Programmierung numerischer Algorithmen

Steffen Börm

Stand 4. Juli 2016, 23 Uhr 33

Inhaltsverzeichnis

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 genähert 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 jede beliebige Genauigkeit erreichen.

In der Praxis müssen häufig große Datenmengen verarbeitet werden, beispielsweise sind bei einer Strömungssimulationen 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 für die Suche in einer Datenstruktur 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 Prozessoeren vorhandenen schnellen Hilfsspeicher geschickt einsetzen und möglichst viele Operationen pro aus dem Speicher geholten 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 für Korrekturen und Verbesserungsvorschläge.

2 Numerische Simulationen

Schon die ersten Computer wurden für die Behandlung naturwissenschaftlicher Fragestellungen verwendet, beispielsweise um den Lauf der Gestirne vorherzusagen oder Flugbahnen zu berechnen.

Auch heute sind derartige Anwendungen von großer Bedeutung, häufig in Form *numerischer Simulationen*, bei denen man ein in der Realität auftretendes Phänomen durch ein mathematisches Modell beschreibt und das Modell analysiert. Da sich die Modelle in den meisten Fällen nicht in geschlossener Form lösen lassen, kommt die *numerische Mathematik* zum Einsatz, die sich mit der Frage beschäftigt, wie sich hinreichend genaue Näherungslösungen effizient berechnen lassen.

2.1 Geschwindigkeit, Beschleunigung, Kraft

Ein erstes Beispiel für eine numerische Simulation finden wir im Bereich der *Kinetik*, einem Teilgebiet der Mechanik, das sich mit bewegten Körpern beschäftigt.

Der Einfachheit halber beschränken wir uns auf *Punktmassen*, also auf Körper, deren gesamte Masse in einem einzigen Punkt im Raum konzentriert ist. Solche Punktmassen sind rein theoretische Konstrukte, die aber in vielen Anwendungsfällen entweder gute Näherungen der tatsächlichen Gegebenheiten darstellen (beispielsweise ist ein Satellit im Verhältnis zum ihn umgebenden Weltraum praktisch ein Punkt) oder aus denen sich kompliziertere Näherungen einfach konstruieren lassen.

Da wir an *bewegten* Punktmassen interessiert sind, schreiben wir die *Position* der Punktmasse als eine Funktion

$$x : \mathbb{R} \to \mathbb{R}^3,$$

die jedem Zeitpunkt $t \in \mathbb{R}$ die Position $x(t)$ im dreidimensionalen Raum zuordnet, an der sich die Punktmasse zu diesem Zeitpunkt befindet.

Bewegungen lassen sich durch ihre *Geschwindigkeit* charakterisieren, also durch die Veränderung der Position in Abhängigkeit von der Zeit: Für zwei Zeitpunkte $a < b$ bezeichnen wir den Quotienten

$$\frac{x(b) - x(a)}{b - a}$$

als die *mittlere Geschwindigkeit* der Punktmasse zwischen a und b. Da die Arbeit mit mittleren Geschwindigkeiten häufig etwas umständlich ist, lässt man die Differenz der Zeitpunkte gegen null streben und bezeichnet die *Momentangeschwindigkeit* zu einem Zeitpunkt $t \in \mathbb{R}$ mit

$$x'(t) := \lim_{h \to 0} \frac{x(t+h) - x(t)}{h},$$

falls dieser Grenzwert existiert. Als Abkürzung für die (Momentan-)Geschwindigkeit führen wir die Funktion

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

ein, wobei der Buchstabe v durch das englische Wort *velocity* motiviert ist.

Wäre die Geschwindigkeit konstant, würde sich die Punktmasse gleichmäßig entlang einer Gerade bewegen. Sehr viel interessanter sind Bewegungen, die von so einer Gerade abweichen. Sie lassen sich elegant durch die *Beschleunigung* beschreiben, die analog zu der Momentangeschwindigkeit als

$$v'(t) := \lim_{h \to 0} \frac{v(t+h) - v(t)}{h},$$

also als die relative Veränderung der Geschwindigkeit über kurzen Zeiträumen definiert ist. Als Abkürzung verwenden wir

$$a(t) := v'(t) \qquad \text{für alle } t \in \mathbb{R},$$

wobei der Buchstabe a auf das englische Wort *acceleration* hindeuten soll.

Nach Isaac Newton entstehen Beschleunigungen durch Kräfte, die auf die Punktmasse wirken. Die Kräfte beschreiben wir durch eine Funktion

$$F : \mathbb{R} \to \mathbb{R}^3,$$

die jedem Zeitpunkt $t \in \mathbb{R}$ die wirkende Kraft $F(t)$ zuordnet (F steht für *force*). Wenn wir die Masse, auf die die Kraft wirkt, mit m bezeichnen, erhalten wir das Newton-Axiom

$$ma(t) = F(t) \qquad \text{für alle } t \in \mathbb{R}.$$

Insgesamt erhalten wir die folgenden Gleichungen:

$$x'(t) = v(t), \qquad v'(t) = a(t), \qquad ma(t) = F(t) \qquad \text{für alle } t \in \mathbb{R}. \tag{2.1}$$

Wenn wir die zu jedem Zeitpunkt wirkenden Kräfte sowie Position und Geschwindigkeit zu einem Anfangszeitpunkt $t = 0$ kennen, können wir die Bewegung der Punktmasse vorhersagen.

Ein einfaches Beispiel ist eine Punktmasse, die in der Nähe der Erdoberfläche geworfen wird. In diesem Fall können wir die Erdanziehungskraft als konstant setzen, etwa durch die Gleichung

$$F(t) = m \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} \qquad \text{für alle } t \in \mathbb{R},$$

wobei $g \approx 9,81$ die *Erdbeschleunigung* angibt und wir davon ausgehen, dass der Erdmittelpunkt in Richtung der negativen y-Achse liegt. Es folgt direkt

$$a(t) = \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} \qquad \text{für alle } t \in \mathbb{R}.$$

Um die Geschwindigkeit aus der Beschleunigung zu gewinnen, müssen wir die Gleichung $v'(t) = a(t)$ lösen, also eine *Stammfunktion* finden. Da a konstant ist, ist das eine einfache Aufgabe mit der Lösung

$$v(t) = v_0 + \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} t \qquad \text{für alle } t \in \mathbb{R},$$

bei der $v_0 = v(0)$ die Geschwindigkeit zum Anfangszeitpunkt $t = 0$ angibt. Um schließlich die Position unseres Flugkörpers zu einem bestimmten Zeitpunkt zu ermitteln, müssen wir $x'(t) = v(t)$ lösen. Die Stammfunktion ist durch

$$x(t) = x_0 + v_0 t + \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} \frac{t^2}{2} \qquad \text{für alle } t \in \mathbb{R}$$

gegeben und erlaubt es uns, die Position zu jedem beliebigen Zeitpunkt $t \in \mathbb{R}$ zu berechnen. Hier ist $x_0 = x(0)$ die Position zum Anfangszeitpunkt $t = 0$.

In diesem Modell wirkt ausschließlich die Erdanziehungskraft auf den Flugkörper. In der Realität werden allerdings auch andere Kräfte wirken, beispielsweise wird der Körper durch den Luftwiderstand abgebremst. Bei nicht allzu hohen Geschwindigkeiten lässt sich dieser Widerstand durch die *Stokes-Reibung* modellieren, die als eine Bremskraft

$$F_{\mathrm{br}}(t) = -\varrho v(t) \qquad \text{für alle } t \in \mathbb{R}$$

mit einer von der Luftdichte (und normalerweise auch der Form des Flugkörpers) abhängenden Konstanten $\varrho \in \mathbb{R}_{\geq 0}$ auftritt und jeweils der Geschwindigkeit entgegenwirkt.

Diese Abhängigkeit der Kraft von der Geschwindigkeit macht das Lösen der Gleichung erheblich schwieriger, denn nun gilt

$$v'(t) = \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} - \frac{\varrho}{m} v(t) \qquad \text{für alle } t \in \mathbb{R},$$

eine *gewöhnliche Differentialgleichung*, bei der die Ableitung der Funktion v von der Funktion selbst abhängt.

Für viele gewöhnliche Differentialgleichungen lassen sich keine Lösungen in Form einer kurzen Gleichung angeben, deshalb kommen numerische Verfahren zum Einsatz, also Algorithmen, die eine beliebig genaue Näherungslösung berechnen.

Bemerkung 2.1 (Lösung) *In unserem Fall ist es noch möglich, eine kurze Formel für die Lösung anzugeben: Gewöhnliche Differentialgleichungen der Gestalt*

$$v'(t) = \alpha - \beta v(t) \qquad \textit{für alle } t \in \mathbb{R}$$

mit $\alpha, \beta \in \mathbb{R}$ und $\beta \neq 0$ können mit einem Ansatz der Form

$$v(t) = \frac{\alpha}{\beta} + c\exp(-\beta t) \qquad \text{für alle } t \in \mathbb{R}$$

gelöst werden, bei dem wir $c \in \mathbb{R}$ beliebig wählen dürfen. Dann gilt nämlich

$$v'(t) = -c\beta\exp(-\beta t) = \alpha - \beta\left(\frac{\alpha}{\beta} + c\exp(-\beta t)\right) = \alpha - \beta v(t) \qquad \text{für alle } t \in \mathbb{R}.$$

Falls ein Anfangswert v_0 zum Zeitpunkt $t = 0$ gegeben ist, können wir $c = v_0 - \alpha/\beta$ setzen und erhalten $v(0) = \alpha/\beta + c = v_0$.

Durch Integration der Gleichung für $v(t)$ ergibt sich die Stammfunktion

$$x(t) = x_0 + \frac{\alpha}{\beta}t - \frac{c}{\beta}\exp(-\beta t) \qquad \text{für alle } t \in \mathbb{R}.$$

2.2 Zeitschrittverfahren

Die Flugbahn im Gravitationsfeld der Erde mit Luftreibung haben wir durch die Gleichungen

$$x(0) = x_0, \qquad v(0) = v_0, \tag{2.2a}$$

$$x'(t) = v(t), \qquad v'(t) = \begin{pmatrix} 0 \\ -g \\ 0 \end{pmatrix} - \frac{\varrho}{m}v(t) \qquad \text{für alle } t \in \mathbb{R} \tag{2.2b}$$

beschrieben. Um uns Schreibarbeit zu sparen, fassen wir x und v zu Vektoren zusammen, setzen also

$$y_0 := \begin{pmatrix} x_0 \\ v_0 \end{pmatrix}, \qquad y(t) := \begin{pmatrix} x(t) \\ v(t) \end{pmatrix} \qquad \text{für alle } t \in \mathbb{R}.$$

Die Ableitungen fassen wir in einer Funktion

$$f(t, z) := \begin{pmatrix} z_4 \\ z_5 \\ z_6 \\ 0 - \beta z_4/m \\ -g - \beta z_5/m \\ -\beta z_6/m \end{pmatrix} \qquad \text{für alle } t \in \mathbb{R},\ z \in \mathbb{R}^6$$

zusammen und stellen fest, dass (2.2) äquivalent zu

$$y(0) = y_0, \qquad y'(t) = f(t, y(t)) \qquad \text{für alle } t \in \mathbb{R} \tag{2.3}$$

ist, so dass wir diese erheblich kürzere Gleichung analysieren können.

Nach Definition gilt

$$y'(t) = \lim_{h \to 0} \frac{y(t+h) - y(t)}{h},$$

also bietet es sich an, den auf der rechten Seite stehenden Quotienten für ein hinreichend kleines $h > 0$ als Approximation der Ableitung zu verwenden:

$$y'(t) \approx \frac{y(t+h) - y(t)}{h}.$$

Indem wir diese Gleichung nach $y(t+h)$ auflösen, erhalten wir

$$y(t+h) \approx y(t) + hy'(t),$$

und durch Einsetzen von (2.3) folgt

$$y(t+h) \approx y(t) + hf(t, y(t)).$$

Mit Hilfe dieser Formel können wir eine Näherung der Lösung zum Zeitpunkt $t+h$ gewinnen, falls uns die Lösung zum Zeitpunkt t bekannt ist.

Für $t = 0$ können wir dank (2.3) den Anfangswert $y(0) = y_0$ einsetzen und so eine Näherung

$$\tilde{y}(h) = y(0) + hf(0, y(0)) = y_0 + hf(0, y_0)$$

für den Zeitpunkt $t = h$ berechnen. Indem wir diese Näherung an die Stelle von $y(h)$ treten lassen, können wir eine Näherung

$$\tilde{y}(2h) = \tilde{y}(h) + hf(h, \tilde{y}(h))$$

für den Zeitpunkt $t = 2h$ finden. In dieser Weise können wir induktiv vorgehen und Näherungslösungen für die Zeitpunkte

$$t_i := hi \qquad \text{für alle } i \in \mathbb{N}_0$$

ermitteln.

Definition 2.2 (Explizites Euler-Verfahren) *Das* explizite Euler-Verfahren *berechnet mit der Vorschrift*

$$\begin{aligned} \tilde{y}(t_0) &= y_0, \\ \tilde{y}(t_{i+1}) &= \tilde{y}(t_i) + hf(t_i, \tilde{y}(t_i)) \qquad \textit{für alle } i \in \mathbb{N}_0 \end{aligned}$$

Näherungen der Lösung y in den Punkten t_i.

Das explizite Euler-Verfahren ist das einfachste Beispiel für ein *Zeitschrittverfahren*, bei dem Näherungen der Lösung einer gewöhnlichen Differentialgleichung zu bestimmten festen Zeitpunkten der Reihe nach berechnet werden.

Bemerkung 2.3 (Genauigkeit) *Selbstverständlich ist es wichtig, die Genauigkeit der Näherung abschätzen zu können, die ein Zeitschrittverfahren berechnet. Im Fall des Euler-Verfahrens können wir dazu die* Taylor-Entwicklung *heran ziehen: Falls y auf dem Intervall $[t, t+h]$ zweimal stetig differenzierbar ist, gilt nach dem Satz von Taylor*

$$y(t+h) = y(t) + hy'(t) + h^2 \int_0^1 (1-s)y''(t+sh)\,ds,$$
$$\frac{y(t+h)-y(t)}{h} = y'(t) + h\int_0^1 (1-s)y''(t+sh)\,ds,$$

und mit der durch

$$\|z\|_{\infty,[a,b]} := \max\{\|z(s)\| \ : \ s \in [a,b]\}$$

definierten Maximumnorm *können wir das Integral durch*

$$\left\| h\int_0^1 (1-s)y''(t+sh)\,ds \right\| \le h\int_0^1 (1-s)\|y''(t+sh)\|\,ds$$
$$\le h\|y''\|_{\infty,[t,t+h]} \int_0^1 (1-s)\,ds = \frac{h}{2}\|y''\|_{\infty,[t,t+h]}$$

abschätzen. Unter diesen Voraussetzungen dürfen wir also erwarten, dass der Fehler sich ungefähr proportional zu h verhalten wird.

Wenn wir den bei unserem Verfahren auftretenden Fehler halbieren wollen, müssen wir demnach die Schrittweite h halbieren, also doppelt so viele Zeitschritte wie zuvor ausführen. Viele numerische Algorithmen zeigen ein ähnliches Verhalten: Für eine höhere Genauigkeit müssen ein höherer Rechenaufwand und oft auch ein höherer Speicherbedarf in Kauf genommen werden.

Häufig ist es allerdings möglich, die „Kosten“ eines Verfahrens zu reduzieren, indem man besondere Eigenschaften des zu behandelnden Problems ausnutzt. In unserem Fall können wir beispielsweise davon profitieren, wenn auch die höheren Ableitungen der Lösung y stetig sind.

Unser Ausgangspunkt ist wieder eine Approximation der Ableitung, allerdings diesmal in der Form des *zentralen Differenzenquotienten*

$$y'(t+h/2) \approx \frac{y(t+h)-y(t)}{h}.$$

Wenn wir wie zuvor verfahren, erhalten wir

$$y(t+h) \approx y(t) + hy'(t+h/2) = y(t) + hf(t+h/2, y(t+h/2))$$

und stellen fest, dass wir diesen Ausdruck nicht praktisch auswerten können, da uns $y(t+h/2)$ nicht zur Verfügung steht. Deshalb behelfen wir uns mit dem bereits bekannten Euler-Verfahren, um die Näherung

$$\tilde{y}(t+h/2) := y(t) + \frac{h}{2}f(t, y(t))$$

zu ermitteln, die wir dann in die erste Formel einsetzen, um zu

$$y(t+h) \approx y(t) + hf(t+h/2, \tilde{y}(t+h/2))$$

zu gelangen. Damit erhalten wir ein neues Zeitschrittverfahren.

Definition 2.4 (Runge-Verfahren) *Das* Runge-Verfahren *berechnet mit der Vorschrift*

$$\begin{aligned} \tilde{y}(t_0) &= y_0, \\ \tilde{y}(t_{i+1/2}) &= \tilde{y}(t_i) + \frac{h}{2} f(t_i, \tilde{y}(t_i)), \\ \tilde{y}(t_{i+1}) &= \tilde{y}(t_i) + hf(t_{i+1/2}, \tilde{y}(t_{i+1/2})) \qquad \text{für alle } i \in \mathbb{N}_0 \end{aligned}$$

Näherungen der Lösung y in den Punkten t_i.

Bemerkung 2.5 (Genauigkeit) *Falls wir voraussetzen, dass die Lösung y dreimal stetig differenzierbar und die rechte Seite f Lipschitz-stetig im zweiten Argument ist, lässt sich zeigen, dass sich der Fehler des Runge-Verfahrens ungefähr proportional zu h^2 verhält.*

Die unterschiedliche Fehlerentwicklung des Euler- und des Runge-Verfahrens kann erhebliche Konsequenzen haben: Angenommen, beide Verfahren erreichen eine gewisse Genauigkeit $\epsilon > 0$, wenn sie m Schritte durchführen. Ein Schritt des Runge-Verfahrens erfordert doppelt so viele Rechenoperationen wie einer des Euler-Verfahrens, also kostet uns das Runge-Verfahren doppelt so viel Zeit wie das Euler-Verfahren.

Angenommen, die Genauigkeit ϵ reicht nicht, sondern wir brauchen eine Genauigkeit von $\epsilon/100$. Da sich der Fehler des Euler-Verfahrens proportional zur Schrittweite h verhält, muss es $100m$ Schritte durchführen, um diese Genauigkeit zu erreichen. Der Fehler des Runge-Verfahrens ist proportional zu h^2, also genügt es, $10m$ Schritte auszuführen. Obwohl ein Schritt des Runge-Verfahrens doppelt so lange dauert wie einer des Euler-Verfahrens, ist die gesamte Rechenzeit des Runge-Verfahrens deshalb in diesem Fall um den Faktor fünf geringer als die des Euler-Verfahrens.

3 Maschinenzahlen und die Kondition einer Berechnung

Es wurde bereits erwähnt, dass bei der Programmierung numerischer Algorithmen die Tatsache von besonderer Bedeutung ist, dass der Computer reelle Zahlen nur näherungsweise darstellen kann. Bei einer längeren Berechnung müssen insbesondere auch alle Zwischenergebnisse approximiert werden, so dass sich die dabei auftretenden Fehler ansammeln und im schlimmsten Fall das Ergebnis unbrauchbar machen können.

Deshalb muss bei der Entwicklung und Implementierung numerischer Algorithmen darauf geachtet werden, dass die Fehlerfortpflanzung beherrschbar bleibt.

3.1 Maschinenzahlen

Natürliche Zahlen werden im Computer — genau wie in der Schulmathematik — typischerweise durch ein *Stellenwertsystem* dargestellt: Bei einer Basis von b wird die Zahl durch eine Folge von Ziffern zwischen 0 und $b-1$ dargestellt, die jeweils mit aufsteigenden Potenzen von b multipliziert werden. Wenn wir die jeweilige Basis b in den Index setzen und die Ziffern in aufsteigender Reihenfolge von rechts nach links anordnen, erhalten wir die gewohnte Darstellung

$$\begin{aligned} 372_{10} &= 3 \times 10^2 + 7 \times 10^1 + 2 \times 10^0 = 372, \\ 101_2 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5. \end{aligned}$$

Mathematisch lässt sich also eine natürliche Zahl mit m Ziffern in der Form

$$x = \sum_{j=0}^{m-1} d_j b^j, \qquad d_j \in [0 : b-1] \text{ für alle } j \in [0 : m-1]$$

schreiben. Heute wird in der Regel das Binärsystem mit der Basis $b = 2$ verwendet, eine Ziffer wird dann als *Bit* bezeichnet und kann entweder den Wert 0 oder den Wert 1 annehmen.

Bemerkung 3.1 (BCD) *Um Fehler bei der kaufmännischen Rundung zu vermeiden, bieten manche Prozessoren auch dezimale Ziffern an, beispielsweise im x86-Befehlssatz in der BCD-Darstellung (für* binary coded decimal*). Die Handhabung ist allerdings recht umständlich und eher nur für spezielle Anwendungen im Bereich der Finanzmathematik interessant.*

Da ein Computer nur über endlich viel Speicher verfügt, müssen wir die Anzahl der Stellen beschränken. Üblich sind heute Zahlen mit 8, 16, 32 oder 64 Bit.

Für Zahlen mit Vorzeichen wird häufig die *Zweierkomplementdarstellung* verwendet: Wenn m Bits zur Verfügung stehen, werden nicht-negative Zahlen zwischen 0 und $2^{m-1}-1$ wie oben beschrieben dargestellt, das $(m-1)$-te Bit ist gleich null und zeigt damit ein positives Vorzeichen an. Eine Zahl wird negiert, indem man alle Ziffern invertiert und eins addiert.

0	1	1	0	0	1	1	1	103

1	1	0	0	0	0	1	1	–61

Bei einem Überlauf (engl. *overflow*) kann so aus einer positiven Zahl eine negative oder umgekehrt werden, beispielsweise gilt bei 16-Bit-Zahlen in Zweierkomplementdarstellung `32767 + 1 = -32768`.

Für die Darstellung reeller Zahlen verwendet man in der Regel die *Gleitkommadarstellung* nach dem Standard IEEE 754: Das Vorzeichen wird separat in einem Bit gespeichert. Eine positive Zahl wird so lange durch $b = 2$ dividiert oder mit 2 multipliziert, bis sie im Intervall $[1, 2)$ liegt. Das Ergebnis wird dann auf p Nachkommastellen gerundet. Die Vorkommastelle ist immer gleich eins und braucht deshalb nicht gespeichert zu werden.

Mathematisch lässt sich diese Darstellung in der Form

$$x = \sigma b^e \left(1 + \sum_{j=1}^{p} d_j 2^{-j}\right)$$

mit dem Vorzeichen $\sigma \in \{-1, 1\}$, einem Exponenten $e \in \mathbb{Z}$ und Ziffern $d_1, \ldots, d_p \in \{0, 1\}$ schreiben.

In der Praxis kann der Exponent $e \in \mathbb{Z}$ natürlich auch keine beliebig großen oder kleinen Werte annehmen. Er wird durch eine r-Bit-Zahl $\hat{e} \in [0 : 2^r - 1]$ mit $r \in \mathbb{N}$ dargestellt, aus der er sich mit der Formel

$$e = \hat{e} - (2^{r-1} - 1)$$

rekonstruieren lässt. Die Werte $\hat{e} = 0$ und $\hat{e} = 2^r - 1$ sind dabei für die Darstellung spezieller „Zahlen“ zuständig, etwa gehört $\hat{e} = 0$ zu der Null, die sich in der obigen Form nicht darstellen lässt, während $\hat{e} = 2^r - 1$ beispielsweise für die Darstellung von ∞ und $-\infty$ verwendet wird.

Bei der im IEEE-Standard definierten *einfachen Genauigkeit* werden $p = 23$ Bits für die Nachkommastellen, die *Mantisse*, und $r = 8$ Bits für den Exponenten verwendet, so dass mit dem Vorzeichen insgesamt 32 Bit benötigt werden.

Bei der Darstellung durch eine 32-Bit-Zahl gehören die niedrigsten Bits zu der Mantisse, die nächsthöheren zu dem Exponenten, und das höchste zu dem Vorzeichen.

In der Programmiersprache C können wir mit Hilfe eines `union`-Typs direkt auf die einzelnen Komponenten zugreifen:

```
union {
  float f;
  struct {
    uint32_t mantissa:23;
    uint8_t exponent:8;
    uint8_t sign:1;
  } b;
} x;

x.f = 2.75f;
printf("%06x %02x %1x\n",
       x.b.mantissa, x.b.exponent, x.b.sign);
```

In diesem Fall müssen wir einen Exponenten von $e = 1$ verwenden, um $2.75 = 1.375 \times 2^1$ zu erhalten, also gilt $\hat{e} = e + e^{r-1} - 1 = e + 127 = 128$.

In binärer Darstellung gilt $1.375 = 1.011_2$, also müssen in der Mantisse das zweit- und das dritthöchste Bit gesetzt sein, so dass wir $2^{21} + 2^{20} = 3 \times 2^{20}$ erhalten. Das Vorzeichenbit ist gleich null, da uns eine positive Zahl vorliegt.

Neben Gleitkommazahlen einfacher Genauigkeit definiert der IEEE-Standard auch Zahlen doppelter Genauigkeit, bei denen die Mantisse aus $p = 52$ Bits und der Exponent aus $r = 11$ Bits besteht, so dass wir inklusive Vorzeichen insgesamt 64 Bits benötigen.

```
union {
  double f;
  struct {
    uint64_t mantissa:52;
    uint16_t exponent:11;
    uint8_t sign:1;
  } b;
} x;

x.f = 2.75;
printf("%013" PRIx64 " %02x %1x\n",
       x.b.mantissa, x.b.exponent, x.b.sign);
```

Wenn wir die Zahl 2.75 in doppelter Genauigkeit darstellen möchten, erhalten wir $\hat{e} = e + e^{r-1} - 1 = e + 1023 = 1024$ und eine Mantisse von $2^{50} + 2^{49} = 6 \times 2^{48}$.

Da uns nur p Ziffern zur Verfügung stehen, muss bei der Umwandlung in die Gleitkommadarstellung gerundet werden. Mathematisch schreiben wir das Runden als eine Abbildung

$$\mathrm{fl} : \mathbb{R} \to \mathbb{R},$$

die einer beliebigen reellen Zahl eine Maschinenzahl zuordnet, im Idealfall natürlich mit möglichst geringem Abstand.

Die üblichen arithmetischen Operationen werden dann durch gerundete Operationen ersetzt:

$$\begin{aligned} x \oplus y &= \mathrm{fl}(x+y), \\ x \ominus y &= \mathrm{fl}(x-y), \\ x \odot y &= \mathrm{fl}(xy) && \text{für alle } x, y \in \mathbb{R}, \\ x \oslash y &= \mathrm{fl}(x/y) && \text{für alle } x \in \mathbb{R},\ y \in \mathbb{R} \setminus \{0\}. \end{aligned}$$

Diese Operationen verwendet der Computer, wenn wir ihm den Auftrag erteilen, eine Addition, Subtraktion, Multiplikation oder Division auszuführen. Moderne Prozessoren bieten auch die Operation *fused multiply-add* (FMA), bei der direkt $\mathrm{fl}(x+yz)$ für $x, y, z \in \mathbb{R}$ berechnet wird, ohne das Zwischenergebnis der Multiplikation zu runden.

Beispielsweise erhalten wir bei einfacher Genauigkeit für die Addition $1 \oplus 2^{-24}$ entweder 1 oder $1 + 2^{-23}$, abhängig davon, ob ab- oder aufgerundet wird. Wenn wir nun 1 subtrahieren, ergibt sich entweder null oder 2^{-23}, der absolute Fehler beträgt also 2^{-24}, der relative demnach 100 Prozent. Dieser Effekt wird als *Auslöschung* bezeichnet: Die 23 gespeicherten Bits löschen sich bei der Berechnung der Differenz gegenseitig aus, so dass nur eine sehr geringe relative Genauigkeit erreicht wird. In der Praxis wird der Begriff „Auslöschung“ auch schon verwendet, wenn „viele“ der Ziffern sich gegenseitig aufheben und die relative Genauigkeit darunter deutlich leidet. Bei der Konstruktion numerischer Algorithmen müssen wir darauf achten, solche Auslöschungseffekte möglichst zu vermeiden.

Als Beispiel untersuchen wir die aus der Schule bekannte pq-Formel für die Nullstellen einer quadratischen Gleichung

$$x^2 - px + q = 0.$$

Bekanntlich erhalten wir mit quadratischer Ergänzung

$$\begin{aligned} 0 = x^2 - px + q = x^2 - 2\frac{p}{2}x + \frac{p^2}{4} - \frac{p^2}{4} + q = \left(x - \frac{p}{2}\right)^2 - \left(\frac{p^2 - 4q}{4}\right), \\ x \in \left\{ \frac{p}{2} + \frac{\sqrt{p^2-4q}}{2}, \frac{p}{2} - \frac{\sqrt{p^2-4q}}{2} \right\}, \end{aligned}$$

so dass eine Implementierung der Gestalt

```
a = sqrt(p*p - 4.0f*q);
x1 = (p + a) * 0.5f;
x2 = (p - a) * 0.5f;
```

nahe liegt. Wenn wir dieses Programm mit Gleitkommazahlen ausführen, stellen wir allerdings fest, dass für kleine Werte von q die jeweils betragskleinere Nullstelle nur relativ ungenau berechnet wird. Die Ursache ist ein Auslöschungseffekt: Für $q \approx 0$ haben wir $p^2 - 4q \approx p^2$, so dass sich $a \approx |p|$ ergibt. Falls p positiv ist, ist also für die Berechnung von x_1 bei $p - a$ Auslöschung zu erwarten, anderenfalls für x_2 bei $p + a$.

Wir können dieses Problem vermeiden, indem wir die *pq*-Formel nur für diejenige Nullstelle verwenden, für die sie eine brauchbare Genauigkeit erreicht. Die zweite Nullstelle können wir rekonstruieren: Wenn x_1 und x_2 die beiden Nullstellen sind, haben wir

$$x^2 - px + q = (x - x_1)(x - x_2) = x^2 - (x_1 + x_2)x + x_1x_2,$$

und aus einem Koeffizientenvergleich folgt $q = x_1x_2$. Falls wir also eine Nullstellen kennen, können wir die zweite ermitteln, indem wir q durch die bekannte Nullstelle dividieren. Der resultierende Algorithmus nimmt die folgende Form an:

```
a = sqrt(p*p - 4.0f*q);
if(p > 0.0f) {
  x1 = (p + a) * 0.5f;
  x2 = q / x1;
}
else {
  x2 = (p - a) * 0.5f;
  x1 = q / x2;
}
```

3.2 Stabilitätsanalyse

Um Algorithmen gezielt so konstruieren zu können, dass sie möglichst wenig von Rundungsfehlern beeinträchtigt werden, empfiehlt es sich, den Einfluss des Rundens mathematisch zu erfassen.

Jede reelle Zahl $x \in \mathbb{R} \setminus \{0\}$ lässt sich in der Form

$$x = \sigma 2^e \sum_{j=0}^{\infty} d_j 2^{-j}$$

mit einem Vorzeichen $\sigma \in \{-1, 1\}$, einem Exponenten $e \in \mathbb{Z}$ und Ziffern $(d_j)_{j=0}^{\infty}$ aus $\{0, 1\}$ darstellen. Wir gehen davon aus, dass die Zahl *normalisiert* dargestellt ist, dass also $d_0 = 1$ gilt.

Wenn wir auf p Nachkommastellen runden, addieren wir 2^{-p}, falls $d_{p+1} = 1$ gilt, und setzen anschließend alle Ziffern ab dem Index $j + 1$ auf null. Das Ergebnis bezeichnen wir mit

$$\mathrm{fl}(x) = \sigma 2^e \sum_{j=0}^{p} d_j 2^{-j} + \begin{cases} 2^{-p} & \text{falls } d_{j+1} = 1, \\ 0 & \text{ansonsten.} \end{cases}$$

Ein großer Vorteil der Gleitkommadarstellung besteht darin, dass wir den durch das Runden verursachten *relativen* Fehler abschätzen können.

Lemma 3.2 (Rundungsfehler) *Es gilt*

$$|x - \mathrm{fl}(x)| \leq 2^{-(p+1)}|x|.$$

Beweis. Wir halten zunächst fest, dass infolge der normalisierten Darstellung $d_0 = 1$ gesichert ist, also

$$|x| = 2^e \sum_{j=0}^{\infty} d_j 2^{-j} \geq 2^e.$$

Falls $d_{p+1} = 0$ gilt, haben wir

$$\begin{aligned}
|x - \mathrm{fl}(x)| &= 2^e \left| \sum_{j=0}^{\infty} d_j 2^{-j} - \sum_{j=0}^{p} d_j 2^{-j} \right| \overset{d_{p+1}=0}{=} 2^e \left| \sum_{j=0}^{\infty} d_j 2^{-j} - \sum_{j=0}^{p+1} d_j 2^{-j} \right| \\
&= 2^e \sum_{j=p+2}^{\infty} d_j 2^{-j} \leq 2^e \sum_{j=p+2}^{\infty} 2^{-j} = 2^{e-(p+1)} \sum_{k=1}^{\infty} 2^{-k} = 2^{e-(p+1)} \leq 2^{-(p+1)}|x|.
\end{aligned}$$

Anderenfalls, also für $d_{p+1} = 1$, haben wir

$$\begin{aligned}
|x - \mathrm{fl}(x)| &= 2^e \left| \sum_{j=0}^{\infty} d_j 2^{-j} - \sum_{j=0}^{p} d_j 2^{-j} - 2^{-p} \right| \overset{d_{p+1}=1}{=} 2^e \left| \sum_{j=0}^{\infty} d_j 2^{-j} - \sum_{j=0}^{p+1} d_j 2^{-j} - 2^{-(p+1)} \right| \\
&= 2^e \left| \sum_{j=p+2}^{\infty} d_j 2^{-j} - 2^{-(p+1)} \right| = 2^{e-(p+1)} \left| \sum_{k=1}^{\infty} d_{k+p+1} 2^{-k} - 1 \right| \\
&= 2^{e-(p+1)} \left| \sum_{k=1}^{\infty} d_{k+p+1} 2^{-k} - \sum_{k=1}^{\infty} 2^{-k} \right| \\
&= 2^{e-(p+1)} \sum_{k=1}^{\infty} |d_{k+p+1} - 1|\, 2^{-k} \leq 2^{e-(p+1)} \sum_{k=1}^{\infty} 2^{-k} = 2^{e-(p+1)} \leq 2^{-(p+1)}|x|.
\end{aligned}$$

■

Für Gleitkommazahlen einfacher Genauigkeit dürfen wir also mit einem relativen Rundungsfehler von höchstens $2^{-24} \approx 5.96 \times 10^{-8}$ rechnen, bei doppelter Genauigkeit ist es $2^{-53} \approx 1.11 \times 10^{-16}$. Diese Schranke nennen wir die *relative Maschinengenauigkeit* des jeweiligen Gleitkommatyps, sie wird häufig mit

$$\epsilon_{\mathrm{mach}} = \begin{cases} 2^{-24} & \text{für einfache Genauigkeit,} \\ 2^{-53} & \text{für doppelte Genauigkeit} \end{cases}$$

bezeichnet. Da die Null als Gleitkommazahl exakt dargestellt werden kann, haben wir

$$|x - \mathrm{fl}(x)| \leq \epsilon_{\mathrm{mach}}|x| \qquad \text{für alle } x \in \mathbb{R} \tag{3.1}$$

bewiesen, wobei wir zur Vereinfachung annehmen, dass der Exponent beliebig groß oder klein werden darf. In der Praxis stehen für den Exponenten nur r Bits zur Verfügung, so dass für Zahlen mit sehr großem oder sehr kleinem Betrag Probleme zu erwarten sind. Wir gehen im Folgenden davon aus, dass in unseren Algorithmen keine Zahlen in diesen Randbereichen auftreten und wir mit der Formel (3.1) arbeiten dürfen.

Da das Rechnen mit Beträgen häufig etwas unhandlich ist, verwendet man gerne die folgende Variante unserer Abschätzung:

Folgerung 3.3 (Rundungsfehler) *Sei $x \in \mathbb{R}$. Dann existiert ein $\delta \in \mathbb{R}$ mit*

$$\mathrm{fl}(x) = (1+\delta)x, \qquad |\delta| \leq \epsilon_{mach}.$$

Beweis. Falls $x = 0$ gilt, können wir $\delta = 0$ setzen und sind fertig.

Anderenfalls definieren wir

$$\delta := \frac{\mathrm{fl}(x) - x}{x}$$

und stellen fest, dass

$$(1+\delta)x = \frac{x + \mathrm{fl}(x) - x}{x} x = \mathrm{fl}(x), \qquad |\delta| = \frac{|\,\mathrm{fl}(x) - x|}{|x|} \leq \epsilon_{\mathrm{mach}}$$

gelten. ∎

Nun können wir daran gehen, unseren verbesserten Algorithmus für die Nullstellenberechnung zu analysieren. Da in der Praxis $\epsilon_{\mathrm{mach}}^2 \ll \epsilon_{\mathrm{mach}}$ gilt, vernachlässigen wir dabei alle Terme, deren Betrag sich durch $\epsilon_{\mathrm{mach}}^2$ mal einer kleinen Konstanten beschränken lässt.

Wir untersuchen nur den Fall $p > 0$ und gehen davon aus, dass eine Näherung

$$\tilde{a} = (1 + c\delta_a)a$$

der Zahl $a = \sqrt{p^2 - 4q}$ mit $|\delta_a| \leq \epsilon_{\mathrm{mach}}$ und einem kleinen Faktor $c > 0$ bereits berechnet worden ist.

Da die Division durch Zweierpotenzen bei Gleitkommazahlen exakt erfolgt (sofern der Exponent nicht überläuft), haben wir dann

$$\begin{aligned}\tilde{x}_1 = \frac{p \oplus \tilde{a}}{2} &= (1+\tilde{\delta}_x)\frac{p+\tilde{a}}{2} = (1+\tilde{\delta}_x)\frac{p + (1+c\delta_a)a}{2} = (1+\tilde{\delta}_x)\frac{p+a}{2} + (1+\tilde{\delta}_x)c\delta_a\frac{a}{2} \\ &\approx (1+\tilde{\delta}_x)x_1 + c\delta_a\frac{a}{2} = (1+\tilde{\delta}_x)x_1 + c\tilde{\delta}_a x_1\end{aligned}$$

mit einem geeigneten $\tilde{\delta}_x \in [-\epsilon_{\mathrm{mach}}, \epsilon_{\mathrm{mach}}]$ und

$$\tilde{\delta}_a := \delta_a \frac{a}{p+a} = \delta_a \frac{a/2}{x_1}.$$

Wegen $a, p \geq 0$ haben wir

$$|\tilde{\delta}_a| = |\delta_a|\frac{|a|}{|p+a|} \leq |\delta_a| \leq \epsilon_{\mathrm{mach}}$$

und erhalten schließlich

$$\tilde{x}_1 = (1 + (c+1)\delta_x)x_1$$

für

$$\delta_x := \frac{\tilde{\delta}_x + c\tilde{\delta}_a}{c+1}, \qquad |\delta_x| = \frac{|\tilde{\delta}_x + c\tilde{\delta}_a|}{c+1} \leq \frac{|\tilde{\delta}_x| + c|\tilde{\delta}_a|}{c+1} \leq \epsilon_{\mathrm{mach}}.$$

Für die Berechnung der betragskleineren Nullstelle haben wir

$$\begin{aligned}\tilde{x}_2 = q \oslash \tilde{x}_1 &= (1+\tilde{\delta}_y)\frac{q}{\tilde{x}_1} = (1+\tilde{\delta}_y)\frac{q}{(1+(c+1)\delta_x)x_1} \\ &= \frac{1+\tilde{\delta}_y}{1+(c+1)\delta_x}\frac{q}{x_1} = \frac{1+\tilde{\delta}_y}{1+(c+1)\delta_x}x_2\end{aligned}$$

mit einem geeigneten $\delta_y \in [-\epsilon_{\text{mach}}, \epsilon_{\text{mach}}]$. Aufgrund unserer Konvention ergibt sich mit der dritten binomischen Formel

$$\frac{1}{1+(c+1)\delta_x} = \frac{1-(c+1)\delta_x}{(1+(c+1)\delta_x)(1-(c+1)\delta_x)} = \frac{1-(c+1)\delta_x}{1-(c+1)^2\delta_x^2} \approx 1-(c+1)\delta_x,$$

also

$$\begin{aligned}\tilde{x}_2 = \frac{1+\tilde{\delta}_y}{1+(c+1)\delta_x}x_2 &\approx (1+\tilde{\delta}_y)(1-(c+1)\delta_x)x_2 \\ &\approx (1+\tilde{\delta}_y-(c+1)\delta_x)x_2 = (1+(c+2)\delta_y)x_2\end{aligned}$$

für

$$\delta_y := \frac{\tilde{\delta}_y-(c+1)\delta_x}{c+2}, \qquad |\delta_y| = \frac{|\tilde{\delta}_y|+(c+1)|\delta_x|}{c+2} \le \epsilon_{\text{mach}}.$$

Insgesamt haben wir also

$$\tilde{x}_1 = (1+(c+1)\delta_x)x_1, \qquad \tilde{x}_2 = (1+(c+2)\delta_y)x_2$$

bewiesen. Die relativen Fehler erfüllen damit die Abschätzungen

$$\frac{|x_1-\tilde{x}_1|}{|x_1|} = (c+1)|\delta_x| \le (c+1)\epsilon_{\text{mach}}, \qquad \frac{|x_2-\tilde{x}_2|}{|x_2|} = (c+2)|\delta_y| \le (c+2)\epsilon_{\text{mach}}$$

mit einer kleinen Konstanten c. Wir dürfen sie als durchaus genau ansehen.

3.3 Kondition und Rückwärtsanalyse

Der im vorherigen Abschnitt illustrierte Zugang zu der Untersuchung der Fortpflanzung des Rundungsfehlers wird als *Vorwärtsanalyse* bezeichnet: Man führt den Algorithmus Schritt für Schritt aus und ermittelt, wie sich die Fehler entwickeln.

Wie man unschwer erkennt, kann diese Methode schnell sehr unhandlich werden.

Glücklicherweise gibt es eine Alternative: Die meisten Algorithmen verarbeiten Eingabedaten und produzieren Ausgabedaten. Angenommen, ein exakt — also ohne Rundungsfehler — arbeitender Algorithmus ermittelt zu den Eingabedaten a die Ausgabedaten b. Infolge der Rundungsfehler erhalten wir bei einer Implementierung im Computer stattdessen die gestörten Ausgabedaten $\widetilde{b}$.

Die Idee der *Rückwärtsanalyse* besteht nun darin, nach gestörten Eingabedaten $\widetilde{a}$ zu suchen, für die der *exakte* Algorithmus das Ergebnis $\widetilde{b}$ berechnen würde.

Da sich der exakte Algorithmus häufig viel einfacher als der gestörte analysieren lässt, erhalten wir so Aussagen über die Genauigkeit von $\widetilde{b}$.

Als Beispiel dient eine Aufgabe aus der linearen Algebra, nämlich das Lösen eines linearen Gleichungssystems

$$Rx = b$$

mit einer oberen Dreiecksmatrix

$$R = \begin{pmatrix} r_{11} & \dots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{pmatrix}.$$

und Vektoren

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}.$$

Im Allgemeinen ist das Lösen eines linearen Gleichungssystems eine schwierige Aufgabe. Da aber R eine obere Dreiecksmatrix ist, können wir einen einfachen Algorithmus anwenden: Wir zerlegen R, x und b in

$$\begin{pmatrix} R_{**} & R_{*1} \\ & r_{nn} \end{pmatrix} \begin{pmatrix} x_* \\ x_n \end{pmatrix} = \begin{pmatrix} b_* \\ b_n \end{pmatrix} \tag{3.2}$$

mit

$$R_{*n} = \begin{pmatrix} r_{1n} \\ \vdots \\ r_{n-1,n} \end{pmatrix}, \qquad R_{**} = \begin{pmatrix} r_{11} & \dots & r_{1,n-1} \\ & \ddots & \vdots \\ & & r_{n-1,n-1} \end{pmatrix},$$
$$x_* = \begin{pmatrix} x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}, \qquad b_* = \begin{pmatrix} b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}.$$

Aus der Definition der Matrix-Vektor-Multiplikation folgt unmittelbar, dass (3.2) äquivalent zu

$$R_{**}x_* + R_{*n}x_n = b_*, \qquad r_{nn}x_n = b_n$$

ist. Falls $r_{nn} \neq 0$ gilt, folgt aus der zweiten Gleichung unmittelbar $x_n = b_n/r_{nn}$, die letzte Komponente des Lösungsvektors können wir also unmittelbar berechnen. Sobald wir sie kennen, können wir die erste Gleichung in die Form

$$R_{**}x_* = b_* - R_{*n}x_n$$

bringen, die wieder ein lineares Gleichungssystem mit der oberen Dreiecksmatrix R_{**} ist. Allerdings ist R_{**} um eine Zeile und eine Spalte kleiner als R, so dass wir induktiv

fortfahren können. Der resultierende Algorithmus trägt aus naheliegenden Gründen den Namen *Rückwärtseinsetzen*.

Man kann sich vorstellen, dass die Vorwärtsanalyse dieses Algorithmus' relativ aufwendig wird: Schon x_n wird nur näherungsweise berechnet, und die Näherung wird anschließend mit R_{*n} multipliziert und von b_* subtrahiert, wobei weitere Rundungsfehler auftreten.

Die Rückwärtsanalyse dagegen ist relativ überschaubar: Die Näherung $\tilde{x}_n$ des Koeffizienten x_n wird durch

$$\tilde{x}_n = b_n \oslash r_{nn}$$

ermittelt, und nach Folgerung 3.3 finden wir ein $\delta_n \in \mathbb{R}$ mit $|\delta_n| \leq \epsilon_{\text{mach}}$ und

$$\tilde{x}_n = (1+\delta_n) b_n / r_{nn} = (1+\delta_n) x_n, \qquad |\delta_n| \leq \epsilon_{\text{mach}}.$$

Die Idee der Rückwärtsanalyse besteht darin, Eingabedaten zu finden, für die der exakte Algorithmus das gestörte Ergebnis berechnen würde. In unserem Fall sind b und R die Eingabedaten, und mit

$$\tilde{b}_n := (1+\delta_n) b_n$$

gilt offenbar

$$\tilde{x}_n = \tilde{b}_n / r_{nn}.$$

Die Näherung $\tilde{d}_*$ des Vektors

$$d_* := b_* - R_{*n} x_n$$

ist durch

$$\tilde{d}_i = b_i \ominus r_{in} \odot \tilde{x}_n \qquad \text{für alle } i \in [1 : n-1]$$

gegeben. Für ein $i \in [1 : n-1]$ finden wir nach Folgerung 3.3 jeweils $\delta, \delta' \in \mathbb{R}$ mit

$$r_{in} \odot \tilde{x}_n = (1+\delta) r_{in} \tilde{x}_n, \qquad \tilde{d}_i = (1+\delta')(b_i - (1+\delta) r_{in} \tilde{x}_n), \qquad |\delta|, |\delta'| \leq \epsilon_{\text{mach}}.$$

Auch in diesem Fall können wir die Rundungsfehler in den Eingabedaten „verstecken“, indem wir

$$\tilde{r}_{in} := (1+\delta')(1+\delta) r_{in}$$

definieren und

$$\tilde{d}_i = (1+\delta') b_i - \tilde{r}_{in} \tilde{x}_n$$

erhalten. Nun können wir den Rest des Lösungsvektors x_* approximieren, indem wir

$$R_{**} x_* = \tilde{d}_i$$

lösen und die dabei auftretenden zusätzlichen Rundungsfehler wieder durch Störungen des Vektors b_* und der Matrix R_{**} ausdrücken.

Am Ende erhalten wir eine obere Dreiecksmatrix $\widetilde{R}$ und einen Vektor $\widetilde{b}$ mit

$$\widetilde{R}\widetilde{x} = \widetilde{b},$$

die praktisch berechnete Näherungslösung $\widetilde{x}$ ist also die exakte Lösung des gestörten Gleichungssystems. Dank unserer Konstruktion wissen wir, wie nahe $\widetilde{R}$ und $\widetilde{b}$ and R und b liegen, so dass wir „nur“ herausfinden müssen, welche Beziehung zwischen $\widetilde{x}$ und x sich daraus ableiten lässt.

Dafür müssen wir quantifizieren können, wie nahe Vektoren und Matrizen einander sind. Das übliche Mittel dafür sind *Normen*.

Eine Norm auf einem $\mathbb{R}$-Vektorraum $\mathcal{V}$ ist eine Abbildung, die jedem Vektor $v \in V$ eine Zahl $\|v\|_V \geq 0$ zuordnet und die folgenden drei Eigenschaften besitzt:

$$\begin{aligned} \|v\|_V = 0 &\iff v = 0 && \text{für alle } v \in V, \\ \|\lambda v\|_V &= |\lambda| \, \|v\|_V && \text{für alle } v \in V,\ \lambda \in \mathbb{R}, \\ \|v + w\|_V &\leq \|v\|_V + \|w\|_V && \text{für alle } v, w \in V. \end{aligned}$$

Diese Zahl können wir als „Länge“ des Vektors v interpretieren.

Für den Raum $V = \mathbb{R}^n$ der n-Tupel reeller Zahlen sind beispielsweise die *Maximumnorm*

$$\|v\|_\infty = \max\{|v_i| \ : \ i \in [1:n]\} \qquad \text{für alle } v \in \mathbb{R}^n, \tag{3.3a}$$

die *Manhattan-Norm*

$$\|v\|_1 = \sum_{i=1}^{n} |v_i| \qquad \text{für alle } v \in \mathbb{R}^n \tag{3.3b}$$

und die *euklidische Norm*

$$\|v\|_2 = \left(\sum_{i=1}^{n} |v_i|^2 \right)^{1/2} \qquad \text{für alle } v \in \mathbb{R}^n \tag{3.3c}$$

üblich. Die euklidische Norm entspricht aufgrund des Satzes von Pythagoras gerade dem Abstandsbegriff, den wir aus der zwei- und dreidimensionalen Geoemtrie gewöhnt sind.

Die „Länge“ einer Matrix $X \in \mathbb{R}^{n \times n}$ würden wir gerne so messen, dass sie mit der für den Raum $\mathbb{R}^n$ verwendeten Norm *verträglich* ist: Wenn $\|\cdot\|_V$ eine Norm auf $\mathbb{R}^n$ ist, würden wir gerne eine Norm $\|\cdot\|_{V \leftarrow V}$ auf $\mathbb{R}^{n \times n}$ definieren, die

$$\|Xv\|_V \leq \|X\|_{V \leftarrow V} \|v\|_V \qquad \text{für alle } X \in \mathbb{R}^{n \times n},\ v \in \mathbb{R}^n \tag{3.4}$$

erfüllt. Diese Eigenschaft erleichtert es erheblich, beispielsweise Aussagen über die Fehlerfortpflanzung zu formulieren und zu beweisen.

Offenbar ist (3.4) äquivalent dazu, dass

$$\frac{\|Xv\|_V}{\|v\|_V} \leq \|X\|_{V \leftarrow V} \qquad \text{für alle } X \in \mathbb{R}^{n \times n},\ v \in \mathbb{R}^n \setminus \{0\}$$

gilt, so dass die „kleinste“ Norm mit der gewünschten Eigenschaft gerade durch

$$\|X\|_{V\leftarrow V} := \sup\left\{\frac{\|Xv\|_V}{\|v\|_V} \;:\; v\in\mathbb{R}^n\setminus\{0\}\right\} \qquad \text{für alle } X\in\mathbb{R}^{n\times n}$$

definiert ist. Diese Norm nennen wir die von $\|\cdot\|_V$ *induzierte Operatornorm* auf $\mathbb{R}^{n\times n}$.

In unserem Fall ist die Maximumnorm besonders bequem, mit der wir den relativen Fehler der Matrix $\widetilde{R}$ und des Vektors $\widetilde{b}$ in der Form

$$\|R-\widetilde{R}\|_{\infty\leftarrow\infty}\leq C_R n\,\epsilon_{\text{mach}}\|R\|_{\infty\leftarrow\infty}, \qquad \|b-\widetilde{b}\|_\infty\leq C_b n\,\epsilon_{\text{mach}}\|b\|_\infty$$

mit Konstanten $C_r, C_b\approx 1$ abschätzen können.

Mit etwas Aufwand lässt sich die folgende Aussage über den Einfluss von Störungen auf das Lösen eines linearen Gleichungssystems beweisen:

Satz 3.4 (Störungssatz für lineare Gleichungssysteme) *Seien $A,\widetilde{A}\in\mathbb{R}^{n\times n}$ und $b,\widetilde{b}\in\mathbb{R}^n$ so gegeben, dass A invertierbar ist und*

$$\|A^{-1}(A-\widetilde{A})\|_{V\leftarrow V}<1.$$

Dann besitzen die linearen Gleichungssysteme

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

eindeutig bestimmte Lösungen $x,\widetilde{x}\in\mathbb{R}^n$, die die Fehlerabschätzung

$$\frac{\|x-\widetilde{x}\|_V}{\|x\|_V}\leq\frac{\kappa_V(A)}{1-\|A^{-1}(A-\widetilde{A})\|_{V\leftarrow V}}\left(\frac{\|A-\widetilde{A}\|_{V\leftarrow V}}{\|A\|_{V\leftarrow V}}+\frac{\|b-\widetilde{b}\|_V}{\|b\|_V}\right)$$

mit der durch

$$\kappa_V(A):=\|A\|_{V\leftarrow V}\|A^{-1}\|_{V\leftarrow V}$$

gegebenen Konditionszahl *erfüllen.*

Bemerkung 3.5 (Relativer Fehler) *Man kann leicht nachprüfen, dass*

$$\|A^{-1}(A-\widetilde{A})\|_{V\leftarrow V}\leq\|A\|_{V\leftarrow V}\|A^{-1}\|_{V\leftarrow V}\frac{\|A-\widetilde{A}\|_{V\leftarrow V}}{\|A\|_{V\leftarrow V}}=\kappa_V(A)\frac{\|A-\widetilde{A}\|_{V\leftarrow V}}{\|A\|_{V\leftarrow V}}$$

gilt, so dass sich unter der Voraussetzung

$$\kappa_V(A)\frac{\|A-\widetilde{A}\|_{V\leftarrow V}}{\|A\|_{V\leftarrow V}}<1$$

die vereinfachte (aber auch ungenauere) Fehlerabschätzung

$$\frac{\|x-\widetilde{x}\|_V}{\|x\|_V}\leq\frac{\kappa_V(A)}{1-\kappa_V(A)\frac{\|A-\widetilde{A}\|_{V\leftarrow V}}{\|A\|_{V\leftarrow V}}}\left(\frac{\|A-\widetilde{A}\|_{V\leftarrow V}}{\|A\|_{V\leftarrow V}}+\frac{\|b-\widetilde{b}\|_V}{\|b\|_V}\right)$$

ergibt. Falls also der relative Fehler der Matrixstörung klein genug ist, ist auch das gestörte Gleichungssystem eindeutig lösbar, und der Fehler lässt sich durch die relativen Fehler der Störungen der Matrix und der rechten Seite abschätzen.

Wenn wir davon ausgehen, dass die Konditionszahl $\kappa_V(A)$ nicht allzu groß ist, erhalten wir für das Rückwärtseinsetzen die Abschätzung

$$\frac{\|x - \widetilde{x}\|_\infty}{\|x\|_\infty} \leq C_x n \, \epsilon_{\text{mach}} \kappa_V(R)$$

mit einer Konstanten $C_x \approx 2$. Von entscheidender Bedeutung für die Verstärkung des Rundungsfehlers ist demnach die Konditionszahl $\kappa_V(R)$.

Diese Zahl ist eine Eigenschaft der Aufgabenstellung, die wir nicht beeinflussen können. Sie legt die Grenze der Genauigkeit fest, die auch ein perfekter Algorithmus nicht übertreffen kann, weil schon die Eingabedaten nur genähert zur Verfügung stehen.

Einen Algorithmus, der ein Ergebnis berechnet, dessen Genauigkeit sich immerhin in der Nähe der theoretisch bestmöglichen bewegt, nennen wir *stabil*.

4 Lineare Gleichungssysteme

Wir haben bereits gesehen, dass sich lineare Gleichungssysteme durch Rückwärtseinsetzen lösen lassen, falls die auftretende Matrix obere Dreiecksgestalt besitzt.

In der Praxis treten meistens Gleichungssysteme auf, die diese Eigenschaft nicht aufweisen. Allerdings ist es möglich, ein beliebiges Gleichungssystem so zu zerlegen, dass nur noch Matrizen in Dreiecksgestalt auftreten und sich das System auflösen lässt.

4.1 Gauß-Elimination

Bereits aus der Schule ist das Gauß'sche Eliminationsverfahren bekannt, mit dem sich ein lineares Gleichungssystem auf obere Dreiecksgestalt bringen lässt.

Dabei eliminiert man der Reihe nach Unbekannte aus dem Gleichungssystem, bis nur noch eine einzige Unbekannte übrig bleibt, die sich direkt berechnen lässt. Anschließend können alle weiteren Unbekannten rekonstruiert werden.

Als Beispiel untersuchen wir das Gleichungssystem

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

Wir eliminieren a_{21}, indem wir die erste Zeile mit $\ell_{21} := a_{21}/a_{11}$ multiplizieren und von der zweiten Zeile subtrahieren, um

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} - \ell_{21}a_{12} & a_{23} - \ell_{21}a_{13} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 - \ell_{21}b_1 \\ b_3 \end{pmatrix}$$

zu erhalten. Entsprechend können wir a_{31} eliminieren, indem wir die erste Zeile mit $\ell_{31} := a_{31}/a_{11}$ multiplizieren und von der dritten Zeile subtrahieren, um zu

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} - \ell_{21}a_{12} & a_{23} - \ell_{21}a_{13} \\ 0 & a_{32} - \ell_{31}a_{12} & a_{33} - \ell_{31}a_{13} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 - \ell_{21}b_1 \\ b_3 - \ell_{31}b_1 \end{pmatrix} \tag{4.1}$$

zu gelangen. Die Komponenten x_2 und x_3 können nun unabhängig von x_1 bestimmt werden, indem wir das Gleichungssystem

$$\begin{pmatrix} a_{22} - \ell_{21}a_{12} & a_{23} - \ell_{21}a_{13} \\ a_{32} - \ell_{31}a_{12} & a_{33} - \ell_{31}a_{13} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_2 - \ell_{21}b_1 \\ b_3 - \ell_{31}b_1 \end{pmatrix}$$

auflösen. In diesem Sinn wurde die Unbekannte x_1 aus der Berechnung eliminiert.

Zur Abkürzung führen wir

$$A^{(1)} := \begin{pmatrix} a_{22} - \ell_{21} a_{12} & a_{23} - \ell_{21} a_{13} \\ a_{32} - \ell_{31} a_{12} & a_{33} - \ell_{31} a_{13} \end{pmatrix}, \qquad b^{(1)} := \begin{pmatrix} b_2 - \ell_{21} b_1 \\ b_3 - \ell_{31} b_1 \end{pmatrix}$$

ein und müssen

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \end{pmatrix}$$

auflösen. Dazu eliminieren wir $a_{21}^{(1)}$, indem wir die erste Zeile mit $\ell_{21}^{(1)} := a_{21}^{(1)} / a_{11}^{(1)}$ multiplizieren und von der zweiten subtrahieren. Das Ergebnis ist

$$\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} \\ 0 & a_{22}^{(1)} - \ell_{21}^{(1)} a_{12}^{(1)} \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} - \ell_{21}^{(1)} b_1^{(1)} \end{pmatrix}. \tag{4.2}$$

Zur Abkürzung führen wir

$$A^{(2)} := \left(a_{22}^{(1)} - \ell_{21}^{(1)} a_{12}^{(1)} \right), \qquad b^{(2)} := \left(b_2^{(1)} - \ell_{21}^{(1)} b_1^{(1)} \right)$$

ein und stellen fest, dass wir x_3 als Lösung des eindimensionalen Systems

$$A^{(2)} x_3 = b^{(2)}$$

bestimmen können, nämlich als

$$x_3 = b_1^{(2)} / a_{11}^{(2)}.$$

Diesen Wert können wir in (4.2) einsetzen, um

$$x_2 = (b_1^{(1)} - a_{12}^{(1)} x_3) / a_{11}^{(1)}$$

zu erhalten, und damit folgt aus (4.1) unmittelbar

$$x_1 = (b_1 - a_{12} x_2 - a_{13} x_3) / a_{11}.$$

Bei der praktischen Implementierung können wir ausnutzen, dass die Einträge der Matrix A und des Vektors b mit neuen Werten überschrieben werden können, um Speicherplatz zu sparen.

Die jeweils eliminierten Koeffizienten können genutzt werden, um die Skalierungsfaktoren ℓ_{21}, ℓ_{31} und $\ell_{21}^{(1)}$ aufzubewahren. Nach dem ersten Schritt des Verfahrens hätten wir also A und b mit

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \ell_{21} & a_{11}^{(1)} & a_{12}^{(1)} \\ \ell_{31} & a_{21}^{(1)} & a_{22}^{(1)} \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} b_1 \\ b_1^{(1)} \\ b_2^{(1)} \end{pmatrix}$$

überschrieben nach dem zweiten Schritt dann mit

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ \ell_{21} & a_{11}^{(1)} & a_{12}^{(1)} \\ \ell_{31} & \ell_{21}^{(1)} & a_{11}^{(2)} \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} b_1 \\ b_1^{(1)} \\ b_1^{(2)} \end{pmatrix}.$$

Entsprechend können wir während des Rückwärtseinsetzens die Koeffizienten der rechten Seite nach und nach durch die berechneten Koeffizienten der Lösung ersetzen.

4.2 Basic Linear Algebra Subroutines

In vielen Anwendungen müssen sehr große lineare Gleichungssysteme mit zehntausend oder mehr Unbekannten gelöst werden. Der Rechenaufwand der Gauß-Elimination wächst *kubisch* in der Anzahl der Unbekannten, so dass es von großer Bedeutung ist, den Algorithmus so effizient wie möglich zu gestalten.

In der Praxis sind Fachleute selten, die sowohl die Hardware optimal ausnutzen können, als auch die numerischen Algorithmen eingehend studiert haben. Deshalb hat man sich auf einen überschaubaren Satz von einfachen Operationen für Vektoren und Matrizen geeinigt, auf die sich die wichtigsten numerischen Algorithmen zurückführen lassen.

Hardware-Fachleute können diese grundlegenden Operationen optimal implementieren, und Numerik-Fachleute können sie dann benutzen, um ihre Algorithmen umzusetzen.

Dieser Satz von Funktionen trägt den Namen BLAS, eine Abkürzung für *basic linear algebra subroutines*. Der Hauptteil des BLAS-Pakets ist in drei Stufen aufgeteilt:

- Zu Stufe 1 gehören Funktionen, die mit Vektoren arbeiten, beispielsweise die Berechnung von Linearkombinationen, Skalarprodukten und Normen.
- Zu Stufe 2 gehören Funktionen, die mit Vektoren und Matrizen arbeiten, beispielsweise die Berechnung der Matrix-Vektor-Multiplikation oder Rang-1-Updates.
- Zu Stufe 3 gehören Funktionen, die mit Matrizen arbeiten, beispielsweise die Berechnung des Matrix-Produkts.

Dabei lassen sich die Funktionen der Stufe 3 auf Funktionen der Stufe 2 zurückführen, die sich wiederum mit Funktionen der Stufe 1 umsetzen lassen. Eine gute BLAS-Implementierung wird allerdings die auf den Stufen 2 und 3 vorhandene Struktur ausnutzen, um die vorhandene Hardware möglichst optimal auszunutzen.

BLAS stellt eine Vektor $x \in \mathbb{R}^n$ als ein Array `x` und ein Inkrement `incx` dar. Der Koeffizient x_i findet sich dann unter `x[i*incx]`. Entsprechend der C-Konvention beginnt die Indizierung dabei mit $i = 0$.

Für Matrizen $A \in \mathbb{R}^{m \times n}$ kennt BLAS mehrere Darstellungen, von denen die allgemeinste die Matrix als ein Array `A` und ein Spalteninkrement `ldA` (für *leading dimension*) verwendet. Der Koeffizient a_{ij} findet sich unter `A[i+j*ldA]`, wobei die Indizierung wieder bei $i = j = 0$ beginnt.

Ein Beispiel für eine Funktion der Stufe 1 ist

```
void cblas_dscal(int n, double alpha,
                 double *x, int incx);
```

Diese Funktion überschreibt einen Vektor $x \in \mathbb{R}^n$ mit dem Vektor αx, führt also die Operation $x \leftarrow \alpha x$ aus. Dabei gibt `n` die Länge des Vektors an, `alpha` den Skalierungsfaktor, `x` das Array des Vektors und `incx` sein Inkrement. Es gibt auch Varianten `sscal` für `float`-Vektoren sowie `cscal` und `zscal` für Vektoren mit Koeffizienten des Typs `complex float` und `complex double`.

Ein weiteres wichtiges Beispiel ist die Funktion

```
void cblas_daxpy(int n, double alpha,
                 const double *x, int incx,
                 double *y, int incy);
```

Sie berechnet eine Linearkombination, indem sie den Vektor y mit $\alpha x + y$ überschreibt, also die Operation $y \leftarrow \alpha x + y$ ausführt.

Etwas weniger zentral, aber immer noch wichtig, sind die Funktionen

```
double cblas_dnrm2(int n, const double *x, int incx);
double cblas_ddot(int n, const double *x, int incx,
                  const double *y, incy);
```

Sie sind für die euklidische Norm

$$\|x\|_2 = \sqrt{x^* x} = \left(\sum_{i=0}^{n-1} x_i \right)^{1/2}$$

und das euklidische Skalarprodukt

$$\langle x, y \rangle_2 = x^* y = \sum_{i=0}^{n-1} x_i y_i$$

zuständig.

Eine zentrale Funktion der Stufe 2 ist

```
void cblas_dgemv(CBLAS_LAYOUT layout, CBLAS_TRANSPOSE trans,
                 int m, int n, double alpha,
                 const double *A, int ldA,
                 const double *x, int incx,
                 double *y, int incy);
```

Sie führt eine Matrix-Vektor-Multiplikation aus. Dabei entscheidet `layout` darüber, wie die Matrix dargestellt wird (in unserem Fall `layout==CblasColMajor`), während `trans` festlegt, ob mit der Matrix A oder ihrer Transponierten A^* multipliziert werden soll. Je nach Wahl wird entweder $y \leftarrow \alpha A x + y$ oder $y \leftarrow \alpha A^* x + y$ berechnet. Dabei gibt `m` die Anzahl der Zeilen und `n` die Anzahl der Spalten der Matrix an.

Für `layout==CblasColMajor` und `trans==CblasNotTrans` könnte diese Funktion wie folgt auf `cblas_daxpy` zurückgeführt werden:

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

Die j-te Spalte der Matrix wird also mit αx_j multipliziert und zu y addiert. Man beachte, dass hier Zeiger-Arithmetik verwendet wird, um mit `A+j*ldA` einen Zeiger auf die j-te Spalte der Matrix A zu erhalten.

Alternativ könnte man auch `cblas_ddot` verwenden und

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

schreiben. Abhängig von der eingesetzten Hardware kann die erste oder die zweite Fassung Vorteile bieten. Beispielsweise ist die erste Fassung auf Rechnern sehr viel schneller, die über Vektor-Rechenwerke verfügen, weil sich bei `cblas_daxpy` alle m Zeilen des Ergebnisses unabhängig voneinander aktualisieren lassen.

Für die Gauß-Elimination ist die ebenfalls zur Stufe 2 gehörende Funktion

```
void cblas_dger(CBLAS_LAYOUT layout,
                int m, int n, double alpha,
                const double *x, int incx,
                const double *y, int incy,
                double *A, int ldA);
```

sehr wichtig. Sie addiert $\alpha x_i y_j$ zu dem Eintrag a_{ij} der Matrix, führt also das *Rang-1-Update* $A \leftarrow A + \alpha x y^*$ durch. Hier beschreibt `layout` wieder die Darstellung, `m` die Anzahl der Zeilen und `n` die Anzahl der Spalten.

Für `layout==CblasColMajor` kann auch diese Funktion auf `cblas_daxpy` zurückgeführt werden:

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

Zu der j-ten Spalte der Matrix wird also der Vektor $\alpha y_j x$ addiert. Wie zuvor wird diese Spalte mit Zeigerarithmetik durch `A+j*ldA` ermittelt.

Die Implementierung der Gauß-Elimination auf Grundlage der BLAS-Funktionen kann beispielsweise so erfolgen, dass man im ersten Schritt mit der Funktion `cblas_dscal` die Koeffizienten $\ell_{i1} = a_{i1}/a_{11}$ berechnet, um dann im zweiten Schritt mit der Funktion `cblas_dger` die Teilmatrix $A^{(1)}$ zu konstruieren. Wenn die ursprüngliche Matrix n Zeilen und Spalten hatte, hat $A^{(1)}$ nur noch $n-1$ Zeilen und Spalten. Einen Zeiger auf den entsprechenden Teil der größeren Matrix A können wir mit Zeiger-Arithmetik als `A+1+ldA` gewinnen. Wichtig ist dabei, dass auch diese Teilmatrix noch dieselbe *leading dimension* wie die ursprüngliche Matrix verwendet, damit korrekt von einer Spalte zur nächsten gewechselt wird.

Die mit großem Abstand wichtigste Funktion der Stufe 3 ist

```
void cblas_dgemm(CBLAS_LAYOUT layout,
                 CBLAS_TRANSPOSE transA, CBLAS_TRANSPOSED transB,
                 int m, int n, int k, double alpha,
                 const double *A, int ldA,
                 const double *B, int ldB,
                 double beta,
                 double *C, int ldC);
```

Sie multipliziert zwei Matrizen A und B miteinander, skaliert das Ergebnis mit α, und addiert es zu einer Matrix C, die zuvor mit β skaliert wurde. Die ausgeführte Operation

ist also $C \leftarrow \alpha AB + \beta C$. Mit `transA` und `transB` kann festgelegt werden, dass die Transponierten von A oder B verwendet werden sollen. Dabei ist $C \in \mathbb{R}^{m\times n}$. Falls A nicht transponiert wird, gilt $A \in \mathbb{R}^{m\times k}$, anderenfalls $A \in \mathbb{R}^{k\times m}$. Falls B nicht transponiert wird, gilt $B \in \mathbb{R}^{k\times n}$, anderenfalls $B \in \mathbb{R}^{n\times k}$.

Für `layout==CblasColMajor` und `transB=CblasNoTrans` kann diese Funktion beispielsweise wie folgt auf `cblas_dgemv` zurückgeführt werden:

```
for(j=0; j<n; j++) {
  cblas_dscal(m, beta, C+j*ldC, 1);
  cblas_dgemv(layout, transA, m, k, alpha,
              A, ldA, B+j*ldB, 1, C+j*ldC, 1);
}
```

Wir multiplizieren einfach jede Spalte der Matrix B mit der Matrix A und addieren das Ergebnis zu der korrespondierenden Spalte der Matrix C.

Auch hier gibt es Alternativen, beispielsweise können wir `cblas_dger` verwenden, um die Produkte aus jeweils einer Spalte der Matrix A und einer Zeile der Matrix B zu der Matrix C zu addieren:

```
for(j=0; j<k; j++)
  cblas_dger(layout, m, n, beta, A+j*ldA, 1, B+j, ldB, C, ldC);
```

In den meisten BLAS-Implementierungen wird sehr großer Wert darauf gelegt, die Funktion `cblas_dgemm` möglichst schnell laufen zu lassen, mindestens durch Vektorisierung, gelegentlich sogar durch Parallelisierung. Deshalb ist es sehr empfehlenswert, numerische Algorithmen so zu formulieren, dass sie einen möglichst großen Teil ihrer Arbeit mit Matrix-Multiplikationen erledigen können.

Im Folgenden werden wir zur Abkürzung auf das Präfix `cblas_` und den Parameter `layout` verzichten, wenn wir Bezug auf BLAS nehmen. Wir gehen davon aus, dass Matrizen immer im *column-major format* vorliegen, also Spalte für Spalte hintereinander im Speicher stehen.

4.3 Faktorisierungen und Block-Algorithmen

Ein genauerer Blick auf die Gauß-Elimination zeigt, dass wir tatsächlich eine *Faktorisierung* der Matrix $A \in \mathbb{R}^{n\times n}$ berechnen, also eine Zerlegung

$$A = LR$$

mit den Dreiecksmatrizen

$$L = \begin{pmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \dots & \ell_{n,n-1} & 1 \end{pmatrix}, \qquad R = \begin{pmatrix} r_{11} & r_{12} & \dots & r_{1n} \\ & \ddots & \ddots & \vdots \\ & & r_{n-1,n-1} & r_{n-1,n} \\ & & & r_{nn} \end{pmatrix}.$$

Das Gleichungssystem

$$Ax = b$$

ist äquivalent zu

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

Bei der Gauß-Elimination wird der Vektor y implizit berechnet, da wir die einzelnen Transformationen auch jeweils auf die rechte Seite unserer Gleichungen anwenden. Die Lösung ergibt sich dann aus der zweiten Gleichung in Dreiecksform.

Die Gleichung $Ly = b$ können wir durch *Vorwärtseinsetzen* lösen: Aus der ersten Zeile der Gleichung folgt $y_1 = b_1$. Damit können wir diese Unbekannte aus den verbliebenen Zeilen eliminieren, indem wir b_i für $i > 1$ jeweils durch $b_i - \ell_{i1} y_1$ ersetzen. Anschließend verfahren wir mit der zweiten Zeile und den weiteren entsprechend. Der Algorithmus lässt sich kompakt mit BLAS ausdrücken:

```
for(j=0; j<n; j++)
  daxpy(n-j-1, -b[j], L+(j+1)+j*ldL, 1, b+(j+1)*incb, incb);
```

Für die Gleichung $Rx = y$ kommt entsprechend das *Rückwärtseinsetzen* zum Einsatz: Aus der letzten Zeile der Gleichung folgt $r_{nn} x_n = b_n$, also $x_n = b_n / r_{nn}$. Damit können wir diese Unbekannte aus der verbliebenen Zeilen eliminieren, indem wir b_i für $i < n$ jeweils durch $b_i - r_{in} x_n$ ersetzen. Mit BLAS kann man den Algorithmus auf `daxpy` zurückführen:

```
for(j=n; j-->0; ) {
  y[j] /= R[j+j*ldR];
  daxpy(j, -y[j], R+j*ldR, 1, y, incy);
}
```

Sobald uns also die *LR-Zerlegung* $A = LR$ einer Matrix A zur Verfügung steht, ist es relativ einfach, Gleichungssysteme mit dieser Matrix zu lösen.

Bei der Gauß-Elimination wird die LR-Zerlegung bestimmt, indem Schritt für Schritt eine Variable nach der anderen eliminiert wird. Das mag für menschliche Rechner sinnvoll sein, für elektronische allerdings eher nicht: Für die Elimination einer Variablen eines n-dimensionalen Gleichungssystems müssen $(n-1)^2$ Koeffizienten der Matrix verändert werden. Falls die Matrix groß ist, wird sie nicht in den Cache des Prozessors passen, so dass sehr viel auf den langsamen Hauptspeicher zugegriffen werden muss.

Dieser Nachteil lässt sich innerhalb gewisser Grenzen vermeiden, indem man zu einem *Blockalgorithmus* übergeht, also nicht mit einzelnen Zeilen und Spalten der Matrix rechnet, sondern mit größeren Blöcken.

Sei dazu $c \in [1 : n]$ eine geeignete Blockgröße. Geeignet sind beispielsweise Größen, die so beschaffen sind, dass eine $c \times c$-Matrix in den Cache des Prozessors passt.

Wir zerlegen die Matrix A

$$A_{11} := \begin{pmatrix} a_{11} & \dots & a_{1c} \\ \vdots & \ddots & \vdots \\ a_{b1} & \dots & a_{bc} \end{pmatrix}, \qquad A_{12} := \begin{pmatrix} a_{1,c+1} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{c,c+1} & \dots & a_{cn} \end{pmatrix},$$

$$A_{21} := \begin{pmatrix} a_{c+1,1} & \dots & a_{c+1,c} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nc} \end{pmatrix}, \qquad A_{22} := \begin{pmatrix} a_{c+1,c+1} & \dots & a_{c+1,n} \\ \vdots & \ddots & \vdots \\ a_{n,c+1} & \dots & a_{nn} \end{pmatrix}$$

und schreiben A in der kompakten *Blockform*

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \left(\begin{array}{ccc|ccc} a_{11} & \dots & a_{1c} & a_{1,c+1} & \dots & a_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{c1} & \dots & a_{cc} & a_{c,c+1} & \dots & a_{cn} \\ \hline a_{c+1,1} & \dots & a_{c+1,c} & a_{c+1,c+1} & \dots & a_{c+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nc} & a_{n,c+1} & \dots & a_{nn} \end{array}\right).$$

Mit den Dreiecksmatrizen können wir entsprechend verfahren, um

$$L = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix}, \qquad R = \begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix}$$

zu erhalten. Aus der von uns angestrebten Gleichung

$$A = LR$$

ergibt sich so

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = A = LR = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix} = \begin{pmatrix} L_{11}R_{11} & L_{11}R_{12} \\ L_{21}R_{11} & L_{21}R_{12} + L_{22}R_{22} \end{pmatrix}.$$

Indem wir diese Gleichung komponentenweise lesen, erhalten wir

$$\begin{gathered} A_{11} = L_{11}R_{11}, \\ A_{12} = L_{11}R_{12}, \qquad A_{21} = L_{21}R_{11}, \\ A_{22} - L_{21}R_{12} = L_{22}R_{22}. \end{gathered}$$

Wir sehen, dass sich die Faktoren L_{11} und R_{11} alleine aus der Matrix A_{11} bestimmen lassen. Falls also die Blockgröße so gewählt ist, dass A_{11} in einen schnellen Cache passt, können bei der Berechnung von L_{11} und R_{11} Zugriffe auf den langsamen Hauptspeicher weitgehend vermieden werden.

Sobald uns L_{11} zur Verfügung steht, können wir die zweite Gleichung durch Vorwärtseinsetzen lösen. Dasselbe gilt für R_{11} und die dritte Gleichung, denn sie ist äquivalent zu $A_{21}^* = R_{11}^* L_{21}^*$, und R_{11}^* ist wieder eine untere Dreiecksmatrix. Es empfiehlt sich, auch diese Operationen im Interesse der Datenlokalität blockweise zu gestalten, also beispielsweise jeweils c Spalten der Matrix R_{12} oder c Zeilen der Matrix L_{21} gemeinsam zu berechnen.

Wenn uns schließlich L_{21} und R_{12} bekannt sind, können wir die Matrix $A_{22} - L_{21}R_{12}$ berechnen. Auch hier empfiehlt sich eine blockweise Vorgehensweise, die sich besonders

effizient gestalten lässt, indem wir ausnutzen, dass uns mit `dgemm` eine — wahrscheinlich sehr effizient implementierte — Funktion für die Berechnung von Matrix-Produkten zur Verfügung steht.

Die LR-Zerlegung der so konstruierten Matrix kann dann beispielsweise durch Rekursion erfolgen.

4.4 Orthogonale Zerlegungen

Gemäß Satz 3.4 reagiert die Lösung eines linearen Gleichungssystems um so empfindlicher auf Störungen, je größer die Konditionszahl der Matrix ist.

Wenn wir das Gleichungssystem mit Hilfe einer LR-Zerlegung behandeln, müssen wir die Systeme

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

mit Vorwärts- und Rückwärtseinsetzen auflösen, so dass die Konditionszahlen der Matrizen L und R für die zu erwartende Genauigkeit den Ausschlag geben.

Unglücklicherweise können diese Konditionszahlen erheblich größer als die Konditionszahl der Matrix A sein, so dass wir eine ungenaue Lösung erhalten, obwohl die Aufgabenstellung an sich gutartig ist.

Um ein stabiles Verfahren zu erhalten, empfiehlt es sich deshalb, nach alternativen Zerlegungen Ausschau zu halten, die die Konditionszahl nicht unnötig vergrößern.

Ein sehr erfolgreicher Zugang sind *orthogonale Zerlegungen*.

Definition 4.1 (Orthogonale Matrix) *Eine Matrix $Q \in \mathbb{R}^{n\times n}$ nennen wir* orthogonal*, falls $Q^*Q = I$ gilt.*

Sei $Q \in \mathbb{R}^{n\times n}$. Aus der Definition ergibt sich unmittelbar, dass Q invertierbar ist mit $Q^{-1} = Q^*$. Aus $I = QQ^{-1} = QQ^*$ folgt dann, dass auch Q^{-1} orthogonal ist. Wir finden außerdem

$$\begin{aligned} \langle Qx, Qy\rangle_2 = \langle x, Q^*Qy\rangle_2 = \langle x, y\rangle_2 &\qquad \text{für alle } x, y \in \mathbb{R}^n, \\ \|Qx\|_2 = \sqrt{\langle Qx, Qx\rangle_2} = \sqrt{\langle x, x\rangle_2} = \|x\|_2 &\qquad \text{für alle } x \in \mathbb{R}^n, \end{aligned}$$

orthogonale Transformationen lassen also die Längen von Vektoren und die Winkel zwischen ihnen unverändert.

Für die Operatornorm ergeben sich daraus die Gleichungen

$$\begin{aligned} \|QB\|_{2\leftarrow 2} &= \max\left\{ \frac{\|QBx\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^m \setminus \{0\} \right\} \\ &= \max\left\{ \frac{\|Bx\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^m \setminus \{0\} \right\} = \|B\|_{2\leftarrow 2} \qquad \text{für alle } B \in \mathbb{R}^{n\times m}, \\ \|BQ^*\|_{2\leftarrow 2} &= \max\left\{ \frac{\|BQ^*x\|_2}{\|x\|_2} \;:\; x \in \mathbb{R}^n \setminus \{0\} \right\} \end{aligned}$$

$$= \max\left\{ \frac{\|By\|_2}{\|Qy\|_2} \;:\; y \in \mathbb{R}^n \setminus \{0\} \right\}$$
$$= \max\left\{ \frac{\|By\|_2}{\|y\|_2} \;:\; y \in \mathbb{R}^n \setminus \{0\} \right\} = \|B\|_{2\leftarrow 2} \qquad \text{für alle } B \in \mathbb{R}^{m\times n},$$

aus denen insbesondere

$$\kappa_2(A) = \|A\|_{2\leftarrow 2}\|A^{-1}\|_{2\leftarrow 2} = \|QA\|_{2\leftarrow 2}\|A^{-1}Q^{-1}\|_{2\leftarrow 2} = \kappa_2(QA)$$

folgt. Die Multiplikation mit einer orthogonalen Matrix lässt also die Konditionszahl völlig unverändert.

Definition 4.2 (QR-Zerlegung) *Sei $A \in \mathbb{R}^{n\times m}$. Ein Paar (Q, R) von Matrizen $Q \in \mathbb{R}^{n\times n}$, $R \in \mathbb{R}^{n\times m}$ nennen wir eine* QR-Zerlegung *der Matrix A, falls*

- *$A = QR$ gilt,*
- *Q eine orthogonale Matrix ist und*
- *R eine obere Dreiecksmatrix.*

Wie bereits gesehen gelten

$$\kappa_2(R) = \kappa_2(QR) = \kappa_2(A),$$
$$\kappa_2(Q) = \kappa_2(QI) = \kappa_2(I) = 1,$$

so dass wir Gleichungssysteme mit den Matrizen Q und R auflösen können, ohne eine allzu ungünstige Fehlerfortpflanzung befürchten zu müssen.

Falls uns eine QR-Zerlegung einer invertierbaren Matrix A zur Verfügung steht, können wir $Ax = b$ lösen, indem wir

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

nach y beziehungsweise x auflösen. Die zweite Aufgabe lässt sich wie zuvor per Rückwärtseinsetzen behandeln. Die erste Aufgabe ist sogar noch einfacher, denn da Q orthogonal ist, gilt

$$y = Q^*Qy = Q^*b,$$

wir müssen also lediglich eine Matrix-Vektor-Multiplikation durchführen.

Es stellt sich die Frage, ob wir QR-Zerlegungen von Matrizen praktisch konstruieren können. Ein sehr nützliches Hilfsmittel dabei sind *Householder-Spiegelungen*, mit deren Hilfe sich ein beliebiger Vektor auf ein Vielfaches des ersten kanonischen Einheitsvektors

$$e = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

abbilden lässt. Indem wir diese Spiegelungen der Reihe nach auf die Spalten einer Matrix A anwenden, können wir sie in obere Dreiecksform überführen, sogar falls es sich um eine allgemeine nicht-quadratische Matrix handelt.

Definition 4.3 (Householder-Spiegelung) *Sei $v \in \mathbb{R}^n \setminus \{0\}$. Die Matrix*

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

nennen wir die Householder-Spiegelung *zu dem Vektor v.*

Householder-Spiegelungen sind orthogonal, denn für jeden Vektor $v \in \mathbb{R}^n \setminus \{0\}$ gilt

$$\begin{aligned} Q_v^* Q_v = Q_v Q_v &= \left(I - 2\frac{vv^*}{v^*v}\right)\left(I - 2\frac{vv^*}{v^*v}\right) \\ &= I - 4\frac{vv^*}{v^*v} + 4\frac{vv^*vv^*}{v^*vv^*v} = I - 4\frac{vv^*}{v^*v} + 4\frac{vv^*}{v^*v} = I. \end{aligned}$$

Householder-Spiegelungen entsprechen auch unserer anschaulichen Vorstellung einer Spiegelung: Vektoren aus der auf v senkrecht stehenden Hyperebene

$$H := \{x \in \mathbb{R}^n \ : \ v^*x = 0\}$$

werden unverändert gelassen, während der Normalenvektor v wegen

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

gerade seine Richtung umkehrt.

Wir suchen Householder-Spiegelungen, mit denen wir eine Matrix $A \in \mathbb{R}^{n\times m}$ auf obere Dreiecksgestalt bringen können. Ein guter erster Schritt besteht darin, in der ersten Spalte

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

alle Einträge unterhalb der Diagonalen zu eliminieren. Dazu bräuchten wir einen Vektor $v \in \mathbb{R}^n \setminus \{0\}$ mit

$$Q_v a = \alpha e$$

für ein $\alpha \in \mathbb{R}$. Falls $a = 0$ gilt, leistet jede beliebige Householder-Spiegelung das Gewünschte. Ansonsten muss

$$\alpha e = Q_v a = a - 2\frac{vv^*a}{v^*v} = a - 2v\frac{v^*a}{v^*v}$$

gelten, der Householder-Vektor v muss also im Aufspann der Vektoren e und a liegen. Wenn wir diesen Ansatz weiter verfolgen, gelangen wir zu

$$v = \begin{pmatrix} a_{11} \pm \|a\|_2 \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}.$$

Wir müssen das Vorzeichen in der ersten Zeile so wählen, dass Auslöschung vermieden wird, so dass sich

$$v = \begin{pmatrix} a_{11} + \operatorname{sgn}(a_{11})\|a\|_2 \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix} = a + \operatorname{sgn}(a_{11})\|a\|_2 e, \qquad \operatorname{sgn}(x) = \begin{cases} 1 & \text{falls } x > 0, \\ -1 & \text{falls } x < 0, \\ 0 & \text{ansonsten} \end{cases}$$

ergibt. Mit diesem Vektor erhalten wir

$$\begin{aligned} v^*a &= (a + \operatorname{sgn}(a_{11})\|a\|_2 e)^* a = a^*a + \operatorname{sgn}(a_{11})\|a\|_2 e^* a \\ &= \|a\|_2^2 + \operatorname{sgn}(a_{11})\|a\|_2 a_{11}, \\ v^*v &= (a + \operatorname{sgn}(a_{11})\|a\|_2 e)^* (a + \operatorname{sgn}(a_{11})\|a\|_2 e) \\ &= a^*a + \operatorname{sgn}(a_{11})\|a\|_2 e^* a + \operatorname{sgn}(a_{11})\|a\|_2 a^* e + \operatorname{sgn}(a_{11})^2 \|a\|_2^2 e^* e \\ &= 2\|a\|_2^2 + 2\operatorname{sgn}(a_{11})\|a\|_2 a_{11} = 2v^*a, \\ Q_v a &= a - 2v\frac{v^*a}{v^*v} = a - v = -\operatorname{sgn}(a_{11})\|a\|_2 e, \end{aligned}$$

also ist $Q_v a$ tatsächlich ein Vielfaches des ersten kanonischen Einheitsvektors. Es folgt

$$Q_v A = \begin{pmatrix} -\operatorname{sgn}(a_{11})\|a\|_2 & A_{1*}^{(1)} \\ 0 & A_{**}^{(1)} \end{pmatrix}$$

mit geeigneten Matrizen $A_{1*}^{(1)} \in \mathbb{R}^{1\times(m-1)}$ und $A_{**}^{(1)} \in \mathbb{R}^{(n-1)\times(m-1)}$.

Im nächsten Schritt können wir nun eine zweite Householder-Spiegelung auf die erste Spalte der Teilmatrix $A_{**}^{(1)}$ anwenden, um alle Einträge unterhalb ihrer Diagonalen zu eliminieren. Diese Vorgehensweise können wir wiederholen, bis wir die gewünschte Dreiecksform erhalten haben.

Wenn wir die dabei konstruierten Householder-Vektoren mit Nulleinträgen auffüllen, können wir das Ergebnis als eine Faktorisierung

$$Q_{v^{(k)}} Q_{v^{(k-1)}} \dots Q_{v^{(1)}} A = R, \qquad k = \min\{n, m\}$$

schreiben. Da die Householder-Spiegelungen symmetrisch und orthogonal sind, sind sie selbstinvers, so dass

$$A = Q_{v^{(1)}} \dots Q_{v^{(k-1)}} Q_{v^{(k)}} R$$

folgt. Das Produkt orthogonaler Matrizen ist selbst orthogonal, so dass wir mit $Q := Q_{v^{(1)}} \dots Q_{v^{(k-1)}} Q_{v^{(k)}}$ die gewünschte Faktorisierung

$$A = QR$$

erhalten. Es bleibt noch zu klären, wie wir die Matrix Q möglichst effizient und sparsam speichern.

Dazu genügt es, die Householder-Vektoren $v^{(1)}, \ldots, v^{(k)}$ zu speichern. Die ersten $i-1$ Einträge eines Vektors $v^{(i)}$ sind dabei gleich null, brauchen also nicht gespeichert zu werden. Für die letzten $n-i$ Einträge können wir den Speicherplatz in der Matrix A verwenden, der durch die Elimination der Unterdiagonalelemente frei geworden ist. Es fehlt uns also nur noch ein Speicherplatz für den i-ten Eintrag des i-ten Householder-Vektors $v^{(i)}$.

Ein besonders eleganter Zugang besteht darin, auszunutzen, dass die Skalierung des Vektors für die Householder-Spiegelung irrelevant ist. Für jedes $\gamma \in \mathbb{R} \setminus \{0\}$ gilt nämlich

$$Q_{\gamma v} = I - 2\frac{\gamma^2 v v^*}{\gamma^2 v^* v} = I - 2\frac{v v^*}{v^* v} = Q_v.$$

Also dürfen wir die Householder-Vektoren so skalieren, dass der i-te Eintrag des i-ten Vektors jeweils gleich eins ist. Diese Eins brauchen wir genauso wenig zu speichern wie die ihr vorangestellten Nulleinträge, so dass unser Speicherproblem gelöst ist. Für eine quadratische Matrix, also im Fall $n = m$, stellen wir unsere QR-Zerlegung in der Form

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

dar, die sich gut eignet, um sowohl Q oder Q^* auf einen Vektor anzuwenden oder ein Gleichungssystem mit der Matrix R zu lösen. Im Fall $n > m$ fügen wir weitere Zeilen für die Householder-Vektoren hinzu, im Fall $n < m$ dagegen weitere Spalten für die Matrix R.

Dieser Skalierungsansatz hat noch eine weitere sehr willkommene Konsequenz: Es lässt sich leicht nachprüfen, dass der i-te Eintrag des i-ten Vektors jeweils der betragsgrößte des gesamten Vektors ist. Wenn wir ihn auf eins skalieren, können die Beträge aller anderen Einträge nicht größer als eins sein, so dass sich das Risiko eines Maschinenzahl-Überlaufs bei der Berechnung der Produkte v^*x und v^*v erheblich reduziert.

Um die Anwendung der Householder-Spiegelungen besonders effizient zu gestalten, werden häufig die Koeffizienten

$$\tau_i := \frac{2}{\|v^{(i)}\|_2^2} = \frac{2}{(v^{(i)})^* v^{(i)}}$$

zusätzlich abgespeichert, so dass sich die Spiegelungen kompakt in der Form

$$Q_{v^{(i)}} = I - \tau_i v^{(i)} (v^{(i)})^*$$

schreiben lassen und für ihre Anwendung auf einen Vektor lediglich ein Skalarprodukt und eine Linearkombination erforderlich sind.

Bei der Implementierung können (und sollten) wir wieder auf BLAS-Funktionen höherer Stufen zurückgreifen. Wenn wir beispielsweise die Matrix A in der Form

$$A = \begin{pmatrix} a_{11} & A_{1*} \\ a_{*1} & A_{**} \end{pmatrix}$$

zerlegen, müssen wir im ersten Schritt des Verfahrens einen Householder-Vektor $v^{(1)}$ mit

$$Q_{v^{(1)}} \begin{pmatrix} a_{11} \\ a_{*1} \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \end{pmatrix}$$

finden. Gemäß unserer Konstruktion ist dafür die Norm des Vektors zu berechnen (`dnrm2`, BLAS-Stufe 1), die wir zu a_{11} addieren oder von diesem Koeffizienten subtrahieren. Anschließend skalieren wir die a_{*1}, um die letzten $n-1$ Komponenten des Vektors $v^{(1)}$ zu erhalten (`dscal`, BLAS-Stufe 1). α können wir in a_{11} speichern und so die Berechnung der ersten Spalte der gewünschten Darstellung abschließen.

Für die Berechnung der verbliebenen Spalten müssen wir nun $Q_{v^{(1)}}$ auf die Restmatrix

$$\begin{pmatrix} A_{1*} \\ A_{**} \end{pmatrix}$$

anwenden. Diese Operation lässt sich durch eine Matrix-Vektor-Multiplikation (`dgemv`, BLAS-Stufe 2) mit der transponierten Matrix und ein Rang-1-Update ausdrücken (`dger`, BLAS-Stufe 2).

Um eine möglichst hohe Geschwindigkeit zu erreichen, wäre es besser, wenn wir mit Funktionen der Stufe 3 arbeiten könnten. Das ist möglich, gestaltet sich aber etwas schwieriger als bei der LR-Zerlegung, weil die Householder-Spiegelung jeweils auf vollständige Spalten der Matrix angewendet wird. Wir können allerdings wie folgt vorgehen: Wenn wir zwei Householder-Spiegelungen

$$Q_{v^{(1)}} = I - \tau_1 v^{(1)} (v^{(1)})^*, \qquad Q_{v^{(2)}} = I - \tau_2 v^{(2)} (v^{(2)})^*$$

multiplizieren, erhalten wir

$$\begin{aligned} Q_{v^{(2)}} Q_{v^{(1)}} &= Q_{v^{(2)}} - \tau_1 Q_{v^{(2)}} v^{(1)} (v^{(1)})^* = I - \tau_2 v^{(2)} (v^{(2)})^* - \tau_1 Q_{v^{(2)}} v^{(1)} (v^{(1)})^* \\ &= I - \begin{pmatrix} \tau_1 Q_{v^{(2)}} v^{(1)} & \tau_2 v^{(2)} \end{pmatrix} \begin{pmatrix} v^{(1)} & v^{(2)} \end{pmatrix}^*, \end{aligned}$$

können also das Produkt in der Form

$$Q_{v^{(2)}} Q_{v^{(1)}} = I - WV^*, \qquad W \in \mathbb{R}^{n \times 2},\ V \in \mathbb{R}^{m \times 2}$$

darstellen, die sich mit Hilfe der Matrix-Multiplikation verarbeiten lässt.

Dieser Ansatz lässt sich verallgemeinern: Sei $\ell \in \{1, \dots, k-1\}$ so gegeben, dass

$$Q_{v^{(\ell)}} \dots Q_{v^{(1)}} = I - WV^*, \qquad W \in \mathbb{R}^{n \times \ell},\ V \in \mathbb{R}^{n \times \ell}$$

gilt. Wenn wir $Q_{v^{(\ell+1)}}$ hinzu nehmen, erhalten wir

$$\begin{aligned} Q_{v^{(\ell+1)}} Q_{v^{(\ell)}} \dots Q_{v^{(1)}} &= Q_{v^{(\ell+1)}} (I - WV^*) = Q_{v^{(\ell+1)}} - Q_{v^{(\ell+1)}} WV^* \\ &= I - \tau_{\ell+1} v^{(\ell+1)} (v^{(\ell+1)})^* - Q_{v^{(\ell+1)}} WV^* \\ &= I - \begin{pmatrix} Q_{v^{(\ell+1)}} W & \tau_{\ell+1} v^{(\ell+1)} \end{pmatrix} \begin{pmatrix} V & v^{(\ell+1)} \end{pmatrix} \\ &= I - \widetilde{W} \widetilde{V}^* \end{aligned}$$

mit

$$\widetilde{W} := \begin{pmatrix} Q_{v^{(\ell+1)}} W & \tau_{\ell+1} v^{(\ell+1)} \end{pmatrix} \in \mathbb{R}^{n \times (\ell+1)}, \qquad \widetilde{V} := \begin{pmatrix} V & v^{(\ell+1)} \end{pmatrix} \in \mathbb{R}^{n \times (\ell+1)}.$$

Wir können nun eine cache-effiziente Implementierung auf Grundlage der Matrix-Multiplikation entwickeln: Die Matrix A wird in der Form

$$A = \begin{pmatrix} A_c & A_* \end{pmatrix}, \qquad A_c \in \mathbb{R}^{n \times c}, \qquad A_* \in \mathbb{R}^{n \times (m-c)}$$

zerlegt, wobei der Parameter $c \in \{1, \ldots, m\}$ so gewählt ist, dass drei $n \times c$-Matrizen und eine $c \times m$-Matrix in den Cache-Speicher passen. Wir berechnen zunächst c Householder-Spiegelungen, die eine QR-Zerlegung

$$Q_c R_c = Q_{v^{(1)}} \ldots Q_{v^{(c)}} R_c = A_c$$

der Matrix A_c darstellen. Die Matrix R_c enthält dann bereits die ersten c Spalten der Matrix R der vollständigen Zerlegung.

Anschließend konstruieren wir mit der oben dargestellten Methode Matrizen $W, V \in \mathbb{R}^{n \times c}$, die

$$Q_{v^{(1)}} \ldots Q_{v^{(c)}} = I - V W^*$$

erfüllen. Nun können wir

$$\widehat{A}_* := Q_c^* A_* = Q_{v^{(c)}} \ldots Q_{v^{(1)}} A_* = A_* - W V^* A_*$$

mit zwei Matrix-Multiplikationen

$$B_* := V^* A_*, \qquad \widehat{A}_* = A_* - W B_*$$

berechnen, also mit Funktionen der Stufe 3.

Sofern die Matrizen A_c, V und W in den Cache-Speicher passen, dürfen wir bei dieser Vorgehensweise eine hohe Effizienz erwarten. Die Cache-Ausnutzung lässt sich noch steigern, indem man jeweils nur c Spalten der Matrix B_* berechnet und das Update der Matrix A_* in entsprechenden Blöcken ausführt.

Dieser Ansatz erfordert zwar zusätzlichen Hilfsspeicher für die Matrizen W und V sowie das Produkt $V^* A_*$, kann sich aber trotzdem lohnen.

5 Eigenwertaufgaben

Eine eng mit linearen Gleichungssysteme verwandte Aufgabenstellung ist die Suche nach Eigenwerten und Eigenvektoren einer Matrix $A \in \mathbb{R}^{n \times n}$, also nach Paaren (λ, x) mit

$$Ax = \lambda x, \qquad x \neq 0.$$

Im Gegensatz zu linearen Gleichungssystemen ist es bei Eigenwertaufgaben im Allgemeinen unmöglich, einen mathematischen Algorithmus anzugeben, der nach einer endlichen Anzahl von Schritten das exakte Ergebnis (unter Vernachlässigung der Rundungsfehler) berechnet.

Deshalb müssen approximative Algorithmen zum Einsatz kommen, also solche, die lediglich eine beliebig genaue Näherungslösung ermitteln. Für Eigenwertaufgaben sind dabei vor allem Iterationsverfahren sehr erfolgreich.

5.1 Vektoriteration

Das einfachste Verfahren für die approximative Berechnung von Eigenwerten und Eigenvektoren ist die *Vektoriteration* (engl. *power iteration*), die auch als *Von-Mises-Iteration* bekannt ist.

Sie beruht auf der Idee, sich die unterschiedliche Größe der Eigenwerte zunutze zu machen. Zur Motivation gehen wir davon aus, dass eine Matrix $A \in \mathbb{R}^{n \times n}$ zwei Eigenwerte $\lambda_1, \lambda_2 \in \mathbb{R}$ mit zugehörigen Eigenvektoren $x_1, x_2 \in \mathbb{R}^n \setminus \{0\}$ besitzt. Dann gilt für die Vektoren $y^{(0)} := x_1 + x_2$ und $y^{(1)} := Ay$ die Gleichung

$$y^{(1)} = Ay^{(0)} = A(x_1 + x_2) = Ax_1 + Ax_2 = \lambda_1 x_1 + \lambda_2 x_2.$$

Falls $|\lambda_1|$ größer als $|\lambda_2|$ ist, wird der Vektor x_1 in dem Vektor $y^{(1)}$ stärker vertreten sein als in dem Vektor $y^{(0)}$. Der Vektor $y^{(1)}$ ist also in einem geeigneten Sinn „einem Eigenvektor näher“ als der Vektor $y^{(0)}$.

Diese Beobachtung legt es nahe, die Matrix A mehrfach mit einem Anfangsvektor $y^{(0)}$ zu multiplizieren, um die Folge

$$y^{(m+1)} := Ay^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0$$

zu konstruieren. Dabei handelt es sich um die einfachste Form der Vektoriteration, bei der letzten Endes die Vektoren $y^{(m)} = A^m y^{(0)}$ der Reihe nach berechnet werden.

Diese Form hat allerdings den Nachteil, dass es bei der praktischen Implementierung mit Maschinenzahlen für $|\lambda_1| > 1$ zu einem Überlauf kommen kann, bei dem zu große Komponenten des Vektors $y^{(m)}$ durch ∞ oder $-\infty$ ersetzt werden. Für $|\lambda_1| < 1$ kann es

entsprechend zu einem „Unterlauf“ kommen, bei dem zu kleine Komponenten durch null ersetzt werden. Diese Schwierigkeiten lassen sich vermeiden, indem wir eine Skalierung einführen: Wenn x ein Eigenvektor ist, ist αx mit $\alpha \neq 0$ ebenfalls einer. Üblich ist eine Skalierung, die dafür sorgt, dass die auftretenden Vektoren Einheitsvektoren sind.

Definition 5.1 (Vektoriteration) *Sei $A \in \mathbb{R}^{n\times n}$. Sei $y^{(0)} \in \mathbb{R}^n$ mit $\|y^{(0)}\| = 1$ gegeben. Die* Vektoriteration *berechnet die Vektoren*

$$z^{(m)} := Ay^{(m)}, \qquad y^{(m+1)} := \frac{z^{(m)}}{\|z^{(m)}\|} \qquad \text{für alle } m \in \mathbb{N}_0.$$

In dieser abstrakten Form lässt sich die Vektoriteration nicht implementieren, da sie unendlich viele Vektoren berechnet. In der Praxis kommen deshalb *Abbruchkriterien* zum Einsatz, die festlegen, wann wir mit der letzten berechneten Näherungslösung zufrieden sind und die Iteration abbrechen. Das einfachste Kriterium besteht natürlich darin, schlicht die Anzahl der Schritte vorzugeben.

Dafür müssten wir allerdings abschätzen können, wie schnell die Näherungslösungen der exakten Lösung approximieren. Wir brauchen also Aussagen über die *Konvergenz* des Verfahrens.

Zur Vereinfachung nehmen wir an, dass A eine symmetrische Matrix ist. Dann existiert eine Orthonormalbasis $(x_i)_{i=1}^n$ von Eigenvektoren zu Eigenwerten $(\lambda_i)_{i=1}^n$, wir haben also

$$\begin{aligned} Ax_i &= \lambda_i x_i && \text{für alle } i \in \{1,\dots,n\},\\ x_i^* x_j &= \begin{cases} 1 & \text{falls } i = j,\\ 0 & \text{ansonsten} \end{cases} && \text{für alle } i,j \in \{1,\dots,n\}. \end{aligned}$$

Da es sich um eine Basis handelt, können wir jeden beliebigen Vektor $y \in \mathbb{R}^n$ in der Form

$$y = \sum_{j=1}^n \alpha_j x_j$$

darstellen, und da die Basisvektoren orthonormal sind, gilt auch

$$\alpha_i = \sum_{j=1}^n \alpha_j x_i^* x_j = x_i^* y \qquad \text{für alle } i \in \{1,\dots,n\},$$

so dass wir die *Parseval-Gleichung*

$$y = \sum_{i=1}^n (x_i^* y) x_i \tag{5.1}$$

erhalten. Die euklidische Norm des Vektors kann gemäß

$$\|y\|_2^2 = y^* y = \sum_{i=1}^n \sum_{j=1}^n (x_j^* y)(x_i^* y) x_i^* x_j = \sum_{i=1}^n (x_i^* y)^2 = \sum_{i=1}^n \alpha_i^2$$

berechnet werden.

Wir können nicht erwarten, dass die Vektoren $y^{(m)}$ im konventionellen Sinn konvergieren. Falls beispielsweise $\lambda_1 = -1$ gilt, wird sich das Vorzeichen in jedem Schritt umkehren.

Wir können allerdings den *Winkel* zwischen den Vektoren $y^{(m)}$ und dem Eigenvektor x_1 als Maß der Konvergenz verwenden. Der Winkel zwischen einem Vektor $y \in \mathbb{R}^n \setminus \{0\}$ und x_1 ist durch

$$\cos \angle(y, x_1) := \frac{x_1^* y}{\|y\|_2}$$

gegeben. Entsprechend der üblichen Rechenregeln für trigonometrische Funktionen erhalten wir auch

$$\sin \angle(y, x_1) := \sqrt{1 - \cos^2 \angle(y, x_1)} = \sqrt{\frac{y^* y - (x_1^* y)^2}{\|y\|_2^2}},$$
$$\tan \angle(y, x_1) := \frac{\sin \angle(y, x_1)}{\cos \angle(y, x_1)} = \sqrt{\frac{y^* y - (x_1^* y)^2}{(x_1^* y)^2}}.$$

Lemma 5.2 (Konvergenz) *Es gelte*

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

Sei $y \in \mathbb{R}^n$ ein Vektor mit $x_1^* y \neq 0$, also $\cos \angle(y, x_1) \neq 0$. Der Vektor $z := Ay$ erfüllt dann

$$\tan \angle(z, x_1) \leq \frac{|\lambda_2|}{|\lambda_1|} \tan \angle(y, x_1).$$

Beweis. Wir setzen

$$\alpha_i := x_i^* y, \qquad \beta_i := \lambda_i \alpha_i \qquad \text{für alle } i \in \{1, \ldots, n\}$$

und erhalten mit (5.1) die Gleichung

$$z = Ay = A \sum_{i=1}^n \alpha_i x_i = \sum_{i=1}^n \alpha_i A x_i = \sum_{i=1}^n \alpha_i \lambda_i x_i = \sum_{i=1}^n \beta_i x_i.$$

Mit den Gleichungen

$$\|y\|_2^2 = \sum_{i=1}^n \alpha_i^2, \qquad x_1^* y = \alpha_1,$$
$$\|z\|_2^2 = \sum_{i=1}^n \beta_i^2, \qquad x_1^* z = \beta_1$$

erhalten wir

$$\tan^2 \angle(z, x_1) = \frac{z^* z - (x_1^* z)^2}{(x_1^* z)^2} = \frac{\sum_{i=1}^n \beta_i^2 - \beta_1^2}{\beta_1^2} = \frac{\sum_{i=2}^n \beta_i^2}{\beta_1^2}$$

$$= \frac{\sum_{i=2}^n \lambda_i^2 \alpha_i^2}{\lambda_1^2 \alpha_1^2} \leq \frac{\lambda_2^2}{\lambda_1^2} \frac{\sum_{i=2}^n \alpha_i^2}{\alpha_1^2} = \left(\frac{|\lambda_2|}{|\lambda_1|}\right)^2 \tan^2 \angle(y, x_1).$$

■

Auf den Fall der Vektoriteration übertragen bedeutet dieses Ergebnis, dass

$$\tan \angle(y^{(m)}, x_1) \leq \left(\frac{|\lambda_2|}{|\lambda_1|}\right)^m \tan \angle(y^{(0)}, x_1) \qquad \text{für alle } m \in \mathbb{N}_0$$

gilt. Falls λ_1 ein *dominanter Eigenwert* ist, falls also $|\lambda_1| > |\lambda_2|$ gilt, folgt

$$\lim_{m \to \infty} \tan \angle(y^{(m)}, x_1) = 0,$$

der Winkel zwischen $y^{(m)}$ und dem Eigenvektor x_1 wird also gegen null streben.

5.2 Inverse Iteration

Die Vektoriteration erlaubt es uns, einen Eigenvektor zu dem betragsgrößten Eigenwert einer Matrix zu berechnen. In der Praxis ist man meistens eher an dem betragskleinsten Eigenwert interessiert, gelegentlich auch an dem Eigenwert, der einem vorgebenen Wert am nächsten liegt.

Für derartige Fragestellungen ist die *inverse Iteration* nützlich, die auf der einfachen Idee beruht, die Vektoriteration auf ein geeignete inverse Matrix anzuwenden.

Lemma 5.3 (Inverse) *Sei $A \in \mathbb{R}^{n \times n}$, sei $\mu \in \mathbb{R}$ derart gegeben, dass $A - \mu I$ invertierbar ist.*

Dann gilt

$$Ax = \lambda x \iff (A - \mu I)^{-1} x = \frac{1}{\lambda - \mu} x \qquad \text{für alle } x \in \mathbb{R}^n \setminus \{0\}.$$

Beweis. Sei $x \in \mathbb{R}^n \setminus \{0\}$. Es gilt

$$Ax = \lambda x \iff (A - \mu I)x = (\lambda - \mu)x \iff x = (\lambda - \mu)(A - \mu I)^{-1} x.$$

Falls nun $Ax = \lambda x$ gilt, folgt mit $x \neq 0$ bereits $\lambda - \mu \neq 0$, so dass wir per Division die gewünschte Gleichung erhalten.

Falls dagegen $(A - \mu I)^{-1} x = \frac{1}{\lambda - \mu} x$ gilt, erhalten wir durch Multiplikation mit $\lambda - \mu$ die rechte Seite unserer Äquivalenzkette, und $Ax = \lambda x$ folgt unmittelbar. ■

Indem wir dieses Lemma auf $\mu = 0$ anwenden, erhalten wir ein Iterationsverfahren für die Berechnung des betragskleinsten Eigenwerts.

Definition 5.4 (Inverse Iteration) *Seien $A \in \mathbb{R}^{n \times n}$ und $\mu \in \mathbb{R}$ so gegeben, dass $A - \mu I$ invertierbar ist. Sei $y^{(0)} \in \mathbb{R}^n$ mit $\|y^{(0)}\| = 1$ gegeben. Die* inverse Iteration *berechnet die Vektoren*

$$z^{(m)} := (A - \mu I)^{-1} y^{(m)}, \qquad y^{(m+1)} := \frac{z^{(m)}}{\|z^{(m)}\|} \qquad \text{für alle } m \in \mathbb{N}_0.$$

Die Zahl μ wird als Shift-Parameter *bezeichnet.*

Gelegentlich bezieht sich der Name inverse Iteration *nur auf den Fall $\mu = 0$, während für $\mu \neq 0$ der Name* inverse Iteration mit Shift *verwendet wird.*

Aus Lemma 5.2 folgt mit Lemma 5.3 unmittelbar eine Konvergenzaussage.

Folgerung 5.5 (Konvergenz) *Es gelte*

$$|\lambda_1 - \mu| \leq |\lambda_2 - \mu| \leq \ldots \leq |\lambda_n - \mu|.$$

Sei $y \in \mathbb{R}^n$ ein Vektor mit $x_1^* y \neq 0$, also $\cos \angle(y, x_1) \neq 0$. Der Vektor $z := (A - \mu I)^{-1} y$ erfüllt dann

$$\tan \angle(z, x_1) \leq \frac{|\lambda_1 - \mu|}{|\lambda_2 - \mu|} \tan \angle(y, x_1).$$

Beweis. Nach Lemma 5.3 sind die Eigenwerte der Matrix $\widehat{A} := (A - \mu I)^{-1}$ durch

$$\widehat{\lambda}_i := \frac{1}{\lambda_i - \mu} \qquad \text{für alle } i \in \{1, \ldots, n\}$$

gegeben. Es folgt

$$|\widehat{\lambda}_1| \geq |\widehat{\lambda}_2| \geq \ldots \geq |\widehat{\lambda}_n|.$$

Also dürfen wir Lemma 5.2 anwenden und erhalten

$$\tan \angle(z, x_1) \leq \frac{|\widehat{\lambda}_2|}{|\widehat{\lambda}_1|} \tan \angle(y, x_1) = \frac{|\lambda_1 - \mu|}{|\lambda_2 - \mu|} \tan \angle(y, x_1).$$

■

Insbesondere konvergiert die inverse Iteration, falls $|\lambda_1 - \mu| < |\lambda_2 - \mu|$ gilt, falls also ein einfacher Eigenwert näher an μ liegt als alle anderen.

Bemerkung 5.6 (Implementierung) *Bei der praktischen Implementierung der inversen Iteration empfiehlt es sich in der Regel, auf die explizite Berechnung der Inversen $(A - \mu I)^{-1}$ zu verzichten.*

Stattdessen können wir beispielsweise eine Faktorisierung der Matrix $A - \mu I$ verwenden, mit deren Hilfe sich die linearen Gleichungssysteme

$$(A - \mu I) z^{(m)} = y^{(m)}$$

effizient lösen lassen.

5.3 Rayleigh-Quotient

Mit Hilfe der Vektoriteration oder der inversen Iteration können wir eine Folge von Vektoren berechnen, die gegen einen Eigenvektor konvergieren.

Es wäre nützlich, wenn wir beurteilen könnten, wie nahe wir bei diesem Prozess schon einem Eigenvektor gekommen sind. Der einfachste Weg bestünde darin, die *Defektnorm*

$$\|Ay - \lambda y\|$$

zu untersuchen. Falls sie klein ist, dürfte y einem Eigenvektor zu dem Eigenwert λ nahe kommen.

In der Praxis kennen wir allerdings häufig den Eigenwert nicht, zu dem ein approximativer Eigenvektor gehört. Wir können ihn allerdings rekonstruieren: Nehmen wir an, dass $x \in \mathbb{R}^n \setminus \{0\}$ ein Eigenvektor zu einem Eigenwert $\lambda \in \mathbb{R}$ ist. Dann gelten die Gleichungen

$$\lambda x = Ax, \qquad \lambda x^* x = x^* Ax, \qquad \lambda = \frac{x^* Ax}{x^* x},$$

und dank $x^* x = \|x\|_2^2 \neq 0$ können wir mit der letzten Gleichung den Eigenwert λ berechnen.

Definition 5.7 (Rayleigh-Quotient) *Die Abbildung*

$$\Lambda_A : \mathbb{R}^n \setminus \{0\} \to \mathbb{R}, \qquad x \mapsto \frac{x^* Ax}{x^* x},$$

nennen wir die Rayleigh-Quotientenabbildung.

Für einen Vektor $x \in \mathbb{R}^n \setminus \{0\}$ nennen wir $\Lambda_A(x)$ den Rayleigh-Quotienten *zu A und x.*

Mit Hilfe des Rayleigh-Quotienten können wir zu einer Näherung $y \in \mathbb{R}^n \setminus \{0\}$ eines Eigenvektors eine Näherung $\lambda \in \mathbb{R}$ des korrespondierenden Eigenwerts berechnen und erhalten den folgenden Algorithmus:

$y \leftarrow y/\|y\|_2$;
Löse $(A - \mu I)z = y$;
$\widehat{\lambda} \leftarrow y^* z$;
while $\|z - \widehat{\lambda} y\| \leq \epsilon$ **do**
  $y \leftarrow z/\|z\|_2$;
  Löse $(A - \mu I)z = y$;
  $\widehat{\lambda} \leftarrow y^* z$
end

Hier ist $\epsilon \in \mathbb{R}_{>0}$ ein frei wählbarer Parameter, der festlegt, wie genau Eigenwert und Eigenvektor berechnet werden sollen. Infolge der zu erwartenden Rundungsfehler sollte

in der Praxis natürlich darauf geachtet werden, ϵ nicht zu gering zu wählen, um eine Endlosschleife zu vermeiden.

Wir können den Rayleigh-Quotienten sogar verwenden, um die inverse Iteration zu beschleunigen: Falls y eine hinreichend gute Näherung des gesuchten Eigenvektors ist, ist $\Lambda_A(y)$ eine gute Näherung des korrespondierenden Eigenwerts λ_1. Gemäß Folgerung 5.5 ist die Konvergenz der inversen Iteration um so schneller, je näher μ an λ_1 liegt, also bietet es sich an, $\mu = \Lambda_A(y)$ zu setzen.

Definition 5.8 (Rayleigh-Iteration) *Seien $A \in \mathbb{R}^{n\times n}$. Sei $y^{(0)} \in \mathbb{R}^n$ mit $\|y^{(0)}\|_2 = 1$ gegeben. Die* Rayleigh-Iteration *berechnet die Folgen*

$$\mu_m := (y^{(m)})^* A y^{(m)}, \qquad z^{(m)} := (A - \mu_m I)^{-1} y^{(m)},$$
$$y^{(m+1)} := \frac{z^{(m)}}{\|z^{(m)}\|_2} \qquad \text{für alle } m \in \mathbb{N}_0.$$

Das Verfahren kann abbrechen, falls $A - \mu_m I$ nicht invertierbar ist. In diesem Fall ist μ_m ein Eigenwert zu dem Eigenvektor $y^{(m)}$.

Eine praktische Implementierung der Rayleigh-Iteration könnte die folgende Gestalt annehmen:

$y \leftarrow y/\|y\|_2$;
Löse $(A - \mu I)z = y$;
$\widehat{\lambda} \leftarrow y^* z$;
while $\|z - \widehat{\lambda} y\| \leq \epsilon$ **do**
  $y \leftarrow z/\|z\|_2$;
  $\mu \leftarrow y^* A y$;
  Löse $(A - \mu I)z = y$;
  $\widehat{\lambda} \leftarrow y^* z$
end

Ein Nachteil der Rayleigh-Iteration gegenüber der inversen Iteration soll hier nicht verschwiegen werden: Da sich der Shift-Parameter in jedem Schritt ändert, muss in jedem Schritt ein Gleichungssystem mit einer neuen Matrix gelöst werden. Falls beispielsweise eine LR- oder QR-Faktorisierung eingesetzt wird, muss diese Faktorisierung in jedem Schleifendurchlauf neu berechnet werden, während es bei der inversen Iteration ausreicht, sie einmal vor dem Eintritt in die Schleife zu ermitteln.

Die Suche nach Konvergenzaussagen für die Rayleigh-Iteration gestaltet sich etwas schwieriger als für die inverse Iteration, weil ein ungünstig gewählter Anfangsvektor $y^{(0)}$ dazu führen kann, dass das Verfahren gegen einen anderen Eigenwert als λ_1 konvergiert.

Falls allerdings der Winkel zwischen $y^{(0)}$ und x_1 hinreichend klein ist, kann man zeigen, dass das Verfahren *quadratisch konvergent* ist, dass also eine Konstante $c \in \mathbb{R}_{\geq 0}$ so existiert, dass

$$\tan \angle(y^{(m+1)}, x_1) \leq c \tan^2 \angle(y^{(m)}, x_1) \qquad \text{für alle } m \in \mathbb{N}_0$$

gilt. Falls $c \tan \angle(y^{(0)}, x_1) < 1$ gilt, konvergiert die Rayleigh-Iteration erheblich schneller als die inverse Iteration. Als Beispiel nehmen wir an, dass $c \tan \angle(y^{(0)}, x_1) < 1/2$ gilt. Mit unserer Abschätzung folgt

$$\begin{aligned} c \tan \angle(y^{(1)}, x_1) &\leq c^2 \tan^2 \angle(y^{(0)}, x_1) < 1/4, \\ c \tan \angle(y^{(2)}, x_1) &\leq c^2 \tan^2 \angle(y^{(1)}, x_1) < 1/16, \\ c \tan \angle(y^{(3)}, x_1) &\leq c^2 \tan^2 \angle(y^{(2)}, x_1) < 1/256, \end{aligned}$$

so dass jeder Schritt des Verfahrens die Anzahl der korrekt berechneten Nachkommastellen verdoppelt.

Falls A eine symmetrische Matrix ist, ist die Rayleigh-Iteration sogar *kubisch konvergent*, es findet sich nämlich eine Konstante $c \in \mathbb{R}_{\geq 0}$ derart, dass

$$\tan \angle(y^{(m+1)}, x_1) \leq c \tan^3 \angle(y^{(m)}, x_1) \qquad \text{für alle } m \in \mathbb{N}_0$$

gilt. In diesem Fall *verdreifacht* jeder Schritt des Verfahrens die Anzahl der korrekten Nachkommastellen. Da es beispielsweise bei IEEE-Maschinenzahlen doppelter Genauigkeit nur 52 Nachkommastellen gibt, kann ein kubisch konvergenter Algorithmus sehr schnell die Maschinengenauigkeit erreichen.

5.4 Hessenberg-Form

Die inverse Iteration und vor allem die Rayleigh-Iteration können sehr schnell konvergieren, falls die Shift-Parameter geeignet gewählt werden.

Allerdings ist ein einzelner Schritt dieser Iterationen relativ aufwendig, beispielsweise muss in jedem Schritt der Rayleigh-Iteration jeweils eine neue Matrix faktorisiert und ein Gleichungssystem gelöst werden. Im Allgemeinen verursacht das einen Rechenaufwand *pro Schritt*, der *kubisch* mit der Matrixdimension wächst, so dass dieses Verfahren in der beschriebenen Form nur im Ausnahmefall eingesetzt werden kann.

Wir können allerdings die Situation erheblich verbessern, indem wir die Matrix in einer Vorbereitungsphase in eine Form bringen, die wir effizienter handhaben können. Auf den ersten Blick könnte man daran denken, die Matrix in Dreiecksform zu bringen, das würde allerdings bereits dem Lösen des gesamten Eigenwertproblems entsprechen und ist deshalb schwierig.

Deshalb bringen wir die Matrix nur „fast“ auf obere Dreiecksform: Wir erlauben auch unmittelbar unterhalb die Diagonalen noch Einträge.

Definition 5.9 (Hessenberg-Matrix) *Eine Matrix $A \in \mathbb{R}^{n \times n}$ bezeichnen wir als* Hessenberg-Matrix *oder als* in Hessenberg-Form*, falls*

$$a_{ij} = 0 \qquad \textit{für alle } i, j \in [1:n],\ i \geq j+1$$

gilt.

Wir schreiben Hessenberg-Matrizen $H \in \mathbb{R}^{n\times n}$ in der Form

$$H = \begin{pmatrix} h_{11} & h_{12} & \dots & h_{1n} \\ h_{21} & \ddots & \ddots & \vdots \\ & \ddots & \ddots & h_{n-1,n} \\ & & h_{n,n-1} & h_{nn} \end{pmatrix},$$

bei der die Nulleinträge unterhalb der Nebendiagonalen weggelassen werden.

Gleichungssysteme mit Hessenberg-Matrizen lassen sich relativ einfach lösen, beispielsweise können wir den ersten Nebendiagonaleintrag h_{21} mit der Transformation

$$\begin{pmatrix} 1 & & & & \\ -h_{21}/h_{11} & 1 & & & \\ & 0 & \ddots & & \\ & & \ddots & 1 & \\ & & & 0 & 1 \end{pmatrix} H = \begin{pmatrix} h_{11} & h_{12} & \dots & \dots & h_{1n} \\ 0 & h_{22}^{(1)} & h_{23}^{(1)} & \dots & h_{2n}^{(1)} \\ & h_{32} & \ddots & \ddots & h_{3n} \\ & & \ddots & \ddots & \vdots \\ & & & h_{n,n-1} & h_{nn} \end{pmatrix}$$

eliminieren, die lediglich die zweite Zeile der Matrix verändert und deshalb nur $\mathcal{O}(n)$ Operationen erfordert. Die Restmatrix

$$H_{**} = \begin{pmatrix} h_{22}^{(1)} & h_{23}^{(1)} & \dots & h_{2n}^{(1)} \\ h_{32} & \ddots & \ddots & h_{3n} \\ & \ddots & \ddots & \vdots \\ & & h_{n,n-1} & h_{nn} \end{pmatrix}$$

ist wieder eine Hessenberg-Matrix, so dass wir induktiv fortfahren können, um die LR-Zerlegung in $\mathcal{O}(n^2)$ Operationen zu berechnen. Im Vergleich zu den ungefähr $\frac{2}{3}n^3$ Operationen, die die LR-Zerlegung einer allgemeinen Matrix erfordert, haben wir also erheblich Arbeit eingespart.

Die QR-Zerlegung einer Hessenberg-Matrix lässt sich in ähnlicher Weise berechnen: Die Householder-Transformationen müssen lediglich auf zweidimensionale Vektoren angewendet werden, da nur die Nebendiagonale eliminiert werden muss. Damit benötigt auch die QR-Zerlegung einer Hessenberg-Matrix nur $\mathcal{O}(n^2)$ Operationen.

Demzufolge lassen sich beispielsweise die inverse Iteration oder die Rayleigh-Iteration für Hessenberg-Matrizen sehr effizient durchführen.

Im Allgemeinen werden wir allerdings mit Matrizen konfrontiert sein, die keine Hessenberg-Matrizen sind. In diesem Fall können wir allerdings diese Eigenschaft *herstellen*: Für jede Matrix $A \in \mathbb{R}^{n\times n}$ lassen sich eine orthogonale Matrix $Q \in \mathbb{R}^{n\times n}$ und eine Hessenberg-Matrix $H \in \mathbb{R}^{n\times n}$ finden, die

$$QAQ^* = H$$

erfüllen. Die Überführung von A in H nennen wir eine *Hessenberg-Transformation*.

Die Hessenberg-Transformation ist eine *Ähnlichkeitstransformation*: Falls $x \in \mathbb{R}^n \setminus \{0\}$ ein Eigenvektor der Matrix A zu dem Eigenwert $\lambda \in \mathbb{R}$ ist, ist $\widehat{x} := Qx$ wegen

$$H\widehat{x} = QAQ^*Qx = QAx = \lambda Qx = \lambda\widehat{x}$$

ein Eigenvektor der Matrix H zu demselben Eigenwert. Falls umgekehrt $\widehat{x} \in \mathbb{R}^n \setminus \{0\}$ ein Eigenvektor der Matrix H zu dem Eigenwert λ ist, ist $x := Q^*\widehat{x}$ wegen

$$Ax = AQ^*\widehat{x} = Q^*QAQ^*\widehat{x} = Q^*H\widehat{x} = \lambda Q^*\widehat{x} = \lambda x$$

ein Eigenvektor der Matrix A zu demselben Eigenwert.

Die Transformation lässt also die Eigenwerte unverändert und bietet uns eine einfache Möglichkeit, die Eigenvektoren der Matrizen A und H inenander zu überführen.

Die Konstruktion der Hessenberg-Transformation kann mit Householder-Spiegelungen bewerkstelligt werden: Wir beginnen mit einer allgemeinen Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}$$

und wählen eine Householder-Spiegelung $Q_{v^{(1)}}$, die

$$a = \begin{pmatrix} a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}$$

auf ein Vielfaches des ersten kanonischen Einheitsvektors

$$Q_{v^{(1)}}a = \begin{pmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

abbildet. Es folgt

$$\begin{pmatrix} 1 & \\ & Q_{v^{(1)}} \end{pmatrix} A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ \alpha & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ & \vdots & \ddots & \vdots \\ & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{pmatrix}.$$

Um eine Ähnlichkeitstransformation zu erhalten, müssen wir $Q_{v^{(1)}}^* = Q_{v^{(1)}}$ auch von rechts mit der Matrix multiplizieren und erhalten

$$\begin{pmatrix} 1 & \\ & Q_{v^{(1)}} \end{pmatrix} A \begin{pmatrix} 1 & \\ & Q_{v^{(1)}}^* \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12}^{(2)} & \dots & a_{1n}^{(2)} \\ \alpha & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ & \vdots & \ddots & \vdots \\ & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix}.$$

Dabei ist wichtig, dass diese zweite Transformation die erste Spalte nicht verändert, so dass unsere im ersten Schritt gewonnenen Nulleinträge erhalten bleiben.

Wie aus einigen vorangegangenen Algorithmen bekannt können wir nun mit der Matrix

$$A_{**} = \begin{pmatrix} a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \ddots & \vdots \\ a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix}$$

induktiv fortfahren, um schließlich die gewünschte Hessenberg-Form zu erreichen.

Da jede Householder-Spiegelung einmal von links und einmal von rechts mit der jeweils aktuellen Matrix multipliziert wird, ist der Rechenaufwand für die Hessenberg-Transformation ungefähr doppelt so hoch wie für die QR-Zerlegung. Gerade bei der Rayleigh-Iteration lohnt sich diese Mehraufwand allerdings, weil er es ermöglicht, einen Schritt der Iteration in $\mathcal{O}(n^2)$ statt $\mathcal{O}(n^3)$ Operationen durchzuführen.

Besonders attraktiv ist die Householder-Transformation für symmetrische Matrizen: Falls $A = A^*$ gilt, haben wir

$$H = QAQ^* = QA^*Q^* = (QAQ^*)^* = H^*,$$

so dass wir eine symmetrische Hessenberg-Matrix erhalten. Aus $j \geq i+1$ folgt $h_{ij} = h_{ji} = 0$, also verschwinden auch oberhalb der oberen Nebendiagonale alle Einträge, so dass wir sogar eine symmetrische *Tridiagonalmatrix*

$$H = \begin{pmatrix} h_{11} & h_{21} & & \\ h_{21} & \ddots & \ddots & \\ & \ddots & \ddots & h_{n,n-1} \\ & & h_{n,n-1} & h_{nn} \end{pmatrix}$$

konstruiert haben. Man kann sich überlegen, dass sich LR- und QR-Zerlegung einer solchen Matrix in $\mathcal{O}(n)$ Operationen berechnen lassen, indem man wieder ausnutzt, dass „fast alle“ Einträge bereits gleich null sind.

6 Iterationsverfahren

Wir haben bereits Iterationsverfahren für die Behandlung von Eigenwertproblemen kennen gelernt: Ausgehend von einer Anfangsnäherung der gesuchten Lösung berechnet ein solches Verfahren eine Folge von Näherungslösungen, die hoffentlich immer besser werden.

Iterationsverfahren sind auch für viele allgemeinere Aufgaben häufig der einzige oder der einzige praktikable Weg, eine Lösung wenigstens approximativ zu bestimmen.

6.1 Iterationsverfahren

Im Allgemeinen dienen Iterationsverfahren der Suche nach einem Element x_* einer Menge V. Beispielsweise suchen wir bei Eigenwertaufgaben nach einem Eigenvektor $x_* \in \mathbb{R}^n \setminus \{0\}$ und einem Eigenwert $\lambda \in \mathbb{R}$ mit $Ax_* = \lambda x_*$. Das Paar (x_*, λ) in ein Element der Menge $U = (\mathbb{R}^n \setminus \{0\}) \times \mathbb{R}$.

Ein weiteres Beispiel ist die Quadratwurzel einer Zahl $\alpha \in \mathbb{R}_{>0}$, die ein Element x_* der Menge $U = \mathbb{R}_{>0}$ ist, das sich beispielsweise über die Gleichung $x_*^2 = \alpha$ beschreiben lässt.

Ein abstraktes Iterationsverfahren können wir durch eine Abbildung

$$\Phi : U \to U$$

beschreiben, die einer gegebenen Näherungslösung eine verbesserte Näherung zuordnet.

Eine minimale Anforderung an ein solches Verfahren ist, dass es die gesuchte Lösung $x_* \in U$ nicht verschlechtert, dass also

$$\Phi(x_*) = x_*$$

gilt. Ein solches x_* nennt man einen *Fixpunkt* der Abbildung Φ.

Für eine Eigenwertaufgabe könnten wir beispielsweise $\lambda = \Lambda_A(x)$ mit Hilfe des Rayleigh-Quotienten rekonstruieren und das Verfahren

$$\Phi : \mathbb{R}^n \setminus \{0\} \to \mathbb{R}^n \setminus \{0\}, \qquad x \mapsto \frac{Ax}{\Lambda_A(x)},$$

definieren, das sämtliche Eigenvektoren als Fixpunkte besitzt.

Für die Quadratwurzel könnten wir

$$\Phi : \mathbb{R}_{>0} \to \mathbb{R}_{>0}, \qquad x \mapsto x + (\alpha - x^2), \tag{6.1a}$$

oder erheblich besser

$$\Phi : \mathbb{R}_{>0} \to \mathbb{R}_{>0}, \qquad x \mapsto x + \frac{\alpha - x^2}{2x}, \tag{6.1b}$$

verwenden.

Offenbar ist es nicht schwer, eine Iterationsabbildung zu finden. Die Frage ist, ob eine solche Abbildung auch die Eigenschaft besitzt, Näherungslösungen zu verbessern.

Erinnerung 6.1 (Mittelwertsatz der Differentialrechnung) *Sei eine stetig differenzierbare Funktion $g \in C^1[a,b]$ gegeben. Dann existiert ein $\eta \in (a,b)$ mit*

$$g(b) - g(a) = (b-a)g'(\eta).$$

Mit Hilfe des Mittelwertsatzes der Differentialrechnung lässt sich immerhin für stetig differenzierbare Abbildungen eine Antwort finden: Sei $\Phi \in C^1[a,b]$, sei $x^* \in [a,b]$ der gesuchte Fixpunkt, und sei $x \in [a,b]$ die aktuelle Näherungslösung. Nach dem Mittelwertsatz finden wir ein $\eta \in [a,b]$ mit

$$\Phi(x) - x_* = \Phi(x) - \Phi(x_*) = (x - x_*)\Phi'(\eta).$$

Daraus folgt

$$|\Phi(x) - x_*| = |x - x_*|\,|\Phi'(\eta)|.$$

Also entscheidet $|\Phi'(\eta)|$ darüber, um welchen Faktor sich der Fehler unserer Näherung verbessert.

Als Beispiel können wir die Iterationsfunktionen (6.1a) und (6.1b) für die Approximation der Quadratwurzel untersuchen. Für (6.1a) erhalten wir

$$\Phi'(x) = 1 - 2x \qquad \text{für alle } x \in \mathbb{R}_{>0},$$

so dass wir bereits für $x \notin (0,1)$ zu $|\Phi'(x)| \geq 1$ gelangen. Falls die gesuchte Wurzel nicht zufällig in $(0,1)$ liegt, wird also diese Iteration nicht funktionieren.

Für (6.1b) ergibt sich wegen

$$\Phi(x) = x + \frac{\alpha - x^2}{2x} = x + \frac{\alpha}{2x} - \frac{x}{2} = \frac{1}{2}\left(x + \frac{\alpha}{x}\right) \qquad \text{für alle } x \in \mathbb{R}_{>0}$$

die Ableitung

$$\Phi'(x) = \frac{1}{2}\left(1 - \frac{\alpha}{x^2}\right) \qquad \text{für alle } x \in \mathbb{R}_{>0}.$$

Es gilt $|\Phi(x)| < 1$, falls $|1 - \alpha/x^2| < 2$ gilt, also $\alpha/x^2 < 3$ und damit $x > \sqrt{\alpha/3}$. Solange x groß genug ist, wird also dieses Verfahren konvergieren.

Es handelt sich dabei um das *Heron-Verfahren*, das tatsächlich sogar für *jeden* beliebigen Wert $x \in \mathbb{R}_{>0}$ den Fehler reduziert, allerdings ist der Nachweis dieser Eigenschaft etwas aufwendiger.

Im Mehrdimensionalen könnten wir im Prinzip genauso vorgehen, allerdings gestaltet sich das Rechnen mit mehrdimensionalen Ableitungen häufig etwas aufwendiger, so dass man es gerne vermeiden würde. Ein möglicher Ansatz besteht darin, statt der Differenzierbarkeit die *Lipschitz-Stetigkeit* zu verlangen, also die Existenz einer Konstanten $L \in \mathbb{R}_{\geq 0}$, die

$$\|\Phi(x) - \Phi(y)\| \leq L\|x - y\| \qquad \text{für alle } x, y \in U$$

erfüllt. Hierbei soll $U \subseteq V$ mit einem Banach-Raum V gelten, dessen Norm wir mit $\|\cdot\|$ bezeichnen. Ein Beispiel wäre $V = \mathbb{R}^n$ mit der euklidischen Norm oder der Maximum-Norm.

Für einen Fixpunkt $x_* \in U$ und eine beliebige Näherungslösung $x \in U$ erhalten wir dann

$$\|\Phi(x) - x_*\| = \|\Phi(x) - \Phi(x_*)\| \leq L\|x - x_*\|.$$

Falls wir also sicherstellen können, dass $L < 1$ gilt, haben wir wieder eine Verbesserung unserer Näherungslösung erreicht.

Es gilt sogar der folgende Satz:

Satz 6.2 (Banach) *Sei V ein Banach-Raum, sei $U \subseteq V$ eine abgeschlossene Teilmenge, und sei $\Phi : U \to U$ eine Abbildung, die*

$$\|\Phi(x) - \Phi(y)\| \leq L\|x - y\| \qquad \textit{für alle } x, y \in U$$

mit einer Konstanten $L \in [0, 1)$ erfüllt.

Dann besitzt Φ genau einen Fixpunkt $x_* \in U$, und für jeden Anfangsvektor $x_0 \in U$ konvergiert die durch

$$x_{m+1} := \Phi(x_m) \qquad \textit{für alle } m \in \mathbb{N}_0$$

gegebene Folge gegen x_*.

Unter den Voraussetzungen dieses Satzes können wir auch konkret abschätzen, wie genau eine Näherungslösung bereits ist. Beispielsweise haben wir

$$\begin{aligned}
\|x - x_*\| &= \|x - \Phi(x) + \Phi(x) - x_*\| \leq \|x - \Phi(x)\| + \|\Phi(x) - x_*\| \\
&= \|x - \Phi(x)\| + \|\Phi(x) - \Phi(x_*)\| \leq \|x - \Phi(x)\| + L\|x - x_*\|, \\
(1 - L)\|x - x_*\| &\leq \|x - \Phi(x)\|, \\
\|x - x_*\| &\leq \frac{\|x - \Phi(x)\|}{1 - L},
\end{aligned}$$

so dass wir nach der Berechnung der nächsten Näherungslösung $\Phi(x)$ Rückschlüsse auf die Genauigkeit der vorigen x ziehen können.

6.2 Eindimensionale Newton-Iteration

Viele in der Praxis wichtige Aufgabenstellungen lassen sich in die Form eines *Nullstellenproblems*

$$f(x_*) = 0$$

bringen, wobei

$$f : U \to V$$

eine Abbildung aus einer Menge $U \subseteq V$ in einen Vektorraum V bezeichnen soll.

Wir suchen nach einer allgemeinen Vorschrift, mit deren Hilfe wir aus f eine Iterationsfunktion Φ konstruieren können.

Erinnerung 6.3 (Taylor-Entwicklung) *Seien $k \in \mathbb{N}$ und eine $(k+1)$-mal stetig differenzierbare Funktion $g \in C^{k+1}[a,b]$ gegeben. Für alle $x, y \in [a,b]$ existiert ein $\eta \in [a,b]$ mit*

$$g(y) = g(x) + \sum_{\nu=1}^{k} \frac{(y-x)^\nu}{\nu!} g^{(\nu)}(x) + \frac{(y-x)^{k+1}}{(k+1)!} g^{(k+1)}(\eta).$$

Zuñachst untersuchen wir den Fall einer zweimal stetig differenzierbaren Funktion $f \in C^2(\mathbb{R})$ auf der reellen Achse. Sei $x_* \in \mathbb{R}$ eine Nullstelle der Funktion, und sei $x \in \mathbb{R}$ eine Näherung. Nach dem Satz von Taylor gilt

$$0 = f(x_*) = f(x) + (x_* - x) f'(x) + \frac{(x_* - x)^2}{2} f''(\eta)$$

für ein $\eta \in [x, x_*]$ (mit der Konvention $\eta \in [x_*, x]$, falls $x_* < x$ gelten sollte).

Wir erhalten

$$\begin{aligned}
0 &= f(x) + (x_* - x) f'(x) + \frac{(x_* - x)^2}{2} f''(\eta), \\
(x_* - x) f'(x) &= -f(x) - \frac{(x_* - x)^2}{2} f''(\eta), \\
x_* - x &= -\frac{f(x)}{f'(x)} - \frac{(x_* - x)^2}{2} \frac{f''(\eta)}{f'(x)}, \\
x_* &= x - \frac{f(x)}{f'(x)} - \frac{(x_* - x)^2}{2} \frac{f''(\eta)}{f'(x)}.
\end{aligned} \tag{6.2}$$

Falls die Näherung x nicht allzu weit von x_* entfernt ist, dürfen wir den letzten Term vernachlässigen und erhalten

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

Die rechte Seite definiert das gesuchte Iterationsverfahren.

Definition 6.4 (Eindimensionale Newton-Iteration) *Für $f \in C^2(\mathbb{R})$ mit $f'(x) \neq 0$ für alle $x \in \mathbb{R}$ ist die* Newton-Iteration *durch*

$$\Phi : \mathbb{R} \to \mathbb{R}, \qquad x \mapsto x - \frac{f(x)}{f'(x)},$$

gegeben.

Bemerkung 6.5 (Geometrische Interpretation) *Die Newton-Iteration lässt sich auch geometrisch herleiten: Die Funktion*

$$t : \mathbb{R} \to \mathbb{R}, \qquad y \mapsto f(x) + (y - x)f'(x),$$

definiert die Tangente *an f in x, also diejenige gerade, die f in x schneidet und dort dieselbe Steigung besitzt.*

Wenn wir nach einer Nullstelle y der Tangente suchen, erhalten wir

$$\begin{aligned} 0 &= t(x) = f(x) + (y - x)f'(x), \\ 0 &= \frac{f(x)}{f'(x)} + y - x, \\ y &= x - \frac{f(x)}{f'(x)} = \Phi(x), \end{aligned}$$

also gerade die Näherung der Newton-Iteration.

Aus der Gleichung (6.2) können wir eine Aussage über das Konvergenzverhalten der Newton-Iteration gewinnen: Es gilt

$$\Phi(x) - x_* = \frac{(x_* - x)^2}{2} \frac{f''(\eta)}{f'(x)}.$$

Falls wir also eine Konstante $C \in \mathbb{R}_{\geq 0}$ mit

$$\frac{|f''(y)|}{|f'(x)|} \leq C \qquad \text{für alle } x, y \in \mathbb{R}$$

finden können, folgt

$$|\Phi(x) - x_*| \leq \frac{C}{2}|x - x_*|^2 \qquad \text{für alle } x \in \mathbb{R}.$$

Bei derartigen Aussagen spricht man von *quadratischer Konvergenz*, da der alte Fehler quadriert auf der rechten Seite auftritt. Quadratisch konvergente Verfahren sind ausgesprochen erstrebenswert, weil sie sehr schnell hohe Genauigkeiten erreichen können.

Sei $x_0 \in \mathbb{R}$ ein Anfangswert, und sei die Folge der aus ihm berechneten Näherungen durch

$$x_{m+1} = \Phi(x_m) \qquad \text{für alle } m \in \mathbb{N}_0$$

gegeben. Nehmen wir an, dass $|x_0 - x_*| < 1/C$ gilt. Dann folgen

$$\begin{aligned}
|x_1 - x_*| &\leq \frac{C}{2}|x_0 - x_*|^2 < \frac{1}{2}|x_0 - x_*|, \\
|x_2 - x_*| &\leq \frac{C}{2}\frac{1}{4}|x_0 - x_*|^2 < \frac{1}{8}|x_0 - x_*|, \\
|x_3 - x_*| &\leq \frac{C}{2}\frac{1}{64}|x_0 - x_*|^2 < \frac{1}{128}|x_0 - x_*|,
\end{aligned}$$

und eine einfache Induktion führt zu

$$|x_m - x_*| \leq \frac{1}{2^{2^m-1}}|x_0 - x_*| \qquad \text{für alle } m \in \mathbb{N}_0.$$

Wir erkennen, dass jeder Schritt des Verfahrens die Anzahl der korrekt berechneten Stellen der Näherungslösung ungefähr verdoppelt. In der Praxis genügen deshalb oft sehr wenige Schritte, um die volle Maschinengenauigkeit zu erreichen.

Das uns bereits bekannte Heron-Verfahren lässt sich als Newton-Verfahren für die Funktion

$$f : \mathbb{R}_{>0} \to \mathbb{R}_{>0}, \qquad x \mapsto a - x^2,$$

interpretieren, so dass wir folgern dürfen, dass dieses Verfahren bei geeigneten Anfangswerten ebenfalls quadratisch konvergieren wird.

In der Praxis ist das Heron-Verfahren weniger attraktiv, weil die erforderliche Division durch x auf vielen Prozessoren erheblich mehr Zeit als Additionen, Subtraktionen oder Multiplikationen erfordert. Deshalb wird häufig eher die *reziproke Wurzel* gesucht, also $x_* = 1/\sqrt{\alpha}$, denn durch Multiplikation mit α lässt sich aus ihr leicht die Wurzel rekonstruieren.

Die reziproke Wurzel ist eine Nullstelle der Funktion

$$f : \mathbb{R}_{>0} \to \mathbb{R}_{>0}, \qquad x \mapsto a - \frac{1}{x^2},$$

und für die Newton-Iteration erhalten wir

$$\begin{aligned}
\Phi(x) = x - \frac{f(x)}{f'(x)} &= x - \frac{a - x^{-2}}{2x^{-3}} = x - \frac{1}{2}(ax^3 - x) \\
&= \frac{3}{2}x - \frac{1}{2}ax^3 = \frac{x}{2}(3 - ax^2) \qquad \text{für alle } x \in \mathbb{R},
\end{aligned}$$

und ein Schritt dieser Iteration erfordert lediglich Multiplikationen und Subtraktionen, die moderne Prozessoren sehr schnell ausführen können.

6.3 Mehrdimensionale Newton-Iteration

Wenden wir uns nun der Verallgemeinerung der Newton-Iteration für mehrdimensionale Probleme zu. Wir suchen eine Nullstelle $x_* \in \mathbb{R}^n$ einer Funktion $f \in C^1(\mathbb{R}^n, \mathbb{R}^n)$.

Wir wählen eine Näherungslösung $x \in \mathbb{R}^n$ und führen unsere Analyse auf den eindimensionalen Fall zurück, indem wir die Strecke von x zu x_* durch

$$\gamma : [0,1] \to \mathbb{R}^n, \qquad t \mapsto x + (x_* - x)t,$$

parametrisieren und die Hilfsfunktion

$$g : [0,1] \to \mathbb{R}^n, \qquad t \mapsto f(x + (x_* - x)t),$$

definieren. Sie erfüllt offenbar

$$g(0) = f(x), \qquad g(1) = f(x_*) = 0,$$

und mit der Kettenregel erhalten wir

$$g'(t) = (f \circ \gamma)'(t) = Df(\gamma(t))\gamma'(t) = Df(\gamma(t))(x_* - x) \qquad \text{für alle } t \in [0,1].$$

Hier bezeichnet

$$Df(y) = \begin{pmatrix} \partial_1 f_1(y) & \dots & \partial_n f_1(y) \\ \vdots & \ddots & \vdots \\ \partial_1 f_n(y) & \dots & \partial_n f_n(x) \end{pmatrix} \qquad \text{für alle } y \in \mathbb{R}^n$$

die *Jacobi-Matrix* der Funktion f, also die Matrix aller partiellen Ableitungen aller Komponenten.

Erinnerung 6.6 (Hauptsatz der Integral- und Differentialrechnung) *Es sei eine stetig differenzierbare Funktion $g \in C^1([a,b], \mathbb{R}^n)$ gegeben. Dann gilt*

$$g(b) - g(a) = \int_a^b g'(t)\,dt.$$

Mit Hilfe des Hauptsatzes der Integral- und Differentialrechnung erhalten wir

$$0 = f(x_*) = g(1) = g(0) + \int_0^1 g'(t)\,dt = f(x) + \int_0^1 Df(\gamma(t))(x_* - x)\,dt.$$

Da uns x_* unbekannt ist, können wir nur $\gamma(0) = x$ praktisch angeben, also schieben wir den Term $Df(x)(x_* - x) = Df(\gamma(0))(x_* - x)$ ein und finden

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

Wenn wir annehmen, dass $Df(x)$ invertierbar ist, ergibt sich

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

$$x_* = x - Df(x)^{-1}f(x) - Df(x)^{-1}\int_0^1 \big(Df(\gamma(t)) - Df(\gamma(0))\big)(x_* - x)\,dt. \tag{6.3}$$

Falls wir wieder davon ausgehen, dass x relativ nahe an x_* liegt, verläuft die gesamte Strecke von x nach x_* ebenfalls in der Nähe der gesuchten Nullstelle, so dass sowohl $x_* - x$ als auch $\gamma(t) - \gamma(0)$ und unter geeigneten Bedingungen auch $Df(\gamma(t)) - Df(\gamma(0))$ hinreichend klein sind und ignoriert werden können, um

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

zu erhalten.

Definition 6.7 (Mehrdimensionale Newton-Iteration) *Für $f \in C^2(\mathbb{R}^n, \mathbb{R}^n)$ mit für alle $x \in \mathbb{R}^n$ invertierbarer Jacobi-Matrix $Df(x)$ ist die* Newton-Iteration *durch*

$$\Phi : \mathbb{R}^n \to \mathbb{R}^n, \qquad\qquad x \mapsto x - Df(x)^{-1}f(x),$$

gegeben.

Auch in diesem Fall können wir eine Konvergenzaussage gewinnen: Nach (6.3) gilt

$$\Phi(x) - x_* = Df(x)^{-1}\int_0^1 \big(Df(\gamma(t)) - Df(\gamma(0))\big)(x_* - x)\,dt.$$

Wäre f zweimal stetig differenzierbar, so könnten wir den Quotienten

$$\frac{\|Df(\gamma(t)) - Df(\gamma(0))\|}{\|\gamma(t) - \gamma(0)\|} = \frac{\|Df(\gamma(t)) - Df(\gamma(0))\|}{t\|x_* - x\|}$$

mit Hilfe der zweiten Ableitung abschätzen und ein Ergebnis erhalten, das dem aus dem eindimensionalen Fall entspricht.

Eine erste Verallgemeinerung würde darin bestehen, lediglich die Lipschitz-Stetigkeit

$$\|Df(x) - Df(y)\| \le L\|x - y\| \qquad\qquad \text{für alle } x, y \in \mathbb{R}^n$$

zu fordern. Das hätte allerdings den Nachteil, dass eine bloße Skalierung der Funktion f schon die Konstante L verändern würde, obwohl die Nullstelle natürlich dieselbe bleibt.

Deshalb wählen wir eine *affin-invariante* Voraussetzung, nämlich

$$\|Df(x)^{-1}(Df(x) - Df(y))\| \le L\|x - y\| \qquad\qquad \text{für alle } x, y \in \mathbb{R}^n.$$

Mit dieser Voraussetzung erhalten wir

$$\begin{aligned}
\|\Phi(x) - x_*\| &= \left\| \int_0^1 Df(x)^{-1}\big(Df(\gamma(t)) - Df(\gamma(0))\big)(x_* - x)\,dt \right\| \\
&\le \int_0^1 \big\|Df(x)^{-1}\big(Df(\gamma(t)) - Df(x)\big)\big\|\,\|x_* - x\|\,dt \\
&\le L\int_0^1 \|\gamma(t) - x\|\,\|x_* - x\|\,dt = L\int_0^1 t\|x_* - x\|^2\,dt = \frac{L}{2}\|x_* - x\|^2,
\end{aligned}$$

dürfen also festhalten, dass auch das mehrdimensionale Newton-Verfahren quadratisch konvergiert.

7 Approximation von Funktionen

Heute übliche Prozessoren stellen uns Befehle für die Berechnung der Grundrechenarten zur Verfügung, allerdings nicht für darüber hinaus gehende Funktionen wie die Berechnung des Sinus, der Exponentialfunktion, oder des Logarithmus'. Diese Funktionen sind allerdings für viele natur- und wirtschaftswissenschaftliche Anwendungen von großer Bedeutung, so dass sich die Frage stellt, wie wir sie auf die Grundrechenarten zurückführen können.

7.1 Taylor-Entwicklung

Ein besonders einfacher Ansatz beruht auf der Taylor-Entwicklung, die in Erinnerung 6.3 bereits angedeutet wurde: Um eine $(k+1)$-mal stetig differenzierbare Funktion auf einem Intervall $[a,b]$ zu approximieren, können wir einen Entwicklungspunkt x_0 wählen und das *Taylor-Polynom*

$$\widetilde{g}_k(x) := \sum_{\nu=0}^{k} (x-x_0)^\nu \frac{g^{(\nu)}(x_0)}{\nu!} \qquad \text{für alle } x \in [a,b]$$

verwenden. Nach Erinnerung 6.3 finden wir für jedes $x \in [a,b]$ ein $\eta \in [a,b]$ mit

$$g(x) - \widetilde{g}_k(x) = (x-x_0)^{k+1} \frac{g^{(k+1)}(\eta)}{(k+1)!},$$

können also mit Hilfe der durch

$$\|f\|_{\infty,[a,b]} := \max\{|f(x)| \ : \ x \in [a,b]\} \qquad \text{für alle } f \in C[a,b] \tag{7.1}$$

definierten *Maximumnorm* die Fehlerabschätzung

$$\|g - \widetilde{g}_k\|_{\infty,[a,b]} \leq (b-a)^{k+1} \frac{\|g^{(k+1)}\|_{\infty,[a,b]}}{(k+1)!} \tag{7.2}$$

festhalten. Das Taylor-Polynom lässt sich besonders elegant auswerten: Wir definieren die Taylor-Basis $t_0, \ldots, t_k$ durch

$$t_\nu : [a,b] \to \mathbb{R}, \qquad x \mapsto (x-x_0)^\nu,$$

und stellen fest, dass

$$t_\nu(x) = (x-x_0) t_{\nu-1}(x) \qquad \text{für alle } x \in [a,b] \tag{7.3}$$

gilt. Die Koeffizienten

$$a_\nu := \frac{g^{(\nu)}(x_0)}{\nu!} \qquad \text{für alle } \nu \in [0:k]$$

hängen nicht von x ab und können von uns deshalb in einem Vorbereitungsschritt ermittelt und in Konstanten abgelegt werden, so dass sie keinen Rechenaufwand in der fertigen Implementierung verursachen.

Die Auswertung des Taylor-Polynoms können wir mit Hilfe des *Horner-Schemas* besonders effizient gestalten: Es gilt

$$\begin{aligned}\widetilde{g}_k(x) &= \sum_{\nu=0}^{k} a_\nu t_\nu(x) = a_0 + \sum_{\nu=1}^{k} a_\nu t_\nu(x) = a_0 + (x-x_0)\sum_{\nu=1}^{k} a_\nu t_{\nu-1}(x) \\ &= a_0 + (x-x_0)\sum_{\nu=0}^{k-1} a_{\nu+1} t_\nu(x) \qquad \text{für alle } x \in [a,b], \end{aligned} \tag{7.4}$$

so dass wir die Summe verkürzen können. Indem wir induktiv fortfahren, erhalten wir den folgenden Algorithmus:

```
const double a[] = { 1.0, 1.0/2.0, 1.0/3.0, 1.0/4.0, 1.0/5.0 };
const double x0 = 0.0;

double eval_taylor(double x)
{
  int k = sizeof(a) / sizeof(double) - 1;
  int i;
  double y, z;

  y = x - x0;
  z = a[k];
  for(i=k; i-->0; )
    z = a[i] + y * z;

  return z;
}
```

Das Array `a` nimmt die Koeffizienten auf, die Konstante `x0` den Entwicklungspunkt x_0, in `y` wird $x - x_0$ berechnet, während in `z` jeweils die Summen

$$s_i := \sum_{\nu=0}^{k-i} a_{\nu+i} t_\nu(x)$$

zu finden sind, die sich durch

$$s_{i-1} = \sum_{\nu=0}^{k-i+1} a_{\nu+i-1} t_\nu(x) = a_{i-1} + (x-x_0)\sum_{\nu=1}^{k-i+1} a_{\nu+i-1} t_{\nu-1}(x)$$

$$= a_{i-1} + (x - x_0) \sum_{\nu=0}^{k-i} a_{\nu+i} t_\nu(x) = a_{i-1} + (x - x_0) s_i \qquad \text{für alle } i \in [1 : k]$$

beginnend mit $s_k = a_k$ der Reihe nach berechnen lassen.

Wir dürfen festhalten, dass dieser Algorithmus lediglich $2k+1$ Gleitkommaoperationen für eine Auswertung des Taylor-Polynoms benötigt, und das darf als recht effizient gelten.

Ein großer Vorteil dieses Ansatzes besteht darin, dass sich die besonderen Eigenschaften der zu approximierenden Funktion gut ausnutzen lassen, um ihn effizienter zu gestalten. Als Beispiel untersuchen wir den Sinus

$$g(x) := \sin(x) \qquad \text{für alle } x \in [-\pi/2 : \pi/2].$$

Mit einer Induktion lässt sich einfach nachprüfen, dass

$$g^{(\nu)}(x) = \begin{cases} (-1)^{\lfloor \nu/2 \rfloor} \sin(x) & \text{falls } \nu \text{ gerade,} \\ (-1)^{\lfloor \nu/2 \rfloor} \cos(x) & \text{falls } \nu \text{ ungerade} \end{cases} \qquad \text{für alle } x \in \mathbb{R},\ \nu \in \mathbb{N}_0$$

gilt, so dass wir für den Entwicklungspunkt $x_0 = 0$ wegen $\sin(0) = 0$ und $\cos(0) = 1$ gerade

$$\widetilde{g}_k(x) = \begin{cases} x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{x^{k-1}}{(k-1)!} & \text{falls } k \text{ gerade,} \\ x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{x^k}{k!} & \text{falls } k \text{ ungerade} \end{cases} \qquad \text{für alle } x \in \mathbb{R}$$

erhalten. Alle geradzahligen Potenzen brauchen wir also nicht zu berücksichtigen, und indem wir x^2 nur einmal berechnen, ergibt sich der folgende Algorithmus:

```
const double a[] = { 1.0, 1.0/6.0, 1.0/120.0, 1.0/5040.0,
                     1.0/362880.0 };

double eval_sine(double x)
{
  int k2 = sizeof(a) / sizeof(double) - 1;
  int i;
  double x2, z;

  x2 = x * x;
  z = a[k2] * x;
  for(i=k2; i-->0; )
    z = a[i] + x2 * z;

  return z;
}
```

Hier werden nur die Koeffizienten a_{2i+1} zu ungeradzahligen Indizes in `a` gespeichert, während `x2` das Quadrat der Variablen x aufnimmt. Da bei dieser Variante nur für

ungeradzahlige Indizes gerechnet wird, ist sie ungefähr doppelt so effizient wie die allgemeine Implementierung.

Natürlich müssen wir uns die Frage stellen, wie genau unsere Approximation der Sinus-Funktion ist. Mit Hilfe der Abschätzung (7.2) ist sie leicht beantwortet: Die Maximumnorm der Ableitungen ist für den Sinus (und auch den Cosinus) immer gerade gleich eins, so dass wir

$$\|g - \widetilde{g}_k\|_{\infty,[-\pi/2,\pi/2]} \leq \frac{(\pi/2)^{k+1}}{(k+1)!}$$

erhalten. Für $k = 9$, also fünf Terme in unserer reduzierten Entwicklung, ergibt sich beispielsweise

$$\|g - \widetilde{g}_k\|_{\infty,[-\pi/2,\pi/2]} \leq 2.6 \times 10^{-5}.$$

7.2 Lagrange-Interpolation

Gelegentlich lassen sich die Ableitungen der zu approximierenden Funktion nicht so einfach berechnen, wie es im Fall des Sinus oder Cosinus der Fall ist. Dann kann die *Interpolation* zum Einsatz kommen, die eine Näherung einer Funktion nur aufgrund ihrer Werte in (hoffentlich) geschickt gewählten Punkten konstruiert.

Wir gehen davon aus, dass wir eine Funktion $g : [a, b] \to \mathbb{R}$ approximieren wollen. Wie schon im Fall der Taylor-Approximation suchen wir ein Polynom, m-ten Grades das die Funktion g möglichst gut wiedergibt.

Dabei bezeichnen wir mit

$$\Pi_m := \Big\{ \sum_{\nu=0}^{m} a_\nu x^\nu \ : \ a_0, \ldots, a_m \in \mathbb{R} \Big\}$$

den *Raum der Polynome höchstens m-ten Grades.*

Der Ansatz der Interpolation besteht darin, $m + 1$ *Stützstellen* (auch bekannt als *Interpolationspunkte*)

$$\xi_0, \ldots, \xi_m \in [a, b]$$

zu wählen und ein *Interpolationspolynom* $p \in \Pi_m$ zu suchen, das

$$p(\xi_\nu) = g(\xi_\nu) \qquad \text{für alle } \nu \in [0 : m] \tag{7.5}$$

erfüllt. Damit keine Bedingung doppelt auftaucht, fordern wir im Folgenden, dass die Stützstellen paarweise verschieden sind.

Es stellt sich zunächst die Frage, ob ein solches Polynom p überhaupt existiert. Diese Frage lässt sich elegant beantworten, indem wir den Vektorraum Π_m mit einer geeigneten Basis ausstatten.

Lemma 7.1 (Lagrange-Polynome) *Für alle $\mu \in [0 : m]$ definieren wir das* μ-te Lagrange-Polynom *zu den Stützstellen $\xi_0, \ldots, \xi_m$ durch*

$$\ell_\mu : \mathbb{R} \to \mathbb{R}, \qquad x \mapsto \prod_{\substack{\nu=0 \\ \nu \neq \mu}}^{m} \frac{x - \xi_\nu}{\xi_\mu - \xi_\nu}.$$

Dann ist $\ell_\mu \in \Pi_m$, und es gilt

$$\ell_\mu(\xi_\nu) = \begin{cases} 1 & \text{falls } \nu = \mu, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } \nu, \mu \in [0:m]. \tag{7.6}$$

Beweis. Sei $\mu \in [0:m]$. Jeder der Faktoren in der Definition des Lagrange-Polynoms ist linear, also ein Polynom ersten Grades. Das Produkt dieser m Faktoren muss dann ein Polynom höchstens m-ten Grades sein. Also gilt $\ell_\mu \in \Pi_m$.

Für $x = \xi_\mu$ ist jeder der Faktoren in der Definition des Lagrange-Polynoms gleich eins, also muss auch das Produkt gleich eins sein.

Für $x = \xi_\nu$ mit einem $\nu \in [0:m]$ und $\nu \neq \mu$ tritt in dem Produkt der Faktor $\frac{x-\xi_\nu}{\xi_\mu-\xi_\nu}$ auf, der für $x = \xi_\nu$ gleich null ist. Also ist in diesem Fall auch das gesamte Produkt gleich null.

Anderenfalls, also für $x = \xi_\mu$, ist jeder der Faktoren von der Form $\frac{\xi_\mu-\xi_\nu}{\xi_\mu-\xi} = 1$, so dass das gesamte Produkt gleich eins sein muss. ∎

Mit Hilfe der Lagrange-Polynome können wir das gesuchte Interpolationspolynom einfach angeben: Wenn wir

$$p := \sum_{\mu=0}^{m} g(\xi_\mu)\ell_\mu$$

setzen, erhalten wir $p \in \Pi_m$ und wegen (7.6) auch

$$p(\xi_\nu) = \sum_{\mu=0}^{m} g(\xi_\mu)\ell_\mu(\xi_\nu) = g(\xi_\nu)\ell_\nu(\xi_\nu) = g(\xi_\nu) \qquad \text{für alle } \nu \in [0:m].$$

Übrigens können wir analog sogar zeigen, dass $\{\ell_\mu : \mu \in [0:m]\}$ eine Basis des Polynomraums ist, denn wenn wir $c_0, \ldots, c_m \in \mathbb{R}$ mit

$$0 = \sum_{\mu=0}^{m} c_\mu \ell_\mu$$

gegeben haben, folgt durch Punktauswertung

$$0 = \sum_{\mu=0}^{m} c_\mu \ell_\mu(\xi_\nu) = c_\nu \qquad \text{für alle } \nu \in [0:m],$$

also sind die Lagrange-Polynome $m+1$ linear unabhängige Vektoren des nach Definition höchstens $(m+1)$-dimensionalen Polynomraums, somit müssen sie eine Basis sein.

Es stellt sich die Frage, wie gut das Interpolationspolynom die Funktion g approximiert.

Bemerkung 7.2 (Interpolationsfehler) *Die zu approximierende Funktion erfülle $g \in C^{m+1}[a,b]$. Für beliebige Stützstellen lässt sich beweisen, dass*

$$\|g - p\|_{\infty,[a,b]} \leq (b-a)^{m+1} \frac{\|g^{(m+1)}\|_{\infty,[a,b]}}{(m+1)!}$$

gilt, wir erhalten also eine ähnliche Abschätzung wie im Fall der Taylor-Approximation.

Für geschickt gewählte Stützstellen, nämlich für die Tschebyscheff-Punkte

$$\xi_\nu := \frac{b+a}{2} + \frac{b-a}{2} \cos\left(\frac{2\nu+1}{2m+2}\right) \qquad \text{für alle } \nu \in [0:m],$$

ergibt sich die wesentlich bessere Abschätzung

$$\|g-p\|_{\infty,[a,b]} \leq 2\left(\frac{b-a}{4}\right)^{m+1} \frac{\|g^{(m+1)}\|_{\infty,[a,b]}}{(m+1)!}.$$

Für diese Stützstellen lässt sich sogar beweisen, dass das Interpolationspolynom bis auf einen kleinen Faktor den Interpolationsfehler minimiert, also „fast das beste“ Polynom ist, mit dem sich g approximieren lässt.

Natürlich sind wir wieder daran interessiert, das Interpolationspolynom p möglichst effizient auszuwerten. Die Lagrange-Polynome sind dafür eher ungeeignet, denn schon die Auswertung eines einzelnen solchen Polynoms entsprechend seiner Definition erfordert mindestens m Subtraktionen und Multiplikationen: Selbst wenn wir die Vorfaktoren

$$\lambda_\mu := \prod_{\substack{\nu=0 \\ \nu\neq\mu}}^{m} \frac{1}{\xi_\mu - \xi_\nu} \qquad \text{für alle } \mu \in [0:m]$$

vorberechnen, müssen wir immer noch

$$\ell_\mu(x) = \lambda_\mu \prod_{\substack{\nu=0 \\ \nu\neq\mu}} x - \xi_\nu \qquad \text{für alle } \mu \in [0:m]$$

auswerten. Anschließend müssen wir die $m+1$ Lagrange-Polynome mit den passenden Koeffizienten multiplizieren und aufaddieren, so dass insgesamt ein Aufwand von mindestens

$$\sum_{\mu=0}^{m} 2m+2 = (m+1)(2m+2) = 2(m+1)^2$$

entsteht. Im Gegensatz zu der Taylor-Entwicklung würde uns die Auswertung des Polynoms p also einen mit m *quadratisch* wachsenden Aufwand kosten.

7.3 Newton-Interpolation

Der Aufwand für die Auswertung des Interpolationspolynoms lässt sich erheblich reduzieren, indem wir eine bessere Basis als die der Lagrange-Polynome für Π_m wählen.

Günstig wäre es, wenn diese Basis eine Eigenschaft ähnlich (7.3) aufweisen würde, denn dann dürften wir darauf hoffen, das effiziente Horner-Schema in modifizierter Form verwenden zu können.

Dabei müssen wir berücksichtigen, dass wir im Fall der Interpolation nicht nur einen Entwicklungspunkt, sondern mehrere Stützstellen haben.

Definition 7.3 (Newton-Polynome) *Wir definieren die* Newton-Polynome *zu den Stützstellen* $\xi_0, \ldots, \xi_m$ *induktiv durch*

$$n_{\nu,\mu}(x) := \begin{cases} 1 & \text{falls } \nu = \mu, \\ (x - \xi_\nu) n_{\nu+1,\mu}(x) & \text{ansonsten} \end{cases} \quad \text{für alle } \nu, \mu \in [0:m],\ \nu \leq \mu,\ x \in \mathbb{R}.$$

Angenommen, wir könnten das Interpolationspolynom in der Form

$$p = \sum_{\mu=0}^{m} d_\mu n_{0,\mu} \tag{7.7}$$

darstellen. Nach Definition hätten wir dann

$$\begin{aligned} p(x) &= \sum_{\mu=0}^{m} d_\mu n_{0,\mu}(x) = d_0 + \sum_{\mu=1}^{m} d_\mu n_{0,\mu}(x) = d_0 + (x - \xi_0) \sum_{\mu=1}^{m} d_\mu n_{1,\mu}(x) \\ &= d_0 + (x - \xi_0) \sum_{\mu=0}^{m-1} d_{\mu+1} n_{1,\mu}(x) \qquad \text{für alle } x \in [a,b]. \end{aligned}$$

Diese Gleichung entspricht dem Schritt (7.4) der Herleitung des Horner-Schemas.

Wir dort können wir für ein gegebenes $x \in \mathbb{R}$ Zwischensummen

$$s_i := \sum_{\mu=0}^{m-i} d_{\mu+i} n_{i,\mu+i}(x) \qquad \text{für alle } i \in [0:m]$$

definieren, die sich durch $s_m = d_m$ und

$$\begin{aligned} s_{i-1} &= \sum_{\mu=0}^{m-i+1} d_{\mu+i-1} n_{i-1,\mu+i-1}(x) = d_{i-1} + (x - \xi_i) \sum_{\mu=1}^{m-i+1} d_{\mu+i-1} n_{i,\mu+i-1}(x) \\ &= d_{i-1} + (x - \xi_i) \sum_{\mu=0}^{m-i} d_{\mu+i} n_{i,\mu+i}(x) = d_{i-1} + (x - \xi_i) s_i \qquad \text{für alle } i \in [1:m] \end{aligned}$$

der Reihe nach auswerten lassen, bis wir $p(x) = x_0$ erhalten. Dafür sind nur $3m$ Operationen erforderlich, der Rechenaufwand wächst wieder nur linear mit m.

Die entscheidende Frage ist natürlich, ob wir jedes Interpolationspolynom in der Form (7.7) darstellen können.

Dazu müssten wir Koeffizienten $d_0, \ldots, d_m \in \mathbb{R}$ finden, die

$$g(\xi_\nu) = \sum_{\mu=0}^{m} d_\mu n_{0,\mu}(\xi_\nu) \qquad \text{für alle } \nu \in [0:m]$$

erfüllen. Das ist ein lineares Gleichungssystem, dessen Lösbarkeit wir untersuchen sollten.

Lemma 7.4 (Matrixdarstellung) *Es gilt*

$$n_{\nu,\mu}(x) = \prod_{\kappa=\nu}^{\mu-1} x - \xi_\kappa \qquad \text{für alle } \nu, \mu \in [0:m],\ \nu \neq \mu,\ x \in \mathbb{R},$$

wobei wir die Konvention verwenden, dass das leere Produkt gleich eins ist.

Die durch

$$a_{\nu\mu} = n_{0,\mu}(\xi_\nu) \qquad \text{für alle } \nu, \mu \in [0:m]$$

gegebene Matrix $A \in \mathbb{R}^{[0:m]\times[0:m]}$ ist eine untere Dreiecksmatrix, deren Koeffizienten auf und unterhalb der Diagonalen alle ungleich null sind.

Beweis. Die erste Behauptung folgt mit einer einfachen Induktion über $\mu - \nu \in \mathbb{N}_0$.

Für den Nachweis des zweiten Behauptung wählen wir $\nu, \mu \in [0:m]$. Im echten oberen Dreiecksteil der Matrix, also für $\mu > \nu$, haben wir

$$a_{\nu\mu} = n_{0,\mu}(\xi_\nu) = \prod_{\kappa=0}^{\mu-1} \xi_\nu - \xi_\kappa = 0,$$

da in dem Produkt der Fall $\kappa = \nu$ auftreten kann, der zu einem Faktor von null führt.

Für $\mu \leq \nu$ haben wir dagegen

$$a_{\nu\mu} = n_{0,\mu}(\xi_\nu) = \prod_{\kappa=0}^{\mu-1} \xi_\nu - \xi_\kappa \neq 0,$$

da wegen $\kappa \leq \mu - 1 < \mu \leq \nu$ im Produkt niemals $\kappa = \nu$ auftreten kann. ∎

Da insbesondere auf der Diagonalen der Matrix A keine Null auftreten kann, ist A invertierbar. Also können wir die Lösung des Gleichungssystems

$$\begin{pmatrix} n_{0,0}(\xi_0) & 0 & \dots & 0 \\ n_{0,0}(\xi_1) & n_{0,1}(\xi_1) & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ n_{0,0}(\xi_m) & \dots & \dots & n_{0,m}(\xi_m) \end{pmatrix} \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_m \end{pmatrix} = \begin{pmatrix} g(\xi_0) \\ g(\xi_1) \\ \vdots \\ g(\xi_m) \end{pmatrix}$$

durch Vorwärtseinsetzen bestimmen und so die für unser verallgemeinertes Horner-Schema erforderlichen Koeffizienten gewinnen.

Folgerung 7.5 (Newton-Basis) $\{n_{0,\mu} : \mu \in [0:m]\}$ *ist eine Basis des Polynomraums* Π_m.

Beweis. Offenbar gilt $n_{0,\mu} \in \Pi_m$ für alle $\mu \in [0:m]$, also brauchen wir nur zu zeigen, dass die Newton-Polynome linear unabhängig sind.

Seien also $d_0, \ldots, d_m \in \mathbb{R}$ so gegeben, dass

$$0 = \sum_{\mu=0}^{m} d_\mu n_{0,\mu}$$

gilt. Durch Punktauswertung dieses Polynoms in $\xi_0, \ldots, \xi_m$ erhalten wir $0 = Ad$, und da A invertierbar ist, folgt $d = 0$. Also sind die Newton-Polynome linear unabhängig, und somit eine Basis. ∎

Anders als bei der Taylor-Approximation, bei der wir die Koeffizienten der Darstellung unmittelbar an den Ableitungen ablesen konnten, müssen wir bei der Interpolation in einem vorbereitenden Schritt die Koeffizienten als Lösung eines linearen Gleichungssystems bestimmen. Das Vorwärtseinsetzen erfordert dabei zwar wieder einen quadratischen Rechenaufwand, muss aber für jedes Interpolationspolynom nur einmal durchgeführt werden.

7.4 Anwendung in der Computergrafik

Polynome werden häufig in der Computergrafik eingesetzt, um Kurven und Oberflächen zu beschreiben. Statt eine skalarwertige Funktion zu interpolieren, wird mit einer vektorwertigen gearbeitet: Für gegebene Stützstellen $\xi_0, \ldots, \xi_m \in [a, b]$ sind Stützpunkte $y_0, \ldots, y_m \in \mathbb{R}^d$ gegeben, und gesucht sind Polynome $p_1, \ldots, p_d \in \Pi_m$ mit

$$p_i(\xi_\nu) = y_{\nu,i} \qquad \text{für alle } \nu \in [0:m],\ i \in [1:d].$$

Da sich alle Komponenten der Vektoren unabhängig voneinander bearbeiten lassen, können wir die im vorangehenden Abschnitt eingeführten Techniken unmittelbar einsetzen. Je nach Prozessorarchitektur kann es allerdings sinnvoll sein, unsere Algorithmen zu *vektorisieren*, also sie für alle Komponenten gleichzeitig auszuführen. Einerseits kann die Vektorisierung bei Prozessoren mit entsprechenden Befehlssätzen erheblich Zeit sparen, andererseits müssen auch beispielsweise die Koeffizienten der Matrix A dabei nur einmal berechnet werden, weil sie für alle Komponenten dieselben sind.

Bemerkung 7.6 (Rationale Approximation) *Für Anwendungen im* computer aided design *(CAD) hat die Verwendung von Polynomen allerdings den Nachteil, dass sich Kreise mit ihnen nur approximieren, aber nicht exakt darstellen lassen. Beispielsweise im Maschinenbau ist es aber sehr wünschenswert, auch kreisförmige Werkstücke am Computer darstellen zu können.*

Dieses Problem lässt sich lösen, indem wir zu einer rationalen Approximation *übergehen, also zu einer Darstellung durch einen Bruch*

$$r(x) := \frac{p(x)}{q(x)}$$

mit $p, q \in \Pi_m$. Falls wir dieselben Stützstellen für p und q verwenden, können wir die rationale Approximation umsetzen, indem wir für ein gegebenes $x \in \mathbb{R}$ die Werte $p(x)$

und $q(x)$ mit den bisher entwickelten Verfahren bestimmen und anschließend mit einer einzigen Division das gewünschte Endergebnis erhalten.

Für einen Halbkreis können wir beispielsweise

$$p_1(x) = 2x, \qquad p_2(x) = 1 - x^2, \qquad q(x) = 1 + x^2 \qquad \text{für alle } x \in [-1, 1]$$

setzen und erhalten mit

$$r(x) = \begin{pmatrix} p_1(x)/q(x) \\ p_2(x)/q(x) \end{pmatrix} \qquad \text{für alle } x \in [-1, 1]$$

das Ergebnis

$$\|r(x)\|_2^2 = \frac{p_1(x)^2 + p_2(x)^2}{q(x)^2} = \frac{4x^2 + 1 - 2x^2 + x^4}{1 + 2x^2 + x^4} = 1 \qquad \text{für alle } x \in [-1, 1],$$

und mit $r(-1) = (0, -1)$, $r(0) = (1, 0)$ und $r(1) = (0, 1)$ stellen wir fest, dass wir die obere Hälfte des Einheitskreises parametrisiert haben.

Die Polynominterpolation ist gerade so definiert, dass das Polynom alle gegebenen Punkte exakt treffen muss. Da die Lagrange-Polynome nur einfache Nullstellen haben, wechseln sie allerdings an jeder Stützstelle das Vorzeichen, so dass das Verschieben eines Stützpunkts in einer Richtung dazu führt, dass sich das Interpolationspolynom auf bestimmten Intervallen in die entgegengesetzte Richtung schiebt. Das ist ein Effekt, den man beispielsweise bei der interaktiven Konstruktion einer Kurve eher als lästig empfindet.

Bemerkung 7.7 (Bézier-Kurven) *Ein Lösungsansatz sind die sogenannten* Bézier-Kurven*, die durch eine Sammlung von* Kontrollpunkten *$z_0, \ldots, z_m \in \mathbb{R}^d$ beschrieben werden. Die zugehörige Kurve ist durch*

$$b(x) = \sum_{\mu=0}^{m} \binom{m}{\mu} z_\mu x^\mu (1 - x)^{m-\mu} \qquad \text{für alle } x \in [0, 1]$$

beschrieben. Da die Gewichtsfunktionen $x \mapsto \binom{m}{\mu} x^\mu (1 - x)^{m-\mu}$ auf $[0, 1]$ immer nichtnegativ sind, vermeiden sie den bei der Interpolation auftretenden Effekt: Wenn wir einen Kontrollpunkt in einer Richtung schieben, schiebt sich die gesamte Kurve in diese Richtung.

Da außerdem

$$\sum_{\mu=0}^{m} \binom{m}{\mu} x^\mu (1 - x)^{m-\mu} = (x + (1 - x))^m = 1 \qquad \text{für alle } x \in [0, 1]$$

gilt, bewegt sich die gesamte Kurve in der konvexen Hülle *der Kontrollpunkte.*

Bézier-Kurven sind aufgrund dieser Eigenschaften gut für die interaktive Konstruktion geometrischer Objekt geeignet.

Sie lassen sich auch elegant mit dem Algorithmus von de Casteljau *berechnen, indem sequentiell Konvexkombinationen zwischen Punkten konstruiert werden: Wenn wir mit*

$$b_\mu^{(0)}(x) = z_\mu \qquad \text{für alle } \mu \in [0:m],\ x \in [0,1]$$

anfangen und aufsteigend für $k \in [1:m]$ *die Hilfspunkte*

$$b_\mu^{(k)}(x) = x b_\mu^{(k-1)}(x) + (1-x) b_{\mu+1}^{(k-1)}(x) \qquad \text{für alle } \mu \in [0:m-k],\ x \in [0,1]$$

berechnen, erhalten wir $b^{(m)}(x) = b(x)$.

8 Schwachbesetzte Matrizen

Bisher haben wir Matrizen durch Arrays dargestellt: eine konventionelle Matrix durch ein zweidimensionales Array ihrer Koeffizienten, eine Tridiagonalmatrix durch drei eindimensionale Arrays für die Diagonale und die beiden Nebendiagonalen.

In der Praxis treten sehr häufig Matrizen auf, die *schwachbesetzt* sind, bei denen also fast alle Einträge gleich null sind, bei der sich die Verteilung übrigen Einträge allerdings nicht leicht voraussagen lässt, da sie beispielsweise von der Geometrie eines zu simulierenden Körpers abhängen kann.

Um derartige Matrizen effizient darstellen zu können, müssen spezialisierte Datenstrukturen eingesetzt werden.

8.1 Beispiel: Wellengleichung

Als motivierendes Beispiel untersuchen wir die Simulation der Ausbreitung von Wellen in einem Medium, zunächst nur in einer eindimensionalen Situation, beispielsweise einer gespannten Gitarrensaite.

Um ein im Computer lösbares Problem zu erhalten, approximieren wir die Saite durch eine endliche Anzahl von Punktmassen, die durch ideale Federn aneinander gekoppelt sind. Die Praxis zeigt, dass bei einer hinreichend großen Anzahl ein brauchbares Modell entsteht.

Sei also $n \in \mathbb{N}$. Wenn wir davon ausgehen, dass die Saite in vertikaler Richtung schwingt (*transversale* Wellen), können wir sie in horizontaler Richtung regelmäßig (*äquidistant*) unterteilen, also gemäß

$$h := \frac{1}{n+1}, \qquad x_i := hi \qquad \text{für alle } i \in [0 : n+1].$$

Die vertikalen Auslenkungen beschreiben wir durch zeitabhängige Funktionen

$$y_i : \mathbb{R} \to \mathbb{R} \qquad \text{für alle } i \in [0 : n+1],$$

so dass sich die i-te Punktmasse zum Zeitpunkt $t \in \mathbb{R}$ an der Position (x_i, y_i) befindet.

Die Saite soll „links“ und „rechts“ eingespannt sein, so dass wir

$$y_0(t) = 0, \qquad y_{n+1}(t) = 0 \qquad \text{für alle } t \in \mathbb{R}$$

erhalten. Um vorherzusagen, wie die Saite schwingen wird, müssen wir eine Beziehung zwischen den einzelnen Punktmassen herstellen. In unserem Fall gehen wir davon aus, dass benachbarte Massen durch ideale Federn aneinander gekoppelt sind.

Eine ideale Feder reagiert auf eine Auslenkung aus der Ruhelage, indem sie eine zu dieser Auslenkung proportionale Kraft in die Gegenrichtung ausübt. In unserem Fall gehen wir davon aus, dass die Feder zwischen dem $(i-1)$-ten und dem i-ten Punkt in Ruhelage eine Länge von $r < h$ besitzt, während sie ausgelenkt die Länge

$$\sqrt{(x_i - x_{i-1})^2 + (y_{i-1}(t) - y_i(t))^2} = \sqrt{h^2 + (y_{i-1}(t) - y_i(t))^2}$$

aufweist. Auf die i-te Masse wirkt dann eine Kraft, die sie in die Richtung

$$\begin{aligned}\frac{1}{\sqrt{(x_i - x_{i-1})^2 + (y_{i-1}(t) - y_i(t))^2}} \begin{pmatrix} x_{i-1} - x_i \\ y_{i-1}(t) - y_i(t) \end{pmatrix} \\ = \frac{1}{\sqrt{h^2 + (y_{i-1}(t) - y_i(t))^2}} \begin{pmatrix} -h \\ y_{i-1}(t) - y_i(t) \end{pmatrix}\end{aligned}$$

der $(i-1)$-ten Masse zieht. Insgesamt erhalten wir die Kraft

$$\begin{aligned} f_{i,i-1}(t) &= c_i \frac{\sqrt{h^2 + (y_{i-1}(t) - y_i(t))^2} - r}{\sqrt{h^2 + (y_{i-1}(t) - y_i(t))^2}} \begin{pmatrix} -h \\ y_{i-1}(t) - y_i(t) \end{pmatrix} \\ &= c_i \left(1 - \frac{r/h}{\sqrt{1 + (y_i(t) - y_{i-1}(t))^2/h^2}}\right) \begin{pmatrix} -h \\ y_{i-1}(t) - y_i(t) \end{pmatrix}, \end{aligned}$$

wobei $c_i \in \mathbb{R}_{>0}$ die *Federkonstante* ist, die beispielsweise die Elastizität der Saite beschreibt.

Falls die „Ruhelänge“ r deutlich geringer als h ist, können wir den Vorfaktor durch eins approximieren, so dass sich

$$f_{i,i-1}(t) = c_i \begin{pmatrix} -h \\ y_{i-1}(t) - y_i(t) \end{pmatrix}$$

ergibt. Die i-te Masse ist auch mit der $(i+1)$-ten Masse verbunden, für die wir analog eine Kraft von

$$f_{i,i+1}(t) = c_i \begin{pmatrix} h \\ y_{i+1}(t) - y_i(t) \end{pmatrix}$$

erhalten. Die beiden Kräfte summieren sich, so dass auf die i-te Masse zu einem Zeitpunkt $t \in \mathbb{R}$ insgesamt die Kraft

$$f_i(t) = c_i \begin{pmatrix} 0 \\ y_{i+1}(t) - 2y_i(t) + y_{i-1}(t) \end{pmatrix} \qquad \text{für alle } i \in [1:n],\ t \in \mathbb{R}$$

wirkt. Offenbar wirkt keine Kraft in horizontaler Richtung, so dass wir im Folgenden nur die vertikale Auslenkung zu betrachten brauchen.

Nach den Newton-Axiomen bewirkt eine Kraft eine Beschleunigung der Masse, also eine Veränderung ihrer Geschwindigkeit. Wenn wir die vertikalen Geschwindigkeiten der Massen durch Funktionen

$$v_i : \mathbb{R} \to \mathbb{R} \qquad \text{für alle } i \in [0:n+1],\ t \in \mathbb{R}$$

beschreiben, ergibt sich so

$$v_i'(t) = \frac{1}{m_i} f_i(t) \qquad \text{für alle } i \in [0 : n+1],\ t \in \mathbb{R},$$

wobei m_i die Masse der i-ten Punktmasse bezeichnet.

Ebenfalls nach den Newton-Axiomen bewirkt eine Geschwindigkeit eine Veränderung der Auslenkung, so dass wir

$$y_i'(t) = v_i(t) \qquad \text{für alle } i \in [0 : n+1],\ t \in \mathbb{R}$$

erhalten. Wir müssen also die Gleichungen

$$y_i'(t) = v_i(x), \quad v_i'(t) = \frac{c_i}{m_i}(y_{i+1}(t) - 2y_i(t) + y_{i-1}(t)) \qquad \text{für alle } i \in [1 : n],\ t \in \mathbb{R}$$

lösen, die ein System gewöhnlicher Differentialgleichungen bilden.

Zur Vereinfachung nehmen wir an, dass sich die Gesamtmasse $m \in \mathbb{R}_{>0}$ der Saite gleichmäßig auf alle Punktmassen verteilt, wobei der linke und rechte Endpunkt nur halb gewichtet werden. Es folgt $m_i = mh$.

Wir nehmen auch an, dass die Saite aus einem homogenen Material besteht. Dann berechnet sich die Federkonstante c_i aus einer Materialkonstante c gemäß $c_i = c/h$, denn eine kurze Feder leistet bei derselben Ausdehnung einen größeren Widerstand als eine lange.

Unser System vereinfacht sich zu

$$y_i'(t) = v_i(x), \quad v_i'(t) = \frac{c}{mh^2}(y_{i+1}(t) - 2y_i(t) + y_{i-1}(t)) \qquad \text{für alle } i \in [1 : n],\ t \in \mathbb{R}.$$

Indem wir $y_1(t), \ldots, y_n(t)$ und $v_1(t), \ldots, v_n(t)$ zu vektorwertigen Funktionen

$$y : \mathbb{R} \to \mathbb{R}^n, \qquad v : \mathbb{R} \to \mathbb{R}^n$$

zusammenfassen und die Matrix

$$A := \frac{c}{mh^2} \begin{pmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}$$

einführen, erhalten wir schließlich die handliche Form

$$y'(t) = v(t), \qquad v'(t) = -Ay(t) \qquad \text{für alle } t \in \mathbb{R}. \tag{8.1}$$

Ein einfacher Lösungsansatz besteht darin, die kontinuierliche Zeit in feste Zeitschritte zu zerlegen: Wir wählen eine Schrittweite $\delta \in \mathbb{R}_{>0}$ und setzen

$$t_i := i\delta \qquad \text{für alle } i \in \mathbb{Z}.$$

Die Ableitungen von y und v in t_i können wir durch den *zentralen Differenzenquotienten*

$$y'(t_i) \approx \frac{y(t_{i+1}) - y(t_{i-1})}{2\delta}, \qquad v'(t_i) \approx \frac{v(t_{i+1}) - v(t_{i-1})}{2\delta} \qquad \text{für alle } i \in \mathbb{Z}$$

approximieren, so dass wir

$$\begin{aligned} y(t_{i+1}) - y(t_{i-1}) &\approx 2\delta y'(t_i) = 2\delta v(t_i), \\ v(t_{i+1}) - v(t_{i-1}) &\approx 2\delta v'(t_i) = -2\delta A y(t_i) \qquad \text{für alle } i \in \mathbb{Z} \end{aligned}$$

erhalten. Wir stellen fest, dass wir für die Berechnung von $y(t_i)$ für geradzahliges i gerade $v(t_j)$ für ungeradzahlige j benötigen. Die Idee des *Leapfrog-Verfahrens* besteht darin, $y(t_i)$ nur für gerade i und $v(t_i)$ nur für ungerade i zu berechnen. Es nimmt die Form

$$\begin{aligned} \widetilde{y}(t_{i+2}) &= \widetilde{y}(t_i) + 2\delta \widetilde{v}(t_{i+1}), \\ \widetilde{v}(t_{i+3}) &= \widetilde{v}(t_{i+1}) - 2\delta A \widetilde{y}(t_{i+2}) \qquad \text{für alle } i \in 2\mathbb{Z} \end{aligned}$$

an, erreicht in der Praxis eine durchaus brauchbare Genauigkeit, und es lässt sich einfach implementieren.

In unserem Modellproblem lässt sich dieser Ansatz besonders effizient umsetzen, wenn wir ausnutzen, dass A eine Tridiagonalmatrix ist, so dass die für die Berechnung von $\widetilde{v}(t_{i+3})$ erforderliche Multiplikation mit der Matrix A in $\mathcal{O}(n)$ Operationen ausgeführt werden kann.

Wenden wir uns nun der zweidimensionalen Wellengleichung zu. Wir simulieren eine zweidimensionale Membran, beispielsweise das Fell einer Trommel. Wie zuvor approximieren wir die Membran durch Punktmassen, die sich lediglich in vertikaler Richtung verschieben dürfen. Die horizontale Position wird nun durch zwei Koordinaten festgelegt. Der Einfachheit halber gehen wir von einer quadratischen Membran aus, so dass wir die Punktmassen in einem regelmäßigen Rechtecksgitter anordnen können. Analog zu dem bereits betrachteten eindimensionalen Fall wählen wir $n \in \mathbb{N}$ und setzen

$$h := \frac{1}{n+1}, \qquad x_i := (i_1 h, i_2 h) \qquad \text{für alle } i = (i_1, i_2) \in \mathcal{I} := [1:n] \times [1:n].$$

Hier tritt die zweidimensionale Indexmenge $\mathcal{I}$ and die eindimensionalen Indizes $[1:n]$, die wir bisher verwendet haben.

Die Punktmassen werden wieder durch Federn miteinander verbunden, und zwar die i-te Masse mit den benachbarten Massen (i_1-1, i_2), (i_1+1, i_2), (i_1, i_2-1) und (i_1, i_2+1). Indem wir wie zuvor vorgehen, können wir zeigen, dass auf die i-te Masse gerade die Kraft

$$f_i(t) = \frac{c}{m} \frac{y_{(i_1-1,i_2)}(t) + y_{(i_1+1,i_2)}(t) + y_{(i_1,i_2-1)}(t) + y_{(i_1,i_2+1)}(t) - 4y_i(t)}{h^2}$$

wirkt. Wir gehen wieder davon aus, dass die Auslenkung der Randpunkte auf den Wert null fixiert ist, fassen die Auslenkungen und Geschwindigkeiten zu vektorwertigen Funktionen

$$y : \mathbb{R} \to \mathbb{R}^{\mathcal{I}}, \qquad v : \mathbb{R} \to \mathbb{R}^{\mathcal{I}},$$

zusammen, und erhalten

$$y'(t) = v(t), \qquad v'(t) = -Ay(t) \qquad \text{für alle } t \in \mathbb{R},$$

wobei die Matrix $A \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ nun durch

$$a_{ij} = \begin{cases} \frac{4c}{mh^2} & \text{falls } j = i, \\ -\frac{c}{mh^2} & \text{falls } |j_1 - i_1| = 1,\ j_2 = i_2, \\ -\frac{c}{mh^2} & \text{falls } |j_2 - i_2| = 1,\ j_1 = i_1, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i, j \in \mathcal{I}$$

gegeben ist. Hier stoßen wir auf zwei Schwierigkeiten: Erstens ist A nun eine Matrix, deren Zeilen und Spalten durch jeweils *zwei* Indizes charakterisiert werden, kann also insgesamt als ein *vierdimensionales* Array interpretiert werden.

Zweitens sind fast alle Einträge dieser Matrix gleich null, allerdings handelt es sich nicht um eine Tridiagonalmatrix oder eine ähnlich einfache Struktur.

Die erste Schwierigkeit lässt sich einfach umgehen, indem wir die Indexmenge $\mathcal{I}$ bijektiv auf eine Indexmenge $[1 : N]$ mit $N = n^2$ abbilden und die Matrix A dann als Matrix in $\mathbb{R}^{N\times N}$ interpretieren.

Da wir eine Implementierung in der Programmiersprache C anstreben, werden wir allerdings die Indexmenge $[0 : N - 1]$ verwenden, die besser zu den C-Konventionen passt. Eine übliche Wahl ist die Bijektion

$$\iota : \mathcal{I} \to [0 : N - 1], \qquad i \mapsto (i_1 - 1) + (i_2 - 1)n,$$

die einer zeilenweisen Numerierung der Indizes entspricht. Für jedes $k \in [0 : N - 1]$ können wir mit ganzzahliger Division durch n mit Rest Indizes $k_1, k_2 \in [0 : n - 1]$ mit $k = k_1 + k_2 n$ erhalten, und mit $i_1 = k_1 + 1$ und $i_2 = k_2 + 1$ folgt $\iota(i) = k$, also ist ι surjektiv. Da $\#\mathcal{I} = n^2 = N = \#[0 : N - 1]$ gilt, also Definitions- und Bildbereich gleich mächtig sind, folgt daraus bereits, dass ι injektiv sein muss, also auch bijektiv.

Die Matrix A könnte nun in C beispielsweise durch das folgende Programmfragment aufgestellt werden.

```
A = malloc(sizeof(double) * N * N);
ldA = N;

diag = 4.0 * c / (m * h * h);
offdiag = -c / (m * h * h);

for(jx=1; jx<=n; jx++)
  for(jy=1; jy<=n; jy++) {
    j = (jx-1) + (jy-1)*n;

    for(ix=1; ix<=n; ix++)
      for(iy=1; iy<=n; iy++) {
```

```
        i = (ix-1) + (iy-1)*n;

        if(ix == jx) {
          if(iy == jy)
            A[i+j*ldA] = diag;
          else if(iy == jy+1 || iy == jy-1)
            A[i+j*ldA] = offdiag;
        }
        else if(ix == jx+1 || ix == jx-1) {
          if(iy == jy)
            A[i+j*ldA] = offdiag;
          else
            A[i+j*ldA] = 0.0;
        }
        else
          A[i+j*ldA] = 0.0;
      }
    }
```

Diese Implementierung weist allerdings erhebliche Nachteile auf: Sie enthält beispielsweise Fallunterscheidungen in der innersten Schleife, die dazu führen können, dass Prozessoren mit Pipelining nur schlecht ausgelastet sind.

Viel kritischer ist allerdings, dass jede Zeile der Matrix höchstens fünf von null verschiedene Einträge enthält. Bei $n = 1000$ würden also mindestens $N - 5 = 999\,995$ Nullen in jeder Zeile gespeichert werden, die auf das Ergebnis der Matrix-Vektor-Multiplikation keinerlei Einfluss haben. Wir verschwenden Speicher und Rechenzeit.

8.2 Schwachbesetzte Matrizen in Listendarstellung

Es gibt eine ganze Reihe von alternativen Darstellungen für schwachbesetzte Matrizen. Eine besonders einfache ist die Listendarstellung: Zu jeder Zeile (oder analog zu jeder Spalte) der Matrix gehört eine Liste der in dieser Zeile von null verschiedenen Koeffizienten.

Da wir ja gerade „Lücken“ in der Zeile lassen möchten, muss dabei zu jedem Koeffizienten auch die Spalte gespeichert werden, zu der er gehört. Die Zeile kennen wir ja bereits.

In der Sprache C könnte ein Element dieser Liste wie folgt realisiert werden:

```
typedef struct {
  int j;                    /* Column index */
  double aij;               /* Matrix coefficient */
  matrixcoeff *next;        /* Next in row */
} matrixcoeff;
```

Die Matrix selbst sollte neben der Anzahl der Zeilen und Spalten auch ein Array aufweisen, das jedem Zeilenindex den Zeiger auf den Kopf der entsprechenden Liste zuordnet, beispielsweise in Form des folgenden Datentyps:

```
typedef struct {
  int rows;                    /* Number of rows */
  int cols;                    /* Number of columns */

  matrixcoeff **nz;            /* Matrix rows */
} listmatrix;
```

Für jeden Zeilenindex i zwischen `0` und `rows-1` können wir den Kopf der zugehörigen Zeile unter `nz[i]` finden. Beispielsweise gibt das folgende Programmfragment alle von null verschiedenen Einträge aus:

```
for(i=0; i<rows; i++)
  for(c=nz[i]; c; c=c->next)
    printf("(%d %d: %f)\n", i, c->j, c->aij);
```

Eine Multiplikation einer Matrix mit einem Vektor lässt sich umsetzen, indem wir für jede Zeile i die Liste durchlaufen, die j-ten Einträge aus dem Eingabevektor lesen, mit a_{ij} multiplizieren, und die Ergebnisse aufaddieren. Der Rechenaufwand ist dann im Wesentlichen proportional zu der Anzahl der Listenelemente.

Gelegentlich benötigen Algorithmen auch die Möglichkeit, mit der Transponierten einer Matrix zu multiplizieren. Das ist in diesem Fall etwas schwieriger, weil wir keinen direkten Zugriff auf die Spalten der Matrix haben. Wir können allerdings für den i-ten Eintrag x_i des Eingabevektors die Liste der i-ten Zeile durchlaufen und das Produkt aus a_{ij} und x_i zu dem j-ten Eintrag des Ergebnisvektor y_j addieren. Falls der Ergebnisvektor vorher auf null gesetzt wurde, erhalten wir das gewünschte Ergebnis.

Etwas aufwendig ist das Aufstellen einer solchen Matrix: Wir erzeugen zunächst ein Objekt `A` des Typs `listmatrix` und setzen alle Einträge des Arrays `A->nz` auf den Nullzeiger, der in diesem Fall leere Listen darstellt. In diesem Zustand treten keine von null verschiedenen Einträge auf, also haben wir die Nullmatrix dargestellt.

Einen Eintrag $a_{ij} \neq 0$ können wir nun einfügen, indem wir die Liste `A->nz[i]` um einen passenden Eintrag erweitern. Da es sich in unserem Beispiel um eine einfach verkettete Liste handelt, empfiehlt es sich, den neuen Eintrag am Kopf einzufügen.

In vielen praktischen Anwendungen, insbesondere bei der in den Natur- und Ingenieurwissenschaften sehr erfolgreichen Finite-Elemente-Methode, ist es hilfreich, einen Wert v zu einem bereits vorhandenen Eintrag a_{ij} hinzu addieren zu können. Das lässt sich umsetzen, indem wir die zu der i-ten Zeile gehörende Liste `A->nz[i]` durchsuchen. Falls in dieser Liste ein Element `mc` mit `mc->j == j` existiert, addieren wir v zu `mc->aij` und sind fertig. Anderenfalls fügen wir ein neues Listenelement hinzu.

Auf einem 64-Bit-System benötigt eine Variable des Typs `matrixcoeff` jeweils 8 Bytes für `j`, `aij` und `next`. Eine `listmatrix` mit n Zeilen und k von null verschiedenen Elementen benötigt mindestens $24k + 16 + 8n$ Bytes. In der Praxis kann der Speicherbedarf

erheblich höher ausfallen, weil jedes Listenelement für die dynamische Speicherverwaltung zusätzliche Daten aufbewahren muss. In unserem Modellproblem haben wir $k \approx 5n$, so dass wir einen Speicherbedarf von etwas weniger als $128n + 16$ Bytes erhalten. Eine Darstellung durch eine vollbesetzte Matrix würde $8n^2$ erfordern, ist als schon für $n > 17$ aufwendiger als die Listendarstellung. Für die nicht ungewöhnliche Problemdimension $n = 1\,000\,000$ benötigt die Listendarstellung gut 122 Megabytes, während die Arraydarstellung mehr als 7 629 394 Megabytes erfordert.

8.3 Schwachbesetzte Matrizen in Arraydarstellung

Listen sind auf modernen Computern häufig wesentlich weniger effizient als Arrays, da die Listenelemente praktisch beliebig im Hauptspeicher verteilt liegen können, so dass häufig neue Adressen an die Speichercontroller übertragen werden müssen und sich die Wartezeiten bei Speicherzugriffen verlängern.

Deshalb sind wir daran interessiert, schwachbesetzte Matrizen durch Arrays darzustellen. Sparsam und relativ elegant ist dabei die Darstellung im CRS-Format (engl. *compressed row storage*), das drei Arrays verwendet:

```
typedef struct {
  int rows;                  /* Number of rows */
  int cols;                  /* Number of columns */

  int *row;                  /* Start indices of row data */
  int *col;                  /* Column indices for each row */
  double *coeff;             /* Coefficients for each row */
} crsmatrix;
```

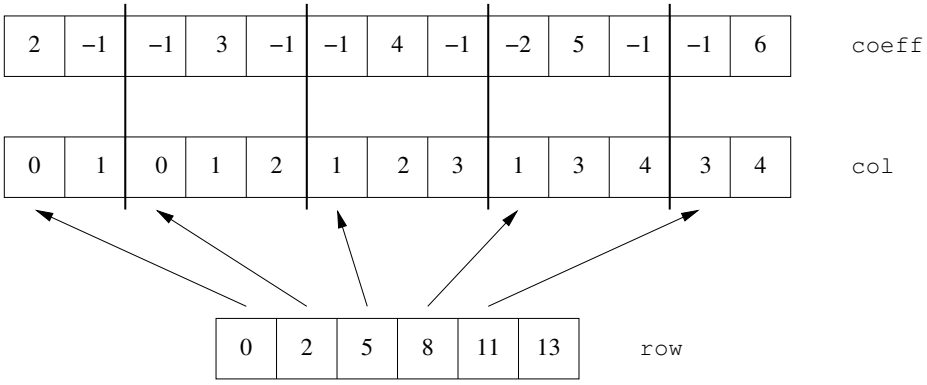
Das Array `coeff` speichert die Koeffizienten der von null verschiedenen Matrixeinträge. Dabei werden zuerst die Einträge der ersten Zeile angegeben, dann die der zweiten, und so weiter. Das Array `col` speichert die zugehörigen Spaltenindizes in entsprechender Anordnung. Das Array `row` speichert zu jeder Zeile den ersten Eintrag in den Arrays `coeff` und `col`, der zu dieser Zeile gehört.

Um lästige Fallunterscheidungen zu vermeiden, empfiehlt es sich, zusätzlich in `row[rows]` die Gesamtzahl der von null verschiedenen Einträge zu speichern, denn dann finden sich die Einträge der i-ten Zeile immer in den Indizes `row[i]` bis `row[i+1]-1` der Arrays `coeff` und `col`.

Beispielsweise würde die Matrix

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 3 & -1 & & \\ & -1 & 4 & -1 & \\ & -2 & & 5 & -1 \\ & & & -1 & 6 \end{pmatrix}$$

in der Form



dargestellt werden. Das Auflisten der von null verschiedenen Einträge ließe sich mit dem folgenden Programmfragment bewerkstelligen:

```
for(i=0; i<rows; i++)
  for(k=row[i]; k<row[i+1]; k++)
    printf("(%d %d: %f)\n", i, col[k], coeff[k]);
```

Man erkennt, dass alle auftretenden Arrays in diesem Fall der Reihe nach durchlaufen werden, so dass die Hoffnung besteht, dass ein moderner Prozessor Operationen mit Matrizen in CRS-Darstellung effizient durchführen kann.

Darüber hinaus spart die Darstellung auch etwas Speicher: Für n Zeilen und für k von null verschiedene Einträge benötigen wir $8(n+1)$ Bytes für das Array `row` sowie jeweils $8k$ Bytes für die Arrays `col` und `coeff`. Insgesamt kommen wir auf $16+8(n+1)+16k$ Bytes. Im Vergleich zu der Listendarstellung haben wir also ungefähr 30% des Speichers gespart.

Ein großer Nachteil der CRS-Darstellung soll hier nicht verschwiegen werden: Sie ist vollständig statisch, so dass das Einfügen eines neuen Eintrags es beispielsweise typischerweise erforderlich macht, die Arrays `coeff` und `col` zu kopieren. Überraschenderweise kann das unter gewissen Umständen immer noch effizienter als der Einsatz einer Liste sein, elegant ist es jedenfalls nicht.

Häufig wird deshalb das Aufstellen einer CRS-Matrix in zwei Phasen erledigt: In einer ersten wird lediglich gezählt, wie viele Nicht-Null-Elemente in jeder Zeile auftreten werden. Danach ist bekannt, wie groß die Arrays `coeff` und `col` sein müssen, so dass sie angelegt werden können. In der zweiten Phase werden sie dann mit ihren endgültigen Werten gefüllt.

9 Schnelle Lösungsverfahren

Die Darstellung einer schwachbesetzten Matrix durch eine Liste oder CRS-Arrays hat den Vorteil, dass sich der Speicherbedarf gegenüber der konventionellen Darstellung erheblich reduziert und sich Matrix-Vektor-Multiplikationen effizient ausführen lassen.

Allerdings stellt man bei vielen praktischen Anwendungen fest, dass die bisher diskutierten Lösungsverfahren wie die LR- oder QR-Zerlegung zu einem Verlust der schwachbesetzten Struktur führen, so dass auch Rechenaufwand und Speicherbedarf erheblich wachsen. Deshalb lohnt es sich, über Lösungsverfahren nachzudenken, die auch für schwachbesetzte Matrizen geeignet sind.

9.1 Klassische Iterationsverfahren

Wir untersuchen ein lineares Gleichungssystem der Form

$$Ax = b, \qquad A \in \mathbb{R}^{n\times n},\ x, b \in \mathbb{R}^n, \tag{9.1}$$

bei dem A und b gegeben sind und wir x berechnen wollen. Unsere Lösungsalgorithmen sollen so beschaffen sein, dass sie es nicht erforderlich machen, die Matrix A „umzubauen“, denn das könnte bedeuten, dass sich die schwachbesetzte Struktur verändert.

Ein erster Ansatz ist eng mit dem Vorwärts- und Rückwärtseinsetzen verwandt: Wir wählen einen Index $i \in [1 : n]$ und versuchen, eine Variable x_i so zu bestimmen, dass zumindest die i-te Zeile des linearen Gleichungssystems erfüllt ist. Wir erhalten

$$\begin{aligned}
\sum_{j=1}^{n} a_{ij}x_j &= b_i,\\
a_{ii}x_i &= b_i - \sum_{\substack{j=1\\ j\neq i}} a_{ij}x_j\\
x_i &= \frac{1}{a_{ii}}\Big(b_i - \sum_{\substack{j=1\\ j\neq i}} a_{ij}x_j\Big),
\end{aligned}$$

und diese Berechnung lässt sich ausführen, ohne die Matrixstruktur zu verändern.

Die Lösung erhalten wir allerdings nur, falls alle x_j für $j \neq i$ bereits den Komponenten der Lösung entsprachen, und das ist hinreichend unwahrscheinlich.

Deshalb verwenden wir einen *iterativen* Zugang: Wir gehen von einer Näherungslösung $x^{(m)} \in \mathbb{R}^n$ aus und berechnen eine, hoffentlich bessere, Näherungslösung $x^{(m+1)} \in \mathbb{R}^n$, indem wir der Reihe nach alle Komponenten aktualisieren:

Definition 9.1 (Gauß-Seidel-Iteration) *Seien $A \in \mathbb{R}^{n\times n}$ und $b, x^{(0)} \in \mathbb{R}^n$ gegeben. Es gelte $a_{ii} \neq 0$ für alle $i \in [1:n]$. Durch*

$$x_i^{(m+1)} = \frac{1}{a_{ii}}\Big(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(m)}\Big) \quad \text{für alle } i \in [1:n],\ m \in \mathbb{N}_0 \tag{9.2}$$

ist die Folge der Näherungslösungen der Gauß-Seidel-Iteration *definiert.*

Es ist zu beachten, dass bei der Berechnung von $x_1^{(m+1)}$ nur Komponenten der alten Lösung $x^{(m)}$ verwendet werden, bei der Berechnung von $x_2^{(m+1)}$ dann bereits $x_1^{(m+1)}$, bei der von $x_3^{(m+1)}$ bereits $x_1^{(m+1)}$ und $x_2^{(m+1)}$, und so weiter. Offenbar ist es zweckmäßig, diese Berechnung so zu implementieren, dass die Komponenten von $i = 1$ bis $i = n$ nacheinander bestimmt werden.

Wir stellen fest, dass wir bei der Berechnung der beiden Summen jeweils nur die von null verschiedenen Einträge der Matrix zu berücksichtigen brauchen, so dass wir bei einer Listen- oder CRS-Darstellung der Matrix einen Rechenaufwand erwarten dürfen, der lediglich proportional zu der Anzahl dieser Einträge ist. Für schwachbesetzte Matrizen lässt sich deshalb ein Schritt der Gauß-Seidel-Iteration sehr effizient durchführen.

Um analysieren zu können, ob das Iterationsverfahren eine gegen die exakte Lösung konvergente Folge berechnet, müssen wir es etwas anders formulieren.

Wir zerlegen $A = D - E - F$ mit einer Diagonalmatrix $D \in \mathbb{R}^{n\times n}$, einer echten unteren Dreiecksmatrix $E \in \mathbb{R}^{n\times n}$ und einer echten oberen Dreiecksmatrix $F \in \mathbb{R}^{n\times n}$ gemäß

$$d_{ij} = \begin{cases} a_{ii} & \text{falls } i = j, \\ 0 & \text{ansonsten,} \end{cases}$$

$$e_{ij} = \begin{cases} -a_{ij} & \text{falls } i > j, \\ 0 & \text{ansonsten,} \end{cases} \qquad f_{ij} = \begin{cases} -a_{ij} & \text{falls } i < j, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } i, j \in [1:n].$$

Die Gauß-Seidel-Iteration nimmt dann die Form

$$\begin{aligned} x_i^{(m+1)} &= \frac{1}{a_{ii}}\Big(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(m)}\Big) \\ &= \frac{1}{d_{ii}}\big(b_i + (Ex^{(m+1)})_i + (Fx^{(m)})_i\big) \\ &= \big(D^{-1}(b + Ex^{(m+1)} + Fx^{(m)})\big)_i \qquad \text{für alle } i \in [1:n],\ m \in \mathbb{N}_0, \end{aligned}$$

so dass wir sie kompakt als

$$\begin{aligned} x^{(m+1)} &= D^{-1}(b + Ex^{(m+1)} + Fx^{(m)}), \\ x^{(m+1)} - D^{-1}Ex^{(m+1)} &= D^{-1}(b + Fx^{(m)}), \\ D^{-1}(D - E)x^{(m+1)} &= D^{-1}(b + Fx^{(m)}), \\ (D - E)x^{(m+1)} &= b + Fx^{(m)}, \end{aligned}$$

$$x^{(m+1)} = (D-E)^{-1}(b + Fx^{(m)}) \qquad \text{für alle } m \in \mathbb{N}_0$$

schreiben können. Die Gauß-Seidel-Iteration löst also in jedem Schritt ein Gleichungssystem mit einer unteren Dreiecksmatrix durch Vorwärtseinsetzen. Die Iterationsvorschrift ist durch

$$\Phi : \mathbb{R}^n \to \mathbb{R}^n, \qquad x \mapsto (D-E)^{-1}(b + Fx),$$

gegeben, und unsere Aufgabe besteht darin, Bedingungen dafür zu identifizieren, dass es sich dabei um eine Kontraktion im Sinne des Banachschen Fixpunktsatzes 6.2 handelt.

Wir haben bereits gesehen, dass

$$x^{(m+1)} = D^{-1}(b + Ex^{(m+1)} + Fx^{(m)})$$

gilt, also liegt die Vermutung nahe, dass D^{-1} im Verhältnis zu E und F in einem geeigneten Sinn „hinreichend klein“ sein müsste, also die Diagonalmatrix D „hinreichend groß“.

Definition 9.2 (Diagonaldominanz) *Eine Matrix $A \in \mathbb{R}^{n\times n}$ nennen wir* diagonaldominant *oder* echt diagonaldominant*, falls*

$$\sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}| < |a_{ii}| \qquad \textit{für alle } i \in [1:n] \textit{ gilt.}$$

Satz 9.3 (Konvergenz) *Sei $A \in \mathbb{R}^{n\times n}$ diagonaldominant. Dann gilt*

$$\varrho := \max\left\{ \frac{1}{|a_{ii}|} \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}| \;:\; i \in [1:n] \right\} < 1$$

und wir erhalten

$$\|\Phi(x) - \Phi(y)\|_\infty \leq \varrho \|x - y\|_\infty \qquad \textit{für alle } x, y \in \mathbb{R}^n.$$

Beweis. $\varrho < 1$ ist äquivalent zu der vorausgesetzten Diagonaldominanz der Matrix.

Seien $x, y \in \mathbb{R}^n$ gegeben. Nach Definition gilt

$$\Phi(x) - \Phi(y) = (D-E)^{-1}F(x-y).$$

Zur Abkürzung definieren wir $z := x - y$ und $\widetilde{z} := \Phi(x) - \Phi(y)$. Wir haben

$$\widetilde{z} = (D-E)^{-1}Fz, \qquad (D-E)\widetilde{z} = Fz,$$
$$D\widetilde{z} = E\widetilde{z} + Fz, \qquad \widetilde{z} = D^{-1}(E\widetilde{z} + Fz).$$

Wir zeigen nun

$$|\widetilde{z}_i| \leq \varrho \|z_i\|_\infty \qquad \text{für alle } i \in [1:n]$$

per Induktion über i.

Induktionsanfang: Für $i = 1$ gilt $(E\widetilde{z})_i = 0$, wir erhalten

$$|\widetilde{z}_1| = \frac{1}{|a_{11}|}\Big|\sum_{j=2}^{n} a_{1j} z_j\Big| \leq \frac{1}{|a_{11}|}\sum_{j=2}^{n} |a_{1j}|\,|z_j| \leq \frac{1}{|a_{11}|}\sum_{j=2}^{n} |a_{1j}|\,\|z\|_\infty \leq \varrho\|z\|_\infty.$$

Induktionsvoraussetzung: Sei $i \in [1:n]$ so gegeben, dass

$$|\widetilde{z}_j| \leq \varrho\|z\|_\infty \qquad \text{für alle } j \in [1:i-1] \text{ gilt.}$$

Induktionsschritt: Wir haben

$$\begin{aligned}|\widetilde{z}_i| &= \frac{1}{|a_{ii}|}\Big|\sum_{j=1}^{i-1} a_{ij}\widetilde{z}_j + \sum_{j=i+1}^{n} a_{ij} z_j\Big| \leq \frac{1}{|a_{ii}|}\Big(\sum_{j=1}^{i-1} |a_{ij}|\,|\widetilde{z}_j| + \sum_{j=i+1}^{n} |a_{ij}|\,|z_j|\Big)\\ &\leq \frac{1}{|a_{ii}|}\Big(\varrho\sum_{j=1}^{i-1} |a_{ij}|\,\|z\|_\infty + \sum_{j=i+1}^{n} |a_{ij}|\,\|z\|_\infty\Big) \leq \frac{1}{|a_{ii}|}\sum_{\substack{j=1\\ j\neq i}} |a_{ij}|\,\|z\|_\infty \leq \varrho\|z\|_\infty.\end{aligned}$$

Nach Definition von z und $\widetilde{z}$ folgt daraus unmittelbar $\|\Phi(x) - \Phi(y)\|_\infty \leq \varrho\|x - y\|_\infty$. ∎

Eine Schwierigkeit des Gauß-Seidel-Verfahrens besteht darin, dass es sich häufig nicht gut parallelisieren lässt, weil die Berechnung von $x_i^{(m+1)}$ von den Werten der $x_j^{(m+1)}$ mit $j < i$ abhängt. Zwar zeigt eine genauere Betrachtung, dass nur diejenigen Indizes benötigt werden, für die auch $a_{ij} \neq 0$ gilt, so dass sich unter gewissen Umständen durch eine geschickte Permutation der Matrixzeilen und -spalten eine Verbesserung erzielen lässt, aber das ist nicht immer möglich.

Wir können das Verfahren so verändern, dass wir für die Berechnung des neuen Werts grundsätzlich die *alten* Werte verwenden. Dann können wir alle Komponenten parallel berechnen lassen.

Definition 9.4 (Jacobi-Iteration) *Seien $A \in \mathbb{R}^{n\times n}$ und $b, x^{(0)} \in \mathbb{R}^n$ gegeben. Es gelte $a_{ii} \neq 0$ für alle $i \in [1:n]$. Durch*

$$x_i^{(m+1)} = \frac{1}{a_{ii}}\Big(b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij} x_j^{(m)}\Big) \qquad \textit{für alle } i \in [1:n],\ m \in \mathbb{N}_0$$

ist die Folge der Näherungslösungen der Jacobi-Iteration *definiert.*

Die Iterationsvorschrift des Jacobi-Verfahrens ist durch

$$\Phi : \mathbb{R}^n \to \mathbb{R}^n, \qquad x \mapsto D^{-1}(b + Ex^{(m)} + Fx^{(m)}),$$

gegeben.

Lemma 9.5 (Konvergenz) *Sei $A \in \mathbb{R}^{n\times n}$ diagonaldominant, sei ϱ wie in Satz 9.3 definiert. Dann gilt*

$$\|\Phi(x) - \Phi(y)\|_\infty \leq \varrho\|x - y\|_\infty \qquad \text{für alle } x, y \in \mathbb{R}^n.$$

Beweis. Seien $x, y \in \mathbb{R}^n$, sei $z := x - y$, und sei $\widetilde{z} := \Phi(x) - \Phi(y)$. Dann gilt

$$\begin{aligned} |\widetilde{z}_i| &= \frac{1}{|a_{ii}|}\Big|\sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij} z_j\Big| \leq \frac{1}{|a_{ii}|}\sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|\,|z_j| \\ &\leq \frac{1}{|a_{ii}|}\sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|\,\|z\|_\infty \leq \varrho\|z\|_\infty \qquad \text{für alle } i \in [1:n]. \end{aligned}$$

Das ist äquivalent zu $\|\Phi(x) - \Phi(y)\|_\infty \leq \varrho\|x - y\|_\infty$. ∎

9.2 Verfahren der konjugierten Gradienten

In der Praxis zeigt sich, dass sowohl das Gauß-Seidel- als auch das Jacobi-Verfahren bei wichtigen Problemklassen nur sehr langsam konvergieren, so dass sie nur sehr eingeschränkt brauchbar sind.

Bessere Eigenschaften findet man bei der Familie der *Krylow-Raum-Verfahren*, unter denen das Verfahren der konjugierten Gradienten eine prominente Stellung einnimmt.

Definition 9.6 (Positiv definit) *Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt* positiv definit, *falls*

$$x^* A x > 0 \qquad \text{für alle } x \in \mathbb{R}^n \setminus \{0\} \text{ gilt.}$$

Die Idee des Verfahrens der konjugierten Gradienten besteht darin, das Gleichungssystem (9.1) durch eine äquivalente Minimierungsaufgabe zu ersetzen. Wir definieren

$$f : \mathbb{R}^n \to \mathbb{R}, \qquad x \mapsto \frac{1}{2} x^* A x - b^* x.$$

Lemma 9.7 (Minimierung) *Sei $A \in \mathbb{R}^{n\times n}$ symmetrisch und positiv definit. Seien $x, p \in \mathbb{R}^n$ gegeben. Es gilt*

$$f(x) \leq f(x + \lambda p) \qquad \text{für alle } \lambda \in \mathbb{R} \tag{9.3a}$$

genau dann, wenn

$$p^*(Ax - b) = 0 \tag{9.3b}$$

gilt. Insbesondere ist die Lösung des linearen Gleichungssystems (9.1) das einzige Minimum der Funktion f.

Beweis. Zunächst stellen wir fest, dass

$$\begin{aligned} f(x+\lambda p) &= \frac{1}{2}(x+\lambda p)^* A(x+\lambda p) - b^*(x+\lambda p) \\ &= \frac{1}{2}x^*Ax + \lambda p^*Ax + \frac{\lambda^2}{2}p^*Ap - b^*x - \lambda b^*p \\ &= f(x) + \lambda p^*(Ax-b) + \frac{\lambda^2}{2}p^*Ap \qquad \text{für alle } \lambda \in \mathbb{R} \end{aligned}$$

gilt. Sei nun (9.3b) vorausgesetzt. Dann folgt unmittelbar

$$f(x+\lambda p) = f(x) + \frac{\lambda^2}{2}p^*Ap \geq f(x) \qquad \text{für alle } \lambda \in \mathbb{R}.$$

Sei nun umgekehrt (9.3a) vorausgesetzt. Falls $p = 0$ gilt, ist (9.3b) trivial erfüllt. Ansonsten gilt $p^*Ap > 0$, da A positiv definit ist, so dass

$$\lambda := -\frac{p^*(Ax-b)}{p^*Ap}$$

wohldefiniert ist und wir

$$\begin{aligned} f(x) \leq f(x+\lambda p) &= f(x) - \frac{(p^*(Ax-b))^2}{p^*Ap} + \frac{(p^*(Ax-b))^2}{(p^*Ap)^2}p^*Ap \\ &= f(x) - \frac{(p^*(Ax-b))^2}{p^*Ap} \leq f(x) \end{aligned}$$

erhalten. Mit $p^*Ap > 0$ folgt $p^*(Ax-b) = 0$, also (9.3b). ∎

Wir können das Minimum der Funktion f iterativ approximieren, indem wir eine Suchrichtung $p \in \mathbb{R}^n \setminus \{0\}$ wählen und nach demjenigen $\lambda \in \mathbb{R}$ suchen, für das $f(x+\lambda p)$ minimal wird. Nach (9.3b) ist das äquivalent zu

$$p^*(A(x+\lambda p)-b) = 0 \iff p^*(Ax-b) = -\lambda p^*Ap \iff \lambda = \frac{p^*(b-Ax)}{p^*Ap}.$$

Der Zähler (keineswegs der gesamte Ausdruck!) wird gemäß der Cauchy-Schwarz-Ungleichung maximal, falls wir $p = b - Ax$ setzen. Damit erhalten wir mit

$$\begin{aligned} r^{(m)} &:= b - Ax^{(m)}, \\ p^{(m)} &:= r^{(m)}, \\ \lambda^{(m)} &:= \frac{(p^{(m)})^* r^{(m)}}{(p^{(m)})^* Ap^{(m)}}, \\ x^{(m+1)} &:= x^{(m)} + \lambda^{(m)} p^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0 \end{aligned}$$

ein erstes Minimierungsverfahren. Es erfordert in jedem Schritt zwei Multiplikationen mit der Matrix A, und diese Multiplikationen sind häufig sehr aufwendig. Deshalb lohnt

es sich, eine der beiden zu vermeiden, indem wir den Hilfsvektor $a^{(m)} := Ap^{(m)}$ einführen und feststellen, dass

$$\begin{aligned} r^{(m+1)} &= b - Ax^{(m+1)} = b - A(x^{(m)} + \lambda^{(m)} p^{(m)}) = b - Ax^{(m)} - \lambda^{(m)} Ap^{(m)} \\ &= r^{(m)} - \lambda^{(m)} a^{(m)} \qquad \text{für alle } m \in \mathbb{N}_0 \text{ gilt.} \end{aligned}$$

Definition 9.8 (Gradientenverfahren) *Seien $A \in \mathbb{R}^{n\times n}$ und $b, x^{(0)} \in \mathbb{R}^n$ gegeben. Die Matrix A sei symmetrisch und positiv definit. Durch*

$$\begin{aligned} r^{(0)} &:= b - Ax^{(0)}, \\ p^{(m)} &:= r^{(m)}, \\ a^{(m)} &:= Ap^{(m)}, \\ \lambda^{(m)} &:= \frac{(p^{(m)})^* r^{(m)}}{(p^{(m)})^* a^{(m)}}, \\ x^{(m+1)} &:= x^{(m)} + \lambda^{(m)} p^{(m)}, \\ r^{(m+1)} &:= r^{(m)} - \lambda^{(m)} a^{(m)} \qquad \textit{für alle } m \in \mathbb{N}_0 \end{aligned}$$

ist die Folge $(x^{(m)})_{m=0}^{\infty}$ der Näherungslösungen des Gradientenverfahrens *definiert.*

Wenn man das Gradientenverfahren praktisch anwendet, stellt man allerdings fest, dass es in typischen Anwendungsfällen nur unwesentlich schneller als beispielsweise die Gauß-Seidel-Iteration arbeitet. Eine Ursache besteht darin, dass die Optimalität bezüglich der Richtung $p^{(0)}$, die wir im ersten Schritt des Verfahrens herstellen, bereits im zweiten Schritt wieder verloren gehen kann und das häufig auch tut.

Dieses Problem lässt sich glücklicherweise lösen, indem wir die Richtungen $p^{(m)}$ etwas geschickter wählen. Nehmen wir an, dass $x^{(m)}$ optimal bezüglich der Richtungen $p^{(0)}, \ldots, p^{(m-1)}$ ist, dass also

$$(p^{(\ell)})^*(b - Ax^{(m)}) = 0 \qquad \text{für alle } \ell \in [0 : m-1]$$

gilt. Wir möchten die Richtung $p^{(m)}$ so wählen, dass diese Optimalität auch für $x^{(m+1)}$ erhalten bleibt, dass also

$$\begin{aligned} 0 = (p^{(\ell)})^*(b - Ax^{(m+1)}) &= (p^{(\ell)})^*(b - Ax^{(m)} - \lambda^{(m)} Ap^{(m)}) \\ = (p^{(\ell)})^*(b - Ax^{(m)}) - \lambda^{(m)} (p^{(\ell)})^* Ap^{(m)} &\qquad \text{für alle } \ell \in [0 : m-1] \end{aligned}$$

gilt. Da $x^{(m)}$ optimal bezüglich $p^{(0)}, \ldots, p^{(m-1)}$ ist und wir $\lambda^{(m)} \neq 0$ wählen wollen, muss also

$$(p^{(\ell)})^* Ap^{(m)} = 0 \qquad \text{für alle } \ell \in [0 : m-1] \tag{9.4}$$

gelten. Vektoren $p^{(\ell)}$ und $p^{(m)}$, die diese Eigenschaft besitzen, nennt man *konjugiert* zueinander. Unser Ziel ist es, $p^{(m)}$ aus $r^{(m)}$ so zu konstruieren, dass (9.4) gilt.

Aufgrund der Ähnlichkeit zu orthogonalen Vektoren können wir das Gram-Schmidt-Orthogonalisierungsverfahren anpassen, um $p^{(m)}$ zu konstruieren, indem wir

$$p^{(m)} := r^{(m)} - \sum_{\ell=0}^{m-1} \frac{(p^{(\ell)})^* A r^{(m)}}{(p^{(\ell)})^* A p^{(\ell)}} p^{(\ell)}$$

setzen. Diese Gleichung für die Berechnung zu verwenden, wäre allerdings relativ aufwendig, insbesondere müssten wir alle bisher verwendeten Richtungen aufbewahren. Glücklicherweise können wir diesen Aufwand vermeiden.

Lemma 9.9 (Krylow-Raum) *Wir definieren die* Krylow-Räume *zu A und $r^{(0)}$ durch*

$$\mathcal{K}_m := \operatorname{span}\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, \ldots, A^m r^{(0)}\} \qquad \text{für alle } m \in \mathbb{N}_0.$$

Sei $m_0 \in \mathbb{N}$ die kleinste Zahl, für die

$$\mathcal{K}_{m_0} = \mathcal{K}_{m_0+1}$$

gilt. Dann haben wir

$$\operatorname{span}\{p^{(0)}, \ldots, p^{(m)}\} = \operatorname{span}\{r^{(0)}, \ldots, r^{(m)}\} = \mathcal{K}_m \qquad \text{für alle } m \in [0 : m_0].$$

Sei $m \in [0 : m_0]$, und sei $\ell \in [0 : m-2]$. Dann gilt nach Lemma 9.9

$$Ap^{(\ell)} \in \mathcal{K}_{m-1} = \operatorname{span}\{p^{(0)}, \ldots, p^{(m-1)}\},$$

und da $x^{(m)}$ optimal bezüglich der Richtungen $p^{(0)}, \ldots, p^{(m-1)}$ ist, gilt auch

$$(p^{(\ell)})^* r^{(m)} = 0 \qquad \text{für alle } \ell \in [0 : m-1],$$

so dass wir schließlich

$$(p^{(\ell)})^* A r^{(m)} = (Ap^{(\ell)})^* r^{(m)} = 0$$

erhalten. Bei der Gram-Schmidt-Orthogonalisierung sind also alle Terme bis auf den für $\ell = m-1$ gleich null, so dass wir einfach

$$p^{(m)} := r^{(m)} - \frac{(p^{(m-1)})^* A r^{(m)}}{(p^{(m-1)})^* A p^{(m-1)}} = r^{(m)} - \frac{(a^{(m-1)})^* r^{(m)}}{(a^{(m-1)})^* p^{(m-1)}}$$

verwenden können.

Definition 9.10 (Verfahren der konjugierten Gradienten) *Seien $A \in \mathbb{R}^{n \times n}$ und $b, x^{(0)} \in \mathbb{R}^n$ gegeben. Die Matrix A sei symmetrisch und positiv definit. Durch*

$$\begin{aligned} r^{(0)} &:= b - Ax^{(0)}, \\ p^{(0)} &:= r^{(0)}, \\ a^{(m)} &:= Ap^{(m)}, \end{aligned}$$

$$
\begin{aligned}
\lambda^{(m)} &:= \frac{(p^{(m)})^* r^{(m)}}{(p^{(m)})^* a^{(m)}}, \\
x^{(m+1)} &:= x^{(m)} + \lambda^{(m)} p^{(m)}, \\
r^{(m+1)} &:= r^{(m)} - \lambda^{(m)} a^{(m)}, \\
\mu^{(m)} &:= \frac{(a^{(m)})^* r^{(m+1)}}{(p^{(m)})^* a^{(m)}}, \\
p^{(m+1)} &:= r^{(m+1)} - \mu^{(m)} p^{(m)} \qquad \textit{für alle } m \in [0 : m_0]
\end{aligned}
$$

ist die Folge $(x^{(m)})_{m=0}^{m_0}$ *der Näherungslösungen des* Verfahrens der konjugierten Gradienten *(auch als* cg-Verfahren *bekannt) definiert.*

10 Paralleles Rechnen

Da aufgrund physikalischer Schranken in absehbarer Zeit nicht mit einer Zunahme der Taktfrequenz moderner Prozessoren zu rechnen ist, ist eine Steigerung der Rechenleistung nur möglich, indem die zu leistende Arbeit auf mehrere Prozessoren verteilt wird.

Dieser Ansatz ist im Bereich des Hochleistungsrechnens schon seit Jahrzehnten sehr erfolgreich, und die Einführung von Mehrkernprozessoren erlaubt es, ihn auch auf bei Workstations, Desktop-Rechnern und sogar Smartphones gewinnbringend einzusetzen.

10.1 Programmiermodelle für das parallele Rechnen

Für unsere Zwecke bietet es sich an, paralleles Rechnen danach zu klassifizieren, welche Operationen parallel ausgeführt werden und wo die benötigten Daten vorliegen.

In der ersten Kategorie ist die einfachste Form des parallelen Rechnens die *Vektorisierung*, bei der mehrere Rechenwerke jeweils identische Operationen ausführen. Dieser Ansatz ist beispielsweise im Bereich der Signalverarbeitung und Computergrafik sehr erfolgreich, denn in beiden Gebieten dominieren Algorithmen, die mehr oder weniger gleiche Operationen für sehr viele Datenpakete ausführen müssen, etwa Tausende von Punkten auf den Bildschirm zu bringen, die gemeinsam ein darzustellendes Objekt beschreiben.

Beispielsweise bietet der C-Compiler der *GNU Compiler Collection* die Möglichkeit, Vektortypen zu definieren und mit ihnen Operationen auszuführen. Eine Linearkombination $y \leftarrow y + \alpha x$ zweier Vektoren der Länge n lässt sich mit dem folgenden Programmfragment umsetzen:

```
typedef float float4 __attribute__((vector_size (16)));

void axpy(int n, float alpha, const float *x, float *y)
{
  const float4 *xv = (const float4 *) x;
  float4 *yv = (float4 *) y;
  int i;

  for(i=0; i<n/4; i++)
    yv[i] += alpha * xv[i];

  for(i*=4; i<n; i++)
    y[i] += alpha * x[i];
}
```

Der Typ `float4` beschreibt hier einen Vektortyp, in dem vier Variablen des Typs `float` zusammengefasst sind. Die erste Schleife führt jeweils vier Multiplikationen und vier Additionen simultan aus. Die zweite Schleife kümmert sich um die wenigen Einträge, die übrig bleiben, falls n nicht durch vier teilbar ist.

Dieser Ansatz bietet den Vorteil, dass sich mit geringem Schaltungsaufwand eine erhebliche Steigerung der Rechenleistung erreichen lässt. Außerdem ist die Programmierung relativ einfach, solange man es mit Algorithmen zu tun hat, die sich gut für die Vektorisierung eignen.

Allerdings eignet sich die Mehrheit der Algorithmen nicht oder nicht gut für eine Vektorisierung. In diesem Fall kann es sich lohnen, *nebenläufige Programme* zu schreiben, bei denen unterschiedliche Folgen von Operationen gleichzeitig (oder zumindest scheinbar gleichzeitig) ausgeführt werden. Gegenüber der Vektorisierung ist dieser Ansatz erheblich allgemeiner einsetzbar, stellt aber auch erheblich höhere Anforderungen an die Hardware und die Fähigkeiten des Programmierenden. Auf Seiten der Hardware muss die Möglichkeit geschaffen werden, die Zustände mehrerer Befehlsketten zu verwalten, Wartezeiten durch geschickte Anordnung der Befehle zu vermeiden, und den Zugriff auf gemeinsam genutzte Ressourcen zu koordinieren. Auf Seiten des Menschen muss dafür gesorgt werden, dass die Beiträge der einzelnen Befehlsketten so zusammengesetzt werden, dass unabhängig von ihrer Verteilung auf die verfügbaren Rechenwerke immer das korrekte Ergebnis ermittelt wird und keine gegenseitige Blockade (*deadlock*) auftritt.

Die Daten für eine Berechnung können entweder in einem gemeinsamen Speicher aufbewahrt werden, auf den alle Rechenwerke und Prozessoren beliebig zugreifen dürfen, oder in einem verteilten Speicher, bei dem beispielsweise jeder Prozessor exklusiv auf einen Bereich zugreifen kann und Daten aus den anderen Bereichen explizit von den zuständigen anderen Prozessoren anfordern muss.

Ein gemeinsamer Speicher ist für die Programmierung oft sehr viel komfortabler, stellt allerdings erhebliche Anforderungen an die Hardware: Um Speicherzugriffe effizient ausführen zu können, kommen zwischen Hauptspeicher und Prozessor in der Regel Cache-Speicher zum Einsatz, die Kopien häufig genutzter Speicherbereiche aufnehmen. Falls nun ein Prozessor Daten in den Hauptspeicher schreibt, müssen die Caches aller anderen Prozessoren darauf aufmerksam gemacht werden, dass eventuell vorhandene Kopien des betreffenden Teils des Hauptspeichers nicht mehr länger gültig sind und vor dem nächsten Zugriff ersetzt werden müssen.

In der Praxis kommen deshalb gemeinsame Speicher nur bei relativ kleinen Systemen zum Einsatz, bei denen der Aufwand für die Synchronisation beherrschbar ist. Große Rechnersysteme verwenden in der Regel verteilten Speicher.

10.2 Multithreading

Ein erfolgreiches Modell für die nebenläufige Programmierung ist das *Multithreading*: Ein Programm besteht aus mehreren „Strängen“ von Befehlen, den *Threads*, die nebenläufig ausgeführt werden. In der Regel obliegt es dabei dem Betriebs- oder Laufzeitsystem, die verfügbaren Ressourcen den Threads zuzuordnen.

Eine beliebte Umsetzung dieses Ansatzes ist der Standard *OpenMP* (für *Open Multi-Processing*), der als Erweiterung der Programmiersprachen C, C++ und FORTRAN spezifiziert ist. Threads werden in diesem Standard durch OpenMP-Direktiven erzeugt und gesteuert, die in C und C++ als `#pragma`-Anweisungen und in FORTRAN als aktive Kommentare umgesetzt werden, so dass ein Compiler, der OpenMP nicht unterstützt, sie ignorieren kann, ohne die Funktionsfähigkeit eines Programms zu beeinträchtigen.

Während der Ausführung eines Programms können wir in einen *parallelen Abschnitt* eintreten. Dabei wird ein *Team* von Threads erzeugt, die den gemeinsamen Abschnitt ausführen. Beispielsweise wird das folgende Programmfragment

```
#pragma omp parallel
{
  printf("Thread %d of %d\n",
         omp_get_thread_num(), omp_get_num_threads());
}
printf("Done\n");
```

dazu führen, dass so viele Threads angelegt werden, wie die OpenMP-Implementierung für sinnvoll hält, und dass jeder dieser Threads seine Nummer und die Anzahl der Threads in seinem Team ausgibt. Der Abschnitt endet, wenn alle Threads ihre Arbeit geleistet haben. Sie werden gelöscht und der letzte Aufruf der Funktion `printf` erfolgt nur noch einmal.

Da beispielsweise aus einem parallelen Abschnitt heraus beliebige Funktionen aufgerufen werden können, ist er als *zeitlicher* Abschnitt zu verstehen, nicht als Abschnitt des Programmcodes. In unserem Beispiel wird das erste `printf` etwa in einem parallelen Abschnitt ausgeführt, das zweite jedoch nicht.

Innerhalb eines parallelen Abschnitts lässt sich mit weiteren Direktiven steuern, wie die zu leistende Arbeit auf die vorhandenen Threads verteilt werden soll. Beispielsweise wird im folgenden Programmfragment

```
#pragma omp parallel
{
  #pragma omp single
  printf("Started parallel section, %d threads\n",
         omp_get_num_threads());

  /* ... */
}
```

nur einer der erzeugten Threads eine Statusmeldung ausgeben, während die anderen warten. Anschließend bearbeiten alle nebenläufig den Rest des parallelen Abschnitts. Diese Direktive ist offenbar nicht besonders hilfreich, wenn es darum geht, Arbeit zu verteilen, aber sie ist häufig nützlich, um beispielsweise Statusmeldungen auszugeben.

Gerade bei der Verarbeitung von Daten, die in Arrays organisiert sind, ist eine Direktive hilfreich, die eine `for`-Schleife automatisch auf alle Threads verteilt.

```
void axpy(int n, float alpha, const float *x, float *y)
{
  #pragma omp parallel
  {
    int i;

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

In diesem Beispiel wird für jeden Index i zwischen 0 und $n-1$ genau einmal die Operation $y_i \leftarrow y_i + \alpha x_i$ ausgeführt. Das Laufzeitsystem verteilt die Aufrufe auf die vorhandenen Threads und versucht dabei, jeden Thread möglichst gleichmäßig zu belasten. Dabei sind nur bestimmte Schleifen erlaubt, denn das OpenMP-System muss die Möglichkeit haben, bereits zu Beginn der Schleife einfach zu ermitteln, wie viele Iterationen stattfinden werden. Diese Iterationen werden dann nach einer von mehreren wählbaren Strategien auf die Threads im aktuellen Team verteilt.

An dieser Stelle müssen wir kurz den Umgang mit Variablen diskutieren. In C werden lokale Variablen in einem Kellerspeicher (engl. *stack*) abgelegt. Damit die Threads eines Teams weitere Funktionen aufrufen können, verfügt jeder von ihnen über einen eigenen Keller. In dem obigen Beispiel besitzt deshalb jeder Thread eine eigene Kopie der Variablen `i`, es handelt sich um eine *private* Variable. Falls eine von einem Thread aufgerufene Funktion weitere lokale Variablen definiert, werden diese in dem Keller des aufrufenden Threads aufbewahrt.

Eine Besonderheit des `for`-Konstrukts besteht darin, dass die Iterationen der Schleife verteilt werden, ohne die Variable `i` tatsächlich zu verwenden. Während bei

```
for(i=0; i<n; i++)
  work(i);
```

die Variable `i` nach dem Ende der Schleife den Wert `n` besitzt, ist ihr Wert bei

```
#pragma omp for
for(i=0; i<n; i++)
  work(i);
```

nach Ausführung der Schleife undefiniert.

Globale Variablen und lokale Variablen, die außerhalb eines parallelen Abschnitts definiert wurden, werden von allen Threads des Teams geteilt. Im folgenden Beispiel

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

werden deshalb alle Threads auf dieselbe Variable im gemeinsamen Speicher zugreifen.

Es mag dabei überraschen, dass dieses Programmfragment auf den meisten Prozessoren nicht zuverlässig die Anzahl der Threads im Team ermittelt: Bei vielen Architekturen wird `i++` in drei Operationen zerlegt: das Lesen des aktuellen Werts der Variablen `i` aus dem Speicher, die Addition von eins, und das Schreiben des Ergebnisses in den Speicher. Da sich die Threads den Speicher teilen, kann es beispielsweise passieren, dass alle Threads den ursprünglichen Wert 0 lesen, ihre Addition durchführen, und alle den Wert 1 in den gemeinsamen Speicher schreiben. Das Ergebnis der Berechnung in einem Thread hängt also davon ab, wie viele andere Threads des Teams den Speicher schon verändert haben, bevor er ihn lesen konnte. Auf üblichen Systemen lässt sich das nicht zuverlässig voraussagen, so dass im Prinzip alle Werte zwischen 1 und der Anzahl der Threads auftreten können.

In diesem einfachen Beispiel können wir das Problem mit Hilfe der `atomic`-Direktive lösen, die sicher stellt, dass ein Befehl „atomar“, also unteilbar, ausgeführt wird:

```
int i = 0;
#pragma omp parallel
{
  #pragma omp atomic
  i++;
}
```

Dieses Programmfragment berechnet zuverlässig die Anzahl der Threads im aktuellen Team. Die `atomic`-Direktive darf nur vor bestimmten C-Statements auftreten. Abhängig von der verwendeten Architektur kann sie die Leistung eines Systems erheblich beeinträchtigen, beispielsweise falls sie intern umgesetzt wird, indem sämtlichen laufenden Threads *alle* Speicherzugriffe verboten werden. Auf älteren Intel-Prozessoren sorgt beispielsweise das `LOCK`-Präfix bei einem Maschinenbefehl dafür, dass der Speicherbus während der Ausführung dieses Befehls völlig gesperrt ist. Modernere Prozessoren verwenden glücklicherweise geschicktere Strategien.

10.3 Task-basierte Parallelisierung

Häufig treten Algorithmen auf, bei denen erst im Laufe der Ausführung ermittelt werden kann, welche Rechenoperationen ausgeführt werden müssen. In solchen Fällen hilft uns die `for`-Direktive nicht weiter, weil sie nur für Schleifen mit ungefähr gleich aufwendigen Iterationsschritten gut funktioniert, während eine Parallelisierung „per Hand“ zu aufwendig ist.

In dieser Situation können *Tasks* helfen, die mit der Version 3.0 des OpenMP-Standards eingeführt wurden. Dabei handelt es sich um strukturierte Blöcke von Anweisungen, die durch das Laufzeitsystem auf die Threads des aktuellen Teams verteilt werden. Eine Besonderheit besteht dabei darin, dass Tasks selbst weitere Tasks anlegen können, die gemeinsam mit den bereits vorhandenen verwaltet werden. Gerade bei dynamischen Datenstrukturen wie Bäumen lässt sich dadurch eine relativ

gleichmäßige Verteilung der Arbeit auf die Threads erreichen: Wir erzeugen für alle Knoten (oder aus Effizienzgründen für alle Teilbäume einer gewissen Größe) Tasks, die durch das Laufzeitsystem auf die Threads verteilt werden. Falls ein Teilbaum sehr viel größer als seine Geschwister ist, werden ihn ihm auch mehr Tasks erzeugt, die von frei werdenden Threads übernommen werden können.

Das folgende Programmfragment zählt beispielsweise die Knoten eines Binärbaums:

```
static void count(tree *t)
{
  if(t->left)
  #pragma omp task
    count(t->left);

  if(t->right)
  #pragma omp task
    count(t->right);

  #pragma omp taskwait

  t->desc = 1;
  if(t->left)
    t->desc += t->left->desc;
  if(t->right)
    t->desc += t->right->desc;
}
```

Mit der `task`-Direktive wird dabei jeweils der folgende strukturierte Anweisungsblock als Task deklariert. Im vorliegenden Fall entstehen zwei Tasks für die beiden rekursiven Aufrufe, falls zwei Söhne vorhanden sind.

Die `taskwait`-Direktive sorgt dafür, dass der aktuelle Task wartet, bis alle von ihm erzeugten Tasks fertig geworden sind. In unserem Beispiel müssen die Tasks der beiden Söhne abgeschlossen sein, bevor die Berechnung des Werts `t->desc` im Vater beginnen kann.

Die taskbasierte Programmierung kann auch Vorteile bieten, wenn viele unterschiedliche Aufgaben auf mehrere Threads zu verteilen sind. Als Beispiel wählen wir die Berechnung der LR-Zerlegung einer Matrix $A \in \mathbb{R}^{n \times n}$. Wir stellen A als Blockmatrix mit $m \times m$ Blöcken dar, also in der Form

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

wobei alle Blöcke ungefähr gleich groß sein sollten. Wenn wir die Faktoren L und R

entsprechend darstellen, also als

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

und die Gleichung $A = LR$ in Blockform schreiben, stellen wir beispielsweise fest, dass

$$\begin{aligned} A_{11} &= L_{11}R_{11}, \\ A_{1j} &= L_{11}R_{1j} && \text{für alle } j \in [2:m], \\ A_{i1} &= L_{i1}R_{11} && \text{für alle } i \in [2:m] \end{aligned}$$

gelten müssen. Also ist zuerst die LR-Zerlegung der Matrix A_{11} zu berechnen.

Anschließend können wir die Matrizen R_{1j} und L_{i1} durch Vorwärtseinsetzen ermitteln (denn es gilt $R_{11}^* L_{i1}^* = A_{i1}^*$). In dieser zweiten Phase des Algorithmus' sind alle Schritte voneinander unabhängig, so dass sie parallelisiert werden können.

Schließlich erhalten wir

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

und die Berechnung der rechten Seite dieser Gleichung kann durch $(m-1)^2$-malige Matrix-Multiplikation erfolgen. Auch in diesem Fall ist wieder jeder Matrixblock unabhängig von allen anderen zu berechnen, so dass sich eine Parallelisierung lohnt.

Sobald die rechte Seite vorliegt, können wir induktiv fortfahren, um den Rest der LR-Zerlegung zu bestimmen.

Wenn wir diesen Algorithmus task-basiert parallelisieren, müssen wir darauf achten, dass die Abhängigkeiten zwischen den einzelnen Teilmatrizen korrekt berücksichtigt werden: Die Matrizen L_{ik} und R_{kj} können für $i, j \in [k+1:m]$ beispielsweise erst berechnet werden, wenn L_{kk} und R_{kk} vorliegen.

Ein einfacher Zugang besteht darin, mit Hilfe der `taskwait`-Direktive dafür zu sorgen, dass jeweils vor der Berechnung der LR-Zerlegung eines Diagonalblocks A_{kk}, vor der Berechnung der Matrizen L_{ik} und R_{kj} für $i, j \in [k+1:m]$ durch Vorwärtseinsetzen, und vor der Aktualisierung der Blöcke A_{ij} für $i, j \in [k+1:m]$ mit Hilfe dieser Direktive dafür gesorgt wird, dass alle vorher erzeugten Tasks erfolgreich abgeschlossen wurden.

10.4 Synchronisation

Der Austausch von Daten zwischen den an einer Berechnung beteiligten Threads erfolgt über den gemeinsamen Speicher, deshalb benötigen wir einen Mechanismus, mit dem wir sicherstellen können, dass beispielsweise ein Thread, der den Wert einer von einem anderen Thread berechneten Variablen lesen muss, diesen Wert erst zu lesen versucht,

wenn der andere Thread ihn in den Speicher geschrieben hat. Bei das task-basierten Programmierung haben wir das mit der `taskwait`-Direktive bewerkstelligt.

Auf einer noch gröberen Stufe kann eine Synchronisation durch eine *Barriere* erfolgen, die sicher stellt, dass alle Threads des aktuellen Teams an einem bestimmten Punkt des Programms warten, bis alle anderen Threads ebenfalls diesen Punkt erreicht haben.

```
#pragma omp parallel
{
  work1();

  #pragma omp barrier

  work2();
}
```

In diesem Beispiel führen alle Threads die Funktion `work1` aus, die beispielsweise Teilergebnisse in den gemeinsamen Speicher schreibt. An der mit der `barrier`-Direktive erzeugten Barriere warten sie dann auf die anderen Threads, so dass sicher gestellt ist, dass `work2` erst aufgerufen wird, wenn alle Threads mit der Bearbeitung von `work1` fertig sind und ihre Ergebnisse vorliegen.

Bei komplexen Algorithmen mit einem nicht einfach vorhersehbaren Fluss von Daten ist dieser Zugang wenig attraktiv, weil sich mit ihm nur *alle* von einem Task erzeugten Tasks synchronisieren lassen, nicht gezielt einzelne von ihnen, die Daten austauschen sollen.

Deshalb bietet OpenMP eine Implementierung von *Locks*, also Sperren, mit denen sich der Zugriff auf Ressourcen des Systems steuern lässt. Ein Lock ist eine Variable des Typs `omp_lock_t`, die wir uns als ein Türschloss vorstellen können, das entweder auf- oder abgeschlossen ist. Die Funktion `omp_set_lock` versucht, das Schloss abzuschließen. Falls das Schloss vorher offen war, ist der Versuch erfolgreich. Falls das Schloss bereits abgeschlossen war, wartet die Funktion, und damit auch der Thread, im dem die Funktion aufgerufen wurde, bis ein anderer Thread das Schloss mit der Funktion `omp_unset_lock` aufschließt.

Im folgenden Beispiel wird das Lock `protect_x` verwendet, um zu verhindern, dass zwei Tasks gleichzeitig auf den Wert der Variablen `x` zugreifen.

```
#include <omp.h>

omp_lock_t protect_x;
int x = 0;

omp_init_lock(&protect_x);

#pragma omp parallel
{
  #pragma omp task
```

```
  {
    omp_set_lock(&protect_x);
    x += 17;
    omp_unset_lock(&protect_x);
  }

  #pragma omp task
  {
    omp_set_lock(&protect_x);
    x += 13;
    omp_unset_lock(&protect_x);
  }
}
```

Einer der beiden Tasks wird als erster die Funktion `omp_set_lock` ausführen und das Lock damit abschließen. Falls anschließend der andere Task dasselbe versucht, muss er warten, bis der frühere mit `omp_unset_lock` das Lock wieder aufschließt und damit signalisiert, dass er seine Arbeit getan hat.

10.5 Systeme mit verteiltem Speicher

Die Konstruktion von Systemen mit gemeinsamem Speicher ist relativ aufwendig und deshalb kostenintensiv, schließlich muss ein Kommunikationsnetz geschaffen werden, dass bei *sämtlichen* Speicherzugriffen dafür sorgt, dass alle Caches konsistent bleiben. Das macht den Einsatz dieser Technik für Systeme mit sehr vielen Prozessoren unattraktiv.

Deshalb verzichtet man bei großen Systemen in der Regel auf geteilten Speicher und stattet jeden Prozessor mit Speicher aus, auf den nur er alleine zugreifen kann. Das System besteht dann aus *Knoten*, eigenständigen Rechnern, die über ein — hoffentlich schnelles — Kommunikationsnetz miteinander verbunden sind. Einerseits vereinfacht sich das System dadurch erheblich, andererseits kann jeder Prozessor auf „seinen“ Speicher zugreifen, ohne auf andere Prozessoren Rücksicht nehmen zu müssen.

Der Datenaustausch zwischen verschiedenen Prozessoren wird allerdings bei diesem Ansatz erheblich schwieriger, denn da ein Prozessor nur auf seinen eigenen Speicher direkt zugreifen kann, ist er bei dem Zugriff auf Daten aus dem Speicher eines anderen Prozessors auf dessen Mitarbeit angewiesen.

Die Kommunikation in einem System mit verteiltem Speicher wird in der Praxis häufig durch den Austausch von Nachrichten zwischen den einzelnen Prozessoren modelliert. Der in diesem Bereich am weitesten verbreitete Standard ist das *Message Passing Interface* (MPI), das eine Reihe von C- und FORTRAN-Funktionen bietet, mit denen sich der Austausch von Nachrichten beschreiben lässt.

MPI sieht es vor, dass Programme auf Systemen laufen können, auf denen sie nicht übersetzt wurden, so dass nicht unbedingt der übliche Compiler des Entwicklungssystems zum Einsatz kommt, sondern ein auf die Zielplattform zugeschnittener, der üblicherweise

als `mpicc` oder `mpifort` aufgerufen wird. Der resultierende Binärcode wird in der Regel nicht unmittelbar ausführbar sein, stattdessen muss er über ein Hilfsprogramm wie `mpirun` gestartet werden, das ihn auf die zur Verfügung stehenden Rechnerknoten verteilt, die für den Datenaustausch nötigen Informationen übermittelt, und Prozesse einrichtet, die die Programme ausführen.

Kommunikation innerhalb eines MPI-Programms erfolgt über eine *Kommunikator* (engl. *communicator*) genannte Datenstruktur, die in C durch den Typ `MPI_Comm` dargestellt wird. Wenn ein MPI-Programm gestartet wird, stehen jedem Prozess zunächst zwei Kommunikatoren zur Verfügung: `MPI_COMM_WORLD` ist für die Kommunikation mit allen anderen Prozessen zuständig, `MPI_COMM_SELF` dagegen nur für die Kommunikation mit dem Prozess selbst.

Das einfachste MPI-Programm muss zu Beginn einmal die Funktion `MPI_Init` aufrufen, um sich mit dem MPI-Laufzeitsystem in Verbindung zu setzen, und es muss am Ende einmal die Funktion `MPI_Finalize` aufrufen, um sich wieder abzumelden. Um es etwas interessanter zu machen, ermittelt das folgende Programm auch die Anzahl der gestarteten Prozesse und die Nummer des jeweils laufenden Prozesses:

```
#include <mpi.h>
#include <stdio.h>

int
main(int argc, char **argv)
{
  int size, rank;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  (void) printf("Hello, this is process %d of %d.\n",
                rank, size);

  MPI_Finalize();

  return 0;
}
```

Die Funktion `MPI_Init` erhält Zeiger auf die Variablen `argc` und `argv`, um Befehlszeilenparameter zu finden und auszusortieren, die MPI für die interne Verwaltung benötigt.

`MPI_Comm_size` ermittelt, wie viele Prozesse ein Kommunikator enthält. In unserem Beispiel wird die Funktion für den von MPI vorgegebenen Kommunikator `MPI_COMM_WORLD` aufgerufen, so dass die Anzahl aller laufenden Prozesse ermittelt wird.

Die innerhalb eines Kommunikators zusammengefassten Prozesse sind mit null beginnend aufsteigend numeriert, und die Funktion `MPI_Comm_rank` gibt uns diese Nummer

zurück.

Sobald wir keine MPI-Funktionen mehr verwenden wollen, üblicherweise unmittelbar vor dem Ende des Programms, rufen wir `MPI_Finalize` auf, um eventuell noch im Hintergrund laufende Arbeiten abzuschließen, durch das Laufzeitsystem angelegten Hilfsspeicher freizugeben, und den Prozess im System abzumelden.

Wenn wir den Prozess mit

```
mpicc mpi_hello.c -o mpi_hello
```

übersetzen und mit

```
mpirun -np 4 ./mpi_hello
```

in vier Prozessen ausführen, könnten wir beispielsweise ein Ergebnis der Form

```
Hello, this is process 2 of 4.
Hello, this is process 1 of 4.
Hello, this is process 3 of 4.
Hello, this is process 0 of 4.
```

erhalten. Die Reihenfolge der Meldungen hängt dabei davon ab, welcher der vier laufenden Prozesse zu welchem Zeitpunkt die `printf`-Funktion aufruft. Da für das Laufzeitsystem alle Prozesse gleichberechtigt sind, kann bei jedem neuen Aufruf eine andere Reihenfolge entstehen.

Die Kommunikationsfunktionen des MPI-Standards lassen sich grob in zwei Kategorien aufteilen: Bei der *Punkt-zu-Punkt-Kommunikation* tauschen zwei Prozesse Daten miteinander aus, bei der *kollektiven Kommunikation* sind alle Prozesse eines Kommunikators beteiligt. Eine Mischform ist die mit dem MPI2-Standard hinzu gekommene *einseitige Kommunikation*, bei der in einem kollektiven Schritt Speicherbereiche in jedem Prozess definiert werden, auf die anschließend einzelne Prozesse fast beliebig zugreifen können.

Die grundlegenden Funktionen der Punkt-zu-Punkt-Kommunikation sind `MPI_Send` für das Senden einer Nachricht und `MPI_Recv` für das Empfangen. Eine Nachricht besteht dabei aus einer Anzahl von Daten desselben Typs, einem Sender, einem Empfänger und einer zusätzlichen Markierung (engl. *tag*), mit der sich beispielsweise an unterschiedlichen Stellen eines Prozesses abgeschickte Nachrichten zuordnen lassen. Die Übertragung erfolgt innerhalb des Kontexts eines Kommunikators, der beispielsweise sicher stellt, dass die Nachrichten in der Reihenfolge eintreffen, in der sie abgeschickt wurden. In dem folgenden einfachen Beispielprogramm berechnet der Prozess mit der Nummer null eine Zufallszahl, die er dann an den Prozess mit der Nummer eins sendet, der sie ausgibt.

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank == 0) {
  x = rand();
  MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

```
    printf("Sender's number:   %d\n", x);
}
else if(rank == 1) {
    MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Receiver's number: %d\n", x);
}
```

Die Funktionen `MPI_Send` und `MPI_Recv` benötigen als Parameter jeweils

1. einen Zeiger auf einen Speicherbereich, der die zu sendenden Daten enthält oder die zu empfangenden Daten aufnehmen kann,
2. die Größe dieses Speicherbereichs,
3. den Typ der in diesem Speicherbereich abgelegten Variablen,
4. die Nummer des Empfängers oder Senders,
5. die zusätzliche Markierung und
6. den Kommunikator, in dessen Kontext der Nachrichtenaustausch stattfindet.
7. Bei `MPI_Recv` kann außerdem noch ein Zeiger auf eine Variable des Typs `MPI_Status` angegeben werden, die zusätzliche Informationen über die empfangenen Daten aufnimmt, beispielsweise die Anzahl der empfangenen Daten oder die Nummer des sendenden Prozesses. Wir können auch `MPI_STATUS_IGNORE` übergeben, um auf diese Daten zu verzichten.

Während das Programm aus der Funktion `MPI_Recv` erst zurückkehrt, sobald die Nachricht vollständig empfangen wurde, darf eine MPI-Implementierung die Funktion `MPI_Send` so umsetzen, dass aus ihr zurückgekehrt wird, sobald das zu sendende Array überschrieben werden darf, aber dessen Inhalt noch nicht unbedingt gesendet oder gar empfangen wurde. Das ist sinnvoll, falls beispielsweise die Daten erst in einen Hilfsspeicher kopiert und erst von dort auf den Weg gebracht werden.

Falls wir Wert darauf legen, dass unser Programm sicher stellt, dass eine gesendete nachricht auch empfangen wurde, können wir die Funktion `MPI_Ssend` (`Ssend` steht für *synchronisierendes* Senden) verwenden, aus der das Programm erst zurück kehrt, wenn der Empfänger damit begonnen hat, die Daten zu erhalten, also insbesondere ein passendes `MPI_Recv` aufgerufen wurde.

Häufig ist es allerdings hilfreich, *weniger* Anforderungen an die Reihenfolge der Programmausführung zu stellen, beispielsweise falls mehrere Prozesse Daten senden und empfangen sollen. Im folgenden Beispiel ermitteln die Prozesse null und eins je eine Zufallszahl, die an den anderen übertragen werden soll:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
if(rank == 0) {
  x = rand();
  MPI_Ssend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  printf("Numbers: %d, %d\n", x, y);
}
else if(rank == 1) {
  y = rand();
  MPI_Ssend(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  printf("Numbers: %d, %d\n", x, y);
}
```

Dieses Programm wird *nicht* funktionieren: Prozess null wartet darauf, dass Prozess eins den Wert x empfängt, während Prozess eins darauf wartet, dass Prozess null den Wert y empfängt.

In diesem einfachen Fall ließe sich das Problem lösen, indem wir die Reihenfolge der Sende- und Empfangsfunktionen entweder in Prozess null oder Prozess eins tauschen. In allgemeinen Situationen ist das oft nicht möglich.

Ein eleganterer Weg besteht darin, *nicht-blockierende* Kommunikationsfunktionen zu verwenden. Derartige Funktionen übergeben den Auftrag für eine Operation an das MPI-System, warten aber nicht auf dessen Ausführung, sondern kehren umgehend in die aufrufende Funktion zurück. Bei `MPI_Recv` wäre das natürlich verhängnisvoll, weil nicht gesichert wäre, dass die zu empfangenden Daten schon im Speicher des aufrufenden Prozesses angekommen sind. Deshalb gehört zu einer nicht-blockierenden Funktion stets eine Variable des Typs `MPI_Request`, mit der sich überprüfen lässt, ob eine Operation erfolgreich abgeschlossen wurde.

Die nicht-blockierenden Funktionen tragen die Namen `MPI_Irecv`, `MPI_Isend` und `MPI_Issend`, wobei das „I“ jeweils für *immediate* steht, also für die unmittelbare Rückkehr aus der Funktion. Eine korrekt arbeitende Fassung unseres Beispielprogramms nimmt die folgende Form an:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank == 0) {
  MPI_Request req;

  x = rand();
  MPI_Issend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
  MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  MPI_Wait(&req, MPI_STATUS_IGNORE);
  printf("Numbers: %d, %d\n", x, y);
```

```
}
else if(rank == 1) {
  MPI_Request req;

  y = rand();
  MPI_Issend(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req);
  MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
  MPI_Wait(&req, MPI_STATUS_IGNORE);
  printf("Numbers: %d, %d\n", x, y);
}
```

Die Aufrufe der Funktion `MPI_Issend` teilen dem MPI-System jeweils nur mit, dass Daten gesendet werden sollen, warten aber nicht darauf, dass das geschieht. Erst nachdem der Prozess aus dem Aufruf der Funktion `MPI_Wait` zurückgekehrt ist, ist sicher gestellt, dass die Sendeoperation ausgeführt wurde. Da in dem Beispielprogramm eine synchronisierende Sendeoperation verwendet wird, ist auch gesichert, dass der Empfänger das passende `MPI_Recv` aufgerufen hat.

Bei der nicht-blockierenden Empfangsoperation ist klar, dass wir erst nach der Rückkehr aus `MPI_Wait` darauf hoffen dürfen, dass der Empfangspeicher sinnvolle Daten enthält. Bei der nicht-blockierenden Sendeoperation ist allerdings auch Vorsicht geboten, denn solange diese Operation nicht abgeschlossen wurde, dürfen wir die Daten im Sendespeicher nicht modifizieren, weil sie schließlich entweder noch gar nicht oder nur zum Teil übertragen worden sein könnten. Auch hier müssen wir also auf die Rückkehr aus `MPI_Wait` warten, bevor wir den Sendespeicher weiterverwenden dürfen.

Ein weiterer Vorteil der nicht-blockierenden Operationen besteht darin, dass sich mit ihrer Hilfe Rechenoperationen und Kommunikation parallel ausführen lassen. Moderne Rechnersysteme verfügen häufig über Netzwerkkarten, die unabhängig vom Hauptprozessor Daten verschicken und empfangen können (beispielsweise per DMA, *direct memory access*), so dass der Hauptprozessor lediglich den Datenaustausch in Auftrag zu geben braucht und sich anschließend um andere Dinge kümmern kann.

Als Beispiel wählen wir die Matrix-Vektor-Multiplikation mit einer Matrix $A \in \mathbb{R}^{n \times n}$, die gemäß

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

in Teilmatrizen der Größe ℓ zerlegt ist, es soll also $n = m\ell$ gelten.

Diese Matrix soll mit einem Vektor $x \in \mathbb{R}^n$ multipliziert werden, und das Ergebnis soll zu einem Vektor $y \in \mathbb{R}^n$ addiert werden, wobei beide passend in

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}$$

zerlegt ist, wobei $x_1, \ldots, x_m$ und $y_1, \ldots, y_m$ diesmal Vektoren der Länge ℓ sind, keine einzelnen Koeffizienten. Wir wollen die Operation auf einem Rechner mit verteiltem Speicher unterbringen, auf dem m Prozesse laufen. Der i-te Prozess verfügt dabei über die Teilmatrizen $A_{i1}, \ldots, A_{im}$, den Eingabevektor x_i und den Ausgabevektor y_i. Um nun

$$y_i \leftarrow y_i + \sum_{j=1}^{m} A_{ij} x_j$$

ausführen zu können, müssen die Vektoren $x_{i+1}, \ldots, x_m$ und $x_1, \ldots, x_{i-1}$ an den i-ten Prozess übermittelt werden.

Diese Berechnung zerlegen wir in m Phasen, wobei in der Phase $k \in [0 : m-1]$ jeweils

$$y_i \leftarrow y_i + A_{ij} x_j, \qquad j = (i+k) \bmod m$$

ausgeführt wird. Da wir wissen, dass in der nächsten Phase $A_{ij'}$ mit $j' = (i+k+1) \bmod m$ benötigt werden wird, können wir in Phase k bereits $x_{j'}$ von Prozess j' empfangen und x_i an Prozess $p' = (i-k-1) \bmod m$ senden. Mit nicht-blockierenden Kommunikationsfunktionen können diese Operationen parallel zu der Berechnung ablaufen. Da die Multiplikation mit der Matrix A_{ij} einen Aufwand von $\mathcal{O}(\ell^2)$ Rechenoperationen hat, aber nur $\mathcal{O}(\ell)$ Daten gesendet und empfangen werden müssen, dürfen wir hoffen, dass die Kommunikationsoperationen abgeschlossen sind, wenn wir die nächsten Teilvektoren benötigen. Die zentrale Schleife unserer Berechnung kann wie folgt umgesetzt werden:

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

i = (rank+size-1) % size;
j = (rank+1) % size;
MPI_Isend(x, l, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &sreq);
MPI_Irecv(xj[1], l, MPI_DOUBLE, j, 0, MPI_COMM_WORLD, &rreq);
mvm_block(rank, rank, x, y);
MPI_Wait(&sreq, MPI_STATUS_IGNORE);
MPI_Wait(&rreq, MPI_STATUS_IGNORE);

for(k=1; k<size; k++) {
  i = (rank+size-(k+1)) % size;
  j = (rank+(k+1)) % size;
  MPI_Isend(x, l, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &sreq);
  MPI_Irecv(xj[(k+1)%2], l, MPI_DOUBLE, j, 0, MPI_COMM_WORLD, &rreq);
  mvm_block(rank, (rank+k) % size, xj[k%2], y);
  MPI_Wait(&sreq, MPI_STATUS_IGNORE);
  MPI_Wait(&rreq, MPI_STATUS_IGNORE);
}
```

Hier gehen wir davon aus, dass `mvm_block` jeweils den durch die ersten beiden Argumente gegebenen Matrixblock mit einem Vektor multipliziert und das Ergebnis zu dem Ergebnisvektor addiert.

Der Diagonalblock wird zuerst behandelt, denn er erfordert keine Kommunikation. Parallel zu dem ersten Aufruf der Funktion `mvm_block` werden bereits die Daten für den ersten Block oberhalb der Diagonalen übertragen.

`xj[0]` und `xj[1]` sind `double`-Arrays, die jeweils den für die aktuelle und die nächste Multiplikation benötigten Teilvektor aufnehmen. In Schritt `k` wird dabei mit `xj[k%2]` gerechnet, während `xj[(k+1)%2]` mit den Daten des nächsten Schritts gefüllt wird.

Der Vektor `x` enthält den Teilvektor x_i zu dem aktuellen Prozess mit der Nummer i (im Programm `rank`), während `y` den Teilvektor y_i des Ergebnisvektors aufnimmt.

10.6 Kollektive Kommunikation

Bei bestimmten Operationen ist es hilfreich oder sogar notwendig, wenn alle Prozesse eines Kommunikators sie gemeinsam ausführen. Ein einfaches Beispiel ist die bereits aus OpenMP bekannte Barriere, die alle Prozesse nur „gemeinsam“ überwinden können, an der also jeder Prozess wartet, bis alle anderen die Barriere erreicht haben.

Solche Barrieren werden in MPI mit der Funktion `MPI_Barrier` realisiert, die als einziges Argument den Kommunikator erhält, in dem die Prozesse organisiert sind, die synchronisiert werden sollen.

Ein interessanteres Beispiel ist die Funktion `MPI_Bcast`, die Daten von einem Prozess an alle anderen Prozesse verteilt (engl. *broadcast*). Eine mögliche Anwendung ist die Übermittlung von in einem Prozess abgefragten Eingabewerten an alle Prozesse, damit sie mit denselben Parameter weiterarbeiten können:

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank == 0) {
  printf("Matrix dimension?\n");
  scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Der Wert der Variablen `n` wird dabei von Prozess null an alle Prozesse im Kommunikator `MPI_COMM_WORLD` übertragen.

Natürlich brauchen wir in diesem Fall nicht wirklich eine gesonderte Funktion, denn wir könnten die Parameter ja auch mittels der bereits bekannten Punkt-zu-Punkt-Kommunikation austauschen. Der Reiz der Broadcast-Operation besteht darin, dass sie erheblich effizienter sein kann: Beispielsweise könnte eine Implementierung zunächst die Daten an einen weiteren Prozess senden. Nun liegen die Daten bei zwei Prozessen vor, die sie *parallel* an zwei weitere senden können. Nun kennen schon vier Prozesse die Daten, die sie wiederum parallel an vier weitere senden. Insgesamt verdoppelt sich in jedem Schritt die Anzahl der Prozesse, die über die Daten verfügen, so dass p Prozesse in $\mathcal{O}(\log p)$ kollektiven Schritten informiert werden können.

Darüber hinaus kann eine gute MPI-Implementierung sich natürlich auch besondere Eigenschaften der Hardware zunutze machen, beispielsweise indem die unterschiedlichen

Geschwindigkeiten der Verbindungen zwischen einzelnen Rechnerknoten berücksichtigt werden.

Das Gegenstück der Broadcast-Operation ist die Reduce-Operation, die von der Funktion `MPI_Reduce` ausgeführt wird. Sie sammelt Daten von allen Prozessen eines Kommunikators ein und führt eine Operation aus, um sie zusammenzufügen. Ein Beispiel ist die Berechnung des Skalarprodukts zweier Vektoren $x, y \in \mathbb{R}^n$, die in Teilvektoren $x_1, \ldots, x_m$ und $y_1, \ldots, y_m$ der Länge ℓ zerlegt sind, die auf m Prozesse verteilt sind. Das Skalarprodukt kann dann als

$$\sum_{i=1}^{m} y_i^* x_i$$

dargestellt werden, so dass $y_i^* x_i$ lokal in einem Prozess berechnet werden kann und lediglich das Aufsummieren Kommunikation erfordert:

```
sum = 0.0;
for(j=0; j<l; j++)
  sum += y[j] * x[j];

MPI_Reduce(&sum, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Der Funktion `MPI_Reduce` müssen wir dabei neben den Eingabedaten in `sum` auch eine Variable `total` für das Ergebnis und eine Operation für das Zusammenfügen mitgeben, in diesem Fall `MPI_SUM`, also die Summe. Das Ergebnis steht anschließend nur dem Prozess null zur Verfügung.

Auch in diesem Fall dürfen wir darauf hoffen, dass sich durch Parallelisierung innerhalb der Reduce-Operation Zeit sparen lässt: Im ersten Schritt können sich die Prozesse zu Paaren zusammenfinden, die jeweils eine Summe berechnen. Anschließend müssen nur noch die Hälfte der Zahlen summiert werden, und wir können induktiv fortfahren, um in jedem Schritt die Anzahl der beteiligten Prozesse zu halbieren, bis nur noch der Prozess null übrig bleibt, der dann das Ergebnis kennt.

Index