Programmierung numerischer Algorithmen

Steffen Börm

Stand 17. Juni 2019, 14 Uhr 04

Alle Rechte beim Autor.

Inhaltsverzeichnis

1 Einleitung **5**

2 Lineare Gleichungssysteme **7**
2.1 LR-Zerlegung 7
2.2 Konstruktion der LR-Zerlegung 11
2.3 Basic Linear Algebra Subprograms 14
2.4 Programmieren mit BLAS 18
2.5 Block-Algorithmen 22
2.6 Parallelisierung 27
2.7 Maschinenzahlen und Rundungsfehler 33
2.8 Konditionszahl 35
2.9 QR-Zerlegung 38

3 Iterationen **45**
3.1 Eigenwertprobleme 45
3.2 Konvergenz der Vektoriteration* 51
3.3 Newton-Iteration 56
3.4 Optimierungsaufgaben 60
3.5 Iterative Verfahren für lineare Gleichungssysteme 63
3.6 Konjugierte Gradienten* 66

4 Simulationen **71**
4.1 Federpendel 71
4.2 Wellengleichung 74
4.3 Mehrkörpersysteme 77
4.4 Populationsdynamik 81

5 Visualisierung **83**
5.1 OpenGL Legacy 83
5.2 GLUT 85
5.3 Homogene Koordinaten 89
5.4 Dreiecke 94
5.5 Beleuchtung 96
5.6 OpenGL Core Profile 98

6 Approximation von Funktionen **103**
6.1 Taylor-Entwicklung 103
6.2 Interpolation 107

6.3 Newton-Interpolation 109
6.4 Interpolationsfehler* 112

7 Numerische Integration **115**
7.1 Interpolatorische Quadratur 115
7.2 Summierte Quadraturformeln 122

8 Transformationen und Kompression **125**
8.1 Fourier-Synthese 125
8.2 Fourier-Analyse 131
8.3 Zirkulante Matrizen 132
8.4 Toeplitz-Matrizen 134

Index **137**

References **139**

1 Einleitung

Wenn wir die uns umgebende Welt beschreiben wollen, erkennen wir schnell, dass es Größen gibt, die sich nicht einfach „abzählen“ lassen, beispielsweise die Länge einer Strecke, die Temperatur eines Körpers, der Salzgehalt des Meerwassers, oder die Helligkeit einer Lichtquelle.

Während für abzählbare Größen die *diskrete Mathematik* Algorithmen bietet, mit denen sich viele zentrale Fragestellungen behandeln lassen, beispielsweise nach kürzesten Wegen in Graphen oder sparsamsten Einfärbungen von Landkarten, sind für nicht abzählbare Größen völlig andere Ansätze erforderlich.

Solche *kontinuierlichen* Größen kann ein digitaler Computer beispielsweise überhaupt nicht darstellen: Da ein Computer nur über einen Speicher endlicher Größe verfügt, kann er auch nur eine endliche Anzahl von Zuständen annehmen. Also kann er insbesondere nur endlich viele Werte darstellen.

Deshalb können beispielsweise die reellen Zahlen nur *approximiert*, also näherungsweise, dargestellt werden. Diese Darstellung kann so geschickt gestaltet werden, dass der auftretende relative Fehler gering ist, so dass die Ergebnisse für praktische Anwendungen ausreichend genau sind.

Ein Algorithmus, der mit reellen Zahlen arbeitet, muss natürlich nicht nur das Endergebnis, sondern auch alle Zwischenergebnisse geeignet approximieren. Falls man dabei ungeschickt vorgeht, können sich die dabei auftretenden kleinen Fehler gegenseitig so weit verstärken, dass ein völlig unbrauchbares Ergebnis entsteht. Eine wichtige Frage ist deshalb die nach der *Stabilität* eines Algorithmus, also danach, wie empfindlich er auf die in Zwischenschritten auftretenden Approximationsfehler reagiert.

Allerdings gibt es Situationen, in denen auch ein sehr stabiler Algorithmus zu schlechten Ergebnissen führen kann, nämlich wenn die auszuführende Berechnung sehr empfindlich auf unvermeidbare Störungen der Eingabedaten reagiert. Bei solchen *schlecht konditionierten* Aufgabenstellungen kann auch ein perfekter Algorithmus kein brauchbares Ergebnis ermitteln, in diesen Fällen lohnt es sich, über alternative Problemformulierungen nachzudenken.

Die Tatsache, dass wir ohnehin keine exakten Ergebnisse erwarten dürfen, ist aber auch eine Chance: Wir können nach Algorithmen suchen, die nur die Genauigkeit erreichen, die für eine konkrete Anwendung erforderlich ist.

Beispielsweise ist schon die Berechnung der Dezimaldarstellung der Quadratwurzel $\sqrt{2}$ praktisch unmöglich, da diese Darstellung unendlich viele Ziffern hat, die ein Computer nicht in endlicher Zeit bestimmen kann. In der Praxis genügen uns allerdings in der Regel wenige Nachkommastellen, und diese lassen sich mit geeigneten Algorithmen sehr schnell bestimmen.

Von besonderer Bedeutung sind dabei *iterative Verfahren*: Ein Schritt des Verfahrens

geht von einer Approximation der Lösung aus und versucht, sie zu verbessern. Durch eine geeignete Steuerung der Anzahl der Schritte können wir auf diese Weise theoretisch jede beliebige Genauigkeit erreichen.

In der Praxis müssen häufig große Datenmengen verarbeitet werden, beispielsweise sind bei einer Strömungssimulation Druck und Geschwindigkeit in vielen Punkten im Raum zu bestimmen. Es stellt sich dann die Frage, wie diese Daten so dargestellt werden können, dass typische Lösungsalgorithmen sie effizient und schnell verarbeiten können. Dabei bedeutet „effizient", dass der theoretische Rechenaufwand möglichst gering ist, während „schnell" bedeutet, dass der Algorithmus auf einem praktisch existierenden Rechner in möglichst kurzer Zeit ausgeführt werden kann.

Für die Untersuchung dieses zweiten Aspekts ist es erforderlich, den internen Aufbau eines Rechnersystems zumindest in Grundzügen zu kennen. Beispielsweise sind Speicherzugriffe auf modernen Rechnern häufig sehr langsam im Vergleich zu Rechenoperationen, so dass wir die bei vielen Prozessoren vorhandenen schnellen Hilfsspeicher geschickt einsetzen und möglichst viele Operationen pro aus dem Speicher gelesenem Datenelement ausführen sollten.

Da heute auch schon sehr einfache Computer über Mehrkernprozessoren verfügen, ist auch die Parallelisierung numerischer Algorithmen von großer Bedeutung für die Reduktion der Rechenzeit, also die Verteilung einer aufwendigen Berechnung auf mehrere Prozessorkerne, die möglichst unabhängig voneinander arbeiten sollen.

Danksagung

Ich bedanke mich bei Dirk Boysen, Christina Börst, Alexander Dobrick, Sven Christophersen, Nils Krütgen und Janne Henningsen für Korrekturen und Verbesserungsvorschläge.

2 Lineare Gleichungssysteme

Eine der wichtigsten Problemklassen in der numerischen Mathematik sind die *linearen Gleichungssysteme*. Viele zentrale Naturgesetze, beispielsweise das HOOKEsche Federgesetz der klassischen Mechanik, die MAXWELLschen Gleichungen der Elektrodynamik oder die KIRCHHOFFschen Regeln für elektrische Schaltungen, führen unmittelbar zu linearen Gleichungssystemen, während andere wichtige Gesetze wie die NAVIER-STOKES-Gleichungen der Strömungsmechanik sich mit Hilfe linearer Approximationen näherungsweise lösen lassen.

Aufgrund der großen Bedeutung dieser Gleichungssysteme überrascht es nicht, dass es eine große Vielfalt an Lösungsverfahren gibt, unter denen wir das für eine konkrete Aufgabenstellung am besten geeignete auswählen können.

In diesem Kapitel beschäftigen wir uns mit *direkten* Lösungsverfahren, die — zumindest theoretisch — mit einer endlichen Anzahl von Rechenoperationen die exakte Lösung eines linearen Gleichungssystems ermitteln.

2.1 LR-Zerlegung

Solange nur ein einziges lineares Gleichungssystem zu behandeln ist, können wir das aus der Schule bekannte GAUSSsche Eliminationsverfahren anwenden. Falls wir allerdings *mehrere* Systeme

$$Ax = b \tag{2.1}$$

mit derselben Matrix A, aber unterschiedlichen rechten Seiten b, lösen müssen, empfiehlt es sich, das Verfahren etwas umzuformulieren, nämlich als Zerlegung in *Dreiecksmatrizen*.

Definition 2.1 (Dreiecksmatrizen) *Seien $L, R \in \mathbb{R}^{n\times n}$ Matrizen.*

Wir nennen L eine linke untere Dreiecksmatrix*, falls*

$$i < j \Rightarrow \ell_{ij} = 0 \qquad \textit{für alle } i, j \in [1:n],$$

und wir nennen R eine rechte obere Dreiecksmatrix*, falls*

$$i > j \Rightarrow r_{ij} = 0 \qquad \textit{für alle } i, j \in [1:n].$$

Es ist üblich, bei Matrizen Nulleinträge auszulassen, solange dabei keine Missverständnisse auftreten können, so dass wir Dreiecksmatrizen häufig in der Gestalt

$$L = \begin{pmatrix} \ell_{11} & & \\ \vdots & \ddots & \\ \ell_{n1} & \cdots & \ell_{nn} \end{pmatrix}, \qquad R = \begin{pmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{pmatrix}$$

schreiben. Das Eliminationsverfahren überführt die Matrix A in eine obere Dreiecksmatrix R, indem eine Reihe von Transformationen $T_1, \dots, T_n$ angewendet werden:

$$T_n \cdots T_1 A = R.$$

Die Transformationen sind invertierbar, so dass wir auch

$$A = T_1^{-1} \cdots T_n^{-1} R$$

schreiben können. Bei genauerer Betrachtung stellt sich heraus, dass die Matrix $L := T_1^{-1} \cdots T_n^{-1}$ eine untere Dreiecksmatrix ist, wir haben also $A = LR$.

Definition 2.2 (LR-Zerlegung) *Seien $A, L, R \in \mathbb{R}^{n\times n}$ Matrizen.*

Falls L eine linke untere und R eine rechte obere Dreiecksmatrix ist und

$$A = LR$$

gilt, nennen wir das Paar (L, R) eine LR-Zerlegung *der Matrix A.*

LR-Zerlegungen sind nützliche Hilfsmittel für eine Vielzahl von Aufgabenstellungen. Wir haben bereits gesehen, dass sie sich eignen, um lineare Gleichungssysteme zu lösen: Falls (L, R) eine LR-Zerlegung der Matrix A ist, ist das Gleichungssystem (2.1) äquivalent zu

$$Ax = b \iff LRx = b \iff Ly = b \text{ und } Rx = y.$$

Falls wir also lineare Gleichungssysteme in Dreiecksgestalt lösen können, können wir auch das allgemeine System (2.1) lösen.

Wir untersuchen zunächst das System $Ly = b$ mit einer beliebigen invertierbaren unteren Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$. Wir zerlegen L, y und b in

$$L_{*1} = \begin{pmatrix} \ell_{21} \\ \vdots \\ \ell_{n1} \end{pmatrix}, \qquad L_{**} = \begin{pmatrix} \ell_{22} & & \\ \vdots & \ddots & \\ \ell_{n2} & \cdots & \ell_{nn} \end{pmatrix}, \qquad y_* = \begin{pmatrix} y_2 \\ \vdots \\ y_n \end{pmatrix}, \qquad b_* = \begin{pmatrix} b_2 \\ \vdots \\ b_n \end{pmatrix}$$

und erhalten damit die Gleichungen

$$Ly = \left(\begin{array}{c|ccc} \ell_{11} & & & \\ \hline \ell_{21} & \ell_{22} & & \\ \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{nn} \end{array}\right) \left(\begin{array}{c} y_1 \\ \hline y_2 \\ \vdots \\ y_n \end{array}\right) = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} y_1 \\ y_* \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_* \end{pmatrix}$$

mit deren Hilfe wir das System in Blockform kurz als

$$\begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} y_1 \\ y_* \end{pmatrix} = \begin{pmatrix} b_1 \\ b_* \end{pmatrix}, \tag{2.2}$$

procedure forsubst(L, **var** b);
for $k = 1$ **to** n **do begin**
  $b_k \leftarrow b_k/\ell_{kk}$;
  for $i \in [k+1 : n]$ **do**
    $b_i \leftarrow b_i - \ell_{ik} b_k$
end

Abbildung 2.1: Vorwärtseinsetzen zur Lösung des linken unteren Dreieckssystems $Ly = b$. Der Vektor b wird mit der Lösung y überschrieben.

schreiben können. Matrizen, die in dieser Form aus Teilmatrizen zusammengesetzt sind, bezeichnen wir als *Blockmatrizen*. Mit ihnen lässt sich analog zu gewöhnlichen Matrizen rechnen, insbesondere ist die Blockgleichung äquivalent zu den Gleichungen

$$\ell_{11} y_1 = b_1, \qquad\qquad L_{*1} y_1 + L_{**} y_* = b_*.$$

Falls $\ell_{11} \neq 0$ gilt, können wir die linke Gleichung direkt auflösen und zu $y_1 = b_1/\ell_{11}$ gelangen. Das Einsetzen in die rechte Gleichung führt zu

$$L_{**} y_* = b_* - L_{*1} y_1. \tag{2.3}$$

Das ist wieder ein lineares Gleichungssystem mit einer unteren Dreiecksmatrix L_{**}, allerdings hat sich die Dimension auf $n-1$ reduziert. Wir können diese Reduktion fortführen, bis wir ein eindimensionales System erreichen, das wir direkt lösen können.

Um möglichst nahe an der mathematischen Herleitung zu bleiben, könnte man den Algorithmus rekursiv formulieren. Da er aber besonders einfach ist, können wir uns den damit verbundenen Aufwand auch sparen und einfach eine Variable k verwenden, die angibt, welches die erste Zeile beziehungsweise Spalte von L_{**}, y_* und b_* ist.

Der resultierende Algorithmus ist in Abbildung 2.1 dargestellt, er ist naheliegenderweise auch als *Vorwärtseinsetzen* bekannt. Um Speicherplatz zu sparen, wird der Vektor b nach und nach mit dem Ergebnis y überschrieben. Das ist möglich, weil wir $b_1, \ldots, b_k$ nicht mehr benötigen, sobald die Koeffizienten $y_1, \ldots, y_k$ berechnet wurden.

Für das zweite System $Rx = y$ können wir ähnlich vorgehen: Wir führen Teilmatrizen und -vektoren

$$R_{*n} = \begin{pmatrix} r_{1n} \\ \vdots \\ r_{n-1,n} \end{pmatrix}, \quad R_{**} = \begin{pmatrix} r_{11} & \cdots & r_{1,n-1} \\ & \ddots & \vdots \\ & & r_{n-1,n-1} \end{pmatrix}, \quad x_* = \begin{pmatrix} x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}, \quad y_* = \begin{pmatrix} y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

ein und erhalten

$$Rx = \left(\begin{array}{ccc|c} r_{11} & \cdots & r_{1,n-1} & r_{1n} \\ & \ddots & \vdots & \vdots \\ & & r_{n-1,n-1} & r_{n-1,n} \\ \hline & & & r_{nn} \end{array}\right) \left(\begin{array}{c} x_1 \\ \vdots \\ x_{n-1} \\ \hline x_n \end{array}\right) = \begin{pmatrix} R_{**} & R_{*n} \\ & r_{nn} \end{pmatrix} \begin{pmatrix} x_* \\ x_n \end{pmatrix},$$

```
procedure backsubst(R, var y);
for k = n downto 1 do begin
  y_k ← y_k/r_kk;
  for i ∈ [1 : k − 1] do
    y_i ← y_i − r_ik y_k
end
```

Abbildung 2.2: Rückwärtseinsetzen zur Lösung des rechten oberen Dreieckssystems $Rx = y$. Der Vektor y wird mit der Lösung x überschrieben.

so dass wir die Gleichung $Rx = y$ kompakt als

$$\begin{pmatrix} R_{**} & R_{*n} \\ & r_{nn} \end{pmatrix} \begin{pmatrix} x_* \\ x_n \end{pmatrix} = \begin{pmatrix} y_* \\ y_n \end{pmatrix} \tag{2.4}$$

schreiben können. Die Block-Gleichung (2.4) ist äquivalent zu

$$R_{**}x_* + R_{*n}x_n = y_*, \qquad r_{nn}x_n = y_n.$$

Aus der rechten Gleichung folgt direkt $x_n = y_n/r_{nn}$, und die linke Gleichung nimmt dann die Form

$$R_{**}x_* = y_* - R_{*n}x_n \tag{2.5}$$

an, also die eines weiteren linearen Gleichungssystems mit einer rechten oberen Dreiecksmatrix R_{**}, das aber nur noch $n - 1$ Unbekannte aufweist. Wir können unseren Algorithmus nun rekursiv auf dieses kleinere Problem anwenden, bis wir bei einem Problem der Dimension eins angekommen sind, das wir direkt lösen können. Auch hier können wir uns eine „echte" Rekursion ersparen, indem wir eine Variable $k \in [1 : n]$ einführen, die die Dimension des gerade aktuellen Teilproblems angibt. Wir beginnen mit $k = n$, gehen „rekursiv" zu $k = n - 1$ über, dann zu $k = n - 2$, und so weiter. Auf diese Weise können wir den gesamten Algorithmus mit einer einfachen Schleife über k formulieren.

Auch hier können wir Speicherplatz sparen, indem wir die Komponenten des Eingabevektors y mit denen der Lösung x überschreiben, da die Werte $y_{k+1}, \ldots, y_n$ nicht mehr benötigt werden, sobald $x_{k+1}, \ldots, x_n$ bekannt sind.

Der resultierende Algorithmus ist in Abbildung 2.2 zusammengefasst. Er ist aus naheliegenden Gründen als *Rückwärtseinsetzen* bekannt.

Bemerkung 2.3 (Lösbarkeit) *Nebenbei haben wir mit unseren Algorithmen auch konstruktiv bewiesen, dass lineare Gleichungssysteme mit Dreiecksmatrizen sich eindeutig lösen lassen, falls alle Diagonalelemente ungleich null sind.*

Man kann sich überlegen, dass das nicht nur eine hinreichende, sondern sogar eine notwendige Bedingung ist.

Bemerkung 2.4 (Rechenaufwand) *Für die Bestimmung des Rechenaufwands numerische Algorithmen werden traditionell nur Operationen mit reellen Zahlen gezählt. In dieser Zählweise benötigt der Rumpf der k-Schleife des Vorwärtseinsetzens gerade $1 + 2(n-k)$ Operationen, so dass sich mit der* GAUSS*schen Summenformel insgesamt*

$$\sum_{k=1}^{n} 1 + 2(n-k) = n + 2\sum_{k=1}^{n} n - k = n + 2\sum_{\hat{k}=0}^{n-1} \hat{k} = n + 2\frac{n(n-1)}{2} = n^2$$

Operationen ergeben. Für das Rückwärtseinsetzen erhalten wir dasselbe Ergebnis.

Übungsaufgabe 2.5 (Matrix-Vektor-Multiplikation) *Seien $L, R \in \mathbb{R}^{n\times n}$ eine linke untere und eine rechte obere Dreiecksmatrix.*

Entwickeln Sie einen Algorithmus, der einen gegebenen Vektor $x \in \mathbb{R}^n$ mit dem Produkt Lx beziehungsweise dem Produkt Rx überschreibt, ohne Hilfsspeicher (abgesehen von Schleifenvariablen) zu benötigen.

Übungsaufgabe 2.6 (Rechteckige Dreiecksmatrizen) *Wie kann der Begriff der Dreiecksmatrix sinnvoll auf allgemeine Matrizen $L, R \in \mathbb{R}^{n\times m}$ übertragen werden?*

Angenommen, es steht ein Vektor $\widehat{x} \in \mathbb{R}^k$ mit $k = \max\{n, m\}$ zur Verfügung, in dessen ersten m Einträgen der Vektor x gespeichert ist. Können Sie analog zu der Übungsaufgabe 2.5 einen Algorithmus angeben, der die ersten n Einträge des Vektors $\widehat{x}$ mit Lx beziehungsweise Rx überschreibt?

Übungsaufgabe 2.7 (Left-looking substitution) *Unser Algorithmus für das Vorwärtseinsetzen wird auch als* right-looking substitution *bezeichnet, weil alle „rechts" von dem aktuellen k liegenden Einträge der rechten Seite b sofort aktualisiert werden, sobald y_k berechnet wurde. Diese Vorgehensweise hat beispielsweise auf Vektorrechnern Vorteile, da sie sich gut für die Parallelisierung eignet.*

Auf manchen anderen Rechnern bevorzugt man die left-looking substitution*, die durch*

```
for k = 1 to n do begin
  for j ∈ [1 : k − 1] do
    b_k ← b_k − ℓ_kj b_j;
  b_k ← b_k/ℓ_kk
end
```

gegeben ist. Begründen Sie, weshalb dieser Algorithmus ebenfalls den Vektor b mit der Lösung y des Gleichungssystems $Ly = b$ überschreibt.

2.2 Konstruktion der LR-Zerlegung

Nachdem wir nun davon überzeugt sind, dass eine LR-Zerlegung ein nützliches Hilfsmittel für die Behandlung linearer Gleichungssysteme sein kann, stellt sich die Frage, wie sich eine solche Zerlegung praktisch konstruieren lässt.

Für eine gegebene Matrix $A \in \mathbb{R}^{n\times n}$ suchen wir also eine linke untere Dreiecksmatrix $L \in \mathbb{R}^{n\times n}$ und eine rechte obere Dreiecksmatrix $R \in \mathbb{R}^{n\times n}$ mit

$$A = LR. \tag{2.6}$$

Wie zuvor können wir A, L und R in Blockmatrizen zerlegen. Dazu definieren wir

$$A_{**} = \begin{pmatrix} a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n2} & \cdots & a_{nn} \end{pmatrix}, \qquad A_{*1} = \begin{pmatrix} a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}, \qquad A_{1*} = \begin{pmatrix} a_{12} & \cdots & a_{1n} \end{pmatrix},$$

$$L_{**} = \begin{pmatrix} \ell_{22} & & \\ \vdots & \ddots & \\ \ell_{n2} & \cdots & \ell_{nn} \end{pmatrix}, \qquad L_{*1} = \begin{pmatrix} \ell_{21} \\ \vdots \\ \ell_{n1} \end{pmatrix},$$

$$R_{**} = \begin{pmatrix} r_{22} & \cdots & r_{2n} \\ & \ddots & \vdots \\ & & r_{nn} \end{pmatrix}, \qquad R_{1*} = \begin{pmatrix} r_{12} & \cdots & r_{1n} \end{pmatrix},$$

und schreiben (2.6) in der Blockdarstellung

$$\begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix} = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix} \begin{pmatrix} r_{11} & R_{1*} \\ & R_{**} \end{pmatrix},$$

an der wir die Gleichungen

$$a_{11} = \ell_{11} r_{11}, \tag{2.7a}$$

$$\ell_{11} R_{1*} = A_{1*}, \tag{2.7b}$$

$$L_{*1} r_{11} = A_{*1}, \tag{2.7c}$$

$$L_{*1} R_{1*} + L_{**} R_{**} = A_{**} \tag{2.7d}$$

ablesen können. Diese Gleichungen können wir nun der Reihe nach auflösen: (2.7a) wird beispielsweise durch $\ell_{11} = 1$ und $r_{11} = a_{11}$ erfüllt. Sobald uns ℓ_{11} und r_{11} vorliegen, können wir (2.7b) und (2.7c) durch

$$R_{1*} = \frac{1}{\ell_{11}} A_{1*}, \qquad L_{*1} = A_{*1} \frac{1}{r_{11}} \tag{2.8}$$

erfüllen, und (2.7d) schließlich führt zu

$$L_{**} R_{**} = A_{**} - L_{*1} R_{1*},$$

der definierenden Gleichung für eine LR-Zerlegung der nur noch $(n-1)$-dimensionalen Matrix $A_{**} - L_{*1} R_{1*}$. Nun können wir wie im Fall des Vorwärtseinsetzens fortfahren, bis nur noch eine eindimensionale Matrix übrig bleibt.

Der resultierende Algorithmus findet sich in Abbildung 2.3. Wie schon im Fall des Vorwärts- und Rückwärtseinsetzens überschreibt er die Einträge der Matrix A mit denen

```
procedure lrdecomp(var A);
for k = 1 to n 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 2.3: Berechnung der LR-Zerlegung $A = LR$. Der rechte obere Dreiecksanteil der Matrix A wird mit R überschrieben, der linke untere ohne die Diagonale mit L.

der Matrizen L und R. Da wir die Diagonaleinträge der Matrix L gleich eins setzen, brauchen sie nicht abgespeichert zu werden, so dass kein zusätzlicher Speicher benötigt wird. Nach dem ersten Schritt ist A durch

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ \ell_{21} & a_{22} - \ell_{21} r_{12} & \cdots & a_{2n} - \ell_{21} r_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & a_{n2} - \ell_{n1} r_{12} & \cdots & a_{nn} - \ell_{n1} r_{1n} \end{pmatrix}$$

überschrieben, am Ende der Prozedur durch

$$\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ \ell_{21} & r_{22} & \cdots & r_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ \ell_{n1} & \cdots & \ell_{n,n-1} & r_{nn} \end{pmatrix}. \tag{2.9}$$

An dieser Matrix können wir L (wegen $\ell_{11} = \ldots = \ell_{nn} = 1$) und R unmittelbar ablesen.

Bemerkung 2.8 (Eindeutigkeit) *Durch die Gleichung (2.7a) sind ℓ_{11} und r_{11} nicht eindeutig definiert, wir können eine der beiden Variablen frei (nicht gleich null!) wählen und die andere dann so anpassen, dass $\ell_{11} r_{11} = a_{11}$ gilt. Deshalb ist eine LR-Zerlegung im Allgemeinen nicht eindeutig bestimmt.*

Wir können allerdings Eindeutigkeit herstellen, indem wir, wie oben geschehen, fordern, dass alle Diagonaleinträge $\ell_{11}, \ldots, \ell_{nn}$ der linken unteren Dreiecksmatrix L gleich eins sind. Man spricht dann von einer normierten *linken unteren Dreiecksmatrix.*

Bemerkung 2.9 (Existenz) *Aus (2.7b) und (2.7c) folgt, dass bei der Konstruktion der LR-Zerlegung Schwierigkeiten auftreten, falls $\ell_{11} = 0$ oder $r_{11} = 0$ gelten, da sich dann diese Gleichungen nicht mehr eindeutig lösen lassen.*

Wegen (2.7a) ist das genau dann der Fall, wenn $a_{11} = 0$ gilt, beispielsweise besitzt die Matrix

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

keine LR-Zerlegung.

Da unser Algorithmus rekursiv arbeitet, treten entsprechende Probleme auf, wenn eine der im Zuge des Algorithmus entstehenden *Matrizen eine Null im linken oberen Koeffizienten aufweise.*

Es lässt sich zeigen, dass eine LR-Zerlegung einer invertierbaren Matrix A genau dann existiert, wenn auch alle Hauptuntermatrizen $A|_{[1:m]\times[1:m]}$ *mit* $m \in [1 : n-1]$ *invertierbar sind.*

Bemerkung 2.10 (Rechenaufwand) *Wir zählen wieder nur Operationen mit reellen Zahlen. Für jedes $k \in [1 : n]$ benötigt der Rumpf der k-Schleife*

$$(n-k) + 2(n-k)^2$$

Operationen, so dass insgesamt

$$\sum_{k=1}^{n} n-k+2(n-k)^2 = \sum_{\hat{k}=0}^{n-1} \hat{k} + 2\hat{k}^2 = \frac{n(n-1)}{2} + 2\frac{n(n-1)(2n-1)}{6}$$
$$= \frac{n(n-1)(4n+1)}{6} = \frac{n(4n^2-3n-1)}{6} \leq \frac{2}{3}n^3$$

Operationen erforderlich sind. Die Berechnung der LR-Zerlegung hat also kubische *Komplexität. Angesichts der Tatsache, dass die Matrix A durch nur n^2 Koeffizienten dargestellt ist, ist dieser Rechenaufwand unerfreulich hoch.*

2.3 Basic Linear Algebra Subprograms

Die bisher diskutierten Algorithmen für die Behandlung linearer Gleichungssysteme können wir im Prinzip unmittelbar implementieren. Allerdings ist zu befürchten, dass diese Implementierung nicht besonders gut sein wird: Moderne Prozessoren bieten eine Reihe von Ansatzpunkten für die Verbesserung der Laufzeit, beispielsweise durch die Ausnutzung von Caches oder durch Vektorisierung.

Beispielsweise können auf einem Prozessor, der die Befehlssatzerweiterung AVX unterstützt, bestimmte Rechenoperationen mit einem einzigen Befehl auf mehrere Eingabewerte angewendet werden, so dass sich bei dafür geeigneten Algorithmen die Laufzeit erheblich reduzieren lässt. Allerdings muss dafür die Implementierung maßgeschneidert werden, beispielsweise kann die Linearkombination $y \leftarrow \alpha x + y$ mit **float**-Vektoren durch

```
void
axpy(int n, float alpha, const float *x, float *y)
{
  __m256 v_alpha;
  int i;
  v_alpha = _mm256_broadcast_ss(&alpha);
  for(i=0; i+7<n; i+=8)
```

```
    _mm256_store_ps(y+i,
                    _mm256_add_ps(_mm256_load_ps(y+i),
                    _mm256_mul_ps(v_alpha,
                                  _mm256_load_ps(x+i))));
  for(; i<n; i++)
    y[i] += alpha * x[i];
}
```

unter optimalen Bedingungen um den Faktor acht beschleunigt werden. Allerdings müsste diese Funktion für jede neue Prozessorarchitektur und eventuell sogar für jede neue Generation derselben Architektur angepasst werden, um die optimale Geschwindigkeit zu erreichen.

Es wäre ineffizient, wenn jeder *Anwender* der numerischen linearen Algebra dazu gezwungen wäre, ein tiefes Verständnis der Hardware zu erwerben, auf der sein Programm laufen soll. Deshalb hat man sich im Laufe der Jahre auf eine Sammlung grundlegender Funktionen geeinigt, die klein genug ist, um es Hardware-Spezialisten zu ermöglichen, hoch optimierte Implementierungen für jede Prozessorgeneration zur Verfügung zu stellen, aber auch allgemein genug, um die wichtigsten Algorithmen der linearen Algebra ausdrücken zu können. Diese Sammlung von Funktionen trägt den Namen *Basic Linear Algebra Subprograms*, in der Regel abgekürzt als *BLAS*.

BLAS-Funktionen erwarten, dass Vektoren und Matrizen in einer bestimmten Weise dargestellt werden: Ein Vektor $x \in \mathbb{R}^n$ wird durch

- seine Dimension n,
- einen Zeiger `x` auf ein Array von **`float`**- oder **`double`**-Variablen und
- ein *Inkrement* `incx`

dargestellt. Der Eintrag x_i ist dann in `x[i*incx]` zu finden, wobei wir davon ausgehen, dass die Numerierung C-typisch mit $i = 0$ beginnt und mit $i = n - 1$ endet.

Eine Matrix $A \in \mathbb{R}^{n \times m}$ wird durch

- ihre Dimensionen n und m,
- einen Zeiger `A` auf ein Array von **`float`**- oder **`double`**-Variablen und
- eine *leading dimension* `ldA`

dargestellt. Dabei wird für die Übersetzung der zweidimensionalen Indizes der Matrix in die eindimensionalen Indizes des Arrays `A` die *column-major*-Anordnung verwendet, der Eintrag a_{ij} ist also in `A[i+j*ldA]` zu finden.

Diese Art der Darstellung bietet allerhand Vorteile:

- Indem wir Zeigerarithmetik verwenden, können wir eine in der k-ten Zeile und ℓ-ten Spalte beginnende Teilmatrix B einer Matrix A durch `B=A+k+l*ldA` mit unveränderter *leading dimension* `ldB=ldA` darstellen. Sofern der für das Produkt verwendete `int`-Typ nicht überläuft, gilt nach den Regeln der Zeigerarithmetik `B[i+j*ldB] = A[k+l*ldA+i+j*ldB] = A[(k+i)+(l+j)*ldA]`.

- In derselben Weise können wir einfach auf Spalten- und Zeilenvektoren einer Matrix zugreifen. Die k-te Zeile ist beispielsweise durch `x=A+k` mit `incx=ldA` charakterisiert, da `x[i*incx] = A[k+i*ldA]` gilt. Die k-te Spalte erhalten wir entsprechend durch `y=A+k*ldA` mit `incy=1`.

BLAS-Funktionen werden in drei Stufen (engl. *levels*) unterteilt: Level 1 enthält Funktionen, die mit Vektoren arbeiten, beispielsweise für die Berechnung einer Linearkombination oder eines Skalarprodukts. Level 2 enthält Funktionen, die mit Matrizen und Vektoren arbeiten, beispielsweise für die Berechnung der Matrix-Vektor-Multiplikation. Level 3 enthält Funktionen, die mit Matrizen arbeiten, unter denen die Matrix-Multiplikation die mit großem Abstand wichtigste ist.

Funktionen höherer Stufen können in der Regel durch Funktionen niedrigerer Stufen ausgedrückt werden, so dass beispielsweise eine optimierte Implementierung der Matrix-Vektor-Multiplikation auch zu einer besseren Effizienz der Matrix-Multiplikation führen kann.

Für die Zwecke der Vorlesung können wir uns auf eine Handvoll zentraler Funktionen beschränken.

Aus der ersten BLAS-Stufe [3] benötigen wir die Funktion

```
void
axpy(int n, real alpha, const real *x, int incx,
     real *y, int incy);
```

die einen Vektor $x \in \mathbb{R}^n$ mit einem Skalar $\alpha \in \mathbb{R}$ multipliziert und zu einem zweiten Vektor $y \in \mathbb{R}^n$ addiert, also die Operation $y \leftarrow \alpha x + y$ ausführt.

Darüber hinaus brauchen wir die Funktionen

```
real
nrm2(int n, const real *x, int incx);

real
dot(int n, const real *x, int incx,
    const real *y, int incy);
```

die die euklidische Norm

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2}$$

eines Vektors $x \in \mathbb{R}^n$ und das euklidische Skalarprodukt

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i$$

zweier Vektoren $x, y \in \mathbb{R}^n$ berechnen.

Schließlich erfordern einige Algorithmen noch die Funktion

```
void
scal(int n, real alpha, real *x, int incx);
```

die einen Vektor $x \in \mathbb{R}^n$ mit einem Skalar $\alpha \in \mathbb{R}$ multipliziert, also die Operation $x \leftarrow \alpha x$ ausführt.

Aus der zweiten BLAS-Stufe [2] benötigen wir die zentrale Funktion

```
void
gemv(bool trans, int n, int m,
      real alpha, const real *A, int ldA,
      const real *x, int incx, real *y, int incy);
```

die eine Matrix $A \in \mathbb{R}^{n \times m}$ mit einem Vektor $x \in \mathbb{R}^m$ und einem Skalar $\alpha \in \mathbb{R}$ multipliziert und das Ergebnis zu einem Vektor $y \in \mathbb{R}^n$ addiert, also die Operation $y \leftarrow \alpha Ax + y$ ausführt, falls `trans == false` gilt.

Falls dagegen `trans == true` gilt, wird $x \in \mathbb{R}^n$ mit α und der *transponierten* Matrix A^* multipliziert und zu $y \in \mathbb{R}^m$ addiert, es wird also $y \leftarrow \alpha A^* x + y$ ausgeführt.

Darüber hinaus ist beispielsweise für die Konstruktion der LR-Zerlegung die Funktion

```
void
ger(int n, int m, real alpha,
    const real *x, int incx,
    const real *y, int incy,
    real *A, int ldA);
```

sehr nützlich, die das Produkt $xy^* \in \mathbb{R}^{n \times m}$ zweier Vektoren $x \in \mathbb{R}^n$ und $y \in \mathbb{R}^m$ mit $\alpha \in \mathbb{R}$ multipliziert und zu einer Matrix $A \in \mathbb{R}^{n \times m}$ addiert, also die Operation $A \leftarrow \alpha xy^* + A$ ausführt. Hier ist xy^* eine Abkürzung für die durch

$$(xy^*)_{ij} = x_i y_j \qquad \text{für alle } i \in [1:n],\ j \in [1:m]$$

definierte Matrix. Derartige *Rang-1-Updates* treten bei der Behandlung von Matrixgleichungen häufig auf.

Gelegentlich arbeiten wir mit *symmetrischen* Matrizen, also mit Matrizen, bei denen $a_{ij} = a_{ji}$ gilt. In diesem Fall bietet es sich an, nur Einträge mit $i \leq j$, also oberhalb der Diagonalen, oder $i \geq j$, also unterhalb der Diagonalen, zu speichern und zu bearbeiten, da die anderen Einträge durch die Symmetriebedingung festgelegt sind und sich die Rechenzeit im Idealfall halbieren lässt. Dann benötigen wir allerdings eine Variante der `ger`-Funktion, die nur Einträge ober- oder unterhalb der Diagonalen verändert:

```
void
syr(bool upper, int n, real alpha,
    const real *x, int incx,
    real *A, int ldA);
```

Falls `upper` gesetzt ist, wird nur der Anteil der Matrix mit $i \leq j$ verändert, ansonsten nur der mit $i \geq j$. Hinzuaddiert wird die Matrix αxx^*.

Aus der dritten BLAS-Stufe [1] benötigen wir nur eine einzige Funktion:

```
void
gemm(bool transA, bool transB,
```

```
        int n, int m, int k, real alpha,
        const real *A, int ldA,
        const real *B, int ldB,
        real beta, real *C, int ldC);
```

Diese Funktion multipliziert eine Matrix $A \in \mathbb{R}^{n \times k}$ mit einer Matrix $B \in \mathbb{R}^{k \times m}$ und einem Skalar $\alpha \in \mathbb{R}$ und addiert das Ergebnis zu einer Matrix $C \in \mathbb{R}^{n \times m}$, die zuvor mit $\beta \in \mathbb{R}$ skaliert wurde. Es wird also die Operation $C \leftarrow \alpha AB + \beta C$ ausgeführt.

Falls `transA == true` gilt, wird die Matrix A durch ihre Transponierte A^* ersetzt. Dann ist n die Anzahl der Spalten und k die Anzahl der Zeilen der Matrix A.

Falls `transB == true` gilt, wird entsprechend die Matrix B durch ihre Transponierte B^* ersetzt. Dann ist m die Anzahl der Zeilen und k die Anzahl der Spalten der Matrix B.

2.4 Programmieren mit BLAS

Um einen Eindruck davon zu gewinnen, wie man mittels BLAS Aufgaben der linearen Algebra lösen kann, diskutieren wir zunächst, wie sich die beiden Funktionen der zweiten Stufe mit Hilfe derer der ersten Stufe ausdrücken lassen.

Die Matrix-Vektor-Multiplikation (`gemv`, *general matrix-vector multiplication*) lässt sich in der Form

```
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    y[i*incy] += alpha * A[i+j*ldA] * x[j*incx];
```

schreiben. Die innere Schleife ist dann das euklidische Skalarprodukt zwischen der i-ten Zeile der Matrix und dem Vektor x, so dass wir mit der Stufe-1-Funktion `dot` zu

```
for(i=0; i<n; i++)
  y[i*incy] += alpha * dot(m, A+i, ldA, x, incx);
```

gelangen. Wenn wir die i- und die j-Schleife vertauschen, stellen wir fest, dass wir n Linearkombinationen ausrechnen müssen, und das können wir mit der Stufe-1-Funktion `axpy` erledigen:

```
for(j=0; j<m; j++)
  axpy(n, alpha * x[j*incx], A+j*ldA, 1, y, incy);
```

Auf einem Intel®-Prozessor des Typs Xeon®E3-1220 v5 benötigt die erste Fassung für eine Dimension von 16 384 ungefähr 3,14 Sekunden, während die zweite mit 0,20 Sekunden ungefähr sechzehnmal schneller ist.

Für diesen erheblichen Geschwindigkeitsunterschied gibt es mindestens zwei Ursachen:

- *Caching:* Moderne Prozessoren greifen nicht direkt auf den Hauptspeicher zu, sondern nur auf einen kleinen und schnellen Hilfsspeicher, den *Cache*, der Kopien bestimmter Daten aus dem Hauptspeicher aufnimmt. Im Cache sind mehrere aufeinander folgende Adressen zu einer *cache line* zusammengefasst, und bei Zugriffen auf den Hauptspeicher werden grundsätzlich nur vollständige *cache lines* übertragen.

Durch die *column-major order*, in der BLAS Matrizen darstellt, ist es deshalb sehr günstig, eine Matrix spaltenweise zu durchlaufen, weil dann alle Einträge einer *cache line* genutzt werden, während es bei zeilenweiser Vorgehensweise nur ein Bruchteil wäre.

- *Vektorisierung:* Viele moderne Prozessoren enthalten Rechenwerke mehrfach, so dass sie eine arithmetische Operation für mehrere Eingabewerte simultan durchführen können (SIMD, *single instruction, multiple data*).

  Die `axpy`-Operation ist dafür gut geeignet, weil sich die Komponenten des Vektors simultan addieren lassen. Bei der `dot`-Operation in unserem Beispiel hingegen liegen die Daten so im Speicher, dass es für den Prozessor schwierig ist, die Rechenwerke auszulasten.

Auch bei der Funktion `ger`, die wir als

```
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    A[i+j*ldA] += alpha * x[i*incx] * y[j*incy];
```

schreiben können, ergeben sich je nach Anordnung der Schleifen zwei Varianten, die diesmal beide mit `axpy` realisiert werden: Falls wir die j-Schleife durch `axpy` ersetzen, erhalten wir

```
for(i=0; i<n; i++)
  axpy(m, alpha * x[i*incx], y, incy, A+i, ldA);
```

während wir bei der i-Schleife zu

```
for(j=0; j<m; j++)
  axpy(n, alpha * y[j*incy], x, incx, A+j*ldA, 1);
```

gelangen. Für die Dimension 16 384 benötigt die erste Variante 3,31 Sekunden, während der zweiten 0,24 Sekunden genügen. Auch in diesem Fall lässt sich der Geschwindigkeitsgewinn mit der Vektorisierung und der günstigeren Reihenfolge der Speicherzugriffe erklären.

Analog zu der überlegenen zweiten Fassung können wir auch die Funktion `syr` realisieren:

```
if(upper)
  for(j=0; j<n; j++)
    axpy(j+1, alpha * x[j*incx], x, incx, A+j*ldA, 1);
else
  for(j=0; j<n; j++)
    axpy(n-j, alpha * x[j*incx], x+j*incx, incx,
         A+j+j*ldA, 1);
```

Wir müssen daür sorgen, dass jeweils nur ein Teil des Vektors x zu einem Teil der Matrix A addiert wird. Falls die symmetrische Matrix durch ihren oberen Dreiecksanteil

dargestellt wird, brauchen wir lediglich die Dimension in dem `axpy`-Aufruf anzupassen. Für den unteren Dreiecksanteil müssen wir auch j zu den beiden Zeigern addieren.

Die Matrix-Multiplikation (`gemm`, *general matrix-matrix multiplication*) der dritten Stufe kann entsprechend durch die Funktionen der zweiten Stufe ausgedrückt werden. Nachdem wir die Skalierung mit β separat ausgeführt haben, könnte eine direkte Implementierung die Form

```
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    for(l=0; l<k; l++)
      C[i+j*ldC] += alpha * A[i+l*ldA] * B[l+j*ldB];
```

annehmen. Wenn wir die i-ten Zeilen der Matrizen A und C mit A_{i*} und C_{i*} bezeichnen, entsprechen die beiden inneren Schleifen der Matrix-Vektor-Multiplikation $C_{i*} \leftarrow C_{i*} + A_{i*}B$ beziehungsweise der transponierten Gleichung $C_{i*}^* \leftarrow C_{i*}^* + \alpha B^* A_{i*}^*$, die wir mit `gemv` realisieren können:

```
for(i=0; i<n; i++)
  gemv(true, k, m, alpha, B, ldB, A+i, ldA,
       C+i, ldC);
```

Eine zweite Variante erhalten wir, indem wir die beiden äußeren Schleifen tauschen, also erst über j und dann über i und ℓ iterieren. Dann entsprechen die beiden inneren Schleifen der Matrix-Vektor-Multiplikation der j-ten Spalte B_{*j} der Matrix B mit der Matrix A, die wir wieder mit `gemv` ausdrücken können:

```
for(j=0; j<m; j++)
  gemv(false, n, k, alpha, A, ldA, B+j*ldB, 1,
       C+j*ldC, 1);
```

Eine dritte und letzte Variante ergibt sich, wenn wir die Schleife über ℓ nach außen ziehen. Die Schleifen über i und j entsprechen dann einem Rang-1-Update, bei dem das Produkt der ℓ-ten Spalte $A_{*\ell}$ von A und der ℓ-ten Zeile $B_{\ell*}$ von B zu der Matrix C addiert wird. Mit der Funktion `ger` erhalten wir

```
for(l=0; l<k; l++)
  ger(n, m, alpha, A+l*ldA, 1, B+l, ldB, C, ldC);
```

Bei einer Dimension von $n = 2\,048$ benötigt die erste Variante 101,58 Sekunden, die zweite nur 6,49 Sekunden und die dritte 6,10 Sekunden. Auch in diesem Fall lassen sich die Unterschiede damit erklären, dass bei der zweiten Variante jeweils auf aufeinander folgende Speicheradressen zugegriffen wird und sich die einzelnen Linearkombinationen mit Vektoroperationen beschleunigen lassen, während diese Optimierungen bei der ersten Variante nicht greifen.

Aufbauend auf den BLAS-Funktionen können wir uns unserer ursprünglichen Aufgabe widmen, nämlich dem Lösen linearer Gleichungssysteme. Der erste Schritt besteht darin, die LR-Zerlegung zu berechnen, die nach unserer Herleitung für

$$A = \begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix}, \qquad L = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad R = \begin{pmatrix} r_{11} & R_{1*} \\ & R_{**} \end{pmatrix}$$

auf die vier Gleichungen

$$a_{11} = \ell_{11} r_{11}, \qquad A_{1*} = \ell_{11} R_{1*}, \qquad A_{*1} = L_{*1} r_{11}, \qquad A_{**} - L_{*1} R_{1*} = L_{**} R_{**}$$

führt. Wie bereits erwähnt, setzen wir $\ell_{11} = 1$ und $r_{11} = a_{11}$, so dass aus der zweiten Gleichung $R_{1*} = A_{1*}$ und aus der dritten $L_{*1} = A_{1*}/r_{11}$ folgen. Die Skalierung der Teilmatrix A_{1*} können wir mit `scal` durchführen. In der von BLAS-Funktionen verwendeten Datenstruktur können wir eine in der k-ten Zeile und Spalte beginnende Teilmatrix A_{**} per Zeiger-Arithmetik als `A+k+k*ldA` konstruieren. Die Subtraktion des Produkts $L_{*1} R_{1*}$ kann dann die Funktion `ger` übernehmen. Insgesamt erhalten wir den folgenden Algorithmus:

```
void
lrdecomp(int n, real *A, int ldA)
{
  int k;

  for(k=0; k<n; k++) {
    scal(n-k-1, 1.0 / A[k+k*ldA], A+(k+1)+k*ldA, 1);
    ger(n-k-1, n-k-1, -1.0,
        A+(k+1)+k*ldA, 1, A+k+(k+1)*ldA, ldA,
        A+(k+1)+(k+1)*ldA, ldA);
  }
}
```

Dieser Algorithmus berechnet die LR-Zerlegung der Matrix A in der in (2.9) gegebenen Form. Für eine Matrix der Dimension $n = 2\,048$ wird die Zerlegung auf dem erwähnten Xeon®-Prozessor in 2,47 Sekunden berechnet, benötigt also ungefähr ein Drittel der Zeit, die der Prozessor für die Matrix-Multiplikation per `ger` braucht.

Wenn uns die LR-Zerlegung zur Verfügung steht, müssen wir vorwärts einsetzen, um $Ly = b$ zu lösen, und rückwärts einsetzen, um schließlich aus $Rx = y$ die Lösung x zu erhalten. Da L eine normierte untere Dreiecksmatrix ist, also $\ell_{ii} = 1$ für alle $i \in [1 : n]$ gilt, ist das Vorwärtseinsetzen besonders einfach: Mit

$$L = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_* \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ y_* \end{pmatrix}$$

erhalten wir

$$y_1 = b_1/\ell_{11} = b_1, \qquad L_{**} y_* = b_* - L_{*1} y_1,$$

und letztere Operation können wir mit `axpy` ausführen, so dass sich die folgende Implementierung ergibt:

```
void
lsolve(int n, const real *L, int ldL, real *b)
{
```

```
  int k;

  for(k=0; k<n; k++)
    axpy(n-k-1, -b[k], L+(k+1)+k*ldL, 1, b+(k+1), 1);
}
```

Hier gibt k wieder die Zeile und Spalte an, in der die gerade aktuelle Teilmatrix von L beginnt, so dass L_{*1} ab `L+(k+1)+k*ldL` im Speicher zu finden ist.

Für das Rückwärtseinsetzen ergibt sich aus

$$R = \begin{pmatrix} R_{**} & R_{*n} \\ & r_{nn} \end{pmatrix}, \qquad y = \begin{pmatrix} y_* \\ y_n \end{pmatrix}, \qquad x = \begin{pmatrix} x_* \\ x_n \end{pmatrix},$$

unmittelbar

$$x_n = y_n / r_{nn}, \qquad R_{**} x_* = y_* - R_{*n} x_n,$$

und letztere Operationen kann auch wieder mit `axpy` ausgeführt werden. Insgesamt nimmt unsere Implementierung die folgende Gestalt an:

```
void
rsolve(int n, const real *R, int ldR, real *b)
{
  int k;

  for(k=n; k-->0; ) {
    b[k] /= R[k+k*ldR];
    axpy(k, -b[k], R+k*ldR, 1, b, 1);
  }
}
```

erhalten. Hier bezeichnet k die letzte Zeile und Spalte der gerade aktuellen Teilmatrix, so dass R_{*n} ab `R+k*ldR` im Speicher liegt.

Bemerkung 2.11 (Rückwärts zählende Schleife) *Die Schleife*

```
for(k=n; k-->0; ) {
  /* ... */
}
```

durchläuft die Werte $n-1$ bis 0: Wir setzen zu Beginn zwar k auf n und prüfen, ob es echt größer *als* 0 *ist, aber danach wird es sofort mit dem Postdekrement-Operator um eins reduziert. Dadurch arbeitet diese Schleife auch korrekt, falls k eine* **`unsigned`***-Variable ist und deshalb keine negativen Werte annehmen kann.*

2.5 Block-Algorithmen

Algorithmen der BLAS-Stufe 3 sind von besonderem Interesse, weil bei ihnen der Speicherbedarf quadratisch von der Dimension der Matrizen abhängt, der Rechenauf-

wand aber kubisch, bei großen Matrizen werden also mit den Koeffizienten relativ viele Rechenoperationen durchgeführt.

Diese Eigenschaft bietet uns die Möglichkeit, den *Cache-Speicher* moderner Prozessoren auszunutzen. Häufig ist der Cache in einer Hierarchie organisiert, beispielsweise ist der *first-level cache* relativ klein und sehr schnell und der *second-level cache* erheblich größer und langsamer. Bei Mehrkernprozessoren teilen sich die Prozessorkerne häufig einen *third-level cache*, der größer und langsamer als der *second-level cache* ist und mit dem sich der Datenaustausch zwischen den Kernen beschleunigen lässt.

Die Caches sind so konstruiert, dass sie für Programmiererinnen und Programmierer transparent sind, wir können also Programme schreiben, ohne uns mit den Caches zu beschäftigen. Allerdings werden diese Programme dann unter Umständen nicht allzu schnell laufen.

Um die Caches gut ausnutzen zu können, sollten wir wissen, wie sie arbeiten. Ein moderner Cache ist in *cache lines* von einigen Bytes organisiert, beispielsweise 64 bei gängigen Intel- oder AMD-Prozessoren. Bei einem Lesezugriff auf den Speicher wird eine *cache line* vollständig aus dem Hauptspeicher in den Cache übertragen. Bei einem Schreibzugriff wird (sofern wir keine speziellen Maschinenbefehle verwenden) ebenfalls eine *cache line* in den Cache übertragen, dort modifiziert, und dann entweder sofort (*write-through cache*) oder später (*write-back cache*) zurück in den Hauptspeicher geschickt. Da grundsätzlich immer vollständige *cache lines* übertragen werden, kann beispielsweise ein Schreibzugriff auf ein einziges Byte dazu führen, dass 64 Byte aus dem Hauptspeicher gelesen, modifiziert, und dann wieder in den Hauptspeicher geschrieben werden.

Entsprechend teilt sich eine **`double`**-Variable in einem Array jeweils eine *cache line* mit sieben weiteren. Falls wir also auf im Speicher hintereinander liegende Variablen zugreifen, kann der Cache besonders effizient arbeiten.

Da ein Cache in der Regel kleiner als der Hauptspeicher ist, müssen sich mehrere Adressen einen Platz im Cache teilen. Die einfachste Variante ist der *direct-mapped cache*, bei dem beispielsweise die niedrigsten Bits der Hauptspeicher-Adresse verwendet werden, um die Cache-Adresse zu ermitteln. Bei dieser Vorgehensweise gehört offenbar zu jeder Hauptspeicher-Adresse genau eine Cache-Adresse, und bei ungünstig verteilten Daten kann es geschehen, dass die Daten sich gegenseitig aus dem Cache verdrängen, obwohl sie noch häufiger gebraucht werden könnten (sog. *cache thrashing*).

Dieses Problem lösen *assoziative Caches*, bei denen zu jeder Hauptspeicher-Adresse eine (üblicherweise geringe) Anzahl von Cache-Adressen gehört. Die meisten heute in Prozessoren eingesetzten Caches sind assoziativ.

Zugriffe auf den Hauptspeicher sind *erheblich* langsamer als Zugriffe auf den Cache: Falls Daten im *first-level cache* sind, können sie bei aktuellen Prozessoren binnen 4 Taktzyklen dem Prozessor zur Verfügung stehen. Daten im *second-level cache* benötigen je nach Architektur 11–21 Taktzyklen, für den *third-level cache* sind es 30–65. Ein Zugriff auf den Hauptspeicher erfordert 167–195 Zyklen, also mehr als das Vierzigfache eines Zugriffs auf den *first-level cache*.

Da der Cache automatisch verwendet wird, brauchen wir lediglich unsere Algorithmen so zu organisieren, dass möglichst selten auf den Hauptspeicher zugegriffen werden muss,

dass also einmal in den Cache transferierte Daten möglichst oft genutzt werden.

Im Kontext der linearen Algebra haben sich dabei *Blockverfahren* als attraktiv erwiesen: Wir behandeln nicht alle Koeffizienten einer Matrix einzeln, sondern fassen sie zu Blöcken zusammen, also etwas größeren Teilmatrizen. Als Beispiel dient uns die LR-Zerlegung einer Matrix $A \in \mathbb{R}^{n\times n}$.

Wir wählen $m \in \mathbb{N}$ und $0 = \alpha_0 < \alpha_1 < \ldots < \alpha_m = n$. Nun setzen wird $n_\nu := \alpha_\nu - \alpha_{\nu-1}$ für alle $\nu \in [1:m]$ und definieren Teilmatrizen $A_{\nu\mu} \in \mathbb{R}^{n_\nu \times n_\mu}$ durch

$$(A_{\nu\mu})_{ij} = a_{\alpha_{\nu-1}+i,\alpha_{\mu-1}+j} \qquad \text{für alle } i \in [1:n_\nu],\ j \in [1:n_\mu],\ \nu,\mu \in [1:m],$$

so dass

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mm} \end{pmatrix}$$

gilt. Mit den Dreiecksmatrizen $L, R \in \mathbb{R}^{n\times n}$ können wir entsprechend verfahren, um

$$L = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{m1} & \cdots & L_{mm} \end{pmatrix}, \qquad R = \begin{pmatrix} R_{11} & \cdots & R_{1m} \\ & \ddots & \vdots \\ & & R_{mm} \end{pmatrix}$$

zu erhalten. Indem wir wie in (2.7) die erste Zeile und Spalte des Produkts

$$\begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mm} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{m1} & \cdots & L_{mm} \end{pmatrix} \begin{pmatrix} R_{11} & \cdots & R_{1m} \\ & \ddots & \vdots \\ & & R_{mm} \end{pmatrix}$$

untersuchen, ergeben sich die Gleichungen

$$\begin{aligned} A_{11} &= L_{11} R_{11}, \\ A_{\nu 1} &= L_{\nu 1} R_{11} \qquad \text{für alle } \nu \in [2:m], \\ A_{1\mu} &= L_{11} R_{1\mu} \qquad \text{für alle } \mu \in [2:m], \end{aligned}$$

$$\begin{pmatrix} A_{22} - L_{21}R_{12} & \cdots & A_{m2} - L_{21}R_{1m} \\ \vdots & \ddots & \vdots \\ A_{m2} - L_{m1}R_{12} & \cdots & A_{mm} - L_{m1}R_{1m} \end{pmatrix} = \begin{pmatrix} L_{22} & & \\ \vdots & \ddots & \\ L_{m2} & \cdots & L_{mm} \end{pmatrix} \begin{pmatrix} R_{22} & \cdots & R_{2m} \\ & \ddots & \vdots \\ & & R_{mm} \end{pmatrix},$$

aus denen wir wieder ein rekursives Verfahren konstruieren können, das in Abbildung 2.4 zusammengefasst ist.

Dieser Algorithmus ist allerdings unvollständig: Wir wissen zwar aus dem vorangehenden Kapitel, wie sich die LR-Zerlegung $L_{\kappa\kappa} R_{\kappa\kappa} = A_{\kappa\kappa}$ berechnen lässt, wir müssen aber auch wissen, wie sich die Systeme $L_{\kappa\kappa} R_{\kappa\mu} = A_{\kappa\mu}$ und $L_{\nu\kappa} R_{\kappa\kappa} = A_{\nu\kappa}$ lösen lassen.

Die erste Aufgabe haben wir im Prinzip bereits gelöst: $L_{\kappa\kappa}$ ist eine untere Dreiecksmatrix, also können wir einfach jede Spalte des Gleichungssystems per Vorwärtseinsetzen behandeln. Im Interesse der Effizienz verwenden wir dabei die Stufe-2-Funktion `ger` anstelle der Stufe-1-Funktion `axpy`, die wir in `lsolve` benutzt haben:

procedure block_lrdecomp(**var** A);
for $\kappa = 1$ **to** m **do begin**
Berechne die Zerlegung $L_{\kappa\kappa} R_{\kappa\kappa} = A_{\kappa\kappa}$;
for $\mu \in [\kappa + 1 : m]$ **do**
Löse $L_{\kappa\kappa} R_{\kappa\mu} = A_{\kappa\mu}$;
for $\nu \in [\kappa + 1 : m]$ **do**
Löse $L_{\nu\kappa} R_{\kappa\kappa} = A_{\nu\kappa}$;
for $\nu, \mu \in [\kappa + 1 : m]$ **do**
$A_{\nu\mu} \leftarrow A_{\nu\mu} - L_{\nu\kappa} R_{\kappa\mu}$
end

Abbildung 2.4: Berechnung der LR-Zerlegung $A = LR$ mit einem Block-Algorithmus.

```
void
block_lsolve(int n, int m, const real *L, int ldL,
             real *B, int ldB)
{
  int k;

  for(k=0; k<n; k++)
    ger(n-k-1, m, -1.0, L+(k+1)+k*ldL, 1, B+k, ldB,
        B+(k+1), ldB);
}
```

Die zweite Aufgabe ist auf den ersten Blick neu: Die obere Dreiecksmatrix $R_{\kappa\kappa}$ ist gegeben, und $L_{\nu\kappa}$ mit

$$L_{\nu\kappa} R_{\kappa\kappa} = A_{\nu\kappa}$$

ist gesucht. Anders als bisher steht die Dreiecksmatrix also auf der rechten Seite des Produkts. Glücklicherweise lässt sich das leicht ändern, indem wir die Gleichung transponieren. Wir erhalten

$$R^*_{\kappa\kappa} L^*_{\nu\kappa} = A^*_{\nu\kappa},$$

und da die transponierte Matrix $R^*_{\kappa\kappa}$ wieder eine *untere* Dreiecksmatrix ist, können wir auch dieses System durch spaltensweises Vorwärtseinsetzen behandeln.

In der konkreten Implementierung wäre es natürlich unattraktiv, die transponierten Matrizen tatsächlich explizit zu konstruieren und dafür Speicher anzufordern. Stattdessen wenden wir den Algorithmus `block_lsolve` direkt auf die transponierten Matrizen an, tauschen also lediglich Zeilen und Spalten.

Da $R_{\kappa\kappa}$ keine normierte Dreiecksmatrix ist, müssen wir die Division durch das Diagonalelement berücksichtigen, die sich für eine ganze Spalte mit der Stufe-1-Funktion `scal` realisieren lässt:

```
void
block_rsolve_trans(int n, int m, const real *R, int ldR,
```

```
                     real *B, int ldB)
{
  int k;

  for(k=0; k<n; k++) {
    scal(m, 1.0/R[k+k*ldR], B+k*ldB, 1);
    ger(m, n-k-1, -1.0, B+k*ldB, 1, R+k+(k+1)*ldR, ldR,
        B+(k+1)*ldB, ldB);
  }
}
```

Nun stehen uns alle benötigten Funktionen zur Verfügung, um die blockweise LR-Zerlegung zu realisieren. Die Hilfsgrößen $\alpha_0, \dots, \alpha_m$ konstruieren wir dabei mittels α_ν =`nu*n/m`, um eine möglichst einheitliche Größe der Matrixblöcke zu erreichen. Die Regeln der ganzzahligen Division in der Sprache C sorgen dabei dafür, dass $\alpha_0 = 0$ und $\alpha_m = n$ gelten, solange der verwendete **int**-Typ nicht bei der Berechnung des Produkts `nu*n` überläuft. Wir erhalten die folgende Implementierung:

```
void
block_lrdecomp(int n, int m, real *A, int ldA)
{
  int i, j, k;
  int oi, oj, ok, ni, nj, nk;

  for(k=0; k<m; k++) {
    ok = n * k / m;
    nk = n * (k+1) / m - ok;
    lrdecomp(nk, A+ok+ok*ldA, ldA);

    for(j=k+1; j<m; j++) {
      oj = n * j / m;
      nj = n * (j+1) / m - oj;
      block_lsolve(nk, nj, A+ok+ok*ldA, ldA,
                   A+ok+oj*ldA, ldA);
      block_rsolve_trans(nk, nj, A+ok+ok*ldA, ldA,
                         A+oj+ok*ldA, ldA);
    }

    for(j=k+1; j<m; j++) {
      oj = n * j / m;
      nj = n * (j+1) / m - oj;
      for(i=k+1; i<m; i++) {
        oi = n * i / m;
        ni = n * (i+1) / m - oi;
        gemm(false, false, ni, nj, nk, -1.0,
```

```
                A+oi+ok*ldA, ldA, A+ok+oj*ldA, ldA,
                1.0, A+oi+oj*ldA, ldA);
        }
      }
    }
  }
```

Auf dem Xeon®-Prozessor benötigt die konventionelle LR-Zerlegung `lrdecomp` für eine Matrix der Dimension $n = 4\,096$ ungefähr 22 Sekunden, während die blockweise Version mit $m = 32$ (also Blöcken mit 128×128 Koeffizienten) weniger als 9 Sekunden erfordert. Beide Varianten berechnen dabei exakt dieselbe Zerlegung, der einzige Unterschied ist die Reihenfolge der Speicherzugriffe.

2.6 Parallelisierung

Die heute üblichen Prozessoren zerlegen die Laufzeit eines Programms in *Takte* und führen in jedem Takt eine bestimmte, üblicherweise relativ geringe, Anzahl von Befehlen aus. Damit gibt die *Taktfrequenz* den Ausschlag für die Geschwindigkeit, mit der Programme ausgeführt werden.

Während im 20. Jahrhundert jede neue Generation von Prozessoren eine deutliche Steigerung der Taktfrequenz brachte, wachsen die Frequenzen im 21. Jahrhundert nur langsam, teilweise sinken sie sogar wieder. Eine Ursache sind physikalische Beschränkungen: Falls wir davon ausgehen, dass ein Prozessor einen Durchmesser von einem Zentimeter hat, kann selbst ein sich mit Lichtgeschwindigkeit fortpflanzendes Signal diese Strecke nur ungefähr 30 Milliarden mal pro Sekunde zurücklegen, damit ist 30 GHz die absolute Obergrenze für die erreichbare Taktfrequenz. Gemessen daran, dass das Signal auch eine gewisse zeitliche Dauer aufweisen muss, ist die heute bei Hochleistungsprozessoren übliche Frequenz von 3 bis 4 GHz vermutlich nahe an dem praktisch erreichbaren Maximum.

Um trotzdem die Rechenleistung weiter steigern zu können, ist man deshalb dazu übergegangen, Programme auf mehrere Prozessoren zu verteilen. Für diesen Ansatz hat sich der Begriff der *Parallelisierung* eingebürgert, teilweise auch der präzisere der *Nebenläufigkeit*.

Rechner mit mehreren unabhängigen Prozessoren sind allerdings relativ aufwendig und damit kostspielig. Deshalb sind als Kompromisslösung heute *Mehrkernprozessoren* im Einsatz, bei denen mehrere aus Steuer-, Recheneinheiten und lokalen Caches bestehende *Prozessorkerne* sich einen weiteren Cache teilen und mit dem Hauptspeicher und den Peripheriegeräten über eine gemeinsame Schnittstelle verbunden sind. Derartige Prozessoren können in relativ einfachen Hauptplatinen eingesetzt werden, bieten aber die Rechenleistung eines Mehrprozessorsystems. Da sich allerdings alle Prozessorkerne dieselbe Schnittstelle für den Hauptspeicher teilen müssen, kann es vorkommen, dass manche Programme diese Rechenleistung nicht voll nutzen können, weil sie nicht schnell genug die nötigen Daten aus dem Speicher erhalten.

Mehrkernprozessoren für allgemeine Aufgaben werden in der Regel so gestaltet, dass

alle Prozessorkerne eine gemeinsame Sicht auf den Speicher teilen („*shared memory*“). Falls also ein Kern einen Wert in den Speicher schreibt, sorgen geeignete Kommunikationsprotokolle dafür, dass alle anderen Kerne davon in Kenntnis gesetzt werden. Dadurch entsteht ein gewisser Mehraufwand, dem allerdings eine erhebliche Vereinfachung der Programmierung gegenüber steht.

Um die im Rahmen eines Programms zu leistende Rechenarbeit auf die Prozessorkerne verteilen zu können, müssen wir dem Rechner mitteilen, welche Aufgaben parallel ausgeführt werden können und welche voneinander abhängig sind. Ein etablierter Standard dafür ist OpenMP, eine Erweiterung der Programmiersprache C, die einige für die parallele Programmierung erforderliche Konzepte ergänzt.

Ein OpenMP-Programm ist unterteilt in *Threads*, also Folgen von Befehlen, die von Prozessorkernen ausgeführt werden. Jeder Thread verfügt dabei über einen eigenen Befehlszähler und über lokale Variablen, kann allerdings auch frei auf den Speicher zugreifen, um mit anderen Threads geteilte Datenstrukturen zu manipulieren.

Das Programm beginnt mit einem einzelnen Thread, der die Funktion `main` ausführt. Um Arbeit auf mehrere Prozessorkerne zu verteilen, kann jeder Thread weitere Threads anlegen, indem er einen *parallelen Abschnitt* erreicht. Bei dem Eintritt in einen solchen Abschnitt wird ein *Team* von Threads angelegt, und alle Threads des Teams bearbeiten die Befehle des Abschnitts nebenläufig. Damit eine sinnvolle Arbeitsteilung möglich ist, können die Threads entscheiden, unterschiedliche Befehle auszuführen.

Wenn alle Threads das Ende des Abschnitts erreicht haben, werden sie beendet und der ursprüngliche Thread führt die folgenden Befehle aus.

OpenMP verwendet die **#pragma**-Direktive des C-Compilers, um seinen Befehlsumfang zu erweitern. Dabei gilt die Konvention, dass ein Compiler, der OpenMP nicht unterstützt, diese Direktive einfach ignoriert. OpenMP ist so konzipiert, dass auch in diesem Fall das Programm noch ein korrektes Verhalten zeigen wird, es wird eben nur nicht parallel ausgeführt werden.

Parallele Abschnitte werden mit der Direktive **#pragma** `omp parallel` angelegt, die beispielsweise wie folgt eingesetzt werden kann:

```
int
main(int argc, char **argv)
{
  int i;

  i = 0;
  #pragma omp parallel
  {
    i++;
  }

  return i;
}
```

In diesem Programm wird die Anweisung `i++` von allen Threads des Teams ausgeführt, der parallele Abschnitt erstreckt sich von der zweiten öffnenden geschweiften Klammer bis zu deren schließendem Gegenstück.

Man könnte erwarten, dass dieses Programm die Anzahl der Threads im Team zurück gibt. Tatsächlich kann es jedes Ergebnis zwischen eins und der Anzahl der Threads zurück geben, weil nicht sicher gestellt ist, dass ein Thread mit der Veränderung der gemeinsam genutzten Variable `i` fertig ist, bevor andere sie lesen.

Falls alle Threads die Variable lesen, bevor ein anderer sie verändern konnte, werden alle den Wert null erhalten, um eins erhöhen, und eins in den Speicher schreiben.

OpenMP bietet verschiedene Möglichkeiten, um dieses Problem zu lösen. Die einfachste besteht darin, die Anweisung mit der Direktive **`#pragma`** `omp atomic` als *atomar*, also unteilbar, zu kennzeichnen. Dann sorgt der Prozessor dafür, dass kein Prozessorkern die Variable verändern kann, während ein anderer sie bearbeitet. Allerdings lassen sich nur wenige Anweisungen als atomar kennzeichnen, so dass diese Lösung nur eingeschränkt nützlich ist. Darüber hinaus kann sie abhängig von der verwendeten Prozessorarchitektur sehr ineffizient arbeiten.

Eine erheblich flexiblere, aber auch langsamere, Lösung besteht darin, *kritische Abschnitte* mit der Direktive **`#pragma`** `omp critical` zu definieren. Dann stellt OpenMP sicher, dass sich jeweils nur ein Thread in einem solchen Abschnitt befindet, so dass er ungestört Datenstrukturen modifizieren kann.

Erheblich besser ist es natürlich, wenn wir solche Konflikte von Anfang an vermeiden können, und bei Operationen aus der linearen Algebra sind wir glücklicherweise oft dazu in der Lage. Ein einfaches Beispiel ist die Realisierung der `axpy`-Funktion: Auf jede Komponente des Ergebnisvektors wird nur einmal lesend und einmal schreibend zugegriffen, und es gibt keine Wechselwirkungen zwischen den Komponenten.

Für solche günstigen Situationen bietet OpenMP die Direktive **`#pragma`** `omp` **`for`**, die eine unmittelbar auf die Direktive folgende **`for`**-Schleife auf alle Threads des aktuellen Teams verteilt:

```
void
axpy(int n, real alpha, const real *x, int incx,
     real *y, int incy)
{
  int i;

  #pragma omp parallel
  {
    #pragma omp for
    for(i=0; i<n; i++)
      y[i*incy] += alpha * x[i*incx];
  }
}
```

Da relativ häufig nur eine einzige **`for`**-Schleife zu parallelisieren ist, können die Direktive **`#pragma`** `omp parallel` für das Anlegen des parallelen Abschnitts und die Direkti-

ve **#pragma** omp **for** für die Verteilung der **for**-Schleife zu einer einzigen Direktive zusammengefasst werden:

```
void
axpy(int n, real alpha, const real *x, int incx,
     real *y, int incy)
{
  int i;

  #pragma omp parallel for
  for(i=0; i<n; i++)
    y[i*incy] += alpha * x[i*incx];
}
```

Dieser Zugang ist emfehlenswert, falls in der Schleife ausreichend viele Iterationen anfallen und darf nur angewendet werden, falls die Anzahl der Iterationen von Anfang an fest steht. Für das axpy-Beispiel genügt er also, für aufwendigere Aufgaben wie die Block-Matrix-Multiplikation oder die Block-LR-Zerlegung sind flexiblere Konstruktionen gefragt.

Ein sehr attraktives Konzept sind die mit Version 3.0 des OpenMP-Standards eingeführten *Tasks*, insbesondere seit in Version 4.0 des Standards die Möglichkeit eingeführt wurde, Abhängigkeiten zwischen Tasks definieren zu können.

Ein Task ist dabei eine Folge von Befehlen, die von *irgendeinem* Thread ausgeführt werden kann. Wir können also die zu lösende Aufgabe in kleinere Teilaufgaben zerlegen und diese Teilaufgaben durch das Laufzeitsystem in möglichst günstiger Weise auf die verfügbaren Threads verteilen lassen.

Als erstes Beispiel kann die blockweise Matrix-Multiplikation dienen:

```
#pragma omp parallel
#pragma omp single
{
  for(k=0; k<m; k++) {
    ok = n * k / m;
    dk = n * (k+1) / m - ok;

    for(i=0; i<m; i++) {
      oi = n * i / m;
      di = n * (i+1) / m - oi;

      for(j=0; j<m; j++) {
        oj = n * j / m;
        dj = n * (j+1) / m - oj;

        #pragma omp task depend(inout: C[oi+oj*ldC]), \
                         firstprivate(di,dj,dk,oi,oj,ok)
        gemm(false, false, di, dj, dk,
```

```
                alpha, A+oi+ok*ldA, ldA,
                       B+ok+oj*ldB, ldB,
                beta, C+oi+oj*ldC, ldC);
          }
        }
      }
    }
```

Wie bereits bekannt, wird mit **#pragma** omp parallel ein paralleler Abschnitt mit einem Team von Threads angelegt. Die Direktive **#pragma** omp single sorgt dann dafür, dass die **for**-Schleifen nur von einem einzigen Thread ausgeführt werden. Dieser Thread legt nun mit **#pragma** omp task Tasks für jeden Aufruf der gemm-Funktion an, und dieses Tasks werden durch das OpenMP-Laufzeitsystem auf die anderen Threads des Teams verteilt.

Jeder Task kann über lokale Variablen verfügen. Das Schlüsselwort firstprivate erlaubt es uns, eine Kopie einer gemeinsam genutzten Variablen anzulegen, die den Wert erhält, den sie zu dem Zeitpunkt der Erzeugung des Tasks hatte. In unserem Beispiel ist das wichtig, um dafür zu sorgen, dass die korrekten Werte der sich verändernden Schleifenvariablen verwendet werden.

Da wir nicht wissen, wie OpenMP die Tasks verteilen wird, insbesondere welche Tasks gleichzeitig bearbeitet werden, könnte es passieren, dass mehrere Threads gleichzeitig schreibend auf dieselbe Teilmatrix des Ergebnisses zugreifen wollen. Wie schon in unserem ersten Beispiel könnte das zu fehlerhaften Ergebnissen führen.

Dieses Problem lässt sich elegant lösen, indem wir mit dem Schlüsselwort depend Abhängigkeiten zwischen Tasks definieren. Ist eine Variable mit in oder inout markiert, kann der Task erst ausgeführt werden, wenn alle vorigen Tasks abgeschlossen wurden, bei denen sie mit out oder inout markiert ist.

In unserem Fall legen wir fest, dass der erste Eintrag C[oi+oj*ldC] des Blocks (i, j) sowohl als Ein- als auch als Ausgabe eines Tasks deklariert wird. Alle Tasks, die vorher auf diesen Eintrag zugreifen, müssen abgeschlossen sein, bevor der neue Task beginnen kann. Da die Blöcke in unserem Beispiel disjunkt sind, sind damit auch alle anderen Einträge des Blocks vor Störungen sicher.

Solange die Anzahl der Blöcke groß genug ist, gibt es auch mit dieser Abhängigkeit noch genug Arbeit, die auf Threads verteilt werden kann. Auf einem Vierkernprozessor des Typs Core™i7-3820 reduziert sich die Rechenzeit für die Matrix-Multiplikation von 26 Sekunden auf 6,2 Sekunden, die Laufzeit wird also tatsächlich um einen Faktor von ungefähr vier reduziert.

Wir können die auf Tasks basierende Parallelisierung auch verwenden, um etwas kompliziertere Algorithmen wie die blockweise LR-Zerlegung zu behandeln. Die bereits fertig gestellten Blöcke der Zerlegung verwenden wir dabei nur als in-Abhängigkeiten, damit sie als Eingabedaten für mehrere Tasks gleichzeitig verwendet werden können.

```
#pragma omp parallel
#pragma omp single
{
```

```
  for(k=0; k<m; k++) {
    ok = n * k / m;
    dk = n * (k+1) / m - ok;

    #pragma omp task depend(inout: A[ok+ok*ldA]), \
                     firstprivate(dk,ok)
    lrdecomp(dk, A+ok+ok*ldA, ldA);

    for(j=k+1; j<m; j++) {
      oj = n * j / m;
      dj = n * (j+1) / m - oj;

      #pragma omp task depend(in: A[ok+ok*ldA]), \
                       depend(inout: A[ok+oj*ldA]), \
                       firstprivate(dk,dj,ok,oj)
      block_lsolve(dk, dj, A+ok+ok*ldA, ldA,
                   A+ok+oj*ldA, ldA);

      #pragma omp task depend(in: A[ok+ok*ldA]),\
                       depend(inout: A[oj+ok*ldA]) \
                       firstprivate(dk,dj,ok,oj)
      block_rsolve_trans(dk, dj, A+ok+ok*ldA, ldA,
                         A+oj+ok*ldA, ldA);
    }

    for(j=k+1; j<m; j++) {
      oj = n * j / m;
      dj = n * (j+1) / m - oj;

      for(i=k+1; i<m; i++) {
        oi = n * i / m;
        di = n * (i+1) / m - oi;

        #pragma omp task depend(in: A[oi+ok*ldA], \
                                    A[ok+oj*ldA]), \
                         depend(inout: A[oi+oj*ldA]), \
                         firstprivate(di,dj,dk,oi,oj,ok)
        gemm(false, false, di, dj, dk,
             -1.0, A+oi+ok*ldA, ldA, A+ok+oj*ldA, ldA,
             1.0, A+oi+oj*ldA, ldA);
      }
    }
  }
}
```

2.7 Maschinenzahlen und Rundungsfehler

Die meisten reellen Zahlen lassen sich in einem Computer nicht exakt darstellen: Die Menge der reellen Zahlen ist unendlich, und ein Computer bietet nur endlich viel Speicherplatz. Deshalb werden solche Zahlen durch *Maschinenzahlen* angenähert. Ein weit verbreiteter Standard für solche Zahlen ist IEEE 754-1989, bei dem Zahlen im Wesentlichen in der Form

$$x = \sigma b^e \sum_{\ell=0}^{m} d_\ell b^{-\ell}$$

mit dem Vorzeichen $\sigma \in \{-1, 1\}$, dem Exponenten $e \in [-2^{r-1} - 2, 2^{r-1} - 1]$, und den Ziffern $d_0, \dots, d_m \in [0 : b - 1]$ dargestellt werden.

Die Menge der darstellbaren Zahlen ist durch die *Basis* $b \in \mathbb{N}_{\geq 2}$, die *Mantissenlänge* $m \in \mathbb{N}$ und die *Exponentengröße* r festgelegt.

Der IEEE-Standard verwendet als Basis grundsätzlich $b = 2$ und *normalisiert* die Zahl, sorgt also für $d_0 = 1$, damit der Exponent e eindeutig festgelegt ist und kein Platz für führende Nullen verschwendet wird. Das ist bei allen Zahlen mit Ausnahme der Null möglich, so dass für die Null ein spezieller Code vorgesehen wurde.

Jede reelle Zahl lässt sich in der Form

$$x = \sigma b^e \sum_{\ell=0}^{\infty} d_\ell b^{-\ell}$$

mit *unendlich* vielen Ziffern $(d_\ell)_{\ell=0}^{\infty}$ darstellen, und indem wir die Summe nach dem m-ten Term abschneiden, können wir eine Maschinenzahl $\tilde{x}$ definieren, die x approximiert. Diesen Algorithmus können wir etwas modifizieren, etwa indem wir kaufmännisch runden. Insgesamt gehört zu jeder Menge von Maschinenzahlen $\mathcal{M}_{m,b,r}$ eine *Kürzungsabbildung*

$$\mathrm{fl} \colon \mathbb{R} \to \mathcal{M}_{m,b,r},$$

die jeder reellen Zahl eine Näherung in der Menge $\mathcal{M}_{m,b,r}$ der Maschinenzahlen zuordnet.

Da der Exponent e abhängig von der Zahl x gewählt werden kann, und damit die Position des „Kommas“, fallen IEEE-Maschinenzahlen in die Kategorie der *Gleitkommazahlen* (engl. *floating point numbers*). Der im Vergleich zu Festkommazahlen (bei denen der Exponent immer derselbe ist und deshalb nicht abgespeichert werden muss) erheblich höhere Schaltungsaufwand ist dadurch gerechtfertigt, dass Gleitkommazahlen eine gewisse *relative Genauigkeit* erreichen: Mit der Zahl $\epsilon_{\mathrm{mach}} = 2^{-m-1}$ gilt (bei kaufmännischem Runden)

$$|x - \mathrm{fl}(x)| \leq \epsilon_{\mathrm{mach}} |x| \qquad \text{für alle } x \in \mathbb{R},$$

solange $|x|$ nicht so groß oder so klein wird, dass der Exponent e nicht mehr dargestellt werden kann.

Dieser Eigenschaft ist es zu verdanken, dass wir mit Gleitkommazahlen sowohl sehr große als auch sehr kleine Werte handhaben können, beispielsweise sowohl die Abstände zwischen Himmelskörpern als auch die zwischen Atomen.

Wann immer wir im Computer Rechenoperationen ausführen, müssen die Ergebnisse grundsätzlich durch Maschinenzahlen dargestellt werden, so dass beispielsweise die „echten“ Grundrechenarten durch

$$x \oplus y = \mathrm{fl}(x+y), \qquad x \ominus y = \mathrm{fl}(x-y), \qquad x \odot y = \mathrm{fl}(xy), \qquad x \oslash y = \mathrm{fl}(x/y)$$

ersetzt werden. Es entstehen also *Rundungsfehler*, die je nach Aufgabenstellung und Programm das Ergebnis der Berechnung erheblich verfälschen können.

Besonders kritisch kann dabei die *Fehlerfortpflanzung* sein: Ein Fehler in einem Zwischenergebnis beeinträchtigt alle folgenden Rechenoperationen. Bei ungünstig gestellten Aufgaben kann das Ergebnis dadurch völlig unbrauchbar werden.

Als Beispiel untersuchen wir die Matrizen

$$L^{(n)} := \begin{pmatrix} 1 & & & \\ -1 & 1 & & \\ \vdots & \ddots & \ddots & \\ -1 & \cdots & -1 & 1 \end{pmatrix} \qquad \text{für alle } n \in \mathbb{N}$$

und die Gleichungssysteme

$$L^{(n)} x^{(n)} = \begin{pmatrix} \alpha \\ \vdots \\ \alpha \end{pmatrix} \qquad \text{für alle } n \in \mathbb{N}$$

mit einer Zahl $\alpha \in \mathbb{R}$, die *keine* Maschinenzahl ist, sich aber durch $\tilde{\alpha} = \mathrm{fl}(\alpha)$ approximieren lässt.

Als akademisches Beispiel nehmen wir an, dass keine weiteren Rundungsfehler auftreten, dass wir also die Systeme

$$L^{(n)} \widetilde{x}^{(n)} = \begin{pmatrix} \tilde{\alpha} \\ \vdots \\ \tilde{\alpha} \end{pmatrix} \qquad \text{für alle } n \in \mathbb{N}$$

exakt auflösen können. In der Praxis werden selbstverständlich weitere Fehler während des Lösungsvorgangs hinzu kommen, aber das können wir zur Vereinfachung vernachlässigen.

Da wir es mit einem linearen Gleichungssystem zu tun haben, erfüllen die Fehler $e^{(n)} := \widetilde{x}^{(n)} - x^{(n)}$ die Gleichungssysteme

$$L^{(n)} e^{(n)} = \begin{pmatrix} \epsilon \\ \vdots \\ \epsilon \end{pmatrix} \qquad \text{für alle } n \in \mathbb{N}$$

mit $\epsilon := \tilde{\alpha} - \alpha$. Diese Systeme können wir durch Vorwärtseinsetzen auflösen: Wir haben

$$L^{(n)} = \left(\begin{array}{c|cccc} 1 & & & & \\ \hline -1 & 1 & & & \\ \vdots & -1 & \ddots & & \\ \vdots & \vdots & \ddots & \ddots & \\ -1 & -1 & \cdots & -1 & 1 \end{array}\right),$$

also folgen unmittelbar $e_1^{(n)} = \epsilon$ sowie im Fall $n > 1$ auch

$$L^{(n-1)} \begin{pmatrix} e_2^{(n)} \\ \vdots \\ e_n^{(n)} \end{pmatrix} = \begin{pmatrix} 2\epsilon \\ \vdots \\ 2\epsilon \end{pmatrix}.$$

Infolge der Linearität besitzt dieses Gleichungssystem die Lösung $2e^{(n-1)}$, so dass wir insgesamt

$$e^{(n)} = \begin{pmatrix} \epsilon \\ 2e^{(n-1)} \end{pmatrix} \qquad \text{für alle } n \in \mathbb{N}_{\geq 2}$$

erhalten. Der Fehler ist also von der Gestalt

$$e^{(n)} = \begin{pmatrix} \epsilon \\ 2\epsilon \\ 4\epsilon \\ \vdots \\ 2^{n-1}\epsilon \end{pmatrix} \qquad \text{für alle } n \in \mathbb{N}.$$

Obwohl die Matrizen $L^{(n)}$ und die rechten Seiten auf den ersten Blick sehr „gutartig“ aussehen, wird der Fehler ϵ während des Vorwärtseinsetzens *exponentiell* verstärkt, so dass schon bei $n = 100$ und IEEE-Gleitkommazahlen doppelter Genauigkeit kein brauchbares Ergebnis mehr zu erwarten ist.

2.8 Konditionszahl

Es ist sinnvoll, sich Gedanken darüber zu machen, woran es liegt, dass sich manche Gleichungssysteme „gutartig“ gegenüber Rundungsfehlern und sonstigen Störungen verhalten und andere nicht.

Ein erster Schritt bei der Untersuchung dieses Phänomens ist es, sich darüber Gedanken zu machen, wie wir Fehler überhaupt messen können. Zweckmäßig sind dafür *Normen*, die einem Element x eines $\mathbb{R}$-Vektorraums V eine nicht-negative Zahl $\|x\| \in \mathbb{R}_{\geq 0}$ zuordnen. Eine Norm ist durch die folgenden drei Eigenschaften charakterisiert:

$$x = 0 \iff \|x\| = 0 \qquad \text{für alle } x \in V, \tag{2.10a}$$

$$\|\alpha x\| = |\alpha| \, \|x\| \qquad \text{für alle } x \in V, \ \alpha \in \mathbb{R}, \tag{2.10b}$$

$$\|x + y\| \leq \|x\| + \|y\| \qquad \text{für alle } x, y \in V. \tag{2.10c}$$

Typische Beispiele für Normen auf $V = \mathbb{R}^n$ sind

- die *Maximumnorm* (oder ℓ^∞-Norm)

$$\|x\|_\infty = \max\{|x_i| \ : \ i \in [1:n]\} \qquad \text{für alle } x \in \mathbb{R}^n, \tag{2.11a}$$

- die *Manhattan-Norm* (oder ℓ^1-Norm)

$$\|x\|_1 = \sum_{i=1}^{n} |x_i| \qquad \text{für alle } x \in \mathbb{R}^n \tag{2.11b}$$

- und die *euklidische Norm* (oder ℓ^2-Norm)

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} \qquad \text{für alle } x \in \mathbb{R}^n. \tag{2.11c}$$

Wir können Normen als „Längenmaß“ für Vektoren interpretieren, beispielsweise korrespondiert die euklidische Norm nach dem Satz von PYTHAGORAS mit unserem alltäglichen Abstandsbegriff.

Um zu untersuchen, wie sich die Norm eines Vektors $x \in \mathbb{R}^n \setminus \{0\}$ verändert, wenn wir ihn mit einer Matrix $A \in \mathbb{R}^{n\times n}$ multiplizieren, bietet es sich an, das Verhältnis

$$\frac{\|Ax\|}{\|x\|}$$

zu untersuchen. Dieses Verhältnis wird in der Regel von x abhängen, so dass wir das Maximum betrachten sollten, um eine verlässliche Schranke für die durch die Matrix A verursachte „Längenänderung“ zu erhalten. Dieses Maximum bezeichnen wir mit

$$\|A\| := \max \left\{ \frac{\|Ax\|}{\|x\|} \ : \ x \in \mathbb{R}^n \setminus \{0\} \right\} \qquad \text{für alle } A \in \mathbb{R}^{n\times n}. \tag{2.12}$$

Es lässt sich relativ einfach nachrechnen, dass wir mit dieser Definition eine Norm auf dem Raum $\mathbb{R}^{n\times n}$ der Matrizen erhalten. Diese Norm nennen wir die *von der Vektornorm* $\|\cdot\|$ *induzierte Matrixnorm.*

Die von $\|\cdot\|_\infty$, $\|\cdot\|_1$ und $\|\cdot\|_2$ induzierte Matrixnormen bezeichnen wir ebenfalls mit $\|\cdot\|_\infty$, $\|\cdot\|_1$ und $\|\cdot\|_2$, da jeweils aus dem Kontext klar wird, ob die Vektornorm oder die Matrixnorm gemeint ist.

Die wichtigste Eigenschaft der induzierten Matrixnorm ist, dass sie mit der Vektornorm *verträglich* ist, dass also

$$\|Ax\| \leq \|A\| \, \|x\| \qquad \text{für alle } A \in \mathbb{R}^{n\times n}, \ x \in \mathbb{R}^n \tag{2.13}$$

gilt, wie sich leicht anhand der Definition nachprüfen lässt. Aus dieser Ungleichung ergibt sich unmittelbar die *Submultiplikativität*

$$\|AB\| \leq \|A\|\,\|B\| \qquad \text{für alle } A, B \in \mathbb{R}^{n\times n}. \tag{2.14}$$

Aus dieser Eigenschaft in Kombination mit der geometrischen Summenformel

$$\sum_{m=0}^{\infty} q^m = \frac{1}{1-q} \qquad \text{für alle } q \in \mathbb{R} \text{ mit } |q| < 1$$

wiederum ergibt sich die folgende Aussage über die NEUMANN*sche Reihe*:

Satz 2.12 (Neumannsche Reihe) *Sei $X \in \mathbb{R}^{n\times n}$ eine Matrix mit $\|X\| < 1$. Dann ist $I - X$ invertierbar und es gelten*

$$\sum_{m=0}^{\infty} X^m = (I - X)^{-1}, \qquad \|(I-X)^{-1}\| \leq \frac{1}{1 - \|X\|}. \tag{2.15}$$

Falls wir also die Identität moderat stören, erhalten wir wieder eine invertierbare Matrix und können sogar deren Norm abschätzen.

Nun können wir uns der Empfindlichkeit linearer Gleichungssysteme gegenüber Störungen widmen.

Satz 2.13 (Störungen linearer Gleichungssysteme) *Seien Matrizen $A, \widetilde{A} \in \mathbb{R}^{n\times n}$ und Vektoren $b, \widetilde{b} \in \mathbb{R}^n$ gegeben. Wir nehmen an, dass Lösungen $x, \widetilde{x} \in \mathbb{R}^n$ der Gleichungssysteme*

$$Ax = b, \qquad \widetilde{A}\widetilde{x} = \widetilde{b}$$

gegeben sind. Falls A invertierbar ist und

$$\|A^{-1}(A - \widetilde{A})\| < 1$$

gilt, ist auch $\widetilde{A}$ invertierbar. Falls $b \neq 0$ gilt, haben wir

$$\frac{\|x - \widetilde{x}\|}{\|x\|} \leq \frac{\|A\|\,\|A^{-1}\|}{1 - \|A^{-1}(A - \widetilde{A})\|} \left(\frac{\|b - \widetilde{b}\|}{\|b\|} + \frac{\|A - \widetilde{A}\|}{\|A\|} \right).$$

Beweis. Sei A invertierbar. Wir definieren $X := I - A^{-1}\widetilde{A}$ und stellen

$$\|X\| = \|I - A^{-1}\widetilde{A}\| = \|A^{-1}(A - \widetilde{A})\| < 1$$

fest. Also dürfen wir Satz 2.12 anwenden, um zu folgern, dass

$$I - X = A^{-1}\widetilde{A}$$

invertierbar ist, also auch $\widetilde{A}$.

Indem wir die Verträglichkeit (2.13), die Submultiplikativität (2.14) sowie die Abschätzung (2.15) ausnutzen, erhalten wir schließlich

$$\begin{aligned}
(I - X)(x - \widetilde{x}) &= A^{-1}\widetilde{A}(x - \widetilde{x}) = A^{-1}(Ax + (\widetilde{A} - A)x - \widetilde{b}) \\
&= A^{-1}(b - \widetilde{b} + (\widetilde{A} - A)x), \\
x - \widetilde{x} &= (I - X)^{-1}A^{-1}(b - \widetilde{b} + (\widetilde{A} - A)x), \\
\|x - \widetilde{x}\| &\le \frac{\|A^{-1}\|}{1 - \|X\|}(\|b - \widetilde{b}\| + \|\widetilde{A} - A\|\,\|x\|) \\
&\le \frac{\|A^{-1}\|}{1 - \|X\|}\left(\frac{\|b - \widetilde{b}\|}{\|b\|}\|Ax\| + \|\widetilde{A} - A\|\,\|x\|\right) \\
&\le \frac{\|A^{-1}\|}{1 - \|X\|}\left(\frac{\|b - \widetilde{b}\|}{\|b\|}\|A\|\,\|x\| + \|\widetilde{A} - A\|\,\|x\|\right) \\
&= \frac{\|A\|\,\|A^{-1}\|}{1 - \|X\|}\left(\frac{\|b - \widetilde{b}\|}{\|b\|} + \frac{\|A - \widetilde{A}\|}{\|A\|}\right)\|x\|.
\end{aligned}$$

Aus $b \neq 0$ folgt $x \neq 0$, so dass wir auf beiden Seiten der Gleichung durch $\|x\|$ dividieren können, um die gewünschte Abschätzung zu erhalten. ∎

Für die Verstärkung des *relativen* Fehlers der rechten Seite b und der Matrix A ist also in erster Linie die *Konditionszahl*

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

der Matrix A entscheidend. Wegen

$$\|A^{-1}(A - \widetilde{A})\| \le \|A\|\,\|A^{-1}\|\,\frac{\|A - \widetilde{A}\|}{\|A\|} = \kappa(A)\frac{\|A - \widetilde{A}\|}{\|A\|}$$

nimmt die Aussage des Störungssatzes 2.13 die Form

$$\frac{\|x - \widetilde{x}\|}{\|x\|} \le \frac{\kappa(A)}{1 - \kappa(A)\frac{\|A - \widetilde{A}\|}{\|A\|}}\left(\frac{\|b - \widetilde{b}\|}{\|b\|} + \frac{\|A - \widetilde{A}\|}{\|A\|}\right) \tag{2.16}$$

an, die die Bedeutung der Konditionszahl deutlich macht.

2.9 QR-Zerlegung

Wie wir in (2.16) gesehen haben, ist damit zu rechnen, dass die Konditionszahl $\kappa(A)$ einen entscheidenden Einfluss auf die Verstärkung von Rundungsfehlern hat.

Wenn wir das Gleichungssystem $Ax = b$ mittels der LR-Zerlegung lösen, lösen wir tatsächlich die Systeme

$$Ly = b, \qquad Rx = y,$$

so dass den Konditionszahlen $\kappa(L)$ und $\kappa(R)$ eine große Bedeutung zukommt. Mit Hilfe der Submultiplikativität (2.14) erhalten wir

$$\kappa(A) = \|A\| \, \|A^{-1}\| = \|LR\| \, \|R^{-1}L^{-1}\| \leq \|L\| \, \|R\| \, \|R^{-1}\| \, \|L^{-1}\| = \kappa(L)\kappa(R),$$

aber leider nicht die umgekehrte Abschätzung, so dass die beiden Faktoren L und R erheblich größere Konditionszahlen als die ursprüngliche Matrix aufweisen können.

Es wäre günstiger, wenn wir eine Faktorisierung finden könnten, die die Konditionszahl unverändert lässt. Dreieckszerlegungen leisten das im Allgemeinen nicht, aber *orthogonale* Zerlegungen schon.

Definition 2.14 (Transponierte Matrix) *Sei $A \in \mathbb{R}^{n\times m}$. Die durch*

$$b_{ij} = a_{ji} \qquad \text{für alle } i \in [1:m],\ j \in [1:n]$$

definierte Matrix $B \in \mathbb{R}^{m\times n}$ bezeichnen wir als die Transponierte *der Matrix A und schreiben sie als $A^* := B$ (in der Literatur gelegentlich auch als A^T).*

Definition 2.15 (Orthogonale Matrix) *Eine Matrix $Q \in \mathbb{R}^{n\times m}$ nennen wir* isometrisch*, falls $Q^*Q = I$ gilt.*

Eine isometrische Matrix $Q \in \mathbb{R}^{n\times m}$ nennen wir orthogonal*, falls $n = m$ gilt.*

Offenbar ist für jede isometrische Matrix $Q \in \mathbb{R}^{n\times m}$ die transponierte Matrix Q^* eine Linksinverse, also muss Q insbesondere injektiv sein.

Falls $Q \in \mathbb{R}^{n\times n}$ orthogonal ist, handelt es sich um eine injektive lineare Abbildung von $\mathbb{R}^n$ nach $\mathbb{R}^n$, die nach Dimensionssatz auch surjektiv sein muss. Also ist die Linksinverse auch eine Rechtsinverse und es gilt

$$Q^{-1} = Q^* \qquad \text{für alle orthogonalen } Q \in \mathbb{R}^{n\times n}.$$

Ihren Namen verdienen sich isometrische Matrizen mit ihrer besonderen Beziehung zu der EUKLIDischen Norm

$$\|x\|_2 = \left(\sum_{i=1}^{n} |x_i|^2 \right)^{1/2} \qquad \text{für alle } x \in \mathbb{R}^n.$$

Diese Norm ergibt sich aus dem EUKLIDischen Skalarprodukt

$$\langle x, y\rangle_2 = \sum_{i=1}^{n} x_i y_i \qquad \text{für alle } x, y \in \mathbb{R}^n$$

mittels der Gleichung

$$\|x\|_2 = \sqrt{\langle x, x\rangle_2} \qquad \text{für alle } x \in \mathbb{R}^n.$$

Transponierte Matrizen stehen in enger Beziehung zu dem Skalarprodukt, es gilt nämlich

$$\langle x, By\rangle_2 = \langle B^*x, y\rangle_2 \qquad \text{für alle } B \in \mathbb{R}^{n\times m},\ x \in \mathbb{R}^n,\ y \in \mathbb{R}^m. \tag{2.17}$$

Wenn nun $Q \in \mathbb{R}^{n\times m}$ eine isometrische Matrix ist, erhalten wir

$$\|x\|_2 = \sqrt{\langle x, x\rangle_2} = \sqrt{\langle Q^*Qx, x\rangle_2} = \sqrt{\langle Qx, Qx\rangle_2} = \|Qx\|_2 \qquad \text{für alle } x \in \mathbb{R}^n, \quad (2.18)$$

isometrische Matrizen lassen also die EUKLIDische Norm unverändert. Ein Blick auf die Definition (2.12) der induzierten Matrixnorm führt sofort zu der Gleichung $\|Q\|_2 = 1$.

Definition 2.16 (QR-Zerlegung) *Sei $A \in \mathbb{R}^{n\times m}$. Sei $Q \in \mathbb{R}^{n\times n}$ eine orthogonale Matrix und $R \in \mathbb{R}^{n\times m}$ eine rechte obere Dreiecksmatrix. Falls*

$$A = QR \quad (2.19)$$

gilt, nennen wir das Paar (Q, R) eine QR-Zerlegung *der Matrix A.*

Eine QR-Zerlegung weist in Hinblick auf die Konditionszahl erheblich bessere Eigenschaften als eine LR-Zerlegung auf: Sei (Q, R) eine QR-Zerlegung einer invertierbaren Matrix $A \in \mathbb{R}^{n\times n}$. Wegen (2.18) gelten

$$\begin{aligned}
\|A\|_2 &= \max\left\{\frac{\|Ax\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^n \setminus \{0\}\right\} = \max\left\{\frac{\|QRx\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^n \setminus \{0\}\right\} \\
&= \max\left\{\frac{\|Rx\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^n \setminus \{0\}\right\} = \|R\|_2, \\
\|A^{-1}\|_2 &= \max\left\{\frac{\|A^{-1}x\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^n \setminus \{0\}\right\} = \max\left\{\frac{\|y\|_2}{\|Ay\|_2} \;:\; y \in \mathbb{R}^n \setminus \{0\}\right\} \\
&= \max\left\{\frac{\|y\|_2}{\|QRy\|_2} \;:\; y \in \mathbb{R}^n \setminus \{0\}\right\} = \max\left\{\frac{\|y\|_2}{\|Ry\|_2} \;:\; y \in \mathbb{R}^n \setminus \{0\}\right\} \\
&= \max\left\{\frac{\|R^{-1}z\|_2}{\|z\|_2} \;:\; z \in \mathbb{R}^n \setminus \{0\}\right\} = \|R^{-1}\|_2,
\end{aligned}$$

so dass wir

$$\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \|R\|_2\|R^{-1}\|_2 = \kappa_2(R), \qquad \kappa_2(Q) = \|Q\|_2\|Q^{-1}\|_2 = 1$$

erhalten. Die QR-Zerlegung führt also zu keiner Verschlechterung der von der EUKLIDischen Norm induzierten Konditionszahl.

Auch eine QR-Zerlegung können wir verwenden, um lineare Gleichungssysteme zu lösen: Es gilt

$$Ax = b \iff QRx = b \iff (Qy = b \wedge Rx = y),$$

und wegen $Q^{-1} = Q^*$ können wir $y = Q^*b$ direkt berechnen und dann das System $Rx = y$ wie gehabt durch Rückwärtseinsetzen lösen.

Es stellt sich die Frage, ob wir zu einer gegebenen Matrix $A \in \mathbb{R}^{n\times m}$ eine QR-Zerlegung praktisch konstruieren können. Als Inspiration können wir die LR-Zerlegung heran ziehen: In ihrem ersten Schritt wenden wir eine linke untere Dreiecksmatrix L_1 an, um dafür zu sorgen, dass in der ersten Spalte der neuen Matrix L_1A alle Einträge unterhalb der Diagonalen verschwinden. Anschließend verfahren wir entsprechend mit den verbliebenen Spalten.

Falls wir also eine orthogonale Matrix $Q_1 \in \mathbb{R}^{n\times n}$ finden können, die in der ersten Spalte der Matrix Q_1A alle Einträge unterhalb der Diagonalen eliminiert, könnten wir analog vorgehen. Ein guter Ansatz sind *Householder-Spiegelungen* der Form

$$Q_v = I - 2\frac{vv^*}{v^*v} \tag{2.20}$$

für einen Vektor $v \in \mathbb{R}^n \setminus \{0\}$, bei denen wir im Interesse einer kompakten Schreibweise den Vektor $v \in \mathbb{R}^n$ in der naheliegenden Weise mit der Matrix $v \in \mathbb{R}^{n\times 1}$ identifizieren und $v^*v \in \mathbb{R}^{1\times 1}$ mit dem Skalar $\langle v, v\rangle_2 = \|v\|_2^2$.

Wegen $(vv^*)^* = vv^*$ haben wir

$$Q_v^*Q_v = \left(I - 2\frac{vv^*}{v^*v}\right)\left(I - 2\frac{vv^*}{v^*v}\right) = I - 4\frac{vv^*}{v^*v} + 4\frac{vv^*vv^*}{(v^*v)^2} = I - 4\frac{vv^*}{v^*v} + 4\frac{vv^*}{v^*v} = I,$$

also sind Householder-Spiegelungen tatsächlich orthogonal.

Wir müssen allerdings noch überprüfen, dass wir tatsächlich eine Householder-Spiegelung finden können, die Einträge unterhalb der Diagonalen eliminiert. Wir bezeichnen dazu die erste Spalte der Matrix A mit

$$a := \begin{pmatrix} a_{11} \\ \vdots \\ a_{n1} \end{pmatrix}$$

und den ersten kanonischen Einheitsvektor mit

$$\delta := \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Unsere Aufgabe besteht darin, einen Vektor $v \in \mathbb{R}^n \setminus \{0\}$ so zu finden, dass

$$Q_v a = \alpha\delta = \begin{pmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

mit einem geeigneten $\alpha \in \mathbb{R}$ gilt, denn dann sind alle Koeffizienten der ersten Spalte a unterhalb der Diagonalen eliminiert.

Wir setzen die Definition der Householder-Spiegelung ein und erhalten

$$\begin{aligned} \alpha\delta &\stackrel{!}{=} Q_v a = a - 2\frac{vv^*}{v^*v}a = a - 2v\frac{\langle v, a\rangle_2}{\|v\|_2^2}, \\ \alpha\delta - a &\stackrel{!}{=} -2v\frac{\langle v, a\rangle_2}{\|v\|_2^2}. \end{aligned}$$

Wir sehen, dass wir auf der rechten Seite den Vektor v mit einem beliebigen Faktor $\lambda \in \mathbb{R} \setminus \{0\}$ multiplizieren können, ohne die Gleichung zu verändern, denn dann tritt sowohl im Zähler als auch im Nenner $\lambda^2 \neq 0$ auf, und beide Terme heben sich auf. Also können wir in unserer Gleichung ebenfalls den Skalierungsfaktor streichen und erhalten

$$a - \alpha\delta = v.$$

Wir müssen nur noch α bestimmen. Da $\|a\|_2 = \|Q_v a\|_2 = \|\alpha\delta\|_2 = |\alpha|$ gilt, bleibt uns nur die Wahl $\alpha \in \{\|a\|_2, -\|a\|_2\}$, die zu

$$v = a - \|a\|_2\delta = \begin{pmatrix} a_1 - \|a\|_2 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} \qquad \text{oder} \qquad v = a + \|a\|_2\delta = \begin{pmatrix} a_1 + \|a\|_2 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

führt. Um Rundungsfehler möglichst gering zu halten, empfiehlt es sich, das Vorzeichen von α so zu wählen, dass es mit dem von $-a_1$ übereinstimmt, also als $\alpha = -\operatorname{sgn}(a_1)\|a\|_2$. Es folgt

$$v = \begin{pmatrix} a_1 + \operatorname{sgn}(a_1)\|a\|_2 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}, \qquad Q_v a = \alpha\delta = \begin{pmatrix} -\operatorname{sgn}(a_1)\|a\|_2 \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Für unseren Algorithmus benötigen wir auch die Norm des Vektors v, die wir mittels

$$\begin{aligned} \|v\|^2 &= \langle v, v\rangle_2 = \langle a + \operatorname{sgn}(a_1)\|a\|_2\delta, a + \operatorname{sgn}(a_1)\|a\|_2\delta\rangle_2 \\ &= \|a\|_2^2 + 2\operatorname{sgn}(a_1)\|a\|_2 a_1 + \operatorname{sgn}(a_1)^2\|a\|_2^2 = 2(\|a\|_2^2 - \alpha a_1) \end{aligned}$$

einfach aus der Norm des Vektors a und seinem ersten Element gewinnen können, so dass die Householder-Spiegelung die Form

$$Q_v = I - 2\frac{vv^*}{v^*v} = I - \tau vv^* \qquad \text{mit} \qquad \tau := \frac{2}{\|v\|_2^2} = \frac{1}{\|a\|_2^2 - \alpha a_1}$$

annimmt. Eine QR-Zerlegung einer beliebigen Matrix lässt sich dann mit dem folgenden Algorithmus konstruieren:

- Berechne $\|a\|_2$.
- $\alpha \leftarrow -\operatorname{sgn}(a_1)\|a\|_2$, $\tau \leftarrow 1/(\|a\|_2^2 - \alpha a_1)$, $v \leftarrow a - \alpha\delta$.
- $B \leftarrow Q_v A = \begin{pmatrix} \alpha & B_{1*} \\ 0 & B_{**} \end{pmatrix}$.
- Wende den Algorithmus auf die Teilmatrix B_{**} an.

Ein Sonderfall ist dabei zu berücksichtigen: Falls $a = 0$ ist, erhalten wir $v = 0$, so dass wir nicht durch $\|v\|_2^2$ dividieren können. In diesem Fall sind allerdings schon alle Koeffizienten des Vektors a gleich null, so dass es egal ist, welchen Householder-Vektor wir wählen. Die einfachste Wahl ist $v = \delta$. Alternativ können wir in diesem Fall auch darauf verzichten, überhaupt eine Spiegelung anzuwenden.

Bemerkung 2.17 (In-place-Darstellung) *Auch im Fall der QR-Zerlegung können wir mit einem Trick die Matrix A direkt mit der QR-Zerlegung überschreiben. Da die Skalierung der Householder-Vektoren keine Rolle spielt, können wir dafür sorgen, dass $v_1 = 1$ gilt, so dass wir die erste Komponente eines Householder-Vektors jeweils nicht abzuspeichern brauchen.*

Dann können wir die Matrix R wie gehabt im rechten oberen Dreiecksteil der Matrix abspeichern und die Householder-Vektoren (ohne ihre erste Komponente) unterhalb der Diagonalen. Wenn wir mit $v^{(1)}, \dots, v^{(n)}$ die Householder-Vektoren bezeichnen, erhalten wir

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \to \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ v_2^{(1)} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_n^{(1)} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

$$\to \cdots \to \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ v_2^{(1)} & r_{22} & \cdots & r_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ v_n^{(1)} & \cdots & v_2^{(n-1)} & r_{nn} \end{pmatrix}.$$

Um unnötige Arbeit bei der Berechnung der Faktoren $\tau_i = 2/\|v^{(i)}\|_2^2$ zu sparen, bietet es sich natürlich an, diese Faktoren zusätzlich abzuspeichern.

Bemerkung 2.18 (Blockvariante) *Auch bei der QR-Zerlegung sind wir natürlich daran interessiert, BLAS-Funktionen der Stufe 3 zu verwenden, um eine möglichst hohe Effizienz zu erreichen.*

Dieses Ziel können wir erreichen, indem wir mehrere Householder-Spiegelungen zusammenfassen: Seien $v_1, \dots, v_k \in \mathbb{R}^n \setminus \{0\}$ und $\tau_1, \dots, \tau_k \in \mathbb{R}$ gegeben, die die Householder-Spiegelungen $Q_{v_\ell} = I - \tau_\ell v_\ell v_\ell^*$ definieren. Wir konstruieren für alle $\ell \in [1 : k]$ Matrizen $W_\ell \in \mathbb{R}^{n\times\ell}$ und $V_\ell \in \mathbb{R}^{n\times\ell}$ mit

$$Q_{v_\ell} \cdots Q_{v_1} = I - W_\ell V_\ell^*.$$

Für $\ell = 1$ ist diese Aufgabe einfach zu lösen, indem wir $W_1 = v_1\tau_1$ und $V_1 = v_1$ setzen, wobei wieder Vektoren aus $\mathbb{R}^n$ mit Matrizen aus $\mathbb{R}^{n\times 1}$ identifiziert werden.

Wenn wir für ein $\ell \in [1 : k-1]$ die Matrizen W_ℓ und V_ℓ konstruiert haben, erhalten wir mit

$$Q_{v_{\ell+1}} Q_{v_\ell} \cdots Q_{v_1} = Q_{v_{\ell+1}}(I - W_\ell V_\ell^*) = I - \tau_{\ell+1} v_{\ell+1} v_{\ell+1}^* - Q_{v_{\ell+1}} W_\ell V_\ell^*$$

$$= I - \begin{pmatrix} Q_{v_{\ell+1}} W_\ell & \tau_{\ell+1} v_{\ell+1} \end{pmatrix} \begin{pmatrix} V_\ell^* \\ v_{\ell+1}^* \end{pmatrix}$$

die Gleichungen

$$W_{\ell+1} := \begin{pmatrix} Q_{v_{\ell+1}} W_\ell & \tau_{\ell+1} v_{\ell+1} \end{pmatrix}, \qquad V_{\ell+1} := \begin{pmatrix} V_\ell & v_{\ell+1} \end{pmatrix}.$$

Wir brauchen also lediglich die neue Householder-Spiegelung $Q_{v_{\ell+1}}$ auf die bisherige Matrix W_ℓ anzuwenden und jeweils eine Spalte hinzu zu fügen, um die neuen Matrizen $W_{\ell+1}$ und $V_{\ell+1}$ zu erhalten.

Wenn uns W_k und V_k vorliegen, können wir die k Householder-Spiegelungen auf eine komplette Matrix A anwenden, indem wir erst die Hilfsmatrix $B := V_k^* A$ und dann das Update $A \leftarrow A - W_k B$ mit der Funktion `gemm` *berechnen. Dafür benötigen wir zwar etwas zusätzlichen Speicher, gewinnen aber unter Umständen erheblich an Geschwindigkeit.*

3 Iterationen

Nicht alle Aufgabenstellungen führen zu linearen Gleichungssystemen, die sich mit den im vorigen Kapitel diskutierten Methoden behandeln lassen.

Bei den meisten nicht-linearen Gleichungen ist es nicht mehr möglich, die exakte Lösung in einer endlichen Anzahl von Operationen zu ermitteln: Nach dem Satz von Abel-Ruffini existieren beispielsweise Polynome fünften Grades, deren Nullstellen sich nicht durch Wurzelausdrücke darstellen lassen. Also können diese Nullstellen auch nicht durch Grundrechenarten und Wurzeloperationen ermittelt werden.

In derartigen Situationen kommen *Iterationsverfahren* zum Einsatz, die eine Folge von Näherungslösungen konstruieren, die hoffentlich schnell gegen die exakte Lösung konvergieren. Indem wir nach einer gewissen Anzahl von Iterationen abbrechen, können wir eine beliebig genaue Approximation der Lösung erhalten.

3.1 Eigenwertprobleme

Als einführendes Beispiel untersuchen wir eine Verallgemeinerung der bisher betrachteten linearen Gleichungssysteme: Wir suchen nach einem *Eigenwert* einer Matrix $A \in \mathbb{R}^{n\times n}$, also nach einer Zahl $\lambda \in \mathbb{R}$ derart, dass ein Vektor $e \in \mathbb{R}^n \setminus \{0\}$ mit

$$Ae = \lambda e \tag{3.1}$$

existiert. Ein solcher Vektor heißt *Eigenvektor* zu der Matrix A und dem Eigenwert λ.

Die Gleichung (3.1) ist offenbar äquivalent zu

$$(\lambda I - A)e = 0.$$

Falls $\lambda I - A$ injektiv wäre, also invertierbar, würde aus dieser Gleichung bereits $e = 0$ folgen, aber Eigenvektoren müssen ungleich null sein.

Also ist λ genau dann ein Eigenwert, wenn die Matrix $\lambda I - A$ nicht invertierbar ist. Eine Matrix ist genau dann nicht invertierbar, wenn ihre Determinante gleich null ist, also sind die Eigenwerte gerade die Nullstellen des *charakteristischen Polynoms*

$$p_a \colon \mathbb{R} \to \mathbb{R}, \qquad \lambda \mapsto \det(\lambda I - A).$$

Daraus ergibt sich beispielsweise, dass die Matrix

$$A := \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

keine reellen Eigenwerte besitzt, da das charakteristische Polynome $p_A(\lambda) = \lambda^2 + 1$ keine reellen Nullstellen besitzt.

Die Matrix

$$A := \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

besitzt genau einen Eigenwert $\lambda = 1$, da das charakteristische Polynom $p_A(\lambda) = (\lambda - 1)^2$ eine doppelte Nullstelle besitzt.

Schließlich besitzt die Matrix

$$A := \begin{pmatrix} 1 & 10 \\ 0 & 2 \end{pmatrix}$$

die zwei Eigenwerte $\lambda_1 = 1$ und $\lambda_2 = 2$, die die Nullstellen des charakteristischen Polynoms $p_A(\lambda) = (\lambda - 1)(\lambda - 2)$ sind.

Bemerkung 3.1 (Begleitmatrix) *Das charakteristische Polynom einer Matrix $A \in \mathbb{R}^{n\times n}$ ist immer ein Polynom n-ten Grades, bei dem der n-te Koeffizient gleich eins ist. Umgekehrt lässt sich mit dem* Laplace*schen Entwicklungssatz zeigen, dass für ein Polynom*

$$q(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1} + x^n$$

die Frobenius-Begleitmatrix

$$B := \begin{pmatrix} 0 & 0 & \cdots & 0 & -b_0 \\ 1 & 0 & \cdots & 0 & -b_1 \\ 0 & 1 & \ddots & \vdots & -b_2 \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & -b_{n-1} \end{pmatrix}$$

gerade $p_B = q$ erfüllt.

Die Suche nach Eigenwerten einer Matrix und die nach den Nullstellen eines Polynoms sind also äquivalente Aufgaben. Der Satz von Abel und Ruffini besagt, dass es Polynome fünften Grades gibt, deren Nullstellen sich nicht durch die Grundrechenarten und Wurzelfunktionen ausdrücken lassen.

Da typische Prozessoren für Gleitkommazahlen in der Regel nur die Grundrechenarten und einige spezielle Operationen wie die Approximation der Quadraturwurzel unterstützen, haben wir keine Chance, die Nullstellen eines solchen Polynoms mit endlich vielen Operationen zu berechnen. Dementsprechend kann es auch keinen Algorithmus geben, der die Eigenwerte einer beliebigen Matrix in endlich vielen Operationen berechnet.

Bei derartigen Aufgabenstellungen können uns *Iterationsverfahren* helfen: Statt direkt die Lösung zu berechnen, zielen sie lediglich darauf, eine vorliegende Näherungslösung zu verbessern.

Ein Beispiel für diese Vorgehensweise ist die *Vektoriteration*, auch als die Von-Mises-Iteration bekannt. Angenommen, wir kennen Eigenwerte $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ mit zugehörigen Eigenvektoren $e_1, \ldots, e_n \in \mathbb{R}^n \setminus \{0\}$ einer Matrix $A \in \mathbb{R}^{n\times n}$. Wir gehen von dem Vektor

$$x^{(0)} = e_1 + e_2 + \ldots + e_n$$

aus und multiplizieren mit A, um weitere Vektoren

$$x^{(m+1)} = Ax^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0 \tag{3.2}$$

zu konstruieren. Da $e_1, \ldots, e_n$ Eigenvektoren sind, gilt

$$x^{(1)} = Ax^{(0)} = Ae_1 + Ae_2 + \ldots + Ae_n = \lambda_1 e_1 + \lambda_2 e_2 + \ldots + \lambda_n e_n,$$

und mit einer einfachen Induktion erhalten wir

$$x^{(m)} = \lambda_1^m e_1 + \lambda_2^m e_2 + \ldots + \lambda_n^m e_n \qquad \text{für alle } m \in \mathbb{N}_0.$$

Falls nun einer der Eigenwerte im Betrag größer als der andere ist, falls beispielsweise

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$$

gilt, bedeutet diese Gleichung, dass sich für großes m der Anteil des ersten Eigenvektors e_1 gegenüber allen anderen „durchsetzen“ wird.

Damit beschreibt die Gleichung (3.2) bereits ein erstes einfaches Iterationsverfahren für die Approximation eines Eigenvektors.

Diese Fassung ist allerdings nicht praxistauglich: Falls $|\lambda_1| > 1$ gilt, wird die Norm der Vektoren $x^{(m)}$ im Zuge der Iteration immer weiter wachsen, bis die für ihre Darstellung verwendeten Gleitkommazahlen überlaufen und wir kein brauchbares Resultat mehr erhalten. Falls dagegen $|\lambda_1| < 1$ gilt, wird die Norm so lange schrumpfen, bis in der Gleitkommazahldarstellung auf null abgerundet wird und wir ebenfalls kein Ergebnis finden, mit dem wir etwas anfangen können.

Beide Probleme lassen sich glücklicherweise einfach lösen: Da wir lediglich einen Eigenvektor suchen, können wir in jedem Schritt des Verfahrens die Vektoren so skalieren, dass Über- und Unterlauf vermieden werden, beispielsweise indem wir dafür sorgen, dass die Vektoren immer Einheitsvektoren bezüglich einer geeigneten Norm bleiben:

$$\widetilde{x}^{(m+1)} := Ax^{(m)}, \tag{3.3a}$$

$$x^{(m+1)} := \widetilde{x}^{(m+1)} / \|\widetilde{x}^{(m+1)}\| \qquad \text{für alle } m \in \mathbb{N}_0. \tag{3.3b}$$

Damit haben wir einen Algorithmus gefunden, bei dem wir darauf hoffen dürfen, dass er „nach unendlich vielen Schritten“ einen Eigenvektor finden wird. Selbstverständlich wollen wir nicht unendlich lange warten, also bietet es sich an, die Iteration abzubrechen, sobald eine hinreichend genaue Näherung gefunden wurde.

Im Prinzip könnten wir dazu die Norm

$$\|Ax^{(m)} - \lambda_1 x^{(m)}\|$$

verwenden: Falls sie gleich null ist, ist $x^{(m)}$ ein Eigenvektor. Falls sie hinreichend klein ist, lässt sich zeigen, dass $x^{(m)}$ in einem geeigneten Sinn immer noch eine gute Näherung darstellt.

Leider kennen wir in der Praxis λ_1 häufig nicht, so dass wir uns mit einer weiteren Näherung behelfen müssen. Glücklicherweise ist es relativ einfach, aus einer Näherung

```
procedure vectorit(A, ϵ, var x);
x̃ ← Ax;
λ̃ ← ⟨x, x̃⟩₂/⟨x, x⟩₂;
while ‖x̃ − λ̃x‖ > ϵ|λ̃| do begin
  x ← x̃/‖x̃‖;
  x̃ ← Ax;
  λ̃ ← ⟨x, x̃⟩₂/⟨x, x⟩₂
end
```

Abbildung 3.1: Vektoriteration

$x^{(m)}$ des Eigenvektors e_1 zu einer Näherung $\tilde{\lambda}_1$ des Eigenwerts λ_1 zu gelangen: Da e_1 ein Eigenvektor ist, gelten

$$\begin{aligned} Ae_1 &= \lambda_1 e_1, \\ \langle e_1, Ae_1\rangle_2 &= \lambda_1 \langle e_1, e_1\rangle_2, \\ \frac{\langle e_1, Ae_1\rangle_2}{\langle e_1, e_1\rangle_2} &= \lambda_1, \end{aligned}$$

so dass für einen *exakten* Eigenvektor e_1 der exakte Eigenwert λ_1 mit Hilfe des links stehenden Quotienten rekonstruiert werden kann.

Dieser RAYLEIGH-Quotient lässt sich selbstverständlich auch für alle anderen von null verschiedenen Vektoren definieren, so dass wir

$$\Lambda_A(x) := \frac{\langle x, Ax\rangle_2}{\langle x, x\rangle_2} \qquad \text{für alle } x \in \mathbb{R}^n \setminus \{0\}$$

erhalten. Indem wir $x^{(m)}$ einsetzen, gewinnen wir genäherte Eigenwerte. Die resultierende praktisch einsetzbare Vektoriteration ist in Abbildung 3.1 zusammengefasst.

Dabei ist das Abbruchkriterium gerade so gewählt, dass bei Verwendung der euklidischen Norm eine relative Genauigkeit von $\frac{\epsilon}{1-\epsilon}$ garantiert wird. Auf den Beweis dieser Aussage verzichten wir an dieser Stelle.

Ein großer Vorteil der Vektoriteration besteht darin, dass die Matrix A nicht explizit vorzuliegen braucht, denn der Algorithmus muss nur dazu in der Lage sein, sie mit einem Vektor zu multiplizieren.

Das eröffnet uns sehr weitreichende Möglichkeiten, den Algorithmus speziellen Anforderungen anzupassen.

Eine der wichtigsten Modifikationen ist die *inverse Iteration*, die es uns erlaubt, *beliebige* Eigenwerte zu behandeln: Wir wählen einen *Shift-Parameter* $\mu \in \mathbb{R}$ und stellen fest, dass

$$Ae = \lambda e \iff (A - \mu I)e = (\lambda - \mu)e$$

gilt. Die Eigenwerte der Matrix $A - \mu I$ sind also gegenüber denen der Matrix A um μ „verschoben“, daher der Name.

```
procedure invit(A, μ, ε, var x);
y ← Ax;
λ̃ ← ⟨x, y⟩₂/⟨x, x⟩₂;
while ‖y − λ̃x‖ > ε|λ̃| do begin
  Löse (A − μI)x̃ = x;
  x ← x̃/‖x̃‖;
  y ← Ax;
  λ̃ ← ⟨x, y⟩₂/⟨x, x⟩₂
end
```

Abbildung 3.2: Inverse Iteration mit Shift μ

Falls μ kein Eigenwert ist, ist $A - \mu I$ invertierbar und wir erhalten

$$Ae = \lambda e \iff (A - \mu I)e = (\lambda - \mu)e \iff (A - \mu I)^{-1}e = \frac{1}{\lambda - \mu}e.$$

Jedem Eigenwert λ der Matrix A ist also ein Eigenwert $\frac{1}{\lambda-\mu}$ der Matrix $(A - \mu I)^{-1}$ zugeordnet, und zu beiden gehört derselbe Eigenvektor.

Insbesondere ist der betragsgrößte Eigenwert der Matrix $(A - \mu I)^{-1}$ gerade derjenige, für den $|\lambda - \mu|$ minimal ist. Indem wir die Vektoriteration auf $(A - \mu I)^{-1}$ statt A anwenden, erhalten wir demnach eine Iteration, die denjenigen Eigenwert approximiert, der dem *fast beliebig wählbaren* Shift-Parameter μ am nächsten liegt:

$$\widetilde{x}^{(m+1)} := (A - \mu I)^{-1}x^{(m)}, \tag{3.4a}$$

$$x^{(m+1)} := \widetilde{x}^{(m+1)}/\|\widetilde{x}^{(m+1)}\| \qquad \text{für alle } m \in \mathbb{N}_0. \tag{3.4b}$$

Bei der Implementierung empfiehlt es sich, die Multiplikation eines Vektors mit der Inversen $(A - \mu I)^{-1}$ durch das Lösen eines Gleichungssystems zu ersetzen, da letzteres in der Regel erheblich effizienter und weniger anfällig gegenüber Rundungsfehlern ist. Der Algorithmus ist in Abbildung 3.2 dargestellt.

Der gewünschte Eigenvektor wird sich um so schneller durchsetzen, je kleiner der Abstand des zugehörigen Eigenwerts zu dem Shift-Parameter μ ist.

Falls uns *a priori* keine hinreichend genauen Informationen über die Lage des gewünschten Eigenwerts vorliegen, empfiehlt es sich, den Shift-Parameter *automatisch* durch den Algorithmus wählen zu lassen. Da wir hoffen dürfen, dass sich im Zuge der Iteration immer bessere Näherungen des Eigenvektors ergeben werden, bietet es sich an, nicht nur einmal zu Beginn der Iteration den Shift-Parameter zu „raten", sondern alle paar Schritte oder sogar in jedem einzelnen Schritt unsere Schätzung zu verbessern.

Da wir wissen, dass der RAYLEIGH-Quotient eine Näherung eines Eigenwerts ermittelt, falls wir eine Näherung eines Eigenvektors einsetzen, ist er der naheliegende Kandidat: Die RAYLEIGH-*Iteration* berechnet in jedem Schritt des Verfahrens einen neuen Shift-Parameter $\mu^{(m)} = \Lambda_A(x^{(m)})$ und führt dann einen Schritt der inversen Iteration mit

diesem Shift durch:

$$\mu^{(m+1)} := \langle x^{(m)}, Ax^{(m)} \rangle_2 / \langle x^{(m)}, x^{(m)} \rangle_2, \tag{3.5a}$$

$$\widetilde{x}^{(m+1)} := (A - \mu^{(m+1)} I)^{-1} x^{(m)}, \tag{3.5b}$$

$$x^{(m+1)} := \widetilde{x}^{(m+1)} / \|\widetilde{x}^{(m+1)}\| \qquad \text{für alle } m \in \mathbb{N}_0. \tag{3.5c}$$

Diese Iteration hat den Vorteil, dass sie *sehr* schnell eine hohe Genauigkeit erreicht, falls unser Anfangsvektor eine passable Näherung eines Eigenvektors ist. Ist er es nicht, kann die Iteration sich dagegen chaotisch verhalten.

Es kann sich daher durchaus lohnen, zunächst einige Schritte der inversen Iteration mit festem Shift-Parameter durchzuführen, um einen brauchbaren Anfangsvektor für die Rayleigh-Iteration zu erhalten.

Bemerkung 3.2 (Callback-Funktionen) *Alle in diesem Abschnitt diskutierten Algorithmen benötigen die Matrix A nicht explizit: Es genügt, wenn uns Funktionen zur Verfügung stehen, die A mit einem Vektor multiplizieren und ein lineares Gleichungssystem mit der Matrix $A - \mu I$ lösen.*

Deshalb kann es sich lohnen, die Algorithmen so zu implementieren, dass sie die Matrix A als Instanz eines abstrakten Datentyps *behandeln, der lediglich die beiden für uns relevanten Funktionen anbietet.*

„Echte“ objektorientierte Programmiersprachen wie Java und C++ bieten dafür geeignete Sprachkonstrukte. In C können wir uns mit Funktionszeigern behelfen, so definiert etwa

```
typedef void
(*addeval_func)(real alpha, void *A,
                const real *x, real *y);
```

einen Typ `addeval_func`*, der Funktionen repräsentiert, die die Parameter* `alpha`*,* `A`*,* `x` *sowie* `y` *aufnehmen. Eine Variable*

```
addeval_func my_addeval;
```

zeigt dann auf eine Funktion, die in der Form

```
my_addeval(alpha, A, x, y);
```

aufgerufen werden kann. Wir können also unseren Implementierungen der drei Iterationsverfahren einfach Zeiger auf Funktionen als Parameter übergeben, die die benötigten Funktionen ausführen.

Für eine vollbesetzte Matrix A in der von BLAS verwendeten Array-Darstellung könnte diese Funktion wie folgt realisiert werden:

```
typedef struct {
  int rows;
  int cols;

  real *A;
```

```
  int ld;
} amatrix;

void
addeval_amatrix(real alpha, const amatrix *A,
                const real *x, real *y)
{
  gemv(false, A->rows, A->cols, alpha, A->A, A->ld,
       x, 1, y, 1);
}

int
main()
{
  amatrix *A;
  /* ... */
  addeval_func my_addeval = (addeval_func) addeval_amatrix;
  /* ... */
  my_addeval(alpha, A, x, y);
  /* ... */
}
```

Bei der Zuweisung der Funktion `addeval_amatrix` *an die Variable* `my_addeval` *muss dabei explizit der Typ angepasst werden, da* `addeval_amatrix` *im zweiten Argument einen Zeiger auf eine* `amatrix` *erwartet, während* `addeval_func` *einen* **`void`**-*Zeiger verwendet.*

Bei dem Aufruf von `my_addeval` *ist keine explizite Typumwandlung erforderlich, da der Zeiger* `A` *automatisch in einen* **`void`**-*Zeiger konvertiert wird.*

3.2 Konvergenz der Vektoriteration*

Eine zentrale Frage bei allen Iterationsverfahren ist die nach der *Konvergenz*, also danach, wie schnell sich die Iterationsvektoren der Lösung nähern.

Bei Eigenwertproblemen ergibt sich eine Komplikation dadurch, dass Eigenvektoren nicht eindeutig bestimmt sind, denn sie können mit einem von null verschiedenen Skalierungsfaktor multipliziert werden, ohne die Eigenvektor-Eigenschaft zu verlieren. Es gibt also keinen eindeutig bestimmten Eigenvektor zu einem gegebenen Eigenwert.

Dieses Problem können wir elegant lösen, indem wir Konvergenzaussagen für den *Winkel* zwischen dem Iterationsvektor und dem Eigenvektor formulieren. Wir definieren Cosinus, Sinus und Tangens zwischen zwei von null verschiedenen Vektoren durch

$$\cos\angle(x,y) := \frac{|\langle x,y\rangle_2|}{\|x\|_2\|y\|_2}, \qquad \sin\angle(x,y) := \sqrt{1-\cos^2\angle(x,y)},$$
$$\tan\angle(x,y) := \frac{\sin\angle(x,y)}{\cos\angle(x,y)} \qquad \text{für alle } x,y \in \mathbb{R}^n \setminus \{0\}.$$

Da der Winkel zwischen zwei Vektoren von der Skalierung der Vektoren unabhängig ist (Skalierungsfaktoren kürzen sich jeweils heraus), können wir uns für die Konvergenzanalyse auf die einfache Vektoriteration (3.2) ohne Normierung der Vektoren beschränken.

Eine erste Konvergenzaussage können wir für Diagonalmatrizen formulieren:

Lemma 3.3 (Vektoriteration für Diagonalmatrizen) *Seien* $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ *mit*

$$|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$$

gegeben, sei

$$D := \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix},$$

und sei $\delta_1 := (1, 0, \ldots, 0) \in \mathbb{R}^n$ *der erste kanonische Einheitsvektor.*

Falls $\cos \angle(x^{(0)}, \delta_1) > 0$ *gilt, erfüllen die Iterationsvektoren der Vektoriteration, angewendet auf die Matrix* D*, die Abschätzung*

$$\tan \angle(x^{(m)}, \delta_1) \leq \left(\frac{|\lambda_2|}{|\lambda_1|} \right)^m \tan \angle(x^{(0)}, \delta_1) \qquad \text{für alle } m \in \mathbb{N}_0.$$

Beweis. Sei $x^{(0)} \in \mathbb{R}^n$ mit $\tan \angle(x^{(0)}, \delta_1) \neq 0$ gegeben.

Aus der Voraussetzung folgt

$$|x_1^{(0)}| = |\langle x^{(0)}, \delta_1 \rangle_2| \neq 0$$

und wir erhalten

$$\begin{aligned}
\cos^2 \angle(x^{(m)}, \delta_1) &= \frac{|\langle x^{(m)}, \delta_1 \rangle_2|^2}{\|x^{(m)}\|_2^2} = \frac{|x_1^{(m)}|^2}{\sum_{j=1}^n |x_j^{(m)}|^2}, \\
\sin^2 \angle(x^{(m)}, \delta_1) &= 1 - \cos^2 \angle(x^{(m)}, \delta_1) = \frac{\sum_{j=1}^n |x_j^{(m)}|^2 - |x_1^{(m)}|^2}{\sum_{j=1}^n |x_j^{(m)}|^2} = \frac{\sum_{j=2}^n |x_j^{(m)}|^2}{\sum_{j=1}^n |x_j^{(m)}|^2}, \\
\tan^2 \angle(x^{(m)}, \delta_1) &= \frac{\sin^2 \angle(x^{(m)}, \delta_1)}{\cos^2 \angle(x^{(m)}, \delta_1)} = \frac{\sum_{j=2}^n |x_j^{(m)}|^2}{|x_1^{(m)}|^2} \qquad \text{für alle } m \in \mathbb{N}_0.
\end{aligned}$$

Sei $m \in \mathbb{N}_0$. Da wir wegen der Skalierungsinvarianz der Winkel die einfache Vektoriteration (3.2) betrachten, haben wir

$$\begin{aligned}
\tan^2 \angle(x^{(m)}, \delta_1) = \tan^2 \angle(D^m x^{(0)}, \delta_1) &= \frac{\sum_{j=2}^n |\lambda_j^m x_j^{(0)}|^2}{|\lambda_1^m x_1^{(0)}|^2} \leq \frac{\sum_{j=2}^n |\lambda_2|^{2m} |x_j^{(0)}|^2}{|\lambda_1|^{2m} |x_1^{(0)}|^2} \\
&= \left(\frac{|\lambda_2|}{|\lambda_1|} \right)^{2m} \frac{\sum_{j=2}^n |x_j^{(0)}|^2}{|x_1^{(0)}|^2} = \left(\frac{|\lambda_2|}{|\lambda_1|} \right)^{2m} \tan^2 \angle(x^{(0)}, \delta_1),
\end{aligned}$$

und unsere Behauptung folgt, indem wir die Wurzel ziehen. ∎

Diagonalmatrizen sind nun keine besonders interessante Anwendung, da sich bei ihnen alle Eigenwerte und Eigenvektoren einfach ablesen lassen. Allerdings lässt sich die Aussage unseres Lemmas elegant auf symmetrische Matrizen verallgemeinern, indem wir das folgende Resultat aus der linearen Algebra verwenden:

Satz 3.4 (Hauptachsentransformation) *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix, es gelte also $A = A^*$.*

Dann existieren eine orthogonale Matrix $Q \in \mathbb{R}^{n\times n}$ und eine Diagonalmatrix

$$D = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}$$

mit $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$ und

$$A = QDQ^*.$$

Wir können also jede symmetrische Matrix so „drehen“, dass sie zu einer Diagonalmatrix wird. Da orthogonale Matrizen das Skalarprodukt und die Norm unverändert lassen, folgt aus Lemma 3.3 unmittelbar das folgende allgemeinere Ergebnis:

Folgerung 3.5 (Vektoriteration) *Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische Matrix, es gelte also $A = A^*$. Seien $\lambda_1, \ldots, \lambda_n \in \mathbb{R}$ und die Matrizen $Q, D \in \mathbb{R}^{n\times n}$ wie in Satz 3.4 gegeben. Dann ist $e_1 := Q\delta_1$ ein Eigenvektor zu dem betragsgrößten Eigenwert λ_1.*

Falls $\cos\angle(x^{(0)}, e_1) > 0$ gilt, erfüllen die Iterationsvektoren der Vektoriteration die Abschätzung

$$\tan\angle(x^{(m)}, e_1) \leq \left(\frac{|\lambda_2|}{|\lambda_1|}\right)^m \tan\angle(x^{(0)}, e_1) \qquad \text{für alle } m \in \mathbb{N}_0.$$

Beweis. Zunächst halten wir fest, dass

$$Ae_1 = QDQ^*Q\delta_1 = QD\delta_1 = \lambda_1 Q\delta_1 = \lambda_1 e_1$$

gilt, so dass e_1 in der Tat ein Eigenvektor für den Eigenwert λ_1 ist.

Wir definieren

$$\widehat{x}^{(m)} := Q^* x^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

und stellen fest, dass

$$\widehat{x}^{(m+1)} = Q^* x^{(m+1)} = Q^* A x^{(m)} = Q^* A Q \widehat{x}^{(m)} = D\widehat{x}^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

gilt, dass also die Vektoren $(\widehat{x}^{(m)})_{m=0}^\infty$ sich aus der Vektoriteration für die Diagonalmatrix D ergeben.

Da Q eine orthogonale Matrix ist, haben wir dank (2.17) die Gleichungen

$$\langle x^{(m)}, e_1\rangle_2 = \langle Q\widehat{x}^{(m)}, Q\delta_1\rangle_2 = \langle \widehat{x}^{(m)}, Q^*Q\delta_1\rangle_2 = \langle \widehat{x}^{(m)}, \delta_1\rangle_2,$$
$$\|x^{(m)}\|_2 = \sqrt{\langle x^{(m)}, x^{(m)}\rangle_2} = \sqrt{\langle \widehat{x}^{(m)}, \widehat{x}^{(m)}\rangle_2} = \|\widehat{x}^{(m)}\|_2 \qquad \text{für alle } m \in \mathbb{N}_0,$$

so dass wir mit Lemma 3.3 unmittelbar

$$\tan \angle(x^{(m)}, e_1) = \tan \angle(\widehat{x}^{(m)}, \delta_1) \leq \left(\frac{|\lambda_2|}{|\lambda_1|}\right)^m \tan \angle(\widehat{x}^{(0)}, \delta_1)$$
$$= \left(\frac{|\lambda_2|}{|\lambda_1|}\right)^m \tan \angle(x^{(0)}, e_1) \qquad \text{für alle } m \in \mathbb{N}_0$$

erhalten. ∎

Damit steht uns — zumindest für den Fall symmetrischer Matrizen — eine quantitative Konvergenzaussage zur Verfügung, die sich in der Praxis als durchaus präzise erweist: Falls der Anfangsvektor $x^{(0)}$ eine Linearkombination aus Eigenvektoren zu λ_1 und λ_2 ist, wird aus der Ungleichung eine Gleichung.

Für die inverse Iteration ergibt sich unter der Voraussetzung

$$|\lambda_1 - \mu| \leq |\lambda_2 - \mu| \leq \ldots \leq |\lambda_n - \mu| \tag{3.6}$$

die Aussage

$$\tan \angle(x^{(m)}, e_1) \leq \left(\frac{|\lambda_1 - \mu|}{|\lambda_2 - \mu|}\right)^m \tan \angle(x^{(0)}, e_1) \qquad \text{für alle } m \in \mathbb{N}_0, \tag{3.7}$$

indem wir Folgerung 3.5 auf die Matrix $(A - \mu I)^{-1}$ anwenden.

Für den Rayleigh-Quotienten können wir Aussagen darüber gewinnen, wie gut er einen Eigenwert approximiert, wenn der eingesetzte Vektor einen Eigenvektor annähert.

Lemma 3.6 (Rayleigh-Quotient) *Sei $x \in \mathbb{R}^n \setminus \{0\}$, und sei $\lambda \in \mathbb{R}$ ein Eigenwert der Matrix $A \in \mathbb{R}^{n \times n}$ mit einem Eigenvektor $e \in \mathbb{R}^n \setminus \{0\}$. Dann gilt*

$$|\lambda - \Lambda_A(x)| \leq \|A - \lambda I\|_2 \sin \angle(x, e).$$

Falls A symmetrisch ist, falls also $A = A^*$ gilt, haben wir sogar

$$|\lambda - \Lambda_A(x)| \leq \|A - \lambda I\|_2 \sin^2 \angle(x, e).$$

Beweis. Zunächst halten wir fest, dass

$$\left\| x - \frac{\langle x, e\rangle_2}{\|e\|_2^2} e \right\|_2^2 = \langle x - \frac{\langle x, e\rangle_2}{\|e\|_2^2} e, x - \frac{\langle x, e\rangle_2}{\|e\|_2^2} e\rangle_2$$
$$= \|x\|_2^2 - 2\frac{\langle x, e\rangle_2^2}{\|e\|_2^2} + \frac{\langle x, e\rangle_2^2}{\|e\|_2^4}\|e\|_2^2 = \|x\|_2^2 - \frac{\langle x, e\rangle_2^2}{\|e\|_2^2}$$

gilt, so dass wir mit $\alpha := \langle x, e\rangle_2 / \|e\|_2^2$ die Gleichung

$$\sin^2 \angle(x,e) = \frac{\|x\|_2^2\|e\|_2^2 - \langle x,e\rangle_2^2}{\|x\|_2^2\|e\|_e^2} = \frac{\|x\|_2^2 - \frac{\langle x,e\rangle_2^2}{\|e\|_2^2}}{\|x\|_2^2} = \frac{\|x-\alpha e\|_2^2}{\|x\|_2^2}$$

erhalten. Mit ihrer Hilfe, der Gleichung $(A - \lambda I)e = 0$ und der CAUCHY-SCHWARZ-Ungleichung

$$|\langle y,z\rangle_2| \leq \|y\|_2\|z\|_2 \qquad \text{für alle } y,z \in \mathbb{R}^n$$

können wir nun die gewünschten Abschätzungen nachrechnen, wobei wir im vorletzten Schritt die Verträglichkeitsungleichung (2.13) ausnutzen:

$$\begin{aligned}
|\lambda - \Lambda_A(x)| &= \left|\frac{\langle x,\lambda x\rangle_2}{\langle x,x\rangle_2} - \frac{\langle x,Ax\rangle_2}{\langle x,x\rangle_2}\right| = \frac{|\langle x,(A-\lambda I)x\rangle_2|}{\|x\|_2^2} = \frac{|\langle x,(A-\lambda I)(x-\alpha e)\rangle_2|}{\|x\|_2^2} \\
&\leq \frac{\|x\|_2\|(A-\lambda I)(x-\alpha e)\|_2}{\|x\|_2^2} \leq \frac{\|A-\lambda I\|_2\|x-\alpha e\|_2}{\|x\|_2} \\
&= \|A-\lambda I\|_2 \sin\angle(x,e).
\end{aligned}$$

Falls A symmetrisch ist, falls also $A^* = A$ gilt, haben wir auch $(A - \lambda I)^* = A^* - \lambda I^* = A - \lambda I$, so dass wir mit der Gleichung (2.17), der CAUCHY-SCHWARZ-Ungleichung und der Verträglichkeitsungleichung (2.13) die Abschätzung

$$\begin{aligned}
|\lambda - \Lambda_A(x)| &= \frac{|\langle x,(A-\lambda I)(x-\alpha e)\rangle_2|}{\|x\|_2^2} = \frac{|\langle (A-\lambda I)^* x, x-\alpha e\rangle_2|}{\|x\|_2^2} \\
&= \frac{|\langle (A-\lambda I)x, x-\alpha e\rangle_2|}{\|x\|_2^2} = \frac{|\langle (A-\lambda I)(x-\alpha e), x-\alpha e\rangle_2|}{\|x\|_2^2} \\
&\leq \frac{\|(A-\lambda I)(x-\alpha e)\|_2\|x-\alpha e\|_2}{\|x\|_2^2} \leq \|A-\lambda I\|_2 \frac{\|x-\alpha e\|_2^2}{\|x\|_2^2} \\
&= \|A-\lambda I\|_2 \sin^2\angle(x,e)
\end{aligned}$$

erhalten. ∎

Folgerung 3.7 (Rayleigh-Iteration) *Sei $A \in \mathbb{R}^{n\times n}$ symmetrisch, es gelte also $A = A^*$. Seien die Eigenwerte gemäß (3.6) angeordnet. Sei $x^{(0)} \in \mathbb{R}^n \setminus \{0\}$. Falls*

$$2\|A-\lambda I\|_2 \tan^2\angle(x^{(0)},e_1) < |\lambda_2 - \lambda_1| \tag{3.8}$$

gilt, führt ein Schritt der RAYLEIGH*-Iteration mit $\mu = \Lambda_A(x^{(0)})$ zu*

$$\tan\angle(x^{(1)},e_1) \leq \frac{2\|A-\lambda_1 I\|_2}{|\lambda_2-\lambda_1|}\tan^3\angle(x^{(0)},e_1).$$

Insbesondere gilt $\tan\angle(x^{(1)},e_1) < \tan\angle(x^{(0)},e_1)$, so dass die Eigenschaft (3.8) auch für die nächste Iterierte gilt.

Beweis. Es gelte (3.8). Mit Lemma 3.6 folgen

$$\begin{aligned}
|\lambda_1 - \mu| &\leq \|A - \lambda_1 I\|_2 \sin^2 \angle(x^{(0)}, e_1) \leq \|A - \lambda_1 I\|_2 \tan^2 \angle(x^{(0)}, e_1), \\
|\lambda_2 - \mu| &\geq |\lambda_2 - \lambda_1| - |\lambda_1 - \mu| \geq |\lambda_2 - \lambda_1| - \|A - \lambda_1 I\|_2 \tan^2 \angle(x^{(0)}, e_1) \\
&\geq |\lambda_2 - \lambda_1| - |\lambda_2 - \lambda_1|/2 = |\lambda_2 - \lambda_1|/2.
\end{aligned}$$

Mit (3.7) und (3.8) erhalten wir

$$\begin{aligned}
\tan \angle(x^{(1)}, e_1) &\leq \frac{|\lambda_1 - \mu|}{|\lambda_2 - \mu|} \tan \angle(x^{(0)}, e_1) \\
&\leq \frac{\|A - \lambda_1 I\|_2 \tan^2 \angle(x^{(0)}, e_1)}{|\lambda_2 - \lambda_1|/2} \tan \angle(x^{(0)}, e_1) \\
&= \frac{2\|A - \lambda_1 I\|_2}{|\lambda_2 - \lambda_1|} \tan^3 \angle(x^{(0)}, e_1).
\end{aligned}$$

■

3.3 Newton-Iteration

Wir haben gesehen, dass die Suche nach den Eigenwerten einer Matrix äquivalent zu der Suche nach den Nullstellen ihres charakteristischen Polynoms ist. Nun wollen wir uns der Frage widmen, wie sich Nullstellen allgemeinerer Funktionen finden lassen.

Genauer gesagt gehen wir davon aus, dass eine Funktion $f\colon \mathbb{R}^n \to \mathbb{R}^n$ gegeben ist und wir eine Nullstelle $x^* \in \mathbb{R}^n$ suchen, also eine Lösung der Gleichung

$$f(x^*) = 0. \tag{3.9}$$

Ein einfaches Beispiel für derartige Aufgabenstellungen ist die Suche nach der Quadratwurzel einer Zahl $a \in \mathbb{R}_{\geq 0}$, die wir bereits bei der Konstruktion der QR-Zerlegung verwendet haben. Wir können sie beispielsweise als Nullstelle der Funktion

$$f\colon \mathbb{R} \to \mathbb{R}, \qquad x \mapsto x^2 - a,$$

oder der Funktion

$$f\colon \mathbb{R} \to \mathbb{R}, \qquad x \mapsto \frac{a}{x^2} - 1,$$

charakterisieren.

Die Idee des NEWTON-Verfahrens besteht darin, die Funktion f durch ein lineares Polynom zu approximieren und eine Nullstelle dieses Polynoms zu suchen. Dazu verwenden wir den Satz von TAYLOR.

Satz 3.8 (Taylor) *Sei $m \in \mathbb{N}_0$, und sei $f \in C^{m+1}[a, b]$. Seien $x, y \in [a, b]$. Es existiert ein $\eta \in [a, b]$ mit*

$$f(y) = \sum_{k=0}^{m} f^{(k)}(x) \frac{(y - x)^k}{k!} + f^{(m+1)}(\eta) \frac{(y - x)^{m+1}}{(m + 1)!}.$$

Wir beschränken uns zunächst auf den eindimensionalen Fall, also auf $n = 1$, und wenden den Satz von Taylor auf $y = x^*$ und $m = 1$ an. Wegen $f(x^*) = 0$ und unter der Annahme $f'(x) \neq 0$ folgt

$$\begin{aligned}
0 = f(x^*) &= f(x) + f'(x)(x^* - x) + f''(\eta)\frac{(x^* - x)^2}{2}, \\
0 &= \frac{f(x)}{f'(x)} + x^* - x + \frac{f''(\eta)}{2f'(x)}(x^* - x)^2, \\
x^* &= x - \frac{f(x)}{f'(x)} - \frac{f''(\eta)}{2f'(x)}(x^* - x)^2. \qquad (3.10)
\end{aligned}$$

Falls wir annehmen, dass $(x^* - x)^2$ „hinreichend klein" ist, können wir den dritten Term auf der rechten Seite wegfallen lassen und erhalten

$$x^* \approx x - \frac{f(x)}{f'(x)}.$$

Wir definieren die Abbildung

$$\Phi\colon \mathbb{R} \setminus \{x \in \mathbb{R} \ : \ f'(x) = 0\} \to \mathbb{R}, \qquad x \mapsto x - \frac{f(x)}{f'(x)},$$

und stellen fest, dass aus (3.10) unmittelbar

$$\begin{aligned}
x^* &= \Phi(x) - \frac{f''(\eta)}{2f'(x)}(x^* - x)^2, \\
\Phi(x) - x^* &= \frac{f''(\eta)}{2f'(x)}(x^* - x)^2, \\
|\Phi(x) - x^*| &= \frac{|f''(\eta)|}{2|f'(x)|}|x - x^*|^2 \qquad (3.11)
\end{aligned}$$

folgt. Falls also

$$\frac{|f''(\eta)|}{2|f'(x)|}|x - x^*| < 1$$

gilt, wird $\Phi(x)$ näher an x^* liegen als x. Damit steht uns eine Rechenvorschrift zur Verfügung, die aus einer Näherungslösung eine verbesserte Näherungslösung konstruiert. Das ist gerade das Grundprinzip eines Iterationsverfahrens.

Aus unserer Beobachtung ergibt sich unmittelbar die Newton-Iteration

$$x^{(m+1)} := \Phi(x^{(m)}) = x^{(m)} - \frac{f(x^{(m)})}{f'(x^{(m)})} \qquad \text{für alle } m \in \mathbb{N}_0, \qquad (3.12)$$

die zu den am häufigsten eingesetzten Methoden für die Behandlung nichtlinearer Gleichungssysteme zählt.

Eine Besonderheit der Newton-Iteration ist ihre *quadratische Konvergenz.*

Satz 3.9 (Quadratische Konvergenz) *Sei $x^* \in [a,b]$ eine Nullstelle einer Funktion $f \in C^2[a,b]$. Seien $\beta, \gamma \in \mathbb{R}_{\geq 0}$ gegeben mit*

$$\frac{1}{|f'(x)|} \leq \beta, \qquad |f''(x)| \leq \gamma \qquad \text{für alle } x \in [a,b].$$

Sei $x^{(0)} \in [a,b]$ gegeben. Falls

$$c := \frac{\beta\gamma}{2}|x^{(0)} - x^*| < 1 \tag{3.13}$$

gilt, konvergieren die Vektoren $(x^{(m)})_{m=0}^\infty$ der Newton*-Iteration (3.12) gegen x^* mit*

$$|x^{(m)} - x^*| \leq c^{2^m-1}|x^{(0)} - x^*| \qquad \text{für alle } m \in \mathbb{N}_0.$$

Beweis. Wir zeigen per Induktion

$$|x^{(m)} - x^*| \leq c^{2^m-1}|x^{(0)} - x^*| \tag{3.14}$$

für alle $m \in \mathbb{N}_0$.

Induktionsanfang: Für $m = 0$ ist die Abschätzung (3.14) wegen $2^m - 1 = 1 - 1 = 0$ und $c^0 = 1$ trivial.

Induktionsvoraussetzung: Sei $m \in \mathbb{N}_0$ so gegeben, dass (3.14) gilt.

Induktionsschritt: Mit (3.11) erhalten wir

$$\begin{aligned}|x^{(m+1)} - x^*| &\leq \frac{|f''(\eta)|}{2|f'(x^{(m)})|}|x^{(m)} - x^*|^2 \leq \frac{\beta\gamma}{2}(c^{2^m-1}|x^{(0)} - x^*|)^2 \\ &= \frac{\beta\gamma}{2}c^{2^{m+1}-2}|x^{(0)} - x^*|^2 = c^{2^{m+1}-1}|x^{(0)} - x^*|\end{aligned}$$

mit einem geeigneten $\eta \in [a,b]$.

Da wegen (3.13) die Ungleichung $0 \leq c < 1$ gelten muss, konvergiert somit die Folge $(x^{(m)})_{m=0}^\infty$ gegen x^*. ∎

Die Abschätzung (3.14) bedeutet, dass die Newton-Iteration *sehr* schnell konvergieren wird, sobald wir eine Näherungslösung gefunden haben, die hinreichend nahe an der exakten Lösung x^* liegt.

Übungsaufgabe 3.10 (Quadratwurzel) *Die Quadratwurzel $x^* := \sqrt{a}$ einer Zahl $a \in \mathbb{R}_{>0}$ lässt sich mit Hilfe der Newton-Iteration in unterschiedlicher Weise approximieren.*

Ein naheliegender Ansatz ist es, die Funktion

$$f\colon \mathbb{R} \to \mathbb{R}, \qquad x \mapsto x^2 - a,$$

zu verwenden, die offenbar die Nullstellen $\sqrt{a}$ und $-\sqrt{a}$ besitzt. Die resultierende Newton*-Iteration hat die Form*

$$\Phi(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x} = x - \frac{x}{2} + \frac{a}{2x} = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

und ist unter dem Namen HERON-*Verfahren bekannt.*

Da moderne Prozessoren in der Regel Additionen und Multiplikationen erheblich schneller als Divisionen ausführen können, stellt sich die Frage, ob sich auch eine Iteration finden lässt, die nur Additionen und Multiplikationen benötigt. Dazu berechnen wir die reziproke Quadratwurzel $x^* = 1/\sqrt{a}$, *aus der sich mittels* $\sqrt{a} = ax^*$ *einfach die Quadratwurzel rekonstruieren lässt. Die reziproke Quadratwurzel ist eine Nullstelle der Funktion*

$$f\colon \mathbb{R} \to \mathbb{R}, \qquad x \mapsto \frac{1}{x^2} - a,$$

aus der sich die NEWTON-*Iteration mit*

$$\Phi(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x^{-2} - a}{-2x^{-3}} = x + \frac{1}{2}(x - ax^3) = \frac{x}{2}(3 - ax^2)$$

ergibt. Diese Iterationsvorschrift erfordert keine Divisionen durch x *oder* a *und lässt sich deshalb sehr schnell ausführen.*

Natürlich sind wir daran interessiert, auch mehrdimensionale Nullstellenprobleme lösen zu können. Eine mehrdimensionale Variante der NEWTON-Iteration lässt sich einfach herleiten, indem wir lediglich das Verhalten der Funktion f auf dem Geradenstück von x zu x^* untersuchen.

Sei nun $\Omega \subseteq \mathbb{R}^n$ eine konvexe Menge, sei $f \in C^2(\Omega, \mathbb{R}^n)$, und sei $x^* \in \Omega$ eine Nullstelle der Funktion f.

Sei $x \in \Omega$ eine Näherung der Nullstelle x^*. Wir parametrisieren das Geradenstück von x zu x^* mit der Abbildung

$$\gamma\colon [0,1] \to \Omega, \qquad t \mapsto x + t(x^* - x),$$

und untersuchen die Funktion

$$g\colon [0,1] \to \mathbb{R}^n, \qquad t \mapsto f(\gamma(t)).$$

Aus den Definitionen und der Kettenregel folgen unmittelbar

$$\begin{aligned} &\gamma(0) = x, && \gamma(1) = x^*, && \gamma'(t) = x^* - x, \\ &g(0) = f(x), && g(1) = f(x^*) = 0, && g'(t) = Df(\gamma(t))(x^* - x) \qquad \text{für alle } t \in [0,1], \end{aligned}$$

wobei $Df(\gamma(t))$ die JACOBI-Matrix der Funktion f im Punkt $\gamma(t)$ bezeichnet. Mit dem Hauptsatz der Integral- und Differentialrechnung erhalten wir

$$\begin{aligned} -f(x) = f(x^*) - f(x) = g(1) - g(0) &= \int_0^1 g'(t)\,dt = \int_0^1 Df(\gamma(t))(x^* - x)\,dt \\ &= Df(x)(x^* - x) + \int_0^1 \Big(Df(\gamma(t)) - Df(\gamma(0))\Big)(x^* - x)\,dt. \end{aligned}$$

Wir gehen davon aus, dass die JACOBI-Matrix invertierbar ist und gelangen zu

$$-Df(x)^{-1}f(x) = x^* - x + \int_0^1 Df(x)^{-1}\Big(Df(\gamma(t)) - Df(\gamma(0))\Big)(x^* - x)\,dt,$$
$$x - Df(x)^{-1}f(x) = x^* + \int_0^1 Df(x)^{-1}\Big(Df(\gamma(t)) - Df(\gamma(0))\Big)(x^* - x)\,dt.$$

Wenn wir annehmen, dass der zweiten Term auf der rechten Seite „klein genug“ ist, folgt

$$x^* \approx x - Df(x)^{-1}f(x),$$

und die Abbildung

$$\Phi\colon \mathbb{R}^n \setminus \{x \in \mathbb{R}^n \ :\ Df(x) \text{ nicht invertierbar } \} \to \mathbb{R}^n, \qquad x \mapsto x - Df(x)^{-1}f(x),$$

ist die mehrdimensionale Variante der NEWTON-Iteration. Im Vergleich zu der eindimensionalen Version tritt die JACOBI-Matrix $Df(x)$ an die Stelle der Ableitung $f'(x)$ und die Inverse an die Stelle des Kehrwerts.

3.4 Optimierungsaufgaben

In vielen Anwendungen sind wir mit *Optimierungsaufgaben* konfrontiert: Man will nicht ein Gleichungssystem lösen, sondern man ist daran interessiert, bestimmte Parameter so zu wählen, dass Kosten minimiert oder Gewinne maximiert werden.

Im einfachsten Fall gelangen wir zu der folgenden Aufgabenstellung: Gegeben eine Funktion $J : \mathbb{R}^n \to \mathbb{R}$, finde ein $x^* \in \mathbb{R}^n$ so, dass

$$J(x^*) \leq J(x) \qquad \text{für alle } x \in \mathbb{R}^n$$

gilt, dass also J ein Minimum in x^* annimmt.

Um die Minimierungsaufgabe auf den eindimensionalen Fall zu reduzieren, fixieren wir eine beliebige Richtung $p \in \mathbb{R}^n \setminus \{0\}$ und betrachten die Funktion

$$g\colon \mathbb{R} \to \mathbb{R}, \qquad t \mapsto J(x + tp),$$

die ihr Minimum in $t = 0$ annehmen muss, falls x die Minimierungsaufgabe löst. Falls J zweimal stetig differenzierbar ist, finden wir mit dem Satz 3.8 von TAYLOR für jedes $t \in \mathbb{R}$ ein $\eta \in \mathbb{R}$ mit

$$g(t) = g(0) + g'(0)t + g''(\eta)\frac{t^2}{2}.$$

Mit Hilfe der Kettenregel erhalten wir

$$g'(0) = \sum_{i=1}^n \frac{\partial J}{\partial x_i}(0)p_i, \qquad g''(\eta) = \sum_{i,j=1}^n \frac{\partial^2 J}{\partial x_i \partial x_j}(x + \eta p)p_i p_j,$$

und indem wir den *Gradienten*

$$\nabla J(x) = \begin{pmatrix} \frac{\partial J}{\partial x_1}(x) \\ \vdots \\ \frac{\partial J}{\partial x_n}(x) \end{pmatrix} \qquad \text{für alle } x \in \mathbb{R}^n$$

und die HESSE-*Matrix*

$$D^2 J(x) = \begin{pmatrix} \frac{\partial^2 J}{\partial x_1 \partial x_1}(x) & \cdots & \frac{\partial^2 J}{\partial x_1 \partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 J}{\partial x_n \partial x_1}(x) & \cdots & \frac{\partial^2 J}{\partial x_n \partial x_n}(x) \end{pmatrix} \qquad \text{für alle } x \in \mathbb{R}^n$$

einführen, können wir diese Gleichung kompakt als

$$g(t) = g(0) + t\langle \nabla J(x), p\rangle + \frac{t^2}{2}\langle D^2 J(x + \eta p)p, p\rangle \tag{3.15}$$

schreiben. Um Rückschlüsse auf die Minimierungsaufgabe ziehen zu können, benötigen wir eine zusätzliche Voraussetzung an die HESSE-Matrix:

Definition 3.11 (Positiv definite Matrix) *Eine Matrix $A \in \mathbb{R}^{n \times n}$ nennen wir* positiv definit*, falls*

$$\langle x, Ax\rangle > 0 \qquad \textit{für alle } x \in \mathbb{R}^n \setminus \{0\}$$

gilt. Falls das Skalarprodukt nur größer oder gleich null ist, nennen wir die Matrix positiv semidefinit*.*

Wir gehen im Folgenden davon aus, dass die HESSE-Matrix in allen Punkten positiv definit ist. Dann dürfen wir aus $p \neq 0$ unmittelbar auf

$$\langle D^2 J(x + \eta p)p, p\rangle > 0$$

schließen. Da $s \mapsto D^2 J(x + sp)$ eine stetige Funktion ist, können wir

$$c := \max\{\langle D^2 J(x + sp)p, p\rangle \;:\; s \in [-1, 1]\}$$

definieren, und da die HESSE-Matrix positiv definit ist, gilt auch $c > 0$. Aus (3.15) folgt

$$g(t) \leq g(0) + t\langle \nabla J(x), p\rangle + \frac{t^2}{2}c \qquad \text{für alle } t \in [-1, 1].$$

Falls nun $\langle \nabla J(x), p\rangle \in [-c, c]$ gilt, können wir diese Ungleichung auf

$$t := -\frac{\langle \nabla J(x), p\rangle}{c} \in [-1, 1]$$

anwenden, um

$$g(t) \leq g(0) - \frac{\langle \nabla J(x), p \rangle^2}{c} + \frac{\langle \nabla J(x), p \rangle^2}{2c} = g(0) - \frac{\langle \nabla J(x), p \rangle^2}{2c} \leq g(0)$$

zu erhalten. Also kann x nur dann ein Minimum sein, wenn $\langle \nabla J(x), p \rangle = 0$ gilt.

Falls $\langle \nabla J(x), p \rangle > c$ gilt, können wir $t = -1$ setzen und erhalten

$$g(t) \leq g(0) - \langle \nabla J(x), p \rangle + c/2 \leq g(0) - c + c/2 = g(0) - c/2 < g(0),$$

also ist x kein Minimum. Entsprechend können wir auch für $\langle \nabla J(x), p \rangle < -c$ vorgehen.

Insgesamt dürfen wir festhalten, dass x nur dann ein Minimum der Funktion J ist, wenn $\langle \nabla J(x), p \rangle = 0$ für alle $p \in \mathbb{R}^n \setminus \{0\}$ gilt. Das kann nur geschehen, wenn $\nabla J(x) = 0$ gilt. Statt nach einem Minimum der Funktion J zu suchen, können wir also auch nach einer Nullstelle des Gradienten ∇J suchen, beispielsweise mit Hilfe des Newton-Verfahrens.

Dieser Ansatz würde es allerdings erforderlich machen, die Jacobi-Matrix des Gradienten zu berechnen, also gerade die Hesse-Matrix, und das kann einen relativ hohen Rechenaufwand nach sich ziehen.

Einen einfacheren Zugang bietet das *Gradientenverfahren*: Wir gehen von (3.15) aus, also von

$$g(t) = g(0) + t\langle \nabla J(x), p \rangle + \frac{t^2}{2} \langle D^2 J(x + \eta p)p, p \rangle,$$

und beschränken uns auf kleine Werte von t. Dann können wir den dritten Term auf der rechten Seite wegfallen lassen und erhalten

$$g(t) \approx g(0) + t\langle \nabla J(x), p \rangle. \tag{3.16}$$

Diese (approximative) Gleichung können wir verwenden, um eine besonders günstige Richtung p auszuwählen: Die Cauchy-Schwarz-Ungleichung beinhaltet, dass das Skalarprodukt maximal wird, wenn p und $\nabla J(x)$ linear abhängig sind, also setzen wir der Einfachheit halber

$$p := -\nabla J(x).$$

Der negative Gradient $-\nabla J(x)$ ist also die Richtung, in der die Funktion J in der Nähe des Punkts x am schnellsten abnimmt.

Solange $J(x)$ nicht schon minimal ist, ist dann nach dem oben Gesagten $p \neq 0$, wir haben also eine akzeptable Richtung gefunden. Nach (3.16) dürfen wir hoffen, dass $g(t) \approx g(0) - t\|\nabla J(x)\|^2$ gilt, sofern wir t hinreichend klein gewählt haben. Damit ist dann $J(x + tp) = g(t)$ näher an dem gesuchten Minimum.

Es ergibt sich die Iterationsvorschrift

$$x^{(m+1)} := x^{(m)} - t_m \nabla J(x^{(m)}) \qquad \text{für alle } m \in \mathbb{N}_0,$$

die bei geeigneter Wahl der Skalierungsparameter t_m eine Folge von Näherungslösungen berechnet, die gegen das gewünschte x^* konvergiert.

3.5 Iterative Verfahren für lineare Gleichungssysteme

Wir haben bereits Verfahren kennen gelernt, mit denen sich beliebige lineare Gleichungssystem lösen lassen. Allerdings ist der Rechenaufwand dieser Verfahren häufig relativ hoch, beispielsweise benötigt die LR-Zerlegung im allgemeinen Fall ungefähr $\frac{2}{3}n^3$ Operationen, so dass für große Gleichungssysteme mit Hunderttausenden oder Millionen von Unbekannten der Aufwand mit bezahlbaren Computern nicht bewältigt werden kann.

In diesen Situationen können iterative Verfahren helfen, die zwar nur eine Näherungslösung berechnen, aber dafür unter Umständen erheblich schneller als die klassischen Algorithmen arbeiten.

Eine wichtige Klasse solcher Verfahren lässt sich herleiten, indem wir das Gleichungssystem $Ax = b$ durch eine äquivalente Minimierungsaufgabe ersetzen. Dazu gehen wir davon aus, dass A eine symmetrische und positiv definite Matrix (Definition 3.11) ist, und betrachten die Funktion

$$\widehat{J}\colon \mathbb{R}^n \to \mathbb{R}, \qquad x \mapsto \langle x - x^*, A(x - x^*)\rangle.$$

Da A als positiv definit vorausgesetzt ist, gilt $\widehat{J}(x) > 0$ für alle $x \neq x^*$, so dass $\widehat{J}(x^*) = 0$ das globale Minimum der Funktion ist.

Allerdings können wir $\widehat{J}$ nicht unmittelbar implementieren, da uns x^* nicht zur Verfügung steht. Mit der binomischen Formel erhalten wir die Gleichung

$$\begin{aligned}\widehat{J}(x) &= \langle x - x^*, A(x - x^*)\rangle \\ &= \langle x, Ax\rangle - \langle x, Ax^*\rangle - \langle x^*, Ax\rangle + \langle x^*, Ax^*\rangle \qquad \text{für alle } x \in \mathbb{R}^n\end{aligned}$$

erfüllt. Da A symmetrisch ist, können wir (2.17) ausnutzen, um

$$\langle x^*, Ax\rangle = \langle A^* x^*, x\rangle = \langle Ax^*, x\rangle = \langle x, Ax^*\rangle \qquad \text{für alle } x \in \mathbb{R}^n$$

zu erhalten, so dass sich mit $Ax^* = b$ die Gleichung

$$\begin{aligned}\widehat{J}(x) &= \langle x - x^*, A(x - x^*)\rangle = \langle x, Ax\rangle - 2\langle x, Ax^*\rangle + \langle x^*, Ax^*\rangle \\ &= \langle x, Ax\rangle - 2\langle x, b\rangle + \langle x^*, b\rangle \qquad \text{für alle } x \in \mathbb{R}^n\end{aligned}$$

ergibt. Der letzte Term $\langle x^*, b\rangle$ ist konstant, also können wir ihn wegfallen lassen, ohne zu beeinflussen, wo das Minimum angenommen wird: Wir definieren

$$J\colon \mathbb{R}^n \to \mathbb{R}, \qquad x \mapsto \langle x, Ax\rangle - 2\langle x, b\rangle,$$

und halten fest, dass $J(x) = \widehat{J}(x) - \langle x^*, b\rangle$ gilt, so dass $J(x)$ ebenfalls sein globales Minimum für $x = x^*$ annimmt. Im Gegensatz zu $\widehat{J}(x)$ können wir $J(x)$ allerdings praktisch auswerten, ohne die Lösung x^* kennen zu müssen.

Wir können das Gradientenverfahren einsetzen, um dieses Minimum zu suchen. Um den Gradienten zu berechnen, fixieren wir $k \in [1:n]$ und halten fest, dass wegen der Symmetrie der Matrix

$$J(x) = \sum_{\substack{i,j=1\\ i\neq k, j\neq k}}^{n} x_i a_{ij} x_j + \sum_{\substack{j=1\\ j\neq k}}^{n} x_k a_{kj} x_j + \sum_{\substack{i=1\\ i\neq k}}^{n} x_i a_{ik} x_k + x_k a_{kk} x_k - 2\sum_{\substack{i=1\\ i\neq k}} x_i b_i - 2x_k b_k$$

$$= \sum_{\substack{i,j=1\\ i\neq k, j\neq k}}^{n} x_i a_{ij} x_j + 2\sum_{\substack{j=1\\ j\neq k}}^{n} x_k a_{kj} x_j + x_k a_{kk} x_k - 2\sum_{\substack{i=1\\ i\neq k}} x_i b_i - 2x_k b_k \qquad \text{für alle } x \in \mathbb{R}^n$$

gilt, so dass wir mit der Potenzregel die partielle Ableitung nach x_k berechnen können:

$$\frac{\partial J}{\partial x_k}(x) = 2\sum_{\substack{j=1\\ j\neq k}}^{n} a_{kj} x_j + 2a_{kk} x_k - 2x_k b_k = 2(Ax - b)_k,$$

so dass wir insgesamt

$$\nabla J(x) = 2(Ax - b) \qquad \text{für alle } x \in \mathbb{R}^n$$

erhalten. Der Gradient unserer Funktion J lässt sich also einfach mit einer einzigen Matrix-Vektor-Multiplikation berechnen.

Um das Gradientenverfahren vollständig zu definieren, müssen wir allerdings auch Skalierungsparameter t so festlegen, dass

$$J(x - t\nabla J(x))$$

so klein wie möglich wird. Für unsere spezielle Funktion J lässt sich tatsächlich der optimale Parameter sogar für beliebige Richtungen $p \in \mathbb{R}^n \setminus \{0\}$ praktisch berechnen:

Lemma 3.12 (Optimale Dämpfung) *Seien $x, p \in \mathbb{R}^n$ mit $p \neq 0$ gegeben.*

$$J(x + t_{opt} p) \leq J(x + tp) \qquad \text{für alle } t \in \mathbb{R}.$$

ist äquivalent zu

$$t_{opt} := \frac{\langle p, b - Ax\rangle}{\langle p, Ap\rangle}.$$

Beweis. Wir betrachten die Funktion

$$g\colon \mathbb{R} \to \mathbb{R}, \qquad t \mapsto J(x + tp),$$

die sich mit der binomischen Formel als

$$\begin{aligned} g(t) &= \langle x + tp, A(x + tp)\rangle - 2\langle x + tp, b\rangle \\ &= \langle x, Ax\rangle + t\langle p, Ax\rangle + t\langle x, Ap\rangle + t^2\langle p, Ap\rangle - 2\langle x, b\rangle - 2t\langle p, b\rangle \\ &= g(0) + t\langle p, Ax\rangle + t\langle x, Ap\rangle + t^2\langle p, Ap\rangle - 2t\langle p, b\rangle \qquad \text{für alle } t \in \mathbb{R} \end{aligned}$$

darstellen lässt. Dank der Symmetrie der Matrix erhalten wir mit (2.17) die Gleichung $\langle x, Ap\rangle = \langle Ax, p\rangle = \langle p, Ax\rangle$, und damit

$$g(t) = g(0) + 2t\langle p, Ax - b\rangle + t^2\langle p, Ap\rangle \qquad \text{für alle } t \in \mathbb{R}.$$

procedure gradients(A, b, ϵ, **var** x);
$r \leftarrow b - Ax$;
while $\|r\| > \epsilon\|b\|$ **do begin**
  $a \leftarrow Ar$;
  $t \leftarrow \langle r, r\rangle / \langle r, a\rangle$;
  $x \leftarrow x + tr$;
  $r \leftarrow r - ta$
end

Abbildung 3.3: Gradientenverfahren für lineare Gleichungssysteme mit einer symmetrischen und positiv definiten Matrix

Um das optimale t zu finden, differenzieren wir die Funktion und suchen eine Nullstelle, müssen also

$$0 = g'(t) = 2\langle p, Ax - b\rangle + 2t\langle p, Ap\rangle$$

lösen. Offenbar ist $t = t_{\mathrm{opt}}$ die einzige Lösung dieser Gleichung, und wegen $\langle p, Ap\rangle > 0$ ist die zweite Ableitung g'' positiv, also haben wir auch ein Minimum gefunden. ∎

Da wir mit t_{opt} die Suchrichtung p optimal skalieren, dürfen wir statt des Gradienten $\nabla J(x)$ auch das *Residuum* $r = b - Ax$ verwenden, das wir für die Berechnung von t_{opt} ohnehin benötigen. Damit erhalten wir als vorläufige Fassung des Gradientenverfahrens

$$\begin{aligned} r^{(m)} &\leftarrow b - Ax^{(m)}, \\ t_m &\leftarrow \frac{\langle r^{(m)}, r^{(m)}\rangle}{\langle r^{(m)}, Ar^{(m)}\rangle}, \\ x^{(m+1)} &\leftarrow x^{(m)} + t_m r^{(m)} && \text{für alle } m \in \mathbb{N}_0. \end{aligned}$$

Ein kleiner Nachteil dieser Version besteht darin, dass wir zwei Multiplikationen mit der Matrix A benötigen, einmal für den Vektor $x^{(m)}$ und einmal für $r^{(m)}$. Dieser Nachteil lässt sich vermeiden, wenn wir ausnutzen, dass sich das Residuum $r^{(m+1)}$ aus dem Residuum $r^{(m)}$ berechnen lässt:

$$\begin{aligned} r^{(m+1)} &= b - Ax^{(m+1)} = b - A(x^{(m)} + t_m r^{(m)}) \\ &= b - Ax^{(m)} - t_m Ar^{(m)} = r^{(m)} - t_m Ar^{(m)} && \text{für alle } m \in \mathbb{N}_0. \end{aligned}$$

Vor Beginn der eigentlichen Iteration berechnen wir das erste Residuum $r^{(0)} = b - Ax^{(0)}$. In jedem Schritt der Iteration brauchen wir dann nur den Hilfsvektor $a^{(m)} = Ar^{(m)}$ zu berechnen, mit dessen Hilfe wir sowohl t_m als auch $r^{(m+1)}$ konstruieren können. Das resultierende Gradientenverfahren ist in Abbildung 3.3 zusammengefasst.

Eine bemerkenswerte Eigenschaft dieser Iteration ist, dass wir lediglich dazu in der Lage sein müssen, die Matrix A mit Vektoren zu multiplizieren. In vielen Anwendungen können wir es deshalb beispielsweise vermeiden, die Matrix abzuspeichern, weil sich ihre Einträge effizient im Zuge der Matrix-Vektor-Multiplikation rekonstruieren lassen.

3.6 Konjugierte Gradienten*

Im Gradientenverfahren wählen wir den Skalierungsparameter t_{opt} gerade so, dass

$$J(x + t_{\text{opt}}p) \leq J(x + tp) \qquad \text{für alle } t \in \mathbb{R}$$

gilt. Die neue Näherungslösung $x' := x + t_{\text{opt}}p$ erfüllt dann

$$J(x') = J(x + t_{\text{opt}}p) \leq J(x + (t_{\text{opt}} + t)p) = J(x' + tp) \qquad \text{für alle } t \in \mathbb{R},$$

wir können x' also nicht mehr verbessern, indem wir Vielfache des Vektors p hinzuaddieren.

Definition 3.13 (Optimalität) *Sei $\mathcal{V} \subseteq \mathbb{R}^n$ ein Teilraum. Wir nennen $x \in \mathbb{R}^n$ optimal bezüglich $\mathcal{V}$, falls*

$$J(x) \leq J(x + p) \qquad \textit{für alle } p \in \mathcal{V}$$

gilt, falls x also nicht durch Addition eines Elements des Vektorraums verbessert werden kann.

Das Gradientenverfahren sorgt durch die geschickte Wahl des Skalierungsparameter t_m dafür, dass der Vektor $x^{(m+1)}$ optimal bezüglich des Aufspanns des Residuums $r^{(m)}$ ist. Leider beobachtet man in der Praxis, dass $x^{(m+2)}$ diese Optimalität bereits wieder verloren haben kann.

Unser Ziel ist es, das Verfahren so zu modifizieren, dass eine einmal erreichte Optimalität bezüglich eines gegebenen Teilraums erhalten bleibt.

Sei also $\mathcal{V} \subseteq \mathbb{R}^n$ ein Teilraum, und sei $x \in \mathbb{R}^n$ optimal bezüglich dieses Teilraums. Wir suchen eine Richtung $p \in \mathbb{R}^n \setminus \{0\}$ derart, dass die nächste Näherungslösung $x' = x + tp$ immer noch optimal bezüglich des Raums $\mathcal{V}$ ist.

Sei $q \in \mathcal{V} \setminus \{0\}$. Da x optimal bezüglich $\mathcal{V}$ ist, gilt insbesondere

$$J(x) \leq J(x + tq) \qquad \text{für alle } t \in \mathbb{R},$$

und mit Lemma 3.12 folgt daraus, dass $t_{\text{opt}} = 0$ gelten muss, also

$$\langle q, b - Ax \rangle = 0. \tag{3.17}$$

Wenn wir die Optimalität erhalten möchten, muss demnach

$$0 = \langle q, b - Ax' \rangle = \langle q, b - A(x + tp) \rangle = \langle q, b - Ax \rangle + t\langle q, Ap \rangle$$

gelten. Mit (3.17) folgt

$$0 = t\langle q, Ap \rangle.$$

Also müssen wir entweder $t = 0$ setzen, also die eher nutzlose Wahl $x' = x$ treffen, oder wir müssen dafür sorgen, dass die neue Richtung $p \in \mathbb{R}^n \setminus \{0\}$ die Eigenschaft

$$\langle q, Ap \rangle = 0 \qquad \text{für alle } q \in \mathcal{V}$$

besitzt. Der Vektor Ap muss also senkrecht auf dem gesamten Teilraum $\mathcal{V}$ stehen. Man spricht davon, dass p *A-konjugiert* auf $\mathcal{V}$ ist.

Im ersten Schritt des Verfahrens ist $x^{(0)}$ nur bezüglich des trivialen Teilraums $\mathcal{V}_0 = \{0\}$ optimal, so dass wir als erste Richtung wie bisher das Residuum $p^{(0)} := r^{(0)}$, also den negativen Gradienten, verwenden können.

Im zweiten Schritt ist $x^{(1)}$ aufgrund der Wahl des Parameters t_0 optimal bezüglich des eindimensionalen Teilraums $\mathcal{V}_1 = \operatorname{span}\{r^{(0)}\} = \operatorname{span}\{p^{(0)}\}$, also sollten wir $p^{(1)}$ so wählen, dass

$$\langle p^{(1)}, Ap^{(0)} \rangle = 0$$

gilt. Im Gradientenverfahren würden wir nun einfach $r^{(1)} = p^{(1)}$ setzen und damit wahrscheinlich diese Gleichung verletzen. Wir können allerdings ein geeignetes Vielfaches von $p^{(0)}$ von $r^{(1)}$ subtrahieren, um dieses Problem zu lösen: Mit dem Ansatz $p^{(1)} = r^{(1)} - \mu p^{(0)}$ erhalten wir

$$0 \stackrel{!}{=} \langle p^{(1)}, Ap^{(0)} \rangle = \langle r^{(1)} - \mu p^{(0)}, Ap^{(0)} \rangle = \langle r^{(1)}, Ap^{(0)} \rangle - \mu \langle p^{(0)}, Ap^{(0)} \rangle,$$

und da A positiv definit ist, können wir

$$\mu = \frac{\langle r^{(1)}, Ap^{(0)} \rangle}{\langle p^{(0)}, Ap^{(0)} \rangle}$$

setzen, um diese Gleichung zu erfüllen. Insgesamt haben wir

$$p^{(1)} = r^{(1)} - \frac{\langle r^{(1)}, Ap^{(0)} \rangle}{\langle p^{(0)}, Ap^{(0)} \rangle} p^{(0)}.$$

Wir können entsprechend fortfahren: $x^{(m)}$ ist optimal bezüglich des m-dimensionalen Teilraums $\mathcal{V}_m = \operatorname{span}\{p^{(0)}, \ldots, p^{(m-1)}\}$, und die Richtung

$$p^{(m)} = r^{(m)} - \sum_{\ell=0}^{m-1} \frac{\langle r^{(m)}, Ap^{(\ell)} \rangle}{\langle p^{(\ell)}, Ap^{(\ell)} \rangle} p^{(\ell)} \tag{3.18}$$

ist A-konjugiert auf $\mathcal{V}_m$. Die Richtung $p^{(m)}$ mit Hilfe dieser Gleichung zu bestimmen, wäre sehr aufwendig, weil viele Skalarprodukte anfallen würden und die Vektoren $Ap^{(\ell)}$ und $p^{(\ell)}$ gespeichert werden müssten.

Glücklicherweise lässt sich der Rechenaufwand erheblich reduzieren:

Definition 3.14 (Krylow-Raum) *Sei $A \in \mathbb{R}^{n \times n}$, sei $z \in \mathbb{R}^n$, sei $m \in \mathbb{N}$. Den Raum*

$$\mathcal{K}(A, z, m) := \operatorname{span}\{z, Az, \ldots, A^{m-1}z\}$$

nennen wir den m-ten KRYLOW-Raum *zu der Matrix A und dem Vektor z.*

Lemma 3.15 (Krylow-Raum) *Falls $p^{(0)}, \ldots, p^{(m-1)} \neq 0$ gilt, haben wir*

$$\mathcal{V}_m = \operatorname{span}\{p^{(0)}, \ldots, p^{(m-1)}\} = \operatorname{span}\{r^{(0)}, \ldots, r^{(m-1)}\} = \mathcal{K}(A, r^{(0)}, m).$$

Beweis. Mit einer einfachen Induktion können wir

$$\mathcal{V}_m = \operatorname{span}\{p^{(0)}, \ldots, p^{(m-1)}\} \subseteq \operatorname{span}\{r^{(0)}, \ldots, r^{(m-1)}\} \qquad \text{für alle } m \in \mathbb{N}_0$$

zeigen, denn die Richtungen werden ja gerade aus den Residuen konstruiert.

Für $m \in \mathbb{N}_0$ ist das nächste Residuum durch

$$r^{(m+1)} = b - Ax^{(m+1)} = b - A(x^{(m)} - t_m p^{(m)}) = r^{(m)} - t_m A p^{(m)}$$

gegeben, und da wir bereits wissen, dass $p^{(m)} \in \operatorname{span}\{r^{(0)}, \ldots, r^{(m)}\}$ gilt, folgt

$$r^{(m+1)} \in \operatorname{span}\{r^{(m)}, Ar^{(0)}, \ldots, Ar^{(m)}\}.$$

Mit einer einfachen Induktion erhalten wir

$$\operatorname{span}\{r^{(0)}, \ldots, r^{(m-1)}\} \subseteq \operatorname{span}\{r^{(0)}, Ar^{(0)}, \ldots, A^{m-1} r^{(0)}\} \qquad \text{für alle } m \in \mathbb{N}_0.$$

Die drei Räume können alle höchstens die Dimension m aufweisen, also müssen wir nur noch zeigen, dass die Vektoren $p^{(0)}, \ldots, p^{(m-1)}$ linear unabhängig sind.

Seien also $\alpha_0, \ldots, \alpha_{m-1} \in \mathbb{R}$ so gegeben, dass

$$0 = \sum_{\ell=0}^{m-1} \alpha_\ell p^{(\ell)}$$

gilt. Für jedes $k \in [0 : m-1]$ folgt

$$0 = \langle p^{(k)}, A \sum_{\ell=0}^{m-1} \alpha_\ell p^{(\ell)} \rangle = \sum_{\ell=0}^{m-1} \alpha_\ell \langle p^{(k)}, Ap^{(\ell)} \rangle = \alpha_k \langle p^{(k)}, Ap^{(k)} \rangle,$$

da die Richtungen A-konjugiert sind. Da A positiv definit ist und $p^{(k)} \neq 0$ nach Voraussetzung gilt, folgt $\alpha_k = 0$. Also sind die Richtung linear unabhängig und die Dimension aller Räume gleich m. Da die Räume ineinander enthalten sind, sind sie damit auch identisch. ■

Nun können wir unser Verfahren verbessern: Wir nehmen an, dass $p^{(0)}, \ldots, p^{(m-1)} \neq 0$ gilt, dass also das Verfahren bis zu dem m-ten Schritt durchgeführt werden kann.

Sei $\ell \in [0 : m-1]$. Dann gilt

$$p^{(\ell)} \in \mathcal{K}(A, r^{(0)}, \ell + 1)$$

also auch

$$Ap^{(\ell)} \in \mathcal{K}(A, r^{(0)}, \ell + 2) = \mathcal{V}_{\ell+2}.$$

Nach Konstruktion ist $x^{(m)}$ optimal bezüglich $\mathcal{V}_m$, also gilt

$$\langle q, r^{(m)} \rangle = \langle q, b - Ax^{(m)} \rangle = 0 \qquad \text{für alle } q \in \mathcal{V}_m.$$

```
procedure conjgrad(A, b, ε, var x);
r ← b − Ax;
p ← r;
while ‖r‖ > ε‖b‖ do begin
  a ← Ap;
  t ← ⟨p, r⟩/⟨p, a⟩;
  x ← x + tp;
  r ← r − ta;
  μ ← ⟨r, a⟩/⟨p, a⟩;
  p ← r − μp
end
```

Abbildung 3.4: Verfahren der konjugierten Gradienten für lineare Gleichungssysteme mit einer symmetrischen und positiv definiten Matrix

Falls nun $\ell \leq m-2$ gilt, also $\ell+2 \leq m$, so haben wir $Ap^{(\ell)} \in \mathcal{V}_m$ und damit

$$\langle r^{(m)}, Ap^{(\ell)} \rangle = 0 \qquad \text{für alle } \ell \in [0:m-2].$$

In der Gleichung (3.18) entfallen damit in der Summe alle Terme bis auf den letzten, so dass nur

$$p^{(m)} = r^{(m)} - \frac{\langle r^{(m)}, Ap^{(m-1)} \rangle}{\langle p^{(m-1)}, Ap^{(m-1)} \rangle} p^{(m-1)}$$

übrig bleibt. Das resultierende Verfahren ist in Abbildung 3.4 zusammengefasst, es ist unter dem Namen *Verfahren der konjugierten Gradienten* (engl. *conjugate gradients method*) bekannt und geht auf Hestenes und Stiefel zurück.

Es stellt sich die Frage, ob dieses Verfahren überhaupt durchführbar ist, denn es ist nicht offensichtlich, dass die im Nenner auftretenden Skalarprodukte $\langle p^{(m)}, Ap^{(m)} \rangle$ ungleich null sind.

Glücklicherweise ist das kein Problem: Da A positiv definit ist, kann das Skalarprodukt nur gleich null sein, wenn $p^{(m)} = 0$ gilt. In diesem Fall haben wir nach Konstruktion

$$0 = p^{(m)} = r^{(m)} - \sum_{\ell=0}^{m-1} \frac{\langle r^{(m)}, Ap^{(\ell)} \rangle}{\langle p^{(\ell)}, Ap^{(\ell)} \rangle} p^{(\ell)},$$

also insbesondere

$$r^{(m)} \in \mathcal{V}_m = \operatorname{span}\{p^{(0)}, \dots, p^{(m-1)}\}.$$

Da aber $x^{(m)}$ nach Konstruktion optimal bezüglich dieses Teilraums ist, folgt

$$\|r^{(m)}\|^2 = \langle r^{(m)}, r^{(m)} \rangle = \langle r^{(m)}, b - Ax^{(m)} \rangle = 0,$$

also $r^{(m)} = 0$. Sollte also $p^{(m)} = 0$ auftreten, gilt auch $r^{(m)} = 0$ und damit $Ax^{(m)} = b$, in diesem Fall haben wir also bereits die exakte Lösung des linearen Gleichungssystems berechnet.

Bemerkung 3.16 (Effizienz) *Im Vergleich zu dem Gradientenverfahren benötigt das Verfahren der konjugierten Gradienten den zusätzlichen Hilfsvektor p, und in jedem Schritt müssen ein weiteres Skalarprodukt und eine weitere Linearkombination ausgeführt werden.*

In der Praxis nimmt man den höheren Rechenaufwand und Speicherbedarf allerdings gerne in Kauf, weil sie mit einer erheblich *schnelleren Konvergenz einher gehen.*

4 Simulationen

Eines der wichtigsten Anwendungsgebiete numerischer Verfahren ist die Simulation naturwissenschaftlicher Phänomene. Ausgehend von Naturgesetzen wie den NEWTON-Axiomen der Mechanik oder den MAXWELL-Gleichungen der Elektrodynamik können mathematische Modelle natürlicher Vorgänge konstruiert werden, und diese Modelle können mit Hilfe numerischer Verfahren analysiert werden.

Dieser Zugang kann beispielsweise verwendet werden, um Prototypen im Computer zu simulieren, statt sie tatsächlich zu bauen, und so Kosten zu sparen. Es ist aber auch möglich, Experimente zu simulieren, die sich in der Realität gar nicht durchführen ließen, beispielsweise um die Entwicklung des Klimas vorherzusagen oder Theorien über die Zustände im Inneren von Sternen zu formulieren.

4.1 Federpendel

Als erstes einfaches Beispiel untersuchen wir das Federpendel: Ein Gewicht der Masse $m \in \mathbb{R}_{>0}$ ist an einem festen Punkt mit einer Feder aufgehängt. Wenn wir die Auslenkung des Gewichts gegenüber seiner Ruhelage zu einem Zeitpunkt $t \in \mathbb{R}$ mit $x(t)$ bezeichnen, erhalten wir eine Funktion

$$x\colon \mathbb{R} \to \mathbb{R}.$$

Entsprechend bezeichnen wir die Geschwindigkeit des Gewichts zu einem Zeitpunkt $t \in \mathbb{R}$ mit $v(t)$ und erhalten so eine zweite Funktion

$$v\colon \mathbb{R} \to \mathbb{R}.$$

Das *erste* NEWTON*sche Gesetz* besagt, dass die Geschwindigkeit die Ableitung der Auslenkung ist, dass also

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

gilt. Die Veränderung der Geschwindigkeit wird durch die *Beschleunigung*

$$a(t) := v'(t) \qquad \text{für alle } t \in \mathbb{R}$$

beschrieben, und das *zweite* NEWTON*sche Gesetz* besagt, dass sie proportional zu der wirkenden Kraft ist. Genauer gilt

$$m\, a(t) = f(t) \qquad \text{für alle } t \in \mathbb{R}, \tag{4.2}$$

wobei m die Masse und $f(t)$ die zu einem Zeitpunkt $t \in \mathbb{R}$ wirkende Kraft ist.

Bisher taucht die Feder noch nicht in unseren Gleichungen auf. Zumindest für geringe Auslenkungen kann sie durch das HOOKEsche Gesetz

$$f(t) = -c\,x(t) \qquad \text{für alle } t \in \mathbb{R} \tag{4.3}$$

beschrieben werden, in der die *Federkonstante* $c \in \mathbb{R}_{>0}$ angibt, welche Kraft die Feder ausübt, um das Gewicht wieder in den Ruhezustand zurück zu bewegen.

Indem wir die Gleichungen ineinander einsetzen erhalten wir

$$x'(t) = v(t), \qquad v'(t) = -\frac{c}{m}x(t) \qquad \text{für alle } t \in \mathbb{R}. \tag{4.4}$$

Es handelt sich um eine *gewöhnliche Differentialgleichung*, die das Verhalten des Federpendels beschreibt.

Um eine eindeutige Lösung zu erhalten, müssen wir Auslenkung und Geschwindigkeit zu einem beliebigen Zeitpunkt vorgeben. Der Einfachheit halber entscheiden wir uns für $t = 0$ und fordern

$$x(0) = x_0, \qquad v(0) = v_0$$

mit einer gegebenen Anfangsauslenkung $x_0 \in \mathbb{R}$ und einer gegebenen Anfangsgeschwindigkeit $v_0 \in \mathbb{R}$.

In diesem sehr einfachen Fall lassen sich die Funktionen x und v explizit angeben, indem man geeignete Sinus- und Cosinus-Funktionen verwendet. In allgemeineren Fällen ist das nicht möglich, dann müssen numerische Näherungsverfahren zum Einsatz kommen.

Für die Behandlung gewöhnlicher Differentialgleichungen haben sich *Zeitschrittverfahren* als sehr erfolgreich erwiesen: Wir betrachten die Funktionen x und v zu festen Zeitpunkten $0 = t_0 < t_1 < t_2 < \ldots$ und suchen nach Algorithmen, die zu bekannten Werten in der Vergangenheit eine Vorhersage der Werte in der Zukunft konstruieren.

Eines der einfachsten Verfahren ist das *explizite* EULER-*Verfahren*, das auf der Idee beruht, die Ableitungen $x'(t)$ und $v'(t)$ durch die Differenzenquotienten

$$x'(t) \approx \frac{x(t+\delta) - x(t)}{\delta}, \qquad v'(t) \approx \frac{v(t+\delta) - v(t)}{\delta} \tag{4.5}$$

anzunähern. Indem wir diese Gleichungen nach $x(t+\delta)$ und $v(t+\delta)$ auflösen, erhalten wir

$$x(t+\delta) \approx x(t) + \delta x'(t), \qquad v(t+\delta) \approx v(t) + \delta v'(t),$$

und wir können (4.4) einsetzen, um zu

$$x(t+\delta) \approx x(t) + \delta v(t), \qquad v(t+\delta) \approx v(t) - \delta\frac{c}{m}x(t)$$

zu gelangen. Wenn uns also $x(t)$ und $v(t)$ bekannt sind, können wir näherungsweise $x(t+\delta)$ und $v(t+\delta)$ vorhersagen.

Wir wählen eine *Zeitschrittweite* $\delta \in \mathbb{R}_{>0}$, definieren die Zeitpunkte durch

$$t_i := \delta i \qquad \text{für alle } i \in \mathbb{N}_0,$$

und erhalten die Regel

$$\widetilde{x}(t_0) := x_0, \qquad \widetilde{v}(t_0) := v_0,$$
$$\widetilde{x}(t_{i+1}) := \widetilde{x}(t_i) + \delta\widetilde{v}(t_i), \qquad \widetilde{v}(t_{i+1}) := \widetilde{v}(t_i) - \delta\frac{c}{m}\widetilde{x}(t_i) \qquad \text{für alle } i \in \mathbb{N}_0,$$

mit der wir Näherungslösungen $\widetilde{x}(t_i) \approx x(t_i)$ und $\widetilde{v}(t_i) \approx v(t_i)$ berechnen können.

Die Genauigkeit dieser Näherungen ist allerdings relativ gering: Wenn wir davon ausgehen, dass x zweimal stetig differenzierbar ist, haben wir mit dem Satz 3.8 von Taylor

$$x(t+\delta) = x(t) + x'(t)\delta + x''(\eta)\frac{\delta^2}{2},$$
$$\frac{x(t+\delta) - x(t)}{\delta} = x'(t) + x''(\eta)\frac{\delta}{2}$$

für ein $\eta \in [t, t+\delta]$, so dass sich der durch den Differenzenquotienten (4.5) eingeführte Fehler ungefähr proportional zu der Schrittweite δ verhalten wird.

Wir können allerdings bessere Differenzenquotienten konstruieren: Sei x dreimal stetig differenzierbar. Wieder mit dem Satz 3.8 von Taylor erhalten wir

$$x(t+\delta) = x(t) + x'(t)\delta + x''(t)\frac{\delta^2}{2} + x'''(\eta_+)\frac{\delta^3}{6},$$
$$x(t-\delta) = x(t) - x'(t)\delta + x''(t)\frac{\delta^2}{2} - x'''(\eta_-)\frac{\delta^3}{6},$$
$$x(t+\delta) - x(t-\delta) = 2x'(t)\delta + (x'''(\eta_+) + x'''(\eta_-))\frac{\delta^3}{6},$$
$$\frac{x(t+\delta) - x(t-\delta)}{2\delta} = x'(t) + \frac{x'''(\eta_+) + x'''(\eta_-)}{2}\frac{\delta^2}{6}$$

mit geeigneten $\eta_+ \in [t, t+\delta]$ und $\eta_- \in [t-\delta, t]$. Da die dritte Ableitung stetig ist, können wir mit dem Zwischenwertsatz ein $\eta \in [t-\delta, t+\delta]$ finden, für das

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

gilt, so dass sich unsere Gleichung kürzer als

$$\frac{x(t+\delta) - x(t-\delta)}{2\delta} = x'(t) + x'''(\eta)\frac{\delta^2}{6}$$

schreiben lässt. Der auf der linken Seite stehende *zentrale Differenzenquotient* wird also ebenfalls eine Näherung der ersten Ableitung berechnen, allerdings mit einer zu δ^2 proportionalen Genauigkeit.

Wenn wir wie zuvor verfahren, erhalten wir

$$\frac{x(t+\delta)-x(t-\delta)}{2\delta} \approx x'(t) = v(t), \qquad \frac{v(t+\delta)-v(t-\delta)}{2\delta} \approx v'(t) = -\frac{c}{m}x(t),$$
$$x(t+\delta) \approx x(t-\delta) + 2\delta v(t), \qquad v(t+\delta) \approx v(t-\delta) - 2\delta\frac{c}{m}x(t).$$

Um $x(t+\delta)$ berechnen zu können, benötigen wir $x(t-\delta)$ und $v(t)$, anders als bei dem zuvor betrachteten EULER-Verfahren werden also x und v zu unterschiedlichen Zeitpunkten gebraucht.

Also geben wir uns damit zufrieden, $x(t_i)$ nur für *gerades* $i \in \mathbb{N}_0$ zu berechnen und $v(t_i)$ nur für *ungerades* $i \in \mathbb{N}_0$:

$$\begin{aligned} \widetilde{x}(t_{2i+2}) &:= \widetilde{x}(t_{2i}) + 2\delta\widetilde{v}(t_{2i+1}), \\ \widetilde{v}(t_{2i+3}) &:= \widetilde{v}(t_{2i+1}) - 2\delta\frac{c}{m}\widetilde{x}(t_{2i+2}) \qquad \text{für alle } i \in \mathbb{N}_0. \end{aligned}$$

Da dabei in jedem Schritt ein Zeitpunkt „übersprungen“ wird, trägt dieses Verfahren den Namen *Leapfrog-Verfahren.* Es wurde offenbar schon von NEWTON verwendet, später von ENCKE und STÖRMER.

Es ist allerdings noch nicht vollständig: Für die Geschwindigkeit v steht uns nur der Anfangswert v_0 zu dem Zeitpunkt t_0 zur Verfügung, wir brauchen aber einen Wert zu dem Zeitpunkt t_1. Wir lösen das Problem, indem wir einen Schritt des EULER-Verfahrens verwenden:

$$\widetilde{x}(t_0) := x_0, \qquad \widetilde{v}(t_1) := v_0 - \delta\frac{c}{m}\widetilde{x}(t_0).$$

Es lässt sich zeigen, dass in diesem Sonderfall $\widetilde{v}(t_1)$ immer noch mit einer zu δ^2 proportionalen Genauigkeit berechnet wird, so dass die durch den zentralen Differenzenquotienten erreichte Verbesserung nicht verloren geht.

Ein großer Vorteil des Leapfrog-Verfahrens besteht darin, dass es die höhere Genauigkeit ohne wesentlichen Mehraufwand gegenüber dem EULER-Verfahren erreicht: In jedem Schritt müssen dieselben Operationen ausgeführt werden, lediglich mit der doppelten Schrittweite.

4.2 Wellengleichung

Ein etwas interessanteres Beispiel bietet uns die *Wellengleichung*, die beispielsweise die Schwingung einer Saite (etwa einer Gitarre oder eines Klaviers) beschreibt. Der Einfachheit halber nehmen wir an, dass die Saite eine Länge von eins aufweist und horizontal von links nach rechts eingespannt ist. Neben der Zeit $t \in \mathbb{R}$ spielt nun auch der Ort $s \in [0,1]$ auf der Saite eine Rolle, bei dem $s = 0$ dem linken und $s = 1$ dem rechten Randpunkt entspricht.

Die vertikale Auslenkung eines Punkts an Position $s \in [0,1]$ zu der Zeit $t \in \mathbb{R}$ gegenüber der Ruhelage bezeichnen wir mit $x(t,s)$, seine vertikale Geschwindigkeit mit

$v(t,s)$. Wie zuvor erhalten wir mit dem ersten NEWTONschen Gesetz

$$\frac{\partial x}{\partial t}(t,s) = v(t,s) \qquad \text{für alle } t \in \mathbb{R},\ s \in [0,1].$$

Die wirkende Kraft ist nach TAYLOR proportional zu der Krümmung der Saite, die sich wiederum für geringe Auslenkungen durch die zweite Ableitung approximieren lässt, so dass wir

$$\frac{\partial v}{\partial t}(t,s) = c\frac{\partial^2 x}{\partial s^2}(t,s) \qquad \text{für alle } t \in \mathbb{R},\ s \in [0,1]$$

erhalten. Wir wissen bereits, dass wir die kontinuierliche Zeit durch eine Folge diskreter Zeitpunkte $t_0 < t_1 < t_2 < \ldots$ ersetzen können.

Bei der Konstruktion unserer Zeitschrittverfahren haben wir die erste Ableitung durch Differenzenquotienten ersetzt, die lediglich Werte in den diskreten Zeitpunkten benötigen.

Auch für einen festen Zeitpunkt t_i sind allerdings $u(t_i,\cdot)$ und $v(t_i,\cdot)$ immer noch Funktionen in der Ortsvariablen s, die ein Computer nicht exakt darstellen kann, da er nur über eine endliche Menge an Speicher verfügt, es aber unendlich viele Punkte $s \in [0,1]$ gibt.

Wir werden also auch im Ort eine *Diskretisierung* vornehmen müssen, bei der wir uns auf endlich viele Punkte s_j beschränken. Analog zu unseren Zeitschrittverfahren ist es dann aber erforderlich, die in der Wellengleichung auftretende zweite Ableitung in der Variablen s durch einen Differenzenquotienten zu ersetzen.

Setzen wir dazu voraus, dass x in der s-Variablen viermal stetig differenzierbar ist. Für ein $s \in (a,b)$ und ein $h \in \mathbb{R}_{>0}$ mit $s+h, s-h \in [a,b]$ erhalten wir mit dem Satz von TAYLOR die Gleichungen

$$\begin{aligned}
x(t,s+h) &= x(t,s) + \frac{\partial x}{\partial s}(t,s)h + \frac{\partial^2 x}{\partial s^2}(t,s)\frac{h^2}{2} + \frac{\partial^3 x}{\partial s^3}(t,s)\frac{h^3}{6} + \frac{\partial^4 x}{\partial s^4}(t,\eta_+)\frac{h^4}{24},\\
x(t,s-h) &= x(t,s) - \frac{\partial x}{\partial s}(t,s)h + \frac{\partial^2 x}{\partial s^2}(t,s)\frac{h^2}{2} - \frac{\partial^3 x}{\partial s^3}(t,s)\frac{h^3}{6} + \frac{\partial^4 x}{\partial s^4}(t,\eta_-)\frac{h^4}{24}
\end{aligned}$$

mit $\eta_+ \in [s, s+h]$ und $\eta_- \in [s-h, s]$, und durch Addition ergibt sich

$$\begin{aligned}
x(t,s+h) - 2x(t,s) + x(t,s-h) &= \frac{\partial^2 x}{\partial s^2}(t,s)h^2 + \left(\frac{\partial^4 x}{\partial s^4}(t,\eta_+) + \frac{\partial^4 x}{\partial s^4}(t,\eta_-)\right)\frac{h^4}{24},\\
\frac{x(t,s+h) - 2x(t,s) + x(t,s-h)}{h^2} &= \frac{\partial^2 x}{\partial s^2}(t,s) + \left(\frac{\partial^4 x}{\partial s^4}(t,\eta_+) + \frac{\partial^4 x}{\partial s^4}(t,\eta_-)\right)\frac{h^2}{24}.
\end{aligned}$$

Da die vierte Ableitung $\frac{\partial^4 x}{\partial s^4}$ stetig ist, finden wir mit dem Zwischenwertsatz ein $\eta \in [s-h, s+h]$ mit

$$\frac{1}{2}\left(\frac{\partial^4 x}{\partial s^4}(t,\eta_+) + \frac{\partial^4 x}{\partial s^4}(t,\eta_-)\right) = \frac{\partial^4 x}{\partial s^4}(t,\eta),$$

so dass wir insgesamt

$$\frac{x(t,s+h) - 2x(t,s) + x(t,s-h)}{h^2} = \frac{\partial^2 x}{\partial s^2}(t,s) + \frac{\partial^4 x}{\partial s^4}(t,\eta)\frac{h^2}{12}$$

gezeigt haben. Der Differenzenquotient auf der linken Seite approximiert also die zweite partielle Ableitung bezüglich der Ortsvariablen s mit einem zu h^2 proportionalen Fehler.

Der Differenzenquotient benötigt lediglich die Werte $s+h$, s und $s-h$, so dass wir das kontinuierliche Intervall $[0,1]$ durch diskrete Punkte $0 = s_0 < s_1 < \ldots < s_n = 1$ ersetzen können. Der Einfachheit halber verwenden wir *äquidistante* Punkte, wählen also $n \in \mathbb{N}$ und setzen

$$h := \frac{1}{n+1}, \qquad s_j := hj \qquad \text{für alle } j \in [0 : n+1].$$

Die Wellengleichung wird dann durch das System

$$\begin{aligned} \frac{\partial x}{\partial t}(t, s_j) &= v(t, s_j), \\ \frac{\partial v}{\partial t}(t, s_j) &= \frac{c}{h^2}\Big(x(t, s_{j+1}) - 2x(t, s_j) + x(t, s_{j-1})\Big) \qquad \text{für alle } t \in \mathbb{R},\ j \in [1 : n] \end{aligned}$$

approximiert. Wir können uns Schreibarbeit sparen, indem wir die Matrix

$$A := \frac{c}{h^2} \begin{pmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}$$

einführen und die Auslenkungen und Geschwindigkeiten in Vektoren

$$x_h(t) := \begin{pmatrix} x(t, s_1) \\ \vdots \\ x(t, s_n) \end{pmatrix}, \qquad v_h(t) := \begin{pmatrix} v(t, s_1) \\ \vdots \\ v(t, s_n) \end{pmatrix} \qquad \text{für alle } t \in \mathbb{R}$$

zusammenfassen und

$$x_h'(t) = v_h(t), \qquad v_h'(t) = -Ax_h(t) + \frac{c}{h^2}(x(t,0) + x(t,1)) \qquad \text{für alle } t \in \mathbb{R} \tag{4.6}$$

schreiben. Der zweite Term auf der rechten Seite beschreibt dabei den Einfluss der Randwerte $x(t,0)$ und $x(t,1)$. Es handelt sich um eine gewöhnliche Differentialgleichung, die im Fall fixierter Randpunkte, also für $x(t,0) = x(t,1) = 0$, der das Federpendel beschreibenden Gleichung (4.4) sehr ähnlich sieht.

Im Folgenden beschränken wir uns auf diesen Fall und erhalten

$$x_h'(t) = v_h(t), \qquad v_h'(t) = -Ax_h(t) \qquad \text{für alle } t \in \mathbb{R}.$$

Dieses System können wir wieder mit Zeitschrittverfahren behandeln. Für das explizite Euler-Verfahren ergibt sich

$$\begin{aligned} \widetilde{x}_h(t_0) &= x_{0,h}, & \widetilde{v}_h(t_0) &= v_{0,h}, \\ \widetilde{x}_h(t_{i+1}) &= \widetilde{x}_h(t_i) + \delta \widetilde{v}_h(t_i), & \widetilde{v}_h(t_{i+1}) &= \widetilde{v}_h(t_i) - \delta A \widetilde{x}_h(t_i) \qquad \text{für alle } i \in \mathbb{N}_0, \end{aligned}$$

während wir für das Leapfrog-Verfahren die Gleichungen

$$\begin{aligned}
\widetilde{x}_h(t_0) &= x_{0,h}, \\
\widetilde{v}_h(t_1) &= v_{0,h} - \delta A x_{0,h}, \\
\widetilde{x}_h(t_{2i+2}) &= \widetilde{x}_h(t_{2i}) + 2\delta\widetilde{v}(t_{2i+1}), \\
\widetilde{v}_h(t_{2i+3}) &= \widetilde{v}_h(t_{2i+1}) - 2\delta A\widetilde{x}(t_{2i+2}) \qquad \text{für alle } i \in \mathbb{N}_0
\end{aligned}$$

erhalten. In beiden Fällen sind die Anfangswerte für Auslenkung und Geschwindigkeit durch

$$(x_{0,h})_j := x(0, t_j), \qquad (v_{0,h})_j := v(0, t_j) \qquad \text{für alle } j \in [1:n]$$

gegeben. Da wir für das Leapfrog-Verfahren erwarten dürfen, dass der Differenzenquotient in der Zeit eine Genauigkeit von δ^2 erreicht und der im Ort eine von h^2, verspricht das Verfahren eine brauchbare Genauigkeit bei vertretbarem Rechenaufwand.

Bemerkung 4.1 (BLAS-Implementierung) *Die Multiplikation eines Vektors mit der Matrix A lässt sich mit der* `axpy`*-Funktion der BLAS-Stufe 1 realisieren, und damit auch sowohl das* EULER- *als auch das Leapfrog-Verfahren.*

4.3 Mehrkörpersysteme

Die bisher betrachteten Beispiele sind relativ einfach, so dass sich mit analytischen Methoden Lösungen explizit angeben lassen. Das folgende Beispiel stellt eine erheblich größere Herausforderung dar, so dass der Einsatz numerischer Verfahren nach heutigem Stand der Forschung der beste Weg zu dessen Lösung ist.

Wir betrachten n Körper im dreidimensionalen Raum, die sich unter dem Einfluss des klassischen auf NEWTON zurückgehenden Gravitationsgesetzes bewegen. Der Einfachheit halber nehmen wir an, dass die Körper punktförmig sind. Beispielsweise bei astrophysikalischen Simulationen ist das angesichts der im Verhältnis zu den auftretenden Entfernungen vernachlässigbaren Durchmesser von Sonnen, Planeten und Monden durchaus vertretbar. Die Positionen der Körper können dann durch Funktionen

$$x_i \colon \mathbb{R} \to \mathbb{R}^3 \qquad \text{für alle } i \in [1:n],$$

beschrieben werden, die einem Zeitpunkt $t \in \mathbb{R}$ die Position $x_i(t)$ des i-ten Körpers zu diesem Zeitpunkt zuordnen. Mit $m_1, \ldots, m_n \in \mathbb{R}_{>0}$ bezeichnen wir die Massen der Körper.

Das NEWTONsche Gravitationsgesetz nimmt dann die Form

$$x_i''(t) = \gamma \sum_{\substack{j=1 \\ j\neq i}}^{n} m_j \frac{x_j(t) - x_i(t)}{\|x_j(t) - x_i(t)\|_2^3} \qquad \text{für alle } t \in \mathbb{R},\ i \in [1:n]$$

mit der *Gravitationskonstante* $\gamma \in \mathbb{R}_{>0}$ an, wir können also die auf die Körper zu einem Zeitpunkt $t \in \mathbb{R}$ wirkenden Beschleunigungen aus der jeweiligen Konfiguration der Körper zu diesem Zeitpunkt bestimmen.

Indem wir wieder die Geschwindigkeiten

$$v_i(t) := x_i'(t) \qquad \text{für alle } t \in \mathbb{R},\ i \in [1:n]$$

als zusätzliche Variablen einführen, erhalten wir das System

$$\begin{aligned} x_i'(t) &= v_i(t), \\ v_i'(t) &= \gamma \sum_{\substack{j=1 \\ j \neq i}}^{n} m_j \frac{x_j(t) - x_i(t)}{\|x_j(t) - x_i(t)\|_2^3} \qquad \text{für alle } t \in \mathbb{R},\ i \in [1:n], \end{aligned}$$

das aus $2n$ gekoppelten gewöhnlichen Differentialgleichungen besteht. Zusammen mit Anfangsbedingungen

$$x_i(0) = x_{0,i}, \qquad v_i(0) = v_{0,i} \qquad \text{für alle } i \in [1:n]$$

erhalten wir eine eindeutige Lösung, solange sich die Körper nicht zu nahe kommen[1].

Mit diesem System können wir wie zuvor verfahren, beispielsweise indem wir das Leapfrog-Verfahren anwenden:

$$\begin{aligned} \widetilde{x}_i(t_0) &:= x_{0,i}, \\ \widetilde{v}_i(t_1) &:= v_{0,i} + \delta\gamma \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{x_j(t_0) - x_i(t_0)}{\|x_j(t_0) - x_i(t_0)\|_2^3}, \\ \widetilde{x}_i(t_{2k+2}) &:= \widetilde{x}_i(t_{2k}) + 2\delta \widetilde{v}_i(t_{2k+1}), \\ \widetilde{v}_i(t_{2k+3}) &:= \widetilde{v}_i(t_{2k+1}) + 2\delta\gamma \sum_{\substack{j=1 \\ j \neq i}}^{n} m_j \frac{x_j(t_{2k+2}) - x_i(t_{2k+2})}{\|x_j(t_{2k+2}) - x_i(t_{2k+2})\|_2^3} \qquad \text{für alle } i \in [1:n] \end{aligned}$$

und alle Zeitschritte $k \in \mathbb{N}_0$.

Solange sich die Körper nicht zu nahe kommen, können wir bei Verwendung einer hinreichend kleinen Zeitschrittweite δ darauf hoffen, dass dieses Verfahren eine brauchbare Näherung der tatsächlichen Lösung berechnet.

Sollten sich zwei Körper sehr nahe kommen, können wir beispielsweise die Zeitschrittweite reduzieren, um trotz der größeren Beschleunigungen noch eine gute Genauigkeit zu erreichen.

Solange die Zahl n der Körper relativ klein ist, stellt die Auswertung der Beschleunigungen

$$a_i(t) := \gamma \sum_{\substack{j=1 \\ j \neq i}} m_j \frac{x_j(t) - x_i(t)}{\|x_j(t) - x_i(t)\|_2^3} \qquad \text{für alle } t \in \mathbb{R}$$

[1] Kämen sich die Körper zu nahe, wäre die für die Analyse des Systems wichtige Bedingung verletzt, dass die rechte Seite der Differentialgleichung Lipschitz-stetig sein muss.

keine große Herausforderung dar. Die Situation ändert sich in *Vielkörpersystemen*, beispielsweise bei der Simulation einer Galaxie oder eines komplizierten Moleküls. Die Anzahl der Sterne in der Milchstraße wird auf 100–400 Milliarden geschätzt, so dass die Auswertung der Beschleunigungen für alle Sterne mindestens $(10^{11})^2 = 10^{22}$ Operationen erfordern würde. Selbst wenn man (bei solchen Datenmengen langsame) Speicherzugriffe vernachlässigt, könnte ein moderner Prozessor höchstens einige Milliarden solcher Operationen pro Sekunde ausführen, so dass ungefähr 10^{12} Sekunden nötig wären. Das entspricht ungefähr 30 000 Jahren. Wenn uns ein Großrechner der Exaflop-Klasse zur Verfügung stünde, und davon gibt es derzeit weltweit nur sehr wenige, ließe sich diese Zeit auf einige Stunden oder Tage reduzieren, aber das wäre immer noch nur ein einziger Zeitschritt von vielen, so dass eine sinnvolle Simulation viel zu aufwendig wäre.

Auch dieses — diesmal nicht mathematische, sondern praktische — Problem lässt sich mit geeigneten Approximationstechniken lösen: Angenommen, wir wollen die Beschleunigung berechnen, die zu einem Zeitpunkt $t \in \mathbb{R}$ auf einen in $y := x_i(t)$ befindlichen Körper wirken. Wir wählen einen Quader $s \subseteq \mathbb{R}^3$, der hinreichend weit von y entfernt ist, und bezeichnen mit

$$\hat{s} := \{j \in [1:n] \ : \ x_j(t) \in s\}$$

die Indizes derjenigen Körper, die derzeit in s liegen. Falls der Durchmesser des Quaders s im Verhältnis zu seinem Abstand zu y klein ist, sieht es für einen Betrachter, der von y aus in Richtung s schaut, so aus, als wäre s im Wesentlichen ein Punkt. Daraus entsteht die Idee, alle in s enthaltenen Körper durch einen „imaginären Körper“ zu ersetzen, der zu ungefähr derselben Beschleunigung führt. Wenn wir diesen imaginären Körper im Punkt $x_s \in s$ ansetzen, haben wir

$$\gamma \sum_{j\in\hat{s}} m_j \frac{x_j(t) - y}{\|x_j(t) - y\|_2^3} \approx \gamma \sum_{j\in\hat{s}} m_j \frac{x_s - y}{\|x_s - y\|_2^3} = \gamma m_s \frac{x_s - y}{\|x_s - y\|_2^3},$$

wobei m_s die Summe der Massen aller Körper in s bezeichnet. Wir könnten also alle Massen in s durch eine einzige imaginäre Masse ersetzen und damit sehr viel Rechenzeit sparen.

In dieser Form ist der Ansatz allerdings sehr ungenau. Eine Lösung besteht darin, nicht nur einen einzigen imaginären Körper zu verwenden, sondern mehrere $(x_{s,\nu})_{\nu=1}^k$. Um zu verhindern, dass die Massen einzelner Körper mehrfach gezählt werden, führt man Gewichtungsfunktionen

$$\ell_{s,\nu} \colon s \to \mathbb{R} \qquad\qquad \text{für alle } \nu \in [1:k]$$

ein, deren Summe gleich eins ist, es soll also $\ell_{s,1}(x) + \ldots + \ell_{s,k}(x) = 1$ für alle $x \in s$ gelten. Die Approximation nimmt dann die Form

$$\gamma \sum_{j\in\hat{s}} m_j \frac{x_j(t) - y}{\|x_j(t) - y\|_2^3} \approx \gamma \sum_{j\in\hat{s}} m_j \sum_{\nu=1}^{k} \frac{x_{s,\nu} - y}{\|x_{s,\nu} - y\|_2^3} \ell_{s,\nu}(x_j(s)) = \gamma \sum_{\nu=1}^{k} m_{s,\nu} \frac{x_{s,\nu} - y}{\|x_{s,\nu} - y\|_2^3}$$

mit den nun durch

$$m_{s,\nu} := \sum_{j\in\hat{s}} m_j \ell_{s,\nu}(x_j(t)) \qquad\qquad \text{für alle } \nu \in [1:k]$$

definierten Hilfsmassen an. Da sich die Gewichtungsfunktionen zu eins summieren, gilt

$$\sum_{\nu=1}^{k} m_{s,\nu} = \sum_{\nu=1}^{k} \sum_{j\in\hat{s}} m_j \ell_{s,\nu}(x_j(t)) = \sum_{j\in\hat{s}} m_j \sum_{\nu=1}^{k} \ell_{s,\nu}(x_j(t)) = \sum_{j\in\hat{s}} m_j,$$

die Gesamtmasse der imaginären Körper entspricht also der der ursprünglichen Körper.

Bei diesem verallgemeinerten Ansatz ersetzen wir alle in s liegenden Körper durch k imaginäre Körper, die bei hinreichend weit von s entfernten Körpern ungefähr dieselbe Beschleunigung hervorrufen. Sofern k deutlich kleiner als die Anzahl der ersetzten Körper ist, sparen wir mit diesem Ansatz immer noch erheblich Rechenzeit.

Allerdings dürfen wir diese Approximation nur durchführen, falls y weit von s entfernt ist, wir können also insbesondere nicht den gesamten Weltraum als s wählen, denn dann wäre der Abstand gleich null. Um trotzdem ein allgemein einsatzbares Verfahren zu erhalten, verwenden wir *mehrere* Gebiete s, die in einer Baumstruktur organisiert sind.

Dieser *Clusterbaum* $\mathcal{T}$ muss die folgenden Voraussetzungen erfüllen:

- Die Wurzel enthält alle Punkte $x_1(t), \dots, x_n(t)$.
- Für jeden Knoten s des Clusterbaums und jedes $j \in \hat{s}$ gilt $x_j(t) \in s$.
- Wenn ein Knoten s des Clusterbaums die Söhne $s_1, \dots, s_m$ besitzt, gilt

  $$\hat{s}_1 \cup \ldots \cup \hat{s}_m = \hat{s}.$$

  Jeder Körper in s ist also in *mindestens* einem der Söhne enthalten.
- Wenn s_1 und s_2 Söhne eines Knoten s sind, gilt $\hat{s}_1 \cap \hat{s}_2 = \emptyset$.

  Jeder Körper in s ist also in *höchstens* einem der Söhne enthalten.

Die Knoten eines Clusterbaums nennen wir *Cluster*. Wir verwenden die Kurzschreibweise $s \in \mathcal{T}$ dafür, dass s ein Knoten des Clusterbaums $\mathcal{T}$ ist. Die Menge der Söhne eines Clusters s bezeichnen wir mit $\operatorname{sons}(s)$.

Der verallgemeinerte Algorithmus geht rekursiv vor: Wenn wir die von allen Körpern eines Clusters $s \in \mathcal{T}$ bewirkte Beschleunigung eines Körpers an der Position y berechnen wollen, prüfen wir zunächst, ob y hinreichend weit von s entfernt ist. Dazu werden in der Praxis *Zulässigkeitsbedingungen* der Form

$$\operatorname{diam}(s) \le \operatorname{dist}(y, s)$$

eingesetzt, die den Durchmesser $\operatorname{diam}(s)$ des Clusters s zu seinem Abstand $\operatorname{dist}(y, s)$ zu dem Punkt y in Beziehung setzen.

Falls die Bedingung erfüllt ist, dürfen wir die Beschleunigung mit Hilfe der imaginären Körper approximieren.

Falls die Bedingung nicht erfüllt ist, können zwei Fälle auftreten: Falls s Söhne besitzt, dürfen wir wegen

$$\gamma \sum_{j\in\hat{s}} m_j \frac{x_j(t) - y}{\|x_j(t) - y\|_2^3} = \sum_{s'\in\operatorname{sons}(s)} \gamma \sum_{j\in\hat{s}'} m_j \frac{x_j(t) - y}{\|x_j(t) - y\|_2^3}$$

```
procedure eval_cluster(y, s, var a);
if diam(s) ≤ dist(y, s) then begin
  for ν = 1 to k do
    a ← a + γ m_{s,ν} (x_{s,ν} − y) / ‖x_{s,ν} − y‖_2^3
end else if sons(s) ≠ ∅ then begin
  for s' ∈ sons(s) do
    eval_cluster(y, s', a)
end else begin
  for j ∈ ŝ do
    a ← a + γ m_j (x_j(t) − y) / ‖x_j(t) − y‖_2^3
end
```

Abbildung 4.1: Approximation der Beschleunigung per Clusterbaum.

rekursiv die Berechnung an die Söhne delegieren und ihre Beiträge aufsummieren.

Falls s hingegen keine Söhne hat, berechnen wir die Summe direkt. Da solche *Blattcluster* in der Regel nur wenige Körper enthalten, ist der resultierende Aufwand vertretbar.

Der resultierende rekursive Algorithmus ist in Abbildung 4.1 zusammengefasst. Da bei jedem Rekursionsschritt keine Körper verloren gehen oder hinzu kommen, berechnet er — abgesehen von der durch die imaginären Körper eingeführte Näherung — die vollständige Beschleunigung.

4.4 Populationsdynamik

Numerische Simulationen werden nicht nur in der Physik eingesetzt, sondern auch in der Biologie, beispielsweise um die Dynamik der Populationen bestimmter Lebenwesen zu untersuchen.

Ein Beispiel sind die Lotka-Volterra-Gleichungen

$$\begin{aligned} b'(t) &= b(t)(\alpha - \beta_1 r(t)), \\ r'(t) &= r(t)(\beta_2 b(t) - \gamma) \qquad \text{für alle } t \in \mathbb{R}, \end{aligned}$$

die ein Raubtier-Beutetier-System modellieren. $b(t) \in \mathbb{R}$ ist dabei eine Näherung der Anzahl der Beutetiere zu einem Zeitpunkt $t \in \mathbb{R}$, während $r(t) \in \mathbb{R}$ eine Näherung der Anzahl der Raubtiere ist.

Der Parameter $\alpha > 0$ gibt an, wie schnell sich die Beutetiere vermehren würden, wenn es keine Raubtiere gäbe, die Parameter $\beta_1, \beta_2 > 0$ beschreiben die Wechselwirkung zwischen Raubtieren und Beutetieren, also dass die Anzahl der Beutetiere sinkt, wenn sie gefressen werden, während die Raubtiere sich um so schneller vermehren, je wohlgenährter sie sind. Der Parameter $\gamma > 0$ schließlich gibt die Geschwindigkeit an, mit der die Raubtiere aussterben würden, falls sie nichts mehr fressen können.

Falls wir die Populationsgrößen $b(t_0)$, $r(t_0)$ zu einem festen Zeitpunkt $t_0 \in \mathbb{R}$ kennen, können wir mit Hilfe des Differentialgleichungssystems die Entwicklung der Population vorhersagen.

Eine geschlossene Formel für die Lösung ist nicht bekannt, also kommen in der Praxis numerische Näherungsverfahren zum Einsatz. Das explizite EULER-Verfahren beispielsweise können wir unmittelbar einsetzen, es verspricht aber nur eine relativ geringe Genauigkeit. Das Leapfrog-Verfahren ist etwas unattraktiv, weil beispielsweise $b'(t)$ sowohl von $b(t)$ als auch von $r(t)$ abhängt, so dass wir beide Größen zu allen Zeitpunkten berechnen müssten und damit den Vorteil des Leapfrog-Verfahrens verlieren.

Eine Alternative ist das RUNGE-*Verfahren*: Wir approximieren wieder die Ableitung mit dem zentralen Differenzenquotienten

$$\begin{aligned}
\frac{b(t+\delta)-b(t)}{\delta} &\approx b'(t+\delta/2) = b(t+\delta/2)(\alpha - \beta_1 r(t+\delta/2)),\\
b(t+\delta) &\approx b(t) + \delta b(t+\delta/2)(\alpha - \beta_1 r(t+\delta/2)),\\
\frac{r(t+\delta)-r(t)}{\delta} &\approx r'(t+\delta/2) = r(t+\delta/2)(\beta_2 b(t+\delta/2) - \gamma),\\
r(t+\delta) &\approx r(t) + \delta r(t+\delta/2)(\beta_2 b(t+\delta/2) - \gamma) \qquad \text{für alle } t \in \mathbb{R}
\end{aligned}$$

und stellen fest, dass wir für die Auswertung der rechten Seite die Werte $b(t+\delta/2)$ und $r(t+\delta/2)$ benötigen, die uns nicht zur Verfügung stehen. Wir können sie allerdings mit einem Schritt des expliziten EULER-Verfahrens mit halber Schrittweite approximieren:

$$\begin{aligned}
b(t+\delta/2) &\approx \tilde{b}(t+\delta/2) := b(t) + \frac{\delta}{2} b(t)(\alpha - \beta r(t)),\\
r(t+\delta/2) &\approx \tilde{r}(t+\delta/2) := r(t) + \frac{\delta}{2} r(t)(\gamma b(t) - \delta) \qquad \text{für alle } t \in \mathbb{R},
\end{aligned}$$

so dass wir insgesamt

$$\begin{aligned}
\tilde{b}(t+\delta/2) &:= b(t) + \frac{\delta}{2} b(t)(\alpha - \beta r(t)),\\
\tilde{r}(t+\delta/2) &:= r(t) + \frac{\delta}{2} r(t)(\gamma b(t) - \delta),\\
\tilde{b}(t+\delta) &:= b(t) + \delta \tilde{b}(t+\delta/2)(\alpha - \beta_1 \tilde{r}(t+\delta/2)),\\
\tilde{r}(t+\delta) &:= r(t) + \delta \tilde{r}(t+\delta/2)(\beta_2 \tilde{b}(t+\delta/2) - \gamma) \qquad \text{für alle } t \in \mathbb{R}
\end{aligned}$$

erhalten. Man kann beweisen, dass dieser Ansatz erheblich genauer als das EULER-Verfahren ist, sich aber immer noch relativ einfach implementieren lässt.

5 Visualisierung

Simulationen führen häufig zu relativ großen Datenmengen, die in der Regel aus einer Ansammlung von Zahlen besteht, die für Menschen nicht unmittelbar verständlich ist.

Um es uns zu ermöglichen, diese Daten sinnvoll zu interpretieren, ist es erforderlich, sie in einer geeigneten Form zu präsentieren. Das ist normalerweise nur möglich, wenn berücksichtigt wird, wie die Daten entstanden sind und in welcher Beziehung sie zu der ursprünglichen Aufgabenstellung stehen, und dieses Wissen müssen wir bei der Darstellung einfließen lassen.

Statt einfach eines der vielen für wissenschaftliche Visualisierung entwickelten Softwarepakete zu verwenden, werden wir uns mit der Frage beschäftigen, wie wir — zumindest für einfache Anwendungen — die entsprechenden Programmteile selbst schreiben können. Dieser Ansatz erlaubt es uns, die grafische Darstellung eng mit unseren Simulationen zu verbinden und beispielsweise auch interaktiv zu beeinflussen.

5.1 OpenGL Legacy

In einem modernen Computer ist in der Regel spezielle Hardware für die Darstellung von Grafik zuständig, entweder in Form einer in den Hauptprozessor integrierten Grafikeinheit oder in Form einer Grafikkarte, die mit dem Prozessor und dem Hauptspeicher verbunden ist. Diese Hardware direkt zu programmieren ist nicht sinnvoll, da unterschiedliche Systemgenerationen unterschiedlicher Hersteller sehr unterschiedliche Zugänge verfolgen und deshalb im schlimmsten Fall jeder neue Computer Anpassungen an unserer Implementierung erforderlich machen würde.

Stattdessen gehen wir wie bereits bei BLAS vor: Die Hersteller der Grafikhardware und deren Anwender einigen sich auf eine standardisierte Schnittstelle (ein API, *application programming interface*), die eine Reihe von Funktionen zur Verfügung stellt. Es liegt in der Verantwortung der Hardwarehersteller, diese Funktionen so gut wie möglich an ihre Systeme anzupassen, während es in der Verantwortung der Anwender liegt, die gewünschten Operationen durch diese Funktionen auszudrücken.

Ein seit Jahren etablierter Standard für die Grafikprogrammierung in zwei und drei Dimensionen ist OpenGL. Wir befassen uns hier zunächst mit dem veralteten Standard OpenGL 2.1, der für viele wissenschaftliche Anwendungen durchaus ausreichend ist, allerdings auf manchen Systemen nicht mehr unterstützt wird.

Um OpenGL verwenden zu können, müssen wir in der Regel eine Headerdatei `gl.h` (oder `OpenGL.h` auf MacOS-Systemen) einbinden, die die nötigen Funktionen definiert. Die definierten Funktionen finden sich in einer Bibliothek, die je nach System beispielsweise `libGL` oder `libOpenGL` heißen mag.

Zu den wichtigsten dieser Funktionen gehören `glBegin` und `glEnd`, mit denen wir eine Folge geometrischer Objekte definieren, und `glVertex2f` oder `glVertex3f`, mit denen wir die Eckpunkte dieser Objekte definieren.

Den Rand des Rechtecks $[-1/2, 1/2] \times [-1/4, 1/4]$ können wir beispielsweise mit

```
glBegin(GL_LINE_LOOP);
glVertex2f(-0.5, -0.25);
glVertex2f(0.5, -0.25);
glVertex2f(0.5, 0.25);
glVertex2f(-0.5, 0.25);
glEnd();
```

zeichnen: Der erste Befehl legt fest, dass wir einen geschlossenen Linienzug zeichnen wollen, die `glVertex2f`-Befehle definieren die Koordinaten der Eckpunkte, und der letzte Befehl beendet den Linienzug.

Bevor wir etwas zeichnen, sollten wir die Zeichenfläche löschen. Dafür stehen uns die Befehle `glClearColor` und `glClear` zur Verfügung. Der erste Befehl erwartet vier Argumente, nämlich die Anteile der Farben rot, grün und blau sowie den sogenannten Alpha-Anteil, den wir für den Moment ignorieren können. Mit

```
glClearColor(0.0, 0.0, 0.0, 0.0);
```

wählen wir schwarz als Hintergrundfarbe. Wenn die Farbe definiert ist, können wir mit

```
glClear(GL_COLOR_BUFFER_BIT);
```

die Zeichenfläche mit der gewählten Hintergrundfarbe füllen. Dabei legt die Konstante `GL_COLOR_BUFFER_BIT` fest, welche Komponenten der Zeichenfläche wir löschen wollen, neben der Farbe gibt es nämlich noch weitere, auf die wir später noch eingehen werden.

Für den in unserem Beispiel gezeichneten Linienzug können wir mit

```
glColor3f(1.0, 1.0, 1.0);
```

die Farbe weiß festlegen. Diese Farbe gilt für alle folgenden Aufrufe der Funktion `glVertex2f`, wir können sie aber zwischen den Aufrufen beliebig verändern, um jedem Eckpunkt des Linienzugs seine eigene Farbe zuzuweisen. Auf den einzelnen Liniensegmenten wird die Farbe dann linear interpoliert (in der Computergrafik nennt man diese Vorgehensweise *Gouraud-Shading*).

Um die vorhandene Hardware möglichst optimal auszunutzen, arbeitet OpenGL grundsätzlich *asynchron*, unsere Zeichenbefehle werden also nicht unbedingt sofort ausgeführt, sie werden lediglich im Grafiksystem gespeichert, bis es sich lohnt, sie tatsächlich zu bearbeiten. Mit der Funktion `glFlush` können wir dafür sorgen, dass alle im Grafiksystem gespeicherten Befehle ausgeführt werden, allerdings ist nicht garantiert, dass dieser Prozess bei der Rückkehr aus der Funktion abgeschlossen ist. Die Funktion `glFinish` dagegen veranlasst ebenfalls die Ausführung aller Befehle, wartet aber, bis tatsächlich alle ausgeführt worden sind.

Insgesamt erhalten wir das folgende Programmfragment, um den Rand des Rechtecks in weiß auf schwarzem Hintergrund zu zeichnen:

```
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_LINE_LOOP);
glVertex2f(-0.5, -0.25);
glVertex2f(0.5, -0.25);
glVertex2f(0.5, 0.25);
glVertex2f(-0.5, 0.25);
glEnd();
glFinish();
```

5.2 GLUT

Allerdings fehlt unserem Programm noch ein sehr wesentlicher Teil: Wir haben noch gar keine Zeichenfläche angelegt, mit der wir etwas anfangen könnten.

Diese Aufgabe hängt nicht nur von der Grafikhardware ab, sondern auch von dem System, das diese Hardware ansteuert, beispielsweise ein grafischer Desktop oder ein Windowmanager. Deshalb ist sie in separate Programmbibliotheken ausgelagert, die sich darum kümmern, Fenster zu öffnen und zu verwalten, in denen dann eine OpenGL-Zeichenfläche dargestellt werden kann.

Eine besonders einfache Schnittstelle bietet dabei FreeGLUT, eine freie Implementierung des *GL Utility Toolkits*. Die von dieser Bibliothek zur Verfügung gestellten Funktionen werden in einer Headerdatei `freeglut.h` definiert, ihre Implementierung findet sich in einer Bibliothek namens `libglut`.

FreeGLUT muss mit der Funktion `glutInit` initialisiert werden, die Zugriff auf Befehlszeilenparameter benötigt, um beispielsweise für die Fensterverwaltung vorgesehene Parameter verarbeiten zu können. Wenn die Bibliothek initialisiert ist, können wir mit der Funktion `glutCreateWindow` ein Fenster anlegen sowie mit den Funktionen `glutPositionWindow` und `glutReshapeWindow` seine Position und Abmessungen festlegen. FreeGLUT verwendet dabei globale Variablen: Eines der erzeugten Fenster ist jeweils aktuell, und Funktionsaufrufe beziehen sich auf dieses Fenster. Mit der Funktion `glutSetWindow` können wir das aktuelle Fenster festlegen. Das folgende Programmfragment erzeugt zwei Fenster:

```
#include <GL/glut.h>
#include <GL/gl.h>

int
main(int argc, char **argv)
{
  int window1, window2;

  glutInit(&argc, argv);

  window1 = glutCreateWindow("Window␣1");
```

```
    window2 = glutCreateWindow("Window␣2");

    glutSetWindow(window1);
    glutPositionWindow(100, 100);
    glutReshapeWindow(200, 200);

    glutSetWindow(window2);
    glutPositionWindow(400, 100);
    glutReshapeWindow(200, 200);

    glutMainLoop();

    return 0;
  }
```

Die Funktion `glutMainLoop` spielt für FreeGLUT eine zentrale Rolle: Wie die meisten grafischen Benutzeroberflächen geht auch FreeGLUT von einem *ereignisgesteuerten* Programmiermodell aus. Es treten *Ereignisse* ein, beispielsweise ein Mausklick oder ein Tastendruck, auf die das System reagieren muss. Die Funktion `glutMainLoop` wartet auf diese Ereignisse und sorgt dafür, dass geeignete Reaktionen erfolgen.

In unserem Beispiel werden die beiden Fenster angezeigt, allerdings sind ihre Inhalte undefiniert, so dass je nach verwendetem Fenstersystem unterschiedliche Effekte auftreten können. Das ist auch nicht verwunderlich, denn wir haben ja noch keine Verbindung zwischen unserer OpenGL-Zeichenfunktion und dem FreeGLUT-Programm hergestellt.

Diese Verbindung wird mit Hilfe der bereits in Bemerkung 3.2 erwähnten *Callback-Funktionen* realisiert: Mit der Funktion `glutDisplayFunc` können wir für das aktuelle Fenster eine Funktion festlegen, die das Zeichnen übernehmen soll. In unserem Beispiel könnte das wie folgt aussehen:

```
static void
display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 1.0, 1.0);
  glBegin(GL_LINE_LOOP);
  glVertex2f(-0.5, -0.25);
  glVertex2f(0.5, -0.25);
  glVertex2f(0.5, 0.25);
  glVertex2f(-0.5, 0.25);
  glEnd();
  glFinish();

  glutSwapBuffers();
}
```

Wir können mit dem Aufruf `glutDisplayFunc(display1)` diese Funktion dem aktuellen Fenster zuweisen[1].

Leider sind wir damit immer noch nicht fertig: In den meisten Benutzeroberflächen können wir die Größe und Abmessungen von Fenstern verändern, beispielsweise indem wir sie mit dem Mauszeiger „anfassen“ und verschieben. Falls sich die Abmessungen des Fensters verändern, wird sich in der Regel auch die Zeichenfläche verändern, so dass OpenGL eine Möglichkeit haben muss, sich anzupassen.

Diesem Zweck dient die Funktion `glReshapeFunc`, mit der wir eine Callback-Funktion festlegen können, die aufgerufen wird, wenn sich die Abmessungen des aktuellen Fensters ändern. Die einfachste Fassung einer solchen Funktion ist die folgende:

```
static void
reshape(int width, int height)
{
  glViewport(0, 0, width, height);
}
```

Die Funktion `glViewport` legt dabei fest, in welchem Teil des Fensters OpenGL etwas zeichnen soll. Die ersten beiden Parameter definierten dabei die x- und y-Koordinaten des linken unteren Eckpunkts der Zeichenfläche, während die nächsten beiden ihre Breite und Höhe angeben. In unserem Fall wollen wir die gesamte Zeichenfläche verwenden.

Mit den Funktionen `display` und `reshape` können wir nun ein vollständiges Programm angeben:

```
int
main(int argc, char **argv)
{
  glutInit(&argc, argv);

  glutCreateWindow("My␣Window");
  glutPositionWindow(100, 100);
  glutReshapeWindow(200, 200);
  glutReshapeFunc(reshape);
  glutDisplayFunc(display);

  glClearColor(0.0, 0.0, 0.0, 0.0);

  glutMainLoop();

  return 0;
}
```

[1] Die Funktion ist als **`static`** definiert, weil sie nicht von außerhalb des aktuellen Moduls aufgerufen werden soll und deshalb ihr Name nicht dem Linker bekannt gegeben werden muss.

Das von der Funktion `display` gezeichnete Rechteck hat offenbar ein Seitenverhältnis von 2 : 1, das von dem von unserem Beispielprogramm angelegten Fenster auch zunächst korrekt wiedergegeben wird.

Sobald wir allerdings die Abmessungen des Fensters ändern, wird sich das Seitenverhältnis ebenfalls ändern, da OpenGL immer das gesamte Quadrat $[-1, 1] \times [-1, 1]$ auf die Zeichenfläche abbildet, unabhängig von deren Abmessungen.

Dieses Problem ließe sich lösen, indem wir den Aufruf der Funktion `glViewport` modifizieren, um dafür zu sorgen, dass die Zeichenfläche immer quadratisch bleibt:

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

Allerdings hat dieser Ansatz den großen Nachteil, dass wir potentiell große Bereiche des Fensters ungenutzt lassen.

Eine elegantere Lösung besteht darin, die Koordinaten zu *skalieren*: Angenommen, das Fenster ist doppelt so breit wie hoch. Dann würde $[-1/2, 1/2] \times [-1, 1]$ als Quadrat dargestellt. Wenn wir also alle x-Koordinaten mit dem Faktor $1/2$, oder allgemeiner `height/width`, skalieren könnten, würden die Seitenverhältnisse korrekt wiedergegeben werden.

Wir können diese Skalierung natürlich per Hand in die Funktion `display` aufnehmen. Sehr viel eleganter ist es, einen in OpenGL für solche Zwecke vorgesehenen Mechanismus zu verwenden: Die von den Funktionen `glVertex2f` oder `glVertex3f` angegebenen Koordinaten werden zu vierdimensionalen Vektoren erweitert (bei `glVertex2f` wird die dritte Koordinate gleich null gesetzt, bei beiden die vierte gleich eins), die dann eine Reihe von Transformationen durchlaufen, bevor sie schließlich verwendet werden:

- Sie werden mit der *modelview*-Matrix multipliziert. Diese Matrix beschreibt beispielsweise, wie die angegebenen Objekte gedreht, verschoben und skaliert werden müssen, um die gewünschte Szene darzustellen.

- Sie werden mit der bereits erwähnten *projection*-Matrix multipliziert, die beispielsweise beschreibt, wie die Szene auf eine Leinwand (oder vielmehr den Bildschirm) übertragen wird.

- Sie werden *normalisiert*, also so skaliert, dass die vierte Koordinate gleich eins ist. Dieser Schritt wird sich im nächsten Abschnitt noch als sehr nützlich erweisen, für den Moment lassen wir die vierte Koordinate konstant gleich eins.

Wir können mit dem Befehl `glMatrixMode` festlegen, auf welche der beiden Matrizen die folgenden Operationen wirken sollen.

Die jeweils gewählte Matrix können wir mit Funktionen wie `glLoadIdentity`, `glLoadMatrix`, `glTranslatef`, `glRotatef` oder `glScalef` modifizieren. Dazu wählen wir zunächst mit `glMatrixMode` die zu modifizierende Matrix aus, in unserem Fall handelt es sich um die *projection*-Matrix, die dafür zuständig ist, die Projektion der Koordinaten in die Zeichenfläche zu beschreiben:

```
glMatrixMode(GL_PROJECTION);
```

Wir setzen die Matrix mit `glLoadIdentity` auf die Identität zurück und verwenden dann die Funktion `glScalef`, um sie *von rechts* mit einer Matrix zu multiplizieren, die eine Skalierung der Koordinaten beschreibt:

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

Alle Koordinaten werden nun je nach Bedarf mit `height/width` oder `width/height` multipliziert. Diese Anpassung des Koordinatensystems lässt sich am besten in der `reshape`-Funktion vornehmen, die wir wie folgt modifizieren:

```
static void
reshape(int width, int height)
{
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  if(width > height)
    glScalef((float) height/width, 1.0, 1.0);
  else
    glScalef(1.0, (float) width/height, 1.0);
}
```

Mit dieser Anpassung bleiben Seitenverhältnisse auch dann noch korrekt, wenn sich die Abmessungen des Fensters verändern, aber OpenGL kann weiterhin das gesamte Fenster verwenden.

5.3 Homogene Koordinaten

Auch wenn OpenGL für zwei- und dreidimensionale Grafik vorgesehen ist, wird intern mit *vierdimensionalen* Vektoren gerechnet. Einerseits sind Zweierpotenzen für die meisten Computer ohnehin passender als andere Zahlen, andererseits lässt sich aber die vierte Koordinate auch verwenden, um einige wichtige Transformationen besonders elegant darzustellen. Während die ersten drei Koordinaten mit x, y und z bezeichnet werden, ist für die vierte Koordinate w eine übliche Bezeichnung. Wenn die vierte Koordinate eines Vektors gleich eins ist, nennen wir ihn *normalisiert*.

Einfache Transformationen wie die bereits verwendete Skalierung lassen sich realisieren, indem wir die ersten drei Koordinaten modifizieren und die vierte unverändert

lassen, beispielsweise in der Form

$$S := \begin{pmatrix} \alpha_1 & & & \\ & \alpha_2 & & \\ & & \alpha_3 & \\ & & & 1 \end{pmatrix},$$

die von `glScalef` verwendet wird.

Eine sehr wichtige Eigenschaft normalisierter Vektoren besteht darin, dass der Nullpunkt nicht durch den Nullvektor dargestellt wird, sondern durch

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Damit ist es uns möglich, Transformationen durch eine lineare Abbildung zu beschreiben, die den Nullpunkt auf einen anderen Vektor abbilden, beispielsweise die Verschiebung des Nullpunkts an eine Position $v \in \mathbb{R}^3$, die durch die Matrix

$$T := \begin{pmatrix} 1 & & & v_1 \\ & 1 & & v_2 \\ & & 1 & v_3 \\ & & & 1 \end{pmatrix}$$

realisiert wird, die bei der Funktion `glTranslatef` von rechts mit der aktuellen Matrix multipliziert wird. Offenbar gilt

$$T \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & & & v_1 \\ & 1 & & v_2 \\ & & 1 & v_3 \\ & & & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 + v_1 \\ x_2 + v_2 \\ x_3 + v_3 \\ 1 \end{pmatrix}.$$

OpenGL interpretiert die vierdimensionalen Vektoren im Sinne *homogener Koordinaten*: Jeder Vektor $a \in \mathbb{R}^4$, dessen vierte Komponente nicht gleich null ist, kann normalisiert werden, indem man ihn durch a_4 dividiert, also die *Normalisierungsabbildung*

$$\mathrm{normalize}\colon \mathbb{R}^3 \times (\mathbb{R} \setminus \{0\}) \to \mathbb{R}^3 \times \{1\}, \qquad a \mapsto a/a_4,$$

anwendet. Zwei Vektoren, die nach der Normalisierung identisch sind, werden von OpenGL auch als identisch angesehen. Es werden also vierdimensionale Vektoren zu Äquivalenzklassen zusammengefasst, die man als *homogene Koordinaten* bezeichnet.

Wie bereits erwähnt multipliziert OpenGL Koordinaten erst mit der *modelview*-Matrix, dann mit der *projection*-Matrix, um sie anschließend zu normalisieren, bevor etwas gezeichnet wird. Diese Vorgehensweise hat zur Folge, dass wir beispielsweise die Skalierung aller Komponenten des Vektors mit dem Faktor 1/2 auch durch die Matrix

$$H := \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 2 \end{pmatrix}$$

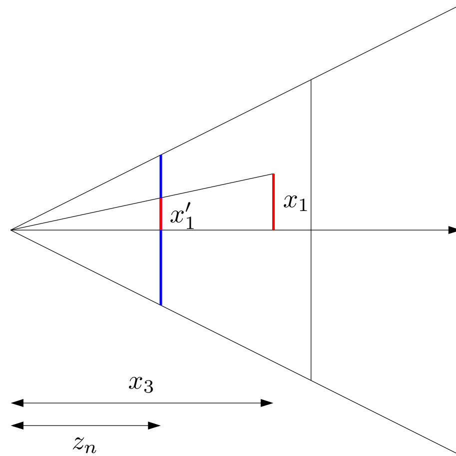

Abbildung 5.1: Anwendung des Strahlensatzes für die Berechnung der perspektivischen Projektion

darstellen können, da sie die w-Koordinate verdoppelt, so dass bei der Normalisierung *alle* Koordinaten durch den doppelten Wert dividiert werden.

Wir können aber auch mit der Matrix

$$P := \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 1 & 0 \end{pmatrix}$$

dafür sorgen, dass ein Vektor mit dem Kehrwert seiner z-Koordinate skaliert wird, denn wir haben

$$P \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_3 \end{pmatrix}, \qquad \text{normalize} \left(P \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} \right) = \begin{pmatrix} x_1/x_3 \\ x_2/x_3 \\ 1 \\ 1 \end{pmatrix}.$$

Diese Eigenschaft können wir ausnutzen, um eine perspektivisch korrekte Darstellung dreidimensionaler Körper zu erreichen: Je weiter sie entfernt sind, je größer also der Abstand zu einer imaginären Kamera ist, desto stärker werden Abstände zwischen Punkten reduziert.

Um die Rechnung zu vereinfachen, gehen wir davon aus, dass die Kamera sich im Nullpunkt befindet und in Richtung der positiven x_3-Achse gerichtet ist. Wir positionieren die imaginäre Leinwand in einer Entfernung von $z_n \in \mathbb{R}_{>0}$ von unserer Kamera und müssen die Position $x' = (x'_1, x'_2, z_n, 1)$ bestimmen, an der ein Punkt $x = (x_1, x_2, x_3, 1) \in \mathbb{R}^4$ auf der Leinwand erscheint.

Nach dem Strahlensatz (siehe Abbildung 5.1) gilt

$$\frac{x'_1}{z_n} = \frac{x_1}{x_3},$$

so dass wir

$$x_1' = \frac{z_n}{x_3} x_1$$

erhalten. Wir wissen bereits, dass OpenGL auf dem Bildschirm grundsätzlich mit Koordinaten aus $[-1,1] \times [-1,1]$ arbeitet. Falls wir eine breitere oder schmalere Leinwand verwenden wollen, sollten wir unsere Gleichung entsprechend modifizieren: Wenn wir die gewünschte Breite mit $w \in \mathbb{R}_{>0}$ bezeichnen, gelangen wir zu

$$x_1' = \frac{2z_n}{wx_3} x_1.$$

Entsprechend erhalten wir für die zweite Koordinate

$$x_2' = \frac{2z_n}{hx_3} x_2,$$

wobei $h \in \mathbb{R}_{>0}$ die Höhe der Leinwand bezeichnet. Im Prinzip wären wir nun fertig und könnten die Matrix

$$\begin{pmatrix} \frac{2z_n}{w} & & & \\ & \frac{2z_n}{h} & & \\ & & 1 & \\ & & 1 & 0 \end{pmatrix}$$

verwenden, um eine perspektivisch korrekte Projektion durchzuführen.

Dabei ergibt sich allerdings eine Schwierigkeit: Bei einer dreidimensionalen Szene kann es durchaus vorkommen, dass ein Objekt ein anderes ganz oder teilweise verdeckt. Da unsere Matrix dafür sorgt, dass alle Punkte auf die Leinwand projiziert werden, geht die Information über ihren Abstand zu unserer Kamera verloren, so dass wir nicht mehr entscheiden können, ob ein Punkt von einem anderen verdeckt wird.

Dieses Problem lässt sich lösen, indem wir die dritte Koordinate so berechnen, dass in ihre Informationen über den Abstand erhalten bleiben. Da OpenGL auch von dieser dritten Koordinate erwartet, dass sie in dem Intervall $[-1,1]$ liegt und standardmäßig alle Punkte jenseits dieses Intervalls ignoriert, legen wir neben $z_n \in \mathbb{R}_{>0}$ auch einen Abstand $z_f \in \mathbb{R}_{>0}$ mit $z_f > z_n$ fest, in dem unser sichtbarer Bereich endet.

Da bereits fest steht, dass wir durch x_3 dividieren werden, suchen wir eine monotone Abbildung der Form

$$\varphi\colon \mathbb{R} \setminus \{0\} \to \mathbb{R}, \qquad x_3 \to \frac{\alpha x_3 + \beta}{x_3},$$

mit

$$\varphi(z_n) = -1, \qquad \varphi(z_f) = 1.$$

Da die Abbildung monoton ist, ist sicher gestellt, dass näher an der Kamera liegende Punkte auch nach der Transformation noch kleinere Koordinaten aufweisen. Aus den beiden Gleichungen ergibt sich

$$\frac{\alpha z_n + \beta}{z_n} = \varphi(z_n) = -1, \qquad \frac{\alpha z_f + \beta}{z_f} = \varphi(z_f) = 1,$$

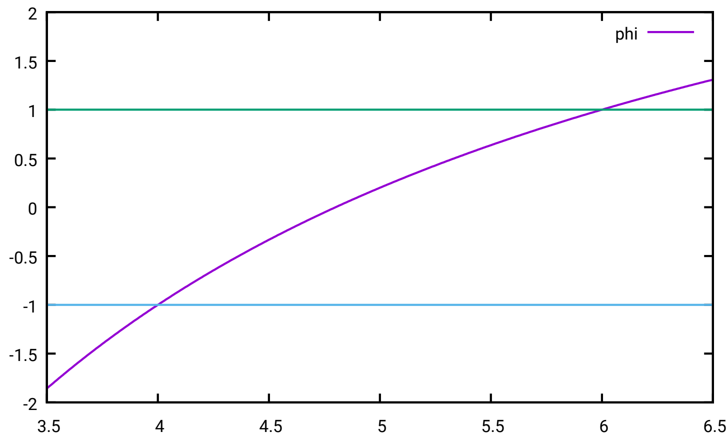

Abbildung 5.2: Monotone Transformation der z-Koordinaten für den Fall $z_n = 4$, $z_f = 6$

$$\alpha z_n + \beta = -z_n, \qquad \alpha z_f + \beta = z_f,$$

und indem wir die erste Gleichung von der zweiten subtrahieren, folgt

$$\alpha(z_f - z_n) = z_f + z_n, \qquad \alpha = \frac{z_f + z_n}{z_f - z_n}.$$

Diesen Wert für α können wir in die zweite Gleichung einsetzen, um zu

$$\frac{z_f + z_n}{z_f - z_n} z_f + \beta = z_f,$$
$$\beta = z_f \left(1 - \frac{z_f + z_n}{z_f - z_n}\right) = z_f \frac{z_f - z_n - (z_f + z_n)}{z_f - z_n} = \frac{-2 z_n z_f}{z_f - z_n}$$

zu gelangen. Die resultierende Funktion φ ist in Abbildung 5.2 dargestellt.

Unsere verbesserte Matrix nimmt damit die Gestalt

$$\begin{pmatrix} \frac{2z_n}{w} & & & \\ & \frac{2z_n}{h} & & \\ & & \frac{z_f+z_n}{z_f-z_n} & \frac{-2z_n z_f}{z_f-z_n} \\ & & 1 & 0 \end{pmatrix}$$

an. Traditionell verwendet OpenGL ein *rechtshändiges Koordinatensystem*: Wir halten die rechte Hand so, dass Daumen, Zeige- und Mittelfinger in zueinander rechtwinklige Richtungen zeigen (siehe Abbildung 5.3). Wir wählen unser Koordinatensystem so, dass die erste Koordinate in Richtung des Daumens wächst, die zweite in Richtung des Zeigefingers, und die dritte in Richtung des Mittelfingers. Falls wir uns an diese Konvention halten wollen, müsste die dritte Koordinate *abnehmen*, wenn wir uns von der Kamera entfernen. Das lässt sich einfach erreichen, indem wir das Vorzeichen der dritten Spalte

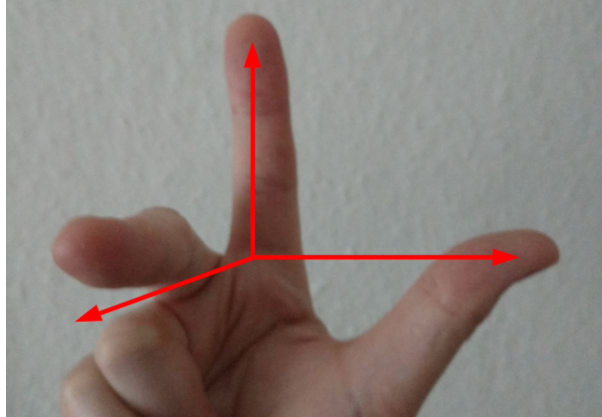

Abbildung 5.3: Rechte-Hand-Regel: Der Daumen zeigt in positive Richtung der ersten Koordinaten, der Zeigefinger in die der zweiten, und der Mittelfinger in die der dritten.

unserer Matrix umdrehen. Damit erhalten wir die endgültige Matrix der perspektivischen Projektion:

$$P := \begin{pmatrix} \frac{2z_n}{w} & & & \\ & \frac{2z_n}{h} & & \\ & & -\frac{z_f+z_n}{z_f-z_n} & -\frac{2z_n z_f}{z_f-z_n} \\ & & -1 & 0 \end{pmatrix},$$

die wir beispielsweise mit der Funktion `glLoadMatrix` dem OpenGL-System für die folgenden `glVertex`-Anweisungen übermitteln können. Alternativ gibt es die Funktion `glFrustum`, die eine Matrix für eine noch etwas allgemeinere Projektion aufstellt.

5.4 Dreiecke

Wenn wir statt einfacher Drahtgittermodelle realistischere Körper darstellen wollen, ersetzen wir Linienzüge durch Dreiecke:

```
glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0.5);
glVertex3f( 0.5,  0.5, 0.5);
glVertex3f(-0.5,  0.5, 0.5);

glVertex3f(-0.5, -0.5, 0.5);
glVertex3f( 0.5, -0.5, 0.5);
glVertex3f( 0.5,  0.5, 0.5);
glEnd();
```

In der Standardeinstellung werden die Dreiecke gefüllt dargestellt. Falls unterschiedliche Farben in den drei Eckpunkten gewählt wurden, wird zwischen ihnen linear interpoliert. Diese Vorgehenweise nennt man GOURAUD-*Shading*[2], sie bietet den Vorteil, dass häufig aufwendigere Berechnungen nur für jeden Eckpunkt statt für jeden Bildpunkt ausgeführt werden müssen und trotzdem ein überzeugender Gesamteindruck entsteht.

In OpenGL hat jedes Dreieck eine Vorder- und eine Rückseite, die durch die Reihenfolge der Eckpunkte festgelegt wird: Die Seite, auf der die Eckpunkte aus Sicht der Kamera entgegen dem Uhrzeigersinn durchlaufen werden, also im mathematisch positiven Sinn, ist die Vorderseite, die andere die Rückseite. Wir können OpenGL dazu auffordern, alle Dreiecke zu ignorieren, die uns ihre Rückseite zeigen, indem wir das sogenannte *back-face culling* einschalten:

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Bei konvexen Körpern (Kugeln, Würfeln, Zylindern, Pyramiden, ...) sind die so eliminierten Dreiecke gerade diejenigen, die die Rückseite des Körpers beschreiben, so dass sie ohnehin verdeckt sind und auch gar nicht sichtbar sein sollen.

Bei nicht-konvexen Körpern und bei Szenen, die sich aus mehreren Körpern zusammensetzen, ist die Situation etwas schwieriger: Die Vorderseite eines weiter von der Kamera entfernten Dreiecks kann durch ein näher an der Kamera liegendes Dreieck verdeckt werden. Neben der Unterscheidung zwischen Vorder- und Rückseite muss also auch noch der Abstand zur Kamera berücksichtigt werden.

Diese Aufgabe löst OpenGL mit einem Hilfsspeicher, dem *depth buffer*, der für jeden Bildpunkt speichert, wie weit er von der Kamera entfernt ist. Wenn OpenGL ein Dreieck zeichnet, erzeugt es für jeden im Dreieck liegenden Bildpunkt ein *Fragment*, also einen Kandidaten für einen Eintrag im Bildspeicher. Dieses Fragment enthält neben der x- und der y-Koordinate auch eine z-Koordinate, die beschreibt, wie weit der Kandidat von der Kamera entfernt ist. Wenn ein Fragment in den Bildspeicher aufgenommen wird, wird seine z-Koordinate im *depth buffer* hinterlegt.

Falls ein neues Fragment dieselbe x- und y-Koordinate aufweist, können wir durch einen Vergleich seiner z-Koordinate mit der im *depth buffer* gespeicherten feststellen, ob es näher an der Kamera liegt. Nur dann wird es gezeichnet, ansonsten wird es verworfen.

In der Standardeinstellung liegen die z-Koordinaten, wie die x- und y-Koordinaten, im Intervall $[-1, 1]$ und sind um so größer, je weiter das Fragment von der Kamera entfernt ist.

Dieser Ansatz hat den Vorteil, dass es völlig egal ist, in welcher Reihenfolge wir die Dreiecke zeichnen. Allerdings müssen wir den *depth buffer* korrekt initialisieren und auch dafür sorgen, dass er tatsächlich verwendet wird. Für die Initialisierung können wir den Aufruf der Funktion `glClear` erweitern:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Damit näher an der Kamera liegende Bildpunkte nicht von weiter entfernten überschrieben werden, müssen wir mit

[2] Benannt nach Henri Gouraud.

```
glEnable(GL_DEPTH_TEST);
```

dafür sorgen, dass die z-Koordinaten der Fragmente mit den Werten im *depth buffer* verglichen werden. Falls die neue z-Koordinate größer als die alte ist, wird das Fragment verworfen, ansonsten wird es in den Bildspeicher aufgenommen und der *depth buffer* wird aktualisiert.

Aus diesem Grund haben wir im vorigen Abschnitt die Funktion φ so konstruiert, dass sie die z-Koordinaten monoton abbildet, um die für die Verdeckung benötigten Informationen zu erhalten.

5.5 Beleuchtung

Für einfache Liniengrafiken genügen die bisher besprochenen Techniken. Falls wir allerdings geometrische Körper etwas realistischer darstellen wollen, müssen wir uns mit der Frage beschäftigen, wie in der Realität die von unseren Augen wahrgenommene Farbe zustande kommt: Gegenstände werden von einer Lichtquelle angestrahlt, und das reflektierte Licht wird von unseren Augen registriert.

Im Interesse einer möglichst geringen Rechenzeit ist es in der Regel nicht möglich, die dahinter stehenden physikalischen Vorgänge exakt nachzubilden, stattdessen werden allerhand Tricks eingesetzt, um einen mehr oder weniger realistischen Eindruck zu erreichen, ohne zu lange Wartezeiten in Kauf nehmen zu müssen.

Der ursprüngliche OpenGL-Standard unterstützt dabei das PHONG-*Modell*[3], bei dem sich die Farbe aus drei Anteilen zusammensetzt:

- Umgebungslicht (engl. *ambient lighting*), das von allen Seiten gleichmäßig den Gegenstand erreicht,
- diffus reflektiertes Licht (engl. *diffuse lighting*), bei dem von einer Lichtquelle eintreffendes Licht in Abhängigkeit von dem Winkel reflektiert wird, unter dem es eine Oberfläche erreicht, und
- gespiegeltes Licht (engl. *specular lighting*), bei dem das von der Lichtquelle eintreffende Licht an der Oberfläche in Richtung des Betrachters gespiegelt wird.

Um dieses Modell einzusetzen, müssen wir zunächst mit

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

dem OpenGL-System mitteilen, dass die Farben der gezeichneten Objekte nicht mehr direkt per `glColor` vorgegeben werden, sondern aus dem Beleuchtungsmodell stammen, und dass bei den Berechnungen die erste Lichtquelle (es sind mehrere möglich) aktiv sein soll.

Eigenschaften der Lichtquellen werden mit Varianten des Befehls `glLight` festgelegt, beispielsweise mit

[3] Benannt nach Bui Thong Phong.

```
float light_position[4] = { 5.0, 5.0, 5.0, 1.0 };
float light_ambient[4] = { 0.1, 0.1, 0.1, 1.0 };
float light_diffuse[4] = { 1.0, 1.0, 1.0, 1.0 };

glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
```

eine Lichtquelle an der Position $(5, 5, -5)$, die ein relativ dunkles Umgebungslicht mit einem Zehntel der Intensität in Rot-, Grün- und Blauanteil beiträgt und bei diffuser Reflektion als weiß mit voller Intensität wirkt.

Wenn das Licht auf ein Objekt trifft, soll es mit dem Material interagieren, aus dem das Objekt besteht. Dieses Material wird mit Varianten des Befehls `glMaterial` beschrieben, so legen beispielsweise die Zeilen

```
float material_ambient[4] = { 1.0, 0.0, 0.0, 0.0 };
float material_diffuse[4] = { 1.0, 0.0, 0.0, 0.0 };

glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,
             material_ambient);
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE,
             material_diffuse);
```

fest, dass sowohl bei Umgebungs- als auch bei diffusem Licht das Material rot erscheinen soll. Dabei besagt `GL_FRONT_AND_BACK`, dass diese Einstellungen sowohl für die Vorder- als auch die Rückseite der gezeichneten Oberflächen gelten sollen.

Sobald das Beleuchtungsmodell eingeschaltet ist, spielt die Funktion `glColor` eigentlich keine Rolle mehr, da die Farben nun aus dem Modell stammen. Allerdings lässt sich die Funktion „recyclen“, wenn wir den *color-material mode* einschalten, beispielsweise durch die Zeilen

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
```

In dieser Einstellung legt ein Aufruf der Funktion `glColor` die Farbe des Materials für die Beleuchtungsberechnung fest, statt direkt die Farbe, in der das Objekt gezeichnet wird, und wir können Änderungen der Farbe sehr viel kürzer formulieren.

Für die Berechnung der diffusen und reflektiven Beleuchtung ist es von entscheidender Bedeutung, dass jedem Eckpunkt eines Dreiecks ein *äußerer Einheitsnormalenvektor* zugeordnet ist, also ein Vektor, der senkrecht auf der Oberfläche steht, der aus dem dargestellten Körper „heraus“ zeigt und der die Länge eins aufweist. Mit Hilfe dieses Vektors kann ermittelt werden, unter welchem Winkel das Licht auf die Oberfläche trifft und wie es gegebenenfalls reflektiert wird.

Im Prinzip können wir für jedes Dreieck einen solchen Normalvektor berechnen: Wenn das Dreieck aus den Punkten $a, b, c \in \mathbb{R}^3$ besteht, können wir die Kantenvektoren $p =$

$b - a$ und $q = c - a$ einführen und ihr *Kreuzprodukt*

$$\widetilde{n} := p \times q = \begin{pmatrix} p_2 q_3 - p_3 q_2 \\ p_3 q_1 - p_1 q_3 \\ p_1 q_2 - p_2 q_1 \end{pmatrix}$$

berechnen. Der Vektor $\widetilde{n}$ steht auf p und q senkrecht, also auch auf dem Dreieck. Durch Division durch seine euklidische Norm erhalten wir

$$n := \widetilde{n}/\|\widetilde{n}\|_2 = \frac{1}{\sqrt{\widetilde{n}_1^2 + \widetilde{n}_2^2 + \widetilde{n}_3^2}} \widetilde{n},$$

den gesuchten äußeren Einheitsnormalenvektor.

OpenGL führt diese Berechnung allerdings nicht automatisch für uns durch, sondern erwartet, dass wir für jeden Eckpunkt mit dem Befehl `glNormal3f` explizit einen Normalvektor angeben. Die Ursache ist, dass sich durch geschickte Modifikation des Normalenvektors der Eindruck erweckt werden kann, die einzelnen Dreiecke seien leicht gebogen, so dass glatte Oberflächen sich realistischer ausleuchten lassen.

Für `glNormal3f` gilt dasselbe wie für `glColor3f`: Der festgelegte Wert wird erst übernommen, wenn ein Eckpunkt mit `glVertex2f` oder `glVertex3f` erzeugt wird. Es steht uns also frei, für jeden Eckpunkt einen separaten Normalenvektor vorzugeben oder denselben für mehrere Eckpunkte zu verwenden.

5.6 OpenGL Core Profile

Bisher stand der alte OpenGL-Standard im Mittelpunkt unserer Betrachtungen, der sich aufgrund seines beschränkten Funktionsumfang etwas besser erklären lässt. Der moderne OpenGL-Standard ist allerdings erheblich flexibler, so dass nun auf die wesentlichen Unterschiede eingegangen werden soll.

Modernes OpenGL sieht vor, dass viele Schritte der Verarbeitung von Eckpunkten, Dreiecken und Fragmenten frei programmiert werden können. Dazu wurde die *OpenGL Shading Language (GLSL)* eingeführt, ein Dialekt der Sprache C, der für die Ausführung auf einer frei programmierbaren Grafikkarte angepasst ist. Beispielsweise kennen GLSL-Programme keine Dateioperationen, weil es auf der Grafikkarte kein Dateisystem gibt. Andererseits unterstützen GLSL-Programme unmittelbar Operationen mit vierdimensionalen Vektoren und 4×4-Matrizen sowie eine weitaus größere Zahl an fest eingebauten Funktionen als C.

Ein Programm, das mit dem *OpenGL Core Profile* kompatibel ist, muss mindestens einen *vertex shader* definieren, der sämtliche Koordinatentransformationen durchführt (also beispielsweise die Multiplikation mit den *modelview*- und *projection*-Matrizen übernehmen kann), und einen *fragment shader*, der die Farbe eines Fragments festlegt.

Ein *vertex shader* erhält jeweils die Daten eines Eckpunkts eines Dreiecks und muss mindestens die Koordinaten berechnen, an denen er auf dem Bildschirm dargestellt werden soll. Wenn wir die Funktionsweise des traditionellen OpenGL-Standards nachahmen wollen, können wir die Matrizen als sogenannte *uniforme* Variablen definieren und die Position als Eingabevektor, um zu dem folgenden Programm zu gelangen:

```
#version 330 core

in vec4 vPosition;
uniform mat4 mModelview;
uniform mat4 mProjection;

void
main()
{
  gl_Position = mProjection * mModelview * vPosition;
}
```

Der Typ `vec4` beschreibt dabei einen vierdimensionalen Vektor, der Typ `mat4` eine 4×4-Matrix.

Die Variable `gl_Position` ist von GLSL vordefiniert und muss mit der Position des Fragments gefüllt werden. In unserem Fall geschieht das durch zwei Matrix-Vektor-Multiplikationen mit `mModelview` und `mProjection`.

Nachdem die Eckpunkte eines Dreiecks transformiert worden sind, wird das Dreieck in Fragmente zerlegt, die dem *fragment shader* übergeben werden, der beispielsweise die Farbe des Bildpunkts definiert. Im einfachsten Fall könnte das wie folgt aussehen:

```
#version 330 core

out vec4 vColor;

void
main()
{
  vColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Wir setzen schlicht weiß als konstante Farbe. Anders als bei einem *vertex shader* gibt es bei einem *fragment shader* keine vordefinierten globalen Variablen, in die wir die Farbe eintragen müssen, sondern die erste mit dem Schlüsselwort `out` deklarierte Variable nimmt die Farbe auf.

Ein GLSL-Programm besteht aus den verwendeten *shader*-Programmen, die übersetzt und gebunden werden müssen:

```
GLuint vertex_shader, fragment_shader, shader_program;

vertex_shader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1, vertex_source, 0);
glCompileShader(vertex_shader);

fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment_shader, 1, fragment_source, 0);
```

```
glCompileShader(fragment_shader);

shader_program = glCreateProgram();
glAttachShader(shader_program, vertex_shader);
glAttachShader(shader_program, fragment_shader);
glLinkProgram(shader_program);

glUseProgram(shader_program);
```

Hier wird vorausgesetzt, dass `vertex_source` und `fragment_source` C-Strings sind, die die beiden oben angegebenen Quelltexte enthalten.

Die im alten OpenGL-Standard verwendeten Funktionen `glBegin` und `glEnd`, mit denen Punkte, Kantenzüge oder Dreiecke definiert werden, werden in modernem OpenGL durch eine auf Arrays basierende Variante ersetzt, bei der die Hoffnung besteht, sie erheblich effizienter implementieren zu können: Daten von Eckpunkten werden in einem *vertex array* gespeichert, und jedes Array wird intern durch einen Namen, einen Wert des Typs `GLuint`, gekennzeichnet. Mit der Funktion `glGenVertexArrays` können wir uns ungenutzte Namen zur Verfügung stellen lassen. Im nächsten Schritt legen wir fest, dass alle folgenden Operationen ein bestimmtes *vertex array* verwenden sollen, indem wir `glBindVertexArray` aufrufen.

Nun benötigen wir Speicher, der die Daten der Eckpunkte aufnehmen kann. Auch solche Pufferspeicher werden in OpenGL durch einen Namen in Form einer `GLuint`-Zahl beschrieben, und wir können uns mit `glGenBuffers` ungenutzte Namen geben lassen. Mit der Funktion `glBufferData` können wir einen Pufferspeicher dann initialisieren und mit Daten füllen:

```
GLfloat vertex_coordinates = { -0.5, -0.5, 0.5,
                                0.5,  0.5, 0.5,
                               -0.5,  0.5, 0.5,
                               -0.5, -0.5, 0.5,
                                0.5, -0.5, 0.5,
                                0.5,  0.5, 0.5 };
GLuint vertex_array;
GLuint vertex_buffer;

glGenVertexArrays(1, &vertex_array);
glBindVertexArray(vertex_array);

glGenBuffers(1, &vertex_buffer);
glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertex_coordinates),
             vertex_coordinates, GL_STATIC_DRAW);
```

Nun müssen wir eine Verbindung zwischen dem *vertex array* und dem GLSL-Programm herstellen: Jedes Element der Arrays gehört zu einem Eckpunkt und kann mehrere Attribute wie Koordinaten oder Farbinformationen aufnehmen. In unserem Fall gibt es nur ein

Attribut namens `vPosition`, und wir können mit `glEnableVertexAttribArray` dem *vertex array* mitteilen, dass es dieses Attribut an das GLSL-Programm übermitteln soll. Da der Pufferspeicher noch völlig unstrukturiert ist, müssen wir auch definieren, wie seine Daten interpretiert werden sollen. Diesem Zweck dient die Funktion `glVertexAttribPointer`, mit der wir für jedes Attribut definieren, wo im Pufferspeicher die Daten zu den einzelnen Eckpunkten zu finden sind: Das Attribut des i-ten Eckpunkts findet sich an der Position `pointer + i*stride` im Array und wird durch `size` Elemente des Typs `type` dargestellt. In unserem Fall können wir `pointer=0`, `stride=3*sizeof(GLfloat)` und `type=GL_FLOAT` verwenden:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
                      3*sizeof(GLfloat), 0);
```

Nun sind alle Eckpunktdaten festgelegt und wir können mit

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

die sechs Eckpunkte verarbeiten und die beiden Dreiecke zeichnen lassen. Sobald wir also das *vertex array* und den Pufferspeicher angelegt haben, fällt der eigentliche Zeichenvorgang relativ einfach aus.

6 Approximation von Funktionen

Sehr häufig trifft man in Anwendungen auf Fragestellungen, die von einem Parameter oder mehreren abhängen. Beispielsweise benötigen wir einen Algorithmus, der die Wurzel einer *beliebigen* positiven Zahl berechnet.

Mathematisch können solche Situationen durch *Funktionen* beschrieben werden, die im einfachsten Fall jedem Parameter $x \in [a, b]$ aus einem Intervall eine reelle Zahl zuordnen:

$$f\colon [a, b] \to \mathbb{R}$$

Unsere Aufgabe besteht also darin, derartige Funktionen möglichst genau und möglichst effizient zu approximieren.

Wenn wir diese Aufgabe gelöst haben, können wir nicht nur unter anderem die Wurzel, den Logarithmus, die Exponentialfunktion oder den Sinus effizient berechnen, wir können sogar Aufgabenstellungen betrachten, bei denen Funktionen als *Unbekannte* auftreten, beispielsweise bestimmte Differentialgleichungen, die strukturmechanische Phänomene oder elektromagnetische Felder beschreiben.

6.1 Taylor-Entwicklung

Einen ersten Ansatz für die Approximation von Funktionen haben wir bereits kennen gelernt: Wenn wir uns die in Satz 3.8 eingeführte Taylor-Entwicklung etwas genauer anschauen, stellen wir fest, dass sie eine hinreichend oft differenzierbare Funktion durch ein Polynom approximiert. Sei $m \in \mathbb{N}_0$, und sei $f \in C^{m+1}[a, b]$ eine beliebige Funktion. Wenn wir einen Entwicklungspunkt $x_0 \in [a, b]$ wählen, erhalten wir mit dem Satz 3.8 für ein beliebiges $y \in [a, b]$ die Darstellung

$$f(y) = \sum_{k=0}^{m} f^{(k)}(x_0)\frac{(y - x_0)^k}{k!} + f^{(m+1)}(\eta)\frac{(y - x_0)^{m+1}}{(m + 1)!}$$

mit einem $\eta \in [a, b]$. Da die Funktionen

$$y \mapsto \frac{(y - x_0)^k}{k!}$$

Polynome k-ten Grades sind, ist ihre Summe

$$p(y) = \sum_{k=0}^{m} f^{(k)}(x_0)\frac{(y - x_0)^k}{k!}$$

ein Polynom höchstens m-ten Grades und die TAYLOR-Formel nimmt die Gestalt

$$f(y) - p(y) = f^{(m+1)}(\eta)\frac{(y - x_0)^{m+1}}{(m+1)!}$$

an. Die rechte Seite dieser Gleichung beschreibt den Fehler, den wir uns einhandeln, wenn wir f durch das Polynom p approximieren.

Da wir η nicht kennen, empfiehlt es sich, die Gleichung durch eine Abschätzung zu ersetzen. Dazu definieren wir die *Maximumnorm*

$$\|g\|_{\infty,[a,b]} := \max\{|g(x)| \ : \ x \in [a,b]\} \qquad \text{für alle } g \in C[a,b],$$

berechnen

$$|y - x_0| = \begin{cases} y - x_0 \leq b - a & \text{falls } y \geq x_0, \\ x_0 - y \leq b - a & \text{ansonsten,} \end{cases}$$

und erhalten

$$|f(y) - p(y)| = |f^{(m+1)}(\eta)|\frac{|y - x_0|^{m+1}}{(m+1)!} \tag{6.1}$$

$$\leq \|f^{(m+1)}\|_{\infty,[a,b]}\frac{(b-a)^{m+1}}{(m+1!)} \qquad \text{für alle } y \in [a,b]. \tag{6.2}$$

Diese Abschätzung lässt sich kompakt als

$$\|f - p\|_{\infty,[a,b]} \leq (b-a)^{m+1}\frac{\|f^{(m+1)}\|_{\infty,[a,b]}}{(m+1)!}$$

schreiben.

Falls wir den Mittelpunkt $x_0 = (b+a)/2$ des Intervalls als Entwicklungspunkt wählen, erhalten wir mit

$$|y - x_0| = \begin{cases} y - x_0 = y - \frac{b+a}{2} \leq b - \frac{b+a}{2} = \frac{b-a}{2} & \text{falls } y \geq x_0, \\ x_0 - y = \frac{b+a}{2} - y \leq \frac{b+a}{2} - a = \frac{b-a}{2} & \text{ansonsten} \end{cases}$$

die verbesserte Abschätzung

$$\|f - p\|_{\infty,[a,b]} \leq \left(\frac{b-a}{2}\right)^{m+1}\frac{\|f^{(m+1)}\|_{\infty,[a,b]}}{(m+1)!}.$$

Selbstverständlich ist die Approximation der Funktion f durch das Polynom p nur hilfreich, wenn wir dieses Polynom auch effizient auswerten können.

Wenn wir davon ausgehen, dass wir eine Funktion f berechnen wollen, deren Ableitungen uns im Vorfeld bekannt sind, können wir die Koeffizienten

$$a_k := \frac{f^{(k)}(x_0)}{k!} \qquad \text{für alle } k \in [0:m]$$

als Konstanten in unseren Algorithmus aufnehmen und müssen nur noch

$$p(y) = \sum_{k=0}^{m} a_k (y - x_0)^k \qquad \text{für alle } y \in [a, b] \tag{6.3}$$

auswerten. Diese Aufgabe lässt sich besonders elegant lösen, indem wir ausnutzen, dass in allen Summanden Potenzen der Zahl $y - x_0$ auftreten: Wir haben

$$p(y) = \sum_{k=0}^{m} a_k (y - x_0)^k = a_0 + \sum_{k=1}^{m} a_k (y - x_0)^k = a_0 + (y - x_0) \sum_{k=1}^{m} a_k (y - x_0)^{k-1}$$

und stellen mit der Substitution $j = k - 1$ fest, dass

$$\sum_{k=1}^{m} a_k (y - x_0)^{k-1} = \sum_{j=0}^{m-1} a_{j+1} (y - x_0)^j$$

nur noch ein Polynom $(m-1)$-ten Grades ist. Wir können diese Operationen wiederholen, bis wir ein Polynom nullten Grades erhalten, das sich trivial auswerten lässt.

Wenn wir die Zwischenergebnisse

$$p_\ell(y) := \sum_{k=m-\ell}^{m} a_\ell (y - x_0)^{k-m+\ell} \qquad \text{für alle } \ell \in [0 : m],\ y \in [a, b]$$

definieren, stellen wir fest, dass wir für $\ell \in [0 : m - 1]$ gerade

$$\begin{aligned} p_{\ell+1}(y) &= \sum_{k=m-\ell-1}^{m} a_\ell (y - x_0)^{k-m+\ell+1} \\ &= a_{m-\ell-1} + (y - x_0) \sum_{k=m-\ell}^{m} a_\ell (y - x_0)^{k-m+\ell} \\ &= a_{m-\ell-1} + (y - x_0) p_\ell(y) \qquad \text{für alle } y \in [a, b] \end{aligned}$$

erhalten. Offenbar gelten auch

$$p_0 = a_m, \qquad p_m = p,$$

so dass wir das Polynom p mit dem in Abbildung 6.1 gegebenen Verfahren auswerten können. Dieser als HORNER-*Schema*[1] bekannte Algorithmus benötigt lediglich $2m + 1$ Gleitkommaoperationen (nämlich m Multiplikationen, m Additionen und eine Subtraktion für die Berechnung der Hilfsvariablen $z = y - x_0$), um das in der Form (6.3) gegebene Polynom auszuwerten.

Die Kombination aus einer TAYLOR-Entwicklung und dem HORNER-Schema lässt sich flexibel unseren Bedürfnissen anpassen. Als Beispiel untersuchen wir die Approximation der Sinusfunktion

$$f\colon [-\pi, \pi] \to \mathbb{R}, \qquad x \mapsto \sin(x).$$

[1] Benannt nach William George Horner.

```
function horner(a, x_0, y);
z ← y − x_0;   p ← a_m;
for ℓ = 1 to m do
  p ← a_{m−ℓ} + z p;
return p
end
```

Abbildung 6.1: Horner-Schema

Es ist bekannt, dass ihre Ableitungen durch

$$f'(x) = \cos(x), \qquad f''(x) = -\sin(x),$$
$$f'''(x) = -\cos(x), \qquad f''''(x) = \sin(x) = f(x) \qquad \text{für alle } x \in \mathbb{R}$$

gegeben sind. Wenn wir nun als Entwicklungspunkt den Mittelpunkt unseres Intervalls, also den Nullpunkt $x_0 = 0$, verwenden, erhalten wir

$$f^{(2k)}(0) = (-1)^k \sin(0) = 0,$$
$$f^{(2k+1)}(0) = (-1)^k \cos(0) = (-1)^k \qquad \text{für alle } k \in \mathbb{N}_0,$$

so dass das Taylor-Polynom die besonders einfache Gestalt

$$p(y) = \sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} y^{2k+1} \qquad \text{für alle } y \in [-\pi, \pi]$$

annimmt, wobei die Konstante $n = \lfloor (m-1)/2 \rfloor$ über die Anzahl der Terme entscheidet. Indem wir $z = y^2$ einführen und

$$y^{2k+1} = y(y^2)^k = yz^k$$

ausnutzen, können wir die Auswertung des Polynoms noch etwas effizienter gestalten, denn wir haben

$$p(y) = y \sum_{k=0}^{n} \frac{(-1)^k}{(2k+1)!} z^k \qquad \text{für alle } y \in [-\pi, \pi],$$

so dass wir analog zu dem gewöhnlichen Horner-Schema vorgehen können, aber $3n+1$ statt $3m$ Operationen benötigen.

Da in unserem Fall $\|f^{(m+1)}\|_{\infty,[-\pi,\pi]} = 1$ gilt, erhalten wir die Schranke

$$\|f - p\|_{\infty,[-\pi,\pi]} \leq \frac{\pi^{2n+3}}{(2n+3)!} \qquad \text{für alle } n \in \mathbb{N}_0$$

für die Genauigkeit, an der wir ablesen können, dass schon eine relativ geringe Anzahl von Termen ausreicht, um den Fehler auf ein akzeptables Maß zu reduzieren. Beispielsweise erreichen wir mit $n = 8$ eine Fehlerschranke von $\|f - p\|_{\infty,[-\pi,\pi]} \leq 3 \times 10^{-8}$ und mit $n = 13$ gelangen wir zu $\|f - p\|_{\infty,[-\pi,\pi]} \leq 3 \times 10^{-17}$.

6.2 Interpolation

Die TAYLOR-Entwicklung beruht auf den Ableitungen der zu approximierenden Funktion, und bei komplizierteren Funktionen kann es sehr aufwendig (und lästig) werden, diese Ableitungen per Hand zu berechnen. Statt alle Ableitungen in einem einzigen Entwicklungspunkt x_0 auszuwerten, können wir auch *die Funktion selbst* in *mehreren* Punkten $x_0, \ldots, x_m \in [a, b]$ auswerten und versuchen, aus diesen Werten eine Approximation zu konstruieren.

Das Polynom wäre dann durch

$$p(x_i) = f(x_i) \qquad \text{für alle } i \in [0 : m] \tag{6.4}$$

definiert. Ein Polynom, das diese Gleichungen erfüllt, bezeichnen wir als ein LAGRANGE-*Interpolationspolynom*[2].

Es stellt sich die Frage, ob ein solches Polynom überhaupt existiert und, falls es existiert, ob es durch die Bedingungen (6.4) eindeutig festgelegt wird.

Für die Untersuchung dieser Frage setzen wir voraus, dass die *Interpolationspunkte* $x_0, \ldots, x_m \in [a, b]$ paarweise verschieden sind, dass also für $i, j \in [0 : m]$ aus $i \neq j$ auch $x_i \neq x_j$ folgt. Unter dieser Voraussetzung dürfen wir die LAGRANGE-*Polynome*

$$\ell_k(x) := \prod_{\substack{j=0 \\ j \neq k}}^{m} \frac{x - x_j}{x_k - x_j} \qquad \text{für alle } k \in [0 : m] \tag{6.5}$$

definieren. Für $i, k \in [0 : m]$ untersuchen wir

$$\ell_k(x_i) = \prod_{\substack{j=0 \\ j \neq k}}^{m} \frac{x_i - x_j}{x_k - x_j}$$

und stellen fest, dass für $i \neq k$ in dem Produkt einmal $j = i$ auftritt, so dass wegen $x_i - x_i = 0$ das gesamte Produkt gleich null ist. Für $i = k$ hingegen ist jeder der Faktoren im Produkt gleich eins, so dass wir insgesamt

$$\ell_k(x_i) = \begin{cases} 1 & \text{falls } i = k, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i, k \in [0 : m] \tag{6.6}$$

erhalten. Ein LAGRANGE-Polynom ist also in genau einem Interpolationspunkt gleich eins und in allen anderen gleich null.

Damit können wir nun

$$p := \sum_{k=0}^{m} f(x_k) \ell_k \tag{6.7}$$

[2]Benannt nach Joseph-Louis de Lagrange.

als Kandidaten für ein LAGRANGE-Interpolationspolynom setzen. Mit (6.6) gilt

$$p(x_i) = \sum_{k=0}^{m} f(x_k)\ell_k(x_i) = f(x_i)\ell_i(x_i) = f(x_i) \qquad \text{für alle } i \in [0:m],$$

also erfüllt p tatsächlich die Bedingungen (6.4). Damit ist die Interpolationsaufgabe immer lösbar.

Um die Frage nach der Eindeutigkeit der Lösung zu klären, halten wir zunächst fest, dass die LAGRANGE-Polynome eine Basis des Raums aller Polynome sind.

Lemma 6.1 (Lagrange-Basis) *Sei $m \in \mathbb{N}_0$, seien $x_0, \ldots, x_m \in \mathbb{R}$ paarweise verschieden. Die durch (6.5) definierten* LAGRANGE*-Polynome $(\ell_k)_{k=0}^m$ sind eine Basis des Raums*

$$\Pi_m := \left\{ x \mapsto \sum_{k=0}^{m} a_k x^k \; : \; a_0, \ldots, a_m \in \mathbb{R} \right\}$$

aller Polynome höchstens m-ten Grades.

Beweis. Seien $a_0, \ldots, a_m \in \mathbb{R}$ Koeffizienten mit

$$0 = \sum_{k=0}^{m} a_k \ell_k.$$

Dann folgt wieder mit (6.6)

$$a_i = \sum_{k=0}^{m} a_k \ell_k(x_i) = \left(\sum_{k=0}^{m} a_k \ell_k \right)(x_i) = 0 \qquad \text{für alle } i \in [0:m],$$

also sind die LAGRANGE-Polynome linear unabhängig. Da wir $m+1$ solche Polynome haben und die Dimension des Raums Π_m höchstens $m+1$ betragen kann, müssen sie auch eine Basis sein. ∎

Seien nun $p, q \in \Pi_m$ zwei Polynome, die die Gleichungen (6.4) erfüllen. Nach Lemma 6.1 können wir beide in der LAGRANGE-Basis darstellen, indem wir Koeffizienten $a_0, \ldots, a_m, b_0, \ldots, b_m \in \mathbb{R}$ mit

$$p = \sum_{k=0}^{m} a_k \ell_k, \qquad q = \sum_{k=0}^{m} b_k \ell_k$$

wählen. Mit (6.6) und (6.4) gilt

$$a_i = \sum_{k=0}^{m} a_k \ell_k(x_i) = p(x_i) = f(x_i) = q(x_i) = \sum_{k=0}^{m} b_k \ell_k(x_i) = b_i \qquad \text{für alle } i \in [0:m],$$

also folgt

$$p = \sum_{k=0}^{m} a_k \ell_k = \sum_{k=0}^{m} b_k \ell_k = q.$$

Damit ist die LAGRANGE-Interpolationsaufgabe *eindeutig* lösbar.

6.3 Newton-Interpolation

Die Darstellung (6.7) des Interpolationspolynoms in der LAGRANGE-Basis ist für theoretische Betrachtungen gut geeignet, für praktische Berechnungen allerdings eher nicht: Die Auswertung eines LAGRANGE-Polynoms erfordert nach Definition mindestens m Subtraktionen und $m-1$ Multiplikationen, so dass eine direkte Auswertung der Formel (6.7) mindestens $(m+1)(m-1) = m^2 - 1$ Operationen erfordern würde.

Wesentlich attraktiver wäre es, wenn wir das bereits bei der TAYLOR-Entwicklung eingesetzte HORNER-Schema verwenden könnten. Das ist möglich, wir müssen es allerdings etwas modifizieren.

Sei p das gesuchte Interpolationspolynom, und sei $q \in \Pi_{m-1}$ ein zweiter Interpolationspolynom, das lediglich die ersten m unserer definierenden Gleichungen erfüllt, also

$$q(x_i) = f(x_i) \qquad \text{für alle } i \in [0 : m-1].$$

Dann gilt offenbar

$$p(x_i) - q(x_i) = f(x_i) - f(x_i) = 0 \qquad \text{für alle } i \in [0 : m-1],$$

so dass die Differenz sich in der Form

$$p(x) - q(x) = (x - x_0) \dots (x - x_{m-1})d \qquad \text{für alle } x \in [a, b]$$

mit einer geeigneten Konstanten d schreiben lässt: Die rechte Seite der Gleichung ist ein Polynom m-ten Grades, das in $x_0, \dots, x_{m-1}$ mit der linken Seite überein stimmt. Wir wählen d so, dass beide Seiten auch in x_m überein stimmen, denn dann müssen sie nach den Betrachtungen des vorigen Abschnitts insgesamt identisch sein.

Wir erhalten die Darstellung

$$p(x) = q(x) + (x - x_0) \cdots (x - x_{m-1})d \qquad \text{für alle } x \in [a, b],$$

mit der wir aus einem Polynom $(m-1)$-ten Grades ein Polynom m-ten Grades konstruieren können, das die Interpolationsbedingung in einem weiteren Punkt erfüllt. Das Produkt $(x - x_0) \cdots (x - x_{m-1})$ soll nun an die Stelle der Potenzen $(x - x_0)^m$ im TAYLOR-Polynom treten.

Wir definieren dazu die NEWTON-*Polynome*

$$n_{i,j}(x) := \begin{cases} 1 & \text{falls } i = j, \\ (x - x_i) n_{i+1,j}(x) & \text{ansonsten} \end{cases} \qquad \text{für alle } i, j \in [0 : m] \text{ mit } i \leq j. \tag{6.8}$$

Mit der Konvention, dass das leere Produkt gleich eins ist, können wir sie anschaulich auch in der Form

$$n_{i,j}(x) = \prod_{k=i}^{j-1} x - x_k \qquad \text{für alle } x \in [a, b],\ i, j \in [0 : m] \text{ mit } i \leq j$$

```
function newton_horner(d, x_0, ..., x_m, y);
p ← d_m;
for ℓ = 1 to m do
  p ← d_{m−ℓ} + (y − x_{m−ℓ}) p;
return p
end
```

Abbildung 6.2: Newton-Horner-Schema

schreiben, aus der sich unmittelbar die enge Verwandtschaft mit den Taylor-Polynomen $(x-x_0)^k$ ablesen lässt.

Wir suchen nach Koeffizienten $d_0, \ldots, d_m \in \mathbb{R}$, mit denen sich das Interpolationspolynom in der Form

$$p(x) = \sum_{k=0}^{m} d_k n_{0,k}(x) \qquad \text{für alle } x \in [a,b] \tag{6.9}$$

schreiben lässt. Falls wir nämlich solche Koeffizienten finden, können wir analog zum Horner-Schema

$$\begin{aligned} p(x) &= \sum_{k=0}^{m} d_k n_{0,k}(x) = d_0 + \sum_{k=1}^{m} d_k n_{0,k}(x) \\ &= d_0 + (x-x_0) \sum_{k=1}^{m} d_k n_{1,k}(x) \qquad \text{für alle } x \in [a,b] \end{aligned}$$

schreiben und mit den Zwischenergebnissen

$$p_\ell(x) := \sum_{k=m-\ell}^{m} d_k n_{m-\ell,k}(x) \qquad \text{für alle } \ell \in [0:m],\ x \in [a,b]$$

zu den Gleichungen $p_0(x) = d_m$, $p_m(x) = p(x)$ sowie

$$\begin{aligned} p_{\ell+1}(x) &= \sum_{k=m-\ell-1}^{m} d_k n_{m-\ell-1,k}(x) \\ &= d_{m-\ell-1} + (x - x_{m-\ell-1}) \sum_{k=m-\ell}^{m} d_k n_{m-\ell,k}(x) \\ &= d_{m-\ell-1} + (x - x_{m-\ell-1}) p_\ell(x) \qquad \text{für alle } \ell \in [0:m-1],\ x \in [a,b] \end{aligned}$$

gelangen. Die resultierende Variante des Horner-Schemas ist in Abbildung 6.2 zusammengefasst, sie erfordert offenbar $3m$ Gleichkommaoperationen, um ein in der Newton-Darstellung (6.9) gegebenes Polynom in einem Punkt $y \in [a,b]$ auszuwerten.

Es bleibt noch zu klären, ob wir das Interpolationspolynom p in der für diesen Algorithmus erforderlichen Form (6.9) darstellen können. Falls das möglich ist, müsste

$$\sum_{k=0}^{m} d_k n_{0,k}(x_i) = p(x_i) = f(x_i) \qquad \text{für alle } i \in [0:m]$$

gelten. Bei genauerer Betrachtung entpuppen sich diese Gleichungen als lineares Gleichungssystem: Wir definieren $A \in \mathbb{R}^{[0:m]\times[0:m]}$ und $b \in \mathbb{R}^{[0:m]}$ durch

$$a_{ik} = n_{0,k}(x_i), \qquad b_i = f(x_i) \qquad \text{für alle } i,k \in [0:m]$$

und erhalten

$$\sum_{k=0}^{m} a_{ik} d_k = b_i \qquad \text{für alle } i \in [0:m],$$

also

$$Ad = b. \tag{6.10}$$

Zu klären sind also die Fragen ob das System immer lösbar ist und wie sich diese Lösung effizient berechnen lässt. Glücklicherweise können wir in diesem speziellen Fall beide Fragen gleichzeitig beantworten: Es gilt

$$a_{ik} = n_{0,k}(x_i) = \prod_{j=0}^{k-1}(x_i - x_j) \qquad \text{für alle } i,k \in [0:m],$$

so dass für $i < k$ unmittelbar $a_{ik} = 0$ folgt, also ist A eine untere Dreiecksmatrix. Für die Diagonalelemente gilt

$$a_{kk} = \prod_{j=0}^{k-1}(x_k - x_j) \qquad \text{für alle } k \in [0:m],$$

und aus $j < k$ folgt $x_k \neq x_j$, also sind alle Diagonalelemente ungleich null. Da A eine Dreiecksmatrix ist, folgt daraus unmittelbar, dass die Matrix auch invertierbar ist.

Das System (6.10) lässt sich also für beliebige rechte Seiten immer durch Vorwärtseinsetzen lösen. Die Berechnung der Matrixkoeffizienten lässt sich mit Hilfe der Gleichungen

$$a_{i,0} = 1, \quad a_{i,k} = n_{0,k}(x_i) = n_{0,k-1}(x_i)(x_i - x_{k-1}) \qquad \text{für alle } i \in [0:m],\ k \in [1:m]$$

in nur

$$\sum_{i=0}^{m} 2i = 2\frac{m(m+1)}{2} = m(m+1)$$

Gleitkommaoperationen bewerkstelligen, das Vorwärtseinsetzen benötigt weitere $(m+1)^2$ Operationen, so dass für die Berechnung der Koeffizienten insgesamt $(2m+1)(m+1) \leq 2(m+1)^2$ Operationen erforderlich sind.

6.4 Interpolationsfehler*

Lemma 6.2 (Nullstellen) *Sei $m \in \mathbb{N}_0$, sei $g \in C^m[a,b]$. Falls g mindestens $m+1$ verschiedene Nullstellen in $[a,b]$ besitzt, besitzt die Ableitung $g^{(m)}$ mindestens eine Nullstelle.*

Beweis. Wir führen den Beweis per Induktion über $m \in \mathbb{N}_0$.

Induktionsanfang: Für $m = 0$ ist die Aussage offensichtlich.

Induktionsvoraussetzung: Sei $m \in \mathbb{N}_0$ so gegeben, dass die Aussage gilt.

Induktionsschritt: Sei $g \in C^{m+1}[a,b]$ eine Funktion mit $m+2$ verschiedenen Nullstellen $a \leq x_0 < x_1 < \ldots < x_m < x_{m+1} \leq b$.

Für jedes $i \in [0 : m]$ gilt dann nach Voraussetzung $g(x_i) = 0 = g(x_{i+1})$. Nach dem Satz von Rolle[3] muss dann die Ableitung g' eine Nullstelle $y_i \in (x_i, x_{i+1})$ besitzen.

Demnach besitzt die Ableitung g' mindestens $m+1$ Nullstellen $y_0 < y_1 < \ldots < y_m$. Da $g' \in C^m[a,b]$ gilt, dürfen wir die Induktionsvoraussetzung anwenden, um zu folgern, dass $g^{(m+1)}$ mindestens eine Nullstelle besitzt. ∎

Satz 6.3 (Interpolationsfehler) *Sei $m \in \mathbb{N}_0$, $f \in C^{m+1}[a,b]$, und sei $p \in \Pi_m$ das durch (6.4) definierte Interpolationspolynom.*

Für jedes $x \in [a,b]$ existiert ein $\eta \in [a,b]$ mit

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

Beweis. Sei $x \in [a,b]$. Falls x einer der Interpolationspunkte ist, also $x \in \{x_0, \ldots, x_m\}$ gilt, ist die Aussage trivial für jedes beliebige $\eta \in [a,b]$ erfüllt.

Gelte also nun $x \notin \{x_0, \ldots, x_m\}$. Wir definieren das Polynom $q \in \Pi_{m+1}$ durch

$$q(y) := p(y) + (y - x_0) \cdots (y - x_m)\alpha \qquad \text{für alle } y \in \mathbb{R}$$

mit der Konstanten

$$\alpha := \frac{f(x) - p(x)}{(x - x_0) \cdots (x - x_m)}.$$

Der Nenner ist nach unserer Voraussetzung wohldefiniert. Dann gelten insbesondere

$$q(x) = p(x) + (x - x_0) \cdots (x - x_m)\alpha = p(x) + f(x) - p(x) = f(x),$$
$$q(x_i) = p(x_i) + (x_i - x_0) \cdots (x_i - x_m)\alpha = p(x_i) = f(x_i) \qquad \text{für alle } i \in [0 : m],$$

so dass die Funktion $g := f - q$ mindestens $m+2$ Nullstellen $x, x_0, \ldots, x_m$ in $[a,b]$ besitzt. Nach Lemma 6.2 besitzt ihre $(m+1)$-te Ableitung dann noch mindestens eine Nullstelle $\eta \in [a,b]$, für die

$$0 = g^{(m+1)}(\eta) = f^{(m+1)}(\eta) - q^{(m+1)}(\eta)$$

[3] Benannt nach Michel Rolle.

$$= f^{(m+1)}(\eta) - p^{(m+1)}(\eta) - (m+1)!\alpha = f^{(m+1)}(\eta) - (m+1)!\alpha$$

gilt, da p ein Polynom höchstens m-ten Grades ist. Es folgt

$$\alpha = \frac{f^{(m+1)}(\eta)}{(m+1)!},$$

also insbesondere

$$f(x) = q(x) = p(x) + (x - x_0)\cdots(x - x_m)\alpha = p(x) + (x - x_0)\cdots(x - x_m)\frac{f^{(m+1)}(\eta)}{(m+1)!}.$$

Das ist die gewünschte Aussage. ∎

Wenn wir das *Stützstellenpolynom*

$$\omega\colon \mathbb{R} \to \mathbb{R}, \qquad x \mapsto (x - x_0)\cdots(x - x_m),$$

einführen, können wir die Aussage des Satzes kompakt als

$$f(x) - p(x) = \omega(x)\frac{f^{(m+1)}(\eta)}{(m+1)!}$$

schreiben und analog zu (6.1) mit der Maximumnorm die kürzere Form

$$\|f - p\|_{\infty,[a,b]} \leq \|\omega\|_{\infty,[a,b]}\frac{\|f^{(m+1)}\|_{\infty,[a,b]}}{(m+1)!}$$

erhalten. Für jede beliebige Wahl der Interpolationspunkte haben wir $\|\omega\|_{\infty,[a,b]} \leq (b - a)^{m+1}$.

Falls wir hingegen die Tschebyscheff-*Interpolationspunkte* [4] verwenden, die durch

$$x_i := \frac{b+a}{2} + \frac{b-a}{2}\cos\left(\pi\frac{2i+1}{2m+2}\right) \qquad \text{für alle } i \in [0:m]$$

gegeben sind, reduziert sich die Größe auf

$$\|\omega\|_{\infty,[a,b]} \leq 2\left(\frac{b-a}{4}\right)^{m+1},$$

und wir können zeigen, dass sie nicht kleiner werden kann. Die so definierte Tschebyscheff-Interpolation kann also in gewisser Weise als bestmögliche Interpolation interpretiert werden. Sie weist auch in andere Hinsicht sehr attraktive Eigenschaften auf.

[4] Benannt nach Pafnuti Lwowitsch Tschebyscheff.

7 Numerische Integration

Neben der bereits diskutierten Differentiation ist auch die Integration eine Aufgabe, die sich häufig nur mit Hilfe numerischer Algorithmen lösen lässt.

Ein Beispiel ist die Berechnung der Wahrscheinlichkeit, dass eine normalverteilte Zufallsvariable einen Wert zwischen zwei Zahlen a und b annimmt: Die Wahrscheinlichkeit ist durch

$$P(\{a \leq X \leq b\}) = \frac{1}{\sqrt{2\pi}} \int_a^b \exp(-x^2)\,dx$$

gegeben, und für die Funktion $x \mapsto \exp(-x^2)$ ist keine Stammfunktion bekannt. Dieses Integral wird deshalb in der Praxis mit Hilfe numerischer Verfahren angenähert.

Ein weiteres Beispiel ist die Bestimmung des Inhalts einer von Kurven eingeschlossenen Fläche oder eines von Oberflächen eingeschlossenen Volumens. Auch hier müssen Integrale berechnet werden, für die häufig keine handlicher analytischer Ansatz bekannt ist.

7.1 Interpolatorische Quadratur

Sei $f \in C[a,b]$. Wir untersuchen die näherungsweise Berechnung des Riemann-Integrals[1]

$$\int_a^b f(x)\,dx.$$

Um die Intervallgrenzen a und b nicht explizit mitführen zu müssen, empfiehlt es sich, die Transformation

$$\Phi_{a,b}\colon [-1,1] \to [a,b], \qquad \hat{x} \mapsto \frac{b+a}{2} + \frac{b-a}{2}\hat{x},$$

einzuführen, die das *Referenzintervall* $[-1,1]$ streng monoton auf $[a,b]$ abbildet und

$$\Phi_{a,b}(-1) = a, \qquad \Phi_{a,b}(1) = b, \qquad \Phi'_{a,b} = \frac{b-a}{2}$$

erfüllt. Durch Substitution erhalten wir

$$\int_a^b f(x)\,dx = \frac{b-a}{2} \int_{-1}^1 f(\Phi_{a,b}(\hat{x}))\,d\hat{x}. \tag{7.1}$$

Falls wir also einen Algorithmus finden können, der Integrale auf dem Referenzintervall approximiert, lässt er sich auf beliebige Intervalle übertragen.

[1] Benannt nach Bernhard Riemann.

Definition 7.1 (Quadraturformel) *Seien* $m \in \mathbb{N}_0$ *und* $x_0, \ldots, x_m \in [a,b]$ *sowie* $w_0, \ldots, w_m \in \mathbb{R}$ *gegeben. Die Abbildung*

$$\mathcal{Q}_{[a,b]} \colon C[a,b] \to \mathbb{R}, \qquad f \mapsto \sum_{k=0}^{m} w_k f(x_k), \tag{7.2}$$

bezeichnen wir als Quadraturformel *der Stufe* m *für das Intervall* $[a,b]$ *zu den* Quadraturpunkten $x_0, \ldots, x_m$ *und den* Quadraturgewichten $w_0, \ldots, w_m$*.*

Falls $[a,b]$ *das Referenzintervall ist, schreiben wir die Quadraturformel auch einfach als* $\mathcal{Q}$*.*

Aus der Gleichung (7.1) ergibt sich unmittelbar ein einfacher Zugang, um aus einer Quadraturformel für das Referenzintervall $[-1,1]$ eine Quadraturformel für ein beliebiges Intervall zu konstruieren.

Definition 7.2 (Transformierte Quadraturformel) *Sei* $\mathcal{Q}$ *eine Quadraturformel der Stufe* m *für das Referenzintervall mit Quadraturpunkten* $\hat{x}_0, \ldots, \hat{x}_m$ *und Quadraturgewichten* $\hat{w}_0, \ldots, \hat{w}_m$*.*

Die durch

$$x_k := \Phi_{a,b}(\hat{x}_k), \qquad w_k := \frac{b-a}{2}\hat{w}_k \qquad \text{für alle } k \in [0:m]$$

definierte Quadraturformel $\mathcal{Q}_{[a,b]}$ *auf dem Intervall* $[a,b]$ *nennen wir die zu* $\mathcal{Q}$ *gehörende* transformierte Quadraturformel*.*

Für eine beliebige Funktion $f \in C[a,b]$ gilt offenbar

$$\mathcal{Q}_{[a,b]}(f) = \sum_{k=0}^{m} w_k f(x_k) = \frac{b-a}{2}\sum_{k=0}^{m} \hat{w}_k f(\Phi_{a,b}(\hat{x}_k)) = \frac{b-a}{2}\mathcal{Q}(f \circ \Phi_{a,b}), \tag{7.3}$$

also dürfen wir erwarten, dass $\mathcal{Q}_{[a,b]}$ Integrale über $[a,b]$ gut approximiert, falls $\mathcal{Q}$ dasselbe für Integrale über $[-1,1]$ leistet.

Ein erfolgreicher Ansatz für die Konstruktion von Quadraturformel beruht auf der in Kapitel 6 beschriebenen Interpolation: Wenn wir Interpolationspunkte $x_0, \ldots, x_m \in [a,b]$ gewählt haben, können wir ein Interpolationspolynom $p \in \Pi_m$ zu einer Funktion f mit Hilfe der Lagrange-Polynome in der Form

$$p = \sum_{k=0}^{m} f(x_k)\ell_k$$

darstellen, und wir dürfen darauf hoffen, dass p eine passable Approximation der Funktion f ist.

Die Idee der *interpolatorischen Quadratur* besteht nun darin, im zu berechnenden Integral die Funktion f durch ihre Approximation p zu ersetzen. Mit der Lagrange-Darstellung (6.7) erhalten wir

$$\int_a^b f(x)\,dx \approx \int_a^b p(x)\,dx = \sum_{k=0}^{m} f(x_k) \int_a^b \ell_k(x)\,dx.$$

Ein Vergleich mit (7.2) legt nahe,

$$w_k := \int_a^b \ell_k(x)\,dx$$

zu setzen, um

$$\int_a^b f(x)\,dx \approx \int_a^b p(x)\,dx = \mathcal{Q}_{[a,b]}(f)$$

zu erhalten.

Definition 7.3 (Interpolatorische Quadratur) *Seien $m \in \mathbb{N}_0$ und paarweise verschiedene $x_0, \ldots, x_m \in [a,b]$ gegeben. Die durch die Quadraturpunkte $x_0, \ldots, x_m$ und die Quadraturgewichte*

$$w_k := \int_a^b \ell_k(x)\,dx \qquad \text{für alle } k \in [0:m]$$

definierte Quadraturformel nennen wir die zu $x_0, \ldots, x_m$ und $[a,b]$ gehörende interpolatorische Quadraturformel*.*

Interpolatorische Quadraturformeln bieten einen einfachen Ansatz, mit dem sich die Quadraturgewichte zu vorliegenden Quadraturpunkten bestimmen lassen: Sei $i \in [0:m]$. Dann ist das Monom $x \mapsto x^i$ ein Element des Raums Π_m der Polynome höchstens m-ten Grades, muss also mit seinem Interpolationspolynom

$$p = \sum_{k=0}^{m} x_k^i \ell_k$$

übereinstimmen. Wenn nun $\mathcal{Q}$ eine interpolatorische Quadraturformel m-ter Stufe für das Referenzintervall ist, muss $\mathcal{Q}(p)$ das exakte Integral sein. Wir haben also

$$\begin{aligned}\sum_{k=0}^{m} w_k x_k^i &= \int_{-1}^{1} x^i\,dx = \left[\frac{x^{i+1}}{i+1}\right]_{x=-1}^{1} \\ &= \begin{cases} 0 & \text{falls } i \text{ ungerade,} \\ \frac{2}{i+1} & \text{ansonsten} \end{cases} \qquad \text{für alle } i \in [0:m].\end{aligned}$$

Das ist ein lineares Gleichungssystem für die Gewichte $w_0, \ldots, w_m \in \mathbb{R}$. Indem wir

$$a_{ik} := x_k^i, \qquad b_i := \begin{cases} 0 & \text{falls } i \text{ ungerade,} \\ \frac{2}{i+1} & \text{ansonsten} \end{cases} \qquad \text{für alle } i, k \in [0:m]$$

definieren, können wir es in der gewohnten Form

$$Aw = b,$$

schreiben und mit den bereits diskutierten Verfahren behandeln.

Zwei besonders einfache Quadraturformeln wollen wir uns etwas genauer anschauen.

Definition 7.4 (Mittelpunktregel) *Die interpolatorische Quadraturformel für den Mittelpunkt* $x_0 = (b+a)/2$ *des Intervalls* $[a,b]$ *nennen wir die* Mittelpunktregel*.*

Für das Referenzintervall ist sie durch

$$\mathcal{M}\colon C[-1,1] \to \mathbb{R}, \qquad\qquad f \mapsto 2f(0),$$

gegeben.

Definition 7.5 (Trapezregel) *Die interpolatorische Quadraturformel für die Endpunkte* $x_0 = a$ *und* $x_1 = b$ *des Intervalls* $[a,b]$ *nennen wir die* Trapezregel*.*

Für das Referenzintervall ist sie durch

$$\mathcal{T}\colon C[-1,1] \to \mathbb{R}, \qquad\qquad f \mapsto f(-1) + f(1),$$

gegeben.

Definition 7.6 (Newton-Cotes-Regeln) *Sei* $m \in \mathbb{N}$*. Wir definieren*

$$x_i := a + \frac{i}{m}(b-a) \qquad\qquad \text{für alle } i \in [0:m].$$

Die zugehörige interpolatorische Quadraturformel m*-ter Stufe nennen wir die* Newton-Cotes-Regel[2] m-ter Stufe*.*

Die Newton-Cotes-*Regel erster Stufe ist offenbar gerade die Trapezregel.*

Es stellt sich die Frage, wie genau diese Quadraturformeln sind. Natürlich können wir Fälle konstruieren, in denen beide Formeln zu beliebig schlechten Ergebnissen führen. Falls der Integrand f allerdings hinreichend oft integrierbar ist, lässt sich der Fehler näher beschreiben.

Lemma 7.7 (Trapezregel) *Sei* $f \in C^2[-1,1]$*. Dann existiert ein* $\eta \in [-1,1]$ *mit*

$$\int_{-1}^{1} f(x)\,dx - \mathcal{T}(f) = -\frac{2}{3} f''(\eta).$$

Beweis. Wir definieren

$$p(x) = \frac{1-x}{2} f(-1) + \frac{x+1}{2} f(1) \qquad\qquad \text{für alle } x \in \mathbb{R}.$$

Offenbar ist p ein lineares Polynom, das $p(-1) = f(-1)$ und $p(1) = f(1)$ erfüllt, also ist es das Interpolationspolynom und es gilt

$$\mathcal{T}(f) = \int_{-1}^{1} p(x)\,dx$$

nach Konstruktion der interpolatorischen Quadraturformel.

[2] Benannt nach Isaac Newton und Roger Cotes.

Wir definieren die Hilfsfunktion

$$\varphi\colon \mathbb{R} \mapsto \mathbb{R}, \qquad\qquad x \mapsto \frac{x^2-1}{2},$$

und halten fest, dass

$$\varphi'(x) = x, \qquad \varphi''(x) = 1 \qquad \text{für alle } x \in \mathbb{R}$$

gelten. Mit Hilfe der partiellen Integration erhalten wir

$$\begin{aligned}\int_{-1}^{1} f(x)\,dx - \mathcal{T}(f) &= \int_{-1}^{1} f(x) - p(x)\,dx = \int_{-1}^{1} \varphi''(x)(f-p)(x)\,dx \\ &= \left[\varphi'(x)\,(f-p)(x)\right]_{x=-1}^{1} - \int_{-1}^{1} \varphi'(x)(f-p)'(x)\,dx,\end{aligned}$$

so dass sich wegen $f(-1) = p(-1)$ und $f(1) = p(1)$ unmittelbar

$$\int_{-1}^{1} f(x)\,dx - \mathcal{T}(f) = -\int_{-1}^{1} \varphi'(x)(f-p)'(x)\,dx$$

ergibt. Wir greifen erneut auf partielle Integration zurück, um zu

$$\begin{aligned}\int_{-1}^{1} f(x)\,dx - \mathcal{T}(f) &= -\int_{-1}^{1} \varphi'(x)(f-p)'(x)\,dx \\ &= \left[-\varphi(x)(f-p)'(x)\right]_{x=-1}^{1} + \int_{-1}^{1} \varphi(x)(f-p)''(x)\,dx\end{aligned}$$

zu gelangen. Da $\varphi(-1) = \varphi(1) = 0$ gilt, entfällt der erste Term, und dank $p'' = 0$ vereinfacht sich auch das verbliebene Integral. Mit dem Mittelwertsatz der Integralrechnung finden wir ein $\eta \in [-1,1]$ mit

$$\begin{aligned}\int_{-1}^{1} f(x)\,dx - \mathcal{T}(f) &= \int_{-1}^{1} \varphi(x) f''(x)\,dx \\ &= f''(\eta) \int_{-1}^{1} \varphi(x)\,dx = f''(\eta) \left[\frac{x^3/3 - x}{2}\right]_{x=-1}^{1} = -\frac{2}{3} f''(\eta).\end{aligned}$$

■

Lemma 7.8 (Mittelpunktregel) *Sei $f \in C^2[-1,1]$. Dann existiert ein $\eta \in [-1,1]$ mit*

$$\int_{-1}^{1} f(x)\,dx - \mathcal{M}(f) = \frac{1}{3} f''(\eta).$$

Beweis. Wir stellen fest, dass

$$\mathcal{M}(f) = 2f(0) = \int_{-1}^{1} f(0)\,dx$$

gilt, so dass sich der Fehler in der Form

$$\int_{-1}^{1} f(x)\,dx - \mathcal{M}(f) = \int_{-1}^{1} f(x) - f(0)\,dx$$

schreiben lässt. Wir definieren die Funktionen

$$\varphi_+(x) := \frac{(x-1)^2}{2}, \qquad \varphi_-(x) := \frac{(x+1)^2}{2} \qquad \text{für alle } x \in \mathbb{R}$$

und halten fest, dass

$$\varphi_+'(x) = x - 1, \qquad \varphi_-'(x) = x + 1, \qquad \varphi_+''(x) = \varphi_-''(x) = 1 \qquad \text{für alle } x \in \mathbb{R}$$

gelten. Wie im vorigen Beweis verwenden wir zweimal partielle Integration und erhalten

$$\begin{aligned}
\int_{-1}^{1} f(x)\,dx - \mathcal{M}(f) &= \int_{-1}^{0} f(x) - f(0)\,dx + \int_{0}^{1} f(x) - f(0)\,dx \\
&= \int_{-1}^{0} \varphi_-''(x)(f(x) - f(0))\,dx + \int_{0}^{1} \varphi_+''(x)(f(x) - f(0))\,dx \\
&= \big[\varphi_-'(x)(f(x) - f(0))\big]_{x=-1}^{0} - \int_{-1}^{0} \varphi_-'(x) f'(x)\,dx \\
&\quad + \big[\varphi_+'(x)(f(x) - f(0))\big]_{x=0}^{1} - \int_{0}^{1} \varphi_+'(x) f'(x)\,dx \\
&= -\int_{-1}^{0} \varphi_-'(x) f'(x)\,dx - \int_{0}^{1} \varphi_+(x) f'(x)\,dx \\
&= -\big[\varphi_-(x) f'(x)\big]_{x=-1}^{0} + \int_{-1}^{0} \varphi_-(x) f''(x)\,dx \\
&\quad - \big[\varphi_+(x) f'(x)\big]_{x=0}^{1} + \int_{0}^{1} \varphi_+(x) f''(x)\,dx \\
&= -\frac{f(0)}{2} + \frac{f(0)}{2} + \int_{-1}^{0} \varphi_-(x) f''(x)\,dx + \int_{0}^{1} \varphi_+(x) f''(x)\,dx \\
&= \int_{-1}^{0} \varphi_-(x) f''(x)\,dx + \int_{0}^{1} \varphi_+(x) f''(x)\,dx.
\end{aligned}$$

Mit dem Mittelwertsatz der Integralrechnung erhalten wir $\eta_- \in [-1, 0]$ und $\eta_+ \in [0, 1]$ mit

$$\begin{aligned}
\int_{-1}^{1} f(x)\,dx - \mathcal{M}(f) &= f''(\eta_-) \int_{-1}^{0} \varphi_-(x)\,dx + f''(\eta_+) \int_{0}^{1} \varphi_+(x)\,dx \\
&= f''(\eta_-) \left[\frac{(x+1)^3}{6}\right]_{x=-1}^{0} + f''(\eta_+) \left[\frac{(x-1)^3}{6}\right]_{x=0}^{1} \\
&= \frac{f''(\eta_-) + f''(\eta_+)}{6}.
\end{aligned}$$

Mit dem Zwischenwertsatz finden wir ein $\eta \in [\eta_-, \eta_+]$ mit $(f''(\eta_-) + f''(\eta_+))/2 = f''(\eta)$, so dass unsere Aussage folgt. ∎

Für sich genommen sind die beiden Aussagen erst einmal nicht besonders ermutigend, denn es ist kein Mechanismus erkennbar, mit dem wir den Quadraturfehler reduzieren könnten, um unsere Genauigkeitsanforderungen zu erfüllen.

Einen Hinweis auf eine mögliche Lösung bieten transformierte Quadraturformeln:

Lemma 7.9 (Transformierte Quadraturformeln) *Wir bezeichnen mit $\mathcal{M}_{[a,b]}$ und $\mathcal{T}_{[a,b]}$ die auf das Intervall $[a,b]$ transformierte Mittelpunkt- und Trapezregel.*

Für jedes $f \in C^2[a,b]$ existieren $\eta_m, \eta_t \in [a,b]$ mit

$$\int_a^b f(x)\,dx - \mathcal{M}_{[a,b]}(f) = \frac{(b-a)^3}{24} f''(\eta_m),$$
$$\int_a^b f(x)\,dx - \mathcal{T}_{[a,b]}(f) = -\frac{(b-a)^3}{12} f''(\eta_t).$$

Beweis. Wir definieren $\hat{f} := f \circ \Phi_{a,b}$. Nach der Kettenregel gelten

$$\hat{f}'(\hat{x}) = \frac{b-a}{2} f'(\Phi_{a,b}(\hat{x})), \quad \hat{f}''(\hat{x}) = \frac{(b-a)^2}{4} f''(\Phi_{a,b}(\hat{x})) \qquad \text{für alle } \hat{x} \in [-1,1]. \tag{7.4}$$

Mit (7.3) erhalten wir

$$\mathcal{M}_{[a,b]}(f) = \frac{b-a}{2}\mathcal{M}(\hat{f}), \qquad \mathcal{T}_{[a,b]}(f) = \frac{b-a}{2}\mathcal{T}(\hat{t}),$$

so dass sich mit den vorigen Lemmas 7.7 und 7.8 $\hat{\eta}_m, \hat{\eta}_t \in [-1,1]$ mit

$$\begin{aligned}
\int_a^b f(x)\,dx - \mathcal{M}_{[a,b]}(f) &= \frac{b-a}{2}\int_{-1}^1 \hat{f}(\hat{x})\,d\hat{x} - \frac{b-a}{2}\mathcal{M}(\hat{f}) \\
&= \frac{b-a}{2}\left(\int_{-1}^1 \hat{f}(\hat{x})\,d\hat{x} - \mathcal{M}(\hat{f})\right) = \frac{b-a}{6}\hat{f}''(\hat{\eta}_m), \\
\int_a^b f(x)\,dx - \mathcal{T}_{[a,b]}(f) &= \frac{b-a}{2}\int_{-1}^1 \hat{f}(\hat{x})\,d\hat{x} - \frac{b-a}{2}\mathcal{T}(\hat{f}) \\
&= \frac{b-a}{2}\left(\int_{-1}^1 \hat{f}(\hat{x})\,d\hat{x} - \mathcal{T}(\hat{f})\right) = -\frac{b-a}{3}\hat{f}''(\hat{\eta}_t)
\end{aligned}$$

finden lassen. Wir definieren $\eta_m := \Phi_{a,b}(\hat{\eta}_m) \in [a,b]$ sowie $\eta_t := \Phi_{a,b}(\hat{\eta}_t) \in [a,b]$ und verwenden (7.4), um

$$\int_a^b f(x)\,dx - \mathcal{M}_{[a,b]}(f) = \frac{b-a}{6}\frac{(b-a)^2}{4} f''(\eta_m) = \frac{(b-a)^3}{24} f''(\eta_m),$$
$$\int_a^b f(x)\,dx - \mathcal{T}_{[a,b]}(f) = -\frac{b-a}{3}\frac{(b-a)^2}{4} f''(\eta_t) = -\frac{(b-a)^3}{12} f''(\eta_t)$$

zu erhalten. ∎

Die entscheidende Beobachtung besteht darin, dass wir jede beliebige Genauigkeit erreichen können, indem wir die Länge des Intervalls reduzieren.

7.2 Summierte Quadraturformeln

Wir haben soeben gesehen, dass kürzere Intervall zu einer höheren Genauigkeit führen. Leider können wir die Länge der Intervalle in der Regel nicht frei wählen, sie ist durch die Aufgabenstellung festgelegt. Wir können allerdings ein Intervall in Teilintervalle zerlegen und hoffen, dass die Summe der Teile günstiger ist als das Ganze.

Der Einfachheit halber zerlegen wir das Intervall $[a, b]$ in gleich große Teile. Für ein $k \in \mathbb{N}$ ist die *äquidistante Zerlegung* des Intervalls $[a, b]$ in k gleich große Teilintervall gegeben durch

$$y_i := a + \frac{b-a}{k} i \qquad \text{für alle } i \in [0 : k].$$

Für das Integral gilt

$$\int_a^b f(x)\,dx = \sum_{i=1}^k \int_{y_{i-1}}^{y_i} f(x)\,dx,$$

also bietet es sich an, die summierten Teilintegrale durch Quadraturformeln anzunähern.

Definition 7.10 (Summierte Quadraturformel) *Seien $k \in \mathbb{N}$ und $y_0, \ldots, y_k \in [a, b]$ wie oben gegeben. Sei $\mathcal{Q}$ eine Quadraturformel auf dem Referenzintervall. Wir bezeichnen*

$$\mathcal{Q}_{[a,b],k} \colon C[a, b] \to \mathbb{R}, \qquad f \mapsto \sum_{i=1}^k \mathcal{Q}_{[y_{i-1},y_i]}(f),$$

als die summierte Quadraturformel *zu $\mathcal{Q}$ und dem Intervall $[a, b]$ mit k Teilintervallen.*

Lemma 7.11 (Summierte Trapezregel) *Die summierte Trapezregel ist durch*

$$\mathcal{T}_{[a,b],k}(f) = \frac{b-a}{k} \left(\frac{f(y_0)}{2} + \sum_{i=1}^{k-1} f(y_i) + \frac{f(y_k)}{2} \right) \qquad \text{für alle } f \in C[a, b],\ k \in \mathbb{N}$$

gegeben. Für jedes $f \in C^2[a, b]$ und jedes $k \in \mathbb{N}$ existiert ein $\eta \in [a, b]$ mit

$$\int_a^b f(x)\,dx - \mathcal{T}_{[a,b],k}(f) = -\frac{(b-a)^3}{12k^2} f''(\eta).$$

Beweis. Sei $f \in C[a, b]$ und $k \in \mathbb{N}$. Die erste Aussage folgt unmittelbar aus

$$\mathcal{Q}_{[y_{i-1},y_i]}(f) = \frac{b-a}{k} \frac{f(y_{i-1}) + f(y_i)}{2} \qquad \text{für alle } i \in [1 : k].$$

Für den Nachweis der zweiten Aussage nehmen wir an, dass $f \in C^2[a, b]$ gilt. Mit Lemma 7.9 finden wir für jedes $i \in [1 : k]$ ein $\eta_i \in [y_{i-1}, y_i]$ mit

$$\int_{y_{i-1}}^{y_i} f(x)\,dx - \mathcal{T}_{[y_{i-1},y_i]}(f) = \frac{(y_i - y_{i-1})^3}{12} f''(\eta_i) = \frac{(b-a)^3}{12k^3} f''(\eta_i),$$

so dass wir für die summierte Quadraturformel

$$\int_a^b f(x)\,dx - \mathcal{T}_{[a,b],k}(f) = \sum_{i=1}^{k} \int_{y_{i-1}}^{y_i} f(x)\,dx - \mathcal{T}_{[y_{i-1},y_i]}(f)$$
$$= \frac{(b-a)^3}{12k^3} \sum_{i=1}^{k} f''(\eta_i)$$

erhalten. Mit dem Zwischenwertsatz finden wir ein $\eta \in [a,b]$ mit

$$\frac{1}{k} \sum_{i=1}^{k} f''(\eta_i) = f''(\eta),$$

so dass sich die Behauptung ergibt. ∎

Bemerkung 7.12 (Summierte Mittelpunktregel) *Die summierte Mittelpunktregel lässt sich entsprechend behandeln, um das folgende Resultat zu erhalten: Für jedes* $f \in C^2[a,b]$ *und jedes* $k \in \mathbb{N}$ *existiert ein* $\eta \in [a,b]$ *mit*

$$\int_a^b f(x)\,dx - \mathcal{M}_{[a,b],k}(f) = \frac{(b-a)^3}{24k^2} f''(\eta).$$

Bei einer summierten Quadraturformel ist es also möglich, die Genauigkeit zu beeinflussen: Sowohl bei der Trapez- als auch bei der Mittelpunktregel führt eine Verdoppelung der Anzahl der Teilintervalle ungefähr zu einer Viertelung des Quadraturfehlers. Wir können also — abgesehen von Rundungsfehlern — jede beliebige Genauigkeit erreichen, sofern wir dazu bereit sind, Rechenzeit zu investieren.

Bemerkung 7.13 (Adaptivität) *Sei* $f \in C^2[a,b]$ *gegeben mit einer zweiten Ableitung, die niemals negativ wird, die also* $f''(x) \geq 0$ *für alle* $x \in [a,b]$ *erfüllt.*

Aus den Lemmas 7.7 und 7.8 folgt

$$\int_a^b f(x)\,dx = \mathcal{T}_{[a,b]}(f) - \frac{(b-a)^3}{12} f''(\eta_t) \leq \mathcal{T}_{[a,b]}(f),$$
$$\int_a^b f(x)\,dx = \mathcal{M}_{[a,b]}(f) + \frac{(b-a)^3}{24} f''(\eta_m) \geq \mathcal{M}_{[a,b]}(f)$$

mit geeigneten $\eta_t, \eta_m \in [a,b]$, *so dass wir das Integral durch die Ergebnisse der Quadraturformeln einschließen können:*

$$\mathcal{M}_{[a,b]}(f) \leq \int_a^b f(x)\,dx \leq \mathcal{T}_{[a,b]}(f).$$

Falls die Differenz $\mathcal{T}_{[a,b]}(f) - \mathcal{M}_{[a,b]}(f)$ *klein ist, müssen demnach auch beide Quadraturfehler klein sein.*

In dieser Situation können wir adaptive *summierte Quadraturformeln konstruieren, also solche, die sich automatisch dem Verhalten der Funktion f anpassen: Für ein gegebenes Intervall $[a, b]$ berechnen wir beide Näherungen und verwenden sie, um den Fehler zu schätzen. Falls er zu groß ist, zerlegen wir das Intervall in zwei Teile und wiederholen die Prozedur für beide Hälften. Anschließend werden die Teilergebnisse addiert, um eine Näherung für das gesamte Intervall zu gewinnen.*

8 Transformationen und Kompression

In modernen Anwendungen sind wir häufig damit konfrontiert, sehr viele Unbekannte behandeln zu müssen, die voneinander abhängen. Ein einfaches Beispiel dafür haben wir bereits bei den schwachbesetzte Matrizen kennen gelernt: Ein lineares Gleichungssystem kann auch dann schwierig zu lösen sein, wenn jede Unbekannte nur mit wenigen anderen in Beziehung steht.

In solchen Situationen kann es attraktiv sein, die Systeme in eine Form zu überführen, die sich besser für die Bearbeitung der relevanten Fragestellung eignet. Ein wichtiges Hilfsmittel in diesem Kontext sind *Variablentransformationen*, bei denen ein Satz von Unbekannten durch einen zweiten, hoffentlich äquivalenten, ersetzt wird.

Falls sich auch das transformierte System noch nicht einfach lösen lässt, kann es zweckmäßig sein, einzelne Gleichungen durch Approximationen zu ersetzen, die Rechenaufwand und Speicherbedarf reduzieren, ohne die Qualität der berechneten Lösung allzu sehr zu beeinträchtigen.

8.1 Fourier-Synthese

Ein wichtiges Beispiel für eine Transformation, mit deren Hilfe sich manche komplizierte Aufgabenstellungen erheblich vereinfachen lassen, ist die FOURIER-*Transformation*[1], die einen Vektor durch eine Überlagerung von Sinus- und Cosinus-Funktionen unterschiedlicher Frequenz darstellt. Diese Transformation ist beispielsweise sehr hilfreich bei der Behandlung von Differentialgleichungen.

Wir führen zur Abkürzung für $n \in \mathbb{N}$ die Notation $Z_n := [0 : n-1]$ für die ganzen Zahlen zwischen 0 und $n-1$ ein.

Die FOURIER-Synthese konstruiert einen Vektor $f \in \mathbb{R}^{Z_n}$ aus Frequenzanteilen $\hat{f}_s, \hat{f}_c \in \mathbb{R}^{Z_n}$ mit der Formel

$$f_i = \sum_{j \in Z_n} \hat{f}_{s,j} \sin(2\pi ij/n) + \sum_{j \in Z_n} \hat{f}_{c,j} \cos(2\pi ij/n) \qquad \text{für alle } i \in Z_n. \tag{8.1}$$

Anschaulich können wir uns vorstellen, dass die Funktion

$$f\colon [0,1] \to \mathbb{R}, \qquad x \mapsto \sum_{j \in Z_n} \hat{f}_{s,j} \sin(2\pi jx) + \sum_{j \in Z_n} \hat{f}_{c,j} \cos(2\pi jx),$$

in den Punkten $x = i/n$ “abgetastet” wird, um den Vektor f zu erhalten.

Die FOURIER-Analyse dagegen nimmt den Vektor f als gegeben an und versucht, die Koeffizienten $\hat{f}_s$ und $\hat{f}_c$ zu rekonstruieren. Beide Transformationen lassen sich erheblich

[1]Benannt nach Jean-Baptiste Joseph Fourier.

einfacher handhaben, indem man komplexe Zahlen einführt: Wir bezeichnen den Körper der komplexen Zahlen mit $\mathbb{C}$ und die *imaginäre Einheit* mit $\iota \in \mathbb{C}$. Sie erfüllt $\iota^2 = -1$. Jede komplexe Zahl $z \in \mathbb{C}$ lässt sich in der Form

$$z = a + \iota b$$

mit dem *Realteil* $a \in \mathbb{R}$ und dem *Imaginärteil* $b \in \mathbb{R}$ darstellen. Summe und Produkt zweier komplexer Zahlen $z = a + \iota b$ und $w = c + \iota d$ lassen sich durch

$$z + w = (a + \iota b) + (c + \iota d) = (a + c) + \iota(b + d), \tag{8.2a}$$

$$zw = (a + \iota b)(c + \iota d) = (ac + \iota ad + \iota bc + \iota^2 bd) = (ac - bd) + \iota(ad + bc) \tag{8.2b}$$

ausdrücken. Die *komplex konjugierte* Zahl zu $z = a + \iota b$ ist durch

$$\bar{z} = a - \iota b,$$

und ihr *Betrag* durch

$$|z| := \sqrt{z\bar{z}} = \sqrt{a^2 + b^2} \tag{8.3}$$

definiert. Die Betragsfunktion ist eine Norm auf $\mathbb{C}$. Real- und Imaginärteil können mittels

$$\Re(z) = a = \tfrac{z+\bar{z}}{2}, \qquad \Im(z) = b = \tfrac{z-\bar{z}}{2\iota}$$

aus z rekonstruiert werden.

Sinus- und Cosinusfunktionen lassen sich besonders elegant mit Hilfe der *komplexen Exponentialfunktion*

$$\exp\colon \mathbb{C} \to \mathbb{C}, \qquad z \mapsto \sum_{i=0}^{\infty} \frac{z^i}{i!},$$

darstellen. Sie erfüllt die üblichen Potenzrechenregeln

$$\exp(z_1 + z_2) = \exp(z_1)\exp(z_2) \qquad \text{für alle } z_1, z_2 \in \mathbb{C}, \tag{8.4a}$$

$$\exp(nz) = \exp(z)^n \qquad \text{für alle } z \in \mathbb{C},\ n \in \mathbb{Z}, \tag{8.4b}$$

und steht über die EULER*sche Gleichung*[2]

$$\exp(\iota\alpha) = \cos(\alpha) + \iota\sin(\alpha) \qquad \text{für alle } \alpha \in \mathbb{R} \tag{8.5}$$

in Beziehung zu der Sinus- und Cosinus-Funktion.

Aufgrund dieser Gleichung gilt mit (8.2b)

$$\begin{aligned}
\hat{f}_{s,i}\sin(2\pi ij/n) + \hat{f}_{c,i}\cos(2\pi ij/n) &= \Re(\hat{f}_{s,i}\sin(2\pi ij/n) + \hat{f}_{c,i}\cos(2\pi ij/n)) \\
&= \Re((\hat{f}_{c,i} - \iota\hat{f}_{s,i})(\cos(2\pi ij/n) + \iota\sin(2\pi ij/n))) \\
&= \Re((\hat{f}_{c,i} - \iota\hat{f}_{s,i})\exp(2\pi\iota ij/n)) \qquad \text{für alle } i, j \in Z_n.
\end{aligned}$$

[2] Benannt nach Leonhard Euler.

Falls wir also die Koeffizienten paarweise zu $\hat{f}_i := \hat{f}_{c,i} - \iota \hat{f}_{s,i}$ zusammenfassen, können wir die FOURIER-Synthese (8.1) kompakt als

$$f_i = \Re\left(\sum_{j \in Z_n} \hat{f}_j \exp(2\pi\iota ij/n)\right) \qquad \text{für alle } i \in Z_n$$

schreiben. Bei genauerer Betrachtung stellt sich heraus, dass diese Gleichung die Koeffizienten $\hat{f}_j$ noch nicht eindeutig festlegt: Bisher ist f lediglich ein Vektor aus reellen Zahlen, während $\hat{f}$ komplexe Zahlen enthält. Um Eindeutigkeit herzustellen, lassen wir auch für f komplexe Zahlen zu und erhalten

$$f_i = \sum_{j \in Z_n} \hat{f}_j \exp(2\pi\iota ij/n) \qquad \text{für alle } i \in Z_n \tag{8.6}$$

mit $f, \hat{f} \in \mathbb{C}^{Z_n}$. Diese Gleichung beschreibt eine Matrix-Vektor-Multiplikation: Wenn wir die FOURIER-Matrix $Q_n \in \mathbb{C}^{Z_n \times Z_n}$ mit

$$q_{n,ij} := \exp(2\pi\iota ij/n) \qquad \text{für alle } i, j \in Z_n$$

definieren, entspricht die FOURIER-Synthese (8.6) gerade der Multiplikation

$$f = Q_n \hat{f}.$$

Bei naiver Vorgehensweise würden wir erwarten, dass die Berechnung des Vektors f aus $\hat{f}$ einen Aufwand proportional zu n^2 aufweist.

Glücklicherweise lassen sich die besonderen Eigenschaften der FOURIER-Matrix Q_n ausnutzen, um die Matrix-Vektor-Multiplikation erheblich effizienter durchzuführen: Die von James Cooley und John W. Tukey entwickelte *schnelle* FOURIER-*Transformation* (engl. *FFT*, *fast* FOURIER *transform*) verwendet einen Teile-und-herrsche-Ansatz (engl. *divide and conquer*, lateinisch *divide et impera*), um den Rechenaufwand drastisch zu reduzieren.

Das entscheidende Hilfsmittel sind dabei die besonderen Eigenschaften der *Einheitswurzel.*

Lemma 8.1 (Einheitswurzel) *Sei $n \in \mathbb{N}$. Die Zahl $\omega_n := \exp(2\pi\iota/n)$ nennen wir die* n-te Einheitswurzel*.*

Sie erfüllt die Gleichungen

$$\omega_n^n = 1, \qquad 1/\omega_n = \bar{\omega}_n. \tag{8.7a}$$

Falls $n = 2m$ für ein $m \in \mathbb{N}$ gilt, haben wir auch

$$\omega_n^m = -1. \tag{8.7b}$$

Beweis. Alle Gleichungen lassen sich aus der Euler-Gleichung (8.5) und den Rechenregeln für Potenzen gewinnen: Wir haben

$$\omega_n^n = \exp(2\pi\iota/n)^n = \exp(2\pi\iota n/n) = \exp(2\pi\iota) = \cos(2\pi) + \iota\sin(2\pi) = 1,$$
$$1/\omega_n = \frac{\bar{\omega}_n}{\omega_n\bar{\omega}_n} = \frac{\bar{\omega}_n}{|\omega_n|^2} = \frac{\bar{\omega}_n}{\cos^2(2\pi/n) + \sin^2(2\pi/n)} = \bar{\omega}_n.$$

Falls $n = 2m$ gilt, erhalten wir

$$\omega_n^m = \exp(2\pi\iota/n)^m = \exp(2\pi\iota m/n) = \exp(\pi\iota) = \cos(\pi) + \iota\sin(\pi) = -1.$$

■

Mit den Rechenregeln für Potenzen gilt $q_{n,ij} = \omega_n^{ij}$. Die Idee des Algorithmus von Cooley und Tukey beruht darauf, die Summe (8.6) in geradzahlige und ungeradzahlige Anteile zu zerlegen. Wir nehmen dazu an, dass $n = 2m$ gilt, so dass jede gerade Zahl $j \in Z_n$ sich in der Form $j = 2k$ mit $k \in Z_m$ darstellen lässt, während ungerade Zahlen die Form $j = 2k + 1$ annehmen. Es folgt

$$\begin{aligned} f_i &= \sum_{j\in Z_n} \omega_n^{ij}\hat{f}_j = \sum_{k\in Z_m} \omega_n^{2ik}\hat{f}_{2k} + \sum_{k\in Z_m} \omega_n^{i(2k+1)}\hat{f}_{2k+1} \\ &= \sum_{k\in Z_m} \omega_m^{ik}\hat{f}_{2k} + \omega_n^i \sum_{k\in Z_m} \omega_n^{2ij}\hat{f}_{2k+1} = \sum_{k\in Z_m} \omega_m^{ik}\hat{f}_{2k} + \omega_n^i \sum_{k\in Z_m} \omega_m^{ij}\hat{f}_{2k+1}. \end{aligned}$$

Wir beobachten, dass auf der rechten Seite zwei Summen auftreten, die dieselbe Struktur wie die ursprüngliche aufweisen, nur mit m Summanden und der m-ten Einheitswurzel statt der n-ten.

Indem wir Hilfsvektoren

$$\hat{x} := \begin{pmatrix} \hat{f}_0 \\ \hat{f}_2 \\ \vdots \\ \hat{f}_{n-2} \end{pmatrix}, \qquad \hat{y} := \begin{pmatrix} \hat{f}_1 \\ \hat{f}_3 \\ \vdots \\ \hat{f}_{n-1} \end{pmatrix}$$

und $x := Q_m\hat{x}$ sowie $y := Q_m\hat{y}$ einführen, erhalten wir

$$f_i = x_i + \omega_n^i y_i \qquad \text{für alle } i \in Z_m.$$

Für $i \geq m$ können wir die Gleichung nicht unmittelbar anwenden, weil x und y lediglich m-dimensionale Vektoren sind. Mit (8.7a) erhalten wir allerdings $\omega_m^{mk} = 1$ für alle $k \in \mathbb{N}_0$, also auch

$$\begin{aligned} f_i &= \sum_{k\in Z_m} \omega_m^{ik}\hat{x}_k + \omega_n^i \sum_{k\in Z_m} \omega_m^{ik}\hat{y}_k = \sum_{k\in Z_m} \omega_m^{(i-m)k}\hat{x}_k + \omega_n^i \sum_{k\in Z_m} \omega_m^{(i-m)k}\hat{y}_k \\ &= x_{i-m} + \omega_n^i y_{i-m} \qquad \text{für alle } i \in Z_n \setminus Z_m. \end{aligned}$$

Diese Formel lässt sich noch etwas verschönern, indem wir (8.7b) verwenden, um zu

$$f_i = x_{i-m} + \omega_n^i y_{i-m} = x_{i-m} - \omega_n^{i-m} y_{i-m} \qquad \text{für alle } i \in Z_n \setminus Z_m$$

zu gelangen. Der resultierende rekursive Algorithmus findet sich in Abbildung 8.1.

```
procedure ifft(n, ω_n, f̂, var f);
if n ≥ 2 then
  m ← n/2;    ω_m ← ω_n^2;
  for k = 0 to m − 1 do begin
    x̂_k ← f̂_{2k};    ŷ_k ← f̂_{2k+1}
  end;
  ifft(m, ω_m, x̂, x);
  ifft(m, ω_m, ŷ, y);
  α ← 1;
  for i = 0 to m − 1 do begin
    f_i ← x_i + αy_i;
    f_{i+m} ← x_i − αy_i;
    α ← ω_n α
  end
end
```

Abbildung 8.1: Schnelle FOURIER-Synthese $f = Q_n \hat{f}$

Bemerkung 8.2 (Komplexe Zahlen in C) *Die Verwendung komplexer Zahlen in C ist besonders einfach, falls der verwendete Compiler den ISO-Standard ISO/IEC 9899:1999 (kurz C99) unterstützt. In diesem Fall definiert die Headerdatei complex.h die Datentypen* `complex float` *und* `complex double` *für komplexe Gleitkommazahlen einfacher und doppelter Genauigkeit.*

Die komplexe Einheit ι ist über die Konstante `I` *erreichbar, die komplexe Exponentialfunktion über die Funktionen* `cexpf` *(für* `complex float`*) und* `cexp` *(für* `complex double`*).*

Grundrechenarten wie Addition, Subtraktion, Multiplikation und Division werden mit den in C üblichen Operatoren ausgeführt.

Natürlich wollen wir noch zeigen, dass dieser Algorithmus es verdient, als „schnell“ bezeichnet zu werden. Wir dürfen dabei davon ausgehen, dass die Addition zweier komplexer Zahlen nach (8.2a) zwei Gleitkommaoperationen erfordert, während für die Multiplikation nach (8.2b) sechs Operationen anfallen.

Wir gehen davon aus, dass $n \in \mathbb{N}$ eine Zweierpotenz ist, dass also $n = 2^p$ für ein $p \in \mathbb{N}_0$ gilt. Der Rechenaufwand, der für die Multiplikation eines Vektors mit der Matrix Q_n mittels unseres Verfahrens anfällt, sei mit $R(n)$ bezeichnet.

Im ersten Schritt des Verfahrens werden die Vektoren y und z angelegt. Dabei fallen keine Gleitkommaoperationen an.

Im zweiten Schritt wird Q_m mit den beiden Vektoren multipliziert, so dass $2R(m) = 2R(n/2)$ Operationen anfallen.

Im dritten Schritt wird der Vektor $\widehat{x}$ konstruiert. Wenn wir das Ergebnis der Multiplikation αy_i zwischenspeichern, fallen pro $i \in Z_m$ genau 16 Operationen an, nämlich

sechs für die Berechnung von αy_i, vier für die Summe und die Differenz, und sechs für die Berechnung von $\omega_n \alpha$.

Die Berechnung der Einheitswurzel $\omega_m = \omega_n^2$ erfordert sechs Operationen, so dass wir insgesamt die Rekursionsgleichung

$$R(n) = \begin{cases} 2R(n/2) + 16n + 6 & \text{falls } n > 1, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } n = 2^p,\ p \in \mathbb{N}_0.$$

Um auf eine Idee für die Lösung dieser Gleichung zu kommen, tabellieren wir ihre Werte für die ersten fünf Zweierpotenzen: Es gilt

$$\begin{aligned} R(1) &= 0, \\ R(2) &= 2R(1) + 32 + 6 = 32 + 6 = 16 \times 1 \times n + 6(n-1), \\ R(4) &= 2R(2) + 64 + 6 = 128 + 18 = 16 \times 2 \times n + 6(n-1), \\ R(8) &= 2R(4) + 128 + 6 = 384 + 42 = 16 \times 3 \times n + 6(n-1), \\ R(16) &= 2R(8) + 256 + 6 = 1024 + 90 = 16 \times 4 \times n + 6(n-1), \end{aligned}$$

und diese Gleichungen legen die Vermutung nahe, dass

$$R(n) = 16pn + 6(n-1) \qquad \text{für alle } n = 2^p,\ p \in \mathbb{N}_0$$

gilt. Wir können den Beweis per Induktion führen: Es gilt $0 = R(1) = 16 \times 0 \times n$. Sei nun $n = 2^p$ mit $p \in \mathbb{N}_0$ so gegeben, dass $R(n) = 16pn + 6(n-1)$ gilt. Dann folgt

$$\begin{aligned} R(2^{p+1}) &= R(2n) = 2R(n) + 16(2n) + 6 = 2 \times (16pn + 6(n-1)) + 16 \times 2 \times n + 6 \\ &= 16p(2n) + 6(2n-2) + 16(2n) + 6 = 16(p+1)(2n) + 6(2n-1) \\ &= 16(p+1)2^{p+1} + 6(2^{p+1} - 1). \end{aligned}$$

Insgesamt dürfen wir also festhalten, dass der Rechenaufwand der schnellen FOURIER-Transformation von der Größenordnung $n \log_2(n)$ ist, und damit für großes n *erheblich* geringer als für den naiven Ansatz.

Bemerkung 8.3 (Bit reversal) *Der FFT-Algorithmus kann ohne größeren Hilfsspeicher auskommen, wenn wir dazu bereit sind, die Koeffizienten des Vektors f in einer anderen als der mathematisch naheliegenden Anordnung zu erhalten.*

Dazu verwenden wir die Inkremente, die wir bereits im Kontext der BLAS-Vektoren kennen gelernt haben: Wenn $\hat{f}$ in einem Array ab `f` *mit einem Inkrement von* `incf` *gespeichert ist, ist $\hat{x}$ in einem Array ab* `x=f` *mit einem Inkrement von* `2*incf` *gespeichert, während sich $\hat{y}$ in einem Array ab* `y=f+1` *mit demselben Inkrement findet. Statt $\hat{f}$ zu kopieren, können wir also einfach Zeiger setzen und das Inkrement verdoppeln.*

Mit dem Ergebnis f wollen wir dann den zuvor von $\hat{f}$ genutzten Speicher überschreiben. Da f_i und f_{i+m} von x_i und y_i abhängen, überschreiben wir den zuvor von x_i genutzten Speicher mit f_i und den von y_i genutzten Speicher mit f_{i+m} (zumindest einer der alten Werte muss dabei zwischengespeichert werden).

Da x und y im Speicher geschachtelt dargestellt sind, steht anschließend f_i "direkt neben" f_{i+m} im Speicher, obwohl es mathematisch einen Abstand von m aufweisen sollte.

Bei $n = 8$ beispielsweise stehen

$$f_0, f_4, f_2, f_6, f_1, f_5, f_3, f_7$$

hintereinander im Speicher. Die modifizierte Numerierung ergibt sich per bit reversal*: Die Reihenfolge der Ziffern der Binärdarstellung dreht sich um, aus $1 = 001_2$ wird $4 = 100_2$ und aus $6 = 110_2$ wird $3 = 011_2$.*

Falls die Anordnung des Ergebnisvektors für unsere Anwendung ohne Belang ist, können wir mit dieser Vorgehensweise den Speicherbedarf halbieren. Da das bit reversal *selbstinvers ist, können wir alternativ auch $\hat{f}$ in der neuen Anordnung darstellen und erhalten dann f in der mathematisch üblichen.*

8.2 Fourier-Analyse

Die Fourier-Synthese konstruiert aus gegebenen Frequenzanteilen $\hat{f}$ einen Vektor $f = Q_n\hat{f}$, in dem die betreffenden Frequenzen mit den gegebenen Anteilen auftreten.

In der Praxis tritt häufig auch die Fourier-Analyse auf, bei der aus dem Vektor f die Frequenzanteile $\hat{f}$ gewonnen werden müssen.

Dabei ist die Hermitesche Matrix[3] nützlich, das komplexe Gegenstück der transponierten Matrix.

Definition 8.4 (Hermitesche Matrix) *Sei $A \in \mathbb{C}^{n\times m}$. Die durch*

$$b_{ij} = \bar{a}_{ji} \qquad \text{für alle } i \in [1:m],\ j \in [1:n]$$

definierte Matrix $B \in \mathbb{C}^{m\times n}$ nennen wir die Hermitesche Matrix *zu A. Wir bezeichnen sie mit A^H.*

Für $A \in \mathbb{C}^{Z_n\times Z_n}$ ist die Hermitesche Matrix A^H analog definiert.

Im Fall der Fourier-Matrizen Q_n besitzt die Hermitesche Matrix die folgende besonders nützliche Eigenschaft.

Lemma 8.5 (Fourier-Inverse) *Sei $n \in \mathbb{N}$, und sei $Q_n \in \mathbb{C}^{Z_n\times Z_n}$ die Fourier-Matrix. Dann gilt*

$$Q_n^H Q_n = nI,$$

also insbesondere $Q_n^{-1} = \frac{1}{n} Q_n^H$.

Beweis. Seien $i, j \in Z_n$ gegeben. Aufgrund der Gleichungen (8.7a) und (8.4a) gilt

$$(Q_n^H Q_n)_{ij} = \sum_{k\in Z_n} \bar{q}_{n,ki} q_{n,kj} = \sum_{k\in Z_n} \bar{\omega}_n^{ki} \omega_n^{kj} = \sum_{k\in Z_n} \omega_n^{-ki} \omega_n^{kj} = \sum_{k\in Z_n} \omega_n^{k(j-i)}.$$

[3] Benannt nach Charles Hermite.

Mit der Definition $q := \omega_n^{j-i}$ können wir diese Gleichung kurz als

$$(Q_n^H Q_n)_{ij} = \sum_{k \in Z_n} q^k$$

schreiben. Falls $j = i$ gilt, haben wir $q = 1$, also $(Q_n^H Q_n)_{ij} = n$. Anderenfalls gilt $q \neq 1$, da $j - i = \pm n$ wegen $i, j \in Z_n$ ausgeschlossen ist, so dass wir mit der geometrischen Summenformel und (8.7a) die Gleichung

$$(Q_n^H Q_n)_{ij} = \sum_{k \in Z_n} q^k = \frac{q^n - 1}{q - 1} = \frac{\omega_n^{(j-i)n} - 1}{q - 1} = 0$$

erhalten.

Damit ist $Q_n^H Q_n = nI$ bewiesen, also ist $\frac{1}{n} Q_n^H$ eine Linksinverse der Matrix Q_n, die somit injektiv sein muss. Da die Matrix quadratisch ist, folgt mit dem Dimensionssatz, dass sie auch surjektiv sein muss, also existiert für jeden Vektor $x \in \mathbb{C}^{Z_n}$ ein Urbild $y \in \mathbb{C}^{Z_n}$ mit $x = Q_n y$. Es folgt

$$Q_n Q_n^H x = Q_n Q_n^H Q_n y = n Q_n y = nx,$$

also $Q_n Q_n^H = nI$. Damit ist $\frac{1}{n} Q_n^H$ auch eine Rechtsinverse, also die Inverse. ■

Um aus einem Vektor f die Frequenzanteile $\hat{f}$ zu rekonstruieren, brauchen wir also nur mit Q_n^H zu multiplizieren und durch n zu dividieren, die Fourier-Analyse wird durch die Gleichung

$$\hat{f} = \tfrac{1}{n} Q_n^H f$$

beschrieben. Die Multiplikation mit der Hermiteschen Matrix lässt sich analog zu der Fourier-Synthese rekursiv bewerkstelligen: Es gilt

$$(Q_n^H)_{ij} = \bar{q}_{n,ji} = \bar{\omega}_n^{ji} = \bar{\omega}_n^{ij} \qquad \text{für alle } i, j \in Z_n,$$

so dass wir den in Abbildung 8.1 dargestellen Algorithmus lediglich mit $\bar{\omega}_n$ anstelle von ω_n aufrufen können, um die Fourier-Analyse effizient durchzuführen.

8.3 Zirkulante Matrizen

Eine wichtige Anwendung der schnellen Fourier-Transformation ist die Multiplikation eines Vektors mit einer *zirkulanten* Matrix.

Zur Vereinfachung der Notation definieren wir

$$i \oplus j := (i + j) \bmod n, \quad i \ominus j := i \oplus (n - j) = (i - j) \bmod n \qquad \text{für alle } i, j \in Z_n.$$

Definition 8.6 (Zirkulante Matrix) *Sei $A \in \mathbb{C}^{Z_n \times Z_n}$. Falls*

$$a_{i \oplus k, j \oplus k} = a_{ij} \qquad \text{für alle } i, j, k \in Z_n$$

gilt, nennen wir A zirkulant.

Eine zirkulante Matrix ist durch ihre erste Zeile bereits eindeutig festgelegt: Wenn wir

$$\alpha_j := a_{0j} \qquad \text{für alle } j \in Z_n$$

setzen, erhalten wir nach Definition

$$a_{ij} = a_{i\ominus i, j\ominus i} = a_{0,j\ominus i} = \alpha_{j\ominus i} \qquad \text{für alle } i,j \in Z_n.$$

Zirkulante Matrizen sind also immer von der Gestalt

$$A = \begin{pmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_{n-2} & \alpha_{n-1} \\ \alpha_{n-1} & \alpha_0 & \alpha_1 & \cdots & \alpha_{n-2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \alpha_2 & \cdots & \alpha_{n-1} & \alpha_0 & \alpha_1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_{n-1} & \alpha_0 \end{pmatrix}. \tag{8.8}$$

Unser Ziel ist es nun, eine solche Matrix mit einem Vektor $x \in \mathbb{C}^{Z_n}$ zu multiplizieren. Da A durch die n Koeffizienten $\alpha_0, \dots, \alpha_{n-1}$ beschrieben ist, hätten wir natürlich gerne einen Algorithmus, der diese Aufgabe in weniger als n^2 Operationen löst. Dabei kann uns die schnelle FOURIER-Transformation helfen.

Lemma 8.7 (Diagonalisierung) *Sei $A \in \mathbb{C}^{Z_n \times Z_n}$ eine zirkulante Matrix in der Darstellung (8.8). Sei $\beta := Q_n\alpha$, und sei*

$$D := \begin{pmatrix} \beta_0 & & \\ & \ddots & \\ & & \beta_{n-1} \end{pmatrix}.$$

Dann gilt

$$AQ_n = Q_nD.$$

Beweis. Seien $i,j \in Z_n$. Es gilt

$$(AQ_n)_{ij} = \sum_{k\in Z_n} a_{ik} q_{n,kj} = \sum_{k\in Z_n} \alpha_{k\ominus i} q_{n,kj}.$$

Wir substituieren $\ell = k \ominus i$, $k = \ell \oplus i$, und erhalten mit (8.7a) die Gleichung

$$\begin{aligned} (AQ_n)_{ij} &= \sum_{\ell\in Z_n} \alpha_\ell q_{n,\ell\oplus i,j} = \sum_{\ell\in Z_n} \alpha_\ell \omega_n^{(\ell\oplus i)j} \\ &= \sum_{\ell\in Z_n} \alpha_\ell \omega_n^{(\ell+i)j} = \omega_n^{ij} \sum_{\ell\in Z_n} \alpha_\ell \omega_n^{\ell j} \\ &= q_{n,ij} \sum_{\ell\in Z_n} \alpha_\ell q_{n,j\ell} = q_{n,ij}(Q_n\alpha)_j = q_{n,ij}\beta_j = (Q_nD)_{ij}. \end{aligned}$$

Das ist bereits unsere Behauptung. ∎

Damit steht uns ein effizienter Algorithmus für die Multiplikation eines Vektors $x \in \mathbb{C}^{Z_n}$ mit der Matrix A zur Verfügung: Aus dem vorigen Lemma folgt mit Lemma 8.5

$$A = \frac{1}{n} Q_n D Q_n^H,$$

also können wir $y = Ax$ mit den Zwischenschritten

$$\beta = Q_n \alpha, \qquad \widehat{x} := Q_n^H x, \qquad \widehat{y} := D\widehat{x}, \qquad y := \frac{1}{n} Q_n \widehat{y}$$

bestimmen. Alle Schritte außer dem dritten benötigen $\sim n \log_2(n)$ Operationen, der dritte lediglich $\sim n$, so dass wir insgesamt einen Rechenaufwand der Größenordnung $n \log_2(n)$ erhalten.

8.4 Toeplitz-Matrizen

Zirkulante Matrizen treten in der Praxis durchaus auf, allerdings nicht sehr häufig. Wesentlich häufiger begegnet man TOEPLITZ-Matrizen.

Definition 8.8 (Toeplitz-Matrix) *Sei $A \in \mathbb{C}^{Z_n \times Z_n}$ eine Matrix. Falls es einen Vektor $\gamma \in \mathbb{C}^{[1-n:n-1]}$ mit*

$$a_{ij} = \gamma_{j-i} \qquad \text{für alle } i, j \in Z_n$$

gibt, nennen wir A eine TOEPLITZ-Matrix[4].

Eine TOEPLITZ-Matrix A besitzt dementsprechend die Darstellung

$$A = \begin{pmatrix} \gamma_0 & \gamma_1 & \cdots & \gamma_{n-2} & \gamma_{n-1} \\ \gamma_{-1} & \gamma_0 & \gamma_1 & \cdots & \gamma_{n-2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \gamma_{2-n} & \cdots & \gamma_{-1} & \gamma_0 & \gamma_1 \\ \gamma_{1-n} & \gamma_{2-n} & \cdots & \gamma_{-1} & \gamma_0 \end{pmatrix}. \tag{8.9}$$

Bedauerlicherweise lassen sich TOEPLITZ-Matrizen nicht unmittelbar mit der FOURIER-Transformation diagonalisieren. Allerdings können wir TOEPLITZ-Matrizen zu *zirkulanten* Matrizen erweitern: Wir definieren

$$\alpha := \begin{pmatrix} \gamma_0 \\ \vdots \\ \gamma_{n-1} \\ \gamma_{1-n} \\ \vdots \\ \gamma_{-1} \end{pmatrix} \in \mathbb{C}^{Z_{2n+1}}$$

[4] Benannt nach Otto Toeplitz.

und erhalten die zirkulante Matrix

$$\widehat{A} = \left(\begin{array}{cccc|ccc} \gamma_0 & \cdots & \cdots & \gamma_{n-1} & \gamma_{1-n} & \cdots & \gamma_{-1} \\ \gamma_{-1} & \gamma_0 & \cdots & \gamma_{n-2} & \gamma_{n-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \gamma_{1-n} \\ \gamma_{1-n} & \cdots & \gamma_{-1} & \gamma_0 & \gamma_1 & \cdots & \gamma_{n-1} \\ \hline \gamma_{n-1} & \cdots & \cdots & \gamma_{-1} & \gamma_0 & \cdots & \gamma_{n-2} \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \gamma_1 & \cdots & \gamma_{n-1} & \gamma_{1-n} & \gamma_{2-n} & \cdots & \gamma_0 \end{array}\right),$$

deren linker oberer Block mit A übereinstimmt.

Um A nun effizient mit einem Vektor $x \in \mathbb{C}^{Z_n}$ zu multiplizieren, ergänzen wir A zu der Matrix $\widehat{A}$ und x zu dem Vektor

$$\widetilde{x} = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}.$$

Nun berechnen wir $\widetilde{y} = \widehat{A}\widetilde{x}$ wie zuvor mit der schnellen Fourier-Transformation.

Da die letzten $n-1$ Komponenten des Vektors $\widetilde{x}$ gleich null sind, spielen die letzten $n-1$ Spalten der Matrix $\widehat{A}$ keine Rolle für das Ergebnis, da sie mit null multipliziert werden.

Demzufolge findet sich in den ersten n Komponenten des Vektors $\widetilde{y}$ das gesuchte Ergebnis $y = Ax$.

Wir müssen für die Behandlung einer Toeplitz-Matrix zwar zu einer erweiterten Matrix ungefähr doppelter Dimension übergehen, können dann aber wieder in einem zu $n \log_2(n)$ proportionalen Rechenaufwand die Matrix-Vektor-Multiplikation durchführen.

Index

BLAS, 15
    Level 1, 16
    Level 2, 17
    Level 3, 17
Blockmatrizen, 9

Cache-Speicher, 23
Callback-Funktion, 50
Callback-Funktionen, 86
Cauchy-Schwarz-Ungleichung, 55

Dreiecksmatrix, 7
    normiert, 13

Eigenvektor, 45
Eigenwert, 45
Einheitswurzel, 127
Euklidisches Skalarprodukt, 39
Euler-Verfahren, 72

FFT, 128
    bit reversal, 130

Gleitkommazahlen, 33
Gouraud-Shading, 95
Gradient, 61
Gradientenverfahren, 62

Hauptachsentransformation, 53
Hermitesche Matrix, 131
Hesse-Matrix, 61
Homogene Koordinaten, 90
Hookesches Gesetz, 71
Horner-Schema, 105
Householder-Spiegelung
    Matrix, Householder, 41

induzierte Matrixnorm, 36
Interpolation, 107
inverse Iteration, 48
isometrische Matrix, 39
Iterationsverfahren, 46

komplexe Zahlen, 125
Konditionszahl, 38
Krylow-Raum, 67

Lagrange-Interpolation, 107
Lagrange-Polynome, 107
Leapfrog-Verfahren, 74
LR-Zerlegung, 8
    BLAS-Implementierung, 20
    Konstruktion, 12

Maschinenzahlen, 33
Matrix
    Dreiecksmatrix, 7
    isometrisch, 39
    orthogonal, 39
    transponiert, 39
    zirkulant, 132
Mittelpunktregel, 118

Neumannsche Reihe, 37
Newton-Cotes-Regeln, 118
Newton-Iteration, 57
Newton-Polynome, 109
Newtonsches Gesetz, 71
Normen, 35

orthogonale Matrix, 39

Phong-Beleuchtung, 96

QR-Zerlegung, 40
Quadraturformel, 116
  interpolatorisch, 117
  summiert, 122
  transformiert, 116
Quadratwurzel, 58

Rayleigh-Iteration, 49
Rayleigh-Quotient, 48
Rechenaufwand
  LR-Zerlegung, 14
  Rückwärtseinsetzen, 11
  Vorwärtseinsetzen, 11
Referenzintervall, 115
Residuum, 65
Rückwärtseinsetzen, 10
  BLAS-Implementierung, 22
Rundungsfehler, 34
RUNGE-Verfahren, 82

Shift-Parameter, 48
Skalarprodukt, 39
Summierte Quadraturformel, 122

TOEPLITZ-Matrix, 134
transponierte Matrix, 39
Trapezregel, 118
TSCHEBYSCHEFF-Interpolation, 113

Vektoriteration, 46
Vorwärtseinsetzen, 9
  BLAS-Implementierung, 21

Zirkulante Matrix, 132

Literaturverzeichnis

[1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16(1):1–17, 1990.

[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM TOMS*, 14(1):1–17, 1988.

[3] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM TOMS*, 5(3):308–323, 1979.