Technische Universität Ilmenau
Institut für Mathematik

Preprint No. M 02/06

Bandbreitenreduktion - Teil 1 - Grundlagen - Sparse Matrizen und ihre Verarbeitung

Neundorf, Werner

September 2002

Impressum:
Hrsg.: Leiter des Instituts für Mathematik
Weimarer Straße 25
98693 Ilmenau
Tel.: +49 3677 69 3621
Fax: +49 3677 69 3270
http://www.tu-ilmenau.de/ifm/

ISSN xxxx-xxxx

ilmedia

Technische Universität Ilmenau
Fakultät für Mathematik
und Naturwissenschaften
Institut für Mathematik

http://www.mathematik.tu-ilmenau.de/Math-Net/js_home_de.html

Postfach 10 05 65
D - 98684 Ilmenau
Germany
Tel.: 03677/69 3267
Fax: 03677/69 3272
Telex: 33 84 23 tuil d.
email: werner.neundorf@tu-ilmenau.de

Preprint No. M 06/02

Bandbreitenreduktion - Teil 1

Grundlagen

Sparse Matrizen und ihre Verarbeitung

Werner Neundorf

September 2002

‡MSC (2000): 65F50, 65F05, 65-01, 65-04, 65-05, 68Q25

Zusammenfassung

Gegenstand des dreiteiligen Preprints ist die Bandbreitenreduktion von Matrizen. Seine einzelnen Ausgaben basieren auf dem Vorlesungsskript "Wissenschaftliches Rechnen - Matrizen und LGS", gehalten als fakultative Veranstaltung am Institut für Mathematik der TU Ilmenau.
Teil 1 enthält die Grundlagen dazu, die insbesondere auf sparse und Bandmatrizen und deren Verarbeitung eingehen. In den Teilen 2 und 3 werden die Bandbreitenreduktion mit dem Algorithmus von Cuthill-McKee bzw. Gibbs-Poole-Stockmeyer ausführlich erläutert, verglichen und an Beispielen illustriert.

Vorwort

Viele Probleme benötigen die Handhabung von Matrizen bzw. die Lösung von linearen Gleichungssystemen. Ausgangspunkt dabei ist, dass man alle bzw. wichtige Informationen über die Matrix nutzt und diese auf ihre weitere Verarbeitung "vorbereitet".

Zu solchen Maßnahmen gehören:
- Feststellung von Eigenschaften der Matrix in Bezug auf Symmetrie, (strenge) Regularität, Definitheit, Diagonaldominanz, Orthogonalität u. a.,
- Erkennen und Anwendung der Besetztheitsstruktur,
- Bandbreiten- und Profilreduzierung,
- (symmetrische) Zeilen/Spaltenpermutation,
- Elementeabgleich,
- Zerlegungs- und Transformationstechniken.

Dabei liegen die Untersuchungen in folgenden Problemklassen.

(1) Skalierung als eine Form der Verbesserung der Kondition der Matrix.

(2) Faktorisierungsmethoden der Form $A = BC$, $A = BCD$ oder ähnlich unter Einbeziehung von Aspekten, die sie numerisch gutartig machen.
Damit ist natürlich formal unter zusätzlichen Bedingungen eine Transformation $C = B^{-1}A$ beschrieben.

(3) Transformationsmethoden der Form $A' = BAC$ möglichst mit Angabe der Transformationsmatrizen B und C, einschließlich der Betrachtung von Sonderfällen.
Ziel dabei ist es, dass die transformierte Matrix A' Eigenschaften besitzt, die ihre weitere Nutzung effizienter machen.

Hier soll der Schwerpunkt auf den Anstrich (3) gelegt werden.
Dazu werden Lösungsalgorithmen bzw. implementierte Routinen in der verschiedenen Programmiersprachen oder Computeralgebrasystemen angegeben.
Einige ergänzende grundlegende Abschnitte sowie zahlreiche Beispiele sollen insgesamt das Verständnis für die Problematik unterstützen.

Inhaltsverzeichnis

1 Sparse Matrizen **1**

1.1 Modellbeispiele 1

1.1.1 Zweipunktrandwertaufgabe - Stabdurchbiegung 1

1.1.2 Temperaturverlauf in einer dünnen quadratischen Platte . . . 5

1.1.3 Poisson-Gleichung auf einem schmalen Streifen 6

1.1.4 Weitere Matrizen 7

1.2 Sparse Matrizen 8

1.3 Band- und besondere Matrizen 29

2 Lösung von LGS und Komplexität der Algorithmen **32**

2.1 Gaußscher Algorithmus 34

2.2 Cholesky-Verfahren 40

2.3 Direkte Verfahren für Bandmatrizen 45

2.4 Tridiagonalmatrizen 49

2.5 Numerische und symbolische direkte Lösung von LGS mit CAS . . . 54

2.6 Komplexität und Geschwindigkeitstest 67

3 Speichern und Generieren von Matrizen **79**

3.1 Matrix-Vektor-Multiplikation für sparse Matrizen 81

3.1.1 Kompaktspeichertechniken 81

3.1.2 Generierung einer sparsen Matrix 85

3.1.3 Programmiertechnik 91

3.1.4 Testbeispiele 100

Literaturverzeichnis 109

Kapitel 1

Sparse Matrizen

1.1 Modellbeispiele

1.1.1 Zweipunktrandwertaufgabe - Stabdurchbiegung

Als Modellproblem wird eine einfache eindimensionale Zweipunktrandwertaufgabe mit inhomogenen Randbedingungen gewählt. Das zur numerischen Behandlung verwendete Diskretisierungsverfahren führt auf ein lineares Gleichungssystem (LGS), dessen Eigenschaften dargestellt werden. Des Weiteren werden für die Lösung des LGS ein Iterationsverfahren (IV) untersucht, insbesondere die Fragen der Konvergenz im Zusammenhang mit dem Spektrum der jeweiligen Iterationsmatrix sowie der Effizienz des Verfahrens.

- Zweipunktrandwertaufgabe mit inhomogenen Randbedingungen:

$$\begin{aligned} -U''(x) &= F(x), \quad x \in \Omega = (0,1) \subset \mathbb{R}, \\ U &= \varphi \;\text{ für }\; x \in \partial\Omega \quad \text{bzw.}\; U(0) = \varphi_0, \;\; U(1) = \varphi_1. \end{aligned} \tag{1.1}$$

- Gitter: $\overline{\Omega}_h = \{x \mid x = ih,\ i = 0(1)N,\ h = 1/N\}$, h Maschenweite.
- Gitterfunktion: $u_h = (u_1, u_2, ..., u_{N-1})^T$ mit $u_i \approx U_i = U(ih)$.
- Analog für rechte Seite: $f_h = (f_1, f_2, ..., f_{N-1})^T$ mit $f_i = F_i = F(ih)$, d. h. auf dem Gitter wird die rechte Seite exakt dargestellt.
- Approximation der Ableitungen (Operatoren) mittels Differenzenausdrücken:

$$U''(x_i) \approx \frac{1}{h^2}(U_{i+1} - 2U_i + U_{i-1}) \;\text{ zentraler Differenzenquotient 2. Ordnung.}$$

- Diskretisierte Aufgabe als LGS:

$$\begin{aligned} -\frac{1}{h^2}(u_{i+1} - 2u_i + u_{i-1}) &= f_i, \quad i = 1, 2, ..., N-1, \\ u_0 = \varphi_0, \quad u_N &= \varphi_1. \end{aligned}$$

- Matrixschreibweise des LGS:

$$A_h u_h \;=\; b_h \quad \text{bzw.} \quad Au \;=\; b \tag{1.2}$$

mit

$$A_h \;=\; \frac{1}{h^2}\begin{pmatrix} 2 & -1 & & \cdots & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 \\ \multicolumn{6}{c}{\dotfill} \\ 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}, \quad b_h \;=\; \begin{pmatrix} f_1 + \varphi_0/h^2 \\ f_2 \\ f_3 \\ \cdots \\ f_{N-2} \\ f_{N-1} + \varphi_1/h^2 \end{pmatrix},$$

$$A \;=\; \begin{pmatrix} 2 & -1 & & \cdots & 0 & 0 \\ -1 & 2 & -1 & \cdots & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 \\ \multicolumn{6}{c}{\dotfill} \\ 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{pmatrix}, \quad b \;=\; h^2\begin{pmatrix} f_1 + \varphi_0/h^2 \\ f_2 \\ f_3 \\ \cdots \\ f_{N-2} \\ f_{N-1} + \varphi_1/h^2 \end{pmatrix},$$

$$A \;=\; \text{tridiag}(-1, 2, -1).$$

Die Koeffizientenmatrix $A(n,n)$ ist schwach besetzt. Sie hat etwa $3n$ nichtverschwindende Elemente ($n = N-1$) an Stelle von n^2 Elementen bei voll besetzten Matrizen. Sie ist eine Tridiagonalmatrix, d. h. ihre Bandbreite ist 3. Dazu ist sie symmetrisch und positiv definit (spd).
Das IV für die Lösung von $Au = b$ notieren in der Basisversionen gemäß [2].

$$u^{(m+1)} = Hu^{(m)} + c = (I - W^{-1}A)u^{(m)} + W^{-1}b = u^{(m)} + W^{-1}r^{(m)}, \tag{1.3}$$

wobei

$u^{(0)}$	Startvektor,
$r^{(m)} = b - Au^{(m)}$	Residuum, manchmal auch $r^{(m)} = Au^{(m)} - b$,
$Au^{(m)} - b$	Defekt,
W	Wichtung, Vorkonditionierungsmatrix,
$W^{-1}(Au^{(m)} - b)$	Korrekturvektor,
$H = I - W^{-1}A$	Iterationsmatrix bedeuten.

Konvergenz des IV

Konvergenzsätze liefern hinreichende und notwendige Konvergenzbedingungen für das IV. Dazu benötigen wir die Eigenwerte (EW) der Iterationsmatrix H oder Näherungen dieser.
Hier ist die Konvergenz im Fall des Gesamtschrittverfahrens (Jacobi-Verfahren, GSV) mit $W = D = \text{diag}(A)$ gesichert durch die Aussage, dass die Koeffizientenmatrix eine irreduzibel diagonaldominante Matrix darstellt.

Die reelle Matrix $A = A(n,n)$ heißt irreduzibel diagonaldominant, wenn

(1) $|a_{ii}| \;\geq\; \sum_{\substack{j=1\\ j\neq i}}^{n} |a_{ij}|, \quad i = 1,2,...,n,$

und für mindestens ein i gilt die strenge Größerbeziehung,

(2) und es gibt keine Permutationsmatrix P, mit der eine Transformation der Matrix gemäß

$$\widetilde{A} \;=\; PAP^T \;=\; \begin{pmatrix} \widetilde{A}_{11} & \widetilde{A}_{12} \\ 0 & \widetilde{A}_{22} \end{pmatrix}, \quad \widetilde{A}_{11} \in \mathbb{R}^{p,p},\; \widetilde{A}_{22} \in \mathbb{R}^{q,q},\; p+q=n,$$

möglich ist (falls es eine solche Transformation gibt, ist die Matrix reduzibel).

Dann gelten für A auch die folgenden Aussagen:

(1) Determinante $\det(A) \neq 0$,

(2) $a_{ii} \neq 0$ für alle $i = 1,2,...,n$.

Die Iterationsmatrix $H = J = I - D^{-1}A = I - \frac{1}{2}A$ hat die Eigenwerte

$$\mu_i = \mu_i(H) = 1 - \frac{1}{2}\lambda_i \in (-1,1),\; \mu_i = -\mu_{n-i},$$

wobei

$$\lambda_i = \lambda_i(A) = 2[1 - \cos(i\pi/(n+1))] = 4\sin^2(i\pi/(2(n+1))), \quad i = 1,2,...,n,$$

die EW von A sind.
Es gilt wegen

$$\begin{aligned} \lambda_1 &= \lambda_{min} = 4\sin^2(\pi h/2), \\ \lambda_n &= \lambda_{max} = 4 - \lambda_{min} = 4\cos^2(\pi h/2) \end{aligned}$$

für den Spektralradius

$$\rho(H) = \max|\mu(H)| = \mu_1 = 1 - \frac{1}{2}\lambda_{min} = \left|1 - \frac{1}{2}\lambda_{max}\right| = 1 - 2\sin^2\left(\frac{\pi h}{2}\right) = \cos(\pi h),$$

also näherungsweise $\rho(H) \approx 1 - \pi^2 h^2/2 < 1$.

Der Spektralradius liegt aber für kleine Schrittweiten h nahe der Eins, so dass die Konvergenzgeschwindigkeit bzw. Konvergenzrate des IV klein ist.

Als direktes Verfahren zur Lösung von $Au = b$ kann man die Gauß-Elimination mit gleichzeitiger LU-Faktorisierung verwenden, die wegen der speziellen Matrixform in verkürzter vektorisierter Variante ohne Pivotstrategie durchführbar ist. Leider ist die Kondition der Matrix A sehr schlecht, d. h. es gilt mit der Spektralnorm (Hilbert-Norm)

$$
\begin{aligned}
\|A\|_2 &= \sqrt{\max_{i=1(1)n} \mu_i}, \quad 0 \le \mu_i \in \sigma(A^T A), \\
&= \sqrt{\rho(A^T A)}, \\
&\qquad \sigma(A^T A) = \{\mu_i(A^T A),\ i = 1, 2, ..., n\} \ \text{ Spektrum}, \\
&\qquad \rho(A^T A) = \max_{i=1(1)n} |\mu_i(A^T A)| \ \text{ Spektralradius},
\end{aligned}
$$

wegen $A = A^T$ die Beziehung

$$
\begin{aligned}
\kappa(A) &= \operatorname{cond}_2(A) = \|A\|_2 \, \|A^{-1}\|_2 \\
&= \frac{\lambda_{max}}{\lambda_{min}} = \frac{4 - \lambda_{min}}{\lambda_{min}} = \frac{4}{\lambda_{min}} - 1 \\
&\approx \frac{4}{4 \sin^2(\pi h/2)} \approx \frac{1}{(\pi h/2)^2} \\
&\approx \frac{4n^2}{\pi^2} \gg 1 \ \text{ für } \ n \gg 1.
\end{aligned}
$$

Um eine akzeptable Näherungslösung u zur exakten u^* zu erhalten, ist eine starke Gleitpunktarithmetik notwendig.
Die Fehlerakkumulation beim Gauß-Algorithmus (GA) mit $A(n,n)$ und t Dualstellen der Mantisse des Gleitpunktformats führt zum absoluten Fehler

$$
\|\delta u\| = \|u^* - u\| = 2^{-t} \operatorname{cond}(A)\ K(n),
$$

wobei

$$
K(n) = \begin{cases} \mathcal{O}(n) & \text{für GA ohne Pivotisierung, } A \text{ diag. dominant oder } A = A^T > 0, \\ \mathcal{O}(2^n) & \text{für GA mit Spaltenpivotisierung,} \\ \mathcal{O}(n^{3/2}) & \text{für GA mit vollständiger Pivotisierung.} \end{cases}
$$

Für $t = 64$ (*extended*-Format), $t = 53$ (*double*-Format) oder $t = 40$ (*real*-Format) kann man die gültigen Dezimalstellen der Näherungslösung ermitteln.

1.1.2 Temperaturverlauf in einer dünnen quadratischen Platte

Gegeben sei die partielle Differentialgleichung für eine Funktion $u(x,y)$ auf dem Einheitsquadrat, die den Temperaturverlauf in einer dünnen Platte beschreibt.

$$-\Delta U(x,y) = -\left(\frac{\partial U}{\partial x^2} + \frac{\partial U}{\partial y^2}\right) = Q(x,y), \quad (x,y) \in \Omega = (0,1)^2. \tag{1.4}$$

Auf dem Rand des Gebietes sei $U(x,y)$ gleich Null.
Das ist eine elliptische Randwertaufgabe bzw. die Poisson-Gleichung.

Der Diskretisierungsparameter bzw. die Maschenweite des quadratischen Gitters sei $h = 1/(n+1)$. Man diskretisiert die partiellen Ableitungen mittels zentraler Differenzenquotienten 2. Ordnung

$$\Delta U(x_i,y_j) \approx \frac{1}{h^2}(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{ij})$$

und notiert die Differenzenformel (Differenzenstern) für alle inneren (zweidimensionalen) Knoten (x_i,y_j), $i,j = 1,2,...,n$, in linearer Reihenfolge zeilenweise gemäß $(j-1)n+i$.
Die diskretisierte RWA schreibt man als LGS $Au = h^2q$.
Welche Struktur und Eigenschaften hat die Matrix A? Wie groß ist ihre Bandbreite?
A besitzt die folgende Blockstruktur.

$$A = \begin{pmatrix} B & -I & & & \\ -I & B & -I & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -I \\ & & & -I & B \end{pmatrix}$$

mit der $(n \times n)$-Matrix

$$B = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & -1 \\ & & & -1 & 4 \end{pmatrix}$$

und der $(n \times n)$-Einheitsmatrix I. A ist eine dünn besetzte symmetrische Matrix mit Bandstruktur. Die Bandbreite beträgt $2n+1$. Die Matrix ist irreduzibel diagonaldominant. EW und Eigenvektoren (EV) von A lassen sich aus dem eindimensionalen Fall einfach herleiten.

1.1.3 Poisson-Gleichung auf einem schmalen Streifen

Gegeben sei die partielle Differentialgleichung (Poisson-Gleichung) für eine Funktion $U(x,y)$ auf einem Rechteckgebiet.

$$-\Delta U(x,y) = -\left(\frac{\partial U}{\partial x^2} + \frac{\partial U}{\partial y^2}\right) = Q(x,y), \quad (x,y) \in \Omega = (a,b) \times (c,d). \qquad (1.5)$$

Auf dem Rand des Gebietes sei die Funktion $U(x,y)$ vorgegeben (Dirichletsche Randbedingungen).
Zur Lösung verwenden wir die finite Differenzenmethode auf einem (11×2)-Gitter (x_i, y_j) mit der Maschenweite h. Wir führen die Nummerierung der Gitterpunkte im rechteckigen Gebiet zeilenweise durch.

o1	o2	o3	o4	o5	o6	o7	o8	o9	o10	o11
o12	o13	o14	o15	o16	o17	o18	o19	o20	o21	o22

So erhalten wir eine Matrix mit der Blockstruktur

$$A = \begin{pmatrix} B & -I \\ -I & B \end{pmatrix}$$

mit (11×11)-Matrizen B und I, welche wie in Kap. 1.1.2 definiert sind. Sie hat die Bandbreite 23.

Jetzt nehmen wir eine Gebietszerlegung vor. Folgende Zerlegung von Ω in Gebiete D_i und "innere Ränder" S_i sei gegeben.

D_1			S_1	D_2			S_2	D_3		
o1	o2	o3	×19	o7	o8	o9	×21	o13	o14	o15
o4	o5	o6	×20	o10	o11	o12	×22	o16	o17	o18

Man diskretisiert die partiellen Ableitungen wiederum mittels zentraler Differenzenquotienten und notiert die Differenzenformel für alle inneren 22 Knoten (x_i, y_j), $i = 1, 2, ..., 12$, $j = 1, 2$, in der Reihenfolge 1,2,...,18,19,...,22.
Hierbei beschreiben die inneren Teilgebiete S_i Punkte, welche bei der Diskretisierung die beiden benachbarten breiten Teilgebiete beeinflussen. Sie werden in der Nummerierung als letzte berücksichtigt.

Dies führt auf die folgende Matrixstruktur

$$A = \begin{pmatrix} B & & & C_1 \\ & B & & C_2 \\ & & B & C_3 \\ C_1^T & C_2^T & C_3^T & D \end{pmatrix}$$

mit der (6×6)-Matrix B den (6×4)-Matrizen C_i und der (4×4)-Matrix D. Natürlich hat auch hier die Matrix A eine sparse Struktur. Dazu kommt die spezielle Blockverteilung der Gestalt eines "Pfeils nach unten"

$$\begin{pmatrix} * & & & * \\ & * & & * \\ & & * & * \\ * & * & * & * \end{pmatrix},$$

die für die Anwendung eines Eliminationsverfahrens und den dabei neu entstehenden Elementen an den bisherigen "Nullstellen" eine günstige Situation darstellt. So kann das sogenannte **Fill-in**, d. h. der Zuwachs der Anzahl der Nichtnullelemente, bei Durchführung der Gauß-Elimination gering gehalten werden.

1.1.4 Weitere Matrizen

Wir notieren eine sparse (5×5)-Matrix mit ihren Nichtnullelementen (NNE)

$$A_1 = (a_{ij}) = \begin{pmatrix} 2 & & 1 & & \\ & 3 & & & 1 \\ 2 & & & 1 & \\ & & 1 & 2 & \\ & & 1 & 2 & 1 \end{pmatrix}.$$

Solche Matrizen entstehen z. B. bei Verflechtungsmodellen in der Ökonomie, bei der Simulation elektrischer Netzwerke oder mit mehr Struktur in der Matrix bei der Lösung von partiellen Differentialgleichungen mittels der Methode der finiten Elemente.

Eine sparse obere Hessenberg-Matrix mit ihren NNE hat die Besetzungsstruktur

$$A_2 = (a_{ij}) = \begin{pmatrix} * & * & \cdots & * & * \\ * & * & \cdots & * & * \\ 0 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & * & * \end{pmatrix}.$$

1.2 Sparse Matrizen

Charakteristisch für eine Reihe von LGS ist, dass die Koeffizientenmatrizen schwach besetzt (dünn, spärlich besetzt, engl. sparse) sind. Wenn die Ordnung (Dimension) der Matrix n beträgt, so ist die Anzahl der NNE der Matrix vergleichsweise gering. Sie beträgt meist nur wenige Prozent im Vergleich zur Gesamtzahl n^2 und erweist sich manchmal als unabhängig von dieser. Eine Rolle spielt dabei auch die Maximalanzahl k der NNE je Zeile oder Spalte. Diese variiert bei Diskretisierungsverfahren für ein- und zweidimensionale Randwertprobleme in typischen Fällen zwischen 3 und 20.

Im Computeralgebrasystem (CAS) MATLAB gibt es Befehle zur Generierung und Verarbeitung sparser Matrizen.

```
Purpose
 Create sparse matrices

Synopsis
 S = sparse(i,j,s,m,n,nzmax)
 S = sparse(i,j,s,m,n)
 S = sparse(i,j,s)
 S = sparse(m,n)
 S = sparse(A)

Description
 S = sparse(...) is the built-in function which generates matrices in
 MATLAB's sparse storage organization. It can be called with 1,2,3,5
 or 6 arguments.

 S = sparse(A) converts a full matrix to sparse form by squeezing out
 any zero elements. If S is already sparse, sparse(S) returns S.

 S = sparse(i,j,s,m,n,nzmax) uses the rows of [i,j,s] to generate an
 m-by-n sparse matrix with space allocated for nzmax nonzeros. The two
 integer index vectors i and j have positive integer elements, while
 the vector s may have real or complex entries.
 Any elements of s which are zero are removed, along with the
 corresponding elements of i and j. Any elements of s which have
 duplicate values of i and j are added together. So all have the same
 length nnz. The length of the  resulting modified s is the number of
 nonzeros in the resulting sparse matrix S.
 There are several simplifications of this six argument call.
 The argument s and one of the arguments i or j may be scalars,
 in which case they are expanded so that the first three arguments
 all have the same length.
```

```
 S = sparse(i,j,s,m,n) uses nzmax = length(s).

 S = sparse(i,j,s) uses m = max(i) and n = max(j). The maxima are
 computed before any zeros in s are removed, so one of the rows of
 [i j s] might be [m n 0].

 S = sparse(m,n) abbreviates sparse([],[],[],m,n,0).  This
 generates the ultimate sparse matrix, an m-by-n all zero matrix.

 All MATLAB's built-in arithmetic, logical, and indexing operations
 can be applied to sparse matrices, or to mixture of sparse and full
 matrices. Operations on sparse matrices return sparse matrices and
 operations on full matrices return full matrices.
 The most cases, operations on mixtures of sparse and full matrices
 return full matrices. The exceptions include situations where the
 result of a mixed operation is structurally sparse, for example,
 A.*S is at least as sparse as S.
 Some operations, such as S>=0, generate "Big Sparse", or
 "BS" matrices - matrices with sparse storage organization
 but few zero elements.

Examples
 S = sparse(1:n,1:n,1) generates a sparse representation of the
 n-by-n identity matrix. The same S results from S = sparse(eye(n,n)),
 but this would also temporarily generate a full n-by-n matrix with
 most of its elements equal to zero.
 B = sparse(10000,10000,pi) is probably not very useful, but is legal
 an works. It set up a 10000-by-10000 matrix with only one nonzero
 element. Don't try full(B); it requires 800 Mbyte of storage.

 This dissects and then reassembles a sparse matrix:
              [i,j,s] = find(S);
              [m,n] = size(S);
              S = sparse(i,j,s,m,n);

 So does this, if the last row and column have nonzero entries:
              [i,j,s] = find(S);
              S = sparse(i,j,s);

See also
 full, find, spy, nonzeros, nnz, nzmax, diags, spones, spalloc,
 spparms, sprand, sprandsym, speye, spconvert, issparse, sprank,
 sparfun (directory)
```

```
Random sparse matrices

sprandn  Sparse normally distributed random matrix.
    R = sprandn(S) has the same sparsity structure as S, but normally
        distributed random entries.

    R = sprandn(m,n,density) is a random, m-by-n, sparse matrix with
        approximately density*m*n normally distributed nonzero entries.
        sprandn is designed to produce large matrices with small density
        and will generate significantly fewer nonzeros than requested if
        m*n is small or density is large.

    R = sprandn(m,n,density,rc) also has reciprocal condition number
        approximately equal to rc. R is constructed from a sum of
        matrices of rank one.

        If rc is a vector of length lr <= min(m,n), then R has
        rc as its first lr singular values, all others are zero.
        In this case, R is generated by random plane rotations
        applied to a diagonal matrix with the given singular values
        It has a great deal of topological and algebraic structure.


sprandsym  Sparse random symmetric matrix.
    R = sprandsym(S) is a symmetric random matrix whose lower triangle
        and diagonal have the same structure as S. The elements are
        normally distributed, with mean 0 and variance 1.

    R = sprandsym(n,density) is a symmetric random, n-by-n, sparse
        matrix with approximately density*n*n nonzeros; each entry is
        the sum of one or more normally distributed random samples.

    R = sprandsym(n,density,rc) also has a reciprocal condition number
        equal to rc. The distribution of entries is nonuniform;
        it is roughly symmetric about 0; all are in [-1,1].

        If rc is a vector of length n, then R has eigenvalues rc.
        Thus, if rc is a positive (nonnegative) vector then R will
        be positive (nonnegative) definite. In either case, R is
        generated by random Jacobi rotations applied to a diagonal
        matrix with the given eigenvalues or condition number. It has
        a great deal of topological and algebraic structure.
```

```
    R = sprandsym(n, density, rc, kind) is positive definite.

        If kind = 1, R is generated by random Jacobi rotation of
           a positive definite diagonal matrix.
           R has the desired condition number exactly.
        If kind = 2, R is a shifted sum of outer products.
           R has the desired condition number only
           approximately, but has less structure.

    R = sprandsym(S,[],rc,3) has the same structure as the MATRIX S
        and approximate condition number 1/rc.
```

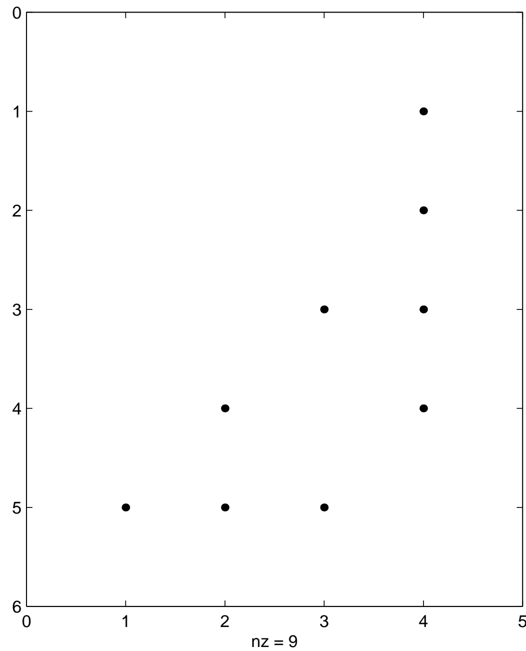
Einige dieser MATLAB-Kommandos sollen nachfolgend angewendet werden.

Beispiel 1.1

Wir generieren Rechteckmatrizen bzw. symmetrische Matrizen mit einer gegebenen Belegungsdichte.

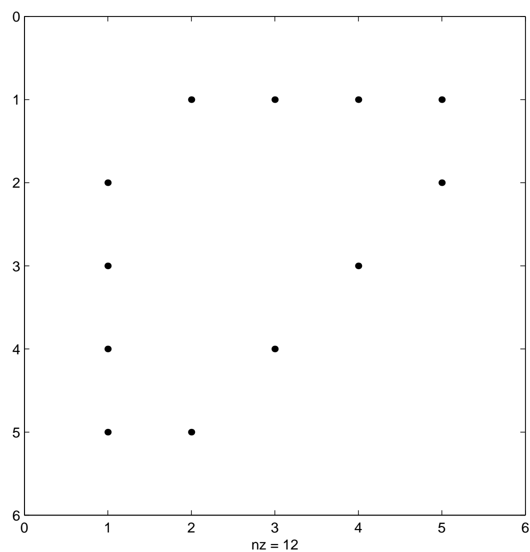
```
>>% a random, m-by-n, sparse matrix with approximately density*m*n
>>% normally distributed nonzero entries
>>R = sprandn(5,4,0.5)
R =
   (5,1)       1.1892
   (4,2)      -1.1465
   (5,2)      -0.0376
   (3,3)       0.1253
   (5,3)       0.3273
   (1,4)      -0.4326
   (2,4)      -1.6656
   (3,4)       0.2877
   (4,4)       1.1909

>>spy(R)
>>print bild09.ps -dps

>>nnz(R)
ans =
     9

>>[i,j,s] = find(R);
>>i'
ans =
     5     4     5     3     5     1     2     3     4
```

```
>>j'
ans =
     1     2     2     3     3     4     4     4     4
>>s'
ans =
    1.1892   -1.1465   -0.0376    0.1253    0.3273   -0.4326   -1.6656
    0.2877    1.1909
```




```
>>% a symmetric random, n-by-n, sparse matrix with approximately
>>% density*n*n nonzeros
>>S = sprandsym(5,0.5)
S =
   (2,1)      -0.0121
   (3,1)       2.2971
   (4,1)      -0.5883
   (5,1)       0.7258
   (1,2)      -0.0121
   (5,2)      -0.1364
   (1,3)       2.2971
   (4,3)       1.0668
   (1,4)      -0.5883
   (3,4)       1.0668
   (1,5)       0.7258
   (2,5)      -0.1364
```

```
>>spy(S)
>>print bild10.ps -dps
>>[i,j,s] = find(S);
>>i'
ans =
     2     3     4     5     1     5     1     4     1     3     1     2

>>j'
ans =
     1     1     1     1     2     2     3     3     4     4     5     5

>>s'
ans =
   -0.0121    2.2971   -0.5883    0.7258   -0.0121   -0.1364    2.2971
    1.0668   -0.5883    1.0668    0.7258   -0.1364

>>nnz(S)
ans =
    12
```



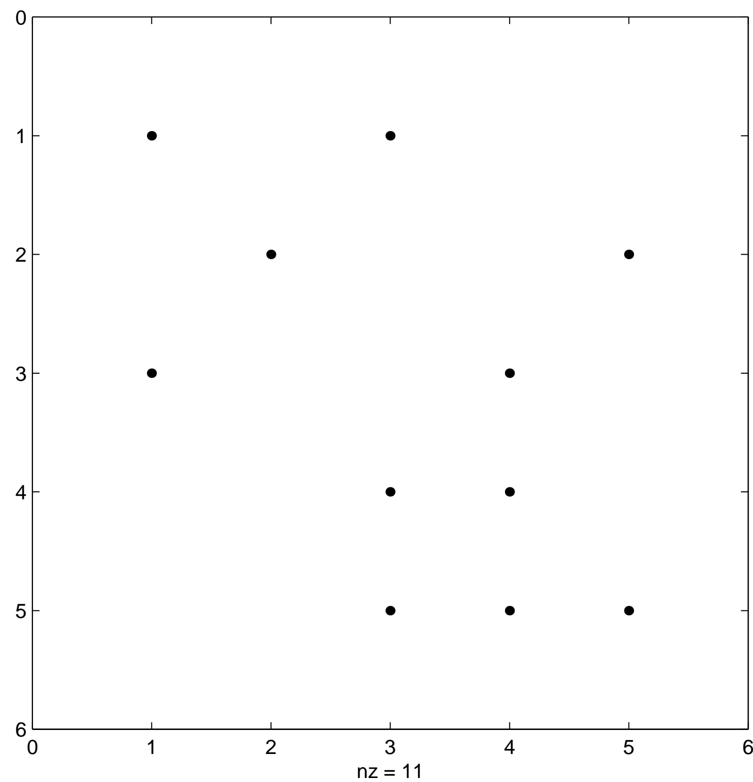
Beispiel 1.2

$$A = (a_{ij}) = \begin{pmatrix} 2 & & 1 & & \\ & 3 & & & 1 \\ 2 & & & 1 & \\ & & 1 & 2 & \\ & & 1 & 2 & 1 \end{pmatrix} \neq A^T$$

```
>>A = [2  0  1  0  0
       0  3  0  0  1
       2  0  0  1  0
       0  0  1  2  0
       0  0  1  2  1]

A =
     2     0     1     0     0
     0     3     0     0     1
     2     0     0     1     0
     0     0     1     2     0
     0     0     1     2     1

>>S = sparse(A)

S =
   (1,1)        2
   (3,1)        2
   (2,2)        3
   (1,3)        1
   (4,3)        1
   (5,3)        1
   (3,4)        1
   (4,4)        2
   (5,4)        2
   (2,5)        1
   (5,5)        1

>>% das gleiche Ergebnis liefert

>>S = sparse([1,3,2,1,4,5,3,4,5,2,5],...
             [1,1,2,3,3,3,4,4,4,5,5],...
             [2,2,3,1,1,1,1,2,2,1,1],5,5,11)
```

```
>>% Check for sparse matrix storage class
>>issparse(A)
ans =
     0
>>issparse(S)
ans =
     1

>>% Number of nonzero entries
>>nnz(S)
ans =
     11

>>% The nonzero entries in a matrix
>>nonzeros(S)'
ans =
     2     2     3     1     1     1     1     2     2     1     1

>>% Find indices and values of nonzero elements
>>find(S)'  % laufende Nummerierung mit allen Spalten nacheinander
ans =
     1     3     7    11    14    15    18    19    20    22    25

>>[i,j] = find(S);
>>i'
ans =
     1     3     2     1     4     5     3     4     5     2     5
>>j'
ans =
     1     1     2     3     3     3     4     4     4     5     5

>>% Bandbreite der Matrix
>>max(i-j)  % Anzahl der unteren Nebendiagonalen
ans =
     2

>>min(i-j)  % -Anzahl der oberen Nebendiagonalen
ans =
    -3
>>bw1 = max(i-j)-min(i-j)+1
bw1 =
     6
```

```
>>% Grafische Belegungsstruktur der Matrix
>>% Visualize matrix sparsity pattern

>>spy(S)
>>print bild01.ps -dps
```



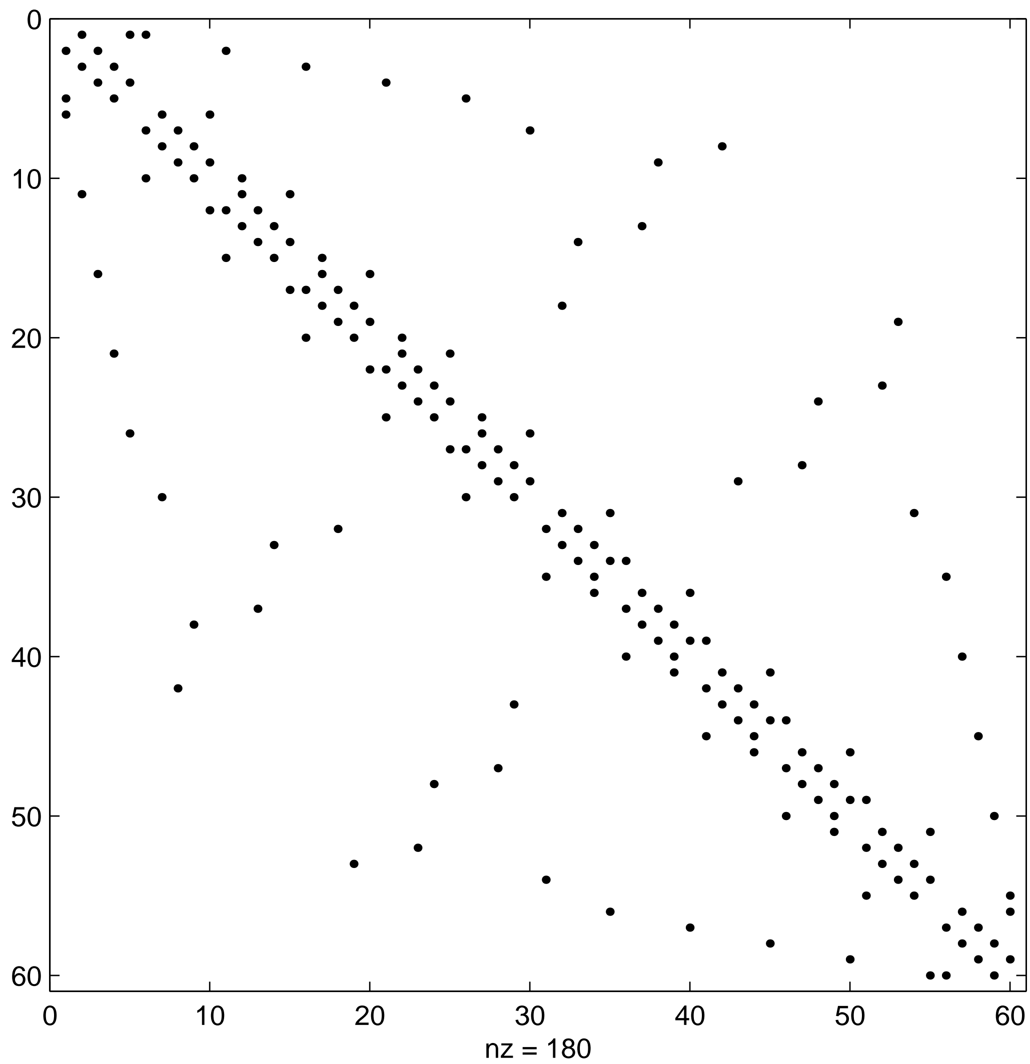

```
>>% Repleace nonzero elements with ones
>>R = spones(S)

R =
   (1,1)  1   (3,1)  1   (2,2)  1   (1,3)  1   (4,3)  1   (5,3)  1
   (3,4)  1   (4,4)  1   (5,4)  1   (2,5)  1   (5,5)  1
```

Beispiel 1.3

$A_2 = (a_{ij})$ ist eine symmetrische (60×60)-Matrix, die in dem m-File `bucky` der Demo-Toolbox von MATLAB enthalten ist. Sie basiert auf dem Adjazenzgraphen eines abgeschnittenen Ikosaeders.

```
>>B = bucky

B =
    (2,1) 1     (5,1) 1     (6,1) 1     (1,2) 1     (3,2) 1    (11,2) 1
    (2,3) 1     (4,3) 1    (16,3) 1     (3,4) 1     (5,4) 1    (21,4) 1
    (1,5) 1     (4,5) 1    (26,5) 1     (1,6) 1     (7,6) 1    (10,6) 1
    (6,7) 1     (8,7) 1    (30,7) 1     (7,8) 1     (9,8) 1    (42,8) 1
    (8,9) 1    (10,9) 1    (38,9) 1    (6,10) 1    (9,10) 1   (12,10) 1
   (2,11) 1   (12,11) 1   (15,11) 1   (10,12) 1   (11,12) 1   (13,12) 1
  (12,13) 1   (14,13) 1   (37,13) 1   (13,14) 1   (15,14) 1   (33,14) 1
  (11,15) 1   (14,15) 1   (17,15) 1    (3,16) 1   (17,16) 1   (20,16) 1
  (15,17) 1   (16,17) 1   (18,17) 1   (17,18) 1   (19,18) 1   (32,18) 1
  (18,19) 1   (20,19) 1   (53,19) 1   (16,20) 1   (19,20) 1   (22,20) 1
   (4,21) 1   (22,21) 1   (25,21) 1   (20,22) 1   (21,22) 1   (23,22) 1
  (22,23) 1   (24,23) 1   (52,23) 1   (23,24) 1   (25,24) 1   (48,24) 1
  (21,25) 1   (24,25) 1   (27,25) 1    (5,26) 1   (27,26) 1   (30,26) 1
  (25,27) 1   (26,27) 1   (28,27) 1   (27,28) 1   (29,28) 1   (47,28) 1
  (28,29) 1   (30,29) 1   (43,29) 1    (7,30) 1   (26,30) 1   (29,30) 1
  (32,31) 1   (35,31) 1   (54,31) 1   (18,32) 1   (31,32) 1   (33,32) 1
  (14,33) 1   (32,33) 1   (34,33) 1   (33,34) 1   (35,34) 1   (36,34) 1
  (31,35) 1   (34,35) 1   (56,35) 1   (34,36) 1   (37,36) 1   (40,36) 1
  (13,37) 1   (36,37) 1   (38,37) 1    (9,38) 1   (37,38) 1   (39,38) 1
  (38,39) 1   (40,39) 1   (41,39) 1   (36,40) 1   (39,40) 1   (57,40) 1
  (39,41) 1   (42,41) 1   (45,41) 1    (8,42) 1   (41,42) 1   (43,42) 1
  (29,43) 1   (42,43) 1   (44,43) 1   (43,44) 1   (45,44) 1   (46,44) 1
  (41,45) 1   (44,45) 1   (58,45) 1   (44,46) 1   (47,46) 1   (50,46) 1
  (28,47) 1   (46,47) 1   (48,47) 1   (24,48) 1   (47,48) 1   (49,48) 1
  (48,49) 1   (50,49) 1   (51,49) 1   (46,50) 1   (49,50) 1   (59,50) 1
  (49,51) 1   (52,51) 1   (55,51) 1   (23,52) 1   (51,52) 1   (53,52) 1
  (19,53) 1   (52,53) 1   (54,53) 1   (31,54) 1   (53,54) 1   (55,54) 1
  (51,55) 1   (54,55) 1   (60,55) 1   (35,56) 1   (57,56) 1   (60,56) 1
  (40,57) 1   (56,57) 1   (58,57) 1   (45,58) 1   (57,58) 1   (59,58) 1
  (50,59) 1   (58,59) 1   (60,59) 1   (55,60) 1   (56,60) 1   (59,60) 1

>>% Number of nonzero entries
>>nnz(B)
ans =
     180
```

```
>>[i,j] = find(B);
>>hbw2 = max(i-j)+1    % halbe Bandbreite
hbw2 =
    35

>>bw2 = max(i-j)-min(i-j)+1    % Bandbreite
bw2 =
    69

>>spy(B)   % Belegungsstruktur
>>print bild02.ps -dps
```



Es gibt verschiedene Möglichkeiten, die sparse Struktur einer Matrix auszunutzen. Natürlich kann man einen Test einer Größe auf Null machen, wenn dieser im Rechner schnell ausführbar ist und damit weitere Oprationen eingespart werden können. Günstiger sind zwei andere Varianten.

(1) Kompaktspeichertechnik

Ziel der kompakten Speicherung sparser Matrizen ist es, Speicherplätze einzusparen und Rechenoperationen mit Nullelementen (NE) zu vermeiden. Sie erfordert beim GA eine relativ aufwendige Anpassung und ist deshalb besonders dann empfehlenswert, wenn viele Systeme mit sparsen Koeffizientenmatrizen derselben Besetztheitsstruktur gelöst werden müssen.
Man kann natürlich die NNE nicht einfach zu einem Vektor zusammenschieben, sondern man muss sich in zusätzlichen Indexvektoren die Informationen zu den zwei Koordinaten der NNE merken.
Für die sparse reelle Matrix

$$A = A(n,n) = (a_{ij}) = \begin{pmatrix} 2 & & 1 & & \\ & 3 & & & 1 \\ 2 & & & 1 & \\ & & 1 & 2 & \\ & & 1 & 2 & 1 \end{pmatrix}$$

mit $n = 5$ und $nne = 11$ NNE wäre folgende Datenstruktur denkbar.
Man braucht 1 *real*-Vektor für die NNE und 2 *integer*-Vektoren für die Positionen. Die NNE werden Zeile für Zeile in den *real*-Vektor $A[1..nne+1]$ geschrieben. Im *integer*-Vektor $J[1..nne+1]$ werden die zugehörigen Spaltenindizes gemerkt. Die i-te Komponente des *integer*-Vektors $I[1..n+1]$ enthält die Position des ersten NNE der i-ten Zeile in den Vektoren A bzw. J. Der Vektor I arbeitet wie ein Zähler, denn die Größe $I_{i+1} - I_i$, $I_1 = 1$, definiert die Anzahl der NNE in der i-ten Zeile. Somit ist $I_{n+1} = nne + 1$. I ist wie ein Zeilenzeiger.

	1	2	3	4	5	$6 = n+1$	
I	1	3	5	7	9	12	$= nne + 1 \leq n^2 + 1$

	1	2	3	4	5	6	7	8	9	10	11	$12 = nne + 1$
J	1	3	2	5	1	4	3	4	3	4	5	$\cdot$
A	2	1	3	1	2	1	1	2	1	2	1	$\cdot$

$A_1 = a_{11} = 2$ ...
$A_5 = a_{31} = 2$ ...

Natürlich gibt es auch andere Möglichkeiten, so z. B. die Speicherung der Tripel (i, j, a_{ij}), wie sie beim MATLAB-Kommando `sparse` auftreten.
Dabei enstehen durch `[i,j]=find(..)` bzw. `nonzeros(...)` die 3 Vektoren $I[1..nne]$, $J[1..nne]$ und $A[1..nne]$ jeweils für die $nne = 11$ Indizes bzw. Werte der NNE der sparsen Matrix.

$I:$	1	3	2	1	4	5	3	4	5	2	5
$J:$	1	1	2	3	3	3	4	4	4	5	5
$A:$	2	2	3	1	1	1	1	2	2	1	1

(2) Bandbreitenreduzierung

Günstiger ist es die NNE in einem möglichst engen Teilbereich der Matrix zu konzentrieren, z. B. nahe der Hauptdiagonalen. Die Bandbreitenreduktion ist ein eigenständiger Algorithmus, der auf graphentheoretischen Betrachtungen mit den NNE als Knoten basiert. Ziel ist es dabei, durch eine Umnummerierung der Knoten im Graphen der Matrix die Bandbreite zu verkleinern. Die minimale Bandbreite zu erreichen, ist ein sehr komplexer Algorithmus. Deshalb begnügt man sich mit "fast optimalen" Verfahren, die auch eine deutliche Verbesserung liefern. Algebraisch bedeutet die Umnummerierung Zeilen- und Spaltenvertauschungen in der Matrix.

Hat man eine symmetrische Matrix A, so will man nach Transformationen die Symmetrie erhalten, so dass sich die Form

$$\tilde{A} \;=\; P^T A P, \;\; \tilde{A} = \tilde{A}^T,$$

anbietet, wobei P eine Permutationsmatrix ist. Diese wird aus dem Permutationsvektor $p = (p_1, p_2, ..., p_n)$ erzeugt, indem in der i-ten Zeile von P an der Stelle (i, p_i) eine Eins steht und sonst Nullen sind. Damit ist auch $P^T P = P P^T = I$.

Beispiel 1.4 (siehe [3])

Sei

$$A^T = A = A(10,10) = (a_{ij}) = \begin{pmatrix}
* & . & . & * & . & . & . & . & . & . \\
. & * & * & * & . & . & . & * & * & . \\
. & * & * & . & . & . & . & * & * & . \\
* & * & . & * & * & . & . & . & * & . \\
. & . & . & * & * & . & * & . & . & . \\
. & . & . & . & . & * & . & * & . & * \\
. & . & . & . & * & . & * & . & . & . \\
. & * & * & . & . & * & . & * & . & . \\
. & * & * & * & . & . & . & . & * & . \\
. & . & . & . & . & * & . & . & . & *
\end{pmatrix}.$$

Die $nne = 2 \cdot 12 + 10 = 34$ NNE sind mit $*$ gekennzeichnet.
Die Matrix hat die Bandbreite $2 \cdot 7 + 1 = 15$, da in 7 Nebendiagonalen NNE auftreten.

Den Permutationsvektor erhält man mit der Betrachtung des zugehörigen Graphens, seiner 10 Knoten und der Zusammenhangsmatrix (Adjazenzmatrix). Wir zeichnen die 10 Knoten 1,2,...,10 und verbinden zwei verschiedene Knoten i und j mit einer Kante, falls $a_{ij} \neq 0$ ist.
Somit haben wir die Kante 1 – 4 (ist gleich Kante 4 - 1), weiter die Kanten 2 – 3, 2 – 4, 2 – 8, 2 – 9, 3 – 8, 3 – 9, 4 – 5, 4 – 9, 5 – 7, 6 – 8, 6 – 10, die sich aus dem oberen Dreieck der Matrix A ablesen lassen. Damit ensteht ein sogenanntes Kugelmodell aus 10 Kugeln und 12 Fäden, das man als Graphen zu A bezeichnet.

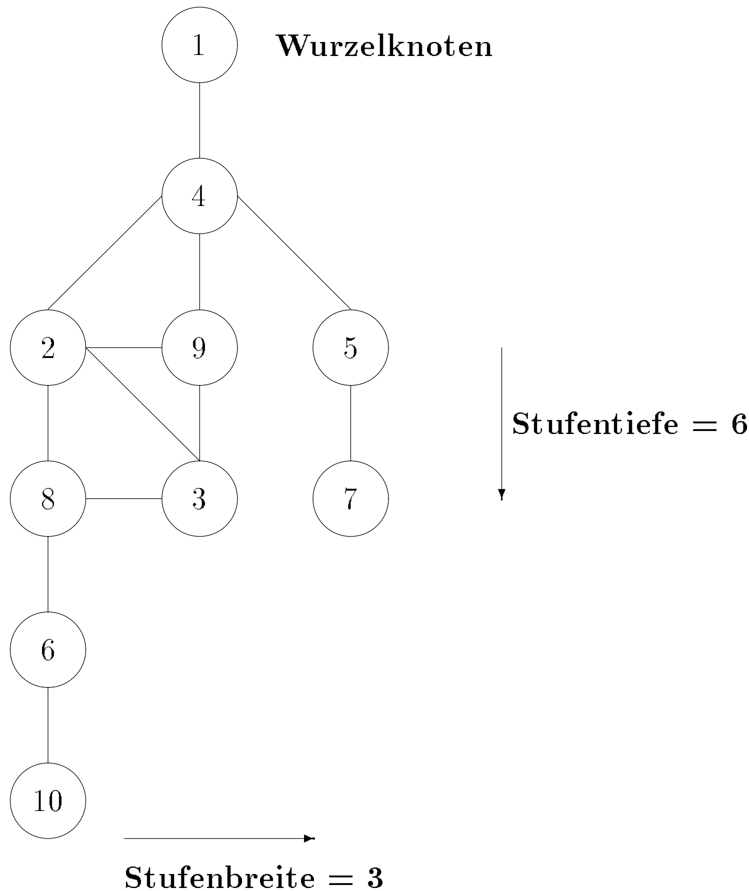


Abb. 1.1 Kugelmodell und Graph G zur Matrix A

Ziel ist es, die zusammenhängenden Knoten auf möglichst viele Niveaus (Stufen, Schichten) zu verteilen, so dass die Stufentiefe wächst und die Stufenbreite abnimmt. Wir machen einige Versuche, das Kugelmodell an einer anderen Kugel anzuheben und zu beobachten, wie sie dann die Kugeln zu neuen Niveaus anordnen. Dieses eher heuristische Prinzip erweist sich als erfolgreich und liegt auch dem eigentlichen Algorithmus zu Grunde.

Man könnte hier also 10 solche Kugelmodelle testen.

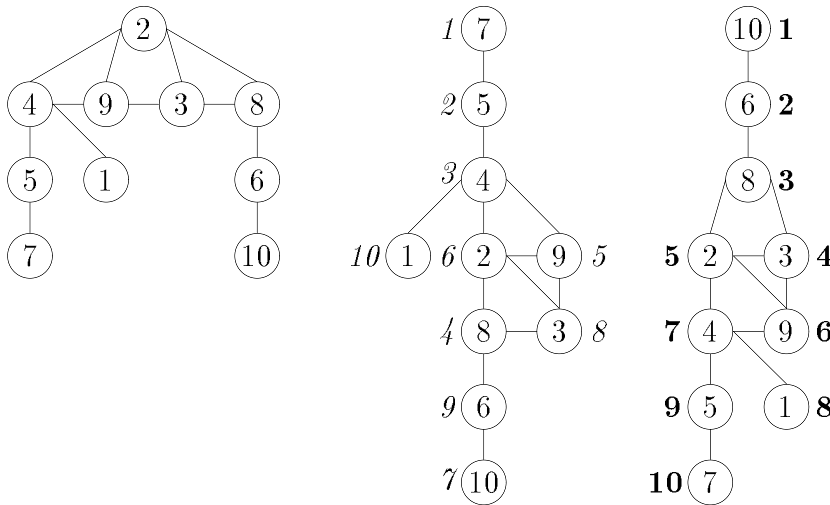


Abb. 1.2 Weitere Kugelmodelle (Graphen) zur Matrix A

Zu den Startknoten ergibt sich jeweils eine Stufenstruktur.
Auf Grund der genannten Zielstellung wählt man Startknoten mit kleinstem Grad (Valenz), das ist die Anzahl der Kanten, die von dem Knoten ausgehen. Dann nummeriert man "von oben nach unten" (siehe rechtes Kugelmodell). Über die Nummerierung der Knoten auf einer Stufe entscheiden sowohl Vorgängerknoten als auch die Grade der Knoten in der aktuellen Stufe.
Somit findet man durch geschicktes Probieren sogar ein optimales Kugelmodell und gleichzeitig die Umnummerierung der Knoten. Damit ergibt sich aus der rechten Darstellung die Neunummerierung der Knoten in den 7 Stufen und daraus der Permutationsvektor $p = (8, 5, 4, 7, 9, 2, 10, 3, 6, 1)$.

Die Permutationsmatrix P ist demzufolge

$$P = \begin{pmatrix} . & . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & 1 & . & . & . & . & . \\ . & . & . & 1 & . & . & . & . & . & . \\ . & . & . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & . & . & 1 & . \\ . & 1 & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & 1 \\ . & . & 1 & . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . & . & . \\ 1 & . & . & . & . & . & . & . & . & . \end{pmatrix}.$$

Die eingetragene Neunummerierung der Knoten in der mittleren Darstellung in Abb. 1.2 ergibt sich mit dem MATLAB-Befehl `symrcm`, auf den später eingegangen wird.

Wir berechnen nun

$$
\begin{aligned}
\tilde{A} &= P^T A P \\
&= P^T \begin{pmatrix}
* & . & . & * & . & . & . & . & . & . \\
. & * & * & * & . & . & . & * & * & . \\
. & * & * & . & . & . & . & * & * & . \\
* & * & . & * & * & . & . & . & * & . \\
. & . & . & * & * & . & * & . & . & . \\
. & . & . & . & . & * & . & * & . & * \\
. & . & . & . & * & . & * & . & . & . \\
. & * & * & . & . & * & . & * & . & . \\
. & * & * & * & . & . & . & . & * & . \\
. & . & . & . & . & * & . & . & . & *
\end{pmatrix}
\begin{pmatrix}
. & . & . & . & . & . & . & 1 & . & . \\
. & . & . & . & 1 & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . & . & . \\
. & . & . & . & . & . & 1 & . & . & . \\
. & . & . & . & . & . & . & . & 1 & . \\
. & 1 & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & 1 \\
. & . & 1 & . & . & . & . & . & . & . \\
. & . & . & . & . & 1 & . & . & . & . \\
1 & . & . & . & . & . & . & . & . & .
\end{pmatrix} \\
&= \begin{pmatrix}
. & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & . & . & 1 & . & . \\
. & . & 1 & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & 1 & . \\
. & . & . & 1 & . & . & . & . & . & . \\
1 & . & . & . & . & . & . & . & . & . \\
. & . & . & . & 1 & . & . & . & . & . \\
. & . & . & . & . & . & 1 & . & . & .
\end{pmatrix}
\begin{pmatrix}
. & . & . & . & . & . & * & * & . & . \\
. & . & * & * & * & * & * & . & . & . \\
. & . & * & * & * & * & . & . & . & . \\
. & . & . & . & * & * & * & * & * & . \\
. & . & . & . & . & . & * & . & * & * \\
* & * & * & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & * & * \\
. & * & * & * & * & . & . & . & . & . \\
. & . & . & * & * & * & * & . & . & . \\
* & * & . & . & . & . & . & . & . & .
\end{pmatrix} \\
&= \begin{pmatrix}
* & * & . & . & . & . & . & . & . & . \\
* & * & * & . & . & . & . & . & . & . \\
. & * & * & * & * & . & . & . & . & . \\
. & . & * & * & * & * & . & . & . & . \\
. & . & * & * & * & * & * & . & . & . \\
. & . & . & * & * & * & * & . & . & . \\
. & . & . & . & * & * & * & * & * & . \\
. & . & . & . & . & . & * & * & . & . \\
. & . & . & . & . & . & * & . & * & * \\
. & . & . & . & . & . & . & . & * & *
\end{pmatrix}
\end{aligned}
$$

Die neue Bandbreite ist $2 \cdot 2 + 1 = 5$. Sie ist optimal, denn es gibt in der Matrix A einige Zeilen mit 5 NNE. Die Bandbreite kann die minimale Anzahl von NNE in einer Matrixzeile nicht unterschreiten.

Wie gut unsere heuristische Vorgehensweise ist, zeigt ein Vergleich mit MATLAB-Befehlen, die auch Bandbreitenreduktionen der Matrix A, gegeben in ihrer sparsen Struktur S, machen.

Wir haben die Kommandos

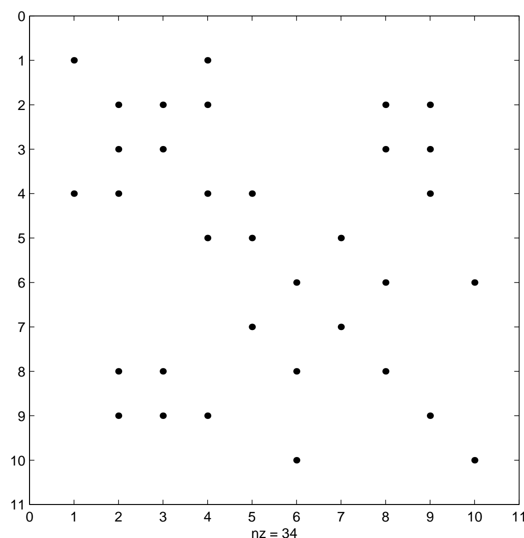
- `p = symmmd(S)`
  returns a symmetric minimum degree ordering of sparse symmetric matrix S using the column minimum degree algorithm; this is a permutation p such that $S(p,p)$ tends to have a sparser structure (not necessarily a smaller bandwidth)
- `r = symrcm(S)`
  returns a symmetric reverse Cuthill-McKee ordering of sparse matrix S using this algorithm; this is a permutation r such that $S(r,r)$ tends to have its nonzero elements closer to the diagonal

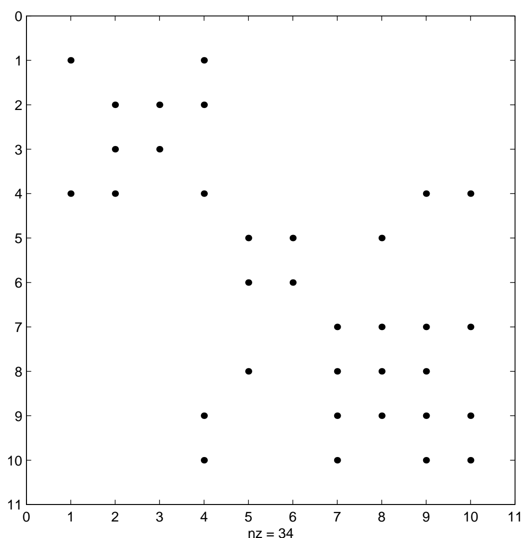
Aus den Permutationsvektoren p bzw. r werden die Permutationsmatrizen P^T bzw. R^T und damit P^TAP (R^TAR) mit der kompakten Elementstruktur berechnet.
Wir betrachten die sparse Struktur vor und nach Anwendung des jeweiligen Algorithmus.

$$A = A(10,10) = (a_{ij}) = \begin{pmatrix} * & . & . & * & . & . & . & . & . & . \\ . & * & * & * & . & . & . & * & * & . \\ . & * & * & . & . & . & . & * & * & . \\ * & * & . & * & * & . & . & . & * & . \\ . & . & . & * & * & . & * & . & . & . \\ . & . & . & . & . & * & . & * & . & * \\ . & . & . & . & * & . & * & . & . & . \\ . & * & * & . & . & * & . & * & . & . \\ . & * & * & * & . & . & . & . & * & . \\ . & . & . & . & . & * & . & . & . & * \end{pmatrix}, \quad nne = 34.$$

```
>>S = sparse(...
  [1,1,2,2,2,2,2,3,3,3,3,4,4,4,4,4,5,5,5,6,6, 6,7,7,8,8,8,8,9,9,9,9,10,10],...
  [1,4,2,3,4,8,9,2,3,8,9,1,2,4,5,9,4,5,7,6,8,10,5,7,2,3,6,8,2,3,4,9, 6,10],...
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, 1,1,1,1,1,1,1,1,1,1,1, 1, 1],...
  10,10,34);

>>spy(S)
>>print bild03.ps -dps
>>p = symmmd(S)    % Bandweite nicht unbedingt kleiner

p =
     1     5     7     4     6    10     3     8     2     9
```

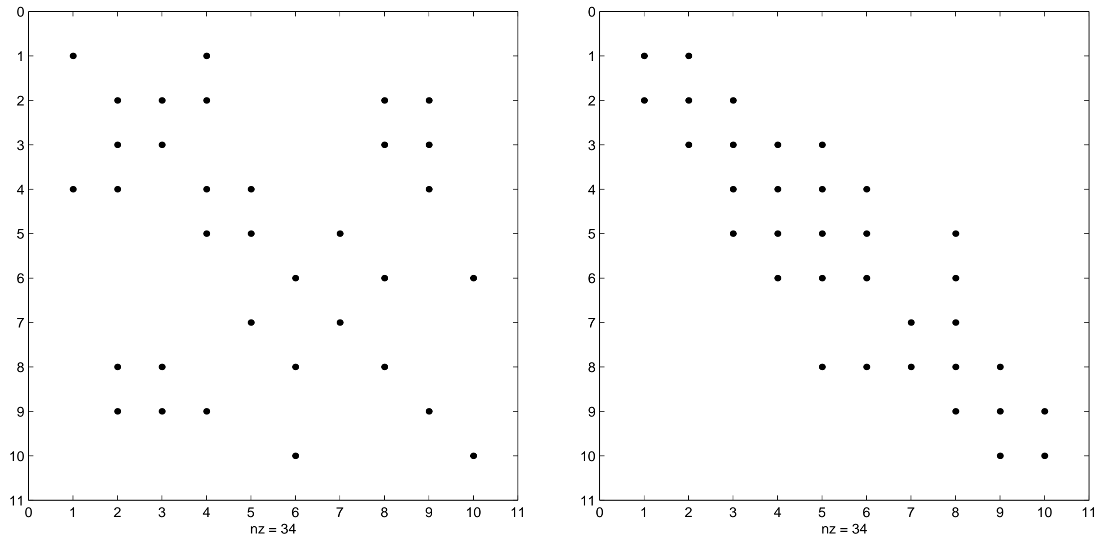
```
>>S(p,p)

ans =
   (1,1)  1   (4,1)  1   (2,2)  1   (3,2)  1   (4,2)  1   (2,3)  1
   (3,3)  1   (1,4)  1   (2,4)  1   (4,4)  1   (9,4)  1  (10,4)  1
   (5,5)  1   (6,5)  1   (8,5)  1   (5,6)  1   (6,6)  1   (7,7)  1
   (8,7)  1   (9,7)  1  (10,7)  1   (5,8)  1   (7,8)  1   (8,8)  1
   (9,8)  1   (4,9)  1   (7,9)  1   (8,9)  1   (9,9)  1  (10,9)  1
  (4,10)  1  (7,10)  1  (9,10)  1 (10,10)  1

>>spy(S(p,p))
>>print bild04.ps -dps
```






```
>>r = symrcm(S)
r =
    10     6     8     3     2     9     1     4     5     7

>>S(r,r)

ans =
   (1,1)  1   (2,1)  1   (1,2)  1   (2,2)  1   (3,2)  1   (2,3)  1
   (3,3)  1   (4,3)  1   (5,3)  1   (3,4)  1   (4,4)  1   (5,4)  1
   (6,4)  1   (3,5)  1   (4,5)  1   (5,5)  1   (6,5)  1   (8,5)  1
   (4,6)  1   (5,6)  1   (6,6)  1   (8,6)  1   (7,7)  1   (8,7)  1
   (5,8)  1   (6,8)  1   (7,8)  1   (8,8)  1   (9,8)  1   (8,9)  1
   (9,9)  1  (10,9)  1  (9,10)  1 (10,10)  1
```

```
>>spy(S(r,r))
>>print bild5.ps -dps
```



Vergleich der Bandbreite $bw(A) = 15$ mit den erzielten neuen Bandbreiten:

- Kugelmodell: $bw = 5$,
- symmetric minimum degree ordering: $bw = 13$,
- reverse Cuthill-McKee ordering: $bw = 7$.

Wir bemerken noch, dass keines der Kugelmodelle zu der Umnummerierung durch die MATLAB-Kommandos passt.

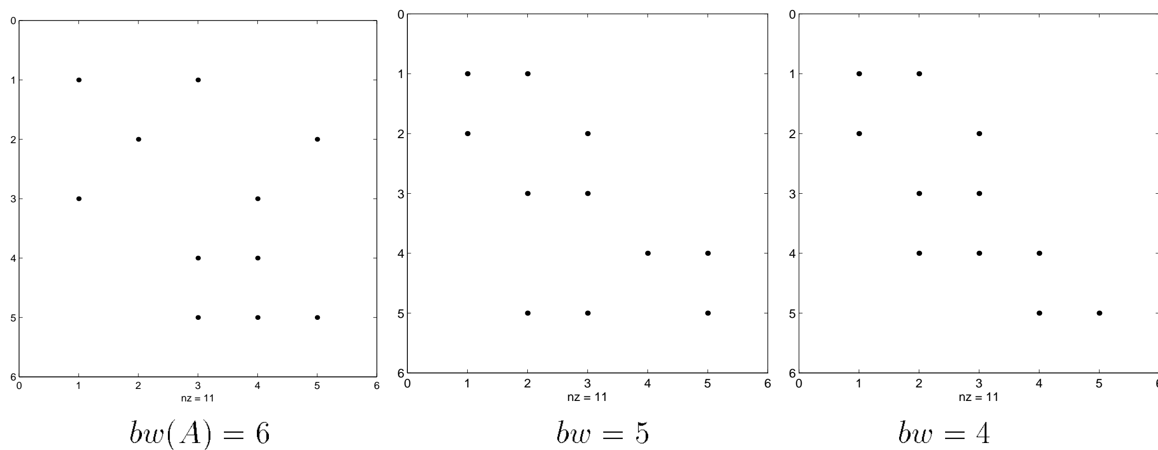
Beispiel 1.5

Wir betrachten erneut die sparse, aber nicht symmetrische Matrix

$$A = (a_{ij}) = \begin{pmatrix} 2 & & 1 & & \\ & 3 & & & 1 \\ 2 & & & 1 & \\ & & 1 & 2 & \\ & & 1 & 2 & 1 \end{pmatrix}, \quad bw(A) = 6, \; nne = 11.$$

```
>>A = [2  0  1  0  0
       0  3  0  0  1
       2  0  0  1  0
       0  0  1  2  0
       0  0  1  2  1];
```

```
>>S = sparse(A)

S =
   (1,1)  2   (3,1)  2   (2,2)  3   (1,3)  1   (4,3)  1   (5,3)  1
   (3,4)  1   (4,4)  2   (5,4)  2   (2,5)  1   (5,5)  1

>>p = symmmd(S)
p =
     1     3     4     2    5

>>spy(S(p,p))
>>print bild06.ps -dps

>>r = symrcm(S)
r =
     1     3     4     5    2

>>spy(S(r,r))
>>print bild07.ps -dps
```



$bw(A) = 6$ $\quad$ $bw = 5$ $\quad$ $bw = 4$

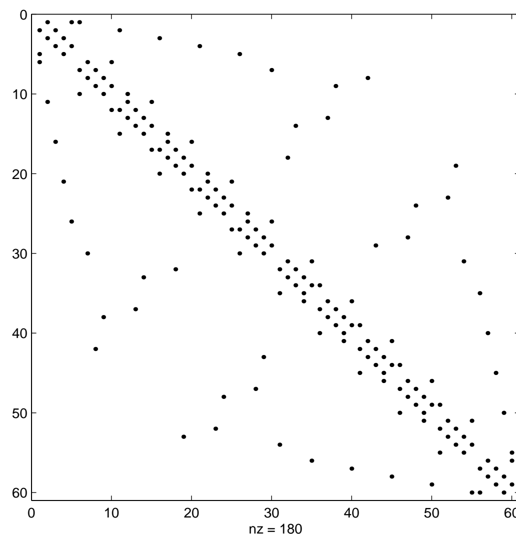
Die Permutationsmatrizen für $\tilde{A} \; = \; P^T A P \; (R^T A R)$ sind

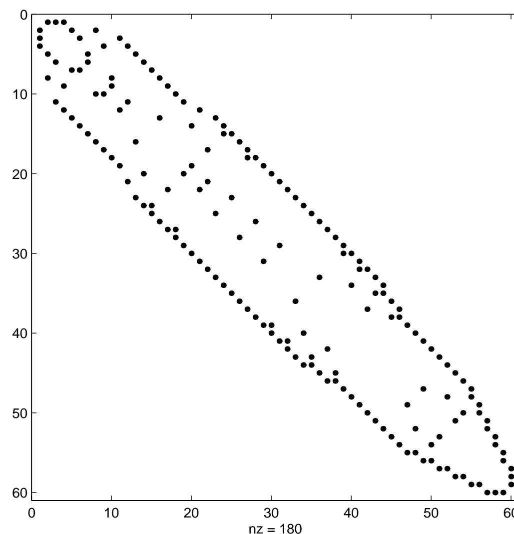
$$P^T = \begin{pmatrix} 1 & & & & \\ & & 1 & & \\ & & & 1 & \\ & 1 & & & \\ & & & & 1 \end{pmatrix}, \quad R^T = \begin{pmatrix} 1 & & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ & 1 & & & \end{pmatrix}.$$

Beispiel 1.6

Wir verringern die Bandbreite der symmetrischen (60×60)-Matrix aus dem m-File `bucky` der Demo-Toolbox von MATLAB.

```
>>B = bucky;
>>spy(B)
>>p = symrcm(B)
p =
    1     6     2     5    10    11    12     7    26    30     3     4
    9    15    13     8    27    29    16    17    21    25    38    14
   37    42    28    43    20    18    22    24    39    33    36    41
   47    44    19    32    23    48    40    34    45    46    53    31
   52    49    57    35    58    50    54    51    56    59    55    60

>>spy(B(p,p))
>>print bild08.ps -dps
>>[i,j] = find(B)
>>bw = max(i-j)-min(i-j)+1
bw =
    69

>>[i,j] = find(B(p,p))
>>bw = max(i-j)-min(i-j)+1
bw =
    21
```





Anschließend kann man weniger aufwendige Bandvarianten zur Faktorisierung der Koeffizientenmatrix oder Lösung des LGS aufrufen.

1.3 Band- und besondere Matrizen

Die Ausgangsmatrix hat die folgende Bandstruktur.

$$A(n,n) = (a_{ij}) = \begin{pmatrix} * & * & * & * & * & . & . & . & . & . \\ * & * & * & * & * & * & . & . & . & . \\ * & * & * & * & * & * & * & . & . & . \\ . & * & * & * & * & * & * & * & . & . \\ . & . & * & * & * & * & * & * & * & . \\ . & . & . & * & * & * & * & * & * & * \\ . & . & . & . & * & * & * & * & * & * \\ . & . & . & . & . & * & * & * & * & * \\ . & . & . & . & . & . & * & * & * & * \\ . & . & . & . & . & . & . & * & * & * \end{pmatrix} \tag{1.6}$$

(Die geschweifte Klammer β rechts umfasst die Zeilen 6 bis 9, die geschweifte Klammer α unten die Spalten 8 und 9.)

Das Symbol $*$ charakterisiert den Bereich der NNE der Matrix und damit die Nebendiagonalen (Kodiagonalen), wo NNE auftreten.
Die Anzahl der unteren Nebendiagonalen mit NNE beträgt $\alpha \geq 0$, die der oberen ist $\beta \geq 0$. Es gilt

$$\begin{aligned} \alpha &= \begin{cases} 0, & \text{falls } a_{ij} = 0 \text{ für alle } i > j \\ \max\limits_{\substack{i>j \\ a_{ij} \neq 0}} (i-j), & \text{sonst,} \end{cases} \\ \beta &= \begin{cases} 0, & \text{falls } a_{ij} = 0 \text{ für alle } j > i \\ \max\limits_{\substack{j>i \\ a_{ij} \neq 0}} (j-i) = -\min\limits_{\substack{j>i \\ a_{ij} \neq 0}} (i-j), & \text{sonst,} \end{cases} \end{aligned} \tag{1.7}$$

Als Bandbreite bezeichnen wir die Größe

$$bw(A) = \alpha + \beta + 1. \tag{1.8}$$

Im Allgemeinen ist $\alpha+\beta+1 \ll n$. Ist $\alpha = \beta = 0$, so erhalten wir eine Diagonalmatrix. Ist einer der Werte gleich Null, so haben wir eine obere bzw. untere Dreiecksmatrix. Für $\alpha = \beta$ liegt eine symmetrische Bandstruktur vor, jedoch noch nicht die Symmetrie der NNE. Für $A = A^T$ bezeichnen wir mit $\alpha+1 = \beta+1$ die halbe Bandbreite, in der Literatur jedoch manchmal auch als Bandbreite genannt. Die Bandbreite einer Nullmatrix ist somit 1.
Als erste modifizierte Bandbreite bezeichnen wir die Größe

$$b1(A) = 1 + 2 \max_{\substack{i,j=1(1)n \\ j \neq i}} (|i-j| \text{ mit } a_{ij} \neq 0 \text{ oder } a_{ji} \neq 0). \tag{1.9}$$

Als zweite modifizierte Bandbreite bezeichnen wir die Größe

$$b2(A) = \min(m, \quad \text{wobei } a_{ij} = 0 \; \forall i, j \text{ mit } |i - j| > m). \tag{1.10}$$

Die Bandbreite $b2$ ist somit gleich der Anzahl der Nebendiagonalen oberhalb, resp. unterhalb der Hauptdiagonalen, welche NNE enthalten.
Jede voll besetzte Matrix hat die Bandbreiten $bw = b1 = 2n - 1, \; b2 = n - 1$. Aber auch viele sparse Matrizen haben diese modifizierten Bandbreiten, z. B. die Matrix

$$\begin{pmatrix} * & & & & * \\ & * & & & \\ & & * & & \\ & & & * & \\ & & & & * \end{pmatrix}$$

mit $bw = n, \; b1 = 2n - 1$ und $b2 = n - 1$.
Ist die Bandbreite wesentlich kleiner als n, so speichert man statt des $(n \times n)$-Feldes der Matrix A nur das $(n \times (\alpha + \beta + 1))$-Rechteck des Bandes gemäß

$$\begin{pmatrix}
+ & * & * & * & * & . & . & . & . & . \\
* & + & * & * & * & * & . & . & . & . \\
* & * & + & * & * & * & * & . & . & . \\
. & * & * & + & * & * & * & * & . & . \\
. & . & * & * & + & * & * & * & * & . \\
. & . & . & * & * & + & * & * & * & * \\
. & . & . & . & * & * & + & * & * & * \\
. & . & . & . & . & * & * & + & * & * \\
. & . & . & . & . & . & * & * & + & * \\
. & . & . & . & . & . & . & * & * & +
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
0 & 0 & + & * & * & * & * \\
0 & * & + & * & * & * & * \\
* & * & + & * & * & * & * \\
* & * & + & * & * & * & * \\
* & * & + & * & * & * & * \\
* & * & + & * & * & * & * \\
* & * & + & * & * & * & 0 \\
* & * & + & * & * & 0 & 0 \\
* & * & + & * & 0 & 0 & 0 \\
* & * & + & 0 & 0 & 0 & 0
\end{pmatrix},$$

wobei die Diagonale extra gekennzeichnet wurde, bzw. im Fall $A = A^T$

$$\begin{pmatrix}
+ & * & * & * & * & . & . & . & . & . \\
* & + & * & * & * & * & . & . & . & . \\
* & * & + & * & * & * & * & . & . & . \\
* & * & * & + & * & * & * & * & . & . \\
* & * & * & * & + & * & * & * & * & . \\
. & * & * & * & * & + & * & * & * & * \\
. & . & * & * & * & * & + & * & * & * \\
. & . & . & * & * & * & * & + & * & * \\
. & . & . & . & * & * & * & * & + & * \\
. & . & . & . & . & * & * & * & * & +
\end{pmatrix}
\Rightarrow
\begin{pmatrix}
+ & * & * & * & * \\
+ & * & * & * & * \\
+ & * & * & * & * \\
+ & * & * & * & * \\
+ & * & * & * & * \\
+ & * & * & * & * \\
+ & * & * & * & 0 \\
+ & * & * & 0 & 0 \\
+ & * & 0 & 0 & 0 \\
+ & 0 & 0 & 0 & 0
\end{pmatrix}. \tag{1.11}$$

Bei der Verwendung von Lösungsalgorithmen sollte es das Ziel sein, in dieser Speicherstruktur des Rechtecks zu bleiben, also mit seinen Spalten als Vektoren zu arbeiten.

Weitere besondere Matrixstrukturen werden beispielhaft mit einer (5×5)-Matrix dargestellt. Dabei kann das Symbol $*$ auch eine Untermatrix bzw. einen Block bedeuten.

Dazu gehören die Tridiagonalform sowie die obere und untere Hessenberg-Form

$$\begin{pmatrix} * & * & & & \\ * & * & * & & \\ & * & * & * & \\ & & * & * & * \\ & & & * & * \end{pmatrix}, \quad \begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ & * & * & * & * \\ & & * & * & * \\ & & & * & * \end{pmatrix}, \quad \begin{pmatrix} * & * & & & \\ * & * & * & & \\ * & * & * & * & \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix},$$

symmetrische Formen mit zusätzlichen Ecken

$$\begin{pmatrix} * & & & & * \\ & * & & & \\ & & * & & \\ & & & * & \\ * & & & & * \end{pmatrix}, \quad \begin{pmatrix} * & * & & & * \\ * & * & * & & \\ & * & * & * & \\ & & * & * & * \\ * & & & * & * \end{pmatrix},$$

die Gestalt eines "Pfeils nach unten bzw. oben"

$$\begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix}, \quad \begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix}.$$

Man kann leicht nachrechnen, das die Matrix A_o "Pfeil nach oben" sich durch eine einfache Zeilen- und Spaltenvertauschung in die Matrix A_u "Pfeil nach unten" überführen lässt. Der Permutationsvektor dazu ist $p = (5, 2, 3, 4, 1)$, so dass gilt

$$A_u = P^T A_o P, \;\; P = P^T = \begin{pmatrix} & & & & 1 \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ 1 & & & & \end{pmatrix}.$$

In den bisher gezeigten Beispielen sind diese Matrixstrukturen teilweise schon aufgetreten.

Kapitel 2

Lösung von LGS und Komplexität der Algorithmen

Direkte Verfahren zur Lösung von reellen LGS enthalten sowohl die Technik der Elimination der Unbekannten als auch die Faktorisierung der Koeffizientenmatrix. Betrachten wir das LGS in seiner Normalform mit möglichen Darstellungen.

$$Ax = b, \quad A(n,n) = (a_{ij})_{i,j=1}^{n}, \quad a_{ij}, b_i \in \mathbb{R}, \tag{2.1}$$

$$\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1, 2, ..., n,$$

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}.$$

Zunächst geben wir eine Übersicht einiger Lösungsalgorithmen an, die im Weiteren eingehender behandelt werden. Ausgehend von (2.1) wird jeweils das Ziel der Umformungen notiert.

- Gauß-Reduktion ohne/mit Spaltenpivotisierung, Resttableau-Algorithmus:
  $Ax = b \;\to\; Bx = c$, streng unteres Dreieck von B ist Null.

- Verketteter Gauß-Algorithmus (VGA) ohne Pivotisierung:
  $Ax = b \;\to\; Ax = -CBx = b \;\to\; Bx = c$ mit LU-Faktorisierung
  $A = -CB = LU$, C, B linke untere bzw. rechte obere Dreiecksmatrix.

- Gauß-Jordan-Algorithmus:
  $Ax = b \;\to\; Ix = c$, I Einheitsmatrix.

- VGA mit Spaltenpivotisierung und Zeilenvertauschung:
  $Ax = b \;\to\; PAx = -CBx = Pb \;\to\; Bx = c$ mit Faktorisierung
  $PA = -CB = LU$, P Permutationsmatrix.

- Cholesky-Verfahren für reelle symmetrische (positiv definite) Matrix:
  $Ax = b \;\to\; LL^Tx = b \;\to\; Ly = b \;\to\; L^Tx = y$ mit LL^T-Faktorisierung
  $A = LL^T$.

Die folgenden Untersuchungen werden i. Allg. ohne Pivotstrategien durchgeführt.

Wie bedeutend der Gaußsche Algorithmus (GA) für Rechnungen mit Matrizen ist, zeigt die kurze Betrachtung des Aufwands zur Berechnung der Determinante.

1. Cramersche Regel und Entwicklungssatz für Determinanten

z_n sei die Anzahl der Multiplikationen bei einer n-reihigen Determinante.
Dann sind $z_1 = 0,\; z_2 = 2$ und

$$z_n \geq nz_{n-1} \geq n(n-1)z_{n-2} \geq \ldots \geq n!$$

Die Cramersche Regel benötigt somit $T(n) \geq (n+1)z_n \geq n!$ Multiplikationen.

Bei einem sehr schnellen Computer im Gflop-Bereich dauert 1 Multiplikation angenommen 1 Nanosekunde (10^{-9} *sec*). Damit ergeben sich folgende Rechenzeiten der Determinantenauswertung für $(n \times n)$-Matrizen, nur die Multiplikationen berücksichtigt.

n	Rechenzeit ist größer als
10	0.0036 Sekunden
15	21.8 Minuten
20	77.1 Jahre
30	8.41 Billiarden Jahre = $8.41 \cdot 10^{15}$ Jahre

Tab. 2.1 Rechenzeit für det(A) mit Cramerscher Regel

2. Gaußscher Algorithmus

Dieses Standardverfahren der numerischen linearen Algebra braucht, wie später auch hergeleitet wird, ungefähr $n^3/3$ Multiplikationen bzw. Divisionen und ähnlich viele Additionen/Subtraktionen. Bei Berücksichtigung nur der Multiplikationen haben wir also die Komplexität $T(n) = n^3/3 + \mathcal{O}(n^2)$.
Wenn wir nun auf Computern wie PC Pentium II, III oder IV rechnen, die weniger als 10^9 Gleitpunktoperationen pro Sekunde ausführen, im Durchschnitt 10 bis 1000 Mal langsamer sind, dann ergeben sich trotzdem noch akzeptable Rechenzeiten für große Dimensionen. Eine Multiplikation möge also 10^{-8} bzw. 10^{-6} *sec* dauern.

	ungefähre Rechenzeit in sec	
n	$*$ mit 10^{-8} *sec*	$*$ mit 10^{-6} *sec*
10	0.000 003	0.003 333
15	0.000 011	0.001 125
20	0.000 027	0.002 667
50	0.000 417	0.041 667
100	0.003 333	0.333 333
400	0.213 333	21.333 333

Tab. 2.2
Rechenzeit für det(A) mit GA

Für die letzte Dimension $n = 400$ liegen praktische Rechnungen dazu auf einem PC Pentium 800MHz im Sekundenbereich, wenn man die Programme BP, Borland C++ oder auch das Computeralgebrasystem (CAS) Matlab mit den verfügbaren *built-in*-Funktionen `det` oder `lu` verwendet.

2.1 Gaußscher Algorithmus

Der GA für $Ax = b$ basiert auf der LU-Faktorisierung der Matrix A mit der unteren und oberen Dreiecksmatrix L bzw. U gemäß

$$A = LU. \tag{2.2}$$

Pivotstrategien sollen dabei nicht vordergründig betrachtet werden. Die vollständige Faktorisierung mit

$$L = (l_{ij}) = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \multicolumn{5}{c}{\dotfill} \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{pmatrix}, \quad U = (u_{ij}) = \begin{pmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \multicolumn{5}{c}{\dotfill} \\ 0 & 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

und der Bedingung

$$a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} \tag{2.3}$$

ergibt die Grundvariante mit der abwechselnden zeilenweisen Berechnung von U und spaltenweisen Berechnung von L. Dabei können die Dreiecksmatrizen auch an der Stelle der Matrix A gespeichert werden. Analog kann man den Ansatz $A = -CB$ machen mit

$$C = (c_{ij}) = \begin{pmatrix} -1 & 0 & 0 & \cdots & 0 \\ c_{21} & -1 & 0 & \cdots & 0 \\ c_{31} & c_{32} & -1 & \cdots & 0 \\ \multicolumn{5}{c}{\dotfill} \\ c_{n1} & c_{n2} & c_{n3} & \cdots & -1 \end{pmatrix}, \quad B = (b_{ij}) = \begin{pmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1n} \\ 0 & b_{22} & b_{23} & \cdots & b_{2n} \\ 0 & 0 & b_{33} & \cdots & b_{3n} \\ \multicolumn{5}{c}{\dotfill} \\ 0 & 0 & & \cdots & b_{nn} \end{pmatrix}.$$

Einige Grundvarianten des Gaußschen Algorithmus

(1) Resttableau-Algorithmus

Ziel ist hier die Überführung der Matrix (evtl. mit rechter Seite) in eine obere Dreiecksform. Diesen Schritt bezeichnet man als **Vorwärtselimination**.

Der Platz der entstehenden NE unterhalb der Diagonalen wird für die Speicherung der unteren Dreicksmatrix genutzt. Die Berechnung erfolgt auf dem Platz der Matrix A und liefert auch die vollständige oder einfach LU-Faktorisierung. Man beachte, dass beim Schleifendurchlauf $k = n$ eigentlich nur der Test des Diagonalelements auf Null gemacht wird und kein Resttableau mehr da ist.

$A = LU,\ l_{ii} = 1.$

$$
\begin{aligned}
k \quad &= \quad 1, 2, ..., n\\
&\quad p = a_{kk},\\
&\qquad \text{vorzeitiger Abbruch bei } \ p = 0\\
&\quad i = k+1, k+2, ..., n\\
&\qquad s = a_{ik}/p\\
&\qquad a_{ik} = s\\
&\qquad a_{ij} = a_{ij} - s\, a_{kj}, \quad j = k+1, k+2, ..., n
\end{aligned}
$$

(2) Verketteter Gauß-Algorithmus

Wiederum wird die Matrix A durch die Faktorisierungskomponenten überspeichert. Im unteren Dreieck unterhalb der Diagonalen von A werden die Elemente $c_{ij},\ i > j$, der Matrix C gespeichert. Weiterhin könnte man die Übernahme der ersten Zeile von A in die Matrix B vor der k-Schleife machen.

$A = -CB = LU,\ c_{ii} = -1, \quad L = -C,\ U = B.$

$$
\begin{aligned}
k \quad &= \quad 1, 2, ..., n\\
&\quad p = a_{kk} + \sum_{i=1}^{k-1} a_{ki} a_{ik},\\
&\qquad \text{vorzeitiger Abbruch bei } \ p = 0\\
&\quad a_{kk} = p\\
&\quad a_{kj} = a_{kj} + \sum_{i=1}^{k-1} a_{ki} a_{ij}, \quad j = k+1, k+2, ..., n\\
&\quad a_{ik} = -\left(a_{ik} + \sum_{j=1}^{k-1} a_{ij} a_{jk} \right) /p, \quad i = k+1, k+2, ..., n
\end{aligned}
$$

(3) Variante VGA/LU-Faktorisierung mit $A = LU,\ l_{ii} = 1$

Initialisierung von U als Nullmatrix und L als Einheitsmatrix, erste Zeile von U extra.

$$
\begin{aligned}
u_{1j} &= a_{1j}, \quad j = 1, 2, ..., n \\
k &= 1, 2, ..., n-1 \\
&\quad p = u_{kk} \\
&\quad l_{ik} = \frac{1}{p}\left(a_{ik} - \sum_{s=1}^{k-1} l_{is} u_{sk}\right), \quad i = k+1, k+2, ..., n \\
&\quad u_{k+1,j} = a_{k+1,j} - \sum_{s=1}^{k} l_{k+1,s} u_{sj}, \quad j = k+1, k+2, ..., n
\end{aligned}
$$

(4) Variante VGA/LU-Faktorisierung mit $A = LU,\ u_{ii} = 1$

Initialisierung von L als Nullmatrix und U als Einheitsmatrix, erste Spalte von L extra.

$$
\begin{aligned}
l_{i1} &= a_{i1}, \quad i = 1, 2, ..., n \\
k &= 1, 2, ..., n-1 \\
&\quad p = l_{kk} \\
&\quad u_{kj} = \frac{1}{p}\left(a_{kj} - \sum_{s=1}^{k-1} l_{ks} u_{sj}\right), \quad j = k+1, k+2, ..., n \\
&\quad l_{i,k+1} = a_{i,k+1} - \sum_{s=1}^{k} l_{is} u_{s,k+1}, \quad i = k+1, k+2, ..., n
\end{aligned}
$$

Bemerkungen

- Die Strategien sind durchführbar, wenn in allen bis auf den letzten Schritt das Diagonalelement $\neq 0$ ist.
- Das LGS ist lösbar, wenn weiterhin das letzte Diagonalelement $\neq 0$ ist.
- Falls ein Diagonalelement $= 0$ ist, dann ist die Anwendung von Pivotstrategien mit Zeilen- und/oder Spaltenvertauschungen ($\Rightarrow$ Permutationsvektoren) erforderlich.
- Die Determinante berechnet sich in Variante 3 aus $\det(A) = \prod_{k=1}^{n} u_{kk}$.
- Meist wird die Matrix systematisch durch die Resttableaus bzw. L, U überschrieben.

Steht nun die LU-Faktorisierung der Matrix A zur Verfügung, so kann man das LGS $Ax = LUx = b$ durch die gestaffelten LGS

$$Ly = b \quad \text{und} \quad Ux = y$$

der Reihe nach lösen. Bezieht man die rechte Seite b bei der Gauß-Elimination gleich mit ein, so entsteht aus der rechten Seite der Vektor y und es bleibt nur die **Rückwärtssubstitution** gemäß $Ux = y$ übrig. Der Gesamtaufwand bleibt in jedem Fall der gleiche.

Vorwärtselimination der rechten Seite als Lösung von $Ly = b$

$$\begin{array}{lll}
y_1 & = & b_1 \\
k & = & 2, 3, ..., n \\
\quad h & = & b_k \\
\quad h & = & h - l_{ki}\, y_i, \quad i = 1, 2, ..., k-1 \\
\quad y_k & = & h
\end{array}$$

Rückwärtssubstitution als Lösung von $Ux = y$

$$\begin{array}{lll}
x_n & = & y_n / u_{nn} \\
k & = & n-1, n-2, ..., 1 \\
\quad h & = & b_k \\
\quad h & = & h - u_{ki}\, x_i, \quad i = k+1, k+2, ..., n \\
\quad x_k & = & h / u_{kk}
\end{array}$$

Effizienz des Gauß-Algorithmus

Der arithmetische Hauptaufwand liegt in der LU-Faktorisierung der Matrix bzw. Vorwärtselimination des LGS mit rechter Seite. Wir betrachten jeden der drei Schritte, also LU-Faktorisierung, Vorwärtselimination sowie Rückwärtssubstitution für sich und fassen schließlich die Strichoperationen $\{+, -\}$ bzw. Punktoperationen $\{*, /\} = \{\cdot, :\}$ jeweils zusammen.

$A = LU$:

In Variante (1) verursacht die Anweisung $a_{ij} = a_{ij} - s\, a_{kj}$ im Inneren aller Schleifen mit einer Addition und einer Multiplikation den Hauptaufwand.

Wir haben mit dieser Vorschrift $n-1$ Resttableaus der Reihe nach zu erzeugen, das sind also $(n-1)^2+(n-2)^2+...+1^2$ Elemente mit je 1 Addition und 1 Multiplikation. Dazu kommen noch $(n-1)+(n-2)+...+1$ Divisionen für die Spalten. Damit ergibt sich der Gesamtaufwand beim Durchlaufen der k-Schleife

$$\begin{aligned}
T_{LU}(n) &= \sum_{k=1}^{n-1}(n-k)^2\ \{+,*\} \quad + \sum_{k=1}^{n-1}(n-k)\ \{/\} \ = \sum_{k=1}^{n-1} k^2\ \{+,*\} \quad + \sum_{k=1}^{n-1} k\ \{/\} \\
&= \frac{(n-1)n(2n-1)}{6}\ \ \{+,*\} \quad + \frac{(n-1)n}{2}\ \ \{/\} \\
&= \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}\ \ \{+\} \quad + \frac{n^3}{3} - \frac{n}{3}\ \ \{*\} \\
&= \frac{2n^3}{3} - \frac{n^2}{2} + \frac{n}{6},
\end{aligned}$$

$Ly = b:$

$$\begin{aligned}
T_V(n) &= \sum_{k=1}^{n-1} k\ \ \{+,*\} \ = \ \frac{(n-1)n}{2}\ \ \{+,*\} \\
&= \frac{n^2}{2} - \frac{n}{2}\ \ \{+\} \quad + \frac{n^2}{2} - \frac{n}{2}\ \ \{*\} \\
&= n^2 - n,
\end{aligned}$$

$Ux = y:$

$$\begin{aligned}
T_R(n) &= \sum_{k=1}^{n-1} k\ \ \{+,*\} \quad + n\ \ \{/\} \ = \ \frac{(n-1)n}{2}\ \ \{+\} \quad + \frac{(n+1)n}{2}\ \ \{*\} \\
&= \frac{n^2}{2} - \frac{n}{2}\ \ \{+\} \quad + \frac{n^2}{2} + \frac{n}{2}\ \ \{*\} \\
&= n^2.
\end{aligned}$$

Damit beträgt der Aufwand zur Lösung des LGS bei gegebener Faktorisierung LU $T_V(n) + T_R(n) = 2n^2 - n$ sowie der arithmetische Gesamtaufwand

$$T(n) = T_{LU}(n) + T_V(n) + T_R(n) = \frac{2n^3}{3} + \frac{3n^2}{2} + \frac{n}{6}. \tag{2.4}$$

Hat man m rechte Seiten im LGS zu verarbeiten, so ist die Komplexität

$$\begin{aligned}
T_m(n) &= T_{LU}(n) + m[T_V(n) + T_R(n)] \ = \ \Big(\frac{2n}{3} + 2m\Big)n^2 - \frac{n^2}{2} + \frac{n}{6} - mn \\
&\approx \frac{2n^3}{3} + 2mn^2, \quad m \gg 1.
\end{aligned}$$

Zur Berechnung der inversen Matrix A^{-1} muss das LGS n Mal gelöst werden, und zwar mit den n rechten Seiten als Einheitsvektoren.
Dazu kommen $2n^2$ *flops* für das Produkt $A^{-1}b$. Damit ist grob gerechnet

$$\begin{aligned} T_i(n) &= n[T_V(n) + T_R(n)] = 2n^3 - n^2, \\ T_{inv}(n) &= T_{LU}(n) + T_i(n) + 2n^2 = \frac{8n^3}{3} + \frac{n^2}{2} + \frac{n}{6}. \end{aligned}$$

Beachtet man jedoch, dass das gestaffelte System $Ly = b$ mit Einheitsvektoren auf der rechten Seite wegen der Nullen nur ca. die Hälfte von $T_V(n)$ braucht, dann ergibt sich eine genauere Abschätzung gemäß

$$\widetilde{T}_{inv}(n) = T_{LU}(n) + n\Big[\frac{1}{2}T_V(n) + T_R(n)\Big] + 2n^2 = \frac{13n^3}{6} + n^2 + \frac{n}{6}. \tag{2.5}$$

Wir machen nun die Aufwandsuntersuchung für die Vorwärtselimination im LGS $Ly = b$ mit $b = e_1, e_2, ..., e_n$ detaillierter. Zu lösen ist das System

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{pmatrix} Y = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

Für die 1. Lösung $y = (y_1, y_2, ..., y_n)^T$ als 1. Spaltenvektor von Y braucht man folgende Multiplikationen und Additionen:

$$\begin{aligned} &y_1: && 0, \\ &y_2: && 1\ \{*\}, \\ &y_3: && 2\ \{*\},\ 1\ \{+\}, \\ &\ldots \\ &y_{n-1}: && n-2\ \{*\},\ n-3\ \{+\}, \\ &y_n: && n-1\ \{*\},\ n-2\ \{+\}, \end{aligned}$$

also insgesamt $(n-1)n/2\ \{*\}$ und $(n-2)(n-1)/2\ \{+\}$.
Für die weiteren Lösungsvektoren reduziert sich der Aufwand jeweils um die "untere Zeile". Damit gilt

$$\begin{aligned} \overline{T}_V(n) &= \sum_{k=2}^{n} \frac{(k-1)k}{2}\ \{*\} + \sum_{k=2}^{n-1} \frac{(k-1)k}{2}\ \{*\} \\ &= 2\sum_{k=2}^{n} \frac{(k-1)k}{2} - \sum_{k=1}^{n-1} k = \sum_{k=2}^{n}(k-1)k - \frac{(n-1)n}{2} \\ &= \sum_{k=2}^{n} k^2 - \sum_{k=2}^{n} k - \frac{(n-1)n}{2} \end{aligned}$$

$$\begin{aligned}\overline{T}_V(n) &= \frac{1}{6}n(n+1)(2n+1) - 1 - \Big(\frac{(n-1)n}{2} - 1\Big) - \frac{(n-1)n}{2} \\ &= \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} - n^2 \;=\; \frac{n^3}{3} - \frac{n^2}{2} + \frac{n}{6}\end{aligned}$$

und

$$\overline{T}_{inv}(n) = T_{LU}(n) + \overline{T}_V(n) + nT_R(n) + 2n^2 = 2n^3 + n^2 + \frac{n}{3}. \tag{2.6}$$

Etwas einfacher stellen sich die Komplexitätsfunktionen dar, wenn man 1 Addition und 1 Multiplikation zu einer Grundoperation $\{\circ\}$ zusammenfasst und in den Betrachtungen nur die führende Ordnung nimmt.

Verfahren	$T_\circ(n)$ bez. $\{\circ\}$
$A = LU$	$n^3/3$
$Ly = b, Ux = y$	n^2
A^{-1}	n^3
$A = LU$ & $Ax = b$	$n^3/3$
$A = LU$ & A^{-1}	$4n^3/3$

Tab. 2.3
Führende Ordnung der Komplexität von Verfahren im Operationsmix $\{+,\ *\}$

In älterer Literatur wird der wesentliche Aufwand meist nach der Anzahl der Multiplikationen/Divisionen gerechnet. Damit ergeben sich

$$T^*_{LU}(n) = \frac{n^3}{3} - \frac{n}{3},\quad T^*_V(n) = \frac{n^2}{2} - \frac{n}{2},\quad T^*_R(n) = \frac{n^2}{2} + \frac{n}{2},$$

$$T^*_{LU}(n) + T^*_V(n) = \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6} = \frac{1}{6}n(n-1)(2n+5), \tag{2.7}$$

$$T^*(n) = T^*_{LU}(n) + T^*_V(n) + T^*_R(n) = \frac{n^3}{3} + n^2 - \frac{n}{3}. \tag{2.8}$$

Bei Berücksichtigung der Additionen/Subtraktionen hat man ungefähr eine Verdopplung des Aufwands, was im Vergleich mit (2.4) nur größenordnungsmäßig richtig ist.

2.2 Cholesky-Verfahren

Das Cholesky-Verfahren, auch Quadratwurzelverfahren genannt, für eine reelle symmetrische (positiv definite) Matrix basiert auf der R^TR- bzw. LL^T-Faktorisierung und läuft in folgenden Schritten ab.

- Ansatz, Faktorisierung und Darstellungsvarianten

$$A \;=\; R^TR,\ \ R = (r_{ij})\ \ \text{obere Dreiecksmatrix},$$

$$\begin{aligned} A &= LL^T, \quad L = (l_{ij}) \text{ untere Dreiecksmatrix,} \\ &= L'DL'^T, \quad D = \operatorname{diag}(d_1, d_2, ..., d_n),\ d_i = l_{ii},\ l'_{ii} = 1, \end{aligned} \tag{2.9}$$

$$L = \begin{pmatrix} l_{11} & 0 & 0 & \cdots & 0 \\ l_{21} & l_{22} & 0 & \cdots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & l_{nn} \end{pmatrix}, \quad L' = (l'_{ij}). \tag{2.10}$$

- Test auf Durchführbarkeit der Faktorisierung und damit auf Singularität bzw. Definitheit der Matrix mit einer gegebenen Toleranz $\varepsilon > 0$

- Lösung des LGS $Ax = b$ gemäß
  - $A = LL^T$ (LL^T-Faktorisierung),
  - $Ly = b, \quad R^Ty = b$ (Vorwärtselimination),
  - $L^Tx = y, \quad Rx = y$ (Rückwärtssubstitution),
  - komponentenweise Lösungsdarstellung mittels L,

$$y_i = \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right), \quad i = 1, 2, ..., n, \tag{2.11}$$

$$x_i = \frac{1}{l_{ii}} \left(y_i - \sum_{k=i+1}^{n} l_{ki} x_k \right), \quad i = 1, 2, ..., n. \tag{2.12}$$

- Die Berechnungsvorschriften für l_{ij} bzw. r_{ij} folgen aus

$$a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik} l_{jk} = \sum_{k=1}^{\min(i,j)} r_{ki} r_{kj}, \quad 1 \le i \le j \le n. \tag{2.13}$$

Das Cholesky-Verfahren ist für symmetrische, positiv definite Matrizen numerisch stabil, und es folgt aus der Beziehung (2.13) bei $i = j$ die Abschätzung $l_{ij}^2 \le a_{ii}$ für alle i, j. Die Elemente der Dreiecksmatrix L werden im Vergleich zu denen von A nicht groß.

Ist die Matrix nur symmetrisch, aber streng regulär, d. h., alle Hauptuntermatrizen

$$\tilde{A}_k = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \cdots & \cdots & \cdots & \cdots \\ a_{k1} & a_{k2} & \cdots & a_{kk} \end{pmatrix}, \quad 1 \le k \le n,$$

sind regulär, so ist das Cholesky-Verfahren im Komplexen durchführbar. Die Dreiecksmatrix L wird dann wegen negativer Radikanten rein imaginäre Diagonalelemente und damit imaginäre Spalten enthalten.

Einige Grundvarianten des Cholesky-Verfahrens

(1) Faktorisierung von A mittels L

$$
\begin{aligned}
i &= 1,2,\dots,n \\
l_{ii} &= \Big(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\Big)^{1/2}, \\
&\qquad \text{vorzeitiger Abbruch bei Radikand} < \varepsilon \\
l_{ji} &= \frac{1}{l_{ii}}\Big(a_{ij} - \sum_{k=1}^{i-1} l_{ik} l_{jk}\Big), \quad j = i+1, i+2, \dots, n,
\end{aligned}
$$

(2) Faktorisierung von $A = LL^T$ zeilenweise

Der erste Schritt im Zyklus wird extra behandelt.

$$
\begin{aligned}
l_{11} &= \sqrt{a_{11}} \\
i &= 2,3,\dots,n \\
l_{ij} &= \frac{1}{l_{jj}}\Big(a_{ji} - \sum_{k=1}^{j-1} l_{ik} l_{jk}\Big), \quad j = 1,2,\dots,i-1 \\
l_{ii} &= \Big(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\Big)^{1/2}
\end{aligned}
$$

(3) Faktorisierung von $A = LL^T$ spaltenweise

Der erste Schritt im Zyklus wird extra behandelt, weil dabei die Summe $\sum_{k=1}^{i-1}$ nicht auftritt.

$$
\begin{aligned}
l_{11} &= \sqrt{a_{11}} \\
l_{i1} &= a_{1i}/l_{11}, \quad i = 2,3,\dots,n \\
i &= 2,3,\dots,n \\
l_{ii} &= \Big(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\Big)^{1/2} \\
l_{ji} &= \frac{1}{l_{ii}}\Big(a_{ij} - \sum_{k=1}^{i-1} l_{jk} l_{ik}\Big), \quad j = i+1, i+2, \dots, n,
\end{aligned}
$$

(4) Faktorisierung von A am Platz

Wir überspeichern dabei von A das untere Dreieck (ohne Diagonale).
Das linke Dreieck von A und der Vektor p als Diagonale von L liefern die untere Dreiecksmatrix L.

$$
\begin{aligned}
i &= 1,2,...,n \\
\quad j &= i,i+1,...,n \\
\qquad h &= a_{ij} - \sum_{k=1}^{i-1} a_{ik}a_{jk}, \quad \text{vorzeitiger Abbruch bei} \quad h < \varepsilon \\
&\text{falls} \quad i = j, \text{ dann} \quad p_i = \sqrt{h}, \quad \text{sonst} \quad a_{ji} = h/p_i
\end{aligned}
$$

Effizienz des Cholesky-Verfahrens

Der arithmetische Hauptaufwand liegt in der LL^T-Faktorisierung der Matrix. Wir betrachten jeden der drei Schritte, also LL^T-Faktorisierung, Vorwärtselimination sowie Rückwärtssubstitution für sich und fassen schließlich die Strichoperationen $\{+,-\}$ bzw. Punktoperationen $\{*,\ /\} = \{\cdot,\ :\}$ jeweils zusammen.

$A = LL^T$:
In Variante (3) verursacht die Anweisung

$$l_{ji} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} l_{jk} l_{ik} \right)$$

im Inneren aller Schleifen mit $i-1$ Additionen, $i-1$ Multiplikationen und einer Division den Hauptaufwand.
Wir haben in der i-Schleife diese Vorschrift $n-i$ Mal zu erzeugen, das sind also jeweils $(n-i)(i-1)$ Additionen und Multiplikationen sowie $n-i$ Divisionen. Dazu kommt die Berechnung des Diagonalelements mit $i-1$ Additionen und $i-1$ Multiplikationen. Weiterhin bleibt eine kleine Vorabrechnung und die n Quadratwurzeln. Damit ergibt sich der Gesamtaufwand

$$
\begin{aligned}
T_{LL^T}(n) &= n \ \{\sqrt{\ }\} \ + \sum_{i=1}^{n-1} i \ \{/\} \ + \sum_{i=1}^{n-1} (n-i)i \ \{+,*\} \\
&= n \ \{\sqrt{\ }\} \ + \frac{(n-1)n}{2} \ \{/\} \ + \left(n \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2 \right) \ \{+,*\} \\
&= n \ \{\sqrt{\ }\} \ + \frac{(n-1)n}{2} \ \{/\} \ + n\frac{(n-1)n}{2} - \frac{(n-1)n(2n-1)}{6} \ \{+,*\} \\
&= n \ \{\sqrt{\ }\} \ + \frac{n^2}{2} - \frac{n}{2} \ \{/\} \ + \frac{n^3}{6} - \frac{n}{6} \ \{+,*\},
\end{aligned}
$$

$$\begin{aligned} T_{LL^T}(n) &= n \;\{\sqrt{}\} \;\; + \frac{n^3}{6} + \frac{n^2}{2} - \frac{2n}{3} \;\{*\} \;\; + \frac{n^3}{6} - \frac{n}{6} \;\{+\} \\ &= n \;\{\sqrt{}\} \;\; + \frac{n^3}{3} + \frac{n^2}{2} - \frac{5n}{6}, \end{aligned}$$

$Ly = b$:

$$\begin{aligned} T_V(n) &= \sum_{k=1}^{n-1} k \;\{+,*\} \;\; + n \;\{/\} \\ &= \frac{(n-1)n}{2} \;\{+\} \;\; + \frac{(n+1)n}{2} \;\{*\} \\ &= \frac{n^2}{2} - \frac{n}{2} \;\{+\} \;\; + \frac{n^2}{2} + \frac{n}{2} \;\{*\} \\ &= n^2, \end{aligned}$$

$L^T x = y$:

$$\begin{aligned} T_R(n) &= \sum_{k=1}^{n-1} k \;\{+,*\} \;\; + n \;\{/\} \\ &= \frac{(n-1)n}{2} \;\{+\} \;\; + \frac{(n+1)n}{2} \;\{*\} \\ &= \frac{n^2}{2} - \frac{n}{2} \;\{+\} \;\; + \frac{n^2}{2} + \frac{n}{2} \;\{*\} \\ &= n^2. \end{aligned}$$

Damit beträgt der Aufwand zur Lösung des LGS bei gegebenem $A = LL^T$

$$T_V(n) + T_R(n) = 2n^2$$

sowie der arithmetische Gesamtaufwand

$$T_{chol}(n) = T_{LL^T}(n) + T_V(n) + T_R(n) = n \;\{\sqrt{}\} \;\; + \frac{n^3}{3} + \frac{5n^2}{2} - \frac{5n}{6}. \qquad (2.14)$$

Betrachtet man wiederum nur die Größenordnung und fasst 1 Addition und 1 Multiplikation zu einer Grundoperation $\{\circ\}$ zusammen, dann ist die Komplexitätsfunktionen

$$T_\circ(n) = \frac{n^3}{6} + \mathcal{O}(n^2),$$

so dass etwa nur halb so viel Aufwand entsteht wie beim GA.

Achtung: MATLAB zählt bei der Ausführung der Wurzelfunktion `sqrt()` mit einem einfachen Argument wie auch bei der Berechnung anderer transzendenter Funktionen 1 *flop*, aber für die (quadratische) Potenz gemäß `a^b` werden 2 *flops* gebraucht.

2.3 Direkte Verfahren für Bandmatrizen

(1) Gauß-Algorithmus

Wenn die Ausgangsmatrix Bandstruktur mit einer Bandbreite $\alpha+\beta+1$ hat, kann man den Rechenaufwand und den Speicherbedarf des GA reduzieren. Es zeigt sich nämlich, dass sich die Bandstruktur von A auf die Dreiecksmatrizen L und U überträgt.
Ist A streng regulär, so kann der GA ohne Pivotisierung durchgeführt werden und L hat die Bandbreite $\alpha+1$ und U die Bandbreite $\beta+1$.

$$\begin{pmatrix} + & * & * & & \\ * & + & * & * & \\ & * & + & * & * \\ & & * & + & * \\ & & & * & + \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ * & 1 & & & \\ & * & 1 & & \\ & & * & 1 & \\ & & & * & 1 \end{pmatrix} \begin{pmatrix} + & * & * & & \\ & + & * & * & \\ & & + & * & * \\ & & & + & * \\ & & & & + \end{pmatrix}$$

Man kann den GA an die Situation anpassen, indem man die Indexgrenzen der Bandbreite entsprechen abändert. Auf diese Weise wird verhindert, dass der Algorithmus überflüssige Operationen mit NE auführt.

Resttableau-Algorithmus für Bandstruktur.

$A = LU,\ l_{ii} = 1.$

$$\begin{array}{lll}
k & = & 1, 2, ..., n \\
\quad p & = & a_{kk}, \\
 & & \quad \text{vorzeitiger Abbruch bei } \ p = 0 \\
\quad i & = & k+1, k+2, ..., \min(k+\alpha, n) \\
 & & s \quad = a_{ik}/p \\
 & & a_{ik} = s \\
 & & a_{ij} = a_{ij} - s\, a_{kj}, \quad j = k+1, k+2, ..., \min(k+\beta, n)
\end{array}$$

Steht nun die LU-Faktorisierung der Bandmatrix A zur Verfügung, so kann man das LGS $Ax = LUx = b$ durch die gestaffelten LGS

$$Ly = b \ \text{ und } \ Ux = y$$

mit Banddreiecksmatrizen der Reihe nach lösen.

Vorwärtselimination der rechten Seite als Lösung von $Ly = b$

$$
\begin{aligned}
y_1 &= b_1 \\
k &= 2, 3, ..., n \\
&\quad h = b_k \\
&\quad h = h - l_{ki}\, y_i, \quad i = \max(1, k+1-\alpha), ..., k-2, k-1 \\
&\quad y_k = h
\end{aligned}
$$

Rückwärtssubstitution als Lösung von $Ux = y$

$$
\begin{aligned}
x_n &= y_n / u_{nn} \\
k &= n-1, n-2, ..., 1 \\
&\quad h = b_k \\
&\quad h = h - u_{ki}\, x_i, \quad i = k+1, k+2, ..., \min(k+\beta, n) \\
&\quad x_k = h / u_{kk}
\end{aligned}
$$

Speichert man das Band der Matrix in ein Rechteckfeld um, so muss der GA wegen der geänderten Indizierung der Elemente neu geschrieben werden.

Muss dagegen pivotisiert werden, so stehen bei Spaltenpivotisierung mit Zeilenvertauschung unterhalb des Diagonalelements jeweils α Pivotkandidaten zur Auswahl. Im ungünstigsten Fall wird im k-ten Schritt die unterste Zeile $k + \alpha$ mit Zeile k getauscht. Maximal verbreitet sich das Band von U also um α Kodiagonalen auf die Gesamtbreite $\alpha + \beta + 1$.

$$
P \begin{pmatrix} + & * & * & & \\ * & + & * & * & \\ & * & + & * & * \\ & & * & + & * \\ & & & * & + \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ * & 1 & & & \\ & * & 1 & & \\ & & * & 1 & \\ & & & * & 1 \end{pmatrix} \begin{pmatrix} + & * & * & \Box & \\ & + & * & * & \Box \\ & & + & * & * \\ & & & + & * \\ & & & & + \end{pmatrix}
$$

Komplexität der LU-Faktorisierung

A habe die Bandbreite $\alpha + \beta + 1$.

Der Rechenaufwand an Additionen und Multiplikationen ergibt sich wie folgt.

(1) Im ersten Schritt mit der 1. Zeile durch die Elimination der Subdiagonalelemente der 1. Spalte: $\alpha\beta$ $\{+, *\}$

(2) Entsprechendes gilt für die Zeilen 2 bis $(n - \alpha)$.

(3) Nach Durchführung der Schritte (1) und (2) bleibt eine vollbesetzte $(\alpha \times \alpha)$-Matrix übrig, welche trianguliert werden muss.

Insgesamt erfordern die Schritte (1), (2) und (3) einen Gesamtaufwand von

$$T_B(n) = (n - \alpha)\alpha\beta + \frac{1}{6}(\alpha - 1)\alpha(2\alpha - 1) \approx n\alpha\beta - \alpha^2\beta + \frac{\alpha^3}{3}$$

Additionen und Multiplikationen.

Für $n \gg \alpha, \beta$ ist dies in führender Ordnung gleich $n\alpha\beta$.

In Beispielen veranschaulichen wir noch den Aufwand bei anderen Besetzungsstrukturen von A.

Beispiel 2.1

(a) A habe die Bandbreite $2\alpha + 1$, d. h. $\beta = \alpha$. Dann gilt insbesondere

$$T_B(n) = (n - \alpha)\alpha^2 + \frac{1}{6}(\alpha - 1)\alpha(2\alpha - 1) \approx n\alpha^2 - \frac{2}{3}\alpha^3.$$

(b) Völlig anders sind die Verhältnisse bei der Matrix

$$\begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix}.$$

Diese Matrix hat lediglich ca. $3n$ NNE. Hier füllt sich bei der Elimination der ersten Spalte der Rest der Matrix auf, und der Gesamtaufwand hat wieder die Ordnung $\mathcal{O}(n^3)$. Der GA ist hier also nicht in der Lage, dünn besetzte Strukturen auszunutzen, um Rechenzeit und Speicheraufwand zu sparen. Das Ersetzen von NE durch NNE wird als **Fill-in** bezeichnet.

(c) Die Lage ändert sich wiederum völlig, wenn A folgendermaßen besetzt ist.

$$\begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix}$$

Man sieht schnell, dass hier die Besetzungsstruktur erhalten bleibt.

(2) Cholesky-Verfahren

Wenn die symmetrische Ausgangsmatrix Bandstruktur mit einer Bandbreite $2\alpha+1$ hat, kann man den Rechenaufwand und den Speicherbedarf des Verfahrens reduzieren. Es zeigt sich nämlich, dass sich die Bandstruktur von A auf die Dreiecksmatrizen L und L^T überträgt. Ist A spd, so kann die Cholesky-Faktorisierung im Reellen durchgeführt werden und L hat die Bandbreite $\alpha+1$.
Wir nehmen die Variante (3) der Faktorisierung von $A = LL^T$ spaltenweise.

$$
\begin{aligned}
l_{11} &= \sqrt{a_{11}} \\
l_{i1} &= a_{1i}/l_{11}, \quad i = 2,3,...,\alpha+1 \\
i &= 2,3,...,n \\
&\quad l_{ii} = \left(a_{ii} - \sum_{k=\max(1,i-\alpha)}^{i-1} l_{ik}^2 \right)^{1/2} \\
&\quad l_{ji} = \frac{1}{l_{ii}} \left(a_{ij} - \sum_{k=\max(1,j-\alpha)}^{i-1} l_{jk} l_{ik} \right), \quad j = i+1, i+2, ..., \min(i+\alpha, n),
\end{aligned}
$$

Der Algorithmus funktioniert formal für alle $\alpha \geq 0$.
Im Gesamtaufwand bleiben auf jeden Fall die Berechnung der n Quadratwurzeln. Dazu kommt für $\alpha \ll n$ in jedem Schritt i die Berechnung von α Spaltenelementen mit jeweils sich verkürzenden Skalarprodukten $\sum_{k=j-\alpha}^{i-1} l_{jk} l_{ik}$, $j = i, i+1, ..., i+\alpha-1$.
Dabei werden jeweils $\alpha(\alpha+1)/2$ Additionen und Multiplikationen benötigt, dazu noch α Divisionen.
Damit ergibt sich der Gesamtaufwand

$$
T(n) \approx n \ \{\sqrt{\ }\} \ + n[\alpha(\alpha+1)+\alpha] = n \ \{\sqrt{\ }\} \ + n\alpha(\alpha+2).
$$

Für eine Tridiagonalmatrix mit $\alpha = 1$ erhält man $T(n) \approx n \ \{\sqrt{\ }\} \ + 3n$.
Die Cholesky-Faktorisierung kann man für $\alpha = 1$ mit Hilfe von Vektoren notieren. Sei $\tilde{l} = (\tilde{l}_1, \tilde{l}_2, ..., \tilde{l}_n)$ die Diagonale und $l = (l_2, l_3, ..., l_n)$ die untere Nebendiagonale der Matrix L. Dann sind die sich vereinfachenden Formeln

$$
\begin{aligned}
\tilde{l}_1 &= \sqrt{a_{11}}, \\
l_i &= \frac{a_{i-1,i}}{\tilde{l}_{i-1}}, \quad \tilde{l}_i = \sqrt{a_{ii} - l_i^2}, \quad i = 2,3,...,n.
\end{aligned}
$$

2.4 Tridiagonalmatrizen

Gegeben sei das LGS $Ax = d$ mit der tridiagonalen Koeffizientenmatrix

$$A = \begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{n-1} & b_{n-1} & c_{n-1} \\ 0 & \dots & 0 & 0 & a_n & b_n \end{pmatrix} \quad \text{und} \quad d = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}.$$

Durch Elimination von $a_2, a_3, ..., a_n$ in A kann man die Matrix in eine obere Dreiecksgestalt überführen und gleichzeitig die rechte Seite transformieren.
Wir wollen aber unter Berücksichtigung der Bandstruktur zunächst die LU-Faktorisierung ohne Pivotstrategie machen und dann die gestaffelten LGS lösen.
Es bietet sich der Ansatz
$A = LU$,

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ & \cdot & \cdot & & \\ & & \cdot & 1 & \\ & & & l_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & & & \\ & u_{22} & u_{23} & & \\ & & \cdot & \cdot & \\ & & & \cdot & u_{n-1,n} \\ & & & & u_{nn} \end{pmatrix}$$

Durch Ausmultiplizieren und Koeffizientenvergleich bestätigt man zunächst, dass $u_{i,i+1} = c_i,\; i = 1, 2, ..., n-1$, ist.
Damit gehen wir auch in den Matrizen L und U zu einer vektoriellen Schreibweise über.

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ l_2 & 1 & & & \\ & \cdot & \cdot & & \\ & & \cdot & 1 & \\ & & & l_{n-1} & 1 \end{pmatrix} \begin{pmatrix} u_1 & c_1 & & & \\ & u_2 & c_2 & & \\ & & \cdot & \cdot & \\ & & & \cdot & c_{n-1} \\ & & & & u_n \end{pmatrix}$$

Somit erhält man die Grundvariante des verkürzten GA für Tridiagonalsysteme.

LU-**Faktorisierung** von $A = LU$

$$\begin{aligned} u_1 &= b_1 \\ k &= 2, 3, ..., n \\ & \quad \text{falls } u_{k-1} = 0, \text{ dann Abbruch mit Info} \\ & \quad l_k = a_k / u_{k-1} \\ & \quad u_k = b_k - l_k c_{k-1} \end{aligned}$$

Vorwärtselimination der rechten Seite als Lösung von $Ly = d$

$$
\begin{aligned}
y_1 &= d_1 \\
y_k &= d_k - l_k y_{k-1}, \quad k = 2, 3, ..., n
\end{aligned}
$$

Rückwärtssubstitution als Lösung von $Ux = y$

$$
\begin{aligned}
x_n &= y_n / u_n \\
x_k &= (y_k - c_k x_{k+1}) / u_k, \quad k = n-1, n-2, ..., 1
\end{aligned}
$$

Effizienz des verkürzten Gauß-Algorithmus

Der arithmetische Aufwand liegt in der Verarbeitung und Erzeugung von Vektoren. Nimmt man alle Operationen, so ist

$$
\begin{aligned}
T_{tri}(n) &= T_{LU}(n) + T_V(n) + T_R(n) \\
&= 3(n-1) + 2(n-1) + [3(n-1) + 1] \\
&\approx 3n + 5n \\
&= 8n.
\end{aligned}
$$

Programmtechnisch kann man die Elemente der Vektoren l und u auf den Vektoren a und b ablegen und für den Hilfsvektor wir der Platz d benutzt, so dass man insgesamt mit 4 Vektoren auskommt.
Die Inverse zu einer Bandmatrix hat leider i. Allg. keine Bandgestalt. Aber sie ist mit ca. $3n + 5n^2$ Operationen berechenbar.

Es sollen noch zwei andere Varianten des verkürzten GA vorgestellt werden, bei denen die Eigenschaften und Überprüfung der Durchführbarkeit gut zu erkennen sind.

(1) Variante mit rekursivem Ansatz

Das LGS notieren wir zeilenweise

$$
\begin{aligned}
b_1 x_1 + c_1 x_2 &= d_1, \\
a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= d_i, \ i = 2, 3, ..., n-1, \\
a_n x_{n-1} + b_n x_n &= d_n.
\end{aligned}
$$

Man erwartet eine Lösung des LGS, die der rekursiven Beziehung

$$x_i = \alpha_i x_{i+1} + \beta_i, \quad i = n-1, n-2, ..., 1,$$

genügt, so dass nun die Koeffizienten α_i und β_i sowie die Startbedingung x_n zu bestimmen sind.

Setzt man diesen Ansatz in die allgemeine Gleichung ein, erhält man

$$\begin{aligned} a_i(\alpha_{i-1}x_i + \beta_{i-1}) + b_i x_i + c_i(x_i - \beta_i)/\alpha_i &= d_i \\ (a_i\alpha_{i-1} + b_i + c_i/\alpha_i)x_i + (a_i\beta_{i-1} - c_i\beta_i/\alpha_i - d_i) &= 0 \end{aligned}$$

$$\begin{aligned} a_i\alpha_{i-1} + b_i + c_i/\alpha_i &= 0 \\ a_i\beta_{i-1} - c_i\beta_i/\alpha_i - d_i &= 0. \end{aligned}$$

Daraus folgen die Rekursionsformeln

$$\begin{aligned} \alpha_i &= -\frac{c_i}{b_i + a_i\alpha_{i-1}}, \\ \beta_i &= \frac{a_i\beta_{i-1} - d_i}{c_i}\alpha_i = \frac{d_i - a_i\beta_{i-1}}{b_i + a_i\alpha_{i-1}}. \end{aligned}$$

Aus

$$\begin{aligned} b_1 x_1 + c_1 x_2 &= d_1, \\ x_1 - \alpha_1 x_2 &= \beta_1 \end{aligned}$$

ergibt sich $\alpha_1 = -\frac{c_1}{b_1}$ und $\beta_1 = \frac{d_1}{b_1}$, so dass man $\alpha_0 = \beta_0 = 0$ setzen kann.
Aus dem kleinen LGS

$$\begin{aligned} a_n x_{n-1} + b_n x_n &= d_n, \\ x_{n-1} - \alpha_{n-1} x_n &= \beta_{n-1} \end{aligned}$$

folgt

$$x_n = \frac{d_n - a_n\beta_{n-1}}{b_n + a_n\alpha_{n-1}},$$

was die gleiche Darstellung wie β_n bedeutet.

Zusammenfassung

$$\begin{aligned} \alpha_0 &= \beta_0 = 0 \\ i &= 1, 2, ..., n-1 \\ &\quad \alpha_i = -\frac{c_i}{h} \quad \text{mit} \quad h = b_i + a_i\alpha_{i-1} \\ &\quad \beta_i = \frac{d_i - a_i\beta_{i-1}}{h} \\ x_n &= \frac{d_n - a_n\beta_{n-1}}{b_n + a_n\alpha_{n-1}} \\ x_i &= \alpha_i x_{i+1} + \beta_i, \quad i = n-1, n-2, ..., 1 \end{aligned}$$

Der Algorithmus wird auch Schießverfahren (engl. chase method, russ. metod progonka) genannt.
Der arithmetische Aufwand beträgt auch hier ca. $8n$ Operationen.
Die Durchführbarkeit ist garantiert, wenn in allen Schritten die Division durch $b_i + a_i\alpha_{i-1}$ machbar ist. Dazu reicht aus, dass $|\alpha_{i-1}| < 1$ und $|b_i| \geq |a_i|$ sind.
Ein hinreichende Bedingung für eine stabile Berechnung ist

$$|b_i| \geq |a_i| + |c_i|, \;\; |b_1| > |c_1| \text{ oder } |b_n > |a_n|,$$

d. h. die Diagonale der Matrix A ist etwas mehr als schwach diagonaldominant.
Genauere Betrachtungen zur Stabilität sind in [21].
Diese Forderungen sind z. B. bei LGS, die aus Diskretisierungsverfahren für Differentialgleichungen entstehen, erfüllt.

Beispiel 2.2 $A = A^T = \text{tridiag}(-1, 2, -1) > 0$, irreduzibel diagonaldominant

LU-Faktorisierung

$$\begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & -1 \\ & & & & -1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & & & & & \\ -\frac{1}{2} & 1 & & & & \\ & -\frac{2}{3} & 1 & & & \\ & & \cdot & \cdot & & \\ & & & \cdot & 1 & \\ & & & & -\frac{n-1}{n} & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & & & & \\ & \frac{3}{2} & -1 & & & \\ & & \frac{4}{3} & -1 & & \\ & & & \cdot & \cdot & \\ & & & & \cdot & -1 \\ & & & & & \frac{n+1}{n} \end{pmatrix}.$$

Cholesky-Faktorisierung

$$\begin{aligned} A &= LU = LDL^T, \quad D = \text{diag}(2, \tfrac{3}{2}, \tfrac{4}{3}, ..., \tfrac{n+1}{n}), \\ &= LD^{1/2}D^{1/2}L^T = LD^{1/2}(D^{1/2})^T L^T = (LD^{1/2})(LD^{1/2})^T \\ &= \widetilde{L}\widetilde{L}^T, \quad \widetilde{L} = LD^{1/2}, \end{aligned}$$

$$\text{mit } \widetilde{L} = \begin{pmatrix} \sqrt{2} & & & & & & \\ -\sqrt{\frac{1}{2}} & \sqrt{\frac{3}{2}} & & & & & \\ & -\sqrt{\frac{2}{3}} & \sqrt{\frac{4}{3}} & & & & \\ & & -\sqrt{\frac{3}{4}} & \sqrt{\frac{5}{4}} & & & \\ & & & \cdot & \cdot & & \\ & & & & \cdot & \sqrt{\frac{n}{n-1}} & \\ & & & & & -\sqrt{\frac{n-1}{n}} & \sqrt{\frac{n+1}{n}} \end{pmatrix}.$$

Beide Dreiecksmatrizen sind jeweils diagonaldominant und $\det(A) = \prod_{i=1}^{n} u_{ii} = n + 1$.

(2) Schießverfahren (andere Variante)

Sind in der Matrix nicht die Diagonalelemente, sondern die Elemente c_i in der oberen Kodiagonalen dominant, so verwendet man diese als Pivotelemente und stellt die Gleichungen des Systems anders um.
Das LGS notieren wir erneut zeilenweise

$$\begin{aligned} b_1x_1 + c_1x_2 &= d_1, \\ a_ix_{i-1} + b_ix_i + c_ix_{i+1} &= d_i, \; i = 2, 3, ..., n-1, \\ a_nx_{n-1} + b_nx_n &= d_n. \end{aligned}$$

Wir schreiben x_2 als lineare Funktion von x_1 gemäß

$$x_2 = (d_1 - b_1x_1)/c_1 = y_1 + z_1x_1$$

mit den Koeffizienten $y_1 = d_1/c_1, \quad z_1 = -b_1/c_1$.
Stellt man die zweite Gleichung nach x_3 um und setzt den Ausdruck für x_2 ein, so ergibt sich

$$x_3 = (d_2 - a_2x_1 - b_2x_2)/c_2 = [d_2 - a_2x_1 - b_2(y_1 + z_1x_1)]/c_2 = y_2 + z_2x_1$$

mit $y_2 = (d_2 - b_2y_1)/c_2, \quad z_2 = -(a_2 + b_2z_1)/c_2$.
Aus den nächsten Gleichungen folgt analog für $i = 3, 4, ..., n-1$

$$\begin{aligned} x_{i+1} &= (d_i - a_ix_{i-1} - b_ix_i)/c_i \\ &= [d_i - a_i(y_{i-2} + z_{i-2}x_1) - b_i(y_{i-1} + z_{i-1}x_1)]/c_i \\ &= y_i + z_ix_1 \end{aligned}$$

mit

$$y_i = (d_i - a_iy_{i-2} - b_iy_{i-1})/c_i, \quad z_i = -(a_iz_{i-2} + b_iz_{i-1})/c_i.$$

Setzt man $x_{n-1} = y_{n-2} + z_{n-2}x_1$ und $x_n = y_{n-1} + z_{n-1}x_1$ in die n-te Gleichung des LGS ein, so bleibt nur x_1 als Unbekannte, und man erhält

$$\begin{aligned} a_nx_{n-1} + b_nx_n &= d_n, \\ a_n(y_{n-2} + z_{n-2}x_1) + b_n(y_{n-1} + z_{n-1}x_1) &= d_n, \\ x_1 &= \frac{d_n - a_ny_{n-2} - b_ny_{n-1}}{a_nz_{n-2} + b_nz_{n-1}}. \end{aligned}$$

Damit können die Komponenten $x_2, x_3, ..., x_n$ mittels $x_{i+1} = y_i + z_ix_1$ bestimmt werden.
Der arithmetische Aufwand beträgt hier ca. $11n$ Operationen.

2.5 Numerische und symbolische direkte Lösung von LGS mit CAS

In die Betrachtungen beziehen wir hauptsächlich das CAS MATLAB ein, wobei manchmal Maple zum Vergleich angeführt wird.
Wir geben zunächst die verwendeten LGS bzw. Matrizen an.

```
% Definitionen mit built-in-Anweisungen/Funktionen
A = hilb(3)         % Hilbert-Matrix
b = sum(A')'        % Zeilensummen von A als Spaltenvektor
xexakt = [1 1 1]';  % exakte Loesung
AE =[A b]           % erweiterte Matrix

A =
     1.0000    0.5000    0.3333
     0.5000    0.3333    0.2500
     0.3333    0.2500    0.2000
b =
     1.8333
     1.0833
     0.7833

AE =
     1.0000    0.5000    0.3333    1.8333
     0.5000    0.3333    0.2500    1.0833
     0.3333    0.2500    0.2000    0.7833

% komponentenweise Definition der Matrix
for i = 1:3, for j = 1:3, AA(i,j) = i+j-2; end; end;
AA(3,3) = 5;
AA
AA =
     0    1    2
     1    2    3
     2    3    5

% symmetrische positiv definite streng diagonaldominante Matrix
A1 = [ 10  -1   2   0
       -1  11  -1   3
        2  -1  10  -1
        0   3  -1   8 ];

b1 = [  6  25 -11  15 ]';
```

Die verschiedenen Funktionen zu Lösung von LGS in MATLAB basieren meist auf der Gauß-Elimination. Dabei werden Pivotstrategien berücksichtigt, die zu Zeilenvertauschungen führen. Letzteres kann durch Ergebnisparameter wie Ausgabe der Permutationmatrix abgefragt werden. Für spezielle Koeffizientenmatrizen, wie z. B. symmetrische oder spd, gibt es natürlich spezielle Algorithmen.
Rundungseffekte in der Gleitkommaarithmetik der numerische Rechnung in MATLAB führen anders als bei symbolischen Umformungen in MATLAB oder auch Maple zu abweichenden Zeilenvertauschungen und/oder Faktorisierungen der Matrix.
Zunächst diskutieren wir einige Lösungsmöglichkeiten.

1. Row reduce Variante auf Gauß-Jordan-Form

Die built-in-Funktion `rref` angewendet nur auf die Koeffizientenmatrix überführt diese in die Gauß-Jordan-Form und kann zu ihrer Rangbestimmung genutzt werden. Ist die Koeffizientenmatrix regulär, so wird diese transformiert auf die Einheitsmatrix. Eventuell notwendige Zeilenvertauschungen, geschweige denn Faktorisierungen, sind aus der Darstellung jedoch nicht ersichtlich.

```
rref(AA)

ans =
     1     0     0
     0     1     0
     0     0     1
```

Anwendung für Lösung eines LGS

```
rref(AE)

ans =
     1.0000         0         0    1.0000
          0    1.0000         0    1.0000
          0         0    1.0000    1.0000

x = ans(1:3,4)

x =
     1.0000
     1.0000
     1.0000

rref([AA [8 14 23]'])     % mit Zeilenvertauschung wegen AA(1,1)=0
```

```
ans =
     1     0     0     1
     0     1     0     2
     0     0     1     3

ans(:,4)

ans =
     1
     2
     3
```

2. Symbolische Lösung

Dazu gibt es das in MATLAB involvierte Maple-Kommado `linsolve`.
Die Lösbarkeit des LGS wird geprüft. Mögliche Nachrichten, Warnungen bzw. Fehlermeldungen sind

Warning: Matrix is rank deficient; solution does not exist.
Warning: Matrix is rank deficient; solution is not unique.

```
x = linsolve(A,b)    % Ergebnis in rationaler Form, da
                     % Hilbert-Matrix A mit gerundeten Elementen
numeric(x)

x =
  [281474976710655/281474976710656]
  [140737488355331/140737488355328]
  [140737488355325/140737488355328]

ans =
     1.00000000000000
     1.00000000000002
     0.99999999999998

x = linsolve(AA,[8 14 23]')

x =
  [1]
  [2]
  [3]
```

3. LU-Faktorisierung

Die LU-Faktorisierung ohne Zeilenvertauschung liefert $A = LU$ mit den entsprechen Dreiecksmatrizen L, $l_{ii} = 1$, und U. Zeilenvertauschungen werden entweder in der unteren Dreiecksmatrix L einbezogen oder als Permutationsmatrix in Form eines zusätzlichen Ergebnisparameters angegeben.

`[L,U] = lu(A)`	$A = LU$,	in L stehen die untere Dreiecksmatrix und Permutationsmatrix, U obere Dreiecksmatrix
`[L,U,P] = lu(A)`	$PA = LU$,	L, U untere bzw. obere Dreiecksmatrix, P Permutationsmatrix

`lu(A)` alleine ergibt das Tableau $C\backslash B$, welches beim VGA entsteht, so dass $A = -CB = LU$ ist (output from LINPACK'S ZGEFA routine).

Nehmen wir zunächst den einfachen Fall einer spd streng diagonaldominanten Matrix.

```
A1
A1 =
     10    -1     2     0
     -1    11    -1     3
      2    -1    10    -1
      0     3    -1     8

% Tableau wie beim VGA
lu(A1)
ans =
     10.0000   -1.0000    2.0000         0
      0.1000   10.9000   -0.8000    3.0000
     -0.2000    0.0734    9.5413   -0.7798
           0   -0.2752    0.0817    7.1106

[L U P] = lu(A1)
L =
      1.0000         0         0         0
     -0.1000    1.0000         0         0
      0.2000   -0.0734    1.0000         0
           0    0.2752   -0.0817    1.0000

U =
     10.0000   -1.0000    2.0000         0
           0   10.9000   -0.8000    3.0000
           0         0    9.5413   -0.7798
           0         0         0    7.1106
```

```
P =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

[L U] = lu(A1)  % analog ohne Ausgabe von P
```

Die Lösung des LGS kann dann mittels Vorwärts- und Rückwärtsrechnung (Vorwärtselimination und Rückwärtssubstitution) unter Verwendung des Array-Operators \ erhalten werden.

```
y = L\b1
x = U\y
x = U\(L\b1)    % Achtung: U\L\b1 = (U\L)\b1 <> U\(L\b1)
```

Das zweite Beispiel sei ein LGS mit der spd Hilbert-Matrix $H(3)$.

```
A =
     1.0000    0.5000    0.3333
     0.5000    0.3333    0.2500
     0.3333    0.2500    0.2000
```

Die symbolische Rechnung und Faktorisierung in Maple läuft ohne Zeilenvertauschung ab und ergibt die erwartete Faktorisierung.

```
A = [ 1    0   0                 # Maple
      1/2  1   0
      1/3  1   1]*[1  1/2   1/3
                   0  1/12  1/12
                   0  0     1/180]
```

MATLAB hingegen führt im 2. Schritt eine Vertauschung von 2. und 3. Zeile durch.

```
format rat      % display rational approximations
lu(A)

ans =
      1           1/2          1/3
    -1/3          1/12         4/45
    -1/2          -1          -1/180
```

```
[L U] = lu(A)   % Zeilenvertauschung zu erkennen

L =
       1            0            0
      1/2           1            1
      1/3           1            0

U =
       1           1/2          1/3
       0           1/12         4/45
       0            0          -1/180

x = (U\(L\b))'

x =
       1     1     1
```

Pivotstrategie: im 2. Schritt beim Vergleich von 1/12=0.0833333 in der 2. Zeile mit 1/12 darunter wird die 3. Zeile bevorzugt.
Zur Kontrolle noch einmal mit der Permutationsmatrix.

```
[L,U,P] = lu(A)   % P*A=L*U, U wie oben

L =
       1            0            0
      1/3           1            0
      1/2           1            1
P =
       1            0            0
       0            0            1
       0            1            0
```

Gehen wir zum symbolischen Rechnen über.

```
AS = sym(A)

AS =
  [  1, 1/2, 1/3]
  [1/2, 1/3, 1/4]
  [1/3, 1/4, 1/5]

% Definition mit Elementen
AS = sym('[1,1/2,1/3;1/2,1/3,1/4;1/3,1/4,1/5]')
```

```
AS =
  [  1, 1/2, 1/3]
  [1/2, 1/3, 1/4]
  [1/3, 1/4, 1/5]

inverse(AS)   % Matrix symbolisch invertieren

ans =
  [  9,  -36,   30]
  [-36,  192, -180]
  [ 30, -180,  180]
```

Die Funktion *lu* ist nicht anwendbar auf die symbolische Matrix AS.
Fehlermeldung: *Error using ⇒ lu*
Matrix must be square.

Um die Zeilenvertauschung bei der Faktorisierung in MATLAB zu unterbinden, vergrößern wir das Element $A(2,2)$ etwas gemäß $A(2,2) = A(2,2)+1e{-}16$. Wir erhalten dann ein mit Maple vergleichbares Ergebnis.

```
A =
      1            1/2           1/3
     1/2           1/3           1/4
     1/3           1/4           1/5

[L,U] = lu(A)    % P=I

L =
      1             0             0
     1/2            1             0
     1/3            1             1

U =
      1            1/2           1/3
      0            1/12          1/12
      0             0            1/180
```

Zum Abschluss noch die Faktorisierung der Matrix AA, wobei zu sehen ist, dass die Pivotstrategie mit Zeilenvertauschung sich jeweils das betragsgrößte Element der Spalte sucht.

```
[L,U,P] = lu(AA)  % P*A=L*U
L =
       1            0            0
       0            1            0
      1/2          1/2           1
U =
       2            3            5
       0            1            2
       0            0          -1/2
P =
       0            0            1
       1            0            0
       0            1            0
```

4. Cholesky-Faktorisierung

Das ist die symmetrische Faktorisierung $R^T R$, R obere rechte Dreiecksmatrix, der spd Matrix A. Natürlich ist eine Faktorisierung im Komplexen auch denkbar. Dabei entstehen Zeilen mit rein imaginären Elementen.
Weiterhin soll hier ein Hinweis auf das Darstellungsformat `format rat` erfolgen. Reelle Zahlen werden dabei durch Brüche (Zähler und Nenner ganzzahlig) mit einer Genauigkeit von 6-7 Mantissenstellen approximiert und repräsentiert.
Wir verwenden die Cholesky-Faktorisierung für die spd Hilbert-Matrix

$$H(3) = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{pmatrix},$$

was bei exakter Rechnung die obere Dreiecksmatrix liefert.

$$R = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ 0 & \frac{\sqrt{3}}{6} & \frac{\sqrt{3}}{6} \\ 0 & 0 & \frac{\sqrt{5}}{30} \end{pmatrix} \approx \begin{pmatrix} 1.00000000000000 & 0.50000000000000 & 0.33333333333333 \\ 0 & 0.28867513459481 & 0.28867513459481 \\ 0 & 0 & 0.07453559924999 \end{pmatrix}$$

```
format long
R = chol(A)
R =
     1.00000000000000    0.50000000000000    0.33333333333333
                    0    0.28867513459481    0.28867513459481
                    0                   0    0.07453559924999
```

```
x = (R\(R'\b))'
x =
     1     1     1

format rat
R1 = chol(A)
R1 =
     1           1/2          1/3
     0         390/1351     390/1351
     0            0         317/4253

R1-R   % ist Nullmatrix im Format rat
L = chol(A)';            % L untere Dreiecksmatrix
```

Man bedenke, dass $R1$ nur die Bruchdarstellung von R ist. Würde man die approximierenden Brüche zur Definition einer neuen Matrix $R2$ verwenden, so wird der Darstellungsfehler von $\approx 1e-6$ sichtbar. Im kurzen Format `short` mit 4 Nachkommastellen der Mantisse bleiben diese Abweichungen oft verborgen.

```
R2 = [ 1   1/2      1/3
       0 390/1351 390/1351
       0    0     317/4253 ]

R2 =
     1.00000000000000   0.50000000000000   0.33333333333333
                    0   0.28867505551443   0.28867505551443
                    0                  0   0.07453562191394
```

Die Cholesky-Faktorisierung `R = chol(AA)` ist wegen der fehlenden Voraussetzungen zur Matrix AA im Reellen nicht durchführbar und bringt eine Fehlermeldung.

5. Invertieren von Matrizen und Lösung mittels Inverser

Wir zeigen mehrere Möglichkeiten zur Invertierung einer Matrix, so dass mit deren Ergebnis die Lösung des LGS dann durch eine Matrix-Vektor-Multiplikation erhalten wird.

```
X1 = inv(A)    % A = Hilbert-Matrix H(3)
X1 =
     9.0000  -36.0000   30.0000
   -36.0000  192.0000 -180.0000
    30.0000 -180.0000  180.0000

rref([A eye(3)])
```

```
ans =
     1.0000         0         0    9.0000  -36.0000   30.0000
          0    1.0000         0  -36.0000  192.0000 -180.0000
          0         0    1.0000   30.0000 -180.0000  180.0000

X2 = ans(1:3,4:6)
X2 =
     9.0000  -36.0000   30.0000
   -36.0000  192.0000 -180.0000
    30.0000 -180.0000  180.0000

x = (inv(A)*b)'    % aufwendiger als x=A\b
x =
     1.0000    1.0000    1.0000

X4 = inverse(A)    % symbolisch

% Bruchdarstellung sehr lang!
X4 =
[   4225502000760779683654435252 9/46950022230676122118317058 73,
      -5070602400912910850587371765 76/14085006669202836635495117619,
          14085006669202498865523064832 0/46950022230676122118317058 73 ]
[-5070602400912910850587371765 76/14085006669202836635495117619,
       8112963841460664116339235880960/4225502000760850990648535285 7,
         -25353012004564588029934064107 52/14085006669202836635495117619 ]
[  14085006669202498865523064832 0/46950022230676122118317058 73,
      -25353012004564588029934064107 52/14085006669202836635495117619,
          8451004001521544355310112604 16/46950022230676122118317058 73 ]

% Kontrolle zum Element an der Position (1,1)
9-4225502000760779683654435252 9/46950022230676122118317058 73

ans =
     1.527666881884215e-013

X3 = inv(AA)
X3 =
     -1    -1     1
     -1     4    -2
      1    -2     1

AA*X3    % Kontrolle
```

6. Lösung mittels LU-Faktorisierung

Hier kommen die für MATLAB typischen Matrixoperatoren \ und / zur Anwendung.

`x=A\b` ist die Lösung von $Ax = b$,

`x=b/A` ist die Lösung von $xA = b$.

In der Links-Division \ wird die quadratische Matrix mittels Gauß-Elimination faktorisiert und damit dann das LGS $Ax = b$ gelöst.
Für eine rechteckige Matrix A liegt die Householder-Orthogonalisierung zugrunde, und man sucht die Lösung im Sinne der Methode der kleinsten Quadrate (least squares sense).
Rechts- und Links-Division sind ineinander überführbar.

$$\texttt{b/A} = (\texttt{A}'\backslash\texttt{b}')', \quad \texttt{A}\backslash\texttt{b} = (\texttt{b}'/\texttt{A}')'.$$

Die genannten Divisionen sind im Vergleich zu anderen Verfahren zu empfehlen, weil sie im Allgemeinen genauer und schneller sind.

```
x = A\b

x =
     1.0000
     1.0000
     1.0000

X1 = A\eye(3)   % Inverse

X1 =
     9.0000  -36.0000   30.0000
   -36.0000  192.0000 -180.0000
    30.0000 -180.0000  180.0000
```

7. Funktion analog zum Kommando `linsolve`

Der Funktionskopf des entsprechenden m-Files `gaussel.m` ist

```
function [B,r] = gaussel(A).
```

Das Eliminationsverfahren erzeugt eine obere Dreiecksmatrix B (nach evtl. Zeilenvertauschungen) und den Rang der Matrix. Die einfache Spaltenpivotisierung wird gemäß $A(r, r) \neq 0$ durchgeführt.
Für eine reguläre quadratische Matrix A kann die Funktion dann zur Lösung des LGS mit oberer Dreiecksmatrix genutzt werden. Für eine beliebige (n, m)-Rechteckmatrix bedeutet es i. Allg. nur eine Transformation auf die obere Dreiecksmatrix mit der Bestimmung des Rangs $r \leq \min(n, m)$.

```
% gaussel.m

function [B,r] = gaussel(A)
% gaussel returns the transformed rectangular matrix
% as upper triangular matrix B (after possible row exchanges,
% pivot strategy with A(r,r)<>0 ) with the rank r<=min(n,m).
% Use for solving Ax=b or transformation of A.
%
[n m] = size(A);  % row and column number of A
B = A;
r = 1;  % row index
c = 1;  % column index
while (c<=m) & (r<=n)
  i = r;
  % find pivot element in column c
  while  (i<=n)   % nicht "for i = r:n"  nehmen, da nach
                  % Laufanweisung stets i<=n ist (also i<>n+1),
                  % anders als in Maple
    if B(i,c) ~= 0, break, end;
    i = i+1;
  end;
  if (i<=n)       % pivot element exists
    if (i~=r)     % row exchange
      for j = c:m
        t = B(i,j); B(i,j) = B(r,j); B(r,j) = t;
      end;
    end;
    for i = r+1:n        % transform rest table
      if B(i,c) ~= 0
        t = B(i,c)/B(r,c);
        for j = c+1:m, B(i,j) = B(i,j)-t*B(r,j); end;
        B(i,c) = 0;
      end;
    end;
    r = r+1;
  end;
  c = c+1;
  % Kontrollausgabe der Zwischenschritte
  % disp(r);
  % disp(B);
end;
r = r-1;
% end of function gaussel
```

Einige Aufrufe der Funktion.

```
gaussel(A)

ans =
     1.0000    0.5000    0.3333
          0    0.0833    0.0833
          0         0    0.0056

[B,r] = gaussel(A)

B =
     1.0000    0.5000    0.3333
          0    0.0833    0.0833
          0         0    0.0056

r =
     3
```

Transformation der erweiterten Matrix mit Lösung des Dreieckssystems.

```
[n m] = size(AE)
[B,r] = gaussel(AE)

B =
     1.0000    0.5000    0.3333    1.8333
          0    0.0833    0.0833    0.1667
          0         0    0.0056    0.0056
r =
     3

if r==n
  for i = n:-1:1
    h = B(i,n+1);
    for j = i+1:n
      h = h - x(j)*B(i,j);
    end;
    x(i) = h/B(i,i);
  end;
  disp('Loesung des Gleichungssystems')
  x
else
  sprintf('Abbruch wegen Rang r = %2i < %2i = n',r,n)
end;
```

2.6 Komplexität und Geschwindigkeitstest

Zur Einschätzung der Effizienz von Algorithmen bietet MATLAB mehrere Möglichkeiten. Zwei Varianten sind die Bestimmung der Anzahl der durchgeführten Gleitpunktoperationen mit `flops` (floating point operations) sowie die Zeitmessungen mit `clock, etime` und `tic, toc`.
Das Kommando `flops(0)` (nicht `flops = 0`) setzt den Zähler auf Null zurück. So kann die Eingabe von `flops(0)` unmittelbar vor dem Beginn des Algorithmus und der Aufruf `flops` gleich nach seiner Beendigung die *flops* ermitteln.
Achtung: In MATLAB Version 6 ist das Kommando `flops` nicht mehr vorhanden.

```
n = 20;
A = rand(n);            % Zufallsmatrix
b = rand(n,1);          % Zufallsspaltenvektor

% Anzahl der Operationen flops
flops(0);
A\b;
flops
ans =
     8034
```

Die MATLAB Funktion `clock` gibt die aktuelle Zeit mit der Genauigkeit auf eine Hundertstel Sekunde an. Mit zwei solchen Zeiten kann `etime` die abgelaufene Zeit (Zeitdifferenz) in Sekunden bestimmen.
Seit der Version 4.0 gibt es die bequemere Variante einer Stoppuhr mit `tic, toc`.

```
% Zeitmessungen in Sekunden mit clock/etime bzw. tic/toc
t0 = clock; A\b; t = etime(clock,t0)
t =
     0.0600

tic; A\b; toc
elapsed_time =
     0.0600

tic; A\b; t = toc
t =
     0.0600
```

Man beachte, dass bei timesharing Arbeit am Rechner das Ergebnis der Zeitmessung natürlich verfälscht sein kann. Rechnungen an einem separaten 486er PC 66MHz zeigen, dass dies bei Arbeit im Netz mit dem PC (hier Pentium II 350MHz) bis zu ca. 10% mehr Rechenzeit ausmacht.

(1) Komplexität von Algorithmen zur Lösung von LGS

Wir vergleichen hier die Operationszahlen und Rechenzeiten von 5 Algorithmen. Die Rechnungen wurden auf einem 486er Einzel-PC 66MHz sowie einem PC im Netz mit Pentium II Prozessor 350MHz durchgeführt.

$Ax = b$, AB ist die erweiterte Koeffizientenmatrix $(A|b)$,

`[L,U] = lu(A); U\(L\b)`
`AT = gaussel(AB); x` mit Rückwärtseinsetzen
`A\b`
`rref(AB)`
`inv(A)*b`

Die verschiedenen Ergebnisse werden so abgespeichert, um sie u. a. als vergleichendes Balkendiagramm darzustellen.

```
title('SPEEDTEST for SOLUTION of Ax=b for n = 20x20 .. 400x400 ')
clear n t s sn ns tn nt q tt ss
echo off
h = 1;     % Skalierung des Balkenabstandes
na = 20; nsw = 20; ne = 200;
for n = na:nsw:ne
  A = rand(n)+n*eye(n);    % A streng diagonaldominant
  b = rand(n,1);
  AB = [A b];
  %
  t0 = clock;  flops(0);
  [L U] = lu(A);
  U\(L\b);
  t(n) = etime(clock,t0);
  s(n) = flops;
  q(n) = 0;
  if (t(n)~=0), q(n) = s(n)./t(n); end;
  %
  t0 = clock;  flops(0);
  AT = gaussel(AB);
  for i = n:-1:1
    hh = AT(i,n+1);
    for j = i+1:n,  hh = hh - x(j)*AT(i,j);  end;
    x(i) = hh/AT(i,i);
  end;
  t(n+2*h) = etime(clock,t0);
  s(n+2*h) = flops;
  q(n+2*h) = 0;
  if (t(n+2*h)~=0), q(n+2*h) = s(n+2*h)./t(n+2*h); end;
  ..............
```

Bei der Anzahl der Gleitpunktoperationen werden mit `flops` alle arithmetischen Operationen extra gezählt.
Die sonst übliche Angabe des Aufwandes für den GA mit $T^*(n) = \frac{n^3}{3} + n^2 - \frac{n}{3}$ im Operationsmix $\{+, *\}$ muss also verdoppelt werden. Trotzdem treten noch Ungenauigkeiten auf.

Sinnvoll ist es also, die Komplexitätsfunktion (2.4) $T(n) = \frac{2n^3}{3} + \frac{3n^2}{2} + \frac{n}{6}$ zu verwenden. Somit ergibt sich die Größenordnung $\mathcal{K} = \frac{2}{3}n^3 + \mathcal{O}(n^2)$ für die Verfahren `lu, gaussel` und `A\b`.

Der Zugang über die inverse Matrix hat auf der Basis von (2.8) $T^*(n) = \frac{n^3}{3} + n^2 - \frac{n}{3}$ bei n rechten Seiten die Komplexität $\hat{T}_{inv}(n) = 2(\frac{n^3}{3} + n \cdot n^2 - \frac{n}{3}) = \frac{8n^3}{3} - \frac{2n}{3}$.

Dieselbe Größenordnung besitzt auch $T_{inv}(n) = \frac{8n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$, während die genauere Betrachtung mit den Einheitsvektoren die Beziehungen (2.5) $\widetilde{T}_{inv}(n) = \frac{13n^3}{6} + n^2 + \frac{n}{6}$ bzw. (2.6) $\overline{T}_{inv}(n) = 2n^3 + n^2 + \frac{n}{3}$ ergab, was auf die Größenordnung $\mathcal{K} = 2n^3 + \mathcal{O}(n^2)$ führt. Gleichzeitig bedeutet der Vergleich der führenden Koeffizienten von $T(n)$ und $\overline{T}_{inv}(n)$, dass die Verwendung der inversen Matrix zur Lösung des LGS $2 : \frac{2}{3} = 3$ Mal mehr arithmetischen Aufwand erfordert. Die Verhältnisse sind bei kleinen Dimensionen n noch nicht so ausgeprägt, werden aber mit wachsendem n immer deutlicher.

Deshalb eine Vergleich von Ergebnissen aus den hergeleiteten Formeln mit den ermittelten *flops* am PC.

Verf.	n	Formel	Wert	*flops*
`lu`	20	$T(n) = \frac{2n^3}{3} + \frac{3n^2}{2} + \frac{n}{6}$	5 937	6 028
		$2T^*(n) = \frac{2n^3}{3} + 2n^2 - \frac{2n}{3}$	6 120	
	200	$T(n)$	5 393 367	5 394 118
		$2T^*(n)$	5 413 200	
`inv`	20	$\overline{T}_{inv}(n) = 2n^3 + n^2 + \frac{n}{3}$	16 407	18 510
		$\widetilde{T}_{inv}(n) = \frac{13n^3}{6} + n^2 + \frac{n}{6}$	17 737	
		$T_{inv}(n) = \frac{8n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$	21 537	
		$\hat{T}_{inv}(n) = \frac{8n^3}{3} - \frac{2n}{3}$	21 320	
	200	$\overline{T}_{inv}(n)$	16 040 067	16 250 802
		$\widetilde{T}_{inv}(n)$	17 373 367	
		$T_{inv}(n)$	21 353 367	
		$\hat{T}_{inv}(n)$	21 333 200	

Tab. 2.4 Aufwand von Verfahren gemäß Formel und am PC

Die Kommandos `lu`, `gaussel` und `A\b` bestätigen die Formel $T(n)$ für die *flops*, wobei `A\b` für kleinere Dimensionen etwas mehr braucht.
Genauso passen die *flops* bei `inv` am ehesten zu $\overline{T}_{inv}(n)$.

Ausgewählte Ergebnisse für 486er PC.

```
title('SPEEDTEST for SOLUTION of Ax=b for n = 20x20 (20x20)) 200x200')
flops
```

n	lu	gaussel	A\b	rref	inv
20	6028	6221	8020	27747	18510
40	45238	46021	53466	132595	138046
60	149648	151421	168298	357901	454568
80	351258	354421	384610	750222	1064170
100	682068	687021	734244	1351968	2062694
120	1174078		1249534		3546474
140	1859288		1962140		5611170
160	2769698		2904036		8352756
180	3937308		4107254		11867264
200	5394118		5603902		16250802

```
times
```

n	lu	gaussel	A\b	rref	inv
20.0000	0	0.9900	0	5.7700	0
40.0000	0	6.4800	0	20.8700	0.0600
60.0000	0.0600	20.4800	0.1100	44.5500	0.1600
80.0000	0.1700	48.1100	0.2200	77.6100	0.4400
100.0000	0.3300	89.5300	0.3300	118.5300	0.8200
120.0000	0.4900		0.4900		1.3700
140.0000	0.7700		0.7100		2.0800
160.0000	1.0500		1.1500		3.0300
180.0000	1.7000		1.6400		4.2900
200.0000	2.3600		2.2500		6.1500

Die Kommandos `lu` und `A\b` zeigen sowohl für *flops* als auch in der Rechenzeit $time = T$ ähnliches Verhalten. Entsprechend seiner Komplexität ist `inv` dreimal schlechter. Dass `gaussel` trotz gleicher *flops* wie `lu` in der Rechenzeit deutlich schlechter ausfällt, liegt nicht hauptsächlich am timesharing Modus des Rechners. Es ist zu vermuten, dass auch die Struktur der einfachen Laufanweisungen in der Funktion `gaussel` zu Rechenzeitverlusten führt gegenüber solchen, die eventuell in den anderen Kommandos Teilvektoren und Untermatrizen nutzt und eine schnelle Adressenmanipulation beim Zugriff auf Feldkomponenten realisiert.
Auch `rref` wird im Vergleich mit `gaussel` trotz größerer *flops* bezüglich der Rechenzeit immer günstiger, kann aber bei weitem nicht mit `lu` konkurrieren.
Der GA ohne Pivotstrategie (Funktion `gausseo`) zeigt ähnliche Komplexität wie *gaussel*. Das unterstreicht, dass die Auswertung von Bedingungen/Tests größenordnungsmäßig die Komplexität nicht beeinflusst.

n	lu	gaussel	A\b	rref	inv
20	6028	6221	8028	27435	18518
40	45238	46021	53482	132595	138062
60	149648	151421	168380	357758	454650
80	351258	354421	384674	748467	1064234
100	682068	687021	734390	1349225	2062840
120	1174078	1181221	1249514	2214142	3546454
140	1859288	1869021	1962066	3378712	5611096
160	2769698	2782421	2903782	4896733	8352502
180	3937308	3953421	4107164	6812084	11867174
200	5394118	5414021	5604018	9170035	16250918
250	10511393	10542521	10840204	17335976	31642579
300	18136168	18181021	18609462	29346143	54564812
350	28768443	28829521	29412540	45912719	86518365
400	42908218	42988021	43750234	67730467	129004034
$\mathcal{K}$	$\frac{2}{3}n^3$			n^3	$2n^3$

Tab. 2.5 Ergebnistableau für PC Pentium II: *flops*

n	lu	gaussel	A\b	rref	inv
20	0	0.05	0	0.39	0
40	0	0.49	0	1.43	0
60	0	1.54	0	3.02	0
80	0	3.51	0	5.28	0
100	0	6.70	0	8.02	0.05
120	0	11.53	0	11.48	0.11
140	0.06	17.96	0	15.43	0.22
160	0.06	26.69	0.11	20.32	0.22
180	0.11	37.79	0.11	25.76	0.33
200	0.11	56.35	0.11	34.49	0.44
250	0.21	99.26	0.32	50.92	0.49
300	0.38	169.34	0.66	74.70	1.26
350	0.71	271.50	0.66	104.63	1.49
400	1.10	400.79	1.04	140.56	3.89
Vergl.	$\frac{1}{3}T$	$100T$	$\frac{1}{3}T$	$35T$	T

Tab. 2.6 Ergebnistableau für PC Pentium II: *time* in *sec*

(2) Komplexität von Algorithmen zur Bestimmung der Inversen

Wir vergleichen hier die Operationszahlen und Rechenzeiten von 4 Algorithmen. Die Rechnungen wurden auf einem PC mit Pentium II Prozessor 350MHz durchgeführt.

A^{-1}, AE ist die um die Einheitsmatrix I erweiterte Koeffizientenmatrix

```
inv(A)
A\I
[L,U] = lu(A); U\(L\I)
rref(AE)
```

n	inv	A\I	lu	rref
20	17708	22840	22750	47497
40	134814	176676	176280	277181
60	447356	589548	588610	826784
80	1051446	1389568	1387740	1840280
100	2042818	2704470	2701670	3457743
150	6846281	9085008	9078745	11084328
200	16171200	21484502	21473320	25605056
250	31517269	41902646	41885395	49264454
300	54384734	72339686	72314970	84330436
350	86273097	114795124	114762045	133003654
400	128683576	171270178	171226620	197482589
$\mathcal{K}$	$2n^3$	$\frac{8}{3}n^3$		$3n^3$

Tab. 2.7 Ergebnistableau für PC Pentium II: *flops*

n	inv	A\I	lu	rref
20	0	0	0	0.55
40	0	0	0	2.04
60	0	0	0	4.72
80	0.06	0	0.05	8.63
100	0.06	0.06	0.11	13.89
150	0.22	0.22	0.27	31.53
200	0.44	0.61	0.87	59.65
250	0.68	1.21	1.48	100.02
300	1.26	2.08	3.40	154.40
350	1.49	2.25	4.01	225.14
400	3.89	6.32	9.39	312.91
Vergl.	T	$T..\frac{3}{2}T$	$T..2T$	

Tab. 2.8 Ergebnistableau für PC Pentium II: *times in sec*

Wir hatten schon vorher die Komplexität für die Bestimmung der Lösung eines LGS mittels `inv(A)*b` untersucht. Die alleinige Invertierung der Matrix macht gemäß (2.6) den Aufwand $2n^3 - n^2 + \frac{n}{3}$ unterscheidet sich in den *flops* eben nur durch $2n^2$ von den für die Lösung `inv(A)*b`. So beträgt die Größenordnung der Komplexität für die inverse Matrix auch $\mathcal{K} = 2n^3 + \mathcal{O}(n^2)$. Auch die Rechenzeiten T sind ungefähr die gleichen. Die Kommandos `A\I` und `lu` zeigen sowohl für *flops* als auch in der Rechenzeit ähnliches Verhalten, sind aber nicht so gut wie `inv`. Ihre Komplexität ist $\approx \frac{8}{3}n^3$. `rref` fällt im Vergleich als sichtbar schlechteste Variante auf.
Die Kommandos `A\I` und `lu` geraten aber bezüglich der Rechenzeit im Vergleich mit `inv` mit wachsendem n immer mehr ins Hintertreffen. Das Verhältnis anders als bei den *flops* verschlechtert sich. Der Befehl `rref` braucht die längste Rechenzeit.
`A\I` braucht ca. 4 Mal mehr *flops* und *time* als `A\b`. Ähnlich ist es bei `lu`, wobei die Rechenzeitrelation von `lu`(A^{-1}) : `lu`$(Ax = b)$ noch zunehmend schlechter wird.
Wird die inverse Matrix explizit gebraucht, so ist die built-in-Funktion `inv` für die Berechnung am besten geeignet.

(3) Vergleich mit einer Implementierung in Pascal

Bisher wurden in MATLAB 5.3 die Operationszahlen und Rechenzeiten von mehreren Algorithmen verglichen. Jetzt kommt noch ein weiteres Verfahren in der Sprache Pascal unter BPW 7.0 (*double* Präzision) dazu.

(3.1) Lösung von LGS

Zu den 5 Algorithmen ergänzen wir den Algorithmus der Gauß-Elimination mit einfacher Spaltenpivotisierung unter BPW 7.0 (*double* Präzision).
Das ist die MATLAB-Funktion `gaussel`, die in Turbo Pascal notiert und implementiert wird. Die Pascal-Routine heißt `gaussel1`. Beide haben somit die gleiche Komplexität an Operationen.
Die Rechnungen wurden auf einem PC im Netz mit Pentium III Prozessor 800 MHz durchgeführt, der ca. zweimal so schnell wie der Pentium II ist.

	MATLAB					BPW 7.0
n	`lu`	`gaussel`	`A\b`	`rref`	`inv`	`gaussel1`
100	682068	687021	734390	1349225	2062840	wie `gaussel`
200	5394118	5414021	5604018	9170035	16250918	
300	18136168	18181021	18609462	29346143	54564812	
400	42908218	42988021	43750234	67730467	129004034	
$\mathcal{K}$	$\frac{2}{3}n^3$			n^3	$2n^3$	$\frac{2}{3}n^3$

Tab. 2.9 Ergebnistableau für PC Pentium III: *flops*

	MATLAB					BPW 7.0
n	lu	gaussel	A\b	rref	inv	gaussel1
100	0	3.46	0	3.18	0.06	0.05
200	0.06	27.08	0.11	11.75	0.16	0.50
300	0.61	91.07	0.54	25.93	1.04	1.81
400	0.95	215.74	0.90	45.97	2.30	4.34
Vergl.	$\frac{1}{3}T$	$100T$	$\frac{1}{3}T$	$20T$	T	$\frac{3}{2}..2T$

Tab. 2.10 Ergebnistableau für PC Pentium III: $T = time$ in *sec*

```
{$N+}
{ UP fuer Loesung eines LGS mit Gauss-Elimination
  und Spaltenpivotisierung
  in  HP  LGS_POI1.PAS                                }

 type
  {$IFOPT N+}  float = double;
  {$ELSE}      float = real;
  {$ENDIF}

type  vektor =array[1..nmax] of float;
      Pvektor=^vektor;
      matrix =array[1..nmax] of Pvektor;
      Pmatrix=^matrix;

procedure gaussel1(n:integer; var a,atr:Pmatrix; var b,x:Pvektor;
                   var rank:integer);
{ gaussel returns the transformed rectangular matrix (A | b)
  as upper triangular matrix A' (after possible row exchanges,
  pivot strategy with A(r,r)<>0 ) with the rank r<=n.
  Use for solving Ax=b and transformation of A.   }

var i,j,m,r,c:integer;
    t,hh:float;
begin
 { Vorwaertsrechnung }
 for i:=1 to n do
  for j:=1 to n do atr^[i]^[j]:=a^[i]^[j];

 (*  atr:=a;
   { nur Zeigerumspeicherung --> Fehler bei Zeigerverwaltung } *)

 m:=n+1;
```

```
 for i:=1 to n do atr^[i]^[m]:=b^[i];
 r:=1;    { row index }
 c:=1;    { column index }
 while (c<=m) and (r<=n) do
 begin
  i:=r;
  { find pivot element in column c }
  while i<=n do
   begin
    if atr^[i]^[c] <> 0 then break;
    i:=i+1;
   end;
  if i<=n then    { pivot element exists }
   begin
    if i<>r then  { row exchange }
     for j:=c to m do
      begin
       t:=atr^[i]^[j]; atr^[i]^[j]:=atr^[r]^[j]; atr^[r]^[j]:=t;
      end;

    { transform rest table }
    for i:=r+1 to n do
     if atr^[i]^[c]<>0 then
      begin
       t:=atr^[i]^[c]/atr^[r]^[c];
       for j:=c+1 to m do atr^[i]^[j]:=atr^[i]^[j]-t*atr^[r]^[j];
       atr^[i]^[c]:=0;
      end;
    r:=r+1;
   end;
  c:=c+1;
 end;
 r:=r-1;
 rank:=r;

 { Rueckwaertsrechnung }
 if r=n then
  for i:=n downto 1 do
   begin
    hh:=atr^[i]^[m];
    for j:=i+1 to n do hh:=hh-x^[j]*atr^[i]^[j];
    x^[i]:=hh/atr^[i]^[i];
   end;
end;
```

(3.2) Bestimmung der Inversen

Bisher wurden die Operationszahlen und Rechenzeiten von 4 Algorithmen verglichen. Jetzt kommt noch das Austauschverfahren mit Spaltenpivotsuche und Zeilenvertauschung als MATLAB-Funktion `austaus` sowie die analoge Pascal-Prozedur in BPW `Invert_Austausch_P` unter Verwendung von Heap und Pointertechnik dazu.
Beide haben somit die gleiche Komplexität an Operationen.
Die Rechnungen wurden auf einem PC im Netz mit Pentium III Prozessor 800MHz durchgeführt, der ca. zweimal so schnell wie der Pentium II ist.

```
% austaus.m
%
function [B,p,t,ind] = austaus(A,eps)

% Austauschverfahren mit Spaltenpivotsuche und Zeilenvertauschung
%  A .. B=A^(-1)
%  eps     Toleranz fuer Test auf Singularitaet a[i,i]<...
%  p       Permutationsvektor der Zeilenvertauschung
%  t       Indikator : 0..default
%                      1..Abbruch mit eps
%  ind     Stufe (m,m) bei vorzeitigem Abbruch                    }

[n n] = size(A); % row and column number of A
B = A;
t = 0;
ind = 0;
for i = 1:n, p(i) = i; end; % Permutationsvektor --> P
for k = 1:n
  max = 0; index = k;       % Pivotsuche
  for i = k:n
    s = B(i,k);
    if (abs(s)>abs(max)) max = s; index = i; end;
  end;
  % Test auf Singularitaet
  if (abs(max)<eps) ind = k; t = 1; break; end;
  if (index>k)              % Zeilenvertauschung
    for i = 1:n
      s = B(k,i);
      B(k,i) = B(index,i);
      B(index,i) = s;
    end;
    i = p(k); p(k) = p(index); p(index) = i;
  end;
  for i = 1:n      % Bestimmung der Elemente der k-ten Spalte
```

```
    B(i,k) = B(i,k)/max;
  end;
  B(k,k) = 1/max;
  for j = 1:n       % Bestimmung der restlichen Elemente
    if (j~=k)
     for i = 1:n
       if (i~=k) B(i,j) = B(i,j)-B(i,k)*B(k,j); end;
       B(k,j) = -B(k,j)/max;
     end;
    end;
  end;
end;
for i = 1:n         % Spaltenvertauschung mittels p
  for j = 1:n, v(p(j)) = B(i,j); end;
  for j = 1:n, B(i,j) = v(j); end;
end;    % end of function austaus
```

	MATLAB					BPW 7.0
n	inv	A\I	lu	rref	austaus	Invert_ Austausch_P
100	2042818	2704470	2701670	3457743	2960320	wie austaus
200	16171200	21484502	21473320	25605056	23840620	
300	54384734	72339686	72314970	84330436	80640920	
400	128683576	171270178	171226620	197482589	191361220	
$\mathcal{K}$	$2n^3$	$\frac{8}{3}n^3$		$3n^3$	$3n^3$	

Tab. 2.11 Ergebnistableau für PC Pentium III: *flops*

	MATLAB					BPW 7.0
n	inv	A\I	lu	rref	austaus	Invert_ Austausch_P
100	0.00	0.00	0.06	4.50	22.7	0.22
200	0.16	0.22	0.28	18.62	182.0	2.03
300	1.05	1.32	2.14	46.14	612.7	7.20
400	3.08	3.29	5.33	89.69	1492.1	17.19
Vergl.	T	$T..\frac{3}{2}T$	$T..2T$	$\approx 40T$	$\approx 500T$	$\approx 6T$

Tab. 2.12 Ergebnistableau für PC Pentium III: *time* in *sec*

Zunächst kann man für die angegebenen Verfahren die Komplexität bez. der Operationsanzahl auswerten und gut vergleichen.
Bei der Zeitkomplexität muss man berücksichtigen, dass die MATLAB-Funktionen bzw. Kommandos

```
lu, A\b, inv, A\I
```

built-in-Funktionen sind, wo der Algorithmus in einer höheren Programmiersprache implementiert ist und schnelle Adressenmanipulation beim Zugriff auf Feldkomponenten genutzt werden. Dadurch sind die sehr kurzen Rechenzeiten im Vergleich zu den Pascal-Routinen

```
gaussel1, Invert_Austausch_P
```

bei vergleichbaren *flops* zu erklären.
Nicht konkurrieren mit dieser Strategie können die selbst programmierten MATLAB-Funktionen

```
gaussel1, austaus,
```

die trotz vergleichbarer *flops* erheblich langsamer als alle anderen Varianten sind. Das wird im Prinzip durch die interpretative Abarbeitung verursacht.

(3.3) Zur Leistungsfähigkeit der Prozessoren in MATLAB und BPW

Wir fassen die Erkenntnisse aus den Berechnungen zusammen.

Matlab 5.3 : PII 350MHz
bestes Ergebnis bei Built-in-Funktion = 40 000 000 $flops/sec$
schlechtestes Ergebnis bei eigener Funktion = 100 000 $flops/sec$

Matlab 5.3 : PIII 800MHz
bestes Ergebnis bei Built-in-Funktion = 70 000 000 $flops/sec$
schlechtestes Ergebnis bei eigener Funktion = 180 000 $flops/sec$

BPW 7.0 : PIII 800MHz
Ergebnis bei eigener Funktion = 12 000 000 $flops/sec$

Ungefähre Zeiten für eine Gleitpunktoperation

PII	(MATLAB, eigene F.) :	10^{-5} *sec*
PII	(MATLAB, built-in-F.) :	$2.5 \cdot 10^{-8}$ *sec*
PIII	(MATLAB, eigene F.) :	$6 \cdot 10^{-6}$ *sec*
PIII	(MATLAB, built-in-F.) :	$1.5 \cdot 10^{-8}$*sec*
PIII	(BPW, *double*) :	10^{-7} *sec*

Ergebnis:

In MATLAB sollte man keine eigenen Funktionen für rechenintensive Algorithmen schreiben und anwenden. Wenn möglich, ist auf vorhandene MATLAB-Funktionen zurückzugreifen.

Kapitel 3

Speichern und Generieren von Matrizen

In diesem Kapitel der Arbeit befassen wir uns damit, größere Matrizen mit bis zu mehreren Millionen Elementen unter Berücksichtigung ihrer Struktureigenschaften zu verarbeiten. Solche Matrizen sind zum Beispiel zu invertieren oder treten bei der Lösung von großen LGS und EWP auf.
Die konkrete Wahl einer Lösungsmethode wird entscheidend beeinflusst durch die hardwaremäßigen Gegebenheiten des verwendeten Rechners. Auf skalaren Rechnern wird dies zu anderen Ergebnissen führen als auf Hochleistungsrechnern, die Vektorinstruktionen verwenden oder sogar Parallelisierung ermöglichen.
Selbst auf skalaren Rechnern wird die Problemstellung von einer Vielzahl von Parametern beeinflusst, wie etwa:

- schnelle und große Rechenregister, Cache-Speicher,
- interne Parallelität des Rechenwerkes,
- Optimierer, die bestimmte Sprachkonstrukte sehr effektiv behandeln, andere aber nur nur weniger effektiv,
- Verwendung von Maschinencode,
- Fragen der ökonomischen Speicherung,
- Kapazität des Speichers mit schnellem Zugriff,
- Übertragungsgeschwindigkeit zwischen dem Zentralspeicher und den Hilfsspeichermedien.

Häufig wird die Rechenzeit des Programms/Algorithmus in direkten Zusammenhang mit der Anzahl der arithmetischen Operationen gebracht. Das ist ein wichtiges Kriterium. Aber im Zuge der Hardwareentwicklung sollte man andere Fakten keinesfalls unberücksichtigt lassen.
So ist zum z. B. eine Multiplikation beim i-Pentium genauso schnell wie eine Addition. Das Quadrieren einer Zahl ist gar doppelt so schnell wie die Addition. Bei den PC-Generationen Pentium II-IV gleichen sich die Rechenzeiten der arithmetischen Grundoperationen $\{+, -, *, ()^2\}$ immer mehr an. Die Division wie auch die Berechnung der Wurzelfunktion bleiben deutlich zurück.

Zahlreiche verbesserte numerische Verfahren beruhen darauf, auf Kosten von Additionen einige Multiplikationen einzusparen (vgl. [18]). In [17] ist der Zeitgewinn untersucht, den diese Algorithmen erbringen. Dabei werden gewisse unrealistische Vorstellungen über den erzielbaren Gewinn häufig korrigiert. Einsparungen an Multiplikationen sind nicht mehr unbedingt sinnvoll, wenn eine Zuweisung mehr Zeit als eine Gleitkommamultiplikation benötigt (beim i-Pentium gar mehr als 200% der Zeit).
Dazu kommt die Unterstützung der Gleitpunktarithmetik (GPA) in der CPU durch spezielle Prozessoren. In jedem Rechner beziehungsweise bei jeder Programmiersprache kann man nur auf eine endliche Darstellung reeller Zahlen zurückgreifen, und es werden nur einige Systeme dieser Zahlen verarbeitet. Numerisches Rechnen ist durch die Einschränkung auf eine begrenzte Stellenzahl der Größen gekennzeichnet. Wir betrachten diese endlichen **Gleitpunktzahlen** (GPZ), auch Gleitkommazahlen genannt.

In der Sprache Pascal kann man durch Schalter (Compilerdirektive, Modus) oder Einstellungen im Menü festlegen, ob Operationen mit Realzahlen durch Routinen der Laufzeitbibliothek (LZB) oder über die direkte Ansteuerung eines numerischen Koprozessors stattfinden. Im Modus {$N−} steht nur der Datentyp *real* zur Verfügung, alle Operationen mit diesem Typ geschehen über Aufrufe der LZB. Genauer gesagt sind es aber keine Laufzeitroutinen, sondern spezielle Unterprogramme (call) im Kern von Pascal zur Multiplikation von zwei 6-Byte-Worten. Diese Art der Verarbeitung ist natürlich zeitaufwendiger. Der Typ *real* gehört auch nicht zum IEEE-Standard.

Im {$N+} Modus generiert Pascal Code für Gleitkpunktberechnungen mit dem 80x87-Koprozessor und bietet weitere vier Real-Typen: *single, double, extended* und *comp*. Bei Einstellung der Option nutzt Pascal für diese den im Rechner vorhandenen 80x87-Koprozessor. Der Ablauf ist z. B. bei einer Multiplikation in *double* unter Anwendung von Koprozessorroutinen wie folgt :

- `FLD` : Laden eines 8-Byte-Wortes (*double*-Zahl) in den Koprozessor,
- `FMUL` : Multiplikation einer Zahl im Koprozessor mit einer Zahl aus dem Speicher,
- `FSTP` : Rücktransfer des Ergebnisses in den Speicher.

Ansonsten können auch die 80x87-Befehle von der LZB emuliert werden. Nimmt man den Typ *real* im {$N+} Modus, dann laufen die Berechnungen über *double*, wobei durch Konvertierungen Zeitverluste auftreten. Die Reihenfolge der Ausführung einer Multiplikation ist im groben:

- Konvertierung von zwei 6-Byte-Worten (*real*-Zahlen) nach *double* und Speicherung im Koprozessor,
- Multiplikation dieser im Koprozessor mittels Routine `FMUL`,
- Rückkonvertierung in den Speicher im 6-Byte-Format.

Es treten Zeitverluste bei der Behandlung von GPZ des Typs *real* mit/ohne Numerik-Koprozessor auf.

Für die Lösung von Problemen der numerischen Mathematik ist die Einhaltung eines gewissen Standards der GPA, auch in Hinblick auf die Kombination verschiedener Programmiersprachen, sowie die schnelle Ausführung einer großen Anzahl arithmetischen Operationen natürlich wichtig. Die betrifft auch die **Matrix-Vektor-Multiplikation mit sparsen Matrizen**.

3.1 Matrix-Vektor-Multiplikation für sparse Matrizen

Die Behandlung von schwach besetzten Matrizen im Zusammenhang mit der Matrix-Vektor-Multiplikation, Lösung von Gleichungssystemen, Parallelisierung von Algorithmen mit Matrizen u. ä., hat sich inzwischen zu einem fast eigenständigen Untersuchungsgebiet etabliert. Wichtige Zielstellungen sind dabei immer wieder die günstige Speicherung der von Null verschiedenen Matrixelemente (NNE) sowie die effiziente Implementierung der notwendigen Operationen.
So findet man in [3] einen guten Einstieg in die Problematik
- Besetztheitsstruktur einer Matrix,
- Bandbreitenreduzierung,
- Datenstrukturen und Kompaktspeichertechnik.

Hier wollen wir eine Variante der Kompaktspeichertechnik und ihre Umsetzung mittels entsprechender Datenstrukturen auf dem PC betrachten.
Bezüglich der Bereitstellung von sparsen symmetrischen Testmatrizen wird die Möglichkeit der Belegung der NNE auf der Basis der Elementstruktur eines FEM-Netzes einbezogen.

3.1.1 Kompaktspeichertechniken

Die kompakte Ablage der NNE einer Matrix erfordert weitere Informationen, aus denen man eindeutig die Position des Elements in der Matrix ablesen kann. Dies sind zusätzliche Indexvektoren. Für jedes NNE sind in der Regel drei Informationen zu speichern, wobei sein (reeller) Wert ($\neq 0$) hier eine untergeordnete Rolle spielt. Nimmt man für alle NNE den Wert 1 an, so kann man die Matrix z. B. als Adjazenzmatrix eines Graphen interpretieren.

Zunächst geben wir einige Kompaktspeichertechniken an. Ihre Veranschaulichung erfolgt jeweils an einer der folgenden Matrizen mit eingetragenen NNE (Leerstelle = Nullelement).

$$a_1 = \begin{pmatrix} a_{11} & & a_{13} & & & a_{16} \\ & a_{22} & & a_{24} & a_{25} & \\ a_{31} & & a_{33} & & & \\ & a_{42} & & a_{44} & & a_{46} \\ & a_{52} & & & a_{55} & \\ a_{61} & & & a_{64} & & a_{66} \end{pmatrix} = a_1^T, \quad a_2 = \begin{pmatrix} a_{11} & & a_{13} & & \\ & a_{22} & & & a_{25} \\ a_{31} & & & a_{34} & \\ & & a_{43} & a_{44} & \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix} \neq a_2^T$$

Im Weiteren sei $a = a(n,n)$ eine quadratische Matrix mit reellen Elementen a_{ij}, n ihre Dimension und $nne \leq n^2$ die Anzahl aller ihrer NNE.
Der NNE-Vektor sei A, wobei $0 \neq a_{ij} \to A(l)$, die Indexvektoren sind mit J und I bezeichnet.

Ausgewählte Kompaktspeichertechniken:

(1) Zeilenweise Speicherung der Matrix
Demonstrationsmatrix : $a_1(n,n),\ n=6$

$A(1..nne),\ nne=16$, Vektor der NNE der Matrix (mit neuer Indizierung)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a_{11}	a_{13}	a_{16}	a_{22}	a_{24}	a_{25}	a_{31}	a_{33}	a_{42}	a_{44}	a_{46}	a_{52}	a_{55}	a_{61}	a_{64}	a_{66}

$J(1..nne)$ Vektor der zunehmenden Kolonnen- bzw. Spaltenindizes
$1 \leq J(l) \leq n$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	**3**	**6**	**2**	**4**	**5**	**1**	**3**	**2**	**4**	**6**	**2**	**5**	**1**	**4**	**6**

$I(1..n+1)$ Zeilenzeigervektor
$1 = I(1) \leq I(2) \leq ... \leq I(n+1) = nne + 1 \leq n^2 + 1$
$I(i+1) - I(i)$ Anzahl der NNE in Zeile i
$g(i) = I(i+1) - I(i) - 1$ Grad der i-ten Knotenvariable in FEM-Netz ($a_{ii} \neq 0$)

1	2	3	4	5	6	7
1	**4**	**7**	**9**	**12**	**14**	**17**

(2) Zeilenweise Speicherung der unteren Hälfte einer symmetrischen Matrix
Demonstrationsmatrix : $a_1(n,n),\ n=6,\ a_{ii} \neq 0,\ a_{ij} = a_{ji}$

$A(1..nneu),\ nneu = (nne+n)/2 = 11$

a_{11}	a_{22}	a_{31}	a_{33}	a_{42}	a_{44}	a_{52}	a_{55}	a_{61}	a_{64}	a_{66}

$J(1..nneu)$ Vektor der zunehmenden Spaltenindizes, $1 \leq J(l) \leq n$

1	2	1	3	2	4	2	5	1	4	6

$I(1..n)$ Zeilenzeigervektor
$1 = I(1) < I(2) < ... < I(n) = nneu,\ I(0) = 0$
$I(i) - I(i-1) - 1 + \{\text{Anzahl der } J(l) \text{ mit } J(l) = i\}$ Anzahl der NNE in Zeile i

1	2	4	6	8	11

(3) Zeilenweise Speicherung der Matrix mit Zeilen- und Spaltenindizes
Demonstrationsmatrix : $a_2(n,n),\ n=5$

$A(1..nne),\ nne=11$

a_{11}	a_{13}	a_{22}	a_{25}	a_{31}	a_{34}	a_{43}	a_{44}	a_{53}	a_{54}	a_{55}

$J(1..nne)$ Vektor der Spaltenindizes der NNE, $1 \le J(l) \le n$

1	3	2	5	1	4	3	4	3	4	5

$I(1..nne)$ Vektor der Zeilenindizes der NNE, $1 \le I(l) \le n$

1	1	2	2	3	3	4	4	5	5	5

(4) Spaltenweise Speicherung der Matrix mit Zeilenindizes und Spaltenbeginn-/Spaltenzeigervektor
Demonstrationsmatrix : $a_2(n,n),\ n=5$

$A(1..nne),\ nne=11$

a_{11}	a_{31}	a_{22}	a_{13}	a_{43}	a_{53}	a_{34}	a_{44}	a_{54}	a_{25}	a_{55}

$I(1..nne)$ Vektor der Zeilenindizes der NNE, $1 \le I(l) \le n$

1	3	2	1	4	5	3	4	5	2	5

$J(1..n)$ Vektor der Indizes derjenigen Komponenten von I, mit denen eine neue Spalte beginnt, $1 \le J(l) \le nne,\ J(n+1) = nne+1$

1	3	4	7	10	12

(5) Zeilenweise Speicherung der Matrix mit Spaltenindizes und Zeilenbeginn-/Zeilenzeigervektor
Demonstrationsmatrix : $a_2(n,n),\ n=5$

$A(1..nne),\ nne=11$

a_{11}	a_{13}	a_{22}	a_{25}	a_{31}	a_{34}	a_{43}	a_{44}	a_{53}	a_{54}	a_{55}

$J(1..nne)$ Vektor der Spaltenindizes der NNE, $1 \le J(l) \le n$

1	3	2	5	1	4	3	4	3	4	5

$I(1..n)$ Vektor der Indizes derjenigen Komponenten von J, mit denen eine neue Zeile beginnt, $1 \le I(l) \le nne,\ I(n+1) = nne+1$

1	3	5	7	9	12

Das entspricht also der Variante (1).

(6) Spaltenweise Speicherung der Matrix mit Speicherabbildungsfunktion
dazu Position eines Elements a_{ij} innerhalb des Vektors A
Demonstrationsmatrix : $a_2(n,n),\ n=5$

$A(1..nne),\ nne = 11$, NNE spaltenweise

a_{11}	a_{31}	a_{22}	a_{13}	a_{43}	a_{53}	a_{34}	a_{44}	a_{54}	a_{25}	a_{55}

$S(1..nne)$ Vektor der Werte der Speicherabbildungsfunktion für NNE
$S(l) = i + (j-1)\,n$ Nummer von a_{ij}

1	3	7	11	14	15	18	19	20	22	25

Für diese Kompaktspeicherstrategien sind Such- und Einsortieralgorithmen ziemlich aufwendig, da entweder das ganze Feld durchgemustert werden muss bzw. viele Komponenten um eine Position nach hinten verschoben werden müssen zwecks Freimachung einer Stelle.
Will man diese beiden Aspekte etwas geschickter handhaben, so bietet sich die Speichertechnik mit verketteten Listen an. Die verschiedenen Versionen ergeben sich dann aus einfach oder doppelt verketteten Listen bzw. spalten- oder zeilenweiser Verkettung (siehe [3]).

Hier wenden wir die Variante (1) an und notieren noch einmal in zusammengefasster Form die von der Matrix a abgeleiteten Vektoren.

Verwendete Kompaktspeichertechnik

$a(n,n)$	quadratische Matrix der Dimension n
nne	Anzahl der NNE der Matrix, $nne \le n^2$
$A(1..nne)$	Vektor der NNE zeilenweise, $a_{ij} \to A(l)$
$J(1..nne)$	Vektor der zunehmenden Spaltenindizes, $1 \le J(l) \le n$
$I(1..n+1)$	Zeilenzeiger-/Zeilenbeginnvektor
	Vektor der Indizes derjenigen Komponenten von J,
	mit denen eine neue Zeile beginnt
	$1 = I(1) \le I(2) \le ... \le I(n) \le I(n+1) = nne+1$
	$I(i+1) - I(i)$ Anzahl der NNE in Zeile i

3.1.2 Generierung einer sparsen Matrix

Neben der Vorgabe akademischer Beispiele für die Matrix-Vektor-Multiplikation mit sparsen Matrizen wurde programmseitig ein Modul eingebaut, das eine Vernetzungsstruktur eines Gebietes, wie sie in der Methode der finiten Elemente auftritt, in die NNE-Struktur und weiter auf die Vektoren A, J, I transformiert. In der Menüführung des Programms ist diese Variante ersichtlich. Sie erfordert natürlich Dateiarbeit.
Betrachten wir folgendes einfache Gebiet mit einer Vernetzung aus Dreiecken und Vierecken bei gegebener Knotennummerierung.

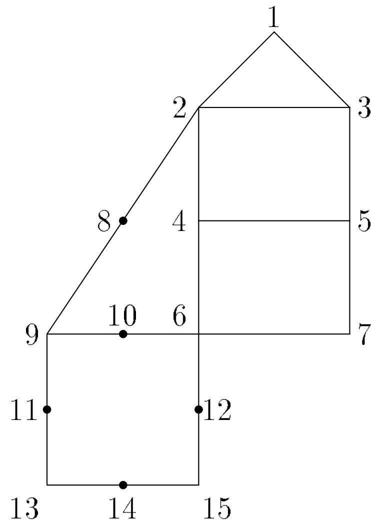


Abb. 3.1
Gebietsvernetzung

Das FEM-Netz liegt entweder in Form einer Textdatei vor oder ist als solche einzugeben. Die NNE haben den Wert 1.
Das zugehörige Netzfile ist

```
15                  % Anzahl der Netzknoten
3                   % Elemente mit 3 Knoten
1  2  3             % 3 Knotennummern (Numm. gegen den Uhrzeigersinn)
-1                  % Ende eines Elementtyps
6                   % Elemente mit 6 Knoten
2  9  6  8  10  4   % 6 Knotennummern (Numm. erst Ecken, dann Mitten)
-1
4
2  4  5  3
4  6  7  5
-1
8
9  13  15  6  11  14  12  10
-1
0                   % Dateiende
```

Die allgemeine Struktur des Netzfiles ist folgende (vergl. [4]).

n	Dimension der Matrix, Anzahl der Knotenpunkte/Freiheitsgrade
$nknot$	Anzahl der Knotenpunkte pro Element der 1. Gruppe
np_i	$i = 1(1)\,nknot$ Knotennummern im Element
np_i	$- \| -$
...	
-1	$(np_1 = -1)$ Schlusszeile für Elementtyp der 1. Gruppe
$nknot$	Anzahl der Knotenpunkte pro Element der 2. Gruppe
np_i	$i = 1(1)\,nknot$ Knotennummern im Element
...	
-1	$(np_1 = -1)$ Schlusszeile für Elementtyp der 2. Gruppe
...	
0	$(nknot = 0)$ Schlusszeile der Textdatei

Die Transformation auf das NNE-File der Matrix erfolgt über einen Zwischenschritt. Dabei werden die Indexpaare (i, j) der NNE oberhalb der Hauptdiagonalen $(i < j)$ zeilenweise gelistet. Ist die Dimension des Problems nicht sehr groß, wird für das Zwischenergebnis ein Vektor bereitgestellt, ansonsten ein File mit Komponenten vom Typ *integer*. Daraus wird das NNE-Textfile abgeleitet. Es hat die Struktur

$isym$	Symmetrie der Matrix (=1, falls Matrix symmetrisch, sonst 0)
n	Dimension der Matrix
nne	Anzahl aller NNE, $nne = n + 2\,nneo$
$nneo$	Anzahl der NNE oberhalb der Hauptdiagonalen
$F(i)$	$i = 1(1)\,2\,nne$, falls $isym = 0$ (Wert $nneo$ ohne Bedeutung)
	enthält die Indizes aller NNE der (nichtsymmetrischen) Matrix
	zeilenweise in geordneter Reihenfolge
	$i_1\ j_1\quad i_2\ j_2\quad i_3\ j_3$ ... (je 5 Paare pro Textzeile)
bzw.	
$F(i)$	$i = 1(1)\,2\,(n1 + nneo)$, falls $isym = 1$
	enthält die Indizes NNE der symmetrischen Matrix
	in (Anzahl=$n1 = nne - 2\,nneo \leq n$) und **oberhalb** (Anzahl=$nneo$)
	der Hauptdiagonalen zeilenweise in geordneter Reihenfolge
	$i_1 \leq j_1\quad i_2 \leq j_2\quad i_3 \leq j_3$ ... (je 5 Paare pro Textzeile)

Die Parameter $isym$ und $nneo$ wurden deshalb mit einbezogen, da natürlich auch solche Matrizen wie a_2 oder symmetrische Matrizen mit Nullelementen auf der Hauptdiagonalen zugelassen sein sollen.
Aus dem NNE-Textfile werden dann unter Beachtung der Symmetrie die Matrixelemente-, Column- und Row-Files A.VAL, A.COL, A.ROW zu A, J bzw. I abgeleitet. Dies sind zwar untypisierte Dateien, aber von wohldefinierter Blockgröße (*double*, *integer*, *longint*). Somit kann man auf ihre Komponenten geeignet zugreifen.

Des Weiteren sind die Problemgrößen einfach zu erhalten mittels

$n = filesize(A.ROW) - 1$ (Dateiname=Matrixname=A),

$nne = filesize(A.VAL) = filesize(A.COL)$.

Die Einbeziehung der Dateiarbeit ermöglicht im Programm auch sogenannte "Seiteneinstiege". Liegen die entsprechenden Textdateien für die Vernetzung oder der NNE der Matrix in der geforderten Struktur anderweilig vor, können diese ohne Weiteres genutzt werden.

Der oben erwähnte Zwischenschritt erfolgt auf der Basis eines FORTRAN-Programms aus [4]. Dieses wurde auf die Sprache Pascal zugeschnitten und durch die Filearbeit ergänzt.

```
C -------------------------------------------------------------------
C     HP : TRANSFER.FOR    DOS-Version
C     UP : -
C -------------------------------------------------------------------
C     HAUPTPROGRAMM ZUR UEBERFUEHRUNG DER ELEMENTSTRUKTUR EINES FEM-
C     NETZES IN DIE MATRIXBELEGUNG IN FORM VON NICHTNULLELEMENTEN
C     (NNE) ZWECKS WEITERER MINIMIERUNG DER BANDBREITE ODER DES
C     PROFILS
C     DAS PROGRAMM IST AUSGELEGT FUER MAXIMAL
C         NMAX  = 10000 KNOTENPUNKTE
C
C         N     = ANZAHL DER KNOTENPUNKTE, ANZAHL DER ZEILEN DER
C                 MATRIX
C         NMAX  = BEGRENZUNG FUER N
C         M     = ANZAHL DER VON NULL VERSCHIEDENEN MATRIXELEMENTE
C                 OBERHALB DER HAUPTDIAGONALEN
C         MMAX  = BEGRENZUNG FUER M
C
C     EINGABE
C         DATEI  MIT  N, (NKNOT,NP(I),-1/) 0/
C         N
C         NKNOT = ANZAHL DER KNOTENPUNKTE PRO ELEMENT (<=8)
C                 NKNOT = 0 : SCHLUSSZEILE
C         NP(I) = KNOTENNUMMERN PRO ELEMENT
C                 NP(1)<0 : SCHLUSSZEILE FUER ELEMENTTYP
C     AUSGABE
C         DATEI  MIT N, M, A(I)
C         N
C         M
C         A(I), I=1(1)2*M
C                 ENTHAELT DIE INDIZES DER NNE DER MATRIX OBER-
C                 HALB DER HAUPTDIAGONALEN IN GEORDNETER REIHEN-
C                 FOLGE  I1,J1, I2,J2, ...
C -------------------------------------------------------------------
      PARAMETER(NMAX=10000,MMAX=2048,KNOTMAX=8)
      INTEGER  NP(8),A(4098)
      INTEGER  N,M,M2,NKNOT,IN,IZ,I,J,K,L,NNP,NZP
      CHARACTER*48  FNAME1,FNAME3
      WRITE(*,899)
  899 FORMAT(' DATENTRANSFER  ELEMENTE--MATRIX'/)
      WRITE(*,900)
  900 FORMAT('  NAME DER EINGABEDATEI (...ELEM*.DAT) : ')
      READ(*,'(A48)')  FNAME1
      OPEN(UNIT=1,FILE=FNAME1,STATUS='OLD')
      WRITE(*,901)
  901 FORMAT('  NAME DER AUSGABEDATEI (...MATR*.DAT) : ')
```

```
      READ(*,'(A48)')  FNAME3
      OPEN(UNIT=3,FILE=FNAME3,STATUS='UNKNOWN')
      READ(1,*)  N
      IF(N.LT.2  .OR. N.GT.NMAX)  STOP  'N ZU KLEIN/GROSS !!'
C ---------------------------------------------------------------------
C     AUFBAU DES NNE-VEKTORS AUFGRUND DER KNOTENNUMMERN DER ELEMENTE
C ---------------------------------------------------------------------
      M = 1
      A(1) = N
      A(2) = N+1
   30 READ(1,*)  NKNOT
      IF(NKNOT.GT.KNOTMAX)  STOP  'NKNOT ZU GROSS !!'
      IF(NKNOT.GT.0)  THEN
   40   READ(1,*)  (NP(I), I=1,NKNOT)
        IF(NP(1).LT.0)  GOTO 30
        DO 90 I = 1,NKNOT-1
          NZP = NP(I)
          DO 80 J = I+1,NKNOT
            NNP = NP(J)
            IN = NNP
            IZ = NZP
            IF(NNP.GT.NZP)  GOTO 70
            IN = NZP
            IZ = NNP
   70       K = 1
   72       IF(IZ.GT.A(K))  GOTO 71
            K = K+1
            IF(IZ.LT.A(K-1))  GOTO 52
   62       IF(IZ.EQ.A(K-1) .AND. (IN.GT.A(K)))  GOTO 61
            IF((A(K-1).EQ.IZ) .AND. (A(K).EQ.IN))  GOTO 80
   52       M = M+1
            IF(M.GT.MMAX)  STOP  'M ZU GROSS !!'
            L = 2*M
   42       A(L) = A(L-2)
            A(L-1) = A(L-3)
            L = L-2
            IF(L.GT.K)  GOTO 42
            A(K) = IN
            A(K-1) = IZ
            GOTO 80
   61       K = K+2
            GOTO 62
   71       K = K+2
            GOTO 72
   80     CONTINUE
   90   CONTINUE
        GOTO  40
      ENDIF
C ---------------------------------------------------------------------
C     MATRIX IN DATEI
C ---------------------------------------------------------------------
      M = M-1
      M2 = 2*M
      WRITE(3,5)  N,M
    5 FORMAT(/1X,2I5)
      WRITE(3,6)  (A(I), I=1,M2)
    6 FORMAT((1X,10I5))
      CLOSE(1)
      CLOSE(3)
      STOP 'S C H L U S S'
      END
```

Beispiel 3.1 Dreiecksvernetzung eines dreieckigen Gebietes mit Loch,
$isym = 1,\ n = 6,\ nne = 30,\ nneo = 12.$

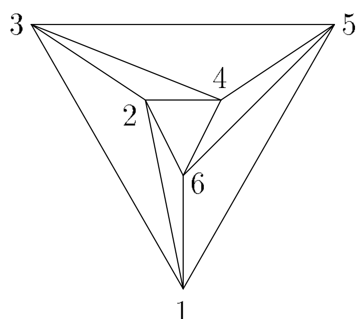


Abb. 3.2 Dreiecksvernetzung

Netz-File **ELEM5.DAT**

```
6
3
1 2 3
1 2 6
2 3 4
3 4 5
5 4 6
5 6 1
-1
0
```

Besetzungsstruktur der symmetrischen Matrix $a(6,6)$

```
x x x   x x
  x x x   x
    x x x
      x x x
        x x
          x
```

NNE-File **NMATR5.DAT**

```
1
6  30  12
1 1   1 2   1 3   1 5   1 6
2 2   2 3   2 4   2 6   3 3
3 4   3 5   4 4   4 5   4 6
5 5   5 6   6 6
```

Zwischenvektor der Indexpaare (i,j) der NNE oberhalb der Hauptdiagonalen:
1 2 1 3 1 5 1 6 2 3 2 4 2 6 3 4 3 5 4 5 4 6 5 6

File **A.VAL** der Matrixelemente (NNE $a_{ij} = 1$, GPZ vom Typ *double*) entsprechend dem Vektor $A(1..nne)$:

a_{11} a_{12} a_{13} a_{15} a_{16} a_{21} a_{22} a_{23} a_{24} a_{26}
a_{31} a_{32} a_{33} a_{34} a_{35} a_{42} a_{43} a_{44} a_{45} a_{46}
a_{51} a_{53} a_{54} a_{55} a_{56} a_{61} a_{62} a_{64} a_{65} a_{66}

File **A.COL** der Spaltenindizes der NNE (ganze Zahlen vom Typ *integer*) entsprechend dem Vektor $J(1..nne)$:

1 2 3 5 6 1 2 3 4 6 1 2 3 4 5 2 3 4 5 6 1 3 4 5 6 1 2 4 5 6

File **A.ROW** der Zeilenzeiger der NNE (ganze Zahlen vom Typ *longint*) entsprechend dem Vektor $I(1..n+1)$:

1 6 11 16 21 26 31

Beispiel 3.2 Gebietsvernetzung mit unterschiedlichen Elementen,
$isym = 1,\ n = 15,\ nne = 117,\ nneo = 51.$

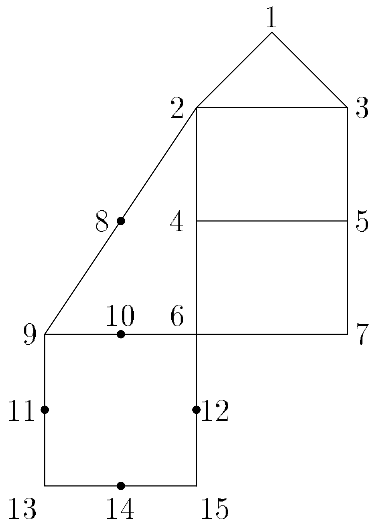
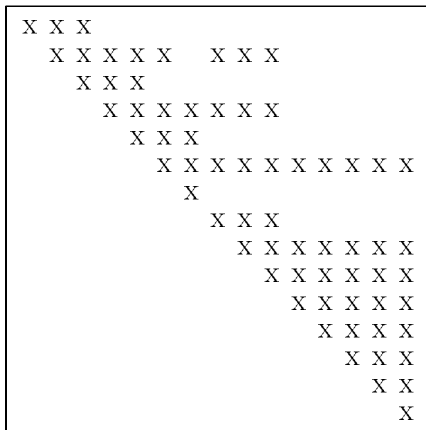


Abb. 3.3 Gebietsvernetzung

Netz-File **ELEM1.DAT**

```
15
3
1 2 3
-1
6
2 9 6 8 10 4
-1
4
2 4 5 3
4 6 7 5
-1
8
9 13 15 6 11 14 12 10
-1
0
```

Besetzungsstruktur der
symmetrischen Matrix $a(15,15)$



NNE-File **NMATR1.DAT**

```
1
15  117  51
 1    1    1    2    1    3    2    2    2    3
 2    4    2    5    2    6    2    8    2    9
 2   10    3    3    3    4    3    5    4    4
 4    5    4    6    4    7    4    8    4    9
 4   10    5    5    5    6    5    7    6    6
 6    7    6    8    6    9    6   10    6   11
 6   12    6   13    6   14    6   15    7    7
 8    8    8    9    8   10    9    9    9   10
 9   11    9   12    9   13    9   14    9   15
10   10   10   11   10   12   10   13   10   14
10   15   11   11   11   12   11   13   11   14
11   15   12   12   12   13   12   14   12   15
13   13   13   14   13   15   14   14   14   15
15   15
```

Die Struktur des Zwischenvektors und der Files **A.VAL, A.COL, A.ROW** ist analog zum Beispiel 3.1 und kann aus der Matrixbelegung bzw. dem NNE-File NMATR1.DAT abgelesen werden.

Wir beschränken uns hier auf die Notation des Files **A.ROW** der Zeilenzeiger der NNE (ganze Zahlen vom Typ *longint*) entsprechend dem Vektor $I(1..n+1)$:

1 4 13 18 27 33 46 50 56 67 78 86 94 102 110 118

3.1.3 Programmiertechnik

Die Kompaktspeicherung der Matrix wird nun integriert in die Matrix-Vektor-Multiplikation $x = a * b$. Für Probleme mittlerer Dimension kann man sich für gewisse Operationen noch im Rahmen des 64KByte-Datensegments bewegen. Wird die Dimension viel größer, muss zusätzlich Filearbeit implementiert werden. Dazu sind abgeleitete Dateien aus der Matrix a sowie weitere Dateien für die Vektoren b und x notwendig.
Das Datensegment sollte man maximal nutzen, um somit den Aufwand für den Datentransfer zwischen externen und Heapspeicher zu minimieren.
In der Unit **DEKLARE.PAS** zum Hauptprogramm **MATVEK1.PAS** (siehe [20], [22]) sind die dafür implementierten Datentypen zu erkennen.

```
{$M 65520}    {Protected-Mode von BP}
{$N+}
unit  Deklare;
{(C)  Neundorf,W. FMN TUI 1996
      Deklarationen
      DEKLARE.PAS

      zum HP  MATVEK1.PAS
              Kompaktspeicherung einer Matrix
              Anwendung bei Matrix-Vektor-Multiplikation }

interface

uses  dos,crt;

const nmax = 8191;    { 8191 = maximale Dimension der Matrix a(n,n)
                        64KByte-Datensegment = 65536Byte
                                             = 8*8192 double-Komponenten }
      nknotmax = 50;  { max. Anzahl von Knoten pro Element }
      kmax = 170;     { Grenze in FEM bei Vektor/File fuer Netz --> NNE }

type  float   =double; { real,single,double,extended }
      vektor  =array[1..nmax] of float;
      column  =array[1..nmax] of integer;
      row     =array[1..nmax+1] of longint;
      knvektor=array[1..nknotmax] of integer;
      pvektor =^vektor;
      pcolumn =^column;
      prow    =^row;

const size_el    =sizeof(float);
      size_row_el=sizeof(longint);
      size_col_el=sizeof(integer);
```

```
var   a_values    :pvektor;    { von Matrix a abgeleitete }
      a_column    :pcolumn;    { Hilfsvektoren im Heap  }
      a_row       :prow;       { fuer A,J,I }
      b_values,x  :pvektor;

      matrixfile  :file; {of float}     { *.VAL }
      columnfile  :file; {of integer}   { *.COL }
      rowfile     :file; {of longint}   { *.ROW }
      b_vektorfile:file; {of float}
      x_vektorfile:file; {of float}
      n           :integer;   { Dimension der Matrix a, Knotenpunktanzahl }
      nne,                    { Anzahl aller NNE }
      nneo        :longint;   { Anzahl der NNE oberhalb der Hauptdiag. }
      version,                { Versionen zur Behandlung der von a
                                abgeleiteten Files fuer  A,J,I }
      isym        :byte;      { Symmetrie von a (isym=1 --> a symm.) }

      matrixname,b_vektorname,x_vektorname,
      netzfname,nnefname:string[8];
      netzfile,nnefile  :text; { Textfiles : *ELEM*.DAT  Netz-File
                                             *MATR*.DAT  NNE-File  }
      eina,einb,berx:boolean;  { Steuergroessen }

implementation

end.
```

Neben dieser Unit mit den wichtigsten Vereinbarungen enthält das Hauptprogramm MATVEK1.PAS noch die Units:

- **MATVEKU1.PAS** UP zur Dateneingabe bei kleinen Systemen, $n \leq 10$,
- **MATVEKU2.PAS** UP zur Dateneingabe bei FEM-Systemen,
Ein-/Vorgabe der Gebietsvernetzung,
Ein-/Vorgabe der NNE-Matrix.

Die Komponentengröße $float = double$, $integer$, $longint$ der untypisierten Files `matrixfile, columnfile, rowfile` für die Vektoren A, J bzw. I wird mittels `sizeof( )` erfragt.

File-Speicher-Datentransfer erfolgt mittels `blockwrite, blockread`.

Dazu gibt es die Pascal-Version `procedure NNE_Matrix_File` von TRANSFER.FOR.

Anfang des Rahmenprogramms **MATVEK1.PAS**

```
{$M 65520}    {Protected-Mode von BP}
{$N+}
program MatVek1;
{(C)  Hohlbein,Dirk FIA, Neundorf,Werner FMN TUI 1996
      Kompaktspeicherung einer Matrix
      Anwendung bei Matrix*Vektor-Multiplikation
      HP    : MATVEK1.PAS
      Units : DEKLARE.PAS  Deklarationen
              MATVEKU1.PAS  Eingabe fuer kleine Systeme, n<=10
              MATVEKU2.PAS  Eingabe Gebietsvernetzung (FEM) --> Matrix }

uses  dos,crt,deklare,matveku1,matveku2;

procedure Info;
begin
 clrscr;
 writeln;
 highvideo;
 writeln('Matrix-Vektor-Multiplikation  x = a * b             ');
 writeln('fuer grossdimensionierte Probleme mit sparser Matrix');
 normvideo;
 writeln;
 writeln(
  'Problemgroessen : a(n,n)  reelle Matrix (Elemente = GKZ vom Typ FLOAT)');
 writeln('                  b(n), x(n)  reelle Vektoren                 ');
 writeln;
 highvideo;
 writeln('Algorithmus : Kompaktspeichertechnik fuer  a                  ');
 normvideo;
 writeln;
 writeln('Die Nichtnullelemente (NNE) der Matrix werden Zeile fuer Zeile');
 writeln('in den Float-Vektor A geschrieben. Im Ganzzahl-Vektor J werden die');
 writeln('dazugehoerigen Spaltenindizes gemerkt. Die i-te Komponente des');
 writeln('Ganzzahl-Vektors I enthaelt die Position des ersten NNE der i-ten');
 writeln('Matrixzeile bezogen auf die Vektoren A (bzw. J).');
 writeln('Die Vektoren A,J,I werden in sequentielle (untypisierte)',
         ' Files transformiert.');
 writeln;
 writeln('Es gilt : A(1..nne),  nne = Anzahl der NNE <= n, a(i,j) -> A(l)');
 writeln('          J(1..nne),  1<=J(l)<=n                               ');
 writeln('          I(1..n+1),  1=I(1)<=I(2)<=...<=I(n+1)=nne+1<=n+1     ');
 writeln('                      I(i+1)-I(i) = Anzahl der NNE in Zeile i   ');
 writeln;
 write  ('                                                           <ET>');
 readln;
 clrscr;
 highvideo;
 writeln('Programmiertechnik');
 normvideo;
 writeln(' - Definition von Vektortypen der Laenge 1..nmax unter Ausnutzung');
 writeln('   des 64KByte-Datensegments');
 writeln(' - Anwendung des Heapspeichers, Pointer auf Vektoren');
 writeln(' - Typlose Files fuer A,J,I, ihre Komponentengroesse FLOAT, INTEGER');
 writeln('   und LONGINT wird mit sizeof() erfragt');
 writeln(' - File-Speicher-Datentransformation mittels blockread, blockwrite,');
 writeln('   nne=filesize(A)=filesize(J)=Anzahl der NNE,  n=filesize(I)-1,');
 writeln(' - Versionen der Filebehandlung');
 ...
```

Versionen der Dateiarbeit

In der Unit der Deklarationen **DEKLARE.PAS** ist der Auswahlparameter `version` zu erkennen.
Er dient zur Festlegung der Version der Filebehandlung und korrespondiert mit dem Verhältnis von n zu $nmax$.

Version Nr.	Verhältnis der Dimensionen	Bemerkung
1	$n \leq nmax$	mit Vektortypen der Länge $1..nmax(+1)$ sind die Inhalte der Files für I, b, x mit **einem** Transfer komplett auf Vektoren im Heap übertragbar, Files für A, J i. Allg. nur abschnittsweise lesbar
2	$n \geq nmax$	Files für A, J, I blockweise transferiert, b, x komponentenweise verarbeitet
3	$n \geq nmax$	Files für A, J, I, b, x blockweise transferiert, Blockgröße $= nmax$

Tab. 3.1 Beschreibung der Versionen der Dateiarbeit im Programm MATVEK1.PAS, in Prozedur `procedure Matrix_mal_Vektor(version:byte);`

(1) Öffnen der typenlosen Dateien mit entsprechender Blockgröße/Datenformat und Abfrage der Problemdimensionen

```
assign(matrixfile,matrixname+'.VAL');
reset (matrixfile,size_el);
assign(columnfile,matrixname+'.COL');
reset (columnfile,size_col_el);
nne:=filesize(matrixfile);
assign(rowfile,matrixname+'.ROW');
reset (rowfile,size_row_el);
n:=filesize(rowfile)-1;
assign(b_vektorfile,b_vektorname+'.VAL');
reset (b_vektorfile,size_el);
assign (x_vektorfile,x_vektorname+'.VAL');
rewrite(x_vektorfile,size_el);
```

(2) Version 1
Die Handhabung der Elemente von I, b, x als Komponenten von Vektoren, die vollständig im Heap abgelegt sind, ist einfach zu durchschauen. Der Zugriff auf die Komponenten der Files für A, J erfolgt im "Gleichschritt", wobei immer nur Blöcke von $nmax$ Komponenten in die entsprechenden Heapvektoren geladen werden (letzter Block eventuell kürzer).
Der fortlaufende Index der Ergebniskomponente ergibt sich auf der Basis des Zeilenzeigervektors. Dabei ist zu berücksichtigen, dass in manchen Zeilen der Matrix nur Nullelemente stehen können. Der äußere Zyklus wird gesteuert durch die Anzahl der NNE.

```
1 : begin
     for i:=1 to n do x^[i]:=0;
     blockread(matrixfile,a_values^,nmax,result);
     blockread(columnfile,a_column^,nmax,result);
     blockread(rowfile,a_row^,n+1,result1);
     blockread(b_vektorfile,b_values^,n,result2);
     k:=0;
     val_count:=0;
     i:=1;
     repeat
      inc(k);
      inc(val_count);
      while k=a_row^[i+1] do inc(i);
      j:=a_column^[val_count];

      x^[i]:=x^[i]+a_values^[val_count]*b_values^[j];

      if val_count=result then
       begin
        blockread(matrixfile,a_values^,nmax,result);
        blockread(columnfile,a_column^,nmax,result);
        val_count:=0;
       end;
     until k=nne;
     blockwrite(x_vektorfile,x^,n);
    end;
```

Beispiel 3.3 Sei

$$a_2(5,5) = \begin{pmatrix} a_{11} & & a_{13} & & \\ & a_{22} & & & a_{25} \\ a_{31} & & & a_{34} & \\ & & a_{43} & a_{44} & \\ & & a_{53} & a_{54} & a_{55} \end{pmatrix} \neq a_2^T.$$

Die zeilenweise Speicherung dieser sparsen Matrix mit Spaltenindizes und Zeilenbeginn-/Zeilenzeigervektor ergibt:

$A(1..nne),\ nne = 11$

a_{11} a_{13}	a_{22} a_{25}	a_{31} a_{34}	a_{43} a_{44}	a_{53} a_{54} a_{55}

$J(1..nne)$ Vektor der Spaltenindizes der NNE, $1 \leq J(l) \leq n = 5$

1 3	2 5	1 4	3 4	3 4 5

$I(1..n+1)$ Vektor der Indizes derjenigen Komponenten von J, mit denen eine neue Zeile beginnt, $1 \leq I(l) \leq nne,\ I(n+1) = nne + 1$,
$1 = I(1) \leq I(2) \leq ... \leq I(n+1) = nne + 1 \leq n^2 + 1$,
$I(i+1) - I(i)$ Anzahl der NNE in Zeile i

1 3 5 7 9	12

Ablauf des Programmteils Version 1 mit $nmax = 5 \geq n,\ nne = 11$

Initialisierung

```
x[1..5]     =[0  0  0  0  0]
a_row[1..6]=[1  3  5  7  9  12]
```

a_values[1..5]=[$a_{11}\, a_{13}\, a_{22}\, a_{25}\, a_{31}$] a_column[1..5]=[1 3 2 5 1] result=5						
	k	val_count	i	a_row[i+1]	j	x[i]
	0	0	1	3		
	1	1			1	$a_{11}b_1$
	2	2			3	$+a_{13}b_3$
k=a_row[i+1]?	3	3	2	5	2	$a_{22}b_2$
	4	4			5	$+a_{25}b_5$
k=a_row[i+1]?	5	5	3	7	1	$a_{31}b_1$
a_values[1..5]=[$a_{34}\, a_{43}\, a_{44}\, a_{53}\, a_{54}$] a_column[1..5]=[4 3 4 3 4] result=5						
		0				
	6	1			4	$+a_{34}b_4$
k=a_row[i+1]?	7	2	4	9	3	$a_{43}b_3$
	8	3			4	$+a_{44}b_4$
k=a_row[i+1]?	9	4	5	12	3	$a_{53}b_3$
	10	5			4	$+a_{54}b_4$
a_values[1..5]=[a_{55}] a_column[1..5]=[5] result=1						
		0				
	11	1			5	$+a_{55}b_5$
		0				
$k = nne$?						

(3) Version 2

Für A, J, I bleiben wir bei der Verarbeitung mit Blöcken der Länge $nmax(+1)$. Der äußere Zyklus wird gesteuert durch den Zeilenzeigervektor und wird in ungefähr $\frac{n}{nmax}$ Zyklen zerlegt. Der Grund dafür ist, das I den Zeilenwechsel bestimmt und mit jedem Zeilenwechsel der Vektor b zurückzusetzen ist (`reset(b_vektorfile,size_el)`).

$nmax$	1	2	3	4
Zyklen	1 3 3 5 5 7 7 9 9 12	1 3 5 5 7 9 9 12	1 3 5 7 7 9 12	1 3 5 7 9 9 12
Anzahl $izykl$	5	3	2	2

Tab. 3.2 Zyklen und ihre Anzahl auf der Basis des Vektors I, Matrix $a_2(5,5)$ aus Kap. 3.1.1, $I(1..6) = (1,3,5,7,9,12)$, $nne = 11$

Die Zyklenanzahl $izykl$ wird berechnet durch Aufrunden des Wertes

$$h = \frac{(n+1)(nmax+1)-2}{nmax(nmax+1)}$$ auf die nächste ganze Zahl.

In jedem Zyklus gibt es maximal $nmax$ innere Schritte. Zu Beginn eines solchen ist die x-Komponente zu initialisieren und der Vektor b zurückzusetzen. Gibt es nur Nullelemente in der Matrixzeile (`diff=0`), dann wird die x-Komponente zu Null gesetzt. Ansonsten muss diese durch Aufdatieren von $a_{ij} * b_j$ (`a_values^[k]*bb`) aktualisiert werden. Dabei kann es passieren, das weitere Blöcke von A, J geholt werden müssen (`blockread(matrixfile,...); blockread(columnfile,...)`).

```
2 : begin
     blockread(matrixfile,a_values^,nmax,result);
     blockread(columnfile,a_column^,nmax,result);
     h:=((n+1.0)*(nmax+1)-2)/(nmax*(nmax+1));
     if frac(h)<1E-6 then izykl:=round(h)
                     else izykl:=trunc(h)+1;
     blockread(rowfile,aa,1,result1);
     k:=0;

     for i:=1 to izykl do
      begin
       blockread(rowfile,a_row^,nmax,result1);
       for l:=result1 downto 1 do a_row^[l+1]:=a_row^[l];
       a_row^[1]:=aa;
       aa:=a_row^[result1+1];
       for val_count:=1 to result1 do
```

```
          begin
           xx:=0;
           reset(b_vektorfile,size_el);
           diff:=a_row^[val_count+1]-a_row^[val_count];
           if diff=0 then blockwrite(x_vektorfile,xx,1)
                     else
            begin
             je:=0;
             for m:=1 to diff do
              begin
               inc(k);
               j:=a_column^[k];
               for j1:=1 to j-je do blockread(b_vektorfile,bb,1);
               je:=j;

               xx:=xx+a_values^[k]*bb;

               if k=result then
                begin
                 blockread(matrixfile,a_values^,nmax,result);
                 blockread(columnfile,a_column^,nmax,result);
                 k:=0;
                end;
              end;
             blockwrite(x_vektorfile,xx,1);
            end; { diff<>0 }
          end; { for val_count }
        end; { for i }
      end;
```

(4) Version 3

Im Vergleich zur Version 2 können wir Filetransfer dadurch einsparen, wenn
- b **nicht** komponentenweise geladen,
- x **nicht** komponentenweise gespeichert

werden.

Wenn die entsprechenden Heapvektoren von b und x abgearbeitet bzw. aufgefüllt worden sind, ist ersterer neu zu belegen bzw. das Ergebnis abzuspeichern. Dieser Zusatz sowie die eventuell besondere Situation im allerletzten Zyklus machen im Wesentlichen den Mehraufwand aus.

```
3 : begin
     blockread(matrixfile,a_values^,nmax,result);
     blockread(columnfile,a_column^,nmax,result);
     h:=((n+1.0)*(nmax+1)-2)/(nmax*(nmax+1));
     if frac(h)<1E-6 then izykl:=round(h)
                     else izykl:=trunc(h)+1;
     h1:=(n+izykl) mod (nmax+1) -1;
     blockread(rowfile,aa,1,result1);
     k:=0;
     ii:=0;

     for i:=1 to izykl do
      begin
       blockread(rowfile,a_row^,nmax,result1);
```

```
      for l:=result1 downto 1 do a_row^[l+1]:=a_row^[l];
      a_row^[1]:=aa;
      aa:=a_row^[result1+1];
      for val_count:=1 to result1 do
       begin
        xx:=0;
        reset(b_vektorfile,size_el);
        blockread(b_vektorfile,b_values^,nmax,result2);
        diff:=a_row^[val_count+1]-a_row^[val_count];
        if diff=0 then
         begin
          inc(ii);
          x^[ii]:=xx;
          if (ii=nmax) or
             (ii=h1) and (i=izykl) and (val_count=result1) then
           begin
            blockwrite(x_vektorfile,x^,ii);
            ii:=0;
           end;
         end
                  else
         begin
          je:=0;
          for m:=1 to diff do
           begin
            inc(k);
            j:=a_column^[k];
            if j<=je+result2 then bb:=b_values^[j-je]
                                 else
             begin
              repeat
               inc(je,result2);
               blockread(b_vektorfile,b_values^,nmax,result2);
              until j<=je+result2;
              bb:=b_values^[j-je];
             end;

            xx:=xx+a_values^[k]*bb;
            if k=result then
             begin
              blockread(matrixfile,a_values^,nmax,result);
              blockread(columnfile,a_column^,nmax,result);
              k:=0;
             end;
           end;
          inc(ii);
          x^[ii]:=xx;
          if (ii=nmax) or
             (ii=h1) and (i=izykl) and (val_count=result1) then
           begin
            blockwrite(x_vektorfile,x^,ii);
            ii:=0;
           end;
         end; { diff<>0}
       end; { for val_count }
    end; { for i }
  end;
```

3.1.4 Testbeispiele

(1) TB1
Multiplikation einer voll besetzten Matrix $a(n,n)$ mit Vektor $b(n)$, $b_i = 1$.
Die Komplexität des Algorithmus beträgt $T(n) = 2n^2 + \mathcal{O}(n)$.

(2) TB2

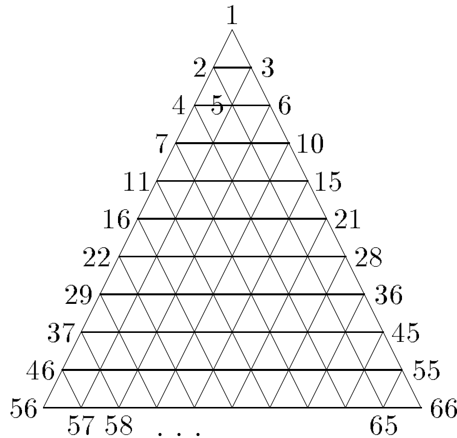


Abb. 3.4 Dreieckvernetzung
Dreieckelemente mit linearem Ansatz
Knotenpunkte zeilenweise durchnummeriert,
$isym = 1$
$n = 66$, $nne = 396$, $nneo = 165$

(3) TB3 (dieses und weitere Beispiele aus [4])

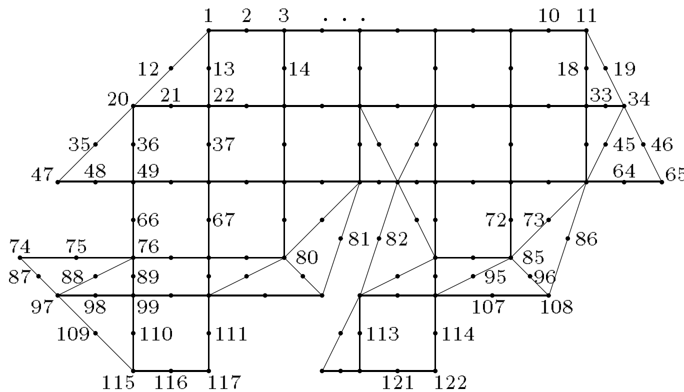


Abb. 3.5 Längsschnitt Autoinnenraum
Dreieck- und Rechteckelemente mit quadratischem Ansatz
Knotenpunkte zeilenweise durchnummeriert,
$isym = 1$
$n = 122$, $nne = 1390$, $nneo = 634$

(4) TB4

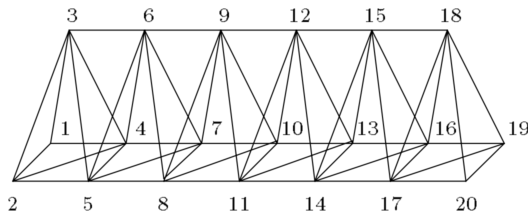


Abb. 3.6 Statisches Fachwerk Kranträger
Stabelemente
Knotenpunkte abschnittsweise durchnummeriert,
$isym = 1$
$n = 20$, $nne = 128$, $nneo = 54$

(5) TB5,6,7

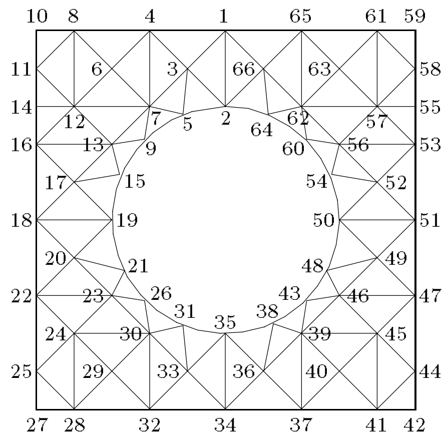


Abb. 3.7 Elliptische RWA
Ebenes Gebiet
mit Öffnung/Loch
Dreieck- und Rechteckelemente
mit linearem/bilinearem Ansatz
Knotenpunkte in positiver
Umlaufrichtung durchnummeriert,
$isym = 1$
$n = 66,\ nne = 378,\ nneo = 156$

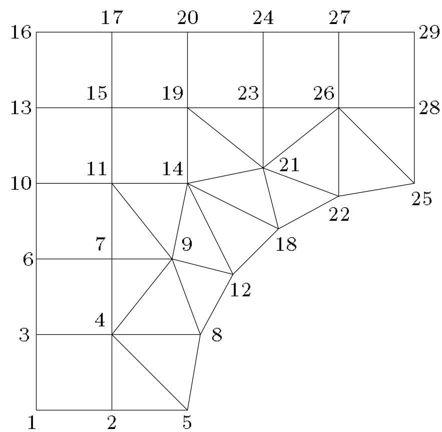


Abb. 3.8 Elliptische RWA
Gebiet mit Öffnung/Loch
wegen Symmetrie
Viertelgebiet
Dreieck- und Rechteckelemente
mit linearem/bilinearem Ansatz
Nummerierung der Knotenpunkte
zwecks günstiger Bandstruktur,
$isym = 1$
$n = 29,\ nne = 179,\ nneo = 75$

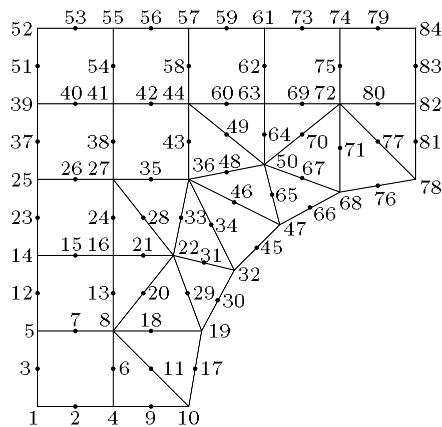


Abb. 3.9 Elliptische RWA
Gebiet mit Öffnung/Loch
wegen Symmetrie
Viertelgebiet
geradlinige Elemente
mit quadratischem Ansatz
Nummerierung der Knotenpunkte
zwecks günstiger Bandstruktur,
$isym = 1$
$n = 84,\ nne = 938,\ nneo = 427$

(6) TB8,9

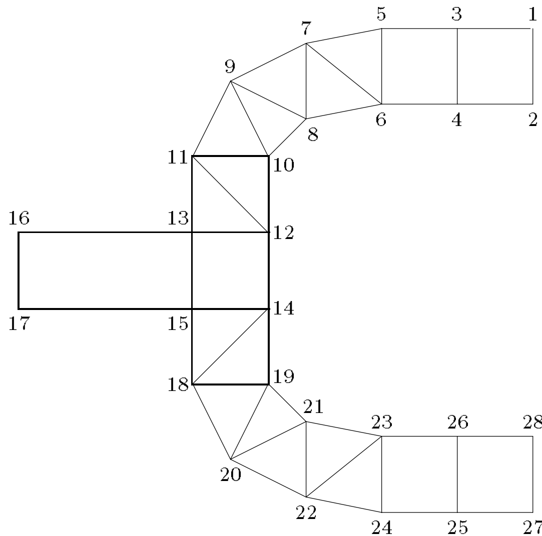


Abb. 3.10 Stimmggabel
Eigenfrequenzen u.
Eigenschwingungs-
formen
Dreieck- und Rechteckelemente
mit linearem/bilinearem Ansatz
Nummerierung der Knotenpunkte
zwecks günstiger Bandstruktur,
$isym = 1$
$n=28,\ nne=146,\ nneo=59$

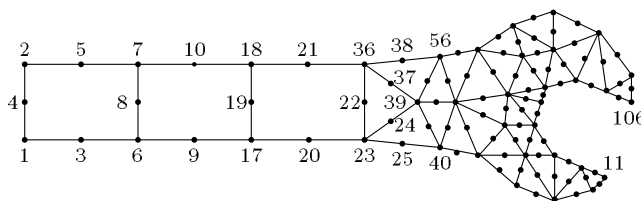


Abb. 3.11 Gabelschlüssel
Deformation
einer Scheibe
Dreieck- und Rechteckelemente
mit quadratischem Ansatz
Nummerierung der Knotenpunkte
zwecks günstiger Bandstruktur,
$isym = 1$
$n=106,\ nne=1036,\ nneo=465$

Rechenzeitvergleiche am PC

Die Rechenzeitangaben erfolgen zwar mit Hunderstelsekunden, aber durch Zeitschwankungen von ungefähr 2% ist die Relevanz der Angaben von Nachkommastellen in manchen Fällen entsprechend gemindert. Die Matrixelemente sind vom GPF *double*.

n	Anzahl der Blocktransfer für A, J bei					Rechenzeit $t(nmax)$ in sec für				
	$nmax =$					$nmax =$				
	8191	8000	6000	4000	2000	8191	8000	6000	4000	2000
100	2	2	2	3	5	0.00	0.00	0.00	0.00	0.05
500	31	32	42	63	125	1.64	2.20	2.47	2.47	2.52
1000	123	125	167	250	500	6.15	8.78	9.17	9.72	9.72
1500	275	282	375	563	1125	13.62	19.33	22.90	22.94	22.96
2000	489	500	667	1000	2000	24.44	34.72	36.58	39.10	39.76
4000	1954	2000	2667	4000		117.10	146.54	150.44	154.51	

Tab. 3.3 Testbeispiel TB1, voll besetzte Matrix $a(n,n)$, $nne = n^2 > nmax \geq n$,
Blocktransfer und Rechenzeiten der Version 1 auf PC i-Pentium

Zwischen den Varianten $nmax = 6000, 4000, 2000$ sind keine signifikanten Unterschiede in der Rechenzeit, d. h., der Filetransferaufwand ist fast der gleiche bei diesen Blockgrößen.
Erstaunlich ist der Rechenzeitgewinn, wenn die maximale Blockgröße $nmax = 8191$ eingestellt ist. Ein Grund für das Abknicken der Zeitfunktion mit Erreichen dieser Grenze wird in einer besonderen inneren Speicherkonstellation gesehen.

$nmax$	n	Anzahl der Blocktransfer für A, J	Rechenzeiten der Versionen		
			1	2	3
4000	4000	4000	154.51	$\approx 30\ min$	154.56
2000	100	5	0.05		
	500	125	2.52		
	1000	500	9.72		
	1500	1125	22.96		
	2000	2000	39.76	577.49	45.64
	4000	8000		$\approx 36\ min$	179.71
1500	100	7	0.05		
	500	167	2.69		
	1000	667	10.44		
	1500	1500	23.75	325.76	25.38
	2000	2667		576.99	47.17
	4000	10667		$\approx 36\ min$	182.51
1000	100	10	0.05		
	500	250	2.69		
	1000	1000	10.55	146.05	11.90
	1500	2250		321.10	26.15
	2000	4000		578.40	47.28
	4000	16000		$\approx 38\ min$	193.67
500	100	20	0.05		
	500	500	2.75	36.69	3.29
	1000	2000		146.26	11.98
	1500	4500		328.84	27.08
	2000	8000		$\approx 10\ min$	49.40
	4000	32000		$\approx 46\ min$	229.59
100	100	100	0.06	1.48	0.16
	500	2500		36.97	4.28
	1000	10000		147.36	16.21
	1500	22500		$\approx 6\ min$	36.30
	2000	40000		$\approx 10\ min$	64.32
	4000	160000			301.41

Tab. 3.4 Testbeispiel TB1, voll besetzte Matrix $a(n, n)$, $nne = n^2 > nmax$, Rechenzeiten in sec der Versionen 1,2,3 auf PC i-Pentium

Die Durchführung zulässiger Testrechnungen zeigt, dass sich insbesondere die Begrenzung des Filetransfers günstig auf die Rechenzeit auswirkt. Sobald noch eine Partitionierung der Datenfiles für I, b, x wegen zu kleinem Datenvektors im Heap oder eine komponentenweise Verarbeitung von b, x notwendig sind, wirkt sich dies mit erheblichen Zeitverlusten aus. Letztere, also Version 2, ist ungefähr 12 Mal langsamer als Version 3 (bei $nmax = 100$ ca. 9 Mal).

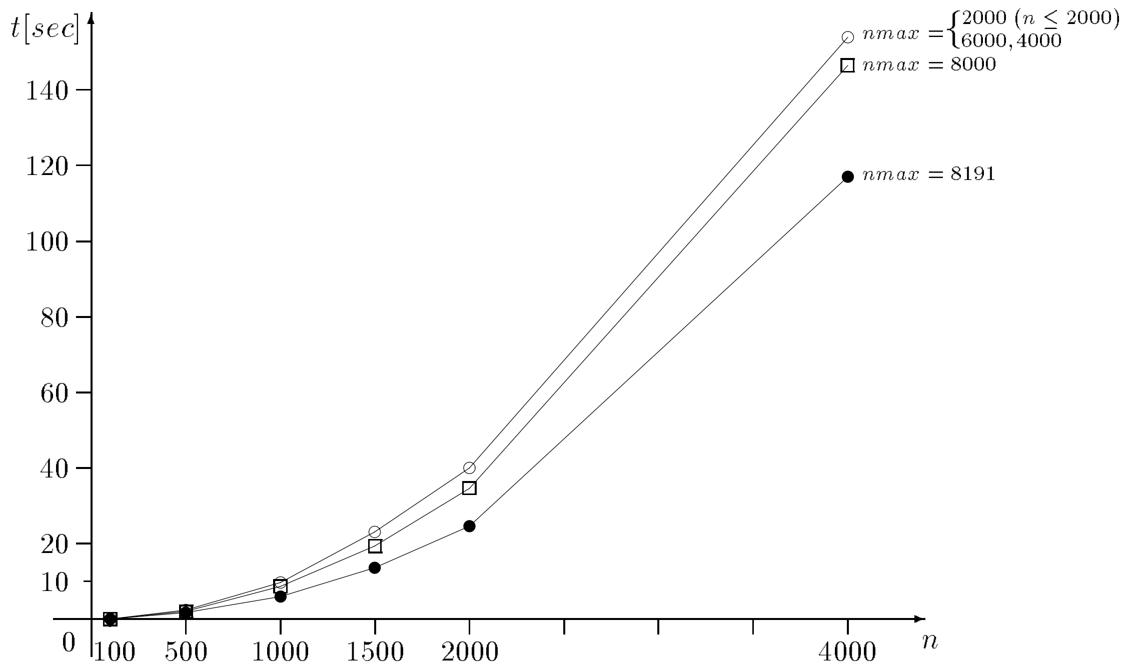


Abb. 3.12 Rechenzeiten $t(n)$ in sec der Version 1 auf PC i-Pentium für $nmax = 8191, 8000(-2000)2000$

Auf Grund der mit wachsendem n auch quadratisch steigenden Anzahl der Blocktransfer verändert sich die Komplexitätsfunktion zu $T(n, nmax) = \mathcal{O}(n^2 + nmax)$. Analoges Verhalten finden wir auch bei der Version 3.

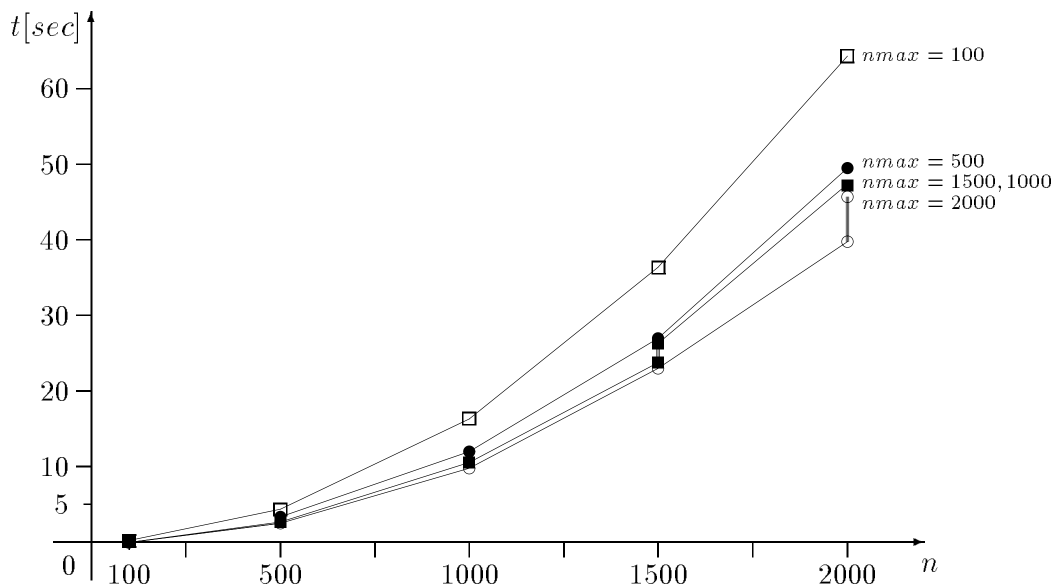


Abb. 3.13 Rechenzeiten $t(n)$ in sec der Versionen 1 ($n \leq nmax$) und 3 ($n \geq nmax$) auf PC i-Pentium für $nmax = 2000(-500)500,\ 100$

An den Stellen $n = nmax$ treten beim Übergang V1→V3 Sprungstellen auf, die in der Grafik zumindest für $nmax = 2000, 1500$ als vertikale Strecken zu erkennen sind.

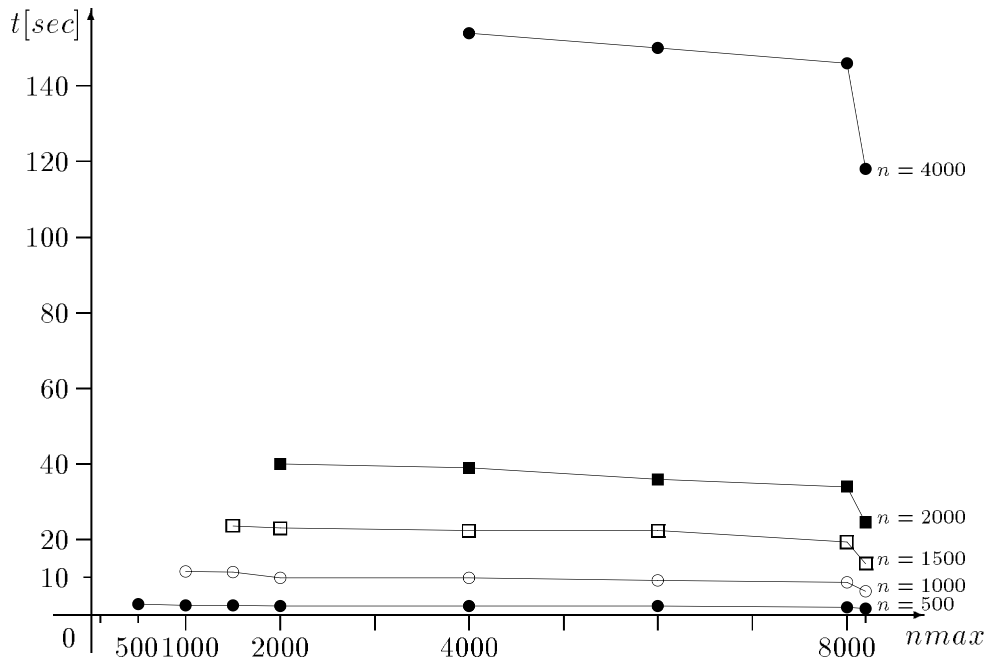


Abb. 3.14 Rechenzeiten $t(nmax)$ in sec der Version 1, $n \leq nmax$, auf PC

Die Größe $nmax$ des Heap-Vektors hat nur ganz geringe Auswirkung auf die Rechenzeit bei kleinem n. Für $n > 2000$ werden Zeitgewinne mit wachsendem $nmax$ deutlicher. Auf den "Knick" wurde in der Bemerkung zur Tabelle 3.3 verwiesen.

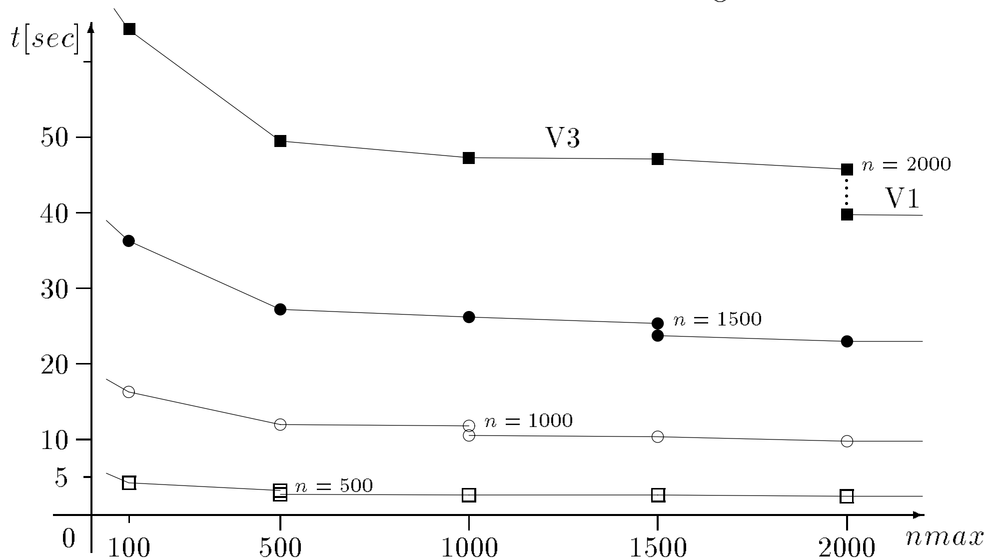


Abb. 3.15 Rechenzeiten $t(nmax)$ in sec der Versionen 3 ($n \geq nmax$) und 1 ($n \leq nmax$) auf PC i-Pentium für verschiedene n

Im ersten Abschnitt der Funktion $t(nmax)$ ist eine stärkere Abnahme zu verzeichnen. Der abfallende Verlauf ist insgesamt gut ausgeprägt bei $n = 4000$ (siehe Tab. 3.4) wie auch für $nmax < 100$ (hier nur angedeutet). Aber er verliert sich schnell mit kleiner werdendem n.

TBm	n	Version 1		V.2	V.3	Versionen 2,3					
		$nmax =$		$nmax$	$nmax$	$nmax =$					
m	nne	nne	n	n	n	1	2	5	10	20	
Kran	20	0.00	0.00	0.11	0.05	0.11	0.11	0.11	0.11	0.11	V2
4	128					0.16	0.11	0.06	0.05	0.05	V3
Stimmg.	28	0.00	0.00	0.16	0.11	0.17	0.16	0.16	0.16	0.16	V2
8	146					0.16	0.16	0.11	0.11	0.11	V3
RWA1	29	0.00	0.00	0.16	0.11	0.22	0.16	0.16	0.16	0.16	V2
6	179					0.22	0.16	0.11	0.11	0.11	V3
Lochgeb.	66	0.00	0.00	0.49	0.22	0.66	0.61	0.61	0.55	0.55	V2
5	378					0.60	0.44	0.33	0.28	0.27	V3
Dreieck	66	0.00	0.00	0.55	0.27	0.66	0.60	0.55	0.55	0.55	V2
2	396					0.66	0.44	0.33	0.33	0.28	V3
RWA2	84	0.03	0.05	0.78	0.38	1.10	0.99	0.88	0.82	0.82	V2
7	938					1.05	0.71	0.49	0.44	0.39	V3
Schlüssel	106	0.05	0.05	1.19	0.49	1.65	1.48	1.32	1.21	1.21	V2
9	1036					1.59	1.04	0.72	0.60	0.49	V3
Auto	122	0.05	0.06	1.59	0.60	2.14	1.92	1.70	1.65	1.61	V2
3	1390					2.09	1.32	0.93	0.71	0.65	V3

TBm	n	Versionen 1,2,3						
		$nmax =$						
m	nne	30,40	60	80	100	120	200	
Lochgeb.	66	-	-	0.00	0.00	0.00	0.00	V1
5	378	0.55	0.49	-	-	-	-	V2
		0.27	0.22	-	-	-	-	V3
Dreieck	66	-	-	0.00	0.00	0.00	0.00	V1
2	396	0.55	0.55	-	-	-	-	V2
		0.27	0.27	-	-	-	-	V3
RWA2	84	-	-	-	0.05	0.05	0.04	V1
7	938	0.82	0.77	0.77	-	-	-	V2
		0.39	0.38	0.38	-	-	-	V3
Schlüssel	106	-	-	-	-	0.05	0.05	V1
9	1036	1.21	1.21	1.21	1.20	-	-	V2
		0.49	0.50	0.50	0.49	-	-	V3
Auto	122	-	-	-	-	-	0.05	V1
3	1390	1.60	1.59	1.59	1.59	1.59	-	V2
		0.60	0.60	0.60	0.60	0.60	-	V3

Tab. 3.5 Testbeispiele TB2..9, FEM-Matrix $a(n, n)$,
Rechenzeiten $t(n, nmax)$ in sec der Versionen 1,2,3 auf PC ASI,
BP im Protected Mode von DOS-Oberfläche gestartet

Es gilt $t \approx 0$ für $n < 84$ und $n \leq nmax$ (Version 1). Die Zeiten der Versionen 2 und 3 nähern sich an bei $nmax \to 1$. Bei festem n bleiben für eine breiten Bereich von $nmax = n/5...n$ die Zeiten bei V2 bzw. V3 annähernd konstant.

Noch einige Bemerkungen zur Komplexitätsfunktion $T(n, nmax)$.
Dabei sollen hier der Knick bei der Version 1 nahe $nmax = 8191$ sowie die Version 2 außer Acht gelassen werden. Wir haben die zwei grundsätzlichen Fälle.

- Testbeispiele TB2...9 (FEM-Matrix)
  Die Anzahl nne der NNE ist abgesehen von $nne \leq n^2$ unabhängig von n. Das führt auf die Komplexität $T(nne, nmax) = \mathcal{O}(nne + nmax)$.

  Für festes $nmax$ ist $T(nne, nmax) = c_V\, nne$, wobei sich der positive Parameter c_V für die Versionen 1 und 3 unterscheidet. Weiterhin hat die Struktur der Matrix schon einen gewissen Einfluss, z. B. durch den Umstand, dass viele Nullzeilen in der Matrix auftreten können.
  Für festes nne ist $T(nne, nmax)$ eine monoton fallende Funktion in $nmax$. Dabei gehört sie in der ersten Phase $nmax = 1..n$ zur Version 3 und hat einen Verlauf ähnlich zur Funktion $\alpha + \beta\, e^{-\gamma\, nmax}$. Im zweiten Abschnitt $nmax \geq n$ verhält sie sich wie eine allmählich abnehmende lineare Funktion (siehe auch Abb. 3.15).

- Testbeispiel TB1 (volle Matrix)
  Hier ist $nne = n^2$. Die Anzahl der Blocktransfer wächst bei festem $nmax$ quadratisch mit n. Die Komplexität ist

$$T(n, nmax) = \mathcal{O}(n^2 + nmax) = c\, n^2 - c_V\, nmax, \quad c, c_V > 0.$$

  Die bisherigen Auswertungen basierten auf folgenden Größen.

Version 3	Version 1 $nmax \geq n$							
	$nmax =$							
n \ $nmax \leq n$	100	500	1000	1500	2000	4000	6000	8000
100	0.06	0.05	0.05	0.05	0.05	0.00	0.00	0.00
	0.16							
500		2.75	2.69	2.69	2.52	2.47	2.47	2.20
	4.28	3.29						
1000			10.55	10.44	9.72	9.72	9.17	8.78
	16.21	11.98	11.90					
1500				23.75	22.96	22.94	22.90	19.33
	36.30	27.08	26.15	25.38				
2000					39.76	39.10	36.58	34.72
	64.32	49.40	47.28	47.17	45.64			
4000						154.51	150.44	146.54
	301.41	229.59	193.67	182.51	179.71	154.56		

Tab. 3.6 Testbeispiel TB1, voll besetzte Matrix $a(n, n)$,
Rechenzeiten $t(n, nmax)\, sec$ der Versionen 1,3 auf PC i-Pentium

Sie lassen auf eine Funktion der Gestalt

$$T(n, nmax) = \begin{cases} c_1 n^2 - c_{V1}\, nmax, & \text{bei Version 1} \\ c_3 n^2 - c_{V3}\, nmax, & \text{bei Version 3} \end{cases}$$

schließen. Für festes n ist in V3 ($nmax \leq n$) der Koeffizient c_{V3} eine Funktion von $nmax$ und anschließend in V1 ($nmax \geq n$) der Koeffizient $c_{V1} \ll 1$. $T(n, nmax)$, $n = nmax$, in beiden Versionen liefert die kleinen Sprungstellen in den Abb. 3.13, 3.15.
In der Version 1 muss $T(n, nmax)$ mit wachsendem $nmax$ gegen eine positive Grenzfunktion $T(n)$ (quadratische Parabel) streben.
Falls $nmax1 > nmax$ ist, dann gilt $T(n, nmax1) < T(n, nmax)$, wobei bei festem $\Delta nmax = nmax1 - nmax$ und wachsendem $nmax$ die Abstände zwischen den Funktionswerten wegen dem Grenzwert immer kleiner werden. Allein der schon erwähnte Knick stört ein wenig dieses Konzept.

Gegenstand der Untersuchungen waren Fragen der ökonomischen Speicherung für große Matrizen und des Aufwands bei Zugriff auf diese bei Speicherung im Zentralspeicher, im Heap oder auf Dateien. Das weitestgehende Ausnutzen des Heap eröffnet schon zusätzliche Möglichkeiten und bringt Aufwandseinsparungen.
Auf spezielle Techniken zur Ausnutzung einer Band-, Block- oder hüllenartigen Struktur der Matrix ist nicht eingegangen worden. Interessant sind in diesem Zusammenhang auch Strategien der "ungepackten oder gepackten Diagonalen", die ausschließlich Diagonalen der Matrix mit NNE speichern, und dabei die jeweilige Diagonale komplett, falls sie mehr als zu 2/3 mit NNE belegt ist, oder kompakt bei nur wenigen NNE im anderen Fall (siehe [7]).

Literaturverzeichnis

[1] Kiełbasiński, A.; Schwetlick, H.: *Numerische lineare Algebra.* Mathematik für Naturwissenschaft und Technik Band 18, DVW, Berlin 1988.

[2] Hackbusch, W.: *Iterative Lösung großer schwach besetzter Gleichungssysteme.* Leitfäden der angewandten Mathematik und Mechanik Band 69. B. G. Teubner Stuttgart 1991.

[3] Maess, G.: *Vorlesungen über numerische Mathematik.* Band 1, 2. Akademie-Verlag Berlin 1984, 1988.

[4] Schwarz, H. R.: 1. *Methode der finiten Elemente.* Leitfäden der angewandten Mathematik und Mechanik Band 47. B. G. Teubner Stuttgart 1991.
2. *FORTRAN-Programme zur Methode der finiten Elemente.* B. G. Teubner Stuttgart 1991.

[5] Zlatev, Z.: *Computational Methods for General Sparse Matrices.* Math. and Its Appl. Vol.65. Kluwer Academic Publishers London 1991.

[6] Gustavson, F.: *A Survey of Some Sparse Matrix Theory and Techniques.* Jahrbuch Überblicke Mathematik. B.I.-Wissenschaftsverlag Mannheim 1981.

[7] Schmauder, M.; Weiss, R.; Schönauer, W.: *The CADSOL Program Package* (Version 1.1). Interner Bericht Nr. 46/92, RZ der Universität Karlsruhe 1992.

[8] Schwetlick, H.; Kretzschmar, H.: *Numerische Verfahren für Naturwissenschaftler und Ingenieure.* Fachbuchverlag Leipzig 1991.

[9] Bramdler, A.; Allan, R. N.; Hamann, Y. M.: *Sparsity.* Pitman Publishing London 1976.

[10] Schendel, U.: *Sparse Matrizen.* Oldenbourg Verlag München/Wien 1976.

[11] Köckler, N.: *Numerische Algorithmen in Softwaresystemen : unter besonderer Berücksichtigung der NAG-Bibliothek.* B. G. Teubner Stuttgart 1990.

[12] Rice, J. R.: *Numerical Methods, Software and Analysis.* 2nd Edition. Academic Press Inc. Boston 1993.

[13] Govaerts, W.; Pryce, J. D.: *Mixed block elimination for linear systems with wide borders.* IMA Journ. of Numerical Analysis (1993)13, 161-180.

[14] Collins, R. J.: *Bandwith Reduction by Automatic Renumbering.* Int. Journ. Num. Methods in Engineering 6(1973) 345-356.

[15] Gibbs, N. E.; Poole, W. G.; Stockmeyer, P. K.: *An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix.* SIAM Journ. Numerical Analysis 13(1976)2, 236-250.

[16] Berger, C.: *Entwurf und Implementierung dünn besetzter Blockmatrizen in C++.* Diplomarbeit TU München IfI 1994.

[17] Spiess, J.: *Untersuchungen des Zeitgewinns durch neue Algorithmen zur Matrix-Multiplikationen.* Computing 17, 23-36 (1976).

[18] Neundorf, W.; Ortlepp, T.: *Berechnung von Matrix-Multiplikationen auf dem PC.* Preprint No. M 15/95, TUI lmenau IfMath August 1995.

[19] Neundorf, W.: *Pascal-Programm* INV_AUSP.PAS .
Invertierung einer quadratischen Matrix mittels Austauschverfahren mit Spaltenpivotsuche und Zeilenvertauschung (Gauß-Jordan) sowie mit Pointertechnik. TU Ilmenau 1995.

[20] Neundorf, W.; Hohlbein, D.: *Pascal-Programm* MATVEK1.PAS .
Kompaktspeicherung einer Matrix und Anwendung bei Matrix-Vektor-Multiplikation. TU Ilmenau 1996.

[21] Samarskij, A. A.: *Theorie der Differenzenverfahren.* Akademische VG Geest & Portig K.-G. Leipzig 1984.

[22] Neundorf, W.: *Behandlung großer Matrizen auf dem PC.* Preprint No M 11/96 Juni 1996 IfMath TU Ilmenau.

[23] Neundorf, W.: *Manipulation von Matrizen I.* Preprint No M 16/96 November 1996 IfMath TU Ilmenau.

[24] Neundorf, W.; Ortlepp, T.: *Unvollständige LU-Zerlegung und approximative Inverse von Blocktridiagonalmatrizen.* Preprint No M 5/97 Juni 1997 IfMath TU Ilmenau.

[25] Neundorf, W.: *MATLAB - Teil I. Vektoren, Matrizen und lineare Gleichungssysteme.* Preprint No M 20/99 Juli 1999 IfMath TU Ilmenau.

[26] Neundorf, W.; *MATLAB - Teil II. Speicheraspekte, spezielle LGS, SVD, EWP, Graphik, NLG, NLGS.* Preprint No M 23/99 September 1999 IfMath TU Ilmenau.

[27] Neundorf, W.:*MATLAB - Teil III. Komplexe LGS, Interpolation, Splines.* Preprint No M 10/00 Mai 2000 IfMath TU Ilmenau.

[28] Meister, A.: *Numerik linearer Gleichungssysteme.* Eine Einführung in moderne Verfahren. Friedr. Vieweg & Sohn VG mbH, Braunschweig 1999.

[29] Meis, Th.; Marcowitz, U.: *Numerische Behandlung partieller Differentialgleichungen.* Springer-Verlag Berlin 1978.

[30] Slavkovsky, P.; Rüde, U.: *Schnellere Berechnung klassischer Matrix-Multiplikationen.* Preprint TUM-I9032, SFB-Bereicht Nr. 342/17/90 A, Mn- chen September 1990.

[31] Bonk, T.; Rüde, U.: *Perfomance Analysis and Optimization of Numerically Intensive Programs.* Preprint TUM-I9238, SFB-Bereicht Nr. 342/26/92 A, München November 1992.

[32] Stoer, J.: *Einführung in die Numerische Mathematik. Band 1.* Springer-Verlag Berlin 1979, 1989.

[33] Spiess, J.: *Untersuchungen des Zeitgewinns durch neue Algorithmen zur Matrix-Multiplikationen.* Computing 17, 23-36 (1976).

[34] Ortlepp, T.: *Schnellere Berechnung der klassischen Matrix-Multiplikation auf PC.* Beleg IfMath TU Ilmenau 1995.

[35] Schwarz, H. R.: *Numerische Mathematik.* B. G. Teubner Stuttgart 1988.

[36] Donner, K.: *Skalierung von Matrizen und numerische Stabilität der Gauß-Elimination.* Preprint Universität Passau, MIP-9514 September 1995.

[37] Bauer, F. L.: *Optimally scaled matrices.* Numer. Mathematik 5(1963)73-87.

[38] Wilkinson, J. H.; Reinsch, C.: *Linear Algebra.* Handbook for automatic computation, Vol. II. Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen, Bd. 186. Berlin-Heidelberg-New York 1971.

[39] Bruaset, A. M.: *A survey of preconditioned iterative methods.* Pitman Research Notes in Mathematics Series 328. Longman Scientific & Technical Essex, John Wiley & Sons, Inc., New York 1995.

[40] Schaback, R; Werner, H.: *Numerische Mathematik.* Springer-Verlag Berlin 1993.

[41] Überhuber, C.: *Computer-Numerik 1,2.* Springer-Verlag Berlin 1995.

[42] Deuflhard, P.; Hohmann, A.: *Numerische Mathematik.* De Gruyter-Verlag Berlin New York 1991.

[43] Ralston, A.: *A First Course in Numerical Analysis.* McGraw-Hill New York 1965.

[44] Cherkasova, M. P.: *Collected Problems in Numerical Methods.* Akademie-Verlag Berlin 1972.

[45] Schwarz, H. R.; Rutishauser, H.; Stiefel, E.: *Numerik symmetrischer Matrizen.* Leitfäden der angewandten Mathematik, Bd. 11. Stuttgart 1968, B. G. Teubner VG Leipzig 1969.

[46] Jankowska, J.; Jankowski, M.: *Przegląd metod i algorytmów numerycznych.* Band 1. WNT Warszawa 1981.

[47] Hämmerlin, G.; Hoffmann, K.-H.: *Numerische Mathematik.* Grundwissen Mathematik 7. Springer-Verlag Berlin 1991.

[48] Stoer, J.; Burlisch, R.: *Einführung in die Numerische Mathematik II.* 3. Aufl. Springer-Verlag Berlin 1990.

[49] Axelsson, O.: *Iterative Solution Methods.* Cambridge University Press 1994.

[50] Engeln-Müllges, G.; Reutter, F.:
1. *Formelsammlung zur Numerischen Mathematik mit FORTRAN 77-Programmen.* Bibliogr. Institut Mannheim 1988.
2. *Formelsammlung zur Numerischen Mathematik mit Turbo Pascal-Programmen.* BI-Wissenschaftsverlag Mannheim 1991.

[51] Engeln-Müllges, G.; Reutter, F.: *Numerik-Algorithmen mit ANSI C-Programmen.* (auch für Turbo Pascal, FORTRAN). BI-Wissenschaftsverlag Mannheim 1993.

[52] Kose, K.; Schröder, R.; Wieliczek, K.: *Numerik sehen und verstehen.* Ein kombiniertes Lehr- und Arbeitsbuch mit Visualisierungssoftware. Vieweg Braunschweig 1992.

[53] Zurmühl, R.; Falk, S.: *Matrizen und ihre Anwendungen.* Teil 2, Numerische Mehtoden. Springer-Verlag Berlin 1984.

[54] Dietel, J.: *Formelsammlung zu Numerischen Mathematik mit Turbo Pascal-Programmen* (TPNUM). Rechenzentrum der RWTH Aachen 1993.

[55] Plato, R.: *Numerische Mathematik kompakt.* Grundlagenwissen für Studium und Praxis. Vieweg Wiesbaden 2000.

[56] Cuthill, E.: *Several strategies for reducing the band width of matrices.* In: Rose, D. J.; Willoughby, R.A. (ed.): *Sparse matrices and their applications.* Plenum, New York 1972, 157-166.

[57] Cuthill, E.; McKee, J.: *Reducing the bandwidth of sparse symmetric matrices.* In: Proc. ACM Nat. Conf., New York 1969, 157-172.

[58] Liu, W. H.; Sherman, A. H.: *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee algorithms for sparse matrices.* SIAM J. Numer. Anal. 13 (1976) 198-213.

[59] King, I. P.: *An automatic reorderimg scheme for simultaneous equations derived from network systems.* Intern. J. Numer. Meth. Engrg. 2 (1970) 523-533.

[60] Hendrich, U.: *Über das Bandbreitenproblem für Produkte zweier Graphen.* Diplomarbeit IfMath TU Ilmenau 1989.

[61] Rosen, R.: *Matrix bandwidth minimization.* Proc. 23rd ACM National Conf., 585-595, 1968.

[62] Lierz, W.: *Lösung von großen Gleichungssystemen mit symmetrischer schwach besetzter Matrix.* Diplomarbeit Universität Köln 1975.

[63] Neundorf, W.: *Wissenschaftliches Rechnen - Matrizen und lineare Gleichungssysteme.* Vorlesungsskript IfMath der TU Ilmenau, August 2002.

Anschrift:

Dr. rer. nat. habil. Werner Neundorf
Technische Universität Ilmenau, Institut für Mathematik
PF 10 05 65
D - 98684 Ilmenau

E-mail : werner.neundorf@tu-ilmenau.de
Homepage : http://www.mathematik.tu-ilmenau.de/~neundorf/index_de.html