3 Lineare Gleichungssysteme, direkte Verfahren

Wir verwenden die Standard-Notation für ein System linearer Gleichungen:

$$A\mathbf{x} = \mathbf{b} \,,$$

worin A die Koeffizientenmatrix, $\mathbf{b}$ die rechte Seite und $\mathbf{x}$ den Lösungsvektor bezeichnet. Wenn das System aus n Gleichungen und Unbekannten besteht, dann ist A eine $n \times n$-Matrix.

Lösungsverfahren für lineare Gleichungssysteme lassen sich in zwei Hauptgruppen unterteilen: direkte und iterative Verfahren.

- Direkte Verfahren berechnen (rundungsfehlerfreie Rechnung vorausgesetzt) eine exakte Lösung. Eliminations- und Substitutionsverfahren sowie die Cramersche Regel fallen in diese Kategorie. Wenn Sie mit Papier und Stift Systeme mit zwei oder drei Unbekannten lösen wollen, sind solche direkte Verfahren die Methoden der Wahl. Computer können heutzutage problemlos für mehrere zehntausend Gleichungen und Unbekannten direkte Lösungsmethoden verwenden.
- Iterative Verfahren berechnen schrittweise immer bessere Näherungslösungen. Diese Methoden eigenen sich aber nur für Gleichungssysteme mit spezieller Matrix-Struktur. Computer lösen damit riesig große Gleichungssysteme (mehrere Millionen Unbekannte), wie sie zum Beispiel bei numerischer Strömungssimulation oder Festigkeitsberechnungen auftreten.

Dieses Kapitel behandelt direkte Verfahren und wiederholt (was aus Mathematik 1 bekannt sein sollte) theoretische Aussagen zur Existenz und Eindeutigkeit der Lösung; iterative Methoden behandelt Kapitel 4.

Software in anerkannt hoher Qualität ist in der Programmbibliothek LAPACK (`http://www.netlib.org/lapack/`) frei verfügbar. Sie werden auch in kommerziell angebotenen Paketen nichts Besseres finden. Auch MATLAB enthält die LAPACK-Algorithmen. (Übrigens ist MATLAB ursprünglich als einfache Benutzerschnittstelle zu diesem Programmpaket entstanden).

3.1 Dreiecksmatrizen

Wenn A eine untere oder obere *Dreiecksmatrix* ist, kann man das Gleichungssystem $A\mathbf{x} = \mathbf{b}$ einfach durch schrittweises Einsetzen lösen (Vorwärts- oder Rückwärtssubstitution).

Andernfalls transformiert man das Gleichungssystem auf Dreiecksgestalt, wie es Abschnitt 3.2 beschreibt. Andere Möglichkeit: man *faktorisiert* A zuerst in ein Produkt von Dreiecksmatrizen. Das entsprechende Verfahren beschreibt Abschnitt 3.5.

Wir bezeichnen Dreiecksmatrizen mit L und R. In der üblichen Notation sind in L nur Einträge im linken unteren Dreieck von Null verschieden und alle Einträge der Hauptdiagonale gleich eins. In R sind nur Einträge im rechten oberen Dreieck einschließlich der Hauptdiagonale ungleich Null. Beispiel für $n = 4$:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \ell_{21} & 1 & 0 & 0 \\ \ell_{31} & \ell_{32} & 1 & 0 \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix}, \qquad R = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} \\ 0 & r_{22} & r_{23} & r_{24} \\ 0 & 0 & r_{33} & r_{34} \\ 0 & 0 & 0 & r_{44} \end{bmatrix}$$

Wenn die Elemente einer Dreiecksmatrix L auf einem Feld `a[i][j]` und die rechte Seite $\mathbf{b}$ auf einem Vektor `b[i]` gespeichert sind, löst das folgende Java-Programmsegment das Gleichungssystem $L\mathbf{x} = \mathbf{b}$ schrittweise durch *Vorwärts-Substitution*.

Achtung, Java zählt Feldindizes von 0 bis $n-1$; konventionelle Mathematik-Schreibweise zählt Zeilen und Spalten von 1 bis n.)

```
for (int i=0; i<n; i++) {
   x[i] = b[i];
   for (int j=0; j<i; j++) {
      x[i] -= a[i][j] * x[j];
   }
}
```

Ähnlich kompakt lässt sich die Lösung eines Gleichungssystems mit rechter oberer Dreiecksmatrixformulieren. Unterschiede zu vorhin: Die *Rückwärts-Substitution* beginnt in der letzten Zeile und schreitet von unten nach oben fort; durch das Hauptdiagonalelement wird dividiert; weiters ist hier die rechte Seite anfangs bereits auf `x[ ]` gespeichert, der Algorithmus überschreibt `x[ ]` mit dem Lösungsvektor.

```
for (int i=n-1; i>-1; i--) {
   for (int j=i+1; j<n; j++) {
      x[i] -= a[i][j] * x[j];
   }
   x[i] /= a[i][i];
}
```

Der Rechenaufwand, gemessen an der Zahl der Punktoperationen (Multiplikationen und Divisionen) beträgt für die Vorwärts-Substitution $n^2/2 - n/2$. Die Rückwärts-Substitution braucht $n^2/2 + n/2$ Punktoperationen. Für große n ist allein der quadratische Term ausschlaggebend. Wir sagen daher

Rechenaufwand bei Vorwärts- oder Rückwärts-Substitution
Lösen eines $n \times n$-Dreieckssystem erfordert $O(n^2)$ Punktoperationen.

Der Rechenaufwand wächst also quadratisch mit der Anzahl der Gleichungen.

Frage: Was tut man, wenn ein Gleichungssystem nicht in Dreiecksform vorliegt? Antwort: man formt es in ein solches um (natürlich so, dass Originalsystem und umgeformtes System äquivalent sind, also genau die gleichen Lösungen haben).

Darum geht es im nächsten Abschnitt.

3.2 Gauß-Elimination

Die einfache Gauß-Elimination läßt sich so formulieren:

Einfache Gauß-Elimination
Gegeben eine $n \times n$-Matrix A und rechte Seite $\mathbf{b}$. Sofern keines der a_{kk} zu einer Division durch Null führt, transformiert dieses Verfahren das System $A\mathbf{x} = \mathbf{b}$ auf ein äquivalentes System in oberer Dreiecksform $R\mathbf{x} = \mathbf{c}$.

Für alle Spalten $k = 1, \ldots n-1$
in Spalte k eliminiere alle Einträge unterhalb des Diagonalelements

Das Eliminieren in Spalte k läuft dabei folgendermaßen ab:

Für alle Zeilen $i = k+1, \ldots, n$ unterhalb der Diagonale
setze $p = a_{ik}/a_{kk}$
subtrahiere das p-fache der k-ten Zeile von Zeile i

Die Subtraktion wird durch folgende Schleife bewerkstelligt:

Für alle Spalten $j = k, \ldots n$
$a_{ij} = a_{ij} - pa_{kj}$
Für rechte Seite: $b_i = b_i - pb_k$

Diese Rechenvorschrift *überschreibt* Einträge in A und $\mathbf{b}$ mit den jeweiligen Zwischenresultaten und letztlich mit den Einträgen von R und $\mathbf{c}$. Sie verzichtet darauf, Einträge unterhalb der Diagonale (die eigentlich gleich Null sein sollen) zu löschen. Es wird sich später, in Abschnitt 3.5, herausstellen, dass diese Einträge eine wichtige Bedeutung haben.

Der Rechenaufwand beträgt $n^3/3 - n/3$ Punktoperationen für die Transformation der Matrix und $n^2/2 - n/2$ Punktoperationen für die Transformation der rechten Seite.

Als JAVA Code sieht Gauß-Elimination verblüffend einfach aus. Zu Beginn muss in `x[ ]` die rechte Seite gespeichert sein. In drei geschachtelte Schleifen wird schließlich die Lösung auf `x[ ]` geschrieben.

```
for (int k=0; k<n; k++) {
   for (int i=k+1; i<n; i++) {
      double p = a[i][k] / a[k][k];
      for (int j=k+1; j<n; j++) {
         a[i][j] -= p * a[k][j];
      }
      x[i] -= p * x[k];
   }
}
```

Das Programm spart es sich, im k-ten Schritt die Elemente unterhalb der Hauptdiagonale in der k-ten Spalte zu berechnen, weil ohnehin 0 herauskommen muss. Es verzichtet auch darauf, diese Nullen explizit in die Matrix hineinzuschreiben, sondern lässt dort die Zwischenresultate einfach stehen.

Das schrittweise Rückwärts-Einsetzen läßt sich mit $n^2/2 + O(n)$ Punktoperationen genäß dem Programmsegment aus dem vorigen Abschnitt erledigen. (Diese Doppelschleife verwendet nur das obere Dreieck von A; die Zwischenresultate $\neq 0$ unterhalb der Diagonale von vorhin stören daher nicht.)

Kombiniert liefern diese beiden Codesegmente einen einfachen Gleichungslöser.

Rechenaufwand bei einfacher Gauss-Elimination wächst kubisch mit der Anzahl der Gleichungen.
Gauß-Elimination löst ein $n \times n$-System $A\mathbf{x} = \mathbf{b}$ mit $O(n^3)$ Rechenoperationen.

(Genau nachgezählt sind es $n^3/3 + n^2 - n/3 = n^3/3 + O(n^2)$ Punktoperationen.)

3.3 Pivotsuche

Die einfache Gauß-Elimination hat einen Haken: Der Rechenschritt $p = a_{ik}/a_{kk}$ kann zu einer Division durch Null führen. Für eine Matrix zufällig gewählter reeller Zahlen ist das extrem unwahrscheinlich, aber Murphy's Gesetz besagt: *If anything can go wrong, it will.* Und tatsächlich versagt das Verfahren bei so simplen Systemen wie

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

weil gleich als erste Rechenoperation durch Null dividiert wird. Das ist definitiv nicht Schuld des Gleichungssystems, es hat nämlich die eindeutige Lösung $x_1 = 1, x_2 = 1$. Vertauscht man andererseits erste und zweite Gleichung, dann läuft das Verfahren problemlos.

Auch aus Gründen der Rechengenauigkeit kann es günstig sein, Gleichungen oder Unbekannte systematisch zu vertauschen. Man nennt diese Vorgehensweise *Pivotierung*.

Gauß-Elimination mit vollständiger Pivotsuche
Gegeben eine $n \times n$-Matrix A und rechte Seite $\mathbf{b}$. Dieses Verfahren transformiert das System $A\mathbf{x} = \mathbf{b}$ auf obere Dreiecksform $R\mathbf{x} = \mathbf{c}$. Die Matrix A wird dabei durch R und der Vektor $\mathbf{b}$ durch $\mathbf{c}$ überschrieben.

```
Für alle Spalten k = 1,...,n-1
    suche das betragsgrößte Element in der
    quadratischen Untermatrix aus den Zeilen
    und Spalten k bis n.

    Bringe dieses Element durch geeignetes Vertauschen
    von Gleichungen und Unbekannten an die Stelle a_kk.

    Wenn a_kk = 0
        Stopp.
    sonst
        in Spalte k eliminiere alle
        Einträge unterhalb des Diagonalelementes
        wie in der simplen Gauß-Elimination.
```

Pivot bedeutet „Drehachse, Angelpunkt". Bei der Elimination dreht sich alles darum, welche Gleichung jeweils benützt werden soll, um die entsprechende Unbekannte in den verbleibenden Gleichungen zu eliminieren. Angelpunkt ist das Element a_{kk}, mit dessen Hilfe der Faktor $p = a_{ik}/a_{kk}$ berechnet wird. Es heißt deswegen das *Pivotelement*. Ein günstiges Pivotelement durch geeignete Vertauschungen zu finden heißt *Pivotsuche* oder *-wahl*.

Die notwendigen Zeilen- und Spaltenvertauschungen komplizieren ein Rechenprogramm in Vergleich zur Dreifachschleife der einfachen Gauß-Elimination erheblich. Der Gewinn an zusätzlicher Einsicht in das Verfahren steht dazu in keinem annehmbaren Verhältnis. Deswegen ist hier kein Programm zur vollständigen Pivotsuche abgedruckt.

Üblicherweise führen Gleichungslöser keine vollständige, sondern nur *Zeilen-Pivotsuche* durch. Das heißt, sie beschränken sich der Einfachheit halber auf Vertauschen von Zeilen (=Gleichungen)[7]. Die Empfindlichkeit gegenüber Rundungsfehlern ist bei Zeilen-Pivotsuche etwas höher als bei vollständiger Pivot-Suche.

3.4 Lösbarkeit linearer Gleichungssysteme

Die Faustregel „Bei genausoviel Gleichungen wie Unbekannten gibt es immer eine Lösung“ *ist falsch*. Ich wiederhole (weil ich es bei Prüfungen immer wieder so höre): *ist falsch*!

Bei den drei Gleichungssystemen

$$\begin{aligned} x+y&=2 \\ 2x+2y&=4 \end{aligned} \qquad \begin{aligned} x+y&=2 \\ x+2y&=3 \end{aligned} \qquad \begin{aligned} x+y&=2 \\ 2x+2y&=3 \end{aligned}$$

sehen Sie hoffentlich mit freiem Auge: Beim ersten und beim zweiten ist $x = 1, y = 1$ eine Lösung. Das dritte System ist unlösbar. Beim ersten System gibt es allerdings noch unendlich viele weitere Lösungen. Diese beispiele illustrieren den allgemeinen Fall.

Lösbarkeit linearer Systeme
Für ein lineares Gleichungssystem gilt genau eine von drei Aussagen: Es gibt

- unendlich viele Lösungen;
- eine eindeutige Lösung;
- keine Lösung.

(Sie haben das in der Mathematik-Grundvorlesung gelernt!)

> Dieser Abschnitt behandelt nur Systeme mit genausoviel Gleichungen wie Unbekannten, aber die obige Aussage gilt auch für lineare Systeme mit mehr Gleichungen als Unbekannten. Bei weniger Gleichungen als Unbekannten sind nur die zwei Fälle möglich: keine Lösung oder unendlich viele Lösungen.

Das Gaußsche Eliminationsverfahren kann entscheiden, welcher Fall vorliegt und mögliche Lösungen berechnen.

3.4.1 Eliminationsverfahren

Gauß-Elimination mit vollständiger oder Zeilen-Pivotsuche transformiert die Originalmatrix A und rechte Seite $\mathbf{b}$ auf ein System in *Stufenform*: Von jeder Zeile zur nächsten nimmt (von links her gesehen) die Zahl der führenden Nullen um mindestens eins zu.

Mögliche Fälle nach Abschluss des Eliminationsverfahrens
Das System ist auf Stufenform transformiert.

- Es treten Nullzeilen in A auf und alle entsprechenden Einträge in b sind ebenfals Null: *unendlich viele Lösungen.*
- Es treten Nullzeilen in A auf, aber zumindest ein entsprechender Eintrag in b ist nicht Null: *keine Lösung.*
- Es treten keine Nullzeilen in A auf: *eine eindeutige Lösung.*

[7] Sie finden in Wikipedia unter dem Stichwort *Gaußsches Eliminationsverfahren* Pseudocode in mehreren Varianten.

Der MATLAB-Befehl `rref([A,b])` (rref steht für *reduced row echelon form*, reduzierte Stufenform) führt eine Variante des Eliminationsverfahrens durch (Gauß-Jordan Verfahren), allerdings nur mit Spalten-Pivotsuche. Für die Ergebnismatrix in der rref-Form gelten die gleichen Aussagen wie oben.

Wenn ein Computer die Elimination in Gleitkomma-Arithmetik durchführt, lässt sich nicht so leicht überprüfen, ob Einträge exakt gleich Null sind. Durch Rundungsfehler in den Eingabedaten und während der Rechnung werden Matrixelemente oft nicht exakt Null, sondern nur extrem klein. Es ist überhaupt nicht trivial, eine Schranke anzugeben, ab der Einträge als Null anzusehen sind. Dubiose Grenzfälle, in denen das Eliminationsverfahren gerade noch eine Lösung findet, obwohl Matrixzeilen schon fast null sind, heißen *numerisch singulär*.

3.4.2 Rang der Matrix und der erweiterten Matrix

Der *Rang einer $m \times n$-Matrix* ist die Anzahl ihrer linear unabhängigen Zeilen- oder Spaltenvektoren.

> Auch wenn die Matrix unterschiedlich viele Zeilen und Spalten hat, gilt: es gibt immer genau so viele linear unabhängige Zeilen wie Spalten; kurz: Zeilenrang = Spaltenrang.

Der MATLAB-Befehl `rank(A)` bestimmt den Rang der $n \times n$-Matrix A, und `rank([A,b])` den Rang der *erweiterten Koeffizientenmatrix* (der Matrix des Gleichungssystems mit rechter Seite als letzte Spalte angefügt).

Rang und Lösbarkeit: Fallunterscheidungen

- `rank(A)` = `rank([A,b])` $< n$: *unendlich viele Lösungen*
- `rank(A)` $< n$ und `rank(A)` $\neq$ `rank([A,b])`: *keine Lösung*
- `rank(A)` $= n$: *eindeutige Lösung*

Es gibt verschiedene Methoden, den Rang einer Matrix zu berechnen, zum Beispiel Transformation auf Stufenform durch Gauß-Elimination: Der Rang ist die Anzahl der von Null verschiedenen Zeilen. Es gibt dabei aber kein einfaches Entscheidungskriterium, ob ein Wert bloß infolge Rundungsfehlern oder „echt“ ungleich Null ist. MATLAB verwendet ein sehr rechenaufwendiges Verfahren (Singulärwertzerlegung), das aber die verläßlichsten Aussagen liefert.

Falls ein Gleichungssystem unendlich viele Lösungen hat, lässt sich die allgemeine Lösung entweder aus dem `rref([A,b])`-Ergebnis ablesen oder durch folgende Befehle finden:

`pinv(A)*b` liefert eine spezielle Lösung
`null(A)` liefert den *Nullraum* von A: eine Liste von linear unabhängigen Lösungen des *homogenen Systems* $A\mathbf{x} = 0$.

Die allgemeine Lösung ist die Summe aus der speziellen Lösung und einer beliebigen Linearkombination aus dem Nullraum.

> `null(A,'r')` liefert ebenfalls eine Liste von linear unabhängigen Lösungen des homogenen Systems, aber mit „schöneren“ Zahlen bei einfachen Beispielmatrizen. Allerdings rechnet diese Variante numerisch weniger zuverlässig. (Lassen Sie sich nie von äußerer Schönheit blenden, wenn dahinter Falschheit lauert!)

Am Beispiel des Systems $A\mathbf{x} = \mathbf{b}$ mit

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 5 & 6 \\ -1 & -2 & -2 & -2 \\ 3 & 6 & 8 & 10 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 2 \end{bmatrix}, \quad \texttt{rref([A,b])} = \begin{bmatrix} 1 & 2 & 0 & -2 & -2 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Aus dem `rref([A,b])`-Ergebnis lässt sich ablesen: $x_2 = \lambda$ und $x_4 = \mu$ sind frei wählbare Parameter, $x_3 = 1 - 2\mu, x_1 = -2 - 2\lambda + 2\mu$. In Vektorform:

$$\mathbf{x} = \begin{bmatrix} -2 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

Bei so einer einfachen Matrix kann MATLAB eine partikuläre Lösung auch in der Form `x=A\b` und den Nullraum mittels `null(A,'r')` berechnen. Dieses Vorgehen ist bei praktischen Problemen mit realen Daten nicht zu empfehlen, aber hier liefert es (abgesehen von einer Warnmeldung) das „schöne“ Ergebnis

$$\mathbf{x} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \mu \begin{bmatrix} 2 \\ 0 \\ -2 \\ 1 \end{bmatrix}$$

`pinv(A)*b` und `null(A)`liefern die aus numerischer Sicht vorteilhafteste Darstellung,

$$\mathbf{x} = \begin{bmatrix} -0.2069 \\ -0.41379 \\ 0.034483 \\ 0.48276 \end{bmatrix} + \lambda \begin{bmatrix} -0.77069 \\ 0.10421 \\ 0.56227 \\ -0.28113 \end{bmatrix} + \mu \begin{bmatrix} -0.48335 \\ 0.54725 \\ -0.61115 \\ 0.30558 \end{bmatrix}$$

MATLAB berechnet dieses Resultat mit aufwändigen und zuverlässigen numerischen Verfahren. Aber anders als bei den vorigen Darstellungen der Lösung ergibt probeweises Einsetzen in den Ausdruck $A\mathbf{x} - \mathbf{b}$ nicht exakt Null, sondern aufgund der Rundungsfehler Werte im Bereich 1×10^{-15}. (Schönheit hängt oft doch auch mit Wahrheit zusammen).

3.4.3 Determinante

Die Determinante determiniert, ob ein Gleichungssystem eindeutig lösbar ist.

> Gleichungssysteme $A\mathbf{x} = \mathbf{b}$ mit $\det A \neq 0$ sind eindeutig lösbar.
>
> Allerdings ist diese Regel für das numerische Rechnen unbrauchbar.

Die folgende symmetrische 8×8-Matrix lässt sich in MATLAB durch `A=rosser` erzeugen.

$$A = \begin{bmatrix} 611 & 196 & -192 & 407 & -8 & -52 & -49 & 29 \\ 196 & 899 & 113 & -192 & -71 & -43 & -8 & -44 \\ -192 & 113 & 899 & 196 & 61 & 49 & 8 & 52 \\ 407 & -192 & 196 & 611 & 8 & 44 & 59 & -23 \\ -8 & -71 & 61 & 8 & 411 & -599 & 208 & 208 \\ -52 & -43 & 49 & 44 & -599 & 411 & 208 & 208 \\ -49 & -8 & 8 & 59 & 208 & 208 & 99 & -911 \\ 29 & -44 & 52 & -23 & 208 & 208 & -911 & 99 \end{bmatrix}$$

Für sie berechnet MATLAB derzeit[8] $\det A = -9480{,}580$ also deutlich $\det A \neq 0$. Damit wären Gleichungssysteme mit A eindeutig lösbar. Als Rang berechnet MATLAB aber (korrekt) `rank(A)`=7, und weil $7 < 8$ ist, kann es keine eindeutigen Lösungen geben. MATLABs Wert für die Determinante ist ziemlich falsch.

Für die 6×6-Matrix H, eine sogenannte Hilbert-Matrix,

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{bmatrix}$$

berechnet MATLAB $\det A = 5{,}3673 \times 10^{-18}$, und das könnte man mit voller Berechtigung gerundet für $\det A = 0$ durchgehen lassen. Als Rang berechnet MATLAB aber (korrekt) `rank(H)`=6. Gleichungssysteme mit H sind also eindeutig lösbar (allerdings extrem anfällig gegenüber Rundungsfehlern).

Diese Beispiele illustrieren:

> Der numerisch berechnete Wert einer Determinante sagt nichts über die Lösbarkeit eines Gleichungssystems aus.

3.5 LR-Zerlegung

Die einfache Gauß-Elimination liefert (wenn sie nicht abbricht) mehr als die Transformation auf Dreiecksgestalt. Sie kann gleichzeitig auch eine Zerlegung

$$A = LR$$

errechnen, wobei L eine untere Dreiecksmatrix mit Einsen in der Hauptdiagonale und R eine obere Dreiecksmatrix ist.

> **LR-Zerlegung**
> Das Gaußsche Eliminationsverfahren ohne Pivotwahl faktorisiert (wenn es nicht abbricht) eine Matrix A in ein Produkt $A = LR$ aus einer linken unteren Dreiecksmatrix L und einer rechten oberen Dreiecksmatrix R.

Nach Durchführen einer Gauß-Elimination mit Pivot-Suche enthalten oberes und unteres Dreieck der Ergebnismatrix die LR-Zerlegung einer Matrix mit - im Vergleich zur Ausgangsmatrix - entsprechend vertauschten Zeilen und Spalten.

Die Elemente von L sind 1 entlang der Hauptdiagonale, und darunter gleich den Multiplikatoren $p = a_{ik}/a_{kk}$ an den entsprechenden Stellen (i, k). Die Elemente von R sind genau jene, die das Eliminationsverfahren in das obere rechte Dreieck schreibt.

Die einzige Änderung im Programm auf Seite 30 zum Eliminationsverfahren ist, sich die Zwischenresultate `p` zu merken. Praktischerweise lässt sich jedes `p` auf dem entsprechenden Feldelement `a[i][k]` speichern; das Verfahren eliminiert nämlich gerade diesen Eintrag, erzeugt also an dieser Stelle eine Null. Die Null muss man sich nicht merken, aber dafür kann man an der dadurch freigewordenen Stelle das Zwischenresultat `p` speichern.

[8] mit Version 2022b. Der Wert mit Version 2021b war $\det A = -10\,611$. Ältere Versionen so um 2018 lieferten $\det A = -9478{,}9$; die 2015-Version bringt $-9448{,}8$; vor 2013 war $\det A = -13\,017$. Es sollte Sie beunruhigen, dass ein Rechenergebnis je nach Programmversion so unterschiedlich ausfällt!

Computerprogramme formulieren das Verfahren in aller Regel so, dass die Originalmatrix A durch R und L überschrieben wird. Das obere Dreieck von A enthält nach erfolgreichem Ablauf die Nichtnull-Einträge von R; die Einsen in der Hauptdiagonale von L verstehen sich von selbst, man braucht sie nicht zu speichern; die restlichen Nichtnull-Elemente von L finden unterhalb der Hauptdiagonale von A Platz. Das Elegante an dieser Speicherung ist, dass sie sich im Verlauf des Verfahrens quasi von selbst ergibt.

Siehe Übungsunterlagen für weitere Informationen!

Für die LR-Zerlegung braucht man keine rechte Seite. Die kommt erst später ins Spiel. Zur Lösung des Systems $A\mathbf{x} = \mathbf{b}$, wenn $A = LR$ bereits gegeben ist, formt man nämlich um:

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} && \\ (LR)\mathbf{x} &= \mathbf{b} && \\ L(R\mathbf{x}) &= \mathbf{b} && \text{setze } \mathbf{y} = R\mathbf{x} \\ L\mathbf{y} &= \mathbf{b} && \text{löse durch Vorwärts-Substitution nach } \mathbf{y} \\ R\mathbf{x} &= \mathbf{y} && \text{löse durch Rückwärts-Substitution nach } \mathbf{x} \end{aligned}$$

Der Rechengang und der Rechenaufwand sind der einfachen Gauß-Elimination völlig äquivalent. Der Vorteil der LR-Zerlegung zeigt sich aber, wenn mehrere Systeme mit der selben Matrix A und unterschiedlichen rechten Seiten $\mathbf{b}_1, \mathbf{b}_2, \ldots$ gelöst werden sollen. Die LR-Zerlegung ist der arbeitsintensive Teil ($\sim n^3/3$ Punktoperationen) und muss nur einmal durchgeführt werden. Die einzelnen Lösungen kosten dann nur mehr $\sim n^2$ Punktoperationen pro rechter Seite.

Für symmetrische Matrizen lassen sich spezielle Varianten der Gauß-Elimination durchführen. Ausnützen der Symmetrie spart Rechenzeit und Speicherplatz. Eine mögliche Zerlegung ist

$$A = LDL^T \,,$$

mit einer Diagonalmatrix D. Die *Cholesky-Zerlegung*

$$A = LL^T$$

ist für symmetrisch positiv definiter Matrizen das Standardverfahren.

3.6 Ein Rechenbeispiel zu Gauß-Elimination und LR-Zerlegung

Das Gaußsche Eliminationsverfahren ist eine Rechenvorschrift zur systematischen Elimination von Unbekannten. Bei Gleichungssystemen mit „einfachen“ Zahlen weicht man oft vom systematischen Weg ab und versucht, den Rechengang abzukürzen (z. B. schon vorhandene Nullen auszunützen). Dabei besteht immer die Gefahr, „im Kreis herum“ zu rechnen, Gleichungen nicht oder doppelt zu verwenden. Es ist daher wichtig, eine allgemein gültige Rechenvorschrift angeben zu können.
Gegeben sei das Gleichungssystem $A \cdot \mathbf{x} = \mathbf{b}$ mit

$$A = \begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix} 6 \\ 6 \\ 14 \end{bmatrix}$$

Man rechnet man mit der *erweiterten Koeffizientenmatrix*

$$[A\,\mathbf{b}] = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 10 & 20 & 23 & 6 \\ 15 & 50 & 67 & 14 \end{bmatrix}$$

Elimination in der ersten Spalte

(Vergleichen Sie mit dem Algorithmus auf Seite 30.)

$n = 3$; in der Spalte $k = 1$ sollen alle Einträge unterhalb des Diagonalelementes eliminiert werden, das sind die Elemente in Zeilen $i = 2, 3$

$k = 1, i = 2$: Elimination in erster Spalte, zweite Zeile

Elimination von $a_{ik} = a_{21} = 10$ mittels des Diagonalelements $a_{kk} = 5$ von Zeile $k = 1$. Multipliziere erste Zeile mit $a_{ik}/a_{kk} = 2$ und subtrahiere:

$$\begin{array}{cccc|} 10 & 20 & 23 & 6 & \\ 10 & 12 & 14 & 12 & |- \\ \hline 0 & 8 & 9 & -6 & \end{array}$$

$k = 1, i = 3$: Elimination in erster Spalte, dritte Zeile

Elimination von $a_{ik} = a_{31} = 15$ mittels des Diagonalelements $a_{kk} = 5$ von Zeile $k = 1$. Multipliziere erste Zeile mit $a_{ik}/a_{kk} = 3$ und subtrahiere:

$$\begin{array}{cccc|} 15 & 50 & 67 & 14 & \\ 15 & 18 & 21 & 18 & |- \\ \hline 0 & 32 & 46 & -4 & \end{array}$$

Transformierte erweiterte Koeffizientenmatrix

nach Elimination in der ersten Spalte:

$$[A\,\mathbf{b}]^{(1)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 32 & 46 & -4 \end{bmatrix}$$

Elimination in der zweiten Spalte

In der Spalte $k = 2$ sollen alle Einträge unterhalb des Diagonalelementes eliminiert werden, das ist nur mehr das Element in Zeile $i = 3$

$k = 2, i = 3$: Elimination in zweiter Spalte, dritte Zeile

Elimination von $a_{ik} = a_{32} = 32$ mittels des Diagonalelements $a_{kk} = 8$ von Zeile $k = 2$. Multipliziere zweite Zeile mit $a_{ik}/a_{kk} = 4$ und subtrahiere:

$$\begin{array}{cccc|} 0 & 32 & 46 & -4 & \\ 0 & 32 & 36 & -24 & |- \\ \hline 0 & 0 & 10 & 20 & \end{array}$$

Transformierte erweiterte Koeffizientenmatrix

Nach Elimination in der zweiten Spalte ist das Eliminationsverfahren beendet.

$$[A\,\mathbf{b}]^{(2)} = \begin{bmatrix} 5 & 6 & 7 & 6 \\ 0 & 8 & 9 & -6 \\ 0 & 0 & 10 & 20 \end{bmatrix}$$

Rücksubstitution

Aus der dritten Zeile:

$$\begin{aligned} 10x_3 &= 20 \\ x_3 &= 2 \end{aligned}$$

Einsetzen für x_3 in zweiter Zeile

$$\begin{aligned} 8x_2 + 9x_3 &= -6 \\ 8x_2 + 18 &= -6 \\ 8x_2 &= -24 \\ x_2 &= -3 \end{aligned}$$

Einsetzen für x_2 und x_3 in erster Zeile

$$\begin{aligned} 5x_1 + 6x_2 + 7x_3 &= 6 \\ 5x_1 - 18 + 14 &= 6 \\ 5x_1 &= 10 \\ x_1 &= 2 \end{aligned}$$

Der Matlab-Befehl `x = A\b` arbeitet im Prinzip nach diesem Verfahren, allerdings in Form der LR-Zerlegung mit Spaltenpivotsuche.

Mehrere rechte Seiten

Sind zur selben Matrix A Lösungen für mehrere rechte Seiten zu berechnen, fügt man diese der erweiterten Koeffizientenmatrix als weitere Spalten an und rechnet in gleicher Weise.

Pivotwahl

Bei diesem Beispiel war es nicht notwendig, Gleichungen zu tauschen, um Divisionen durch Null zu vermeiden. Bei Spaltenpivotwahl hätte man vor dem ersten Schritt die dritte Gleichung zur ersten gemacht (weil sie den betragsgrößten Eintrag in der ersten Spalte hat).

LR-Zerlegung

Die auf Dreiecksform transformierte Matrix und die entsprechenden Pivot-Faktoren liefern auch die LR-Zerlegung. Eine rechte Seite ist für die LR-Zerlegung nicht notwendig.

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 0 & 8 & 9 \\ 0 & 0 & 10 \end{bmatrix}$$

Der MATLAB-Befehl `[L, U]=lu(A)` verwendet im Prinzip das Gaußsche Eliminationsverfahren, aber liefert zu diesem Zahlenbeispiel eine andere „quasi“-LR-Zerlegung. Die U- bzw. R-Matrix[9] ist eine echte obere Dreiecksmatrix, L ist eine „durcheinandergeratene“ untere Dreiecksmatrix. Begründung: Spaltenpivotsuche vertauscht Zeilen in der Matrix und ändert im Rechengang Reihenfolge und Zahlenwerte. Sie verringert dadurch die Rundungsfehler.

[9] Dem deutschen Fachbegriff Links-Rechts-Zerlegung entspricht auf Englisch *lower–upper (LU) decomposition* oder *factorization*. Deswegen heißt der entsprechende MATLAB-Befehl `lu` und nicht `lr`.

Zerlegung mit dem Befehl `[L, U]=lu(A)` liefert

$$\begin{bmatrix} 5 & 6 & 7 \\ 10 & 20 & 23 \\ 15 & 50 & 67 \end{bmatrix} = \begin{bmatrix} 1/3 & 4/5 & 1 \\ 2/3 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 15 & 50 & 67 \\ 0 & -40/3 & -65/3 \\ 0 & 0 & 2 \end{bmatrix}$$

3.7 Weitere Anwendungen der LR-Zerlegung

3.7.1 Determinante

Determinante
Wenn eine Faktorisierung $A = LR$ gegeben ist, ist $\det A$ das Produkt der Hauptdiagonalelemente von R.

Begründung: Die Determinante einer Dreiecksmatrix ist das Produkt der Hauptdiagonalelemente. In L stehen dort lauter Einsen, somit ist $\det L = 1$. Nach den Rechenregeln für Determinanten ist dann

$$\det A = \det(LR) = (\det L)(\det R) = \det R \; .$$

Rechenaufwand für eine $n \times n$-Matrix: $n^3/3 + 2n/3 - 1$ Punktoperationen.

Vergleiche dazu das klassische Verfahren, Entwickeln nach Zeilen oder Spalten: Für eine $n \times n$-Matrix sei dafür $w(n)$ der Rechenaufwand an Punktoperationen. Im allgemeinen Fall sind n Unterdeterminanten von $(n-1) \times (n-1)$-Matrizen zu berechnen und noch mit den jeweiligen Elementen zu multiplizieren. Für den Rechenaufwand gilt also

$$w(n) = nw(n-1) + n = n(w(n-1) + 1) \; .$$

Die Funktion $w(n)$ wächst rasch, stärker als die Faktorielle $n!$. Für die Determinante einer 10×10-Matrix braucht ein schneller PC (10^7 Multiplikationen/sec) eine knappe Sekunde, schon bei einer 13×13-Matrix geht sich eine Viertelstunde Kaffeepause aus, auf die Determinante einer 15×15-Matrix warten Sie zweieinhalb Tage, dreizehn Jahrtausende bei einer 20×20-Matrix, und eine 25×25-Matrix wäre nach 80 Milliarden Jahren noch nicht fertig.
Das rasante Wachsen von $w(n)$ und den vergleichsweise geringen Aufwand der LR-Zerlegung illustriert die folgende Tabelle. Sie soll eindringlich auf die Bedeutung rechengünstiger Algorithmen und den Unterschied zwischen polynomialer und exponentieller Laufzeit hinweisen.

n	$w(n)$	$n^3/3 + 2n/3 - 1$
2	2	3
3	9	10
4	40	23
5	205	44
6	1 236	75
7	8 659	118
8	69 280	175
9	623 529	248
10	6 235 300	339
15	2 246 953 104 075	1 134
20	4 180 411 311 071 440 000	2 679
25	26 652 630 354 867 072 870 693 625	5 224

3.7.2 Inverse

Die zu einer (nichtsingulären) Matrix A inverse Matrix A^{-1} braucht kaum ein Rechenverfahren in expliziter Form. Ist etwa der Vektor $\mathbf{x} = A^{-1}\mathbf{y}$ gefragt, so erhält man $\mathbf{x}$ auch genauso gut als Lösung des Gleichungssystems $A\mathbf{x} = \mathbf{y}$, und das ist von Rechenaufwand und -genauigkeit her günstiger.

Warnung 1: Bevor Sie eine Inverse berechnen, fragen Sie dreimal nach, ob Sie diese wirklich brauchen.

Warnung 2: Falls Sie sich noch an die aus der linearen Algebra bekannte Formel (die mit den Determinanten der Kofaktoren) erinnern: vergessen Sie diese. Sie ist von theoretischer Bedeutung, weil sie die Existenz von Inversen nichtsingulärer Matrizen zeigt. Berechnen sollten Sie die Inverse so nicht (überlegen Sie: Rechenaufwand $O(n^5)$, wenn Sie die einzelnen Unterdeterminanten alle mittels LR-Zerlegung rechnen; exponentieller Rechenaufwand, wenn Sie die Determinanten entwickeln).

Wenn es denn sein muss, gehen Sie so vor: Nennen Sie die erste Spalte der Inversen $\mathbf{x}_1$. Die erste Spalte der Einheitsmatrix I ist der Einheitsvektor $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$. Laut Definition gilt

$$AA^{-1} = I \ .$$

Daher gilt insbesondere

$$A\mathbf{x}_1 = \mathbf{e}_1 \ .$$

Das heißt: die erste Spalte der Inversen erhalten Sie als Lösung eines Gleichungssystems mit dem ersten Einheitsvektor als rechter Seite.

In offenkundiger Verallgemeinerung:

Inverse
Die i-te Spalte von A^{-1} ist Lösung des Gleichungssystems

$$A\mathbf{x}_i = \mathbf{e}_i \ .$$

Hier haben Sie also Gleichungssysteme mit derselben Matrix A und mehreren rechten Seiten zu lösen. Vorgangsweise:

Zerlegung $A = LR$; (Aufwand $(n^3 - n)/3$).
Für $i = 1, \ldots, n$
    Lösung $LR\mathbf{x}_i = \mathbf{e}_i$; (Aufwand jeweils n^2).

Rechenaufwand gesamt $(4n^3 - n)/3$.

Der *Gauß-Jordan-Algorithmus* ist eine speziell günstig organisierte Form des Eliminationsverfahrens; er eignet sich gut zur Berechnung mit Stift und Papier. (Sie kennen dieses Verfahren aus Mathematik 1.)

3.7.3 Symmetrisch positiv definite Matrizen

Einfache Argumente der linearen Algebra zeigen: Für symmetrisch positiv definite Matrizen bricht die einfache Gauß-Elimination nie wegen $a_{kk} = 0$ ab. Pivot-Suche ist daher unnötig. Umgekehrt kann man symmetrisch positiv definite Matrizen dadurch charakterisieren, dass die im k-ten Schritt der LR-Zerlegung auftretenden a_{kk} alle > 0 sind.

Allerdings führt man bei symmetrisch positiv definiten Matrizen (wie bereits auf Seite 36 erwähnt) eher Zerlegungen der Form $A = LL^T$ (Cholesky-Zerlegung) oder $A = LDL^T$ durch. Vorteil: bei geschickter Programmierung weniger Rechenzeit und Speicherplatz erforderlich.

3.7.4 Unvollständige Zerlegung

Auch wenn in einer Matrix A die meisten Elemente null sind, können die Faktoren L und R deutlich mehr Nichtnull-Einträge enthalten. Durch Gauß-Elimination werden zusätzliche *Fällterme* erzeugt (also Elemente $\neq 0$ an Stellen, wo die Ausgangsmatrix Elemente $= 0$ hatte). Wenn man alle (oder kleine) Fällterme einfach vernachlässigt, reduzieren sich der Rechenaufwand und benötigte Speicherplatz einer solchen *unvollständigen Zerlegung* drastisch. Natürlich gilt dann nicht mehr $LR = A$, sondern $LR = \tilde{A}$ für eine Näherung $\tilde{A}$ an A. Unvollständigen Zerlegungen sind leistungsfähige Präkonditionierer für iterative Gleichungslöser.

Das Prinzip lässt sich durch eine geringfügige Änderung im Beispielprogramm zur LR-Zerlegung illustrieren (mehr Informationen im Übungsteil): Man ersetze dort in der innersten Schleife

```
For j = k + 1 To n
   a(i, j) = a(i, j) - p * a(k, j)
Next
```

durch

```
For j = k + 1 To n
   if a(i, j) <> 0 then
      a(i, j) = a(i, j) - p * a(k, j)
Next
```

Damit ist aus der LR-Zerlegung eine unvollständige Zerlegung ohne zusätzliche Fällterme geworden. In dieser Form spart das Programm allerdings keinen Speicherplatz und nicht wirklich Rechenzeit. Dazu wären Datenstrukturen notwendig, die gezielt nur die Nichtnull-Elemente speichern, auf diese zugreifen und damit manipulieren (Speicherformate für spärlich besetzte Matrizen).

3.8 Fehlerempfindlichkeit

Rundungsfehler und Fehler in den Eingabedaten verfälschen eine Matrix A zu $A + \delta A$ und die rechte Seite $\mathbf{b}$ zu $\mathbf{b} + \delta\mathbf{b}$. Die Lösung dieses leicht veränderten Systems wird um ein (hoffentlich nicht zu großes) $\delta\mathbf{x}$ von der Lösung des unverfälschten Systems abweichen:

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b} \quad .$$

Wie hängt $\delta\mathbf{x}$ von δA und $\delta\mathbf{b}$ ab?

Konditionszahl
Die *Konditionszahl* $\kappa(A)$ misst, wie empfindlich in einem System $A\mathbf{x} = \mathbf{b}$ der relative Fehler von $\mathbf{x}$ von kleinen relativen Änderungen in A und $\mathbf{b}$ abhängt.

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \leq \kappa(A) \left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} \right)$$

In der obigen Ungleichung bezeichnet $\|\cdot\|$ sowohl eine Vektornorm (zum Beispiel 1-, 2- oder ∞-Norm) als auch die entsprechende Matrixnorm. Aus den Rechenregeln für Vektor- und Matrixnormen (vergleiche Abschnitt 2.3) lässt sich herleiten:

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

Beweisskizze: Beginne mit dem gestörten System

$$(A + \delta A)(\mathbf{x} + \delta\mathbf{x}) = \mathbf{b} + \delta\mathbf{b}\,,$$

löse Klammern auf

$$A\mathbf{x} + A \cdot \delta\mathbf{x} + \delta A \cdot \mathbf{x} + \delta A \cdot \delta\mathbf{x} = \mathbf{b} + \delta\mathbf{b}\,;$$

weil $A\mathbf{x} = \mathbf{b}$, können wir links $A\mathbf{x}$ und rechts $\mathbf{b}$ streichen. Für kleine $\delta\mathbf{b}$ und δA ist das Produkt $\delta A \cdot \delta\mathbf{x}$ von höherer Ordnung klein; wir vernachlässigen ihn hier. Somit bleibt

$$A \cdot \delta\mathbf{x} + \delta A \cdot \mathbf{x} = \delta\mathbf{b}\ .$$

Daraus lässt sich $\delta\mathbf{x}$ ausdrücken:

$$\delta\mathbf{x} = A^{-1}\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\ ,$$

wende auf beiden Seiten eine Vektornorm an,

$$\|\delta\mathbf{x}\| = \|A^{-1}\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\|\,,$$

nütze eine Eigenschaft der Matrixnorm, Ungleichung (9),

$$\|\delta\mathbf{x}\| \le \|A^{-1}\| \cdot \|\left(\delta\mathbf{b} - \delta A \cdot \mathbf{x}\right)\|\,,$$

verwende die Dreiecksungleichung,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\left(\|\delta\mathbf{b}\| + \|\delta A \cdot \mathbf{x}\|\right)\ ,$$

erweitere die Terme in der Klammer,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\|\left(\frac{\|A\mathbf{x}\|}{\|\mathbf{b}\|}\|\delta\mathbf{b}\| + \frac{\|A\|}{\|A\|}\|\delta A \cdot \mathbf{x}\|\right)\ ,$$

vewende noch einmal eine Ungleichung für die Matrixnormen und hebe $\|A\|$ heraus,

$$\|\delta\mathbf{x}\| \le \|A^{-1}\| \cdot \|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|}\|\mathbf{x}\| + \frac{\|\delta A\|}{\|A\|}\|\mathbf{x}\|\right)\ ,$$

eine letzte Division durch $\|\mathbf{x}\|$, und wir sind fertig:

$$\frac{\|\delta\mathbf{x}\|}{\|\mathbf{x}\|} \le \|A^{-1}\| \cdot \|A\|\left(\frac{\|\delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\delta A\|}{\|A\|}\right)\ .$$

Der relative Fehler in $\mathbf{x}$ kann also $\kappa(A)$-mal größer als der relative Fehler in A und $\mathbf{b}$ sein. Ein Gleichungssystem, dessen Matrix eine große Konditionszahl hat, wird sehr empfindlich auf Fehler in den Eingabedaten reagieren. Ein solches System heißt *schlecht konditioniert*. Geometrische Veranschaulichung: schleifender Schnitt zweier Geraden.

Die Berechnung der Konditionszahl direkt gemäß der Definition würde die Berechnung der Inversen erfordern und wäre unsinnig aufwendig. Viele Gleichungslöser liefern Schätzungen von $\kappa(A)$ als Nebenprodukt. Es gilt zum Beispiel

$$\kappa(A) \ge \frac{\max|\lambda|}{\min|\lambda|}$$

(Verhältnis vongrößtem zu kleinsten Eigenwert-Bertag; Eigenwerte werden in Abschnitt **??** behandelt).

4 Iterative Verfahren für lineare Gleichungssysteme

Gegeben sei ein lineares Gleichungssystem in n Gleichungen und Unbekannten.

$$\begin{array}{lcl} a_{11}x_1 + a_{12}x_2 + & \ldots & + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + & \ldots & + a_{2n}x_n = b_2 \\ \vdots & & \vdots \quad\; \vdots \\ a_{n1}x_1 + a_{n2}x_2 + & \ldots & + a_{nn}x_n = b_n \end{array} \tag{10}$$

In Matrixschreibweise

$$A\mathbf{x} = \mathbf{b}\,. \tag{11}$$

Das klassische Lösungsverfahren dafür, jedenfalls für Systeme von zwei bis einigen hundert Gleichungen, ist Gauß-Elimination. Sie wird in Kapitel 3 systematisch behandelt. Aus vielen Anwendungen (Strömungssimulation, Seismik, Computertomographie, Festigkeitsrechnungen mit finiten Elementen...) resultieren Gleichungssysteme mit vielen tausend oder sogar Millionen Unbekannten. Solche Systeme werden meist iterativ gelöst. Sie lernen hier nur einige Basis-Verfahren kennen, auf denen die leistungsfähigeren Methoden aufbauen.

4.1 Einfache iterative Verfahren: Jacobi, Gauß-Seidel, SOR

Angenommen, die Diagonalelemente a_{ii} einer $n \times n$-Matrix A sind alle ungleich null. Dann wäre folgendes Rezept zur Lösung von $A\mathbf{x} = \mathbf{b}$ möglich (ein Fixpunkt-Verfahren):

Jacobi-Verfahren für $A\mathrm{x} = \mathrm{b}$, einfach formuliert
Löse jede Gleichung nach ihrem Diagonal-Term auf, setze Startwerte ein, iteriere.

Ausführlicher in der komponentenweisen Schreibweise (10) formuliert: Bringen Sie jeweils in der i-ten Zeile alle Terme bis auf den i-ten auf die rechte Seite und lösen Sie nach x_i auf. Ein entsprechend umgeformtes 3×3-System sieht dann so aus:

$$\begin{aligned} x_1 &= (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11} \\ x_2 &= (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \end{aligned}$$

Angenommen, $\mathbf{x}^{(k)}$ ist ein näherungsweiser Lösungsvektor. Das Jacobi-Verfahren erzeugt eine neue Näherung durch

$$\begin{aligned} x_1^{(k+1)} &= (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11} \\ x_2^{(k+1)} &= (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22} \\ x_3^{(k+1)} &= (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33} \end{aligned}$$

Vielleicht finden Sie Matrix-Schreibweise übersichtlicher. Dazu definieren wir eine Matrix $D = [d_{ij}]$ mit den gleichen Diagonalelementen wie A und null in allen Nichtdiagonalelementen. Die restlichen Elemente von A schreiben wir in eine Matrix E:

$$A = D + E \text{ mit } D = [d_{ij}], \quad d_{ij} = \begin{cases} a_{ii} & \text{falls } i = j, \\ 0 & \text{sonst.} \end{cases} \qquad E = A - D\,. \tag{12}$$

Das Gleichungssystem (11) lässt sich dann äquivalent umformen zu

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (D+E)\mathbf{x} &= \mathbf{b} \\ D\mathbf{x} &= \mathbf{b} - E\mathbf{x} \\ \mathbf{x} &= D^{-1}(\mathbf{b} - E\mathbf{x}) \ . \end{aligned}$$

Die letzte Gleichung ist eine Fixpunktgleichung. Die entsprechende Fixpunkt-Iteration

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{13}$$

heißt *Jacobi-Verfahren* .is called *Jacobi method* .

Iterationsschritt des Jacobi-Verfahrens
In Matrix-Schreibweise für Zerlegung $A = D + E$:

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Komponentenweise geschrieben: für $i = 1, \ldots, n$

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

Das Jacobi-Verfahren nützt nicht die aktuellste Information zur Berechnung von $x_i^{(k+1)}$. Beispielsweise verwendet es $x_1^{(k)}$ zur Berechnung von $x_2^{(k+1)}$, obwohl $x_1^{(k+1)}$ bereits verfügbar ist. Wenn wir das Verfahren so formulieren, dass es immer die aktuellsten Näherungswerte an die x_i verwendet, erhalten wir das *Gauß-Seidel-Verfahren* .

Für die Matrix-Schreibweise des Gauß-Seidel-Verfahrens definieren wir eine Matrix $C = [c_{ij}]$ mit den gleichen Elementen wie A in und unterhalb der Hauptdiagonale und Null oberhalb der Hauptdiagonale. Die restlichen Elemente von A schreiben wir in eine Matrix E:

$$A = C + E \text{ mit } C = [c_{ij}], \quad c_{ij} = \begin{cases} a_{ij} & \text{falls } i \geq j, \\ 0 & \text{sonst.} \end{cases} \qquad E = A - C \ . \tag{14}$$

Dieselben Schritte, die für das Jacobi-Verfahren zur Fixpunkt-Gleichung 13 geführt haben, können wir mit der Matrix C statt D wiederholen und erhalten die Iterationsvorschrift für das Gauß-Seidel-Verfahren:

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{15}$$

Iterationsschritt des Gauß-Seidel-Verfahrens
einfach formuliert
Löse der Reihe nach jede Gleichung nach ihrem Diagonal-Term auf, setze Startwerte ein, iteriere, verwende jeweils neuerste Näherungswerte.

Matrix-Schreibweise für Zerlegung $A = C + E$

$$\mathbf{x}^{(k+1)} = C^{-1}(\mathbf{b} - E\mathbf{x}^{(k)})$$

Komponenten-Schreibweise
für $i = 1, \ldots, n$

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

Das Gauß-Seidel-Verfahren lässt sich oft deutlich beschleunigen, wenn man den aus der Iterationsformel erhaltenen Wert nicht direkt verwendet, sondern die Änderung von $x_i^{(k)}$ zu $x_i^{(k+1)}$ noch um einen Faktor $\omega > 1$ vergrößert. Dieses iterative Verfahren heißt *SOR-Verfahren* (SOR steht für *successive overrelaxation*)

Ein geeigneten Wert für ω lässt sich aber nicht so einfach angeben. Die Theorie sagt: $1 \leq \omega < 2$, mit Werten eher in der Nähe von 2. Für $\omega = 1$ reduziert sich SOR auf Gauß-Seidel.

Iterationsschritt des SOR-Verfahrens
einfach formuliert
Jeweils neuer Näherungswert zuerst als Zwischenresultat $y_i^{(k+1)}$ aus Gauß-Seidel-Schritt; endgültiger Näherungswert durch Extrapolation (Überrelaxation) aus alter Näherung und Zwischenresultat: $x_i^{(k+1)} = \omega y_i^{(k+1)} + (1 - \omega) x_i^{(k)}$

Die Komponenten-Schreibweise sieht hier bereits etwas unübersichtlich aus.

für $i = 1, \ldots, n$

$$y_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}$$

$$x_i^{(k+1)} = \omega y_i^{(k+1)} + (1 - \omega) x_i^{(k)}$$

Auch dieses Verfahren lässt sich mit einer Zerlegung $A = B + E$ ähnlich wie die Gleichungen 13 und 15 anschreiben:

$$\mathbf{x}^{(k+1)} = B^{-1}(\mathbf{b} - E\mathbf{x}^{(k)}) \tag{16}$$

Darin ist B eine Kombination der Matrizen C und D von vorhin (12, 14), und zwar

$$B = C + \left(\frac{1}{\omega} - 1 \right) D$$

4.2 Konvergenz des Jacobi- und des Gauß-Seidel-Verfahrens

Nicht für jede beliebige Matrix A konvergieren die drei oben vorgestellten Verfahren. Die Konvergenz des Jacobi-Verfahrens lässt sich zeigen, indem man nachweist, dass es sich bei der Fixpunkt-Iteration um eine kontrahierende Abbildung handelt. Dazu definieren wir:

Eine $n \times n$-Matrix $A = [a_{ij}]$ heißt *stark diagonaldominant*, wenn

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{für} \quad i = 1, 2, \ldots, n$$

Es muss also in jeder Zeile die Summe der Beträge der Nichtdiagonalelemente kleiner sein als der Betrag des Diagonalelementes.

Konvergenz des Jacobi-Verfahrens
Das Jacobi-Verfahren konvergiert bei Gleichungssystemen mit stark diagonaldominanter Matrix für beliebige Startwerte zur eindeutigen Lösung.

Beweis: Wir zeigen, dass die zur Iterationsvorschrift (13) gehörige Funktion $\mathbf{\Phi}(\mathbf{x}) = D^{-1}(\mathbf{b} - E\mathbf{x})$ für beliebige $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ eine kontrahierende Abbildung in der Maximumsnorm ist. Laut Abschnitt 2.4 ist damit Konvergenz gewährleistet. Wir vereinfachen erst einmal

$$\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y}) = D^{-1}(\mathbf{b} - E\mathbf{x}) - D^{-1}(\mathbf{b} - E\mathbf{y}) = D^{-1}E(\mathbf{y} - \mathbf{x})$$

Die i-te Zeile der Matrix $D^{-1}E$ lautet

$$\frac{a_{i1}}{a_{ii}} \quad \frac{a_{i2}}{a_{ii}} \quad \ldots \quad \frac{a_{i,i-1}}{a_{ii}} \quad 0 \quad \frac{a_{i,i+1}}{a_{ii}} \quad \ldots \quad \frac{a_{in}}{a_{ii}}$$

Die Summe der Elementbeträge in dieser Zeile ist < 1 (Begründung: Aus der Summe $1/|a_{ii}|$ herausheben, Diagonaldominanz ausnützen). Damit ist auch für die Zeilensummen- oder Unendlich-Norm der Matrix $D^{-1}E$ gezeigt:

$$\|D^{-1}E\|_\infty < 1 \ .$$

Aus einer Eigenschaft der Matrixnorm, Ungleichung (9), folgt sofort die Kontraktionseigenschaft

$$\|\mathbf{\Phi}(\mathbf{x}) - \mathbf{\Phi}(\mathbf{y})\|_\infty = \|D^{-1}E(\mathbf{y} - \mathbf{x})\|_\infty \leq \|D^{-1}E\|_\infty \cdot \|\mathbf{y} - \mathbf{x}\|_\infty \leq C\|\mathbf{y} - \mathbf{x}\|_\infty$$

mit $C = \|D^{-1}E\|_\infty < 1$.

Mit beträchtlich mehr Aufwand lässt sich zeigen, dass auch für eine größere Klasse von Matrizen, nämlich *schwach diagonaldominante, irreduzible* Matrizen, das Jacobi-Verfahren konvergiert. Diese Aussage ist wichtig, weil viele Aufgaben aus der Praxis (numerische Lösung partieller Differentialgleichungen) genau solche Matrizen liefern. Der Vollständigkeit halber hier die Definitionen:
Eine $n \times n$-Matrix $A = [a_{ij}]$ ist *schwach diagonaldominant*, wenn

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \text{für} \quad i = 1, 2, \ldots, n$$

und zumindest für ein i echte Ungleichheit gilt. Um Irreduzibilität zu untersuchen, zeichnen Sie für jedes i einen Punkt. Für jedes Matrixelement $a_{ij} \neq 0$ in A verbinden Sie die Punkte i und j durch einen Pfeil in Richtung $i \to j$. Die Zeichnung stellt einen *gerichteten Graph* dar. Wenn man von jedem beliebigen Punkt zu jedem anderen gelangen kann, indem man den Pfeilen folgt, dann heißt dieser Graph *zusammenhängend* und die Matrix A ist *irreduzibel*.

In der Regel konvergiert das Gauß-Seidel-Verfahren rascher als das Jacobi-Verfahren. Es braucht für die gleiche Genauigkeit typischerweise nur halb so viele Iterationen. Das SOR-Verfahren mit optimal gewähltem Relaxationsparameter ω braucht größenordnungsmäßig $\sqrt{N}$ Iterationen, wo das Jacobi-Verfahren N Iterationen braucht.

Es gibt aber auch Matrizen, für die ein Verfahren konvergiert, das andere aber nicht.

Wir zitieren hier ohne Beweis zwei weitere Sätze, die Konvergenzbedingungen formulieren.

> Wenn A positive Elemente in der Hauptdiagonale hat und alle andern Elemente ≤ 0 sind, dann konvergiert das Gauß-Seidel-Verfahren genau dann, wenn das Jacobi-Verfahren konvergiert. Wenn beide Verfahren konvergieren, dann ist das Gauß-Seidel-Verfahren asymptotisch schneller (*Satz von Stein und Rosenberg*).
>
> Ist A symmetrisch positiv definit, dann konvergiert das Gauß-Seidel-Verfahren.

4.3 Moderne iterative Gleichungslöser

Gleichungssysteme aus dem Bereich der Strömungssimulation, der Festigkeitsanalyse, der Finanzmathematik und vieler weiterer Anwendungsgebiete erreichen leicht eine Größe von mehreren Millionen Unbekannten. Dafür sind aber pro Matrixzeile nur wenige Elemente von Null verschieden (So eine Matrix heißt *schwach besetzt*). Für die Lösung solcher Systeme werden heute fast ausschließlich iterative Verfahren eingesetzt. Die klassischen Methoden (Jacobi, Gauß-Seidel) konvergieren aber zu langsam und erfordern daher zuviel Rechenaufwand.

Diese Unterlagen können nur eine einführende Übersicht auf einige Prinzipien geben, die moderne iterative Gleichungslöser verwenden.

4.3.1 Splittings, Präkonditionierung

Angenommen, Sie sollen das System

$$A\mathbf{x} = \mathbf{b}$$

lösen.

Günstige Taktik: Sie ersetzen die Matrix A in dieser Aufgabe durch eine andere Matrix $\tilde{A}$, für die Sie Gleichungssysteme viel leichter lösen können. Sie können es sich dabei einfach machen und für $\tilde{A}$ die Einheitsmatrix I wählen, oder den diagonalen Anteil von A, oder gezielt nur bestimmte Matrixelemente aus A herausstreichen.

Schreiben Sie $A = \tilde{A} + E$. Eine solche Aufspaltung heißt ein *Splitting* von A in eine Näherung (auch: *Präkonditionierer*) und einen Restanteil E. Das ursprüngliche Gleichungssystem formulieren Sie dann als Fixpunkt-Aufgabe um.

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ (\tilde{A} + E)\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x} + E\mathbf{x} &= \mathbf{b} \\ \tilde{A}\mathbf{x} &= \mathbf{b} - E\mathbf{x} \\ \mathbf{x} &= \tilde{A}^{-1}(\mathbf{b} - E\mathbf{x}) \end{aligned}$$

Das Jacobi-Verfahren benützt diese Idee mit $\tilde{A} = D$, dem diagonalen Anteil. Das Gauß-Seidel-Verfahren lässt sich ebenfalls in dieser Form darstellen, indem man $\tilde{A}$ aus A durch Streichen sämtlicher Elemente oberhalb der Hauptdiagonale gewinnt.

Allgemein gilt, je besser $\tilde{A}$ die Original-Matrix approximiert, um so rascher konvergiert ein solches iterative Verfahren. Besonders gute Splittings entstehen aus *unvollständiger LR-Zerlegung*. Diese Methoden werden bei den Eliminationsverfahren in Abschnitt 3.7.4 kurz behandelt.

In der oben angegebenen Fixpunkt-Form wird das Verfahren aber nicht implementiert, da man die Matrix $\tilde{A}^{-1}$ nur in einfachsten Fällen (wie etwa $\tilde{A} = D$) explizit bilden sollte. Eine algebraisch gleichwertige, aber für Rechner geeignete Form lautet

Iterativer Gleichungslöser, Grundschema für $A = \tilde{A} + E$
Für ein geeignetes Splitting $A = \tilde{A} + E$, einen beliebigen Startvektor $\mathbf{x}^{(0)}$ und eine vorgegebene Genauigkeitsschranke $\epsilon > 0$ findet dieser Algorithmus eine Näherungslösung von $A\mathbf{x} = \mathbf{b}$.

Beginne mit Startvektor $\mathbf{x}^{(0)}$
setze $\mathbf{r}^{(0)} = \mathbf{b} - A\mathbf{x}^{(0)}$
iteriere für $k = 0, 1, 2, \ldots$
  löse $\tilde{A}\mathbf{d}^{(k+1)} = \mathbf{r}^{(k)}$
  setze $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{d}^{(k+1)}$
  setze $\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - A\mathbf{d}^{(k+1)}$
bis $\|\mathbf{r}^{(k+1)}\| < \epsilon$
Ergebnis: Näherungslösung $\mathbf{x}^{(k+1)}$

Bei Iterationen dieser Form nennt man $\tilde{A}$ auch die *Präkonditionierungsmatrix*.

Für einen Vektor $\mathbf{x}$ und gegebenes A und $\mathbf{b}$ bezeichnet man den Ausdruck $\mathbf{b} - A\mathbf{x}$ als *Residuum*. Die Aufgabe, ein Gleichungssystem $A\mathbf{x} = \mathbf{b}$ zu lösen, kann man also gleichwertig umformulieren in die Aufgabe, ein $\mathbf{x}$ mit verschwindendem Residuum zu finden. Man kann leicht nachprüfen, dass die Vektoren $\mathbf{r}^{(k)}$ im obigen Grundschema tatsächlich die jeweiligen Residuen sind: $\mathbf{r}^{(k)} = \mathbf{b} - A\mathbf{x}^{(k)}$. Das Abbruchkriterium des Verfahrens fordert also ein Residuum, das betragsmäßig kleiner als eine vorgegebene Schranke ist.

Vorsicht! Ein kleines Residuum bedeutet nicht automatisch, dass auch der Fehler $\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}$ zwischen exakter und genäherter Lösung klein ist. Es gilt beispielsweise, falls A symmetrisch ist, die Abschätzung

$$\frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\max}|} \leq \|\mathbf{x}_{\text{exc}} - \mathbf{x}^{(k)}\|_2 \leq \frac{\|\mathbf{r}^{(k)}\|_2}{|\lambda_{\min}|}$$

wobei $\lambda_{\max}$ und $\lambda_{\min}$ den betragsgrößten bzw. -kleinsten Eigenwert von A bezeichnen. Wenn also $\lambda_{\min}$ nahe Null liegt, sagt ein kleines Residuum noch nicht viel über die Größe des Fehlers aus.

Ebensowenig kann man aus der Kleinheit der Korrekturen $\mathbf{d}^{(k+1)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ unmittelbar auf die Kleinheit des Fehlers schließen. Moderne iterative Verfahren können allerdings praktischerweise Näherungen für die Eigenwerte mit geringem Zusatzaufwand mitrechnen und somit verlässliche Schranken für den Fehler liefern.

4.3.2 Konvergenzbeschleunigung durch Minimieren des Residuums

Häufig beobachtet man beim obigen Grundschema, dass sich die Vektoren $\mathbf{d}^{(k)}$, um die sich die Näherungsvektoren pro Iterationsschritt ändern, zwar in eine günstige Richtung zeigen, aber nicht die passende Länge haben. Anstatt den Näherungsvektor $\mathbf{x}^{(k)}$ pro Iterationsschritt nur um den Vektor $\mathbf{d}^{(k+1)}$ zu korrigieren, kann man daher versuchen, gleich ein Vielfaches ω dieser Korrektur anzubringen. (Eine ähnliche Idee verwendet auch schon das SOR-Verfahren.)

Wenn sich der Näherungsvektor beim Schritt von k nach $k+1$ um $\omega\mathbf{d}^{(k+1)}$ ändert, dann lässt sich leicht nachrechnen, dass sich der Restvektor um $-\omega A\mathbf{d}^{(k+1)}$ ändert. Man ändert also das Grundschema, setzt

$$\begin{aligned}\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} + \omega\mathbf{d}^{(k+1)}\\ \mathbf{r}^{(k+1)} &= \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}\end{aligned}$$

und wählt ω in jedem Schritt so, dass der Betrag $\|\mathbf{r}^{(k+1)}\|$ dadurch möglichst klein wird. Wie geht das? Mit der üblichen Vorgehensweise, wenn man einen Extremwert sucht: Differenzieren und Nullsetzen der Ableitung. (Es bedeutet hier$\|\cdot\|$ immer die 2-Norm, die euklidische Länge eines Vektors.)

Wir arbeiten der Einfachheit halber mit dem Betragsquadrat von $\mathbf{r}^{(k+1)}$. Es lässt sich als Funktion von ω schreiben:

$$\begin{aligned}\|\mathbf{r}^{(k+1)}\|^2 &= (\mathbf{r}^{(k+1)} \cdot \mathbf{r}^{(k+1)})\\ &= \Big((\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}) \cdot (\mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)})\Big)\\ &= \Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big)\end{aligned}$$

Die einzelnen inneren Produkte sind hier konstante skalare Größen. Differenzieren nach ω und Nullsetzen der Ableitung liefert

$$\begin{aligned}0 &= \frac{d}{d\omega}\|\mathbf{r}^{(k+1)}\|^2\\ &= \frac{d}{d\omega}\Big(\mathbf{r}^{(k)} \cdot \mathbf{r}^{(k)} - 2\omega(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + \omega^2(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)})\Big)\\ &= -2(\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}) + 2\omega(A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}) \quad \text{, daraus}\\ \omega &= \frac{\mathbf{r}^{(k)} \cdot A\mathbf{d}^{(k+1)}}{A\mathbf{d}^{(k+1)} \cdot A\mathbf{d}^{(k+1)}}\end{aligned}$$

4.3.3 Konvergenzbeschleunigung durch orthogonale Suchrichtungen

Wir setzen nun also

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \omega A\mathbf{d}^{(k+1)}$$

wobei ω so optimal gewählt ist, dass der Betrag von $\mathbf{r}^{(k+1)}$ minimal wird. Das heißt, jede weitere Korrektur des Residuums in Richtung $A\mathbf{d}^{(k+1)}$ kann nur eine Verschlechterung bringen. Falls im nächsten Iterationsschritt

$$\mathbf{r}^{(k+2)} = \mathbf{r}^{(k+1)} - \omega A\mathbf{d}^{(k+2)}$$

die Korrektur $A\mathbf{d}^{(k+2)}$ eine Komponente in Richtung $A\mathbf{d}^{(k+1)}$ enthält, tritt aber genau das ein.

Daher: Ist das Residuum entlang einer Richtung bereits minimiert, dann darf es entlang dieser Richtung nicht mehr korrigiert werden. Wir brauchen also ein Verfahren, das aus der Residuums-Korrektur $A\mathbf{d}^{(k+2)}$ die unerwünschte Komponente in Richtung $A\mathbf{d}^{(k+1)}$ herausnimmt. Das läßt sich durch *Orthogonalisierung* erreichen.

Seien $\mathbf{p}$ und $\mathbf{q}$ zwei Vektoren $\neq 0$. Die Komponente von $\mathbf{p}$ in Richtung von $\mathbf{q}$ ist gegeben durch

$$\left(\frac{\mathbf{p}\cdot\mathbf{q}}{\mathbf{q}\cdot\mathbf{q}}\right)\mathbf{q}$$

Der Vektor

$$\mathbf{p}-\left(\frac{\mathbf{p}\cdot\mathbf{q}}{\mathbf{q}\cdot\mathbf{q}}\right)\mathbf{q}$$

enthält also keine Komponente mehr in Richtung $\mathbf{q}$, steht also orthogonal auf $\mathbf{q}$. (Sonderfall: wenn $\mathbf{p}$ ein skalares Vielfaches von $\mathbf{q}$ ist, liefert diese Rechnung den Nullvektor.)

In dieser Weise lassen sich aus einem Vektor $\mathbf{p}$ auch sukzessive alle Komponenten in Bezug auf ein System von Vektoren herausnehmen.

Orthogonalisieren
Gegeben m von 0 verschiedene Vektoren $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ und ein Vektor $\mathbf{p} \in \mathbb{R}^n$. Dieser Algorithmus entfernt aus $\mathbf{p}$ alle Komponenten in Richtung $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_m$.

Für $i = 1 \ldots m$
  rechne inneres Produkt $r_i = \mathbf{p}\cdot\mathbf{q}_i / \mathbf{q}_i\cdot\mathbf{q}_i$
  subtrahiere Komponente: ersetze $\mathbf{p} = \mathbf{p} - r_i\mathbf{q}_i$

P. K. W. Vinsome, damals bei einer Erdölgesellschaft angestellt, veröffentlichte 1976 ein Verfahren, ORTHOMIN, welches alle diese Ideen (Präkonditionierun, Minimierung, Orthogonalisierung) verwendet. Inzwischen wurde eine Fülle von Verfahren entwickelt, die auf ähnlichen Prinzipien beruhen und heute alle als *Krylov-Unterraum-Verfahren* bezeichnet werden.

Für symmetrisch positiv definite Matrizen wurde aus diesen Ideen schon früher ein besonders elegantes und effizientes Verfahren entwickelt, die Methode der konjugierten Gradienten (Hestenes und Stiefel, 1952). Sie ist das Standardverfahren zur iterativen Lösung schwach besetzter, symmetrisch positiv definiter Systeme.

Für unsymmetrische Matrizen sind GMRES (für *generalized minimal residual method*), BiCG (für *biconjugate gradients*) oder CGS (für *conjugate gradient squared*) gängige Verfahren.