Wissenschaftliches Rechnen

Steffen Börm

Stand 17. Februar 2012

Inhaltsverzeichnis

1 Einleitung

Unter „Wissenschaftlichem Rechnen“ versteht man den Einsatz von Computern für die Lösung wissenschaftlicher Probleme. In der Regel sind dabei eine Reihe von Teilproblemen zu behandeln:

1. Die mathematische **Modellierung** überführt das Problem in die Form mathematischer Gleichungen, aus deren Analyse sich die Lösung ergibt.

2. Häufig wird durch eine **Diskretisierung** eine kontinuierliche Formulierung durch eine angenähert, die durch *endlich viele* Größen beschrieben werden kann und deshalb für einen Computer handhabbar ist.

3. Anschließend werden **Lösungsverfahren** angewendet, um aus dem mathematischen Modell die für die konkrete Fragestellung wichtigen Größen zu gewinnen.

Häufig ist auch eine **Optimierung** erforderlich, beispielsweise um während der Modellierung bestimmte Parameter des Modells so zu wählen, dass reale Experimente möglichst gut erfasst werden, oder um herauszufinden, wie sich ein simulierter Prozess so steuern lässt, dass er bestimmten Zielvorgaben entspricht.

Der Schwerpunkt dieser Vorlesung liegt auf der Diskretisierung und den Lösungsverfahren. Dabei stehen weniger die abstrakten mathematischen Eigenschaften der Verfahren im Mittelpunkt (die in den Vorlesungen „Numerik von Differentialgleichungen“, „Finite Elemente“, „Iterative Verfahren für große Gleichungssysteme“ und „Numerik nicht-lokaler Operatoren“ ausführlich behandelt werden), sondern praktische und algorithmische Aspekte:

- Wie wird aus einer mathematischen Aufgabenstellung ein Programm, das sie löst?

- Wie muss das Programm geschrieben werden, damit es möglichst schnell arbeitet?

- Wie kann die Lösung (die häufig durch eine große Menge von Zahlen dargestellt ist) einem Menschen vermittelt werden?

Die erste Frage untersuchen wir am für die Praxis wichtigen Beispiel partieller Differentialgleichungen auf einem beschränkten Gebiet. Derartige Gleichungen treten beispielsweise bei der Simulation elektromagnetischer, strömungsdynamischer oder strukturmechanischer Probleme auf.

Ein erster Schritt bei der Behandlung der Gleichungen besteht darin, die Geometrie des Gebiets zu beschreiben, auf dem gerechnet werden soll, beispielsweise die Form eines Werkstücks oder die Kontur eines mit Flüssigkeit gefüllten Hohlraums. Eine relativ einfache Lösung dieser Aufgabe bieten *Finite-Elemente-Gitter*: Das Gebiet wird in einfach

geformte Teilgebiete, beispielsweise Dreiecke, Quader oder Tetraeder, zerlegt, die sich mathematisch kompakt beschreiben lassen.

Der zweite Schritt besteht darin, die Differentialgleichungen in eine Form zu überführen, die der Computer verarbeiten kann. Dazu beschäftigen wir uns mit der Methode der *finiten Elemente*, mit der aus dem die Geometrie beschreibenden Gitter und der Differentialgleichung ein Gleichungssystem mit einer endlichen Zahl von Unbekannten wird, das ein Computer lösen kann.

In der Regel wird dieses System sehr viele Unbekannte aufweisen, so dass hocheffiziente Verfahren eingesetzt werden müssen, um die Lösung in vertretbarer Zeit finden zu können. Sehr erfolgreich in diesem Bereich sind *iterative Verfahren*, die eine Folge von Näherungslösungen berechnen, die (hoffentlich) schnell gegen die exakte Lösung konvergieren, aber auf modernen Computern sehr effizient umgesetzt werden können.

Bei der Beantwortung der zweiten Frage, also der nach der Effizienz eines Programms, sind zwei Aspekte von Bedeutung: Erstens muss der richtige Algorithmus gewählt werden, beispielsweise sollten Speicherbedarf, Rechenaufwand und erreichbare Genauigkeit in einem angemessenen Verhältnis zueinander stehen. Zweitens muss der Algorithmus geeignet umgesetzt, also möglichst gut an den zur Verfügung stehenden Computer angepasst werden, beispielsweise in dem Besonderheiten des Prozessors oder der Speicherarchitektur berücksichtigt werden.

Die dritte Frage wird heute in der Regel durch *Visualisierungsverfahren* beantwortet, die die bei der Behandlung der Aufgabe gewonnenen Daten grafisch aufbereiten und, in schwierigen Fällen animiert oder interaktiv, dem Anwender präsentieren.

2 Einfluss der Hardware auf die Laufzeit

Algorithmen werden in Büchern und Vorlesungen in der Regel in mathematisch abstraktem Pseudo-Code angegeben. In dieser Weise lassen sich die zugrundeliegenden Ideen gut vermitteln, allerdings geht dabei ein wichtiger Aspekt verloren: Die Algorithmen werden in der Regel in einer spezifischen Programmiersprache auf einem spezifischen Computer ausgeführt, und sowohl Programmiersprachen als auch Computer entsprechen nur in erster Näherung dem dem Pseudo-Code zugrunde liegenden Ideal.

2.1 Speicherzugriff

Der Einsatz des Computers wird unumgänglich, wenn sehr viele Gleichungen und Unbekannte zu behandeln sind. In diesem Fall müssen die Unbekannten im Speicher des Computers aufbewahrt werden, und bei Rechenoperationen müssen sie schnell für den Prozessor verfügbar gemacht werden. Bei der abstrakten Diskussion mathematischer Algorithmen geht man in der Regel davon aus, dass der Inhalt des Speichers jederzeit zur Verfügung steht und nur Zeit für die eigentlichen Rechenoperationen anfällt. In der Praxis ist die Geschwindigkeit der Prozessoren wesentlich schneller als die der Speicherbausteine gewachsen, so dass heute viele Prozessoren einen großen Teil ihrer Zeit damit verbringen, auf die Ankunft von Daten aus dem Hauptspeicher zu warten.

LR-Zerlegung Als Beispiel untersuchen wir die LR-Zerlegung, also die Zerlegung einer Matrix $A \in \mathbb{R}^{n\times n}$ in eine untere Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$ und eine obere Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$:

$$A = LR.$$

Indem wir die Matrizen in der Form

$$A = \begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix}, \qquad L = \begin{pmatrix} l_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad R = \begin{pmatrix} r_{11} & R_{1*} \\ & R_{**} \end{pmatrix},$$
$$A_{**}, L_{**}, R_{**} \in \mathbb{R}^{(n-1)\times(n-1)}, \qquad A_{1*}, R_{1*} \in \mathbb{R}^{1\times(n-1)}, \qquad A_{*1}, L_{*1} \in \mathbb{R}^{(n-1)\times 1}$$

zerlegen erhalten wir

$$\begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix} = A = LR = \begin{pmatrix} l_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} r_{11} & R_{1*} \\ & R_{**} \end{pmatrix} = \begin{pmatrix} l_{11}r_{11} & l_{11}R_{1*} \\ L_{*1}r_{11} & L_{*1}R_{1*} + L_{**}R_{**} \end{pmatrix},$$

und die einzelnen Komponenten dieser Gleichung ergeben

$$a_{11} = l_{11}r_{11},$$

$$
\begin{aligned}
A_{*1} &= L_{*1} r_{11}, & \iff \quad L_{*1} &= A_{*1} / r_{11}, \\
A_{1*} &= l_{11} R_{1*}, & \iff \quad R_{1*} &= A_{1*} / l_{11}, \\
A_{**} &= L_{*1} R_{1*} + L_{**} R_{**}, & \iff \quad L_{**} R_{**} &= A_{**} - L_{*1} R_{1*}.
\end{aligned}
$$

Mit Hilfe dieser Gleichungen lässt sich direkt ein Algorithmus für die Berechnung der Zerlegung angeben: Einen der Faktoren l_{11} und r_{11} dürfen wir frei wählen, und wir entscheiden uns für $l_{11} = 1$, also $r_{11} = a_{11}$. Nun lassen sich die erste Spalte L_{*1} der Matrix L mit der zweiten Gleichung und die erste Zeile R_{1*} der Matrix R mit der dritten Gleichung berechnen. Damit lässt sich das sogenannte *Schurkomplement* $A_{**} - L_{*1} R_{1*} = A_{**} - A_{*1} a_{11}^{-1} A_{1*}$ aufstellen, aus dessen LR-Zerlegung sich die verbliebenen Matrizen L_{**} und R_{**} per Rekursion ergeben.

Effiziente Darstellung Um Speicherplatz zu sparen, speichern wir die Matrizen L und R in der unteren und oberen Hälfte der ursprünglichen Matrix A ab, wollen also als Ergebnis

$$
\begin{pmatrix}
r_{11} & r_{12} & r_{13} & \dots & r_{1n} \\
l_{21} & r_{22} & r_{23} & \dots & r_{2n} \\
l_{31} & l_{32} & \ddots & \ddots & r_{3n} \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
l_{n1} & l_{n2} & \dots & l_{n,n-1} & r_{nn}
\end{pmatrix}
$$

erhalten. Da die Diagonaleinträge der Matrix L gleich eins sind, brauchen wir sie nicht abzuspeichern. Wir setzen den Algorithmus so um, dass jeweils in dem rechten unteren Teil der Matrix das Schur-Komplement des jeweils aktuellen Schritts abgelegt wird. Nach dem ersten Schritt haben wir also die Struktur

$$
\begin{pmatrix}
r_{11} & r_{12} & r_{13} & \dots & r_{1n} \\
l_{21} & a_{22}^{(1)} & a_{23}^{(1)} & \dots & a_{2n}^{(1)} \\
l_{31} & a_{32}^{(1)} & \ddots & \ddots & a_{3n}^{(1)} \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
l_{n1} & a_{n2}^{(1)} & \dots & a_{n,n-1}^{(1)} & a_{nn}^{(1)}
\end{pmatrix},
$$

wobei die Einträge des Schur-Komplements durch

$$
a_{ij}^{(1)} = a_{ij} - l_{i1} r_{1j} \qquad \text{für alle } i, j \in \{2, \dots, n\}
$$

gegeben sind. Der vollständige Algorithmus nimmt die Abbildung 2.1 gegebene Gestalt an.

Darstellung von Matrizen Für die konkrete Umsetzung des Algorithmus auf einem Rechner müssen wir zunächst festlegen, wie die zweidimensionale Struktur der Matrizen in die eindimensionale Struktur des Speichers überführt werden soll. Üblich ist die sogenannte *column-major* Anordnung, bei der die Spalten der Matrix hintereinander in den

```
procedure lr_decomposition(var A);
for k = 1 to n − 1 do begin
  for i ∈ {k + 1, . . . , n} do
    a_ik ← a_ik/a_kk;
  for i, j ∈ {k + 1, . . . , n} do
    a_ij ← a_ij − a_ik a_kj
end
```

Abbildung 2.1: Pseudocode der LR-Zerlegung

```
procedure lr_ij_decomposition(var A);
for k = 1 to n − 1 do begin
  for i = k + 1 to n do
    a_ik ← a_ik/a_kk;
  for i = k + 1 to n do
    for j = k + 1 to n do
      a_ij ← a_ij − a_ik a_kj
end
```

Abbildung 2.2: Erste Variante der LR-Zerlegung: Innerste Schleife läuft über Zeilen

Speicher geschrieben werden:

$$\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} \qquad \rightarrow \qquad \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}.$$

Eine Matrix $A \in \mathbb{R}^{n \times m}$ wird dann durch einen Vektor $\hat{a} \in \mathbb{R}^{nm}$ dargestellt, der Zusammenhang ist durch

$$a_{ij} = \hat{a}_{i+(j-1)n} \qquad \text{für alle } i \in \{1, \ldots, n\},\ j \in \{1, \ldots, m\}$$

gegeben.

Bei der konkreten Umsetzung des Algorithmus müssen wir auch festlegen, wie die Schleifen durchlaufen werden. Eine erste Möglichkeit ist in Abbildung 2.2 dargestellt.

Bei dieser Variante wird das Schur-Komplement Zeile für Zeile berechnet: Die äußere Schleife läuft über i, die innere über j. Alternativ können wir es auch Spalte für Spalte

Tabelle 2.1: Laufzeiten für verschiedene Varianten der LR-Zerlegung auf verschiedenen Prozessoren

	Atom	Opteron	Ultra IIIi	Core i7
Variante ij	17.8s	8.4s	9.0s	4.5s
Variante ji	3.2s	0.7s	3.0s	0.4s

```
procedure lr_ji_decomposition(var A);
for k = 1 to n − 1 do begin
  for i = k + 1 to n do
    a_ik ← a_ik/a_kk;
  for j = k + 1 to n do
    for i = k + 1 to n do
      a_ij ← a_ij − a_ik a_kj
end
```

Abbildung 2.3: Zweite Variante der LR-Zerlegung: Innerste Schleife läuft über Spalten

berechnen, also die äußere Schleife über j und die innere über i laufen lassen, und erhalten die in Abbildung 2.3 dargestellt zweite Möglichkeit.

Aus mathematischer Sicht sind beide Varianten des Algorithmus völlig gleichwertig. Führen wir sie allerdings auf konkreten Computern aus, erhalten wir die in Tabelle 2.1 aufgeführten Laufzeiten: Die zweite Variante ist auf manchen Computern mehr als *zehnmal* schneller als die erste.

Struktur des Hauptspeichers Die Ursache ist die Organisation des Speichers: Speicherbausteine sind in der Regel in Form eines zweidimensionalen Gitters aus Speicherzellen organisiert, von denen jede Zelle ein einzelnes Bit repräsentiert. Der Zugriff auf den Speicher erfolgt zeilenweise: Durch eine geeignete Beschaltung wird eine komplette Zeile in einen Puffer übernommen, aus dem dann die Werte der einzelnen Spaltenbits gelesen werden können. Bei modernen Bausteinen entspricht eine Zeile mehreren tausend Bits, die an aufeinander folgenden Adressen liegen. Sobald diese Zeile in den Puffer übernommen wurde, kann sehr effizient auf sie zugegriffen werden. Diese Architektur hat zur Folge, dass Zugriffe auf aufeinander folgende Speicherzellen wesentlich schneller als andere Zugriffsmuster sind, und diese Eigenschaft erklärt den Geschwindigkeitsunterschied zwischen den beiden Varianten unseres Algorithmus: Bei der j-i-Variante wird auf fortlaufende Speicheradressen zugegriffen, bei der i-j-Variante dagegen liegt ein „Abstand“ von n Zellen zwischen zwei aufeinander folgenden Zugriffen, so dass wiederholte Wechsel der Speicherzeile auftreten, die die Zugriffszeiten erheblich verlängern.

Caching Natürlich ist es einem Programmierer in der Regel nicht zuzumuten, bei komplizierteren Algorithmen auf die Details der Speicherarchitektur zu achten. Deshalb besitzen moderne Prozessoren kleine Hilfsspeicher, sogenannte *Caches*, die sehr viel schneller als der konventionelle Speicher sind und die als „Zwischenlager“ für Daten aus dem Hauptspeicher dienen. Sobald der Inhalt einer Speicherzelle aus dem Hauptspeicher angefordert wird, wird er in den Cache übernommen. Sollte wieder auf dieselbe Speicherzelle zugegriffen werden, muss nicht der Hauptspeicher bemüht werden, sondern der Inhalt kann dem Cache entnommen werden. Moderne Prozessoren besitzen eine Hierarchie von Caches, die sich von kleinen und sehr schnellen zu großen und relativ langsamen erstreckt.

```
procedure rl_ijk1_mult(L, R, var B);
for j = 1 to n do
  for i = 1 to n do begin
    b_ij ← 0;
    for k = 1 to n do
      if i ≤ k and k ≥ j do
        b_ij ← b_ij + r_ik l_kj
  end
end
```

Abbildung 2.4: Erste Variante der RL-Multiplikation: Fallunterscheidung in der innersten Schleife

Unabhängig von der konkreten Struktur der Cache-Hierarchie empfiehlt es sich deshalb, Algorithmen so zu organisieren, dass einmal aus dem Speicher gelesene Daten möglichst häufig verwendet werden. Diese *Datenlokalität* sicherzustellen kann zu deutlichen Geschwindigkeitsgewinnen führen.

2.2 Pipelines

Neben der Struktur des Speichers spielt natürlich auch die Struktur des Prozessors eine Rolle, der mit den Daten arbeitet. Als Beispiel untersuchen wir die Multiplikation zweier Dreiecksmatrizen, also die Berechnung von

$$B = RL$$

mit einer oberen Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$ und einer unteren Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$. Wenn wir die Multiplikation direkt umsetzen und ausnutzen, dass wegen der Dreiecksstruktur

$$\begin{aligned} r_{ik} &= 0 && \text{für alle } i,k \in \{1,\ldots,n\} \text{ mit } i > k, \\ l_{kj} &= 0 && \text{für alle } k,j \in \{1,\ldots,n\} \text{ mit } k < j \end{aligned}$$

gilt, erhalten wir den in Abbildung 2.4 gegebenen Algorithmus.

Bei genauerer Betrachtung stellen wir fest, dass in der inneren Schleife nur solche Werte von k relevant sind, die die Bedingungen $i, j \leq k$ erfüllen. Diese Eigenschaft können wir ausnutzen, um die Fallunterscheidung in der innersten Schleife zu vermeiden, indem wir die Grenzen der Schleife anpassen. Damit erhalten wir die in Abbildung 2.5 gegebene Variante.

Dieser Ansatz weist den Nachteil auf, dass innerhalb der innersten Schleife auf ständig wechselnde Spalten der Matrix R zugegriffen wird, so dass damit zu rechnen ist, dass der bereits bei der LR-Zerlegung beobachtete Effekt auftreten wird. Dieser Nachteil lässt sich vermeiden, indem wir die Berechnung umstrukturieren: Wenn wir mit δ_k den k-ten

```
procedure rl_ijk2_mult(L, R, var B);
for j = 1 to n do
  for i = 1 to n do begin
    b_ij ← 0;
    for k = max{i, j} to n do
      b_ij ← b_ij + r_ik l_kj
  end
end
```

Abbildung 2.5: Zweite Variante der RL-Multiplikation: Keine Fallunterscheidung in der innersten Schleife

kanonischen Einheitsvektor bezeichnen, also den Vektor, der in der k-ten Komponente gleich eins ist und ansonsten gleich null, können wir die Identitätsmatrix in der Form

$$I = \sum_{k=1}^{n} \delta_k \delta_k^*$$

schreiben. Indem wir diese Matrix in unsere Gleichung einschieben, erhalten wir

$$B = RL = RIL = R \sum_{k=1}^{n} \delta_k \delta_k^* L = \sum_{k=1}^{n} (R\delta_k)(\delta_k^* L).$$

Der Vektor $R\delta_k$ ist gerade die k-te Spalte der Matrix R, während der Vektor $\delta_k^* L$ der k-ten Zeile der Matrix L entspricht. Damit können wir die Berechnung der Matrix B als eine Folge von Produkten eines Spalten- und eines Zeilenvektors darstellen und eine ähnliche Struktur wie im Fall der LR-Zerlegung zu erreichen. Mit Hilfe dieser Darstellung erhalten wir die in Abbildung 2.6 gegebene dritte Variante des Algorithmus.

```
procedure rl_kji_mult(L, R, var B);
B ← 0;
for k = 1 to n do
  for j = 1 to k do
    for i = 1 to k do
      b_ij ← b_ij + r_ik l_kj
end
```

Abbildung 2.6: Dritte Variante der RL-Multiplikation: Folge von Rang-1-Updates

Die Laufzeiten für die drei Varianten unseres Algorithmus sind in Tabelle 2.2 zusammengestellt. Nach unserer Diskussion des Einflusses der Speicherstruktur auf die Geschwindigkeit der LR-Zerlegung überrascht es nicht, dass die dritte Variante am schnellsten ist, denn sie ist in ihrer Struktur der der LR-Zerlegung sehr ähnlich.

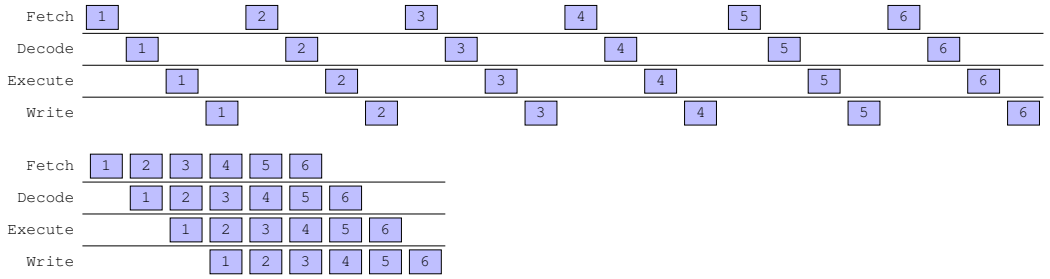

Abbildung 2.7: Effekt des Fließbandprinzips: Die Ausführung von sechs Befehlen benötigt 9 statt 24 Taktzyklen.

Aber wie erklärt sich der teilweise deutliche Vorsprung der zweiten Variante vor der ersten? Da bei beiden in exakt derselben Reihenfolge auf exakt dieselben Speicherzellen zugegriffen wird, kann die Struktur des Hauptspeichers nicht für den Unterschied verantwortlich sein.

Pipelining Die Ursache liegt diesmal in der Architektur der Prozessors begründet. Moderne Prozessoren verarbeiten Befehle nach dem Fließbandprinzip (engl. *pipelining*): Die Verarbeitung eines Befehls verläuft in mehreren Phasen, beispielsweise

1. Befehlscode holen (engl. *instruction fetch*),
2. Befehl decodieren und Daten beschaffen (engl. *instruction decoding*),
3. Befehl ausführen (engl. *execution*) und
4. Ergebnisse abspeichern (engl. *write back*).

Für jede Phase gibt es einen Schaltkreis auf dem Prozessor, der die entsprechenden Aufgaben erfüllt. Wenn wir mit der Ausführung eines Befehls warten würden, bis sein Vorgänger alle Phasen durchlaufen hat, würden zu jedem Zeitpunkt drei der vier Schaltkreise ungenutzt sein.

Die Lösung bietet das Fließbandprinzip (siehe Abbildung 2.7): Sobald der erste Befehl Phase 2 erreicht hat, kann der zweite Befehl in Phase 1 eintreten. Sobald der erste Befehl Phase 3 erreicht hat, kann der zweite in Phase 2 eintreten und der dritte in Phase

Tabelle 2.2: Laufzeiten für verschiedene Varianten der RL-Multiplikation auf verschiedenen Prozessoren

	Atom	Opteron	Ultra IIIi	Core i7
Variante ijk1	12.5s	2.1s	10.9s	1.9s
Variante ijk2	9.2s	1.9s	4.6s	1.4s
Variante kji	3.2s	0.7s	2.7s	0.4s

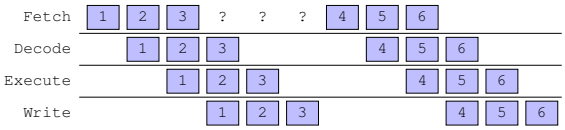

Abbildung 2.8: Nachteil bedingter Sprünge: Falls erst nach der Ausführung des dritten Befehls fest steht, welches der vierte Befehl sein wird, entsteht eine Lücke in der Befehlsfolge.

1. Ab dem vierten Befehl sind so alle vier Schaltkreise gleichzeitig ausgelastet und die Rechenzeit wird deutlich reduziert. Um den Prozessor mit hoher Taktfrequenz betreiben zu können ist es erstrebenswert, die einzelnen Schaltkreise so einfach wie möglich zu halten, deshalb wird die Verarbeitung eines Befehls bei modernen Prozessoren häufig in mehr als zehn besonders einfache Phasen unterteilt.

Bedingte Sprünge Das entscheidende Problem tritt auf, sobald das Programm *bedingte Sprünge* enthält, sobald also der nächste Befehl von dem Ergebnis vorangehender Berechnungen abhängt. Falls der erste Befehl des Programms darüber entscheidet, ob der zweite Befehl an Adresse 5 oder Adresse 10 zu finden ist, kann mit der Verarbeitung des zweiten Befehls eigentlich erst begonnen werden, sobald der erste Befehl vollständig abgearbeitet wurde (siehe Abbildung 2.8). Dadurch kann das Fließbandprinzip nicht mehr wie gewünscht arbeiten, die Effizienz des Prozessors sinkt. Dieser Effekt macht sich offensichtlich um so deutlicher bemerkbar je höher die Anzahl der Phasen ist.

Besonders ungünstig ist in dieser Hinsicht die erste Variante unseres Programms: Die innerste Schleife enthält eine Fallunterscheidung, die von den Werten der Variablen i, j und k abhängt, also gerät innerhalb der innersten Schleife immer wieder das Fließband in's Stocken. Die zweite Variante des Programms ist wesentlich günstiger: Da i und j nur einmal bei der Berechnung des Startwerts für die Variable k eingehen, lässt sich die Anzahl der bedingten Sprünge mit Hilfe einiger einfacher Tricks deutlich reduzieren und so die Geschwindigkeit deutlich verbessern. Bei der dritten Variante schließlich profitieren wir zusätzlich von dem besseren Zugriffsmuster auf den Hauptspeicher.

Sprungvorhersage Auch in diesem Fall versuchen moderne Prozessoren, dem Programmierer zu helfen. Eine übliche Technik besteht in der Sprungvorhersage (engl. *branch prediction*), bei der der Prozessor zu „raten“ versucht, an welche Stelle ein bedingter Sprung springen wird, um mit der Verarbeitung der vermutlich nächsten Befehle zu beginnen (engl. *speculative execution*). Falls der Prozessor richtig geraten hat, erreicht er wieder seine volle Effizienz. Anderenfalls muss er die fälschlicherweise ausgeführten Befehle verwerfen und an der korrekten Position neu ansetzen.

In unserem Fall wird in der zweiten und dritten Variante fast immer ein Sprung an den Beginn der jeweils aktuellen Schleife erfolgen, so dass die Sprungvorhersage relativ zuverlässig arbeitet und falsche Vorhersagen selten sind. Bei der ersten Variante dagegen hängt der Sprung von zwei Variablen und einer komplizierteren Bedingung ab, die eine Vorhersage erschwert.

2.3 BLAS

Um dem vorhandenen Computer die bestmögliche Leistung zu entlocken ist es offenbar erforderlich, relativ viel über seinen internen Aufbau zu wissen, und viele moderne Prozessoren sind relativ kompliziert. Da das wissenschaftliche Rechnen als sehr prestigeträchtige Disziplin gilt, gibt es allerdings einen Ausweg: Gerade für die wichtigen Operationen aus dem Bereich der linearen Algebra hat man sich auf einen Satz von standardisierten Funktionen geeinigt, die für viele Systeme in Form einer optimierten Bibliothek zur Verfügung gestellt werden. So kann das Know-How der Hardware-Hersteller in die Verbesserung der Bibliothek einfließen, während die Anwender Programme schreiben können, die auf unterschiedlichen Rechnern die jeweils passenden Bibliotheken verwenden und so hoffentlich immer die bestmögliche Leistung erreichen.

Diese Sammlung von Funktionen trägt den Namen *BLAS*, *basic linear algebra subprograms*, und wurde ursprünglich in der Sprache FORTRAN entwickelt. Die Dokumentation der Bibliothek sowie die Referenz-Implementierung findet sich unter der Adresse `http://www.netlib.org/blas`.

BLAS-Datentypen Die zentralen Datentypen der BLAS-Bibliothek sind naheliegenderweise Vektoren und Matrizen. Ein Vektor wird durch einen Zeiger auf seinen ersten Eintrag und ein Inkrement (eine ganze Zahl) beschrieben, das angibt, um wieviele Stellen im Speicher gesprungen werden muss, um den nächsten Eintrag zu erhalten. Wenn der Zeiger `x` und das Inkrement `xinc` sind, ist der i-te Eintrag gerade `x[xinc*(i-1)]` (denn der erste Eintrag findet sich an der Stelle `x[0]`).

Eine Matrix wird durch einen Zeiger auf ihren linken oberen Eintrag und ein Spalteninkrement (engl. *leading dimension*) beschrieben, das angibt, um wieviele Stellen im Speicher gesprungen werden muss, um die nächste Spalte zu erhalten. Implizit wird dabei davon ausgegangen, dass die nächste Zeile in der unmittelbar folgenden Speicherzelle zu finden ist. Wenn der Zeiger `a` und das Spalteninkrement `lda` sind, ist der Eintrag in der i-ten Zeile der j-ten Spalte gerade `a[(i-1)+(j-1)*lda]`.

Diese auf den ersten Blick einfachen Datentypen bieten sehr viel Flexibilität: Wenn wir beispielsweise aus der Matrix den zu der j-ten Spalte gehörenden Vektor gewinnen wollen, können wir `y=a+(j-1)*lda` und `yinc=1` verwenden. Für den zu der i-ten Zeile gehörenden Vektor erhalten wir entsprechend `z=a+(i-1)` und `zinc=lda`. Wenn wir bei der Berechnung der LR-Zerlegung zum Schur-Komplement übergehen, ist die entsprechende Teilmatrix durch `b=a+1+lda` und `ldb=lda` beschrieben.

BLAS-Stufen BLAS-Funktionen werden in der Regel danach klassifiziert, mit welchen Datentypen sie arbeiten: Funktionen der Stufe 1 führen Operationen mit Vektoren durch, beispielsweise die Berechnung von Linearkombinationen, Skalarprodukten und Normen. Funktionen der Stufe 2 führen Operationen mit Matrizen und Vektoren durch, beispielsweise die Matrix-Vektor-Multiplikation und das Vorwärts- und Rückwärtseinsetzen, aber auch die Addition der Rang-1-Matrix xy^* zu einer Matrix. Funktionen der Stufe 3 arbeiten mit Matrizen, das wichtigste Beispiel dabei ist die Matrix-Matrix-Multiplikation.

Üblicherweise darf man davon ausgehen, dass Funktionen der Stufe 3 effizienter als solche der Stufe 2 sind, da sie die „gehaltvollere Struktur“ der Matrizen besser ausnutzen können. Aus demselben Grund darf man auch erwarten, dass Funktionen der Stufe 2 effizienter als solche der Stufe 1 sind. Aus diesem Grund versucht man bei der Konstruktion numerischer Algorithmen häufig, Rechenschritte zusammenzufassen, um Funktionen der Stufe 3 verwenden zu können und so die Geschwindigkeit zu steigern.

Umgekehrt lassen sich Funktionen der höheren Stufen (häufig suboptimal) implementieren, indem man Funktionen der niedrigeren Stufen verwendet. Beispielsweise lässt sich die Matrix-Matrix-Multiplikation auf eine Folge von Matrix-Vektor-Multiplikationen zurückführen, die sich wiederum auf eine Folge von Linearkombinationen von Vektoren zurückführen lässt.

BLAS-Funktionen tragen in der Regel Namen mit höchstens sechs Buchstaben, bei denen typischerweise der erste Buchstabe den Datentyp beschreibt: „s“ für reelle Zahlen einfacher Genauigkeit (engl. *single precision*), „d“ für solche doppelter Genauigkeit (engl. *double precision*), „c“ für komplexe Zahlen einfacher Genauigkeit und „z“ für solche doppelter Genauigkeit. Bei Funktionen der Stufen 2 und 3 geben der zweite und dritte Buchstabe die Organisation der Matrix an: „ge“ für allgemeine Matrizen in der hier eingeführten spaltenweisen Darstellung, „sy“ für symmetrische Matrizen, bei denen nur die untere oder obere Dreieckshälfte verwendet, aber trotzdem die gesamte Matrix gespeichert wird, „sp“ für symmetrische Matrizen, bei denen nur die notwendige Hälfte der Koeffizienten gespeichert wird (engl. *packed symmetric*). Es gibt noch weitere Matrixdarstellungen für speziellere Anwendungen, beispielsweise Dreiecks- und Bandmatrizen. Die verbliebenen Buchstaben beschreiben die auszuführende Funktion.

BLAS Stufe 1 Typische Funktionen der Stufe 1 sind die Linearkombination $y \leftarrow \alpha x + y$ (z.B. `daxpy`), das euklidische Skalarprodukt $\langle x, y \rangle_2$ (z.B. `sdot`) oder die euklidische Norm $\|x\|_2$ (z.B. `dnrm2`). Diese Funktionen erhalten als Parameter neben den wie oben beschrieben dargestellten Vektoren auch die Dimension der Vektoren.

BLAS Stufe 2 Die wichtigste Funktion der Stufe 2 ist die Matrix-Vektor-Multiplikation $y \leftarrow \alpha Ax + \beta y$ (z.B. `cgemv`), die neben den Parametern der Matrix und der Vektoren auch die Zeilen- und Spaltenzahl der Matrix erwartet und mit einem separaten Parameter angewiesen werden kann, statt mit der Matrix A mit ihrer adjungierten Matrix A^* zu multiplizieren. Wichtig sind ebenfalls das Rang-1-Update $A \leftarrow A + \alpha xy^*$ (z.B. `sger`) und das Vorwärts- und Rückwärtseinsetzen (z.B. `dtrsv`).

BLAS Stufe 3 Unter den Funktionen der Stufe 3 ist die Matrix-Matrix-Multiplikation $C \leftarrow \alpha AB + \beta C$ (z.B. `sgemm`) sicherlich die wichtigste. Neben den die Matrizen beschreibenden Parametern erwartet sie weitere Daten, die die Anzahl der Zeilen und Spalten der beteiligten Matrizen festlegen und entscheiden, ob A oder B transponiert werden sollen. Außerdem gibt es beispielsweise Funktionen für Rang-k-Updates $C \leftarrow C + \alpha AB^*$ (z.B. `dsyrk`) und das simultane Vorwärts- oder Rückwärtseinsetzen mit mehreren rechten Seiten (z.B. `ctrsm`).

Der Einsatz der BLAS-Funktionen lohnt sich in der Regel erst, wenn Matrizen und Vektoren eine gewisse Größe erreichen, da ansonsten der mit dem Aufruf der Funktion verbundene Verwaltungsaufwand (z.B. für die Übergabe der Parameter und deren Überprüfung) zu sehr in's Gewicht fällt.

Neben der Referenz-Implementierung, die im Quelltext verfügbar ist und sich mit jedem FORTRAN-Compiler übersetzen lassen sollte, gibt es von verschiedenen Herstellern optimierte Implementierungen der BLAS-Funktionen, beispielsweise die *Intel Math Kernel Library* für Computer mit Intel-Prozessor, die *SUN Performance Library* für Solaris-Rechner oder die *AMD Core Math Library* für AMD-Prozessoren. Auf vielen Systemen lässt sich auch *ATLAS* (Abkürzung für *Automatically Tuned Linear Algebra Software*) übersetzen, eine BLAS-Implementierung, die so geschrieben wurde, dass sie sich innerhalb gewisser Grenzen automatisch an die Konfiguration der jeweiligen Hardware anpassen kann. Unter der Adresse `http://math-atlas.sourceforge.net` finden sich die Dokumentation und die Quelltexte.

2.4 Speziellere Techniken

Prefetching Bestimmte Prozessoren bieten Befehle, mit denen sie darauf hingewiesen werden können, dass demnächst der Inhalt bestimmter Speicherzellen benötigt werden dürfte. Diese Funktion (engl. *prefetch* genannt) kann vor allem bei komplexen Datenstrukturen mit für den Prozessor schwer vorhersehbaren Zugriffsmustern die Effizienz erheblich verbessern, denn Daten können parallel zu einer laufenden Berechnung in den Cache transportiert werden, um spätere Wartezeiten zu vermeiden. Manche hochentwickelte Prozessoren besitzen Schaltkreise, die versuchen, die Zugriffsmuster auch ohne Hilfe des Programmierers beziehungsweise des Compilers vorherzusagen.

SIMD Eine weitere Modifikation des Befehlssatzes moderner Prozessoren ist die Aufnahme von SIMD-Befehlen (engl. *single instruction multiple data*), mit denen sich bestimmte Berechnungen mit größeren Datenmengen simultan durchführen lassen. Ein Beispiel ist der SSE-Befehlssatz (engl. *Streaming SIMD Extension*), der auf aktuellen Prozessoren der Hersteller Intel und AMD Verwendung findet: Eine Operation lässt sich simultan auf zwei oder vier Zahlen anwenden, beispielsweise lassen sich mit einem einzigen Befehl vier Paare von Zahlen addieren oder multiplizieren. Die Ausführung dieses Befehls dauert dabei unter idealen Bedingungen nicht länger als die Ausführung einer einzelnen Addition oder Multiplikation, so dass sich erhebliche Geschwindigkeitsgewinne erreichen lassen. In der Praxis ist es häufig schwierig, Algorithmen so zu strukturieren, dass sie von SIMD-Befehlssätzen profitieren können, deshalb verlässt man sich häufig eher auf Bibliotheken wie BLAS, die hoffentlich die Hardware optimal ausnutzen.

Grafikkarten Praktisch jeder Computer verfügt über Schaltkreise, mit denen sich Texte und Grafiken ausgeben lassen. Da von spezialisierten Grafikchips heute erwartet wird, dass sie auch dreidimensionale Grafiken darstellen können, müssen sie in der Lage

sein, die dafür erforderlichen Rechenoperationen sehr schnell durchzuführen. Seit einigen Jahren wird die ursprünglich nur für diesen Zweck vorgesehene Rechenleistung auch für allgemeinere Anwendungen nutzbar gemacht, aktuelle Grafikchips lassen sich mit Hilfe von Schnittstellen wie CUDA (engl. *compute unified device architecture*, siehe `http://www.nvidia.com/object/cuda_home_new.html`) oder OpenCL (siehe `http://www.khronos.org/opencl`) wesentlich freier als ihre Vorgänger programmieren. Da Grafikchips eine Rechenleistung zur Verfügung stellen, die die moderner Prozessoren um zweistellige Faktoren übersteigt, können sie für bestimmte Algorithmen sehr attraktiv sein. Allerdings sind ihren Anwendungen immer noch gewisse Grenzen gesetzt: Grafikchips sind in der Regel für geringere Mengen an Hauptspeicher als „echte“ Prozessoren ausgelegt, der Speicher ist gelegentlich weniger zuverlässig (schließlich sollte er ursprünglich nur für einen Sekundenbruchteil ein Bild anzeigen, nicht über Stunden eine Matrix aufbewahren) und ihr Befehlssatz ist weniger reichhaltig.

3 Visualisierung

Bei vielen Aufgaben im bereich des wissenschaftlichen Rechnens fallen große Mengen an Daten an, beispielsweise in Form simulierter Messwerte an vielen im Raum verteilten Punkten. Da die Interpretation dieser Daten letztendlich wieder Aufgabe eines Menschen ist, ist es wichtig, die Daten in einer Form zu präsentieren, die für uns verständlich ist.

3.1 Einfache Zeichenbefehle

Bereits frühe Heimcomputer konnten zweidimensionale Grafiken darstellen: Ein Teil des Hauptspeichers wurde als Grafikspeicher deklariert, dessen Bits von einem dafür zuständigen separaten Schaltkreis als Bildpunkte interpretiert wurden. Durch Manipulation des Speichers konnte der Prozessor dann (zunächst monochrome, später auch farbige) Grafiken erstellen.

Im Zeitalter grafischer Benutzeroberflächen ist dieser Zugang nicht mehr angemessen: Ein Grafikprogramm sollte *portabel* sein, also auf möglichst vielen verschiedenen Rechnern ausführbar sein und zu vergleichbaren Ergebnissen führen. Deshalb verwendet man *Grafikbibliotheken*, also Sammlungen von Funktionen, die ein Programmierer verwenden kann, um Grafiken zu erstellen. Die Funktionen sind unabhängig von den Eigenschaften der Grafikhardware (wie der Auflösung oder der Anzahl der verfügbaren Farben) definiert, damit die auf ihnen aufsetzenden Programme dieses Eigenschaft erben. Sobald für eine gegebene Hardwareplattform die Bibliothek implementiert wurde, sind alle auf dieser Bibliothek aufsetzenden Programme lauffähig.

Als Beispiel befassen wir uns mit der frei verfügbaren Grafikbibliothek Cairo (siehe `http://cairographics.org`), die relativ leistungsfähig und auf den meisten Rechnern einsetzbar ist.

Cairo kann seine Grafiken nicht nur auf dem Bildschirm ausgeben, sondern auch in Postscript- oder PDF-Dateien schreiben. Das gewünschte Ausgabemedium wird mit dem Typ `cairo_surface_t` beschrieben, Objekte dieses Typs lassen sich mit entsprechenden Funktionen anlegen. Beispielsweise erhalten wir mit

```
cairo_surface_t *surface;

surface = cairo_pdf_surface_create("Bild.pdf", 400.0, 250.0);
```

ein Objekt, das seine Zeichenbefehle in eine PDF-Datei der Breite 400 und Höhe 250 schreibt. Breite und Höhe sind dabei in abstrakten Längeneinheiten gegeben, der Ursprung des Koordinatensystems liegt in der linken oberen Ecke.

Kontext. Leistungsfähige Grafiksysteme bieten eine Vielzahl von Parametern, mit denen sich das Aussehen grafischer Objekte beeinflussen lässt. Schon wenn wir nur eine Linie zwischen zwei Punkten (x_1, y_1) und (x_2, y_2) zeichnen wollen, spielen beispielsweise auch die Farbe und die Dicke der Linie eine Rolle. Alle diese Parameter bei jeder Zeichenoperation anzugeben wäre sehr umständlich, deshalb werden die meisten in einem separaten Objekt gespeichert, das den *Kontext* bezeichnet, in dem eine Operation ausgeführt werden soll. In Cairo ist dafür der Typ `cairo_t` zuständig. Wenn uns ein Objekt des Typs `cairo_surface_t` zur Verfügung steht, können wir mit

```
cairo_t *cr;

cr = cairo_create(surface);
cairo_surface_destroy(surface);
```

ein entsprechendes Objekt anlegen. Der Aufruf der Funktion `cairo_surface_destroy` wirkt auf den ersten Blick etwas deplaziert: Warum sollen wir das Objekt `surface` zerstören, wenn wir es doch noch benutzen wollen, um Zeichenbefehle auszugeben? Tatsächlich zählt Cairo, wieviele Objekte `surface` verwenden, und löscht es erst, wenn es nicht mehr gebraucht wird. Der Funktionsaufruf bewirkt also lediglich, dass `surface` genau so lange bestehen bleiben wir, wie `cr` es braucht. Danach wird es automatisch gelöscht. Der Kontext `cr` wird mit sinnvollen Werten initialisiert, so dass wir sofort etwas zeichnen können.

Pfade. Grafiken setzen sich in Cairo aus *Pfaden* zusammen, das sind Sequenzen von Zeichenbefehlen, die zunächst nur gespeichert und erst mit dem Aufruf einer entsprechenden Funktion gezeichnet werden. Dadurch ist es Cairo möglich, beispielsweise die Übergänge zwischen Teilstücken einer Kurve zu glätten oder das Innere einer Kurve gleichmäßig einzufärben. Jeder Pfad weist unter anderem einen aktuellen Endpunkt auf, an dem weitere Zeichenbefehle ansetzen. Um eine Linie von dem Punkt $(10, 10)$ zu dem Punkt $(390, 240)$ zu zeichnen, müssen wir zuerst mit dem Befehl `cairo_move_to` den Endpunkt in $(10, 10)$ plazieren und dann mit dem Befehl `cairo_line_to` eine Linie vom soeben gesetzten Endpunkt zu dem neuen Punkt zeichnen, der danach der neue Endpunkt wird:

```
cairo_move_to(cr, 10.0, 10.0);
cairo_line_to(cr, 390.0, 240.0);
```

Nun haben wir einen Pfad erzeugt, der die gewünschte Linie enthält, also können wir nun Cairo den Auftrag erteilen, diesen Pfad zu zeichnen. Der einfachste Befehl dafür ist `cairo_stroke`, der einen Linienzug ausgibt:

```
cairo_stroke(cr);
```

Einen einzelnen Strich zu zeichnen ist zwar ein wichtiger Schritt auf dem Weg zu wesentlich komplexeren Grafiken, aber für sich genommen wenig interessant. Also zeichnen wir als nächstes ein Dreieck:

```
cairo_move_to(cr, 80.0, 20.0);
cairo_line_to(cr, 290.0, 160.0);
cairo_line_to(cr, 330.0, 40.0);
cairo_close_path(cr);
cairo_stroke(cr);
```

Der uns bisher unbekannte Befehl `cairo_close_path` fügt dem aktuellen Pfad eine Verbindungslinie von dem aktuellen Endpunkt zu dem Anfangspunkt des Pfads hinzu, in unserem Fall also die dritte Kante des Dreiecks. Falls wir einen Pfad mehrfach verwenden wollen, können wir ihn mit dem Befehl `cairo_copy_path` in einem Objekt des Typs `cairo_path_t` speichern und später mit `cairo_append_path` dem jeweils aktuellen Pfad hinzufügen. Das Pfad-Objekt kann mit `cairo_path_destroy` wieder freigegeben werden, sobald wir es nicht mehr benötigen.

Polygone. Mit den bisher eingeführten Befehlen können wir lediglich Linien zeichnen. In vielen Anwendungen sind *Polygone*, also von Linienzügen eingeschlossene Flächen, von Interesse. Um statt des durch einen Pfad beschriebenen Linienzugs ein von ihm berandetes Polygon zu zeichnen, genügt es, den Befehl `cairo_stroke` durch den neuen Befehl `cairo_fill` zu ersetzen. Beispielsweise zeichnen die Befehle

```
cairo_move_to(cr, 20.0, 230.0);
cairo_line_to(cr, 80.0, 210.0);
cairo_line_to(cr, 100.0, 120.0);
cairo_line_to(cr, 30.0, 100.0);
cairo_close_path(cr);
cairo_fill(cr);
```

ein ausgefülltes Viereck.

Weitere Zeichenoperationen. Neben Linien kann Cairo unter anderem auch Kreisbögen, Rechtecke und Text ausgeben. Für Kreisbögen sind die Befehle `cairo_arc` und `cairo_arc_negative` zuständig, die für einen gegebenen Mittelpunkt und Radius Kreisbögen im beziehungsweise gegen den Uhrzeigersinn von einem Anfangswinkel bis zu einem Endwinkel zeichnen (die Bibliothek zeichnet die Kreisbögen im mathematisch positiven beziehungsweise negativen Sinn, aber die y-Achse zeigt nach unten, nicht nach oben). Achsenparallele Rechtecke lassen sich mit `cairo_rectangle` durch Angabe der Koordinaten der oberen linken Ecke und ihrer Breite sowie Höhe zeichnen. Text kann mit dem Befehl `cairo_show_text` ausgegeben werden, wobei sich die Schriftgröße mit dem Befehl `cairo_set_font_size` festlegen lässt. Das folgende Programmfragment zeichnet einen Halbkreis, ein Rechteck und gibt ein Wort aus:

```
cairo_arc(cr, 250.0, 210.0, 20.0, 0.0, 3.141);
cairo_rectangle(cr, 230.0, 180.0, 40.0, 30.0);
cairo_move_to(cr, 240.0, 198.0);
cairo_show_text(cr, "Box");
cairo_stroke(cr);
```

Es ist dabei zu beachten, dass der Winkel für den Kreisbogen zwischen 0 und 2π liegt, nicht zwischen 0 und 360 Grad.

Linienbreite, Strichelung. In der Ausgangskonfiguration zeichnet Cairo Linien einheitlicher Dicke. Um die Ausdrucksmöglichkeiten zu erweitern, bietet die Bibliothek die Möglichkeit, diese Grundeinstellungen zu verändern. Breite und Strichelung der Linien lassen sich mit den Befehlen

```
double dashes[2] = { 4.0, 6.0 };

cairo_set_line_width(cr, 0.8);
cairo_set_dash(cr, dashes, 2, 0.0);
```

vorgeben. Der erste Befehl `cairo_set_line_width` sorgt dafür, dass alle Linien mit einer Breite von $0,8$ Einheiten gezeichnet werden.

Der zweite Befehl `cairo_set_dash` legt das Muster der Strichelung fest: Jede Linie beginnt mit einem Strich der Länge 4, dann folgt ein Leerraum von 6 Einheiten, und dann wiederholt sich dieses Muster so lange, bis die gewünschte Länge der Linie erreicht ist.

Farbe. Die Farben, mit denen Cairo Zeichenbefehle ausführt, können sehr flexibel gewählt werden, beispielsweise können sie direkt einer zweiten Grafik entnommen werden. Für die Zwecke dieser Einführung genügt es, den Befehl `cairo_set_source_rgb` zu beschreiben, der die Farbe durch Mischung der Grundfarben rot, grün und blau beschreibt. Der Anteil jeder Farbe wird durch eine Zahl zwischen null und eins festgelegt:

```
cairo_set_source_rgb(cr, 1.0, 0.0, 0.0);
cairo_set_source_rgb(cr, 0.5, 0.5, 0.5);
cairo_set_source_rgb(cr, 0.7, 0.0, 0.7);
```

Der erste Befehl legt das intensivste Rot fest, das das Grafiksystem zu bieten hat. Bei dem zweiten Befehl sind alle Farben in mittlerer Intensität vorhanden, so dass ein Grauton entsteht. Bei dem dritten Befehl sind rot und blau gemischt, so dass eine gelbe Farbe das Ergebnis ist.

Kontext speichern und wiederherstellen. Die Befehle `cairo_stroke` und `cairo_fill` zeichnen den aktuellen Pfad beziehungsweise füllen das von ihm berandete Polygon, danach wird der Pfad gelöscht. Falls wir beispielsweise ein berandetes Polygon zeichnen wollen, wäre es praktisch, wenn wir beide Befehle, eventuell mit unterschiedlichen Farben, auf denselben Pfad anwenden könnten. Ein erster Schritt zu diesem Ziel sind die Befehle `cairo_stroke_preserve` und `cairo_fill_preserve`, die die Zeichenbefehle ausführen, aber den zugehörigen Pfad nicht löschen. Um das Innere des Polygons mit einer anderen Farbe als seinen Rand zu zeichnen, wäre es hilfreich, wenn wir die Farbe *temporär* ändern könnten, um nach dem Füllbefehl mit der ursprünglichen Farbe weiterarbeiten zu können. Diesem Zweck dienen die Funktionen `cairo_save` und `cairo_restore`, die den aktuellen Kontext speichern und wiederherstellen. Ein schwarz berandetes grünes Dreieck können wir so mit Hilfe der folgenden Befehle zeichnen:

```
cairo_move_to(cr, 100.0, 190.0);
cairo_line_to(cr, 200.0, 220.0);
cairo_line_to(cr, 110.0, 90.0);
cairo_close_path(cr);
cairo_save(cr);
cairo_set_source_rgb(cr, 0.0, 1.0, 0.0);
cairo_fill_preserve(cr);
cairo_restore(cr);
cairo_stroke(cr);
```

Es ist zu beachten, dass der aktuellen Pfad *nicht* zum Kontext gehört, deshalb ist es in diesem Beispiel wichtig, die Funktion `cairo_fill_preserve` zu verwenden, die den Pfad nicht löschen.

Die Ergebnisse unserer bisherigen Zeichenbefehle sind in der Abbildung 3.1 zu sehen.

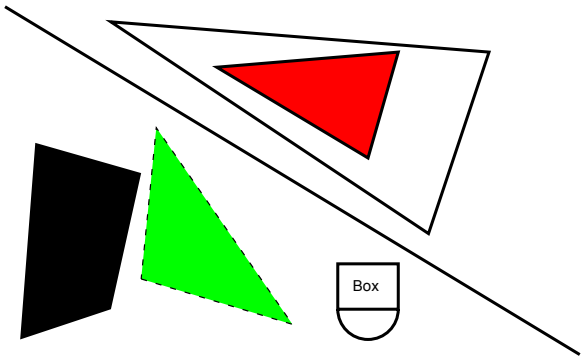

Abbildung 3.1: Einfache Cairo-Zeichenbefehle

3.2 Transformationen

Bisher wurde das Koordinatensystem, in dem wir unsere Grafik beschreiben, in dem Moment festgelegt, in dem wir das Objekt des Typs `cairo_surface_t` anlegen, das die Grafik ausgeben soll. In vielen Anwendungen ist es wünschenswert, das Koordinatensystem modifizieren zu können, beispielsweise um seinen Mittelpunkt in die Mitte der Zeichenfläche zu legen. Aus diesem Grund enthält der Cairo-Kontext eine affine Transformation, die auf jedes Koordinatenpaar angewendet wird und sich durch geeignete Funktionsaufrufe verändern lässt.

Als Beispiel gehen wir wieder von einer PDF-Datei der Breite 400 und der Höhe 250 aus, die wir wie gehabt mit den folgenden Befehlen anlegen:

```
surface = cairo_pdf_surface_create("Bild.pdf", 400.0, 250.0);
cr = cairo_create(surface);
cairo_surface_destroy(surface);
```

Um den Ursprung des Koordinatensystems in die Mitte der Zeichenfläche zu verschieben, verwenden wir den Befehl `cairo_translate` in der folgenden Weise:

```
cairo_translate(cr, 200.0, 125.0);
```

Im neuen Koordinatensystem liegt der Punkt $(0, 0)$ jetzt an dem Ort, an dem im alten Koordinatensystem der Punkt $(200, 125)$ lag.

Wir würden auch gerne das Koordinatensystem so skalieren, dass die Entfernung vom Ursprung zu der oberen beziehungsweise unteren Kante der Zeichenfläche gerade gleich eins ist. Diesem Zweck dient der Befehl `cairo_scale`, der zwei Skalierungsfaktoren für x- und y-Koordinaten entgegennimmt:

```
cairo_scale(cr, 125.0, 125.0);
cairo_set_line_width(cr, 0.01);
```

Wir wählen beide Faktoren identisch, damit Längen in x- und y-Richtung kongruent bleiben. Der Aufruf der Funktion `cairo_set_line_width` ist erforderlich, weil auch die Liniendicke von der Skalierung betroffen ist und deshalb nach der Skalierung Linien 125mal breiter als vorher erscheinen würden. Wollten wir Text ausgeben, müssten wir an dieser Stelle auch die Schriftgröße anpassen.

In diesem Koordinatensystem lassen sich nun einfach beispielsweise parametrisierte Kurven wie

$$\gamma : [0, 2\pi] \to \mathbb{R}^2, \qquad t \mapsto \begin{pmatrix} \alpha \sin(t) + \beta \sin(tk) \\ \alpha \cos(t) + \beta \cos(tk) \end{pmatrix}, \tag{3.1}$$

darstellen, indem man den Definitionsbereich in n Teilintervalle der Form $[t_{i-1}, t_i]$ mit $0 = t_0 < t_1 < \ldots < t_n = b$ zerlegt, die Funktion in den Stützstellen t_i auswertet und einen Pfad konstruiert, der jeweils $\gamma(t_{i-1})$ mit $\gamma(t_i)$ verbindet. Für die durch $t_i = i/n$ gegebene äquidistante Unterteilung ergibt sich beispielsweise das folgende Programmstück:

```
cairo_move_to(cr, 0.0, alpha + beta);
for(i=1; i<n; i++)
  cairo_line_to(cr, alpha * sin(2.0*M_PI*i/n)
                    + beta * sin(2.0*M_PI*k*i/n),
                   alpha * cos(2.0*M_PI*i/n)
                    + beta * cos(2.0*M_PI*k*i/n));
cairo_close_path(cr);
cairo_stroke(cr);
```

Mathematisch entspricht der so dargestellte Polygonzug gerade der stückweise linearen Splinefunktion, die γ in den Stützstellen interpoliert. Da über diese Art der Interpolation viel bekannt ist, lässt sich der Fehler, der bei der Approximation der „echten“ Kurve γ durch den Polygonzug entsteht, leicht abschätzen und so ein n wählen, das zu einer akzeptablen Grafik (siehe Abbildung 3.2) führt.

Geschachtelte Transformationen. Da die Transformation des Koordinatensystems Bestandteil des Kontexts ist, wird sie insbesondere von den Funktionen `cairo_save` und `cairo_restore` gespeichert und wiederhergestellt. Diese Eigenschaft ermöglicht es uns, komplexere grafische Objekte aus geeignet verschobenen, skalierten oder rotierten einfacheren Objekten zusammenzusetzen. Besonders praktisch ist diese Technik in Verbindung mit rekursiven Algorithmen, da sich bei ihnen für jeden rekursiven Aufruf ein

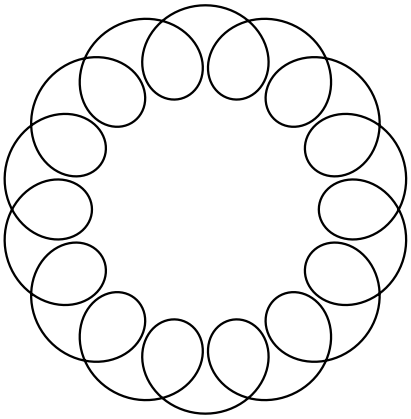

Abbildung 3.2: Visualisierung der parametrisierten Kurve aus Gleichung (3.1)

anderes Koordinatensystem verwenden lässt. Als Beispiel konstruieren wir eine einfache Schneeflockenkurve: Unsere „Schneeflocken“ sollen sich aus fünf „Armen“ zusammensetzen. Ein „Arm“ der Schneeflocke soll aus einer Linie bestehen, die sich einer gewissen Länge in drei kleinere Arme verästelt (siehe Abbildung 3.3), die dieselbe Struktur aufweisen.

Einen einzelnen „Arm“ konstruieren wir, indem wir eine Linie zeichnen und dann für jeden der „Zweige“ ein geeignet verschobenes, skaliertes und rotiertes Koordinatensystem erzeugen, für das wir unsere Zeichenfunktion rekursiv aufrufen:

```
static void
snowflake_arm(cairo_t *cr, int l)
{
  cairo_move_to(cr, 0.0, 0.0);
  cairo_line_to(cr, 0.0, 0.5);
  cairo_stroke(cr);
```

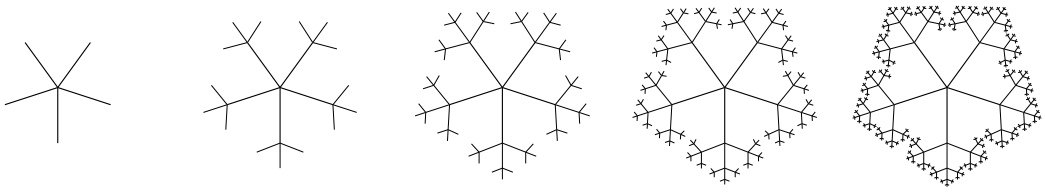

Abbildung 3.3: Einfache Schneeflockenkurve, rekursiv konstruiert

```
  if(l > 0) {
    cairo_save(cr);

    cairo_translate(cr, 0.0, 0.5);
    cairo_scale(cr, 0.45, 0.45);

    snowflake_arm(cr, l-1);

    cairo_rotate(cr, 1.2);
    snowflake_arm(cr, l-1);

    cairo_rotate(cr, -2.4);
    snowflake_arm(cr, l-1);

    cairo_restore(cr);
  }
}
```

Die ersten drei Zeilen der Funktion zeichnen das Linienstück. Falls keine weitere Rekursion erforderlich ist, werden die restlichen Zeilen übersprungen. Anderenfalls wird der Kontext gespeichert, und damit insbesondere die aktuelle Koordinatentransformation. Der Mittelpunkt der Koordinatensystems wird an den Endpunkt des Linienstücks verschoben und das Koordinatensystem skaliert, damit die weiteren Arme kleiner werden. Einen weiteren Arme können wir nun direkt per rekursivem Aufruf zeichnen, weil er in dieselbe Richtung wie das Linienstück zeigt. Für die beiden anderen Arme muss das Koordinatensystem mit der Funktion `cairo_rotate` gedreht werden, bevor der rekursive Aufruf erfolgen kann. Schließlich wird mit `cairo_restore` der ursprüngliche Kontext, also auch die entsprechende Koordinatentransformation, wiederhergestellt, damit in der aufrufenden Funktion keine Schwierigkeiten entstehen.

In Abbildung 3.3 sind fünf Kurven für die Parameter $\ell \in \{0, \ldots, 4\}$ dargestellt. Es kommen wieder Koordinatentransformationen zum Einsatz, um die fünf Grafiken zu einer größeren Grafik zusammenzusetzen:

```
for(l=0; l<5; l++) {
  cairo_save(cr);
  cairo_translate(cr, 40.0+l*80.0, 40.0);
  cairo_scale(cr, 40.0, 40.0);
  cairo_set_line_width(cr, 0.01);

  for(i=0; i<5; i++) {
    cairo_save(cr);
```

```
      cairo_rotate(cr, 2.0 * M_PI * i / 5);
      snowflake_arm(cr, l);
      cairo_restore(cr);
    }
    cairo_restore(cr);
  }
```

Jeder Aufruf der Funktion `snowflake_arm` erfolgt so in einem Koordinatensystem, das so gewählt ist, dass es zu keinen Überschneidungen mit den Ergebnissen der anderen Aufrufe kommen kann.

Homogene Koordinaten. Mathematisch werden Transformationen in der Computergrafik in der Regel durch Matrizen dargestellt. Da das Produkt einer Matrix mit dem Nullvektor immer den Nullvektor ergibt, könnten wir in dieser Weise keine Verschiebungen handhaben, so dass die Struktur der Bibliothek potentiell sehr viel komplizierter werden würde.

Glücklicherweise gibt es eine elegante Möglichkeit, um dieses Problem zu vermeiden: Obwohl wir im zweidimensionalen Raum arbeiten, führen wir eine dritte Koordinate ein, die immer gleich eins ist, man spricht von *homogenen Koordinaten*. Bei diesem Ansatz kann kein Nullvektor mehr auftreten, lediglich der Vektor $(0, 0, 1)$, und es lässt sich leicht nachrechnen, dass

$$\begin{pmatrix} 1 & & a \\ & 1 & b \\ & & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ 1 \end{pmatrix}$$

gilt, wir also Verschiebungen als Multiplikation mit einer 3×3-Matrix darstellen können. Entsprechend ist eine Skalierung durch

$$\begin{pmatrix} \alpha & & \\ & \beta & \\ & & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha x \\ \beta y \\ 1 \end{pmatrix}$$

und eine Rotation durch

$$\begin{pmatrix} \cos(\varphi) & -\sin(\varphi) & \\ \sin(\varphi) & \cos(\varphi) & \\ & & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos(\varphi) - y\sin(\varphi) \\ x\sin(\varphi) + y\cos(\varphi) \\ 1 \end{pmatrix}$$

darstellbar. Im Kontext der Cairo-Bibliothek kann also die gerade aktuelle Koordinatentransformation durch eine einzige 3×3-Matrix dargestellt werden. Sobald eine neue Transformation hinzu kommt, wird die entsprechende Matrix *von rechts* mit der aktuellen Transformation multipliziert.

Es lässt sich leicht nachprüfen, dass bei jeder in dieser Weise auftretenden Matrix die letzte Zeile immer gleich sein wird: Zwei Nullen, dann eine Eins. Deshalb verzichtet Cairo darauf, diese Zeile abzuspeichern.

3.3 Grafische Benutzerschnittstelle

In vielen Anwendungen ist es wünschenswert, dass ein Programm mit dem Anwender interagieren kann, beispielsweise indem es die Möglichkeit bietet, bestimmte Parameter zu verändern oder bei der Visualisierung der Ergebnisse die Perspektive oder die Detailstufe zu wählen.

Seit einigen Jahren ist es üblich, diese Kommunikation mit dem Programm über grafische Benutzerschnittstellen (engl. *GUI*, *graphical user interface*) abzuwickeln: Das Programm stellt Bedienelemente wie Knöpfe, Schieberegler und Eingabefelder zur Verfügung, deren Benutzung zu passenden Reaktionen führt.

Es gibt eine Reihe von Programmbibliotheken, mit denen sich solche Schnittstellen implementieren lassen, allerdings sind sie meistens auf spezielle Betriebssysteme zugeschnitten und deshalb für Programme eher uninteressant, die auf möglichst vielen Systemen lauffähig sein sollen. Für unsere Zwecke ist deshalb die Bibliothek GTK+ `http://www.gtk.org` attraktiver, die auf vielen Systemen verwendet werden kann und den Vorteil bietet, in der Sprache C geschrieben zu sein und auf Cairo aufzusetzen, so dass sich Cairo-Befehle ohne großen Aufwand verwenden lassen.

Wie viele andere Bibliotheken für grafische Schnittstellen beruht GTK+ auf zwei Konzepten: Die Bibliothek ist *ereignisgesteuert* (engl. *event-driven*) und *objektorientiert* (engl. *object-oriented*).

Ereignisgesteuerte Programme. Ein konventionelles Programm führt die durch den Programmierer festgelegten Befehle der Reihe nach aus, bis das Ergebnis fest steht. Diese Vorgehensweise ist sinnvoll, um eine längere Berechnung durchzuführen, sie ist allerdings wesentlich weniger geeignet, wenn eine Interaktion mit dem Benutzer stattfinden soll: Der Benutzer wird die ihm angebotenen Bedienelemente in einer für das Programm nicht vorhersehbaren Reihenfolge verwenden und erwarten, dass das Programm angemessen reagiert.

Ein ereignisgesteuertes Programm tut genau das: Die Aktionen des Benutzers werden als *Ereignisse* registriert, die das Programm verarbeiten soll. Der Programmierer kann für jedes Ereignis festlegen, wie das Programm reagieren soll, in der Regel in Form eines Funktionsaufrufs (engl. bezeichnet als *event handler*). In der Programmiersprache C lässt sich diese Struktur einfach umsetzen, da Funktionen als Zeiger gespeichert werden:

```
typedef int callback_func(int, double);

int example(int x, double y) { return (y < 0.0 ? x+1 : x-1) }

callback_func f = example;

printf("f(4, 5.0)=%d\n", f(4, 5.0));
```

Wir können den Typ `callback_func` definieren, der in diesem Fall Zeiger auf Funktionen aufnimmt, die ein `int`- und ein `double`-Argument erwarten und ein `int`-Ergebnis zurückgeben. Die Funktion `example` ist so eine Funktion, also dürfen wir sie der Variablen `f` dieses Typs zuweisen. Ähnlich wie bei Arrays und Zeigern unterscheidet C auch nicht zwischen Funktionen und Zeigern auf Funktionen, so dass wir den in `f` gespeicherten Funktionszeiger wie eine Funktion aufrufen können, um ein Ergebnis zu erhalten.

Diese Eigenschaft der Programmiersprache ermöglicht es uns, ereignisgesteuerte Programme relativ einfach umzusetzen: Die GUI-Bibliothek muss lediglich eine Tabelle verwalten, in der zu jedem Ereignis ein Zeiger steht, der definiert, welche Funktion ausgeführt werden soll, falls dieses Ereignis eintritt. In Analogie zum Telefon bezeichnet man derartige Funktionen als *Callback-Funktionen*: Der Programmierer hinterlässt bei der GUI-Bibliothek die „Telefonnummern“ der Funktionen (also die Funktionszeiger), und die GUI-Bibliothek „ruft zurück“, falls sie es für nötig erachtet.

Objektorientierte Programmierung. Unter einem *Objekt* im Sinne der objektorientierten Programmierung versteht man eine Einheit aus Daten und Funktionen, die mit diesen Daten arbeiten. In der Sprache C ist eine derartige Einheit relativ einfach zu realisieren, da die Funktionen durch ihre Funktionszeiger beschrieben werden können, die sich wie gewöhnliche Variablen verhalten. Beispielsweise muss ein `struct` lediglich um geeignete Felder erweitert werden.

Der Typ eines Objekts wird auch als seine *Klasse* bezeichnet, während eine Variable dieses Typs als *Instanz* der Klasse bezeichnet wird. Verschiedene Instanzen einer Klasse unterscheiden sich also in ihren Daten, aber nicht in den Typen der Daten oder den zugehörigen Funktionen. Die zu einer Klasse gehörenden Funktionen bezeichnet man als *Methoden*.

Im Kontext der GUI-Programmierung sind diese Konzepte gut geeignet: Ein Eingabefeld für Text beispielsweise muss über Daten verfügen, beispielsweise müssen seine Position auf dem Bildschirm, seine Größe, sein Inhalt und ähnliches mehr gespeichert werden, aber auch über Methoden, beispielsweise muss es reagieren, wenn der Anwender Text eingeben möchte.

Verschiedene Klassen von Objekten besitzen häufig gemeinsame Eigenschaften. Beispielsweise muss jedes Bedienelement einer GUI-Bibliothek sich auf dem Bildschirm grafisch darstellen können, horizontale und vertikale Schieberegler müssen auf Mausbewegungen reagieren, und Knöpfe und Eingabefelder auf Mausklicks. Diese gemeinsamen Eigenschaften drückt man im Kontext der objektorientierten Programmierung durch eine *Klassenhierarchie* aus: Von einer Klasse `A` kann eine zweite Klasse `B` abgeleitet werden, die alle Datenelemente und Funktionen der Klasse `A` bietet, aber weitere hinzufügen und die Funktionen verändern kann. Mathematisch entspricht eine Klasse dem Konzept der Menge, eine Instanz einem Element, und eine abgeleitete Klasse einer Teilmenge.

Klassenhierarchie in GTK+. In der Bibliothek GTK+ ist die Klasse `GtkWidget` (die Bezeichnung *widget* entstand aus *window gadget*, wobei ein *gadget* im Englischen ein originelles kleines technisches Objekt bezeichnet) der „kleinste gemeinsame Nenner“ al-

ler auf dem Bildschirm dargestellten Bedienelemente: Jede Instanz dieser Klasse kann sich auf dem Bildschirm darstellen, auf Mausklicks reagieren, seine Größe anpassen und ähnliches. Von `GtkWidget` leiten sich verschiedene weitere Elemente ab, von denen nun einige behandelt werden sollen.

Fenster. Die Bedienelemente unseres Programms werden wir in der Regel in einem oder mehreren Fenstern auf dem Bildschirm arrangieren. Ein Fenster ist von `GtkWidget` abgeleitet und kann mit der Funktion `gtk_window_new` angelegt werden:

```
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

Der Parameter `GTK_WINDOW_TOPLEVEL` legt dabei fest, dass das Fenster eigenständig auf dem Bildschirm dargestellt und durch das System verwaltet werden soll. Die Alternative ist `GTK_WINDOW_POPUP`, dann entsteht ein Fenster, das beispielsweise nicht durch den Benutzer verschoben oder in seiner Größe geändert werden kann. Diese zweite Sorte von Fenstern ist für Menüfenster und ähnliches gedacht. Wir können mit einigen Funktionen die Eigenschaften des Fensters modifizieren:

```
gtk_window_set_title(GTK_WINDOW(window), "My first window");
gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);
gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
```

Der erste Befehl legt den Titel des Fensters fest, der üblicherweise in der Titelzeile angezeigt wird, der zweite die Ausgangsgröße, der dritte die Ausgangsposition (in diesem Fall die Mitte des Bildschirms). Hier fällt eine Besonderheit der Bibliothek GTK+ auf: Funktionen wie `gtk_window_new` geben immer ein Objekt des Typs `GtkWidget` zurück. Wenn wir eine der Methoden der Klasse `GtkWindow` verwenden wollen, müssen wir deshalb erst den Typ in `GtkWindow` umwandeln. Das übernimmt das Makro `GTK_WINDOW`. Da jedes Widget zunächst unsichtbar ist, müssen wir schließlich mit

```
gtk_widget_show(window);
```

unser Fenster aus der Unsichtbarkeit erlösen.

Knöpfe. Bisher würde unser Programm lediglich ein leeres Fenster auf dem Bildschirm anzeigen. Um es etwas nützlicher zu machen, fügen wir einen Knopf hinzu. Die entsprechende Klasse heißt `GtkButton`, die Befehle

```
widget = gtk_button_new();
gtk_button_set_label(GTK_BUTTON(widget), "Hello");
gtk_widget_show(widget);
```

legen eine neue Instanz an, sorgen für eine sinnvolle Beschriftung und machen den Knopf sichtbar. Trotzdem würde er nicht angezeigt werden, da er, wie die meisten Bedienelemente, erst innerhalb eines Fensters seinen Platz finden muss. Unser `GtkWindow` ist nicht direkt von `GtkWidget` abgeleitet, es gibt die Klasse `GtkContainer` als Zwischenstufe, die Objekte beschreibt, die andere Objekte enthalten können. Diese Objekte werden mit der Methode `gtk_container_add` hinzugefügt, im Fall unseres Fenster wird also

```
gtk_container_add(GTK_CONTAINER(window), widget);
```

den Knopf in dem Fenster deponieren.

Callback-Funktion. Wenn wir unser Programm jetzt starten, würde der Knopf im Fenster angezeigt werden, und er würde optisch reagieren, wenn wir ihn anklicken. Geschehen würde ansonsten nichts, denn wir haben noch nicht festgelegt, was passieren soll. Hier kommt die ereignisorientierte Programmierung zum Zug: Wenn der Knopf angeklickt wird, erzeugt GTK+ ein entsprechendes Ereignis und sucht nach einem Weg, das Ereignis zu behandeln. Wir können die entsprechende Callback-Funktion wie folgt festlegen:

```
g_signal_connect(widget, "clicked",
                 G_CALLBACK(hello_clicked), NULL);
```

Die Verarbeitung von Ereignissen übernimmt eine separate Bibliothek (namens GLib), deshalb beginnt dieser Funktionsaufruf nicht mit dem Kürzel `gtk`. Der Funktionsaufruf besagt, dass die noch zu definierende Funktion `hello_clicked` aufgerufen werden soll, falls bei unserem Knopf `widget` das Ereignis `clicked` eintritt. Die nötige Callback-Funktion können wir für die Zwecke unseres Beispiels relativ einfach halten:

```
static void
hello_clicked(GtkButton *button, gpointer data)
{
  printf("Hello world\n");
}
```

Hauptschleife. Jetzt sind wir fast fertig, wir müssen nur noch den Einstieg in die und den Ausstieg aus der Ereignisverarbeitung regeln. Die Ereignisse werden in einer zentralen (Fast-)Endlosschleife entgegengenommen und verarbeitet, die wir mit `gtk_main` starten und mit `gtk_main_quit` wieder verlassen. Wir wollen, dass die letztere Funktion aufgerufen wird, sobald wir das Fenster schließen. Das Schließen des Fensters ist ebenfalls ein Ereignis, nämlich eines des Typs `destroy`, für das wir eine Callback-Funktion definieren können:

```
g_signal_connect(window, "destroy",
                 G_CALLBACK(gtk_main_quit), NULL);
```

Jetzt ist die gesamte grafische Schnittstelle unseres Programms definiert, und wir können die Hauptschleife mit

```
gtk_main();
```

starten. GTK+ ist zu diesem Zeitpunkt bekannt, dass es unser Fenster gibt, und es wird damit beginnen, Ereignisse der Typen `realize`, `configure-event` und `expose` zu verteilen, um die einzelnen Widgets zu initialisieren, ihre Größe und Position festzulegen und sie schließlich anzuzeigen.

Initialisierung. Der eine Funktionsaufruf, der in jedem auf GTK+ beruhenden Programm vor praktisch allen anderen Aufrufen der Bibliothek stattfinden muss, hat mit der Funktion der Bibliothek wenig zu tun und wird deshalb erst jetzt erwähnt: Der Bibliothek muss mit einem Aufruf der Funktion `gtk_init` (oder einer ihrer Varianten) die Möglichkeit gegeben werden, einige Vorbereitungen zu treffen, beispielsweise um die Eigenschaften des Bildschirms zu ermitteln und auf Kommandozeilenparameter zu reagieren. Diese Parameter erhält die Funktion `main` unseres Hauptprogramms, so dass sie direkt an `gtk_init` weitergereicht werden können:

```
int
main(int argc, char **argv)
{
  GtkWidget *window, *widget;
  gtk_init(&argc, &argv);

  /* ... */

  return 0;
}
```

Container. Eine Benutzerschnittstelle, bei der jedes Fenster nur jeweils einen einzigen Knopf enthalten kann, wäre wenig nützlich. Glücklicherweise sind von `GtkContainer` noch weitere Widgets abgeleitet, die andere Widgets aufnehmen können. Sehr nützlich sind dabei die von `GtkBox` abgeleiteten Klassen `GtkHBox` und `GtkVBox`, die es uns ermöglichen, mehrere Widgets horizontal oder vertikal zu arrangieren. Sie können mit den Funktionen `gtk_hbox_new` und `gtk_vbox_new` angelegt werden:

```
hbox = gtk_hbox_new(TRUE, 3);
vbox = gtk_vbox_new(TRUE, 3);
```

Beide Funktionen erwarten zwei Parameter. Der erste kann die Werte `TRUE` oder `FALSE` annehmen. `TRUE` führt dazu, dass alle in dem Container enthaltenen Widgets gleich breit (für eine `GtkHBox`) oder hoch (für eine `GtkVBox`) sein müssen, während bei `FALSE` auf Kosten des regelmäßigen Erscheinungsbilds Platz gespart wird. Der zweite Parameter legt fest, wie groß der Abstand zwischen den einzelnen Widgets mindestens sein muss.

Da die Klasse `GtkBox` von der Klasse `GtkContainer` abgeleitet ist, können wir mit `gtk_container_add` Widgets hinzufügen:

```
widget = gtk_button_new();
gtk_button_set_label(GTK_BUTTON(widget), "Button A");
gtk_container_add(GTK_CONTAINER(hbox), widget);
```

Allerdings bietet uns die Klasse `GtkBox` die Möglichkeit, das Arrangement der Widgets genauer zu beeinflussen, indem wir die Funktionen `gtk_box_pack_start` und `gtk_box_pack_end` verwenden. Die erste füllt den Container von links nach rechts (bei einer `GtkHBox`) beziehungsweise von oben nach unten (bei einer `GtkVBox`), die zweite jeweils von der anderen Seite. Beide akzeptieren drei zusätzliche Parameter:

```
widget = gtk_button_new();
gtk_button_set_label(GTK_BUTTON(widget), "Button B");
gtk_box_pack_start(GTK_BOX(hbox), widget, TRUE, FALSE, 0);
```

Neben der `GtkBox`, in die etwas eingefügt werden soll, und dem Widget, das eingefügt werden soll, steuern die drei weiteren Parameter den Umgang mit Breite (bzw. Höhe) der einzelnen Widgets innerhalb des Containers. In der Regel wird dem gesamten Container mehr Platz zugewiesen werden, als die in ihm enthaltenen Widgets minimal benötigen. Falls der dritte Parameter den Wert `TRUE` hat, wird in diesem Fall dem einzufügenden Widget ein Anteil des zusätzlichen Platzes überlassen, falls er den Wert `FALSE` hat, bleibt

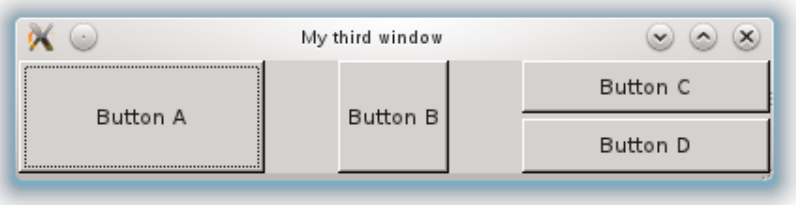

Abbildung 3.4: Eine `GtkVBox` mit zwei Knöpfen (C und D) in einer `GtkHBox` mit zwei weiteren Knöpfen (A und B) in einem `GtkWindow`

die Breite (bzw. Höhe) immer gleich. Der vierte Parameter legt fest, ob der zusätzliche Raum von dem Widget ausgefüllt werden soll (`TRUE`) oder es lediglich als Leerraum umgibt (`FALSE`). Der fünfte Parameter ermöglicht es uns, etwas zusätzlichen Freiraum links und rechts (bzw. oben und unten) des einzufügenden Widgets anzufordern, um den Abstand zu seinen Nachbarn zu steuern.

Da auch `GtkHBox` und `GtkVBox` von `GtkWidget` abgeleitet sind, können wir Instanzen dieser Klassen in andere Instanzen einfügen:

```
gtk_box_pack_start(GTK_BOX(hbox), vbox, TRUE, TRUE, 0);
widget = gtk_button_new();
gtk_button_set_label(GTK_BUTTON(widget), "Button C");
gtk_box_pack_start(GTK_BOX(vbox), widget, TRUE, TRUE, 0);
widget = gtk_button_new();
gtk_button_set_label(GTK_BUTTON(widget), "Button D");
gtk_box_pack_start(GTK_BOX(vbox), widget, TRUE, TRUE, 0);

gtk_container_add(GTK_CONTAINER(window), hbox);
```

Spätestens bei derartigen etwas tiefer verschachtelten Strukturen ist die Funktion `gtk_widget_show_all` nützlich, die nicht nur ein Widget, sondern auch alle in ihm enthaltenen sichtbar macht:

```
gtk_widget_show_all(window);
```

Das resultierende Fenster ist in Abbildung 3.4 dargestellt.

Dateneingabe. Mit einem Klick auf einen Knopf können wir eine beliebige Funktion aufrufen und Berechnungen durchführen. In der Regel werden wir diese Berechnungen auch mit Daten versorgen wollen. Dafür stehen uns eine Reihe weiterer Widgets zur Verfügung.

Für binäre und Boole'sche Werte sind die Klasse `GtkToggleButton` und die von ihr abgeleitete Klasse `GtkCheckButton` zuständig, die sich mit `gtk_toggle_button_new` und `gtk_check_button_new` anlegen lassen. Der jeweilige Zustand lässt sich mit `gtk_toggle_button_get_active` abfragen und mit `gtk_toggle_button_set_active` setzen:

```
widget = gtk_check_button_new();
gtk_button_set_label(GTK_BUTTON(widget), "Boole");
gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(widget), TRUE);
value = gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget));
```

Zeichenketten können wir mit der Klasse `GtkEntry` entgegennehmen, deren Instanzen als einfache Text-Eingabefelder dargestellt werden. Sie können mit `gtk_entry_new` angelegt werden, der aktuelle Inhalt lässt sich mit `gtk_entry_get_text` abfragen und mit `gtk_entry_set_text` ändern:

```
widget = gtk_entry_new();
gtk_entry_set_text(GTK_ENTRY(widget), "3.141");
buf = gtk_entry_get_text(GTK_ENTRY(widget));
sscanf(buf, "%f", &value);
```

Für Zahlenwerte eignet sich die von `GtkEntry` abgeleitete Klasse `GtkSpinButton`, die das Eingabefeld um zwei Knöpfe erweitert, mit der sich numerische Werte variieren lassen. Instanzen dieser Klasse werden mit `gtk_spin_button_new_with_range` angelegt, wobei die Größe des Wertebereichs und die Schrittweite festzulegen sind. Der aktuelle Wert lässt sich mit `gtk_spin_button_get_value` (als `double`) oder `gtk_spin_button_get_value_as_int` (als `int`) auslesen. Ein Aufruf der Funktion `gtk_spin_button_set_value` setzt einen neuen Wert:

```
widget = gtk_spin_button_new_with_range(0.0, 10.0, 1.0);
gtk_spin_button_set_value(GTK_SPIN_BUTTON(widget), 3.0);
value = gtk_spin_button_get_value(GTK_SPIN_BUTTON(widget));
```

Etwas intuitiver lassen sich Zahlenwerte über Schieberegler einstellen, die es in einer horizontalen Fassung `GtkHScale` und einer vertikalen `GtkVScale` gibt. Instanzen dieser Klassen können mit `gtk_hscale_new_with_range` und `gtk_vscale_new_with_range` angelegt werden. Beide sind von der Klasse `GtkRange` abgeleitet, die die gemeinsamen Funktionen `gtk_range_get_value` für die Abfrage des aktuellen Werts und `gtk_range_set_value` für das Setzen desselben bietet:

```
widget = gtk_hscale_new_with_range(-10.0, 10.0, 0.5);
gtk_range_set_value(GTK_RANGE(widget), 2.5);
value = gtk_range_get_value(GTK_RANGE(widget));
```

Ein Auswahl unter mehreren Möglichkeiten kann mit der Klasse `GtkRadioButton` umgesetzt werden, mit der mehrere Knöpfe zu einer Gruppe zusammengefasst werden, in der jeweils nur einer zur Zeit gedrückt sein kann. Mit `gtk_radio_button_new_with_label` können wir eine Instanz anlegen, die keiner Gruppe angehört. Mit der Funktion `gtk_radio_button_new_with_label_from_widget` entsteht eine Instanz, die zu derselben Gruppe wie ein gegebener anderer Knopf gehört:

```
fbox = gtk_vbox_new(FALSE, 0);
widget = gtk_radio_button_new_with_label(NULL, "A");
gtk_box_pack_start(GTK_BOX(fbox), widget, TRUE, FALSE, 0);
widget = gtk_radio_button_new_with_label_from_widget(
              GTK_RADIO_BUTTON(widget), "B");
gtk_box_pack_start(GTK_BOX(fbox), widget, TRUE, FALSE, 0);
widget = gtk_radio_button_new_with_label_from_widget(
              GTK_RADIO_BUTTON(widget), "C");
gtk_box_pack_start(GTK_BOX(fbox), widget, TRUE, FALSE, 0);
```

Hier haben wir die drei Knöpfe in eine neue `GtkVBox` namens `fbox` sortiert, die wir der Übersichtlichkeit halber mit einem beschrifteten Rahmen umschließen wollen. Für solche Rahmen gibt es ebenfalls ein Widget namens `GtkFrame`, das mit `gtk_frame_new` angelegt wird und von `GtkContainer` abgeleitet ist, so dass es `fbox` aufnehmen kann:

```
widget = gtk_frame_new("Choice");
gtk_container_add(GTK_CONTAINER(widget), fbox);
```

Ein Fenster, in dem die so entstandenen Widgets in einer `GtkVBox` angeordnet sind, findet sich in Abbildung 3.5.

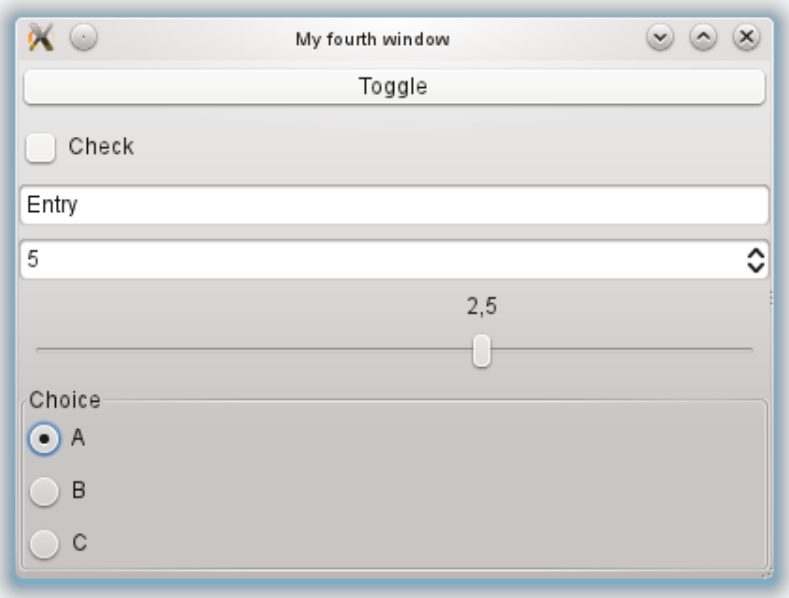

Abbildung 3.5: Eine `GtkVBox` mit einem `GtkToggleButton`, einem `GtkCheckButton`, einem `GtkEntry`, einem `GtkSpinButton`, einem `GtkHScale` und einem `GtkFrame`, der wiederum eine Gruppe mit drei Instanzen der Klasse `GtkRadioButton` enthält.

Zeichenfläche. Selbstverständlich werden wir in der Regel die Ergebnisse unserer Berechnungen grafisch darstellen wollen. Dazu bietet sich das Widget `GtkDrawingArea` an, das eine leere Fläche darstellt, auf der wir mit Hilfe der Cairo-Bibliothek zeichnen können, was wir für sinnvoll erachten. Eine Instanz dieser Klasse wird mit `gtk_drawing_area_new` angelegt, Damit etwas anderes als ein leeres Rechteck angezeigt wird, müssen wir dafür sorgen, dass das Widget die richtige Callback-Funktion aufruft, wenn es gezeichnet werden muss.

In GTK+ 2.x handelt es sich um das Ereignis `expose-event`, und wir können das folgende Programmfragment verwenden, um etwas zu zeichnen:

```
static gboolean
canvas_expose(GtkWidget *widget, GdkEvent *event, gpointer data)
{
  cairo_t *cr;
  gint width, height;
```

```
  gdk_drawable_get_size(GDK_DRAWABLE(widget->window),
                        &width, &height);
  cr = gdk_cairo_create(GDK_DRAWABLE(widget->window));

  cairo_move_to(cr, 0.0, 0.0);
  cairo_line_to(cr, width, height);
  cairo_stroke(cr);

  cairo_destroy(cr);

  return TRUE;
}

/* ... */

canvas = gtk_drawing_area_new();
g_signal_connect(canvas, "expose-event",
                 G_CALLBACK(canvas_expose), NULL);
```

Die Interaktion mit dem Grafiksystem erfolgt in GTK+ mit Hilfe einer separaten Bibliothek namens GDK. Jede Instanz der Klasse `GtkWidget` enthält einen Zeiger auf ein Objekt der Klasse `GdkWindow`, die einen rechteckigen Bereich beschreibt, auf dem gezeichnet werden kann. Die letztere Eigenschaft wird dadurch beschrieben, dass die Klasse von der Klasse `GdkDrawable` abgeleitet ist. Diese Klasse ist für uns interessant, weil sie auf Cairo beruht: Wir können mit `gdk_drawable_get_size` ihre Größe abfragen und mit `gdk_cairo_create` einen Cairo-Kontext anlegen lassen, mit dem wir wie bereits bekannt zeichnen können. Anschließend sollte der Kontext wieder gelöscht werden, um zu vermeiden, dass es bei wiederholten Aufrufen der Callback-Funktion zu Speicherlecks kommt.

In der moderneren Version GTK+ 3.x können wir uns dem Umweg um GDK sparen, hier übernimmt das Ereignis `draw` die Rolle, die `expose-event` in GTK+ 2.x spielte, und die Callback-Funktion erhält einen passenden Cairo-Kontext als Parameter. Damit nimmt unser Beispielprogramm die folgende Form an:

```
static gboolean
canvas_draw(GtkWidget *widget, cairo_t *cr, gpointer data)
{
  gint width, height;

  width = gtk_widget_get_allocated_width(widget);
  height = gtk_widget_get_allocated_height(widget);
```

```
  cairo_move_to(cr, 0.0, 0.0);
  cairo_line_to(cr, width, height);
  cairo_stroke(cr);

  return TRUE;
}

/* ... */

canvas = gtk_drawing_area_new();
g_signal_connect(canvas, "draw",
                 G_CALLBACK(canvas_draw), NULL);
```

Die Bibliothek sorgt dafür, dass der Zustand des Cairo-Kontexts vor dem Aufruf unserer Callback-Funktion gesichert und anschließend wiederhergestellt wird, so dass wir innerhalb der Funktion mit ihm tun können, was wir wollen, ohne Störungen des Gesamtsystems befürchten zu müssen.

Interaktion mit der Zeichenfläche. In manchen Anwendungen kann es sinnvoll sein, dem Benutzer die Möglichkeit zu geben, direkt mit der Zeichenfläche zu interagieren, beispielsweise um angezeigte Objekte auszuwählen oder einen Ausschnitt zu vergrößern. Diese Möglichkeit bieten die Ereignisse `button-press-event` und `motion-notify-event`. Das erste Ereignis wird ausgelöst, falls der Benutzer eine Maustaste innerhalb der Zeichenfläche drückt, das zweite, falls die Maus im Fenster bewegt wird. Wenn wir diese Ereignisse in Verbindung mit der Klasse `GtkDrawingArea` verwenden wollen, müssen wir beachten, dass die Klasse zu Beginn so konfiguriert ist, dass sie die Ereignisse ignoriert. Wir müssen sie deshalb erst mit `gtk_widget_add_events` darauf hinweisen, dass die Ereignisse bearbeitet werden sollen:

```
g_signal_connect(canvas, "button-press-event",
                 G_CALLBACK(canvas_button), canvas);
gtk_widget_add_events(canvas, GDK_BUTTON_PRESS_MASK);
g_signal_connect(canvas, "motion-notify-event",
                 G_CALLBACK(canvas_motion), canvas);
gtk_widget_add_events(canvas, GDK_BUTTON_MOTION_MASK);
```

Die Funktion `gtk_widget_add_events` erwartet dabei im zweiten Argument eine Zahl, deren Bits den einzelnen Ereignissen zugeordnet sind. GDK definiert die Konstanten `GDK_BUTTON_PRESS_MASK` und `GDK_BUTTON_MOTION_MASK`, in denen die Bits gesetzt sind, die zu den beiden für uns interessanten Ereignissen gehören.

Bei der Callback-Funktion für Mausklicks ist relativ wenig zu beachten:

```
static gboolean
canvas_button(GtkWidget *widget, GdkEvent *event, gpointer data)
{
  GdkEventButton *ebutton = (GdkEventButton *) event;
  GtkWidget *canvas = (GtkWidget *) data;

  xstart = ebutton->x;
  ystart = ebutton->y;

  gtk_widget_queue_draw(canvas);
  return FALSE;
}
```

Sobald ein Mausknopf gedrückt wird, wird `canvas_button` aufgerufen und erhält einen Zeiger auf ein Objekt des Typs `GdkEvent`, das nähere Informationen über das Ereignis enthält. Wir wissen, dass es den Typ `GdkEventButton` aufweisen muss, weil wir schließlich einen Mausklick verarbeiten sollen, also können wir es in diesen Typ umwandeln. In den Feldern `x` und `y` finden wir dann die Koordinaten innerhalb der Zeichenfläche, an der sich der Mauszeiger befand, als die Maustaste gedrückt wurde. Mit der Funktion `gtk_widget_queue_draw` wird schließlich veranlasst, dass die Zeichenfläche `canvas` bei nächster Gelegenheit wieder ein Ereignis des Typs `expose-event` beziehungsweise `draw` erhalten wird, um eine visuelle Rückmeldung zu dem Mausklick geben zu können.

Bei der Callback-Funktion für Mausbewegungen ist zu beachten, dass wir GDK explizit darauf hinweisen müssen, dass wir weitere Ereignisse dieses Typs erwarten. Diesem Zweck dient die Funktion `gdk_event_request_motions`, die wie folgt verwendet wird:

```
static gboolean
canvas_motion(GtkWidget *widget, GdkEvent *event, gpointer data)
{
  GdkEventMotion *emotion = (GdkEventMotion *) event;
  GtkWidget *canvas = (GtkWidget *) data;

  xend = emotion->x;
  yend = emotion->y;

  gdk_event_request_motions(emotion);
  gtk_widget_queue_draw(canvas);
  return FALSE;
}
```

3.4 OpenGL

Während in der Vergangenheit infolge der geringen verfügbaren Rechenleistung Simulationen mit ein- oder zweidimensionalen Modellen durchgeführt wurden, sind in den letzten Jahren dreidimensionale Modelle in das Zentrum des Interesses gerückt. Also besteht auch Bedarf an Visualisierungstechniken für derartige Modelle. Da die Darstellung dreidimensionaler Grafik relativ rechenintensiv ist, erfordert sie in der Regel spezialisierte Hardware. Die Kommunikation mit der Hardware erfolgt in Regel über standardisierte Schnittstellen wie DirectX und OpenGL. Im Interesse der Portabilität beschränken wir uns auf OpenGL in der Version 2.1 (aus didaktischen Gründen, denn in späteren Versionen wurden viele der benutzerfreundlicheren Funktionen als „veraltet“ gebrandmarkt, an deren Stelle allgemeinere verwendet werden sollen).

Zeichenoperationen. Im Interesse der Effizienz setzt OpenGL Grafiken aus sehr einfachen geometrischen Objekten zusammen: Aus Punkten, Linien und Dreiecken. Die Definition eines Objekts wird mit einem Aufruf der Funktion `glBegin` begonnen. Es folgt eine Reihe von Aufrufen, die das Objekt beschreibende Eckpunkte beschreiben, beispielsweise `glVertex`. Zum Abschluss wird die Funktion `glEnd` aufgerufen. Ein Beispiel:

```
glBegin(GL_LINES);

glVertex3f(0.0, 0.0, 0.0);
glVertex3f(1.0, 1.0, 0.0);

glVertex3f(1.0, 0.0, 1.0);
glVertex3f(0.0, 1.0, 1.0);

glEnd();
```

Der Parameter des Aufrufs `glBegin` legt fest, wie die bis zu dem Aufruf `glEnd` definierten Eckpunkte $p_1, p_2, \ldots, p_n$ zu interpretieren sind. Wichtig sind die folgenden Objekttypen:

- `GL_POINTS`: Es werden die Punkte $p_1, p_2, \ldots, p_n$ gezeichnet.
- `GL_LINES`: Falls n geradzahlig ist, werden Linien mit den Eckpunkten (p_1, p_2), (p_3, p_4), $\ldots$, (p_{n-1}, p_n) gezeichnet. Falls n ungeradzahlig ist, wird der letzte Punkt ignoriert.
- `GL_LINE_STRIP`: Es werden Linien mit den Eckpunkten $(p_1, p_2), (p_2, p_3), \ldots,$ (p_{n-1}, p_n) gezeichnet.
- `GL_LINE_LOOP`: Es werden die Linien mit den Eckpunkten $(p_1, p_2), (p_2, p_3), \ldots,$ (p_{n-1,p_n}), (p_n, p_1) gezeichnet.

- `GL_TRIANGLES`: Falls n durch drei teilbar ist, werden Dreiecke mit den Eckpunkten $(p_1, p_2, p_3), (p_4, p_5, p_6), \ldots, (p_{n-2}, p_{n-1}, p_n)$ gezeichnet. Ansonsten werden ein bis zwei überzählige Punkte ignoriert.

- `GL_TRIANGLE_STRIP`: Falls n ungerade ist, werden Dreiecke mit den Eckpunkten $(p_1, p_2, p_3), (p_3, p_2, p_4), (p_3, p_4, p_5), \ldots, (p_{n-2}, p_{n-1}, p_n)$ gezeichnet, anderenfalls mit den Eckpunkten $(p_1, p_2, p_3), (p_3, p_2, p_4), (p_3, p_4, p_5), \ldots, (p_{n-1}, p_{n-2}, p_n)$.

Bei Dreiecken ist wichtig, dass OpenGL ihnen eine Vorder- und eine Rückseite zuordnet: Auf der Vorderseite werden die drei Eckpunkte gegen den Uhrzeigersinn, also in mathematisch positiver Reihenfolge, durchlaufen, auf der Rückseite in entgegengesetzter Richtung. Wenn die Oberfläche eines dreidimensionalen Körpers durch Dreiecke dargestellt wird, kann OpenGL so die Dreiecke auf der Rückseite des Körpers aussortieren und den Rechenaufwand reduzieren. Bei der Darstellung eines `GL_TRIANGLE_STRIP` wird die Reihenfolge der Eckpunkte bei jedem zweiten Dreieck umgedreht, um die richtige Orientierung sicherzustellen.

Koordinaten. Das grundlegende Koordinatensystem ist in OpenGL so skaliert, dass Koordinatentupel $(x, y, z) \in [-1, 1]^3$ dargestellt werden. Punkte, deren Koordinaten außerhalb dieses Würfels liegen, werden nicht gezeichnet. Die Punkte werden orthogonal auf die Bildfläche $[-1, 1]^2 \times \{0\}$ projiziert, so dass die z-Koordinate den Abstand eines Punkts zu der Bildfläche beschreibt. Dabei wird das Koordinatensystem nach der Linke-Hand-Regel definiert: Wenn wir uns Daumen, Zeige- und Mittelfinger der linken Hand an die x-, y- und z-Achse angelegt denken, zeigen sie jeweils in die positive Richtung der Achsen.

Die z-Koordinate wird für den *Tiefentest* (engl. *depth test*) verwendet: Ein neuer Punkt in (x, y) wird nur gezeichnet, wenn seine z-Koordinate geringer als die des aktuellen Punkts an dieser Stelle ist. Mit Hilfe dieser Technik lässt sich sehr einfach sicherstellen, dass weiter vorne liegende Objekte physikalisch korrekt weiter hinten liegende verdecken. Dabei spielt es keine Rolle, in welcher Reihenfolge die Objekte gezeichnet werden.

Für die Darstellung der Punkte in einem konkreten Fenster muss festgelegt werden, welcher Teil des Fensters der Bildfläche entsprechen soll. Diesem Zweck dient der Befehl `glViewport`, der die Koordinaten des rechten unteren Eckpunkts sowie Breite und Höhe des Fensters entgegennimmt. Das folgende Programmfragment erfragt die Abmessungen des aktuellen Fensters und sorgt dann dafür, dass das gesamte Fenster als Zeichenfläche dient:

```
gdk_drawable_get_size(GDK_DRAWABLE(widget->window),
                      &width, &height);
glViewport(0, 0, width, height);
```

Da die OpenGL-Zeichenfläche immer quadratisch ist, kann es sinnvoll sein, auch nur einen quadratischen Teil des Fensters zu verwenden, um zu verhindern, dass die Grafik verzerrt erscheint: In x-Richtung gedehnt, falls `width>height` gilt, ansonsten gestaucht. In diesem Fall würden wir wie folgt vorgehen:

```
if(width > height)
  glViewport((width-height)/2, 0, height, height);
else
  glViewport(0, (height-width)/2, width, width);
```

Bei diesem Zugang würden zwar Kreise Kreise bleiben, aber es bliebe auch ein Teil des Fensters ungenutzt.

Transformationen. Eleganter ist es, immer das vollständige Fenster auszunutzen, aber dem unerwünschten Verzerrungseffekt durch eine Skalierung entgegenzuwirken. Für diesen Zweck bietet uns OpenGL, wie schon Cairo, die Möglichkeit, jedes Koordinatentupel einer Transformation zu unterwerfen. Auch OpenGL verwendet homogene Koordinaten, so dass sich Verschiebungen wieder als lineare Transformationen darstellen lassen. Die gewünschte Skalierung lässt sich mit dem Befehl `glScalef` erreichen:

```
glViewport(0, 0, width, height);
if(width > height)
  glScalef((float) height/width, 1.0, 1.0);
else
  glScalef(1.0, (float) width/height, 1.0);
```

Die drei Argumente des Befehls geben die Skalierungsfaktoren für die x-, y- und z-Koordinaten an. Dass die Funktion den Namen `glScalef` anstelle des eigentlich zu erwartenden `glScale` trägt, hängt damit zusammen, dass OpenGL den Typ der übergebenen Parameter im Namen der Funktion verschlüsselt: Neben `glScalef` gibt es auch `glScaled`, und die zweite Funktion erwartet `double`-Parameter, während die erste `float`-Parameter erwartet.

Entsprechend gibt es Funktionen `glTranslatef` und `glTranslated`, die eine Translation durchführen:

```
glTranslatef(0.1, -0.2, 0.0);
```

Mit diesem Funktionsaufruf verschieben wir das Koordinatensystem an die neue Position $(1/10, -1/5, 0)$.

Etwas komplizierter wird es bei Rotationen, denn im dreidimensionalen Raum gibt es mehr als eine Achse, um die wir eine Rotation durchführen könnten, deshalb müssen wir diese Achse explizit angeben:

```
glRotatef(30.0, 0.0, 0.0, 1.0);
```

Anders als Cairo werden Winkel von OpenGL in Grad angegeben, und der obige Befehl rotiert das Koordinatensystem um 30 Grad um die z-Achse: Der erste Parameter gibt den Winkel an, die drei weiteren die x-, y- und z-Koordinaten des Richtungsvektors der Rotationsachse.

Perspektivische Darstellung. Die von OpenGL verwendete orthogonale Projektion entspricht nicht der physikalischen Realität: Weit entfernt liegende Objekte sollten kleiner als nahe erscheinen. Das ist eine Folge des Strahlensatzes, der in Abbildung 3.6 illustriert ist.

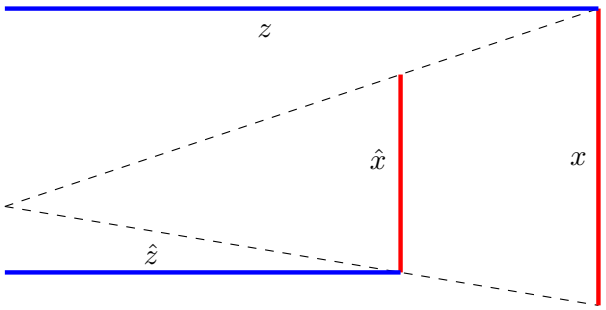

Abbildung 3.6: Perspektivische Projektion per Strahlensatz

Eine Strecke der Länge x in einer Entfernung z von einem Blickpunkt erscheint genauso lang wie eine Strecke der Länge $\hat{x}$ in einer Entfernung $\hat{z}$, falls das Verhältnis zwischen Länge und Entfernung gleich ist, falls also

$$\frac{x}{z} = \frac{\hat{x}}{\hat{z}} \tag{3.2}$$

gilt. Hier ergibt sich ein Problem: Mit Hilfe der Transformationsmatrizen können wir lineare und affine Transformationen durchführen, Divisionen stehen uns aber nicht zur Verfügung. Das Problem lässt sich durch einen Trick lösen: Wir rechnen mit homogenen

Koordinaten und untersuchen die Gleichung

$$\begin{pmatrix} 1 & & & \\ & 1 & & \\ & & & 1 \\ & & 1 & \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ z \end{pmatrix}.$$

Das Ergebnis der Matrix-Vektor-Multiplikation weist in der vierten Komponente nicht mehr die Eins auf, die dort bei homogenen Koordinaten eigentlich stehen sollte. OpenGL führt eine *Renormalisierung* durch, bevor die Koordinaten verwendet werden: Der gesamte Vektor wird so skaliert, dass in der vierten Komponente wieder eine Eins steht.

$$\begin{pmatrix} x \\ y \\ 1 \\ z \end{pmatrix} \quad \mapsto \quad \begin{pmatrix} x/z \\ y/z \\ 1/z \\ 1 \end{pmatrix}.$$

Wie wir sehen haben wir unser Ziel erreicht und die x- und die y-Koordinate durch die z-Koordinate dividiert. In der dritten Komponente des Vektors steht $1/z$, so dass uns auch noch eine Information über die Entfernung des ursprünglichen Punkts zur Bildfläche erhalten geblieben ist.

Aus dem Strahlensatz (3.2) folgt, dass die projizierten Koordinaten durch die Gleichungen

$$\hat{x} = \frac{\hat{z}}{z}x, \qquad\qquad \hat{y} = \frac{\hat{z}}{z}y$$

gegeben sind, wir sollten also in unserer Matrix auch eine Skalierung mit $\hat{z}$ vorsehen. Das ist kein Problem.

Ein Problem ist dagegen die dritte Komponente $d(z) = 1/z$, die die Information über die „Tiefe“ des Bildpunkts enthält: Erstens wird sie um so kleiner, je weiter der Punkt von uns entfernt ist, so dass weiter entfernte Objekte nähere verdecken würden. Zweitens ignoriert OpenGL alle Punkte, deren dritte Komponente nicht im Intervall $[-1, 1]$ liegt, so dass wir nur für $z \geq 1$ etwas sehen würden. Beide Schwierigkeiten lassen sich umgehen, indem wir die dritte Zeile der Transformation etwas verändern. Wir gelangen zu dem Ansatz

$$\begin{pmatrix} \hat{z} & & & \\ & \hat{z} & & \\ & & \alpha & \beta \\ & & 1 & \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x\hat{z} \\ y\hat{z} \\ \alpha z + \beta \\ z \end{pmatrix} \mapsto \begin{pmatrix} x\hat{z}/z \\ y\hat{z}/z \\ (\alpha z + \beta)/z \\ 1 \end{pmatrix} = \begin{pmatrix} x\hat{z}/z \\ y\hat{z}/z \\ \alpha + \beta/z \\ 1 \end{pmatrix}.$$

Die ersten beiden Komponenten sind so, wie sie sein sollen, wir müssen nur noch α und β geeignet wählen. Ein negativer Wert für β sorgt dafür, dass große Werte von z auch zu großen Werten von $d(z) = \alpha + \beta/z$ führen. Da $\hat{z}$ die Entfernung zur Bildfläche beschreiben soll, die im ursprünglichen Koordinatensystem null beträgt, wäre es sicherlich sinnvoll, wenn

$$0 = d(\hat{z}) = \alpha + \beta/\hat{z}$$

gelten würde, also

$$\beta = -\alpha\hat{z}$$

und damit

$$d(z) = \alpha - \alpha\frac{\hat{z}}{z} = \alpha\left(1 - \frac{\hat{z}}{z}\right).$$

Die einfachste Wahl wäre nun $\alpha = 1$. Sie würde dazu führen, dass alle $z \in \mathbb{R}_{\geq\hat{z}}$ in das Intervall $[0, 1]$ abgebildet werden, und zwar monoton wachsend wie gewünscht. Allerdings würden für sehr große Werte von z die Unterschiede zwischen den Werten von $d(z)$ sehr klein werden, und das könnte zu Bildstörungen führen, da Grafikhardware in der Regel nur eine beschränkte Genauigkeit für die Speicherung der Tiefeninformation vorsieht.

Deshalb ist es in der Praxis besser, eine maximale Tiefe $z_{\max} > \hat{z}$ vorzusehen, jenseits der Punkte nicht mehr gezeichnet werden. Im ursprünglichen Koordinatensystem werden Punkte außerhalb des Würfels $[-1, 1]^3$ ignoriert, also liegt die kritische Grenze bei eins, so dass $z_{\max}$ auf 1 abgebildet werden sollte. Wir erhalten die Gleichung

$$1 = d(z_{\max}) = \alpha\left(1 - \frac{\hat{z}}{z_{\max}}\right)$$

und damit

$$\alpha = \frac{1}{1 - \hat{z}/z_{\max}} = \frac{z_{\max}}{z_{\max} - \hat{z}},$$
$$\beta = -\alpha\hat{z} = \frac{\hat{z}}{\hat{z}/z_{\max} - 1} = \frac{\hat{z}z_{\max}}{\hat{z} - z_{\max}}$$

Wenn wir also die Matrix

$$P_{\hat{z},z_{\max}} := \begin{pmatrix} \hat{z} & & & \\ & \hat{z} & & \\ & & z_{\max}/(z_{\max} - \hat{z}) & \hat{z}z_{\max}/(\hat{z} - z_{\max}) \\ & & 1 & \end{pmatrix}$$

als Transformation verwenden, wird die Renormalisierung dazu führen, dass wir eine perspektivisch korrekte Darstellung der dreidimensionalen Geometrie erhalten. Wir können die Transformation mit Hilfe der Funktionen `glLoadMatrixf` (für `float`-Matrizen) und `glLoadMatrixd` (für `double`-Matrizen) OpenGL mitteilen. Die Funktion erwartet eine Matrix in der von uns bereits in Kapitel 2 verwendeten Form: Die vier Spaltenvektoren werden hintereinander im Speicher abgelegt, so dass die 4×4-Matrix als ein Vektor der Länge 16 dargestellt ist.

```
GLfloat P[16];

P[0] = znear; P[4] = 0.0;   P[8] = 0.0;
P[1] = 0.0;   P[5] = znear; P[9] = 0.0;
```

```
P[2] = 0.0;   P[6] = 0.0;   P[10] = zfar / (zfar-znear);
P[3] = 0.0;   P[7] = 0.0;   P[11] = 1.0;

P[12] = 0.0;
P[13] = 0.0;
P[14] = znear * zfar / (znear-zfar);
P[15] = 0.0;

glLoadMatrixf(P);
```

Falls wir die Projektionsmatrix lediglich (von rechts) mit der bereits bestehenden Transformation multiplizieren wollen, können wir alternativ die Funktionen `glMultMatrixf` und `glMultMatrixd` verwenden.

Beleuchtung. In der Regel werden wir relativ enttäuscht sein, wenn wir versuchen, mit den bisher eingeführten Techniken ein dreidimensionales Objekt darzustellen: Auch mit einer physikalisch korrekten Projektion werden wir lediglich einfarbige Polygone auf dem Bildschirm sehen, die keinen räumlichen Eindruck vermitteln (siehe Abbildung 3.7 links).

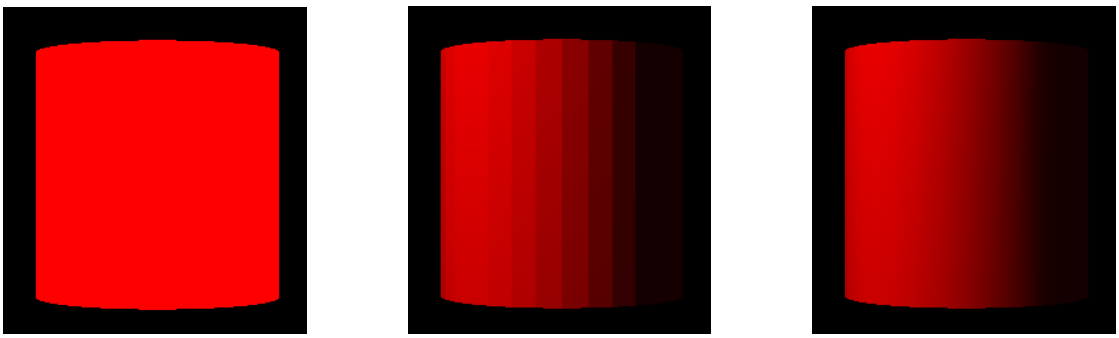

Abbildung 3.7: Durch Rechtecke approximierter Zylinder ohne Beleuchtung, mit einfacher und an die Geometrie angepasster Beleuchtung

Die Ursache liegt darin, dass reale Objekte nicht gleichmäßig eingefärbt sind: Die Oberfläche eines Objekts wirkt heller, wenn sie von einer Lichtquelle beleuchtet wird, und häufig hängt die Helligkeit auch davon ab, in welchem Winkel Lichtstrahlen von der Quelle die Oberfläche erreichen und reflektiert oder gestreut werden.

Aus diesem Grund bietet OpenGL einige einfache *Beleuchtungsmodelle*, mit deren Hilfe sich die Beleuchtung einer Objekts simulieren lässt. Obwohl viele physikalische Effekte wie Schattenwurf, Spiegelungen oder Brechung dabei nicht berücksichtigt werden, lässt sich der räumliche Eindruck der von uns dargestellten Objekte mit diesen Techniken deutlich verbessern (siehe Abbildung 3.7 rechts).

Das einfachste Modell trägt den Namen `GL_EMISSION`: Wir können für jeden von uns gezeichneten Eckpunkt eines Dreiecks eine Farbe festlegen, in der er „leuchten“ soll.

Ähnlich einfach ist das Modell `GL_AMBIENT`, bei dem davon ausgegangen wird, dass die gesamte dargestellte Szene von allen Seiten gleichmäßig mit Licht einer gewissen Farbe beleuchtet wird. Sobald das Licht auf ein von uns gezeichnetes Objekt trifft, werden die Rot-, Grün- und Blau-Komponenten des Lichts mit denen der Objektfarbe multipliziert und so eine neue Farbe berechnet.

Wesentlich überzeugendere Ergebnisse erreichen wir mit dem Modell `GL_DIFFUSE`, bei dem der Winkel zwischen einer Lichtquelle und dem Punkt, der beleuchtet wird, Berücksichtigung findet. Dazu bietet OpenGL eine Reihe von Lichtquellen, deren Position, Farbe und weitere Eigenschaften wir individuell festlegen können. Für jeden zu beleuchtenden Punkt müssen wir einen Einheits-Normalenvektor festlegen, also einen Vektor, der an diesem Punkt senkrecht auf der darzustellenden Oberfläche steht. Um die Stärke des diffus reflektierten Lichts zu bestimmen, berechnet OpenGL das Skalarprodukt zwischen dem Normalenvektor und dem Einheits-Richtungsvektor, der von dem Punkt zu der Lichtquelle weist. Da zwei Einheitsvektoren multipliziert werden, liegt das Ergebnis zwischen -1 und 1. Das Skalarprodukt wird maximal, wenn die Normale in Richtung der Lichtquelle zeigt, die Fläche also frontal von dem Licht getroffen wird, es nimmt stetig ab und erreicht die Null, wenn die Oberfläche parallel zu den Lichtstrahlen liegt, und wird negativ, falls die Oberfläche von der Lichtquelle abgewandt ist. Der diffuse Farbanteil wird wie folgt berechnet: Falls das Skalarprodukt negativ ist, ist die Farbe schwarz. Anderenfalls multiplizieren wir den Wert des Skalarprodukts mit den Rot-, Grün- und Blau-Anteilen der Farbe des Lichts und den korrespondierenden Anteilen der Farbe des Materials.

Die Gesamtfarbe in einem Punkt x mit Normalenvektor n und Richtungsvektor r zu einer Lichtquelle errechnet sich bei Einsatz dieser Modelle als

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} = \begin{pmatrix} r_{\mathrm{em}} \\ g_{\mathrm{em}} \\ b_{\mathrm{em}} \end{pmatrix} + \begin{pmatrix} r_{\mathrm{am}}\hat{r}_{\mathrm{am}} \\ g_{\mathrm{am}}\hat{g}_{\mathrm{am}} \\ b_{\mathrm{am}}\hat{b}_{\mathrm{am}} \end{pmatrix} + \max\{\langle n, r\rangle_2, 0\} \begin{pmatrix} r_{\mathrm{di}}\hat{r}_{\mathrm{di}} \\ g_{\mathrm{di}}\hat{g}_{\mathrm{di}} \\ b_{\mathrm{di}}\hat{b}_{\mathrm{di}} \end{pmatrix}.$$

Hier bezeichnen r_{em}, r_{am} und r_{di} jeweils den Rot-Anteil der Farbe des Objekts für `GL_EMISSION`, `GL_AMBIENT` und `GL_DIFFUSE`, während $\hat{r}_{\mathrm{am}}$ und $\hat{r}_{\mathrm{di}}$ die Farbe des Lichts für die letzten beiden Modelle definiert. Die Grün- und Blau-Anteile sind entsprechend mit g und b bezeichnet.

Die Beleuchtungsberechnung erfolgt bei diesen Modellen nur in den Eckpunkten der gezeichneten Objekte, zwischen den Eckpunkten wir die Farbe linear interpoliert. Dadurch wird der Rechenaufwand gegenüber der Berechnung für jeden Bildpunkt deutlich reduziert, der Preis dafür ist allerdings eine potentiell weniger realistische Darstellung. Den im Modell `GL_DIFFUSE` benötigten Normalenvektor können wir für jeden Punkt mit der Funktion `glNormal` festlegen, der jeweils vor dem korrespondierenden Aufruf der Funktion `glVertex` erfolgen muss.

Dabei gibt es eine Feinheit zu beachten: Der in Abbildung 3.7 dargestellte Zylinder setzt sich aus Rechtecken zusammen, die im mittleren Bild auch gut zu erkennen sind. Sie sind zu erkennen, weil hier die Normalenvektoren verwendet wurden, die zu der Geometrie gehören, mit der OpenGL tatsächlich rechnet, nämlich zu einer Ansammlung von Rechtecken.

In dem rechten Bild dagegen werden die Normalenvektoren verwendet, die zu der eigentlich beabsichtigten Geometrie gehören würden, also zu dem „glatten“ Zylinder. Obwohl OpenGL in beiden Fällen dieselben Rechtecke zeichnet, glauben wir, einen echten Zylinder zu sehen. Das zugehörige Programmfragment sieht wie folgt aus:

```
glBegin(GL_TRIANGLE_STRIP);
for(i=0; i<=segments; i++) {
  glNormal3f(cos(2.0*M_PI*i/segments), 0.0,
             sin(2.0*M_PI*i/segments));
  glVertex3f(radius * cos(2.0*M_PI*i/segments), height,
             radius * sin(2.0*M_PI*i/segments));
  glVertex3f(radius * cos(2.0*M_PI*i/segments), -height,
             radius * sin(2.0*M_PI*i/segments));
}
glEnd();
```

Dabei wird ausgenutzt, dass der Normalenvektor im Fall des Zylinders nicht von der Höhe des Punkts abhängt: Ein einmal festgelegter Normalenvektor gilt für alle folgenden Aufrufe von `glVertex`, bis er geändert wird.

4 Gewöhnliche Differentialgleichungen

Häufig werden numerische Simulationen durchgeführt, um Vorhersagen über das Verhalten eines Systems zu erhalten, beispielsweise über die Entwicklung des Wetters oder Klimas, über das Strömungsverhalten von Luft um einen Flügel oder von Wasser um den Rumpf eines Schiffs oder über die Verformung eines Werkstücks unter Last. Dabei spielt in der Regel die *Zeit* eine wichtige Rolle: Ausgehend von einem bekannten Zustand soll die zukünftige Entwicklung simuliert werden.

In vielen Fällen kann dabei die Veränderung des Zustands des betrachteten Systems in Abhängigkeit von dem aktuellen Zustand ausgedrückt werden, so dass eine *gewöhnliche Differentialgleichung* entsteht.

4.1 Beispiel: Rollende Kugel

Als Beispiel untersuchen wir die Bewegung einer Kugel, die sich auf einer gekrümmten Bahn in zwei Dimensionen frei bewegen kann. Wir wollen die Bewegung der Kugel vorhersagen.

Dazu beschreiben wir zunächst die Bahn durch eine Funktion $g : \mathbb{R} \to \mathbb{R}$: Zu jedem Punkte $x \in \mathbb{R}$ soll $g(x)$ die Höhe der Bahn über diesem Punkt angeben.

Die aktuelle Position der Kugel zu einem Zeitpunkt $t \in \mathbb{R}$ schreiben wir als $x(t)$, sie wird also durch eine Funktion $x : \mathbb{R} \to \mathbb{R}^2$ dargestellt. Entsprechend beschreiben wir die Geschwindigkeit durch eine Funktion $v : \mathbb{R} \to \mathbb{R}^2$ und die wirkende Kraft durch eine Funktion $F : \mathbb{R} \to \mathbb{R}^2$.

Die Newton-Gesetze der Festkörpermechanik besagen, dass die Geschwindigkeit die Ableitung des Orts nach der Zeit ist, dass also

$$x'(t) = v(t) \qquad \text{für alle } t \in \mathbb{R} \tag{4.1}$$

gilt, und dass für eine Einheitsmasse, von der wir hier ausgehen, die Kraft gleich der Ableitung der Geschwindigkeit nach der Zeit ist, dass wir also auch von

$$v'(t) = F(t) \qquad \text{für alle } t \in \mathbb{R} \tag{4.2}$$

ausgehen können. Wenn wir die Kraft F bestimmen können, haben wir eine Gleichung erhalten, die mathematisch analysiert werden kann.

In unserem Fall wirken zwei Kräfte: Einerseits zieht die Gravitation die Kugel nach unten, andererseits verhindert der Kontakt mit der Rollbahn, dass die Kugel unendlich weit senkrecht fällt. Die *Gravitationskraft* ist einfach handzuhaben: Für unser Beispiel setzen wir sie konstant als

$$G := \begin{pmatrix} 0 \\ -1 \end{pmatrix}.$$

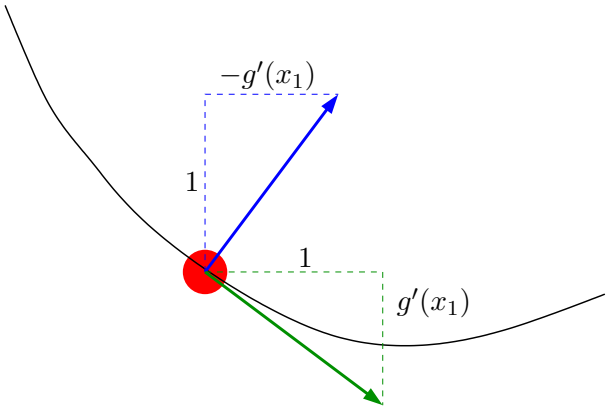

Abbildung 4.1: Tangential- (grün) und Normalenvektor (blau) an der Kurve g in einem Punkt $x = (x_1, x_2) = (x_1, g(x_1))$.

Die Kraft, die dafür sorgt, dass die Kugel im Kontakt mit der Bahn bleibt, ist etwas schwieriger zu berechnen: Sie wirkt senkrecht zur Bahn, also entlang des durch

$$n(x_1) := \begin{pmatrix} -g'(x_1) \\ 1 \end{pmatrix} \qquad \text{für alle } x_1 \in \mathbb{R}$$

definierten *Normalenvektors* (siehe Abbildung 4.1), ihre Stärke hängt allerdings über einen noch unbekannten zeitabhängigen Faktor $\lambda(t)$ von der aktuellen Position und Geschwindigkeit ab, so dass die *Rückstellkraft* durch

$$R(t) := \lambda(t) n(x_1(t)) \qquad \text{für alle } t \in \mathbb{R} \tag{4.3}$$

gegeben ist. Die Kräfte addieren sich zu

$$F(t) = G + R(t) \qquad \text{für alle } t \in \mathbb{R}. \tag{4.4}$$

Der Faktor $\lambda(t)$ muss so gewählt werden, dass zu jedem Zeitpunkt die Kugel auf der Bahn bleibt. Dieser Sachverhalt lässt sich elegant mit Hilfe einer *Niveaufunktion*

$$L : \mathbb{R}^2 \to \mathbb{R}, \qquad x \mapsto x_2 - g(x_1),$$

darstellen, deren Nullstellenmenge offenbar genau aus den Punkten $x \in \mathbb{R}^2$ besteht, die auf dem Graphen der Kurve g liegen. Damit unsere Kugel im Kontakt mit der Bahn bleibt, muss also

$$L(x(t)) = 0 \qquad \text{für alle } t \in \mathbb{R}$$

gelten. Indem wir diese Gleichung zweimal differenzieren erhalten wir

$$DL(x(t))x'(t) = 0 \qquad \text{für alle } t \in \mathbb{R},$$

$$D^2L(x(t))(x'(t),x'(t)) + DL(x(t))x''(t) = 0 \qquad \text{für alle } t \in \mathbb{R},$$

und indem wir $x'(t) = v(t)$ und $x''(t) = v'(t) = F(t)$ einsetzen, erhalten wir

$$\begin{aligned} DL(x(t))v(t) &= 0 && \text{für alle } t \in \mathbb{R}, \\ D^2L(x(t))(v(t),v(t)) + DL(x(t))F(t) &= 0 && \text{für alle } t \in \mathbb{R}, \end{aligned}$$

Für unsere konkrete Niveaufunktion L gelten

$$\begin{aligned} DL(x)y &= \begin{pmatrix} -g'(x_1) & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -g'(x_1)y_1 + y_2, \\ D^2L(x)(y,z) &= \begin{pmatrix} z_1 & z_2 \end{pmatrix} \begin{pmatrix} -g''(x_1) & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -g''(x_1)y_1z_1 \\ &\text{für alle } x,y,z \in \mathbb{R}^2, \end{aligned}$$

also erhalten wir die Gleichung

$$-g''(x_1(t))v_1(t)^2 - g'(x_1(t))F_1(t) + F_2(t) = 0 \qquad \text{für alle } t \in \mathbb{R}.$$

Indem wir unsere Gleichungen (4.4) und (4.3) einsetzen, erhalten wir

$$\begin{aligned} 0 &= -g''(x_1(t))v_1(t)^2 - g'(x_1(t))\lambda(t)(-g'(x_1(t))) + \lambda(t) - 1 \\ &= \lambda(t)(g'(x_1(t))^2 + 1) - (g''(x_1(t))v_1(t)^2 + 1) \qquad \text{für alle } t \in \mathbb{R} \end{aligned}$$

und können

$$\lambda(t) = \frac{g''(x_1(t))v_1(t)^2 + 1}{g'(x_1(t))^2 + 1} \qquad \text{für alle } t \in \mathbb{R}$$

folgern. Insgesamt erhalten wir

$$x'(t) = v(t) \tag{4.5a}$$

$$v'(t) = \frac{g''(x_1(t))v_1(t)^2 + 1}{g'(x_1(t))^2 + 1} \begin{pmatrix} -g'(x_1(t)) \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix} \qquad \text{für alle } t \in \mathbb{R}, \tag{4.5b}$$

eine *gewöhnliche Differentialgleichung*. Da die Ableitung zum Zeitpunkt $t \in \mathbb{R}$ in nichttrivialer Weise von $x(t)$ und $v(t)$ abhängt, ist es in der Regel schwierig, derartige Gleichungen analytisch zu lösen.

4.2 Theorie

Als gewöhnliche Differentialgleichung bezeichnet man Gleichungen der Form

$$y'(t) = f(t,y(t)) \qquad \text{für alle } t \in J, \tag{4.6}$$

wobei y eine stetig differenzierbare Abbildung von einem Intervall $J \subseteq \mathbb{R}$ in eine Teilmenge $\Omega \subseteq E$ eines Banachraums [1] E ist, kurz $y \in C^1(J, \Omega)$, während die Abbildung f die Form

$$f : J \times \Omega \to E, \qquad \Omega \subseteq E$$

besitzt. Gegeben ist dabei in der Regel f, gesucht ist y.

Beispiel 4.1 (Rollende Kugel) *Wir können die Gleichungen unseres Beispiels in die geforderte Form (4.6) bringen, indem wir*

$$y(t) := \begin{pmatrix} x(t) \\ v(t) \end{pmatrix} \qquad \text{für alle } t \in J := \mathbb{R}$$

setzen und die Funktion

$$f : \mathbb{R} \times \mathbb{R}^4 \to \mathbb{R}^4, \qquad (t, y) \mapsto \begin{pmatrix} y_3 \\ y_4 \\ -\frac{g''(y_1)y_3^2+1}{g'(y_1)^2+1} g'(y_1) \\ \frac{g''(y_1)y_3^2+1}{g'(y_1)^2+1} - 1 \end{pmatrix},$$

verwenden. Falls g zweimal stetig differenzierbar ist, ist f eine wohldefinierte stetige Funktion. Falls g m-mal stetig differenzierbar sein sollte, muss f mindestens $(m-2)$-mal stetig differenzierbar sein.

Anschaulich kann man die Gleichung (4.6) so interpretieren, dass sie zu jedem Punkt $y(t)$ der Lösungskurve angibt, in welche „Richtung“ sich die Kurve von diesem Punkt aus bewegen wird. Daraus folgt anschaulich, dass die Gleichung alleine die Lösung y nicht eindeutig definiert: Wir können an einem beliebigen Punkt $y_0 \in \Omega$ zu einem Zeitpunkt $a \in J$ ansetzen und die Kurve verfolgen. Falls wir y_0 und a vorgeben, lässt sich allerdings unter bestimmten Bedingungen eine eindeutige Lösung finden. Ein solches *Anfangswertproblem* ist beschrieben durch

$$y(a) = y_0, \qquad y'(t) = f(t, y(t)) \qquad \text{für alle } t \in J. \tag{4.7}$$

Die Analyse von Anfangswertproblemen, und auch die Konstruktion numerischer Verfahren zu ihrer Lösung, vereinfacht sich erheblich, wenn wir sie in eine äquivalente Form überführen, bei der die formale Forderung der Differenzierbarkeit durch die deutlich schwächere Forderung der Stetigkeit ersetzt wird.

Erinnerung 4.2 (Hauptsatz Integral- und Differentialrechnung) *Sei eine stetig differenzierbare Funktion $f \in C^1([a, b], E)$ auf dem Intervall $[a, b]$ gegeben. Dann gilt*

$$f(b) - f(a) = \int_a^b f'(s)\, ds.$$

[1] Für unsere Zwecke reichen in der Regel die Räume $\mathbb{R}^n$.

Erinnerung 4.3 (Mittelwertsatz Integralrechnung) *Sei eine stetige Funktion $f \in C([a,b],E)$ auf dem Intervall $[a,b]$ gegeben. Dann existiert ein $\eta \in [a,b]$ mit*

$$(b-a)f(\eta) = \int_a^b f(s)\,ds.$$

Lemma 4.4 (Integralgleichung) *Seien $a \in J$, $y_0 \in \Omega$ und $f \in C(J \times \Omega, E)$ gegeben.*

Falls eine Funktion $y \in C^1(J,\Omega)$ Lösung des Anfangswertproblems (4.7) ist, erfüllt sie auch die Integralgleichung

$$y(t) = y_0 + \int_a^t f(s,y(s))\,ds \qquad \text{für alle } t \in J. \tag{4.8}$$

Falls eine Funktion $y \in C(J,\Omega)$ die Integralgleichung erfüllt, ist sie stetig differenzierbar und Lösung des Anfangswertproblems.

Beweis. Sei $y \in C^1(J,\Omega)$ Lösung des Anfangswertproblems (4.7). Aus dem Hauptsatz der Integral- und Differentialrechnung 4.2 folgt

$$y(t) = y(a) + \int_a^t y'(s)\,ds = y_0 + \int_a^t f(s,y(s))\,ds \qquad \text{für alle } t \in J,$$

also die Integralgleichung (4.8).

Sei nun $y \in C(J,\Omega)$ Lösung der Integralgleichung. Dann gilt offenbar

$$y(a) = y_0 + \int_a^a f(s,y(s))\,ds = y_0,$$

die Anfangsbedingung aus (4.7) ist also erfüllt. Um die Ableitung an einer Stelle $t \in J$ zu untersuchen, wählen wir ein $h \in \mathbb{R}$ so, dass $t+h \in J$ gilt. Dann folgt

$$\frac{y(t+h)-y(t)}{h} = \frac{1}{h}\left(\int_a^{t+h} f(s,y(s))\,ds - \int_a^t f(s,y(s))\,ds\right) = \frac{1}{h}\int_t^{t+h} f(s,y(s))\,ds.$$

Mit Hilfe des Mittelwertsatzes der Integralrechnung 4.3 finden wir ein $\eta_h \in J$ zwischen t und $t+h$ so, dass

$$\int_t^{t+h} f(s,y(s))\,ds = hf(\eta_h, y(\eta_h))$$

gilt, und es folgt

$$\frac{y(t+h)-y(t)}{h} = f(\eta_h, y(\eta_h)).$$

Für $h \to 0$ haben wir $\eta_h \to t$, also erhalten wir dank der Stetigkeit der Funktion f

$$\lim_{h\to 0} \frac{y(t+h)-y(t)}{h} = \lim_{h\to 0} f(\eta_h, y(\eta_h)) = f(t,y(t)).$$

Damit ist y in t differenzierbar mit $y'(t) = f(t,y(t))$. Also folgt $y \in C^1(J,E)$, und y erfüllt (4.7). ∎

Die Integralformulierung (4.8) bietet den großen Vorteil, dass sie nicht die Differenzierbarkeit der Funktion y voraussetzt. Auf dieser Formulierung setzt der *Satz von Picard-Lindelöf* auf, der die Existenz und Eindeutigkeit einer Lösung sicherstellt, falls das betrachtete Intervall J nicht zu groß ist. Sein Beweis, den wir hier nur skizzieren, beruht auf dem *Banachschen Fixpunktsatz*.

Erinnerung 4.5 (Banach) *Sei F ein Banachraum, und sei $V \subseteq F$ eine in ihm abgeschlossene Teilmenge. Sei $\Phi : V \to V$ eine Abbildung, die die* Kontraktionseigenschaft *besitzt, für die also eine Konstante $L \in [0, 1)$ mit*

$$\|\Phi(x_1) - \Phi(x_2)\|_F \leq L\|x_1 - x_2\|_F \qquad \text{für alle } x_1, x_2 \in V \tag{4.9}$$

existiert. Dann besitzt Φ genau einen Fixpunkt $x^* \in V$, also ein Element mit $\Phi(x^*) = x^*$.

Der Schlüssel zu unserem Existenzbeweis liegt in der Beobachtung, dass es sich bei der Integralgleichung (4.8) um eine Fixpunktgleichung im Raum der stetigen Funktionen handelt: Die Funktion y tritt sowohl auf der linken als auch auf der rechten Seite auf. Um Erinnerung 4.5 anwenden zu können, müssen wir lediglich eine passende Abbildung Φ konstruieren und nachweisen, dass sie die Kontraktionseigenschaft besitzt. Da wir im Raum der stetigen Funktionen arbeiten, muss Φ eine stetige Funktion auf eine stetige Funktion abbilden. Ein Blick auf (4.8) legt die Definition

$$\Phi[z](t) := y_0 + \int_a^t f(s, z(s))\, ds \qquad \text{für alle } t \in J$$

nahe, mit der wir zu einem $z \in C(J)$ eine neue Funktion $\Phi[z] \in C(J)$ konstruieren können.

Wenn eine Lösung $y \in C(J)$ der Integralgleichung existieren würde, hätten wir

$$\Phi[y](t) = y_0 + \int_a^t f(s, y(s))\, ds = y(t) \qquad \text{für alle } t \in J,$$

also $\Phi[y] = y$, somit ist jede Lösung der Integralgleichung ein Fixpunkt der Abbildung Φ. Die Umkehrung gilt offenbar ebenfalls.

Wir müssen also nur noch dafür sorgen, dass Φ die Kontraktionseigenschaft (4.9) besitzt. Dazu beschränken wir uns auf kompakte Intervalle $[a, b] \subseteq J$ und fordern, dass f *Lipschitz-stetig im zweiten Argument* ist, dass also eine Konstante $L_f \in \mathbb{R}_{\geq 0}$ so existiert, dass

$$\|f(t, x_1) - f(t, x_2)\| \leq L_f\|x_1 - x_2\| \qquad \text{für alle } t \in [a, b],\ x_1, x_2 \in E \tag{4.10}$$

gilt. Mit dieser Voraussetzung erhalten wir

$$\begin{aligned}\|\Phi[z_1](t) - \Phi[z_2](t)\| &= \left\| y_0 + \int_a^t f(s, z_1(s))\, ds - y_0 - \int_a^t f(s, z_2(s))\, ds \right\| \\ &= \left\| \int_a^t f(s, z_1(s)) - f(s, z_2(s))\, ds \right\|\end{aligned}$$

$$\leq \int_a^t \|f(s, z_1(s)) - f(s, z_2(s))\| \, ds$$
$$\leq \int_a^t L_f \|z_1(s) - z_2(s)\| \, ds \qquad \text{für alle } z_1, z_2 \in C[a,b]. \quad (4.11)$$

Diese Abschätzung können wir in die von uns benötigte Form bringen, indem wir die richtige Norm verwenden, nämlich die *Supremumsnorm*

$$\|z\|_{\infty,[a,b]} := \sup\{\|z(t)\| \ : \ t \in [a,b]\} \qquad \text{für alle } z \in C[a,b],$$

mit deren Hilfe (4.11) die kurze Form

$$\begin{aligned}\|\Phi[z_1] - \Phi[z_2]\|_{\infty,[a,b]} &\leq \sup\left\{\int_a^t L_f \|z_1(s) - z_2(s)\| \, ds \ : \ t \in [a,b]\right\} \\ &\leq \sup\left\{\int_a^t L_f \|z_1 - z_2\|_{\infty,[a,b]} \, ds \ : \ t \in [a,b]\right\} \\ &= L_f(b-a)\|z_1 - z_2\|_{\infty,[a,b]} \qquad \text{für alle } z_1, z_2 \in C[a,b]\end{aligned}$$

annimmt, wir haben also

$$\|\Phi[z_1] - \Phi[z_2]\|_{\infty,[a,b]} \leq L\|z_1 - z_2\|_{\infty,[a,b]} \qquad \text{für alle } z_1, z_2 \in C[a,b]$$

mit $L := L_f(b-a)$ bewiesen. Damit die Kontraktionseigenschaft gilt, müssen wir $L < 1$ sicherstellen. Im Sonderfall $L_f = 0$ ist das offenbar immer trivial erfüllt, aber dieser Fall ist nicht besonders interessant. Falls $L_f > 0$ gilt, können wir die Kontraktionseigenschaft sicherstellen, indem wir fordern, dass das Intervall klein genug ist, also

$$b - a < 1/L_f.$$

Die Lipschitz-Stetigkeit der Funktion f zu fordern ist zwar für den Beweis ausreichend, aber häufig etwas unhandlich. Da Differenzierbarkeit die Lipschitz-Stetigkeit impliziert (nachweisbar durch Kombination der Erinnerungen 4.2 und 4.3), können wir unsere Aussage wesentlich eleganter formulieren, indem wir voraussetzen, dass f stetig differenzierbar ist:

Erinnerung 4.6 (Picard-Lindelöf) *Sei J offen und $f : J \times \Omega \to E$ stetig differenzierbar. Dann existieren eine offene Menge $\widehat{J} \subseteq J$ mit $a \in \widehat{J}$ und eine stetig differenzierbare Funktion $\hat{y} \in C^1(\widehat{J}, \Omega)$ mit*

$$y(a) = y_0, \qquad y'(t) = f(t, y(t)) \qquad \textit{für alle } t \in \widehat{J}.$$

Jede weitere Lösung dieses Anfangswertproblems stimmt mit y überein.

In dem Modellproblem aus Beispiel 4.1 genügt so die dreifache stetige Differenzierbarkeit der Funktion g, um die Existenz und Eindeutigkeit zumindest lokaler Lösungen zu garantieren.

4.3 Einschrittverfahren

Wir interessieren uns für Verfahren, mit deren Hilfe wir die Lösung y des Anfangswertproblems (4.7) konkret berechnen oder wenigstens approximieren können. Ein Ausgangspunkt für die Konstruktion solcher Verfahren ist die Integralgleichung (4.8): Die Lösung in einem Punkt $t \in J$ mit $t \geq a$ ist durch

$$y(t) = y_0 + \int_a^t f(s, y(s))\,ds$$

gegeben, falls wir also das Integral approximieren könnten, könnten wir auch die Lösung angeben. Es bietet sich an, eine Quadraturformel zu verwenden, die zu einem Ergebnis der Form

$$y(t) \approx y_0 + \sum_{i=1}^{m} w_i f(s_i, y(s_i))$$

mit Gewichten $w_1, \ldots, w_m$ und Quadraturpunkten $s_1, \ldots, s_m$ führen würde. Leider hilft uns diese Formel nicht viel weiter, denn wir kennen die Werte der Funktion y in den Quadraturpunkten im Allgemeinen nicht. In *einem* Punkt kennen wir den Wert allerdings nach Definition: Es gilt $y(a) = y_0$, also können wir eine besonders einfache Quadraturformel mit nur einem einzigen Quadraturpunkt verwenden:

$$y(t) \approx \tilde{y}_{\mathrm{eu}}(t) := y_0 + (t-a) f(a, y_0). \tag{4.12}$$

Diese Näherung wird allerdings in den meisten Fällen sehr ungenau sein.

Falls y zweimal stetig differenzierbar ist, können wir mit Hilfe des Mittelwertsatzes der Integralrechnung (vgl. Erinnerung 4.3) und des Hauptsatzes der Integral- und Differentialrechnung (vgl. Erinnerung 4.2) immerhin die Aussage

$$\begin{aligned} y(t) - \tilde{y}_{\mathrm{eu}}(t) &= y_0 + \int_a^t f(s, y(s))\,ds - y_0 - \int_a^t f(a, y_0)\,ds \\ &= \int_a^t y'(s) - y'(a)\,ds = (t-a)(y'(\eta) - y'(a)) \\ &= (t-a)\int_a^\eta y''(s)\,ds = (t-a)(\eta - a) y''(\hat{\eta}) \end{aligned}$$

für $\eta \in [a,t]$, $\hat{\eta} \in [a,\eta] \subseteq [a,b]$ erhalten, der wir unmittelbar

$$\|y - \tilde{y}_{\mathrm{eu}}\|_{\infty,[a,b]} \leq (b-a)^2 \|y''\|_{\infty,[a,b]} \tag{4.13}$$

entnehmen. Für kurze Intervalle sollte unser Ansatz also zu brauchbaren Ergebnissen führen, für lange wirkt er weniger vertrauenswürdig.

Wir umgehen dieses Problem, indem wir das Intervall $[a,b]$ in Teilintervalle zerlegen, die wir so klein wählen können, dass sich der auf jedem einzelnen Intervall eingeführte Fehler beherrschen lässt. Für unsere Zwecke genügt einen *äquidistante Zerlegung*: Wir wählen ein $n \in \mathbb{N}$, setzen $h := (b-a)/n$ und

$$t_i := a + hi \qquad \text{für all } i \in \{0, \ldots, n\}.$$

Unser Algorithmus besteht darin, die Näherung (4.12) zuerst auf dem Intervall $[t_0, t_1]$ anzuwenden, um eine Näherungslösung

$$\tilde{y}(t_1) := y_0 + hf(t_0, y_0)$$

zu berechnen. Diese Näherungslösung verwenden wir dann als Ausgangspunkt für die Berechnung auf dem nächsten Intervall $[t_1, t_2]$ und erhalten

$$\tilde{y}(t_2) := \tilde{y}(t_1) + hf(t_1, \tilde{y}(t_1)).$$

In dieser Weise können wir fortfahren und erhalten das folgende Näherungsverfahren:

$$\tilde{y}(t_0) := y_0, \qquad \tilde{y}(t_{i+1}) := \tilde{y}(t_i) + hf(t_i, \tilde{y}(t_i)) \qquad \text{für alle } i \in \{0, \ldots, n-1\}.$$

Der Algorithmus ist leicht umzusetzen: In jedem Schritt müssen wir lediglich einmal die Funktion f auswerten und eine Linearkombination bilden. Allerdings sehen wir an (4.13) auch, dass wir keine besonders hohe Genauigkeit in jedem einzelnen Schritt erwarten dürfen, so dass auch die Genauigkeit des gesamten Verfahrens nicht allzu hoch sein dürfte. Glücklicherweise lässt sich der Ansatz etwas verallgemeinern und so eine wesentlich höhere Genauigkeit erreichen:

Definition 4.7 (Einschrittverfahren) *Sei* $h_0 \in \mathbb{R}_{>0} \cup \{\infty\}$*, und sei*

$$\Delta := \{(t, h) \ : \ t \in [a, b],\ h \in [0, h_0],\ t + h \in [a, b]\}.$$

Eine Funktion $\Psi : \Delta \times \Omega \to E$ *nennen wir* Inkrementfunktion*. Für jedes* $n \in \mathbb{N}$ *mit* $h := (b-a)/n \leq h_0$ *definiert sie ein* Einschrittverfahren *zur Berechnung der Näherungswerte*

$$\begin{aligned} \hat{y}(t_0) &:= y_0, \\ \hat{y}(t_{i+1}) &:= \hat{y}(t_i) + h\Psi(t_i, h, \hat{y}(t_i)) \qquad \text{für alle } i \in \{0, \ldots, n-1\} \end{aligned}$$

der Lösung des Anfangswertproblems (4.7) in den Punkten $a = t_0, t_1, \ldots, t_n = b$*.*

Ein besonders einfaches Einschrittverfahren haben wir bereits kennen gelernt:

Definition 4.8 (Explizites Euler-Verfahren) *Das Einschrittverfahren mit der Inkrementfunktion*

$$\Psi : \Delta \times \Omega \to E, \qquad (t, h, x) \mapsto f(t, x),$$

nennen wir das explizite Euler-Verfahren*.*

Der Name *explizites* Euler-Verfahren legt nahe, dass es auch ein *implizites* Euler-Verfahren gibt, und das ist auch der Fall: Wenn wir das Integral durch den *rechten* Randpunkt approximieren, also

$$y(t_{i+1}) \approx y(t_i) + hf(t_{i+1}, y(t_{i+1}))$$

verwenden. Wir kennen $y(t_{i+1})$ nicht, aber wir können es als Lösung des, im Allgemeinen nichtlinearen, Gleichungssystems

$$y(t_{i+1}) - hf(t_{i+1}, y(t_{i+1})) \approx y(t_i)$$

auffassen und eine Näherung durch

$$\tilde{y}_{\mathrm{im}}(t_{i+1}) - hf(t_{i+1}, \tilde{y}_{\mathrm{im}}(t_{i+1})) = \tilde{y}(t_i) \tag{4.14}$$

definieren, falls das System lösbar ist. Die Lösbarkeit lässt sich auf den Fixpunktsatz von Banach zurückführen, indem wir das Gleichungssystem als Fixpunktgleichung schreiben:

$$\begin{aligned}\tilde{y}_{\mathrm{im}}(t_{i+1}) &= \tilde{y}(t_i) + hf(t_{i+1}, \tilde{y}_{\mathrm{im}}(t_{i+1})) = \Phi(\tilde{y}(t_{i+1})),\\ \Phi(x) &:= \tilde{y}(t_i) + hf(t_{i+1}, x).\end{aligned}$$

Falls wir wieder voraussetzen, dass f Lipschitz-stetig im zweiten Argument ist, also die Abschätzung (4.10) gilt, haben wir

$$\begin{aligned}\|\Phi(x_1) - \Phi(x_2)\| &= \|\tilde{y}(t_i) + hf(t_{i+1}, x_1) - \tilde{y}(t_i) - hf(t_{i+1}, x_2)\|\\ &= \|hf(t_{i+1}, x_1) - hf(t_{i+1}, x_2)\| = h\|f(t_{i+1}, x_1) - f(t_{i+1}, x_2)\|\\ &\leq L_f h\|x_1 - x_2\| \qquad \text{für alle } x_1, x_2 \in E,\end{aligned}$$

für $L_f = 0$ oder $h < 1/L_f$ wird also Φ eine Kontraktion sein, so dass nach Erinnerung 4.5 die gesuchte Lösung $\tilde{y}_{\mathrm{im}}(t_{i+1})$ existiert und eindeutig ist.

In diesem Fall können wir die Gleichung in der Form

$$(I - hf(t_{i+1}, \cdot))(\tilde{y}_{\mathrm{im}}(t_{i+1})) = \tilde{y}(t_i)$$

schreiben und aus der Lösbarkeit folgern, dass die Abbildung auf der rechten Seite eine Inverse $(I - hf(t_{i+1}, \cdot))^{-1}$ besitzen muss. Mir ihr erhalten wir

$$\tilde{y}_{\mathrm{im}}(t_{i+1}) = (I - hf(t_{i+1}, \cdot))^{-1}(\tilde{y}(t_i)) = \tilde{y}(t_i) + h\left(\frac{(I - hf(t_{i+1}, \cdot))^{-1}(\tilde{y}(t_i)) - \tilde{y}(t_i)}{h}\right)$$

und können die Inkrementfunktion des impliziten Euler-Verfahrens direkt ablesen.

Definition 4.9 (Implizites Euler-Verfahren) *Gelte (4.10), und sei $L_f = 0$ oder $h_0 < 1/L_f$. Das Einschrittverfahren mit der Inkrementfunktion*

$$\Psi : \Delta \times \Omega \to E, \qquad (t, h, x) \mapsto \frac{(I - hf(t, \cdot))^{-1}(x) - x}{h},$$

nennen wir das implizite Euler-Verfahren. *Praktisch durchgeführt wird es in der Regel, indem das Gleichungssystem (4.14) gelöst wird.*

Ähnlich wie im Fall des expliziten Euler-Verfahrens lässt sich auch für das implizite Verfahren zeigen, dass die Fehlerabschätzung (4.13) für jeweils einzelne Schritte gilt. Wir sind daran interessiert, den Exponenten des von der Länge des Intervalls abhängenden Terms zu erhöhen, denn dann würde eine Verkleinerung des Intervalls zu einer besseren Reduktion des Fehlers führen.

Eine Möglichkeit besteht darin, ein anderes Quadraturverfahren zu verwenden: Wir gehen wieder von der Formel

$$y(t) = y_0 + \int_a^t f(s, y(s))\, ds$$

aus, approximieren das Integral aber jetzt mit der Mittelpunktregel

$$y(t) \approx \tilde{y}_q(t) := y_0 + (t-a) f((a+t)/2, y((a+t)/2)).$$

Für diese Regel ist bekannt, dass

$$\|y - \tilde{y}_q\|_{\infty,[a,b]} \leq (b-a)^3 \frac{\|y^{(3)}\|_{\infty,[a,b]}}{24}$$

gilt, falls y dreimal stetig differenzierbar ist. In diesem Fall hätten wir also unser Ziel erreicht und einen Exponenten von 3 anstatt von 2.

Leider können wir $\tilde{y}_q$ nicht berechnen, weil uns der Wert $y((a+t)/2)$ im Mittelpunkt des Intervalls nicht zur Verfügung steht. Allerdings können wir diesen Wert mit Hilfe des expliziten Euler-Verfahrens durch

$$y((a+t)/2) \approx \tilde{y}_e((a+t)/2) := y_0 + \frac{a+t}{2} f(a, y_0)$$

annähern und wissen dank (4.13), dass

$$\|y - \tilde{y}_e\|_{\infty,[a,b]} \leq \frac{(b-a)^2}{4} \|y''\|_{\infty,[a,b]}$$

gilt. Auf diesem Weg erhalten wir

$$\begin{aligned} y(t) &\approx \tilde{y}_{\mathrm{ru}}(t) := y_0 + (t-a) f((a+t)/2, \tilde{y}_e((a+t)/2)) \\ &= y_0 + (t-a) f((a+t)/2, y_0 + (a+t)/2 f(a, y_0)), \end{aligned}$$

und diese Formel können wir praktisch auswerten. Es ist noch zu prüfen, ob wir mit dieser aufwendigeren Formel tatsächlich etwas gewonnen haben. Für $\tilde{y}_q$ wissen wir bereits, dass es gut ist, also genügt es, den Unterschied zwischen $\tilde{y}_q$ und $\tilde{y}$ zu untersuchen. Indem wir wieder voraussetzen, dass f im zweiten Argument Lipschitz-stetig ist, also (4.10) gilt, erhalten wir

$$\begin{aligned} \|\tilde{y}_q(t) - \tilde{y}_{\mathrm{ru}}(t)\| &= \|y_0 + (t-a) f((t+a)/2, y((t+a)/2)) \\ &\quad - y_0 + (t-a) f((t+a)/2, \tilde{y}_e((t+a)/2)\| \\ &= (t-a) \|f((t+a)/2, y((t+a)/2)) - f((t+a)/2, \tilde{y}_e((t+a)/2))\| \end{aligned}$$

$$\leq (t-a)L_f\|y((t+a)/2) - \tilde{y}_e((t+a)/2)\|$$
$$\leq (b-a)L_f\frac{(b-a)^2}{4}\|y''\|_{\infty,[a,b]} = (b-a)^3\frac{L_f}{4}\|y''\|_{\infty,[a,b]}.$$

Indem wir diese Abschätzung mit der des Quadraturfehlers kombinieren ergibt sich das Endergebnis

$$\|y - \tilde{y}_{\mathrm{ru}}\|_{\infty,[a,b]} \leq (b-a)^3\left(\frac{\|y^{(3)}\|_{\infty,[a,b]}}{24} + \frac{L_f\|y''\|_{\infty,[a,b]}}{4}\right), \tag{4.15}$$

wir haben also tatsächlich ein besseres Verfahren erhalten.

Definition 4.10 (Runge-Verfahren) *Das Einschrittverfahren mit der Inkrementfunktion*

$$\Psi : \Delta \times \Omega \to E, \qquad (t,h,x) \mapsto f(t+h/2, x+h/2f(t,x)),$$

nennen wir das Runge-Verfahren *oder das* Euler-Collatz-Verfahren*.*

Das Runge-Verfahren konvergiert zwar schneller als das Euler-Verfahren, allerdings erfordert es pro Schritt auch zwei Auswertungen der Funktion f, und da diese Auswertungen in der Praxis häufig den Rechenaufwand bestimmen, kann man grob abschätzen, dass ein Schritt des Runge-Verfahrens ungefähr so aufwendig wie zwei Schritte des Euler-Verfahrens ist. Trotzdem lohnt sich das Runge-Verfahren: Die Genauigkeit, die das Euler-Verfahren in n Schritten erreicht, erreicht das Runge-Verfahren mit einer zu $\sqrt{n}$ proportionalen Schrittzahl.

Wenn wir bereit sind, weitere Auswertungen der Funktion f in Kauf zu nehmen, können wir die Konvergenz weiter beschleunigen:

Definition 4.11 (Runge-Kutta-Verfahren) *Sei $m \in \mathbb{N}$, seien $A \in \mathbb{R}^{m\times m}$, $b, c \in \mathbb{R}^m$ gegeben. Mit den durch*

$$k_i(t,h,x) := f\left(t + c_i h, x + h\sum_{j=1}^{m} a_{ij}k_j(t,h,x)\right) \qquad \text{für alle } i \in \{1,\ldots,m\}$$

definierten Hilfsgrößen bezeichnen wir

$$\Psi : \Delta \times \Omega \to E, \qquad (t,h,x) \mapsto \sum_{i=1}^{m} b_i k_i(t,h,x),$$

als die Inkrementfunktion des durch A, b und c definierten Runge-Kutta-Verfahrens*, und das zugehörige Einschrittverfahren als das entsprechende* Runge-Kutta-Verfahren*. Die Zahl m nennt man die* Stufenzahl *des Verfahrens.*

Im Allgemeinen sind die Größen $k_1, \ldots, k_m$ die Lösung eines nichtlinearen Gleichungssystems, die nicht unbedingt existieren muss. Ein wichtiger Spezialfall sind deshalb die

expliziten Runge-Kutta-Verfahren*, bei denen A eine strike untere Dreiecksmatrix ist, so dass sich*

$$k_i(t,h,x) = f\left(t + c_i h, x + h\sum_{j=1}^{i-1} a_{ij} k_j(t,h,x)\right) \qquad \text{für alle } i \in \{1,\dots,m\}$$

ergibt und sich die k_i der Reihe nach direkt berechnen lassen.

Runge-Kutta-Verfahren werden in der Regel durch ihr *Butcher-Tableau* beschrieben, in dem die Matrix und die beiden Vektoren in der Form

$$\begin{array}{c|ccc} c_1 & a_{11} & \dots & a_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ c_m & a_{m1} & \dots & a_{mm} \\ \hline & b_1 & \dots & b_m \end{array}$$

zusammengefasst sind. Dem expliziten Euler-Verfahren entspricht dabei das Tableau

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array},$$

dem impliziten das Tableau

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array},$$

bei dem A keine strikte Dreiecksmatrix ist, und dem Runge-Verfahren das Tableau

$$\begin{array}{c|cc} 0 & 0 & \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array}.$$

Das *klassische Runge-Kutta-Verfahren* ist durch die Wahl

$$\begin{array}{c|cccc} 0 & 0 & & & \\ 1/2 & 1/2 & 0 & & \\ 1/2 & 0 & 1/2 & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

definiert und erreicht, bei hinreichend oft differenzierbarer Lösung, einen Fehler, der wie n^{-4} fällt.

Höherstufige Runge-Kutta-Verfahren sind im Allgemeinen nicht ganz einfach zu konstruieren und erfordern wegen der vielen Auswertungen der Funktion f einen hohen Rechenaufwand. Deshalb empfiehlt es sich, nach Alternativen zu suchen.

4.4 Mehrschrittverfahren

Bei den Einschrittverfahren wird die Näherung $\tilde{y}(t_{i+1})$ direkt aus der Näherung $\tilde{y}(t_i)$ des unmittelbar vorangehenden Zeitschritts berechnet. Die Idee der *Mehrschrittverfahren* besteht darin, auch weiter in der Vergangenheit liegende Näherungswerte in die Berechnung einfließen zu lassen und so die Effizienz zu verbessern.

Adams-Bashforth-Verfahren. Für die Herleitung gehen wir davon aus, dass uns nicht nur $y(t_i)$, sondern auch $y(t_{i-1}), \dots, y(t_{i-r+1})$ zur Verfügung stehen. Wir wollen wieder das Integral

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} y'(s)\,ds = y(t_i) + \int_{t_i}^{t_{i+1}} f(s, y(s))\,ds$$

approximieren, um eine Näherung des Werts $y(t_{i+1})$ zu erhalten. Dazu ersetzen wir den Integranden

$$g : [a, b] \to E, \qquad s \mapsto y'(s) = f(s, y(s))$$

durch ein Polynom $p \in \Pi_{r-1}$ des Grades $r - 1$, das die Funktion in den Punkten $t_{i-r+1}, \dots, t_i$ interpoliert. Mit Hilfe der Lagrange-Polynome

$$\ell_{i,j}(s) := \prod_{\substack{k=0 \\ k \neq j}}^{r-1} \frac{s - t_{i-k}}{t_{i-j} - t_{i-k}}$$

können wir dieses Polynom in der Form

$$p_i = \sum_{j=0}^{r-1} g(t_{i-j}) \ell_{i,j}$$

schreiben, und indem wir es als Näherung des Integranden verwenden folgt

$$\begin{aligned} y(t_{i+1}) &= y(t_i) + \int_{t_i}^{t_{i+1}} g(s)\,ds \approx y(t_i) + \int_{t_i}^{t_{i+1}} p_i(s)\,ds \\ &= y(t_i) + \sum_{j=0}^{r-1} g(t_{i-j}) \int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds \\ &= y(t_i) + \sum_{j=0}^{r-1} f(t_{i-j}, y(t_{i-j})) \int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds, \end{aligned}$$

wir benötigen also lediglich die Werte der Funktion g in den Punkten $t_{i-r+1}, \dots, t_i$, um eine Näherung der Lösung y im Punkt t_{i+1} zu berechnen.

Koeffizienten. Es ist unpraktisch, die Integrale der Lagrange-Polynome für jeden Zeitschritt neu zu berechnen. Glücklicherweise können wir das vermeiden, weil wir vorausgesetzt haben, dass unsere Zeitpunkt äquidistant sind: Es gilt

$$\begin{aligned}\ell_{i,j}(s) &= \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{s-t_{i-k}}{t_{i-j}-t_{i-k}} = \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{s-(a+h(i-k))}{(a+h(i-j))-(a+h(i-k))} \\ &= \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{s-(a+hi)+hk}{h(k-j)} = \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{(s-t_i)+hk}{h(k-j)} \qquad \text{für alle } s\in\mathbb{R},\end{aligned}$$

und mit einer Variablentransformation folgt

$$\begin{aligned}\int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds &= \int_{t_i}^{t_i+h} \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{(s-t_i)+hk}{h(k-j)}\,ds = \int_0^h \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{s+hk}{h(k-j)}\,ds \\ &= h\int_0^1 \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{hs+hk}{h(k-j)}\,ds = h\int_0^1 \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{s+k}{k-j}\,ds,\end{aligned}$$

also hängt das Integral nicht von i ab, so dass wir es nicht für jeden Schritt neu berechnen müssen. Indem wir die Konstanten

$$w_j := \int_0^1 \prod_{\substack{k=0\\k\neq j}}^{r-1} \frac{s+k}{k-j}\,ds \qquad \text{für alle } j\in\{0,\dots,r-1\} \tag{4.16}$$

vorberechnen, erhalten wir die Approximation

$$y(t_{i+1}) \approx \tilde{y}_{\mathrm{ab}}(t_{i+1}) := y(t_i) + h\sum_{j=0}^{r-1} w_j f(t_{i-j}, y(t_{i-j})).$$

Uns sind die exakten Werte $y(t_i),\dots,y(t_{i-r+1})$ in der Regel nicht bekannt, deshalb ersetzen wir sie durch die im Zuge des Verfahrens berechneten Näherungswerte, um einen praktisch durchführbaren Algorithmus zu erhalten.

Definition 4.12 (Adams-Bashforth-Verfahren) *Sei $r\in\mathbb{N}$, und seien Gewichte $w_0,\dots,w_{r-1}$ wie in (4.16) definiert.*

Seien Näherungen $\tilde{y}(t_0),\dots,\tilde{y}(t_{r-1})\in E$ der Lösung y des Anfangswertproblems (4.7) gegeben. Das Adams-Bashforth-r-Schritt-Verfahren *berechnet mit der Vorschrift*

$$\tilde{y}(t_{i+1}) := \tilde{y}(t_i) + h\sum_{j=0}^{r-1} w_j f(t_{i-j}, \tilde{y}(t_{i-j})) \qquad \textit{für alle } i\in\{r-1,\dots,n\}$$

Näherungen der Lösung y in allen weiteren Punkten.

r	w_0	w_1	w_2	w_3
1	1			
2	3/2	−1/2		
3	23/12	−4/3	5/12	
4	55/24	−59/24	37/24	−9/24

Tabelle 4.1: Koeffizienten w_j des Adams-Bashforth-Verfahrens für $r \in \{1, \ldots, 4\}$.

An dieser Stelle ist wichtig, zwischen zwei Näherungslösungen zu unterscheiden: Die Näherung $\tilde{y}_{\mathrm{ab}}$ berechnet sich aus den *exakten* vorangehenden Werten, während die Näherung $\tilde{y}$ auf den bereits genäherten Werten beruht, also vermutlich wesentlich weniger genau sein dürfte.

Für $r \in \{1, \ldots, 4\}$ sind die Gewichte w_j in der Tabelle 4.1 zusammengestellt. Da die Gewichte als Integrale der jeweiligen Lagrange-Polynome entstehen, deren Summe immer eins ist, muss auch die Summe der Gewichte jeweils eins ergeben.

Effiziente Implementierung. Auf den ersten Blick wirkt das Adams-Bashforth-Verfahren wenig attraktiv, weil die Durchführung jedes Schritts scheinbar r Auswertungen der Funktion f erforderlich macht. Eine genauere Untersuchung zeigt allerdings, dass ein und derselbe Funktionswerte in der Regel in r Zeitschritten verwendet wird, so dass wir ihn zwischenspeichern können. Damit erhalten wir den in Abbildung 4.2 angegebenen Algorithmus.

procedure adams_bashforth(r, f, **var** $\tilde{y}$);
for $i \in \{0, \ldots, r-1\}$ **do begin**
  Berechne $\tilde{y}(t_i)$ und $\tilde{g}(t_i) := f(t_i, \tilde{y}(t_i))$
end;
for $i = r-1$ **to** $n-1$ **do begin**
  $\tilde{y}(t_{i+1}) \leftarrow \tilde{y}(t_i) + \sum_{j=0}^{r-1} w_j \tilde{g}(t_{i-j})$;
  $\tilde{g}(t_{i+1}) \leftarrow f(t_{i+1}, \tilde{y}(t_{i+1}))$
end

Abbildung 4.2: Pseudocode des Adams-Bashforth-Verfahrens

Sobald die Näherungen für die ersten r Werte der Lösung in der ersten Schleife berechnet wurden, erfordert die zweite Schleife, die für großes n die überwiegende Anzahl der Zeitschritte bearbeitet, lediglich *eine* weitere Auswertung der Funktion f pro Schritt, wäre also ausgesprochen sparsam, falls eine angemessene Genauigkeit erreicht wird.

Fehlerabschätzung. Im Rahmen der Herleitung des Adams-Bashforth-Verfahrens haben wir lediglich die Funktion

$$g : [a, b] \to E, \qquad s \mapsto f(s, y(s)) = y'(s),$$

durch ein Interpolationspolynom p in den Punkten $t_i, \dots, t_{i-r+1}$ ersetzt. Für den Interpolationsfehler gilt die folgende allgemeine Aussage:

Erinnerung 4.13 (Interpolationsfehler) *Sei $g \in C^{m+1}([a,b], E)$ eine Funktion, und sei p ein Polynom m-ten Grades, das g in paarweise verschiedenen Punkten $x_0, \dots, x_m \in [a,b]$ interpoliert. Für jedes $x \in [a,b]$ existiert dann ein $\eta \in [a,b]$ mit*

$$g(x) - p(x) = (x - x_0) \dots (x - x_m) \frac{g^{(m+1)}(\eta)}{(m+1)!}.$$

Wir setzen voraus, dass g r-mal stetig differenzierbar ist, also muss wegen $g(s) = y'(s)$ die Lösung des Anfangswertproblems $(r+1)$-mal stetig differenzierbar sein. In diesem Fall folgt aus Erinnerung 4.13 für jedes $s \in [t_i, t_{i+1}]$ die Existenz eines $\eta \in [t_{i-r+1}, t_{i+1}]$, für das die Abschätzung

$$\begin{aligned}
\|g(s) - p(s)\| &\leq |s - t_i| \dots |s - t_{i-r+1}| \frac{\|g^{(r)}(\eta)\|}{r!} = |s - t_i| \dots |s - t_{i-r+1}| \frac{\|y^{(r+1)}(\eta)\|}{r!} \\
&\leq |s - t_i| \; |s - t_{i-1}| \dots |s - t_{i-r+1}| \frac{\|y^{(r+1)}\|_{\infty,[a,b]}}{r!} \\
&= |s - t_i| \; |s - t_i + h| \dots |s - t_i + (r-1)h| \frac{\|y^{(r+1)}\|_{\infty,[a,b]}}{r!} \\
&\leq h(2h) \dots (rh) \frac{\|y^{(r+1)}\|_{\infty,[a,b]}}{r!} = h^r \|y^{(r+1)}\|_{\infty,[a,b]}
\end{aligned}$$

gilt. Durch Integration dieser Fehlerabschätzung erhalten wir

$$\begin{aligned}
\|y(t_{i+1}) - \tilde{y}_{\mathrm{ab}}(t_{i+1})\| &= \left\| y(t_i) + \int_{t_i}^{t_{i+1}} g(s)\, ds - y(t_i) - \int_{t_i}^{t_{i+1}} p(s)\, ds \right\| \\
&\leq \int_{t_i}^{t_{i+1}} \|g(s) - p(s)\|\, ds \leq h^{r+1} \|y^{(r+1)}\|_{\infty,[a,b]}.
\end{aligned}$$

An dieser Stelle sollte noch einmal darauf hingewiesen werden, dass $\tilde{y}_{\mathrm{ab}}$ aus den *exakten* Werten der Lösung in den Punkten $t_i, \dots, t_{i-r+1}$ berechnet wird, uns also in der Praxis so gut wie nie zur Verfügung steht. Die soeben hergeleitete Abschätzung beschreibt also nur den *lokalen* Fehler, der sich ergibt, wenn wir einen einzelnen Schritt des Verfahrens ausgehend von exakten Werten durchführen.

Die Analyse des Gesamtfehlers können wir per Induktion (unter Zuhilfenahme einer noch näher zu diskutierenden Stabilitätsaussage) ausgehend von diesen lokalen Abschätzungen entwickeln.

Adams-Moulton-Verfahren. Ähnlich wie im Fall des Euler-Verfahrens können wir auch implizite Mehrschrittverfahren konstruieren, indem wir den unbekannten Wert $y(t_{i+1})$ in die Approximation des Integrals einfließen lassen. Wir gehen wieder von der Funktion

$$g : [a,b] \to E, \qquad s \mapsto f(s, y(s)) = y'(s),$$

aus, die wir diesmal mit einem Polynom $p \in \Pi_r$ r-ter Ordnung in den Punkten $t_{i+1}, t_i, \ldots, t_{i-r+1}$ interpolieren. Wir erhalten

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} g(s)\,ds \approx y(t_i) + \int_{t_i}^{t_{i+1}} p(s)\,ds$$

und können die Approximation mit Hilfe der Lagrange-Darstellung

$$p = \sum_{j=-1}^{r-1} g(t_{i-j})\ell_{i,j},$$
$$\ell_{i,j}(s) := \prod_{\substack{k=-1\\k\neq j}}^{r-1} \frac{s - t_{i-k}}{t_{i-j} - t_{i-k}}$$

in der Form

$$y(t_{i+1}) \approx \sum_{j=-1}^{r-1} g(t_{i-j}) \int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds$$

schreiben. Die Integrale der Lagrange-Polynome sind wegen

$$\begin{aligned}\int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds &= \int_{t_i}^{t_{i+1}} \prod_{\substack{k=-1\\k\neq j}}^{r-1} \frac{s - t_{i-k}}{t_{i-j} - t_{i-k}}\,ds = \int_{t_i}^{t_{i+1}} \prod_{\substack{k=-1\\k\neq j}}^{r-1} \frac{s - t_i + kh}{t_i - hj - t_i + hk}\,ds \\ &= h\int_0^1 \prod_{\substack{k=-1\\k\neq j}}^{r-1} \frac{t_i + hs - t_i + kh}{h(k-j)}\,ds = h\int_0^1 \prod_{\substack{k=-1\\k\neq j}}^{r-1} \frac{s+k}{k-j}\,ds\end{aligned}$$

wieder von i unabhängig, so dass wir

$$\begin{aligned} y(t_{i+1}) &\approx y(t_i) + h\sum_{j=-1}^{r-1} w_j f(t_{i-j}, y(t_{i-j})), \\ w_j &:= \int_0^1 \prod_{\substack{k=-1\\k\neq j}}^{r-1} \frac{s+k}{k-j}\,ds \qquad \text{für alle } j \in \{-1, \ldots, r-1\} \end{aligned} \tag{4.17}$$

erhalten. Die Näherungslösung für t_{i+1} ist die Lösung der Fixpunktgleichung

$$\tilde{y}_{\mathrm{am}}(t_{i+1}) = y(t_i) + hw_{-1}f(t_{i+1}, \tilde{y}_{\mathrm{am}}(t_{i+1})) + h\sum_{j=0}^{r-1} w_j f(t_{i-j}, y(t_{i-j})),$$

die, wie im Fall des impliziten Euler-Verfahrens, für hinreichend kleine Zeitschrittweiten h eindeutig lösbar ist, falls f die Lipschitz-Bedingung (4.10) erfüllt.

Definition 4.14 (Adams-Moulton-Verfahren) *Sei $r \in \mathbb{N}$, und seien die Gewichte $w_{-1}, \dots, w_{r-1}$ wie in (4.17) definiert. Seien Näherungen $\tilde{y}(t_0), \dots, \tilde{y}(r_{r-1}) \in E$ der Lösung y des Anfangswertproblems (4.7) gegeben. Das* Adams-Moulton-r-Schritt-Verfahren *ist durch die Vorschrift*

$$\tilde{y}(t_{i+1}) = \tilde{y}(t_i) + h \sum_{j=-1}^{r-1} w_j f(t_{i-j}, \tilde{y}(t_{i-j})) \qquad \text{für alle } i \in \{r-1, \dots, n\}$$

definiert, bei der zu beachten ist, dass $\tilde{y}(t_{i+1})$ auch auf der rechten Seite der Gleichung auftritt (für $j = -1$), so dass in jedem Schritt eine Gleichung zu lösen ist. Die Lösbarkeit ist sichergestellt, falls f die Lipschitz-Bedingung (4.10) erfüllt und die Schrittweite h klein genug ist.

Indem wir wie im Fall des Adams-Bashforth-Verfahrens vorgehen können wir beweisen, dass der lokale Fehler des Adams-Moulton-Verfahrens sich wie

$$\|y(t_{i+1}) - \tilde{y}_{\mathrm{am}}(t_{i+1})\| \leq h^{r+2} \|y^{(r+2)}\|_{\infty,[a,b]}$$

verhält, falls $y \in C^{r+2}([a,b], E)$ gilt, also hinreichend oft stetig differenzierbar ist.

Backward Differencing Formula. Statt die Integralgleichung (4.8) zu approximieren, können wir auch direkt mit der Differentialgleichung arbeiten: Falls die Lösung y dreimal stetig differenzierbar ist, können wir sie durch ein quadratisches Polynom $p \in \Pi_2$ annähern. Gut geeignet wäre ein Interpolationspolynom in den Punkten t_{i-1}, t_i und t_{i+1}, allerdings kennen wir den Wert im Punkt t_{i+1} nicht. Wir wissen allerdings, dass in diesem Punkt die Differentialgleichung (4.7) gelten muss, also können wir das Polynom durch die Bedingungen

$$\begin{aligned} p(t_{i-1}) = y(t_{i-1}), \qquad p(t_i) = y(t_i), \\ p'(t_{i+1}) \approx y'(t_{i+1}) = f(t_{i+1}, y(t_{i+1})) \approx f(t_{i+1}, p(t_{i+1})) \end{aligned}$$

definieren. In der Lagrange-Darstellung erhalten wir mit

$$\begin{aligned} p(s) &= \frac{(s-t_i)(s-t_{i+1})}{(t_{i-1}-t_i)(t_{i-1}-t_{i+1})} p(t_{i-1}) + \frac{(s-t_{i-1})(s-t_{i+1})}{(t_i-t_{i-1})(t_i-t_{i+1})} p(t_i) \\ &\quad + \frac{(s-t_{i-1})(s-t_i)}{(t_{i+1}-t_{i-1})(t_{i+1}-t_i)} p(t_{i+1}) \\ &= \frac{(s-t_i)(s-t_{i+1})}{2h^2} p(t_{i-1}) - \frac{(s-t_{i-1})(s-t_{i+1})}{h^2} p(t_i) + \frac{(s-t_{i-1})(s-t_i)}{2h^2} p(t_{i+1}) \end{aligned}$$

und der Produktregel den Ausdruck

$$p'(s) = \frac{2s - t_i - t_{i+1}}{2h^2} p(t_{i-1}) - \frac{2s - t_{i-1} - t_{i+1}}{h^2} p(t_i) + \frac{2s - t_{i-1} - t_i}{2h^2} p(t_{i+1}),$$

so dass wir die Formel

$$f(t_{i+1}, p(t_{i+1})) \approx p'(t_{i+1})$$

$$
\begin{aligned}
&= \frac{t_{i+1}-t_i}{2h^2}p(t_{i-1}) - \frac{t_{i+1}-t_{i-1}}{h^2}p(t_i) + \frac{2t_{i+1}-t_{i-1}-t_i}{2h^2}p(t_{i+1}) \\
&= \frac{1}{2h}p(t_{i-1}) - \frac{2}{h}p(t_i) + \frac{3}{2h}p(t_{i+1}) \\
&= \frac{1}{2h}y(t_{i-1}) - \frac{2}{h}y(t_i) + \frac{3}{2h}p(t_{i+1})
\end{aligned}
$$

erhalten, mit deren Hilfe wir $p(t_{i+1})$ annähern können:

$$
\begin{aligned}
f(t_{i+1}, \tilde{y}_{\mathrm{bd}}(t_{i+1})) &= \frac{1}{2h}(y(t_{i-1}) - 4y(t_i) + 3\tilde{y}_{\mathrm{bd}}(t_{i+1})), \\
\frac{1}{3}(4y(t_i) - y(t_{i-1}) + 2hf(t_{i+1}, \tilde{y}_{\mathrm{bd}}(t_{i+1}))) &= \tilde{y}_{\mathrm{bd}}(t_{i+1}).
\end{aligned}
$$

Die in der zweiten Zeile gegebene Fixpunktgleichung lässt sich wieder lösen, falls f die Lipschitz-Bedingung (4.10) erfüllt und die Schrittweite h klein genug ist.

Definition 4.15 (Rückwärtsdifferenzen-Verfahren, BDF(2)) *Seien Näherungen $\tilde{y}(t_0)$, $\tilde{y}(t_1) \in E$ der Lösung y des Anfangswertproblems (4.7) gegeben. Das* Rückwärtsdifferenzen-Verfahren *(engl.* backward differentiation formula*, BDF(2)) zweiter Ordnung ist durch die Vorschrift*

$$
f(t_{i+1}, \tilde{y}(t_{i+1})) = \frac{1}{2h}(\tilde{y}(t_{i-1}) - 4\tilde{y}(t_i) + \tilde{y}(t_{i+1})) \qquad \text{für alle } i \in \{1, \dots, n\}
$$

definiert, bei der zu beachten ist, dass $\tilde{y}(t_{i+1})$ auf beiden Seiten der Gleichung auftritt, also implizit als Lösung der Gleichung gegeben ist. Die Lösbarkeit ist sichergestellt, falls f die Lipschitz-Bedingung (4.10) erfüllt und die Schrittweite h klein genug ist.

4.5 Fehlerschranken

Bisher haben wir lediglich die Fehler analysiert, die bei *einem* Schritt des Verfahrens, ausgehend von korrekten Anfangswerten, entstehen. In der Praxis gehen wir in der Regel bereits im zweiten Schritt von Näherungswerten aus, sollten also deren Einfluss auf unsere Berechnung analysieren.

Um die Behandlung von Spezialfällen zu vermeiden setzen wir voraus, dass f *global Lipschitz-stetig im zweiten Argument ist*, dass also

$$
\|f(t, x_1) - f(t, x_2)\| \leq L_f \|x_1 - x_2\| \qquad \text{für alle } x_1, x_2 \in E,\ t \in [a, b] \tag{4.18}
$$

gilt, denn dann besitzt das Anfangswertproblem (4.7) für *jeden* Anfangswert eine eindeutig bestimmte Lösung.

Es bietet sich an, die Abhängig von den Anfangswerten in der Notation explizit darzustellen. Seien also ein Anfangszeitpunkt $c \in [a, b]$ und ein Anfangswert $x \in E$ gegeben. Mit $y(\cdot; c, x) \in C^1([c, b], E)$ bezeichnen wir die Lösung des Anfangswertproblems

$$
y(c; c, x) = x,
$$

$$y'(t;c,x) = f(t, y(t;c,x)) \qquad \text{für alle } t \in [c,b],$$

die nach dem bereits Gesagten existiert und eindeutig ist. Mit dieser Notation erhalten wir aus der Eindeutigkeit auch die folgende *Fortsetzungseigenschaft*:

$$y(b;c,x) = y(b;d,y(d;c,x)) \qquad \text{für alle } a \le c \le d \le b,\ x \in E, \tag{4.19}$$

denn $y(\cdot;c,x)|_{[d,b]}$ löst dasselbe Anfangswertproblem wie $y(\cdot;d,y(d;c,x))$.

Wir benötigen eine entsprechende Notation und eine entsprechende Aussage auch für die Näherungslösungen. Wie zuvor bezeichnen wir mit Ψ die Inkrementfunktion eines Einschrittverfahrens und mit $a = t_0 < t_1 < \ldots < t_n = b$ die Zerlegung des Zeitintervalls. Seien $i \in \{0,\ldots,n\}$ und $x \in E$ gegeben. Wir definieren

$$\begin{aligned} \tilde{y}(t_i;t_i,x) &:= x, \\ \tilde{y}(t_{j+1};t_i,x) &:= \tilde{y}(t_j;t_i,x) + h\Psi(t_j,h,\tilde{y}(t_j;t_i,x)) \qquad \text{für alle } j \in \{i,\ldots,n-1\}. \end{aligned}$$

Aus der Definition folgt direkt die Fortsetzungseigenschaft

$$\tilde{y}(b;t_i,x) = \tilde{y}(b;t_j,\tilde{y}(t_j;t_i,x)) \qquad \text{für alle } 0 \le i \le j \le n,\ x \in E, \tag{4.20}$$

denn sowohl $\tilde{y}(\cdot,t_i,x)$ als auch $\tilde{y}(b;t_j,\tilde{y}(t_j;t_i,x))$ nehmen in t_j denselben Wert an, so dass auch alle folgenden Werte gleich sein müssen.

Für die Abschätzung des Fehlers gehen wir wie folgt vor: Wir wählen ein $i \in \{0,\ldots,n\}$ und wollen den Fehler

$$\|y(b;t_i,x) - \tilde{y}(b;t_i,x)\|$$

abschätzen, der im Endzeitpunkt b entsteht, wenn wir von dem Anfangszeitpunkt t_i mit dem Anfangswert x ausgehen. Für $i = n$ gilt $t_i = b$, also ist der Fehler gleich null. Für $i = n-1$ tritt lediglich der lokale Fehler auf, da wir nur einen einzigen Zeitschritt durchführen und sich der Fehler deshalb nicht weiter fortpflanzen kann. Für $i \in \{0,\ldots,n-2\}$ können wir mit Hilfe der Fortsetzungseigenschaften den Zeitpunkt t_{i+1} einschieben:

$$\begin{aligned} \|y(b;t_i,x) - \tilde{y}(b;t_i,x)\| &= \|y(b;t_{i+1},y(t_{i+1};t_i,x)) - \tilde{y}(b;t_{i+1},\tilde{y}(t_{i+1};t_i,x))\| \\ &= \|y(b;t_{i+1},y(t_{i+1};t_i,x)) - \tilde{y}(b;t_{i+1},y(t_{i+1};t_i,x)) \\ &\quad + \tilde{y}(b;t_{i+1},y(t_{i+1};t_i,x)) - \tilde{y}(b;t_{i+1},\tilde{y}(t_{i+1};t_i,x))\| \\ &\le \|y(b;t_{i+1},y(t_{i+1};t_i,x)) - \tilde{y}(b;t_{i+1};y(t_{i+1};t_i,x))\| \\ &\quad + \|\tilde{y}(b;t_{i+1},y(t_{i+1};t_i,x)) - \tilde{y}(b;t_{i+1},\tilde{y}(t_{i+1};t_i,x))\|. \end{aligned}$$

Der erste Term dieser Abschätzung beschreibt die Differenz zwischen exakter und Näherungslösung für den Anfangszeitpunkt t_{i+1}, ihn sollten wir mit einer Induktion in den Griff bekommen können.

Der zweite Term beschreibt die Fortpflanzung einer Störung des Anfangswerts im Zuge des Näherungsverfahrens. Diese Störung wiederum ist gerade der Unterschied zwischen $y(t_{i+1};t_i,x)$ und $\tilde{y}(t_{i+1};t_i,x)$, also der lokale Fehler, den wir bereits untersucht haben.

Das Verhalten des Näherungsverfahrens bei gestörten Anfangswerten lässt sich mit Hilfe des folgenden Resultats beschreiben:

Lemma 4.16 (Gestörte Anfangswerte) *Die Inkrementfunktion Ψ sei im dritten Argument Lipschitz-stetig, es gelte also*

$$\|\Psi(t,h,x_1) - \Psi(t,h,x_2)\| \leq L_\Psi \|x_1 - x_2\| \qquad \text{für alle } x_1, x_2 \in E. \tag{4.21}$$

Dann gilt

$$\|\tilde{y}(t_j;t_i,x_1) - \tilde{y}(t_j;t_i,x_2)\| \leq e^{L_\Psi(t_j-t_i)}\|x_1 - x_2\| \quad \text{für alle } 0 \leq i \leq j \leq n,\ x_1, x_2 \in E. \tag{4.22}$$

Beweis. Per Induktion über $j - i$.

Sei zunächst $i = j \in \{0, \ldots, n\}$. Dann folgt aus der Definition

$$\|\tilde{y}(t_i;t_i,x_1) - \tilde{y}(t_i;t_i,x_2)\| = \|x_1 - x_2\| = e^{L_\Psi(t_i-t_i)}\|x_1 - x_2\|.$$

Sei nun $m \in \{0, \ldots, n\}$ so gegeben, dass (4.22) für alle $0 \leq i \leq j \leq n$ mit $j - i \leq m$ gilt.

Seien $0 \leq i \leq j \leq n$ mit $j - i = m + 1$ gegeben. Nach Definition gilt

$$\begin{aligned}
\|\tilde{y}(t_j;t_i,x_1) - \tilde{y}(t_j;t_i,x_2)\| &= \|\tilde{y}(t_{j-1};t_i,x_1) + h\Psi(t_{j-1},h,\tilde{y}(t_{j-1};t_i,x_1)) \\
&\quad - \tilde{y}(t_{j-1};t_i,x_2) - h\Psi(t_{j-1},h,\tilde{y}(t_{j-1};t_i,x_2))\| \\
&\leq \|\tilde{y}(t_{j-1};t_i,x_1) - \tilde{y}(t_{j-1};t_i,x_2)\| \\
&\quad + h\|\Psi(t_{j-1},t_i,\tilde{y}(t_{j-1};t_i,x_1)) - \Psi(t_{j-1},t_i,\tilde{y}(t_{j-1};t_i,x_2))\| \\
&\leq \|\tilde{y}(t_{j-1};t_i,x_1) - \tilde{y}(t_{j-1};t_i,x_2)\| \\
&\quad + L_\Psi h\|\tilde{y}(t_{j-1};t_i,x_1) - \tilde{y}(t_{j-1};t_i,x_2)\| \\
&= (1 + L_\Psi h)\|\tilde{y}(t_{j-1};t_i,x_1) - \tilde{y}(t_{j-1};t_i,x_2)\|.
\end{aligned}$$

Dank $(j-1) - i = j - i - 1 = m$ dürfen wir die Induktionsvoraussetzung anwenden und erhalten

$$\|\tilde{y}(t_{j-1};t_i,x_1) - \tilde{y}(t_{j-1};t_i,x_2)\| \leq e^{L_\Psi(t_{j-1}-t_i)}\|x_1 - x_2\|,$$

so dass sich

$$\|\tilde{y}(t_j;t_i,x_1) - \tilde{y}(t_j;t_i,x_2)\| \leq (1 + L_\Psi h)e^{L_\Psi(t_{j-1}-t_i)}\|x_1 - x_2\|$$

ergibt. Aus der Reihendarstellung der Exponentialfunktion folgt

$$1 + L_\Psi h \leq e^{L_\Psi h},$$

so dass sich unsere Abschätzung vereinfachen lässt und wir

$$\|\tilde{y}(t_j;t_i,x_1) - \tilde{y}(t_j;t_i,x_2)\| \leq e^{L_\Psi h}e^{L_\Psi(t_{j-1}-t_i)}\|x_1 - x_2\| = e^{L_\Psi(t_j-t_i)}\|x_1 - x_2\|$$

bewiesen haben. Damit ist der Induktionsschritt abgeschlossen. ∎

Nun steht uns das wesentliche Hilfsmittel zur Verfügung, um eine Schranke für den Gesamtfehler zu gewinnen:

Satz 4.17 (Globale Fehlerschranke) *Die Inkrementfunktion Ψ sei im dritten Argument Lipschitz-stetig. Sei*

$$K_\Psi := \max\{\|y(t_{i+1}; t_i, y(t_i)) - \tilde{y}(t_{i+1}; t_i, y(t_i))\| \ : \ i \in \{0, \ldots, n-1\}\}$$

die Schranke der lokalen *Fehler. Dann gilt*

$$\|y(t_j) - \tilde{y}(t_j; t_i, y(t_i))\| \leq K_\Psi \sum_{k=i+1}^{j} e^{L_\Psi(t_j - t_k)}. \tag{4.23}$$

Beweis. Per Induktion über $j - i$.

Sei zunächst $i = j \in \{0, \ldots, n\}$. Dann gilt

$$\tilde{y}(t_j; t_i, y(t_i)) = y(t_i),$$

also auch die Abschätzung (4.23).

Sei nun $m \in \mathbb{N}_0$ so gegeben, dass (4.23) für alle $0 \leq i \leq j \leq n$ mit $j - i \leq m$ gilt. Seien $0 \leq i \leq j \leq n$ mit $j - i = m + 1$ gegeben. Aus den Fortsetzungseigenschaften (4.19) und (4.20) erhalten wir

$$\begin{aligned}
\|y(t_j) - \tilde{y}(t_j; t_i, y(t_i))\| &= \|y(t_j; t_{i+1}, y(t_{i+1}; t_i, y(t_i))) - \tilde{y}(t_j; t_{i+1}, y(t_{i+1}; t_i, y(t_i))) \\
&\quad + \tilde{y}(t_j; t_{i+1}, y(t_{i+1}; t_i, y(t_i))) - \tilde{y}(t_j; t_{i+1}, \tilde{y}(t_{i+1}; t_i, y(t_i)))\| \\
&\leq \|y(t_j; t_{i+1}, y(t_{i+1})) - \tilde{y}(t_j; t_{i+1}, y(t_{i+1}))\| \\
&\quad + \|\tilde{y}(t_j; t_{i+1}, y(t_{i+1}; t_i, y(t_i))) - \tilde{y}(t_j; t_{i+1}, \tilde{y}(t_{i+1}; t_i, y(t_i)))\|.
\end{aligned}$$

Wegen $j - (i+1) = i - j - 1 = m$ dürfen wir die Induktionsvoraussetzung auf den ersten Term anwenden und erhalten

$$\|y(t_j; t_{i+1}, y(t_{i+1})) - \tilde{y}(t_j; t_{i+1}, y(t_{i+1}))\| \leq K_\Psi \sum_{k=i+2}^{j} e^{L_\Psi(t_j - t_k)}.$$

Mit Lemma 4.16 erhalten wir für den zweiten Term

$$\begin{aligned}
&\|\tilde{y}(t_j; t_{i+1}, y(t_{i+1}; t_i, y(t_i))) - \tilde{y}(t_j; t_{i+1}, \tilde{y}(t_{i+1}; t_i, y(t_i)))\| \\
&\qquad \leq e^{L_\Psi(t_j - t_{i+1})} \|y(t_{i+1}; t_i, y(t_i)) - \tilde{y}(t_{i+1}; t_i, y(t_i))\| \\
&\qquad \leq e^{L_\Psi(t_j - t_{i+1})} K_\Psi,
\end{aligned}$$

und die Addition beider Abschätzungen ergibt das gewünschte Resultat (4.23). ∎

Die Abschätzung (4.23) kann für den hier betrachteten Fall konstanter Zeitschrittweiten etwas handlicher gestaltet werden:

Folgerung 4.18 (Globale Fehlerschranke) *Unter den Voraussetzungen des Konvergenzsatzes 4.17 erhalten wir die Abschätzung*

$$\|y(t_j) - \tilde{y}(t_j; t_i, y(t_i))\| \leq \begin{cases} \frac{K_\Psi}{h} \frac{e^{L_\Psi(t_j - t_i)} - 1}{L_\Psi} & \text{falls } L_\Psi \neq 0, \\ \frac{K_\Psi}{h}(t_j - t_i) & \text{ansonsten} \end{cases} \quad \text{für alle } 0 \leq i \leq j \leq n.$$

Beweis. Seien $0 \leq i \leq j \leq n$ gegeben. Falls $L_\Psi = 0$ gelten sollte, haben wir

$$\sum_{k=i+1}^{j} e^{L_\Psi(t_j - t_k)} = \sum_{k=i+1}^{j} 1 = j - i = \frac{hj - hi}{h} = \frac{t_j - t_i}{h},$$

also folgt mit (4.23) die zu beweisende Abschätzung.

Gelte nun $L_\Psi > 0$. Wir definieren die Hilfsgröße

$$q := e^{L_\Psi h} \neq 1$$

und stellen fest, dass

$$e^{L_\Psi(t_j - t_k)} = e^{L_\Psi h(j-k)} = q^{j-k} \qquad \text{für alle } k \in \{i+1, \ldots, j\}$$

gilt. Per geometrischer Summenformel erhalten wir

$$\sum_{k=i+1}^{j} e^{L_\Psi(t_j - t_k)} = \sum_{k=i+1}^{j} q^{j-k} = \sum_{\ell=0}^{j-i-1} q^\ell = \frac{q^{j-i} - 1}{q - 1} = \frac{e^{L_\Psi h(j-i)} - 1}{q - 1} = \frac{e^{L_\Psi(t_j - t_i)} - 1}{q - 1}.$$

Aus der Reihendarstellung der Exponentialfunktion folgt

$$q = e^{L_\Psi h} \geq 1 + L_\Psi h,$$

also

$$\sum_{k=i+1}^{j} e^{L_\Psi(t_j - t_k)} = \frac{e^{L_\Psi(t_j - t_i)} - 1}{q - 1} \leq \frac{e^{L_\Psi(t_j - t_i)} - 1}{L_\Psi h}.$$

Durch Einsetzen in (4.23) erhalten wir die gewünschte Abschätzung. ∎

Diese Fehlerabschätzungen können wir mit den Resultaten für den lokalen Fehler kombinieren, um den Gesamtfehler der bisher diskutierten Verfahren zu analysieren.

Beispiel 4.19 (Explizites Euler-Verfahren) *Die Funktion f erfülle die Lipschitz-Bedingung (4.10) mit einer Lipschitz-Konstanten $L_f > 0$. Die Lösung y sei zweimal stetig differenzierbar. Dann gilt für die durch das explizite Euler-Verfahren berechnete Näherung die Abschätzung*

$$\|y(t_j) - \tilde{y}(t_j)\| \leq h\|y''\|_{\infty,[a,b]} \frac{e^{L_f(t_j - a)} - 1}{L_f} \qquad \textit{für alle } j \in \{0, \ldots, n\}.$$

Beweis. Dank der lokalen Fehlerabschätzung (4.13) gilt

$$K_\Psi \leq h^2\|y''\|_{\infty,[a,b]}.$$

Aus der Lipschitz-Stetigkeit der Funktion f folgt, dass die Inkrementfunktion die Lipschitz-Bedingung (4.21) mit $L_\Psi = L_f$ erfüllt. Mit Folgerung 4.18 erhalten wir die gewünschte Aussage. ∎

Beispiel 4.20 (Runge-Verfahren) *Die Funktion f erfülle die Lipschitz-Bedingung (4.10) mit einer Lipschitz-Konstanten $L_f > 0$. Die Lösung y sei dreimal stetig differenzierbar. Dann gilt für die durch das Runge-Verfahren berechnete Näherung die Abschätzung*

$$\|y(t_j) - \tilde{y}(t_j)\| \leq Ch^2 \frac{e^{L_\Psi(t_j-a)} - 1}{L_\Psi} \qquad \text{für alle } j \in \{0, \ldots, n\}$$

mit

$$C = \frac{\|y^{(3)}\|_{\infty,[a,b]}}{24} + \frac{L_f \|y''\|_{\infty,[a,b]}}{4}, \qquad L_\Psi = L_f(1 + L_f h/2).$$

Beweis. Aus (4.15) folgt

$$K_\Psi \leq h^3 \left(\frac{\|y^{(3)}\|_{\infty,[a,b]}}{24} + \frac{L_f \|y''\|_{\infty,[a,b]}}{4} \right).$$

Zum Nachweis der Lipschitz-Bedingung (4.21) wählen wir $x_1, x_2 \in E$ sowie $t \in [a, b]$ und erhalten

$$\begin{aligned}
&\|\Psi(t, h, x_1) - \Psi(t, h, x_2)\| \\
&\qquad = \|f(t + h/2, x_1 + hf(t, x_1)/2) - f(t + h/2, x_2 + hf(t, x_2)/2)\| \\
&\qquad \leq L_f \|x_1 + hf(t, x_1)/2 - x_2 - hf(t, x_2)/2\| \\
&\qquad \leq L_f (\|x_1 - x_2\| + h\|f(t, x_1) - f(t, x_2)\|/2) \\
&\qquad \leq L_f (\|x_1 - x_2\| + hL_f \|x_1 - x_2\|/2) \\
&\qquad = L_f(1 + L_f h/2)\|x_1 - x_2\| = L_\Psi \|x_1 - x_2\|.
\end{aligned}$$

Kombination beider Abschätzungen mit Folgerung 4.18 führt zu dem gewünschten Resultat. ∎

Die Fehleranalyse expliziter Mehrschrittverfahren, also in unserem Fall des Adams-Bashforth-Verfahrens, lässt sich mit ähnlichen Mitteln durchführen, indem wir r Zeitschritte zu einem Vektor zusammenfassen:

$$Y(t_i) := \begin{pmatrix} y(t_{i-r+1}) \\ \vdots \\ y(t_i) \end{pmatrix}, \qquad \widetilde{Y}(t_i) := \begin{pmatrix} \tilde{y}(t_{i-r+1}) \\ \vdots \\ \tilde{y}(t_i) \end{pmatrix} \qquad \text{für alle } i \in \{r-1, \ldots, n\}.$$

Die Rechenvorschrift des Adams-Bashforth-Verfahrens nimmt damit die Gestalt

$$\widetilde{Y}(t_{i+1}) = \begin{pmatrix} 0 & I & & \\ & \ddots & \ddots & \\ & & 0 & I \\ & & & I \end{pmatrix} \widetilde{Y}(t_i) + h \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \sum_{j=0}^{r-1} w_j f(t_{i-j}, \widetilde{Y}_{r-j}(t_i)) \end{pmatrix}$$

$$\text{für alle } i \in \{r-1, \dots, n-1\}$$

an. Indem wir

$$A := \begin{pmatrix} 0 & I & & \\ & \ddots & \ddots & \\ & & 0 & I \\ & & & I \end{pmatrix}, \qquad \Psi(t,h,X) := \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \sum_{j=0}^{r-1} w_j f(t-hj, X_{r-j}) \end{pmatrix}$$

definieren, erhalten wir die Form

$$\widetilde{Y}(t_{i+1}) = A\widetilde{Y}(t_i) + h\Psi(t_i, h, \widetilde{Y}(t_i)) \qquad \text{für alle } i \in \{r-1, \dots, n-1\}, \tag{4.24}$$

die der eines Einschrittverfahrens sehr nahe kommt.

Der Beweis des Satzes 4.17 lässt sich deshalb wie im Fall der Einschrittverfahren durchführen, sofern uns ein Gegenstück des Störungslemmas 4.16 zur Verfügung steht. Um dieses Resultat zu verallgemeinern, ist es von entscheidender Bedeutung, die richtige Norm zu verwenden.

Definition 4.21 (Stabiles Mehrschrittverfahren) *Wir bezeichnen ein durch A und Ψ gemäß (4.24) definiertes Mehrschrittverfahren als* stabil*, falls eine Norm $\|\cdot\|_r$ auf E^r existiert, die*

$$\|\Psi(t,h,X_1) - \Psi(t,h,X_2)\|_r \le L_\Psi \|X_1 - X_2\|_r \qquad \textit{für alle } X_1, X_2 \in E^r, \tag{4.25a}$$

$$\|AX\|_r \le \|X\|_r \qquad \textit{für alle } X \in E^r \tag{4.25b}$$

erfüllt. Die erste Bedingung ist die bereits in Lemma 4.16 verwendete Lipschitz-Stetigkeit, die zweite ist eine Besonderheit der Mehrschrittverfahren.

Für ein stabiles Mehrschrittverfahren können wir Lemma 4.16 verallgemeinern: Wir definieren die von dem Anfangswert X zu dem Anfangszeitpunkt t_i ausgehende Näherungslösung durch

$$\begin{aligned} \widetilde{Y}(t_i; t_i, X) &:= X, \\ \widetilde{Y}(t_{j+1}; t_i, X) &:= A\widetilde{Y}(t_j; t_i, X) + h\Psi(t_j, h, \widetilde{Y}(t_j; t_i, X)) \qquad \text{für alle } j \in \{i, \dots, n\} \end{aligned}$$

und erhalten das folgende Störungsresultat:

Lemma 4.22 (Gestörte Anfangswerte) *Das durch A und Ψ gemäß (4.24) definierte Mehrschrittverfahren sei stabil mit der Norm $\|\cdot\|_r$. Für die Näherungslösungen gilt dann*

$$\|\widetilde{Y}(t_j; t_i, X_1) - \widetilde{Y}(t_j; t_i, X_2)\|_r \le e^{L_\Psi(t_j - t_i)} \|X_1 - X_2\|_r$$
$$\textit{für alle } r-1 \le i \le j \le n,\ X_1, X_2 \in E^r.$$

Beweis. Wie im Beweis des Lemmas 4.16 per Induktion über $j - i$ unter Zuhilfenahme der Abschätzung

$$\begin{aligned}
&\|\widetilde{Y}(t_j; t_i, X_1) - \widetilde{Y}(t_j; t_i, X_2)\|_r \\
&\qquad = \|A\widetilde{Y}(t_{j-1}; t_i, X_1) + h\Psi(t_{j-1}, h, \widetilde{Y}(t_{j-1}; t_i, X_1)) \\
&\qquad\qquad - A\widetilde{Y}(t_{j-1}; t_i, X_2) - h\Psi(t_{j-1}, h, \widetilde{Y}(t_{j-1}; t_i, X_2))\|_r \\
&\qquad \leq \|A(\widetilde{Y}(t_{j-1}; t_i, X_1) - \widetilde{Y}(t_{j-1}; t_i, X_2))\|_r \\
&\qquad\qquad + h\|\Psi(t_{j-1}, h, \widetilde{Y}(t_{j-1}, t_i, X_1)) - \Psi(t_{j-1}, h, \widetilde{Y}(t_{j-1}, t_i, X_2))\|_r \\
&\qquad \leq (1 + hL_\Psi)\|\widetilde{Y}(t_{j-1}, t_i, X_1) - \widetilde{Y}(t_{j-1}, t_i, X_2)\|_r.
\end{aligned}$$

■

5 Finite Differenzen

Bei gewöhnlichen Differentialgleichungen treten nur Ableitungen nach einer bestimmten Variablen, in der Regel der Zeit, auf. Deshalb können derartige Gleichungen relativ einfach behandelt werden, indem man entlang der Zeitachse „integriert“ und so eine Näherung berechnet.

Bei vielen Anwendungen, beispielsweise in der Strukturmechanik, der Theorie elektromagnetischer Felder oder auch in der Strömungsdynamik, treten hingegen Ableitungen in mehreren Variablen auf, die sich nicht so einfach handhaben lassen. In diesen Situationen führt eine Approximation der Ableitungen zu einem Gleichungssystem, in dem sämtlichen Punkte des Definitionsbereichs Unbekannte zugeordnet sind, die wir berechnen müssen.

5.1 Beispiel: Elektrostatik

Elektromagnetische Felder in einem offenen Gebiet $\Omega \subseteq \mathbb{R}^3$ und einem Zeitintervall $[a, b]$ sind durch

- das *elektrische Feld* $E : [a, b] \times \Omega \to \mathbb{R}^3$,
- das *magnetische Feld* $H : [a, b] \times \Omega \to \mathbb{R}^3$,
- die *elektrische Flussdichte* $D : [a, b] \times \Omega \to \mathbb{R}^3$ und
- die *magnetische Flussdichte* $B : [a, b] \times \Omega \to \mathbb{R}$

beschrieben. Die Felder üben auf Ladungsträger (beispielsweise Elektronen) die *Lorentz-Kraft*

$$F(t) = q(E(t, x(t)) + v(t) \times B(t, x(t))) \tag{5.1}$$

aus, wobei $x(t)$ die Position des Ladungsträgers, $v(t)$ seine Geschwindigkeit und q die Größe seiner Ladung beschreibt. Felder werden hervorgerufen durch

- die *Stromdichte* $j : [a, b] \times \Omega \to \mathbb{R}^3$ und
- die *Raumladungsdichte* $\varrho : [a, b] \times \Omega \to \mathbb{R}$.

Die Eigenschaften der Felder sind durch die *Maxwell-Gleichungen* beschrieben:

Faraday'sches Gesetz. Die erste Maxwell-Gleichung besitzt die Form

$$\nabla \times E(t,x) = -\frac{\partial B}{\partial t}(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega, \tag{5.2}$$

wobei der durch

$$\nabla \times f(x) := \begin{pmatrix} \partial_2 f_3(x) - \partial_3 f_2(x) \\ \partial_3 f_1(x) - \partial_1 f_3(x) \\ \partial_1 f_2(x) - \partial_2 f_1(x) \end{pmatrix} \qquad \text{für alle } f \in C^1(\Omega, \mathbb{R}^3),\ x \in \Omega$$

definierte *Rotationsoperator* ein Maß für die Wirbel des Vektorfelds ist. Die erste Maxwell-Gleichung entspricht dem *Faraday'schen Gesetz* und besagt anschaulich, dass eine Veränderung der magnetischen Flussdichte ein elektrisches Feld hervorruft, beispielsweise in einem Dynamo.

Ampère'sches Gesetz. Die zweite Maxwell-Gleichung lautet

$$\nabla \times H(t,x) = j(t,x) + \frac{\partial D}{\partial t}(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega \tag{5.3}$$

und besagt, dass ein fließender Strom und eine Veränderung der elektrischen Flussdichte Wirbel im magnetischen Feld hervorrufen. Diese Gleichung entspricht dem *Ampère'schen Gesetz*, das besagt, dass das magnetische Feld in einer geschlossenen Schleife durch einen elektrischen Strom entsteht, der durch die Schleife fließt. Eine Anwendung ist der Elektromagnet, bei dem ein elektrischer Strom ein magnetisches Feld hervorruft.

Gauß'sches Gesetz für elektrische Felder. Die dritte Maxwellgleichung

$$\nabla \cdot D(t,x) = \varrho(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega \tag{5.4}$$

verwendet den durch

$$\nabla \cdot f(x) := \partial_1 f_1(x) + \partial_2 f_2(x) + \partial_3 f_3(x) \qquad \text{für alle } f \in C^1(\Omega, \mathbb{R}^3),\ x \in \Omega$$

definierten *Divergenzoperator*, der ein Maß für die Quellen des Vektorfelds ist. Die Gleichung entspricht dem *Gauß'schen Gesetz* (für den elektrischen Fluss), das besagt, dass Ladungen die Quellen des elektrischen Flusses sind: Ladungsträger werden durch die Lorentz-Kraft von anderen Ladungsträgern angezogen oder abgestoßen.

Gauß'sches Gesetz für magnetische Felder. Die vierte und letzte Maxwellgleichung

$$\nabla \cdot B(t,x) = 0 \qquad \text{für alle } t \in [a,b],\ x \in \Omega \tag{5.5}$$

beschreibt, dass der magnetische Fluss quellenfrei ist. Die Gleichung trägt ebenfalls den Namen *Gaußsches Gesetz* (diesmal für das magnetische Feld) und impliziert beispielsweise, dass es keine magnetischen Monopole (also Südpole ohne Nordpole oder umgekehrt) geben kann.

Materialgesetze. Die Verbindung zwischen dem Feld und der Flussdichte ist von den Eigenschaften des Materials abhängig. Wir gehen hier von *linearen*, *homogenen* und *isotropen* Materialien aus, bei denen die elektrische Flussdichte proportional zu dem elektrischen Feld ist, also

$$D(t,x) = \epsilon E(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega \tag{5.6}$$

mit der *Dielektrizitätskonstante* $\epsilon \in \mathbb{R}_{>0}$. Entsprechend ist die magnetische Flussdichte proportional zu dem magnetischen Feld, es gilt also

$$B(t,x) = \mu H(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega \tag{5.7}$$

mit der *Permeabilitätskonstanten* $\mu \in \mathbb{R}_{>0}$. Schließlich übt das elektrische Feld eine Kraft auf freie Ladungsträger aus, die zu einem Strom führt. Im einfachsten Fall beschreiben wir diesen Effekt durch die Gleichung

$$j_{\mathrm{ind}}(t,x) = \sigma E(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega \tag{5.8}$$

mit der *Leitfähigkeitskonstanten* $\sigma \in \mathbb{R}_{\geq 0}$, die beschreibt, wie durchlässig das Material für freie Ladungsträger ist.

Elektrostatik. Wir untersuchen zunächst einen besonders einfachen Fall: Falls die magnetische Flussdichte B nicht von der Zeit abhängt, falls also $\partial B/\partial t = 0$ gilt, nimmt das Faraday'sche Gesetz (5.2) die Form

$$\nabla \times E(t,x) = 0 \qquad \text{für alle } t \in [a,b],\ x \in \Omega$$

an. Falls wir voraussetzen, dass das Gebiet Ω einfach zusammenhängend ist, falls sich also zwei beliebige Kurven, die dieselben Anfangs- und Endpunkte verbinden, stetig ineinander überführen lassen (anschaulich: Falls Ω keine „Löcher“ aufweist), besagt diese Gleichung, dass sich $E(t,x)$ in der Form

$$E(t,x) = -\nabla\varphi(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega$$

mit einem *Potential* $\varphi \in C^1(\Omega)$ darstellen lässt, wobei

$$\nabla f(x) = \begin{pmatrix} \partial_1 f(x) \\ \partial_2 f(x) \\ \partial_3 f(x) \end{pmatrix} \qquad \text{für alle } x \in \Omega$$

den *Gradienten* der Funktion $f \in C^1(\Omega)$ bezeichnet. Indem wir diese Darstellung in das Gauß'sche Gesetz für elektrische Felder (5.4) einsetzen, erhalten wir Indem wir mit der Dielektrizitätskonstanten multiplizieren, finden wir dank (5.6)

$$D(t,x) = -\epsilon\nabla\varphi(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega,$$

und mit dem Gauß'schen Gesetz für elektrische Felder (5.4) folgt

$$-\nabla \cdot (\epsilon \nabla \varphi)(t,x) = \varrho(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega.$$

Offenbar spielt die Zeit in dieser Gleichung keine besondere Rolle, so dass es genügt, sie für ein festes t zu untersuchen und t nicht mehr zu erwähnen, damit erhalten wir

$$-\dot{\nabla}(\epsilon \nabla \varphi)(x) = \varrho(x) \qquad \text{für alle } x \in \Omega, \tag{5.9}$$

die *Poisson-Gleichung* (auch bekannt unter dem Namen *Potentialgleichung*).

Bemerkung 5.1 (∇-Notation) *Die Schreibweisen der Rotation, der Divergenz und des Gradienten lassen sich motivieren, indem wir formal den „Vektor“*

$$\nabla := \begin{pmatrix} \partial_1 \\ \partial_2 \\ \partial_3 \end{pmatrix}$$

einführen. Die Rotation ergibt sich, indem wir formal das Kreuzprodukt dieses „Vektors“ mit einem Vektorfeld bilden, die Divergenz ist das euklidische Skalarprodukt, und der Gradient die Skalar-Multiplikation mit einer skalarwertigen Funktion.

Randbedingungen. Bei genauerer Betrachtung stellt man fest, dass die Gleichung (5.9) nicht eindeutig lösbar ist. Beispielsweise können wir zu φ eine Konstante hinzuaddieren, ohne dass dadurch die rechte Seite verändert würde. Um die Eindeutigkeit einer Lösung zu garantieren, geben wir *Randbedingungen* vor: Besonders einfach wird unser Problem, wenn wir annehmen, dass das Material am Rand unseres Gebiets Ω *supraleitend* ist. Anschaulich bedeutet das, dass die Elektronen sich innerhalb des Rands völlig frei bewegen können und so den Anteil des elektrischen Felds E, der tangential zum Rand wirkt, aufheben. Da $E = \nabla\varphi$ gilt, folgt per Hauptsatz der Integral- und Differentialrechnung, dass φ auf dem Rand konstant sein muss. Da Konstanten $\nabla\varphi$ nicht verändern, können wir der Einfachheit halber fordern, dass φ auf dem Rand des Gebiets konstant gleich null ist. Damit erhalten wir das System

$$-\nabla \cdot (\epsilon \nabla \varphi)(x) = \varrho(x) \qquad \text{für alle } x \in \Omega, \tag{5.10a}$$

$$\varphi(x) = 0 \qquad \text{für alle } x \in \partial\Omega = \overline{\Omega} \cap \overline{\mathbb{R}^d \setminus \Omega}. \tag{5.10b}$$

Da Ableitungen in alle drei Koordinatenrichtungen gleichzeitig auftreten, können wir die für gewöhnliche Differentialgleichungen entwickelten Techniken nicht ohne weiteres anwenden.

Differenzenquotient. Indem wir durch ϵ dividieren erhalten wir

$$\frac{\varrho(x)}{\epsilon} = -\nabla \cdot \nabla\varphi(x) = -\partial_1^2\varphi(x) - \partial_2^2\varphi(x) - \partial_3^2\varphi(x) \qquad \text{für alle } x \in \Omega.$$

Es bietet sich an, bei der Konstruktion eines Näherungsverfahrens für diese Gleichung bei den partiellen Ableitungen anzusetzen: Die zweite Ableitung lässt sich durch einen Differenzenquotienten approximieren.

Lemma 5.2 (Zweiter Differenzenquotient) *Seien $h \in \mathbb{R}_{>0}$ und $g \in C^4[-h,h]$ gegeben. Dann existiert ein $\eta \in [-h,h]$ mit*

$$\frac{g(h) - 2g(0) + g(-h)}{h^2} - g''(0) = \frac{h^2}{12} g^{(4)}(\eta).$$

Beweis. Mit dem Taylor'schen Satz finden wir $\eta_+, \eta_- \in [-h,h]$ mit

$$\begin{aligned} g(h) &= g(0) + hg'(0) + \frac{h^2}{2} g''(0) + \frac{h^3}{6} g'''(0) + \frac{h^4}{24} g^{(4)}(\eta_+), \\ g(-h) &= g(0) - hg'(0) + \frac{h^2}{2} g''(0) - \frac{h^3}{6} g'''(0) + \frac{h^4}{24} g^{(4)}(\eta_-). \end{aligned}$$

Durch Addition der Gleichungen folgt

$$\begin{aligned} g(h) + g(-h) &= 2g(0) + h^2 g''(0) + \frac{h^4}{24}(g^{(4)}(\eta_+) + g^{(4)}(\eta_-)), \\ \frac{g(h) + g(-h) - 2g(0)}{h^2} &= g''(0) + \frac{h^2}{24}(g^{(4)}(\eta_+) + g^{(4)}(\eta_-)). \end{aligned}$$

Dank des Zwischenwertsatzes finden wir ein $\eta \in [-h,h]$ mit

$$g^{(4)}(\eta) = \frac{g^{(4)}(\eta_+) + g^{(4)}(\eta_-)}{2}$$

und stellen fest, dass wir

$$\frac{g(h) - 2g(0) + g(-h)}{h^2} - g''(0) = \frac{h^2}{12} g^{(4)}(\eta)$$

bewiesen haben. ∎

Diese Approximation der zweiten Ableitung können wir selbstverständlich auch auf die zweiten partiellen Ableitungen anwenden. Wir fixieren $i \in \{1,2,3\}$, bezeichnen mit $\delta_i \in \mathbb{R}^3$ den i-ten kanonischen Einheitsvektor und setzen voraus, dass $\varphi \in C^4(\Omega)$ gilt. Sei $x \in \Omega$, und sei $h \in \mathbb{R}_{>0}$ so klein, dass die abgeschlossene Kugel $\bar{K}(x,h)$ um x mit Radius h vollständig in Ω enthalten ist. Mit Lemma 5.2 folgt dann

$$-\partial_i^2 \varphi(x) - \frac{2\varphi(x) - \varphi(x + h\delta_i) - \varphi(x - h\delta_i)}{h^2} = \frac{h^2}{12} \partial_i^4 \varphi(\eta_i)$$

für ein $\eta_i \in x + \delta_i[-h,h]$, wir dürfen also darauf hoffen, dass der Differenzenquotient wie h^2 gegen die zweite partielle Ableitung konvergiert.

Diskretisierung. Da im Differenzenquotienten nur die Werte der Funktion φ in x, $x+h\delta_i$ und $x - h\delta_i$ auftreten, bietet es sich an, die Werte der Funktion in diesen Punkten als Unbekannte eines Gleichungssystems zu verwenden, das die Differentialgleichung approximiert. Diesen Schritt, also der Übergang von unendlich vielen Unbekannten (Werte $\varphi(x)$ für alle $x \in \Omega$) zu endlich vielen, bezeichnet man als *Diskretisierung*. Nur durch eine

Diskretisierung können wir auf einem Kontinuum definierte Funktionen approximieren, da der Computer nur endlich viele Zahlen speichern kann.

Vorläufig beschränken wir uns auf den einfachsten Fall des offenen Einheitswürfels

$$\begin{aligned}\Omega &= (0,1)^3,\\ \partial\Omega &= \{0,1\}\times[0,1]\times[0,1] \cup [0,1]\times\{0,1\}\times[0,1] \cup [0,1]\times[0,1]\times\{0,1\}\end{aligned}$$

Für ein $n \in \mathbb{N}$ setzen wir

$$\begin{aligned}h &:= \frac{1}{n+1},\\ \Omega_h &:= \{x \in \Omega \,:\, x_1/h, x_2/h, x_3/h \in \mathbb{Z}\},\\ \partial\Omega_h &:= \{x \in \partial\Omega \,:\, x_1/h, x_2/h, x_3/h \in \mathbb{Z}\}\end{aligned}$$

und wollen das kontinuierliche Gebiet Ω durch das Punktgitter Ω_h ersetzen. Es bietet sich an, die Punkte in Ω_h durchzunumerieren:

$$x_i := (hi_x, hi_y, hi_z) \qquad \text{für alle } i = (i_x, i_y, i_z) \in \{0, \dots, n+1\}^3.$$

Unser Ziel ist es, die Werte der Funktion φ in den Punkten des Gitters Ω_h zu approximieren, also die Werte

$$\hat{u}_i := \varphi(x_i) \qquad \text{für alle } i = (i_x, i_y, i_z) \in \{0, \dots, n+1\}^3$$

zu bestimmen. Die Randwerte sind uns dabei bereits bekannt, unbekannt sind nur die Werte φ_i für

$$i = (i_x, i_y, i_z) \in \mathcal{I} := \{1, \dots, n\}^3.$$

Für ein $i \in \mathcal{I}$ erhalten wir mit Hilfe des Differenzenquotienten die Beziehung

$$\begin{aligned}\frac{\varrho(x_i)}{\epsilon} &= -\partial_1^2\varphi(x_i) - \partial_2^2\varphi(x_i) - \partial_3^2\varphi(x_i)\\ &\approx \frac{2\varphi(x_i) - \varphi(x_i + h\delta_1) - \varphi(x_i - h\delta_1)}{h^2} + \frac{2\varphi(x_i) - \varphi(x_i + h\delta_2) - \varphi(x_i - h\delta_2)}{h^2}\\ &\quad + \frac{2\varphi(x_i) - \varphi(x_i + h\delta_3) - \varphi(x_i - h\delta_3)}{h^2}\\ &= \frac{2\hat{u}_i - \hat{u}_{i_x+1,i_y,i_z} - \hat{u}_{i_x-1,i_y,i_z}}{h^2} + \frac{2\hat{u}_i - \hat{u}_{i_x,i_y+1,i_z} - \hat{u}_{i_x,i_y-1,i_z}}{h^2}\\ &\quad + \frac{2\hat{u}_i - \hat{u}_{i_x,i_y,i_z+1} - \hat{u}_{i_x,i_y,i_z-1}}{h^2}\\ &= \frac{1}{h^2}\big(6\hat{u}_i - \hat{u}_{i_x+1,i_y,i_z} - \hat{u}_{i_x-1,i_y,i_z} - \hat{u}_{i_x,i_y+1,i_z} - \hat{u}_{i_x,i_y-1,i_z}\\ &\qquad -\hat{u}_{i_x,i_y,i_z+1} - \hat{u}_{i_x,i_y,i_z-1}\big),\end{aligned}$$

also, abgesehen von der Näherung in der zweiten Zeile, ein lineares Gleichungssystem mit den für uns relevanten Größen.

Der Ansatz des *Finite-Differenzen-Verfahrens* besteht darin, die Lösung dieses Gleichungssystems als Näherung der Lösung der Differentialgleichung zu verwenden. Dazu definieren wir

$$\overline{\mathcal{I}} := \{0, \dots, n+1\}^3, \qquad \partial\mathcal{I} := \overline{\mathcal{I}} \setminus \mathcal{I}$$

und die Matrix $\widehat{L} \in \mathbb{R}^{\mathcal{I}\times\overline{\mathcal{I}}}$, die durch

$$\hat{\ell}_{ij} := \begin{cases} 6h^{-2} & \text{falls } i = j, \\ -h^{-2} & \text{falls } |i_x - j_x| = 1,\ i_y = j_y,\ i_z = j_z, \\ -h^{-2} & \text{falls } i_x = j_x,\ |i_y - j_y| = 1,\ i_z = j_z, \\ -h^{-2} & \text{falls } i_x = j_x,\ i_y = j_y,\ |i_z - j_z| = 1, \\ 0 & \text{ansonsten} \end{cases} \quad \begin{array}{l} \text{für alle } i = (i_x, i_y, i_z) \in \mathcal{I}, \\ \qquad\qquad j = (j_x, j_y, j_z) \in \overline{\mathcal{I}} \end{array} \tag{5.11}$$

definiert ist und den Vektor $f \in \mathbb{R}^{\mathcal{I}}$, der durch

$$f_i := \varrho(x_i)/\epsilon \qquad \text{für alle } i \in \mathcal{I}$$

festgelegt wird. Da in dem zur Zeit untersuchten Problem die Lösung auf dem Rand des Gebiets gleich null ist, können wir die Randpunkte ignorieren und lediglich mit der Matrix $L := \widehat{L}|_{\mathcal{I}\times\mathcal{I}}$ arbeiten. Wir wissen, dass die exakten Werte $\hat{u} := (\varphi_i)_{i\in\mathcal{I}}$ die Beziehung

$$\|L\hat{u} - f\|_{\infty,\mathcal{I}} \leq \frac{h^2}{12}(\|\partial_1^4\varphi\|_{\infty,\Omega} + \|\partial_2^4\varphi\|_{\infty,\Omega} + \|\partial_3^4\varphi\|_{\infty,\Omega})$$

erfüllen, also bietet es sich an, nach der Lösung des Gleichungssystems

$$Lu = f \tag{5.12}$$

zu suchen. Es lässt sich zeigen, dass die Matrix L invertierbar ist und dass die von der Maximumnorm induzierte Norm $\|L^{-1}\|_{\infty,\mathcal{I}}$ ihrer Inversen durch eine von h unabhängige Konstante C_s beschränkt werden kann, so dass wir

$$\begin{aligned} \|\hat{u} - u\|_{\infty,\mathcal{I}} = \|L^{-1}(L\hat{u} - Lu)\|_{\infty,\mathcal{I}} &\leq \|L^{-1}\|_{\infty,\mathcal{I}}\|L\hat{u} - f\|_{\infty,\mathcal{I}} \\ &\leq C_s\frac{h^2}{12}(\|\partial_1^4\varphi\|_{\infty,\Omega} + \|\partial_2^4\varphi\|_{\infty,\Omega} + \|\partial_3^4\varphi\|_{\infty,\Omega}) \end{aligned}$$

erhalten. Die Werte unseres Lösungsvektors u werden also wie h^2 gegen die Werte des Vektors $\hat{u}$ konvergieren, der die *exakten* Werte der Lösung φ in den Gitterpunkten x_i enthält.

5.2 Beispiel: Grundwasserströmung

Die Bewegung von Wasser in einem porösen Medium, beispielsweise in der Erde, kann mit Hilfe des *Darcy'schen Gesetzes* beschrieben werden. Dieses Gesetz stellt eine Beziehung zwischen dem *Druck* her, unter dem das Wasser steht, und dem *Fluss* des Wassers.

Der Druck in einem Gebiet $\Omega \subseteq \mathbb{R}^3$ ist eine skalarwertige Funktion $p : \Omega \to \mathbb{R}$, die jedem Punkt des Gebiets die Kraft zuordnet, die pro Fläche in ihm wirkt.

Der Fluss ist ein Vektorfeld $f : \Omega \to \mathbb{R}^3$, das angibt, wieviel Wasser pro Fläche austritt: Wenn $n \in \mathbb{R}^3$ der Normalenvektor einer Einheitsfläche ist, beschreibt $\langle f(x), n\rangle_2$, wieviel Wasser diese Fläche in Richtung des Vektors überschreitet.

Darcy'sches Gesetz. Das *Darcy'sche Gesetz* besagt, dass der Fluss direkt proportional zu dem Gradienten des Drucks ist, dass also

$$f(x) + k(x)\nabla p(x) = 0 \qquad \text{für alle } x \in \Omega \tag{5.13}$$

gilt. Die Größe $k(x)$ beschreibt die Durchlässigkeit des Materials in einem Punkt $x \in \Omega$: Wenn $k(x)$ groß ist, fließt das Wasser besonders schnell. Ein Blick auf die Taylor-Entwicklung zeigt, dass der Gradient gerade diejenige Richtung beschreibt, in der der Druck am schnellsten zunimmt, und die Gleichung besagt, dass das Wasser in die entgegengesetzte Richtung fließt. Diese Eigenschaft wird in der Strömungsdynamik als *Impulserhaltung* bezeichnet.

Massenerhaltung. Man sieht leicht, dass durch die Gleichung (5.13) Fluss und Druck nicht eindeutig bestimmt sind: Für eine beliebige Funktion φ können wir zu f den Gradienten $\nabla\varphi$ hinzuaddieren und $k\varphi$ von p subtrahieren, ohne die Gültigkeit der Gleichung zu verlieren. Wir müssen noch eine weitere Gleichung hinzunehmen, die eine weiteres physikalisches Gesetz beschreibt: Wasser kann nicht einfach entstehen oder verschwinden, und es kann auch (fast) nicht komprimiert werden. Dieses Gesetz der *Massenerhaltung* lässt sich knapp als

$$\nabla \cdot f(x) = 0 \qquad \text{für alle } x \in \Omega \tag{5.14}$$

formulieren. Es stellt sich die Frage, wieso das Verschwinden der Divergenz etwas mit der Massenerhaltung zu tun haben sollte.

Gauß'scher Integralsatz. Die Antwort erhalten wir mit Hilfe des *Gauß'schen Integralsatzes*. Wir untersuchen einen Quader

$$Q := [a_1 - h/2, a_1 + h/2] \times [a_2 - h/2, a_2 + h/2] \times [a_3 - h/2, a_3 + h/2]$$

mit den Randflächen

$$\begin{aligned}
R_1 &:= [a_1 - h/2, a_1 + h/2] \times [a_2 - h/2, a_2 + h/2] \times \{a_3 - h/2\},\\
R_2 &:= [a_1 - h/2, a_1 + h/2] \times [a_2 - h/2, a_2 + h/2] \times \{a_3 + h/2\},\\
R_3 &:= [a_1 - h/2, a_1 + h/2] \times \{a_2 - h/2\} \times [a_3 - h/2, a_3 + h/2],\\
R_4 &:= [a_1 - h/2, a_1 + h/2] \times \{a_2 + h/2\} \times [a_3 - h/2, a_3 + h/2],\\
R_5 &:= \{a_1 - h/2\} \times [a_2 - h/2, a_2 + h/2] \times [a_3 - h/2, a_3 + h/2],\\
R_6 &:= \{a_1 + h/2\} \times [a_2 - h/2, a_2 + h/2] \times [a_3 - h/2, a_3 + h/2],
\end{aligned}$$

die die nach außen zeigenden Einheitsnormalenvektoren

$$\begin{aligned} n_1 &:= (0,0,-1), & n_2 &:= (0,0,1),\\ n_3 &:= (0,-1,0), & n_4 &:= (0,1,0),\\ n_5 &:= (-1,0,0), & n_6 &:= (1,0,0) \end{aligned}$$

besitzen und durch die Funktionen

$$\begin{aligned} \Psi_1 &: [-h/2,h/2]^2 \to R_1, & s &\mapsto (a_1+s_1, a_2-s_2, a_3-h/2),\\ \Psi_2 &: [-h/2,h/2]^2 \to R_2, & s &\mapsto (a_1+s_1, a_2+s_2, a_3+h/2),\\ \Psi_3 &: [-h/2,h/2]^2 \to R_3, & s &\mapsto (a_1-s_1, a_2-h/2, a_3+s_2),\\ \Psi_4 &: [-h/2,h/2]^2 \to R_4, & s &\mapsto (a_1+s_1, a_2+h/2, a_3+s_2),\\ \Psi_5 &: [-h/2,h/2]^2 \to R_5, & s &\mapsto (a_1-h/2, a_2+s_1, a_3-s_2),\\ \Psi_6 &: [-h/2,h/2]^2 \to R_6, & s &\mapsto (a_1+h/2, a_2+s_1, a_3+s_2), \end{aligned}$$

parametrisiert werden. Der Gauß'sche Integralsatz besagt, dass das Integral der Divergenz eines Vektorfelds über das Volumen gleich dem Integral seiner Normalenkomponente über die Oberfläche ist. In unserem Fall nimmt dieses Resultat die Form

$$0 = \int_Q \nabla \cdot f(x)\,dx = \sum_{j=1}^{6} \int_{R_j} \langle f(x), n_j\rangle_2\,dx \tag{5.15}$$

an, wobei die Oberflächenintegrale wie üblich durch

$$\begin{aligned} \int_{R_j} g(x)\,dx &= \int_{-h/2}^{h/2}\int_{-h/2}^{h/2} \sqrt{\det(D\Psi_j^*(\hat{x})D\Psi_j(\hat{x}))}\, g(\Psi_j(\hat{x}))\,d\hat{x}_2\,d\hat{x}_1\\ &= \int_{-h/2}^{h/2}\int_{-h/2}^{h/2} g(\Psi_j(\hat{x}))\,d\hat{x}_2\,d\hat{x}_1 \qquad \text{für alle } j\in\{1,\dots,6\} \end{aligned}$$

definiert sind. Diese Gleichung lässt sich anschaulich interpretieren: Jedes Oberflächenintegral beschreibt die Menge des Wassers, die auf der entsprechenden Seitenfläche aus dem Quader heraus (bei einem positiven Vorzeichen) oder in den Quader hinein (bei einem negativen) fließt. Die Voraussetzung, dass die Summe dieser Mengen gleich null ist, besagt gerade, dass genau soviel Wasser hinein- wie herausfließen muss.

Diskretisierung. Um die Grundwasserströmung im Rechner simulieren zu können, müssen wir wieder eine Diskretisierung durchführen, die zu einem Gleichungssystem mit endlich vielen Unbekannten führt. Dazu approximieren wir die in (5.15) auftretenden Integrale mit der *Mittelpunktregel*.

Lemma 5.3 (Mittelpunktregel) *Seien $h \in \mathbb{R}_{>0}$ und eine Funktion $g \in C^2[-h/2,h/2]$ gegeben. Dann existiert ein $\eta \in [-h/2,h/2]$ mit*

$$\int_{-h/2}^{h/2} g(s)\,ds - hg(0) = \frac{h^3}{24} g''(\eta)$$

Beweis. Wir nutzen aus, dass

$$\int_{-h/2}^{h/2} 1\,ds = h, \qquad \int_{-h/2}^{h/2} s\,ds = 0$$

gelten, um

$$\int_{-h/2}^{h/2} g(s)\,ds - hg(0) = \int_{-h/2}^{h/2} g(s) - g(0) - sg'(0)\,ds$$

zu erhalten. Wir stellen fest, dass der Integrand eine doppelte Nullstelle im Nullpunkt besitzt. Um diese Eigenschaft auszunutzen, zerlegen wir das Integral in zwei Teilintegrale über die Intervalle $[0, h/2]$ und $[-h/2, 0]$. Mit der Funktion $\varphi_+(s) := (s-h/2)^2/2$, deren zweite Ableitung gleich eins ist und die eine doppelte Nullstelle in $s = h/2$ besitzt, erhalten wir per partieller Integration

$$\begin{aligned}
&\int_0^{h/2} g(s) - g(0) - sg'(0)\,ds \\
&\quad = \int_0^{h/2} \varphi_+''(s)(g(s) - g(0) - sg'(0))\,ds \\
&\quad = \left[\varphi_+'(s)(g(s) - g(0) - sg'(0))\right]_0^{h/2} - \int_0^{h/2} \varphi_+'(s)(g'(s) - g'(0))\,ds \\
&\quad = -\int_0^{h/2} \varphi_+'(s)(g'(s) - g'(0))\,ds \\
&\quad = -\left[\varphi_+(s)(g'(s) - g'(0))\right]_0^{h/2} + \int_0^{h/2} \varphi_+(s)g''(s)\,ds \\
&\quad = \int_0^{h/2} \varphi_+(s)g''(s)\,ds.
\end{aligned}$$

Da die Funktion φ_+ nicht-negativ ist, können wir den Mittelwertsatz der Integralrechnung verwenden, um ein $\eta_+ \in [0, h/2]$ mit

$$\int_0^{h/2} \varphi_+(s)g''(s)\,ds = g''(\eta_+)\int_0^{h/2} \varphi_+(s)\,ds = g''(\eta_+)\frac{h^3}{48}$$

zu finden. Wir können die Funktion $\varphi_-(s) := (s+h/2)^2/2$ verwenden, um für das zweite Integral die Gleichung

$$\int_{-h/2}^{0} g(s) - g(0) - sg'(0)\,ds = g''(\eta_-)\frac{h^3}{48}$$

mit einem $\eta_- \in [-h/2, 0]$ zu erhalten. Mit dem Zwischenwertsatz finden wir ein $\eta \in [-h/2, h/2]$ mit

$$g''(\eta) = \frac{g''(\eta_+) + g''(\eta_-)}{2},$$

so dass wir

$$\int_{-h/2}^{h/2} g(s)\,ds - hg(0) = \frac{h^3}{48}(g''(\eta_+) + g''(\eta_-)) = \frac{h^3}{24}g''(\eta)$$

bewiesen haben. ∎

Diskrete Massenerhaltung. Wir verwenden wieder die Gitterpunkte x_i, $i \in \overline{\mathcal{I}}$, die schon im Fall der Potentialgleichung zum Einsatz gekommen sind. Indem wir die Gleichung (5.15) auf einen Quader $Q = [x_{i,1}, x_{i,1}+h] \times [x_{i,2}, x_{i,2}+h] \times [x_{i,3}, x_{i,3}+h]$ mit dem Mittelpunkt $a = (x_{i,1}+h/2, x_{i,2}+h/2, x_{i,3}+h/2)$ anwenden und die Oberflächenintegrale mit der Mittelpunktregel approximieren, erhalten wir die Näherung

$$\begin{aligned} 0 &= \sum_{j=1}^{6} \int_{R_j} \langle f(x), n_j\rangle_2\,dx = \sum_{j=1}^{6} \int_{-h/2}^{h/2}\int_{-h/2}^{h/2} \langle f(\Psi_j(s)), n_j\rangle_2\,ds_2\,ds_1 \\ &\approx \sum_{j=1}^{6} h^2 \langle f(\Psi_j(0,0), n_j\rangle_2, \end{aligned}$$

in der nur noch die Normalenkomponenten des Flusses in den Mittelpunkten der sechs Seitenflächen auftreten. Diese Werte verwenden wir als Unbekannte, also

$$\begin{aligned} f_{x,i} &:= f_1(h(i_x, i_y + 1/2, i_z + 1/2)) && \text{für } i_x \in \{0,\dots,n+1\},\ i_y, i_z \in \{0,\dots,n\}, \\ f_{y,i} &:= f_2(h(i_x + 1/2, i_y, i_z + 1/2)) && \text{für } i_y \in \{0,\dots,n+1\},\ i_x, i_z \in \{0,\dots,n\}, \\ f_{z,i} &:= f_3(h(i_x + 1/2, i_y + 1/2, i_z)) && \text{für } i_z \in \{0,\dots,n+1\},\ i_x, i_y \in \{0,\dots,n\}, \end{aligned}$$

und unter Berücksichtigung der Vorzeichen der Normalenvektoren erhalten wir die genäherte Massenerhaltungsgleichung

$$\begin{aligned} 0 \approx h^2(&-f_{x,i} + f_{x,i+\delta_1} - f_{y,i} + f_{y,i+\delta_2} \\ &- f_{z,i} + f_{z,i+\delta_3}) \qquad \text{für alle } i_x, i_y, i_z \in \{0,\dots,n\}. \end{aligned} \tag{5.16}$$

Diskrete Darcy-Gleichung. Die Darcy-Gleichung (5.13) diskretisieren wir, indem wir fordern, dass sie punkt- und komponentenweise gilt, dass also

$$\begin{aligned} f_{x,i} &= -k(h(i_x, i_y + 1/2, i_z + 1/2))\partial_1 p(h(i_x, i_y + 1/2, i_z + 1/2)) \\ &\qquad \text{für alle } i_x \in \{1,\dots,n\},\ i_y, i_z \in \{0,\dots,n\}, \\ f_{y,i} &= -k(h(i_x + 1/2, i_y, i_z + 1/2))\partial_2 p(h(i_x + 1/2, i_y, i_z + 1/2)) \\ &\qquad \text{für alle } i_y \in \{1,\dots,n\},\ i_x, i_z \in \{0,\dots,n\}, \\ f_{z,i} &= -k(h(i_x + 1/2, i_y + 1/2, i_z))\partial_3 p(h(i_x + 1/2, i_y + 1/2, i_z)) \\ &\qquad \text{für alle } i_z \in \{1,\dots,n\},\ i_x, i_y \in \{0,\dots,n\} \end{aligned}$$

gilt. Um die Diskretisierung zu vervollständigen, müssen wir auch den kontinuierlichen Druck durch endlich viele Werte beschreiben. Diese Werte sollten so gewählt sein, dass sich die Ableitungen $\partial_1 p$, $\partial_2 p$ und $\partial_3 p$ in den Seitenmittelpunkten gut approximieren lassen. Eine gute Näherung bietet der *zentrale Differenzenquotient*:

Lemma 5.4 (Zentraler Differenzenquotient) *Seien ein $h \in \mathbb{R}_{>0}$ und eine Funktion $g \in C^3[-h/2, h/2]$ gegeben. Dann existiert ein $\eta \in [-h/2, h/2]$ mit*

$$\frac{g(h/2) - g(-h/2)}{h} - g'(0) = \frac{h^2}{24} g'''(\eta).$$

Beweis. Mit dem Taylor'schen Satz finden wir $\eta_+, \eta_- \in [-h/2, h/2]$ mit

$$\begin{aligned} g(h/2) &= g(0) + \frac{h}{2} g'(0) + \frac{h^2}{8} g''(0) + \frac{h^3}{48} g'''(\eta_+), \\ g(-h/2) &= g(0) - \frac{h}{2} g'(0) + \frac{h^2}{8} g''(0) - \frac{h^3}{48} g'''(\eta_-). \end{aligned}$$

Durch Subtraktion der Gleichungen folgt

$$\begin{aligned} g(h/2) - g(-h/2) &= h g'(0) + \frac{h^3}{48}(g'''(\eta_+) + g'''(\eta_-)), \\ \frac{g(h/2) - g(-h/2)}{h} &= g'(0) + \frac{h^2}{48}(g'''(\eta_+) + g'''(\eta_-)), \end{aligned}$$

und mit dem Zwischenwertsatz finden wir ein $\eta \in [-h/2, h/2]$ mit

$$g'''(\eta) = \frac{g'''(\eta_+) + g'''(\eta_-)}{2}.$$

Damit ergibt sich

$$\frac{g(h/2) - g(-h/2)}{h} = g'(0) + \frac{h^2}{24} g'''(\eta),$$

und Subtraktion von $g'(h/2)$ führt zu der gewünschten Gleichung. ∎

Indem wir die in der Darcy-Gleichung auftretenden partiellen Ableitungen mit Hilfe des zentralen Differenzenquotienten approximieren, erhalten wir die Näherungen

$$\begin{aligned} \partial_1 p(h(i_x, i_y + 1/2, i_z + 1/2)) \approx \frac{1}{h}(&p(h(i_x + 1/2, i_y + 1/2, i_z + 1/2)) \\ &- p(h(i_x - 1/2, i_y + 1/2, i_z + 1/2))), \\ \partial_2 p(h(i_x + 1/2, i_y, i_z + 1/2)) \approx \frac{1}{h}(&p(h(i_x + 1/2, i_y + 1/2, i_z + 1/2)) \\ &- p(h(i_x + 1/2, i_y - 1/2, i_z + 1/2))), \\ \partial_3 p(h(i_x + 1/2, i_y + 1/2, i_z)) \approx \frac{1}{h}(&p(h(i_x + 1/2, i_y + 1/2, i_z + 1/2)) \\ &- p(h(i_x + 1/2, i_y + 1/2, i_z - 1/2))), \end{aligned}$$

die nur noch die Werte des Drucks in den Mittelpunkten der Quader benötigen. Also wählen wir diese Werte als unsere Druck-Unbekannten

$$p_i := p(h(i_x + 1/2, i_y + 1/2, i_z + 1/2)) \qquad \text{für alle } i_x, i_y, i_z \in \{0, \ldots, n\}$$

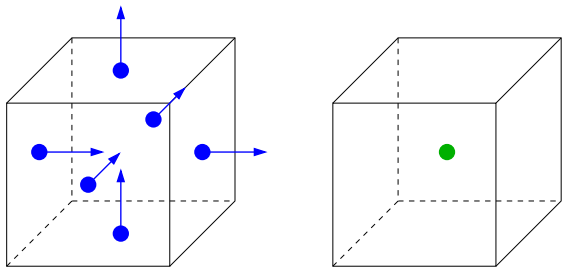

Abbildung 5.1: Diskretisierung der Darcy-Gleichung: Das Gebiet wird in Würfel der Kantenlänge h zerlegt, Freiheitsgrade sind die Normalenanteile des Flusses in den Mittelpunkten der Seitenflächen (links) und der Druck im Mittelpunkt jedes Würfels (rechts).

und können die Darcy-Gleichung in der Form

$$\begin{aligned}
f_{x,i} &\approx -\frac{k(h(i_x, i_y + 1/2, i_z + 1/2))}{h}(p_i - p_{i-\delta_1}) \quad \text{für } i_x \in \{1,\dots,n\},\ i_y, i_z \in \{0,\dots,n\},\\
f_{y,i} &\approx -\frac{k(h(i_x + 1/2, i_y, i_z + 1/2))}{h}(p_i - p_{i-\delta_2}) \quad \text{für } i_y \in \{1,\dots,n\},\ i_x, i_z \in \{0,\dots,n\},\\
f_{z,i} &\approx -\frac{k(h(i_x + 1/2, i_y + 1/2, i_z))}{h}(p_i - p_{i-\delta_3}) \quad \text{für } i_z \in \{1,\dots,n\},\ i_x, i_y \in \{0,\dots,n\}
\end{aligned}$$

approximieren. Nun steht uns alles Nötige zur Verfügung, um die diskrete Formulierung zu definieren. Wir beginnen mit den Indexmengen

$$\begin{aligned}
\mathcal{I}_p &:= \{0,\dots,n\} \times \{0,\dots,n\} \times \{0,\dots,n\},\\
\mathcal{I}_x &:= \{0,\dots,n\} \times \{1,\dots,n\} \times \{1,\dots,n\},\\
\overline{\mathcal{I}}_x &:= \{0,\dots,n\} \times \{0,\dots,n+1\} \times \{0,\dots,n+1\},\\
\mathcal{I}_y &:= \{1,\dots,n\} \times \{0,\dots,n\} \times \{1,\dots,n\},\\
\overline{\mathcal{I}}_y &:= \{0,\dots,n+1\} \times \{0,\dots,n\} \times \{0,\dots,n+1\},\\
\mathcal{I}_z &:= \{1,\dots,n\} \times \{1,\dots,n\} \times \{0,\dots,n\},\\
\overline{\mathcal{I}}_z &:= \{0,\dots,n+1\} \times \{0,\dots,n+1\} \times \{0,\dots,n\}
\end{aligned}$$

für den Druck und die x-, y- und z-Anteile des Flusses. Die Positionen der Freiheitsgrade innerhalb der Gitterwürfel sind in Abbildung 5.1 illustriert.

Matrixdarstellung der Darcy-Gleichung. Die Darcy-Gleichung schreiben wir in der Form

$$\begin{aligned}
\frac{1}{k(h(i_x, i_y + 1/2, i_z + 1/2))} f_{x,i} &\approx -\frac{p_i - p_{i-\delta_1}}{h} \qquad \text{für alle } i = (i_x, i_y, i_z) \in \mathcal{I}_x,\\
\frac{1}{k(h(i_x + 1/2, i_y, i_z + 1/2))} f_{y,i} &\approx -\frac{p_i - p_{i-\delta_2}}{h} \qquad \text{für alle } i = (i_x, i_y, i_z) \in \mathcal{I}_y,
\end{aligned}$$

$$\frac{1}{k(h(i_x+1/2,i_y+1/2,i_z))} f_{z,i} \approx -\frac{p_i - p_{i-\delta_3}}{h} \qquad \text{für alle } i=(i_x,i_y,i_z) \in \mathcal{I}_z,$$

die der Gleichung

$$\begin{pmatrix} A_x & & \\ & A_y & \\ & & A_z \end{pmatrix} \begin{pmatrix} f_x|_{\mathcal{I}_x} \\ f_y|_{\mathcal{I}_y} \\ f_z|_{\mathcal{I}_z} \end{pmatrix} \approx - \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} p \tag{5.17}$$

mit den durch

$$a_{x,i,j} := \begin{cases} 1/k(h(i_x,i_y+1/2,i_z+1/2)) & \text{falls } j=i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i,j \in \mathcal{I}_x,$$

$$a_{y,i,j} := \begin{cases} 1/k(h(i_x+1/2,i_y,i_z+1/2)) & \text{falls } j=i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i,j \in \mathcal{I}_y,$$

$$a_{z,i,j} := \begin{cases} 1/k(h(i_x+1/2,i_y+1/2,i_z)) & \text{falls } j=i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i,j \in \mathcal{I}_z$$

definierten Matrizen $A_x \in \mathbb{R}^{\mathcal{I}_x \times \mathcal{I}_x}$, $A_y \in \mathbb{R}^{\mathcal{I}_y \times \mathcal{I}_y}$ und $A_z \in \mathbb{R}^{\mathcal{I}_z \times \mathcal{I}_z}$ sowie den durch

$$b_{x,i,j} := \begin{cases} 1/h & \text{falls } j=i, \\ -1/h & \text{falls } j=i-\delta_1, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I}_x,\ j \in \mathcal{I}_p,$$

$$b_{y,i,j} := \begin{cases} 1/h & \text{falls } j=i, \\ -1/h & \text{falls } j=i-\delta_2, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I}_y,\ j \in \mathcal{I}_p,$$

$$b_{z,i,j} := \begin{cases} 1/h & \text{falls } j=i, \\ -1/h & \text{falls } j=i-\delta_3, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I}_x,\ j \in \mathcal{I}_p$$

definierten Matrizen $B_x \in \mathbb{R}^{\mathcal{I}_x \times \mathcal{I}_p}$, $B_y \in \mathbb{R}^{\mathcal{I}_y \times \mathcal{I}_p}$ und $B_z \in \mathbb{R}^{\mathcal{I}_z \times \mathcal{I}_p}$ entspricht.

Matrixdarstellung der Massenerhaltungsgleichung. Die approximative Massenerhaltungsgleichung (5.16) dividieren wir durch $-h^3$, um die Gleichung

$$\frac{1}{h} f_{x,i} - \frac{1}{h} f_{x,i+\delta_1} + \frac{1}{h} f_{y,i} - \frac{1}{h} f_{y,i+\delta_2} + \frac{1}{h} f_{z,i} - \frac{1}{h} f_{z,i+\delta_3} \approx 0 \qquad \text{für alle } i \in \mathcal{I}_p$$

zu erhalten, die sich kompakt als

$$\begin{pmatrix} D_x & D_y & D_z \end{pmatrix} \begin{pmatrix} f_x \\ f_y \\ f_z \end{pmatrix} \approx 0$$

mit den durch

$$d_{x,i,j} := \begin{cases} 1/h & \text{falls } j = i, \\ -1/h & \text{falls } j = i + \delta_1, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I}_p,\ j \in \overline{\mathcal{I}}_x,$$

$$d_{y,i,j} := \begin{cases} 1/h & \text{falls } j = i, \\ -1/h & \text{falls } j = i + \delta_2, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I}_p,\ j \in \overline{\mathcal{I}}_y,$$

$$d_{z,i,j} := \begin{cases} 1/h & \text{falls } j = i, \\ -1/h & \text{falls } j = i + \delta_3, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I}_p,\ j \in \overline{\mathcal{I}}_z$$

definierten Matrizen $D_x \in \mathbb{R}^{\mathcal{I}_p \times \overline{\mathcal{I}}_x}$, $D_y \in \mathbb{R}^{\mathcal{I}_p \times \overline{\mathcal{I}}_y}$ und $D_z \in \mathbb{R}^{\mathcal{I}_p \times \overline{\mathcal{I}_z}}$ schreiben lässt.

Durch die neue Skalierung der Gleichung erhalten wir

$$\begin{aligned} D_x|_{\mathcal{I}_p \times \mathcal{I}_x} &= B_x^*, \\ D_y|_{\mathcal{I}_p \times \mathcal{I}_y} &= B_y^*, \\ D_z|_{\mathcal{I}_p \times \mathcal{I}_z} &= B_z^*, \end{aligned}$$

können also die approximative Massenerhaltungsgleichung in der Form

$$\begin{pmatrix} B_x^* & B_y^* & B_z^* \end{pmatrix} \begin{pmatrix} f_x|_{\mathcal{I}_x} \\ f_y|_{\mathcal{I}_y} \\ f_z|_{\mathcal{I}_z} \end{pmatrix} \approx \begin{pmatrix} -D_x|_{\mathcal{I}_p \times \partial\mathcal{I}_x} & -D_y|_{\mathcal{I}_p \times \partial\mathcal{I}_y} & -D_z|_{\mathcal{I}_p \times \partial\mathcal{I}_z} \end{pmatrix} \begin{pmatrix} f_x|_{\partial\mathcal{I}_x} \\ f_y|_{\partial\mathcal{I}_y} \\ f_z|_{\partial\mathcal{I}_z} \end{pmatrix} \tag{5.18}$$

mit den Indexmengen $\partial\mathcal{I}_x := \overline{\mathcal{I}}_x \setminus \mathcal{I}_x$, $\partial\mathcal{I}_y := \overline{\mathcal{I}}_y \setminus \mathcal{I}_y$, $\partial\mathcal{I}_z := \overline{\mathcal{I}}_z \setminus \mathcal{I}_z$ für die Randfreiheitsgrade schreiben. Insgesamt erhalten wir das System

$$\begin{pmatrix} A_x & & & B_x \\ & A_y & & B_y \\ & & A_z & B_z \\ B_x^* & B_y^* & B_z^* & \end{pmatrix} \begin{pmatrix} f_x|_{\mathcal{I}_x} \\ f_y|_{\mathcal{I}_y} \\ f_z|_{\mathcal{I}_z} \\ p \end{pmatrix} \approx \begin{pmatrix} 0 \\ 0 \\ 0 \\ b \end{pmatrix},$$

$$\begin{pmatrix} -D_x|_{\mathcal{I}_p \times \partial\mathcal{I}_x} & -D_y|_{\mathcal{I}_p \times \partial\mathcal{I}_y} & -D_z|_{\mathcal{I}_p \times \partial\mathcal{I}_z} \end{pmatrix} \begin{pmatrix} f_x|_{\partial\mathcal{I}_x} \\ f_y|_{\partial\mathcal{I}_y} \\ f_z|_{\partial\mathcal{I}_z} \end{pmatrix} \approx b.$$

Mit Hilfe der zweiten Gleichung können wir für den gegebenen Fluss am Rand des Gebiets eine Näherung

$$\tilde{b} := \begin{pmatrix} -D_x|_{\mathcal{I}_p \times \partial\mathcal{I}_x} & -D_y|_{\mathcal{I}_p \times \partial\mathcal{I}_y} & -D_z|_{\mathcal{I}_p \times \partial\mathcal{I}_z} \end{pmatrix} \begin{pmatrix} f_x|_{\partial\mathcal{I}_x} \\ f_y|_{\partial\mathcal{I}_y} \\ f_z|_{\partial\mathcal{I}_z} \end{pmatrix}$$

des Vektors b bestimmen, mit dessen Hilfe wir durch Auflösen der ersten Gleichung

$$\begin{pmatrix} A_x & & & B_x \\ & A_y & & B_y \\ & & A_z & B_z \\ B_x^* & B_y^* & B_z^* & \end{pmatrix} \begin{pmatrix} \tilde{f}_x|_{\mathcal{I}_x} \\ \tilde{f}_y|_{\mathcal{I}_y} \\ \tilde{f}_z|_{\mathcal{I}_z} \\ \tilde{p} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \tilde{b} \end{pmatrix} \tag{5.19}$$

den Druck und den Fluss im Inneren des Gebiets approximieren können. Dank unserer Konstruktion ist die Gesamtmatrix symmetrisch, kann allerdings wegen des singulären rechten unteren Blocks insgesamt nicht positiv definit sein, während die ersten drei Diagonalblöcke diese Eigenschaft besitzen. Es handelt sich um ein sogenanntes *Sattelpunktproblem*, das sich mit spezialisierten Verfahren lösen lässt.

5.3 Beispiel: Elektromagnetische Wellen

Den Fall eines von der Zeit unabhängigen elektrischen Felds haben wir bereits behandelt, er führt zu der relativ einfach zu diskretisierenden Potentialgleichung (5.9). Wenn wir zulassen, dass das elektrische Feld sich zeitabhängig verändert, besagt das Ampère'sche Gesetz (5.3), dass das magnetische Feld verändert wird, das wiederum über das Faraday'sche Gesetz (5.2) das elektrische Feld beeinflusst: Es entstehen elektromagnetische Wellen.

Indem wir das Faraday'schen Gesetz (5.2) mit dem Materialgesetz (5.7) kombinieren, erhalten wir

$$\nabla \times E(t,x) = -\frac{\partial B}{\partial t}(t,x) = -\mu \frac{\partial H}{\partial t}(t,x) \qquad \text{für alle} t \in [a,b],\ x \in \Omega. \tag{5.20}$$

Entsprechend ergibt sich aus dem Ampère'schen Gesetz (5.3) in Kombination mit dem Materialgesetz (5.6) die Gleichung

$$\nabla \times H(t,x) = j(t,x) + \frac{\partial D}{\partial t}(t,x) = j(t,x) + \epsilon \frac{\partial E}{\partial t}(t,x) \qquad \text{für alle } t \in [a,b],\ x \in \Omega. \tag{5.21}$$

Unser Ziel besteht nun darin, diese Gleichungen zu diskretisieren, um beide Felder näherungsweise berechnen zu können.

Im Fall der Darcy-Gleichung verhilft uns der Gauß'sche Integralsatz zu der Darstellung (5.15), die als Ausgangspunkt für die Diskretisierung des Massenerhaltungsgesetzes diente. Im Kontext der Maxwell-Gleichungen verwenden wir den *Stokes'schen Integralsatz*, um die auftretenden Rotationsoperatoren in eine handlichere Form zu überführen. Dazu untersuchen wir das in einem Punkt $a \in [h/2, 1-h/2] \times [h/2, 1-h/2] \times [0,1]$ zentrierte Quadrat

$$Q := [a - h/2, a + h/2] \times [a - h/2, a + h/2] \times \{a_3\}$$

mit dem Einheitsnormalenvektor

$$n := (0,0,1)$$

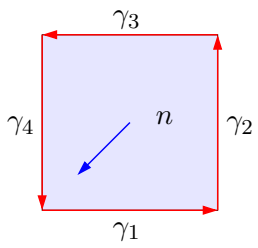

Abbildung 5.2: Quadrat und umlaufende Kurve für die Anwendung des Stokes'schen Integralsatzes

und der durch die Parametrisierungen

$$\begin{aligned}
\gamma_1 &: [-h/2, h/2] \to \partial Q, & s &\mapsto (a_1 + s, a_2 - h/2, a_3),\\
\gamma_2 &: [-h/2, h/2] \to \partial Q, & s &\mapsto (a_1 + h/2, a_2 + s, a_3),\\
\gamma_3 &: [-h/2, h/2] \to \partial Q, & s &\mapsto (a_1 - s, a_2 + h/2, a_3),\\
\gamma_4 &: [-h/2, h/2] \to \partial Q, & s &\mapsto (a_1 - h/2, a_2 - s, a_3)
\end{aligned}$$

beschriebenen Randkurve, die das Quadrat Q „aus Richtung des Normalenvektors gesehen" im mathematisch positiven Sinn umläuft (vgl. Abbildung 5.2).

In diesem Kontext besagt der Stokes'sche Integralsatz, dass sich das Integral des Skalarprodukts der Rotation mit dem Normalenvektor über das Quadrat als das Integral der Tangentialkomponenten des Felds entlang der Randkurve schreiben lässt, also in der Form

$$\int_Q \langle \nabla \times E(t,x), n\rangle_2 \, dx = \sum_{k=1}^{4} \int_{\gamma_k} \langle E(t,x), \gamma_k'\rangle_2 \, dx.$$

Indem wir Q durch die Parametrisierung

$$\Psi : [-h/2, h/2] \times [-h/2, h/2] \to Q, \qquad s \mapsto (a_1 + s_1, a_2 + s_2, a_3),$$

beschreiben, erhalten wir

$$\int_{-h/2}^{h/2} \int_{-h/2}^{h/2} \langle \nabla \times E(t, \Psi(s)), n\rangle_2 \, ds_2 \, ds_1 = \sum_{k=1}^{4} \int_{-h/2}^{h/2} \langle E(t, \gamma_k(s)), \gamma_k'\rangle_2 \, ds,$$

wobei wir $\sqrt{\det(D\Psi^* D\Psi)} = 1$ sowie $\|\gamma_k'\|_2 = 1$ ausnutzen. Wir setzen (5.20) ein und gelangen zu der Gleichung

$$-\int_{-h/2}^{h/2} \int_{-h/2}^{h/2} \mu \langle \frac{\partial H}{\partial t}(t, \Psi(s)), n\rangle_2 \, dx = \sum_{k=1}^{4} \int_{-h/2}^{h/2} \langle E(t, \gamma_k(s)), \gamma_k'\rangle_2 \, ds.$$

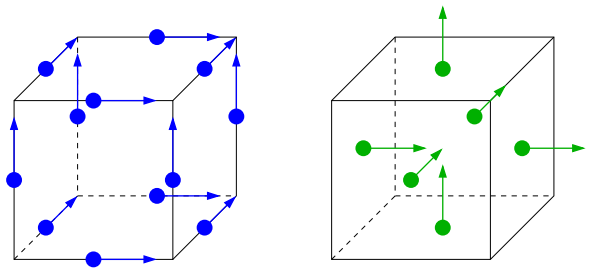

Abbildung 5.3: Beziehung zwischen E- und H-Gitter bei der Yee-Methode

Mit Hilfe der Mittelpunktregel (vgl. Lemma 5.3) folgt

$$-h^2\mu\langle\frac{\partial H}{\partial t}(t,a),n\rangle_2 \approx h\langle E(t,a-h/2\delta_2),\delta_1\rangle_2 + h\langle E(t,a+h/2\delta_1),\delta_2\rangle_2 \\ + h\langle E(t,a+h/2\delta_2),-\delta_1\rangle_2 + h\langle E(t,a-h/2\delta_1),-\delta_2\rangle_2,$$

und wir können durch $-\mu h^2$ dividieren und die besonders einfache Form der Vektoren n, δ_1 und δ_2 ausnutzen, um

$$\frac{\partial H_3}{\partial t}(t,a) \approx \frac{E_2(t,a-h/2\delta_1)-E_2(t,a+h/2\delta_1)}{\mu h} \\ + \frac{E_1(t,a+h/2\delta_2)-E_1(t,a-h/2\delta_2)}{\mu h} \tag{5.22}$$

zu erhalten.

Freiheitsgrade. Da bei dieser Gleichung nur die Komponenten der Vektorfelder E und H in bestimmten Punkten auftreten, können wir sie als Grundlage einer Diskretisierung verwenden. Das Yee-Verfahren geht dabei von den Werten

$$\begin{aligned}
E_{x,i}(t) &:= E_1(t,h(i_x+1/2,i_y,i_z)) && \text{für } i=(i_x,i_y,i_z)\in\overline{\mathcal{I}}_x,\\
E_{y,i}(t) &:= E_2(t,h(i_x,i_y+1/2,i_z)) && \text{für } i=(i_x,i_y,i_z)\in\overline{\mathcal{I}}_y,\\
E_{z,i}(t) &:= E_3(t,h(i_x,i_y,i_z+1/2)) && \text{für } i=(i_x,i_y,i_z)\in\overline{\mathcal{I}}_z,\\
H_{x,i}(t) &:= H_1(t,h(i_x,i_y+1/2,i_z+1/2)) && \text{für } i=(i_x,i_y,i_z)\in\mathcal{J}_x,\\
H_{y,i}(t) &:= H_2(t,h(i_x+1/2,i_y,i_z+1/2)) && \text{für } i=(i_x,i_y,i_z)\in\mathcal{J}_y,\\
H_{z,i}(t) &:= H_3(t,h(i_x+1/2,i_y+1/2,i_z)) && \text{für } i=(i_x,i_y,i_z)\in\mathcal{J}_z
\end{aligned}$$

mit den Indexmengen

$$\begin{aligned}
\overline{\mathcal{I}}_x &:= \{0,\ldots,n\}\times\{0,\ldots,n+1\}\times\{0,\ldots,n+1\},\\
\overline{\mathcal{I}}_y &:= \{0,\ldots,n+1\}\times\{0,\ldots,n\}\times\{0,\ldots,n+1\},\\
\overline{\mathcal{I}}_z &:= \{0,\ldots,n+1\}\times\{0,\ldots,n+1\}\times\{0,\ldots,n\},
\end{aligned}$$

$$
\begin{aligned}
\mathcal{I}_x &:= \{0,\dots,n\} \times \{1,\dots,n\} \times \{1,\dots,n\},\\
\mathcal{I}_y &:= \{1,\dots,n\} \times \{0,\dots,n\} \times \{1,\dots,n\},\\
\mathcal{I}_z &:= \{1,\dots,n\} \times \{1,\dots,n\} \times \{0,\dots,n\},\\
\mathcal{J}_x &:= \{1,\dots,n\} \times \{0,\dots,n\} \times \{0,\dots,n\},\\
\mathcal{J}_y &:= \{0,\dots,n\} \times \{1,\dots,n\} \times \{0,\dots,n\},\\
\mathcal{J}_z &:= \{0,\dots,n\} \times \{0,\dots,n\} \times \{1,\dots,n\}
\end{aligned}
$$

aus (vgl Abbildung 5.3).

Diskretes Faraday'sches Gesetz. Die Gleichung (5.22) und ihre Entsprechungen für Quadrate in zy- und xz-Ebenen lassen sich damit kurz in der Form

$$\frac{\partial H_{x,i}}{\partial t} \approx \frac{1}{\mu h}(E_{z,i} - E_{z,i+\delta_2} + E_{y,i+\delta_3} - E_{y,i}) \qquad \text{für alle } i \in \mathcal{J}_x, \tag{5.23a}$$

$$\frac{\partial H_{y,i}}{\partial t} \approx \frac{1}{\mu h}(E_{x,i} - E_{x,i+\delta_3} + E_{z,i+\delta_1} - E_{z,i}) \qquad \text{für alle } i \in \mathcal{J}_y, \tag{5.23b}$$

$$\frac{\partial H_{z,i}}{\partial t} \approx \frac{1}{\mu h}(E_{y,i} - E_{y,i+\delta_1} + E_{x,i+\delta_2} - E_{x,i}) \qquad \text{für alle } i \in \mathcal{J}_z \tag{5.23c}$$

zusammenfassen. Diese drei Formeln lassen sich mit Hilfe von Matrizen $D_{xy} \in \mathbb{R}^{\mathcal{J}_x \times \overline{\mathcal{I}}_y}$, $D_{xz} \in \mathbb{R}^{\mathcal{J}_x \times \overline{\mathcal{I}}_z}$, $D_{yx} \in \mathbb{R}^{\mathcal{J}_y \times \overline{\mathcal{I}}_x}$, $D_{yz} \in \mathbb{R}^{\mathcal{J}_y \times \overline{\mathcal{I}}_z}$, $D_{zx} \in \mathbb{R}^{\mathcal{J}_z \times \overline{\mathcal{I}}_x}$ und $D_{zy} \in \mathbb{R}^{\mathcal{J}_z \times \overline{\mathcal{I}}_y}$ mit den Einträgen

$$d_{xy,ij} := \begin{cases} 1/h & \text{falls } j = i + \delta_3, \\ -1/h & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{J}_x,\ j \in \overline{\mathcal{I}}_y, \tag{5.24a}$$

$$d_{xz,ij} := \begin{cases} 1/h & \text{falls } j = i + \delta_2, \\ -1/h & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{J}_x,\ j \in \overline{\mathcal{I}}_z, \tag{5.24b}$$

$$d_{yx,ij} := \begin{cases} 1/h & \text{falls } j = i + \delta_3, \\ -1/h & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{J}_y,\ i \in \overline{\mathcal{I}}_x, \tag{5.24c}$$

$$d_{yz,ij} := \begin{cases} 1/h & \text{falls } j = i + \delta_1, \\ -1/h & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{J}_y,\ i \in \overline{\mathcal{I}}_z, \tag{5.24d}$$

$$d_{zx,ij} := \begin{cases} 1/h & \text{falls } j = i + \delta_2, \\ -1/h & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{J}_z,\ i \in \overline{\mathcal{I}}_x, \tag{5.24e}$$

$$d_{zy,ij} := \begin{cases} 1/h & \text{falls } j = i + \delta_1, \\ -1/h & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{J}_z,\ i \in \overline{\mathcal{I}}_y \tag{5.24f}$$

in der kurzen Form

$$\frac{\partial}{\partial t} \begin{pmatrix} H_x \\ H_y \\ H_z \end{pmatrix} \approx -\frac{1}{\mu} \begin{pmatrix} 0 & -D_{xy} & D_{xz} \\ D_{yx} & 0 & -D_{yz} \\ -D_{zx} & D_{zy} & 0 \end{pmatrix} \begin{pmatrix} E_x \\ E_y \\ E_z \end{pmatrix}$$

darstellen. Jede der Teilmatrizen entspricht dabei einer Approximation der partiellen Ableitungen in x-, y- oder z-Richtung mittels des in Lemma 5.4 untersuchten zentralen Differenzenquotienten, die 3×3-Blockmatrix stellt damit eine Approximation des Rotationsoperators dar, die gesamte Gleichung approximiert das Faraday'sche Gesetz (5.20).

Randbedingungen. Wie schon im Fall der Poisson- und Darcy-Gleichung müssen wir Randbedingungen vorsehen, in diesem Fall die Tangentialanteile des elektrischen Felds E, die gerade durch die Indexmengen

$$\partial\mathcal{I}_x := \overline{\mathcal{I}}_x \setminus \mathcal{I}_x, \qquad \partial\mathcal{I}_y := \overline{\mathcal{I}}_y \setminus \mathcal{I}_y, \qquad \partial\mathcal{I}_z := \overline{\mathcal{I}}_z \setminus \mathcal{I}_z$$

beschrieben sind. Dazu setzen wir

$$\begin{aligned} A_{xy} &:= D_{xy}|_{\mathcal{J}_x \times \mathcal{I}_y}, & A_{xz} &:= D_{xz}|_{\mathcal{J}_x \times \mathcal{I}_z}, \\ A_{yx} &:= D_{yx}|_{\mathcal{J}_y \times \mathcal{I}_x}, & A_{yz} &:= D_{yz}|_{\mathcal{J}_y \times \mathcal{I}_z}, \\ A_{zx} &:= D_{zx}|_{\mathcal{J}_z \times \mathcal{I}_x}, & A_{zy} &:= D_{zy}|_{\mathcal{J}_z \times \mathcal{I}_y} \end{aligned}$$

und erhalten

$$\begin{aligned} \frac{\partial}{\partial t} \begin{pmatrix} H_x \\ H_y \\ H_z \end{pmatrix} &\approx -\frac{1}{\mu} \begin{pmatrix} 0 & -A_{xy} & A_{xz} \\ A_{yx} & 0 & -A_{yz} \\ -A_{zx} & A_{zy} & 0 \end{pmatrix} \begin{pmatrix} E_x|_{\mathcal{I}_x} \\ E_y|_{\mathcal{I}_y} \\ E_z|_{\mathcal{I}_z} \end{pmatrix} + \begin{pmatrix} R_x \\ R_y \\ R_z \end{pmatrix}, \\ \begin{pmatrix} R_x \\ R_y \\ R_z \end{pmatrix} &:= -\frac{1}{\mu} \begin{pmatrix} 0 & -D_{xy}|_{\mathcal{J}_x \times \partial\mathcal{I}_y} & D_{xz}|_{\mathcal{J}_x \times \partial\mathcal{I}_z} \\ D_{yx}|_{\mathcal{J}_y \times \partial\mathcal{I}_x} & 0 & -D_{yz}|_{\mathcal{J}_y \times \partial\mathcal{I}_z} \\ -D_{zx}|_{\mathcal{J}_z \times \partial\mathcal{I}_x} & D_{zy}|_{\mathcal{J}_z \times \partial\mathcal{I}_y} & 0 \end{pmatrix} \begin{pmatrix} E_x|_{\partial\mathcal{I}_x} \\ E_y|_{\partial\mathcal{I}_y} \\ E_z|_{\partial\mathcal{I}_z} \end{pmatrix}, \end{aligned} \tag{5.25}$$

wobei die Vektoren R_x, R_y und R_z den Einfluss der konstanten Randwerte beschreiben. Im Fall eines supraleitenden Rands ist die Tangentialkomponente des elektrischen Felds gleich null, so dass R_x, R_y und R_z verschwinden und nur die Gleichung (5.25) übrig bleibt.

Diskretes Ampère'sches Gesetz. Um das elektromagnetische Feld vollständig zu beschreiben, müssen wir auch den Einfluss berücksichtigen, den das magnetische Feld auf das elektrische ausübt, also die Ampère'sche Gleichung (5.21). Indem wir beide Seiten

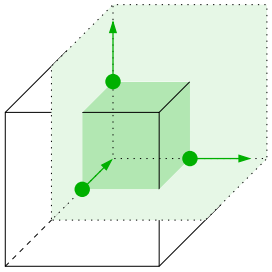

Abbildung 5.4: Duales Gitter im Yee-Verfahren: Indem die Integrationsflächen für das Ampère'sche Gesetz gegenüber denen für das Faraday'sche Gesetz um $h/2$ verschoben werden, lassen sich beide Gesetze mit denselben Unbekannten approximieren.

der Gleichung wieder über Q integrieren, erhalten wir mit dem Stokes'schen Satz die Gleichung

$$\epsilon \int_Q \langle \frac{\partial E}{\partial t}(t,x), n\rangle_2 \, dx + \int_Q \langle j(t,x), n\rangle_2 \, dx = \sum_{k=1}^{4} \int_{\gamma_k} \langle H(t,x), \gamma_k'\rangle_2 \, dx,$$

die wir per Mittelpunktregel (vgl. Lemma 5.3) in die Approximation

$$\begin{aligned}\epsilon h^2 \frac{\partial E_3}{\partial t}(t,a) + h^2 j_3(t,a) \approx h(&H_2(t, a + h/2\delta_1) - H_2(t, a - h/2\delta_1) \\ &+ H_1(t, a - h/2\delta_2) - H_1(t, a + h/2\delta_2))\end{aligned}$$

überführen können. Per Division durch ϵh^2 erhalten wir

$$\begin{aligned}\frac{\partial E_3}{\partial t}(t,a) + j_3(t,a) \approx \frac{1}{\epsilon h}(&H_2(t, a + h/2\delta_1) - H_2(t, a - h/2\delta_1) \\ &+ H_1(t, a - h/2\delta_2) - H_1(t, a + h/2\delta_2)).\end{aligned}$$

Die Eleganz des Yee-Verfahrens liegt darin, dass die Punkte a, in denen diese Gleichung gilt, so gewählt werden, dass genau dieselben Variablen wie im Fall des diskretisierten Faraday'schen Gesetzes auftreten (vgl. Abbildung 5.4).

Während wir im Fall des Faraday'schen Gesetzes die Flächen mit Mittelpunkt $a = h(i_x + 1/2, i_y + 1/2, i_z)$ verwendet haben, benutzen wir für das Faraday'sche Gesetz die Flächen mit dem Mittelpunkt $a = h(i_x, i_y, i_z + 1/2)$, verschieben sie also um $-h/2$ in x- und y-Richtung und um $h/2$ in z-Richtung. Dann erhalten wir

$$\begin{aligned}\frac{\partial E_3}{\partial t}(t,a) &= \frac{\partial E_{z,i}}{\partial t}(t), \\ H_2(t, a + h/2\delta_1) &= H_2(t, h(i_x + 1/2, i_y, i_z + 1/2)) = H_{y,i}(t), \\ H_2(t, a - h/2\delta_1) &= H_2(t, h(i_x - 1/2, i_y, i_z + 1/2)) = H_{y,i-\delta_1}(t),\end{aligned}$$

$$H_1(t, a - h/2\delta_2) = H_1(t, h(i_x, i_y - 1/2, i_z + 1/2)) = H_{x,i-\delta_2},$$
$$H_1(t, a + h/2\delta_2) = H_1(t, h(i_x, i_y + 1/2, i_z + 1/2)) = H_{x,i}$$

und die Gleichung

$$\frac{\partial E_{z,i}}{\partial t} + j(t, h(i_x, i_y, i_z + 1/2)) \approx \frac{1}{\epsilon h}(H_{y,i} - H_{y,i-\delta_1} + H_{x,i-\delta_2} - H_{x,i}).$$

Ein Blick auf die Indexmengen zeigt, dass die Bedingungen

$$i_x \in \{1, \dots, n\}, \qquad i_y \in \{1, \dots, n\}, \qquad i_z \in \{0, \dots, n\}$$

gelten müssen, die zusammen gerade $i \in \mathcal{I}_z$ entsprechen. Die Werte der Stromdichte j in diesen Punkten fassen wir in den Vektoren $j_x \in \mathbb{R}^{\mathcal{I}_x}$, $j_y \in \mathbb{R}^{\mathcal{I}_y}$, $j_z \in \mathbb{R}^{\mathcal{I}_z}$ zusammen, die durch

$$\begin{aligned}
j_{x,i}(t) &:= j(t, h(i_x + 1/2, i_y, i_z)) && \text{für alle } i = (i_x, i_y, i_z) \in \mathcal{I}_x,\\
j_{y,i}(t) &:= j(t, h(i_x, i_y + 1/2, i_z)) && \text{für alle } i = (i_x, i_y, i_z) \in \mathcal{I}_y,\\
j_{z,i}(t) &:= j(t, h(i_x, i_y, i_z + 1/2)) && \text{für alle } i = (i_x, i_y, i_z) \in \mathcal{I}_z
\end{aligned}$$

definiert sind.

Für die x- und y-Komponenten des elektrischen Felds verwenden wir Quadrate parallel zu der zy- oder xz-Ebene und erhalten

$$\begin{aligned}
\frac{\partial E_{x,i}}{\partial t} &\approx \frac{1}{\epsilon h}(H_{z,i} - H_{z,i-\delta_2} + H_{y,i-\delta_3} - H_{y,i}) - j_{x,i} && \text{für alle } i \in \mathcal{I}_x,\\
\frac{\partial E_{y,i}}{\partial t} &\approx \frac{1}{\epsilon h}(H_{x,i} - H_{x,i-\delta_3} + H_{z,i-\delta_1} - H_{z,i}) - j_{y,i} && \text{für alle } i \in \mathcal{I}_y,\\
\frac{\partial E_{z,i}}{\partial t} &\approx \frac{1}{\epsilon h}(H_{y,i} - H_{y,i-\delta_1} + H_{x,i-\delta_2} - H_{x,i}) - j_{z,i} && \text{für alle } i \in \mathcal{I}_z.
\end{aligned}$$

Ein Vergleich mit (5.24) erlaubt es uns, diese Gleichungen kompakt in der Form

$$\frac{\partial}{\partial t}\begin{pmatrix} E_x|_{\mathcal{I}_x} \\ E_y|_{\mathcal{I}_y} \\ E_z|_{\mathcal{I}_z} \end{pmatrix} \approx \frac{1}{\epsilon}\begin{pmatrix} 0 & A_{yx}^* & -A_{zx}^* \\ -A_{xy}^* & 0 & A_{zy}^* \\ A_{xz}^* & -A_{yz}^* & 0 \end{pmatrix}\begin{pmatrix} H_x \\ H_y \\ H_z \end{pmatrix} - \begin{pmatrix} j_x \\ j_y \\ j_z \end{pmatrix} \tag{5.26}$$

zu schreiben. Diese Formel ist das diskrete Gegenstück des Ampère'schen Gesetzes.

Zeitdiskretisierung. Sowohl (5.25) als auch (5.26) weisen noch eine Zeitableitung auf, die in der Regel ebenfalls numerisch approximiert werden muss. Im Prinzip könnten wir sämtliche in Kapitel 4 beschriebenen Verfahren einsetzen, bei den impliziten Verfahren wäre dazu lediglich ein lineares Gleichungssystem in jedem Schritt zu lösen.

Eleganter ist es allerdings, die besondere Struktur der Gleichungen ausnutzen: In (5.25) hängt die Ableitung des magnetischen Felds nur von den Werten des elektrischen Felds ab, in (5.26) gilt das Gegenteil. Die Idee des *Leapfrog-Verfahrens* besteht darin, diesen Zusammenhang auszunutzen. Um eine Verwechslung mit der Ortsschrittweite h zu

vermeiden bezeichnen wir die Zeitschrittweite mit $\delta \in \mathbb{R}_{>0}$. Wir approximieren die Zeitableitung in (5.25) durch die zentrale Differenz (vgl. Lemma 5.4) und erhalten

$$\begin{aligned}\frac{1}{\delta}\left[\begin{pmatrix}H_x(t+\delta/2)\\H_y(t+\delta/2)\\H_z(t+\delta/2)\end{pmatrix} - \begin{pmatrix}H_x(t-\delta/2)\\H_y(t-\delta/2)\\H_z(t-\delta/2)\end{pmatrix}\right] &\approx \frac{\partial}{\partial t}\begin{pmatrix}H_x(t)\\H_y(t)\\H_z(t)\end{pmatrix}\\ &\approx -\frac{1}{\mu}\begin{pmatrix}0 & -A_{xy} & A_{xz}\\ A_{yx} & 0 & -A_{yz}\\ -A_{zx} & A_{zy} & 0\end{pmatrix}\begin{pmatrix}E_x(t)|_{\mathcal{I}_x}\\E_y(t)|_{\mathcal{I}_y}\\E_z(t)|_{\mathcal{I}_z}\end{pmatrix} + \begin{pmatrix}R_x\\R_y\\R_z\end{pmatrix}.\end{aligned}$$

Indem wir die Daten für den Zeitpunkt $t-\delta/2$ auf die rechte Seite der Gleichung bringen und mit δ multiplizieren, erhalten wir

$$\begin{pmatrix}H_x(t+\delta/2)\\H_y(t+\delta/2)\\H_z(t+\delta/2)\end{pmatrix} \approx \begin{pmatrix}H_x(t-\delta/2)\\H_y(t-\delta/2)\\H_z(t-\delta/2)\end{pmatrix} - \frac{\delta}{\mu}\begin{pmatrix}0 & -A_{xy} & A_{xz}\\ A_{yx} & 0 & -A_{yz}\\ -A_{zx} & A_{zy} & 0\end{pmatrix}\begin{pmatrix}E_x(t)|_{\mathcal{I}_x}\\E_y(t)|_{\mathcal{I}_y}\\E_z(t)|_{\mathcal{I}_z}\end{pmatrix} + \delta\begin{pmatrix}R_x\\R_y\\R_z\end{pmatrix}.$$

Entsprechend können wir mit dem diskretisierten Gegenstück (5.26) der Ampère'schen Gleichung verfahren, um zu der Formel

$$\begin{pmatrix}E_x(t+\delta/2)|_{\mathcal{I}_x}\\E_y(t+\delta/2)|_{\mathcal{I}_y}\\E_z(t+\delta/2)|_{\mathcal{I}_z}\end{pmatrix} \approx \begin{pmatrix}E_x(t-\delta/2)|_{\mathcal{I}_x}\\E_y(t-\delta/2)|_{\mathcal{I}_y}\\E_z(t-\delta/2)|_{\mathcal{I}_z}\end{pmatrix} + \frac{\delta}{\epsilon}\begin{pmatrix}0 & A^*_{yz} & -A^*_{zx}\\ -A^*_{xy} & 0 & A^*_{zy}\\ A^*_{xz} & -A^*_{yz} & 0\end{pmatrix}\begin{pmatrix}H_x(t)\\H_y(t)\\H_z(t)\end{pmatrix} - \delta\begin{pmatrix}j_x(t)\\j_y(t)\\j_z(t)\end{pmatrix}$$

zu gelangen. Die Idee des Leapfrog-Verfahrens (zu deutsch ist *leapfrog* ungefähr als „Bockspringen" zu übersetzen) besteht darin, die beiden Felder zu unterschiedlichen Zeitpunkten zu berechnen: Wenn $a \in \mathbb{R}$ der Anfangszeitpunkt ist, wird das elektrische Feld zu den Zeitpunkten $a, a+\delta, a+2\delta, a+3\delta, \ldots$ berechnet, während das magnetische zu den Zeitpunkten $a+\delta/2, a+3\delta/2, a+5\delta/2, a+7\delta/2, \ldots$ berechnet wird. Ausgehend von Anfangswerten

$$\begin{pmatrix}\widetilde{E}_x(a)\\\widetilde{E}_y(a)\\\widetilde{E}_z(a)\end{pmatrix} := \begin{pmatrix}E_x(a)|_{\mathcal{I}_x}\\E_y(a)|_{\mathcal{I}_y}\\E_z(a)|_{\mathcal{I}_z}\end{pmatrix}, \qquad \begin{pmatrix}\widetilde{H}_x(a+\delta/2)\\\widetilde{H}_y(a+\delta/2)\\\widetilde{H}_z(a+\delta/2)\end{pmatrix} := \begin{pmatrix}H_x(a+\delta/2)\\H_y(a+\delta/2)\\H_z(a+\delta/2)\end{pmatrix}$$

definieren sich die Näherungslösungen des Yee-Verfahrens dann durch

$$\begin{aligned}\begin{pmatrix}\widetilde{E}_x(a+(m+1)\delta)\\\widetilde{E}_y(a+(m+1)\delta)\\\widetilde{E}_z(a+(m+1)\delta)\end{pmatrix} &:= \begin{pmatrix}\widetilde{E}_x(a+m\delta)\\\widetilde{E}_y(a+m\delta)\\\widetilde{E}_z(a+m\delta)\end{pmatrix} - \delta\begin{pmatrix}j_x(a+(m+1/2)\delta)\\j_y(a+(m+1/2)\delta)\\j_z(a+(m+1/2)\delta)\end{pmatrix}\\ &\quad + \frac{\delta}{\epsilon}\begin{pmatrix}0 & A^*_{yz} & -A^*_{zx}\\ -A^*_{xy} & 0 & A^*_{zy}\\ A^*_{xz} & -A^*_{yz} & 0\end{pmatrix}\begin{pmatrix}\widetilde{H}_x(a+(m+1/2)\delta)\\\widetilde{H}_y(a+(m+1/2)\delta)\\\widetilde{H}_z(a+(m+1/2)\delta)\end{pmatrix},\\ \begin{pmatrix}\widetilde{H}_x(a+(m+3/2)\delta)\\\widetilde{H}_y(a+(m+3/2)\delta)\\\widetilde{H}_z(a+(m+3/2)\delta)\end{pmatrix} &:= \begin{pmatrix}\widetilde{H}_x(a+(m+1/2)\delta)\\\widetilde{H}_y(a+(m+1/2)\delta)\\\widetilde{H}_z(a+(m+1/2)\delta)\end{pmatrix} + \delta\begin{pmatrix}R_x\\R_y\\R_z\end{pmatrix}\end{aligned}$$

$$-\frac{\delta}{\mu}\begin{pmatrix} 0 & -A_{xy} & A_{xz} \\ A_{yx} & 0 & -A_{yz} \\ -A_{zx} & A_{zy} & 0 \end{pmatrix}\begin{pmatrix} \widetilde{E}_x(t+(m+1)\delta) \\ \widetilde{E}_y(t+(m+1)\delta) \\ \widetilde{E}_z(t+(m+1)\delta) \end{pmatrix}$$

für alle $m \in \mathbb{N}_0$ definiert.

6 Schnelle Lösungsverfahren für lineare Gleichungssysteme

Die in Kapitel 5 auftretenden linearen Gleichungssysteme werden in der Regel schnell groß: In Abhängigkeit von der Schrittweite h des Gitters verhält sich der Fehler ungefähr wie h^2, wir werden also relativ kleine Schrittweiten in Kauf nehmen müssen, wenn wir an einer hohen Genauigkeit interessiert sind. Die Anzahl der Unbekannten ist proportional zu n^3 mit $n \approx 1/h$, also werden wir es in der Regel mit sehr vielen Unbekannten zu tun bekommen. Beispielsweise führt $n = 100$ zu einer Genauigkeit von ungefähr vier Nachkommastellen, aber das Gleichungssystem enthält eine Million Unbekannte. Um lineare Gleichungssysteme dieser Größe lösen zu können, müssen Verfahren zum Einsatz kommen, die deren besondere Eigenschaften ausnutzen können.

6.1 LR-Zerlegung für Bandmatrizen

Ein klassisches Lösungsverfahren für lineare Gleichungssysteme haben wir bereits in Abschnitt 2.1 kennen gelernt: Die LR-Zerlegung, also die Zerlegung einer Matrix $A \in \mathbb{R}^{m\times m}$ in eine untere Dreiecksmatrix $L \in \mathbb{R}^{m\times m}$ und eine obere Dreiecksmatrix $R \in \mathbb{R}^{m\times m}$ mit

$$A = LR.$$

Im Allgemeinen beträgt der Rechenaufwand für diese Zerlegung ungefähr $2m^3/3$, wächst also kubisch mit der Anzahl der Unbekannten. Damit ist diese Form der LR-Zerlegung für die von uns untersuchten großen Gleichungssystem uninteressant.

Die LR-Zerlegung kann allerdings wesentlich effizienter werden, falls die Matrix A bestimmte Eigenschaften besitzt. Die einfachste solche Eigenschaft ist die beschränkte *Bandbreite* der Matrix.

Definition 6.1 (Bandbreite) *Sei $A \in \mathbb{R}^{m\times m}$, und seien $\ell, r \in \mathbb{N}_0$ gegeben. Falls*

$$a_{ij} = 0 \qquad \text{für alle } i,j \in \{1,\dots,m\} \text{ mit } j > i + r$$

gilt, nennen wir r eine obere Bandbreitenschranke *für die Matrix A. Falls dagegen*

$$a_{ij} = 0 \qquad \text{für alle } i,j \in \{1,\dots,m\} \text{ mit } i > j + \ell$$

gilt, nennen wir ℓ eine untere Bandbreitenschranke.

Die kleinste obere Bandbreitenschranke nennen wir die obere Bandbreite *der Matrix, die kleinste untere Bandbreitenschranke die* untere Bandbreite.

Eine Matrix ist genau dann eine untere Dreiecksmatrix, wenn ihre obere Bandbreite gleich null ist. Sie ist genau dann eine obere Dreiecksmatrix, wenn ihre untere Bandbreite gleich null ist. Und sie ist genau dann eine Diagonalmatrix, wenn obere und untere Bandbreite gleich null sind.

LR-Zerlegung bei beschränkter Bandbreite. Bei der Berechnung der LR-Zerlegung mit dem in Abschnitt 2.1 diskutierten Algorithmus bleiben untere und obere Bandbreite erhalten. Als Beispiel untersuchen wir die LR-Zerlegung der Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix},$$

die eine obere Bandbreitenschranke $r = 1$ und eine untere von $\ell = 2$ besitzt. Unser Algorithmus skaliert zunächst die erste Spalte der Matrix, um die erste Spalte der Matrix L und die erste Zeile der Matrix R zu gewinnen:

$$\begin{pmatrix} r_{11} & r_{12} & & & \\ \ell_{21} & a_{22} & a_{23} & & \\ \ell_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

Offenbar bleiben bei dieser Operation die Null-Einträge der Matrix unverändert. Der erste Schritt des Algorithmus subtrahiert das Produkt aus erster Spalte und erstem Vektor von dem rechten unteren Rest der Matrix, und dank der Null-Einträge sind davon nur zwei Koeffizienten betroffen:

$$A^{(1)} = \begin{pmatrix} r_{11} & r_{12} & & & \\ \ell_{21} & a_{22}^{(1)} & a_{23} & & \\ \ell_{31} & a_{32}^{(1)} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}.$$

Auch bei dieser Operation bleiben alle Null-Einträge erhalten. Wir können entsprechend fortfahren und der Reihe nach die Matrizen

$$A^{(2)} = \begin{pmatrix} r_{11} & r_{12} & & & \\ \ell_{21} & r_{22} & r_{23} & & \\ \ell_{31} & \ell_{32} & a_{33}^{(2)} & a_{34} & \\ & \ell_{42} & a_{43}^{(2)} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix}, \qquad A^{(3)} = \begin{pmatrix} r_{11} & r_{12} & & & \\ \ell_{21} & r_{22} & r_{23} & & \\ \ell_{31} & \ell_{32} & r_{33} & r_{34} & \\ & \ell_{42} & \ell_{43} & a_{44}^{(3)} & a_{45} \\ & & \ell_{53} & a_{54}^{(3)} & a_{55} \end{pmatrix},$$

$$A^{(4)} = \begin{pmatrix} r_{11} & r_{12} & & & \\ \ell_{21} & r_{22} & r_{23} & & \\ \ell_{31} & \ell_{32} & r_{33} & r_{34} & \\ & \ell_{42} & \ell_{43} & r_{44} & r_{45} \\ & & \ell_{53} & \ell_{54} & a_{55}^{(4)} \end{pmatrix}, \qquad A^{(5)} = \begin{pmatrix} r_{11} & r_{12} & & & \\ \ell_{21} & r_{22} & r_{23} & & \\ \ell_{31} & \ell_{32} & r_{33} & r_{34} & \\ & \ell_{42} & \ell_{43} & r_{44} & r_{45} \\ & & \ell_{53} & \ell_{54} & r_{55} \end{pmatrix}$$

berechnen. Offenbar müssen wir in jedem Schritt höchstens ℓ Divisionen und ℓr Multiplikationen und Subtraktionen durchführen, so dass sich die Anzahl der Rechenoperationen durch $m\ell(2r+1)$ beschränken lässt. Damit wächst der Rechenaufwand in diesem Fall lediglich *linear* mit der Anzahl der Unbekannten. Eine Implementierung des LR-Verfahrens, die Bandbreitenschranken ausnutzt, kann also auch für sehr große lineare Gleichungssysteme effizient sein. Besonders schnell wird ein derartiger Algorithmus, wenn wir die Einträge der Matrix „von links oben nach rechts unten“ im Speicher anordnen, denn dann sind in jedem Schritt des Verfahrens nur wenige im Speicher benachbarte Koeffizienten betroffen, so dass die Speicherzugriffe ein für den Prozessor günstiges Muster aufweisen.

6.2 LR-Zerlegung mit Gebietszerlegung

Die von uns im Kapitel 5 untersuchten Matrizen weisen Bandbreiten auf, die proportional zu $n^2 = m^{2/3}$ wachsen, so dass eine die Bandbreite ausnutzende Variante der LR-Zerlegung immer noch eine zu $n^7 = m^{7/3}$ proportionale Anzahl von Rechenoperationen benötigt.

Modellproblem. Es ist allerdings möglich, eine weitere Eigenschaft der Matrizen auszunutzen: Als Beispiel untersuchen wir das zweidimensionale Gegenstück der Potentialgleichung. Wir wählen $N \in \mathbb{N}_{\geq 3}$, setzen $h := 1/(N+1)$ und definieren die Matrix $A \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ für die Indexmenge

$$\mathcal{I} := \{1,\dots,N\} \times \{1,\dots,N\}$$

durch

$$a_{ij} = \begin{cases} 4h^{-2} & \text{falls } i = j, \\ -h^{-2} & \text{falls } |i_x - j_x| = 1,\ i_y = j_y, \\ -h^{-2} & \text{falls } |i_y - j_y| = 1,\ i_x = j_x \end{cases} \qquad \text{für alle } i = (i_x, i_y), j = (j_x, j_y) \in \mathcal{I}.$$

Sei $x \in \mathbb{R}^{\mathcal{I}}$. Die i-te Komponente des Vektors $y := Ax$ hängt dann lediglich von den unmittelbaren Nachbarn der Punkts i in dem durch die Indexmenge $\mathcal{I}$ beschriebenen Gitter ab.

Gebietszerlegung. Diese Eigenschaft können wir ausnutzen, um die Matrix A in eine Form zu bringen, die für unsere Zwecke nützlich ist. Dazu zerlegen wir die Indexmenge „horizontal“ in drei Teilmengen: Wir definieren $N_1 := \lfloor (N+1)/2 \rfloor$ und setzen

$$\mathcal{I}_1 := \{1,\dots,N_1-1\} \times \{1,\dots,N\}, \qquad \mathcal{I}_2 := \{N_1+1,\dots,N\} \times \{1,\dots,N\},$$

$$\mathcal{I}_0 := \{N_1\} \times \{1, \dots, N\}.$$

Anschaulich ist $\mathcal{I}_1$ die „linke Hälfte“ der Indexmenge, $\mathcal{I}_2$ die „rechte Hälfte“ und $\mathcal{I}_\partial$ die „Kontaktfläche“ der beiden.

Diese Unterteilung hat eine wichtige Konsequenz: Für jedes $i = (i_x, i_y) \in \mathcal{I}_1$ und jedes $j = (j_x, j_y) \in \mathcal{I}_2$ gilt $|i_x - j_x| > 1$, also folgt $a_{ij} = 0$. Kompakt lässt sich diese Eigenschaft in der Form

$$A|_{\mathcal{I}_1 \times \mathcal{I}_2} = 0, \qquad A|_{\mathcal{I}_2 \times \mathcal{I}_1} = 0$$

schreiben, zwei sehr große Blöcke von Matrixkoeffizienten sind also gleich null.

Struktur der Matrix. Indem wir die verbliebenen Teilmatrizen mit

$$\begin{gathered} A_{11} := A|_{\mathcal{I}_1 \times \mathcal{I}_1}, \qquad A_{22} := A|_{\mathcal{I}_2 \times \mathcal{I}_2}, \\ A_{10} := A|_{\mathcal{I}_1 \times \mathcal{I}_0}, \qquad A_{20} := A|_{\mathcal{I}_2 \times \mathcal{I}_0}, \qquad A_{01} := A|_{\mathcal{I}_0 \times \mathcal{I}_1}, \qquad A_{02} := A|_{\mathcal{I}_0 \times \mathcal{I}_2}, \\ A_{00} := A|_{\mathcal{I}_0 \times \mathcal{I}_0}, \end{gathered}$$

erhalten wir

$$A = \begin{pmatrix} A_{10} & & A_{10} \\ & A_{22} & A_{20} \\ A_{01} & A_{02} & A_{00} \end{pmatrix}.$$

Diese Form der Matrix wird im Englischen als *arrowhead structure* bezeichnet.

Block-LR-Zerlegung. Wenn wir voraussetzen, dass A_{11} und A_{22} invertierbar sind, können wir eine *Block-LR-Zerlegung* der Form

$$\begin{pmatrix} A_{11} & & A_{10} \\ & A_{22} & A_{20} \\ A_{01} & A_{02} & A_{00} \end{pmatrix} = \begin{pmatrix} I & & \\ & I & \\ A_{01}A_{11}^{-1} & A_{02}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & & A_{10} \\ & A_{22} & A_{20} \\ & & S_{00} \end{pmatrix}$$

mit dem *Schur-Komplement*

$$S_{00} := A_{00} - A_{01}A_{11}^{-1}A_{10} - A_{02}A_{22}^{-1}A_{20}$$

berechnen. Wie die übliche LR-Zerlegung können wir auch die Block-LR-Zerlegung verwenden, um das lineare Gleichungssystem aufzulösen: Wir setzen

$$\begin{aligned} x_1 &:= x|_{\mathcal{I}_1}, & x_2 &:= x|_{\mathcal{I}_2}, & x_0 &:= x|_{\mathcal{I}_0}, \\ b_1 &:= b|_{\mathcal{I}_1}, & b_2 &:= b|_{\mathcal{I}_2}, & b_0 &:= b|_{\mathcal{I}_0} \end{aligned}$$

und stellen das Gleichungssystem in der Form

$$\begin{pmatrix} A_{11} & & A_{10} \\ & A_{22} & A_{20} \\ A_{01} & A_{02} & A_{00} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_0 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_0 \end{pmatrix}$$

dar, die zu

$$\begin{pmatrix} I & & \\ & I & \\ A_{01}A_{11}^{-1} & A_{02}A_{22}^{-1} & I \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_0 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_0 \end{pmatrix},$$
$$\begin{pmatrix} A_{11} & & A_{10} \\ & A_{22} & A_{20} \\ & & S_{00} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_0 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_0 \end{pmatrix}$$

äquivalent ist.

Block-Einsetzen. Da beide Gleichungssysteme Block-Dreiecksstruktur aufweisen, bietet es sich an, sie durch Block-Vorwärtseinsetzen und Block-Rückwärtseinsetzen zu lösen. Für das erste System in unterer Block-Dreiecksform erhalten wir so

$$\begin{aligned} y_1 &= b_1, \\ y_2 &= b_2, \\ y_0 &= b_0 - A_{01}A_{11}^{-1}y_1 - A_{02}A_{22}^{-1}y_2, \end{aligned}$$

während für das zweite System das Rückwärtseinsetzen sinnvoll ist, nämlich

$$\begin{aligned} x_0 &= S_{00}^{-1}y_0, \\ x_2 &= A_{22}^{-1}y_2 - A_{22}^{-1}A_{20}x_0, \\ x_1 &= A_{11}^{-1}y_1 - A_{11}^{-1}A_{10}x_0. \end{aligned}$$

Indem wir Hilfsmatrizen

$$B_{10} := A_{11}^{-1}A_{10}, \qquad B_{20} := A_{22}^{-1}A_{20}$$

verwenden und Hilfsvektoren $z_1 \in \mathbb{R}^{\mathcal{I}_1}, z_2 \in \mathbb{R}^{\mathcal{I}_2}$ einführen, können wir den Lösungsalgorithmus in die folgende Form bringen:

$$\begin{aligned} y_1 = b_1, \qquad & y_2 = b_2, \\ z_1 = A_{11}^{-1}y_1, \qquad & z_2 = A_{22}^{-1}y_2, \\ y_0 = b_0 - A_{01}z_1 - A_{02}z_2, & \\ x_0 = S_{00}^{-1}y_0, & \\ x_2 = z_2 - B_{20}x_0, \qquad & x_1 = z_1 - B_{10}x_0. \end{aligned}$$

Falls wir dazu in der Lage sind, die Gleichungssysteme

$$A_{11}z_1 = y_1, \qquad A_{22}z_2 = y_2, \qquad S_{00}x_0 = y_0$$

aufzulösen, die in der zweiten und vierten Zeile des Algorithmus auftreten, können wir also mit geringem Mehraufwand auch die Lösung des Gesamtsystems berechnen.

procedure model_mvm(x, α, **var** y);
for $i_x, i_y \in \{1, \ldots, N\}$ **do**
$y_{i_x,i_y} \leftarrow y_{i_x,i_y} + \alpha h^{-2}(4x_{i_x,i_y} - x_{i_x+1,i_y} - x_{i_x-1,i_y} - x_{i_x,i_y+1} - x_{i_x,i_y-1})$

Abbildung 6.1: Matrix-Vektor-Multiplikation $y \leftarrow y + \alpha Ax$ für die Matrix A des Modellproblems

Schur-Komplement-System. Das Schur-Komplement-System ist relativ zugänglich: Da $\mathcal{I}_0$ nur N Elemente enthält, ist die Matrix S_{00} gemessen an A_{11} und A_{22} relativ klein, also eventuell schon für einfache direkte Lösungsverfahren zugänglich. Bei genauerer Betrachtung stellt sich heraus, dass S_{00} eine besondere Struktur aufweist (bestimmte Teilmatrizen außerhalb der Diagonalen lassen sich in spezieller Form darstellen), die sich ausnutzen lässt, um effizient mit dem Schur-Komplement umzugehen.

Rekursion. Die Behandlung der Systeme mit den Matrizen A_{11} und A_{22} kann *rekursiv* erfolgen: Die Struktur beider Matrizen entspricht der der ursprünglichen Matrix A, allerdings sind sie ungefähr halb so groß. Sollten sie noch zu groß für die Behandlung mit einer einfachen LR-Zerlegung sein, können wir deshalb auch die zu $\mathcal{I}_1$ und $\mathcal{I}_2$ gehörenden Indexmengen wieder halbieren, diesmal eventuell in horizontaler Richtung, und die gesuchten Lösungen aus Lösungen der Teilprobleme zusammensetzen.

Parallelisierung. Ein weiterer Vorteil der Zerlegung besteht darin, dass die Berechnung der Hilfsvektoren z_1 und z_2 sowie der Komponenten x_1 und x_2 des Endergebnisses unabhängig voneinander geschehen kann. Falls also ein Computer zum Einsatz kommt, der über mehrere Recheneinheiten verfügt, können sie parallel rechnen.

Die beschriebene Vorgehensweise ist sehr erfolgreich bei zweidimensionalen Problemen, weil die Indexmenge $\mathcal{I}_0$ zu einem eindimensionalen Problem führt. Bei der Behandlung dreidimensionaler Probleme können wir zwar auch erheblich effizienter als mit einer einfachen LR-Zerlegung rechnen, allerdings bleibt der Rechenaufwand relativ hoch.

6.3 Einfache iterative Verfahren

Verfahren auf Grundlage der LR-Zerlegung benötigen zusätzlichen Speicher, um beispielsweise das Schur-Komplement darzustellen. Wir haben bereits gesehen, dass es bei Finite-Differenzen-Verfahren möglich ist, die Matrix-Vektor-Multiplikation praktisch ohne Hilfsspeicher durchzuführen, es müssen lediglich der Eingabe- und der Ausgabevektor aufbewahrt werden. Für das Modellproblem genügt beispielsweise der in Abbildung 6.1 angegebene Algorithmus, der die Rechenoperation $y \leftarrow y + \alpha Ax$ ausführt.

Bei großen linearen Gleichungssystemen kann dieser sparsame Umgang mit dem Speicher den Unterschied zwischen der Durchführbarkeit einer Berechnung und dem Scheitern ausmachen, also interessieren wir uns für Verfahren, die mit Operationen ähnlich der Matrix-Vektor-Multiplikation auskommen können.

Bei der Behandlung diskretisierter Differentialgleichungen kommt ein weiterer Aspekt hinzu: Die Lösung des linearen Gleichungssystems stellt lediglich eine Näherung der Lösung der Differentialgleichung dar. Falls also schon ein *Diskretisierungsfehler* proportional zu h^2 im Spiel ist, genügt es, auch die Lösung des linearen Gleichungssystems mit einer vergleichbaren Genauigkeit zu bestimmen.

Iterationsverfahren. Näherungslösungen lassen sich sehr elegant mit Hilfe eines *Iterationsverfahrens* berechnen: Wir gehen davon aus, dass eine invertierbare Matrix $A \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ und eine *rechte Seite* $b \in \mathbb{R}^{\mathcal{I}}$ gegeben sind und wir eine Lösung $x = A^{-1}b$ des Gleichungssystems

$$Ax = b \tag{6.1}$$

berechnen wollen. Wir definieren eine *Iterationsfunktion*

$$\Phi : \mathbb{R}^{\mathcal{I}} \times \mathbb{R}^{\mathcal{I}} \to \mathbb{R}^{\mathcal{I}},$$

die aus einer *Anfangsnäherung* $x^{(0)} \in \mathbb{R}^{\mathcal{I}}$ eine hoffentlich verbesserte Näherung $x^{(1)} = \Phi(x^{(0)}, b)$ berechnet. Indem wir Φ wiederholt anwenden, dürfen wir hoffen, dass die durch

$$x^{(m+1)} := \Phi(x^{(m)}, b) \qquad \text{für alle } m \in \mathbb{N}_0$$

definierte *Folge der Iterierten* gegen einen Grenzwert

$$x^* = \lim_{m\to\infty} x^{(m)}$$

konvergiert, der das Gleichungssystem $Ax^* = b$ löst.

Lineare Iterationsverfahren. Da wir ein lineares Gleichungssystem lösen wollen, bietet es sich an, auch *lineare Iterationsverfahren* zu verwenden, die sich in der Form

$$\Phi(x, b) = Mx + Nb \qquad \text{für alle } x, b \in \mathbb{R}^{\mathcal{I}} \tag{6.2}$$

darstellen lassen.

Konsistenz. Offensichtlich sind nur Iterationsfunktionen von Interesse, die die Qualität der Näherung nicht verschlechtern, also sollte insbesondere die exakte Lösung $x = A^{-1}b$ ein *Fixpunkt* in dem Sinn

$$\Phi(x, b) = x \tag{6.3}$$

sein. Im Fall eines linearen Verfahrens ist diese Gleichung äquivalent zu

$$x = \Phi(x, b) = Mx + Nb = Mx + NAx = (M + NA)x.$$

Um die Eigenschaft (6.3) für alle Vektoren $b \in \mathbb{R}^{\mathcal{I}}$ mit zugehörigen Lösungen $x = A^{-1}b$ sicher zu stellen, bietet es sich demnach an, zu fordern, dass die Gleichung

$$M = I - NA \tag{6.4}$$

gilt. Ein derartiges Verfahren bezeichnen wir als *konsistent*, und durch Einsetzen in (6.2) ergibt sich die Darstellung

$$\Phi(x, b) = x - N(Ax - b) \qquad \text{für alle } x, b \in \mathbb{R}^{\mathcal{I}}. \tag{6.5}$$

Fehlerdarstellung. Für konsistente Verfahren lässt sich der Approximationsfehler besonders einfach untersuchen: Wenn wir die Fehler durch

$$e^{(m)} := x^{(m)} - x^* \qquad \text{für alle } m \in \mathbb{N}_0$$

definieren, erhalten wir

$$\begin{aligned} e^{(m+1)} = x^{(m+1)} - x^* &= \Phi(x^{(m)}, b^*) - \Phi(x^*, b^*) = Mx^{(m)} + Nb^* - Mx^* - Nb^* \\ &= Mx^{(m)} - Mx^* = M(x^{(m)} - x^*) = Me^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0. \end{aligned}$$

Wir suchen also nach Verfahren, die einerseits die Konsistenzbedingung (6.4) erfüllen und bei denen andererseits die Multiplikation mit der Matrix M Vektoren schnell schrumpfen lässt. Ideal wäre natürlich ein Verfahren mit $M = 0$, das den Fehler sofort auf null reduzieren würde. Aus (6.4) folgt für ein derartiges Verfahren bereits $N = A^{-1}$, wir müssten also die Inverse bereits kennen, um es durchführen zu können, und in diesem Fall hätten wir es nicht nötig, es durchzuführen.

Immerhin legt dieser Fall nahe, dass es eine gute Idee ist, die Matrix N so zu wählen, dass sie einerseits effizient auf einen Vektor angewendet werden kann und andererseits eine gute Näherung der Inversen der Matrix A darstellt.

Richardson-Iteration. Eines der einfachsten linearen Iterationsverfahren ist die (stationäre) *Richardson-Iteration*, bei der wir $N = \theta I$ mit einer Zahl $\theta \in \mathbb{R}$ wählen. Die Iterationsvorschrift nimmt dann die Form

$$\Phi_{\mathrm{Rich},\theta}(x, b) = x - \theta(Ax - b) \qquad \text{für alle } x, b \in \mathbb{R}^{\mathcal{I}}$$

an, für die Durchführung eines Schritts genügt es also, die Matrix A mit einem Vektor multiplizieren und Linearkombinationen von Vektoren berechnen zu können. Für den Fall des Modellproblems ist der Algorithmus in Abbildung 6.2 dargestellt.

```
procedure model_richardson(b, x, θ, var x′);
for i_x, i_y ∈ {1, ..., N} do begin
  r ← b_i − h^{-2}(4x_{i_x,i_y} − x_{i_x+1,i_y} − x_{i_x−1,i_y} − x_{i_x,i_y+1} − x_{i_x,i_y−1});
  x′_{i_x,i_y} ← x_{i_x,i_y} + θr
end
```

Abbildung 6.2: Pseudocode für einen Schritt des Richardson-Verfahrens für die Matrix des Modellproblems

Randpunkte. Bei dem Algorithmus ist zu beachten, dass wir der Einfachheit halber davon ausgehen, dass die Werte x_{i_x,i_y} für $i_x, i_y \in \{0, \ldots, N+1\}$ vorliegen, dass also auch Randpunkte des Gebiets in den Vektoren enthalten sind, obwohl sie keine Unbekannten unseres Gleichungssystems sind. Dank dieser Konvention ist der Algorithmus kürzer

und die Behandlung inhomogener Randdaten einfacher: Wir wollen in jedem Punkt $i = (i_x, i_y) \in \mathcal{I}$ die Gleichung

$$h^{-2}(4x_{i_x,i_y} - x_{i_x+1,i_y} - x_{i_x-1,i_y} - x_{i_x,i_y+1} - x_{i_x,i_y-1}) = b_{i_x,i_y}$$

erfüllen. Falls nun beispielsweise $(i_x + 1, i_y) \notin \mathcal{I}$ gilt, weil $i_x + 1 = N + 1$ eintritt, ist x_{i_x+1,i_y} keine Unbekannte mehr, sondern durch unsere Randbedingungen gegeben. Also können wir den betreffenden Term einfach auf die andere Seite der Gleichung bringen und erhalten

$$h^{-2}(4x_{i_x,i_y} - x_{i_x-1,i_y} - x_{i_x,i_y+1} - x_{i_x,i_y-1}) = b_{i_x,i_y} + h^{-2}x_{i_x+1,i_y}.$$

Diese Darstellung ist für die Theorie nützlich, weil sie zu einem linearen Gleichungssystem der üblichen Form führt: Unbekannte Koeffizienten links, bekannte rechts. Bei der für das Richardson-Verfahren relevanten Berechnung des Vektors $b - Ax$ dagegen sind beide Darstellung völlig äquivalent und die erste aufgrund der einfachen Programmstruktur vorzuziehen, denn sie erspart uns sonst erforderliche Fallunterscheidungen für die Randpunkte.

Konvergenz. Obwohl die Richardson-Iteration sehr einfach ist (wir berechnen eigentlich nur den Fehler $Ax-b$ der aktuellen Näherungslösung und subtrahieren ein geeignetes Vielfaches von ihr), lässt sich beweisen, dass es für positiv definite Matrizen A konvergiert.

Lemma 6.2 (Konvergenz) *Sei $A \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ positiv definit. Es existieren $\alpha, \beta \in \mathbb{R}_{>0}$ so, dass für jedes $\theta \in \mathbb{R}_{\geq 0}$ die Abschätzung*

$$\|\Phi_{\mathrm{Rich},\theta}(x, b) - \Phi_{\mathrm{Rich},\theta}(y, b)\|_2 \leq L_\theta \|x - y\|_2 \qquad \text{für alle } x, y, b \in \mathbb{R}^{\mathcal{I}}$$

mit der durch

$$L_\theta := \sqrt{1 - 2\theta\alpha + \theta^2\beta} \qquad \text{für alle } \theta \in \mathbb{R}_{\geq 0}$$

definierten Konstanten gilt. Für jedes $\theta \in (0, \theta_{\max})$ mit

$$\theta_{\max} := 2\alpha/\beta^2$$

gilt $L_\theta < 1$, und indem wir $y = A^{-1}b$ einsetzen, folgt, dass die Iterierten gegen die Lösung $A^{-1}b$ konvergieren. Minimal wird L_θ für $\theta_{\mathrm{opt}} = \theta_{\max}/2$, und es gilt

$$L_{\theta_{\mathrm{opt}}} = \sqrt{1 - \alpha^2/\beta^2} \leq 1 - \frac{\alpha^2}{2\beta^2}.$$

Beweis. Die Menge

$$R := \{\langle Ax, x\rangle_2 \ : \ x \in \mathbb{R}^{\mathcal{I}},\ \|x\|_2 = 1\}$$

ist als Bild der kompakten Einheitssphäre unter einer stetigen Abbildung kompakt. Da A positiv definit ist, gilt $R \subseteq \mathbb{R}_{>0}$, also muss R ein Minimum

$$\alpha := \min\{\langle Ax, x\rangle_2 \ : \ x \in \mathbb{R}^{\mathcal{I}},\ \|x\|_2 = 1\} > 0$$

besitzen, für das

$$\alpha\|x\|_2^2 \leq \langle Ax, x\rangle_2 \qquad \text{für alle } x \in \mathbb{R}^{\mathcal{I}}$$

gilt. Wir definieren $\beta := \|A\|_2$.

Seien nun $x, y, b \in \mathbb{R}^{\mathcal{I}}$ gegeben. Dann gilt

$$\begin{aligned}\Phi_{\mathrm{Rich},\theta}(x,b) - \Phi_{\mathrm{Rich},\theta}(y,b) &= x - \theta(Ax - b) - y + \theta(Ay - b)\\ &= (x - y) - \theta A(x - y) = (I - \theta A)(x - y).\end{aligned}$$

Wir setzen $z := x - y$ und erhalten

$$\begin{aligned}\|(I - \theta A)z\|_2^2 &= \|z\|_2^2 - 2\theta\langle Az, z\rangle_2 + \theta^2\|Az\|_2^2\\ &\leq \|z\|_2^2 - 2\theta\alpha\|z\|_2^2 + \theta^2\beta^2\|z\|_2^2\\ &= (1 - 2\theta\alpha + \theta^2\beta^2)\|z\|_2^2 = L_\theta^2\|z\|_2^2\end{aligned}$$

Offenbar gilt

$$L_\epsilon^2 < 1 \iff \theta^2\beta^2 < 2\theta\alpha \iff \theta < 2\alpha/\beta^2 = \theta_{\max}.$$

Ein Blick auf die Ableitungen der Funktion $\theta \mapsto 1 - 2\theta\alpha + \theta^2\beta^2$ zeigt, dass sie für den Wert $\theta = \alpha/\beta^2 = \theta_{\mathrm{opt}}$ ihr Minimum

$$L_{\theta_{\mathrm{opt}}} = \sqrt{1 - \frac{\alpha^2}{\beta^2}} \leq \sqrt{1 - 2\frac{\alpha^2}{2\beta^2} + \frac{\alpha^4}{4\beta^4}} = 1 - \frac{\alpha^2}{2\beta^2}$$

annimmt. ∎

Falls wir den *Dämpfungsparameter* θ klein genug wählen, werden die Iterierten des Richardson-Verfahrens also gegen die Lösung des linearen Gleichungssystems konvergieren. Unsere Abschätzung für die Konvergenzgeschwindigkeit L_θ lässt sich übrigens erheblich verbessern, wenn wir voraussetzen, dass die Matrix A nicht nur positiv definit, sondern auch symmetrisch ist.

Jacobi-Iteration. Die Matrix A^{-1} durch θI zu approximieren dürfte in der Regel zu keinen sehr guten Ergebnissen führen. Deshalb sind wir daran interessiert, die Approximation zu verbessern, ohne den Rechenaufwand wesentlich zu steigern. Eine Möglichkeit besteht darin, die Diagonale $D \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ der Matrix A zu verwenden, die durch

$$d_{ij} = \begin{cases} a_{ii} & \text{falls } i = j,\\ 0 & \text{ansonsten}\end{cases} \qquad \text{für alle } i, j \in \mathcal{I}$$

gegeben ist. Indem wir A^{-1} durch D^{-1} annähern, erhalten wir die *Jacobi-Iteration* mit der Vorschrift

$$\Phi_{\mathrm{Jac}}(x,b) = x - D^{-1}(Ax-b) \qquad \text{für alle } x,b \in \mathbb{R}^{\mathcal{I}}.$$

Auch hier kann es sinnvoll sein, einen Dämpfungsparameter $\theta \in \mathbb{R}_{>0}$ einzuführen, um die Konvergenz sicherzustellen. So erhalten wir die *gedämpfte Jacobi-Iteration*

$$\Phi_{\mathrm{Jac},\theta}(x,b) = x - \theta D^{-1}(Ax-b) \qquad \text{für alle } x,b \in \mathbb{R}^{\mathcal{I}}.$$

Die Jacobi-Iteration folgt einem einfachen Konstruktionsprinzip, das wir erkennen können, indem wir $x' := \Phi_{\mathrm{Jac}}(x,b)$ etwas genauer untersuchen. Für jedes $i \in \mathcal{I}$ gilt nach Definition

$$x'_i = x_i - \frac{1}{a_{ii}} \left(\sum_{j \in \mathcal{I}} a_{ij} x_j - b_i \right),$$

und indem wir den Fall $j = i$ aus der Summe herausziehen, folgt

$$\begin{aligned} x'_i &= x_i - \frac{1}{a_{ii}} \left(a_{ii} x_i + \sum_{\substack{j \in \mathcal{I} \\ j \neq i}} a_{ij} x_j - b_i \right) \\ &= x_i - x_i - \frac{1}{a_{ii}} \left(\sum_{\substack{j \in \mathcal{I} \\ j \neq i}} a_{ij} x_j - b_i \right) = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j \in \mathcal{I} \\ j \neq i}} a_{ij} x_j \right). \end{aligned} \tag{6.6}$$

Wir multiplizieren mit a_{ii} und bringen die Summe auf die linke Seite, um schließlich

$$a_{ii} x'_i + \sum_{\substack{j \in \mathcal{I} \\ j \neq i}} a_{ij} x_j = b_i$$

zu erhalten. Das Jacobi-Verfahren wählt also den Wert x'_i gerade so, dass zumindest die i-te Zeile des Gleichungssystems gelöst wird, das Lösen des Gesamtsystems wird so auf das Lösen eindimensionaler Teilprobleme zurückgeführt.

Für das gedämpfte Verfahren erhalten wir die Gleichung

$$\begin{aligned} \Phi_{\mathrm{Jac},\theta}(x,b) &= x - \theta D^{-1}(Ax-b) = (1-\theta)x + \theta x - \theta D^{-1}(Ax-b) \\ &= (1-\theta)x + \theta(x - D^{-1}(Ax-b)) = (1-\theta)x + \theta \Phi_{\mathrm{Jac}}(x,b), \end{aligned}$$

die Iterierten ergeben sich also als Linearkombinationen (für $\theta \in [0,1]$ sogar als *Konvexkombinationen*) zwischen den Ausgangsvektoren und den Vektoren des ungedämpften Verfahrens.

Das Jacobi-Verfahren zeigt in der Praxis häufig ein wesentlich bessers Konvergenzverhalten als das Richardson-Verfahren. Besonders nützlich ist die Tatsache, dass in vielen

procedure jacobi(b, x, θ, **var** x');
for $k \in \{1, \ldots, n\}$ **do begin**
  $i \leftarrow \iota(k)$;
  $r \leftarrow b_i$;
  for $j \in \mathcal{I} \setminus \{i\}$ **do** $r \leftarrow r - a_{ij}x_j$;
  $x'_i \leftarrow (1-\theta)x_i + \theta r/a_{ii}$
end

Abbildung 6.3: Pseudocode für einen Schritt der gedämpften Jacobi-Iteration

Fällen die Wahl des Dämpfungsparameters sich wesentlich einfacher gestaltet, weil beispielsweise bei einer Skalierung der Matrix A die Matrix D entsprechend mitskaliert wird, so dass der Dämpfungsparameter nicht geändert werden muss.

Die gedämpfte Jacobi-Iteration ist in Abbildung 6.3 allgemein dargestellt und in Abbildung 6.4 für den Fall des Modellproblems.

procedure model_jacobi(b, x, θ, **var** x');
for $i_x, i_y \in \{1, \ldots, N\}$ **do begin**
  $r \leftarrow b_{i_x,i_y} + h^{-2}(x_{i_x+1,i_y} + x_{i_x-1,i_y} + x_{i_x,i_y+1} + x_{i_x,i_y-1})$;
  $x'_{i_x,i_y} \leftarrow (1-\theta)x_{i_x,i_y} + \theta r/(4h^{-2})$
end

Abbildung 6.4: Pseudocode für einen Schritt der gedämpften Jacobi-Iteration für die Matrix des Modellproblems

Gauß-Seidel-Iteration. Die Jacobi-Iteration lässt sich effizient durchführen, weil sich D^{-1} leicht für einen gegebenen Vektor auswerten lässt, wir also leicht Gleichungssysteme der Form

$$Dp = Ax - b$$

lösen können, um $\Phi_{\text{Jac}}(x, b) = x + p$ zu berechnen. Eine weitere Klasse von Matrizen, für die sich Gleichungssysteme relativ leicht lösen lassen, sind die Dreiecksmatrizen, die wir auch schon bei der LR-Zerlegung verwendet haben.

Dreiecksmatrizen sind nur für geordnete Indexmengen definiert, also setzen wir voraus, dass wir die Indexmenge $\mathcal{I}$ durch eine bijektive Abbildung

$$\iota : \{1, \ldots, n\} \to \mathcal{I}$$

mit $n = \#\mathcal{I}$ aufzählen können. Mit ihrer Hilfe können wir Dreiecksmatrizen verallgemeinern: Wir definieren

$$i < j :\iff \iota^{-1}(i) < \iota^{-1}(j),$$

$$i \leq j :\iff \iota^{-1}(i) \leq \iota^{-1}(j) \qquad \text{für alle } i,j \in \mathcal{I}$$

und stellen uns vor, dass ein Eintrag a_{ij} „unterhalb der Diagonalen“ leigt, falls $j < i$ gilt, während er für $i < j$ „oberhalb“ liegt. Wir zerlegen A in die bereits bekannte Diagonalmatrix D sowie ihren *negativen* Unter- und Überdiagonalanteil $E, F \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$, definiert durch

$$e_{ij} = \begin{cases} a_{ij} & \text{falls } j < i, \\ 0 & \text{ansonsten,} \end{cases}$$
$$f_{ij} = \begin{cases} a_{ij} & \text{falls } i < j, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i,j \in \mathcal{I}.$$

Dann gilt

$$A = D - E - F,$$

und $D - E$ ist in unserer verallgemeinerten Sprechweise eine untere Dreiecksmatrix. Die Gauß-Seidel-Iteration besitzt nun die Gestalt

$$\Phi_{\mathrm{GS}}(x,b) = x - (D-E)^{-1}(Ax - b) \qquad \text{für alle } x,b \in \mathbb{R}^{\mathcal{I}}.$$

Diese Schreibweise der Iteration ist für theoretische Untersuchungen brauchbar, für die praktische Umsetzung können wir sie etwas umformen:

$$\begin{aligned} x' := \Phi_{\mathrm{GS}}(x,b) &= x - (D-E)^{-1}(Ax-b) = x - (D-E)^{-1}((D-E-F)x - b) \\ &= x - (D-E)^{-1}(D-E)x - (D-E)^{-1}(-Fx - b) = (D-E)^{-1}(b + Fx). \end{aligned}$$

Wir müssen also den Hilfsvektor $y := b + Fx$ berechnen und dann das Gleichungssystem

$$(D-E)x' = y$$

lösen. Diese Aufgabe lässt sich aufgrund der Dreiecksgestalt der Matrix $D - E$ durch Vorärtseinsetzen lösen: Es muss

$$y_i = ((D-E)x')_i = \sum_{\substack{j\in\mathcal{I} \\ j\leq i}} a_{ij} x'_j \qquad \text{für alle } i \in \mathcal{I}$$

gelten. In einem ersten Schritt können wir $i = \iota(1)$ behandeln, da die Summe dann nur das Diagonalelement betrifft. In einem zweiten Schritt können wir $x'_{\iota(1)}$ in der Summe einsetzen und nach $x'_{\iota(2)}$ auflösen.

Werfen wir nun einen Blick auf den Vektor y, dessen Einträge durch

$$y_i = b_i + \sum_{j\in\mathcal{I}} f_{ij} x_j = b_i - \sum_{\substack{j\in\mathcal{I} \\ i<j}} a_{ij} x_j \qquad \text{für alle } i \in \mathcal{I}$$

gegeben sind. Durch Einsetzen erhalten wir

$$b_i - \sum_{\substack{j\in\mathcal{I}\\ i<j}} a_{ij}x_j = \sum_{\substack{j\in\mathcal{I}\\ j\le i}} a_{ij}x'_j,$$

$$b_i - \sum_{\substack{j\in\mathcal{I}\\ i<j}} a_{ij}x_j - \sum_{\substack{j\in\mathcal{I}\\ j<i}} a_{ij}x'_j = a_{ii}x'_i \qquad \text{für alle } i \in \mathcal{I}. \tag{6.7}$$

Dieses System wollen wir durch Vorwärtseinsetzen auflösen. Man sieht, dass im ersten Schritt die zweite Summe auf der linken Seite leer ist. Im zweiten Schritt enthält sie nur den Summanden für $j = \iota(1)$, für den wir $x'_{\iota(1)}$ im ersten Schritt berechnet haben. Im dritten Schritt treten nur $j \in \{\iota(1), \iota(2)\}$ auf. Entsprechend fehlen diese Einträge in der ersten Summe auf der linken Seite.

Um den Prozess des Vorwärtseinsetzens etwas genauer zu fassen, führen wir Hilfsvektoren $x^{(k)} \in \mathbb{R}^{\mathcal{I}}$ für $k \in \{0, \ldots, n\}$ ein, die durch

$$x_i^{(k)} := \begin{cases} x'_i & \text{falls } \iota^{-1}(i) \le k, \\ x_i & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I},\ k \in \{0, \ldots, n\}$$

definiert sind. Offenbar gelten $x^{(0)} = x$ und $x^{(n)} = x'$, und die Gleichung (6.7) nimmt die Form

$$a_{ii}x'_i = b_i - \sum_{\substack{j\in\mathcal{I}\\ j\neq i}} a_{ij}x_j^{(k-1)} \qquad \text{für alle } k \in \{1, \ldots, n\},\ i = \iota(k) \tag{6.8}$$

an. Da sich die Vektoren $x^{(k-1)}$ und $x^{(k)}$ nur in der Komponente $\iota(k)$ unterscheiden, können wir sie mit Hilfe dieser Formel der Reihe nach ausrechnen und erhalten den in Abbildung 6.5 dargestellten Algorithmus, dessen besondere Eleganz darin liegt, dass wir den ursprünglichen Iterationsvektor x nach und nach mit dem neuen Vektor x' überschreiben können und deshalb keinen zusätzlichen Speicher benötigen.

```
procedure gauss_seidel(b, var x);
for k = 1 to n do begin
  i ← ι(k);
  r = b_i;
  for j ∈ I \ {i} do r ← r − a_ij x_j;
  x_i ← r/a_ii
end
```

Abbildung 6.5: Pseudocode für einen Schritt der Gauß-Seidel-Iteration

In Abbildung 6.6 ist seine Entsprechung für den Fall des Modellproblems zusammengefasst, wobei wir festlegen, dass wir die Gitterpunkte von unten nach oben zeilenweise von links nach rechts durchlaufen. Diese Numerierung wird als *lexikographische Anordnung* bezeichnet.

```
procedure model_gauss_seidel(b, var x);
for i_y = 1 to N do
  for i_x = 1 to N do begin
    r ← b_{i_x,i_y} + h^{-2}(x_{i_x+1,i_y} + x_{i_x-1,i_y} + x_{i_x,i_y+1} + x_{i_x,i_y-1});
    x_{i_x,i_y} ← r/(4h^{-2})
  end
```

Abbildung 6.6: Pseudocode für einen Schritt der Gauß-Seidel-Iteration für die Matrix des Modellproblems

Blockverfahren. Wie wir bereits gesehen haben, können die Jacobi- und die Gauß-Seidel-Iteration als Verfahren interpretiert werden, die für die i-te Zeile des linearen Gleichungssystems eine Gleichung lösen, um x_i' zu berechnen. Statt mit einzelnen Koeffizienten zu rechnen, können wir auch ganze Blöcke von Koeffizienten simultan behandeln: Wir zerlegen die Indexmenge $\mathcal{I}$ in Teilmengen $\mathcal{I}_1, \ldots, \mathcal{I}_m$, die so gewählt sind, dass sich Gleichungssysteme mit den Matrizen

$$A_{kk} := A|_{\mathcal{I}_k \times \mathcal{I}_k} \qquad \text{für alle } k \in \{1, \ldots, m\}$$

einfach auflösen lassen. Im Modellproblem könnte man etwa die Zeilen

$$\mathcal{I}_k := \{1, \ldots, N\} \times \{k\} \qquad \text{für alle } k \in \{1, \ldots, N\}$$

verwenden, denn dann ist jede Matrix A_{kk} tridiagonal (weist also eine obere und untere Bandbreite von eins auf), so dass sich mit Hilfe einer LR-Zerlegung Gleichungssysteme schnell lösen lassen.

Um die die Jacobi-Iteration definierende Gleichung (6.6) elegant für das Blockverfahren schreiben zu können, führen wir die Hilfsvektoren $r_1 \in \mathbb{R}^{\mathcal{I}_1}, \ldots, r_m \in \mathbb{R}^{\mathcal{I}_m}$ ein, die durch

$$r_{k,i} := b_i - \sum_{j \in \mathcal{I} \setminus \mathcal{I}_k} a_{ij} x_j \qquad \text{für alle } k \in \{1, \ldots, m\},\ i \in \mathcal{I}_k$$

definiert sind. An die Stelle der Gleichung (6.6) treten für das Blockverfahren die Gleichungen

$$A_{kk} x'|_{\mathcal{I}_k} = r_k \qquad \text{für alle } k \in \{1, \ldots, m\}.$$

An die Stelle der Division durch das Diagonalelement a_{ii} tritt also die Multiplikation mit der Inversen A_{kk}^{-1}, die dem Lösen eines kleineren linearen Gleichungssystems entspricht.

Für den Fall eines Zeilenblock-Gauß-Seidel-Verfahrens ist der resultierende Algorithmus in Abbildung 6.7 dargestellt: In einem ersten Schritt wird die LR-Zerlegung der Tridiagonalmatrix

$$A_{kk} = \begin{pmatrix} 4h^{-2} & -h^{-2} & & \\ -h^{-2} & \ddots & \ddots & \\ & \ddots & \ddots & -h^{-2} \\ & & -h^{-2} & 4h^{-2} \end{pmatrix}$$

in der Form

$$A_{kk} = \begin{pmatrix} 1 & & & \\ \ell_1 & \ddots & & \\ & \ddots & \ddots & \\ & & \ell_{N-1} & 1 \end{pmatrix} \begin{pmatrix} d_1 & r_1 & & \\ & \ddots & \ddots & \\ & & \ddots & r_{N-1} \\ & & & d_N \end{pmatrix}$$

berechnet. Da A_{kk} für alle $k \in \{1, \ldots, N\}$ dieselbe Matrix ist, genügt es, die LR-Zerlegung einmal auszurechnen und anschließend zu verwenden, um die Gleichungssysteme der einzelnen Zeilen durch Vorwärts- und Rückwärtseinsetzen zu lösen.

```
procedure model_row_gauss_seidel(b, var x);
d_1 ← 4h^{-2};
for i = 1 to N − 1 do begin        { LR-Zerlegung }
  ℓ_i ← −h^{-2}/d_i;   r_i ← −h^{-2};   d_{i+1} ← 4h^{-2} − ℓ_i r_i
end;
for i_y = 1 to N do begin
  for i_x ∈ {1, ..., N} do x̂_{i_x} ← b_{i_x,i_y} + h^{-2}(x_{i_x,i_y+1} + x_{i_x,i_y−1});
  x̂_1 ← x̂_1 + h^{-2} x_{0,i_y};   x̂_N ← x̂_N + h^{-2} x_{N+1,i_y};
  for i = 1 to N − 1 do x̂_{i+1} ← x̂_{i+1} − ℓ_i x̂_i;        { Vorwärtseinsetzen }
  x̂_N ← x̂_N / d_N;        { Rückwärtseinsetzen }
  for i = N − 1 downto 1 do x̂_i ← (x̂_i − r_i x̂_{i+1})/d_i;
  for i_x ∈ {1, ..., N} do x_{i_x,i_y} ← x̂_{i_x}
end
```

Abbildung 6.7: Pseudocode für einen Schritt der Zeilenblock-Gauß-Seidel-Iteration für die Matrix des Modellproblems

6.4 Krylow-Verfahren

Das Richardson-Verfahren bietet den Vorteil, die Matrix ausschließlich in Gestalt der Matrix-Vektor-Multiplikation zu benötigen. Leider konvergiert es in der Regel auch sehr langsam, so dass es in der Praxis nur selten Anwendung findet. Allerdings ist es möglich, wesentlich schnellere Verfahren zu konstruieren, die ebenfalls auf der Matrix-Vektor-Multiplikation aufsetzen.

Selbstadjungiert positiv definit. Zur Motivation gehen wir davon aus, dass $A \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ *positiv definit* ist, dass also

$$\langle Ax, x\rangle_2 \in \mathbb{R}_{>0} \qquad \text{für alle } x \in \mathbb{R}^{\mathcal{I}} \setminus \{0\}$$

gilt, und dass A *selbstadjungiert* ist, dass also

$$\langle Ax, y\rangle_2 = \langle x, Ay\rangle_2 \qquad \text{für alle } x, y \in \mathbb{R}^{\mathcal{I}}$$

erfüllt ist. Diese zweite Eigenschaft ist äquivalent zu

$$a_{ij} = a_{ji} \qquad \text{für alle } i,j \in \mathcal{I},$$

lässt sich also relativ einfach nachprüfen. Die Matrix unseres Modellproblems erfüllt diese Voraussetzungen, die Matrix der dreidimensionalen Potentialgleichung (5.11) ebenfalls.

Minimierungsproblem. Unter diesen Voraussetzungen können wir die Lösung des linearen Gleichungssystems als Lösung eines Minimierungsproblems beschreiben:

Lemma 6.3 (Minimierungsproblem) *Wir definieren*

$$f : \mathbb{R}^{\mathcal{I}} \to \mathbb{R}, \qquad x \mapsto \langle Ax, x\rangle_2 - 2\langle b, x\rangle_2.$$

Dann gilt

$$f(x - \lambda y) = f(x) - 2\lambda\langle Ax - b, y\rangle_2 + \lambda^2\langle Ay, y\rangle_2 \qquad \text{für alle } x, y \in \mathbb{R}^{\mathcal{I}},\ \lambda \in \mathbb{R}. \tag{6.9}$$

Die Funktion f nimmt ihr Minimum für genau ein Argument an, nämlich für $x^* = A^{-1}b$.

Beweis. Seien $x, y \in \mathbb{R}^{\mathcal{I}}$ und $\lambda \in \mathbb{R}$ gegeben. Dann gilt

$$\begin{aligned}
f(x - \lambda y) &= \langle A(x - \lambda y), x - \lambda y\rangle_2 - 2\langle b, x - \lambda y\rangle_2 \\
&= \langle Ax, x\rangle_2 - \langle \lambda Ay, x\rangle_2 - \langle Ax, \lambda y\rangle_2 + \langle \lambda Ay, \lambda y\rangle_2 - 2\langle b, x\rangle_2 + 2\langle b, \lambda y\rangle_2 \\
&= f(x) - \lambda\langle y, Ax\rangle_2 - \lambda\langle Ax, y\rangle_2 + \lambda^2\langle Ay, y\rangle_2 + 2\lambda\langle b, y\rangle_2 \\
&= f(x) - \lambda\langle Ax, y\rangle_2 - \lambda\langle Ax, y\rangle_2 + \lambda^2\langle Ay, y\rangle_2 + 2\lambda\langle b, y\rangle_2 \\
&= f(x) - 2\lambda\langle Ax, y\rangle_2 + 2\lambda\langle b, y\rangle_2 + \lambda^2\langle Ay, y\rangle_2 \\
&= f(x) - 2\lambda\langle Ax - b, y\rangle_2 + \lambda^2\langle Ay, y\rangle_2.
\end{aligned}$$

Indem wir diese Gleichung auf $\lambda = 1$ und $y = x^* - x$ anwenden, erhalten wir

$$f(x) = f(x^* - \lambda y) = f(x^*) - 2\lambda\langle Ax^* - b, y\rangle_2 + \lambda^2\langle Ay, y\rangle_2 = f(x^*) + \langle Ay, y\rangle_2.$$

Da A positiv definit ist, folgt

$$f(x) = f(x^*) + \langle Ay, y\rangle_2 \geq f(x^*),$$

also muss f sein globales Minimum in $f(x^*)$ annehmen. Falls $f(x) = f(x^*)$ gelten sollte, folgt aus der Gleichung bereits $\langle Ay, y\rangle_2 = 0$. Da A positiv definit ist, muss dann $x^* - x = y = 0$ gelten, x^* ist also das einzige Argument, für das f minimal ist. ∎

Statt direkt nach der Lösung x^* zu suchen, können wir also auch versuchen, das Minimum der Funktion f zu finden. Es bietet sich an, dazu ein *Abstiegsverfahren* zu verwenden: Ausgehend von einer Näherungslösung $x^{(m)} \in \mathbb{R}^{\mathcal{I}}$ wählen wir eine *Suchrichtung* $p^{(m)} \in \mathbb{R}^{\mathcal{I}}$ und eine *Schrittweite* $\lambda^{(m)} \in \mathbb{R}$ und setzen

$$x^{(m+1)} := x^{(m)} - \lambda^{(m)} p^{(m)}.$$

Dabei stellen wir sicher, dass zumindest

$$f(x^{(m+1)}) \leq f(x^{(m)})$$

gilt, und hoffen, dass wir $p^{(m)}$ und $\lambda^{(m)}$ so wählen können, dass die linke Seite wesentlich kleiner als die rechte ist, wir also dem Minimum erheblich näher kommen.

Optimalität. Unser Ziel besteht also darin, die Suchrichtung $p^{(m)}$ und die Schrittweite $\lambda^{(m)}$ möglichst geschickt zu wählen. Für die Schrittweite ist diese Aufgabe einfach zu lösen: Wir gehen davon aus, dass $p^{(m)}$ gegeben ist, und wir wollen $\lambda^{(m)}$ so wählen, dass sich die nächste Iterierte $x^{(m+1)} = x^{(m)} + \lambda^{(m)}p^{(m)}$ nicht mehr weiter verbessern lässt, indem wir ein Vielfaches von $p^{(m)}$ hinzuaddieren.

Dazu führen wir die folgende Sprechweise ein: $x \in \mathbb{R}^{\mathcal{I}}$ nennen wir *optimal bezüglich* $p \in \mathbb{R}^{\mathcal{I}}$, falls

$$f(x) \leq f(x - \lambda p) \qquad \text{für alle } \lambda \in \mathbb{R}$$

gilt. Für $p \neq 0$ stellen wir fest (etwa indem wir das Minimum der Gleichung (6.9) durch Differenzieren nach λ bestimmen), dass die rechte Seite dieses Ausdrucks für

$$\lambda = \frac{\langle Ax - b, p\rangle_2}{\langle Ap, p\rangle_2}$$

ihr Minimum

$$f(x - \lambda p) = f(x) - \frac{\langle Ax - b, p\rangle_2^2}{\langle Ap, p\rangle_2}$$

annimmt. Falls x optimal bezüglich p ist, muss

$$f(x) \leq f(x - \lambda p) = f(x) - \frac{\langle Ax - b, p\rangle_2^2}{\langle Ap, p\rangle_2}$$

gelten. Da A positiv definit ist, ist der rechte Term niemals negativ, also muss

$$\langle Ax - b, p\rangle_2 = 0 \tag{6.10}$$

gelten. Der Vektor x ist also genau dann optimal bezüglich der Richtung p, wenn (6.10) erfüllt ist.

Optimale Schrittweite. Wir gehen davon aus, dass eine Suchrichtung $p^{(m)} \in \mathbb{R}^{\mathcal{I}} \setminus \{0\}$ gegeben ist. Wir wollen die Schrittweite $\lambda^{(m)}$ so wählen, dass $x^{(m+1)}$ optimal bezüglich der Richtung $p^{(m)}$ ist, dass also

$$\begin{aligned} 0 &= \langle Ax^{(m+1)} - b, p^{(m)}\rangle_2 = \langle A(x^{(m)} - \lambda^{(m)}p^{(m)}) - b, p^{(m)}\rangle_2 \\ &= \langle Ax^{(m)} - b, p^{(m)}\rangle_2 - \lambda^{(m)}\langle Ap^{(m)}, p^{(m)}\rangle_2 \end{aligned}$$

gilt. Da A positiv definit ist, ist der letzte Faktor echt positiv, also muss die optimale Schrittweite durch

$$\lambda^{(m)} = \frac{\langle Ax^{(m)} - b, p^{(m)}\rangle_2}{\langle Ap^{(m)}, p^{(m)}\rangle_2}$$

gegeben sein. Mit dieser Schrittweite erhalten wir

$$f(x^{(m+1)}) = f(x^{(m)}) - \frac{\langle Ax^{(m)} - b, p^{(m)}\rangle_2^2}{\langle Ap^{(m)}, p^{(m)}\rangle_2}, \tag{6.11}$$

also wird der Funktionswert abnehmen, falls $p^{(m)}$ nicht gerade senkrecht auf dem Vektor $Ax^{(m)} - b$ steht.

Gradientenverfahren. Die Wahl einer geeigneten Suchrichtung erweist sich als wesentlich schwieriger. Ein einfacher Ansatz besteht darin, davon auszugehen, dass wir nur eine „hinreichend kleine“ Schrittweite in Betracht ziehen und deshalb in der Gleichung (6.9) den letzten Term vernachlässigen können. Dann bleibt

$$f(x^{(m)} - \lambda^{(m)} p^{(m)}) \approx f(x^{(m)}) - 2\lambda \langle Ax^{(m)} - b, p^{(m)} \rangle_2,$$

und nach der Cauchy-Schwarz-Ungleichung nimmt der letzte Faktor sein Maximum gerade an, wenn die Suchrichtung $p^{(m)}$ linear abhängig von $Ax^{(m)} - b$ ist. Da die Skalierung der Suchrichtung keine Rolle spielt, können wir einfach

$$p^{(m)} = Ax^{(m)} - b$$

verwenden. Die optimale Suchrichtung ist in diesem Fall durch

$$\lambda^{(m)} = \frac{\langle Ax^{(m)} - b, p^{(m)} \rangle_2}{\langle Ap^{(m)}, p^{(m)} \rangle_2} = \frac{\|Ax^{(m)} - b\|_2^2}{\langle Ap^{(m)}, p^{(m)} \rangle_2}$$

gegeben. Die offensichtlich wichtigen Vektoren

$$d^{(m)} := Ax^{(m)} - b \qquad \text{für alle } m \in \mathbb{N}_0$$

bezeichnet man als die *Defekte* zu den Näherungslösungen, und da sie die Gleichung

$$d^{(m)} = Ax^{(m)} - b = Ax^{(m)} - Ax^* = A(x^{(m)} - x^*) = Ae^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

erfüllen, können sie verwendet werden, um den Iterationsfehler praktisch abzuschätzen.

Die partiellen Ableitungen der Funktion f sind durch

$$\partial_i f(x) = 2 \sum_{j \in \mathcal{I}} a_{ij} x_j - 2b_i \qquad \text{für alle } x \in \mathbb{R}^{\mathcal{I}},\ i \in \mathcal{I}$$

gegeben, so dass wir für den Gradienten in den Iterierten die Gleichung

$$\nabla f(x^{(m)}) = 2Ax^{(m)} - 2b = 2d^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

erhalten, der Defektvektor ist also gerade der halbe Gradient der Funktion f. Dieser Eigenschaft verdankt das durch

$$d^{(m)} := Ax^{(m)} - b, \qquad \lambda^{(m)} := \frac{\|d^{(m)}\|_2^2}{\langle Ad^{(m)}, d^{(m)} \rangle_2}, \qquad x^{(m+1)} := x^{(m)} - \lambda^{(m)} d^{(m)} \tag{6.12}$$

beschriebene Iterationsverfahren den Namen *Gradientenverfahren.*

Auf den ersten Blick erfordert ein Schritt des Gradientenverfahrens zwei Multiplikationen mit der Matrix A, einmal für die Berechnung des Defekts $d^{(m)}$ und einmal für die Bestimmung der Schrittweite λ. Indem wir die Hilfsgröße

$$a^{(m)} := Ad^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

einführen, können wir den Defekt durch

$$\begin{aligned} d^{(m+1)} &= Ax^{(m+1)} - b = A(x^{(m)} - \lambda d^{(m)}) - b \\ &= d^{(m)} - \lambda A d^{(m)} = d^{(m)} - \lambda a^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0 \end{aligned}$$

ohne eine weitere Matrix-Vektor-Multiplikation berechnen. Der resultierende Algorithmus ist in Abbildung 6.8 dargestellt.

```
procedure gradient_init(b, var x, d);
d ← Ax − b
end;
procedure gradient_step(b, var x, d);
a ← Ad;
λ ← ⟨d, d, ⟩2/⟨a, d⟩2;
x ← x − λd;
d ← x − λa
end
```

Abbildung 6.8: Pseudocode für Initialisierung und einen Schritt des Gradientenverfahrens

Vergleich mit Richardson-Iteration. Ein Vergleich mit der Richardson-Iteration zeigt, dass sich beide lediglich durch die Wahl des Skalierungsfaktors vor dem Defektvektor unterscheiden: Für die Richardson-Iteration ist er fest gegeben, im Gradientenverfahren wird in jedem Schritt der bestmögliche Faktor gewählt. Deshalb muss die Konvergenz des Gradientenverfahrens mindestens so schnell wie die der Richardson-Iteration für deren optimalen Dämpfungsparameter sein. Leider zeigt die Praxis, dass sie oft auch nicht viel schneller ist. Trotzdem ist es von Vorteil, die Wahl des Dämpfungsparameters automatisieren zu können.

Ursache der langsamen Konvergenz. Unsere Wahl der Schrittweite führt dazu, dass

$$\begin{aligned}\langle p^{(m)}, p^{(m+1)}\rangle_2 &= \langle d^{(m)}, d^{(m+1)}\rangle_2 = \langle d^{(m)}, A(x^{(m)} - \lambda^{(m)} d^{(m)}) - b\rangle_2 \\ &= \langle d^{(m)}, d^{(m)}\rangle_2 - \lambda^{(m)}\langle d^{(m)}, Ad^{(m)}\rangle_2 \\ &= \langle d^{(m)}, d^{(m)}\rangle_2 - \langle d^{(m)}, d^{(m)}\rangle_2 = 0 \qquad \text{für alle } m \in \mathbb{N}_0\end{aligned}$$

gilt, dass aufeinander folgende Suchrichtungen also senkrecht aufeinander stehen. Falls $\mathbb{R}^{\mathcal{I}}$ beispielsweise ein zweidimensionaler Vektorraum ist, bedeutet das, dass in jedem zweiten Schritt *dieselbe* Suchrichtung verwendet wird. Das legt nahe, dass die durch die geschickte Wahl von $\lambda^{(m)}$ erzielte Optimalität in der Suchrichtung $p^{(m)}$ schon im unmittelbar folgenden Schritt verloren gehen kann.

Erhaltung der Optimalität. Natürlich liegt es nicht in unserem Interesse, die in einem Schritt erreichte Optimalität sofort wieder zu verlieren. An der Schrittweite können wir dabei wenig ändern, sie wird ja schon optimal gewählt, also bleibt uns nur die Suchrichtung als Ansatzpunkt. Wir nehmen an, dass $x^{(m)}$ optimal beüglich *aller* bisherigen Suchrichtungen $p^{(0)}, \ldots, p^{(m-1)}$ ist, und streben an, diese Eigenschaft auch für $x^{(m+1)}$ zu erhalten. Nach (6.10) muss also

$$0 = \langle Ax^{(m+1)} - b, p^{(\ell)}\rangle_2 = \langle A(x^{(m)} - \lambda^{(m)} p^{(m)}) - b, p^{(\ell)}\rangle_2$$

$$= \langle Ax^{(m)} - b, p^{(\ell)} \rangle_2 - \lambda^{(m)} \langle Ap^{(m)}, p^{(\ell)} \rangle_2 \qquad \text{für alle } \ell \in \{0, \dots, m-1\}$$

gelten. Unserer Voraussetzung zufolge ist $x^{(m)}$ bereits optimal bezüglich $p^{(0)}, \dots, p^{(m-1)}$, und da $\lambda^{(m)} \neq 0$ gelten sollte, bleibt lediglich die Bedingung

$$\langle Ap^{(m)}, p^{(\ell)} \rangle_2 = 0 \qquad \text{für alle } \ell \in \{0, \dots, m-1\} \tag{6.13}$$

übrig. Wir müssen also $p^{(m)}$ so wählen, dass $Ap^{(m)}$ senkrecht auf allen vorangehenden Suchrichtungen steht. Falls uns das gelingt, wird $x^{(m+1)}$ bezüglich $p^{(0)}, \dots, p^{(m-1)}$ optimal sein, und infolge der optimalen Wahl der Schrittweite auch bezüglich $p^{(m)}$, so dass wir induktiv fortfahren können.

Orthogonalisierung. Unser Ziel lässt sich relativ einfach mit Hilfe der *Gram-Schmidt-Orthogonalisierung* erreichen: Ausgehend von $d^{(m)}$ konstruieren wir $p^{(m)}$, indem wir die Bedingung (6.13) sicherstellen:

$$p^{(m)} := d^{(m)} - \sum_{\ell=0}^{m-1} \frac{\langle Ad^{(m)}, p^{(\ell)} \rangle_2}{\langle Ap^{(\ell)}, p^{(\ell)} \rangle_2} p^{(\ell)}. \tag{6.14}$$

Diese Formel lässt sich anwenden, solange alle Richtungen $p^{(0)}, \dots, p^{(m-1)}$ ungleich null sind, denn da A positiv definit ist, sind die Nenner der Brüche es dann auch. Der Fall $p^{(m)} = 0$ kann nur dann auftreten, wenn $d^{(m)}$ im Aufspann der Richtungen $p^{(0)}, \dots, p^{(m-1)}$ liegt. In dieser Situation finden wir also $\alpha_0, \dots, \alpha_{m-1} \in \mathbb{R}$ mit

$$d^{(m)} = \sum_{\ell=0}^{m-1} \alpha_\ell p^{(\ell)}.$$

Da $x^{(m)}$ nach Konstruktion optimal bezüglich der Richtungen $p^{(0)}, \dots, p^{(m-1)}$ ist, gilt

$$\|d^{(m)}\|_2^2 = \langle d^{(m)}, d^{(m)} \rangle_2 = \sum_{\ell=0}^{m-1} \alpha_\ell \langle Ax^{(m)} - b, p^{(\ell)} \rangle_2 = 0,$$

somit ist $x^{(m)}$ bereits die Lösung. Wir finden also genau dann keine weitere Suchrichtung mehr, wenn wir keine weitere Suchrichtung brauchen, weil wir die exakte Lösung schon gefunden haben.

Die Formel (6.14) ist zwar für die Theorie hilfreich, für die praktische Implementierung allerdings unattraktiv, da sie es erforderlich macht, im m-ten Schritt immerhin m Skalarprodukte zu berechnen. Glücklicherweise können wir sie erheblich vereinfachen, indem wir die besondere Struktur der Richtungen $p^{(0)}, \dots, p^{(m-1)}$ ausnutzen.

Krylow-Raum. Ausgehend von $p^{(0)} = d^{(0)}$ können wir mit einer einfachen Induktion zeigen, dass

$$\operatorname{span}\{p^{(0)}, \dots, p^{(m)}\} \subseteq \operatorname{span}\{d^{(0)}, \dots, d^{(m)}\} \qquad \text{für alle } m \in \mathbb{N}_0$$

gilt. Mit

$$d^{(m+1)} = Ax^{(m+1)} - b = A(x^{(m)} - \lambda^{(m)}p^{(m)}) - b = d^{(m)} - \lambda^{(m)}Ap^{(m)} \quad \text{für alle } m \in \mathbb{N}_0$$

und einer weiteren Induktion folgt daraus

$$\operatorname{span}\{d^{(0)}, \dots, d^{(m)}\} \subseteq \operatorname{span}\{d^{(0)}, Ad^{(0)}, \dots, A^m d^{(0)}\} \qquad \text{für alle } m \in \mathbb{N}_0.$$

Wir setzen voraus, dass alle Vektoren $p^{(0)}, \dots, p^{(m)}$ ungleich null sind (ansonsten hätten wir die exakte Lösung bereits berechnet). Da sie die Gleichung (6.13) erfüllen, können sie nicht linear abhängig sein, also muss

$$\dim \operatorname{span}\{p^{(0)}, \dots, p^{(m)}\} = m + 1$$

gelten, und aus Dimensionsgründen auch

$$\operatorname{span}\{p^{(0)}, \dots, p^{(m)}\} = \operatorname{span}\{d^{(0)}, \dots, d^{(m)}\} = \operatorname{span}\{d^{(0)}, Ad^{(0)}, \dots, A^m d^{(0)}\}.$$

Diesen Raum nennen wir den *Krylow-Raum m-ter Ordnung* zu der Matrix A und dem Vektor $d^{(0)}$, wir bezeichnen ihn mit $\mathcal{K}(d^{(0)}, m)$.

Vereinfachte Orthogonalisierung. Nun können wir die Formel (6.14) vereinfachen: Wir wählen ein $\ell \in \{0, \dots, m-2\}$. Da A selbstadjungiert ist, erhalten wir

$$\langle Ad^{(m)}, p^{(\ell)} \rangle_2 = \langle d^{(m)}, Ap^{(\ell)} \rangle_2.$$

Aus dem zuvor Gesagten folgt

$$p^{(\ell)} \in \mathcal{K}(d^{(0)}, \ell), \qquad Ap^{(\ell)} \in \mathcal{K}(d^{(0)}, \ell + 1) \subseteq \mathcal{K}(d^{(0)}, m - 1).$$

Nach Konstruktion ist $x^{(m)}$ optimal bezüglich $p^{(0)}, \dots, p^{(m-1)}$, also gilt

$$\langle d^{(m)}, q \rangle_2 = 0 \qquad \text{für alle } q \in \operatorname{span}\{p^{(0)}, \dots, p^{(m-1)}\} = \mathcal{K}(d^{(0)}, m-1)$$

und damit auch

$$\langle d^{(m)}, Ap^{(\ell)} \rangle_2 = 0.$$

Demzufolge fallen in (6.14) die Summanden für $\ell \in \{0, \dots, m-2\}$ weg, so dass nur

$$p^{(m)} = d^{(m)} - \frac{\langle Ad^{(m)}, p^{(m-1)} \rangle_2}{\langle Ap^{(m-1)}, p^{(m-1)} \rangle_2}$$

übrig bleibt. Mit Hilfe dieser Formel lässt sich die neue Suchrichtung sehr effizient bestimmen.

Verfahren der konjugierten Gradienten. Vektoren mit der Eigenschaft (6.13) nennt man *konjugiert*, das resultierende Verfahren

$$d^{(m)} = Ax^{(m)} - b,$$
$$\mu^{(m)} := \frac{\langle Ad^{(m)}, p^{(m-1)}\rangle_2}{\langle Ap^{(m-1)}, p^{(m-1)}\rangle_2}, \qquad p^{(m)} = d^{(m)} - \mu^{(m)} p^{(m)},$$
$$\lambda^{(m)} := \frac{\langle d^{(m)}, p^{(m)}\rangle_2}{\langle Ap^{(m)}, p^{(m)}\rangle_2}, \qquad x^{(m+1)} := x^{(m)} - \lambda^{(m)} p^{(m)}$$

wird deshalb als das *Verfahren der konjugierten Gradienten* (oder kurz cg-Verfahren für engl. *conjugate gradients*) bezeichnet.

procedure cg_init(b, **var** x, d, p);
$d \leftarrow Ax - b$;
$p \leftarrow d$
end;
procedure cg_step(b, **var** x, d, p);
$a \leftarrow Ap$;
$\alpha \leftarrow \langle a, p\rangle_2$;
$\lambda \leftarrow \langle d, p\rangle_2 / \alpha$;
$x \leftarrow x - \lambda p$;
$d \leftarrow d - \lambda a$;
$\mu \leftarrow \langle d, a\rangle_2 / \alpha$;
$p \leftarrow d - \mu p$
end

Abbildung 6.9: Pseudocode für Initialisierung und einen Schritt des cg-Verfahrens

Auch bei diesem Verfahren können wir mit einer einzigen Matrix-Vektor-Multiplikation pro Schritt auskommen, wenn wir die Hilfsvektoren

$$a^{(m)} := Ap^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

einführen und feststellen, dass die Gleichungen

$$\mu^{(m)} = \frac{\langle Ad^{(m)}, p^{(m-1)}\rangle_2}{\langle Ap^{(m-1)}, p^{(m-1)}\rangle_2} = \frac{\langle d^{(m)}, Ap^{(m-1)}\rangle_2}{\langle Ap^{(m-1)}, p^{(m-1)}\rangle_2} = \frac{\langle d^{(m)}, a^{(m-1)}\rangle_2}{\langle a^{(m-1)}, p^{(m-1)}\rangle_2},$$
$$\lambda^{(m)} = \frac{\langle d^{(m)}, p^{(m)}\rangle_2}{\langle a^{(m)}, p^{(m)}\rangle_2}, \qquad d^{(m+1)} = A(x^{(m)} - \lambda^{(m)} p^{(m)}) - b = d^{(m)} - \lambda^{(m)} a^{(m)}$$

verwendet werden können, um weitere Matrix-Vektor-Multiplikationen einzusparen. Der resultierende Algorithmus ist in Abbildung 6.9 zusammengefasst. Wie im Fall des Gradientenverfahrens besteht er aus zwei Teilen: Der Initialisierung, bei der der erste Defekt und die erste Suchrichtung bestimmt werden, und dem eigentlichen Iterationsschritt, bei dem beide und die Lösung aktualisiert werden.

6.5 Mehrgitterverfahren

Alle bisher diskutierten Verfahren nutzen nicht aus, dass die linearen Gleichungssysteme, die sie lösen, aus der Diskretisierung einer partiellen Differentialgleichung entstanden sind. Indem wir diese Eigenschaft ausnutzen, können wir Iterationsverfahren konstruieren, die die *optimale Komplexitätsordnung* erreichen, die also ein lineares Gleichungssystem mit n Unbekannten in $\mathcal{O}(n)$ Rechenoperationen lösen.

Glättung des Fehlers. Bei den einfachen linearen Iterationsverfahren wie etwas dem Jacobi-Verfahren gehen bei der Berechnung der Komponente $x_i^{(m+1)}$ der nächsten Ierierten lediglich diejenigen Werte $x_j^{(m)}$ ein, für die $a_{ij} \neq 0$ gilt. Im Fall unserer Finite-Differenzen-Verfahren sind das gerade die unmittelbaren Nachbarpunkte im Gitter.

Es lässt sich nachrechnen, dass deshalb das Jacobi-Verfahren Anteile des Fehlers $e^{(m)}$, die sich schnell von Gitterpunkt zu Gitterpunkt verändern, sehr gut reduziert, während sich langsam verändernde Anteile nur geringfügig beeinflusst werden. Aus dieser Eigenschaft folgt die langsame Konvergenz des Verfahrens.

Wenn wir uns die Entwicklung des Fehlers während der Durchführung des Jacobi- oder Gauß-Seidel-Verfahrens ansehen, stellen wir fest, dass er *geglättet* wird: Hochfrequente Anteile verschwinden relativ schnell, niedrigfrequente werden kaum reduziert.

Gröberes Gitter. An diesem Punkt können wir ausnutzen, dass das lineare Gleichungssystem aus einer Diskretisierung mit einer gewissen Schrittweite h entstanden ist: Ein „hinreichend glatter“ Fehler sollte sich auch mit einer gröberen Schrittweite noch gut annähern lassen. Also diskretisieren wir die Differentialgleichung mit einer Schrittweite $\tilde{h} = 2h$ und erhalten eine Matrix $\widetilde{A} \in \mathbb{R}^{\widetilde{\mathcal{I}} \times \widetilde{\mathcal{I}}}$. Wir wollen eine Approximation $\tilde{e}$ des Fehlers $e^{(m)} = x^{(m)} - x^*$ auf dem gröberen Gitter berechnen, unser Ziel ist also die Berechnung eines Vektors $\tilde{e}^{(m)} \in \mathbb{R}^{\widetilde{\mathcal{I}}}$, der

$$\tilde{e}^{(m)} \approx e^{(m)}$$

in einem geeigneten Sinn erfüllt. Wir können $e^{(m)}$ nicht berechnen, den Defekt $Ae^{(m)} = Ax^{(m)} - b$ dagegen schon, so dass wir

$$\widetilde{A}\tilde{e}^{(m)} \approx Ae^{(m)} = Ax^{(m)} - b$$

erhalten. Da das gröbere Gitter wesentlich weniger Gitterpunkte als das feinere aufweist, lässt sich ein derartiges Gleichungssystem wesentlich einfacher auflösen.

Prolongation und Restriktion. Natürlich passen bei dieser „Gleichung“ die Indexmengen der Vektoren auf beiden Seiten nicht zusammen: Die linke Seite ist auf dem gröberen Gitter formuliert, die rechte auf dem ursprünglichen.

Eine Verbindung zwischen Vektoren auf beiden Gittern stellen wir mit Hilfe zweier Matrizen her: Einer *Prolongation*

$$p : \mathbb{R}^{\widetilde{\mathcal{I}}} \to \mathbb{R}^{\mathcal{I}},$$

die einem Vektor auf dem groben Gitter eine Funktion auf dem feinen Gitter zuordnet, und einer *Restriktion*

$$r : \mathbb{R}^{\mathcal{I}} \to \mathbb{R}^{\widetilde{\mathcal{I}}},$$

die aus einem Vektor auf dem feinen Gitter einen auf dem groben konstruiert.

Grobgitterkorrektur. Formal sauber ergibt sich dann unsere Näherung $\tilde{e}^{(m)}$ des Fehlers als Lösung des Gleichungssystems

$$\widetilde{A}\tilde{e}^{(m)} = r(Ax^{(m)} - b),$$

und eine Verbesserung der aktuellen Iterierten $x^{(m)}$ erreichen wir, indem wir die Näherung des Fehlers von ihr abziehen, also

$$x^{(m+1)} = x^{(m)} - p\tilde{e}^{(m)}$$

setzen. Die gesamte Berechnung können wir in der Formel

$$x^{(m+1)} = x^{(m)} - p\widetilde{A}^{-1}r(Ax^{(m)} - b)$$

zusammenfassen, die offenbar ein neues lineares Iterationsverfahren beschreibt, bei dem die Matrix

$$N := p\widetilde{A}^{-1}r$$

als Approximation der Inversen der Matrix A verwendet wird. Dieses Verfahren trägt den Namen *Grobgitterkorrektur*, und es ist im Allgemeinen zwar sehr gut darin, glatte Anteile des Fehlers zu reduzieren, versagt jedoch bei oszillierenden Anteilen.

Glättungsverfahren. Zu einem konvergenten Verfahren gelangen wir deshalb erst, indem wir die Grobgitterkorrektur im Wechsel mit klassischen Verfahren wie der Jacobi- oder Gauß-Seidel-Iteration einsetzen, die die oszillierenden Anteile des Fehlers reduzieren. Aufgrund dieser Eigenschaft werden diese Verfahren in diesem Kontext als *Glättungsverfahren* bezeichnet, wir schreiben die entsprechende Iterationsfunktion als Φ_{Gl}.

Gitterhierarchie. Auch wenn das gröbere Gitter wesentlich weniger Freiheitsgrade als das feinere enthalten dürfte, könnten es immer noch zuviele für ein direktes Lösungsverfahren sein. Deswegen bietet es sich an, das Gleichungssystem auf dem gröberen Gitter zu lösen, indem man ein noch gröberes Gitter einführt und rekursiv vorgeht. Wir brauchen also eine Familie $(A_\ell)_{\ell=0}^{L}$ von Matrizen

$$A_\ell \in \mathbb{R}^{\mathcal{I}_\ell \times \mathcal{I}_\ell} \qquad \text{für alle } \ell \in \{0, \dots, L\}$$

mit passenden Prolongationen und Restriktionen

$$p_\ell \in \mathbb{R}^{\mathcal{I}_\ell \times \mathcal{I}_{\ell-1}}, \qquad r_\ell \in \mathbb{R}^{\mathcal{I}_{\ell-1} \times \mathcal{I}_\ell} \qquad \text{für alle } \ell \in \{1, \dots, L\}.$$

Die höchste Stufe $\ell = L$ entspricht dann dem ursprünglichen feinen Gitter, die niedrigste $\ell = 0$ enthält hoffentlich nur noch sehr wenige Freiheitsgrade, so dass wir Gleichungssysteme auf diese Stufe einfach lösen können.

```
procedure mehrgitter(ℓ, var b, x);
if ℓ = 0 then
  Löse A_0 x_0 = b_0
else begin
  for k = 1 to ν do
    x_ℓ ← Φ_GI(x_ℓ, b_ℓ);
  d_ℓ ← A_ℓ x_ℓ − b_ℓ;
  b_{ℓ−1} ← r_ℓ d_ℓ;
  x_{ℓ−1} ← 0;
  for k = 1 to γ do
    mehrgitter(ℓ − 1, b, x);
  x_ℓ ← x_ℓ − p_ℓ x_{ℓ−1};
  for k = 1 to ν do
    x_ℓ ← Φ_GI(x_ℓ, b_ℓ)
end
```

Abbildung 6.10: Pseudocode des Mehrgitterverfahrens

Mehrgitterverfahren. Um ein Gleichungssystem

$$A_\ell x_\ell = b_\ell$$

auf der Stufe $\ell \in \{1, \ldots, L\}$ zu lösen, führen wir zunächst einige Schritte des Glättungsverfahrens durch, um oszillierende Anteile des Fehlers zu reduzieren. Anschließend dürfen wir annehmen, dass der Fehler glatt genug ist, um auf dem gröberen Gitter approximiert zu werden, also berechnen wir den Defekt

$$d_\ell = Ax_\ell - b_\ell$$

und übertragen ihn mit Hilfe der Restriktion auf das nächstgröbere Gitter

$$b_{\ell-1} = r_\ell d_\ell.$$

Das System

$$A_{\ell-1} x_{\ell-1} = b_{\ell-1}$$

lösen wir für $\ell = 1$ direkt, anderenfalls rekursiv. Damit steht uns mit $x_{\ell-1}$ eine Näherung des Fehlers zur Verfügung, die wir mit Hilfe der Prolongation auf das feinere Gitter übertragen und von der aktuellen Näherung abziehen, um

$$x_\ell \leftarrow x_\ell - p_\ell x_{\ell-1}$$

zu erhalten. Da durch die Prolongation eventuell wieder oszillierende Anteile entstanden sein könnten, schließen sich häufig noch einige Glättungsschritte an, um x_ℓ zu verbessern.

Das resultierende lineare Iterationsverfahren trägt den Namen *Mehrgitterverfahren* und ist in Abbildung 6.10 zusammengefasst.

6.6 Verfahren für Sattelpunktprobleme

Die Simulation der Grundwasserströmung führt zu der Gleichung (5.19), die wir kompakt in der Form

$$\begin{pmatrix} A & B \\ B^* & \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{6.15}$$

mit den Matrizen

$$A = \begin{pmatrix} A_x & & \\ & A_y & \\ & & A_Z \end{pmatrix} \in \mathbb{R}^{\mathcal{I}_1 \times \mathcal{I}_1}, \qquad B = \begin{pmatrix} B_x \\ B_y \\ B_z \end{pmatrix} \in \mathbb{R}^{\mathcal{I}_1 \times \mathcal{I}_2}$$

und geeigneten Indexmengen $\mathcal{I}_1, \mathcal{I}_2$ schreiben können. Falls die Matrix A in (6.15) positiv definit und selbstadjungiert ist, bezeichnen wir (6.15) als ein *Sattelpunktproblem*. In unserem Fall ist diese Voraussetzung erfüllt, da A eine Diagonalmatrix mit positiven Diagonaleinträgen ist.

Block-Elimination. Da die Matrix A positiv definit ist, ist sie auch invertierbar, also können wir die erste Zeile der Gleichung (6.15) mit B^*A^{-1} multiplizieren und von der zweiten Zeile subtrahieren, um

$$\begin{pmatrix} A & B \\ & -B^*A^{-1}B \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 - B^*A^{-1}b_1 \end{pmatrix}$$

zu erhalten. Anschaulich entspricht dieser Schritt einer Gauß-Elimination mit Matrizen anstelle von Koeffizienten. Indem wir die zweite Zeile mit -1 multiplizieren, folgt

$$\begin{pmatrix} A & B \\ & B^*A^{-1}B \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ B^*A^{-1}b_1 - b_2 \end{pmatrix}. \tag{6.16}$$

Die erste Diagonalmatrix ist nach Definition positiv definit. Die zweite Diagonalmatrix $S := B^*A^{-1}B$ bezeichnet man als das *Schur-Komplement*. Wir können festhalten, dass S selbstadjungiert ist.

Schur-Komplement-System. Damit wir das System (6.16) durch einfaches Block-Rückwärtseinsetzen lösen können, müssen wir die Lösbarkeit des Systems

$$Sx_2 = B^*A^{-1}b_1 - b_2 \tag{6.17}$$

untersuchen. Weil A positiv definit ist, gilt

$$\langle A^{-1}x, x, \rangle_2 = \langle A^{-1}AA^{-1}x, x\rangle_2 = \langle A(A^{-1}x), A^{-1}x\rangle_2 > 0 \qquad \text{für alle } x \in \mathbb{R}^{\mathcal{I}_1} \setminus \{0\},$$

also muss auch A^{-1} positiv definit sein. Daraus folgt

$$\langle B^*A^{-1}Bx, x\rangle_2 = \langle A^{-1}Bx, Bx\rangle_2 \geq 0 \qquad \text{für alle } x \in \mathbb{R}^{\mathcal{I}_2},$$

also ist das *Schur-Komplement* $S := B^* A^{-1} B$ positiv semidefinit und der Kern dieser Matrix ist gerade der Kern der Matrix B.

Für die Lösbarkeit des Gleichungssystems (6.17) ist eher das Bild der Matrix S von Bedeutung, das wir als nächstes untersuchen. Es gilt

$$\langle B^* x, y \rangle_2 = \langle x, By \rangle_2 = \langle x, 0 \rangle_2 = 0 \qquad \text{für alle } x \in \mathbb{R}^{\mathcal{I}_1},\ y \in \operatorname{Kern} B, \tag{6.18}$$

also steht das Bild der Matrix B^* senkrecht auf dem Kern der Matrix B, so dass wir

$$\dim \operatorname{Bild} B^* + \dim \operatorname{Kern} B \leq n := \#\mathcal{I}_2 \tag{6.19}$$

erhalten. Nach Dimensionssatz gilt

$$\dim \operatorname{Bild} S + \dim \operatorname{Kern} S = n,$$

und da wir bereits die Gleichungen

$$\operatorname{Bild} S \subseteq \operatorname{Bild} B^*, \qquad \operatorname{Kern} S = \operatorname{Kern} B$$

kennen, folgt

$$\dim \operatorname{Bild} B^* + \dim \operatorname{Kern} B \geq \dim \operatorname{Bild} S + \dim \operatorname{Kern} S = n.$$

Mit (6.19) erhalten wir so

$$\dim \operatorname{Bild} B^* + \dim \operatorname{Kern} B = n$$

und damit auch

$$\dim \operatorname{Bild} B^* = n - \dim \operatorname{Kern} B = n - \dim \operatorname{Kern} S = \dim \operatorname{Bild} S.$$

Aus $\operatorname{Bild} S \subseteq \operatorname{Bild} B^*$ folgt damit

$$\operatorname{Bild} S = \operatorname{Bild} B^*,$$

das Bild des Schur-Komplements ist also gerade das Bild der Matrix B^*. Damit (6.17) lösbar ist, muss also die rechte Seite im Bild der Matrix B^* liegen. Das ist offenbar genau dann der Fall, wenn b_2 in diesem Bildraum liegt, und das ist wegen (6.18) wiederum genau dann der Fall, wenn b_2 senkrecht auf dem Kern der Matrix B steht.

Für das die Grundwasserströmung beschreibende System (5.19) entspricht B dem Gradienten, der nur für konstante Funktionen verschwindet, also muss b_2 senkrecht auf dem konstanten Vektor stehen: Die Summe über alle Komponenten muss gleich null sein, damit eine Lösung existiert.

cg-Verfahren. Das cg-Verfahren (siehe Abbildung 6.9) lässt sich auch auf selbstadjungierte positiv *semi*definite Matrizen anwenden, wir können also das System (6.18) mit seiner Hilfe angehen.

Die Berechnung des Defekts im m-ten Schritt nimmt dann die Form

$$d_2^{(m)} = Sx_2^{(m)} - (B^*A^{-1}b_1 - b_2) = B^*A^{-1}(Bx_2^{(m)} - b_1) + b_2$$

an, der Vektor $a_2^{(m)}$ berechnet sich durch

$$a_2^{(m)} = Sp^{(m)} = B^*A^{-1}Bp_2^{(m)}.$$

Die Koeffizienten $\alpha^{(m)}$ und $\lambda^{(m)}$ lassen sich wie zuvor durch

$$\alpha^{(m)} = \langle a_2^{(m)}, p_2^{(m)} \rangle_2, \qquad \lambda^{(m)} = \langle d_2^{(m)}, p_2^{(m)} \rangle_2 / \alpha^{(m)}$$

berechnen, die neuen Iterierten ergeben sich per

$$x_2^{(m+1)} = x_2^{(m)} - \lambda^{(m)} p_2^{(m)}, \qquad d_2^{(m+1)} = d_2^{(m)} - \lambda^{(m)} a_2^{(m)}, \tag{6.20}$$

und die nächste Suchrichtung können wir durch

$$\mu^{(m)} = \langle d_2^{(m+1)}, a_2^{(m)} \rangle_2 / \alpha^{(m)}, \qquad p^{(m+1)} = d^{(m+1)} - \mu^{(m)} p^{(m)}$$

bestimmen.

Uzawa-Verfahren. Wir sind nicht nur an x_2, sondern auch an x_1 interessiert. Ein naiver Zugang bestünde darin, x_2 hinreichend genau zu berechnen und dann in (6.16) den Vektor x_1 durch Rückwärtseinsetzen zu bestimmen, also

$$Ax_1 = b_1 - Bx_2$$

zu lösen. Die Idee des *Uzawa-Verfahrens* besteht darin, die Berechnung des Vektors x_1 simultan zu der Berechnung des Vektors x_2 durchzuführen: Wir definieren

$$x_1^{(m)} = A^{-1}(b_1 - Bx_2^{(m)}) \qquad \text{für alle } m \in \mathbb{N}_0$$

und stellen fest, dass dann

$$d_2^{(m)} = B^*A^{-1}(Bx_2^{(m)} - b_1) + b_2 = b_2 - B^*x_1^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

gilt, zumindest im ersten Schritt wird $x_1^{(0)}$ also „nebenbei“ ausgerechnet.

Durch Einsetzen von (6.20) in die Gleichung für $x_1^{(m+1)}$ erhalten wir

$$\begin{aligned} x_1^{(m+1)} &= A^{-1}(b_1 - Bx_2^{(m+1)}) = A^{-1}(b_1 - B(x_2^{(m)} - \lambda^{(m)} p_2^{(m)})) \\ &= x_1^{(m)} + \lambda^{(m)} A^{-1} B p_2^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0, \end{aligned}$$

procedure uzawa_init(b_1, b_2, **var** x_1, x_2, d_2, p_2);
Löse $Ax_1 = b_1 - Bx_2$;
$d_2 \leftarrow b_2 - B^* x_1$
$p_2 \leftarrow d_2$
end;
procedure uzawa_step(b_1, b_2, **var** x_1, x_2, d_2, p_2);
Löse $Ap_1 = Bp_2$;
$a_2 \leftarrow B^* p_1$;
$\alpha \leftarrow \langle a_2, p_2 \rangle_2$;
$\lambda \leftarrow \langle d_2, p_2 \rangle_2 / \alpha$;
$x_2 \leftarrow x_2 - \lambda p_2$;
$d_2 \leftarrow d_2 - \lambda a_2$;
$x_1 \leftarrow x_1 + \lambda p_1$;
$\mu \leftarrow \langle d_2, a_2 \rangle / \alpha$;
$p_2 \leftarrow d_2 - \mu p_2$
end

Abbildung 6.11: Pseudocode für die Initialisierung und einen Schritt des Uzawa-cg-Verfahrens

und der Vektor $p_1^{(m)} := A^{-1} B p_2^{(m)}$ muss bei der Berechnung des Vektors $a_2^{(m)}$ ohnehin bestimmt werden. Wir können also mit einer einzigen Linearkombination dafür sorgen, dass in jedem Schritt das zu $x_2^{(m)}$ passende $x_1^{(m)}$ zu unserer Verfügung steht.

Der Algorithmus ist in Abbildung 6.11 zusammengefasst und erfordert neben einigen Skalarprodukten und Linearkombinationen lediglich je eine Multiplikation mit B, eine mit B^* und das Lösen eines Gleichungssystems mit der Matrix A. Im Fall der Grundwasserströmung ist A eine Diagonalmatrix, so dass sich diese Aufgabe sehr einfach lösen lässt. Im allgemeinen Fall verwendet man häufig ein iteratives Verfahren, um Näherungslösungen zu gewinnen. Unter diesen Umständen ist es natürlich wichtig, darauf zu achten, dass die Näherungslösungen hinreichend genau sind.

7 Finite-Elemente-Verfahren

In Kapitel 5 haben wir einige Beispiele für Finite-Differenzen-Verfahren kennen gelernt, mit deren Hilfe sich partielle Differentialgleichungen in lineare Gleichungssysteme überführen lassen. Diese *Diskretisierung* ist von entscheidender Bedeutung, um die Lösung der Gleichung zu approximieren. In der Praxis erweisen sich Finite-Differenzen-Verfahren häufig als unflexibel, da beispielsweise die Behandlung komplizierterer Geometrien oder nicht hinreichend oft differenzierbarer Lösungen schwierig ist.

Aus diesem Grund beschäftigen wir uns in diesem Kapitel mit einer alternativen Diskretisierungtechnik, für die wesentlich allgemeinere Gebiete und wesentlich weniger glatte Funktionen keine großen Schwierigkeiten verursachen: Die Methode der *finiten Elemente*.

7.1 Variationsformulierung

Ein Problem bei der Analyse der Finite-Differenzen-Verfahren sind die hohen Anforderungen an die Differenzierbarkeit, beispielsweise mussten wir für die Potentialgleichung (5.9) voraussetzen, dass die Lösung *viermal* stetig differenzierbar ist. Diese Schwierigkeit können wir vermeiden, indem wir zu einer *Variationsformulierung* übergehen, bei der einige Ableitungen per partieller Integration auf eine zweite Funktion übertragen werden und nicht mehr stören.

Als Beispiel untersuchen wir die verallgemeinerte Potentialgleichung

$$-\nabla \cdot (E\nabla u)(x) = f(x) \qquad \text{für alle } x \in \Omega \tag{7.1}$$

auf einem Gebiet $\Omega \subseteq \mathbb{R}^2$. Gegeben sind hier Funktionen

$$f : \Omega \to \mathbb{R}, \qquad E : \Omega \to \mathbb{R}^{2\times 2},$$

gesucht ist eine hinreichend oft differenzierbare Funktion

$$u : \overline{\Omega} \to \mathbb{R}.$$

Wie zuvor benötigen wir Randbedingungen, um die Eindeutigkeit der Lösung sicherzustellen. Der Einfachheit halber verwenden wir *Dirichlet-Randbedingungen*, schreiben also den Wert der Funktion u auf dem Rand vor. Besonders einfach ist die *homogene Dirichlet-Randbedingung*

$$u|_{\partial\Omega} = 0,$$

die besagt, dass die Funktion u auf dem Rand $\partial\Omega$ des Gebiets gleich null sein soll. Zur Abkürzung führen wir den Raum

$$C_0^1(\Omega) := \{u \in C(\overline{\Omega}) \ : \ u|_\Omega \in C^1(\Omega),\ u|_{\partial\Omega} = 0\}$$

ein und schreiben kurz, dass wir ein $u \in C_0^1(\Omega)$ suchen, das die Gleichung (7.1) erfüllt.

Gauß'sche Integralsatz. Wir setzen nun voraus, dass Ω ein *Normalengebiet* ist. Die genaue Definition soll uns hier nicht interessieren, relevant ist, dass es eine Abbildung

$$n : \partial\Omega \to \mathbb{R}^2$$

gibt, die jedem Randpunkt $x \in \partial\Omega$ einen *äußeren Einheitsnormalenvektor* derart zuordnet, dass der *Gauß'sche Integralsatz*

$$\int_\Omega \nabla \cdot w(x)\,dx = \int_{\partial\Omega} \langle w(x), n(x)\rangle_2\,dx \quad \text{für alle } w \in C^0(\overline{\Omega}, \mathbb{R}^2) \text{ mit } w|_\Omega \in C^1(\Omega, \mathbb{R}^2) \tag{7.2}$$

gilt. Beispiele für Normalengebiete sind etwa Polygone, bei denen der äußere Einheitsnormalenvektor in allen Randpunkten mit Ausnahme der Ecken in der offensichtlichen Weise (senkrecht auf der Kante, normiert, aus dem Gebiet heraus orientiert) definiert ist, während wir ihn in den Ecken beliebig wählen können, da sie eine Nullmenge bilden und das Integral (7.1) nicht beeinflussen.

Bei Gebieten mit einem gebogenen Rand können wir zu einem Randpunkt $x \in \partial\Omega$ eine Tangente der Randkurve konstruieren und den Einheitsnormalenvektor dieser Tangente verwenden.

Partielle Integration. Mit Hilfe der Produktregel können wir aus dem Gauß'schen Integralsatz eine Formel für die partielle Integration im mehrdimensionalen Raum gewinnen: Wir fixieren Funktionen $u, v \in C^0(\overline{\Omega})$ mit $u|_\Omega, v|_\Omega \in C^1(\Omega)$ und setzen

$$w(x) := \begin{pmatrix} u(x)v(x) \\ 0 \end{pmatrix} \qquad \text{für alle } x \in \overline{\Omega}.$$

Dann folgt

$$\nabla \cdot w(x) = \partial_1(uv)(x) = u(x)\partial_1 v(x) + v(x)\partial_1 u(x) \qquad \text{für alle } x \in \Omega,$$

so dass der Gauß'sche Integralsatz die Form

$$\int_\Omega u(x)\partial_1 v(x) + v(x)\partial_1 u(x)\,dx = \int_{\partial\Omega} u(x)v(x)n_1(x)\,dx$$

annimmt, also insbesondere

$$\int_\Omega v(x)\partial_1 u(x)\,dx = -\int_\Omega u(x)\partial_1 v(x)\,dx + \int_{\partial\Omega} u(x)v(x)n_1(x)\,dx. \tag{7.3a}$$

Entsprechend erhalten wir auch

$$\int_\Omega v(x)\partial_2 u(x)\,dx = -\int_\Omega u(x)\partial_2 v(x)\,dx + \int_{\partial\Omega} u(x)v(x)n_2(x)\,dx. \tag{7.3b}$$

Ein Vergleich mit der bekannten partiellen Integration auf Intervallen zeigt, dass sich lediglich die Randterme geändert haben.

Variationsformulierung. Da unser Ziel darin besteht, die Gleichung (7.1) durch partielle Integration umzuformen, müssen wir ein Integral einführen. Dazu multiplizieren wir die Gleichung mit einer Funktion $v \in C_0^1(\Omega)$ und integrieren das Produkt, um

$$-\int_\Omega v(x)\nabla \cdot (E\nabla u)(x)\,dx = \int_\Omega v(x)f(x)\,dx$$

zu erhalten. Wir schreiben die Divergenz aus und erhalten

$$-\int_\Omega v(x)\partial_1(E\nabla u)_1(x)\,dx - \int_\Omega v(x)\partial_2(E\nabla u)_2(x)\,dx = \int_\Omega v(x)f(x)\,dx,$$

so dass wir wie gewünscht partiell integrieren können: Es gelten die Gleichungen

$$\begin{aligned}
-\int_\Omega v(x)\partial_1(E\nabla u)_1(x)\,dx &= \int_\Omega \partial_1 v(x)(E\nabla u)_1(x)\,dx - \int_{\partial\Omega} v(x)(E\nabla u)_1(x)n_1(x)\,dx,\\
-\int_\Omega v(x)\partial_2(E\nabla u)_2(x)\,dx &= \int_\Omega \partial_2 v(x)(E\nabla u)_2(x)\,dx - \int_{\partial\Omega} v(x)(E\nabla u)_2(x)n_2(x)\,dx,
\end{aligned}$$

mit deren Hilfe wir zu

$$\int_\Omega \langle \nabla v(x), (E\nabla u)(x)\rangle_2 = \int_\Omega v(x)f(x)\,dx + \int_{\partial\Omega} v(x)\langle n(x), (E\nabla u)(x)\rangle_2\,dx$$

gelangen. Da $v|_{\partial\Omega} = 0$ nach Definition gilt, fällt das Randintegral weg und wir erhalten die folgende *Variationsaufgabe*:

Finde eine Funktion $u \in C_0^1(\Omega)$ mit

$$\int_\Omega \langle \nabla v(x), E(x)\nabla u(x)\rangle_2\,dx = \int_\Omega v(x)f(x)\,dx \quad \text{für alle } v \in C_0^1(\Omega). \tag{7.4}$$

Die Funktionen $v \in C_0^1(\Omega)$ nennt man in diesem Kontext aus naheliegenden Gründen *Testfunktionen*.

Entsprechend unserer Herleitung ist jede Lösung der ursprünglichen Gleichung (7.1) auch eine Lösung dieses Variationsproblems. Falls wir zeigen können, dass die Variationsaufgabe *eindeutig* lösbar ist, folgt daraus, dass ihre Lösung mit der der ursprünglichen Gleichung übereinstimmen, falls letztere eine besitzt. Ansonsten nennen wir die Lösung der Variationsaufgabe eine *verallgemeinerte Lösung*.

7.2 Schwache Differenzierbarkeit

Der entscheidende Vorteil der Variationsaufgabe besteht darin, dass die Funktionen u und v lediglich *einmal* stetig differenzierbar zu sein brauchen. Da wir das Produkt der Ableitungen auch noch integrieren, stellt sich die Frage, ob es ausreichen könnte, die Differenzierbarkeit nur in einem geeigneten Sinn „fast überall“ zu fordern.

Das ist in der Tat möglich, und dieser abgeschwächte Begriff der Differenzierbarkeit erlaubt es, relativ einfach die eindeutige Lösbarkeit der Variationsaufgabe nachzuweisen.

Ein Blick auf (7.4) legt nahe, dass die Funktionen ∇v und $A\nabla u$ gar nicht stetig zu sein brauchen, ihr Produkt muss lediglich in einem geeigneten Sinn *integrierbar* sein. Der „geeignete Sinn“ ist in diesem Fall der des Lebesgue-Integrals, denn dieser Integralbegriff besitzt bestimmte Eigenschaften, die es uns ermöglichen, elegant mit Grenzwerten integrierbarer Funktionen zu arbeiten und so sowohl die Begriff der Ableitung zu verallgemeinern als auch die Lösbarkeit zu etablieren.

Definition 7.1 (L^2-Raum) *Wir definieren den Raum*

$$L^2(\Omega) := \{u : \Omega \to \mathbb{R} \ : \ u \text{ ist messbar und } u^2 \text{ ist Lebesgue-integrierbar}\}$$

mit der Norm

$$\|u\|_{L^2} := \left(\int u(x)^2 \, dx \right)^{1/2} \qquad \text{für alle } u \in L^2(\Omega)$$

und dem mit ihr verträglichen Skalarprodukt

$$\langle u, v \rangle_{L^2} := \int u(x) v(x) \, dx \qquad \text{für alle } u, v \in L^2(\Omega).$$

Den in der Integrationstheorie üblichen Konventionen entsprechend werden zwei Funktionen $u_1, u_2 \in L^2(\Omega)$ als gleich angesehen, falls sie sich nur auf einer Lebesgue-Nullmenge unterscheiden.

Aus der *Cauchy-Schwarz-Ungleichung*

$$|\langle u, v \rangle_{L^2}| \leq \|u\|_{L^2} \|v\|_{L^2} \qquad \text{für alle } u, v \in L^2(\Omega) \tag{7.5}$$

folgt, dass das Skalarprodukt *stetig* bezüglich der L^2-Norm ist. Falls $u, v \in L^2(\Omega)$ gegeben sind und eine Folge $(u_n)_{n=1}^{\infty}$ in $L^2(\Omega)$ mit

$$\lim_{n \to \infty} \|u - u_n\|_{L^2} = 0$$

existiert, erhalten wir

$$\lim_{n \to \infty} |\langle u, v \rangle_{L^2} - \langle u_n, v \rangle_{L^2}| = \lim_{n \to \infty} |\langle u - u_n, v \rangle_{L^2}| \leq \lim_{n \to \infty} \|u - u_n\|_{L^2} \|v\|_{L^2} = 0. \tag{7.6}$$

Schwache Ableitung per Grenzwert. Wir verallgemeinern den Begriff der Ableitung, indem wir Grenzwerte klassisch differenzierbarer Funktionen untersuchen. Für jede Funktion $u \in L^2(\Omega)$ existiert eine Folge $(u_n)_{n=1}^{\infty}$ in $C_0^1(\Omega)$, die

$$\lim_{n \to \infty} \|u - u_n\|_{L^2} = 0$$

erfüllt, also in der L^2-Norm gegen u konvergiert. Sei $i \in \{1, 2\}$ gegeben. Falls eine Funktion $v \in L^2(\Omega)$ existiert, für die

$$\lim_{n \to \infty} \|v - \partial_i u_n\|_{L^2} = 0$$

gilt, die also der L^2-Grenzwert der partiellen Ableitungen der gegen u konvergenten Folge ist, nennen wir v die *schwache i-te Ableitung* der Funktion u.

Da diese Definition für die Praxis eher selten brauchbar ist, leiten wir eine Alternative her: Für eine beliebige Funktion $\varphi \in C_0^1(\Omega)$ erhalten wir durch partielle Integration gemäß (7.3) und (7.6) die Gleichung

$$\begin{aligned}\langle \varphi', u\rangle_{L^2} &= \lim_{n\to\infty} \langle \varphi', u_n\rangle_{L^2} = \lim_{n\to\infty} \int_\Omega \varphi'(x) u_n(x)\, dx \\ &= \lim_{n\to\infty} -\int_\Omega \varphi(x) u_n'(x)\, dx = \lim_{n\to\infty} -\langle \varphi, u_n'\rangle_{L^2} = -\langle \varphi, v\rangle_{L^2},\end{aligned}$$

da die Randterme wegen $\varphi|_{\partial\Omega} = 0$ wegfallen. Durch diese Gleichung lässt sich die schwache Ableitung definieren, ohne den Umweg über Folgen gehen zu müssen.

Definition 7.2 (Schwache Ableitung) *Sei $u \in L^2(\Omega)$ und $i \in \{1, 2\}$. Falls eine Funktion $v \in L^2(\Omega)$ existiert, die*

$$\langle \varphi', u\rangle_{L^2} = -\langle \varphi, v\rangle_{L^2} \qquad \text{für alle } \varphi \in C_0^1(\Omega)$$

erfüllt, nennen wir v eine schwache i-te Ableitung *der Funktion u und bezeichnen sie mit $\partial_i u$.*

Die schwache Ableitung ist eindeutig: Falls es zwei Funktionen $v_1, v_2 \in L^2(\Omega)$ gibt, die die Gleichung

$$-\langle \varphi, v_1\rangle_{L^2} = \langle \varphi', u\rangle_{L^2} = -\langle \varphi, v_2\rangle_{L^2} \qquad \text{für alle } \varphi \in C_0^1(\Omega)$$

erfüllen, folgt

$$\langle \varphi, v_1 - v_2\rangle_{L^2} = 0 \qquad \text{für alle } \varphi \in C_0^1(\Omega).$$

Wir wählen eine Folge $(\varphi_n)_{n=1}^\infty$ mit

$$\lim_{n\to\infty} \|(v_1 - v_2) - \varphi_n\|_{L^2} = 0$$

und erhalten mit (7.6)

$$\|v_1 - v_2\|_{L^2}^2 = \langle v_1 - v_2, v_1 - v_2\rangle_{L^2} = \lim_{n\to\infty} \langle \varphi_n, v_1 - v_2\rangle_{L^2} = 0,$$

also folgt $v_1 = v_2$. Insbesondere stimmen unsere beiden Definitionen der schwachen Ableitung überein. Falls u klassisch differenzierbar ist, entspricht die klassische Ableitung der schwachen.

Definition 7.3 (Sobolew-Raum) *Wir bezeichnen*

$$H^1(\Omega) := \{u \in L^2(\Omega) \ : \ \text{schwache Ableitungen } \partial_1 u, \partial_2 u \in L^2(\Omega) \text{ existieren}\}$$

als den Sobolew-Raum erster Ordnung*, ausgestattet mit der* Sobolew-Norm

$$\|u\|_{H^1} := \left(\|u\|_{L^2}^2 + \|\partial_1 u\|_{L^2}^2 + \|\partial_2 u\|_{L^2}^2\right)^{1/2} \qquad \text{für alle } u \in H^1(\Omega).$$

Den Teilraum

$$H_0^1(\Omega) := \{u \in H^1(\Omega) \ : \ \lim_{n\to\infty} \|u - u_n\|_{H^1} = 0 \text{ für eine Folge } (u_n)_{n=1}^\infty \text{ in } C_0^1(\Omega)\}$$

nennen wir entsprechend den Sobolew-Raum erster Ordnung mit Nullrandbedingungen.

Für eine Funktion $u \in H^1(\Omega)$ können wir den *schwachen Gradienten* $\nabla u \in L^2(\Omega, \mathbb{R}^2)$ definieren, indem wir die klassischen partiellen Ableitungen durch schwache Ableitungen ersetzen. Dank unserer Notation ändert sich die Notation nicht, und wir können die folgende verallgemeinerte Variationsaufgabe untersuchen:

Finde eine Funktion $u \in H_0^1(\Omega)$ mit

$$\int_\Omega \langle \nabla v(x), E(x)\nabla u(x)\rangle_2 \, dx = \int_\Omega v(x) f(x)\, dx \quad \text{für alle } v \in H_0^1(\Omega). \quad (7.7)$$

Verallgemeinerte Variationsaufgabe. Da die schwache Ableitung eine Verallgemeinerung der klassischen Ableitung ist, ist eine klassisch differenzierbare Lösung der ursprünglichen Variationsaufgabe (7.4) auch eine Lösung der verallgemeinerten Aufgabe.

Mit der Abkürzung $V := H_0^1(\Omega)$ und den Funktionen

$$a : V \times V \to \mathbb{R}, \qquad (u,v) \mapsto \int_\Omega \langle \nabla v(x), E(x)\nabla u(x)\rangle_2 \, dx,$$
$$\lambda : V \to \mathbb{R}, \qquad v \mapsto \int_\Omega v(x) f(x),$$

können wir (7.7) kurz als

$$a(v,u) = \lambda(v) \qquad \text{für alle } v \in V \qquad (7.8)$$

schreiben. Es stellt sich die Frage, unter welchen Bedingungen diese Gleichung eine eindeutig bestimmte Lösung $u \in V$ besitzt.

Bemerkung 7.4 (Lösbarkeit) *Die Lösbarkeit der Aufgabe (7.8) können wir sicherstellen, indem wir von geeigneten Annahmen an das Gebiet Ω, die Koeffizientenmatrizen E und die rechte Seite f ausgehen:*

- *Das Gebiet $\Omega \subseteq \mathbb{R}^2$ muss eine* beschränktes Normalengebiet *sein, beispielsweise ein Polygon.*
- *Es muss $f \in L^2(\Omega)$ gelten. Das folgt beispielsweise aus $f \in C(\overline{\Omega})$.*

- *Es müssen Konstanten $e_1, e_2 \in \mathbb{R}_{>0}$ existieren mit*

$$E(x)^* = E(x), \quad e_1\|y\|_2^2 \leq \langle E(x)y, y\rangle_2 \leq e_2\|y\|_2^2 \qquad \text{für alle } x \in \Omega,\ y \in \mathbb{R}^2,$$

  die Matrizen müssen also selbstadjungiert sowie gleichmäßig beschränkt und positiv definit sein.

Unter diesen Bedingungen lässt sich mit etwas Funktionalanalysis, insbesondere dem Lemma von Céa*, nachweisen, dass genau ein $u \in V$ existiert, das (7.8) erfüllt.*

Für den Beweis dieser Aussage ist entscheidend, dass V ein Hilbert-Raum *ist. $H_0^1(\Omega)$ erbt diese Eigenschaft von $L^2(\Omega)$, während $C_0^1(\Omega)$ sie nicht besitzt.*

7.3 Galerkin-Diskretisierung

Aus der Variationsaufgabe (7.8) lässt sich direkt ein lineares Gleichungssystem gewinnen, mit dessen Hilfe sich die Lösung approximieren lässt: Die Idee des *Galerkin-Verfahrens* besteht darin, den Raum V durch einen endlich-dimensionalen Teilraum V_h zu ersetzen und die Lösung der Variationsaufgabe auf diesem Teilraum als Näherung der Lösung des ursprünglichen Problems zu verwenden.

Es bietet sich an, den Raum V_h über eine Basis zu definieren: Für eine endliche Indexmenge $\mathcal{I}$ wählen wir Funktionen

$$\varphi_i \in V \qquad \text{für alle } i \in \mathcal{I},$$

die linear unabhängig sein sollen. Dann ist die Abbildung

$$\Phi : \mathbb{R}^{\mathcal{I}} \to V, \qquad x \mapsto \sum_{i\in\mathcal{I}} x_i\varphi_i,$$

injektiv, ihr Kern enthält also nur den Nullvektor. Ihr Bild

$$V_h := \operatorname{Bild}\Phi = \left\{ \sum_{i\in\mathcal{I}} x_i\varphi_i \ :\ x \in \mathbb{R}^{\mathcal{I}} \right\}$$

definiert den Raum, auf den wir die Variationsaufgabe einschränken:

Finde eine Funktion $u_h \in V_h$ mit

$$a(v_h, u_h) = \lambda(v_h) \qquad \text{für alle } v_h \in V_h. \tag{7.9}$$

Bemerkung 7.5 (Lösbarkeit) *Unter den bereits in Bemerkung 7.4 geforderten Bedingungen besitzt auch (7.9) eine eindeutige Lösung $u_h \in V_h$.*

Bemerkung 7.6 (Approximation) *Ebenfalls unter den Bedingungen der Bemerkung 7.4 folgt aus dem* Satz von Lax-Milgram *die Fehlerabschätzung*

$$\|u - u_h\|_{H^1} \leq C\|u - v_h\|_{H^1} \qquad \text{für alle } v_h \in V_h$$

mit einer nur von e_1, e_2 und dem Gebiet Ω abhängenden Konstanten $C \in \mathbb{R}_{>0}$: Die Näherungslösung u_h ist „fast“ die in V_h bestmögliche Approximation der Lösung u.

Lineares Gleichungssystem. Der Vorteil des Variationsproblems (7.9) besteht darin, dass es sich in ein lineares Gleichungssystem überführen lässt: Für beliebige Vektoren $x, y \in \mathbb{R}^{\mathcal{I}}$ können wir

$$u_h := \sum_{j \in \mathcal{I}} x_j \varphi_j, \qquad v_h := \sum_{i \in \mathcal{I}} y_i \varphi_i$$

definieren und erhalten

$$a(v_h, u_h) = a\left(\sum_{i \in \mathcal{I}} y_i \varphi_i, \sum_{j \in \mathcal{I}} x_j \varphi_j\right) = \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{I}} y_i x_j a(\varphi_i, \varphi_j),$$
$$\lambda(v_h) = \lambda\left(\sum_{i \in \mathcal{I}} y_i \varphi_i\right) = \sum_{i \in \mathcal{I}} y_i \lambda(\varphi_i).$$

Wir führen die Matrix $A \in \mathbb{R}^{\mathcal{I} \times \mathcal{I}}$ und den Vektor $b \in \mathbb{R}^{\mathcal{I}}$ durch

$$a_{ij} = a(\varphi_i, \varphi_j), \qquad b_i = \lambda(\varphi_i) \qquad \text{für alle } i, j \in \mathcal{I} \tag{7.10}$$

ein und folgern

$$a(v_h, u_h) = \langle y, Ax \rangle_2, \qquad \lambda(v_h) = \langle y, b \rangle_2.$$

Falls nun der Vektor x gerade so gewählt ist, dass u_h die Lösung der Variationsaufgabe (7.9) ist, folgt

$$\langle y, Ax \rangle_2 = a(v_h, u_h) = \lambda(v_h) = \langle y, b \rangle_2 \qquad \text{für alle } v_h \in \mathbb{R}^{\mathcal{I}}.$$

Indem wir $y = Ax - b$ setzten, erhalten wir unmittelbar

$$\|Ax - b\|_2^2 = \langle y, Ax - b \rangle_2 = \langle y, Ax \rangle_2 - \langle y, b \rangle_2 = 0,$$

also ist x gerade die Lösung des Gleichungssystems $Ax = b$.

Arbeitsschritte. Wir können also u_h in drei Schritten berechnen:

1. Konstruiere die *Steifigkeitsmatrix* $A \in \mathbb{R}^{\mathcal{I} \times \mathcal{I}}$ und den *Lastvektor* $b \in \mathbb{R}^{\mathcal{I}}$, berechne also die Koeffizienten gemäß (7.10).
2. Berechne die Lösung $x \in \mathbb{R}^{\mathcal{I}}$ des linearen Gleichungssystems $Ax = b$.
3. Setze $u_h = \sum_{j \in \mathcal{I}} x_j \varphi_j$.

Im Gegensatz zu Finite-Differenzen-Verfahren, bei denen die Matrizen explizit gegeben sind, müssen sie bei Galerkin-Verfahren im Allgemeinen berechnet werden. Der resultierende erhöhte Rechenaufwand wird dadurch ausgegliechen, dass das Galerkin-Verfahren wesentlich flexibler ist: Sowohl das Gebiet als auch die Differentialgleichung können wesentlich allgemeiner gewählt werden, und auch bei der Wahl des Ansatzraums V_h gibt es nur wenige Einschränkungen.

7.4 Integration

Integration auf Rechtecken. Sowohl bei der Berechnung des Vektors b als auch bei der der Matrix A müssen wir Integrale auf dem Gebiet $\Omega \subseteq \mathbb{R}^2$ berechnen, wir müssen also einen Weg finden, um den Integralbegriff auf mehrdimensionale Mengen zu übertragen. Besonders einfach ist diese Aufgabe für achsenparallele Rechtecke zu lösen: Für ein Rechteck $R = [a, b] \times [c, d]$ würden wir erwarten, dass das Integral über die konstante Funktion 1 gerade den Flächeninhalt des Rechtecks R ergibt, also $(b - a)(d - c)$, denn das ist gerade das Produkt aus dem Flächeninhalt der Grundfläche und der Höhe des Volumens. Dieser Flächeninhalt lässt sich in der Form

$$\int_R 1\,dx = (b-a)(c-d) = \left(\int_a^b 1\,dx_1\right)\left(\int_c^d 1\,dx_2\right) = \int_a^b \int_c^d 1\,dx_2\,dx_1$$

schreiben, die nahe legt, wie das zweidimensionale Integral definiert werden sollte:

$$\int_R f(x) = \int_a^b \int_c^d f(x_1, x_2)\,dx_2\,dx_1.$$

Alle Eigenschaften des eindimensionalen Integralbegriffs übertragen sich mit Hilfe dieser Eigenschaft auch auf zweidimensionale Integrale, insbesondere können Quadraturformeln wie die Mittelpunktregel (vgl. Lemma 5.3) zum Einsatz kommen

$$\int_R f(x) \approx (b-a)(d-c) f\left(\frac{b+a}{2}, \frac{d+c}{2}\right),$$

falls $b - a$ und $d - c$ klein genug sind.

Integration auf Dreiecken. Mit dreieckigen Gebieten können wir ähnlich verfahren: Wir erwarten, dass das Integral der konstanten Einsfunktion über dem Dreieck

$$\widehat{T} := \{x \in \mathbb{R}^2 \ : \ 0 \le x_1,\ 0 \le x_2,\ x_1 + x_2 \le 1\}$$

gerade der Fläche dieses Dreiecks entspricht, es sollte also

$$\int_{\widehat{T}} 1\,dx = 1/2 = \int_0^1 1 - x_1\,dx_1 = \int_0^1 \int_0^{1-x_1} 1\,dx_2\,dx_1$$

gelten. Dementsprechend definieren wir das Integral durch

$$\int_{\widehat{T}} f(x)\,dx = \int_0^1 \int_0^{1-x_1} f(x_1, x_2)\,dx_2\,dx_1.$$

Variablentransformation. Um auch allgemeinere Dreiecke behandeln zu können, bietet es sich an, sie mit Hilfe einer *Variablenstransformation* auf das bereits bekannte Dreieck $\widehat{T}$ zurückzuführen. Im mehrdimensionalen Fall wird aus der wohlbekannten Substitutionsregel des Riemann-Integrals

$$\int_a^b g'(\hat{x}) f(g(\hat{x}))\,d\hat{x} = \int_{g(a)}^{g(b)} f(x)\,dx$$

die *Variablentransformationsformel*

$$\int_{\widehat{\Omega}} |\det D\Phi(\hat{x})|\ f(\Phi(\hat{x}))\,d\hat{x} = \int_{\Phi(\Omega)} f(x)\,dx, \tag{7.11}$$

bei der $\Phi : \widehat{\Omega} \to \Omega$ ein C^1-*Diffeomorphismus* sein muss, also bijektiv und in jedem Punkt $\hat{x} \in \widehat{\Omega}$ stetig differenzierbar mit invertierbarer Ableitung $D\Phi(\hat{x})$.

Falls nun $T \subseteq \mathbb{R}^2$ ein Dreieck mit den Eckpunkten $a, b, c \in \mathbb{R}^2$ ist, falls also

$$T = \{s_1 a + s_2 b + s_3 c \ :\ 0 \le s_1,\ 0 \le s_2,\ 0 \le s_3,\ s_1 + s_2 + s_3 = 1\}$$

gilt, können wir es mit Hilfe der Abbildung

$$\Phi_T : \widehat{T} \to T, \qquad\qquad \hat{x} \mapsto (1 - \hat{x}_1 - \hat{x}_2)a + \hat{x}_1 b + \hat{x}_2 c,$$

als Bild $\Phi_T(\widehat{T})$ des bereits bekannten Dreieckes $\widehat{T}$ darstellen. Es gilt

$$D\Phi_T = \begin{pmatrix} b_1 - a_1 & c_1 - a_1 \\ b_2 - a_2 & c_2 - a_2 \end{pmatrix},$$

also ist Φ_T genau dann ein C^1-Diffeomorphismus, wenn $b-a$ und $c-a$ linear unabhängig sind. Das ist genau dann der Fall, wenn T nicht ausgeartet ist, also einen echt positiven Flächeninhalt aufweist.

Mit der Variablentransformationsformel (7.11) erhalten wir dann die Gleichung

$$|\det D\Phi_T| \int_{\widehat{T}} f(\Phi_T(\hat{x}))\,d\hat{x} = \int_T f(x)\,dx, \tag{7.12}$$

mit deren Hilfe wir Integrale über beliebigen Dreiecken berechnen können.

Quadratur auf Dreiecken. Die Konstruktion mathematisch eleganter Quadraturformeln auf Dreiecken ist schwierig, weil einige wichtige Eigenschaften eindimensionaler Funktionen verloren gehen. Üblich sind die *Schwerpunktregel*

$$\int_{\widehat{T}} f(x)\,dx \approx \frac{1}{2} f(1/3, 1/3),$$

die *Eckenregel*

$$\int_{\widehat{T}} f(x)\,dx \approx \frac{1}{6}(f(0,0) + f(1,0) + f(0,1)),$$

sowie die *Kantenregel*

$$\int_{\widehat{T}} f(x)\,dx \approx \frac{1}{6}(f(1/2,1/2) + f(0,1/2) + f(1/2,0)).$$

Mit Hilfe der *Duffy-Transformation*

$$\Phi_{\text{Duffy}} : [0,1] \times [0,1] \to \widehat{T}, \qquad \hat{x} \mapsto \begin{pmatrix} \hat{x}_1 \\ (1-\hat{x}_1)\hat{x}_2 \end{pmatrix},$$

lässt sich das Dreieck $\widehat{T}$ auf das Einheitsquadrat $[0,1] \times [0,1]$ zurückführen und die Variablentransformationsformel (7.11) anwenden, um

$$\int_{\widehat{T}} f(x)\,dx = \int_0^1 (1-x_1) \int_0^1 f(x_1, (1-x_1)x_2)\,dx_2\,dx_1$$

zu erhalten. Aus konventionellen Quadraturformeln für das Intervall $[0,1]$ lassen sich so einfach Quadraturformeln für das Dreieck $\widehat{T}$ konstruieren. Etwas unbefriedigend bei diesem Ansatz ist, dass dadurch die einzelnen Eckpunkte des Dreiecks nicht mehr gleich behandelt werden, so dass die Symmetrie verloren geht.

Gitter. Bei der Approximation eindimensionaler Integrale wird das Intervall häufig in kleinere Teilintervalle unterteilt, um den Quadraturfehler zu reduzieren. Entsprechend können wir auch bei zweidimensionalen Gebieten verfahren: Wir versuchen, sie als (fast) disjunkte Vereinigung von Gebieten darzustellen, mit denen sich gut arbeiten lässt.

Im zweidimensionalen Fall bietet es sich an, *Polygongebiete* zu untersuchen, die sich als Vereinigung von Dreiecken darstellen lassen.

Definition 7.7 (Triangulierung) *Sei $\Omega \subseteq \mathbb{R}^2$ ein Gebiet. Sei $\mathcal{T}$ eine Familie nicht ausgearteter Dreiecke. Wir nennen $\mathcal{T}$ eine* Triangulierung *(manchmal auch* Gitter*) des Gebiets Ω, falls*

$$\overline{\Omega} = \bigcup_{T \in \mathcal{T}} T$$

gilt und für alle $T_1, T_2 \in \mathcal{T}$ der Schnitt $T_1 \cap T_2$ entweder

- *leer,*
- *ein gemeinsamer Eckpunkt,*
- *eine gemeinsame Kante oder*
- *$T_1 = T_2$ ist.*

Falls eine Triangulierung des Gebiets Ω existiert und Ω zusammenhängend ist, nennen wir Ω ein Polygongebiet.

Es lässt sich einfach nachprüfen, dass für eine Triangulierung $\mathcal{T}$ eines Gebiets Ω gerade

$$\int_\Omega f(x)\,dx = \sum_{T \in \mathcal{T}} \int_T f(x)\,dx$$

gilt. Da wir bereits wissen, wie sich Integrale über Dreiecke berechnen lassen, können wir mit Hilfe einer Triangulierung auch Integrale auf beliebigen Polygongebieten berechnen.

Bemerkung 7.8 (Gekrümmte Ränder) *Wir können wesentlich allgemeinere Gebiete behandeln, wenn wir allgemeinere Funktionen Φ_T zulassen. Solange Φ_T bijektiv und $D\Phi_T(\hat{x})$ in jedem Punkt $\hat{x} \in \widehat{T}$ des Definitionsgebiets invertierbar bleibt, erhalten wir mit $T := \Phi_T(\widehat{T})$ verallgemeinerte Dreiecke. Die Formel (7.11) bleibt gültig, aber wir können die Determinante im Allgemeinen nicht mehr aus dem Integral herausziehen, weil sie nicht länger konstant ist.*

Datenstruktur. In einem Programm stellen wir die Triangulierung $\mathcal{T}$ durch drei Zahlen und drei Arrays dar: Die Zahlen `vertices`, `edges` und `triangles` geben die Anzahl der Eckpunkte, Kanten und Dreiecke an. Die Koordinaten der Eckpunkte werden in einem Array `x[vertices][2]` gespeichert, die Nummern der Anfangs- und Endpunkte der Kanten in diesem Array werden in einem weiteren Array `e[edges][2]` untergebracht, während die Nummern der drei Kanten eines Dreiecks in dem dritten Array `t[triangles][3]` aufbewahrt werden.

Die Koordinaten des Mittelpunkts der Kante `i` können beispielsweise in dieser Struktur durch

```
m0 = 0.5 * (x[e[i][0]][0] + x[e[i][1]][0]);
m1 = 0.5 * (x[e[i][0]][1] + x[e[i][1]][1]);
```

berechnet werden. Da wir keinen direkten Zugriff auf die Eckpunkte des Dreiecks haben, müssen wir sie mit Hilfe der Kanten rekonstruieren:

```
if(e[t[i][(j+1)%3]][0] != e[t[i][j]][0] &&
   e[t[i][(j+1)%3]][0] != e[t[i][j]][1])
  return e[t[i][(j+1)%3]][0];
else
  return e[t[i][(j+1)%3]][1];
```

Die der Kante `j` gegenüber liegende Ecke ist eine der Ecken der Kante `(j+1)%3`, und zwar genau diejenige, die nicht auch eine Ecke der Kante `j` ist.

7.5 Ansatzraum

Da wir nun wissen, wie wir Integrale auf Polygongebieten berechnen können, bietet es sich an, unsere Aufmerksamkeit wieder der Variationsaufgabe zuzuwenden, genauer gesagt der Berechnung der Koeffizienten 7.10 des linearen Gleichungssystems, das ihr entspricht.

Wir gehen davon aus, dass Ω ein Polygongebiet mit einer Triangulierung $\mathcal{T}$ ist, und wir fixieren für jedes $T \in \mathcal{T}$ eine affine Abbildung Φ_T, die das Referenzdreieck $\widehat{T}$ bijektiv auf T abbildet.

Für den Lastvektor müssen wir Integrale der Form

$$b_i = \int_\Omega \varphi_i(x) f(x)\, dx \qquad \text{für alle } i \in \mathcal{I}$$

berechnen. Da $\mathcal{T}$ eine Triangulierung des Gebiets Ω ist, erhalten wir

$$b_i = \sum_{T \in \mathcal{T}} \int_T \varphi_i(x) f(x)\,dx \qquad \text{für alle } i \in \mathcal{I}.$$

Die Integrale über die Dreiecke berechnen wir per Variablentransformationsformel, erhalten also

$$\int_T \varphi_i(x) f(x)\,dx = |\det \Phi_T| \int_{\widehat{T}} \varphi_i \circ \Phi_T(\hat{x}) f \circ \Phi_T(\hat{x})\,d\hat{x} \qquad \text{für alle } i \in \mathcal{I},\ T \in \mathcal{T}.$$

Für unsere Zwecke wäre es natürlich sehr hilfreich, wenn die Funktion $\varphi_i \circ \Phi_T$ eine möglichst einfache Form aufweisen würde.

Referenzelement. Besonders elegant ist der Ansatz, eine Familie $(\hat{\varphi}_k)_{k \in \widehat{\mathcal{I}}}$ von Funktionen auf dem Dreieck $\widehat{T}$ zu definieren, die dann mit Hilfe der Abbildung

$$\varphi_{T,k} := \hat{\varphi}_k \circ \Phi_T^{-1} \qquad \text{für alle } k \in \widehat{\mathcal{I}},\ T \in \mathcal{T}$$

Funktionen auf dem Gitter definieren. Die auf dem gesamten Gebiet Ω definierten *globalen* Basisfunktionen φ_i können wir dann aus den nur auf einem Dreieck definierten *lokalen* Basisfunktionen $\varphi_{T,k}$ zusammensetzen, indem wir für jedes Dreieck $T \in \mathcal{T}$ eine injektive *Indexabbildung* $\iota_T : \widehat{\mathcal{I}} \to \mathcal{I}$ wählen und

$$\varphi_i|_T := \begin{cases} \varphi_{T,k} & \text{falls } \iota_T(k) = i \text{ für ein } k \in \widehat{\mathcal{I}}, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{I},\ T \in \mathcal{T} \tag{7.13}$$

setzen. Sofern für jedes $i \in \mathcal{I}$ mindestens ein $T \in \mathcal{T}$ und ein $k \in \widehat{\mathcal{I}}$ mit $i = \iota_T(k)$ existieren, besteht die so konstruierte Familie $(\varphi_i)_{i \in \mathcal{I}}$ aus linear unabhängigen Funktionen, bildet also wie gewünscht eine Basis eines Funktionenraums.

Assemblierung des Lastvektors. Mit Hilfe der lokalen Basisfunktionen lässt sich der Lastvektor $b \in \mathbb{R}^{\mathcal{I}}$ elegant konstruieren: Es gilt

$$\begin{aligned} b_i &= \sum_{T \in \mathcal{T}} \int_T \varphi_i(x) f(x)\,dx = \sum_{T \in \mathcal{T}} |\det D\Phi_T| \int_{\widehat{T}} \varphi_i \circ \Phi_T(\hat{x}) f \circ \Phi_T(\hat{x})\,d\hat{x} \\ &= \sum_{\substack{T \in \mathcal{T} \\ \iota_T(k) = i \\ \text{für ein } k \in \widehat{\mathcal{I}}}} |\det D\Phi_T| \int_{\widehat{T}} \varphi_{T,k}(\hat{x}) f \circ \Phi_T(\hat{x})\,d\hat{x} \qquad \text{für alle } i \in \mathcal{I}. \end{aligned}$$

Diese Gleichung kann so gelesen werden, dass ein Dreieck $T \in \mathcal{T}$ zu genau den Komponenten b_i einen Beitrag leistet, für die ein $k \in \widehat{\mathcal{I}}$ mit $\iota_T(k) = i$ existiert. Also können wir den Lastvektor b auch aufstellen, indem wir ausgehend von dem Nullvektor die Beiträge aller Dreiecke aufsummieren.

procedure assemble_load($\mathcal{T}$, f, **var** b);
$b \leftarrow 0$;
for $T \in \mathcal{T}$ **do begin**
  $\alpha_T \leftarrow |\det D\Phi_T|$;
  for $k \in \widehat{\mathcal{I}}$ **do**
    Berechne $\hat{b}_{T,k} = \alpha_T \int_{\widehat{T}} \varphi_{T,k}(\hat{x}) f(\Phi_T(\hat{x}))\, d\hat{x}$;
  for $k \in \widehat{\mathcal{I}}$ **do**
    $b_{\iota_T(k)} \leftarrow b_{\iota_T(k)} + \hat{b}_{T,k}$
end

Abbildung 7.1: Assemblierung des Lastvektors

Dieser in Abbildung 7.1 zusammengefasste Algorithmus wird als *Assemblierung* bezeichnet: Für jedes Dreieck wird ein durch

$$\hat{b}_{T,k} = |\det D\Phi_T| \int_{\widehat{T}} \varphi_{T,k}(\hat{x}) f(\Phi_T(\hat{x}))\, d\hat{x} \qquad \text{für alle } k \in \widehat{\mathcal{I}}$$

definierter lokaler *Elementlastvektor* $\hat{b}_T \in \mathbb{R}^{\widehat{\mathcal{I}}}$ berechnet, dessen Komponenten dann an der durch ι_T vorgegebenen Stelle in den *globalen* Lastvektor eingefügt werden. Der Vorteil dieses Zugangs liegt darin, dass direkt mit der für die Darstellung der Triangulierung ohnehin erforderlichen Datenstruktur gearbeitet wird und die Berechnung von ein Dreieck betreffenden Größen wie $\det D\Phi_T$ nur einmal erfolgen muss. Ein Nachteil liegt darin, dass je nach Struktur der Triangulierung eventuell sehr ungeordnete Lese- und Schreibzugriffe auf den Vektor b erfolgen können, die dazu führen können, dass der Hauptspeicher nicht mit optimaler Effizienz arbeitet.

Assemblierung der Steifigkeitsmatrix. Auch die Steifigkeitsmatrix kann in ähnlicher Weise konstruiert werden: Nach Definition haben wir

$$\begin{aligned} a_{ij} &= \int_\Omega \langle \nabla\varphi_i(x), E(x)\nabla\varphi_j(x)\rangle_2\, dx = \sum_{T\in\mathcal{T}} \int_T \langle \nabla\varphi_i(x), E(x)\nabla\varphi_j(x)\rangle_2\, dx \\ &= \sum_{T\in\mathcal{T}} |\det D\Phi_T| \int_{\widehat{T}} \langle \nabla\varphi_i\circ\Phi_T(\hat{x}), E(\Phi_T(\hat{x}))\nabla\varphi_j\circ\Phi_T(\hat{x})\rangle_2\, d\hat{x} \\ &= \sum_{\substack{T\in\mathcal{T} \\ \iota_T(k)=i,\iota_T(\ell)=j \\ \text{für } k,\ell\in\widehat{\mathcal{I}}}} |\det D\Phi_T| \int_{\widehat{T}} \langle \nabla\varphi_{T,k}\circ\Phi_T(\hat{x}), E(\Phi_T(\hat{x}))\nabla\varphi_{T,\ell}\circ\Phi_T(\hat{x})\rangle_2\, d\hat{x} \end{aligned}$$

für alle $i, j \in \mathcal{I}$, so dass wir auch in diesem Fall mit einer über alle Dreiecke $T \in \mathcal{T}$ laufenden Schleife alle Einträge der Matrix berechnen können.

Gradient. Allerdings tritt eine kleine Komplikation auf: Wir benötigen den Gradienten der lokalen Basisfunktionen $\varphi_{T,k}$. Aus deren Definition und der Kettenregel folgt

$$D\varphi_{T,k}(x) = D(\hat{\varphi}_k \circ \Phi_T^{-1})(x) = D\hat{\varphi}_k(\Phi_T^{-1}(x))D(\Phi_T^{-1})(x) \quad \text{für alle } T \in \mathcal{T},\ k \in \widehat{\mathcal{I}},\ \hat{x} \in \widehat{T},$$

und da dank Umkehrsatz gerade

$$D(\Phi_T^{-1})(x) = (D\Phi_T(\Phi_T^{-1}(x)))^{-1} \qquad \text{für alle } T \in \mathcal{T},\ x \in T$$

gilt, ergibt sich

$$D\varphi_{T,k}(\Phi_T(\hat{x})) = D\hat{\varphi}_k(\hat{x})(D\Phi_T(\hat{x}))^{-1} \qquad \text{für alle } T \in \mathcal{T},\ k \in \widehat{\mathcal{I}},\ \hat{x} \in \widehat{T}.$$

Da $D\Phi_T$ in unserem Fall konstant ist, vereinfacht sich der Ausdruck zu

$$D\varphi_{T,k}(\Phi_T(\hat{x})) = D\hat{\varphi}_k(\hat{x})(D\Phi_T)^{-1} \qquad \text{für alle } T \in \mathcal{T},\ k \in \widehat{\mathcal{I}},\ \hat{x} \in \widehat{T}.$$

Der Gradient ist nach Definition gerade die Adjungierte der Jacobi-Matrix, also gilt

$$\nabla\varphi_{T,k}(\Phi_T(\hat{x})) = (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k(\hat{x}) \qquad \text{für alle } T \in \mathcal{T},\ k \in \widehat{\mathcal{I}},\ \hat{x} \in \widehat{T}.$$

Es folgt

$$a_{ij} = \sum_{\substack{T\in\mathcal{T}\\ \iota_T(k)=i,\iota_T(\ell)=j\\ \text{für } k,\ell\in\widehat{\mathcal{I}}}} |\det D\Phi_T| \int_{\widehat{T}} \langle (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k(\hat{x}), E(\Phi_T(x))(D\Phi_T^*)^{-1}\nabla\hat{\varphi}_\ell(\hat{x})\rangle_2 \, d\hat{x}.$$

Elementsteifigkeitsmatrix. Es bietet sich an, zunächst eine zu einem Dreieck gehörende lokale *Elementsteifigkeitsmatrix* $\widehat{A}_T \in \mathbb{R}^{\widehat{\mathcal{I}}\times\widehat{\mathcal{I}}}$ mit den Einträgen

$$\hat{a}_{T,k\ell} = |\det D\Phi_T| \int_{\widehat{T}} \langle (D\Phi_T^*)^{-1}\nabla\varphi_k(\hat{x}), E(\Phi_T(\hat{x}))(D\Phi_T^*)^{-1}\nabla\varphi_\ell(\hat{x})\rangle_2 \, d\hat{x}$$
$$\text{für alle } k, \ell \in \widehat{\mathcal{I}}$$

zu berechnen und dann ihre Einträge mit der Indexabbildung zu den passenden Einträgen der globalen Steifigkeitsmatrix zu addieren.

Der resultierende Algorithmus für die Assemblierung der Steifigkeitsmatrix ist ein Abbildung 7.2 zusammengefasst. In diesem Fall profitieren wir in größerem Umfang als bei der Assemblierung des Lastvektors von der Tatsache, dass wir jedes Dreieck nur einmal besuchen müssen: Wir müssen auch die Matrix $J_T := (D\Phi_T^*)^{-1}$ nur einmal berechnen. In der Regel wird $\hat{a}_{T,k\ell}$ mit einer Quadraturformel ermittelt, und in diesem Fall müssen auch die Gradienten $\nabla\varphi_{T,k} := J_T\nabla\hat{\varphi}_k$ nur einmal für jeden Quadraturpunkt berechnet werden, bevor sie in die Berechnung der Elementsteifigkeitsmatrix einfließen.

procedure assemble_stiffness($\mathcal{T}$, E, **var** a);
$a \leftarrow 0$;
for $T \in \mathcal{T}$ **do begin**
  Berechnet $J_T := (D\Phi_T^*)^{-1}$;
  $\alpha_T \leftarrow |\det D\Phi_T|$;
  for $k, \ell \in \widehat{\mathcal{I}}$ **do**
    Berechne $\hat{a}_{T,k\ell} = \alpha_T \int_{\widehat{T}} \langle J_T \nabla\hat{\varphi}_k(\hat{x}), E(\Phi_T(\hat{x})) J_T \nabla\hat{\varphi}_\ell(\hat{x})\rangle_2 \, d\hat{x}$;
  for $k, \ell \in \widehat{\mathcal{I}}$ **do**
    $a_{\iota_T(k),\iota_T(\ell)} \leftarrow a_{\iota_T(k),\iota_T(\ell)} + \hat{a}_{T,k\ell}$
end

Abbildung 7.2: Assemblierung der Steifigkeitsmatrix

7.6 Stückweise lineare Basisfunktionen

Der Rechenaufwand für die Assemblierung des Lastvektors ist proportional zu dem Produkt aus der Anzahl der Dreiecke $\#\widehat{\mathcal{T}}$ und der Anzahl der lokalen Basisfunktionen $\#\widehat{\mathcal{I}}$, bei der Assemblierung der Steifigkeitsmatrix geht die zweite Zahl quadratisch ein. Wir können also das Gleichungssystem um so schneller aufstellen, je weniger lokale Basisfunktionen wir verwenden.

Polynome. In der Praxis verwendet man in der Regel Polynome als lokale Basis, beispielsweise aus dem Raum

$$\Pi_m^2 := \{x \mapsto x_1^\alpha x_2^\beta \ : \ \alpha, \beta \in \mathbb{N}_0, \ \alpha + \beta \leq m\}$$

der Polynome m-ten Grades in zwei Variablen. Es lässt sich leicht nachrechnen, dass dieser Raum gerade die Dimension

$$\dim \Pi_m^2 = \frac{1}{2}(m+1)(m+2)$$

besitzt, beispielsweise ist Π_0^2 eindimensional, Π_1^2 dreidimensional und Π_2^2 sechsdimensional.

Stückweise Polynome. Sei $\mathcal{T}$ eine Triangulierung eines Gebiets Ω, und sei $(\hat{\varphi}_k)_{k\in\widehat{\mathcal{I}}}$ eine Basis des Polynomraums Π_m^2. Da mit Φ_T auch Φ_T^{-1} für jedes Dreieck $T \in \mathcal{T}$ affin ist, folgt

$$\varphi_{T,k} \in \Pi_m^2|_T \qquad \text{für alle } T \in \mathcal{T}, \ k \in \widehat{\mathcal{I}},$$

und diese Funktionen bilden eine Basis des Raum $\Pi_m^2|_T$ der auf den Definitionsbereich T beschränkten Polynome m-ten Grades.

Aus der Definition (7.13) der Basisfunktionen folgt dann direkt

$$\varphi_i|_T \in \Pi_m^2|_T \qquad \text{für alle } i \in \mathcal{I},\ T \in \mathcal{T}.$$

Da der Raum V_h von diesen Funktionen aufgespannt wird, überträgt sich diese Eigenschaft auf beliebige Funktionen $v_h \in V_h$, wir arbeiten also mit einem Raum *stückweiser Polynome*.

Schwache Differenzierbarkeit. Im Rahmen der Variationsaufgabe (7.7) dürfen wir V_h nur verwenden, falls die Funktionen dieses Raums zu $H_0^1(\Omega)$ gehören. In $L^2(\Omega)$ liegen Funktionen aus V_h auf jeden Fall, da sie auf jedem Dreieck der Triangulierung Polynome und damit insbesondere integrierbar sind. Wir müssen also nur noch untersuchen, ob die Funktionen aus V_h auch schwach differenzierbar sind. Seien $v_h \in V_h$, $\varphi \in C_0^1(\Omega)$ und $p \in \{1,2\}$ gegeben. Für jedes Dreieck $T \in \mathcal{T}$ bezeichnen wir mit $v_T \in \Pi_m^2|_T$ das Polynom, das auf T (abgesehen von Nullmengen) mit v_h übereinstimmt. Damit ist v_T insbesondere stetig differenzierbar und wir können *auf jedem Dreieck* partiell integrieren, um die Gleichung

$$\begin{aligned} -\int_\Omega \partial_p\varphi(x)v_h(x)\,dx &= \sum_{T\in\mathcal{T}} -\int_T \partial_p\varphi(x)v_T(x)\,dx \\ &= \sum_{T\in\mathcal{T}} \int_T \varphi(x)\partial_p v_T(x)\,dx - \int_{\partial T} n_{T,p}(x)\varphi(x)v_T(x)\,dx. \end{aligned} \tag{7.14}$$

zu erhalten, wobei n_T die äußeren Einheitsnormalenvektoren der Dreiecke bezeichnet. Würde der die Randintegrale enthaltende Term entfallen, hätten wir nachgewiesen, dass die schwache Ableitung $\partial_p v_h$ existiert, also sollten wir uns diesen Term etwas näher ansehen.

Wir bezeichnen mit $\mathcal{E}$ die Menge aller Kanten aller Dreiecke der Triangulierung $\mathcal{T}$. Dann können wir das Integral über den Rand eines Dreiecks durch Integrale über seine Kanten ausdrücken:

$$\int_{\partial T} f(x)\,dx = \sum_{\substack{e\in\mathcal{E}\\ e\subseteq T}} \int_e f(x)\,dx.$$

Für jede Kante $e \in \mathcal{E}$ fixieren wir einen Einheitsnormalenvektor n_e.

Untersuchung der Gitterkanten. Wir unterscheiden zwischen Randkanten und inneren Kanten:

$$\mathcal{E}_\partial := \{e \in \mathcal{E} \ :\ e \subseteq \partial\Omega\}, \qquad \mathcal{E}_\Omega := \mathcal{E} \setminus \mathcal{E}_\partial.$$

Für Randkanten $e \in \mathcal{E}_\partial$ folgt aus $\varphi|_{\partial\Omega}$ bereits

$$\int_e n_{T,p}(x)\varphi(x)v_T(x)\,dx = 0.$$

Für innere Kanten $e \in \mathcal{E}_\Omega$ können wir nach Definition der Triangulierung jeweils genau zwei Dreiecke $T_{e,+}, T_{e,-} \in \mathcal{T}$ finden, die e als Kante besitzen, und es muss

$$n_{T_{e,+}}(x) = -n_{T_{e,-}}(x) \qquad \text{für alle } x \in e$$

gelten. Wir wählen die Dreiecke so, dass

$$n_{T_{e,+}} = n_e, \qquad n_{T_{e,-}} = -n_e \qquad \text{für alle } e \in \mathcal{E}_\Omega$$

gilt. Dann folgt

$$\begin{aligned}
\sum_{T\in\mathcal{T}} \int_{\partial T} n_{T,p}(x)\varphi(x)v_T(x)\,dx &= \sum_{T\in\mathcal{T}} \sum_{\substack{e\in\mathcal{E}_\Omega \\ e\subseteq T}} \int_e n_{T,p}(x)\varphi(x)v_T(x)\,dx \\
&= \sum_{e\in\mathcal{E}_\Omega} \sum_{\substack{T\in\mathcal{T} \\ e\subseteq T}} \int_e n_{T,p}(x)\varphi(x)v_T(x)\,dx \\
&= \sum_{e\in\mathcal{E}_\Omega} \int_e n_{T_{e,+},p}(x)\varphi(x)v_{T_{e,+}}(x) - n_{T_{e,+},p}(x)\varphi(x)v_{T_{e,-}}(x)\,dx \\
&= \sum_{e\in\mathcal{E}_\Omega} \int_e \varphi(x)n_{e,p}(x)(v_{T_{e,+}}(x) - v_{T_{e,-}}(x))\,dx.
\end{aligned}$$

Damit dieses Integral für alle $\varphi \in C_0^1(\Omega)$ und für alle $p \in \{1,2\}$ gleich null ist, muss $v_{T_{e,+}}|_e = v_{T_{e,-}}|_e$ gelten, die Funktion v_h muss also gerade *stetig* sein.

Für stückweise konstante Polynome gilt diese Bedingung nur dann, wenn die zusammengesetzte Funktion auf ganz Ω konstant ist. Da wir Funktionen benötigen, die auf dem Rand des Gebiets gleich null sein, bliebe so nur die Nullfunktion übrig.

Stückweise linearer Ansatzraum. Stückweise lineare Polynome sind besser geeignet, also setzen wir

$$V_h := \{v_h \in C(\overline{\Omega}) \ : \ v_h|_T \in \Pi_1^2 \text{ für alle } T \in \mathcal{T},\ v_h|_{\partial\Omega} = 0\}$$

und erhalten mit (7.14) für jedes $v_h \in V_h$ sowohl $v_h \in H_0^1(\Omega)$ als auch

$$(\partial_p v_h)|_T = \partial_p(v_h|_T) \qquad \text{für alle } T \in \mathcal{T},\ p \in \{1,2\},$$

die schwache Ableitung stimmt also auf jedem Dreieck mit der klassischen Ableitung überein. Dadurch wird es uns natürlich erheblich erleichtert, die Gradienten zu berechnen, die wir für die Elementsteifigkeitsmatrix benötigen.

Hutfunktionen. Es bleibt noch die Basis $(\varphi_i)_{i\in\mathcal{I}}$ des Raums V_h zu definieren. Dazu können wir eine nützliche Eigenschaft linearer Polynome ausnutzen: Falls $w \in \Pi_1^2$ gilt, können wir $a, b \in \mathbb{R}^2$ wählen und feststellen, dass

$$\hat{w}_{a,b} : [0,1] \to \mathbb{R}^2, \qquad t \mapsto w((1-t)a + tb),$$

ein lineares Polynom auf $[0,1]$ ist. Nach dem Identitätssatz für (eindimensionale!) Polynome folgt, dass $\hat{w}_{a,b}$ mit dem linearen Polynom

$$t \mapsto (1-t)w(a) + tw(b) \qquad \text{für alle } t \in [0,1]$$

übereinstimmt, denn beide nehmen für $t = 0$ und $t = 1$ dieselben Werte an. Also folgt

$$w((1-t)a + tb) = \hat{w}_{a,b}(t) = (1-t)w(a) + tw(b) \quad \text{für alle } a, b \in \mathbb{R}^2,\ t \in [0,1]. \quad (7.15)$$

Untersuchen wir nun eine Funktion $v_h \in V_h$. Ihre Einschränkung $v_h|_T$ auf ein Dreieck $t \in \mathcal{T}$ ist ein lineares Polynom, also lässt sich (7.15) für Punkte $a, b \in T$ anwenden. Indem wir a und b als die Ecken des Dreiecks wählen, können wir so die Werte von v_h auf den Kanten des Dreiecks berechnen. Dann können wir a und b auf den Kanten wählen, um die Werte im Inneren zu bestimmen. Also ist die Funktion $v_h|_T$ auf dem gesamten Dreieck bereits durch ihre Werte in den drei Eckpunkten des Dreiecks festgelegt, und die gesamte Funktion v_h durch ihre Werte in den Eckpunkten der Triangulierung.

Wir bezeichnen die Menge aller Eckpunkte der Triangulierung $\mathcal{T}$ mit $\mathcal{N}$. Da wir nun wissen, dass v_h durch den Vektor $(v_h(x))_{x\in\mathcal{N}}$ eindeutig definiert ist, bietet es sich an, die Basisfunktionen so zu wählen, dass sich mit ihrer Hilfe aus den Werten in den Ecken die Funktion v_h rekonstruieren lässt. Diese Eigenschaft besitzt gerade die *Lagrange-Basis*, die durch

$$\varphi_i(j) = \begin{cases} 1 & \text{falls } j = i, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } j \in \mathcal{N}$$

definiert ist: Die Basisfunktion φ_i nimmt in dem Eckpunkt i den Wert eins an und ist in allen anderen gleich null. Insbesondere gilt $\varphi_i|_T = 0$ für alle Dreiecke $T \in \mathcal{T}$ mit $i \notin T$, denn φ_i muss auf den drei Ecken von T gleich null sein, also auch auf den Kanten und im Inneren. Die Menge der Dreiecke, auf denen φ_i ungleich null ist, nennt man den *Träger* der Basisfunktion. In der Abbildung 7.3 ist eine Basisfunktion dargestellt. Aufgrund ihrer typischen Form wird sie oft als *Hutfunktion* oder *Zeltfunktion* (engl. *tent function*) bezeichnet. Da die Ecken $\mathcal{N}$ der Triangulierung oft auch als *Knoten* des Gitters bezeichnet werden, heißt $(\varphi_i)_{i\in\mathcal{N}}$ *Knotenbasis*.

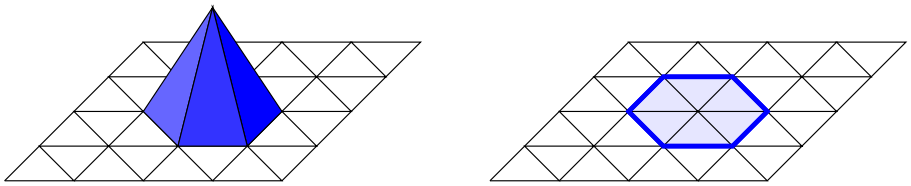

Abbildung 7.3: Hutfunktion (links) und ihr Träger (rechts)

Diese Funktionen sind zwar stückweise linear und stetig, allerdings erfüllen nicht alle von ihnen die Randbedingung: Für $i \in \mathcal{N} \cap \partial\Omega$ erhalten wir offensichtlich eine Basisfunktion φ_i, die auf dem Rand des Gebiets ungleich null ist. Wir können diese Funktionen

einfach ausschließen, indem wir uns auf die Indexmenge

$$\mathcal{I} := \{i \in \mathcal{N} \ : \ i \notin \partial\Omega\}$$

beschränken und haben eine Basis $(\varphi_i)_{i\in\mathcal{I}}$ des gewünschten Raums V_h konstruiert.

Zuordnung lokaler Basisfunktionen. Um den Lastvektor und die Steifigkeitsmatrix assemblieren zu können, benötigen wir auch lokale Basisfunktionen auf dem Referenzdreieck $\widehat{T}$ sowie eine Möglichkeit, diesen Funktionen mit den Indexabbildungen $(\iota_T)_{T\in\mathcal{T}}$ globale Basisfunktionen zuordnen zu können. Es bietet sich an, die lokalen Basisfunktionen so zu wählen, dass sie zu den Eckpunkten

$$\hat{p}_0 := \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \qquad \hat{p}_1 := \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad \hat{p}_2 := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

des Referenzdreiecks passen:

$$\hat{\varphi}_0(x) = 1 - x_1 - x_2, \qquad \hat{\varphi}_1(x) = x_1, \qquad \hat{\varphi}_2(x) = x_2 \qquad \text{für alle } x \in \widehat{T}$$

sind drei lineare Polynome, die die Lagrange-Eigenschaft

$$\hat{\varphi}_k(\hat{p}_\ell) = \begin{cases} 1 & \text{falls } k = \ell, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } k, \ell \in \widehat{\mathcal{I}} := \{0, 1, 2\}$$

besitzen. Diese Eigenschaft „vererbt“ sich auf die Funktionen $\varphi_{T,k}$, aus denen wir die globalen Basisfunktionen zusammensetzen.

Indexzuordnung. Für ein Dreieck $T \in \mathcal{T}$ bildet die Transformation Φ_T das Referenzdreieck $\widehat{T}$ auf das Dreieck T ab, und es werden insbesondere die Eckpunkte aufeinander abgebildet. Diese Eigenschaft können wir ausnutzen, um die Indexabbildung ι_T zu definieren: Die lokale Basisfunktion $\hat{\varphi}_k$ ist in der Ecke $\hat{p}_k$ des Dreiecks $\widehat{T}$ gleich eins und in allen anderen gleich null, also ist $\varphi_{T,k} = \hat{\varphi}_k \circ \Phi_T^{-1}$ gerade in der Ecke $\Phi_T(\hat{p}_k)$ des Dreiecks T gleich eins und in allen anderen gleich null. Damit ist $\varphi_{T,k}$ die Einschränkung der globalen Basisfunktion φ_i mit $i = \Phi_T(\hat{p}_k)$, also sollten wir die Indexabbildungen durch

$$\iota_T(k) = \Phi_T(\hat{p}_k) \qquad \text{für alle } T \in \mathcal{T},\ k \in \widehat{\mathcal{I}}$$

definieren. Mit ihrer Hilfe können wir den Elementlastvektoren und Elementsteifigkeitsmatrizen die richtigen Koeffizienten des globalen Lastvektors und der Steifigkeitsmatrix zuordnen.

Elementlastvektor und Elementsteifigkeitsmatrix. Sei $T \in \mathcal{T}$. Den durch

$$\hat{b}_{T,k} := |\det D\Phi_T| \int_{\widehat{T}} \hat{\varphi}_k(\hat{x}) f(\Phi_T(\hat{x}))\, d\hat{x} \qquad \text{für alle } k \in \widehat{\mathcal{I}}$$

definierten Elementlastvektor können wir beispielsweise mit der Kantenregel approximieren. $\hat{\varphi}_k$ nimmt jeweils in zwei Kantenmittelpunkten den Wert $1/2$ an und ist in dem dritten gleich null, so dass sich

$$\begin{aligned}
\hat{b}_{T,0} &\approx \frac{|\det D\Phi_T|}{12}(f(\Phi_T(1/2,0)) + f(\Phi_T(0,1/2))),\\
\hat{b}_{T,1} &\approx \frac{|\det D\Phi_T|}{12}(f(\Phi_T(1/2,1/2)) + f(\Phi_T(1/2,0))),\\
\hat{b}_{T,2} &\approx \frac{|\det D\Phi_T|}{12}(f(\Phi_T(0,1/2)) + f(\Phi_T(1/2,1/2)))
\end{aligned}$$

ergibt. Bei eine praktischen Implementierung bietet es sich selbstverständlich an, die Determinante nur einmal zu berechnen und die Werte der Funktion f in den drei Kantenmittelpunkten des Dreiecks T nur einmal auszuwerten.

Um die Elementsteifigkeitsmatrix zu berechnen, benötigen wir die Gradienten der transformierten Basisfunktionen $\varphi_{T,k}$, die durch

$$\nabla\varphi_{T,k} = (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k \circ \Phi_T^{-1} \qquad \text{für alle } k \in \widehat{\mathcal{I}}$$

gegeben sind. Da $\hat{\varphi}_k$ ein lineares Polynom ist, ist der Gradient konstant, es gilt nämlich

$$\nabla\hat{\varphi}_0 = \begin{pmatrix}-1\\-1\end{pmatrix}, \qquad \nabla\hat{\varphi}_1 = \begin{pmatrix}1\\0\end{pmatrix}, \qquad \nabla\hat{\varphi}_2 = \begin{pmatrix}0\\1\end{pmatrix}.$$

Da Φ_T affin ist, ist $D\Phi_T$ ebenfalls konstant, also auch die transformierten Gradienten $\nabla\varphi_{T,k}$.

Damit können wir die Formel für die Berechnung der Einträge der Elementsteifigkeitsmatrix erheblich vereinfachen: Es gilt

$$\begin{aligned}
\hat{a}_{T,k\ell} &= |\det D\Phi_T| \int_{\widehat{T}} \langle (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k(\hat{x}), E(\Phi_T(\hat{x}))(D\Phi_T^*)^{-1}\nabla\hat{\varphi}_\ell(\hat{x})\rangle_2 \, d\hat{x}\\
&= |\det D\Phi_T| \left\langle (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k, \int_{\widehat{T}} E(\Phi_T(\hat{x}))\, d\hat{x} (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_\ell \right\rangle_2 \qquad \text{für alle } k,\ell \in \widehat{\mathcal{I}}.
\end{aligned}$$

Mit dem (matrixwertigen) Integral

$$E_T := |\det D\Phi_T| \int_{\widehat{T}} E(\varphi_T(\hat{x}))\, d\hat{x} = \int_T E(x)\, dx$$

erhalten wir

$$\hat{a}_{T,k\ell} = \langle (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k(\hat{x}), E_T(D\Phi_T^*)^{-1}\nabla\hat{\varphi}_\ell(\hat{x})\rangle_2 \qquad \text{für alle } k,\ell \in \widehat{\mathcal{I}},$$

benötigen also keine Quadraturformel.

In der Praxis wird oft angenommen, dass E auf jedem Dreieck konstant ist oder dass es sich gut durch den Wert im Mittelpunkt des Dreiecks approximieren lässt, so dass E_T mit der Schwerpunktregel angenähert werden kann.

In einer Implementierung wird man die Gradienten $\varphi_{T,k} = (D\Phi_T^*)^{-1}\nabla\hat{\varphi}_k$ jeweils nur einmal berechnen. Da E in jedem Punkt als selbstadjungiert vorausgesetzt ist, besitzt auch E_T diese Eigenschaft, also ist die Elementsteifigkeitsmatrix ebenfalls selbstadjungiert. Damit genügt es, nur die obere oder untere Dreieckshälfte der Matrix zu berechnen und die andere durch Spiegelung an der Diagonale zu gewinnen.

Bemerkung 7.9 (Fehlersuche) *Für die Fehlersuche ist es oft nützlich, auszunutzen, dass* $\hat{\varphi}_0+\hat{\varphi}_1+\hat{\varphi}_2 = 1$ *gilt, denn daraus folgen* $\varphi_{T,0}+\varphi_{T,1}+\varphi_{T,2} = 1$ *und* $\nabla\varphi_{T,0}+\nabla\varphi_{T,1}+\nabla\varphi_{T,2} = 0$*. Eine Konsequenz besteht darin, dass die Zeilensummen und Spaltensummen der Elementsteifigkeitsmatrix gleich null sein müssen. Eine weitere darin, dass für* $f = 1$ *die Summe der Koeffizienten des Elementlastvektors den Flächeninhalt des Dreiecks ergeben sollte.*

Randpunkte. Falls $i \in \mathcal{N}$ ein Randpunkt ist, falls also $i \in \partial\Omega$ gilt, ist ihm keine Basisfunktion zugeordnet. Falls $T \in \mathcal{T}$ ein Dreieck ist, das i als Eckpunkt aufweist, wäre nach unserer bisherigen Notation eine *lokale* Basisfunktion $\varphi_{T,k}$ definiert, der aber keine *globale* Basisfunktion φ_i entspricht. Um derartige Probleme zu vermeiden, bietet es sich an, die Bildmenge der Indexabbildungen $\iota_T : \widehat{\mathcal{I}} \to \mathcal{I}$ um die Randpunkte $\partial\mathcal{I} := \mathcal{N} \setminus \mathcal{I}$ zu erweitern und die Konvention festzulegen, dass $\iota_T(k) \notin \mathcal{I}$ bedeutet, dass der lokalen Basisfunktion $\varphi_{T,k}$ keine globale Basisfunktion zugeordnet ist und wir entsprechende Werte in Elementlastvektor und Elementsteifigkeitsmatrix ignorieren können. Wir würden also

$$\iota_T : \widehat{\mathcal{I}} \to \mathcal{N} \qquad \text{für alle } T \in \mathcal{T}$$

vorgeben und bei der Assemblierung des Lastvektors in Abbildung 7.1 die letzte Schleife durch

for $k \in \widehat{\mathcal{I}}$ **do**
  if $\iota_T(k) \in \mathcal{I}$ **then begin**
    $i \leftarrow \iota_T(k)$; $b_i \leftarrow b_i + \hat{b}_{T,k}$
  end

ersetzen. Bei der Assemblierung der Steifigkeitsmatrix in Abbildung 7.2 verwenden wir entsprechend

for $k, \ell \in \widehat{\mathcal{I}}$ **do**
  if $\iota_T(k) \in \mathcal{I}$ und $\iota_T(\ell) \in \mathcal{I}$ **then begin**
    $i \leftarrow \iota_T(k)$; $j \leftarrow \iota_T(\ell)$; $a_{ij} \leftarrow a_{ij} + \hat{a}_{T,k\ell}$
  end

in der letzten Schleife, um die Einträge der Steifigkeitsmatrix zu aktualisieren.

Inhomogene Randwerte. Bisher sind wir von der Randbedingung $u|_{\partial\Omega} = 0$ ausgegangen, also von der *homogenen Dirichlet-Randbedingung*. Allgemeiner können wir auch die *inhomogene Dirichlet-Randbedingung*

$$u|_{\partial\Omega} = g|_{\partial\Omega}$$

mit einer Funktion $g \in H^1(\Omega)$ untersuchen. Um weiterhin nur ein Problem mit homogener Randbedingung lösen zu müssen, schreiben wir

$$u = u_0 + g$$

mit einer Funktion $u_0 \in V = H_0^1(\Omega)$. Offenbar besitzt u dann die korrekten Randwerte, und die Variationsgleichung nimmt die Form

$$a(v, u_0) + a(v, g) = a(v, u) = \lambda(v) \qquad \text{für alle } v \in V$$

an. Indem wir $a(v, g)$ auf die rechte Seite übertragen, ergibt sich die folgende Variationsaufgabe:

Finde $u_0 \in V = H_0^1(\Omega)$ mit

$$a(v, u_0) = \lambda(v) - a(v, g) \qquad \text{für alle } v \in V.$$

In der diskretisierten Fassung der Variationsaufgabe können wir g durch eine stückweise lineare Funktion $g_h \in H^1(\Omega)$ ersetzen, die durch

$$g_h(i) = \begin{cases} g(i) & \text{falls } i \in \partial\Omega, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in \mathcal{N}$$

definiert ist. Von der i-ten Komponente des Lastvektors wird dann die Größe $a(\varphi_i, g_h)$ subtrahiert, die sich mit Hilfe der bei der Assemblierung der Steifigkeitsmatrix aussortierten Einträgen der Elementsteifigkeitsmatrix einfach berechnen lässt: Wir bezeichnen mit $\partial\mathcal{I} := \mathcal{N} \cap \partial\Omega$ die Menge der Ecken der Triangulierung, die auf dem Rand des Gebiets liegen und definieren eine neue Matrix $R \in \mathbb{R}^{\mathcal{I} \times \partial\mathcal{I}}$ durch

$$r_{ij} := \sum_{\substack{T \in \mathcal{T} \\ \iota_T(k) = i, \iota_T(\ell) = j \\ \text{für } k,\ell \in \widehat{\mathcal{I}}}} \hat{a}_{T,k\ell} \qquad \text{für alle } i \in \mathcal{I},\ j \in \partial\mathcal{I}.$$

Der durch

$$d_i := g(i) \qquad \text{für alle } i \in \partial\mathcal{I}$$

definierte Vektor $d \in \mathbb{R}^{\partial\mathcal{I}}$ kann dann in die rechte Seite des Gleichungssystems einfließen: Wir lösen

$$Ax = b - Rd,$$

um die Koeffizienten der Lösung u_h in den Punkten der Menge $\mathcal{I}$ zu erhalten (denn g_h ist in ihnen nach Definition gleich null). Die Koeffizienten auf dem Rand $\partial\mathcal{I}$ finden sich in dem Vektor d.

7.7 Schwachbesetzte Matrizen

Im Fall der Finite-Differenzen-Verfahren lässt sich die Multiplikation eines Vektors mit einer Matrix häufig sehr effizient berechnen: Da $(Ax)_i$ eine Approximation eines Differentialoperators in einem bestimmten Punkt darstellt, hängt die Größe nur von wenigen Nachbarpunkten ab und kann mit relativ wenigen Rechenoperationen berechnet werden.

Die Steifigkeitsmatrix eines typischen Finite-Elemente-Verfahrens besitzt eine ähnliche Eigenschaft: Für $i, j \in \mathcal{I}$ ist ein Eintrag durch

$$a_{ij} = \sum_{\substack{T \in \mathcal{T} \\ \iota_T(k)=i, \iota_T(\ell)=j \\ \text{für } k,\ell \in \widehat{\mathcal{I}}}} \hat{a}_{T,k\ell}$$

gegeben. Demzufolge kann a_{ij} nur dann ungleich null sein, wenn mindestens ein Dreieck existiert, in dem je ein lokaler Freiheitsgrad den globalen Freiheitsgraden i und j zugeordnet ist.

Kanten pro Ecke. Im Spezialfall der stetigen stückweise linearen Basisfunktionen entsprechen die lokalen Freiheitsgrade den Eckpunkten eines Dreiecks, also kann a_{ij} nur dann ungleich null sein, wenn mindestens ein Dreieck existiert, das *gleichzeitig* i und j als Eckpunkte besitzt. Geometrisch entspricht das gerade der Eigenschaft, dass die Eckpunkte i und j identisch oder durch eine Kante der Triangulation verbunden sind.

In der Regel wird bei der Konstruktion einer Triangulierung $\mathcal{T}$ eines Gebiets Ω darauf geachtet, dass die Innenwinkel der beteiligten Dreiecke nicht zu klein werden, denn das hätte zur Folge, dass die Approximationseigenschaften des auf der Triangulierung konstruierten stückweise polynomiellen Ansatzraums sich verschlechtern. Als willkommener Nebeneffekt folgt auch, dass an einer Ecke des Gitters nur eine kleine Zahl von Kanten aufeinander stoßen können: Falls der kleinste Innenwinkel aller Dreiecke der Triangulierung $\alpha_0 \in (0, \pi)$ beträgt, können an einem beliebigen Eckpunkt höchstens $2\pi/\alpha_0$ Dreiecke aufeinanderstoßen, also kann der Eckpunkt von höchstens $2\pi/\alpha_0 + 1$ Kanten getroffen werden. Für ein festes $i \in \mathcal{I}$ können demzufolge neben $j = i$ höchstens $2\pi/\alpha_0 + 1$ weitere Indizes $j \in \mathcal{I}$ mit $a_{ij} \neq 0$ existieren.

Besetztheitsmuster. Bei der Multiplikation der Matrix A können wir diese Eigenschaft ausnutzen, indem wir gezielt nur die von null verschiedenen Matrixeinträge berücksichtigen. Die Information darüber, welche Einträge wir ignorieren können, beschreiben wir mit Hilfe einer geeigneten Menge:

Definition 7.10 (Graph einer Matrix) *Seien $\mathcal{I}$ und $\mathcal{J}$ endliche Indexmengen, und sei $A \in \mathbb{R}^{\mathcal{I} \times \mathcal{J}}$. Eine Menge $\mathcal{G} \subseteq \mathcal{I} \times \mathcal{J}$ nennen wir ein* Besetztheitsmuster *der Matrix A, falls*

$$a_{ij} \neq 0 \Rightarrow (i, j) \in \mathcal{G} \qquad \text{für alle } i \in \mathcal{I},\ j \in \mathcal{J}$$

gilt. Das durch

$$\mathcal{G}_A := \{(i,j) \in \mathcal{I} \times \mathcal{J} \ : \ a_{ij} \neq 0\}$$

definierte minimale Besetztheitsmuster nennen wir den Graphen *der Matrix A.*

Bei der Auswertung des Matrix-Vektor-Produkts Ax müssen wir lediglich die Koeffizienten berücksichtigen, die in einem Besetztheitsmuster $\mathcal{G}$ der Matrix A auftreten:

$$(Ax)_i = \sum_{j \in \mathcal{I}} a_{ij} x_j = \sum_{\substack{j \in \mathcal{I} \\ a_{ij} \neq 0}} a_{ij} x_j = \sum_{\substack{j \in \mathcal{I} \\ (i,j) \in \mathcal{G}}} a_{ij} x_j \qquad \text{für alle } i \in \mathcal{I}.$$

Offensichtlich ist die Auswertung um so effizienter, je weniger Elemente das Besetztheitsmuster entält.

Definition 7.11 (Schwachbesetzt) *Ein Besetztheitsmuster $\mathcal{G}$ einer Matrix $A \in \mathbb{R}^{\mathcal{I} \times \mathcal{J}}$ nennen wir* C_{sp}-schwachbesetzt *für ein $C_{\mathrm{sp}} \in \mathbb{N}$, falls*

$$\#\mathcal{G} \leq C_{\mathrm{sp}} \# \mathcal{I}$$

gilt. Wir nennen es streng C_{sp}-schwachbesetzt*, falls*

$$\begin{aligned} \{j \in \mathcal{J} \ : \ (i,j) \in \mathcal{G}\} &\leq C_{\mathrm{sp}} && \text{für alle } i \in \mathcal{I}, \\ \{i \in \mathcal{I} \ : \ (i,j) \in \mathcal{G}\} &\leq C_{\mathrm{sp}} && \text{für alle } j \in \mathcal{J} \end{aligned}$$

gelten. Offenbar ist jedes streng schwachbesetzte Muster auch schwachbesetzt.

Die Matrix A nennen wir C_{sp}-schwachbesetzt, falls ihr Graph $\mathcal{G}_A$ C_{sp}-schwachbesetzt ist. Entsprechendes gilt für den streng schwachbesetzten Fall.

Im Fall der Hutfunktionen bietet es sich an, das Besetztheitsmuster

$$\mathcal{G} = \{(i,j) \in \mathcal{I} \times \mathcal{I} \ : \ \text{es existiert ein Dreieck } T \in \mathcal{T} \text{ mit } i,j \in T\}$$

auf Grundlage der Dreiecke zu verwenden, das sich auch durch

$$\mathcal{G} = \{(i,j) \in \mathcal{I} \times \mathcal{I} \ : \ \text{es existiert eine Kante } e \in \mathcal{E} \text{ mit } i,j \in e\}$$

auf Grundlage der Kanten definieren lässt. Falls die Innenwinkel aller Dreiecke der Triangulierung nicht kleiner als $\alpha_0 > 0$ sind, folgt aus unseren Überlegungen dann, dass $\mathcal{G}$ C_{sp}-schwachbesetzt ist mit

$$C_{\mathrm{sp}} := \left\lfloor \frac{2\pi}{\alpha_0} + 2 \right\rfloor.$$

Datenstruktur. Es gibt verschiedene Datenstrukturen, mit denen sich schwachbesetzte Matrizen darstellen lassen. Besonders einfach ist es beispielsweise, alle Indexpaare $(i, j) \in \mathcal{G}$ eines Besetztheitsmusters in zwei Arrays `row` und `col` der Länge $n_{\text{nz}} := \#\mathcal{G}$ aufzulisten. In einem weiteren Array `coeff` derselben Länge kann dann zu jedem Indexpaar `i=row[k],j=col[k]` der Koeffizient a_{ij} in `coeff[k]` gespeichert werden.

Diese Struktur hat den Nachteil, dass wir das Ergebnis der wichtigen Matrix-Vektor-Multiplikation nicht komponentenweise berechnen können, stattdessen müssten wir einen Algorithmus der Form

```
for(k=0; k<nnz; k++)
  y[row[k]] += coeff[k] * x[col[k]];
```

verwenden, der möglicherweise völlig ungeordnet die Einträge der Vektoren x und y liest und ebenso ungeordnet die Einträge des Vektors y wieder schreibt.

Wesentlich eleganter ist die CSR-Darstellung (engl. *compressed sparse row*), bei der die Einträge so sortiert sind, dass die Einträge einer Zeile aufeinander folgen. Dank dieser Sortierung ist es nicht erforderlich, die Zeilenindizes abzuspeichern, wir müssen lediglich notieren, wo in der Liste der Koeffizienten und Spaltenindizes eine Zeile beginnt.

In der CSR-Darstellung verwenden wir wieder ein Array `coeff`, das für jedes Element $(i, j) \in \mathcal{G}$ des Besetztheitsmusters einen Koeffizienten a_{ij} aufnimmt und ein Array `col`, das den entsprechendne Spaltenindex enthält. Für $n := \#\mathcal{I}$ enthält das Array `row` enthält nun $n + 1$ Einträge: `row[i]` bezeichnet den ersten Index in den beiden anderen Arrays, der zu der Zeile i gehört. Der letzte Index dieses Arrays entspricht der Anzahl n_{nz} der von null verschiedenen Einträge des Besetztheitsmusters. Die Matrix

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 3 & -1 & & \\ & -1 & 4 & -1 & \\ & -2 & & 5 & -1 \\ & & & -1 & 6 \end{pmatrix}$$

könnte beispielsweise in der in Abbildung 7.4 angegebenen Weise dargestellt werden.

Konstruktion des Besetztheitsmusters. Um eine Matrix in der CSR-Darstellung speichern zu können, müssen wir die Arrays `row` und `col` so konstruieren, dass sie zu der Triangulierung und dem verwendeten Ansatzraum passen. In unserem Spezialfall könnte man versuchen, die Anzahl der Kanten, die an einer Ecke zusammentreffen $i \in \mathcal{I}$, zu zählen, um so die Anzahl der von Null verschiedenen Einträge in der entsprechenden Zeile zu ermitteln. Für allgemeinere Anwendungen bietet es sich an, die beiden Arrays auf einem Umweg zu berechnen: Wir verwenden eine Hilfsstruktur `sparsepattern`, die für jeden Zeilenindex eine Liste von Spaltenindizes verwaltet, zu der wir mit einer Funktion `addnz_sparsepattern` Einträge hinzufügen können. Falls `sp` ein Zeiger auf ein `sparsepattern`-Objekt ist, lässt sich mit

```
addnz_sparsepattern(sp, i, j);
```

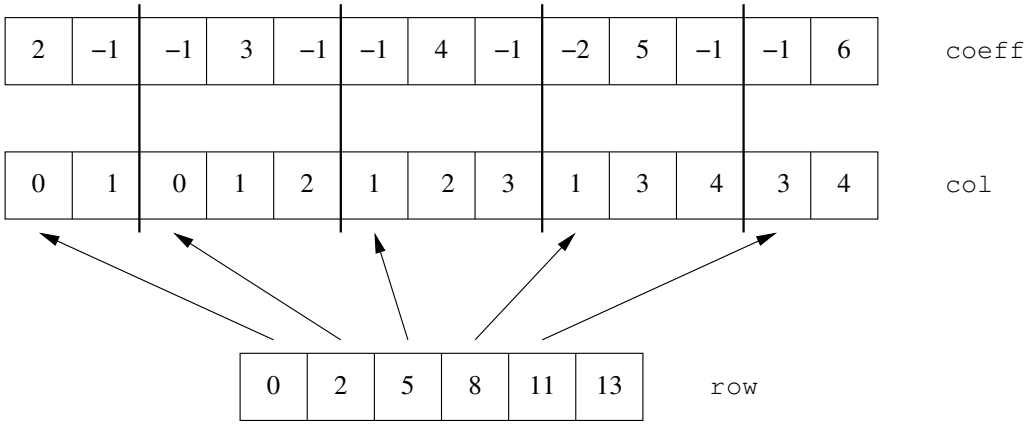

Abbildung 7.4: CSR-Darstellung einer schwachbesetzten Matrix

in der Liste für den Zeilenindex `i` der Spaltenindex `j` hinzufügen, falls er nicht schon vorhanden ist.

Sobald die Listen des `sparsepattern`-Objekts gefüllt sind, können wir sie einfach durchzählen und die Arrays für die CSR-Darstellung der Steifigkeitsmatrix konstruieren.

Numerierung der Indizes. In unserem Anwendungsfall ist nicht jeder Ecke der Triangulierung eine Basisfunktion zugeordnet, denn Ecken auf dem Rand haben wir herausgenommen, um die Randbedingungen einzubeziehen. Da in unserem Gleichungssystem nur fortlaufend numerierte Unbekannte auftreten sollen, bietet es sich an, mit einem zusätzlichen Array `idx2dof` die Zuordnung von Indizes im Gitter zu Freiheitsgraden im Gleichungssystem (engl. *degree of freedom*) zu beschreiben. Am einfachsten geschieht das, indem wir ein Array `is_dof` verwenden, das für jede Ecke angibt, ob in ihr ein Freiheitsgrad liegen soll oder nicht. Falls die für die Triangulierung verwendete Datenstruktur auch Informationen darüber enthält, ob eine Ecke oder eine Kante auf dem Rand des Gebiets liegen, lässt sich das Array `is_dof` leicht füllen. Das Array `idx2dof` können wir dann durch einfaches Abzählen konstruieren:

```
n = 0;
for(i=0; i<vertices; i++)
  if(is_dof[i]) {
    idx2dof[i] = n;
    n++;
  }
```

Besetztheitsmuster für Hutfunktionen. Mit Hilfe der Arrays `is_dof` und `idx2dof` können wir das Besetztheitsmuster berechnen: Wir müssen mit einem von null verschiedenen Eintrag rechnen, falls ein Dreieck existiert, das zwei Eckpunkte verbindet, die

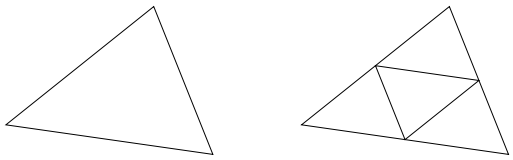

Abbildung 7.5: Rote Verfeinerung eines Dreiecks

beide keine Randpunkte sind. Letzteres lässt sich mit `is_dof` feststellen, die Nummern der Eckpunkte erhalten wir gegebenenfalls aus `idx2dof`:

```
for(i=0; i<triangles; i++) {
  getvertices_tri2d(t2, i, t);
  for(j=0; j<3; j++)
    for(k=0; k<3; k++)
      if(is_dof[t[j]] && is_dof[t[k]])
        addnz_sparsepattern(sp, idx2dof[t[j]], idx2dof[t[k]]);
}
```

Hier soll die Funktion `getvertices_tri2d` aus einer Triangulierung `t2` die Ecken des `i`-ten Dreiecks bestimmen und in das dreielementige Array `t` schreiben. Diese Funktion ist erforderlich, weil in unserer Datenstruktur ein Dreieck nur auf seine Kanten direkt zugreifen kann.

Eine ähnliche Schleifenstruktur lässt sich verwenden, um im Zuge der Assemblierung die Einträge der Elementsteifigkeitsmatrix zu denen der entstehenden Steifigkeitsmatrix zu addieren: Auch hier muss geprüft werden, ob die Eckpunkte zu Freiheitsgraden gehören und, falls ja, welche Nummer diese Freiheitsgrade tragen.

7.8 Gitterverfeinerung

Sowohl der bei der Berechnung des Lastvektors auftretende Quadraturfehler als auch der Diskretisierungsfehler $\|u - u_h\|_V$ hängen entscheidend von der Auflösung der verwendeten Triangulierung ab: Je kleiner die Dreiecke sind, desto besser ist in der Regel die Approximation.

Deshalb sind Verfahren von Interesse, mit denen sich eine Triangulierung *verfeinern* lässt, darunter versteht man die Konstruktion einer neuen Triangulierung $\mathcal{T}'$, die durch Unterteilung der Dreiecke der ursprünglichen Triangulierung $\mathcal{T}$ entstanden ist.

Rote Verfeinerung. Besonders einfach ist die sogenannte „rote Verfeinerung“, bei der ein Dreieck durch Verbinden der drei Kantenmittelpunkte in vier kongruente (d.h. durch Verschiebung, Rotation und Skalierung ineinander überführbare) Dreiecke zerlegt wird (siehe Abbildung 7.5).

Diese relativ einfach umsetzbare Technik bietet den großen Vorteil, dass die Form der Dreiecke der Triangulierung erhalten bleibt und sich die Kongruenz zwischen dem „Vaterdreieck“ und seinen Söhnen ausnutzen lässt, um beispielsweise die Matrizen $(D\Phi_T^*)^{-1}$ sehr einfach zu berechnen, die wir für die Berechnung der Elementsteifigkeitsmatrizen benötigen.

Im Rahmen unserer Datenstruktur müssen wir darauf achten, dass Eckpunkte und Kanten korrekt von benachbarten Dreiecken geteilt werden: Falls bei der Unterteilung eines Dreiecks ein neuer Eckpunkt im Mittelpunkt einer Kante angelegt wurde, muss sichergestellt sein, dass ein Dreieck auf der anderen Seite der Kante denselben Eckpunkt verwendet. Würde es nämlich ebenfalls einen neuen Eckpunkt anlegen, wäre der Kantenmittelpunkt in der neuen Triangulierung doppelt vertreten und die Stetigkeit der Basisfunktionen nicht mehr gesichert.

Diese Aufgabe lässt sich relativ einfach lösen, indem wir den Kantenmittelpunkt der Kante zuordnen, statt dem Dreieck, das ihn für seine Unterteilung benötigt. Entsprechend können wir mit den Kanten verfahren: Bei der Zerlegung einer Kante entstehen zwei neue Kanten, die wir dieser Kante zuordnen. Hinzu kommen für jedes Dreieck drei Kanten, die die Kantenmittelpunkte verbinden. Der Algorithmus kann also in vier Phasen ablaufen:

1. Konstruiere für jede Kante einen Kantenmittelpunkt.
2. Konstruiere für jede Kante zwei Kanten, die ihre beiden Eckpunkte jeweils mit dem Kantenmittelpunkt verbinden.
3. Konstruiere für jedes Dreieck drei Kanten, die ihre Kantenmittelpunkte verbinden.
4. Konstruiere für jedes Dreieck vier Teildreiecke.

Da in unserer Datenstruktur die Ecken, Kanten und Dreiecke durchnumeriert sind, können wir die Zuordnung zu Kanten und Dreiecken elegant mit Hilfe von Arrays beschreiben.

Da wir wissen, wieviele Ecken, Kanten und Dreiecke die verfeinerte Triangulierung besitzen wird, können wir die sie beschreibende Datenstruktur ohne größere Schwierigkeiten vor Beginn der eigentlichen Konstruktion anlegen und die neuen Ecken, Kanten und Dreiecke direkt hineinschreiben.

Lokale Verfeinerung. Falls die Lösung u des Variationsproblems sich in Teilen des Gebiets sehr schnell verändert, während sie in anderen Teilen „glatt“ ist, empfiehlt es sich, die Triangulierung *lokal* zu verfeinern, also nur die Dreiecke zu zerlegen, die in dem interessanten Bereich liegen.

Die rote Verfeinerung eignet sich zu diesem Zweck nur sehr bedingt: Wenn wir ein einzelnes Dreieck zerlegen, entstehen neue Ecken in seinen Kantenmittelpunkten, denen keine Ecken in den benachbarten Dreiecken entsprechen. Diese Ecken bezeichnet man als *hängende Knoten*, sie verletzen die Bedingungen der Definition 7.7 der Triangulierung und würden zu unstetigen Basisfunktionen führen. Um hängende Knoten zu vermeiden,

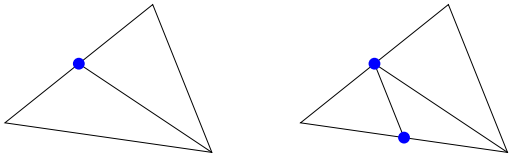

Abbildung 7.6: Grün/blauer Abschluss eines Dreiecks, falls nur eine oder nur zwei Kanten zerlegt werden sollen

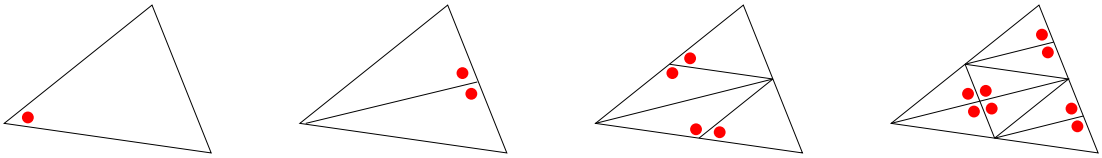

Abbildung 7.7: *Newest vertex bisection* angewendet auf ein Dreieck und die dabei entstehenden Teildreiecke. Der jeweils neueste Eckpunkt in jedem Dreieck ist markiert.

bliebe uns bei der roten Verfeinerung nur die Möglichkeit, auch die Nachbardreiecke zu zerlegen, und dieser Effekt würde sich fortsetzen, bis die gesamte Triangulierung verfeinert ist.

Grüner/blauer Abschluss. Einen unschönen Ausweg bieten die *grüne und blaue Verfeinerung*: Falls in einem Dreieck, das nicht verfeinert werden soll, ein Kantenmittelpunkt Eckpunkt eines verfeinerten Nachbardreiecks ist, wird er mit dem ihm gegenüberliegenden Punkt verbunden, um sicherzustellen, dass eine wohldefinierte Triangulierung entsteht. Falls in einem Dreieck, das nicht verfeinert werden soll, sogar zwei Kantenmittelpunkte Eckpunkte zweier verfeinerter Nachbardreiecke sind, wird der erste mit dem ihm gegenüber liegenden Eckpunkt verbunden, der zweite mit dem ihm in dem so entstandenen kleineren Dreieck gegenüber liegenden.

Da die grüne und blaue Verfeinerung lediglich die Konsistenz der Triangulierung herstellt, nachdem die interessanten Dreiecke rot verfeinert wurden, spricht man bei dieser Technik von dem „grün/blauen Abschluss“ der Triangulierung.

Durch den Abschluss (siehe Abbildung 7.6) erhalten wir zwar eine lokal verfeinerte Triangulierung, bei wiederholter Anwendung können allerdings die Innenwinkel der Dreiecke beliebig klein werden und die Approximationsgüte der Diskretisierung gefährden.

Bisektion. Das Problem der schrumpfenden Innenwinkel lässt sich mit Algorithmen verhindern, die von Anfang an für die lokale Gitterverfeinerung entwickelt wurden. Besonders einfach umzusetzen sind dabei Bisektionsverfahren, die die Dreieckskanten so unterteilen, dass einerseits eine lokale Verfeinerung möglich ist und andererseits die Innenwinkel nicht zu klein werden.

Als Beispiel untersuchen wir die Strategie „*newest vertex bisection*“, bei der wir uns in jedem Dreieck merken, welcher ihrer Eckpunkte als letzter entstanden ist. Unterteilt wird dann jeweils die Kante, die ihm gegenüber liegt. Zwar entstehen bei diesem Ansatz zunächst Dreiecke, die nicht zueinander kongruent sind und deren Innenwinkel kleiner geworden sind, allerdings verschlechtert sich die Lage bei wiederholten Verfeinerungsschritten nicht weiter (siehe Abbildung 7.7), weil entstehende Dreiecke kongruent zu ihren Vorgängern sind.

Elimination hängender Knoten. Wenn wir diese Strategie anwenden, um eine lokale Verfeinerung zu erreichen, werden in der Regel hängende Knoten auftreten, die wir geeignet behandeln müssen. Ein einfacher Zugang besteht darin, in mehreren Generationen vorzugehen: In der ersten Generation werden alle Dreiecke unterteilt, in denen wir die Auflösung steigern wollen. Dabei können hängende Knoten in den Kantenmittelpunkten der Dreiecke entstehen. In der zweiten Generation werden alle Dreiecke der ersten Generation unterteilt, bei denen einer der Kantenmittelpunkte einen hängenden Knoten enthält. Es ist nicht garantiert, dass dadurch die hängenden Knoten eliminiert werden, es können sogar weitere hängede Knoten entstehen. In der dritten Generation werden alle Dreiecke der zweiten Generation unterteilt, bei denen einer der Kantenmittelpunkte eine hängenden Knoten enthält. Auch dabei können weitere hängende Knoten entstehen, also wiederholen wir die Prozedur.

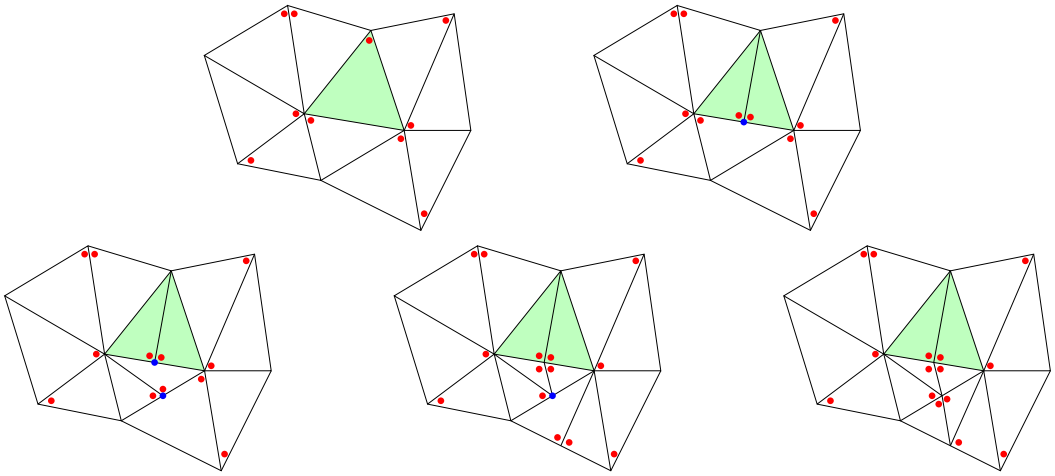

Abbildung 7.8: Folgen der lokalen Verfeinerung eines Dreiecks (grün): Um hängende Knoten (blau) zu vermeiden, müssen zwei weitere Dreiecke ebenfalls verfeinert werden.

Man kann sich überlegen, dass bei dieser Konstruktion hängende Knoten nur in den Kantenmittelpunkten der Ausgangstriangulation auftreten können. Daraus folgt, dass nach einer endlichen Anzahl von Schritten alle hängenden Knoten eliminiert sein werden. Allerdings kann bei einer ungünstigen Verteilung der „jüngsten Ecken“ in dem

Ausgangsgitter die Situation eintreten, dass alle Dreiecke verfeinert werden müssen, um eine wohldefinierte Triangulierung zu erhalten, so dass die Verfeinerung nicht mehr als „lokal“ zu bezeichnen wäre. Ein Beispiel für eine Triangulierung, bei der die Verfeinerung eines Dreiecks erst nach vier Generationen abgeschlossen ist, findet sich in Abbildung 7.8.

7.9 Fehlerschätzer

Wirklich nützlich ist eine lokale Verfeinerung der Triangulierung nur, wenn wir wissen, wo wir das Gitter verfeinern sollen. Dazu können *lokale Fehlerschätzer* zum Einsatz kommen, die jedem Dreieck eine Zahl zuordnen, die als Maß für den lokalen Fehler verwendet werden kann. Sobald diese Zahlen berechnet sind, werden die Dreiecke verfeinert, in denen man einen hohen Fehler erwartet.

Als Beispiel untersuchen wir unsere Bilinearform für den besonders einfachen Fall $E = I$. Es lässt sich zeigen, dass dann Konstanten $C_E, C_S \in \mathbb{R}_{>0}$ mit

$$\begin{aligned} |a(v,u)| &\leq C_S \|v\|_V \|u\|_V && \text{für alle } v,u \in V, \\ a(v,v) &\geq C_E \|v\|_V^2 && \text{für alle } v \in V \end{aligned}$$

existieren. Die erste Eigenschaft wird als *Stetigkeit* bezeichnet, die zweite als *Elliptizität* (im englischen Sprachraum ist der Begriff *coercive* üblich).

Residuum. Wenn $u_h \in V_h$ die Lösung des diskreten Problems

$$a(v_h, u_h) = \lambda(v_h) \qquad \text{für alle } v_h \in V_h$$

und $u \in V$ die Lösung des ursprünglichen Problems

$$a(v,u) = \lambda(v) \qquad \text{für alle } v \in V$$

ist, bezeichnen wir die Abbildung

$$r : V \to \mathbb{R}, \qquad v \mapsto a(v, u - u_h),$$

als das *Residuum* der Näherungslösung u_h. Es handelt sich um eine stetige lineare Abbildung des Raums V in den Körper, und solche Abbildungen sind so wichtig, dass sie einen eigenen Namen besitzen:

Definition 7.12 (Funktionale) *Sei V ein reeller normierter Vektorraum. Eine stetige lineare Abbildung $\mu : V \to \mathbb{R}$ nennen wir* (stetiges) Funktional. *Den Raum der stetigen Funktionale nennen wir den* Dualraum *des Raums V, er wird mit V' bezeichnet.*

Der Dualraum V' ist offenbar ein Vektorraum. Indem wir ihn mit der Operatornorm

$$\|\mu\|_{V'} := \sup \left\{ \frac{|\mu(v)|}{\|v\|_V} \;:\; v \in V \setminus \{0\} \right\} \qquad \textit{für alle } \mu \in V'$$

ausstatten, wird er zu einem Banach-Raum.

Residuum und Fehler. In unserem Fall besitzt die Operatornorm des Residuums eine nützliche Eigenschaft: Aus der Stetigkeit folgt

$$\begin{aligned}\|r\|_{V'} &= \sup\left\{\frac{|a(v,u-u_h)|}{\|v\|_V} \;:\; v\in V\setminus\{0\}\right\}\\ &\leq \sup\left\{\frac{C_S\|v\|_V\|u-u_h\|_V}{\|v\|_V} \;:\; v\in V\setminus\{0\}\right\} = C_S\|u-u_h\|_V,\end{aligned}$$

während wir aus der Elliptizität

$$\begin{aligned}\|r\|_{V'} &= \sup\left\{\frac{|a(v,u-u_h)|}{\|v\|_V} \;:\; v\in V\setminus\{0\}\right\}\\ &\geq \frac{|a(u-u_h,u-u_h)|}{\|u-u_h\|_V} \geq \frac{C_E\|u-u_h\|_V^2}{\|u-u_h\|_V} = C_E\|u-u_h\|_V\end{aligned}$$

erhalten. Dabei müssen wir für die Zwischenschritte $u \neq u_h$ voraussetzen, aber das Endergebnis gilt offenbar auch für $u = u_h$.

In diesem Sinn ist die Operatornorm des Residuums äquivalent zu der Norm des Fehlers $\|u-u_h\|_V$, unser Ziel sollte also darin bestehen, die Operatornorm abzuschätzen.

Galerkin-Orthogonalität. Dabei können wir auf eine nützliche Eigenschaft zurückgreifen: Nach Definition erfüllt $u_h \in V_h$ die Gleichung

$$a(v_h,u_h) = \lambda(v_h) \qquad \text{für alle } v_h \in V_h.$$

Die exakte Lösung u erfüllt ebenfalls

$$a(v_h,u) = \lambda(v_h) \qquad \text{für alle } v_h \in V_h \subseteq V.$$

Durch Subtraktion beider Gleichungen erhalten wir die Beziehung

$$a(v_h,u-u_h) = 0 \qquad \text{für alle } v_h \in V_h,$$

die als *Galerkin-Orthogonalität* bezeichnet wird.

Partielle Integration. Da $C_E\|u-u_h\|_V \leq \|r\|_{V'}$ gilt, können wir eine obere Schranke der Operatornorm des Residuums zur Schätzung des Diskretisierungsfehlers verwenden. Unser Ziel besteht darin, eine Schranke der Form

$$|r(v)| \leq C\|v\|_V \qquad \text{für alle } v \in V$$

mit einem geeigneten C zu finden, denn daraus folgt $\|r\|_{V'} \leq C$ und wir haben die gesuchte obere Schranke gefunden.

Sei also $v \in V$. Um später die Galerkin-Orthogonalität ausnutzen zu können, wählen wir zusätzlich ein $v_h \in V_h$ und stellen fest, dass

$$r(v_h) = a(v_h,u-u_h) = 0,$$

$$r(v) = r(v - v_h) = a(v - v_h, u - u_h)$$

gilt. Mit der Definition der Bilinearform a und der rechten Seite λ folgt

$$\begin{aligned} r(v) &= a(v - v_h, u - u_h) = a(v - v_h, u) - a(v - v_h, u_h) = \lambda(v - v_h) - a(v - v_h, u_h) \\ &= \int_\Omega (v - v_h)(x) f(x)\, dx - \int_\Omega \langle \nabla(v - v_h)(x), \nabla u_h(x) \rangle_2\, dx \\ &= \int_\Omega (v - v_h)(x) f(x)\, dx - \sum_{T \in \mathcal{T}} \int_T \langle \nabla(v - v_h)(x), \nabla u_h(x) \rangle_2\, dx. \end{aligned}$$

Da u_h ein stückweises Polynom ist, können wir das zweite Integral per partieller Integration umformen und erhalten

$$\begin{aligned} r(v) = &\int_\Omega (v - v_h)(x) f(x)\, dx \\ &+ \sum_{T \in \mathcal{T}} \int_T (v - v_h)(x) \nabla \cdot \nabla u_h(x)\, dx - \sum_{T \in \mathcal{T}} \int_{\partial T} (v - v_h)(x) \langle n_T(x), \nabla u_h(x) \rangle_2\, dx \\ = &\sum_{T \in \mathcal{T}} \int_T (v - v_h)(x)(f(x) + \nabla \cdot \nabla u_h(x))\, dx \\ &- \sum_{T \in \mathcal{T}} \int_{\partial T} (v - v_h)(x) \langle n_T(x), \nabla u_h(x) \rangle_2\, dx. \end{aligned}$$

Um ähnliche wie in der Gleichung (7.14) die Summe über alle Dreiecksränder durch eine Summer über alle Kanten ersetzen zu können, definieren wir die Terme

$$j_e(x) := \langle n_e(x), \nabla u_{T_{e,+}}(x) - \nabla u_{T_{e,-}}(x) \rangle_2 \qquad \text{für alle } e \in \mathcal{E}_\Omega,\ x \in e.$$

Hier bezeichnet $T_{e,+}$ jeweils das an $e \in \mathcal{E}$ angrenzende Dreieck, dessen äußerer Normalenvektor mit dem Normalenvektor n_e der Kante übereinstimmt, während $T_{e,-}$ dasjenige Dreieck bezeichnet, bei dem beide Normalenvektoren in entgegengesetzte Richtungen zeigen.

Anschaulich messen die Ausdrücke $j_e(u_h)$, wie stark sich die Ableitung der Funktion u_h in Normalenrichtung bei Überschreiten der Kante verändert. Man bezeichnet sie deshalb als *Sprungterme*. Da uns u_h explizit zur Verfügung steht, können wir sämtliche Sprungterme direkt berechnen. Bei stückweise linearen Basisfunktionen ist die Situation noch einfacher: ∇u_h ist stückweise konstant, also reduziert sich das Kantenintegral auf die Multiplikation des konstanten Gradienten mit dem konstanten Normalenvektor.

Da $v - v_h \in V$ auf Randkanten gleich null ist, brauchen wir nur die Sprungterme zu inneren Kanten zu berücksichtigen und erhalten die Darstellung

$$r(v) = \sum_{T \in \mathcal{T}} \int_T (v - v_h)(x)(f + \nabla \cdot \nabla u_h)(x)\, dx - \sum_{e \in \mathcal{E}_\Omega} \int_e (v - v_h)(x) j_e(u_h)\, dx$$

des Residuums. Indem wir beide Integrale per Cauchy-Schwarz-Ungleichung (7.5) abschätzen, gelangen wir zu der Gleichung

$$|r(v)| \leq \sum_{T\in\mathcal{T}} \left(\int_T (v-v_h)^2(x)\,dx\right)^{1/2} \left(\int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx\right)^{1/2} + \sum_{e\in\mathcal{E}_\Omega} \left(\int_e (v-v_h)^2(x)\,dx\right)^{1/2} \left(\int_e j_e^2(x)\,dx\right)^{1/2}.$$

Bemerkung 7.13 (Stückweise lineare Ansatzfunktionen) *Im von uns untersuchten Fall stückweise linearer Ansatzfunktionen ist $\nabla u_h|_T$ konstant und $\nabla\cdot\nabla u_h|_T$ deshalb gleich null.*

Clément-Interpolation. Jetzt ist der Punkt gekommen, an dem wir $v_h \in V_h$ so wählen können, dass unsere Abschätzung möglichst günstig wird. Üblich ist die Verwendung eines *Clément-Interpolationsoperators* $\mathfrak{I}_h : V \to V_h$, der die Ungleichungen

$$\sum_{T\in\mathcal{T}} h_T^{-2}\|v-\mathfrak{I}_h[v]\|_{L^2(T)}^2 \leq C_{\mathrm{cl}}\|v\|_V^2,$$
$$\sum_{e\in\mathcal{E}} h_e^{-1}\|v-\mathfrak{I}_h[v]\|_{L^2(e)}^2 \leq C_{\mathrm{cl}}\|v\|_V^2$$

mit einer nur von der Triangulierung abhängenden Konstanten C_{cl} erfüllt. Hier bezeichnet h_T den Durchmesser des Dreiecks $T \in \mathcal{T}$, während h_e die Länge der Kante $e \in \mathcal{E}$ angibt.

Indem wir $v_h := \mathfrak{I}_h[v]$ setzen und die Cauchy-Schwarz-Ungleichung für Summen anwenden folgt

$$\begin{aligned}
|r(v)| &\leq \sum_{T\in\mathcal{T}} \left(h_T^{-2}\int_T (v-v_h)^2(x)\,dx\right)^{1/2} \left(h_T^2\int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx\right)^{1/2} \\
&\quad + \sum_{e\in\mathcal{E}_\Omega} \left(h_e^{-1}\int_e (v-v_h)^2(x)\,dx\right)^{1/2} \left(h_e\int_e j_e^2(x)\,dx\right)^{1/2} \\
&\leq \left(\sum_{T\in\mathcal{T}} h_T^{-2}\int_T (v-v_h)^2(x)\,dx\right)^{1/2} \left(\sum_{T\in\mathcal{T}} h_T^2\int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx\right)^{1/2} \\
&\quad + \left(\sum_{e\in\mathcal{E}_\Omega} h_e^{-1}\int_e (v-v_h)^2(x)\,dx\right)^{1/2} \left(\sum_{e\in\mathcal{E}_\Omega} h_e\int_e j_e^2(x)\,dx\right)^{1/2} \\
&\leq C_{\mathrm{cl}}^{1/2}\|v\|_V \left(\sum_{T\in\mathcal{T}} h_T^2\int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx\right)^{1/2} \\
&\quad + C_{\mathrm{cl}}^{1/2}\|v\|_V \left(\sum_{e\in\mathcal{E}_\Omega} h_e\int_e j_e^2(x)\,dx\right)^{1/2}.
\end{aligned}$$

Dank der elementaren Ungleichung $\sqrt{a}+\sqrt{b} \leq \sqrt{2(a+b)}$ können wir die Summen zusammenfassen und

$$|r(v)| \leq (2C_{\mathrm{cl}})^{1/2}\|v\|_V \left(\sum_{T\in\mathcal{T}} h_T^2 \int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx + \sum_{e\in\mathcal{E}_\Omega} h_e \int_e j_e^2(x)\,dx\right)^{1/2}$$

erhalten. Diese Abschätzung haben wir für beliebige $v \in V$ bewiesen, also folgt

$$\|r\|_{V'} \leq (2C_{\mathrm{cl}})^{1/2} \left(\sum_{T\in\mathcal{T}} h_T^2 \int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx + \sum_{e\in\mathcal{E}_\Omega} h_e \int_e j_e^2(x)\,dx\right)^{1/2}.$$

Abgesehen von der Konstanten C_{cl} können wir alle Terme auf der rechten Seite dieser Gleichung explizit berechnen, da uns u_h und f zur Verfügung stehen. Dank

$$\|u-u_h\|_V \leq C_E^{-1}\|r\|_{V'}$$

mit einer Konstanten C_E, die sich in unserem Fall explizit angeben lässt, haben wir damit eine (bis auf C_{cl}) *konkret berechenbare* Schranke für den Diskretisierungsfehler erhalten.

Lokaler Fehlerschätzer. Um Dreiecke identifizieren zu können, auf denen der Fehler besonders groß ist und die deshalb verfeinert werden sollten, verteilt man die Kantenbeiträge auf die Dreiecke: Wir definieren

$$\eta_T(u_h) := \left(h_T^2 \int_T (f+\nabla\cdot\nabla u_h)^2(x)\,dx + \frac{1}{2}\sum_{\substack{e\in\mathcal{E}_\Omega\\ e\subseteq T}} h_e \int_e j_e^2(x)\,dx\right)^{1/2},$$

$$\eta(u_h) := \left(\sum_{T\in\mathcal{T}} \eta_T^2(u_h)\right)^{1/2}.$$

Da an jede innere Kante exakt zwei Dreiecke angrenzen, erhalten wir

$$\|u-u_h\|_V \leq C_E^{-1}(2C_{\mathrm{cl}})^{1/2}\eta(u_h).$$

Ein typischer Ansatz für die lokale Verfeinerung ist die *Markierungsstrategie von Dörfler*, bei der man ein $\theta \in (0,1)$ vorgibt und die Menge $\mathcal{T}_{\mathrm{ref}} \subseteq \mathcal{T}$ der zu verfeinernden Dreiecke so wählt, dass sie möglichst wenige Elemente enthält und

$$\sum_{T\in\mathcal{T}_{\mathrm{ref}}} \eta_T^2(u_h) \geq \theta^2 \sum_{T\in\mathcal{T}} \eta_T^2(u_h)$$

erfüllt. Die Menge $\mathcal{T}_{\mathrm{ref}}$ kann konstruiert werden, indem man die Dreiecksfehler $\eta_T(u_h)$ absteigend sortiert und der Reihe nach hinzu nimmt, bis die Bedingung erfüllt ist.

7.10 Mehrgitterverfahren

Da Finite-Elemente-Verfahren in der Regel zu ähnlich großen und ähnlich schlecht konditionierten linearen Gleichungssystemen wie Finite-Differenzen-Verfahren führen, empfiehlt es sich, der Frage nach effizienten Lösungsverfahren etwas Aufmerksamkeit zu widmen.

Falls die Bilinearform a elliptisch ist, folgt aus

$$\langle x, Ax\rangle_2 = \sum_{i,j\in\mathcal{I}} x_i a_{ij} x_j = a\left(\sum_{i\in\mathcal{I}} x_i\varphi_i, \sum_{j\in\mathcal{I}} x_j\varphi_j\right) \geq C_E \left\|\sum_{i\in\mathcal{I}} x_i\varphi_i\right\|_V^2 \qquad \text{für alle } x\in\mathbb{R}^{\mathcal{I}}$$

bereits, dass die Steifigkeitsmatrix positiv definit sein muss. In unserem Fall ist die Bilinearform symmetrisch, und auch diese Eigenschaft wird an die Steifigkeitsmatrix vererbt. Also kann das cg-Verfahren direkt zur Berechnung der Lösung u_h eingesetzt werden.

Gitterhierarchie. Effizienter ist ein Mehrgitterverfahren, für das wir allerdings eine Hierarchie von Gleichungssystemen benötigen, die über Prolongations- und Restriktionsmatrizen aneinander gekoppelt sind. Hier erweisen sich die Techniken zur Gitterverfeinerung als hilfreich: Ausgehend von einer groben Triangulierung $\mathcal{T}_0$ konstruieren wir eine feinere Triangulierung $\mathcal{T}_1$, auf deren Grundlage wir eine noch feinere Triangulierung $\mathcal{T}_2$ konstruieren. In dieser Weise können wir fortfahren, bis wir über eine Reihe $\mathcal{T}_0,\ldots,\mathcal{T}_L$ von Triangulierungen verfügen, die per Verfeinerung auseinander hervorgegangen sind.

Hierarchie der Gleichungssysteme. Auf jeder Triangulierung $\mathcal{T}_\ell$ haben wir eine Familie $(\varphi_{\ell,i})_{i\in\mathcal{I}_\ell}$ von Basisfunktionen, die zu einer Steifigkeitsmatrix $A_\ell\in\mathbb{R}^{\mathcal{I}_\ell\times\mathcal{I}_\ell}$ und einem Lastvektor $b_\ell\in\mathbb{R}^{\mathcal{I}_\ell}$ führen. Damit steht uns eine Hierarchie von Gleichungssystemen

$$A_\ell x_\ell = b_\ell \qquad\qquad \text{für alle } \ell\in\{0,\ldots,L\}$$

zur Verfügung.

Prolongation. Die Prolongationsmatrix lässt sich besonders einfach konstruieren, wenn wir bedenken, dass die Triangulierungen per Verfeinerung definiert wurden: Jedes Dreieck wurde in mehrere kleinere Dreiecke zerlegt, also ist eine Funktion, die auf der Triangulierung $\mathcal{T}_{\ell-1}$ stückweise polynomiell ist, es auch auf der feineren Triangulierung $\mathcal{T}_\ell$. Demzufolge muss sich eine Basisfunktion $\varphi_{\ell-1,j}$ als Linearkombination

$$\varphi_{\ell-1,j} = \sum_{i\in\mathcal{I}_\ell} p_{\ell,ij}\varphi_{\ell,i}$$

der Basisfunktionen der feineren Triangulierung darstellen lassen. Die Koeffizienten definieren eine Matrix $P_\ell\in\mathbb{R}^{\mathcal{I}_\ell\times\mathcal{I}_{\ell-1}}$, die

$$\sum_{j\in\mathcal{I}_{\ell-1}} x_{\ell-1,j}\varphi_{\ell-1,j} = \sum_{i\in\mathcal{I}_\ell} (P_\ell x_{\ell-1})_i\varphi_{\ell,i} \qquad\qquad \text{für alle } x_{\ell-1}\in\mathbb{R}^{\mathcal{I}_{\ell-1}}$$

erfüllt, mit der sich also eine Funktion auf der gröberen Triangulierung *exakt* als Funktion auf der feineren darstellen lässt. Eine bessere Prolongation kann man sich kaum vorstellen.

Bemerkung 7.14 (Lineare Basisfunktionen) *Die Konstruktion der Prolongationsmatrix P_ℓ ist besonders einfach, wenn wir mit stückweise linearen Basisfunktionen arbeiten: Da unsere Basisfunktionen eine Lagrange-Basis bilden, können wir die Koeffizienten berechnen, indem wir die Basisfunktion $\varphi_{\ell-1,j}$ der groben Triangulierung in den Eckpunkten $i \in \mathcal{I}_\ell$ der feineren auswerten:*

$$p_{\ell,ij} = \varphi_{\ell-1,j}(i) \qquad \text{für alle } i \in \mathcal{I}_\ell.$$

Bei den bisher vorgestellten Verfeinerungsstrategien entspricht ein Eckpunkt $i \in \mathcal{I}_\ell$ des feinen Gitters entweder einem Eckpunkt des groben Gitters, dann gilt

$$p_{\ell,ij} = \begin{cases} 1 & \text{falls } j = i \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } j \in \mathcal{I}_{\ell-1},$$

oder der Eckpunkt liegt im Mittelpunkt zwischen zwei Eckpunkten $j_1, j_2 \in \mathcal{I}_{\ell-1}$ des gröberen Gitters, dann folgt aus der Linearität der Basisfunktionen

$$p_{\ell,ij} = \begin{cases} 1/2 & \text{falls } j \in \{j_1, j_2\}, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } j \in \mathcal{I}_{\ell-1}.$$

Wenn wir im Verfeinerungsalgorithmus Buch darüber führen, wie die neuen Eckpunkte entstanden sind, können wir die Prolongationsmatrizen sehr einfach aufstellen.

Restriktion. Bei der Konstruktion der Restriktionsmatrix ist zu beachten, dass im Kontext der Finite-Elemente-Verfahren Vektoren in unterschiedlicher Weise interpretiert werden: Die Lösung x_ℓ des Gleichungssystems $A_\ell x_\ell = b_\ell$ beschreibt die Koeffizienten der Lösung, während die Komponenten des Lastvektors b_ℓ gerade durch die Auswertung des Funktionals λ in den Basisfunktionen entstehen. Vektoren des ersten Typs nennen wir *Koeffizientenvektoren*, die des zweiten Typs *Funktionalvektoren*. Die Steifigkeitsmatrix A_ℓ bildet Koeffizientenvektoren auf Funktionalvektoren ab, deshalb ist das Residuum $b_\ell - A_\ell \tilde{x}_\ell$ einer Näherungslösung $\tilde{x}_\ell$ ebenfalls ein Funktionalvektor.

Die Prolongationsmatrix wirkt auf Koeffizientenvektoren, während die Restriktionsmatrix auf Funktionalvektoren wirkt. Nehmen wir an, dass ein Funktionalvektor $y \in \mathbb{R}^{\mathcal{I}_\ell}$ zu einem Funktional $\mu \in V'$ gehört, dass also

$$y_i = \mu(\varphi_{\ell,i}) \qquad \text{für alle } i \in \mathcal{I}_\ell$$

gilt. Wir suchen nach einem Funktionalvektor $z \in \mathbb{R}^{\mathcal{I}_{\ell-1}}$, der zu dem selben Funktional gehört. Da wir Basisfunktionen der groben Triangulierung mit Hilfe der Prolongationsmatrix darstellen können, erhalten wir

$$z_j = \mu(\varphi_{\ell-1,j}) = \mu\left(\sum_{i \in \mathcal{I}_\ell} p_{\ell,ij}\varphi_{\ell,i}\right) = \sum_{i \in \mathcal{I}_\ell} p_{\ell,ij}\mu(\varphi_{\ell,i})$$

$$= \sum_{i \in \mathcal{I}_\ell} p_{\ell,ij} y_i = (P_\ell^* y)_j \qquad \text{für alle } j \in \mathcal{I}_{\ell-1},$$

wir können also den Funktionalvektor z auf der groben Triangulierung durch Multiplikation mit der Adjungierten der Prolongationsmatrix aus dem Funktionalvektor y auf der feinen Triangulierung berechnen. Damit ist $R_\ell := P_\ell^*$ die bestmögliche Restriktion, da beiden Funktionalvektoren exakt dasselbe Funktional μ zugeordnet ist, so wie im Fall der Prolongation zwei Koeffizientenvektoren exakt dieselbe Funktion zugeordnet wurde.

Bemerkung 7.15 (Galerkin-Produkt) *Um zu überprüfen, ob Steifigkeits- und Prolongationsmatrizen korrekt aufgestellt wurden, bietet es sich an, die* Galerkin-Produkt-Eigenschaft *heranzuziehen: Für $i, j \in \mathcal{I}_{\ell-1}$ gilt*

$$\begin{aligned} a_{\ell-1,ij} &= a(\varphi_{\ell-1,i}, \varphi_{\ell-1,j}) = a\left(\sum_{\hat{i} \in \mathcal{I}_\ell} p_{\ell,\hat{i}i} \varphi_{\ell,\hat{i}}, \sum_{\hat{j} \in \mathcal{I}_\ell} p_{\ell,\hat{j}j} \varphi_{\ell,\hat{j}} \right) \\ &= \sum_{\hat{i} \in \mathcal{I}_\ell} \sum_{\hat{j} \in \mathcal{I}_\ell} p_{\ell,\hat{i}i} a(\varphi_{\ell,\hat{i}}, \varphi_{\ell,\hat{j}}) p_{\ell,\hat{j}j} = \sum_{\hat{i} \in \mathcal{I}_\ell} \sum_{\hat{j} \in \mathcal{I}_\ell} p_{\ell,\hat{i}i} a_{\ell,\hat{i}\hat{j}} p_{\ell,\hat{j}j} = (P_\ell^* A_\ell P_\ell)_{ij}, \end{aligned}$$

also folgt

$$A_{\ell-1} = R_\ell A_\ell P_\ell.$$

Falls uns eine Funktion zur Verfügung steht, die einen Vektor mit den beteiligten Matrizen multipliziert, können wir die Gültigkeit dieser Gleichung beispielsweise anhand einiger Zufallsvektoren überprüfen.

Alternativ kann (mit nicht unerheblichem „Verwaltungsaufwand“) die Gleichung auch verwendet werden, um $A_{\ell-1}$ zu definieren. *Bei diesem Ansatz müssten lediglich die Steifigkeitsmatrix A_L auf der feinsten Triangulierung und die Prolongationsmatrizen gegeben sein, die restlichen Steifigkeitsmatrizen ließen sich induktiv konstruieren.*

8 Parallelisierung

Die für numerische Simulationen verwendeten mathematischen Modelle werden zunehmend komplexer, so dass auch der Rechenaufwand für ihre Behandlung wächst. Falls wir uns nicht darauf verlassen wollen, dass jemand einen neuen Algorithmus erfindet, der den Rechenaufwand deutlich reduziert, müssen wir uns mit der Frage beschäftigen, wie wir eine gegebene Anzahl an Rechenoperationen in möglichst kurzer Zeit ausführen können.

Über einige Jahrzehnte trugen Fortschritte in der Prozessorentwicklung dazu bei, dass jede Prozessorgeneration etwas schneller als ihre Vorgängerin war, so dass derselbe Algorithmus, oft sogar dasselbe Programm, auf einem neuen Computer schneller als auf einem alten lief. Heute dagegen stoßen wir an die durch die Physik gesetzten Grenzen: Die Vakuum-Lichtgeschwindigkeit von ungefähr 300,000 Kilometern pro Sekunde kann nicht überschritten werden. Bei einem mit 3 GHz arbeitenden Prozessor können also Signale in einem Taktzyklus höchstens eine Entfernung von

$$\frac{3 \times 10^8}{3 \times 10^9} = 10^{-1}$$

Metern zurücklegen, das sind zehn Zentimeter. Da ein Prozessor nicht aus Vakuum besteht, reduziert sich die Zahl noch etwas weiter.

Da die einzelnen Funktionseinheiten des Prozessors miteinander Daten austauschen müssen, kann die Taktfrequenz nur dann gesteigert werden, wenn der Prozessor kleiner wird. Der weiteren Miniaturisierung setzt allerdings die Quantenphysik Grenzen: Je kleiner die Strukturen sind, desto größer ist das Risiko, dass sie sich gegenseitig stören.

Der Ausweg besteht darin, nicht zu versuchen, einen Prozessor immer schneller zu machen, sondern stattdessen die zu bewältigende Arbeit auf mehrere Prozessoren zu verteilen, die gleichzeitig arbeiten. Man spricht dann von *Parallelverarbeitung*.

8.1 Parallelrechner

Cluster. Es gibt verschiedene Ansätze zur Konstruktion von Parallelrechnern. Technisch wenig anspruchsvoll ist ein *Cluster*: Eine Reihe von Rechnern (in diesem Kontext als *Knoten* bezeichnet) wird mit einem schnellen Netzwerk verbunden. Die Rechner führen Programme aus und können über das Netzwerk Daten austauschen, beispielsweise Zwischenergebnisse. Cluster bieten den Vorteil, relativ kosteneffizient konstruiert werden zu können, indem man sie aus einfachen Standard-PCs zusammensetzt. Falls einer der Knoten ausfällt, kann er leicht durch einen anderen ersetzt werden. Leider steht diesem großen Vorteil auch ein nicht zu vernachlässigender Nachteil gegenüber:

Cluster sind nicht leicht zu programmieren. Die Kommunikation zwischen den einzelnen Knoten muss explizit durch den Programmierer umgesetzt werden, und die Menge an ausgetauschten Daten sollte nicht zu groß sein, da der Transport über das Netzwerk in der Regel wesentlich langsamer als beispielsweise ein Speicherzugriff ist.

Im Bereich des Hochleistungsrechnens (engl. *high performance computing*) scheinen Cluster zur Zeit weitgehend konkurrenzlos zu sein: Systeme mit mehreren tausend oder zehntausend Prozessoren werden in der Regel als Cluster aufgebaut, ihre Leistung voll auszunutzen erfordert eingehende Kenntnisse ihrer Architektur.

SMP-Rechner. Wesentlich einfacher zu programmieren sind *SMP-Rechner* (engl. für *simultaneous multiprocessing*), bei denen sich mehrere Prozessoren einen gemeinsamen Hauptspeicher teilen. Der gemeinsame Speicher bietet den Vorteil, dass ein Programm sich nicht um den Datenaustausch zu kümmern braucht, es liest einfach Daten aus dem Speicher und schreibt sie dorthin zurück.

Es kann zu Schwierigkeiten kommen, falls mehrere Prozessoren gleichzeitig Daten in dieselbe Speicherzelle schreiben müssen. Der beste Ausgang so einer Kollision ist noch ein Absturz des Programms mit einer sinnvollen Fehlermeldung, der schlechteste sind fehlerhafte Ergebnisse im Speicher, die spätere Rechnungen in die Irre führen und sehr schwer die diagnostizierende Störungen verursachen können.

Ein Betriebssystem, das SMP-Rechner unterstützt, bietet in der Regel eine Reihe einfacher Synchronisationsmechanismen, mit denen Schreibkollisionen verhindert werden können.

Viele heute verkaufte Standard-PCs sind SMP-Rechner: Sie weisen zwar in der Regel nur einen Prozessor auf, der allerdings zwei, vier oder mehr *Prozessorkerne* besitzt, die sich ein gemeinsames Speicherinterface und häufig auch Teile des Cache-Speichers teilen.

Der gemeinsam Zugriff auf den Speicher kann zwar die Programmierung eines SMP-Rechners erheblich vereinfachen, begrenzt allerdings auch den Grad der Effizienz eines derartigen Systems: Ein Prozessor mit DDR3-1600-Vierkanalspeicher kann unter optimalen Bedingungen ungefähr 50 GB an Daten pro Sekunde übertragen. Wenn er mit 3 GHz getaktet ist, können also ungefähr 16 Bytes pro Taktzyklus aus dem Hauptspeicher gelesen werden. Bei einfachen arithmetischen Operationen, beispielsweise der Berechnung des euklidischen Skalarprodukts zweier Vektoren, kann schon ein einziger Kern 16 Bytes pro Taktzyklus verarbeiten, so dass mehrere Kerne kaum ausgelastet werden.

Um einen SMP-Rechner effizient zu nutzen, ist es deshalb sehr wichtig, bei der Programmierung darauf zu achten, dass möglichst viele Rechenoperationen mit jedem aus dem Hauptspeicher gelesenen Datenelement ausgeführt werden.

NUMA-Rechner. Die von allen Prozessoren oder Prozessorkernen gemeinsam genutzte Speicherschnittstelle beschränkt den Nutzen eines SMP-Rechners, sobald eine größere Anzahl von Prozessoren zum Einsatz kommen soll. Einen Ausweg bietet ein *NUMA-Rechner* (engl. für *non-uniform memory access*), bei dem eine Reihe von Prozessoren zum Einsatz kommen, von denen jeder über seine eigene Speicherschnittstelle und damit auch seinen eigenen Speicher verfügt. Allerdings wird über ein spezialisiertes Netzwerk

sichergestellt, dass jeder Prozessor auch auf den Speicher zugreifen kann, der an andere Prozessoren angeschlossen ist. Da dieser Zugriff über das Netzwerk erfolgt, ist er etwas langsamer als der Zugriff auf den lokalen Speicher und belastet natürlich auch den Prozessor, der den Speicher verwaltet.

Im Prinzip kann ein NUMA-Rechner ähnlich wie ein SMP-Rechner programmiert werden, im Interesse einer möglichst hohen Geschwindigkeit empfiehlt es sich aber natürlich, sicher zu stellen, dass ein Prozessor möglichst viel mit seinem lokalen Speicher arbeitet und nur selten auf den Speicher anderer Prozessoren zugreifen muss: Es ist relativ einfach, einen NUMA-Rechner zu programmieren, aber es ist relativ schwierig, ihn *gut* zu programmieren.

Moderne Prozessoren verfügen in der Regel über Cache-Speicher, mit deren Hilfe versucht wird, die Anzahl der Zugriffe auf den Hauptspeicher möglichst gering zu halten. In NUMA-Systemen können dadurch Schwierigkeiten entstehen: Angenommen, Prozessor 1 liest Daten aus dem Speicher des Prozessors 2 und legt sie in seinem Cache ab. Dann verändert Prozessor 2 die Daten. Ohne weitere Maßnahmen würde das auf Prozessor 1 arbeitende Programm die Daten aus dessen Cache verwenden, obwohl diese Daten mit den Daten im Speicher nichts mehr zu tun haben, so dass eine Speicherzelle scheinbar verschiedene Werte aufweisen könnte, abhängig davon, welcher Prozessor diese Speicherzelle anspricht.

Deshalb verwendet man heute in der Regel ccNUMA-Architekturen (engl. für *cache-coherent non-uniform memory access*), bei denen über geeignete Protokolle sichergestellt ist, dass alle Caches informiert werden, falls die in ihnen enthaltenen Daten durch Schreibzugriffe anderer Prozessoren ungültig geworden sind.

Vektorrechner. Bei allen bisher beschriebenen Parallelrechnern ist ein mehr oder weniger hoher Aufwand erforderlich, um die Synchronisation der beteiligten Prozessoren sicherzustellen: Falls Prozessor 1 ein Zwischenresultat berechnet, das Prozessor 2 benötigt, muss das Programm dafür sorgen, dass Prozessor 2 das Zwischenresultat erst benutzt, wenn es berechnet wurde und ihm zur Verfügung steht. Die Abhängigkeiten zwischen verschiedenen Prozessoren können zu sporadisch auftretenden Fehlern führen, die sehr schwer zu diagnostizieren sind.

Ein *Vektorrechner* umgeht dieses Problem, indem er nur ein einziges Programm ausführt, bei dem jeder Befehl auf mehrere parallele Datenströme wirkt: Beispielsweise könnten in einem ersten Schritt 1000 Zahlen aus dem Speicher gelesen werden, in einem zweiten Schritt werden diese 1000 Zahlen mit zwei multipliziert, um in einem dritten Schritt in den Speicher zurückgeschrieben zu werden. Bei dieser *datenparallelen* Vorgehensweise gibt es keine Schwierigkeiten mit der Synchronisation mehrerer Programme, weil nur ein einziges Programm zur Zeit ausgeführt wird.

Offensichtlich ist dieser Ansatz aber auch wesentlich weniger flexibel als die bisher diskutierten, denn er lässt sich sinnvoll nur auf Probleme anwenden, bei denen exakt dieselben Operationen auf große Datenmengen angewendet werden müssen.

Ein klassisches Anwendungsgebiet für Vektorrechner ist das Rechnen mit Vektoren, also die lineare Algebra. Um beispielsweise eine Linearkombination $x = y + \alpha z$ zu be-

rechnen, müssen eine Addition und eine Multiplikation mit dem Faktor α für alle Komponenten der Vektoren durchgeführt werden. Das sind ideale Bedingungen für einen Vektorrechner. Die Berechnung eines Matrix-Vektor-Produkts lässt sich auf eine Folge von Linearkombinationen zurückführen, die Berechnung des Produkts zweier Matrizen wiederum auf eine Folge von Matrix-Vektor-Produkten.

Ein weiteres Anwendungsgebiet für Vektorrechner ist die Verarbeitung von Bilddaten: Um beispielsweise ein Bild zu glätten, müssen in jedem Bildpunkt gewichtete Mittelwerte zwischen der Farbe in diesem Punkt in den Farben in seinen Nachbarn berechnet werden. In jedem Punkt sind exakt dieselben Operationen auszuführen, also eignet sich auch dieser Algorithmus gut für Vektorrechner.

Moderne PCs sind nicht nur in der Regel SMP-Rechner, sie sind auch Vektorrechner: Um die Verarbeitung von Audio- und Videodaten zu unterstützen wurden die Prozessoren in geringem Umfang um Vektoroperationen erweitert, mit denen sich eine Rechenoperation auf bis zu 32 Datenelemente gleichzeitig anwenden lassen. Dadurch verschärft sich in gewisser Weise ein bereits bei der Besprechung der SMP-Architektur erwähnte Problem: Wenn der Prozessor pro Taktzyklus eine größere Datenmenge verarbeiten kann, muss der Speicherzugriff schnell genug sein, um diese Datenmenge rechtzeitig bereitstellen zu können. Auch hier gilt also: Einen Vektorrechner zu programmieren ist relativ einfach, ihn *gut* zu programmieren erfordert ein genaueres Verständnis seiner Eigenschaften.

8.2 OpenMP

Einen relativ einfachen Einstieg in die parallele Programmierung bietet der Standard OpenMP (siehe `http://www.openmp.org`). Er beschreibt einerseits eine Erweiterung des Funktionsumfangs von C- oder Fortran-Compilern und andererseits eine Reihe von Bibliotheksfunktionen. Das zugrundeliegende Programmiermodell ist das eines Rechners mit geteiltem Speicher, also beispielsweise eines SMP- oder NUMA-Rechners. Anders als frühere Standards (beispielsweise POSIX Threads, siehe `http://standards.ieee.org/findstds/standard/1003.1-2008.html` oder `http://www.unix.org/version3/ieee_std.html`) ist OpenMP eng mit dem Compiler verbunden und kann so beispielsweise direkt die Verarbeitung von Schleifen auf mehrere Prozessoren verteilen oder die Zugriffe auf einzelne Variablen steuern.

Ein OpenMP-Programm wird zunächst von einem einzigen Prozessor beziehungsweise Prozessorkern ausgeführt. Der Programmierer kann festlegen, dass sich das Programm an bestimmten Stellen in mehrere Ausführungsstränge (engl. *threads*) aufspalten kann, die parallel ablaufen, bis sie wieder zusammengeführt werden. Alle Threads greifen auf denselben Hauptspeicher zu, für Funktionsaufrufe steht allerdings jedem Thread ein eigener Speicherbereich (deutsch *Keller*, engl. *stack*) zur Verfügung in dem er Rücksprungadressen und lokale Variablen ablegen kann.

Ein einfaches Beispiel für ein OpenMP-Programm könnte wie folgt aussehen:

```
int main()
{
#pragma omp parallel
  {
    printf("Hallo Welt!\n");
  }
  return 0;
}
```

Die Direktive `#pragma omp parallel` veranlasst den Compiler dazu, an dieser Stelle des Programms mehrere Threads (man spricht von einem *Team* von Threads) zu erzeugen, die die Befehle in den auf die Anweisung folgenden geschweiften Klammern ausführen. Diesen Teil des Programms bezeichnet man als *parallelen Abschnitt*.

Die Anzahl der Threads ist dabei von der OpenMP-Implementierung, dem verwendeten Rechner und dem Wert der Umgebungsvariablen `OMP_NUM_THREADS` abhängig. Falls wir Glück haben, wird unser Beispielprogramm in einigen Zeilen den Begrüßungstext ausgeben. Falls wir weniger Glück haben, könnte ein erster Thread beispielsweise die Hälfte des Texts ausgeben, bevor ein zweiter damit beginnt, seinen Text auszugeben, so dass eine Mischung aus beiden Texten entsteht, die vermutlich nicht die Intention des Programmierers widerspiegelt (es sei denn, er möchte die Tücken der parallelen Programmierung illustrieren).

Kollidierende Speicherzugriffe. Der gemeinsam genutzte Speicher kann unerwünschte Konsequenzen haben, wenn mehrere Threads gleichzeitig auf dieselbe Speicherzelle zugreifen. Ein Beispiel ist das folgende Programm, bei der mehrere Threads dieselbe Variable verwenden:

```
int main()
{
  int i = 0;
#pragma omp parallel
  {
    i++;
  }
  printf("%d\n", i);
  return 0;
}
```

Falls dieses Programm mit n Threads ausgeführt wird, können beliebige Ergebnisse zwischen 1 und n auftreten. Die Ursache liegt darin, dass sich die Anweisung `i++` aus mehreren Teilen zusammensetzt: Zuerst wird der aktuelle Wert der Variablen `i` aus dem

Speicher in eine interne Variable des Prozessors, ein *Register*, gelesen, dann wird der Wert um eins erhöht und schließlich das Ergebnis in den Speicher zurückgeschrieben. Da nicht festgelegt ist, in welcher Reihenfolge die Threads auf den Speicher zugreifen, kann es passieren, dass alle den Wert 0 lesen, bevor der erste den aktualisierten Wert zurückschreibt. In diesem Fall berechnen alle Threads denselben Wert 1, der dann auch von dem letzten Thread in die `i` zugeordnete Speicherzelle geschrieben wird. Es kann aber auch passieren, dass der erste Thread den aktualisierten Wert zurückgeschrieben hat, bevor der zweite Thread ihn liest. Dann würde das Endergebnis mindestens 2 sein.

Atomare Operationen. Für derartige Probleme bietet OpenMP eine einfache Lösung in Gestalt der Direktive `#pragma omp atomic`, mit der sich einzelne Befehle als unteilbar (daher „atomar“) kennzeichnen lassen. OpenMP stellt dann sicher, dass keine Zugriffe auf die von diesem Befehl betroffenen Speicherzellen geschehen, bevor der Befehl nicht vollständig ausgeführt wurde. Das entsprechend modifizierte Programm

```
int main()
{
  int i = 0;
#pragma omp parallel
  {
#pragma omp atomic
    i++;
  }
  printf("%d\n", i);
  return 0;
}
```

berechnet immer den korrekten Wert, nämlich die Anzahl der Threads in dem Team, das dieses Programm ausgeführt hat.

Kritische Abschnitte. Die Direktive `#pragma omp atomic` darf nur für eine kleine Menge von C-Ausdrücken verwendet werden, für längere Befehlsfolgen, die nicht von anderen Threads unterbrochen werden sollen, ist die Direktive `#pragma omp critical` vorgesehen:

```
int main()
{
  int i = 0;
#pragma omp parallel
  {
```

```
#pragma omp critical(zaehlen)
    {
      i++;
    }
  }
  return 0;
}
```

Die Direktive definiert einen *kritischen Abschnitt* (engl. *critical section*) des Programms, in dem sich jeweils nur ein einziger Thread aufhalten darf. Eventuelle weitere Threads, die den Abschnitt erreichen, müssen warten, bis er wieder freigegeben ist. Kritische Abschnitte können mit einem Namen, hier `zaehlen`, versehen werden, mit deren Hilfe sich auch mehrere im Programm verteilte Befehlsfolgen zu einem größeren kritischen Abschnitt zusammenfassen lassen: Falls ein Thread irgendeinen der Teilabschnitte ausführt, müssen alle anderen Threads vor *allen* Teilabschnitten warten. Mit diesem Mechanismus lassen sich beispielsweise gemeinsam genutzte Variablen über das gesamte Programm hinweg vor Kollisionen schützen.

Locking. Ein sehr flexibler Mechanismus zur Absicherung von Zugriffen auf gemeinsame Ressourcen (und unter gewissen Bedingungen auch zur Synchronisation mehrerer Threads) ist *Locking*. Anders als die bisher behandelten Techniken wird es in OpenMP über Bibliotheksfunktionen umgesetzt, die in der Include-Datei `omp.h` definiert werden:

```
#include "omp.h"

void omp_init_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Eine Variable des Typs `omp_lock_t` wird mit `omp_init_lock` vorbereitet, nach getaner Arbeit sollte als Gegenstück `omp_destroy_lock` aufgerufen werden, um beispielsweise eventuell zusätzlich angelegten Speicher freizugeben. Aus der Perspektive des Programmierers kann eine Variable dieses Typs nur zwei Zustände annehmen: Sie ist entweder „offen“ oder „abgeschlossen“. Nach dem Aufruf der Funktion `omp_init_lock` ist sie zunächst offen.

Ein Aufruf von `omp_set_lock` versetzt die Variable in den Zustand „abgeschlossen“, falls sie vorher offen war. Falls sie vorher bereits abgeschlossen war, wartet der Thread, bis sie von einem anderen Thread in den Zustand „offen“ zurückversetzt wird, und unternimmt dann einen neuen Versuch. Falls mehrere Threads warten, ist unbestimmt, welcher von ihnen erfolgreich ist.

Ein Aufruf von `omp_unset_lock` versetzt die Variable in den Zustand „offen“, und zwar unabhängig davon, ob sie vielleicht vorher schon in diesem Zustand war. Falls wir beispielsweise ein Programm schreiben, bei dem zu jedem Aufruf der Funktion `omp_set_lock` genau ein Aufruf der Funktion `omp_unset_lock` gehört, müssen wir sorgfältig darauf achten, dass keiner der zweiten Aufrufe „verloren geht“, weil er auf eine bereits offene Variable trifft.

Die Funktion `omp_test_lock` versucht ebenfalls, die Variable in den Zustand „abgeschlossen“ zu versetzen, wartet aber nicht, falls sie bereits abgeschlossen sein sollte. Stattdessen gibt sie den Wert null zurück, falls die Variable bereits abgeschlossen war, oder einen anderen Wert, falls sie offen war und jetzt abgeschlossen ist.

Deadlock. Bei fehlerhafter Programmierung kann es zu einer sogenannten *Deadlock-Situation* (in deutsch manchmal als *Verklemmung* bezeichnet) kommen, in der ein Thread endlos darauf wartet, dass eine Lock-Variable in den Zustand „offen“ wechselt. Ein klassisches Beispiel ist das Problem der *dining philosophers*: Fünf Philosophen sitzen um einen runden Tisch, zwischen ihnen liegt jeweils eine Gabel, vor jedem steht sein Mittagessen. Um essen zu können, muss ein Philosoph beide Gabeln in den Händen halten. Da die Hand-Auge-Koordination der Philosophen nicht die beste ist, können sie nur nach einer Gabel zur Zeit greifen. Da sie hungrig sind, legen sie eine einmal gegriffene Gabel erst wieder zurück, wenn sie auch eine zweite erhalten und ihren Hunger für den Moment gestillt haben. In diesem Fall legen sie beide Gabeln zurück.

Je nachdem, in welcher Reihenfolge sie das tun, kann es zu verschiedenen Ergebnissen kommen: Es können alle Philosophen reihum etwas essen, es können einige essen und andere verhungern, und es können auch alle verhungern. Letzteres tritt ein, falls alle Philosophen nach der zu ihrer Linken liegenden Gabel greifen, bevor einer der anderen nach der rechts liegenden Gabel greifen kann. Damit ist jeder Philosoph im Besitz genau einer Gabel, die er auch nicht zurück legt, keine weitere liegt auf dem Tisch, also kann kein Philosoph etwas essen.

Eine einfache Lösung des Problems besteht darin, den Zugriff auf den Tisch mit einer Lock-Variablen zu sichern. Sobald ein Philosoph mit `omp_set_lock` erfolgreich war, kann er sich soviele Gabeln nehmen, wie er möchte, essen, und die Gabeln wieder zurücklegen, bevor er den Tisch mit `omp_unset_lock` wieder freigibt. Eine ähnliche Strategie verfolgten früher Versionen des Linux-Kernels, bei dem Zugriffe auf gemeinsam genutzte zentrale Speicherbereiche mit dem *big kernel lock* geschützt wurden. Einfach und sicher, aber nicht sehr effizient, da auch die Philosophen warten müssen, für die eigentlich beide Gabeln vorhanden wären. Im Fall des Linux-Kernels wurde viel Arbeit darauf verwenden, das *big kernel lock* nach und nach zu beseitigen und durch Lock-Variablen für Teilmodule zu ersetzen.

Eine weitere Möglichkeit besteht darin, die benötigten Ressourcen zu numerieren und immer in aufsteigender Reihenfolge anzufordern: Gabeln und Philosophen sind entgegen den Uhrzeigersinn von 0 bis 4 durchnumeriert und Philosoph $n \in \{0, \ldots, 4\}$ versucht, die Gabeln n und $(n+1) mod 5$ zu greifen. Die Philosophen $n \in \{0, \ldots, 3\}$ greifen zuerst nach der Gabel zu ihrer Linken, Philosoph $n = 4$ aber zuerst nach der zu seiner Rechten.

Man sieht, dass es nicht einfach ist, sicher zu stellen, dass ein paralleles Programm überhaupt arbeitet, statt in einer Deadlock-Situation stecken zu bleiben.

Arbeitsteilung. Nach der Direktive `#pragma omp parallel` wird ein Teil des Programms von mehreren Threads gleichzeitig abgearbeitet. Um diese Threads gewinnbringend einsetzen zu können, müssen wir die zu leistende Arbeit auf sie verteilen. Besonders einfach lässt sich diese Aufgabe lösen, falls ähnliche Rechenoperationen sehr oft wiederholt werden müssen: In der Regel wird man dann eine Schleife programmieren. Falls bekannt ist, wie oft die Schleife durchlaufen werden wird, können wir mit der Direktive `#pragma omp for` dafür sorgen, dass die einzelnen Schleifendurchläufe auf mehrere Threads verteilt werden:

```
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<n; i++)
    x[i] = y[i] + alpha * z[i];
}
```

Die Direktive `#pragma omp for` muss unmittelbar vor einer `for`-Schleife stehen, die eine Reihe von Bedingungen erfüllen muss, die es dem Compiler ermöglichen, bereits vor Eintritt in die Schleife festzustellen, wie oft sie durchlaufen werden wird, damit er festlegen kann, wie die Arbeit gleichmäßig auf die Threads zu verteilen ist. Beispielsweise muss die Schleifenvariable vom Typ `int` sein, und die Abbruchbedingung des Schleife muss die Form `i<n`, `i>m`, `i<=n` oder `i>=m` aufweisen.

Alle Threads in dem die Schleife verarbeitenden Team warten am Ende darauf, dass alle anderen fertig werden. Dadurch ist sichergestellt, dass alle Speicheradressen, die in der Schleife aktualisiert werden sollen, bei Verlassen der Schleife ihre endgültigen Werte aufweisen.

Die Direktiven `#pragma omp parallel` und `#pragma omp for` können der Übersicht halber zu einer Direktive `#pragma omp parallel for` zusammengefasst werden, die beide kombiniert:

```
#pragma omp parallel for
for(i=0; i<n; i++)
  x[i] = y[i] + alpha * z[i];
```

Vor Eintritt in die `for`-Schleife werden mehrere Threads angelegt, dann wird die Schleife auf alle verteilt und verarbeitet. Nach Verlassen der Schleife werden die Threads wieder zu einem einzelnen Ausführungsstrang zusammengefasst, so wie es auch am Ende eines parallelen Abschnitts geschehen würde.

Private Variablen. Alle Variablen, die vor `#pragma omp parallel` definiert wurden, werden von allen Threads geteilt. Das verursacht Schwierigkeiten, falls wir beispielsweise das Matrix-Vektor-Produkt mit einem Programm der folgenden Form zu berechnen versuchen:

```
double sum;
int i, j;
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<n; i++) {
    sum = 0.0;
    for(j=0; j<n; j++)
      sum += a[i+j*n] * x[j];
    y[i] += alpha * sum;
  }
}
```

Da alle Threads die Variable `sum` teilen, ist unbestimmt, welche Werte die Variable annehmen wird. Dasselbe gilt für die Variable `j`, mit der wir die innere Schleife steuern.

OpenMP bietet uns für derartige Situationen die Möglichkeit, einzelne Variablen bei Eintritt in einen parallelen Abschnitt als „privat“ zu deklarieren. Dabei legt jeder Thread eine Kopie der genannten Variablen an, die nur er verwendet, so dass Kollisionen vermieden werden. Eine korrekte Fassung des Matrix-Vektor-Produkts könnte dann wie folgt aussehen:

```
double sum;
int i, j;
#pragma omp parallel private(sum,j)
{
  #pragma omp for
  for(i=0; i<n; i++) {
    sum = 0.0;
    for(j=0; j<n; j++)
      sum += a[i+j*n] * x[j];
    y[i] += alpha * sum;
  }
}
```

Der Zusatz `private` enthält eine Liste derjenigen Variablen, die jeder Thread durch eine private Kopie ersetzen soll.

Explizite Arbeitsteilung. Wir können die anliegende Arbeit auch explizit auf mehrere Threads verteilen, indem wir Bibliotheksfunktionen verwenden:

```
#include "omp.h"

int omp_in_parallel();
int omp_get_num_threads();
int omp_get_thread_num();
```

Die Funktion `omp_in_parallel` gibt genau dann einen von null verschiedenen Wert zurück, falls sie innerhalb eines parallelen Abschnitts aufgerufen wurde. Das ist nützlich, weil es beispielsweise erlaubt ist, aus einem parallelen Abschnitt heraus andere Funktionen aufzurufen, und diese Funktionen können mit Hilfe der Funktion `omp_in_parallel` feststellen, ob sie bei dem Zugriff auf eventuell gemeinsam genutzte Speicherbereiche oder sonstige Ressourcen Vorsicht walten lassen müssen.

Falls die Funktion `omp_get_num_threads` in einem parallelen Abschnitt aufgerufen wird, gibt sie die Anzahl der Threads zurück, mit denen dieser Abschnitt ausgeführt wird. Die Funktion `omp_get_thread_num` dagegen gibt unter diesen Bedingungen die Nummer des Threads zurück, der sie aufgerufen hat, beginnend mit dem *master thread*, der die Nummer null trägt:

```
if(omp_in_parallel())
  printf("Thread %d von %d arbeitet.\n",
         omp_get_thread_num(), omp_get_num_threads());
else
  printf("Keine parallelen Threads.\n");
```

Mit diesen Funktionen können wir beispielsweise die Matrix-Vektor-Multiplikation so aufteilen, dass jeder Thread einen möglichst großen zusammenhängenden Zeilenblock bearbeitet:

```
start = n * omp_get_thread_num() / omp_get_num_threads();
stop = n * (omp_get_thread_num()+1) / omp_get_num_threads();
for(i=start; i<stop; i++) {
  sum = 0.0;
  for(j=0; j<n; j++)
    sum += a[i+j*n] * x[j];
  y[i] += alpha * sum;
}
```

Speicherzugriff auf NUMA-Systemen. Eine explizite Arbeitsverteilung ist für NUMA-Architekturen von Vorteil, da wir mit ihrer Hilfe dafür sorgen können, dass jeder Prozessor überwiegend mit Daten arbeitet, die in seinem lokalen Speicher liegen, statt zu häufig auf den Speicher anderer Prozessoren zugreifen zu müssen. Wenn auf einem modernen Betriebssystem mit einem Aufruf einer Funktion wie `malloc` Speicher angefordert wird, wird dabei in der Regel noch kein realer Speicherbereich zur Verfügung gestellt, sondern lediglich ein Vermerk in einer Verwaltungsstruktur angelegt. Reale Speicherblöcke werden erst angefordert, wenn das Programm auf sie zugreift. Manche Betriebssysteme verfolgen dabei auf NUMA-Systemen die Strategie, dass zu diesem Zeitpunkt der lokale Speicher des Prozessors verwendet wird, der denjenigen Thread ausführt, der den Speicherzugriff verursacht hat. Falls wir also dafür sorgen können, dass möglichst immer dieselben Threads auf bestimmte Speicheradressen zugreifen, dürfen wir auf erhebliche Geschwindigkeitsvorteile auf NUMA-Architekturen hoffen.

8.3 Message Passing

Ein gemeinsam genutzter Speicher kann in vielen Fällen die Umsetzung von Algorithmen erleichtern, er kann aber auch zu unvorhergesehenen Schwierigkeiten führen, beispielsweise wenn mehrere Threads gleichzeitig in eine Speicherzelle zu schreiben versuchen. Abgesehen davon wächst die Komplexität eines Rechners erheblich, wenn beispielsweise in einer ccNUMA-Architektur sichergestellt werden muss, dass viele Prozessoren über Veränderung in Speicherzellen fremder Prozessoren informiert werden müssen. Mit dem steigenden Verwaltungsaufwand sinkt die Effizienz.

Deshalb sind Cluster-Systeme von großem Interesse, insbesondere im Bereich des Hochleistungsrechnens. Cluster können wesentlich einfacher strukturiert sein, und damit auch günstiger und unter Umständen auch effizienter.

Für die Programmierung entstehen durch den Einsatz eines Cluster-Systems allerdings neue Herausforderungen: Wir können nicht mehr direkt auf Speicherbereiche fremder Prozessoren zugreifen, stattdessen müssen wir Daten bei ihnen explizit anfordern und darauf hoffen, dass sie unsere Anfrage beantworten.

MPI. In diesem Bereich hat sich der Standard MPI (engl. *message passing interface*, siehe `http://www.mpi-forum.org`) durchgesetzt, der beschreibt, wie auf mehreren Prozessoren laufende Programme sich miteinander verständigen können. Ein MPI-Programm wird üblicherweise mittels eines separaten Steuerprogramms auf mehreren Knoten eines Clusters parallel gestartet, dabei entstehen parallel laufende Prozesse mit völlig unabhängigen Speicherbereichen. Im Extremfall können die Prozesse sogar beispielsweise auf unterschiedlichen Rechnerarchitekturen an geographisch weit voneinander entfernten Orten laufen.

Mit dem Aufruf der Funktion `MPI_Init` im Hauptprogramm wird die MPI-Implementierung initialisiert, anschließend können MPI-Funktionen benutzt werden. Mit einem Aufruf der Funktion `MPI_Finalize` wird der Bibliothek die Gelegenheit gegeben, vor Ende des Programms eventuelle Aufräumarbeiten zu leisten.

Anzahl und Rang. Wichtig für die Arbeitsteilung sind vor allem zwei Funktionen, die an die Stelle der uns vons OpenMP bekannten Funktionen `omp_get_num_threads` und `omp_get_thread_num` treten:

```
#include "mpi.h"
int main(int argc, char *argv[]) {
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Process %d of %d\n", rank, size);
  MPI_Finalize();
}
```

Eine Gruppe von parallel laufenden MPI-Prozessen wird durch eine Datenstruktur namens `MPI_Comm` repräsentiert, genannt *communicator*. Ein solcher *communicator* ist in jedem MPI-Programm vordefiniert, nämlich `MPI_COMM_WORLD`, die Repräsentation aller Prozesse, die gemeinsam gestartet wurden.

Die Anzahl der Prozesse in einem *communicator* lässt sich mit der Funktion `MPI_Comm_size` abfragen. Die Prozesse sind mit null beginnend fortlaufen numeriert, die Nummer des aufrufenden Prozesses wird als dessen *Rang* bezeichnet und kann mit der Funktion `MPI_Comm_rank` ermittelt werden.

Nur mit Hilfe dieser beiden Funktionen muss ein MPI-Programm entscheiden, welchen Teil der anstehenden Arbeit die einzelnen Prozesse übernehmen sollen.

Punkt-zu-Punkt-Kommunikation. Für die meisten praktischen Anwendungen ist es erforderlich, dass die einzelnen Prozesse untereinander Daten austauschen können. Zu diesem Zweck stellt MPI eine Reihe von Funktionen zur Verfügung. Die einfachsten von ihnen sind die folgenden:

```
double x[23];

if(rank == 0)
  MPI_Send(x, 23, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
else if(rank == 1)
  MPI_Recv(x, 23, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
```

Prozess null sendet 23 `double`-Werte aus dem Array `x` an Prozess eins, versehen mit einer Markierung (fünftes Argument, engl. *tag*) von null und innerhalb der durch den *communicator* `MPI_COMM_WORLD` gegebenen Kontexts. Prozess eins empfängt dieser 23

Werte und kümmert sich nicht darum, ob die Operation erfolgreich war oder nicht. Falls wir daran interessiert sind, gegebenenfalls eine Fehlermeldung zu erhalten, können wir ein Objekt des Typs `MPI_Status` anlegen und einen Zeiger darauf als letztes Argument an `MPI_Recv` übergeben.

Die Funktion `MPI_Send` ist fertig, sobald die ihr übergebenen Daten entweder schon erfolgreich verschickt oder wenigstens in einen Zwischenspeicher übertragen wurden. Die Spezifikation garantiert lediglich, dass die in dem Array `x` enthaltenen Daten von der Funktion nicht mehr benötigt werden, wir können das Array also beispielsweise mit neuen Daten füllen, ohne das Ergebnis des abgeschlossenen `MPI_Send`-Aufrufs zu verändern. Es ist nicht garantiert, dass die Daten bereits in Prozess eins eingetroffen sind, es muss nicht einmal ein passendes `MPI_Recv` in Prozess eins aufgerufen worden sein.

Die Funktion `MPI_Recv` ist fertig, sobald eine Nachricht vollständig empfangen wurde, die von dem angegebenen Prozess mit der angegebenen Markierung verschickt wurde. Die letzten beiden Bedingungen können wir abschwächen, indem wir für die Markierung den Wert `MPI_ANY_TAG` und für den Absender `MPI_ANY_SOURCE` einsetzen. Nach der Rückkehr in das aufrufende Programm enthält `x` die empfangenen Daten. Falls wir statt `MPI_STATUS_IGNORE` einen Zeiger auf ein Objekt des Typs `MPI_Status` als letztes Argument an die Funktion übergeben haben, können wir weitere Informationen über die eingegangene Nachricht erhalten:

```
MPI_Status status;
int count, error;

MPI_Recv(x, 23, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
count = MPI_Get_count(&status, MPI_DOUBLE, &error);
printf("%d Zahlen empfangen, Absender %d, "
       "Tag %d, Fehlercode %d\n",
       count, status.MPI_SOURCE, status.MPI_TAG, error);
```

Nicht-blockierende Kommunikation. Sowohl `MPI_Send` als auch `MPI_Recv` können dazu führen, dass das aufrufende Programm warten muss, bis der Empfänger die Daten akzeptiert oder der Sender sie abgeschickt hat: Sie können den Programmablauf *blockieren*. In manchen Situationen kann es nützlich sein, nicht auf den Abschluss eines Nachrichtenaustauschs warten zu müssen. Diesem Zweck dienen *nicht-blockierende* Sende- und Empfangsfunktionen, deren Namen sich durch den zusätzlichen Buchstaben I (für engl. *immediate*, im Sinne von „sofortige Rückkehr“) von ihren blockierenden Gegenstücken unterscheiden, beispielsweise heißen sie `MPI_Isend` oder `MPI_Irecv`. Die Funktionen verwenden ein zusätzliches Objekt des Typs `MPI_Request`, das den Zustand der Sende- oder Empfangsoperation beschreibt. Mit Hilfe der Funktion `MPI_Wait` kann darauf gewartet werden, dass die Operation abgeschlossen wurde:

```
MPI_Request request;

target_rank = (rank + 1) % size;
source_rank = (rank + size - 1) % size;
MPI_Isend(x, 23, MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD,
          &request);
MPI_Recv(y, 23, MPI_DOUBLE, source_rank, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Dieses Programmstück überträgt den Inhalt des Arrays `x` des aktuellen Prozesses in das Array `y` des Prozesses mit dem nächsthöheren Rang (modulo Anzahl der Prozesse). Wenn wir `MPI_Send` anstelle von `MPI_Recv` verwenden würden, könnte es passieren, dass das Programm nie aus der Funktion zurückkehrt, weil jeder Prozess senden will, aber keiner empfangen. Durch die Verwendung des nicht-blockierenden `MPI_Isend` können wir die Empfangsfunktion „parallel“ zu der Sendefunktion abarbeiten lassen.

Nicht-blockierende Sende- und Empfangsfunktionen können die Geschwindigkeit eines Programms verbessern, weil die MPI-Implementierung im Idealfall keine Zwischenspeicher einsetzen muss. In unserem Beispiel könnte beispielsweise eine optimierte Implementierung direkt die Daten aus dem Array `x` des einen Prozesses in das Array `y` des anderen kopieren. Auf gut ausgestatteten Systemen kann dieser Kopiervorgang an spezielle Hardware (Stichwort DMA, engl. *direct memory access*) delegiert werden, so dass (fast) keine zusätzliche Rechenzeit benötigt wird.

Kollektive Kommunikation. Gelegentlich ist es nützlich, Nachrichten nicht nur zwischen zwei Prozessen, sondern simultan zwischen allen Prozessen austauschen zu können. Beispielsweise könnte Prozess null einen Wert berechnen, den anschließend alle anderen Prozesse verwenden sollen. Diese Aufgabe lässt sich mit Funktionen für die *kollektive Kommunikation* lösen. In unserem Beispiel leistet `MPI_Bcast` das Gewünschte:

```
int n;

if(rank == 0) {
  printf("Problemdimension?\n");
  scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

In diesem Programmfragment erbittet der Prozess mit Rang null eine Eingabe des Benutzers, deren Ergebnis dann mit der Funktion `MPI_Bcast` an alle anderen Prozesse übermittelt wird. Die ersten drei Parameter beschreiben wie zuvor das Array der Eingabedaten, nämlich dessen Adresse, Länge und Typ, der vierte Parameter gibt die *Wurzel*

der Kommunikationsoperation an, also den Rang des Prozesses, dessen Daten an alle anderen übertragen werden sollen. Offenbar können alle Prozesse außer der Wurzel erst weiterarbeiten, wenn der Wurzelprozess die Funktion erreicht hat und die Nachricht übertragen wurde.

Selbstverständlich wäre es möglich, den Aufruf der Funktion `MPI_Bcast` durch `size-1` Aufrufe der Funktion `MPI_Send` in Prozess null und Aufrufe der Funktion `MPI_Recv` in den anderen Prozessen zu ersetzen, allerdings bieten sich dann der MPI-Implementierung weniger Möglichkeiten, den Datenaustausch zu optimieren. Falls in unserem Beispiel eine Gruppe von 2^p Prozessen auftritt, würde der naive Ansatz $2^p - 1$ Aufrufe der Funktion `MPI_Send` in Prozess null erfordern, denn jede Nachricht würde von ihm ausgehen. Falls wir `MPI_Bcast` verwenden, kann die MPI-Implementierung geschickter vorgehen: In einem ersten Schritt wird die Nachricht an Prozess 2^{p-1} übermittelt. In einem zweiten Schritt kann dieser Prozess die Nachricht weitergeben, beispielsweise an Prozess $3 \times 2^{p-2}$, während Prozess null sie an den Prozess 2^{p-2} übermittelt. In einem dritten Schritt besitzen schon vier Prozesse die Nachricht, so dass Prozess $i2^{p-2}$ sie an Prozess $(2i+1)2^{p-3}$ schicken kann, und so weiter. Bei dieser Vorgehensweise fallen in jedem Prozess nur höchstens p Sendeoperationen statt $2^p - 1$ an. Zwar bleiben insgesamt natürlich $2^p - 1$ Nachrichten auszutauschen, aber der Austausch kann parallel erfolgen. Bei geeigneter Vernetzung der beteiligten Rechner lässt sich der Zeitbedarf in dieser Weise erheblich reduzieren.

Übersetzen und Ausführen. Im Gegensatz zu den bisher diskutierten Bibliotheken und Erweiterungen des Compilers ist es bei MPI-Programmen üblich, einen speziellen Befehl zu verwenden, um sie zu übersetzen: An die Stelle des Compilers `cc` tritt `mpicc`. `mpicc` wird in der Regel die übergebenen Parameter um einige für die Verarbeitung von MPI-Programmen erforderliche ergänzen und anschließend den normalen Compiler `cc` aufrufen. Deshalb sollte `mpicc` auch dieselben Parameter wie `cc` verarbeiten können und sich genauso verhalten.

Da ein MPI-Programm typischerweise mehrfach gestartet wird, schließlich sollen mehrere Prozesse parallel arbeiten, genügt es nicht, einfach den Namen des Programms in der Befehlszeile einzugeben. Stattdessen wird das separate Programm `mpirun` verwendet, das mehrere Kopien des MPI-Programms startet, gegebenenfalls auf mehreren Rechnern eines Netzwerks. Das Übersetzen und Ausführen eines Testprogramms könnte also wie folgt ablaufen:

```
mpicc test_mpi1.c -o test_mpi1 -lm
mpirun -np 4 ./test_mpi1
```

Der Parameter `-np 4` sorgt dabei dafür, dass vier Prozesse gestartet werden, so dass der *communicator* `MPI_COMM_WORLD` genau vier Prozesse enthält.

Index