Algorithmen zur parallelen Determinantenberechnung

Holger Burbach

Oktober 1992

Diplomarbeit, geschrieben am Lehrstuhl Informatik II
der Universität Dortmund

Betreut von Prof. Ingo Wegener

Inhaltsverzeichnis

Kapitel 1

Vorbemerkungen

1.1 Einführung

Die Berechnung der Determinante einer quadratischen Matrix ist ein Problem, dessen effiziente Lösung in vielen Bereichen von Interesse ist, in der Informatik z. B. in der Computergrafik und der Kodierungstheorie.

Ein Problem in der analytischen Geometrie ist es, die Lage von geometrischen Objekten zueinander festzustellen und Schnittpunkte oder -ebenen zu berechnen. Ein Teilproblem dabei ist die Prüfung der linearen Unabhängigkeit von Vektoren. Es läßt sich auf die Berechnung einer Determinante zurückführen.

Ein weiterer Bereich, in dem die Determiantenberechnung angewendet wird, ist die theoretische Physik. Dort wird in vielen Theorien auf Matrizen zu Beschreibung der verschiedenen Sachverhalte zurückgegriffen. In der Einstein'sche Relativitätstheorie z. B. wird die *Tersorrechnung*, die die Eigenschaften sich von Koordinatensystem zu Koordinatensystem ändernder Maßzahlen untersucht, ausgibig verwendet. Die Determinante ist eine solche Maßzahl. Beim Studium von Literatur, die diese Thematiken behandelt (z. B. [BS86] ab S. 70), stößt man immer wieder auf Matrizen und ihre Determinanten.

Seitdem sich die Forschung im Bereich der Informatik zunehmend mit Parallelrechnern beschäftigt, werden für alle bekannten Probleme Algorithmen gesucht, die die Tatsache, daß auf einem Parallelrechner mehrere Prozessoren gleichzeitig an der Lösung desselben Problems arbeiten, besonders effizient ausnutzen.

Betrachtet man eine Matrix aus der Sicht der Informatik als Datenstruktur, so drängt sich die Benutzung dieser Datenstruktur in Parallelrechnern geradezu auf, denn intuitiv, ohne zunächst alle Probleme ausgearbeitet zu haben, kann man auf die Idee kommen, die Matrizenelemente jeweils einzelnen Prozessoren oder Gruppen von Prozessoren zuzuordnen, die das zugrundeliegende Problem für dieses Matrizenelement bearbeiten. Selbstverständlich ist die praktische Verwendung dieser Idee nicht in jedem Fall ganz so einfach.

So war die effiziente Parallelisierung der Determinantenberechnung lange Zeit ein ungelöstes Problem, bis 1976, als Laszlo Csanky einen in jenen Tagen überraschenden Algorithmus veröffentlichte [Csa76]. Es folgten eine Reihe weiterer Algorithmen verschiedener Autoren mit vergleichbaren Leistungsmerkmalen.

In all diesen Veröffentlichungen wird vorrangig die Größenordnung der Laufzeiten und Anzahlen der Prozessoren betrachtet. Es werden in einzelnen Veröffentlichungen auch bereits einige Vergleiche mit den anderen Algorithmen durchgeführt. So ist es wünschenswert einen Überblick über die existierenden Algorithmen zu bekommen und sie insgesamt miteinander zu vergleichen.

Die vorliegende Diplomarbeit behandelt vier Algorithmen zur parallelen Determinantenberechnung[1]. Dabei wird auf die Verwendung von Größenordnungen (O-Notation) in Aufwandsana-

[1] [Csa76], [BvzGH82], [Ber84] und [Pan85]

lysen weitgehend verzichtet. Um den Einsteig in das Thema zu erleichtern, wird zusätzlich noch der Entwicklungssatz von Laplace zur Berechnung der Determinante erwähnt.

Die Darstellung der vier Algorithmen in den zugehörigen Kapiteln 3 bis 6 umfaßt neben den Grundlagen und der Algorithmen selbst, jeweils eine Analyse der Rechenzeit und des Grades der Parallelisierung[2]. Da in der Praxis die Größe des benötigen Speicherplatzes kein vorrangiges Problem mehr darstellt, wird dieser Wert nicht analysiert. Matrizen mit Elementen aus $\mathbb{C}$ werden nicht betrachtet. In diesen vier Kapitel wird versucht, auf Unterschiede und Gemeinsamkeiten der Algorithmen einzugehen.

Im Anschluß an die Darstellung der Algorithmen wird in Kapitel 7 ihre Implementierung beschrieben. Die Quelltexte sind im Anhang zu finden. Schließlich erfolgt in Kapitel 8 ein zusammenfassender Vergleich der Algorithmen.

Der Text soll es ermöglichen, die Algorithmen ohne weitere Literatur anhand von Grundkenntnissen aus der Mathematik und Informatik zu verstehen. Aus diesem Grund und um einheitliche Bezeichnungen zu vereinbaren sind Grundlagen, insbesondere aus der Linearen Algebra, an den benötigten Stellen aufgeführt. Für den Fall, daß die im Text enthaltenen Informationen nicht ausreichen, sind die benutzten Quellen an den jeweiligen Stellen angegeben.

Alle Betrachtungen abstrahieren von technischen Problemen bei der Konstruktion von Parallelrechnern. Dazu wird das Rechnermodell der PRAM benutzt. Die Beschreibung dieses Modells erfolgt in Kapitel 1.2.

Vor anderen Teilen dieser Diplomarbeit sollten zunächst die Kapitel 1.2 und 1.3 gelesen werden. Alle weiteren Teile von Kapitel 1 sowie Kapitel 2 sind als Sammlung von Grundlagen zu verstehen, auf die bei Bedarf zurückgegriffen werden kann[3].

1.2 Das Berechnungsmodell

In diesem Kapitel wird der für Komplexitätsbetrachtungen verwendete Modellrechner beschrieben. Es ist die *Arbitrary Concurrent Read Concurrent Write Parallel Random Access Machine (arbitrary CRCW PRAM)* . Sie besteht aus gleichen Prozessoren, die alle auf denselben Arbeitsspeicher zugreifen. Innerhalb einer Zeiteinheit können diese Prozessoren, und zwar alle gleichzeitig, zwei Operanden aus dem Speicher lesen, eine der in Tabelle 1.1 aufgeführten Operationen ausführen und das Ergebnis wieder im Speicher ablegen. Falls beim Schreiben mehrere Prozessoren auf eine Speicherzelle zugreifen, muß der Algorithmus unabhängig davon korrekt sein, welcher Prozessor seinen Schreibzugriff tatsächlich ausführt. Da die Komplexität der vier

Operation	Symbol
Addition	$+$
Subtraktion	$-$
Multiplikation	$*$
Division (Ergebnis in $\mathbb{Q}$)	$/$
Division (Ergebnis in $\mathbb{Z}$)	div
$x-(x \text{ div } y)*y$	$x \bmod y$

Tabelle 1.1: Operationen des Modellrechners

hauptsächlich interessierenden Algorithmen nicht nur auf ihre Größenordnung hin untersucht wird, sondern die Ausdrücke zur Beschreibung der Komplexität genau angegeben werden sollen, ist es erforderlich, von Details der Implementierung, die die Konstanten beeinflussen, zu abstrahieren, so daß die Aussagen allgemeingültig sind. Aus diesem Grund wird

[2] Diese Begriffe sind in Kapitel 1.3 definiert.

[3] Im Text kommen häufig Punkte und Kommata als Satzzeichen direkt im Anschluß an abgesetzte Gleichungen vor. An einigen Stellen, besonders hinter Vektoren und Matrizen, fehlen diese Satzzeichen aus technischen Gründen.

- für die Verarbeitung von Schleifenbedingungen,
- für die Verarbeitung von Verzweigungsbedingungen,
- für die Ein- und Ausgabe von Daten,
- für die Initialisierung von Speicherbereichen,
- und für komplexe Adressierungsarten (z. B. indirekte Adressierung) beim Zugriff auf Speicherbereiche

kein zusätzlicher Aufwand in Rechnung gestellt. Es werden also nur die arithmetischen Operationen gezählt.

Zu beachten ist, daß bei einer PRAM jeder Aufwand zur Verteilung von Aufgaben auf verschiedene Prozessoren vernachlässigt wird. Diese Eigenschaft bietet die Möglichkeit zur Kritik, da so jedes Problem deutlich vereinfacht wird, jedoch in der Praxis die Organisation der Aufgabenverteilung nicht unerheblichen Aufwand erfordert. Eine genaue Analyse der Auswirkungen dieser Vernachlässigung ist umfangreich und nicht Thema des vorliegenden Textes.

1.3 Bezeichnungen

In diesem Kapitel werden die verwendeten Begriffe und Symbole definiert.

Um auf die in der Arbeit hauptsächlich behandelten Algorithmen einfach Bezug nehmen zu können, werden mit Hilfe der Namen ihrer Autoren die folgenden Abkürzungen vereinbart:

- C-Alg. steht für den Algorithmus von Csanky (Unterkapitel 3.8 ab S. 40).
- BGH-Alg. steht für den Algorithmus von Borodin, von zur Gathen und Hopcroft (Unterkapitel 4.6 ab S. 59).
- B-Alg. steht für den Algorithmus von Berkowitz (Unterkapitel 5.3 ab S. 72).
- P-Alg. steht für den Algorithmus von Pan (Unterkapitel 6.7 ab S. 99).

Die Menge der positiven ganzen Zahl *ohne Null* wird mit

$$\mathbb{N}$$

bezeichnet. Die Menge der positiven ganzen Zahl einschließlich der Null wird mit

$$\mathbb{N}_0$$

bezeichnet. Falls nicht im Einzelfall anders festgelegt erfolgen alle Darstellungen von Zahlen zur Basis 10.

Der Vorgang, in dem beliebig viele Prozessoren gleichzeitig je zwei Operanden aus dem Arbeitsspeicher lesen, aus diesen Operanden ein Ergebnis berechnen und dieses Ergebnis wieder im Arbeitsspeicher ablegen, wird als ein *Schritt* bezeichnet.

Die *parallele Zeitkomplexität eines Algorithmus* bezeichnet die Anzahl der Schritte, die dieser benötigt, um die Lösung[4] für das zugrunde liegende Problem zu berechnen. Die maximale Anzahl der Prozessoren, die dabei gleichzeitig beschäftigt werden, wird mit *Parallelisierungsgrad des Algorithmus* bezeichnet.

Falls nicht im Einzelfall anders festgelegt, gilt folgende Regelung: Großbuchstaben bezeichnen Matrizen und Kleinbuchstaben Zahlen oder Vektoren, A bezeichnet eine $n \times n$-Matrix, indizierte Kleinbuchstaben beziehen sich auf die Elemente der mit dem zugehörigen Großbuchstaben bezeichneten Matrix.

Die in Tabelle 1.2 aufgelisteten Schreibweisen werden benutzt.

[4] Die von uns betrachteten Probleme besitzen nur eine Lösung.

Begriff	Schreibweise
Element in Zeile i, Spalte j von A	$a_{i,j}$
i-tes Element des Vektors v	v_i
Matrix, die aus A durch Streichen der Zeilen v und der Spalten w entsteht (dabei seien v und w echte Teilmengen der Menge der Zahlen von 1 bis n; diese Mengen werden hier als durch Kommata getrennt Zahlenfolge geschrieben)	$A_{(v\|w)}$
Einheitsmatrix (Elemente der Hauptdiagonalen gleich 1; alle anderen Elemente gleich 0) mit n Zeilen und Spalten	E_n
Einheitsmatrix (Anzahl der Zeilen und Spalten aus dem Zusammenhang klar)	E
Nullmatrix (alle Elemente sind gleich 0) mit m Zeilen und n Spalten	$0_{m,n}$
Nullvektor (alle Elemente sind gleich 0) der Länge m	0_m
Logarithmus von x zur Basis 2	$\log(x)$
Menge aller n-stelligen Permutationen	$\mathcal{S}_n$
$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n \qquad (=2.718281\ldots)$	e
Logarithmus von x zur Basis e	$\ln(x)$
Anzahl der Elemente der Menge M	$\|M\|$

Tabelle 1.2: Bezeichnungen

1.4 Das Präfixproblem

Von einer effizienten Lösung des Präfixproblems wird an verschiedenen Stellen Gebrauch gemacht. Es ist also von übergreifendem Interesse und wird deshalb hier behandelt ([LF80], [Weg89] S. 83 ff.). Es läßt sich folgendermaßen formulieren:

> Gegeben sei die Halbgruppe
> $$(M, \circ) \ .$$
> D. h. die Verknüpfung $\circ$ ist assoziativ auf M. Weiterhin seien
> $$x_1, x_2, x_3, \ldots, x_n$$
> Elemente aus M. Es wird definiert
> $$p_i := x_1 \circ x_2 \circ x_3 \circ \ldots \circ x_i \ .$$
> Das Präfixproblem besteht darin, alle Elemente der Menge
> $$\{p_i | 1 \leq i \leq n\}$$
> zu berechnen.

Es sind u. a. zwei Möglichkeiten[5] denkbar, dies mit parallelen Algorithmen zu erreichen.

- Die erste Möglichkeit:
  1. Löse das Präfixproblem parallel für
     $$x_1, \ldots, x_{\lceil n/2 \rceil}$$
     und
     $$x_{\lceil n/2+1 \rceil}, \ldots, x_n \ ,$$
     so daß nach diesem Schritt
     $$p_1, \ldots, p_{\lceil n/2 \rceil}$$
     bereits berechnet sind.

[5] die sich zu einer dritten zusammenfassen lassen (Satz 1.4.1)

2. Berechne aus
$$p_{\lceil n/2 \rceil}$$
und der Lösung des Problems für
$$x_{\lceil n/2+1 \rceil}, \ldots, x_n$$
parallel in einem weiteren Schritt
$$p_{\lceil n/2+1 \rceil}, \ldots, p_n$$

- Die zweite Möglichkeit, die hier kurz dargestellt werden soll, sieht folgendermaßen aus (o. B. d. A. sei n eine Zweierpotenz):
  1. Berechne parallel in einem Schritt
  $$x_1 \circ x_2, x_3 \circ x_4, \ldots, x_{n-1} \circ x_n$$
  2. Löse das Präfixproblem für diese $n/2$ Werte. Damit werden alle p_i mit geradem i berechnet.
  3. Die noch fehlenden p_i für ungerade i können nun parallel in einem weiteren Schritt aus der Lösung für die $n/2$ Werte und den x_i mit ungeradem i berechnet werden.

Diese beiden Möglichkeiten können zu einem Algorithmus zusammengefaßt werden:

Satz 1.4.1 *Gegeben sei die Halbgruppe*
$$(M, \circ) \ .$$
Das Präfixproblem für n Elemente
$$x_1, x_2, \ldots, x_n$$
von M läßt sich von
$$\left\lfloor \frac{3}{4} n \right\rfloor$$
Prozessoren in
$$\lceil \log(n) \rceil$$
Schritten lösen.

Beweis O. B. d. A. sei n eine Zweierpotenz. In dem Fall, daß n keine Zweierpotenz ist, wird n durch die nächst höhere Zweierpotenz n' ersetzt und alle Verknüpfungen mit Elementen x_i von M für
$$i > n$$
werden nicht durchgeführt.

Benutze folgenden Algorithmus:

1. Wenn
$$n = 1$$
dann ist x_1 das Ergebnis.
2. Wenn
$$n = 2$$
dann ist
$$x_1, x_1 \circ x_2$$
das Ergebnis.
3. Schritte 3a und 3b parallel:

(a) i. Berechne parallel in einem Schritt

$$x_1 \circ x_2, \ldots, x_{n/2-1} \circ x_{n/2}$$

ii. Benutze den Algorithmus rekursiv zur Lösung des Problems für die in Schritt 3(a)i erhaltenen $n/4$ Werte. Auf diese Weise sind die p_i für

$$1 \leq i \leq n/2$$

mit geraden i, u. a. auch $p_{n/2}$, bereits berechnet.

(b) Benutze den Algorithmus rekursiv zur Lösung des Problems für

$$x_{n/2+1}, \ldots, x_n$$

4. Schritte 4a und 4b parallel:

(a) Für i gelte

$$1 \leq i \leq n/2 \ .$$

Wenn $n/2 > 2$, dann berechne parallel in einem Schritt mit Hilfe der p_i aus 3(a)ii und der x_i mit ungeradem i die fehlenden p_i mit ungeradem i.

(b) Berechne aus $p_{n/2}$ und den Ergebnissen von 3b die p_i mit

$$n/2 + 1 \leq i \leq n \ .$$

Zur Analyse des Algorithmus bezeichnet $s(n)$ die Anzahl der Schritte, die er benötigt, um das Präfixproblem für n Eingabewerte zu lösen, und $p(n)$ die Anzahl der Prozessoren, die dabei beschäftigt werden können.

- Hier wird zunächst die Anzahl der Schritte betrachtet. Es gilt

$$s(1) = 0,\ s(2) = 1,\ s(4) = 2 \ .$$

Bei der Betrachtung des Algorithmus erkennt man, daß folgende Rekursionsgleichung Gültigkeit besitzt:

$$\forall n > 4 :\ s(n) = \max(s(n/4) + 1,\ s(n/2)) + 1 \ . \tag{1.1}$$

Wenn man diese Formel auf $s(n/2)$ anwendet und das Ergebnis in die obige Formel einsetzt, erhält man

$$\forall n > 4 :\ s(n) = \max(\ s(n/4) + 1,\ \max(s(n/8) + 1,\ s(n/4)) + 1\ )\ + 1$$

Aufgrund der Assoziativität der max-Funktion ist dies gleichbedeutend mit

$$\begin{aligned} & \forall n > 4 :\ s(n) &&= \max(s(n/4) + 1,\ s(n/8) + 2,\ s(n/4) + 1) + 1 \\ \Rightarrow\ & \forall n > 4 :\ s(n) &&= s(n/4) + 2 \end{aligned}$$

Es gilt also für jedes i:

$$\forall n > 4 :\ s(n) = s(n/2^{2i}) + 2i$$

Mit

$$i = \frac{\log(n)}{2}$$

erhält man als Endergebnis

$$s(n) = \log(n) \ .$$

- Für die Anzahl der beschäftigten Prozessoren $p(n)$ gilt:

$$p(1) = 0, p(2) = 1, p(4) = 2$$

Ferner gilt offensichtlich folgende Rekursionsgleichung:

$$\forall n > 4 : \; p(n) = \max(n/2 + n/4,\, p(n/4) + p(n/2),\, n/4 + p(n/2)) \tag{1.2}$$

Beim Ausrechnen der Werte von $p(8), p(16)$ und $p(32)$ mit Hilfe dieser Rekursionsgleichung gelangt man zu der Vermutung, daß gilt:

$$\forall n > 4 : \; p(n) = \frac{3}{4}n \tag{1.3}$$

Dies wird durch Induktion bewiesen. Zu beachten ist, daß nach Voraussetzung nur die Potenzen von 2 als Werte für n in Frage kommen.

Sei also nun

$$n > 4$$

und es gelte

$$\begin{aligned} p(n) &= \frac{3}{4}n \\ p(n/2) &= \frac{3}{8}n \; . \end{aligned}$$

Es ist zu zeigen, daß dann auch

$$p(2n) = \frac{3}{2}n$$

richtig ist. Nach der Rekursionsgleichung (1.2) gilt

$$p(2n) = \max(3/2 * n, p(n/2) + p(n), n/2 + p(n))$$

Die Anwendung der Induktionsvoraussetzung führt zu

$$p(2n) = \max(3/2 * n, 3/8 * n + 3/4 * n, n/2 + 3/4 * n)$$

und somit zu

$$p(2n) = \frac{3}{2}n$$

was zu zeigen war. Für die Anzahl der benötigen Prozessoren gilt also

$$\forall n > 4 : \; p(n) = \frac{3}{4}n$$

Da die Lösung des Problems für

$$n \leq 4$$

einfach ist, wird die Quantifizierung nicht weiter beachtet.

Damit die Aussagen nicht nur für Zweierpotenzen, werden die Werte für $s(n)$ und $p(n)$ mit Gaußklammern versehen. Für $p(n)$ ist dies ohne weitere Begründung problematisch. Betrachtet man den Algorithmus jedoch genauer, stellt man fest, daß beim ersten ausgeführten Schritt die meisten Prozessoren beschäftigt werden. Die Anzahl dieser Prozessoren gibt der Term in Gaußklammern an. □

1.5 Lösungen grundlegender Probleme

In diesem Kapitel werden Algorithmen zur Lösung einiger grundlegender Probleme angegeben und auf ihre Komplexität hin untersucht.

Satz 1.5.1 (Binärbaummethode) *Wird das Präfixproblem (siehe Beschreibung Seite 10) dahingehend vereinfacht, daß nur p_n zu berechnen ist, so läßt sich dieses vereinfachte Problem in*

$$\lceil \log(n) \rceil$$

Schritten von

$$\left\lfloor \frac{n}{2} \right\rfloor$$

Prozessoren lösen.

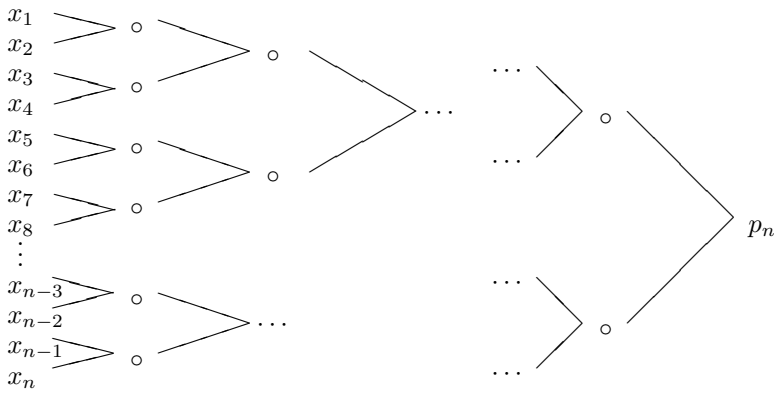

Abbildung 1.1: Binärbaummethode

Beweis Verknüpfe die x_i nach dem Schema in Abbildung 1.1. Falls n keine Zweierpotenz ist, werden die Verknüpfungen mit den x_j, für die gilt

$$n < j \leq 2^{\lceil \log(n) \rceil} \ ,$$

nicht durchgeführt. Die Lösung des Problems erfordert offensichtlich den angegebenen Aufwand. □

Folgerung 1.5.2 (Parallele Grundrechenarten) *Seien n Zahlen durch die gleiche Rechenoperation miteinander zu verknüpfen. Diese Rechenoperation sei eine der Grundrechenarten Addition oder Multiplikation. Die Verknüpfung kann in*

$$\lceil \log(n) \rceil$$

Schritten von

$$\left\lfloor \frac{n}{2} \right\rfloor$$

Prozessoren durchgeführt werden.

Beweis Aufgrund der Assoziativität der Rechenoperationen folgt dies direkt aus Satz 1.5.1. □

Satz 1.5.3 (Parallele Matrizenmultiplikation) *Sei A eine $m \times p$-Matrix und B eine $p \times n$-Matrix. Sie lassen sich in*

$$\lceil \log(p) \rceil + 1$$

Schritten von

$$m * n * p$$

Prozessoren miteinander multiplizieren.

Beweis Sei C die $m \times n$-Ergebnismatrix. Sie wird mit Hilfe der Gleichung

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j}$$

berechnet. Dazu werden zuerst parallel in einem Schritt

$$d_{i,k,j} := a_{i,k} b_{k,j}$$

mit

$$\begin{aligned} 1 \leq \quad & i \quad \leq m \\ 1 \leq \quad & j \quad \leq n \\ 1 \leq \quad & k \quad \leq p \end{aligned}$$

von

$$m * n * p$$

Prozessoren berechnet. Die Ergebnismatrix erhält man dann nach der Gleichung

$$c_{i,j} = \sum_{k=1}^{p} d_{i,k,j}$$

Die Berechnung der Matrix C aus den $d_{i,k,j}$ kann nach 1.5.2 für ein Matrizenelement in

$$\lceil \log(p) \rceil$$

Schritten von

$$\left\lfloor \frac{p}{2} \right\rfloor$$

Prozessoren durchgeführt werden, also für die gesamte Matrix in genauso vielen Schritten von

$$m * n * \left\lfloor \frac{p}{2} \right\rfloor$$

Prozessoren. Die Werte für Schritte und Prozessoren zusammengenommen ergeben die Behauptung. □

Zwei $n \times n$-Matrizen lassen sich also in

$$\lceil \log(n) \rceil + 1$$

Schritten von

$$n^3$$

Prozessoren miteinander multiplizieren.

Die Matrizenmultiplikation läßt sich asymptotisch, d. h. für $n \to \infty$ auch mit

$$O(n^{2+\gamma}),\ \gamma = 0.376$$

Prozessoren durchführen [CW90]. Gegenüber 1.5.3 ergibt sich wegen des erheblichen konstanten Aufwandes nur für große n eine Verbesserung. Es wird jeweils gesondert darauf hingewiesen, falls auf diese Möglichkeit zurückgegriffen wird.

Kapitel 2

Grundlagen aus der Linearen Algebra

In diesem Kapitel werden die für den gesamten weiteren Text wichtigen Begriffe und Sätze aus der Linearen Algebra behandelt. Falls in späteren Kapiteln an einzelnen Stellen weitergehende Grundlagen insbesondere aus anderen Bereichen nötig sind, werden diese an den jeweiligen Stellen behandelt.

Da es sich bei dem Inhalt dieses Kapitels um Grundlagen handelt, sind einige Beweise etwas oberflächlicher bzw. fehlen ganz.

Literatur:

- [MM64] Kapitel 1 und 2
- [Dö77] Kapitel 6, 9 und 12
- [BS87] ab Seite 148

Im folgenden sind A und B $n \times n$-Matrizen. Für uns reichen Betrachtungen im Körper der rationalen Zahlen aus.

2.1 Matrizen und Determinanten

In diesem Kapitel werden die grundlegendsten Begriffe über Matrizen und Determinanten aufgeführt, um eine Grundlage für den weiteren Text zu vereinbaren.

Definition 2.1.1 A heißt *invertierbar*, wenn es eine Matrix B gibt, so daß

$$AB = BA = E_n$$

In diesem Fall heißt B *Inverse von* A und wird auch mit

$$A^{-1}$$

bezeichnet.

Definition 2.1.2 Falls für die Matrizen A und B gilt

$$b_{i,j} = a_{j,i}$$

so heißt B *Transponierte von* A. Für die Transponierte von A wird auch A^T geschrieben.

Definition 2.1.3

$$\mathrm{tr}\,(A) := \sum_{i=1}^{n} a_{i,i}$$

heißt *Spur der Matrix A*.

Definition 2.1.4 Eine bijektive Abbildung

$$f : \{1, \ldots, n\} \to \{1, \ldots, n\}$$

heißt *n-Permutation*. Sei

$$1 \leq i < j \leq n$$

Falls gilt

$$f(i) > f(j) \ ,$$

so heißt diese Bedingung *Inversion der n-Permutation*.

$$\mathcal{S}_n$$

bezeichnet die Menge aller n-Permutationen. Zusammen mit der Konkatenation von Abbildungen bildet sie eine Gruppe, die *symmetrische Gruppe* $\mathcal{S}_n$.

Definition 2.1.5 Sei f eine n-Permutation. Dann heißt

$$\mathrm{sig}(f) := \prod_{1 \leq i < j \leq n} \frac{f(i) - f(j)}{i - j} \tag{2.1}$$

Signatur von f.

Die so definierte Signatur besitzt folgende Eigenschaften:

- Es gilt
$$\forall f \in \mathcal{S}_n : \mathrm{sig}(f) \in \{1, -1\}$$
Aus der Permutationseigenschaft ergibt sich, daß es für jede Differenz, die als Faktor im Zähler von (2.1) auftaucht, eine Differenz im Nenner mit dem gleichen Betrag existiert, so daß der Wert des gesamten Produktes den Betrag 1 besitzt. Das Vorzeichen wird durch die Anzahl der Inversionen beeinflußt.

- Falls die Anzahl der Inversionen der n-Permutation f gerade ist, so gilt
$$\mathrm{sig}(f) = 1 \ ,$$
andernfalls gilt
$$\mathrm{sig}(f) = -1$$

- Die Anzahl der n-Permutationen f mit
$$\mathrm{sig}(f) = 1 \ ,$$
ist gleich der Anzahl der n-Permutationen g mit
$$\mathrm{sig}(g) = -1 \ .$$
([Dö77] Seite 196)

Die Permutationen mit

$$\operatorname{sig}(f) = 1$$

nennt man *gerade*, die anderen *ungerade*.

Definition 2.1.6 Seien

$$A, B \in \mathbb{Q}^{n^2}$$

Sei

$$\det : \mathbb{Q}^{n^2} \to \mathbb{Q}$$

eine Abbildung mit folgenden Eigenschaften:

D1
Entsteht B aus A durch Multiplikation einer Zeile mit

$$r \in \mathbb{Q}$$

so gilt:

$$\det(B) = r \det(A)$$

D2
Enthält A zwei gleiche Zeilen, so gilt:

$$\det(A) = 0$$

D3
Entsteht B aus A durch Addition des r-fachen einer Zeile zu einer anderen, so gilt:

$$\det(B) = \det(A)$$

D4
Für die Einheitsmatrix gilt:

$$\det(E_n) = 1$$

Dann heißt

$$\det(A)$$

Determinante der Matrix A.

Definition 2.1.7 Die auf einer Matrix definierten Operationen

- Vertauschung zweier Zeilen
- Multiplikation einer Zeile mit einem Faktor[1]
- Addition des Vielfachen einer Zeile zu einer anderen[2]

werden *elementare Zeilenoperationen* genannt. Die entsprechenden Operationen auf Matrizenspalten werden *elementare Spaltenoperationen* genannt.

Satz 2.1.8

$$g(A) = \sum_{f \in \mathcal{S}_n} \operatorname{sig}(f) a_{1,f(1)} a_{2,f(2)} \dots a_{n,f(n)} \tag{2.2}$$

besitzt die Eigenschaften aus 2.1.6.

[1] vgl. D1 in 2.1.6
[2] vgl. D3 in 2.1.6

Beweis Da es sich hier um Grundlagen handelt, wird der Beweis weniger ausführlich angegeben:

D1

Für jede Permutation kommt im entsprechenden Summanden in (2.2) aus jeder Zeile der Matrix genau ein Element als Faktor vor. Falls eine Zeile mit r multipliziert wurde, kann man also aus jedem Summanden r ausklammern und erhält die Behauptung.

D2

Seien Zeile i und Zeile j gleich ($i \neq j$). Berechne die Summe in (2.2) getrennt für die ungeraden und die geraden Permutationen.

Die n-Permutation g vertausche i mit j und lasse alles andere gleich. Aus den Grundlagen der Theorie der Halbgruppen und Gruppen ergibt sich, daß man die ungeraden n-Permutationen erhält, indem man die geraden n-Permutationen jeweils einzeln mit der Permutation g zusammen ausführt.

Deshalb entsprechen sich die Summanden der beiden Teilsummen paarweise und unterscheiden sich nur durch das Vorzeichen. Der Gesamtausdruck besitzt also den Wert 0.

D3

Es werde das r-fache von Zeile i zu Zeile j addiert. Dadurch enthält jeder Summand in (2.2) genau einen Faktor, der seinerseits wieder die Summe zweier Matrizenelemente ist. Deshalb kann man die gesamte Summe in zwei Summen aufteilen. Die eine entspricht genau der Summe in (2.2), die andere enthält in jedem Summanden zwei gleiche Faktoren, sowie den Faktor r. Diesen Faktor kann man ausklammern (mit Hilfe von D1). Nach D2 ist der Wert dieser Summe dann gleich Null, und nur die andere bleibt übrig.

D4

Außer für die identische Abbildung enthält jeder Summand der entsprechenden Permutation in (2.2) mindestens zwei Nullen als Faktoren und ist deshalb gleich Null. Der Summand, der der identischen Abbildung entspricht, hat den Wert 1.

□

2.2 Der Rang einer Matrix

Zum Verständnis des weiteren Textes wird in diesem Kapitel der Begriff des *Rangs* einer Matrix eingeführt. Da dieser Begriff für die Untersuchung linearer Gleichungssysteme wichtig ist, werden teilweise auch nichtquadratische Matrizen betrachtet[3] .

Da in diesem Kapitel wiederum Grundlagen aus der Linearen Algebra behandelt werden, ist die Darstellung auf die für uns wichtigen Aspekte beschränkt.

Definition 2.2.1 Die maximale Anzahl linear unabängiger Spaltenvektoren einer Matrix wird mit *Spaltenrang* bezeichet.

Die maximale Anzahl linear unabhängiger Zeilenvektoren einer Matrix wird mit *Zeilenrang* bezeichnet.

[3]Literatur: siehe 2.1

Die folgenden Betrachtungen gelten für den Zeilenrang analog.

Die $m \times n$-Matrix A habe den Spaltenrang r. Es gilt also

$$0 \leq r \leq n \ .$$

Die Spaltenvektoren von A werden mit

$$a_1, \ldots, a_n$$

bezeichnet. Seien die Spaltenvektoren

$$a_{i_1}, \ldots, a_{i_r}$$

linear unabhängig. Das bedeutet, aus

$$\begin{aligned} & d_1 a_{i_1} + \cdots + d_r a_{i_r} \\ = & \begin{bmatrix} d_1 a_{1,i_1} \\ \vdots \\ d_1 a_{n,i_1} \end{bmatrix} + \cdots + \begin{bmatrix} d_r a_{1,i_r} \\ \vdots \\ d_r a_{n,i_r} \end{bmatrix} \\ = & \begin{bmatrix} d_1 a_{1,i_1} + \cdots + d_r a_{1,i_r} \\ \vdots \\ d_1 a_{n,i_1} + \cdots + d_r a_{n,i_r} \end{bmatrix} \\ = & \ 0_n \end{aligned} \tag{2.3}$$

folgt

$$d_1 = \ldots = d_r = 0 \ .$$

Vertauscht man zwei Elemente des Vektors (2.3), bleibt die Gültigkeit dieser Bedingung davon unberührt. Daraus folgt, daß die Vertauschung zweier Matrixzeilen den Spaltenrang der Matrix unberührt läßt.

Ebenso verhält es sich mit der Multiplikation einer Zeile der Matrix mit einem Faktor $c \neq 0$. Analog zur Argumentation bei der Vertauschung zweier Zeilen erhält man

$$\begin{bmatrix} c(d_1 a_{1,i_1} + \cdots + d_r a_{1,i_r}) \\ \vdots \\ (d_1 a_{n,i_1} + \cdots + d_r a_{n,i_r}) \end{bmatrix} \ .$$

Die Bedingung $d_1 = \ldots = d_r = 0$ bleibt durch den Faktor unberührt.

Das gleiche Ergebnis erhält man für die Addition des Vielfachen einer Zeile zu einer anderen.

Die elementaren Zeilenoperationen[4] lassen den Spaltenrang also unverändert. Analog verhält es sich mit den elementare Spaltenoperationen und dem Zeilenrang.

Durch die Zeilen- und Spaltenoperationen läßt sich jede Matrix in Diagonalform überführen[5], ohne daß dadurch der Rang verändert wird.

Für eine Matrix, bei der nur die Elemente der Hauptdiagonalen von Null verschieden sein können, stimmen Zeilen- und Spaltenrang offensichtlich überein. Da die Zeilen und Spaltenoperationen den Rang unverändert lassen, erhält man:

Folgerung 2.2.2 *Für jede Matrix stimmen Zeilen- und Spaltenrang überein.*

Aus diesem Grund ist die folgende Definition sinnvoll:

Definition 2.2.3 Die Anzahl linear unabhängiger Spalten einer Matrix A wird als *Rang von* A bezeichnet, kurz

$$\operatorname{rg}(A) \ .$$

[4] siehe 2.1.7

[5] nicht zu verwechseln mit Diagonalisierbarkeit!

Es gilt offensichtlich:

$$rg(A) \leq \min(m, n) \ .$$

An dieser Stelle können wir drei wichtige Begriffe zueinander in Beziehung setzen:

Satz 2.2.4 *Für eine $n \times n$-Matrix A sind folgende Aussagen äquivalent:*

- *Matrix A ist invertierbar.*
- *Für die Determinante gilt:*
$$\det(A) \neq 0$$
- *Für den Rang gilt:*
$$\mathrm{rg}\,(A) = n$$

Beweis Es ist zu beachten, daß jede Matrix durch elementare Zeilen- und Spaltenoperationen in eine Diagonalmatrix überführt werden kann, ohne daß sich die Invertierbarkeitseigenschaft, der Betrag der Determinante oder der Rang dadurch veränderen. Man kann also o. B. d. A. davon ausgehen, daß A Diagonalform besitzt.

Durch diese Überlegung wird die Gültigkeit der Aussage offensichtlich. □

2.3 Lösbarkeit linearer Gleichungssysteme

Da lineare Gleichungssysteme eine wichtige Rolle beim Verständnis noch folgender Ausführungen spielen, werden sie in diesem Kapitel näher betrachtet. Die nun folgenden Grundlagen aus der Linearen Algebra sind in der in 2.1 aufgelisteten Literatur ausführlich behandelt. Wir beschränken uns hier auf die für den nachfolgenden Text wichtigen Sachverhalte.

Für die weiteren Beschreibungen drei grundlegende Begriffe aus der Linearen Algebra von Bedeutung, deren Definitionen deshalb hier angegeben sind:

Sei K ein Körper. Eine Menge V zusammen mit zwei Verknüpfungen

$$+ \ : \ V \times V \to V \tag{2.4}$$
$$: \ K \times V \to V \ , \tag{2.5}$$

die die folgenden Bedingungen erfüllt:

- Die Menge V in Verbindung mit $+$ ist eine Gruppe.
- Für alle $v, w \in V$ und alle $r, s \in K$ gelten die Gleichungen

$$\begin{aligned} (r+s)v &= rv + sv \\ r(v+w) &= rv + rw \\ (rs)v &= r(sv) \\ 1v &= v \ . \end{aligned}$$

wird als K-Vektorraum bezeichnet.

Definition 2.3.1 Eine nichtleere Teilmenge U eines K-Vektorraumes V wird als *Unterraum* von V bezeichnet, falls gilt

$$\forall u, v \in U, \, r \in K : \left\{ \begin{array}{rcl} u + v & \in & U \\ ru & \in & U \ . \end{array} \right.$$

Definition 2.3.2 Die Menge aller Vektoren x, für die bei einer gegebenen $m \times n$-Matrix A gilt

$$Ax = 0_n \tag{2.6}$$

wird als *Kern der Matrix A* , kurz $\ker(A)$, bezeichnet[6] .

Für unsere Zwecke benötigen wir noch zwei weitere Feststellungen:

Bemerkung 2.3.3 *Betrachtet man 2.3.1, erkennt man, daß alle x, die Gleichung (2.6) erfüllen, einen Unterraum bilden.*

Der Rang einer Matrix und die Dimension von $\ker(A)$ hängen auf eine für uns wichtige Weise zusammen:

Lemma 2.3.4 *Sei V ein K-Vektorraum der Dimension n und W ein K-Vektorraum der Dimension m. Sei*

$$f : V \to W$$

eine lineare Abbildung mit der entsprechenden $m \times n$-Matrix A. Dann gilt:

$$\mathrm{rg}(A) + \dim(\ker(A)) = \dim(V) \ .$$

Beweis Zum Beweis der obigen Gleichung wird die Dimension von $\ker(A)$ in Abhängigkeit vom Rang von A betrachtet.

Die Matrix A habe den Rang r. Die maximale Anzahl linear unabhängiger Spaltenvektoren beträgt also r. O. B. d. A. seien die ersten r Spaltenvektoren linear unabhängig.

Es wird in zwei Schritten gezeigt, daß in $\ker(A)$ die maximale Anzahl der linear unabhängigen Vektoren, also die Dimension von A, $n - r$ beträgt.

- Bezeichne a_i den i-ten Spaltenvektor von A. Der Kern von A ist die Menge

$$\begin{aligned} & \{x \in V \mid Ax = 0_m\} \\ = \ & \{x \in V \mid a_1x_1 + a_2x_2 + \cdots + a_nx_n = 0_m\} \ . \end{aligned}$$

  Die Dimension dieser Menge ist die maximale Anzahl linear unabhängiger Vektoren x, die in ihr enthalten sind. Die Elemente eines Vektors x kann man, wie an der obigen Darstellung ersichtlich ist, als Faktoren in einer Linearkombination von Spaltenvektoren von A betrachten. Anhand der Voraussetzungen, die für die Spaltenvektoren gelten, lassen sich Aussagen für die Vektoren x machen.

  Da $r + 1$ Spaltenvektoren von A immer linear abhängig sind, ist es möglich, durch Linearkombination von jeweils $r + 1$ Spaltenvektoren den Nullvektor zu erhalten. Sind die übrigen $n - (r + 1)$ Spaltenvektoren nicht an einer Linearkombination beteiligt, entspricht das einer Null als entsprechendes Element von x.

  Es gibt $n - r$ Möglichkeiten, zu den ersten r linear unabhängigen Spaltenvektoren von A genau einen weiteren auszusuchen. Aufgrund der Verteilung der Nullen in den entsprechenden Vektoren x muß es mindestens $n - r$ linear unabhängige Vektoren im Kern von A geben.

- Angenommen, die Anzahl der linear unabhängigen Vektoren in $\ker(A)$ ist größer als $n-r$. O. B. d. A. seien die ersten $n-r$ dieser Vektoren so gewählt, daß bei jedem dieser Vektoren unter den Vektorelementen x_{r+1} bis x_n genau eines ungleich Null ist. Aus den vorangegangenen Ausführungen folgt, daß solche Vektoren existieren müssen. Die Vektoren werden mit $v_{r+1}, \ldots, v_n$ bezeichnet. Für den Vektor v_i sei das i-te Vektorelement ungleich Null.

[6] In der Literatur wird häufig an dieser Stelle der Kern einer linearen Abbildung betrachtet. Da die theoretischen Hintergründe hier nicht von Interesse sind, ist in unser Betrachtung von Matrizen die Rede.

Da die ersten r Spaltenvektoren von A linear unabhängig sind, gibt es für die ersten r Elemente jedes Vektors v_i keine Wahlmöglichkeit, sobald das i-te Element im Rahmen der Randbedingungen dem Wert nach festliegt.

Sei w ein weiterer Vektor aus $\ker(A)$, der zu den Vektoren v linear unabhängig ist. Wird das j-te Element eines Vektors v_i mit $v_{i,j}$ bezeichnet, und das k-te Element von w mit w_k, dann gibt es Faktoren $d_{r+1}, \ldots, d_n$, so daß

$$\forall r+1 \leq k \leq n : \; w_k = d_k * v_{k,k} \; .$$

Ebenso wie für die Vektoren v liegen damit auch die ersten r Elemente des Vektors w fest. Sie können anhand der obigen Beziehung aus den Elementen $r+1$ bis n der Vektoren v und w berechnet werden, sofern w überhaupt die Rahmenbedinungen erfüllt und in $\ker(A)$ ist.

Damit w dennoch zu den Vektoren v linear unabhängig ist, muß es einen $r+1$-ten Spaltenvektor von A geben, der zu den ersten r Spaltenvektoren linear unabhängig ist und der bei der Zusammenstellung der Vektoren v mit Hilfe der Betrachtung von Linearkombinationen (s. o.) noch nicht benutzt wurde. Dies ist jedoch im Widerspruch zu den Voraussetzungen.

Somit folgt:

$$\mathrm{rg}(A) + \dim(\ker(A)) = r + (n-r) = n = \dim(V) \; .$$

□

Mit Hilfe von 2.3.3 und 2.3.4 können wir die für uns wichtigen Eigenschaften linearer Gleichungssysteme betrachten.

Ein lineares Gleichungssystem besteht aus n Gleichungen mit m Unbekannten:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ \vdots \quad \vdots \quad \vdots & \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{m,n}x_n &= b_m \end{aligned}$$

Die x_i sind die zu bestimmenden Unbekannten. Betrachtet man die gegebenen Konstanten $a_{i,j}$ und b_i als Elemente einer Matrix bzw. eines Vektors, kann man das Gleichungssystem auch kompakter darstellen:

$$Ax = b \; .$$

Dabei ist A ein $n \times m$-Matrix und b ein Vektor der Länge m. Ist $b = 0_m$, so bezeichnet man das Gleichungssystem als homogen . Ist $b \neq 0_m$, so bezeichnet man das Gleichungssystem als inhomogen .

Uns interessieren die Lösungsmengen eines solchen Gleichungssystems.

Satz 2.3.5 *Sei V der zugrundeliegende K-Vektorraum der Dimension n. Die Lösungsmenge $L(A, 0_m)$ des homogenen Gleichungssystems*

$$Ax = 0_m$$

ist ein Unterraum von V der Dimension

$$n - \mathrm{rg}(A) \; .$$

Beweis Die Lösungsmenge $L(A, 0_m)$ ist der Kern von A. Damit folgt die Behauptung aus 2.3.3 und 2.3.4. □

Ist $\mathrm{rg}\,(A) = n$, gilt also $L(A, 0_m) = \{0_n\}$.

Uns interessieren auch die Lösungen inhomogener Gleichungssysteme. Dazu betrachtet man die erweiterte Matrix $[A, b]$, die aus der Matrix A und dem Vektor b als $n+1$-te Spalte besteht:

Satz 2.3.6 *Die Gleichung*

$$Ax = b \ . \tag{2.7}$$

ist genau dann lösbar, wenn gilt

$$\mathrm{rg}\,(A) = \mathrm{rg}\,([A, b]) \ . \tag{2.8}$$

Beweis Man kann die linke Seite von (2.7) als Linearkombination von Spaltenvektoren von A betrachten. Die Faktoren der Linearkombination bilden die Elemente des Lösungsvektors x.

Der Vektor b ist genau dann als Linearkombination der Spaltenvektoren von A darstellbar, wenn die maximale Anzahl linear unabhängiger Vektoren in A und $[A, b]$ gleich ist. Dies ist gleichbedeutend mit der Gültigkeit von (2.8). □

Dieser Satz führt zu einer für uns bedeutsamen Charakterisierung der Lösbarkeit:

Folgerung 2.3.7 *Ist (2.7) lösbar und* $\mathrm{rg}\,(A) = n$*, also* $n \leq m$*, dann sind die Spaltenvektoren von A linear unabhängig und werden alle für die Linearkombination zur Darstellung von b benötigt. Es gibt in diesem Fall genau einen Vektor x, der (2.7) löst.*

Ist $\mathrm{rg}\,(A) = m$, also $m \leq n$, dann ist (2.7) für jedes b lösbar, denn es gilt allgemein

$$\mathrm{rg}\,(A) \leq \mathrm{rg}\,([A, b]) \leq m \ .$$

Für $\mathrm{rg}\,(A) = m$ folgt daraus mit Hilfe der vorangegangenen Bemerkungen die Lösbarkeit.

2.4 Das charakteristische Polynom

Betrachtet man A als lineare Abbildung im n-dimensionalen Vektorraum, so lautet eine interessante Fragestellung:

Gibt es einen Vektor x ungleich dem Nullvektor sowie einen Skalar λ, so daß gilt

$$Ax = \lambda x \quad ? \tag{2.9}$$

Man kann (2.9) umformen in

$$(A - \lambda E_n)x = 0 \ .$$

Dies ist eine homogenes lineares Gleichungssystem mit n Gleichungen in n Unbekannten. Aus 2.3.5 und 2.2.4 folgt, daß es genau dann eine nichttriviale Lösung besitzt, wenn gilt

$$\det(A - \lambda E_n) = 0 \ .$$

Dies motiviert die folgende Definition:

Definition 2.4.1

$$\det(A - \lambda E_n)$$

heißt[7] *charakteristisches Polynom der Matrix A*. Die obige Darstellung als Determinante einer Matrix heißt *Matrizendarstellung* des charakteristischen Polynoms.

$$\det(A - \lambda E_n) = 0$$

[7] Die Form $\det(\lambda E_n - A)$ ist in der Literatur auch gebräuchlich. Welche Form gewählt wird, hängt von den Vorlieben im Umgang mit den Vorzeichen ab. Bis auf die Vorzeichen bleibt die Gültigkeit von Aussagen unberührt. Für uns ist die in der Definition angegebene Form bequemer.

heißt *charakteristische Gleichung der Matrix A*. Die Matrix

$$A - \lambda E_n$$

wird als *charakteristische Matrix* bezeichnet.

Das charakteristische Polynom einer Matrix läßt sich auch in der *Koeffizientendarstellung*

$$c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda + c_0 \tag{2.10}$$

angeben.

In der Literatur sind als Vorzeichen von c_i nicht nur $+$, sondern auch $-$ oder $(-1)^i$ gebräuchlich. Mancherorts herrscht auch die Gewohnheit, die Indizierung der Koeffizienten in der Form $c_0\lambda^n + c_1\lambda^{n-1} + \cdots + c_{n-1}\lambda + c_n$ durchzuführen. Da diese Vielfalt der Gebräuche in der Literatur nicht nur auf die Koeffizientendarstellung beschränkt ist, kann es u. U. sein, daß c_n sowohl das Vorzeichen $+$ als auch immer den Wert 1 besitzt. In diesen Fällen wird c_n auch häufig weggelassen. Für uns ist die gewählte Form (2.10) der Darstellung am sinnvollsten.

Definition 2.4.2 Die Nullstellen des charakteristischen Polynoms einer Matrix heißen *Eigenwerte* der Matrix. Ein Vektor x der Gleichung (2.9) zusammen mit einem Eigenwert λ erfüllt, heißt *Eigenvektor*. Der Nullvektor ist als Eigenvektor nicht zugelassen.

Definition 2.4.3 Sei λ_i ein Eigenwert der $n \times n$-Matrix A. Dann wird

$$\operatorname{rg}(A) - \operatorname{rg}(A - \lambda_i E_n)$$

als *Rangabfall* des Eigenwertes λ_i bezeichnet.

Die Determinante und die Spur findet man unter den Koeffizienten des charakteristischen Polynoms wieder. Besonders für die Berechnung der Determinante ist dies eine wichtige Beobachtung:

Bemerkung 2.4.4

$$\begin{aligned} \det(A) &= c_0 \\ \operatorname{tr}(A) &= (-1)^{n-1}c_{n-1} \end{aligned} \tag{2.11}$$

Wenn $p(\lambda)$ das charakteristische Polynom der Matrix A ist, gilt also:

$$\det(A) = p(0) \ .$$

Rechnet man die Matrizendarstellung des charakteristischen Polynoms um in die Koeffizientendarstellung, wird dabei die Gültigkeit von (2.11) deutlich. Der Wert von c_{n-1} wird nur durch die Matrixelemente in der Hauptdiagonalen beeinflußt. Das Produkt dieser Elemente besteht aus n Linearfaktoren der Form

$$a_{i,i} - \lambda \ .$$

Berechnet man den Wert dieses Produktes, so wird c_n durch alle Summanden der Form

$$(-1)^{n-1}a_{i,i}\lambda^{n-1}$$

beeinflußt. Insgesamt gilt also

$$c_{n-1} = (-1)^{n-1}\sum_{i=1}^{n} a_{i,i} \ ,$$

so daß man mit (2.11) die Spur erhält.

Ein Algorithmus zur Berechnung der Koeffizienten des charakteristischen Polynoms einer Matrix berechnet somit u. a. auch deren Determinante und deren Spur.

Kapitel 3

Die Algorithmen von Csanky

In diesem Kapitel werden zwei von L. Csanky[1] vorgeschlagene Algorithmen behandelt. Der erste davon verwendet eine relativ bekannte Methode zur Determinantenberechnung[2]. Die Effizienz dieser Methode ist jedoch nicht befriedigend. Ihre Darstellung ist als Einstieg gedacht.

3.1 Die Stirling'schen Ungleichungen

Als Vorarbeit für den ersten Algorithmus werden in diesem Unterkapitel weitere Grundlagen behandelt.

Satz 3.1.1 (Stirling'sche Ungleichungen) *Sei*

$$n \in \mathbb{N}$$

Dann gilt

$$n! \geq \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n \mathrm{e}^{\frac{1}{12n+1}} \tag{3.1}$$

$$n! \leq \sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n \mathrm{e}^{\frac{1}{12n}} \tag{3.2}$$

Beweis Die Ungleichungen folgen aus [vM67] S. 154. □

Die Ungleichungen werden in den folgenden Lemmata angewendet. Die Richtung der Abschätzung der jeweiligen Werte wird durch deren Verwendung im späteren Text bestimmt.

Lemma 3.1.2

$$\binom{n}{\frac{n}{2}} \geq \frac{2^n}{\sqrt{\pi\frac{n}{2}}\,\mathrm{e}}$$

Beweis Es soll nach unten abgeschätzt werden. Deshalb wird der Zähler mit Hilfe von (3.1) und der Nenner mit Hilfe von (3.2) abgeschätzt. Man erhält:

$$\begin{aligned}\binom{n}{\frac{n}{2}} &= \frac{n!}{\left(\frac{n}{2}\right)^2} \\ &\geq \frac{\sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n \mathrm{e}^{\frac{1}{12n+1}}}{\left(\sqrt{\pi n}\left(\frac{\frac{n}{2}}{\mathrm{e}}\right)^{\frac{n}{2}} \mathrm{e}^{\frac{1}{6n}}\right)^2}\end{aligned} \tag{3.3}$$

[1] [Csa74] und [Csa76]
[2] Entwicklungssatz von Laplace

$$\geq \frac{\sqrt{2}}{\sqrt{\pi n}\left(\frac{1}{2}\right)^n \mathrm{e}^{\frac{-18n-2}{(12n+1)6n}}} \tag{3.4}$$
$$\geq \frac{2^n}{\sqrt{\pi\frac{n}{2}}\,\mathrm{e}}$$

□

Auf die gleiche Weise, wie im vorangegangenen Lemma 3.1.3 eine Abschätzung nach unten erfolgt, erhält man eine Abschätzung von

$$\log\binom{n}{\frac{n}{2}}$$

nach oben:

Lemma 3.1.3

$$\log\binom{n}{\frac{n}{2}} \leq n - \frac{\log\left(\frac{n}{2}\right)+1}{2}$$

Beweis Aus (3.1) folgt:

$$\left(\frac{n}{2}\right)! \geq \sqrt{\pi n}\left(\frac{\frac{n}{2}}{\mathrm{e}}\right)^{\frac{n}{2}} \mathrm{e}^{\frac{1}{6n+1}}$$

Also gilt:

$$\frac{n!}{\left(\frac{n}{2}!\right)^2} \leq \frac{\sqrt{2\pi n}\left(\frac{n}{\mathrm{e}}\right)^n \mathrm{e}^{\frac{1}{12n}}}{\left(\sqrt{\pi n}\left(\frac{\frac{n}{2}}{\mathrm{e}}\right)^{\frac{n}{2}} \mathrm{e}^{\frac{1}{6n+1}}\right)^2} \tag{3.5}$$
$$= \frac{\sqrt{2}}{\sqrt{\pi n}\left(\frac{1}{2}\right)^n \mathrm{e}^{\frac{1-18n}{12n(6n+1)}}} \tag{3.6}$$
$$\leq \frac{2^n}{\sqrt{\pi\frac{n}{2}}}$$

Bildet man nun auf beiden Seiten der Ungleichungskette den Logarithmus, erhält man mit Hilfe einiger Logarithmengesetze

$$\log\binom{n}{\frac{n}{2}} \leq n - \frac{1}{2}\log\left(\frac{n}{2}\right) - \frac{1}{2}\log(\pi) \leq n - \frac{\log\left(\frac{n}{2}\right)+1}{2}$$

□

3.2 Der Entwicklungssatz von Laplace

Für den ersten im Kapitel 3 darzustellenden Algorithmus wird ein bekannter Satz verwendet. Er wird in diesem Unterkapitel zusammen mit einer häufig benutzten Folgerung dargestellt.

Satz 3.2.1 (Entwicklungssatz von Laplace) *Sei k eine natürliche Zahl mit*

$$1 \leq k \leq n-1$$

Sei D_n die Determinante der Matrix A.

Für die natürliche Zahl i gelte

$$1 \leq i \leq \binom{n}{k}$$

Sei $D_k^{(2i-1)}$ die Determinante einer Untermatrix $A_k^{(2i-1)}$ von A, die aus k Spalten der ersten k Zeilen gebildet werde. von A, die aus den für $A_k^{(2i-1)}$ nicht gewählten $n-k$ Zeilen und Spalten gebildet werde.

Für jedes i werde eine andere der

$$\binom{n}{k}$$

möglichen Auswahlen für die k Spalten für $A^{(2i-1)}$ getroffen.

Für eine Untermatrix $A_k^{(2i-1)}$ bezeichne

$$f(A_k^{(2i-1)})$$

die n-Permutation, die die Spalten von A so vertauscht, daß $A_k^{(2i-1)}$ aus den ersten k und $A_{n-k}^{(2i)}$ aus den weiteren $n-k$ Zeilen und Spalten von A besteht.

Dann gilt:

$$D_n = \operatorname{sig}(f(A_k^1))D_k^1 D_{n-k}^2 + \operatorname{sig}(f(A_k^3))D_k^3 D_{n-k}^4 + \cdots + \operatorname{sig}\left(f\left(A_k^{2\binom{n}{k}-1}\right)\right) D_k^{2\binom{n}{k}-1} D_k^{2\binom{n}{k}} \quad (3.7)$$

Beweis (vergl. [Csa74] Seite 21) Es ist zu zeigen, daß die Berechnung der Determinante nach (3.7) mit der Berechnung nach (2.2) übereinstimmt.

Die einzelnen Determinanten in (3.7) werden auf die in (2.2) angegebene Weise berechnet. Dazu ist zu beachten, daß $\mathcal{S}_n$ genau $n!$ Elemente besitzt. Es genügt zu zeigen, daß in (3.7) ebenso viele Permutationen auf die Indizes von 1 bis n angewendet werden und alle voneinander verschieden sind.

Berechnet man einen Summanden

$$\operatorname{sig}\left(f\left(A_k^{(2i-1)}\right)\right) D_k^{(2i-1)} D_{n-k}^{(2i)}$$

in (3.7) anhand von (2.2), kann man die zwei Summen durch Ausmultiplizieren zu einer zusammenfassen. Den $k!$ Permutationen zur Berechnung von $D_k^{(2i-1)}$ und den $(n-k)!$ Permutationen zur Berechnung von $D_{n-k}^{(2i)}$ entsprechen

$$k!\,(n-k)!$$

n-Permutationen zur Berechnung der zusammengefaßten Summen für diesen einen Summanden. Die Menge dieser n-Permutationen wird mit M_i bezeichnet.

Die k-Permutationen bzw. $(n-k)$-Permutationen zur Berechnung aller Determinanten D_k und D_{n-k} werden jeweils auf verschiedene Mengen von Indizes angewendet, d. h. für zwei beliebige dieser Mengen von Indizes gibt es in der einen mindestens einen Index, der in der anderen nicht enthalten ist. Das bedeutet, daß alle Mengen M_i voneinander verschieden sind.

Die Anzahl der Mengen M_i, die man auf diese Weise für (3.7) erhält, beträgt

$$\binom{n}{k}$$

Ihre Vereinigung ergibt eine Menge mit

$$\binom{n}{k} k!\,(n-k)!$$

also

$$n!$$

Elementen, was zu zeigen war. □

Aus diesem Satz erhält man mit $k = 1$:

Folgerung 3.2.2 (Zeilen- und Spaltenentwicklung) *Seien*

$$1 \leq p \leq n$$

und

$$1 \leq q \leq n$$

beliebig. Dann gilt die Entwicklung nach Zeile p

$$\det(A) = \sum_{j=1}^{n} (-1)^{p+j} a_{p,j} \det(A_{(p|j)})$$

und die Entwicklung nach Spalte q

$$\det(A) = \sum_{i=1}^{n} (-1)^{i+q} a_{i,q} \det(A_{(i|q)})$$

3.3 Determinantenberechnung durch 'Divide and Conquer'

Der Algorithmus ([Csa74] S. 21 ff.), der hier betrachtet wird, berechnet die Determinante rekursiv mit Hilfe der Methode *Divide and Conquer*, d. h. durch die Berechnung der Determinanten von Untermatrizen der gegebenen Matrix.

Da es sich hierbei nur um ein einleitendes Beispiel handelt, wird zur Vereinfachung angenommen, daß die Anzahl der Zeilen und Spalten n der Matrix eine Zweierpotenz ist. Falls dies nicht der Fall ist, wird sie um entsprechend viele Zeilen und Spalten erweitert, so daß die neuen Elemente der Hauptdiagonalen jeweils gleich 1 und alle weiteren neuen Elemente jeweils gleich 0 sind.

Zur rekursiven Berechnung der Determinate wird Satz 3.2.1 benutzt. Man wählt

$$k := \frac{1}{2} n \ .$$

Somit gilt auch

$$n - k = \frac{1}{2} n \ .$$

Die Anzahl der Schritte, die der Algorithmus benötigt, um mit Hilfe dieses Satzes die Determinante einer $n \times n$-Matrix zu berechnen, wird mit

$$s(n)$$

bezeichnet. Die Anzahl der Prozessoren, die dabei beschäftigt werden, wird mit

$$p(n)$$

bezeichnet.

Bei der Berechnung wird Gleichung (3.7) rekursiv ausgewertet. Das führt dazu, daß

$$\binom{n}{\frac{n}{2}}$$

Determinanten von Untermatrizen zu berechnen sind. Die Berechnung einer Determinanten erfordert

$$s\left(\frac{n}{2}\right)$$

Schritte und

$$2 \binom{n}{\frac{n}{2}} p\left(\frac{n}{2}\right)$$

Prozessoren.

Aus 1.5.2 folgt, daß die in (3.7) auftretenden Additionen in

$$\log \binom{n}{\frac{n}{2}}$$

Schritten von

$$\frac{\binom{n}{\frac{n}{2}}}{2}$$

Prozessoren erledigt werden können. Die Multiplikationen können in einem Schritt von ebenfalls

$$\frac{\binom{n}{\frac{n}{2}}}{2}$$

Prozessoren durchgeführt werden. Also gilt für die Anzahl der Schritte, die der Algorithmus benötigt, folgende Rekursionsgleichung:

$$s(n) = \begin{cases} 0 & : \quad n = 1 \\ s\left(\frac{n}{2}\right) + 1 + \log \binom{n}{\frac{n}{2}} & : \quad n > 1 \end{cases}$$

Mit 3.1.3 folgt

$$s(n) \leq s\left(\frac{n}{2}\right) + 1 + n - \frac{\log\left(\frac{n}{2}\right) + 1}{2}$$

Das ist äquivalent zu

$$s(n) \leq s\left(\frac{n}{2}\right) + 1 + n - \frac{\log(n)}{2}$$

Die Auflösung der Rekursion ergibt:

$$s(n) \leq \log(n) + \sum_{j=1}^{\log(n)} \frac{n}{j} - \frac{1}{2} \sum_{k=1}^{\log(n)} \log\left(\frac{n}{k}\right)$$

Man kann nun die folgenden Abschätzungen vornehmen:

$$\begin{aligned} \sum_{j=1}^{\log(n)} \frac{1}{j} &= 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\log(n)} \\ &\leq 1 + \int_{1}^{\log(n)} \frac{1}{x}\, dx \\ &= [\ln(x)]_1^{\log(n)} = 1 + \ln(\log(n)) \\ &\leq 1 + \log(\log(n)) \end{aligned}$$

und

$$\sum_{k=1}^{\log(n)} \log(k) \leq \sum_{k=1}^{\log(n)} \log(\log(n)) = \log(n) \log(\log(n))$$

und kommt so in Verbindung mit einigen Logarithmengesetzen auf

$$s(n) \leq \log(n) + n(1 + \lceil \log(\log(n)) \rceil) - \frac{1}{2} \log^2(n) + \log(n) \lceil \log(\log(n)) \rceil$$

Also gilt

$$s(n) = O(n \log(\log(n)))$$

Für die Anzahl der beschäftigten Prozessoren gilt

$$p(n) = \begin{cases} 0 & : \quad n = 1 \\ 2 & : \quad n = 2 \\ \max\left(2\binom{n}{\frac{n}{2}} p\left(\frac{n}{2}\right), \frac{\binom{n}{\frac{n}{2}}}{2}\right) & : \quad \text{sonst} \end{cases}$$

Bei diesem Beispielalgorithmus interessiert uns nur die Größenordnung der Anzahl beschäftigter Prozessoren. Deshalb wird diese Anzahl grob nach unten abgeschätzt durch

$$p(n) > \binom{n}{\frac{n}{2}}$$

Mit 3.1.2 folgt

$$p(n) > \frac{2^n}{\sqrt{\pi \frac{n}{2}}\,\mathrm{e}}$$

Da nach unten abgeschätzt wurde, folgt aus dieser Ungleichung

$$p(n) = \Omega\left(\frac{2^n}{\sqrt{n}}\right)$$

Die Anzahl der Prozessoren ist also von exponentieller Größenordnung.

Es wird sich bei den noch zu betrachtenden Algorithmen zeigen, daß sowohl für die Anzahl der Schritte als auch für die Anzahl der beschäftigten Prozessoren deutlich bessere Werte möglich sind.

3.4 Die Linearfaktorendarstellung

In diesem Unterkapitel werden einige Aussagen behandelt, die sich in Verbindung mit einer weiteren Darstellungsmöglichkeit für das charakteristische Polynom ergeben. Diese Aussagen werden für den Beweis des Satzes von Frame (siehe 3.7) benötigt. Literatur dazu ist bereits in Kapitel 2 aufgelistet.

Neben der Matrizendarstellung und der Koeffizientendarstellung gibt es noch eine dritte für uns wichtige Darstellung für das charakteristische Polynom, die *Linearfaktorendarstellung*. Ein Polynom n-ten Grades kann man auch als Produkt von n Linearfaktoren darstellen, so daß das charakteristische Polynom folgendes Aussehen bekommt:

$$(\lambda_1 - \lambda)(\lambda_2 - \lambda)\dots(\lambda_n - \lambda) \tag{3.8}$$

Dabei sind die λ_i die Eigenwerte der zugrunde liegenden $n \times n$-Matrix. Sie sind *nicht* paarweise verschieden. Diese Tatsache führt uns zum Begriff der *Vielfachheit*. Man kann in der obigen Linearfaktorendarstellung gleiche Faktoren mit Hilfe von Potenzen beschreiben. Dazu gelte

$$k \in \mathbb{N},\ 1 \le k \le n\ .$$

Die Eigenwerte seien

$$\lambda'_1, \lambda'_2, \dots, \lambda'_k\ ,$$

jedoch alle paarweise voneinander verschieden. So bekommt (3.8) folgendes Aussehen:

$$(\lambda'_1 - \lambda)^{m(\lambda'_1)}(\lambda'_2 - \lambda)^{m(\lambda'_2)}\dots(\lambda'_k - \lambda)^{m(\lambda'_k)}$$

In dieser Darstellung wird $m(\lambda'_i)$ mit *Vielfachheit* des Eigenwertes λ'_i bezeichnet.

Die n Eigenwerte einer $n \times n$-Matrix besitzen zur Determinante und zur Spur jeweils eine wichtige Beziehung, welche in den nächsten beiden Sätzen dargestellt wird:

Satz 3.4.1

$$\det(A) = \prod_{i=1}^{n} \lambda_i$$

Beweis Die Gültigkeit der Behauptung wird bei Betrachtung der Berechnung der Koeffizientendarstellung des charakteristischen Polynoms aus dessen Linearfaktorendarstellung deutlich. Beim Ausmultiplizieren der Linearfaktoren wird der Wert von c_0 nur durch das Produkt der Eigenwerte beeinflußt. □

Satz 3.4.2

$$\mathrm{tr}\,(A) = \sum_{i=1}^{n} \lambda_i$$

Beweis Betrachtet man die Berechnung der Koeffizientendarstellung des charakteristischen Polynoms aus dessen Linearfaktorendarstellung durch Ausmultiplizieren, erkennt man, daß gilt:

$$c_{n-1} = (-1)^{n-1} \sum_{i=1}^{n} \lambda_i \ .$$

Mit 2.4.4 folgt daraus die Behauptung. □

Die Eigenwerte besitzen weiterhin folgende interessante Eigenschaft:

Satz 3.4.3 *Seien $\lambda_1, \lambda_2, \ldots, \lambda_n$ die Eigenwerte der $n \times n$-Matrix A. Sei $k \in \mathbb{N}$. Dann gilt:*

$$\lambda_1^k, \lambda_2^k, \ldots, \lambda_n^k$$

sind die Eigenwerte von A^k.

Beweis Ein Skalar λ ist genau dann Eigenwert von A, wenn es einen Vektor $x \neq 0_n$ gibt, so daß gilt[3]

$$Ax = \lambda x \ . \tag{3.9}$$

Mit Hilfe dieser Beziehung wird die Behauptung per Induktion bewiesen.

Nach Voraussetzung ist die Behauptung für $k = 1$ erfüllt. Es gelte also nun

$$\forall \lambda = \lambda_1, \ldots, \lambda_n \exists x : \ A^k x = \lambda^k x \ .$$

Zu zeigen ist, daß dann auch gilt

$$\forall \lambda = \lambda_1, \ldots, \lambda_n \exists x : \ A^{k+1} x = \lambda^{k+1} x$$

Diese Gleichung kann man umformen in

$$\forall \lambda = \lambda_1, \ldots, \lambda_n \exists x : \ A^k \underbrace{Ax}_{(*)} = \lambda^k \underbrace{\lambda x}_{(*)}$$

Nach Voraussetzung sind die Terme $(*)$ gleich. Sie werden mit y bezeichnet. Die Gleichung bekommt dann folgendes Aussehen:

$$\forall \lambda = \lambda_1, \ldots, \lambda_n \exists y : \ A^k y = \lambda^k y$$

Dies ist wiederum nach Voraussetzung richtig. □

Aus diesem Satz ergibt sich mit Hilfe von 3.4.2 eine für uns wichtige Beziehung:

Folgerung 3.4.4

$$\mathrm{tr}\,(A^k) = \sum_{i=1}^{n} \lambda_i^k$$

[3] vgl. Erläuterungen auf S. 25

3.5 Die Newton'schen Gleichungen für Potenzsummen

Mit den *Newton'schen Gleichungen für Potenzsummen* werden in diesem Unterkapitel weitere Grundlagen für den Beweis der Sätze von Frame (siehe 3.7) behandelt. Die gesamten Hintergründe für diese Gleichungen werden z. B. in [Hau52] (Kapitel 7 und 8) behandelt.

Eine *Potenzsumme* ist eine Summe von Potenzen einer oder mehrerer Unbestimmter. Auf Seite 4.6.1 sind weitere Beispiele für einfachere Potenzsummengleichungen zu finden.

Da uns diese Gleichungen im Zusammenhang mit dem charakteristischen Polynom einer Matrix interessieren, werden sie anhand dieses Polynoms entwickelt. Dazu werden folgende Vereinbarungen getroffen:

- Das charakteristische Polynom der $n \times n$-Matrix A sei
$$p(\lambda) := c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda + c_0 \ .$$
- Die erste Ableitung von $p(\lambda)$ wird mit
$$p'(\lambda)$$
bezeichnet.
- Die Eigenwerte von A seien
$$\lambda_1, \lambda_2, \ldots, \lambda_n \ .$$
Es wird definiert
$$s_k := \sum_{i=1}^{n} \lambda_i^k \ .$$

Zunächst beschäftigen wir uns mit der Polynomdivision. Sei dazu ein i mit $1 \leq i \leq n$ gegeben. Da λ_i Eigenwert von A und somit Nullstelle von $p(\lambda)$ ist, läßt sich $p(\lambda)$ ohne Rest durch
$$\lambda_i - \lambda$$
teilen. Das Ergebnis ist ein Polynom vom Grad $n-1$. Das folgende Lemma liefert eine Aussage über dessen Koeffizienten:

Lemma 3.5.1 *Gegeben sei die Gleichung:*
$$\frac{p(\lambda)}{(\lambda_i - \lambda)} = \hat{c}_{n-1}\lambda^{n-1} + \hat{c}_{n-2}\lambda^{n-2} + \ldots + \hat{c}_1\lambda + \hat{c}_0 \tag{3.10}$$
Dann gilt für $1 \leq k \leq n-1$*:*
$$\hat{c}_k = -\sum_{j=1}^{n-k} \lambda_i^{j-1} c_{k+j} \tag{3.11}$$

Beweis Multipliziert man beide Seiten von Gleichung (3.10) mit
$$(\lambda_i - \lambda)$$
und multipliziert die rechte Seite der so gewonnenen Gleichung aus, ergibt sich:
$$\begin{aligned} & c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda + c_0 \\ = \ & -\hat{c}_{n-1}\lambda^n + (\lambda_i\hat{c}_{n-1} - \hat{c}_{n-2})\lambda^{n-1} + (\lambda_i\hat{c}_{n-2} - \hat{c}_{n-3})\lambda^{n-2} + \cdots + (\lambda_i\hat{c}_1 - \hat{c}_0)\lambda + \lambda_i\hat{c}_0 \end{aligned}$$
Setzt man
$$\hat{c}_n = \hat{c}_{-1} = 0 \ ,$$

erhält man durch Koeffizientenvergleich:

$$\begin{aligned} & \forall 1 \leq l \leq n : c_k = \lambda_i \hat{c}_k - \hat{c}_{k-1} \\ \Leftrightarrow \quad & \forall 1 \leq l \leq n : \hat{c}_{l-1} = \lambda_i \hat{c}_l - c_l \end{aligned} \tag{3.12}$$

Gleichung (3.11) wird durch Induktion[4] bewiesen. Für $k = n-1$ folgt die Gültigkeit von (3.11) aus (3.12).

Gelte (3.11) also nun für $k = l$. Es ist zu zeigen, daß die Gleichung dann auch für $k = l-1$ gilt:

$$\hat{c}_{l-1} = -\sum_{j=1}^{n-(l-1)} \lambda_i^{j-1} c_{l-1+j}$$

Zieht man c_l aus der Summe heraus und indiziert neu, erhält man:

$$\hat{c}_{l-1} = -c_l - \sum_{j=1}^{n-l} \lambda_i^{j} c_{l+j}$$

Zieht man nun noch λ_i aus der Summe heraus, erhält man:

$$\hat{c}_{l-1} = -c_l - \lambda_i \sum_{j=1}^{n-l} \lambda_i^{j-1} c_{l+j}$$

Nach Induktionsvoraussetzung ist dies gleichbedeutend mit:

$$\hat{c}_{l-1} = -c_l + \lambda_i \hat{c}_l$$

Die Gültigkeit dieser Gleichung folgt aus (3.12). □

Eine an dieser Stelle wichtige Regel für das Differenzieren lautet ([Hau52] S. 160):

Bemerkung 3.5.2 *Seien*

$$g_1(x), g_2(x), \ldots, g_n(x)$$

auf einem Intervall I differenzierbare Funktionen. Es gelte

$$f(x) = \prod_{i=1}^{n} g_i(x) \ .$$

Dann ist auch $f(x)$ auf I differenzierbar mit

$$f'(x) = \sum_{i=1}^{n} g_1(x) g_2(x) \cdots g_{i-1}(x) g_i'(x) g_{i+1}(x) \cdots g_{n-1}(x) g_n(x) \ .$$

Falls gilt

$$\forall x \in I, 1 \leq i \leq n : \ g_i(x) \neq 0 \ ,$$

läßt sich diese Regel auch einfacher formulieren:

$$f'(x) = \sum_{i=1}^{n} \frac{f(x)}{g_i(x)} g_i'(x)$$

Betrachtet man das charakteristische Polynom $p(\lambda)$ in Linearfaktorendarstellung und beachtet, daß die Ableitung von

$$(\lambda_i - \lambda)$$

-1 ergibt, dann erhält man mit Hilfe von 3.5.2:

[4] Um die Art und Weise, wie die Koeffizienten der Polynome indiziert sind, einheitlich zu halten, verläuft die Induktion etwas ungewohnt.

Folgerung 3.5.3

$$p'(\lambda) = -\sum_{i=1}^{n} \frac{p(\lambda)}{(\lambda_i - \lambda)} \tag{3.13}$$

Mit Hilfe von 3.5.1 und 3.5.3 erhält man nun die gesuchten Gleichungen ([Hau52] S. 181):

Satz 3.5.4 (Newton'sche Gleichungen für Potenzsummen)
Für die Koeffizienten des charakteristischen Polynoms gilt[5]*:*

$$\forall 0 \leq k \leq n-1 : \; -(n-(k+1))c_{k+1} - \sum_{j=2}^{n-k} s_{j-1} c_{k+j} = 0$$

Beweis Die $\hat{c}_k$ aus Lemma 3.5.1 sind abhängig vom gewählten λ_i. Deshalb definieren wir diese Koeffizienten als Funktionen von λ_i:

$$\hat{c}_k(\lambda_i) := -\sum_{j=1}^{n-k} \lambda_i^{j-1} c_{k+j} \quad .$$

Dann folgt aus Lemma 3.5.1:

$$\sum_{i=1}^{n} \hat{c}_k(\lambda_i) = \overbrace{-\sum_{j=1}^{n-k} s_{j-1} c_{k+j}}^{\dot{c}_k :=} \quad . \tag{3.14}$$

Überträgt man diese Beziehung zwischen den Koeffizienten auf die entsprechenden Polynome erhält man:

$$\sum_{i=1}^{n} \frac{p(\lambda)}{\lambda_i - \lambda} = \sum_{k=0}^{n-1} \dot{c}_k \lambda^k \tag{3.15}$$

Die erste Ableitung des charakteristischen Polynoms in der Koeffizientendarstellung sieht folgendermaßen aus:

$$p'(\lambda) = n c_n \lambda^{n-1} + (n-1) c_{n-1} \lambda^{n-2} + \cdots + 2c_2 \lambda + c_1 \tag{3.16}$$

Aus den drei Gleichungen (3.13), (3.15) und (3.16) folgt:

$$-\sum_{k=0}^{n-1} \dot{c}_k \lambda^k = n c_n \lambda^{n-1} + (n-1) c_{n-1} \lambda^{n-2} + \cdots + 2c_2 \lambda + c_1$$

Durch Koeffizientenvergleich erhält man aus dieser Gleichung:

$$\forall 0 \leq k \leq n-1 : \; -\dot{c}_k = (k+1) c_{k+1}$$

Setzt man für $\dot{c}_j$ den entsprechenden Term aus Gleichung (3.14) ein, ergibt sich:

$$\begin{aligned}
& \forall 0 \leq k \leq n-1 : \sum_{j=1}^{n-k} s_{j-1} c_{k+j} &&= (k+1) c_{k+1} \\
\Leftrightarrow \quad & \forall 0 \leq k \leq n-1 : \; n c_{k+1} + \sum_{j=2}^{n-k} s_{j-1} c_{k+j} &&= (k+1) c_{k+1} \\
\Leftrightarrow \quad & \forall 0 \leq k \leq n-1 : \; (n-(k+1)) c_{k+1} + \sum_{j=2}^{n-k} s_{j-1} c_{k+j} &&= 0
\end{aligned}$$

□

[5] Die Terme s_i sind am Beginn des Unterkapitels definiert.

3.6 Die Adjunkte einer Matrix

Bei den Beweisen des Satzes von Frame in Unterkapitel 3.7 spielt die Adjunkte einer Matrix eine bedeutende Rolle und wird deshalb hier behandelt.

Definition 3.6.1 Sei A eine $n \times n$-Matrix. Erhält man die Matrix B aus der Matrix A nach

$$b_{i,j} := (-1)^{i+j} \det(A_{(j|i)}) \ ,$$

so heißt B *Adjunkte der Matrix* A. Die Adjunkte wird mit

$$\operatorname{adj}(A)$$

bezeichnet.

Zum Beweis einer uns besonders interessierenden Eigenschaft der Adjunkten benötigen wir zunächst noch ein Lemma. Es behandelt den Fall einer Zeilen- bzw. Spaltenentwicklung[6], bei der jedoch als Faktoren für die Unterdeterminanten die Matrizenelemente nicht aus der Zeile bzw. Spalte entnommen werden, nach der die Determinante entwickelt wird:

Lemma 3.6.2 *Seien*

$$1 \leq p, p' \leq n$$

und

$$1 \leq q, q' \leq n$$

mit

$$p \neq p'$$

und

$$q \neq q'$$

Dann gilt:

$$\sum_{j=1}^{n} (-1)^{p+j} a_{p',j} \det(A_{(p|j)}) = 0$$

und

$$\sum_{i=1}^{n} (-1)^{i+q} a_{i,q'} \det(A_{(i|q)}) = 0$$

Beweis Betrachtet man die Berechnung der Unterdeterminanten in den obigen Gleichungen nach (2.2), erkennt man beim Vergleich mit der Berechnung der Determinante einer Matrix, die zwei gleiche Zeilen enthält, daß die Terme beider Berechnungen nach einigen Vereinfachungen übereinstimmen. Nach Satz 2.1.8 ist die Determinante in diesem Fall gleich 0. □

Mit Hilfe von 3.6.2 erhalten wir nun:

Satz 3.6.3

$$A * \operatorname{adj}(A) = E_n * \det(A) \tag{3.17}$$

$$\operatorname{adj}(A) * A = E_n * \det(A) \tag{3.18}$$

[6] vgl. 3.2.2

Beweis Spalte k der Matrix $\mathrm{adj}\,(A)$ sieht so aus:

$$\begin{bmatrix} (-1)^{1+k}\det(A_{(k|1)}) \\ (-1)^{2+k}\det(A_{(k|2)}) \\ \vdots \\ (-1)^{n+k}\det(A_{(k|n)}) \end{bmatrix}$$

Das Element an der Stelle (i,k) der Produktmatrix

$$A * \mathrm{adj}\,(A)$$

ist also gleich

$$\sum_{j=1}^{n} a_{i,j}(-1)^{j+k}A_{(k|j)}$$

Dies ist nach 3.2.2 gleich $\det(A)$ für

$$i = k$$

und nach 3.6.2 gleich 0 für

$$i \neq k$$

Daraus folgt die Gültigkeit von (3.17). Die Argumentation für (3.18) verläuft analog. □

3.7 Der Satz von Frame

In diesem Unterkapitel wird eine Methode von J. S. Frame ([Fra49]; [Dwy51] S. 225-235) vorgestellt[7] , die es u. a. erlaubt, die Determinante einer Matrix zu berechnen. Diese Methode ist im wesentlichen eine Neuentdeckung der Methode von Leverrier aus dem 19. Jahrhundert zur Bestimmung der Koeffizienten des charakteristischen Polynoms (z. B. [Hou64] S. 166 ff.). Die Darstellung wird hier auf die Teile beschränkt, die für die Determinantenberechnung wichtig sind.

Die Adjunkte von

$$A - \lambda E_n \tag{3.19}$$

besteht aus lauter Determinanten von $(n-1)\times(n-1)$-Matrizen. Sie kann deshalb durch ein Polynom vom Grad $n-1$ dargestellt werden (siehe dazu auch 2.4.1 und 3.6.1). Dies motiviert die folgende Vereinbarung zusätzlich zu den in 3.5 aufgeführten Bezeichnungen:

Seien

$$B_i\ ,\ 0 \leq i \leq n-1$$

geeignet gewählte $n \times n$-Matrizen. Dann bezeichnet

$$c(\lambda) := B_{n-1}\lambda^{n-1} + B_{n-2}\lambda^{n-2} + \cdots + B_2\lambda^2 + B_1\lambda + B_0\ .$$

die Adjunkte von (3.19).

Lemma 3.7.1

$$\begin{aligned} B_{n-1} &= (-1)E_n \\ \forall n-2 \geq i \geq 0:\ B_i &= AB_{i+1} - c_{i+1}E_n \end{aligned}$$

[7] Die Originalveröffentlichung [Fra49] enthält keinen Beweis. Dieser Beweis ist schwer zu bekommen. Er wird hier deshalb frei nachvollzogen und dürfte sich vom Original nicht wesentlich unterscheiden. In diesem Zusammenhang möchte ich mich bei R. T. Bumby für seinen Hinweis auf die Newton'schen Gleichungen für Potenzsummen bedanken.

Beweis Aus Satz 3.6.3 in Verbindung mit Definition 2.4.1 folgt:

$$(A - \lambda E_n)c(\lambda) = p(\lambda)E_n$$

Setzt man die Koeffizientendarstellungen von $c(\lambda)$ und $p(\lambda)$ in diese Gleichung ein, erhält man

$$\begin{aligned}
&(A - \lambda E_n)(B_{n-1}\lambda^{n-1} + B_{n-2}\lambda^{n-2} + \cdots + B_2\lambda^2 + B_1\lambda + B_0) \\
&\qquad = (c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_2\lambda^2 + c_1\lambda + c_0)E_n \\
\Leftrightarrow \quad &AB_{n-1}\lambda^{n-1} + AB_{n-2}\lambda^{n-2} + \cdots + AB_2\lambda^2 + AB_1\lambda + AB_0 \\
&- B_{n-1}\lambda^n - B_{n-2}\lambda^{n-1} - \cdots - B_2\lambda^3 - B_1\lambda^2 - B_0\lambda \\
&\qquad = (c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_2\lambda^2 + c_1\lambda + c_0)E_n \\
\Leftrightarrow \quad &- B_{n-1}\lambda^n + (AB_{n-1} - B_{n-2})\lambda^{n-1} + (AB_{n-2} - B_{n-3})\lambda^{n-2} + \cdots \\
&+ (AB_2 - B_1)\lambda^2 + (AB_1 - B_0)\lambda \\
&\qquad = (c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_2\lambda^2 + c_1\lambda + c_0)E_n
\end{aligned}$$

Koeffizientenvergleich ergibt die Behauptung. □

Lemma 3.7.2 *Es gilt*[8]

$$p'(\lambda) = -\mathrm{tr}\,(c(\lambda)) \ .$$

Beweis Zu zeigen ist:

$$\sum_{j=1}^{n} jc_j\lambda^{j-1} = -\mathrm{tr}\left(\sum_{j=1}^{n} B_{j-1}\lambda^{j-1}\right)$$

Durch Koeffizientenvergleich erhält man:

$$\forall 1 \le j \le n : \ jc_j = -\mathrm{tr}\,(B_{j-1})$$

Durch wiederholte Anwendung von 3.7.1 ergibt sich daraus:

$$\begin{aligned}
jc_j &= -\mathrm{tr}\,(AB_j - c_jE_n) \\
\Leftrightarrow (n-j)c_j &= \mathrm{tr}\,(AB_j) \\
\Leftrightarrow (n-j)c_j &= \mathrm{tr}\,(A(AB_{j+1} - c_{j+1}E_n)) \\
\Leftrightarrow (n-j)c_j &= \mathrm{tr}\,(A^{n-j}B_{n-1}) - \sum_{k=1}^{n-j-1} \mathrm{tr}\,(A^k)c_{j+k} \\
\Leftrightarrow (n-j)c_j &= -\mathrm{tr}\,(A^{n-j}) - \sum_{k=1}^{n-j-1} \mathrm{tr}\,(A^k)c_{j+k} \\
\Leftrightarrow (n-j)c_j &= -\mathrm{tr}\,(A^{n-j}) - \sum_{k=1}^{n-j-1} \mathrm{tr}\,(A^k)c_{j+k}
\end{aligned}$$

Nach 3.4.4 ist dies gleichbedeutend mit

$$(n-j)c_j + s_{n-j} + \sum_{k=1}^{n-j-1} s_kc_{j+k} = 0$$

Da für das charakteristische Polynom $c_n = 1$ gilt, ist diese Gleichung nach Satz 3.5.4 richtig. □

[8] Da in dieser Arbeit verschiedene Arten von hoch und tiefgestellten Indizes und Markierungen verwendet werden, sei hiermit exiplizit darauf hingewiesen, daß mit $p'(\lambda)$,wie allgemein üblich, die erste Ableitung von $p(\lambda)$ gemeint ist.

Lemma 3.7.3

$$\forall 1 \leq i \leq n : \; c_i = \frac{1}{n-i}\mathrm{tr}\,(AB_i)$$

Beweis Wie in 3.7.1 folgt zunächst aus 3.6.3 in Verbindung mit Definition 2.4.1:

$$\begin{aligned} & (A - \lambda E_n)c(\lambda) &&= p(\lambda)E_n \\ \Leftrightarrow \quad & Ac(\lambda) - \lambda c(\lambda) &&= p(\lambda)E_n \\ \Leftrightarrow \quad & \mathrm{tr}\,(Ac(\lambda)) &&= \mathrm{tr}\,(\lambda c(\lambda)) + \mathrm{tr}\,(p(\lambda)E_n) \end{aligned}$$

Mit Hilfe von 3.7.2 folgt:

$$\begin{aligned} & \mathrm{tr}\,(Ac(\lambda)) = np(\lambda) - \lambda p'(\lambda) \\ \Leftrightarrow \quad & \mathrm{tr}\,(AB_{n-1}\lambda^{n-1} + AB_{n-2}\lambda^{n-2} + \cdots + AB_2\lambda^2 + AB_1\lambda + AB_0) \\ = \quad & n(\lambda^n c_n + c_{n-1}\lambda^{n-1} + \cdots + c_2\lambda^2 + c_1\lambda + c_0) \\ & - (n\lambda^n c_n + (n-1)c_{n-1}\lambda^{n-1} + \cdots + 2c_2\lambda^2 + c_1\lambda) \end{aligned}$$

Koeffizientenvergleich ergibt

$$\forall 1 \leq i \leq n : \; \mathrm{tr}\,(AB_i) = nc_i - ic_i \;\; .$$

□

Satz 3.7.4 (Frame)

$$\det(A) = \frac{\mathrm{tr}\,(AB_0)}{n} \tag{3.20}$$

Beweis Man erhält die Behauptung aus 3.7.1, 3.7.3 und 2.4.4. □

3.8 Determinantenberechnung mit Hilfe des Satzes von Frame

In diesem Unterkapitel wird eine effiziente Methode zur parallelen Determinantenberechnung [Csa76] vorgestellt (abgekürzt mit *C-Alg.*; vgl. Unterkapitel 1.3). Sie benutzt Divisionen und kann deshalb nur angewendet werden, wenn die Berechnungen in einem Körper stattfinden. Dies ist problematisch, weil in realen Rechnern nur mit begrenzter Genauigkeit gearbeitet werden kann und somit immer auf die eine oder andere Weise modulo gerechnet wird. Z. B. besitzt 6 im Ring $\mathbb{Z}_8$ kein multiplikatives Inverses.

Dies motiviert den Entwurf von Algorithmen, die ohne Divisionen auskommen, und somit auch in Ringen anwendbar sind, wie BGH-Alg. und B-Alg. . P-Alg. benutzt wie C-Alg. ebenfalls Divisionen.

Die Determinantenberechnung erfolgt in dem Algorithmus, der in diesem Unterkapitel vorgestellt wird, mit Hilfe des Satzes von Frame (Satz 3.7.4). Der Satz nutzt die bereits in 2.4.4 erwähnte Tatsache aus, daß sich unter den Koeffizienten des charakteristischen Polynoms auch die Determiante befindet. Diese Eigenschaft des charakteristischen Polynoms wird auch in B-Alg. und P-Alg. in Verbindung mit anderen Verfahren zur Bestimmung der Koeffizienten verwendet.

Die Berechnung der Determinante nach Satz 3.7.4 erfolgt mit Hilfe einer Rekursionsgleichung. Für eine effiziente parallele Berechnung ist dies nicht befriedigend. Deshalb ist sind einige Umformungen [Csa76] erforderlich. Für diese Umformungen wird ein Operator benötigt. Dazu seien M und N jeweils $n \times n$-Matrizen:

Definition 3.8.1 Der Operator T wird definiert durch:

$$TN := \mathrm{tr}\,(N)$$

Er wird *Spuroperator* genannt.

Es gilt also

$$(E + MT)N = N + MTN = N + M\mathrm{tr}\,(N)\ .$$

Für die Determinantenberechnung nach Satz 3.7.4 ist die dort auftretende Matrix B_0 zu berechnen. Mit Hilfe der Lemmata 3.7.1 und 3.7.2 sowie des soeben definierten Operators T erhält diese Berechnung folgendes Aussehen:

$$\begin{aligned}
B_0 &= AB_1 - c_1E_n \\
&= AB_1 - \frac{E_n}{n-1}\mathrm{tr}\,(AB_1) \\
&= \left(E_n - \frac{E_n}{n-1}T\right)AB_1 \\
&\vdots \\
&= \left(E_n - \frac{E_n}{n-1}T\right)\left\{A\left[\left(E_n - \frac{E_n}{n-2}T\right)\{A[\cdots(E_n - E_nT)\{A[E_n]\}\cdots]\}\right]\right\}
\end{aligned}$$

Mit Hilfe der Assoziativität der Matrizenmultiplikation erhält man:

$$B_0 = \left(\underbrace{A - \frac{\overbrace{E}^{\text{Term 1}}}{n-1}\overbrace{TA}^{\text{Term 2}}}_{\text{Term 3}}\right)\left(A - \frac{E}{n-2}TA\right)\cdots\left(A - \frac{E}{2}TA\right)(A - ETA) \tag{3.21}$$

Da E_n nur in der Hauptdiagonalen von 0 verschiedene Elemente besitzt, läßt sich Term 1 in einem Schritt von

$$n$$

Prozessoren berechnen. Parallel dazu läßt sich Term 2 nach Satz 1.5.2 in

$$\lceil \log(n) \rceil$$

Schritten von

$$\left\lfloor \frac{n}{2} \right\rfloor$$

Prozessoren berechnen.

Anschließend ist die Ergebnismatrix von Term 1 mit dem Ergebnis von Term 2 zu multiplizieren. Dies kann, wie bei der Berechnung von Term 1 in einem Schritt von

$$n$$

Prozessoren durchgeführt werden. Die darauf folgende Matrizensubtraktion zur Berechnung von Term 3 kann in einem Schritt von

$$n^2$$

Prozessoren erledigt werden.

Insgesamt kann Term 3 also in

$$\lceil \log(n) \rceil + 2$$

Schritten von

$$n^2$$

Prozessoren berechnet werden.

Zur Berechnung von B_0 sind n Terme auf die gleiche Weise wie Term 3 zu berechnen. Term 2 braucht für all diese Terme nur einmal berechnet zu werden. Insgesamt kann die Berechnung der n Terme in

$$\lceil \log(n) \rceil + 2$$

Schritten von

$$n^3$$

Prozessoren erledigt werden.

Um das Endergebnis B_0 zu erhalten sind schließlich noch die Ergebnismatrizen der n Terme miteinander zu multiplizieren. Die Anzahl der Schritte und Prozessoren dafür folgt aus den Sätzen 1.5.1 und 1.5.3. Zu beachten ist dabei, daß eine Verknüpfung nicht in einem Schritt von einem Prozessor durchgeführt wird, sondern nach 1.5.3 in

$$\lceil \log(n) \rceil + 1$$

Schritten von

$$n^3$$

Prozessoren. Deshalb werden diese Matrizenmultiplikationen in

$$(\lceil \log(n) \rceil + 1)\lceil \log(n) \rceil$$

Schritten von

$$n^3 \left\lfloor \frac{n}{2} \right\rfloor$$

Prozessoren durchgeführt.

Insgesamt wird die Berechnung von B_0 also in

$$\lceil \log(n) \rceil^2 + 2\lceil \log(n) \rceil + 2$$

Schritten von

$$n^3 \left\lfloor \frac{n}{2} \right\rfloor$$

Prozessoren durchgeführt.

Um die Determinante zu berechnen, sind noch nacheinander durchzuführen:

1. eine Matrizenmultiplikation nach Satz 1.5.3 in

   $$\lceil \log(n) \rceil + 1$$

   Schritten von

   $$n^3$$

   Prozessoren,

2. die Berechnung der Spur[9] einer Matrix nach Satz 1.5.1 in

   $$\lceil \log(n) \rceil$$

   Schritten von

   $$\left\lfloor \frac{n}{2} \right\rfloor$$

   Prozessoren und

3. eine Division in einem Schritt von einem Prozessor.

[9] siehe Definition 2.1.3

Diese drei Berechnungsstufen werden insgesamt in

$$2\lceil \log(n) \rceil + 2$$

Schritten von

$$n^3$$

Prozessoren durchgeführt.

Die Berechnung der Determinanten mit Hilfe von Satz 3.7.4 kann also in

$$\lceil \log(n) \rceil^2 + 4 \lceil \log(n) \rceil + 4$$

Schritten durchgeführt werden. Die Anzahl der Prozessoren beträgt

$$\begin{aligned} & n^3 \left\lfloor \frac{n}{2} \right\rfloor \\ \leq \quad & \left\lceil \frac{n^4}{2} \right\rceil \end{aligned}$$

Man erkennt, daß C-Alg. keine Fallunterscheidungen verwendet. Dies ist ein Vorteil bei der Konstruktion von Schaltkreisen, da somit keine Teilschaltkreise für einzelne Zweige entworfen werden müssen. Dadurch wird ein Schaltkreis zur Determinantenberechnung mit Hilfe von C-Alg. nicht unnötig vergrößert. Es wird sich zeigen, daß die anderen Algorithmen (BGH-Alg., B-Alg. und P-Alg.) die Eigenschaft fehlender Fallunterscheidungen ebenfalls besitzen. In dieser Hinsicht besitzt keiner der Algorithmen einen Vorteil gegenüber den anderen.

Vergleicht man den Aufwand an Schritten und Prozessoren mit dem der anderen Algorithmen, zeigt sich, daß C-Alg. bereits sehr effizient ist.

Kapitel 4

Der Algorithmus von Borodin, Von zur Gathen und Hopcroft

Der Algorithmus [BvzGH82], der in diesem Kapitel dargestellt wird, verbindet die Vermeidung von Divisionen [Str73], das Gauß'sche Eliminationsverfahren (z. B. [BS87] S. 735) und die parallele Berechnung von Termen [VSBR83] miteinander, um die Determinante einer Matrix zu berechnen. Auf diesen Algorithmus wird mit *BGH-Alg.* Bezug genommen (vgl. Unterkapitel 1.3).

Er unterschiedet sich in seiner Methodik von den anderen Algorithemen (C-Alg., B-Alg. und P-Alg.) vor allem dadurch, daß er die Koeffizienten des charakteristischen Polynoms in keiner Weise beachtet (vgl. 2.4.4), sondern die Determinante durch miteinander verknüpfte Transformationen, nicht zuletzt auch durch Ausnutzung von Satz 2.1.8, direkt berechnet.

Wie sich in diesem Kapitel zeigen wird, besitzt der Algorithmus durch die Verbindung der drei o. g. Verfahren eine gewisse Eleganz, besonders, was die Handhabung der Konvergenz von Potenzreihen angeht.

Ein Nachteil des Algorithmus ist die vergleichsweise schlechte Effizienz.

4.1 Das Gauß'sche Eliminationsverfahren

Das Gauß'sche Eliminationsverfahren wird dazu benutzt, lineare Gleichungssysteme der Form

$$Ax = b$$

zu lösen. Dazu wird die sogenannte *erweiterte Koeffizientenmatrix* betrachtet. Sie ist eine $n \times (n+1)$-Matrix, deren erste n Spalten aus den Spalten der Koeffizientenmatrix A bestehen und deren $(n+1)$-te Spalte aus dem Vektor b besteht.

Die Idee des Gauß'schen Eliminationsverfahrens ist es, die erweiterte Koeffizientenmatrix so zu transformieren, daß die darin enthaltene Matrix A die Form einer *oberen Dreiecksmatrix* bekommt. Für eine solche $n \times n$-Matrix gilt:

$$\forall 1 \leq j < i \leq n : a_{i,j} = 0$$

Falls für die Matrix

$$\forall 1 \leq i < j \leq n : a_{i,j} = 0$$

erfüllt ist, nennt man sie *untere Dreiecksmatrix*. Zur Transformation werden *elementare Zeilenoperationen* verwendet. Sie werden in Definition 2.1.6 der Determinanten einer Matrix unter D1 und D3 beschrieben. Sie haben nicht nur die dort genannten Beziehungen zur Determinanten einer Matrix, sondern noch zusätzlich die Eigenschaft, daß sie, angewandt auf die erweiterte Koeffizientenmatrix, die Lösungsmenge des linearen Gleichungssystems unverändert lassen.

Für die Determinantenberechnung wird die erweiterte Koeffizientenmatrix nicht weiter beachtet. Alle Operationen beziehen sich nur auf die Matrix A. Die Matrizenelemente unterhalb der Hauptdiagonalen[1] werden spaltenweise durch Nullen ersetzt, beginnend mit der ersten Spalte. Die Transformationen werden durch folgende Gleichungen beschrieben[2]:

$$a_{i,j}^{(0)} := a_{i,j} \tag{4.1}$$

$$a_{i,j}^{(k)} := \begin{cases} a_{i,j}^{(k-1)} & : \quad i \leq k \\ a_{i,j}^{(k-1)} - a_{k,j}^{(k-1)} \frac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}} & : \quad i > k \end{cases} \tag{4.2}$$

Die so gewonnene Matrix $A^{(n)}$ ist die gesuchte obere Dreiecksmatrix. Betrachtet man Satz 2.1.8, erkennt man, daß sich die Determinante dieser Dreiecksmatrix dadurch berechnen läßt, daß man die Elemente der Hauptdiagonalen miteinander multipliziert. Da man nur die in 2.1.6 erwähnten Operationen verwendet hat, erhält man so auch die Determinante der Matrix A.

4.2 Potenzreihenringe

Im darzustellenden Algorithmus spielen Potenzreihenringe eine wichtige Rolle. Deshalb werden in diesem Unterkapitel die für uns interessanten Eigenschaften dieser Ringe behandelt. Für unsere Betrachtungen sind Ringe mit einer zusätzlichen Eigenschaft von besonderem Interesse:

Definition 4.2.1 Sei R ein Ring. Ein $x \in R$ wird als *Einheit*[3] bezeichnet, wenn es ein $y \in R$ gibt, so daß

$$x * y = 1 \ .$$

Gibt es in R solche Elemente, so wird R als *Ring mit Division durch Einheiten* bezeichnet.

Falls in diesem Kapitel von Ringen die Rede ist, sind immer Ringe mit Division durch Einheiten gemeint, falls nicht ausdrücklich etwas anderes angegeben wird.

Sei M eine Menge von Unbestimmten:

$$M := \{x_1, x_2, \ldots, x_u\} \ .$$

Dann heißt $R[M]$ *Ring über* M. Für $R[M]$ schreiben wir auch abkürzend $R[]$. Die Elemente von $R[]$ sind Terme, in denen neben den Elementen von R zusätzlich Elemente von M als Unbestimmte auftreten dürfen.

Analog zur Definition von $R[]$ wird $R[[M]]$ definiert als *Potenzreihenring über* M. Für $R[[M]]$ wird auch $R[[]]$ geschrieben. Die Elemente von $R[[]]$ besitzen folgendes Aussehen:

- Sei T eine Teilmenge[4] von $\mathbb{N}^{n^2}$.
- Für ein $e \in T$ bezeichne e_i das i-te Element.
- Für $e \in T$ sei
$$k_{e_1,e_2,\ldots,e_{n^2}} \in R$$
- Jedes $u \in R[[]]$ hat für geeignete k_i und eine geeignete Menge T die Form:
$$\sum_{e:\{e_1,e_2,\ldots,e_u\} \in T} k_{e_1,e_2,\cdots,e_u} \prod_{i=1}^{u} x_i^{e_i} \tag{4.3}$$

[1] Die Hauptdiagonale bilden $a_{1,1}$ bis $a_{n,n}$.

[2] Das Gauß'sche Eliminationsverfahren wird im weiteren Text so modifiziert, daß Divisionen durch Null nicht vorkommen können. Dieser Fall wird deshalb schon hier außer Acht gelassen.

[3] nicht zu verwechseln mit Einselement

[4] T kann unendlich groß sein

Die Summe der Glieder von u, für die gilt

$$\sum_{i=1}^{u} e_i = p$$

wird *homogene Komponente vom Grad p* genannt. Die homogene Komponente vom Grad 0 wird auch *konstanter Term* genannt.

Der Ring $R[]$ enthält $R[[]]$ als Unterring und dieser wiederum als Unterring den Ring der Polynome über den Unbestimmten M.

Der Potenzreihenring $R[[]]$ besitzt eine für uns besonders interessante Eigenschaft. Dazu zunächst der folgende Satz:

Satz 4.2.2 (Taylor) *Eine Funktion f sei in*

$$(x_0 - \alpha, x_0 + \alpha)$$

mit

$$\alpha > 0$$

$(n+1)$*-mal differenzierbar. Dann gilt für*

$$x \in (x_0 - \alpha, x_0 + \alpha)$$

die Taylorentwicklung

$$f(x) = \sum_{\nu=0}^{n} \frac{f^{(\nu)}(x_0)}{\nu!}(x - x_0)^{\nu} + R_n(x)$$

mit

$$R_n(x) := \frac{f^{(n+1)}(x_0 + \vartheta(x - x_0))}{(n+1)!}(x - x_0)^{n+1}$$

wobei

$$\vartheta \in (0, 1)$$

und x_0 *der sogenannte* Entwicklungspunkt *ist.*

Beweis [Hil74] S. 33-35 □

Weitere Literatur zum Thema 'Taylorreihen' ist z. B. [BS87] (S. 31 und 269). Ein Beispiel für die Anwendung von Satz 4.2.2 ist die Funktion

$$f_1(x) := \frac{1}{1 - x} \quad . \tag{4.4}$$

Sie ist unendlich oft differenzierbar mit dem Entwicklungspunkt $x_0 = 0$ erhält man die Potenzreihe

$$f_2(x) = \sum_{i=0}^{\infty} x^i \quad . \tag{4.5}$$

Der *Konvergenzradius* ([BS87] S. 366) beträgt 1, d. h. nur für

$$|x| < 1$$

gilt

$$f_1(x) = f_2(x) \quad .$$

Für den Konvergenzradius k wird das Intervall

$$(k, -k)$$

als *Konvergenzbereich* bezeichnet.

Satz 4.2.2 läßt sich auch auf mehrere Unbestimmte verallgemeinern. Für uns ist dabei nur folgendes interessant:

Seien

$$f, g \in R[[]] \ .$$

Der konstante Term von g sei gleich Null. Für die Unbestimmten gelte

$$x_1, \ldots, x_u \in (-1, 1) \ . \tag{4.6}$$

Sei g konvergent. Dann folgt aus Satz 4.2.2, daß sich in $R[[]]$ Divisionen der Form

$$\frac{f}{1-g} \tag{4.7}$$

ersetzen lassen durch

$$f * \underbrace{(1 + g + g^2 + \ldots)}_{(*)} \ . \tag{4.8}$$

Die Potenzreihe g wird als *innere* Reihe bezeichnet. Die Terme $(*)$ sind ebenfalls Potenzreihen und werden als *äußere* Reihen bezeichnet. Setzt man die *innere* Reihe in eine der *äußeren* ein, erhält man wiederum eine Potenzreihe. Diese wird als *Gesamtreihe* bezeichnet.

Im obigen Beispiel konvergiert die Gesamtreihe, falls die innere Reihe konvergiert und ihre Unbestimmten innerhalb des Konvergenzradius der äußeren liegen. Da diese Bedingungen erfüllt sind, folgt die Konvergenz der Reihe (4.8). Um die Konvergenz in praktisch nutzbarem Maße sicherzustellen, sollte der Betrag der Werte, die für die Unbestimmten eingesetzt werden, nicht beliebig nahe bei 1 liegen.

Konvergenz ist beim Umgang mit Potenzreihen ein wichtiges Thema. Besonders beim Verknüpfen von Potenzreihen mit mehreren Unbestimmten, wie im vorliegenden Fall, sind Konvergenzbetrachtungen u. U. komplex. Allgemeine Betrachtungen der Konvergenz von Potenzreihen mit mehreren Unbestimmten führen an dieser Stelle zu weit und sind z. B. in

- [BT70] ab S. 1 sowie ab S. 49 und
- [Hö73] ab S. 34

zu finden.

Bei praktischen Berechnungen können Potenzreihen nicht beliebig weit entwickelt werden, da die Rechenleistung beschränkt ist. Deshalb muß ein Grad festgelegt werden, bis zu dem die Potenzreihen entwickelt werden. Dieser Grad ist i. A. besonders von der Stärke der Konvergenz der Reihe abhängig, die entwickelt werden soll. Die Festsetzung eines solchen Grades erfordert eine Analyse des jeweiligen Problems, das mit Hilfe der Entwicklung in Potenzreihen gelöst werden soll. So kann eine Potenzreihe als Endergebnis mehrerer hintereinander durchgeführter Verknüpfungen von Potenzreihen u. U. auch dann konvergieren, wenn als Zwischenergebnis auftretende Reihen divergieren[5]

Da für uns an dieser Stelle weitere allgemeine Betrachtungen uninteressant sind, erfolgt die Konvergenzanalyse im Zusammenhang mit der Anwendung der Potenzreihenentwicklung auf unser Problem der Determinantenberechnung.

4.3 Das Gauß'sche Eliminationsverfahren ohne Divisionen

Die Möglichkeiten zur Vermeidung von Divisionen wurden von V. Strassen [Str73] allgemein untersucht. In diesem Unterkapitel wird dargestellt, wie sich Strassens Ergebnisse auf das Gauß'sche Eliminationsverfahren anwenden lassen.

[5] In dem Algorithmus zur Determinantenberechnung, der in diesem Kapitel vorgestellt wird, tritt diese Besonderheit auf. In [BvzGH82] wird darauf in keiner Weise eingegangen, was sich bei der Bearbeitung als störend herausgestellt hat.

Die Hauptidee zur Vermeidung von Divisionen ist es, alle Berechnungen nicht in einem Ring R mit Division durch Einheiten durchzuführen, sondern im zugehörigen Potenzreihenring $R[[]]$, wobei Matrizenelemente als Unbestimmte auftreten. Um die Berechnungen in diesen Ring zu übertragen, wird das Kroneckersymbol definiert als

$$\delta_{i,j} := \begin{cases} 1 & : \quad i = j \\ 0 & : \quad i \neq j \end{cases}$$

Es sei die Determinante der $n \times n$-Matrix A zu berechnen. Ihre Elemente werden mit Hilfe der Definition

$$a'_{i,j} := \delta_{i,j} - a_{i,j} \tag{4.9}$$

ersetzt. Das bedeutet, jedes Matrizenelement $a_{i,j}$ wird ersetzt durch

$$\delta_{i,j} - a'_{i,j} \ .$$

Wendet man nun das Gauß'sche Eliminationsverfahren an, bekommt jede Division die Form (4.7) und kann somit ersetzt werden durch (4.8), wie durch das Beispiel in Unterkapitel 4.4 deutlich wird.

Berechnet man mit Hilfe des Eliminationsverfahrens die Determinante von A, wie in 4.1 beschrieben ist, und rechnet dabei in $R[[]]$ statt in $R[]$, erhält man als Endergebnis eine Potenzreihe d' über den Unbestimmten $a'_{i,j}$, die die Determinante von A beschreibt.

In der praktischen Berechnung ersetzt man die $a'_{i,j}$ mit Hilfe von (4.9) durch konkrete Werte und wertet die Potenzreihe d' aus, um die Determinante als Element von R zu erhalten.

Ein bis hierhin ungelöstes Problem ist die Sicherstellung der Konvergenz von d'. Dazu ist die Frage zu beantworten:

> Wie groß ist der Konvergenzradius von d'?

Hierfür müssen wir zunächst eine Eigenschaft der Determinante näher betrachten[6]:

Lemma 4.3.1 *Es gibt genau eine Abbildung, die jeder Matrix ihre Determinante zuordnet.*

Beweis Der Beweis wird anhand der Matrix A geführt.

Seien f und $\hat{(}f)$ zwei voneinander verschiedene Abbildungen, mit den in der Definition 2.1.6 der Determinante beschriebenen Eigenschaften.

Es werden zwei Fälle unterschieden:

- Bei
  $$\mathrm{rg}\,(A) < n$$
  gilt nach Satz 2.2.4
  $$f(A) = \hat{f}(A) = 0 \ .$$
- Sei
  $$\mathrm{rg}\,(A) = n \ . \tag{4.10}$$
  Entsteht die Matrix B aus A durch Zeilenumformungen entsprechend D1 in Definition 2.1.6, dann gibt es ein $c \neq 0$, so daß gilt:
  $$f(B) = c * f(A) \ .$$
  Das gleiche gilt auch für $\hat{f}$:
  $$\hat{(}B) = c * \hat{(}A) \ .$$
  Wegen (4.10) ist es möglich, durch Zeilenumformungen
  $$B = E_n$$
  zu erreichen. Aus D4 in Definition 2.1.6 folgt dann:
  $$f(A) = \frac{1}{c} f(E_n) = \frac{1}{c} = \frac{1}{c} \hat{f}(E_n) = \hat{f}(A) \ .$$

[6] Literatur zu diesem Lemma ist die in Kapitel 2 aufgelistete Grundlagenliteratur.

In beiden Fällen gilt also $f = \hat{(}f)$ im Widerspruch zur Voraussetzung, daß f und $\hat{(}f)$ verschieden sind. □

Mit der Unterstützung durch dieses Lemma gelangt man zu einer wichtigen Aussage:

Satz 4.3.2 *Bezeichne d die Potenzreihe über den Unbestimmten $a_{i,j}$, die man aus d' (s. o.) dadurch erhält, daß man alle Unbestimmten $a'_{i,j}$ mit Hilfe von (4.9) ersetzt.*

Für d gilt:

Alle homogenen Komponenten mit einem Grad ungleich n sind gleich Null.

Beweis Aus der Richtigkeit der im vorliegenden Kapitel beschriebenen Verfahren folgt, daß d eine Determinante von A entsprechend der Definition 2.1.6 beschreibt.

Bezeichne f die nach Satz 2.1.8 berechnete Determinante von A als Summe, deren Summanden jeweils aus einem Produkt von n Matrizenelementen bestehen.

Nach Lemma 4.3.1 gilt:

$$d = f \ .$$

Betrachtet man die Termstruktur von d und beachtet, daß für die Matrizenelemente keine zusätzlichen Eigenschaften vorausgesetzt werden, folgt aus dieser Gleichung die Behauptung. □

Der Satz wird in Unterkapitel 4.4 an einer 3×3-Matrix demonstriert.

Sowohl d als auch d' beschreiben die Determinante von A. Der Konvergenzradius von beiden Reihen ist also *Unendlich*.

Aus Satz 4.3.2 folgt insbesondere, daß sich alle homogenen Komponenten mit einem Grad größer als n gegenseitig aufheben. Da alle Divisionen durch Additionen und Multiplikationen ersetzt worden sind, gehen diese Komponenten nicht in den Wert von Komponenten geringeren Grades ein. Komponenten mit einem bestimmten Grad beeinflussen im Verlauf der Rechnungen lediglich die Werte von Komponenten gleichen oder höheren Grades.

Also ist es unnötig, die homogenen Komponenten mit einem Grad größer als n überhaupt zu berechnen. Dies ist ein wichtiges Ergebnis für die Analyse der Effizienz des Algorithmus.

4.4 Beispiel zur Vermeidung von Divisionen

Für eine 3×3-Matrix wird in diesem Unterkapitel gezeigt, wie die Determinante mit Hilfe des Gauß'schen Eliminationsverfahrens ohne Divisionen berechnet wird[7]. Wie im vorangegangenen Unterkapitel 4.3 begründet ist, werden bei allen Potenzreihen nur die homogenenen Komponenten bis maximal zum Grad 3 betrachtet.

Es ist die Determinante von

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \tag{4.11}$$

[7] Das Beispiel wurde mit Hilfe eines Programms zur symbolischen Manipulation von Termen berechnet. Wegen der vielen Indizes ist das Nachrechnen ohne Computer nicht ratsam.

zu berechnen. Die Ersetzung mit Hilfe von Gleichung (4.9) ergibt:

$$\begin{bmatrix} 1-a'_{1,1} & 0-a'_{1,2} & 0-a'_{1,3} \\ 0-a'_{2,1} & 1-a'_{2,2} & 0-a'_{2,3} \\ 0-a'_{3,1} & 0-a'_{3,2} & 1-a'_{3,3} \end{bmatrix} .$$

Nun werden Vielfache der ersten Zeile von den folgenden Zeilen subtrahiert, und man erhält:

$$\begin{bmatrix} 1-a'_{1,1} & 0-a'_{1,2} & 0-a'_{1,3} \\ 0 & (1-a'_{2,2})-(0-a'_{1,2})\frac{(0-a'_{2,1})}{(1-a'_{1,1})} & (0-a'_{2,3})-(0-a'_{1,3})\frac{(0-a'_{2,1})}{(1-a'_{1,1})} \\ 0 & (0-a'_{3,2})-(0-a'_{1,2})\frac{(0-a'_{3,1})}{(1-a'_{1,1})} & (1-a'_{3,3})-(0-a'_{1,3})\frac{(0-a'_{3,1})}{(1-a'_{1,1})} \end{bmatrix} .$$

Durch Ersetzung der Divisionen und Vereinfachung der Terme erhält man:

$$\begin{bmatrix} 1-a'_{1,1} & 0-a'_{1,2} & 0-a'_{1,3} \\ 0 & \begin{array}{c}(1-a'_{2,2})-a'_{1,2}a'_{2,1}* \\ (1+a'_{1,1}+(a'_{1,1})^2+(a'_{1,1})^3)\end{array} & \begin{array}{c}(0-a'_{2,3})-a'_{1,3}a'_{2,1}* \\ (1+a'_{1,1}+(a'_{1,1})^2+(a'_{1,1})^3)\end{array} \\ 0 & \begin{array}{c}(0-a'_{3,2})-a'_{1,2}a'_{3,1}* \\ (1+a'_{1,1}+(a'_{1,1})^2+(a'_{1,1})^3)\end{array} & \begin{array}{c}(1-a'_{3,3})-a'_{1,3}a'_{3,1}* \\ (1+a'_{1,1}+(a'_{1,1})^2+(a'_{1,1})^3)\end{array} \end{bmatrix} .$$

Da nur die homogenen Komponenten bis maximal zum Grad 3 berücksichtigt werden, erhält man durch weitere Vereinfachung der Terme:

$$\begin{bmatrix} 1-a'_{1,1} & 0-a'_{1,2} & 0-a'_{1,3} \\ 0 & \begin{array}{c}1-(a'_{2,2}+a'_{1,2}a'_{2,1}+ \\ a'_{1,2}a'_{2,1}a'_{1,1})\end{array} & \begin{array}{c}0-(a'_{2,3}+a'_{1,3}a'_{2,1}+ \\ a'_{1,3}a'_{2,1}a'_{1,1})\end{array} \\ 0 & \begin{array}{c}0-(a'_{3,2}+a'_{1,2}a'_{3,1}+ \\ a'_{1,2}a'_{3,1}a'_{1,1})\end{array} & \begin{array}{c}1-(a'_{3,3}+a'_{1,3}a'_{3,1}+ \\ a'_{1,3}a'_{3,1}a'_{1,1})\end{array} \end{bmatrix} .$$

Man erkennt, daß alle Elemente der Hauptdiagonalen wieder die Form

$$1-g_{i,j}$$

und alle anderen Elemente wieder die Form

$$0-g_{i,j}$$

besitzen, wobei der konstante Term der $g_{i,j}$ jeweils gleich 0 ist. Bei der Fortsetzung des Eliminationsverfahrens können auftretende Divisionen also wiederum auf die gleiche Weise ersetzt werden.

Als nächstes wird ein Vielfaches der zweiten Zeile von der dritten subtrahiert. Dazu sei

$$\begin{aligned} a''_{2,2} &:= 1-(a'_{2,2}+a'_{1,2}a'_{2,1}+a'_{1,2}a'_{2,1}a'_{1,1}) \\ a''_{2,3} &:= 0-(a'_{2,3}+a'_{1,3}a'_{2,1}+a'_{1,3}a'_{2,1}a'_{1,1}) \\ a''_{3,2} &:= 0-(a'_{3,2}+a'_{1,2}a'_{3,1}+a'_{1,2}a'_{3,1}a'_{1,1}) \\ a''_{3,3} &:= 1-(a'_{3,3}+a'_{1,3}a'_{3,1}+a'_{1,3}a'_{3,1}a'_{1,1}) \\ a'''_{3,3} &:= a''_{3,3}-\frac{a''_{3,2}}{a''_{2,2}}a''_{2,3} . \end{aligned}$$

Man erhält die Matrix:

$$\begin{bmatrix} 1-a'_{1,1} & 0-a'_{1,2} & 0-a'_{1,3} \\ 0 & a''_{2,2} & a''_{2,3} \\ 0 & 0 & a'''_{3,3} \end{bmatrix} .$$

Da nur die homogenen Komponenten bis zum Grad 3 betrachtet werden sollen, erhält man durch Ersetzung der Division in der beschriebenen Weise und Vereinfachung der Terme[8] für $a''_{3,3}$:

$$\begin{aligned}
& a''_{3,3} - \frac{a''_{3,2}}{a''_{2,2}} a''_{2,3} \\
\rightarrow \quad & a''_{3,3} - a''_{3,2} \\
& *(1 + a'_{2,2} + a'_{1,2}a'_{2,1} + a'^2_{2,2} + a'_{1,2}a'_{2,1}a'_{1,1} \\
& \quad +2a'_{2,2}a'_{1,2}a'_{2,1} + a'^3_{2,2}) \\
& *a''_{2,3} \\
\rightarrow \quad & 1 - a'_{1,1}a'_{1,3}a'_{3,1} - a'_{1,2}a'_{2,3}a'_{3,1} - a'_{1,3}a'_{2,1}a'_{3,2} \\
& -a'_{1,3}a'_{3,1} - a'_{2,2}a'_{2,3}a'_{3,2} - a'_{2,3}a'_{3,2} - a'_{3,3} \ .
\end{aligned}$$

Um die Determinante zu berechnen, werden die Elemente der Hauptdiagonalen $a'_{1,1}$, $a''_{2,2}$ und $a'''_{3,3}$ miteinander multipliziert. Wiederum werden die Komponenten mit zu großem Grad weggelassen. Man erhält:

$$\begin{aligned}
& a'_{1,1}a''_{2,2}a'''_{3,3} \\
\rightarrow \quad & 1 - a'_{1,1}a'_{2,2}a'_{3,3} + a'_{1,1}a'_{2,2} + a'_{1,1}a'_{2,3}a'_{3,2} + a'_{1,1}a'_{3,3} - a'_{1,1} + a'_{1,2}a'_{2,1}a'_{3,3} \\
& -a'_{1,2}a'_{2,1} - a'_{1,2}a'_{2,3}a'_{3,1} - a'_{1,3}a'_{2,1}a'_{3,2} + a'_{1,3}a'_{2,2}a'_{3,1} - a'_{1,3}a'_{3,1} \\
& +a'_{2,2}a'_{3,3} - a'_{2,2} - a'_{2,3}a'_{3,2} - a'_{3,3} \ .
\end{aligned}$$

Um die gesuchte Determinante zu erhalten, setzt man die mit Hilfe von Gleichung (4.9) aus den $a_{i,j}$ erhaltenen Werte für die $a'_{i,j}$ ein.

Zum Beweis, daß wir richtig gerechnet haben, machen wir nun die durch (4.9) definierte Substitution im obigen Term wieder rückgängig. Nach der Vereinfachung des Terms lautet das Ergebnis, ohne daß zusätzlich irgendwelche Teilterme weggelassen worden sind:

$$\begin{aligned}
& a'_{1,1}a''_{2,2}a'''_{3,3} = \\
& \quad a_{1,1}a_{2,2}a_{3,3} + a_{1,2}a_{2,3}a_{3,1} + a_{1,3}a_{2,1}a_{3,2} \\
& \quad -a_{1,1}a_{2,3}a_{3,2} - a_{1,2}a_{2,1}a_{3,3} - a_{1,3}a_{2,2}a_{3,1} \ .
\end{aligned}$$

Die Richtigkeit dieses Ergebnisses wird beim Vergleich mit Satz 2.1.8 deutlich.

4.5 Parallele Berechnung von Termen

Durch die Methode von Strassen zur Vermeidung von Divisionen entstehen Terme, die es parallel auszuwerten gilt. In diesem Unterkapitel wird ein Verfahren [VSBR83] beschrieben, welches diese Auswertung ermöglicht. Die Beschreibung des Verfahrens ist auf die Verwendung im Rahmen des Kapitels angepaßt. Eine ausführliche Beschreibung ist auch in [Wal87] ab S. 22 zu finden.

Zunächst wird die Berechnung von Termen formalisiert. Dazu wird die Menge

$$\{v_i \mid 1 \leq i \leq c\}$$

mit V bezeichnet. Die Menge der Elemente $a_{i,j}$ der $n \times n$-Matrix A, die hier als Unbestimmte auftreten, wird mit X bezeichnet. Sei R der Ring, in dem alle Rechnungen durchgeführt werden. Es wird definiert

$$\bar{V} := V \cup X \cup R \ .$$

Definition 4.5.1 Sei $R[]$ der bereits erwähnte Ring über den Elementen von X. Sei $c \in \mathbb{N}$ gegeben. Seien $+$ und $*$ die beiden Ringoperatoren für Addition bzw. Multiplikation. Es gelte

$$\circ \in \{+, *\} \ .$$

[8] Da alle Prokukte aus mehr als 3 Unbestimmten sofort weggelassen werden, dürfen die Rechenschritte nicht durch = verbunden werden.

Weiterhin gelte

$$\forall 1 \leq i \leq c : \; v_i', v_i'' \in \bar{V} \backslash \{v_i, v_{i+1}, \ldots, v_c\} \; .$$

Jede Folge der Form

$$v_i := v_i' \circ v_i'', \; i = 1, \ldots, c$$

heißt dann *Programm über R[]* . Ein Element einer solchen Folge wird *Anweisung* genannt. Abhängig davon, ob $\circ$ die Addition oder die Multiplikation bezeichnet, wird das v_i auch als *Additions-* bzw. *Multiplikationsknoten* bezeichnet. Falls das genaue Aussehen von Anweisungen von untergeordnetem Interesse ist, werden diese zur Abkürzung durch ihren Additions- bzw. Multiplikationsknoten repräsentiert.

Falls in diesem Unterkapitel im Einzelfall nichts anderes festgelegt wird, ist mit v_i' bzw. v_i'' jeweils der erste bzw. zweite Operand der Anweisung v_i gemeint. Dies gilt auch dann, wenn andere Buchstaben benutzt werden oder keine Indizierung erfolgt.

Jeder Term über $R[]$ läßt sich durch ein Programm über $R[]$ berechnen. Um Aussagen über solche Programme machen zu können, sind eine Reihe weiterer Vereinbarungen erforderlich, die im folgenden aufgeführt sind.

Der durch eine Anweisung v_i berechnet Term wird mit $f(v_i)$ bezeichnet.

Sei $x \in \bar{V}$. Seien $x', x'' \in V$. Dann wird der *Grad von x* mit $g(x)$ bezeichnet und folgendermaßen definiert:

$$g(x) := \begin{cases} 0 & : \quad x \in R \\ 1 & : \quad x \in X \\ g(x') + g(x'') & : \quad (x \in V) \wedge (x := x' * x'') \\ \max(g(x'), g(x'')) & : \quad (x \in V) \wedge (x := x' + x'') \end{cases}$$

Der Grad von x stimmt nicht mit dem Grad des Polynoms überein, daß dem Term $f(x)$ entspricht. Dazu ein Beispiel:

$$\begin{array}{lll} v_1 := y * (-1) & f(v_1) = -y & g(v_1) = 1 \\ v_2 := y + v_1 & f(v_2) = 0 & g(v_2) = 1 \end{array}$$

Es wird o. B. d. A. angenommen, daß für jede Anweisung

$$x := x' \circ x''$$

die Bedingung

$$g(x') \geq g(x'')$$

erfüllt ist.

Für alle $a \in \mathbb{N}$ wird definiert:

$$\begin{array}{lll} V_a & := & \{u \in V \mid g(u) > a, \; u := u' * u'', \; g(u') \leq a\} \\ V_a' & := & \{u \in V \mid g(u) > a, \; u := u' + u'', \; g(u'') \leq a\} \end{array}$$

Definition 4.5.2 Sei $v \in V$. Sei $v_1, \ldots, v_k$ die längste Folge von Elementen von $\bar{V}$, so daß gilt

$$\begin{array}{rcl} & & v_1 = v \\ \forall 1 \leq i \leq k-1 & : & (v_{i+1} = v_i') \vee (v_{i+1} = v_i'') \\ & & v_k \in F \cup X \; . \end{array}$$

Dann bezeichnet $d(v) = k$ die *Tiefe von v*.

Definition 4.5.3 Sei $v, w \in \bar{V}$. Dann wird $f(v; w) \in R[]$ wie folgt definiert:

Bezeichnen v und w denselben Knoten, so gilt:

$$f(v; w) := 1 \ .$$

Falls dies nicht erfüllt ist und $w \in R \cup X$, dann gilt:

$$f(v; w) := 0 \ .$$

Falls dies ebenfalls nicht erfüllt ist und

$$w := w' + w'' \ ,$$

dann gilt

$$f(v; w) := f(v; w') + f(v; w'') \ .$$

Falls auch dies nicht erfüllt ist, bleibt nur noch der Fall übrig daß gilt

$$w := w' * w'' \ .$$

Dafür wird definiert

$$f(v; w) := f(v; w') * f(w'') \ .$$

Durch die Art und Weise, wie $f(v; w)$ definiert ist, ergibt sich eine besondere Eigenschaft für den Fall, daß $g(w) < 2g(v)$ erfüllt ist. Falls nämlich in dem Programm, zu dem v und w gehören, der Knoten v durch eine neue Unbestimmte v' ersetzt wird, dann ist $f(v; w)$ der Koeffizient von v' in $f(w)$.

In Verbindung mit $f(v; w)$ besitzen die Funktionen $g()$ und $d()$ eine Eigenschaft, die weiter unten von Bedeutung ist:

Lemma 4.5.4

$$g(v) > g(w) \Rightarrow f(v; w) = 0$$

Beweis Der Beweis erfolgt durch Induktion nach $d(w)$.

$d(w) = 0$

Es gilt:

$$\begin{aligned} g(w) = 0 &\Rightarrow w \in R \\ g(v) > g(w) &\Rightarrow v \in V \cup X \end{aligned}$$

Also ist $f(v; w) = 0$.

$d(w) > 0$

Das Lemma gelte für alle $u \in \bar{V}$, $d(u) < d(w)$. Aus 4.5.2 folgt direkt, daß für jede Anweisung

$$w := w' \circ w''$$

gilt

$$d(w) > d(w'),\ d(w) > d(w'') \ .$$

Mit Hilfe von 4.5.3 folgt daraus die Gültigkeit des Lemmas.

□

Lemma 4.5.5

$$d(v) > d(w) \Rightarrow f(v; w) = 0$$

Beweis analog zu 4.5.4 □

Es lassen sich nun zwei Aussagen formulieren. Dazu gelte jeweils $v, w \in V$ und $0 < g(v) \leq a < g(w)$.

Lemma 4.5.6

$$f(v;w) = \sum_{u \in V_a} (f(v;u) * f(u;w)) + \sum_{u \in V'_a} (f(v;u'') * f(u;w))$$

Beweis Der Beweis erfolgt durch Induktion nach $d(w)$. Aufgrund der Struktur der zu beweisenden Aussage sind die Beweise von Induktionsanfang und Induktionsschluß nicht voneinander getrennt.

Wegen der Voraussetzung

$$0 < a < d(w)$$

folgt aus

$$d(w) \leq 1 \Rightarrow w \in R \cup X \ ,$$

daß $d(w) = 1$ nicht auftreten kann. Sei im folgenden also $d(w) > 1$.

Vier Fälle sind zu unterscheiden:

$w := w' + w''$, $g(w'') \leq a$

Das Lemma gelte für w'. Aus der Voraussetzung folgt:

$$w \in V'_a \ .$$

Aus 4.5.5 folgt:

$$f(w;w') = 0 \ .$$

Außerdem gilt:

$$g(w'') \leq a \Rightarrow \forall u \in V'_a : \ f(u;w'') = 0 \ .$$

Nach 4.5.3 gilt:

$$f(w,w) = 1 \ .$$

So ergibt sich:

$$\begin{aligned} f(v;w') &= \sum_{u \in V_a} (f(v;u) * f(u,w')) \\ &\quad + \sum_{u \in V'_a} (f(v;u'') * f(u;w')) \\ &= \sum_{u \in V_a} (f(v;u) * (f(u,w') + f(u;w''))) \\ &\quad + \sum_{u \in V'_a} (f(v;u'') * (f(u;w') + f(u;w''))) \\ &= \sum_{u \in V_a} (f(v;u) * f(u,w)) \\ &\quad + \sum_{u \in V'_a \setminus \{w\}} (f(v;u'') * f(u;w)) \end{aligned}$$

Es folgt mit Hilfe von 4.5.3:

$$\begin{aligned} f(v;w) &= f(v;w') + f(v;w'') \\ &= f(v;w') + f(v;w'') * f(w;w) \\ &= \sum_{u \in V_a} (f(v;u) * f(u,w)) \\ &\quad + \sum_{u \in V'_a \setminus \{w\}} (f(v;u'') * f(u;w)) + f(v;w'') * f(w;w) \\ &= \sum_{u \in V_a} (f(v;u) * f(u,w)) \\ &\quad + \sum_{u \in V'_a} (f(v;u'') * f(u;w)) \end{aligned}$$

$w := w' + w''$, $g(w'') > a$

Das Lemma gelte für w' und w''.

$$\begin{aligned} f(v;w) &= f(v;w') + f(v;w'') \\ &= \sum_{u\in V_a} (f(v;u) * f(u;w')) + \sum_{u\in \bar{V}_a} (f(v;u'') * f(u;w')) \\ &\quad + \sum_{u\in V_a} (f(v;u) * f(u;w'')) + \sum_{u\in \bar{V}_a} f(v;u'') * f(u;w'')) \\ &= \sum_{u\in V_a} (f(v;u) * (f(u;w') + f(u;w''))) \\ &\quad + \sum_{u\in \bar{V}_a} (f(v;u'') * (f(u;w') + f(u;w''))) \\ &= \sum_{u\in V_a} (f(v;u) * f(u;w)) + \sum_{u\in \bar{V}_a} (f(v;u'') * f(u;w)) \end{aligned}$$

$w := w' * w''$, $g(w') \leq a$

Es gilt:

$$\begin{aligned} w &\in V_a \\ f(v;w) &= f(v;w) * f(w;w) \ . \end{aligned}$$

Andererseits gilt:

$$\begin{aligned} \forall u \in V_a\backslash\{w\} &: f(u;w') = 0 \\ \Rightarrow \forall u \in V_a\backslash\{w\} &: f(u;w) = f(w'') * f(u;w') = f(w'') * 0 = 0 \end{aligned}$$

Also folgt:

$$\sum_{u\in V_a} (f(v;u) * f(u;w)) = f(v;w) * f(w;w) = f(v;w) \ .$$

Weiterhin folgt aus 4.5.4:

$$\begin{aligned} &\forall u \in V'_a : \ f(u;w') = 0 \\ &\quad \Rightarrow \sum_{u\in V'_a} (f(u'') * f(u;w)) \\ &\qquad = \sum_{u\in V'_a} (f(u'') * f(w'') * f(u;w')) = 0 \end{aligned}$$

Also ist das Lemma für diesen Fall richtig.

$w := w' * w''$, $g(w') > a$

Das Lemma gelte für w'.

$$\begin{aligned} f(v;w) &= f(w'') * f(v;w') \\ &= f(w'') * \left(\sum_{u\in V_a} (f(v;u) * f(u;w')) + \sum_{u\in V'_a} (f(v;u'') * f(u;w')) \right) \\ &= \sum_{u\in V_a} (f(v;u) * f(w'') * f(u;w')) + \sum_{u\in V'_a} (f(v;u'') * f(w'') * f(u;w')) \\ &= \sum_{u\in V_a} (f(v;u) * f(u;w)) + \sum_{u\in V'_a} (f(v;u'') * f(u;w)) \end{aligned}$$

□

Lemma 4.5.7

$$f(w) = \sum_{u \in V_a} (f(u) * f(u;w)) + \sum_{u \in V'_a} (f(u'') * f(u;w))$$

Beweis Bis auf den Unterschied, daß die auftretenden Terme entsprechend unterschiedlich sind, ist der Beweis identisch zum Beweis von 4.5.6. □

Mit Hilfe von 4.5.6 und 4.5.7 läßt sich ein Verfahren zur parallelen Berechnung von Termen angeben, das im folgenden beschrieben wird.

Gegeben sei ein Programm der Länge c, d. h.

$$V = \{v_1, v_2, \ldots, v_c\} \ .$$

Es ist $f(v_c)$ zu berechnen. Die Berechnung erfolgt stufenweise. Seien $v, w \in V$. In Stufe 0 werden alle $f(w)$ mit

$$g(w) = 1$$

und alle $f(v;w)$ mit

$$g(w) - g(v) = 1$$

berechnet.

In Stufe i werden alle $f(w)$ mit

$$2^{i-1} < g(w) \leq 2^i$$

und alle $f(v;w)$ mit

$$2^{i-1} < g(w) - g(v) \leq 2^i$$

berechnet. Dabei werden die Ergebnisse der vorangegangenen Stufen benutzt.

Auf diese Weise ist $f(v_c)$ nach

$$\lceil \log(g(v_c)) \rceil$$

Stufen berechnet.

In Stufe i werden zunächst die $f(w)$ mit Hilfe von 4.5.7 berechnet. Dazu wird $a = 2^{i-1}$ gewählt:

$$\begin{aligned} f(w) &= \sum_{u \in V_a} (f(u) * f(u;w)) + \sum_{u \in V'_a} (f(u'') * f(u;w)) \\ &= \sum_{u \in V_a} (f(u') * f(u'') * f(u;w)) + \sum_{u \in V'_a} (f(u'') * f(u;w)) \end{aligned} \tag{4.12}$$

Anhand der Definitionen erkennt man, daß für alle auftretenden $f(\ldots)$ gilt:

$$g(f(\ldots)) \leq 2^{i-1} \ .$$

Also wurden alle zu benutzenden Terme bereits in einer der vorangegangenen Stufen berechnet.

Man erkennt anhand der bisher angestellten Betrachtungen über Programme zur Berechnung von Termen, daß der Aufwand für alle Programme der Länge c gleich ist. Da eine Aufgabe, die in a Schritten von b Prozessoren erledigt wird, auch in $2a$ Schritten von $b/2$ Prozessoren erledigt werden kann, erfolgt die Analyse des Aufwandes für eine bestimmte Stufe i zunächst mit Hilfe der durchschnittlich für eine Stufe zu erwartenden erforderlichen Operationen[9].

Für eine bestimmte Stufe läßt sich die Größe der Mengen V_a und V'_a nicht genau vorherbestimmen. Falls die Berechnung in z Stufen durchgeführt wird, dann gilt jedoch

$$\begin{aligned} &0 \leq i, j \leq z \\ &i \neq j \\ &a_k := 2^{k-1} \\ &V_{a_i} \cap V_{a_k} = V'_{a_i} \cap V'_{a_k} = \emptyset \\ &\sum_{0 \leq i \leq z} |V_{a_i}| \leq c \\ &\sum_{0 \leq i \leq z} |V'_{a_i}| \leq c \end{aligned}$$

[9] Diese Betrachtungsweise kennt man in der Literatur unter dem Begriff *Rescheduling*.

Die Mengen V_a und V'_a besitzen also durchschnittlich höchstens

$$\frac{c}{z} = \frac{c}{\lceil \log(g(v_c)) \rceil}$$

Elemente. Dieser Wert wird mit m bezeichnet.

Aus den vorangegangenen Überlegungen folgt, daß (4.12) in

$$\lceil \log(m) \rceil + 3 = \left\lceil \log \left(\frac{c}{\lceil \log(g(v_c)) \rceil} \right) \right\rceil + 3$$

Schritten von

$$2m = 2\frac{c}{\lceil \log(g(v_c)) \rceil}$$

Prozessoren berechnet werden kann.

Nachdem in Stufe i die $f(w)$ berechnet worden sind, werden die $f(v;w)$ mit Hilfe von 4.5.6 ausgerechnet. Dazu wird $a = g(v) + 2^{i-1}$ gewählt:

$$\begin{aligned} f(v;w) &= \sum_{u \in V_a} (f(v;u) * f(u;w)) + \sum_{u \in V'_a} (f(v;u'') * f(u;w)) \\ &= \sum_{u \in V_a} (f(u'') * f(v;u') * f(u;w)) + \sum_{u \in V'_a} (f(v;u'') * f(u;w)) \end{aligned}$$

Anhand der Definitionen erkennt man, daß alle $f(v;u')$, $f(u;w)$ und $f(v;u'')$ bereits berechnet wurden. Für $f(u'')$ gibt es kritische Fälle, die separat untersucht werden müssen:

$g(u') \geq g(u'') > 2^i,\ f(v;u') = 0$
: Der Fall ist kein Problem, da der Wert des jeweiligen gesamten Terms gleich Null ist.

$g(u') \geq g(u'') > 2^i,\ f(v;u') \neq 0$
: Es muß gelten:

$$g(u') \geq g(v) \ .$$

Daraus ergibt sich:

$$\begin{aligned} g(u) &= g(u') + g(u'') \\ &> g(v) + 2^i \\ &\geq g(w) \\ &\Rightarrow f(u;w) = 0 \end{aligned}$$

Der Wert des Terms ist also wiederum gleich Null.

Für die Analyse des Aufwandes gelten die gleichen Bemerkungen wie für die Berechnung der $f(w)$.

Insgesamt kann $f(v_c)$ in

$$2\lceil \log(g(v_c)) \rceil (\lceil \log(m) \rceil + 3) = 2\lceil \log(g(v_c)) \rceil \left(\left\lceil \log \left(\frac{c}{\lceil \log(g(v_c)) \rceil} \right) \right\rceil + 3 \right) \tag{4.13}$$

Schritten von

$$2m = 2\frac{c}{\lceil \log(g(v_c)) \rceil} \tag{4.14}$$

Prozessoren berechnet werden.

Für das bis hierhin beschriebene und analysierte Verfahren gibt es einen Sonderfall, der mit geringerem Aufwand gelöst werden kann.

Definition 4.5.8 Sei $v_1, \ldots, v_c$ ein Programm im Sinne von 4.5.1. Falls für alle Additionsknoten $v_i := v_i' + v_i''$ dieses Programms gilt:

$$g(v_i') = g(v_i'') \ ,$$

so wird das Programm als *homogen* bezeichnet.

Für homogene Programme sind alle Mengen V_a' leer. Mit dieser Feststellung ergeben sich aus 4.5.6 und 4.5.7 zwei Folgerungen für homogene Programme:

Folgerung 4.5.9

$$f(v;w) = \sum_{u \in V_a} (f(v;u) * f(u;w))$$

Folgerung 4.5.10

$$f(w) = \sum_{u \in V_a} (f(u) * f(u;w))$$

Werden im angegebenen Verfahren zur parallelen Berechnung von Termen die Folgerungen 4.5.9 und 4.5.10 statt der Lemmata 4.5.6 und 4.5.7 benutzt, so führt das zu leicht verringertem Berechnungsaufwand. Dann kann $f(v_c)$ analog zur obigen Analyse für $f(v_c)$ bei nicht homogenen Programmen in

$$2\lceil\log(g(v_c))\rceil \left(\left\lceil \log \left(\frac{c}{\lceil\log(g(v_c))\rceil} \right) \right\rceil + 2 \right) \tag{4.15}$$

Schritten von

$$\frac{c}{\lceil\log(g(v_c))\rceil} \tag{4.16}$$

Prozessoren berechnet werden.

4.6 Das Gauß'sche Eliminationsverfahren parallelisiert

Das Thema dieses Unterkapitels ist es, wie das in 4.5 beschriebene Verfahren benutzt werden kann, um mit Hilfe des in 4.3 angegebenen Gauß'schen Eliminationsverfahrens ohne Divisionen parallel die Determinante einer $n \times n$-Matrix zu berechnen. Auf den so entstehenden Algorithmus wird mit BGH-Alg. Bezug genommen (vgl. Unterkapitel 1.3).

Da keine Divisionen durchgeführt werden, ist BGH-Alg. ebenso wie B-Alg. auch in Ringen anwendbar. In dieser Hinsicht besitzen die beiden Algorithmen gegenüber C-Alg. und P-Alg. einen Vorteil.

Im folgenden wird beschrieben, wie ein Programm im Sinne von 4.5 anhand der Ergebnisse von 4.3 aufgebaut wird. Um die Auswirkungen des Grades, bis zu dem Potenzreihen entwickelt werden, auf die Effizienz der Rechnung besser demonstrieren zu können, werden die folgenden Betrachtungen zunächst unabhängig von einem konkreten Grad durchgeführt. Für alle Potenzreihen werden die homogenen Komponenten bis maximal zum Grad s betrachtet.

Es gibt drei wesentliche Elementaroperationen für Potenzreihen, die zunächst auf ihren Aufwand hin untersucht werden. Da sich nach 4.5 die Anzahl der Prozessoren und der Schritte aus der Programmlänge ergibt, wird im folgenden nur die Anzahl der Anweisungen im Sinne von 4.5.1 betrachtet:

Addition
: Dieser Fall gilt für *Subtraktion* analog. Es werden die homogenen Komponenten gleichen Grades addiert. Da die homogenen Komponenten bis zum Grad s betrachtet werden, sind hierfür $s + 1$ Anweisungen erforderlich.

Multiplikation

Seien a und b die zu multiplizierenden Potenzreihen. Für eine Potenzreihe x bezeichne x_i deren homogene Komponente vom Grad i. Das Ergebnis der zu Multiplikation von a und b sei c. Man erhält c mit:

$$c_i := \sum_{j=0}^{i} a_j * b_{i-j} \ .$$

Da für c auch nur die homogenen Komponenten bis zum Grad s berechnet werden müssen, folgt mit Hilfe der Gleichung

$$2^i = \sum_{j=0}^{i-1} 2^j + 1$$

für die Anzahl der Anweisungen bei Benutzung der Binärbaummethode nach 1.5.1:

$$\begin{aligned}
& \sum_{i=0}^{s} \left(i + \sum_{j=0}^{\lceil \log(i) \rceil - 1} 2^j \right) \\
= & \sum_{i=0}^{s} \left(i + 2^{\lceil \log(i) \rceil} - 1 \right) \\
\leq & \sum_{i=0}^{s} \left(i + 2^{\log(i)+1} - 1 \right) \\
= & \sum_{i=0}^{s} (3i - 1) \\
= & \ 3 \sum_{i=0}^{s} i - (s+1) \\
= & \ \frac{3}{2} s(s+1) - (s+1) \\
= & \ \frac{3s^2 + s - 2}{2} \ .
\end{aligned}$$

Division

Die Divisionen werden entsprechend der Ausführungen in 4.2 und 4.3 durch Additionen und Multiplikationen ersetzt (vgl. S. 48). Da nur die homogenen Komponenten bis zum Grad s betrachtet werden, erfolgt die Potenzreihenentwicklung wie in (4.8) nur bis zum s-ten Glied.

Somit sind s Multiplikationen und $s-1$ Additionen von Potenzreihen sowie die Addition des konstanten Terms durchzuführen. In Verbindung mit den vorangegangenen Analysen von Addition und Multiplikation ergibt sich für die Anzahl der Anweisungen:

$$\begin{aligned}
& s * \left(\frac{3s^2 + s - 2}{2} \right) + (s-1) * (s+1) + 1 \\
= & \ \frac{3s^3 + 3s^2 - 2s}{2} \ .
\end{aligned}$$

Als nächstes wird untersucht, wieviele der einzelnen Elementaroperationen zur Berechnung der Determinante benutzt werden. Dazu werden zwei Gleichungen benutzt:

Bemerkung 4.6.1 *Sei $n \in \mathbb{N}_0$. Dann gilt:*

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

Bemerkung 4.6.2 *Sei $n \in \mathbb{N}_0$. Dann gilt:*

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Das in 4.1 beschriebene Verfahren verwendet die Gleichungen (4.1) und (4.2). Werden mit Hilfe dieser Gleichungen zunächst alle Matrizenelemente transformiert, beträgt die Anzahl der Berechnungen neuer Elemente:

$$\begin{aligned}
& \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (n-(j-1)) \\
= \quad & \sum_{i=2}^{n} \sum_{j=i}^{n} (n-(j-1)) \\
= \quad & \sum_{i=2}^{n} \sum_{j=1}^{n-(i-1)} j \\
\overset{\text{nach 4.6.1}}{=} \quad & \sum_{i=2}^{n} \frac{(n-(i-1)) * ((n-(i-1))+1)}{2} \\
= \quad & \frac{1}{2} \sum_{i=2}^{n} ((n-i+1) * (n-i+2)) \\
= \quad & \frac{1}{2} \sum_{i=2}^{n} (n^2 + 3n - 2ni + i^2 - 3i + 2) \\
= \quad & \frac{1}{2} \left((n-1)(n^2+3n+2) + \sum_{i=2}^{n} i^2 - \sum_{i=2}^{n} 2ni - \sum_{i=2}^{n} 3i \right) \\
\overset{\text{nach 4.6.1,4.6.2}}{=} \quad & \frac{1}{2} \left((n^3 + 2n^2 - n - 2) + \frac{1}{6} n(n+1)(2n+1) - 1 \right. \\
& \left. -2n \left(\frac{n(n+1)}{2} - 1 \right) - 3 \left(\frac{n(n+1)}{2} - 1 \right) \right) \\
= \quad & \frac{1}{2} \left((n^3 + 2n^2 - n - 2) + \frac{2n^3 + 3n^2 + n}{6} - 1 \right. \\
& \left. -(n^3 + n^2 - 2n) - \left(\frac{3(n^2+n)}{2} - 3 \right) \right) \\
= \quad & \frac{1}{6}(n^3 - n)
\end{aligned}$$

Für jede einzelne Transformation eines Matrixelements werden nach (4.2) eine Subtraktion, eine Multiplikation und eine Division durchgeführt. Da alle Rechnungen in $R[[]]$ erfolgen, werden dabei Potenzreihen miteinander verknüpft, wofür der Aufwand gemessen in durchzuführenden Anweisungen bereits analysiert worden ist (s. o.). Für die identische Abbildung nach (4.1) wird kein Aufwand in Rechnung gestellt. So kommt man auf

$$\begin{aligned}
& \frac{1}{6}(n^3 - n) * \left(s + 1 + \frac{3s^2 + s - 2}{2} + \frac{3s^3 + 3s^2 - 2s}{2} \right) \\
= \quad & \frac{1}{4} \left(n^3 s^3 + 2n^3 s^2 + \frac{n^3 s}{3} - ns^3 - 2ns^2 - \frac{ns}{3} \right)
\end{aligned} \tag{4.17}$$

Anweisungen, um eine gegebene Matrix mit Hilfe des Gauß'schen Verfahrens in eine obere Dreiecksmatrix zu überführen. Zu Berechnung der Determinante sind im Anschluß daran noch die Elemente der Hauptdiagonalen miteinander zu multiplizieren. Dies kann mit $n-1$ Multiplikationen geleistet werden, denen

$$\begin{aligned}
& (n-1) * \frac{3s^2 + s - 2}{2} \\
= \quad & \frac{1}{2}(3ns^2 + ns - 2n - 3s^2 - s + 2)
\end{aligned} \tag{4.18}$$

Anweisungen entsprechen. So hat man bereits das Ergebnis als Element von $R[[]]$. Um die Determinante als Element von R zu erhalten, müssen nun noch die homogenen Komponenenten bis zum Grad s addiert werden. Dies kann mit Hilfe von s Anweisungen erfolgen. Abgesehen von diesen Additionen ist das Programm homogen im Sinne von 4.5.8. Deshalb ist es von Vorteil, die in 4.5 beschriebene Methode auf das Programm ohne die letzten Additionen anzuwenden und diese Additionen mit Hilfe der Binärbaummethode nach 1.5.1 durchzuführen. Die Addition der homogenen Komponenten kann so in

$$\lceil \log(s+1) \rceil$$

Schritten von

$$\left\lfloor \frac{s+1}{2} \right\rfloor$$

Prozessoren geleistet werden.

Man erhält das Gesamtergebnis für die Programmlänge ohne die letzten Additionen als Summe von (4.17) und (4.18):

$$\frac{1}{4}\left(n^3s^3 + 2n^3s^2 + \frac{1}{3}n^3s - ns^3 + ns^2 + \frac{5}{3}ns - 4n - 6s^2 - 2s + 4\right)$$

Anweisungen. Dieser Wert wird entsprechend der Terminologie in 4.5 mit c bezeichnet. Da bei allen Rechnungen nur die homogenen Komponenten bis zum Grad s beachtet werden, gilt

$$g(v_c) = s \ .$$

Aus c und $g(v_c)$ erhält man mit Hilfe der Analyseergebnisse (4.15) und (4.16) aus 4.5 für die in diesem Kapitel beschriebene Methode zur parallelen Determinantenberechnung einen Aufwand von[10]

$$\begin{aligned} & 2\lceil \log(s) \rceil * (\lceil \log(c) \rceil + 2) + \lceil \log(s+1) \rceil \\ = \ & 2\lceil \log(s) \rceil \\ & * \left(\left\lceil \log\left(\frac{1}{4}\left(n^3s^3 + 2n^3s^2 + \frac{1}{3}n^3s - ns^3 + ns^2 + \frac{5}{3}ns - 4n - 6s^2 - 2s + 4\right)\right)\right\rceil + 2\right) \\ & + \lceil \log(s+1) \rceil \end{aligned}$$

Schritten und

$$\begin{aligned} & \max\left(c\,,\left\lfloor \frac{s}{2} \right\rfloor\right) \\ = \ & c \\ = \ & \frac{1}{4}\left(n^3s^3 + 2n^3s^2 + \frac{1}{3}n^3s - ns^3 + ns^2 + \frac{5}{3}ns - 4n - 6s^2 - 2s + 4\right) \end{aligned}$$

Prozessoren. Man erkennt an diesen Werte die Bedeutung des Parameters s, dem maximal berücksichtigten Grad der homogenen Komponenten der Potenzreihen. Betrachtet man s als Konstante, so kann der Algorithmus in

$$O(\log(n))$$

Schritten von

$$O(n^3)$$

Prozessoren bearbeitet werden.

Die Analyse in Unterkapitel 4.3 ergibt, daß $s = n$ zu setzen ist, so daß der Algorithmus in

$$O(\log^2(n))$$

Schritten von

$$O(n^6)$$

[10] Genau genommen muß der Wert noch um 1 erhöht werden für die Berechnung der $a'_{i,j}$ aus den ursprünglichen Matrizenelementen $a_{i,j}$ entsprechend Gleichung (4.9).

Prozessoren bearbeitet werden kann.

Die Aufwandanalyse ergibt, daß BGH-Alg. insbesondere bei der Größenordnung der Anzahl der Prozessoren deutlich hinter C-Alg., B-Alg. und P-Alg. zurückliegt. Die Konstanten bei der Anzahl der Schritte sind ebenfalls vergleichsweise schlecht.

In BGH-Alg. werden, wie bei den anderen drei Algorithmen, keine Fallunterscheidungen verwendet, was aus den bereits in Unterkapitel 3.8 erwähnten Gründen beim Entwurf von Schaltkreisen vorteilhaft ist.

Betrachtet man die Methodik von BGH-Alg., so ist er P-Alg. am ähnlichsten. Beide fassen mehrere auch unabhängig voneinander bedeutsame Verfahren zu einem Algorithmus zur Determiantenberechnung zusammen. C-Alg. und B-Alg. hingegen stützen sich jeweils auf bestimmte schon seit 40 bis 50 Jahren bekannte Sätze, die nach einigen Umformungen für einen parallelen Algorithmus verwendet werden.

Kapitel 5

Der Algorithmus von Berkowitz

Der in diesem Kapitel vorgestellte Algorithmus [Ber84] berechnet die Determinante mit Hilfe einer rekursiven Beziehung zwischen den charakteristischen Polynomen einer Matrix und ihren Untermatrizen. Dabei wird 2.4.4 ausgenutzt. Auf den Algorithmus wird mit *B-Alg.* Bezug genommen[1].

Wie in BGH-Alg., werden keine Divisionen verwendet[2].

5.1 Toepliz-Matrizen

Im darzustellenden Algorithmus spielen Toepliz-Matrizen (Definition s. u.) eine wichtige Rolle und werden deshalb in diesem Unterkapitel behandelt.

Eine Matrix $n \times p$-Matrix A heißt *Toepliz-Matrix*, falls gilt:

$$a_{i,j} = a_{i-1,j-1},\ 1 < i \leq n,\ 1 < j \leq p\ .$$

Sie hat also folgendes Aussehen:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & \cdots \\ a_{2,1} & a_{1,1} & a_{1,2} & a_{1,3} & \ddots \\ a_{3,1} & a_{2,1} & a_{1,1} & a_{1,2} & \ddots \\ a_{4,1} & a_{3,1} & a_{2,1} & a_{1,1} & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}$$

Die folgende Eigenschaft von Toepliz-Matrizen ist für uns wichtig:

Satz 5.1.1 *Sei A eine $n \times p$-Matrix und B eine $p \times m$-Matrix. Beide seien untere Dreiecks-Toeplitz-Matrizen. Falls für die Matrix C gilt*

$$C = A * B\ ,$$

dann ist C ebenfalls eine untere Dreiecks-Toeplitz-Matrix. Sie kann in

$$\lceil \log(p) \rceil + 1$$

[1] vgl. Unterkapitel 1.3
[2] vgl. Erläuterungen in Unterkapitel 3.8

Schritten von

$$\begin{aligned} & \frac{\min(p,\, m) * (\min(p,\, m) + 1)}{2} + p * \max(n - p, 0) \\ \leq \quad & n * p \end{aligned}$$

Prozessoren berechnet werden.

Beweis Es sind drei Eigenschaften von C zu zeigen:

1. C ist eine untere Dreiecksmatrix.
2. C ist eine Toeplitz-Matrix.
3. C kann mit dem oben angegebenen Aufwand an Schritten und Prozessoren berechnet werden.

Dies geschieht in drei entsprechenden Beweisschritten. Dazu ist zu beachten, daß die einzelnen Elemente von C nach der Gleichung für die Matrizenmultiplikation berechnet werden:

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} b_{k,j} \tag{5.1}$$

1. Um zu beweisen, daß C ebenfalls eine untere Dreiecksmatrix darstellt, ist zu zeigen

   $$i < j \Rightarrow c_{i,j} = 0$$

   Dies erfolgt durch Fallunterscheidung anhand des Index k in Gleichung (5.1). Es gibt zwei Fälle:

   $i < k$
   : Da A nach Voraussetzung eine untere Dreiecksmatrix ist und somit

     $$i < j \Rightarrow a_{i,j} = 0$$

     gilt, folgt

     $$a_{i,k} = 0 \ ,$$

     wodurch der entsprechende Summand in Gleichung (5.1) zu 0 wird.

   $i \geq k$
   : Nach Voraussetzung gilt

     $$i < j \ ,$$

     da für die Elemente oberhalb der Hauptdiagonalen von C zu zeigen ist, daß sie gleich 0 sind. Daraus folgt aber

     $$k < j \ .$$

     Nach Voraussetzung ist B ebenfalls eine untere Dreiecksmatrix und es gilt somit

     $$i < j \Rightarrow b_{i,j} = 0$$

     Daraus folgt

     $$b_{k,j} = 0 \ ,$$

     wodurch wiederum der entsprechende Summand in Gleichung (5.1) zu 0 wird.

   In beiden Fällen sind die betrachteten Summanden von Gleichung (5.1) gleich 0. Also ist dann auch

   $$c_{i,j} = 0 \ ,$$

   was zu zeigen war.

2. Damit C eine Toeplitz-Matrix ist, muß gelten

$$c_{i,j} = c_{i+1,j+1} \ .$$

Mit Hilfe von Gleichung (5.1) ausgedrückt bedeutet dies

$$\sum_{k=1}^{p} a_{i,k} b_{k,j} = \sum_{l=1}^{p} a_{i+1,l} b_{l,j+1} \ . \tag{5.2}$$

Da C eine untere Dreiecksmatrix ist, wie oben bewiesen wurde, müssen nur

$$c_{i,j}$$

betrachtet werden, für die gilt

$$i \geq j \ .$$

Man kann Fallunterscheidungen anhand der Indizes k und l durchführen. Es gibt für jeden Index drei Fälle, also insgesamt sechs:

$k > i$
: Da A nach Voraussetzung eine untere Dreiecksmatrix ist, gilt in diesem Fall

$$a_{i,k} = 0 \ ,$$

und der entsprechende Summand wird zu 0.

$j > k$
: Da B nach Voraussetzung ebenfalls eine untere Dreiecksmatrix ist, gilt in diesem Fall

$$b_{k,j} = 0 \ ,$$

und der entsprechende Summand wird zu 0.

$i \geq k \geq j$
: Nur in diesem Fall ergibt sich auf der linken Seite von Gleichung (5.2) für den jeweiligen Summand ein von 0 verschiedener Wert. Deshalb kann man die linke Seite dieser Gleichung auch schreiben als

$$\sum_{k=j}^{i} a_{i,k} b_{k,j} \ .$$

$l > i + 1$
: In diesem Fall gilt, da A eine obere Dreiecksmatrix ist,

$$a_{i+1,l} = 0 \ .$$

Der entsprechende Summand der Summe in Gleichung (5.2) wird somit zu 0 und muß nicht länger betrachtet werden.

$j + 1 > l$
: In diesem Fall gilt

$$b_{l,j+1} = 0 \ ,$$

da B eine obere Dreiecksmatrix ist und der entsprechende Summand in Gleichung (5.2) muß nicht länger betrachtet werden.

$i + 1 \geq l \geq j + 1$
: Nur in diesem Fall ergibt sich auf der rechten Seite von Gleichung (5.2) ein von 0

verschiedener Wert für den entsprechenden Summanden. Man kann also die rechte Seite dieser Gleichung auch schreiben als

$$\sum_{l=j+1}^{i+1} a_{i+1,l} b_{l,j+1}$$

Nach der Betrachtung dieser sechs Fälle reduziert sich Gleichung (5.2) also, falls man nur die von 0 verschiedenen Summanden betrachtet, auf die Form

$$\sum_{k=j}^{i} a_{i,k} b_{k,j} = \sum_{l=j+1}^{i+1} a_{i+1,l} b_{l,j+1}$$

Anders geschrieben hat diese Gleichung die Form

$$a_{i,j}b_{j,j} + a_{i,j+1}b_{j+1,j} + a_{i,j+2}b_{j+2,j} + \ldots + a_{i,i}b_{i,j} =$$
$$a_{i+1,j+1}b_{j+1,j+1} + a_{i+1,j+2}b_{j+2,j+1} + a_{i+1,j+3}b_{j+3,j+1} + \ldots + a_{i+1,i+1}b_{i+1,j+1}$$

Da A und B Toeplitz-Matrizen sind, haben die beiden Seiten dieser Gleichung den gleichen Wert, was zu beweisen war.

3. Da C wiederum eine Toeplitz-Matrix ist, müssen nur die $c_{i,j}$ mit $j = 1$ neu berechnet werden. Alle anderen Elemente sind entweder gleich Null oder gleich einem $c_{i,1}$. Läßt man zusätzlich alle Multiplikationen mit Null weg, kommt man zur Berechnung von C insgesamt mit

$$\lceil \log(p) \rceil + 1$$

Schritten und

$$\begin{aligned} & \sum_{k=1}^{\min(p,\, m)} k + p * \max(n-p,\, 0) \\ = \quad & \frac{\min(p,\, m) * (\min(p,\, m) + 1)}{2} + p * \max(n-p, 0) \end{aligned}$$

Prozessoren aus. Einschließlich der Multiplikationen mit Null erhält man

$$\lceil \log(p) \rceil + 1$$

Schritte und

$$n * p$$

Prozessoren.

□

5.2 Der Satz von Samuelson

In diesem Unterkapitel wird der theoretische Hintergrund des darzustellenden Algorithmus behandelt.

Zur Beschreibung des Satzes von Samuelson [Sam42] wird folgende Schreibweise eingeführt (A ist eine $n \times n$-Matrix):

- Den Vektor S_i erhält man aus dem i-ten Spaltenvektor von A durch Entfernen der ersten i Elemente. Er hat also folgendes Aussehen:

$$\begin{bmatrix} a_{i+1,i} \\ a_{i+2,i} \\ \vdots \\ a_{n,i} \end{bmatrix}$$

- Den Vektor R_i erhält man aus dem i-ten Zeilenvektor von A durch Entfernen der ersten i Elemente. Er hat also folgendes Aussehen:

$$[a_{i,i+1}, a_{i,i+2}, \ldots, a_{i,n}]$$

- Die Matrix M_i erhält man aus der Matrix A durch Entfernen der ersten i Zeilen und Spalten. Sie hat also folgendes Aussehen:

$$\begin{bmatrix} a_{i+1,i+1} & a_{i+1,i+2} & \cdots & a_{i+1,n} \\ a_{i+2,i+1} & a_{i+2,i+2} & \cdots & a_{i+2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,i+1} & a_{n,i+2} & \cdots & a_{n,n} \end{bmatrix}$$

- Statt S_1, R_1 und M_1 wird auch S, R und M geschrieben.

Die Matrix A läßt sich also auch in den Formen

$$\begin{bmatrix} a_{11} & R \\ S & M \end{bmatrix}$$

oder

$$\begin{bmatrix} a_{11} & R_1 & \rightarrow & & \\ S_1 & a_{22} & R_2 & \rightarrow & \\ \downarrow & S_2 & a_{33} & R_3 & \rightarrow \\ & \downarrow & S_3 & \ddots & \ddots \\ & & \downarrow & \ddots & \end{bmatrix}$$

darstellen.

Im folgenden Lemma wird das charakteristische Polynom einer Matrix mit Hilfe der oben definierten R, S und M ausgedrückt:

Lemma 5.2.1 *Sei $p(\lambda)$ das charakteristische Polynom der $n \times n$-Matrix A. Dann gilt:*

$$p(\lambda) = (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) - R * \operatorname{adj}(M - \lambda * E_{n-1}) * S$$

Beweis Es gilt

$$p(\lambda) = \det(A - \lambda * E_n)$$

Durch Entwicklung nach der ersten Zeile erhält man:

$$p(\lambda) = (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) + \sum_{j=2}^{n} (-1)^{1+j} a_{1,j} \underline{\det((A - \lambda * E_n)_{(1|j)})}$$

Nun werden die in der obigen Gleichung unterstrichenen Determinanten jeweils nach der ersten Spalte entwickelt:

$$\begin{aligned} p(\lambda) = \quad & (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) + \\ & \sum_{j=2}^{n} \underbrace{(-1)^{1+j} a_{1,j}}_{(*1)} \sum_{k=2}^{n} (-1)^{1+(k-1)} a_{k,1} \det(\underbrace{(A - \lambda * E_n)_{(1,k|1,j)}}_{(*2)}) \end{aligned}$$

Wenn man in dieser Gleichung (*1) mit der inneren Summe multipliziert und (*2) mit Hilfe von M ausdrückt erhält man:

$$p(\lambda) = (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) + \sum_{j=2}^{n}\sum_{k=2}^{n}(-1)^{1+j+k}\underbrace{a_{1,j}a_{k,1}}_{(*)}\det((M - \lambda * E_{n-1})_{(k-1|j-1)})$$

Hier läßt sich (*) mit Hilfe von R und S formulieren:

$$p(\lambda) = (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) + \sum_{j=2}^{n}\sum_{k=2}^{n}(-1)^{1+j+k}r_j s_k \det((M - \lambda * E_{n-1})_{(k-1|j-1)})$$

Dies wiederum ist in Matrizenschreibweise und mit Hilfe der Adjunkten einer Matrix ausgedrückt nichts anderes als

$$p(\lambda) = (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) - R * \mathrm{adj}\,(M - \lambda * E_{n-1}) * S \ ,$$

was zu beweisen war. □

Vor Lemma 5.2.3 müssen wir hier zunächst einen wichtigen Satz behandeln ([MM64] S. 50 f):

Satz 5.2.2 (Cayley und Hamilton) *Sei $p(\lambda)$ das charakteristische Polynom von A. Dann gilt:*

$$p(A) = 0_{n,n}$$

Beweis Aus Satz 3.6.3 folgt

$$(A - \lambda E_n)\mathrm{adj}\,(A - \lambda E_n) = p(A)E_n \tag{5.3}$$

Da die Elemente von

$$\mathrm{adj}\,(A - \lambda E_n)$$

aus Unterdeterminanten von A gewonnen werden, bestehen diese Elemente aus Polynomen über λ vom maximalen Grad

$$n - 1$$

Also gilt für geeignete $n \times n$-Matrizen

$$B_j, 1 \leq j \leq n - 1$$

die folgende Beziehung:

$$\mathrm{adj}\,(A - \lambda E_n) = B_{n-1}\lambda^{n-1} + \ldots + B_1\lambda + B_0 \tag{5.4}$$

Außerdem kann man $p(A)$ schreiben als

$$p(A) = c_n\lambda^n + \ldots + c_1\lambda + c_0 \tag{5.5}$$

Drückt man (5.3) mit Hilfe von (5.4) und (5.5) aus, erhält man

$$(A - \lambda E_n)(B_{n-1}\lambda^{n-1} + \ldots + B_1\lambda + B_0) = (c_n\lambda^n + \ldots + c_1\lambda + c_0)E_n$$

Multipliziert man die Terme auf beiden Seiten aus und vergleicht die Koeffizienten miteinander, erhält man folgende Gleichungen:

$$\begin{array}{rcrcr}
 & - & B_{n-1} & = & c_nE_n \\
AB_{n-1} & - & B_{n-2} & = & c_{n-1}E_n \\
AB_{n-2} & - & B_{n-3} & = & c_{n-2}E_n \\
 & & & \vdots & \\
AB_1 & - & B_0 & = & c_1E_n \\
AB_0 & & & = & c_0E_n
\end{array}$$

Multipliziert man beide Seiten der ersten dieser Gleichungen mit A^n, beide Seiten der zweiten mit A^{n-1}, allgemein beide Seiten der j-ten mit A^{n-j+1}, und addiert sie, erhält man

$$0_{n,n} = c_n\lambda^n + c_{n-1}\lambda^{n-1} + \ldots + c_1A = p(A)$$

□

Die Adjunkte in Lemma 5.2.1 läßt sich mit Hilfe der Koeffizienten des charakteristischen Polynoms $q(\lambda)$ von M ausdrücken. In Koeffizientendarstellung besitzt $q(\lambda)$ die Form:

$$q(\lambda) = q_{n-1}\lambda^{n-1} + q_{n-2}\lambda^{n-2} + \ldots + q_1\lambda + q_0$$

Es gilt folgende Aussage:

Lemma 5.2.3

$$\text{adj}\,(M - \lambda * E_{n-1}) = -\sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k-1} q_l \tag{5.6}$$

Beweis Multipliziert man beide Seiten von (5.6) mit

$$M - \lambda * E_{n-1} \ ,$$

erhält man auf der linken Seite

$$\text{adj}\,(M - \lambda * E_{n-1}) * (M - \lambda * E_{n-1}) \ .$$

Dies ist nach Satz 3.6.3 gleich

$$\begin{aligned} & E_{n-1} * \det(M - \lambda * E_{n-1}) \\ = \quad & q(\lambda) * E_{n-1} \ . \end{aligned}$$

Auf der rechten Seite von Gleichung (5.6) erhält man

$$-(\underbrace{M}_{(*1)} \underbrace{-\lambda * E_{n-1})}_{(*2)} \sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k-1} q_l$$

Bei der Multiplikation erhält man für (*1) und (*2) im obigen je eine Doppelsumme:

$$-\sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k} q_l + \sum_{k=0}^{n-2} \lambda^{k+1} \sum_{l=k+1}^{n-1} M^{l-k-1} q_l$$

Durch Umordnen der Indizes der zweiten Doppelsumme erhält man

$$-\sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k} q_l + \sum_{k=1}^{n-1} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k} q_l \tag{5.7}$$

Nach Satz 5.2.2 gilt

$$\sum_{l=0}^{n-1} M^l q_l = 0 \tag{5.8}$$

Somit kann man die linke Seite von Gleichung (5.8) zur zweiten Doppelsumme von Term (5.7) addieren und erhält

$$-\sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k} q_l + \sum_{k=0}^{n-1} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k} q_l$$

Wenn man nun die Vorzeichen der beiden Doppelsummen sowie die benutzten Indizes betrachtet, erkennt man, daß sich der Gesamtterm vereinfacht darstellen läßt, da große Teile zusammengenommen 0 ergeben. Die Teile, die sich nicht auf diese Weise gegenseitig aufheben, lassen sich schreiben als

$$\sum_{k=0}^{n-1} \lambda^k E_{n-1} q_k \ ,$$

was gleichbedeutend ist mit

$$q(\lambda) * E_{n-1} \ .$$

Also stimmen die beiden Seiten von Gleichung (5.6) überein. □

Die beiden Lemmata 5.2.1 und 5.2.3 führen zu folgendem Satz [Sam42]:

Satz 5.2.4 (Samuelson)

$$p(\lambda) = (a_{1,1} - \lambda) * \det(M - \lambda * E_{n-1}) + R * \left(\sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k-1} q_l \right) * S \tag{5.9}$$

Beweis Lemma 5.2.3 angewendet auf Lemma 5.2.1 ergibt die Behauptung. □

5.3 Determinantenberechnung mit Hilfe des Satzes von Samuelson

Um Satz 5.2.4 zur Determinantenberechnung zu benutzen [Ber84], sind weitere Überlegungen notwendig, die in diesem Unterkapitel behandelt werden.

Betrachtet man die Methodik des entstehenden Algorithmus, erkennt man Ähnlichkeit zu C-Alg. . Auch dort wird ein schon länger bekannter Satz mit Hilfe von zusätzlichen Überlegungen für eine parallelen Algorithmus verwendet.

Zu beachten ist, daß in diesem Unterkapitel für die Multiplikation zweier $n \times n$-Matrizen $n^{2+\gamma}$ Prozessoren in Rechnung gestellt werden (vgl. S. 15).

Benutzt man die Koeffizientendarstellung für die charakteristischen Polynome von A und M, läßt sich Gleichung (5.9) umformulieren in

$$\sum_{i=0}^{n} p_i \lambda^i = (a_{1,1} - \lambda) * \sum_{i=0}^{n-1} q_i \lambda^i + R * \left(\sum_{k=0}^{n-2} \lambda^k \sum_{l=k+1}^{n-1} M^{l-k-1} q_l \right) * S \ .$$

Vergleicht man die Koeffizienten der λ^i auf beiden Seiten der Gleichung und definiert

$$q_{-1} := 0 \ ,$$

erhält man

$$p_n = -q_{n-1} \tag{5.10}$$

$$p_{n-1} = a_{1,1} q_{n-1} - q_{n-2} \tag{5.11}$$

$$\forall i = n-2 \ldots 0 : \ p_i = a_{1,1} q_i - q_{i-1} + \sum_{j=i+1}^{n-1} R M^{j-i-1} S q_i \tag{5.12}$$

Die Beziehungen zwischen den Koeffizienten, die diese Gleichungen beschreiben, kann man auch durch eine Matrizengleichung ausdrücken. Dazu wird Matrix C_t definiert als untere Dreiecks-Toeplitz-Matrix der Größe $(n - t + 2) \times (n - t + 1)$. Ihre Elemente werden definiert durch

$$(c_t)_{i,j} := \begin{cases} -1 & : \ i = 1 \\ a_{t,t} & : \ i = 2 \\ R_t M_t^{i-3} S_t & : \ i > 2 \end{cases}$$

Die Matrix hat also das folgende Aussehen:

$$\begin{bmatrix} -1 & 0 & \cdots \\ a_{t,t} & -1 & \ddots \\ R_t S_t & a_{t,t} & \ddots \\ R_t M_t S_t & R_t S_t & \ddots \\ \vdots & \ddots & \ddots \\ R_t M_t^{n-t-1} S_t & & \end{bmatrix}$$

Insbesondere hat C_n die Form

$$\begin{bmatrix} -1 \\ a_{n,n} \end{bmatrix}$$

Mit Hilfe dieser Definition erhält man aus den Gleichungen (5.10), (5.11) und (5.12) die folgende Matrizengleichung:

$$\begin{bmatrix} p_n \\ p_{n-1} \\ \vdots \\ p_0 \end{bmatrix} = C_1 \begin{bmatrix} q_{n-1} \\ q_{n-2} \\ \vdots \\ q_0 \end{bmatrix} \tag{5.13}$$

Auf die gleiche Weise, wie man Satz 5.2.4 auf die Matrizen A und M anwendet, kann man diesen Satz auch auf die Matrizen M und M_2, M_2 und M_3, etc. anwenden und erhält so Matrizengleichungen, die in ihrer Form der Gleichung (5.13) entsprechen.

Wendet man diese Matrizengleichungen aufeinander an, erhält man:

$$\begin{bmatrix} p_n \\ p_{n-1} \\ \vdots \\ p_0 \end{bmatrix} = \prod_{i=1}^{n} C_i \tag{5.14}$$

Um die Koeffizienten des charakteristischen Polynoms von A auf die geschilderte Weise zu berechnen, muß man also die Matrizen C_i berechnen und dann miteinander multiplizieren. Nach 2.4.4 ist damit auch die Determinante der Matrix A berechnet.

Für jede $(n-i+2) \times (n-i+1)$-Matrix C_i bei ist der $(n-i)$-elementige Vektor

$$T_t := [R_i S_i,\, R_i M_i S_i,\, R_i M_i^2 S_i,\, \ldots,\, R_i M_i^m S_i],\; m := n-i-1 \tag{5.15}$$

zu berechnen. Da also T_n keine Elemente enthält, ist die Berechnung der Vektoren T_1 bis T_{n-1} erforderlich.

Man kann jeden Exponenten k eines Elementes

$$R_i * M_i^k * S_i$$

von T_i in der Form

$$k = u + v * \lceil \sqrt{m} \rceil$$

mit

$$0 \le u < \lceil \sqrt{m} \rceil$$
$$0 \le v \le \lfloor \sqrt{m} \rfloor$$

eindeutig darstellen. Man könnte statt $\sqrt{m}$ auch einen anderen Wert zwischen 0 und m nehmen. Jedoch führt die Wahl von $\sqrt{m}$ dazu, daß sich die Größe der Mengen aller u und v um höchstens 1 unterscheidet.

Um T_i effizient zu erhalten, kann man zunächst die den Mengen der u und v entsprechenden Vektoren

$$U_i := \left[R_i,\, R_i M_i,\, R_i M_i^2,\, \ldots,\, R_i M_i^{\lceil \sqrt{m} \rceil - 1} \right]$$

und

$$V_i := \left[S_i,\, M_i^{\lceil \sqrt{m} \rceil} S_i,\, M_i^{2\lceil \sqrt{m} \rceil} S_i,\, \ldots,\, M_i^{\lfloor \sqrt{m} \rfloor \lceil \sqrt{m} \rceil} S_i \right]$$

berechnen und danach jedes Element des einen Vektors mit jedem Element des anderen multiplizieren.

Genau genommen werden auf diese Weise einige Werte zuviel berechnet, wie sich bei noch exakterer Analyse des Algorithmus zeigt. Es sind jedoch vernachlässigbar wenige. Die Berechnung

dieser Werte kann durch vernachlässigbar geringen zusätzlichen Aufwand verhindert werden. Um die Darstellung des Algorithmus nicht unnötig unübersichtlich zu machen, werden diese Werte nicht weiter beachtet.

Vor Beginn der Rechnung wird ein

$$\epsilon \in \mathbb{Q}\,,\ 0 < \epsilon \leq 0.5$$

festgelegt[3]. O. B. d. A. sei ϵ so gewählt, daß gilt[4]

$$\exists\, p \in \mathbb{N} :\ p * \epsilon = 0.5\ .$$

Die Wahl von ϵ beeinflußt das Verhältnis zwischen der Anzahl der Schritte und der Anzahl der dabei beschäftigten Prozessoren. Dies wird weiter unten durch die Analyse deutlich.

Für den Rest dieses Unterkapitels gelte die Vereinbarung, daß mit

$$a^b$$

der Wert

$$\lceil a^b \rceil$$

gemeint ist.

Mit Hinweis auf die Bemerkungen im Anschluß an die Behandlung der Matrizenmultiplikation in Satz 1.5.3 wird im folgenden für die Multiplikation zweier $n \times n$-Matrizen ein Aufwand von

$$\gamma_S(\lceil \log(n) \rceil + 1)$$

Schritten und

$$\gamma_P n^{2+\gamma}$$

Prozessoren in Rechnung gestellt.

Im folgenden ist mit T, U und V jeweils T_i, U_i bzw. V_i gemeint, wobei $1 \leq i < n$ gilt.

Um den Vektor U zu berechnen, benutzen wir folgenden iterativen Algorithmus[5]:

- Der Vektor Z_α wird wie folgt definiert:

  $$Z_\alpha := \left[R_i,\ R_i M_i,\ R_i M_i^2,\ \ldots,\ R_i M_i^{m^\alpha - 1}\right]$$

  Das bedeutet, es gilt

  $$Z_0 = [R_i]$$

  Das Ziel ist es, $Z_{0.5} = U$ zu berechnen. Der Vektor Z_0 ist bekannt, da R_i Teil der Eingabe ist.

- Wenn Z_α bekannt ist, im ersten Schleifendurchlauf also Z_0, dann wird daraus $Z_{\alpha+\epsilon}$ wie folgt berechnet:

  – Berechne

  $$Y_{\alpha+\epsilon} := \left[M_i^{m^\alpha},\ M_i^{2m^\alpha},\ M_i^{3m^\alpha},\ \ldots,\ M_i^{m^\epsilon m^\alpha}\right]$$

  Nach 1.4.1 in Verbindung mit 1.5.3 erhält man für die Anzahl der Schritte

  $$\begin{aligned} & \gamma_S \lceil \log(m^\epsilon) \rceil (\lceil \log(m) \rceil + 1) \\ \leq\ & \gamma_S \lceil (\epsilon \lceil \log(m) \rceil + 1)(\lceil \log(m) \rceil + 1) \rceil \\ =\ & \gamma_S \left\lceil \epsilon \lceil \log(m) \rceil^2 + (\epsilon + 1) \lceil \log(m) \rceil + 1 \right\rceil \end{aligned}$$

  und für die Anzahl der Prozessoren

  $$\gamma_P \left\lfloor 0.75 m^\epsilon \right\rfloor m^{2+\gamma} < \gamma_P m^{2+\gamma+\epsilon}\ .$$

  Die dafür nötige Startmatrix M_i^α erhält man als Nebenergebnis aus der Berechnung von Y_α. Die Startmatrix für die Berechnung von Y_ϵ ist M_i.

[3] ein Wert $\epsilon > 0.5$ ist möglich, jedoch von seinen Auswirkungen her uninteressant
[4] erfüllt ϵ diese Bedingung nicht, wird dadurch die Analyse des Algorithmus unnötig unübersichtlich
[5] zur Vereinfachung der Darstellung werden keine ganzzahligen Werte zur Indizierung benutzt

– Der Vektor $X_{\alpha+\epsilon}$ wird folgendermaßen definiert:

$$X_{\alpha+\epsilon} := \left[R_i M_i^{m^\alpha},\, R_i M_i^{m^\alpha+1},\, R_i M_i^{m^\alpha+2},\, \ldots,\, R_i M_i^{m^{\alpha+\epsilon}-1}\right]$$

Es wird nun $X_{\alpha+\epsilon}$ aus Z_α und $Y_{\alpha+\epsilon}$ berechnet.
Der Vektor Z_α besitzt m^α Elemente, die ihrerseits Vektoren der Länge m darstellen. Sie werden mit

$$z_{\alpha,1},\, z_{\alpha,2},\, \ldots,\, z_{\alpha,m^\alpha}$$

bezeichnet. Der Vektor $Y_{\alpha+\epsilon}$ besitzt m^ϵ Elemente. Diese Elemente sind $m \times m$-Matrizen und werden mit

$$y_{\alpha+\epsilon,1},\, y_{\alpha+\epsilon,2},\, \ldots,\, y_{\alpha+\epsilon,m^\epsilon}$$

bezeichnet.

Der Vektor $X_{\alpha+\epsilon}$ wird wie folgt berechnet:
Parallel für $i := 1$ bis $m^\epsilon - 1$:
Parallel für $j := 1$ bis m^α:

$$x_{\alpha+\epsilon,(i-1)*m^\epsilon+j} := z_{\alpha,j} * y_{\alpha+\epsilon,i}$$

Bei dieser Berechnung fällt auf, daß $y_{\alpha+\epsilon,m^\epsilon}$ nicht verwendet wird. Diese Matrix bildet die Startmatrix für die Berechnung von $Y_{\alpha+2\epsilon}$ im nächsten Schleifendurchlauf (s. o.).
Für die Analyse des Aufwandes der Berechnung von $X_{\alpha+\epsilon}$ wird Z_α als Matrix betrachtet. Die $z_{\alpha,j}$ bilden die Zeilenvektoren dieser Matrix. So gesehen sind also m^ϵ Matrizenmultiplikationen durchzuführen. Dies kann von

$$\gamma_P m^{2+\gamma+\epsilon}$$

Prozessoren in

$$\gamma_S \lceil \log(m) \rceil + 1$$

Schritten durchgeführt werden.

– Die ersten m^α Elemente des in diesem Schleifendurchlauf gesuchten Vektors $Z_{\alpha+\epsilon}$ werden durch die Elemente des Vektors Z_α gebildet und alle weiteren durch die Elemente des soeben berechneten Vektors $X_{\alpha+\epsilon}$.

Betrachtet man den Aufwand zur Berechnung von $Y_{\alpha+\epsilon}$ und $X_{\alpha+\epsilon}$ zusammen, erhält man für die Berechnung von $Z_{\alpha+\epsilon}$ aus Z_α

$$\gamma_S \left[\epsilon \lceil \log(m) \rceil^2 + (\epsilon+2) \lceil \log(m) \rceil + 2\right]$$

Schritte und

$$\gamma_P m^{2+\gamma+\epsilon}$$

Prozessoren.

- Insgesamt erfolgen

$$\frac{1}{2\epsilon}$$

Schleifendurchläufe. Der Aufwand zur Berechnung von U beträgt deshalb

$$\begin{aligned} & \frac{0.5\gamma_S}{\epsilon} \left[\epsilon \lceil \log(m) \rceil^2 + (\epsilon+2) \lceil \log(m) \rceil + 2\right] \\ \leq\ & 0.5\gamma_S \left\lceil \lceil \log(m) \rceil^2 + \left(1 + \frac{2}{\epsilon}\right) \lceil \log(m) \rceil + \frac{2}{\epsilon} \right\rceil \end{aligned}$$

Schritte und

$$\gamma_P m^{2+\gamma+\epsilon}$$

Prozessoren.

Im Anschluß an die Berechnung von U erfolgt die Berechnung von V auf die gleiche Weise. Der einzige wesentliche Unterschied zwischen den beiden Berechnungsvorgängen ist die andere Startmatrix zur Berechnung des Y_ϵ entsprechenden Vektors. Hier wird $M_i^{m^{0.5}}$ statt M_i benötigt. Man erhält $M_i^{m^{0.5}}$ aus M_i mit Hilfe der Binärbaummethode nach 1.5.1. Dies kann in

$$\begin{aligned} & \gamma_S \left\lceil \log(m^{0.5}) \right\rceil (\lceil \log(m) \rceil + 1) \\ \leq\ & \gamma_S \lceil 0.5 \lceil \log(m) \rceil (\lceil \log(m) \rceil + 1) \rceil \\ =\ & \gamma_S \left\lceil 0.5(\left\lceil \log^2(m) \right\rceil + \lceil \log(m) \rceil + 1) \right\rceil \end{aligned}$$

Schritten von

$$\begin{aligned} & \gamma_P \left\lceil 0.5 m^{0.5} m^{2+\gamma} \right\rceil \\ \leq\ & \gamma_P \left\lceil 0.5 m^{2.5+\gamma} \right\rceil \end{aligned}$$

Prozessoren geleistet werden. Ist die Startmatrix berechnet, ist der weitere Aufwand zur Berechnung von V gleich dem Aufwand zur Berechnung von U. Also kann V insgesamt in[6]

$$\begin{aligned} & \gamma_S \left\lceil 0.5(\lceil \log(m) \rceil^2 + \lceil \log(m) \rceil + 1) + 0.5 \left(\lceil \log(m) \rceil^2 + \left(1 + \frac{2}{\epsilon}\right) \lceil \log(m) \rceil + \frac{2}{\epsilon} \right) \right\rceil \\ =\ & \gamma_S \left\lceil \lceil \log(m) \rceil^2 + \left(1 + \frac{1}{\epsilon}\right) \lceil \log(m) \rceil + \frac{1}{\epsilon} + 0.5 \right\rceil \end{aligned}$$

Schritten erledigt werden. Die Anzahl der Prozessoren beträgt

$$\max \left(\underbrace{\gamma_P m^{2+\gamma+\epsilon}}_{\text{Term 1}}, \underbrace{\gamma_P \left\lceil 0.5 m^{2.5+\gamma} \right\rceil}_{\text{Term 2}} \right) .$$

Da mit steigendem m Term 2 stärker wächst als Term 1, wird die Analyse mit Term 2 für die Anzahl der Prozessoren fortgesetzt.

Parallel zur Berechnung von U wird zuerst $M_i^{m^{0.5}}$ und mit Hilfe dieser Matrix dann V berechnet. Der Aufwand dafür beträgt insgesamt

$$\gamma_S \left\lceil \lceil \log(m) \rceil^2 + \left(1 + \frac{1}{\epsilon}\right) \lceil \log(m) \rceil + \frac{1}{\epsilon} + 0.5 \right\rceil$$

Schritte und

$$\gamma_P \left(m^{2+\gamma+\epsilon} + \left\lceil 0.5 m^{2.5+\gamma} \right\rceil \right)$$

Prozessoren.

Um den nach (5.15) gesuchten Vektor T zu erhalten, müssen noch die Elemente der Vektoren U und V, die ja ihrerseits wiederum Vektoren darstellen, miteinander multipliziert werden. Die Vektoren U und V besitzen eine Länge von $m^{0.5}$. Die Multiplikation zweier Elemente dieser Vektoren können analog zur Matrizenmultiplikation in 1.5.3 in

$$\lceil \log(m) \rceil + 1$$

Schritten von

$$m$$

Prozessoren erledigt werden. Insgesamt sind

$$m^{0.5} * m^{0.5} = m$$

solcher Multiplikationen durchzuführen. Die Berechnung von T aus U und V kann also in

$$\lceil \log(m) \rceil + 1$$

[6] Da die Terme, die die Anzahl der Schritte und Prozessoren beschreiben, bereits nach oben abgeschätzt sind, wird bei der Zusammenfassung von Termen, die durch Gaußklammern eingefaßt sind, auf eine weitere Abschätzung verzichtet.

Schritten von

$$m^2$$

Prozessoren durchgeführt werden.

Betrachtet man den Gesamtaufwand zur Berechnung von U, V und T, kommt man auf

$$\gamma_S \left\lceil \lceil \log(m) \rceil^2 + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon}\right) \lceil \log(m) \rceil + \frac{1}{\epsilon} + \frac{1}{\gamma_S} + 0.5 \right\rceil$$

Schritte und

$$\gamma_P \left(m^{2+\gamma+\epsilon} + \left\lceil 0.5 m^{2.5+\gamma} \right\rceil\right) \leq \gamma_P \left(m^{2+\gamma+\epsilon} + 0.5 m^{2.5+\gamma} + 1\right) \tag{5.16}$$

Prozessoren.

Nach der obigen Analyse der Berechnung einer der Vektoren kann die parallele Berechnung aller Vektoren T_1 bis T_{n-1} in

$$\gamma_S \left\lceil \lceil \log(n-2) \rceil^2 + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon}\right) \lceil \log(n-2) \rceil + \frac{1}{\epsilon} + \frac{1}{\gamma_S} + 0.5 \right\rceil \tag{5.17}$$

Schritten durchgeführt werden. Da die Berechnung eines Vektors T_i für $i > 1$ bei gleichem ϵ schneller ist als die Berechnung von T_1, ist es möglich, dadurch Prozessoren zu sparen, daß man ϵ für jeden Vektor T_i verschieden wählt, und zwar als Funktion von

- der Größe n der Eingabematrix A,
- der Länge $m+1$ des jeweiligen Vektors T_i und
- dem ϵ, daß zur Berechnung des Vektors T_1 verwendet wird.

Das separat für jeden Vektor T_i zu wählende ϵ wird mit[7] ϵ_m bezeichnet.

Da die Vektoren T für $m \leq n-2$ berechnet werden sollen, muß für jedes ϵ_m mit $\epsilon_m \neq \epsilon$ die Bedingung $m \leq n-3$ erfüllt sein. Da gleichzeitig $m \geq 1$ erfüllt sein muß, wird für die folgenden Analysen $n \geq 4$ angenommen. Andernfalls ist die Anwendung der Idee zur Wahl der ϵ_m nicht sinnvoll.

Wie ϵ_m zu wählen ist, ergibt sich aus Term (5.17). Es muß gelten:

$$\lceil \log(m) \rceil^2 + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon_m}\right) \lceil \log(m) \rceil + \frac{1}{\epsilon_m} \leq \lceil \log(n-2) \rceil^2 + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon}\right) \lceil \log(n-2) \rceil + \frac{1}{\epsilon}$$

Löst man diese Ungleichung nach ϵ_m auf erhält man:

$$\epsilon_m \geq \frac{\lceil \log(m) \rceil + 1}{\lceil \log(n-2) \rceil^2 + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon}\right) \lceil \log(n-2) \rceil + \frac{1}{\epsilon} - \lceil \log(m) \rceil^2 - \left(1 + \frac{1}{\gamma_S}\right) \lceil \log(m) \rceil} \tag{5.18}$$

Die Gaußklammern in dieser Ungleichung führen zu einigen wichtigen Konsequenzen für n, m und ϵ_m. Es gelte dazu

$$\begin{gathered} k \in \mathbb{N} \\ 1 \leq 2^k < m_1 \leq 2^{k+1} \leq n-2 \\ 1 \leq 2^k < m_2 \leq 2^{k+1} \leq n-2 \quad . \end{gathered}$$

Aus (5.18) folgt dann

$$\epsilon_{m_1} = \epsilon_{m_2} \ .$$

Das bedeutet insbesondere, daß es u. U. einige m mit $m \leq n-3$ gibt, für die gilt

$$\epsilon_m = \epsilon \ .$$

[7] Es wurde bereits definiert: $m := n - i - 1$.

Der ungünstigste Fall tritt für

$$n - 2 = 2^{k+1}$$

ein. Bei diesem Fall ist nur für

$$m \leq \frac{n-2}{2}$$

die Bedingung

$$\epsilon_m < \epsilon$$

erfüllt.

Für die weitere Analyse ist es an dieser Stelle sinnvoll, die Gaußklammern im Term auf der rechten Seite von (5.18) zu beseitigen. Dazu wird der Term nach oben abgeschätzt.

Terme, die in Gaußklammern eingefaßt sind, kann man mit Hilfe der Beziehung

$$a \leq \lceil a \rceil \leq a + 1 \tag{5.19}$$

abschätzen. Es gilt jedoch

$$\begin{aligned} & \lceil \log(x) \rceil \\ \leq \; & \log(x) + 1 \\ = \; & \log(2x) \ . \end{aligned}$$

Zu beachten sind hier die Konsequenzen, wenn die Abschätzung mit Hilfe von (5.19) vorgenommen werden. Falls $n - 2$ eine Zweierpotenz ist, ergibt die auf diese Weise abgeschätzte Ungleichung (5.18) nur für

$$m \leq \frac{n-2}{4}$$

Werte für ϵ_m, so daß

$$\epsilon_m < \epsilon \ .$$

Eine Verbesserung dieser Abschätzung der Gaußklammerfunktion ist wünschenswert.

Dazu wird definiert, daß eine Funktion $f(x)$ *konkav auf einem Intervall* I ist, falls für ihre zweite Ableitung $f''(x)$ gilt:

$$\forall x \in I : \ f''(x) \leq 0 \ .$$

Soll die Gaußklammer einer konkaven Funktion $h(x)$ gebildet und die Fläche unter der resultierenden Kurve berechnet werden, so läßt sich der Ausdruck auch durch

$$\int \lceil h(x)dx \rceil \leq \int (h(x) + 0.5)dx$$

nach oben abschätzen. In Abbildung 5.1 ist dies verdeutlicht. Dort ist Fläche 1 größer als Fläche 2. Somit kommen wir auf

$$\begin{aligned} & \int \lceil \log(x) \rceil \, dx \\ \leq \; & \int (\log(x) + 0.5)dx \\ = \; & \int \log(\sqrt{2}x)dx \ . \end{aligned} \tag{5.20}$$

Ist der abzuschätzende Gaußklammerterm Teil einer Funktion

$$h_2(\lceil h(x) \rceil, x) \ ,$$

so läßt sich die beschriebene Abschätzung durchführen, falls h_2 monoton ist. Diese Bedingung ist bei den folgenden Anwendungen erfüllt.

Für die folgenden Untersuchungen wird die Funktion

$$g : \mathbb{Q} \to \mathbb{Q}$$

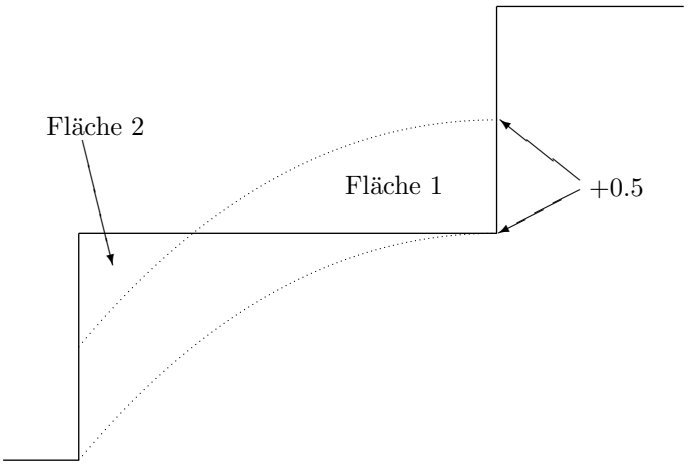

Abbildung 5.1: Integration der Gaußklammer einer konkaven Funktion

eingeführt. Sie sei *im Riemann'schen Sinne integrierbar* [BS87] (S. 289) auf dem Intervall

$$(-\infty, \infty)$$

und wird als Platzhalter für die Funktion verwendet, die schließlich zur Abschätzung der Gaußklammerfunktion benutzt wird. Je genauer sie die Gaußklammerfunktion abschätzt, umso besser werden die Analyseergebnisse.

Mit Hilfe von (5.18) wird folgende Funktion zur Berechnung von ϵ_m bei gegebenen n und ϵ definiert:

$$f(m) := \frac{g(\log(m)) + 1}{g^2(\log(n-2)) + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon}\right) g(\log(n-2)) + \frac{1}{\epsilon} - g^2(\log(m)) - \left(1 + \frac{1}{\gamma_S}\right) g(\log(m))}$$

Mit Hilfe dieser Funktion kommt man anhand von (5.16) für die Anzahl der Prozessoren zur Berechnung aller Vektoren T auf:

$$\begin{aligned}
& 4 + (n-2)^{2+\gamma+\epsilon} + \frac{(n-2)^{2.5+\gamma}}{2} + \sum_{m=2}^{n-3} \left(m^{2+\gamma+f(m)} + \frac{m^{2.5+\gamma}}{2} + 1 \right) \\
\leq\ & \overbrace{4 + (n-2)^{2+\gamma+\epsilon} + \frac{(n-2)^{2.5+\gamma}}{2}}^{t_1:=} + \int_2^{n-2} \left(m^{2+\gamma+f(m)} + \frac{m^{2.5+\gamma}}{2} + 1 \right) dm \\
=\ & \overbrace{t_1 + m|_2^{n-2} + \left(\frac{1}{7+2\gamma} m^{3.5+\gamma} \right)\Big|_2^{n-2}}^{t_2:=} + \int_2^{n-2} m^{2+\gamma+f(m)} dm \qquad (5.21) \\
=\ & t_2 + \int_2^{n-2} \overbrace{\mathrm{e}^{(2+\gamma+f(m))\ln(m)}}^{t_3:=} dm \qquad (5.22)
\end{aligned}$$

Für die Integration von t_3 sind 3 Methoden von Bedeutung:

Numerische Berechnung
: Diese Methode ist für gegebene n und ϵ eine gangbare Möglichkeit [EMR88] (Kap. 9). Für eine allgemeine Analyse ist sie jedoch nicht geeignet.

Analytische Berechnung

Bezeichne t_4 die erste Ableitung von

$$(2 + \gamma + f(m)) \ln(m) \ .$$

Bezeichne t_5' die erste Ableitung des zu bestimmenden Terms t_5. Da t_3 eine Exponentialfunktion ist, muß die gesuchte Stammfunktion, die Form $t_3 t_5$ besitzen. Die Ableitung dieses Ausdrucks ergibt

$$(t_3 t_5)' = t_3 t_4 t_5 + t_3 t_5' = t_3 \underline{(t_4 t_5 + t_5')} \ .$$

Da der unterstrichene Teil den Wert 1 besitzen muß, ist die Differentialgleichung

$$t_5' = 1 - t_4 t_5$$

zu lösen. Dies ist eine *explizite gewöhnliche Differentialgleichung erster Ordnung* [BS87] (S. 414 ff.).

Man gelangt zu der Vermutung, daß zu t_3 keine Stammfunktion existiert, da sowohl die Integration der Differentialgleichung, als auch die Integration von t_3 mit Hilfe der Eigenschaften unbestimmter Integrale [BS87] (S. 295) nicht zu einem Ergebnis zu führen scheinen. Unterstützt wird diese Vermutung durch die Tatsache, daß für

$$\mathrm{e}^{m^2}$$

keine Stammfunktion existiert.

Abschätzung der Stammfunktion nach oben

Diese Methode ist am besten geeignet und wird im folgenden benutzt. Dazu wird der oben erwähnte Term t_5 so bestimmt, daß gilt

$$t_4 t_5 + t_5' \geq 1 \ . \tag{5.23}$$

Der Nenner von $f(m)$ wird mit t_6 bezeichnet, der Zähler mit t_7. Die Ableitung von t_3 ergibt:

$$\begin{aligned} t_3' = t_3 & \left(\frac{2}{m} + \frac{\gamma}{m} \right) \\ + & \frac{\left(g'(\log(m)) \frac{\log(m)}{m} + \frac{g(\log(m))}{m} + \frac{1}{m} \right) t_6}{t_6^2} \\ + & \frac{t_7 \ln(m) \left(\frac{2 g(\log(m)) g'(\log(m))}{m \ln(2)} + \frac{\left(1 + \frac{1}{\gamma_S}\right) g'(\log(m))}{m \ln(2)} \right)}{t_6^2} \end{aligned}$$

Wie bereits beschrieben wurde, ist es an dieser Stelle möglich

$$g(x) := x + 0.5 \tag{5.24}$$

als obere Abschätzung der Gaußklammerfunktion zu verwenden. Es ist zu beachten, daß diese Abschätzung nicht für die Berechnung von ϵ_m bei der Anwendung des Algorithmus in einer konkreten Situation benutzt werden darf. Die Benutzung von (5.24) an dieser Stelle ist nur zulässig, weil die Abschätzung des gesamten Integrals in (5.22) das Ziel ist.

Wie ebenfalls bereits beschrieben wurde, ist darauf zu achten, daß die auftretenden Werte für ϵ_m kleiner oder gleich ϵ sind. Damit dies der Fall ist muss wegen der mit (5.24) gewählten Abschätzung gelten:

$$\begin{aligned} & \log(\sqrt{2} m) && \leq && \lceil \log(n-2) \rceil \\ \Rightarrow \quad & \log(m) && \leq && \left\lceil \log\left(\frac{n-2}{\sqrt{2}} \right) \right\rceil \\ \Rightarrow \quad & m && \leq && 2^{\left\lceil \log\left(\frac{n-2}{\sqrt{2}} \right) \right\rceil} \end{aligned}$$

Ungleichung (5.23) wird erfüllt, wenn man

$$t_5 = m$$

wählt. Die Gültigkeit dieser Behauptung ergibt sich insbesondere aus der Betrachtung der Größenordnungen der Zähler und Nenner in der Ableitung von t_3.

So erhält man durch Abschätzung von (5.22) nach oben für die Anzahl der Prozessoren:

$$\begin{aligned}
& t_2 + \int_{2^{\lfloor \log\left(\frac{n-2}{\sqrt{2}}\right)\rfloor}}^{n-2} m^{2+\gamma+0.5} dm + \int_{2}^{2^{\lfloor \log\left(\frac{n-2}{\sqrt{2}}\right)\rfloor}} \mathrm{e}^{(2+\gamma+f(m))\ln(m)} dm \\
\leq\ & t_2 + \frac{1}{3.5+\gamma} m^{3.5+\gamma}\Big|_{2^{\lfloor \log\left(\frac{n-2}{\sqrt{2}}\right)\rfloor}}^{n-2} + \left(m^{2+\gamma+f(m)} m\right)\Big|_{2}^{2^{\lfloor \log\left(\frac{n-2}{\sqrt{2}}\right)\rfloor}} \\
=\ & 4 + (n-2)^{2+\gamma+\epsilon} + \frac{(n-2)^{2.5+\gamma}}{2} \\
& + m|_2^{n-2} + \left(\frac{1}{7+2\gamma} m^{3.5+\gamma}\right)\Big|_2^{n-2} + \frac{1}{3.5+\gamma} m^{3.5+\gamma}\Big|_{2^{\lfloor \log\left(\frac{n-2}{\sqrt{2}}\right)\rfloor}}^{n-2} \\
& + m^{3+\gamma+f(m)}\Big|_2^{2^{\lfloor \log\left(\frac{n-2}{\sqrt{2}}\right)\rfloor}}
\end{aligned}$$

Dieser Term wird mit t_8 bezeichnet. An ihm erkennt man, daß die Anzahl der Prozessoren mit wachsendem n asymptotisch

$$\frac{3}{7+2\gamma}(n-2)^{3.5+\gamma}$$

beträgt.

Schließlich müssen noch die Matrizen C_1 bis C_n miteinander multipliziert werden. Dies geschieht mit Hilfe der Binärbaummethode nach 1.5.1. Es wird definiert

$$n' := \left\lfloor \frac{n}{2} \right\rfloor \ .$$

Da nach 5.1.1 bei allen Multiplikationen Dreiecks-Toeplitz-Matrizen verknüpft werden und C_i eine $(n-i+2) \times (n-i+1)$-Matrix handelt, können diese Multiplikationen in

$$(\lceil \log(n+1) \rceil + 1) \lceil \log(n) \rceil \tag{5.25}$$

Schritten von weniger als

$$\begin{aligned}
& \sum_{k=1}^{n'} (n-2k+2) * (n-2k+1) \\
=\ & \sum_{k=1}^{n'} (2k(2k-1)) \\
=\ & 4\sum_{k=1}^{n'} k^2 - 2\sum_{k=1}^{n'} k \\
\overset{4.6.1,\ 4.6.2}{=}\ & 4\frac{n'(n'+1)(2n'+1)}{6} - 2\frac{n'(n'+1)}{2}
\end{aligned}$$

Prozessoren durchgeführt werden. Für ein gegebenes n ist der Wert dieses Terms ist kleiner als der Wert von t_8.

In Verbindung mit 2.4.4 ergibt sich als Endergebnis der Analyse, daß mit Hilfe des in diesem Kapitel vorgestellen Algorithmus die Determinante einer $n \times n$-Matrix in weniger als[8]

$$\gamma_S \left\lceil \lceil \log(n-2) \rceil^2 + \left(1 + \frac{1}{\gamma_S} + \frac{1}{\epsilon}\right) \lceil \log(n-2) \rceil + \frac{1}{\epsilon} + \frac{1}{\gamma_S} + 0.5 \right\rceil + (\lceil \log(n+1) \rceil + 1) \lceil \log(n) \rceil$$

[8] Summe von (5.17) und (5.25)

Schritten von weniger als t_8 Prozessoren berechnet werden kann.

Da in der Praxis für die Matrizenmultiplikation Satz 1.5.3 statt der in [CW90] angegebenen Methode benutzt wird, ist für diesen Fall in allen obigen Termen

$$\gamma = \gamma_S = \gamma_P = 1$$

zu setzen.

Vergleicht man B-Alg. mit C-Alg., BGH-Alg. und P-Alg., fällt wiederum das Fehlen von Fallunterscheidungen auf. Weiterhin werden wie bei BGH-Alg. keine Divisionen verwendet, so daß B-Alg. auch in Ringen anwendbar ist.

Betrachtet man die Aufwandsanalyse, so erkennt man, daß B-Alg. leicht schlechter ist als C-Alg. und deutlich besser als BGH-Alg. . In Kapitel 6 wird P-Alg. in diese Rangfolge eingereiht.

Kapitel 6

Der Algorithmus von Pan

In diesem Kapitel wird der Algorithmus von V. Pan [Pan85] zur Determinantenberechnung vorgestellt. Er kommt ebenfalls ohne Divisionen[1] aus und berechnet die Determinante iterativ. Auf diesen Algorithmus wird mit *P-Alg.* Bezug genommen[2].

Man erhält insbesondere durch Variation der in den Unterkapitel 6.5 und 6.6 dargestellten Inhalte einige weitere Versionen des Algorithmus. Für eine vollständige Darstellung müssen alle diese Varianten beschrieben und auf ihre Effizienz hin untersucht werden[3]. Da dies jedoch den Rahmen dieses Textes sprengt, beschränken sich die folgenden Darstellungen auf die effizienteste Version des Algorithmus.

P-Alg. bietet erheblich mehr Variationsmöglichkeiten als C-Alg., BGH-Alg. und B-Alg., die, wie erwähnt, nicht alle hier behandelt werden können. Vergleicht man P-Alg. von seiner Methodik her mit den drei anderen, so erkennt man Ähnlichkeiten zu BGH-Alg. . In beiden Algorithmen werden mehrere auch separat bedeutsame teilweise schon länger bekannte Verfahren zusammen verwendet. Von diesen Verfahren hebt sich lediglich die in Unterkapitel 6.5 dargestellte Methode zur Berechnung einer Näherungsinversen ab. Sie wurde in [Pan85] erstmalig veröffentlicht.

6.1 Diagonalisierbarkeit

In diesem Unterkapitel wird die Diagonalisierbarkeit von Matrizen behandelt. Es ist für das Verständnis des in diesem Kapitel dargestellten Algorithmus zur Determiantenberechnung nicht unbedingt erforderlich und kann daher beim Lesen auch übersprungen werden. Im folgenden werden jedoch einige Hintergründe der im Unterkapitel 6.3 dargestellten Methode von Krylov näher beleuchtet, die ein paar Zusammenhänge klarer werden lassen.

Literatur zu diesem Thema ist neben den in Kapitel 2 genannten Stellen auch [Zur64] S. 169 ff .

Zum Problem der Diagonalisierbarkeit[4] gelangt man über den Begriff der Basis eines Vektorraumes. Da es sich dabei um Grundlagen der Linearen Algebra handelt, erfolgt die Darstellung vergleichsweise oberflächlich.

Sei K ein Körper und V ein K-Vektorraum. Seien $k_1, k_2, \ldots, k_n \in K\backslash\{0\}$ und $v_1, v_2, \ldots, v_n \in V$. Dann wird

$$k_1v_1 + k_2v_2 + \ldots + k_nv_n \tag{6.1}$$

als *Linearkombination* der Vektoren v_1 bis v_n bezeichnet. Sei 0_m der Nullvektor in V. Falls für die Vektoren die Bedingung

$$k_1v_1 + k_2v_2 + \ldots + k_nv_n = 0_m \Rightarrow k_1 = k_2 = \ldots = k_n = 0$$

[1] vgl. Bemerkungen in 3.8
[2] vgl. Unterkapitel 1.3
[3] Näheres dazu ist insbesondere in [PR85b] zu finden
[4] Definition s. u.

erfüllt ist, werden sie als *linear unabhängig* bezeichnet, ansonsten als *linear abhängig* . Falls jedes Element von V als Linearkombination der Vektoren $v_1, \ldots, v_n$ darstellbar ist, werden diese Vektoren als *Basis* bezeichnet.

Für alle Basen gilt die Aussage:

> Je zwei Basen eines K-Vektorraumes bestehen aus dergleichen Anzahl von Vektoren.

Sei B die Basis des K-Vektorraumes V. Dann wird die Anzahl der Vektoren, die B bilden, als *Dimension von* V , kurz $\dim(V)$, bezeichnet.

Zur Darstellung von Elementen eines Vektorraumes V wählt man sich eine Basis und beschreibt jedes Element des Vektorraumes als Linearkombination der Elemente der Basis. Sei n die Dimension des Vektorraumes. Dann kann man auf diese Weise jedes Element von V als n-Tupel von Elementen des zugrunde liegenden Körpers K betrachten. Man erhält die Vektorschreibweise:

$$\begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_n \end{bmatrix} \tag{6.2}$$

Die Basis der Form

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

wird als *kanonische Basis* bezeichnet.

Zur Betrachtung der Beziehungen verschiedener Basen zueinander werden diese Basen ihrerseits bzgl. der kanonischen Basis dargestellt.

Wird ein Vektor v bzgl. einer Basis B dargestellt, so wird dies folgendermaßen ausgedrückt:

$$\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}_B$$

Werden Vektoren zu Matrizen zusammengefaßt, so wird die gleiche Schreibweise auch für diese Matrizen verwendet.

Sei V ein K-Vektorraum der Dimension n und W ein K-Vektorraum der Dimension m. Ein Ergebnis der Linearen Algebra lautet, daß dann die Menge aller K-Vektorraumhomomorphismen[5] f

$$f : V \to W$$

isomorph ist zur Menge aller $m \times n$-Matrizen A, wenn man die Abbildung definiert als[6]

$$f\left(\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}\right) := \begin{bmatrix} \sum_{j=1}^{n} a_{1,j} v_j \\ \vdots \\ \sum_{j=1}^{n} a_{m,j} v_j \end{bmatrix}$$

[5] also die Menge aller *strukurverträglichen* linearen Abbildungen

[6] Dies entspricht der Matrizenmultiplikation (vgl. 1.5.3), wenn man den abzubildenden Vektor v als $n \times 1$-Matrix betrachtet (vgl. 1.5.3).

Die Untersuchung der K-Vektorraumhomomorphismen kann man also anhand der entsprechenden Matrien vornehmen. Im folgenden sind nur quadratische Matrizen von Interesse. Deshalb werden in den weiteren Ausführungen nur diese Matrizen beachtet.

Stellt man die Vektoren einer Basis B_V bzgl. einer anderen Basis B_W (normalerweise der kanonischen Basis) dar und betrachtet sie als Spaltenvektoren einer Matrix

$$[B_V]_{B_W} \ ,$$

so erkennt man beim Vergleich von (6.1) und (6.2) miteinander, daß man einen bzgl. B_V dargestellten Vektor x in seine Darstellung bzgl. B_W umrechnen kann durch

$$[x]_{B_W} = B_V[x] \ . \tag{6.3}$$

Man erkennt also, daß man eine Basis auch als Vektorraumhomomorphismus betrachten kann. Die umgekehrte Betrachtungsweise ist natürlich nicht möglich.

Um zum Begriff der *Diagonalisierbarkeit* zu gelangen, betrachten wir nun, was passiert, wenn man Basen austauscht und die Darstellungen bzgl. der neuen Basen vornehmen will.

Seien V und W jeweils K-Vektorräume sowie B_V und B_W jeweils Basen dieser Vektorräume. Sei

$$f : V \to W$$

ein Vektorraumhomomorphismus und A die entsprechende Matrix. Es gelte

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}_{B_W} = A \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{B_V} \tag{6.4}$$

Wechselt man nun zu den Basen $\tilde{B_V}$ und $\tilde{B_W}$ und stellt die neuen Basen bzgl. der alten durch die Matrizen C und D dar, so gilt entsprechend (6.3):

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}_{B_V} = C \begin{bmatrix} \tilde{x_1} \\ \tilde{x_2} \\ \vdots \\ \tilde{x_n} \end{bmatrix}_{\tilde{B_V}}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}_{B_W} = D \begin{bmatrix} \tilde{y_1} \\ \tilde{y_2} \\ \vdots \\ \tilde{y_n} \end{bmatrix}_{\tilde{B_W}}$$

Gleichung (6.4) bekommt also folgendes Aussehen:

$$D \begin{bmatrix} \tilde{y_1} \\ \tilde{y_2} \\ \vdots \\ \tilde{y_n} \end{bmatrix}_{\tilde{B_W}} = A\,C \begin{bmatrix} \tilde{x_1} \\ \tilde{x_2} \\ \vdots \\ \tilde{x_n} \end{bmatrix}_{\tilde{B_V}}$$

Multipliziert man beide Seiten mit D^{-1}, erhält man:

$$\begin{bmatrix} \tilde{y_1} \\ \tilde{y_2} \\ \vdots \\ \tilde{y_n} \end{bmatrix}_{\tilde{B_W}} = D^{-1}\,A\,C \begin{bmatrix} \tilde{x_1} \\ \tilde{x_2} \\ \vdots \\ \tilde{x_n} \end{bmatrix}_{\tilde{B_V}}$$

Die Abbildung f wird bzgl. der neuen Basen $\tilde{B_V}$ und $\tilde{B_W}$ also durch die Matrix $A' := D^{-1}AC$ dargestellt. Wählt man die neuen Basen geeignet, so ist es immer möglich zu erreichen, daß A'

die Form

$$\begin{bmatrix} a'_{1,1} & 0 & \cdots & 0 \\ 0 & a'_{2,2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a'_{n,n} \end{bmatrix}$$

bekommt. Eine Matrix dieser Form wird als *Diagonalmatrix* bezeichnet.

Betrachtet man nun statt einer Abbildung zwischen den Vektorräumen eine Abbildung in V und wechselt die Basis von V, so besitzt die Abbildung bzgl. der neuen Basis die Form $C^{-1}AC$. Es stellt sich wiederum die Frage, ob es möglich ist, die neue Basis so zu wählen, daß $C^{-1}AC$ eine Diagonalmatrix ist. Eine Matrix A, für die das möglich ist, wird als *diagonalisierbar* bezeichnet.

Um eine Beziehung zur Methode von Krylov (siehe Unterkapitel 6.3) herstellen zu können, folgt eine Charakterisierung der Diagonalisierbarkeit. Dazu greifen wir auf den bereits in 2.3.1 definierten Begriff des *Unterraumes* zurück. Anhand dieser Definitionen erkennt man, daß alle zu einem Eigenwert gehörenden Eigenvektoren zusammen mit dem Nullvektor einen Unterraum des zugrunde liegenden Vektorraumes bilden. Er wird als *Eigenraum* bezeichnet.

An dieser Stelle werden die Eigenwerte mit der Diagonalisierbarkeit in Verbindung gebracht:

Satz 6.1.1 *Eine $n \times n$-Matrix A ist genau dann diagonalisierbar, wenn der zugrunde liegende K-Vektorraum V eine Basis aus Eigenvektoren von A besitzt.*

Beweis Wir verzichten hier auf einen ausführlichen Beweis. Die Ideen für die beiden Beweisrichtungen sind:

- Ist eine Matrix A diagonalisierbar und wechselt man die Basis so, daß A Diagonalgestalt bekommt, werden dadurch die Vektoren der kanonischen Basis zu Eigenvektoren der Matrix.
- Bilden die Eigenvektoren einer Matrix A eine Basis von V und benutzt man diese Basis zur Darstellung bekommt A die Form einer Diagonalmatrix.

□

Satz 6.1.2 *Seien $\lambda_1, \ldots, \lambda_k$ paarweise verschiedene Eigenwerte der $n \times n$-Matrix A. Sei v_i ein Eigenvektor zu λ_i. Dann sind die Eigenvektoren $v_1, \ldots, v_k$ linear unabhängig.*

Beweis Der Beweis erfolgt durch Induktion nach der Anzahl der verschiedenen Eigenwerte k:

$k = 1$
: Für diesen Fall ist die Behauptung offensichtlich richtig.

$k > 1$
: Die Behauptung gelte für $k-1$ und sei für k zu zeigen. Induktionsvoraussetzung ist also, daß $v_1, \ldots, v_{k-1}$ linear unabhängig sind. Angenommen $v_1, \ldots, v_k$ sind linear abhängig. Dann existieren eindeutig bestimmte $r_1, \ldots, r_{k-1} \in K$ mit

$$v_k = r_1 v_1 + \cdots + r_{k-1} v_{k-1} \ . \tag{6.5}$$

Da v_k nicht der Nullvektor sein kann, muß mindestens einer der Faktoren $r_1, \ldots, r_k$ ungleich Null sein, z. B. r_i.

Betrachtet man A als Abbildung und wendet diese Abbildung auf (6.5) an, so kann man, da es sich um Eigenvektoren handelt, auch mit den Eigenwerten multiplizieren und erhält

$$\lambda_k v_k = \lambda_1 r_1 v_1 + \cdots + \lambda_k r_k v_k \ .$$

Ist nun $\lambda_k = 0$, dann muß wegen der Verschiedenheit der Eigenwerte $\lambda_i \neq 0$ sein und man erhält einen Widerspruch zur linearen Unabhängigkeit von $v_1, \ldots, v_k$.

Ist $\lambda_k \neq 0$ erhält man einen Widerspruch zur Eindeutigkeit der Darstellung von v_k.

□

Aus 6.1.1 und 6.1.2 ergibt sich:

Folgerung 6.1.3 *Die maximale Anzahl verschiedener Eigenwerte einer Matrix ist gleich der Dimension des zugrunde liegenden Vektorraumes.*

Besitzt eine Matrix maximal viele verschiedene Eigenwerte, so ist sie diagonalisierbar.

Für die zweite Folgerung benötigen wird einen weiteren Betriff. Seien dazu T und U Unterräume des K-Vektorraumes V. Dann wird die Menge

$$\{w \mid \exists k, l \in K,\, t \in T,\, u \in U :\ w = kt + lu\}$$

aller Linearkombinationen zweier Vektoren aus T und U als *direkte Summe von T und U* bezeichnet. Anhand von 2.3.1 erkennt man, daß die direkte Summe zweier Unterräume von V wiederum ein Unterraum von V ist.

Somit erhält man aus 6.1.1 und 6.1.2:

Folgerung 6.1.4 *Eine Matrix ist genau dann diagonalisierbar, wenn die direkte Summe aller Eigenräume der Matrix den zugrundliegenden Vektorraum ergibt.*

Die Bedeutung der in diesem Unterkapitel dargestellten Sachverhalte wird deutlich, wenn man sie mit den in Unterkapitel 6.3 erwähnten Einschränkungen für die Verwendbarkeit der Methode von Krylov zur Berechnung der Koeffizienten des charakteristischen Polynoms vergleicht.

6.2 Das Minimalpolynom

Die Methode von Krylov (siehe 6.3) dient zur Bestimmung der Koeffizienten des Minimalpolynoms einer Matrix. Deshalb wird hier dieses Minimalpolynom zunächst näher betrachtet. Eine tiefergreifende Behandlung befindet sich z. B. in [Zur64] S. 233 ff .

In Satz 5.2.2 wird bewiesen, daß eine $n \times n$-Matrix A ihre eigene charakteristische Gleichung erfüllt. Diese Beobachtung führt zu:

Definition 6.2.1 Das Polynom $m(\lambda)$ mit dem kleinsten Grad, für das die Gleichung

$$m(A) = 0_n$$

erfüllt ist, wird *Minimalpolynom* genannt. Die Gleichung wird als *Minimumgleichung* bezeichnet.

Um die Methode von Krylov verstehen zu können, müssen wir verschiedene Eigenschaften des Minimalpolynoms beleuchten:

Satz 6.2.2 *Sei A eine $n \times n$-Matrix und $f(\lambda)$ ein Polynom. Es gelte*

$$f(A) = 0_n \ .$$

Dann ist $f(\lambda)$ ein Vielfaches des Minimalpolynoms $m(\lambda)$ von A.

Beweis Angenommen die Behauptung ist falsch. Dann entsteht bei der Division von $f(\lambda)$ durch $m(\lambda)$ ein Rest $r(\lambda)$ und für ein geeignetes Polynom $q(\lambda)$ gilt:

$$f(\lambda) = q(\lambda)m(\lambda) + r(\lambda) \ .$$

Der Grad von $r(\lambda)$ ist kleiner als der Grad von $m(\lambda)$. Wird nun in diese Gleichung A eingesetzt, erhält man

$$0_n = 0_n + r(A) \ .$$

Also muß auch gelten

$$r(A) = 0_n \ .$$

Da jedoch das Minimalpolynom das Polynom mit dem kleinsten Grad ist, das diese Bedingung erfüllt, führt dies zu einem Widerspruch. □

Aus 5.2.2 und 6.2.2 ergibt sich:

Folgerung 6.2.3 *Das charakteristische Polynom ist ein Vielfaches des Minimalpolynoms.*

Da wir die Methode von Krylov zur Berechnung des charakteristischen Polynoms verwenden wollen, müssen wir wissen, unter welchen Umständen es mit dem Minimalpolynom zusammenfällt. Diese Frage beantwortet der folgende Satz:

Satz 6.2.4 *Sei A eine $n \times n$-Matrix. Es wird definiert:*

$$C := A - \lambda E_n \ .$$

Sei

$$p(\lambda) = \det(C)$$

das charakteristische Polynom von A. Es gelte

$$m(\lambda) = \frac{p(\lambda)}{q(\lambda)} \tag{6.6}$$

.

Das Polynom $m(\lambda)$ ist genau dann das Minimalpolynom von A, wenn $q(\lambda)$ der größte gemeinsame Teiler (ggT) der Determinanten aller $(n-1) \times (n-1)$-Untermatrizen von C ist.

Beweis Der Beweis erfolgt in zwei Schritten:

- Sei zunächst $q(\lambda)$ der ggT Determinanten der Untermatrizen. Es ist zu zeigen, daß dann $m(\lambda)$ das Minimalpolynom ist.

  Mit Hilfe von Satz 3.2.2 (Zeilen- und Spaltenentwicklung) folgt, daß $p(\lambda)$ durch $q(\lambda)$ teilbar ist. D. h. es gibt ein Polynom $m'(\lambda)$, so daß

  $$p(\lambda) = m'(\lambda)q(\lambda) \ . \tag{6.7}$$

  Weiterhin gibt es eine $n \times n$-Matrix M aus teilerfremden Polynomen über λ, so daß gilt:

  $$\mathrm{adj}\,(C) = Mq(\lambda) \ . \tag{6.8}$$

  Nach Satz 3.6.3 gilt:

  $$C\,\mathrm{adj}\,(C) = E_n p(\lambda) \ . \tag{6.9}$$

Mit (6.7) folgt aus (6.9):

$$C\,\mathrm{adj}\,(C) = E_n m'(\lambda) q(\lambda) \ . \tag{6.10}$$

Mit (6.8) folgt aus (6.9):

$$C\,\mathrm{adj}\,(C) = CMq(\lambda) \ . \tag{6.11}$$

Aus (6.11) und (6.10) folgt:

$$CM = E_n m'(\lambda) \ .$$

Benutzt man die Definition von C, erhält man

$$(A - \lambda E_n)M = E_n m'(\lambda) \ .$$

Setzt man nun in dieser Gleichung A für λ ein, ergibt sich

$$m'(A) = 0_n \ .$$

Nach Satz 6.2.2 ist $m'(\lambda)$ also ein Vielfaches des Minimalpolynoms $m(\lambda)$.

Angenommen es gibt ein Polynom $m''(\lambda)$ mit

$$m''(A) = 0_n \ ,$$

dessen Grad kleiner ist als der Grad von $m'(\lambda)$. Da der Grad des charakteristischen Polynoms immer n ist, muß dann der Grad von $q(\lambda)$ in (6.7) und somit auch in (6.8) entsprechend größer sein, im Widerspruch dazu, daß $q(\lambda)$ der ggT ist. Also ist $m'(\lambda)$ das Minimalpolynom.

- Sei nun $m(\lambda)$ das Minimalpolynom. Dann ist zu zeigen, daß $q(\lambda)$ der ggT der Unterdeterminanten ist.

  Nach 6.2.3 gibt es ein Polynom $q'(\lambda)$, so daß

  $$p(\lambda) = q'(\lambda) m(\lambda) \ . \tag{6.12}$$

  Benutzt man für das Minimalpolynom die Koeffizientendarstellung, erhält man mit geeigneten Koeffizienten b_i:

  $$\begin{aligned} & -E_n m(\lambda) \\ &= \ m(A) - E_n m(\lambda) \\ &= \ b_m(A^m - \lambda^m E_n) + b_{m-1}(A^{m-1} - \lambda^{m-1} E_n) + \cdots + b_1(A - \lambda E_n) \ . \end{aligned}$$

  Also ist $m(\lambda)$ durch $(A - \lambda E_n)$ teilbar und es gibt eine Matrix N aus Polynomen über λ, so daß gilt:

  $$m(\lambda) E_n = (A - \lambda E_n) N \ . \tag{6.13}$$

  Mutipliziert man beide Seiten mit $q'(\lambda)$, erhält man

  $$p(\lambda) E_n = q'(\lambda)(A - \lambda E_n) N \ .$$

  Subtrahiert man nun auf beiden Seiten

  $$\begin{aligned} & p(\lambda) E_n \\ \overset{\text{nach } 3.6.3}{=} \ & C\,\mathrm{adj}\,(C) \\ = \ & (A - \lambda E_n)\mathrm{adj}\,(C) \ , \end{aligned}$$

  erhält man

  $$0_{n,n} = \underbrace{(A - \lambda E_n)}_{(*1)} \ \underbrace{(q'(\lambda) N - \mathrm{adj}\,(C))}_{(*2)}$$

  In dieser Gleichung ist Term $(*1)$ für ein beliebig gewähltes λ ungleich der Nullmatix[7]. Also muß Term $(*2)$ gleich der Nullmatrix sein, so daß gilt:

  $$q'(\lambda) N = \mathrm{adj}\,(C)$$

[7] abgesehen von einigen Sonderfällen, deren Existenz den Beweis jedoch nicht beeinträchtigt

Also ist $q'(\lambda)$ Teiler der Elemente von adj (C).

Angenommen es gibt ein Polynom $q''(\lambda)$, dessen Grad größer ist als der Grad von $q'(\lambda)$ und das ebenfalls Teiler der Elemente von adj (C) ist. Da der Grad von $p(\lambda)$ immer n ist, muß dann der Grad von $m(\lambda)$ in (6.12) kleiner sein, im Widerspruch zu der Voraussetzung, daß $m(\lambda)$ das Minimalpolynom ist.

□

Berechnet man in (6.13) auf beiden Seiten die Determinante, erhält man

$$m^n(\lambda) = p(\lambda) \det(N) \ . \tag{6.14}$$

Aus (6.6) und (6.14) ergibt sich:

Folgerung 6.2.5 *Es ist λ_i genau dann Nullstelle von $m(\lambda)$, wenn es auch Nullstelle von $p(\lambda)$ ist.*

Anders ausgedrückt: $m(\lambda)$ und $p(\lambda)$ besitzen die gleichen Nullstellen mit evtl. verschiedenen Vielfachheiten. Das führt zu einer weiteren Schlußfolgerung:

Folgerung 6.2.6 *Besitzt eine $n \times n$-Matrix n paarweise verschiedene Eigenwerte, so stimmen ihr Minimalpolynom und ihr charakteristisches Polynom überein.*

Falls die Eigenwerte nicht paarweise verschieden sind, können Minimalpolynom und charakteristisches Polynom also verschieden sein, was eine Einschränkung für die Methode von Krylov (siehe Unterkapitel 6.3) bedeutet, wenn man sie zur Berechnung der Koeffizienten des charakteristischen Polynoms verwendet. Wann die beiden Polynome verschieden sind, zeigt der folgende Satz:

Satz 6.2.7 *Das Minimalpolynom $m(\lambda)$ einer Matrix A stimmt genau dann mit dem charakteristischen Polynom $p(\lambda)$ der Matrix überein, wenn die Dimension jedes Eigenraumes 1 beträgt*[8].

Beweis Aus 6.2.4 folgt, daß $m(\lambda)$ und $p(\lambda)$ genau dann übereinstimmen, wenn der ggT der Determinanten aller $(n-1) \times (n-1)$-Untermatrizen der charakteristischen Matrix von A gleich 1 ist.

Betrachtet man die beiden Polynome in ihrer Linearfaktorendarstellung, muß der genannte ggT, falls er ungleich 1 ist, mit einem Linearfaktor von $p(\lambda)$ übereinstimmen. Für die entsprechende Nullstelle von $p(\lambda)$ verschwinden auch alle $(n-1) \times (n-1)$-Unterdeterminanten. Es gibt also für diese Nullstelle keine $n-1$ linear unabhängigen Spaltenvektoren der charakteristischen Matrix. Der Rangabfall der Nullstelle ist also größer als 1 und es gibt mehr als einen linear unabhängigen Eigenvektor zu diesem Eigenwert. □

Aus 6.2.6 und 6.2.7 erhält man:

Folgerung 6.2.8 *Falls die Eigenwerte nicht paarweise verschieden sind und das Minimalpolynom mit dem charakteristischen Polynom übereinstimmt, zerfällt es im Körper der reellen Zahlen nicht in Linearfaktoren.*

Wie bereits mehrfach erwähnt, erfolgt nun die Anwendung der Ergebnisse dieses Unterkapitels auf die Methode von Krylov.

[8] Es ist zu beachten, daß diese Aussage nicht dazu äquivalent ist, daß die direkte Summe der Eigenräume den gesamten Vektorraum ergibt.

6.3 Die Methode von Krylov

In diesem Unterkapitel wird die Methode von Krylov zur Bestimmung der Koeffizienten des Minimalpolynoms einer Matrix beschrieben (siehe z. B. [Zur64] ab S. 171 oder [Hou64] ab S. 149; Originalveröffentlichung [Kry31]). Wie in Unterkapitel 6.2 beschrieben wird, ist das Minimalpolynom unter bestimmten Bedingungen mit dem charakteristischen Polynom identisch. Da sich unter den Koeffizienten des charakteristischen Polynoms auch die Determinante der zugrunde liegenden Matrix befindet (vgl. 2.4.4), ist es möglich, Krylovs Methode zur Determinantenberechnung zu verwenden, was im Algorithmus von Pan ausgenutzt wird.

Sei A die $n \times n$-Matrix, deren Minimalpolynom zu berechnen ist. Sei z_0 ein geeigneter Vektor der Länge n. Wie z_0 beschaffen ist, wird noch behandelt. Sei $i \in \mathbb{N}$ gegeben. Die Vektoren

$$z_1, \ldots, z_i$$

erhält man durch

$$\begin{aligned} z_1 &:= Az_0 \\ z_2 &:= Az_1 = A^2 z_0 \\ &\vdots \\ z_i &:= Az_{i-1} = A^i z_0 \ . \end{aligned} \tag{6.15}$$

Die Vektoren

$$z_0, \ldots, z_i$$

werden als *iterierte Vektoren* bezeichnet. Betrachtet man die iterierten Vektoren als Spaltenvektoren einer Matrix, erhält man eine sogenannte *Krylov*-Matrix:

$$K(A, z_0, i) := [z_0, z_1, z_2, \ldots, z_{i-1}] \ .$$

Zwischen den iterierten Vektoren besteht eine lineare Abhängigkeit besonderer Form, die von Krylov [Kry31] für das hier zu beschreibende Verfahren entdeckt wurde.

Das Minimalpolynom von A wird mit $m(\lambda)$ bezeichnet. Es gilt also

$$m(A) = 0_{n,n} \ . \tag{6.16}$$

Das Polynom habe den Grad j. Seien $c_0, \ldots c_{j-1}$ die Koeffizienten des Polynoms. Definiert man

$$c := \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{j-1} \end{bmatrix}$$

dann ergibt sich

$$\begin{aligned} & m(A) &&= 0_{n,n} \\ \Leftrightarrow\ & A^j + c_{j-1}A^{j-1} + \ldots + c_1 A + c_0 E_n &&= 0_{n,n} \\ \Leftrightarrow\ & c_{j-1}A^{j-1} + \ldots + c_1 A + c_0 E_n &&= -A^j \\ \Leftrightarrow\ & c_0 E_n z_0 + c_1 A z_0 + \ldots + c_{j-1}A^{j-1} z_0 &&= -A^j z_0 \\ \Leftrightarrow\ & K(A, z_0, j)c &&= -A^j z_0 \end{aligned} \tag{6.17}$$

Gleichung (6.17) kann man als lineares Gleichungssystem in Matrizenschreibweise betrachten. Multipliziert man die rechte Seite dieser Gleichung aus, erhält man einen Vektor der Länge n. Nach 2.3.6 ist das entsprechende Gleichungssystem genau dann lösbar, wenn

$$\operatorname{rg}(K(A, z_0, j) = \operatorname{rg}([K(A, z_0, j),\ A^j z_0]) \ .$$

Wir suchen eine eindeutige Lösung des Gleichungssystems und erhalten diese durch Verwendung von 2.3.7, wonach der bis hierhin nicht näher beschriebene Vektor z_0 so gewählt werden

muß, daß die Spaltenvektoren von $K(A, z_0, j)$ linear unabhängig und die Spaltenvektoren von $[K(A, z_0, j),\ A^j z_0]$ linear abhängig sind.

An dieser Stelle wird deutlich, daß $m(\lambda)$ das Polynom mit dem kleinsten Grad sein muß, das (6.16) erfüllt, damit (6.17) eine eindeutige Lösung besitzt. Falls ein Polynom $m_1(\lambda)$ existiert, daß (6.16) erfüllt und dessen Grad kleiner ist als der Grad von $m(\lambda)$, dann ist die lineare Abhängigkeit unabhängig von der Wahl von z_0 bereits für weniger als j iterierte Vektoren gegeben.

Da nach (6.17) die ersten $j+1$ iterierten Vektoren linear abhängig sind, bleibt zu zeigen, daß die ersten j dieser Vektoren linear unabhängig sind. Dazu betrachten wir das $s \in \mathbb{N}$ mit der Eigenschaft, daß s paarweise verschiedene Eigenvektoren von A immer linear unabhängig und $s+1$ von ihnen immer linear abhängig sind. Aus den Grundlagen über Eigenvektoren und lineare Unabhängigkeit geht hervor, daß ein solches s gibt.

Seien somit

$$x_1, \ldots, x_s$$

linear unabhängige Eigenvektoren von A. Sei der iterierte Vektor z_0 darstellbar als Linearkombination einer maximalen Anzahl (also s) von Eigenvektoren von A. Demnach gilt für geeignete

$$d_1, \ldots, d_s \ ,$$

ungleich Null[9]:

$$z_0 = d_1 x_1 + \cdots + d_s x_s \ . \tag{6.18}$$

Eine lineare Abhängigkeit zwischen den ersten j iterierten Vektoren hat die Form

$$h(z_0) := e_0 z_0 + e_1 z_1 + \cdots + e_{j-1} z_{j-1} = 0_n \ , \tag{6.19}$$

wobei nicht alle e_i gleich Null sind. Mit Hilfe der Gleichungen (6.15) und (6.18) in Verbindung mit den Eigenwertgleichungen[10]

$$Ax_i = \lambda_i x_i$$

erhält man:

$$\begin{array}{rcll} z_0 &=& d_1 x_1 + \cdots + d_s x_s & \mid * e_0 \\ z_1 &=& \lambda_1 d_1 x_1 + \cdots + \lambda_s d_s x_s & \mid * e_1 \\ z_2 &=& \lambda_1^2 d_1 x_1 + \cdots + \lambda_s^2 d_s x_s & \mid * e_2 \\ & \vdots & & \\ z_{j-1} &=& \lambda_1^{j-1} d_1 x_1 + \cdots + \lambda_s^{j-1} d_s x_s & \mid * 1 \end{array}$$

Diese Gleichungen für die z_i werden mit den am rechten Rand angegebenen Werten (vgl. (6.19)) multipliziert und anschließend addiert. Definiert man

$$g(\lambda) := e_0 + e_1 \lambda + \cdots + e_{j-1} \lambda^{j-1} \ ,$$

so lautet das Ergebnis in Verbindung mit (6.19) :

$$h(z_0) = g(\lambda_1) d_1 x_1 + \cdots + g(\lambda_s) d_s x_s = 0_n \ .$$

Da die x_k nach Voraussetzung linear unabhängig sind, muß also gelten

$$\forall 1 \le k \le s : \ g(\lambda_k) d_k = 0 \ .$$

Da wiederum nach Voraussetzung z_0 eine Linearkombination aller s linear unabhängigen Eigenvektoren x_i (s. o.) ist, folgt

$$\forall 1 \le i \le s : \ g(\lambda_i) = 0 \ . \tag{6.20}$$

Da die Dimension jedes Eigenraumes mindestens 1 beträgt, folgt mit Hilfe von 6.2.5, daß die maximale Anzahl linear unabhängiger Eigenvektoren s mindestens so groß ist wie der Grad des Minimalpolynoms j.

[9] Eine ausführliche Diskussion der Eigenschaften der iterierten Vektoren, insbesondere ihrer Beziehung zu den Eigenvektoren, befindet sich in [Bod59] (Teil 2, Kapitel 2).

[10] vgl. Gleichung (2.9)

Da $g(\lambda)$ als Polynom vom Grad $j-1$ nur maximal $j-1$ Nullstellen besitzen kann, folgt aus (6.20)

$$e_0 = e_1 = \cdots = e_{j-1} = 0 \ ,$$

Daraus wiederum folgt mit (6.19), daß die iterierten Vektoren linear unabhängig sind.

Der bis hierhin geführte Beweis der linearen Unabhängigkeit iterierter Vektoren verwendet die maximale Anzahl s linear unabhängiger Eigenvektoren. Betrachten wir deshalb diesen Wert s genauer. Es gibt zwei Fälle:

- Minimalpolynom und charakteristisches Polynom sind identisch. Satz 6.2.7 führt zu zwei Unterfällen:
  - Die Eigenwerte sind paarweise verschieden. In diesem Fall ist s gleich dem Grad n des charakteristischen Polynoms und somit gleich j.
  - Die Eigenwerte sind nicht paarweise verschieden. Nach 6.2.7 ist s gleich der Anzahl verschiedener Eigenwerte und damit in $\mathbb{Q}$ kleiner als n und mindestens 1. In diesem Fall läßt sich die lineare Unabhängigkeit nur für weniger als j iterierte Vektoren beweisen und die Methode von Krylov ist zur Bestimmung der Koeffizienten des Minimalpolynoms nicht anwendbar.
- Minimalpolynom und charakteristisches Polynom sind nicht identisch. Es gibt wiederum zwei Unterfälle:
  - Die direkte Summe der Eigenräume ergibt den gesamten Vektorraum. In diesem Fall ist $s = n > j$.
  - Die direkte Summe der Eigenräume ergibt nicht den gesamten Vektorraum. In diesem Fall ist $s \geq j$.

Wählt man also z_0 als Linearkombination einer maximalen Anzahl linear unabhängiger Eigenvektoren, so sind die ersten j iterierten Vektoren linear unabhängig, es sei denn, Minimalpolynom und charakteristisches Polynom sind identisch und die Eigenwerte nicht paarweise verschieden.

Das nun noch verbliebene Problem ist die Wahl von z_0 für eine gegebene Matrix A, da im Normalfall die Eigenvektoren nicht bekannt sind. Diese Schwierigkeit kann dadurch überwunden werden, daß man die Methode von Krylov mit verschiedenen Vektoren z_0 auf die Matrix A anwendet und dabei die Vektoren z_0 so auswählt, daß mindestens einer unter ihnen eine Linearkombination aller Eigenvektoren x_1 bis x_s ist.

Wählt man eine Basis des zugrunde liegenden Vektorraumes, bestehend aus n Vektoren, sowie einen weiteren Vektor so aus, daß je n dieser $n+1$ Vektoren linear unabhängig sind und der $n+1$-te jeweils eine Linearkombination aller n anderen ist, so besitzt mindestens einer dieser Vektoren die geforderte Eigenschaft. Dies erkennt man durch folgende Überlegung: Stellt man die n Vektoren als Linearkombinationen der Eigenvektoren dar, so wird jeder Eigenvektor mindestens einmal benötigt. Da der $n+1$-te Vektor eine Linearkombination der anderen n ist, ist er also auch eine Linearkombination aller beteiligen Eigenvektoren. Ein Beispiel für $n+1$ Vektoren, die offensichtlich diese Eigenschaft besitzen, sind die Vektoren der kanonische Basis und deren Summe:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Uns interessiert die Berechnung der Determinante mit Hilfe der Methode von Krylov. Zusammengefaßt sieht das Vorgehen dazu folgendermaßen aus:

- Zunächst werden auf die soeben beschriebene Weise $n+1$ für den iterierten Vektor z_0 bestimmt. In der praktischen Anwendung beschränkt man sich in der Regel auf einen

Vektor, da die Anzahl der Fälle, in denen dieser Vektor nicht ausreicht, so gering ist, daß diese Fälle vernachlässigt werden können.

Die folgenden Schritte werden mit jedem Vektor z_0 parallel durchgeführt.

Falls mehrere der parallelen Zweige ein Ergebnis liefern, müssen diese Ergebisse gleich sein. Falls sie nicht gleich sind, wurde die Rechnung nicht korrekt durchgeführt.

Falls keiner der Zweige ein Ergebnis liefert, sind entweder die Eigenwerte der zugrunde liegenden Matrix nicht paarweise verschieden, oder die Matrix ist nicht invertierbar. Diese beiden Unterfälle können mit dem hier dargestellten Algorithmus nicht unterschieden werden.

- Es werden die iterierten Vektoren von z_1 bis z_n berechnet.
- Das Gleichungssystem (6.17) wird dadurch gelöst, daß die aus den iterierten Vektoren bestehende Krylov-Matrix inveritert wird. Ist die Krylov-Matrix nicht invertierbar, so ist der Berechnungsversuch aus den bereits beschriebenen Gründen ein Fehlschlag und wird abgebrochen.
- Die Koeffizienten des charakteristischen Polynoms, und somit auch die Determinante, werden dadurch berechnet, daß die Inverse der Krylov-Matrix mit dem iterierten Vektor z_n multipliziert wird (vgl. (6.17)).

6.4 Vektor- und Matrixnormen

Die Darstellungen der Wahl einer Näherungsinversen in Unterkapitel 6.5 und der iterativen Matrizeninvertierung in Unterkapitel 6.6 benutzen Normen von Matrizen. Deshalb werden im vorliegenden Unterkapitel die Normen eingeführt, die dort zur Beschreibung erforderlich sind. Literatur dazu ist z. B. [GvL83] ab S. 12 oder [IK73] ab S. 3.

Umgangssprachlich formuliert, stellt der Begriff der *Norm* eine Verallgemeinerung des Begriffs der *Länge* dar. Um zu Matrixnormen zu gelangen, klären wir zunächst, was eine Vektornorm ist:

Definition 6.4.1 Eine Funktion

$$f : \mathbb{Q}^n \to \mathbb{Q} \ ,$$

die die Bedingungen

1. $\forall x \in \mathbb{Q} : f(x) \geq 0, f(x) = 0 \Leftrightarrow x = 0_n$
2. $\forall x, y \in \mathbb{Q} : f(x + y) \leq f(x) + f(y)$
3. $\forall a \in \mathbb{Q}, x \in \mathbb{Q}^n : f(ax) = |a| f(x)$

erfüllt, heißt *Norm über* $\mathbb{Q}^n$.

Definition 6.4.2 Sei

$$p \in \mathbb{N}$$

fest gewählt und

$$x \in \mathbb{Q}^n$$

beliebig.

$$||x||_p := (|x_1|^p + \ldots + |x_n|^p)^{1/p}$$

Die so definierte Funktion heißt *p-Norm* .

$$||x||_\infty := \max_{1 \leq i \leq n} |x_i|$$

Diese Funktion wird mit ∞-*Norm* bezeichnet.

Die p-Normen sowie die ∞-Norm werden als *Höldernormen* bezeichnet.

Mit der vorangegangenen Definition werden zwar einige Begriffe angegeben, es ist jedoch nicht selbstverständlich, daß es sich bei den Funktionen auch wirklich um Normen handelt:

Satz 6.4.3 *Die in 6.4.2 definierten Funktionen sind Normen über* $\mathbb{Q}^n$.

Beweis Die Funktionen erfüllen die Bedingungen aus 6.4.1.

Der Beweis dieser Behauptung ist für die 1-Norm, 2-Norm und ∞-Norm in [IK73] ab S. 4 und für die anderen Normen in [Ach67] S. 4-7 angegeben. □

Den Begriff der Norm kann man auch auf Matrizen ausdehnen. Für uns genügt die Betrachtung quadratischer Matrizen.

Definition 6.4.4 Eine Funktion

$$f : \mathbb{Q}^{n^2} \to \mathbb{Q} \ ,$$

die die Bedingungen

$$\begin{aligned} \forall A \in \mathbb{Q}^{n^2} &: \quad f(A) \geq 0, f(A) = 0 \leftrightarrow A = 0_n \\ \forall A, B \in \mathbb{Q}^{n^2} &: \quad f(A+B) \leq f(A) + f(B) \\ \forall c \in \mathbb{Q}, A \in \mathbb{Q}^{n^2} &: \quad f(cA) = |c| f(A) \\ \forall A, B \in \mathbb{Q}^{n^2} &: \quad f(AB) \leq f(A) * f(B) \end{aligned}$$

erfüllt, heißt *Matrixnorm über* $\mathbb{Q}^{n^2}$.

Die vierte der obigen Bedingungen wird in der Literatur nicht immer für Matrixnormen gefordert. In solchen Fällen wird unterschieden zwischen Matrixnormen, die diese Bedingungen erfüllen, und solchen, die diese Bedingung nicht erfüllen (vgl. [IK73] S. 8 und [GvL83] S. 14). Für uns sind diese Unterschiede nicht von Bedeutung.

Die von uns benutzten Matrixnormen sind folgendermaßen definiert:

Definition 6.4.5 Sei

$$A \in \mathbb{Q}^{n^2}$$

Sei

$$x \in \mathbb{Q}^n$$

Es gelte

$$||x|| = 1$$

für eine fest gewählte Vektornorm. Die Funktion

$$||A|| := ||Ax||$$

heißt dann *durch die Vektornorm induzierte Matrixnorm* .

Sie ist in der Literatur auch noch unter den Namen *natürliche Norm* und *Operatornorm* bekannt und wird häufig noch anders definiert (vgl. [IK73] S. 8). Das hat jedoch für unsere Anwendungen keine Bedeutung.

Satz 6.4.6 *Die in 6.4.5 definierte Funktion ist eine Matrixnorm.*

Beweis Die Funktion erfüllt die Bedingungen in 6.4.4 ([IK73] ab S. 8). □

Es folgen Beispiele für induzierte Matrixnormen, die im weiteren Text benutzt werden. Dazu benötigen wir vorher noch einen weiteren Begriff:

Seien $\lambda_1, \ldots, \lambda_n$ die Eigenwerte von A. Dann wird der Spektralradius $\rho(A)$ definiert als

$$\rho(A) := \max\{|\lambda_1|, \ldots, |\lambda_n|\} \ .$$

Durch Indizes wird jeweils kenntlich gemacht, durch welche Vektornorm die jeweilige Matrixnorm induziert wird[11].

$$||A||_1 = \max_j \sum_{k=1}^{n} |a_{k,j}| \tag{6.21}$$

$$||A||_2 = \sqrt{\rho(A * A)} \tag{6.22}$$

$$||A||_\infty = \max_i \sum_{k=1}^{n} |a_{i,k}| \tag{6.23}$$

$$\tag{6.24}$$

Die Beweise der Gleichungen (6.21), (6.22) und (6.23) sind in [IK73] ab S. 9 zu finden.

Falls es nicht im Einzelfall anders festgelegt ist, gilt im weiteren Text $||\ || = ||\ ||_2$.

6.5 Wahl einer Näherungsinversen

In dem in Kapitel 6 vorzustellenden Algorithmus wird die Krylov-Matrix dadurch invertiert, daß eine Näherungsinverse berechnet und diese dann iterativ verbessert wird. In diesem Unterkapitel wird die Wahl der Näherungsinversen beschrieben. Literatur dazu sind [PR85a] und [PR85b].

Für die weiteren Betrachtungen benötigen wir:

$$t := \frac{1}{||A^T A||_1} \tag{6.25}$$

$$B := tA^T \tag{6.26}$$

$$R(B) := E_n - BA \tag{6.27}$$

Im folgenden wird in mehreren Schritten eine Ungleichung für $||R(B)||$ bewiesen, aus der folgt, daß das in Unterkapitel 6.6 beschriebene Verfahren effizient auf B angewendet werden kann, um eine Inverse von A mit zufriedenstellender Näherung zu erhalten.

Definition 6.5.1 Gilt für eine Matrix A

$$A = A^T \ ,$$

dann wird sie als *symmetrisch* bezeichnet.

Da die beiden folgenden Lemmata aus den Grundlagen über Normen stammen, werden die Beweise auf einen Literaturverweis beschränkt. Sie sind in [Atk78] ab S. 416 zu finden.

Lemma 6.5.2 *Für jede symmetrische Matrix A gilt:*

$$||A||_2 = \rho(A) \ .$$

[11] Bei der Vielzahl der in diesem Unterkapitel auftauchenden Normen, sollte man nicht vergessen, daß $|x|$ für einen Skalar x einfach nur den Absolutwert bezeichnet.

Lemma 6.5.3

$$||A^TA||_2 = \rho(A^TA) = ||A||^2 \leq ||A^TA||_1 \leq \max_i \sum_j |a_{i,j}| \max_j \sum_i |a_{i,j}| \leq n||A^TA||$$

Lemma 6.5.4 *Sei λ ein Eigenwert der invertierbaren $n \times n$-Matrix A. Dann ist $1/\lambda$ ein Eigenwert von A^{-1}.*

Beweis Da A invertierbar ist, ist $\lambda \neq 0$. Sei v ein Eigenvektor zu λ. Dann gilt $v \neq 0_n$, sowie

$$\begin{aligned} Av &\neq 0_n \\ Av &= \lambda v \ . \end{aligned}$$

Damit ergibt sich:

$$A^{-1}(Av) = v = (1/\lambda)\lambda v = (1/\lambda)Av \ ,$$

woraus die Behauptung folgt. □

Lemma 6.5.5 *Sei A invertierbar. Sei λ ein Eigenwert von A^TA. Dann gilt:*

$$\frac{1}{||A^{-1}||^2} \leq \lambda \leq ||A||^2 \ . \tag{6.28}$$

Beweis Die rechte Ungleichung von (6.28) folgt aus 6.5.3.

Die linke Ungleichung von (6.28) folgt aus 6.5.4 in Verbindung mit 6.5.3. □

Satz 6.5.6 *Seien t, B und $R(B)$ entsprechend der Gleichungen (6.26), (6.25) und (6.27) definiert. Sei μ ein Eigenwert von $R(B)$. Dann gilt*

$$0 \leq \mu \leq 1 - \frac{1}{||A^TA||_1||A^{-1}||^2} \ .$$

Beweis Sei v Eigenvektor von μ (d. h. v ist nicht der Nullvektor). Dann gilt:

$$R(B)v = \mu v \ .$$

Mit Hilfe von (6.27) und (6.26) erhält man:

$$(E_n - tA^TA)v = v - tA^TAv = \mu v \ .$$

Daraus folgt

$$A^TAv = \lambda v, \ \lambda = \frac{1-\mu}{t}$$

Also ist λ ein Eigenwert von A^TA und mit 6.5.5 folgt

$$\begin{aligned} &\frac{1}{||A^{-1}||^2} \leq \lambda = \frac{1-\mu}{t} \leq ||A||^2 \\ \Leftrightarrow\ &1 - t||A||^2 \leq \mu \leq 1 - \frac{t}{||A^{-1}||^2} \end{aligned}$$

Mit Hilfe von Lemma 6.5.3 und (6.25) ergibt sich die Behauptung. □

Da die Matrix

$$R(B) = E_n - tA^TA$$

symmetrisch ist, folgt aus 6.5.2 und 6.5.6 eine Ungleichung, die es erlaubt, die in (6.26) definierte Matrix B für unsere Zwecke zu verwenden:

Folgerung 6.5.7

$$||R(B)|| \leq 1 - \frac{1}{||A^TA||_1||A^{-1}||^2}$$

Die Benutzung dieser Folgerung wird in Unterkapitel 6.6 beschrieben.

6.6 Iterative Matrizeninvertierung

In diesem Unterkapitel wird beschrieben, wie man eine gegebene Näherungsinverse B einer invertierbaren Matrix A iterativ schrittweise verbessern kann. Diese Methode ist in der Literatur als *Newton-Verfahren* bekannt und wird in [PR85a] und [PR85b] sowie in [Hou64] ab S. 64 behandelt.

Um zu einem Iterationsverfahren zu gelangen, nehmen wir einige Umformungen an einer Gleichung vor, in der das mit (6.27) $R(B)$ vorkommt:

$$\begin{aligned} R(B) &= 0_{n,n} \\ \Leftrightarrow \quad R(B) + E_n &= E_n \\ \Leftrightarrow \quad (R(B) + E_n)B &= B \\ \Leftrightarrow \quad (2E_n - BA)B &= B \end{aligned}$$

Man definiert mit Hilfe der letzten dieser Gleichungen die Iteration

$$\begin{aligned} B_0 &:= B \\ B_i &:= (2E_n - B_{i-1}A)B_{i-1} \ . \end{aligned} \tag{6.29}$$

Betrachtet man nun (6.27) für B_i statt B, bekommt man eine Aussage über die Stärke der Konvergenz der Iteration:

$$\begin{aligned} R(B_i) &= E_n - B_iA \\ &= E_n - (2E_n - B_{i-1}A)B_{i-1}A \\ &= E_n - 2B_{i-1}A + (B_{i-1}A)^2 \\ &= (R(B_{i-1}))^2 \end{aligned}$$

Daraus folgt, daß für jede Matrixnorm gilt:

$$||R(B_i)|| = ||(R(B_{i-1}))^2|| \ .$$

Unter der Voraussetzung

$$||R(B)|| < 1 \tag{6.30}$$

konvergiert $||R(B_i)||$ also quadratisch gegen Null und die B_i entsprechend gegen A^{-1}.

Um das Iterationsverfahren einsetzen zu können, müssen wir noch feststellen, wieviele Iterationen erforderlich sind, um eine bestimmte Genauigkeit zu erreichen. Dazu definieren wir als Abkürzung

$$q := ||R(B)||, \ q_i := ||R(B_i)||, \ q_0 := q \ .$$

Damit das Verfahren überhaupt in praktisch nutzbarer Weise konvergiert, darf q nicht beliebig nahe bei 1 liegen. Sei $c \in \mathbb{N}$ gegeben. Dann nehmen wir an, daß für q gilt:

$$q = 1 - \frac{1}{n^c} \ . \tag{6.31}$$

Da das Verfahren quadratisch konvergiert, gilt für $k \in \mathbb{N}_0$:

$$q_k = q^{2^k} \ .$$

Mit (6.31) erhalten wir:

$$q_k = \left(1 - \frac{1}{n^c}\right)^{2^k} \ .$$

Für unsere Anwendung sei

$$q_k < \frac{1}{d}, \ d \in \mathbb{N}$$

ausreichend. Mit

$$k := e\log(n),\ e \in \mathbb{Q}_{>0}$$

erhalten wir:

$$\left(1 - \frac{1}{n^c}\right)^{n^e} = \frac{1}{d}\ .$$

Da uns die Anzahl der Iterationen interessiert, lösen wir diese Gleichung nach e auf und nehmen dazu $n \to \infty$ an:

$$\begin{aligned}
& \left(1 - \frac{1}{n^c}\right)^{n^e} && = && \frac{1}{d} \\
\Rightarrow\quad & \left(\left(1 - \frac{1}{n^c}\right)^{n^c}\right)^{n^e/n^c} && = && \frac{1}{d} \\
\Rightarrow\quad & \mathrm{e}^{-(n^e/n^c)} && \approx && \frac{1}{d} \\
\Rightarrow\quad & \ln(d) && \approx && n^{e-c} \\
\Rightarrow\quad & \frac{\ln(\ln(d))}{\ln(n)} + c && \approx && e
\end{aligned}$$

Wir können also für wachsende n annehmen, daß gilt

$$c \approx e\ .$$

Um die — bei gegebenem n — von e bestimmte Anzahl der Iterationen zu erhalten, müssen wir c näher bestimmen. Vergleichen wir dazu (6.31) mit der Aussage von 6.5.7. Aus Grundlagen über Normen (z. B. [IK73] ab S. 12) erhalten wir für eine beliebige Matrix W:

$$||W||_2 \leq \sqrt{||W^2||_1}\ .$$

Wir können also die Berechnung von c auf die Bestimmung der 1-Norm, also der maximalen Betragsspaltensumme, einer Matrix zurückführen und erkennen, daß c von der Größe der Matrizenelemente abhängt.

Somit erkennen wir eine weitere Einschränkung für die Anwendbarkeit des Algorithmus. Die Matrix muß die Bedingung

$$||A^T A||_1 ||A^{-1}||^2 \leq n^c \tag{6.32}$$

für ein $c \in \mathbb{N}$ erfüllen. Die Wahl von c hängt davon ab, auf welche Matrizen man das Verfahren anwendet. Um P-Alg. mit den anderen Algorithmen vergleichen zu können, nehmen wir im weiteren Verlauf des Textes $c = 1$ an.

6.7 Determinatenber. mit Hilfe der Methoden von Krylov und Newton

In diesem Unterkapitel werden die in den vorangegangenen Abschnitten dargestellten Methoden zu einem Algorithmus zur Determinantenberechnung zusammengefaßt, auf den mit *P-Alg.* Bezug genommen wird[12]. Weiterhin wird die Anzahl der Schritte und der Prozessoren analysiert.

Im folgenden ist mit n^x jeweils $\lceil n^x \rceil$ gemeint.

Im folgenden wird mit Hinweis auf die Bemerkungen auf S. 15 für die Multplikation zweier $n \times n$-Matrizen ein Aufwand von

$$\gamma_S(\lceil \log(n) \rceil + 1)$$

Schritten und

$$\gamma_P n^{2+\gamma}$$

Prozessoren in Rechnung gestellt.

[12] vgl. Unterkapitel 1.3

Es sei die Determinante der $n \times n$-Matrix A zu berechnen. Zuerst wird die Krylov-Matrix entsprechend der Darstellung in Abschnitt 6.3 berechnet. Es ist üblich, für den dort erwähnten Vektor z zur Berechnung der iterierten Vektoren den Einheitsvektor zu verwenden, dessen sämtliche Elemente gleich 1 sind. Aufgrund der aus theoretischer Sicht ohnehin bereits eingeschränkten Verwendbarkeit der Methode von Krylov für unsere Zwecke, bedeutet die Beschränkung auf diesen Vektor nur eine unbedeutende Verschlechterung des Algorithmus.

Zuerst sind die iterierten Vektoren anhand des Vektors z und der Matrix A zu berechnen. Dies geschieht anhand der nachstehenden Gleichungen ([Pan85], [BM75] S. 128, [Kel82]). Auf den rechten Seiten werden nur Ergebnisse vorangegangener Gleichungen oder z bzw. A verwendet. Auf den linken Seiten stehen neu berechnete Ergebnisse. Die Terme auf den rechten Seiten beschreiben Matrizenmultiplikationen und die linken Seiten deren Ergebnisse.

$$\begin{aligned} [A^3v,\, A^2v] &= A^2[Av,\, v] \\ [A^7v,\, A^6v,\, A^5v,\, A^4v] &= A^4[A^3v,\, A^2v,\, Av,\, v] \\ &\vdots \\ [A^{2*2^h}v,\, \ldots,\, A^{2^h}v] &= A^{2^h}[A^{2^h-1}v,\, \ldots,\, v] \end{aligned}$$

Um die jeweils nächste Gleichung mit Hilfe der schon bekannten Ergebnisse aufzustellen, sind zwei Matrizenmultiplikationen erforderlich. Mit der ersten wird die nächste benötigte Potenz von A berechnet. Mit der zweiten wird der jeweilige Term der rechten Seite der Gleichung ausgewertet. Die n gesuchten iterierten Vektoren können auf diese Weise in

$$\gamma_S(\lceil \log(n) \rceil + 1)\, \lceil \log(n) \rceil$$

Schritten von

$$\gamma_P n^{2+\gamma}$$

Prozessoren berechnet werden.

Anschließend ist für die aus iterierten Vektoren bestehende Krylov-Matrix eine Näherungsinverse entsprechend der Ausführungen in Unterkapitel 6.5 zu berechnen. Dazu werden die Gleichungen (6.26) und (6.25) verwendet. Betrachtet man diese Gleichungen, erkennt man, daß zur Berechnung der dortigen Matrix B aus der Matrix A eine Matrizenmultiplikation, eine Berechnung der 1-Norm einer Matrix, eine Division durch einen Skalar sowie eine Matrix–Skalar–Multiplikation erforderlich ist. An dieser Stelle wird deutlich, daß der Algorithmus nicht ohne Divisionen auskommt.

Um die 1-Norm zu erhalten, sind n Summen von je n-Matrizenelementen zu berechnen und miteinander zu vergleichen. Dies kann mit Hilfe der Binärbaummethode nach Satz 1.5.1 in

$$2\, \lceil \log(n) \rceil$$

Schritten von

$$n \left\lfloor \frac{n}{2} \right\rfloor$$

Prozessoren durchgeführt werden.

Insgesamt kann die Berechnung der Näherungsinversen also in

$$\gamma_S(\lceil \log(n) \rceil + 1) + 2\, \lceil \log(n) \rceil + 2$$

Schritten von

$$\gamma_P n^{2+\gamma}$$

Prozessoren geleistet werden.

Nachdem sie zur Verfügung steht, kann sie mit Hilfe des in Unterkapitel 6.6 beschriebenen Verfahrens verbessert werden.

Ein Iterationsschritt mit Hilfe von Gleichung (6.29) erfordert zwei Multiplikationen und eine Addtition von Matrizen. Setzt man für die Anzahl der durchzuführenden Iterationen zunächst die Unbestimmte I ein, kann das Iterationsverfahren in

$$I * 2\gamma_S(\lceil \log(n) \rceil + 1) + 1$$

Schritten von

$$\gamma_P n^{2+\gamma}$$

Prozessoren auf die Näherungsinverse angewendet werden um diese bis auf eine ausreichende Genauigkeit an die gesuchte Inverse anzunähern.

Schließlich ist noch eine Matrix–Vektor–Multiplikation durchzuführen, um die Methode von Krylov zur Berechnung der Koeffizienten des charakteristischen Polynoms zu vollenden. Betrachtet man den Vektor wiederum als Matrix, kann dies in

$$\gamma_S(\lceil \log(n) \rceil + 1)$$

Schritten von

$$\gamma_P n^{2+\gamma}$$

Prozessoren erledigt werden. Nach 2.4.4 hat man mit den Koeffizienten auch die Determinante berechnet.

Hier wird eine Einschränkung für den Algorithmus deutlich. Da ein Näherungsverfahren verwendet wird, ist es nicht möglich, die Determinante für Matrizen mit Elementen aus $\mathbb{Q}$ genau zu berechnen. Diese müssen deshalb aus $\mathbb{Z}$ stammen, denn in diesem Fall sind die Koeffizienten des charakteristischen Polynoms ebenfalls ganzzahlig und die Determinante kann bei genügend genau durchgeführtem Newton-Verfahren durch Rundung genau ermittelt werden.

Betrachtet man die Einschränkungen für die Verwendbarkeit der Methode von Krylov (vgl. Unterkaptel 6.3) sowie die Bedingung (6.32) auf S. 99, erkennt man, daß insgesamt nicht unerhebliche Anforderungen an die Matrix, deren Determianten zu berechnen ist, gestellt werden müssen, damit der in diesem Kapitel dargestellte Algorithmus anwendbar ist.

Als Gesamtaufwand für den Algorithmus erhält man für die Anzahl der Schritte:

$$\begin{aligned}
& \gamma_S(\lceil \log(n) \rceil + 1) \lceil \log(n) \rceil \\
+ \quad & \gamma_S(\lceil \log(n) \rceil + 1) + 2 \lceil \log(n) \rceil + 2 \\
+ \quad & I * 2\gamma_S(\lceil \log(n) \rceil + 1) + 1 \\
+ \quad & \gamma_S(\lceil \log(n) \rceil + 1) \\
= \quad & \gamma_S \left(\lceil \log(n) \rceil^2 + 5 \lceil \log(n) \rceil + I \left(2 \lceil \log(n) \rceil + 2 \right) + \frac{3}{\gamma_S} + 2 \right)
\end{aligned}$$

Mit Verweis auf Unterkapitel 6.6 nehmen wir $I = \log(n)$ an. Dadurch lautet der Term für die Anzahl der Schritte:

$$\gamma_S \left(3 \lceil \log(n) \rceil^2 + 7 \lceil \log(n) \rceil + \frac{3}{\gamma_S} + 2 \right) \quad .$$

Vergleicht man die Annahme für I mit den Ausführungen in Unterkapitel 6.6, erkennt man, daß dies der beste mit dem Algorithmus zu erreichende Wert ist. Bei schlechteren Randbedingungen erhält man für die Schritte einen entsprechend schlechteren Wert.

Für die Prozessoren ergibt sich:

$$\gamma_P n^{2+\gamma} \quad .$$

Da die Matrizenmultiplikation nach [CW90] nur für sehr große n Vorteile bringt, wird

$$\gamma_P = \gamma_S = \gamma = 1$$

gesetzt, um eine für die Anwendung des Algorithmus realistische Vergleichsmöglichkeit mit den anderen Algorithmen zu bekommen.

Vergleicht man P-Alg. mit C-Alg., BGH-Alg. und B-Alg., so erkennt man, daß die Anzahl der Prozessoren in P-Alg. gleich ist mit der Anzahl der Prozessoren für die Matrizenmultiplikation ist und somit um eine Potenz geringer als beim besten der drei anderen Algorithmen.

Ein gravierender Nachteil von P-Alg. sind die Einschränkungen für die Benutzbarkeit. Die Bedingung, daß die Matrizenelemente ganzzahlig sein müssen, läßt sich durch Ausnutzung der Eigenschaften einer Determinaten (siehe Definition 2.1.6) ausgleichen. Dazu werden die Matrizenelemente durch einen geeigneten Faktor multipliziert, so daß die resultierende Matrix nur

ganzzahlige Elemente enthält. Nachdem der Algorithmus auf diese Matrix angewendet worden ist, kann man dann aus dem Ergebnis auf die Determinate der ursprünglichen Matrix schließen.

Durch die in Unterkapitel 6.3 beschriebenen Einschränkungen, kann für P-Alg. nicht garantiert werden, daß er für jede invertierbare Matrix eine Determinante ungleich Null liefert. Die Wahrscheinlichkeit für einen solchen Fall ist zwar gering, jedoch unterscheidet diese Beschränkung P-Alg. von den anderen drei Algorithmen.

Weiterhin muß man bei P-Alg. natürlich wiederum auf die Existenz von Divisionen und das Fehlen von Fallunterscheidungen hinweisen.

Kapitel 7

Implementierung

In diesem Kapitel wird die Implementierung der in den Kapitel 3 bis 6 behandelten Algorithmen beschrieben[1].

7.1 Erfüllte Anforderungen

In diesem Unterkapitel werden die wesentlichen Eigenschaften des implementierten Programms beschrieben um einen Überblick über dessen Leistungsmerkmale zu geben.

Die Algorithmen sind in Modula-2 auf einem Rechner mit einem Prozessor implementiert[2]. Als Literatur über die Programmiersprache Modula-2 ist z. B. [CLR86] empfehlenswert.

Alle Algorithmusteile, die parallel ausgeführt werden sollen, werden mit Hilfe von Schleifen nacheinander ausgeführt. Mit Hilfe von Zählprozeduren (siehe Modul 'Pram') wird während der Programmausführung ermittelt, in wieviel Schritten und mit Hilfe von wieviel Prozessoren der Algorithmus durch eine PRAM abgearbeitet werden kann.

Das Programm ermöglicht es, Matrizen anhand von Parametervorgaben zufällig zu erzeugen. Die Parameter sind

- Größe der Matrix,
- Rang (um auch Matrizen mit der Determinante Null gezielt erzeugen zu können),
- Wahl einer der Mengen[3] $\mathbb{N}$, $\mathbb{Z}, \mathbb{Q}$ oder $\mathbb{Q}^+$ [4] für die Matrizenelemente, da Algorithmen evtl. nicht auf jede dieser Mengen anwendbar sind, und
- Vielfachheiten der Eigenwerte (wichtig für den Algorithmus von Pan).

Für erzeugte Matrizen kann jeder der implementierten Algorithmen aufgerufen werden. Eine Matrix wird zusammen mit den durch die verschiedenen Algorithmen berechneten Determinanten und den Meßergebnissen der Zählprozeduren unter einem Namen auf dem Hintergrundspeicher abgelegt. Diese Verwaltung geschieht automatisch. Dazu muß der Benutzer vor jedem neuen Anlegen eines aus den obigen Daten bestehenden Datensatzes einen Namen für diesen Datensatz angeben.

Das Programm ist kommandoorientiert. D. h. nach dem Start wird der Benutzer aufgefordert, Befehle einzugeben, die nacheinander ausgeführt werden.

Die Lesbarkeit des Quelltextes wird durch häufige Kommentare gesteigert.

[1] Der Algorithmus nach dem Entwicklungssatz von Laplace (siehe Unterkapitel 3.3) wird hier nicht berücksichtigt.

[2] Megamax Modula-2 auf einem ATARI ST

[3] durch geeignete Parameterwahl mit dem Befehl 'param'

[4] alle Elemente aus $\mathbb{Q}$, die größer oder gleich Null sind

7.2 Bedienung des Programms

In diesem Unterkapitel wird die Benutzung des Programms beschrieben.

Nach dem Programmstart muß zunächst mit Hilfe des Befehls *find* ein Datensatz festgelegt werden, der bearbeitet werden soll. Dies kann ein bereits existierender oder ein neu anzulegender sein.

Bei einem neu angelegten Datensatz müssen danach mit Hilfe des Befehls *param* die Parameter für die zu erzeugende Matrix festgelegt werden.

Anschließend kann man mit *gen* eine neue Matrix erzeugen lassen. Nachdem mit *param* die Parameter festgelegt sind, kann mit mit *gen* zu jeder Zeit eine neue Matrix generieren lassen, wodurch jedoch die vorangegangene Matrix verloren geht.

Wenn eine Matrix erzeugt worden ist, kann man mit Hilfe der Befehle *berk, bgh, csanky* und *pan* die entsprechenden Algorithmen auf die generierte Matrix anwenden.

Mit *show* kann man sich zu jeder Zeit den aktuellen mit *find* bestimmten Datensatz auf dem Bildschirm anzeigen lassen. Die generierte Matrix wird nur ausgegeben, wenn eine Matrixzeile in eine Bildschirmzeile paßt. Größere Matrizen kann man mit *mshow* trotzdem ausgeben lassen.

Falls man einen weiteren Datensatz anlegen oder einen bereits vorhandenen wieder bearbeiten will, benutzt man erneut den Befehl *find*. Die Speicherung des alten Datensatzes geschieht automatisch.

Weitere Befehle, die man nach dem Programmstart zu jedem Zeitpunkt angeben kann, sind: *del, exit, h, help, hilfe, ?, ls* und *q* . Ihre Bedeutung ist in der folgenden Liste aller erlaubten Befehle erklärt:

berk
: Der Algorithmus von Berkowitz aus Kapitel 5 wird auf die Matrix des aktuellen Datensatzes angewendet. Die berechnete Determinante sowie die Ergebisse der Zählprozeduren (siehe Modul 'Pram') werden im aktuellen Datensatz abgelegt.

bgh
: Die Wirkung dieses Befehls ist analog zu der des Befehls *berk*, jedoch bezogen auf den Algorithmus von Borodin, von zur Gathen und Hopcroft aus Kapitel 4.

csanky
: Die Wirkung dieses Befehls ist analog zu der des Befehls *berk*, jedoch bezogen auf den Algorithmus von Csanky aus Unterkapitel 3.8.

del
: Es wird nach einem Datensatznamen gefragt. Der zugehörige Datensatz wird im Hauptspeicher und auf dem Hintergrundspeicher gelöscht.

exit
: Das Programm wird beendet. Der aktuelle mit *find* festgelegte Datensatz wird automatisch auf dem Hintergrundspeicher abgelegt.

find
: Es wird nach einem Datensatznamen gefragt. Falls ein Datensatz mit diesem Namen bereits im Hauptspeicher abgelegt ist, wird dieser erneut zum aktuellen Datensatz. Falls dies nicht der Fall ist und auf dem Hintergrundspeicher ein Datensatz mit diesem Namen abgelegt ist, wird dieser in den Hauptspeicher geladen und zum aktuellen Datensatz. Falls beide Fälle nicht zutreffen, wird ein neuer Datensatz mit dem angegebenen Namen im Hauptspeicher angelegt und als aktueller Datensatz betrachtet.

gen
: Entsprechend der mit *param* festgelegten Parameter wird eine neue Matrix für den aktuellen Datensatz generiert. Die dort durch die Befehle *berk, bgh, csanky* und *pan* abgelegten Daten werden gelöscht.

h, help, hilfe, ?
: Durch diese Befehle wird eine Kurzbeschreibung aller erlaubten Befehle auf den Bildschirm ausgegeben.

ls
: Auf dem Bildschirm wird eine Liste der Namen der im Hauptspeicher befindlichen Datensätze ausgegeben.

mshow
: Die Matrix des aktuellen Datensatzes wird auf dem Bildschirm ausgegeben.

pan
: Die Wirkung dieses Befehls ist analog zu der des Befehls *berk*, jedoch bezogen auf den Algorithmus von Pan aus Kapitel 6.

param
: Es wird nach den Parametern für die mit Hilfe von *gen* zu generierende Matrix gefragt. Alle zuvor im aktuellen Datensatz abgelegten Daten werden gelöscht.

q
: Die Wirkung dieses Befehls ist mit der des Befehls *exit* identisch.

show
: Der aktuelle Datensatz wird auf dem Bildschirm ausgegeben. Die Matrix wird nur ausgegeben, falls eine Matrixzeile in eine Bildschirmzeile paßt. Mit dem Befehl *mshow* kann die Matrix dennoch ausgegeben werden.

7.3 Die Modulstruktur

In diesem Unterkapitel wird die Struktur des implementierten Programms beschrieben. Dazu wird zu jedem Modul dessen Aufgabe und evtl. dessen Beziehung zu anderen Modulen angegeben. Alle Beschreibungen von Details der Implementierung, die für die Benutzung des Programms und für das Verständnis von dessen Gesamtstruktur unwichtig sind, erfolgen durch Kommentare innerhalb des Quelltextes (siehe Anhang).

Eine Beschreibung des Programmoduls *main* entspricht einer Erklärung der Benutzung des Programms. Diese ist in Unterkapitel 7.2 zu finden.

Die Beschreibung der Programmodule zum Test einzelner Teile des Gesamtprogramms ist von untergeordnetem Interesse und beschränkt sich deshalb auf kurze Bemerkungen über ihren Zweck ihm Rahmen der alphabetischen Auflistung (s. u.).

Die vorrangige Aufmerksamkeit des an der Implementierung der Algorithmen Interessierten sollte sich auf die Module *Det* und *Pram* sowie auf das Programmodul *algtest* richten.

Die genannten vorrangig interessanten Module sind im Anhang A zusammen mit dem Modul *main* gesammelt. Die weniger interessanten Testprogrammodule sind im Anhang C aufgeführt. Alle weiteren Module sind in alphabetischer Reihenfolge in Anhang B zu finden.

Zunächst wird anhand eines *reduzierten Ebenenstrukturbildes* (Erklärung s. u.) ein Überblick über die Programmstruktur geben. Anschließend erfolgt eine alphabetische Auflistung der Module und ihrer Erklärungen.

In das erwähnte Strukturbild sind alle Module nach folgenden Regeln eingetragen:

- Die Eintragung erfolgt ebenenweise. Die niedrigste Ebene ist im Bild unten zu finden und die höchste oben. Jedes Modul gehört genau einer Ebene an.
- Jedes Modul wird im Rahmen der Maßgaben durch die anderen Regeln in einer möglichst niedrigen Ebene eingetragen.
- Von jedem Modul A aus, das ein Modul B benutzt, z. B. durch Aufruf von Prozeduren des Moduls B, wird im Rahmen der Einschränkungen durch andere Regeln ein Pfeil auf dieses Modul *B* gerichtet.
- Jedes Modul wird so eingetragen, daß kein Pfeil von ihm auf ein Modul auf der gleichen oder einer höheren Ebene gerichtet ist.
- Alle Pfeile von einem Modul aus auf Module, die nicht genau eine Ebene tiefer angeordnet sind, werden weggelassen.

Durch die letzte Regel gewinnt das Strukturbild erheblich an Übersichtlichkeit, ohne wesentlichen Informationsgehalt zu verlieren. Aus dem Quelltext jedes Moduls ist zu entnehmen, welche Module, außer den im Bild angegebenen, sonst noch benutzt werden. Die aufgeführten Regeln liefern das in Abbildung 7.1 angegebene Bild.

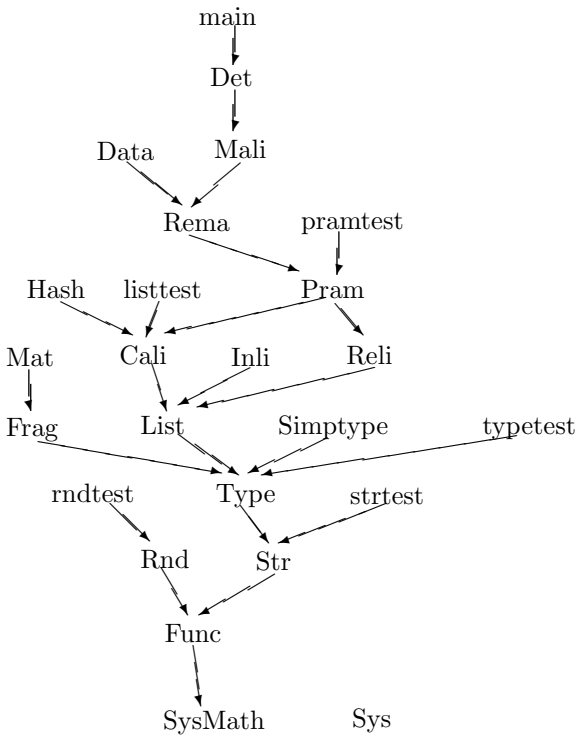

Abbildung 7.1: reduziertes Ebenenstrukturbild

Es folgen die Kurzbeschreibungen der einzelnen Module in alphabetischer Reihenfolge.

algtest (Programmodul)
Dieses Modul dient zum Test der Algorithmen zur Determinantenberechnung ohne Behinderung durch Anforderungen irgendwelcher Art, insbesondere ohne Beachtung der Parallelisierung und der Maßgabe, Matrizen beliebiger Größe zu verarbeiten.

Cali (CArdinal LIst)
In diesem Modul sind lineare Listen positiver ganzer Zahlen implementiert. Es stützt sich auf das Modul 'List'.

Data
Das Modul 'Data' dient der Verwaltung der Datensätze bestehend aus Matrizen und ihren Parametern, sowie der berechneten Determinanten und der dabei gezählten Schritte und Prozessoren.

Det
In diesem Modul sind die Algorithmen zur parallelen Determinantenberechnung implementiert.

Frag (array FRAGments)
Im Modul 'Frag' sind Felder beliebiger variabler Länge und beliebigen Inhalts implementiert. Das Modul ist erforderlich, um Matrizen verarbeiten zu können, deren Größe durch den Benutzer erst während der Laufzeit des Programms festgelegt wird.

Das Modul profitiert von der Verwaltung von Elementen beliebiger Typen durch das Modul 'Type'.

Func (FUNCtions)
In diesem Modul sind verschiedene Prozeduren und Funktionen insbesondere für mathematische Zwecke zusammengefaßt.

Hash
Durch dieses Modul werden Prozeduren zur Streuspeicherung, auch unter dem Namen 'Hashing' bekannt, zur Verfügung gestellt. Das Modul wird durch den Algorithmus von Borodin, von zur Gathen und Hopcroft im Modul 'Det' benötigt, um Zwischenergebnisse bei der parallelen Berechnung von Termen zu speichern.

Das Modul 'Hash' erlaubt es, beliebige Daten zu speichern. Dabei wird auf das Modul 'Type' zur Verwaltung von Elementen beliebiger Typen zurückgegriffen.

Inli (INteger LIst)
In diesem Modul sind lineare Listen ganzer Zahlen implementiert. Es stützt sich auf das Modul 'List'.

List
Das Modul 'List' stellt Prozeduren zur Verwaltung von linearen doppelt verketteten Listen bliebiger Elemente zur Verfügung. Analog zu den Modulen 'Frag', 'Hash' und 'Mat' benutzt 'List' das Modul 'Type' zur Verwaltung von Elementen beliebiger Typen.

Auf dem Modul 'List' bauen verschiedene Module zur Implementierung von Listen spezieller Typen auf.

listtest (Programmodul)
Dieses Programmodul dient zum Test des Moduls 'List'. Es verwendet dazu das Modul

'Cali'.

main (Programmodul)
Dies ist das Hauptmodul des gesamten Programms. Es nimmt die Befehle des Benutzers entgegen und ruft die entsprechenden Prozeduren auf. Die Benutzung ist in Unterkapitel 7.2 beschrieben.

Mali (MAtrix LIst)
In diesem Modul sind lineare Listen von Matrizen implementiert. Es stützt sich auf die Module 'List' und 'Mat'.

Mat (MATrix)
Dieses Modul stellt Prozeduren zur Verwaltung von zweidimensionalen Matrizen beliebiger Größe für beliebige Elemente zur Verfügung. Es stützt sich auf das Modul 'Frag' zur Verwaltung der Felder beliebiger Größe und auf das Modul 'Type' zur Verwaltung von Elementen beliebiger Typen.

Pram
Das Modul 'Pram' stellt die Zählprozeduren zur Verfügung, die zur Ermittlung des Aufwandes für eine PRAM zur Abarbeitung der verschiedenen Algorithmen zur Determiantenberechnung erforderlich sind. Das Modul wird durch die Module 'Det' und 'Rema' benutzt und verwendet seinerseits insbesondere das Modul 'Cali' für Verwaltungsaufgaben.

pramtest (Programmodul)
Dieses Programmodul dient zum Test des Modul 'Pram'.

Reli (REal LIst)
In diesem Modul sind lineare Listen von Fließkommazahlen implementiert. Es stützt sich auf das Modul 'List'.

Rema (REal MAtrix)
Dieses Modul implementiert Matrizen aus Fließkommazahlen. Es stützt sich dazu auf das Modul 'Mat'.

Rnd (RaNDomize)
Das Modul 'Rnd' erlaubt es, Zufallszahlen nach der linearen Kongruenzmethode zu erzeugen. Es wird vom Modul 'Rema' dazu benutzt, anhand von verschiedenen Parametern zufällige Matrizen zu generieren.

rndtest (Programmodul)
Dieses Programmodul dient zum Test des Moduls 'Rnd'.

simptype (SIMPle TYPE)
Dieses Modul stellt Verwaltungsprozeduren für die einfachen Datentypen
[3] 'LONGCARD', 'LONGINT' und 'LONGREAL' zur Verfügung, damit sie in Verbindung mit dem Modul 'Type' verwendet werden können.

Str (STRing)
Im Modul 'Str' sind diverse Prozeduren zur Verarbeitung von Zeichenketten implementiert.

strtest (Programmodul)
Dieses Programmodul dient zum Test des Moduls 'Str'.

Sys
Dieses Modul stellt Prozeduren zum Ablegen von Daten auf dem Hintergrundspeicher zur Verfügung. Da die Behandlung der Massenspeicher auf den verschiendenen Rechnersystemen unterschiedlich ist, muß das Modul 'Sys' bei der Portierung des Programms auf einen anderen Rechner neu implementiert werden.

SysMath
Die zur Verfügung gestellten mathematischen Funktionen sind von System zu System unterschiedlich. Deshalb sind im Modul 'SysMath' die Funktionen gesammelt, die im Programm benutzt werden. Bei der Portierung des Programms auf ein anderes Computersystem muß dieses Modul evtl. angepaßt werden.

Type
Dieses Modul dient der Verwaltung von Elementen beliebiger Datentypen. Ein neuer Typ wird im Rahmen dieses Moduls durch die Angabe verschiedener Verwaltungsprozeduren definiert. Das Modul übernimmt auf diese Weise die Sammlung der Eigenschaften verschiedener Typen, um so die Übersichtlichkeit zu steigern. Ohne dieses Modul muß jedes der Module 'Frag', 'Hash', 'List' und 'Mat' eine entsprechende Verwaltung separat enthalten.

typetest (Programmodul)
Dieses Programmodul dient zum Test des Moduls 'Type'.

7.4 Anmerkungen zur Implementierung

An dieser Stelle werden einige praktische Gesichtspunkte der Implementierung kommentiert.

Vergleicht man das Modul 'algtest' mit dem Rest des Quelltextes, so erkennt man, daß insbesondere die Anforderung der *Pseudoparallelisierung* die Länge des Quelltextes stark vergrößert. Bei den Algorithmen im Modul 'Det' handelt es sich ungefähr um eine Vergrößerung um den Faktor 5.

Weiterhin zeigt sich, daß eine flexible auch nachträglich erweiterbare Programmstruktur, die unter dem Gesichtspunkt sich evtl. anschließender Arbeiten wünschenswert ist, nicht unerheblichen Aufwand bedeutet. So machen die eigentlich interessierenden Programmteile nur ca. 30 Prozent des Quelltextes[5] aus. Der gesamte weitere Aufwand ergibt sich einerseits aus verschiedenen Anforderungen an Leistungen und Struktur des Programms, andererseits aus der Notwendigkeit, Datentypen zu implementieren, die die verwendete Sprache nicht standardmäßig zur Verfügung stellt.

Auf eine Implementierung auf leistungsfähigeren Rechnern wurde verzichtet, da der benötigte Speicherplatz quadratisch mit der Anzahl der Zeilen und Spalten der Matrizen wächst. Der zusätzliche Speicherplatz führt nicht zu einer Steigerung der Matrizengröße von weitreichendem Interesse. Diese Beschränkung erlaubt es, mit der relativ geringen Rechenleistung eine ATARI ST auszukommen.

Größere Matrizen sind auch aus einem weiteren Grund nicht ohne erheblichen weiteren Aufwand sinnvoll. Die Standardarithmetiken verschiedener Implementierungen von Programmiersprachen erlauben eine Rechengenauigkeit von typischerweise ca. 19 Stellen. Dies reicht nicht aus, um Determinanten größerer Matrizen überhaupt darzustellen. Deshalb ist es für eine deutliche

[5] Gesamtlänge ca. 9500 Zeilen

Steigerung der Matrizengröße erforderlich, eine eigene Fließkommaarithmetik zu implementieren, die es ermöglicht, mit beliebiger Genauigkeit[6] zu rechnen.

Die praktischen Erfahrungen in verschiedenen Bereichen der angewandten Informatik zeigen, daß in bestehenden in der Regel zufriedenstellend laufenden Programmen eine Restquote an Programmierfehlern im Quelltext von ca. einem Fehler pro 1000 Zeilen Quelltext existiert. Beim gegenwärtigen Stand der Technik ist es nicht möglich, Programme wesentlich fehlerfreier zu bekommen.

Besonders bei mathematischen Programmen ist es in der Regel erforderlich, trotzdem nahezu Fehlerfreiheit zu erreichen, was die Implementierung solcher Programme zusätzlich erschwert. Ein praktisches Beispiel für diese Probleme sind die implementierten Algorithmen zur Determiantenberechnung. Die pseudoparallelen Algorithmen im Modul 'Det' besitzen eine Gesamtlänge von ca. 2300 Zeilen, erheblich mehr als die Implementierungen im Programmodul 'algtest'.

Da die Dauer einer Fehlersuche schwer abzuschätzen ist und nur begrenzte Zeit zur Verfügung stand, haben die genannten Schwierigkeiten dazu geführt, daß zwar alle Algorithmen lauffähig sind, jedoch die im Anhang zu findenden Implementierungen leider keine Determinanten berechnen:

- P-Alg. im Programmodul 'algtest' sowie
- BGH-Alg., B-Alg. und P-Alg. im Modul 'Det'.

Durch umfangreiche Testläufe kann ausgeschlossen werden, daß die Fehler außerhalb der Module zu suchen sind. Es muß sich jeweils um fehlerhafte Implementierung der Algorithmusbeschreibungen in den jeweiligen Kapiteln handeln (z. B. Vorzeichenfehler oder falsche Indizes).

[6] im Rahmen der physikalischen Grenzen

Kapitel 8

Nachbetrachtungen

In diesem Kapitel werden die Ergebnisse dieser Arbeit abschließend betrachtet und bewertet.

8.1 Vergleich der Algorithmen

In diesem Unterkapitel werden die vier hauptsächlich in dieser Arbeit behandelten Algorithmen zusammenfassend miteinander verglichen.

Auf die Algorithmen wird, wie in Unterkapitel 1.3 definiert wird, mit Hilfe der Anfangsbuchstaben ihrer Autoren Bezug genommen: C-Alg., BGH-Alg., B-Alg. und P-Alg..

Der Vergleich wird nach folgenden Kriterien durchgeführt:

- Werden Divisionen benutzt?
- Werden Fallunterscheidungen durchgeführt?
- Wie groß ist die Anzahl der Schritte?
- Wie groß ist die Anzahl der Prozessoren?
- Welche Einschränkungen für die Anwendbarkeit gibt es?
- Wie aufwendig ist die Implementierung?
- Welche Ähnlichkeiten in der Methodik gibt es?

Im Anschluß an diesen Vergleich wird eine Bewertung der Algorithmen anhand der genannten Gesichtspunkte vorgenommen.

Zunächst sei angemerkt, daß keiner der Algorithmen Fallunterscheidungen verwendet. Dies vereinfacht eine Betrachtung aus der Sicht des Schaltkreisentwurfs.

Es kommen lediglich B-Alg. und BGH-Alg. ohne Divisionen aus und können somit in beliebigen Ringen angewendet werden. C-Alg. und P-Alg. können nur in Körpern verwendet werden, falls man exakte Ergebnisse verlangt

P-Alg. darf nur auf ganzzahlige Matrizen angewendet werden. Da P-Alg. Divisionen verwendet, ist es jedoch auch dann nicht gewährleistet, daß er die Determinante exakt liefert. Hinzu kommt bei P-Alg. als erheblicher Nachteil die eingeschränkte Verwendbarkeit, auch wenn sich dies in der praktischen Anwendung nicht sehr stark auswirkt. Diese Eigenschaft von P-Alg. ist mit dem Laufzeitverhalten von 'Quicksort' vergleichbar, das im *Average Case* sehr gut, jedoch im *Worst Case* sehr schlecht ist. P-Alg. ist theoretisch nur eingeschränkt verwendbar, praktisch jedoch nahezu uneingeschränkt, da zufällig zusammengestellte Matrizen mit sehr hoher Wahrscheinlichkeit invertierbar sind und die Bedingungen für P-Alg. erfüllen.

Betrachtet man die Anzahl der Schritte, stellt man fest, daß C-Alg. am schnellsten ist, gefolgt von B-Alg. und P-Alg.. Das Schlußlicht bildet mit einigem Abstand BGH-Alg.

Bezüglich der Anzahl der Prozessoren hat P-Alg. die Nase vorn, gefolgt von C-Alg. und B-Alg.. Das Schlußlicht bildet BGH-Alg. wiederum mit Abstand. Zu beachten ist der sehr gute Wert für die Prozessoren bei P-Alg., der sich vermutlich kaum weiter verbessern läßt.

Auf einem Rechner mit einem Prozessor ist P-Alg. der effizienteste, da er insgesamt die wenigsten Operationen benötigt. Er wird gefolgt von C-Alg. B-Alg. liegt hier an dritter Stelle gefolgt von BGH-Alg..

Betrachtet man den Aufwand für die Implementierung gemessen in Anzahl der Quelltextzeilen[1], stellt man fest, daß C-Alg. bei weitem am einfachsten zu implementieren ist. Ihm folgt P-Alg. dichtauf. Die Implementierung von B-Alg. ist bereits etwas aufwendiger, aber noch zumutbar. BGH-Alg. bildet auch in diesem Punkt mit relativ großem Abstand das Schlußlicht.

Bei der Betrachtung der Verfahren, die die einzelnen Algorithmen verwenden, erkennt man Parallelen zwischen C-Alg. und B-Alg. sowie zwischen BGH-Alg. und P-Alg.. Sowohl in C-Alg. als auch in B-Alg. wird jeweils ein Satz verwendet, der zum Zeitpunkt der Veröffentlichung der Algorithmen bereits seit mehreren Jahrzehnte bekannt war. Die beiden Sätze sind durch diverse Umformungen für die parallele Determinanteberechnung nutzbar gemacht worden. In BGH-Alg. und P-Alg. hingegen werden jeweils mehrere auch separat bedeutsame Verfahren zu einem Algorithmus in Verbindung gebracht.

Unter dem Gesichtspunkt, daß BGH-Alg. nicht der einzige divisionsfreie Algorithmus mit polynomiellem Aufwand ist, besitzt er wegen des erforderlichen Implementierungsaufwandes und seiner relativen Langsamkeit nur theoretisches Interesse.

C-Alg. ist am brauchbarsten für schnelle Erstellung einer Implementierung, falls das Vorhandensein von Divisionen nicht weiter stört.

P-Alg. sollte verwendet werden, falls es im wesentlichen auf Geschwindigkeit ankommt und die Einschränkungen für die Anwendbarkeit sowie die Existenz von Divisionen nicht stören.

B-Alg. ist der Algorithmus unter den vieren mit den ausgewogensten Leistungsmerkmalen und geht insgesamt als Sieger aus dem Vergleich hervor. Er besitzt eine ausreichende Effizienz, kommt ohne Divisionen aus und unterliegt keinen sonstigen Einschränkungen, wie z. B. P-Alg.. Im Zweifelsfall sollte immer B-Alg. verwendet werden.

Um zum Abschluß einen Eindruck von der Effizienz der Algorithmen im direkten Vergleich zu liefern, sind für Tabelle 8.1 die aus den Aufwandsanalysen hervorgegangene Terme beispielhaft ausgewertet worden. In der Tabelle werden bei den Anzahlen der Prozessoren die guten Werte für P-Alg. und die auffällig schlechten Werte für BGH-Alg. besonders deutlich.

n	C-Alg.		BGH-Alg.		B-Alg.		P-Alg.	
	Schr.	Proz.	Schr.	Proz.	Schr.	Proz.	Schr.	Proz.
2	9	8	16	24	12	4	15	8
4	16	128	55	1486	19	27	31	64
6	25	648	99	15343	27	361	53	216
8	25	2048	118	81284	36	2234	53	512
10	36	5000	172	298460	41	6417	81	1000
12	36	10368	180	867715	49	15670	81	1728
14	36	19208	196	2145346	49	41922	81	2744
16	36	32768	205	4708392	53	72317	81	4096
18	49	52488	265	9431425	59	120961	115	5832
20	49	80000	275	17574870	69	194580	115	8000

Tabelle 8.1: Vergleich der Algorithmen

[1] Dies besitzt in der Praxis in Verbindung mit den anderen Eigenschaften der Algorithmen eine nicht zu unterschätzende Bedeutung.

8.2 Ausblick

Zum Schluß folgt noch eine kurze Liste weiterer Themen, auf die man bei der Bearbeitung der vorliegenden Arbeit stößt:

- Analyse von Schaltkreisen für die beschriebenen Algorithmen
- Betrachtung von Matrizen mit Elementen aus $\mathbb{C}$
- Analyse verschiedener Varianten der beschriebenen Algorithmen
- Analyse des Aufwandes für die Aufgabenverteilung zwischen mehreren Prozessoren
- Betrachtung anderer Rechnermodelle (insbesondere Rechnermodelle ohne gemeinsamen Speicher für die Prozessoren)
- Analyse des Speicherplatzverbrauchs

Es bleibt also einiges zu tun... . Für dieses Mal soll das jedoch alles sein.

Wenn die Gedanken wieder leichter fließen...

> Himmlische Stille rauscht durch die Nacht
> Samtenes Schweigen strömt durch die Luft
> glitzernd breitet sich das Tal in der Ferne
>
> kein Laut
> ohne Hast läßt die Ruhe verbreiten ihr Glück

Literaturverzeichnis

[Ach67] N. I. Achieser. Vorlesungen über Approximationstheorie. Akademie-Verlag, Berlin, 1967.

[Atk78] K. E. Atkinson. An Introduction to Numerical Analysis. Wiley, New York, 1978.

[Ber84] S. J. Berkowitz. On Computing the Determinant in Small Parallel Time Using a Small Number of Processors. Information Processing Letters 18, 147–150, 1984.

[BM75] A. Borodin und I. Munro. The Computational Complexity of Algebraic and Numeric Problems. American Elsevier, New York, 1975.

[Bod59] E. Bodewig. Matrix Calculus. North-Holland Pub. Co., 1959.

[BS86] I. N. Bronstein und K. A. Semendjajew. Taschenbuch der Mathematik, Ergänzende Kapitel. Verlag Harri Deutsch, Thun und Frankfurt/Main, 4. Ausgabe, 1986.

[BS87] I. N. Bronstein und K. A. Semendjajew. Taschenbuch der Mathematik. Verlag Harri Deutsch, Thun und Frankfurt/Main, 23. Ausgabe, 1987.

[BT70] H. Behnke und P. Thullen. Theorie der Funktionen mehrerer komplexer Veränderlicher. Springer Verlag, 1970.

[BvzGH82] A. Borodin, J. von zur Gathen, und J. Hopcroft. Fast Parallel Matrix and GCD Computation. Information and Control 52, 241–256, 1982.

[CLR86] M. Dal Cin, J. Lutz, und T. Risse. Programmierung in Modula-2. Teubner Verlag, 2. Ausgabe, 1986.

[Csa74] L. Csanky. On the Parallel Complexity of Some Computational Problems. PhD thesis, Computer Science Division, University of California, Berkeley, 1974.

[Csa76] L. Csanky. Fast Parallel Matrix Inversion Algorithms. Siam Journal on Computing, 618–623, 1976.

[CW90] D. Coppersmith und S. Winograd. Matrix Multiplikation via Arithmetic Progressions. Journal of Symbolic Computation 9, 251–280, 1990.

[Dö77] W. Dörfler. Mathematik für Informatiker, Jahrgang Band 1. Hanser Verlag, 1977.

[Dwy51] P. S. Dwyer. Linear Computations. John Wiley and Sons, New York, 1951.

[EMR88] G. Engeln-Müllges und F. Reutter. Formelsammlung zur Numerischen Mathematik mit Modula2-Programmen. BI-Wissenschaftsverlag, 1988.

[Fra49] J. S. Frame. A Simple Recurrent Formula for Inverting a Matrix. Bulletins of the American Mathematic Society 55, 1045, 1949.

[GvL83] G. H. Golub und C. F. van Loan. Matrix Computation. The John Hopkins University Press, Baltimore, 1983.

[Hau52] O. Haupt. Einführung in die Algebra, 1952.

[Hil74] F. B. Hildebrand. Introduction to Numerical Analysis, 1974.

[Hö73] L. Hörsmander. An Introduction to Complex Analysis in Several Variables. North Holland Publishing Company, 1973.

[Hou64] A. Housholder. The Theory of Matrices in Numerical Analysis, 1964.

[IK73] E. Isaacson und H. B. Keller. Analyse numerischer Verfahren, 1973.

[Kel82] W. Keller. Fast Algorithms for the Characteristic Polynomial, 1982. Diplomarbeit, Institut für Angewandte Mathematik, Univ. Zürich.

[Kry31] A. N. Krylov. 7. Ser. Classe Mathem. . Bulletins of the Academie of Sciences of URSS, 491–538, 1931.

[LF80] R. Ladner und M. Fischer. Parallel Prefix Computations. Journal of the ACM 27, 831–838, 1980.

[MM64] M. Marcus und H. Ming. A Survey of Matrix Theory and Matrix Inequalities. Allyn and Bacon, Boston, 1964.

[Pan85] V. Pan. Fast and Efficient Parallel Algorithms for the Exact Inversion of Integer Matrices. Lecture Notes on Computer Science 206, 504–521, 1985.

[PR85a] V. Pan und J. Reif. Efficient Parallel Solution of Linear Systems. Proceedings of the 17th Annual ACM Symposium on Theory of Computing, 143–152, 1985.

[PR85b] V. Pan und J. Reif. Fast and Efficient Parallel Solution of Linear Systems. Interner Bericht TR-02-85, Center for Research in Computer Technology, Aiken Computation Laboratory, Harvard University, 1985.

[Sam42] P. A. Samuelson. A Method for Determining Explicitly the Coefficients of the Characteristic Equation. Annals of Mathematical Statistics 13, 424–429, 1942.

[Str73] V. Strassen. Vermeidung von Divisionen. J. Reine Angew. Math. 264, 184–202, 1973.

[vM67] H. von Mangoldt. Einführung in die höhere Mathematik, 1967.

[VSBR83] L. Valiant, S. Skyum, S. Berkowitz, und C. Rackoff. Fast Parallel Computation of Polynomials Using Few Processors. SIAM Journal on Computing 12 (4), 641–644, 1983.

[Wal87] I. Wald. Automatische Parallelisierung von Registermaschinen, 1987. Diplomarbeit, Johann Wolfgang Goethe-Universität, Frankfurt am Main.

[Weg89] I. Wegener. Effiziente Algorithmen für grundlegende Funktionen. B. G. Teubner, Stuttgart, 1989.

[Zur64] R. Zurmühl. Matrizen, 1964.

Index

Algorithmen zur parallelen Determinantenberechnung

Holger Burbach

Oktober 1992

Anhang (Quelltexte)

Anhang A

Implementierung der parallelen Determinantenberechnung

In diesem Kapitel sind alle Module gesammelt, denen das vorrangige Interesse der Implementierung gilt.

A.1 Programmodul 'main'

```
MODULE main;

(* Hauptmodul der Implementierungen zur Diplomarbeit
   'Algorithmen zur parallelen Determinantenberechnung'

   Dieses Modul erlaubt es, die implementierten Algorithmen
   auszuprobieren.
*)

FROM InOut IMPORT WriteReal, WriteString, WriteLn,
                  ReadCard, ReadString, ReadLn;

(* Um die korrekte Reihenfolge bei automatischer Kompilation und
   Initialisierung zu gewaehrleisten, werden hier auch die Module
   importiert, die nicht direkt in 'main' verwendet werden.
*)

IMPORT SysMath, Sys,
       Func,
       Rnd, Str,
       Type,
       Frag, List, Simptype,
       Mat, Cali, Inli, Reli,
       Hash, Pram,
       Rema,
       Data, Mali,
       Det;

VAR
    cur: Data.Id;
        (* aktuell in Bearbeitung befindlicher Testdatensatz *)

PROCEDURE WaitReturn;
VAR
    input: ARRAY [1..2] OF CHAR;
```

```
BEGIN
    WriteString("    <RETURN> ...");
    ReadString(input)
END WaitReturn;

PROCEDURE PrintHelp();
(* Diese Prozedur gibt einen Hilfstext fuer den Benutzer aus. *)
BEGIN
    WriteString(
    ">>>> Hilfe zur Programmbedienung <<<<");
    WriteLn; WriteString(
    "Das Programm versteht folgende Befehle (Gross- / ");
    WriteLn; WriteString(
    "Kleinschreibung wird nicht beachtet):");
    WriteLn; WriteLn; WriteString(
    "  Befehl       Wirkung");
    WriteLn; WriteString(
    "  ------       -------");
    WriteLn; WriteString(
    "  ?, h,        diesen Text ausgegeben");
    WriteLn; WriteString(
    "  help, hilfe");
    WriteLn; WriteString(
    "  q, exit      Programm beenden");
    WriteLn; WriteString(
    "  find         anderen/neuen Testdatensatz bearbeiten");
    WriteLn; WriteString(
    "  show         Testdatensatz anzeigen (Falls die Matrix zu");
    WriteLn; WriteString(
    "                   gross ist, wird sie nicht automatisch mit");
    WriteLn; WriteString(
    "                   angezeigt. Dies kann mit 'mshow' geschehen)");
    WriteLn; WriteString(
    "  mshow        Matrix des Testdatensatzes anzeigen");
    WriteLn; WriteString(
    "  del          Testdatensatz loeschen");
    WriteLn; WriteString(
    "  ls           Testdatensaetze auflisten");
    WriteString(" (die im Speicher stehen)");
    WriteLn; WriteString(
    "  param        Parameter fuer Matrizengenerierung festlegen");
    WriteLn; WriteString(
    "  gen          Matrix generieren");
    WriteLn; WaitReturn; WriteString(
    "  csanky       Det. mit Alg. von Csanky berechnen");
    WriteLn; WriteString(
    "  bgh          Det. mit Alg. von Borodin, von zur Gathen");
    WriteLn; WriteString(
    "                   und Hopcroft berechnen");
    WriteLn; WriteString(
    "  berk         Det. mit Alg. von Berkowitz berechnen");
    WriteLn; WriteString(
    "  pan          Det. mit Alg. von Pan berechnen");
    WriteLn;
END PrintHelp;

PROCEDURE CommandNumber(VAR c, i: ARRAY OF CHAR): CARDINAL;
(* 'c' muss eine Zeichenkette entsprechend der Beschreibung fuer
   die Variable 'commands' enthalten. 'i' wird als Befehl
   betrachtet. Anhand von 'commands' wird diesem Befehl eine
   Befehlsnummer zugeordnet und als Funktionswert zurueck-
   gegeben. *)
VAR
    pos: CARDINAL;
    erg: CARDINAL; (* Funktionsergebnis *)
```

```
BEGIN
    IF Str.In(i,c,pos) THEN
        WHILE (c[pos] # ',') DO
            INC(pos)
        END;
        INC(pos);
        erg:= 0;
        REPEAT
            erg:= erg*10 + (ORD(c[pos]) - ORD('0'));
            INC(pos);
        UNTIL (c[pos] = ';') OR (c[pos] = CHR(0));
    ELSE
        erg:= 0
    END;
    RETURN erg
END CommandNumber;

PROCEDURE dummy;
BEGIN
    WriteString(" Befehl noch nicht implementiert"); WriteLn
END dummy;

PROCEDURE IsSet(dat: Data.Id): BOOLEAN;
(* Falls 'dat = NIL' wird eine Fehlermeldung ausgeben. In diesem
   Fall ist der Funktionswert FALSE, andernfalls TRUE. *)
BEGIN
    IF (dat = Data.Id(NIL)) THEN
        WriteString(" kein Datensatz ausgewaehlt");
        WriteString(" (zuerst  find  benutzen)");
        WriteLn;
        RETURN FALSE
    END;
    RETURN TRUE
END IsSet;

VAR
    input: ARRAY [1..10] OF CHAR;
        (* Eingabe des Benutzers *)
    commands: ARRAY [1..160] OF CHAR;
        (* Zeichenkette zur Zuordnung von Befehlen zu Befehlsnummern;
           Format:
               command    :  Befehlskode ";" { Befehlskode ";" }
               Befehlskode:  <Zeichenkette> "," <Ziffernfolge>
        *)
    CommNum: CARDINAL;
        (* Befehlsnummer zu 'input' *)
    name: ARRAY [1..16] OF CHAR;
        (* vom Benutzer eingegebener Dateiname *)

PROCEDURE ComParam;
(* Befehl 'param' *)
VAR
    input: ARRAY [1..2] OF CHAR;
    size: CARDINAL; (* vom Benutzer angegebene Matrizengroesse *)
    rank: CARDINAL; (* ... Rang *)
    mult: CARDINAL; (* ... Vielfachheit eines Eigenwertes *)
    i: Data.tAlg;
BEGIN
    IF IsSet(cur) THEN
        FOR i:= Data.laplace TO Data.pan DO
            Data.SetAlg(cur, i, 0.0, 0, 0)
        END;

        WriteLn;
        WriteString("Anzahl der Zeilen und Spalten? ");
```

```
        ReadCard(size); WriteLn;
        WriteString("Rang? "); ReadCard(rank); WriteLn;
        WriteString("Nachkommastellen (j/n)? ");
        ReadString(input); WriteLn;
        Rema.SetSize( Data.GetMat(cur), size, size);
        Rema.SetRank( Data.GetMat(cur), size);
        Rema.SetReal( Data.GetMat(cur),
                     Str.Equal(input,"j")
        );
        REPEAT
            WriteString("Vielfachheit ungleich 1 eines");
            WriteString(" Eigenwertes (0,1 : Ende) ? ");
            ReadCard(mult); WriteLn;
            IF mult > 1 THEN
                Rema.SetMultiplicity(
                    Data.GetMat(cur), mult
                );
            END
        UNTIL mult <= 1;
        Data.HasChanged(cur)
    END
END ComParam;

PROCEDURE ComAlg(alg: Data.tAlg);
(* Befehle: csanky, bgh, berk, pan;
   'alg' gibt zu benutzenden Algorithmus an *)
VAR
    det: LONGREAL;
BEGIN
    Pram.Start;
    CASE alg OF
        Data.csanky :
            det:= Det.Csanky( Data.GetMat(cur) )
      | Data.bgh :
            det:= Det.BGH( Data.GetMat(cur) )
      | Data.berk :
            det:= Det.Berkowitz( Data.GetMat(cur) )
      | Data.pan :
            det:= Det.Pan( Data.GetMat(cur) )
    END;
    Pram.Ende;
    Data.SetAlg(cur, alg, det, Pram.GezaehlteProzessoren(),
                Pram.GezaehlteSchritte()
    )
END ComAlg;

BEGIN
    Str.Empty(commands);
    Str.Assign(commands,"h,1;?,1;help,1;hilfe,1;q,2;exit,2;");
    Str.Append(commands,"find,3;show,4;mshow,5;param,6;gen,7;");
    Str.Append(commands,";csanky,9;bgh,10;berk,11;pan,12;");
    Str.Append(commands,"del,13;ls,14;");
    cur:= Data.Id(NIL);

    WriteLn;
    WriteString("*** Algorithmen zur parallelen ");
    WriteString("Determinantenberechnung ***"); WriteLn;
    REPEAT
        WriteLn;
        WriteString(">> "); ReadString(input);
        WriteLn;
        Str.Lower(input);
        CommNum:= CommandNumber(commands, input);
        CASE CommNum OF
            1 : (* h, ?, help, hilfe *)
```

```
            PrintHelp
        | 2 : (* q, exist *)
            (* tue nichts *)
        | 3 : (* find *)
            IF cur # Data.Id(NIL) THEN
                Data.FlushOnly(cur)
            END;
            WriteString("Name? "); ReadString(name); WriteLn;
            Data.Find(cur,name);
        | 4 : (* show *)
            IF IsSet(cur) THEN
                Data.Write(cur);
                WriteString("Bei der Generierung bestimmte");
                WriteString(" Determinante: ");
                WriteReal( Rema.Det( Data.GetMat(cur) ), 12, 4);
                WriteLn;
                WriteString(
                " ... zur Fortsetzung 'RETURN'-Taste ...");
                ReadString(input);
                IF Rema.Rows( Data.GetMat(cur) )
                   <= Rema.MaxIoRow
                THEN
                    Rema.Write( Data.GetMat(cur) )
                END
            END
        | 5 : (* mshow *)
            IF IsSet(cur) THEN
                Rema.Write(Data.GetMat(cur))
            END
        | 6 : (* param *)
            ComParam
        | 7 : (* gen *)
            IF IsSet(cur) THEN
                Rema.Randomize( Data.GetMat(cur) )
            END;
        | 9..12 :
            (* csanky, bgh, berk, pan *)
            IF IsSet(cur) THEN
                ComAlg(VAL(Data.tAlg, CommNum-8))
            END;
        | 13: (* del *)
            IF cur # Data.Id(NIL) THEN
                Data.FlushOnly(cur)
            END;
            WriteString("Name? "); ReadString(name); WriteLn;
            Data.Find(cur,name);
            Data.Del(cur)
        | 14: (* ls *)
            Data.ListNames;
            WHILE Data.NextName(name) DO
                WriteString(name); WriteLn
            END
      ELSE
          WriteString(" Befehl unbekannt");
          WriteString(" (fuer Hilfestellung  h  eingeben)");
          WriteLn
      END
   UNTIL CommNum= 2;
   Data.End
END main.
```

A.2 Definitionsmodul 'Det'

```
DEFINITION MODULE Det;

(* Verschiedene Algorithmen zur Determinantenberechnung

   In den Algorithmen dieses Moduls werden die Zaehlprozeduren
   des Moduls 'Pram' aufgerufen, so dass nach jeder Berechnung
   einer Determinante festgestellt werden kann, wieviele
   Schritte und Prozessoren eine PRAM zur Abarbeitung des
   Algorithmus benoetigt.
*)

IMPORT Sys, SysMath, Func, Type, List, Frag, Hash, Reli, Mat,
       Pram, Rema, Mali;
FROM Rema IMPORT tMat;

PROCEDURE EineSpalte(mat: tMat): LONGREAL;
(* ... Entwicklungssatz von Laplace (Entwicklung nach einer
       Spalte) *)

PROCEDURE Laplace(mat: tMat): LONGREAL;
(* ... Entwicklungssatz von Laplace (Entwicklung nach k Spalten) *)

PROCEDURE Csanky(mat: tMat): LONGREAL;
(* ... Algorithmus von Csanky *)

PROCEDURE BGH(mat: tMat): LONGREAL;
(* ... Algorithmus von Borodin, von zur Gathen und Hopcroft *)

PROCEDURE Berkowitz(mat: tMat): LONGREAL;
(* ... Algorithmus von Berkowitz *)

PROCEDURE Pan(mat: tMat): LONGREAL;
(* ... Algorithmus von Pan *)

END Det.
```

A.3 Implementierungsmodul 'Det'

```
IMPLEMENTATION MODULE Det;

(* Verschiedene Algorithmen zur Determinatenberechnung

         ( Erklaerungen im Definitionsmodul )
*)

FROM SYSTEM IMPORT TSIZE, ADR;
FROM InOut IMPORT WriteLn, WriteString, WriteCard, WriteReal;
IMPORT Sys, SysMath, Func, Type, List, Frag, Hash, Reli, Mat,
       Pram, Rema, Mali;
FROM Sys IMPORT tPOINTER;
FROM SysMath IMPORT ld, lg, real, power, Card2LCard, Card2LReal,
                    LReal2Card, LReal2LCard, LCard2Card, LCard2LReal,
                    LInt2LReal;
FROM Func IMPORT Message, Error;
FROM Rema IMPORT tMat, Rows, Columns, Elem, Set;

CONST (* Schalter zur Fehlersuche (TRUE: entsprechender Programm-
         teil gibt Testmeldungen aus):  *)
```

```
      BGHDebug = TRUE;
          (* ... Algorithmus von Borodin, von zur Gathen und
                 Hopcroft (Prozedur 'BGH' ... ) *)
      BerkDebug = TRUE;
          (* ... Algorithmus von Berkowitz *)

      BerkEpsilon = 0.5;
          (* 'Epsilon' fuer den Berkowitz-Algorithmus *)

      HashSize = 10000L;
          (* Groessenvorgabe fuer den verwendeten Hash-Speicher *)

VAR MatId: Type.Id; (* Typidentifikator fuer 'Matrix' *)
    ListId: Type.Id; (* Typidentifikator fuer 'List' *)

(*********************************************************)
(***** gemeinsame Prozeduren verschiedener Algorithmen: *****)

PROCEDURE CheckSquare(mat: tMat; proc: ARRAY OF CHAR);
(* Falls 'mat' keine quadratische Matrix ist, wird eine
   Fehlermeldung ausgegeben. Dabei erscheint 'proc' als
   Funktionsname in der Meldung. *)
BEGIN
    IF Rows(mat) # Columns(mat) THEN
        Error(proc,
            "Die Matrix ist nicht quadratisch.")
    END;
    IF (Rows(mat) < 1) OR (Columns(mat) < 1) THEN
        Error(proc,
        "Die Matrix besitzt weniger als eine Zeile oder Spalte !??");
    END
END CheckSquare;

PROCEDURE SetToeplitz(a: tMat);
(* Anhand der Elemente der in ihrer ersten Spalte wird 'a' in eine
   untere Dreiecks-Toeplitz-Matrix ueberfuehrt.

   Nach dem Aufruf gilt also:
       a_{i,j} = a_{i-1,j-1}  und  i<j \Rightarrow a_{i,j} = 0
*)
VAR i, (* Zeile *)
    j, (* Spalte *)
    r, (* Zeilen insgesamt *)
    c: (* Spalten insgesamt *)
       LONGCARD;
BEGIN
    r:= Rows(a);
    c:= Columns(a);
    FOR j:= 2 TO c DO
        Set(a, 1, j, 0.0)
    END;
    FOR i:= 2 TO r DO
        FOR j:= 2 TO c DO
            Set(a, i, j, Elem(a, i-1, j-1))
        END
    END
END SetToeplitz;

PROCEDURE MultMatList(VAR res: tMat;
                      matlist: List.tList;
                      toepliz: BOOLEAN);
(* In 'matlist' muss eine Liste von Matrizen (Typ tMat)
   uebergeben werden.
   In 'res' wird das Produkt aller dieser Matrizen zurueckgegeben.
   Der Inhalt von 'matlist' geht verloren.
```

```
   Falls in 'toepliz' der Wert TRUE uebergeben wird, geht die
   Prozedur davon aus, das 'matlist' untere Dreiecks-Toepliz-Matrizen
   enthaelt und benutzt einen effizienteren Multiplikations-
   algorithmus.
   Bei 'toepliz = FALSE' wird 'Rema.Mult' zur Multiplikation verwendet.
   'res' wird in 'MultMatList' angelegt. 'matlist' muss ausserhalb
   von 'MultMatList' angelegt und geloescht werden. *)
VAR l: ARRAY [1..2] OF Mali.tMali;
    m1, m2, m: tMat;
    source, target: [1..2];
    i,k: LONGCARD;
    sum: LONGREAL;
    AddList: Reli.tReli;
        (* Liste fuer Addition nach der Binaerbaummethode *)
BEGIN
    Reli.Use(AddList);
    Mali.Use(l[1]);
    l[2]:= matlist;
    source:= 1; target:= 2;
    IF List.Count( l[target] ) = 0 THEN
       Error("Det.MultMatList", "Die Liste ist leer")
    END;

    WHILE List.Count( l[target] ) > 1 DO
        target:= 3 - target; source:= 3 - source;
        Pram.ParallelStart("Det.MultMatList");
        REPEAT
            List.First( l[source] );
            m1:= Mali.OutCur( l[source] );
            m2:= Mali.OutCur( l[source] );
            Rema.Use(m, Rows(m1), Columns(m2) );

            IF toepliz THEN
                (* effizienter Multiplikationsalgorithmus fuer
                   untere Dreiecks-Toepliz-Matrizen: *)
                IF Columns(m1) # Rows(m2) THEN
                    Error("Det.MultMatList",
                        "Die Liste enthaelt inkompatible Matrizen.");
                END;
                Pram.ParallelStart("Det.MultMatList:Toeplitz");
                FOR i:= 1 TO Rows(m1) DO
                    List.Empty(AddList);

                    (* untere Dreieckmatrix; deshalb
                       Rows(m) statt Columns(m) moeglich: *)
                    FOR k:= 1 TO Columns(m1) DO
                        Reli.InsertBehind(AddList,
                            Elem(m1, i, k) * Elem(m2, k, 1)
                        )
                    END;

                    (* Die Durchlaeufe der obigen Schleife werden
                       parallel durchgefuehrt: *)
                    Pram.Prozessoren(Rows(m));
                    Pram.Schritte(1);

                    Set(m, i, 1, Pram.AddList(AddList) );
                    Pram.NaechsterBlock("Det.MultMatList:Toeplitz")
                END;
                Pram.ParallelEnde("Det.MultMatList:Toeplitz");
                SetToeplitz(m)
            ELSE
                Rema.Mult(m1, m2, m)
            END;
```

```
            Rema.DontUse(m1);
            Rema.DontUse(m2);
            Mali.InsertBehind( l[target], m );

            IF List.Count( l[source] ) = 1 THEN
                Mali.InsertBehind(
                    l[target], Mali.OutCur( l[source] )
                )
            END;
            Pram.NaechsterBlock("Det.MultMatList");
        UNTIL List.Count( l[source] ) = 0;
        Pram.ParallelEnde("Det.MultMatList");
    END;

    List.First( l[target] );
    res:= Mali.OutCur( l[target] );
    List.DontUse(l[1]);
    List.DontUse(AddList)
END MultMatList;

PROCEDURE MatFragSet(f: Frag.tFrag; index: LONGCARD; item: tMat);
(* ... zur vereinfachten Handhabung *)
BEGIN
    Frag.SetItem(f, index, tPOINTER(item))
END MatFragSet;

PROCEDURE MatFragGet(f: Frag.tFrag; index: LONGCARD): tMat;
(* ... zur vereinfachten Handhabung *)
BEGIN
    RETURN tMat( Frag.GetItem(f, index) )
END MatFragGet;

PROCEDURE Praefixalg(x, res: Frag.tFrag);
(* 'x' und 'res' muessen Felder von Matrizen (Typ 'Rema.tMat') sein.
   In jedem Element 'i' von 'res' wird das Produkt der ersten 'i'
   Elemente von 'x' zurueckgegeben. Die Produkte werden mit Hilfe
   des Ladner-Fischer-Praefixalgorithmus berechnet.
   'x' darf leere Elemente ( = NIL ) enthalten. Die minimal zulaessige
   Laenge von 'x' ist 1.
*)
VAR UpperHalf, ResUpperHalf,
    LowerQuarter, ResLowerQuarter: Frag.tFrag;
    frontier: LONGCARD;
        (* groesster Index der unteren Haelfte von 'x' bzw. 'res' *)
    i: LONGCARD;
BEGIN
    IF Frag.GetHigh(x) = 1 THEN
        (* Es ist nichts zu berechnen (kopiere Eingabe): *)
        MatFragSet( res, 1, Rema.Copy(MatFragGet(x, 1)) )

    ELSIF Frag.GetHigh(x) = 2 THEN
        (* Es ist nur das zweite Element von 'res' zu berechnen
           (eine Matrizenmultiplikation): *)
        MatFragSet( res, 1, Rema.Copy(MatFragGet(x, 1)) );
        IF (MatFragGet(x, 1) # tMat(NIL))
           AND (MatFragGet(x, 2) # tMat(NIL))
        THEN
            MatFragSet( res, 2, Rema.CreateMult(
                                    MatFragGet(x, 1),
                                    MatFragGet(x, 2)
                                )
            )
        END
    ELSE
        IF (Frag.GetHigh(x) MOD 4) # 0 THEN
```

```
    Error("Det.Praefixalg",
        "Die Feldgroese ist nicht durch 4 teilbar.")
END;
frontier:= Frag.GetHigh(x) DIV 2;
Pram.ParallelStart("Det.Praefixalg:halves");
    (* multipliziere die Elemente der unteren Haelfte des
       Eingabefeldes 'x' paarweise miteinander: *)

    Frag.Use(LowerQuarter, MatId, 1, frontier DIV 2);
    Frag.Use(ResLowerQuarter, MatId, 1, frontier DIV 2);
    Frag.AddRef(ResLowerQuarter, TRUE);

    Pram.ParallelStart("Det.Praefixalg:lower");
        FOR i:= 1 TO frontier DIV 2 DO
            IF (MatFragGet(x, 2 * i - 1) # tMat(NIL))
               AND (MatFragGet(x, 2 * i) # tMat(NIL))
            THEN
                MatFragSet(LowerQuarter, i,
                    Rema.CreateMult(
                        MatFragGet(x, 2 * i - 1),
                        MatFragGet(x, 2 * i)
                    )
                )
            END;
        Pram.NaechsterBlock("Det.Praefixalg:lower")
        END;
    Pram.ParallelEnde("Det.Praefixalg:lower");

    (* berechne die Elemente mit geradem Index der unteren
       Haelfte des Ergebnisfeldes 'res': *)

    Praefixalg(LowerQuarter, ResLowerQuarter);
    FOR i:= 1 TO frontier DIV 2 DO
        MatFragSet(res, i * 2,
                   MatFragGet(ResLowerQuarter, i))
    END;

    Frag.DontUse(LowerQuarter);
    Frag.DontUse(ResLowerQuarter);
Pram.NaechsterBlock("Det.Praefixalg:halves");
    (* loese das Problem rekursiv fuer die obere Haelfte des
       Eingabefeldes 'x': *)

    Frag.Use(UpperHalf, MatId, 1, frontier);
    Frag.AddRef(UpperHalf, TRUE);
    Frag.Use(ResUpperHalf, MatId, 1, frontier);

    FOR i:= frontier + 1 TO Frag.GetHigh(x) DO
        MatFragSet(UpperHalf, i - frontier, MatFragGet(x, i))
    END;
    Praefixalg(UpperHalf, ResUpperHalf);
Pram.ParallelEnde("Det.Praefixalg:halves");
Pram.ParallelStart("Det.Praefixalg:nextstep");
    (* berechne die Elemente mit ungeradem Index der unteren
       Haelfte des Ergebnisfeldes 'res': *)

    MatFragSet( res, 1, Rema.Copy(MatFragGet(x, 1)) );
    Pram.ParallelStart("Det.Praefixalg:oddlower");
        FOR i:= 3 TO (frontier DIV 2) BY 2 DO
            IF (MatFragGet(res, i - 1) # tMat(NIL))
            AND (MatFragGet(x, i) # tMat(NIL)) THEN
                MatFragSet( res, i,
                    Rema.CreateMult(
                        MatFragGet(res, i - 1),
                        MatFragGet(x, i)
```

```
                        )
                    );
                END;
            Pram.NaechsterBlock("Det.Praefixalg:oddlower")
            END;
        Pram.ParallelEnde("Det.Praefixalg:oddlower");
    Pram.NaechsterBlock("Det.Praefixalg:nextstep");
        (* berechne die Elemente der oberen Haelfte des Ergebnis-
           feldes 'res': *)

        Pram.ParallelStart("Det.Praefixalg:resupper");
            FOR i:= frontier + 1 TO Frag.GetHigh(res) DO
                IF (MatFragGet(res, frontier) # tMat(NIL)) AND
                   (MatFragGet(ResUpperHalf, i - frontier) # tMat(NIL))
                THEN
                    MatFragSet(res, i,
                        Rema.CreateMult(
                            MatFragGet(res, frontier),
                            MatFragGet(ResUpperHalf, i-frontier)
                        )
                    )
                END;
                Pram.NaechsterBlock("Det.Praefixalg:resupper")
            END;
        Pram.ParallelEnde("Det.Praefixalg:resupper");
        Frag.DontUse(UpperHalf);
        Frag.DontUse(ResUpperHalf);
    Pram.ParallelEnde("Det.Praefixalg:nextstep");
END
END Praefixalg;

PROCEDURE MultLadnerFischer(l: Mali.tMali; m: tMat; max: LONGCARD);
(* Mit Hilfe des Ladner-Fischer-Praefixalgorithmus werden
   fuer 'm' alle Potenzen von 1 bis 'max' berechnet und in
   'l' zurueckgegeben.
   Dabei werden Zaehlprozeduren des Moduls 'Pram' aufgerufen.
*)
VAR i, size: LONGCARD;
    x, res: Frag.tFrag;
BEGIN
    size:= LReal2LCard(
            power( 2.0,
                LCard2LReal( Func.Ceil(ld(LCard2LReal(max))) )
            )
        );
    Frag.Use(x, MatId, 1, size);
    Frag.Use(res, MatId, 1, size);
    FOR i:= 1 TO max DO
        MatFragSet(x, i, Rema.Copy(m))
    END;

    Praefixalg(x, res);

    List.Empty(l);
    FOR i:= 1 TO max DO
        Mali.InsertBehind(l, MatFragGet(res, i))
    END;

    Frag.DontUse(x);
    Frag.DontUse(res)
END MultLadnerFischer;

(*************************************************************)
(***** Algorithmus nach der Spaltenentwicklung von Laplace *****)
(*****       ( Spezialfall des Entwicklungssatzes )        *****)
```

```
PROCEDURE RemoveRowAndColumn(a: tMat; r, c: LONGCARD): tMat;
(* Funktionsergebnis ist die Matrix, die man aus 'a' durch
   Streichen von Zeile 'r' und Spalte 'c' erhaelt.
*)
VAR res: tMat; (* Funktionsergebnis *)
    i, j, i2, j2: LONGCARD;
BEGIN
    Rema.Use(res, Rows(a) - 1, Columns(a) - 1);
    FOR i:= 1 TO Rows(res) DO
        FOR j:= 1 TO Columns(res) DO
            IF i < r THEN
                i2:= i
            ELSE
                i2:= i+1
            END;
            IF j < c THEN
                j2:= j
            ELSE
                j2:= j+1
            END;
            Set(res, i, j, Elem(a, i2, j2))
        END
    END;
    RETURN res
END RemoveRowAndColumn;

PROCEDURE EineSpalte(a: tMat): LONGREAL;
(* Funktionsergebnis ist die Determianten von 'a'. Sie wird durch
   rekursive Entwicklung nach der ersten Spalte berechnet.
*)
VAR WorkMat: tMat;
        (* Untermatrix von 'a', deren Determinante als naechste
           zu berechnen ist *)
    DetList: Reli.tReli;
        (* Liste der vorzeichenbehafteten Determinanten von Unter-
           Matrizen von 'a' *)
    i: LONGCARD; (* Schleifenzaehler *)
    n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    AddList: Reli.tReli;
    res: LONGREAL;
BEGIN
    CheckSquare(a,"Det.Spalten");
    n:= Rows(a);
    IF n = 1 THEN
        RETURN Elem(a,1,1)
    END;

    Reli.Use(DetList);
    Reli.Use(AddList);

    Pram.ParallelStart("Det.EineSpalte");
    FOR i:= 1 TO n DO
        WorkMat:= RemoveRowAndColumn(a, i, 1);
        Reli.InsertBehind(
            DetList, EineSpalte(WorkMat) * power( -1.0, real(i+1) )
        );
        Rema.DontUse(WorkMat);
        Pram.NaechsterBlock("Det.EineSpalte")
    END;
    Pram.ParallelEnde("Det.EineSpalte");

    List.First(DetList);
    FOR i:= 1 TO n DO
        Reli.InsertBehind(AddList,
```

```
            Elem(a, i, 1) * Reli.Cur(DetList)
        );
        List.Next(DetList)
    END;
    (* Die Schleifendurchlaufe werden parallel durchgefuehrt: *)
    Pram.Prozessoren( n );
    Pram.Schritte( 1 );

    res:= Pram.AddList(AddList);
        (* Der zur Addition der Elemente von 'AddList' wird in
           'Pram.AddList' gezaehlt. *)

    List.DontUse(DetList);
    List.DontUse(AddList);
    RETURN res
END EineSpalte;

PROCEDURE Laplace(a: tMat): LONGREAL;
BEGIN
    CheckSquare(a,"Det.Laplace");
    IF Rows(a) = 1 THEN
        RETURN Elem(a,1,1)
    END;

    RETURN EineSpalte(a)
END Laplace;

(***********************************)
(*****  Algorithmus von Csanky  *****)
(***** nach dem Satz von Frame: *****)

PROCEDURE Csanky(a: tMat): LONGREAL;
VAR b0: tMat;
    tr: LONGREAL; (* Spur von 'a' *)
    MatList: Mali.tMali;
        (* Liste von zu multiplizierenden Matrizen *)
    work: tMat;
        (* aktuelle Arbeitsmatrix *)
    i,j: LONGCARD; (* Schleifenzaehler *)
    z: LONGREAL; (* Zwischenergebnis im Algorithmus *)
    det: LONGREAL; (* Determinante von 'a' *)
    size: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
BEGIN
    Mali.Use(MatList);
    CheckSquare(a,"Det.Csanky");
    IF Rows(a) = 1 THEN
        RETURN Elem(a,1,1)
    END;

    size:= Rows(a);

    tr:= Rema.Trace(a);
    (* Der Aufwand wird in 'Rema.Trace' gezaehlt. *)

    Pram.ParallelStart("Det.Csanky");
    FOR i:= size - 1 TO 1 BY -1 DO
        Rema.Use(work, Rows(a), Columns(a));
        Rema.Assign(a, work);
        (* Zuweisungen werden nicht gezaehlt. *)

        z:= tr / real(i);
        Pram.Prozessoren(1);
        Pram.Schritte(1);

        FOR j:= size TO 1 BY -1 DO
```

```
            Set(work, j, j, Elem(work, j, j) - z)
        END;
        Pram.Prozessoren(size);
        Pram.Schritte(1);

        Mali.InsertBefore(MatList, work);
        (* Verwaltungsaufwand wird nicht gezaehlt. *)

        Pram.NaechsterBlock("Det.Csanky");
        (* Die Schleifendurchlaeufe werden parallel durchgefuehrt. *)
    END;
    Pram.ParallelEnde("Det.Csanky");

    MultMatList(b0, MatList, FALSE);

    (* 'b0' wird als Zwischenspeicher missbraucht: *)
    Rema.Mult(a,b0,b0);
        (* Der Aufwand wird im Modul 'Rema' gezaehlt. *)

    det:= Rema.Trace(b0) / real( size );
        (* Der Aufwand fuer 'Rema.Trace' wird dort gezaehlt. *)
    Pram.Prozessoren(1);
    Pram.Schritte(1);

    List.DontUse(MatList);

    RETURN det
END Csanky;

(*****************************************************************)
(***** Algorithmus von Borodin, von zur Gathen und Hopcroft: *****)

(* Typen fuer den Algorithmus: *)

TYPE tOpType = ( constant, indeterminate, node );
     tOp = POINTER TO tOpRec;
     tOpRec
         = RECORD (* ein Operand eines Knotens *)
               CASE vOpType: tOpType OF
                   constant:
                   (* der Operand ist eine Konstante *)
                       ConstVal: LONGREAL (* Wert der Konstante *)
                 | indeterminate:
                   (* der Operand ist eine Unbestimmte *)
                       IndetVal: LONGREAL
                           (* Wert, der bei der Ausfuehrung des
                              Programms fuer die Unbestimmte
                              eingesetzt werden soll *)
                 | node:
                   (* der Operand ist das Ergebnis eines anderen
                      Knotens *)
                       NodePos: List.tPos
                           (* Position des Knotens in der
                              Knotenliste *)
               END
           END;
     tNode = POINTER TO tNodeRec;
     tNodeRec
           = RECORD (* ein Additions- oder Multiplikationsknoten *)
                 DegValid: BOOLEAN;
                     (* TRUE: 'degree' wurde bereits gesetzt *)
                 degree: LONGCARD;
                     (* Grad des Knotens *)
                 ResValid: BOOLEAN;
                     (* TRUE: 'res' wurde bereits berechnet *)
```

```
                res: LONGREAL;
                    (* Ergebnis des Knotens *)
                add: BOOLEAN;
                    (* TRUE:  der Knoten ist ein Additionsknoten;
                       FALSE: ... ein Multiplikationsknoten *)
                op1, op2: tOp
                    (* die beiden Operanden *)
            END;
     tSer = Frag.tFrag;  (* power SERies *)
         (* jede Potenzreihe wird als Feld ihrer homogenen Komponen-
            ten bis zum festgelegten Grad implementiert *)

VAR NodeId: Type.Id; (* Typidentifikator eines Knotens *)
    OpId  : Type.Id; (* ... eines Operanden *)
    SerId: Type.Id; (* ... eine Potenzreihe *)

    vMaxDeg: LONGCARD;
        (* maximaler Grad der homogenen Komponenten, die fuer
           Potenzreihen betrachtet werden *)

PROCEDURE MaxDeg(): LONGCARD;
(* Funktionsergebnis: siehe 'vMaxDeg';
   die Deklaration der Prozedur 'MaxDeg' erlaubt, verglichen mit
   anderen Moeglichkeiten der Handhabung des maximalen Grades,
   groessere Flexibilitaet beim Experimentieren mit der
   Implementierung des Algorithmus *)
BEGIN
    RETURN vMaxDeg
END MaxDeg;

PROCEDURE ConvertMat(a: tMat; VAR m2: LONGREAL);
(* Die Elemente von 'a' werden so transformiert, dass sie im Inter-
   vall von -0.01 bis 0.01 liegen. In 'm2' wird der Faktor zurueck-
   gegeben, mit dem die Determinante der transformierten Matrix 'a'
   multipliziert werden muss, um die Determinante der urspruenglichen
   Matrix zu erhalten.
*)
VAR m, m1, max: LONGREAL;
    i, j, size: LONGCARD;
BEGIN
    size:= Rows(a);
    max:= 0.0;
    FOR i:= 1 TO size DO
        FOR j:= 1 TO size DO
            max:= Func.MaxReal(ABS(Elem(a, i, j)), max)
        END
    END;

    m:= power( 10.0, real(Func.Ceil(lg(max)) + 1) );
    m1:= 1.0 / m;
    m2:= power(m, real(size));

    FOR i:= 1 TO size DO
        FOR j:= 1 TO size DO
            Set(a, i, j, Elem(a, i, j) * m1)
        END
    END
END ConvertMat;

(* ---------------------------------------- *)
(* Operationsprozeduren fuer Modul 'Type': *)

PROCEDURE NewOpI(o: tPOINTER);
(* Initialisierungsprozedur fuer Modul 'Type' *)
VAR O: tOp;
```

```
BEGIN
    O:= o;
    WITH O^ DO
        vOpType:= constant;
        ConstVal:= 0.0
    END
END NewOpI;

PROCEDURE NewNodeI(n: tPOINTER);
(* Initialisierungsprozedur fuer Modul 'Type' (Typ: NodeId) *)
VAR N: tNode;
BEGIN
    N:= n;
    WITH N^ DO
        DegValid:= FALSE;
        degree:= 0;
        ResValid:= FALSE;
        add:= TRUE;
        NewOpI(op1);
        NewOpI(op2)
    END
END NewNodeI;

PROCEDURE DelNodeI(n: tPOINTER);
(* Loeschprozedur fuer Modul 'Type' (Typ: NodeId) *)
VAR N: tNode;
BEGIN
    N:= n;
    Type.DelI(OpId, N^.op1);
    Type.DelI(OpId, N^.op2)
END DelNodeI;

PROCEDURE NewSerI(s: tPOINTER);
VAR S: tSer;
    i: LONGCARD;
    op: tOp;
BEGIN
    S:= tSer(s);
    Frag.Use(S, OpId, 0, MaxDeg());
    FOR i:= 0 TO MaxDeg() DO
        op:= Type.NewI(OpId);
        Frag.SetItem(S, i, op)
    END
END NewSerI;

PROCEDURE DelSerI(s: tPOINTER);
VAR S: tSer;
BEGIN
    S:= tSer(s);
    Frag.DontUse(S)
END DelSerI;

(* ------------------------------------ *)
(* Handhabung von Operanden (Typ 'tOp'): *)

PROCEDURE OpAssign(a: tOp; VAR b: tOp);
(* In 'b' muss entweder NIL oder ein initialisierte Operator ueber-
   geben werden. Dieser wird in 'OpAssign' geloescht. In 'b' wird
   eine neu angelegte Kopie von 'a' zurueckgegeben.
*)
BEGIN
    Type.DelI(OpId, b);
    b:= Type.NewI(OpId);

    b^.vOpType:= a^.vOpType;
```

```
    CASE b^.vOpType OF
        constant     : b^.ConstVal:= a^.ConstVal;
      | indeterminate: b^.IndetVal:= a^.IndetVal;
      | node         : b^.NodePos:= a^.NodePos
    END
END OpAssign;

PROCEDURE OpAdd(prog: List.tList; a,b: tOp; VAR res: tOp);
(* An das Ende von 'prog' wird eine Anweisung zur Addition von
   'a' und 'b' angehaengt. Die Position dieser Anweisung innerhalb
   von 'prog' (als Operator gespeichert) wird in 'res' zurueck-
   gegeben. Die Zuweisung der Ergebnisses an 'res' erfolgt wie bei
   'OpAssign'.
   'a', 'b' und 'res' duerfen beim Aufruf identisch sein.
*)
VAR NewNode: tNode; (* neue Anweisung fuer 'prog' *)
BEGIN
   NewNode:= Type.NewI(NodeId);
   OpAssign(a, NewNode^.op1);
   OpAssign(b, NewNode^.op2);
   List.InsertBehind(prog, NewNode);

   Type.DelI(OpId, res);
   res:= Type.NewI(OpId);
   res^.vOpType:= node;
   res^.NodePos:= List.GetPos(prog)
END OpAdd;

PROCEDURE OpMult(prog: List.tList; a,b: tOp; VAR res: tOp);
(* ... analog 'OpAdd', jedoch Multiplikation *)
VAR NewNode: tNode; (* neue Anweisung fuer 'prog' *)
BEGIN
   NewNode:= Type.NewI(NodeId);
   NewNode^.add:= FALSE;
   OpAssign(a, NewNode^.op1);
   OpAssign(b, NewNode^.op2);
   List.InsertBehind(prog, NewNode);

   Type.DelI(OpId, res);
   res:= Type.NewI(OpId);
   res^.vOpType:= node;
   res^.NodePos:= List.GetPos(prog)
END OpMult;

PROCEDURE OpGetDeg(prog: List.tList; op: tOp): LONGCARD;
(* Funktionswert ist der Grad von 'op'. In 'prog' muss das zuge-
   hoerige Programm uebergeben werden. *)
VAR res: LONGCARD;
    cur: List.tPos;
    pred: tNode;
BEGIN
    CASE op^.vOpType OF
        constant:
            res:= 0
      | indeterminate:
            res:= 1
      | node:
            cur:= List.GetPos(prog);
            List.SetPos(prog, op^.NodePos);

            pred:= List.Cur(prog);
            IF NOT pred^.DegValid THEN
                Error("Det.OpGetDeg",
                    "Grad von Vorgaengerknoten unbekannt")
            END;
```

```
            res:= pred^.degree;

            List.SetPos(prog, cur)
    END;
    RETURN res
END OpGetDeg;

PROCEDURE NodeGetCur(prog: List.tList): tNode; FORWARD;
PROCEDURE NodeGetRes(node: tNode): LONGREAL; FORWARD;

PROCEDURE OpGetRes(prog: List.tList; op: tOp): LONGREAL;
(* Funktionsergebnis ist der Wert des Operanden 'op' aus dem
   Programm 'prog'. Falls der Operand ein Anweisungsknoten ist,
   muss dessen Ergebnis bereits bekannt sein.
*)
VAR res: LONGREAL;
    cur: List.tPos;
BEGIN
    CASE op^.vOpType OF
        constant:
            res:= op^.ConstVal
      | indeterminate:
            res:= op^.IndetVal
      | node:
            cur:= List.GetPos(prog);
            List.SetPos(prog, op^.NodePos);

            res:= NodeGetRes(NodeGetCur(prog));

            List.SetPos(prog, cur)
    END;
    RETURN res
END OpGetRes;

PROCEDURE OpGetNode(prog: List.tList; o: tOp): tNode;
(* 'o' muss ein Operand vom mit 'vOpType = node' sein. Er muss
   weiterhin Operand eines Knotens von 'prog' sein. Funktionswert
   ist der Knoten, auf den im Datensatz des Operanden verwiesen
   wird.
*)
VAR cur: List.tPos;
    res: tNode;
BEGIN
    IF o^.vOpType # node THEN
        Error("Det.OpGetNode", "Operand ist kein Knoten")
    END;
    cur:= List.GetPos(prog);

    List.SetPos(prog, o^.NodePos);
    res:= NodeGetCur(prog);

    List.SetPos(prog, cur);
    RETURN res
END OpGetNode;

(* ---------------------------------------- *)
(* Handhabung von Potenzreihen (Typ 'tSer') *)

PROCEDURE TypeNewSerI(VAR a: tSer);
VAR point: tPOINTER;
BEGIN
    point:= Type.NewI(SerId);
    a:= tSer(point)
END TypeNewSerI;
```

```
PROCEDURE TypeDelSerI(VAR a: tSer);
VAR point: tPOINTER;
BEGIN
    point:= tPOINTER(a);
    Type.DelI(SerId, point);
    a:= tSer(point)
END TypeDelSerI;

PROCEDURE SerAssign(a: tSer; VAR b: tSer);
(* ... analog 'OpAssign', jedoch fuer Typ 'tSer' *)
VAR i: LONGCARD;
    NewOp: tOp;
BEGIN
    TypeDelSerI(b);
    TypeNewSerI(b);
    Frag.SetRange(b, 0, MaxDeg());
    NewOp:= NIL;

    FOR i:= 0 TO MaxDeg() DO
        OpAssign(Frag.GetItem(a, i), NewOp);
        Frag.SetItem(b, i, NewOp);
        NewOp:= NIL
    END
END SerAssign;

PROCEDURE SerAdd(prog: List.tList; a,b: tSer; VAR res: tSer);
(* An das Ende von 'prog' wird ein Programmstueck angehaengt, das
   die Summe der Potenzreihen 'a' und 'b' berechnet. Die Summe
   wird in 'res' zurueckgegeben. Die Zuweisung an 'res' erfolgt
   wie bei 'SerAssign'.
   Es werden nur die homogenen Komponenten bis zum Grad 'MaxDeg()'
   berechnet.
   'a', 'b' und 'res' duerfen beim Aufruf identisch sein.
*)
VAR i: LONGCARD;
    NewOp: tOp;
    HilfRes: tSer;
BEGIN
    TypeNewSerI(HilfRes);
    Frag.SetRange(HilfRes, 0, MaxDeg());
    NewOp:= NIL;

    FOR i:= 0 TO MaxDeg() DO
        OpAdd(prog, Frag.GetItem(a,i), Frag.GetItem(b,i), NewOp);
        Frag.SetItem(HilfRes, i, NewOp);
        NewOp:= NIL
    END;

    TypeDelSerI(res);
    res:= HilfRes
END SerAdd;

PROCEDURE SerMult(prog: List.tList; a,b: tSer; VAR res: tSer);
(* ... analog 'SerAdd', jedoch Multiplikation *)
VAR ResComp: LONGCARD;
    SumOp, MultOp: tOp;
    i: LONGCARD;
    HilfRes: tSer;
BEGIN
    TypeNewSerI(HilfRes);
    Frag.SetRange(HilfRes, 0, MaxDeg());

    MultOp:= NIL;

    FOR ResComp:= 0 TO MaxDeg() DO
```

```
        SumOp:= Type.NewI(OpId);
        FOR i:= 0 TO ResComp DO
            OpMult(prog,
                Frag.GetItem(a,i), Frag.GetItem(b, ResComp - i), MultOp
            );
            OpAdd(prog, SumOp, MultOp, SumOp)
        END;
        Frag.SetItem(HilfRes, ResComp, SumOp)
    END;

    Type.DelI(OpId, MultOp);
    TypeDelSerI(res);
    res:= HilfRes
END SerMult;

PROCEDURE SerSetConst(a: tSer; component: LONGCARD; val: LONGREAL);
(* Die Komponente von 'a' mit dem Grad 'component' wird zu einer
   Konstanten mit dem Wert 'val' gemacht. *)
VAR op: tOp;
BEGIN
    op:= Frag.GetItem(a, component);
    op^.vOpType:= constant;
    op^.ConstVal:= val
END SerSetConst;

PROCEDURE SerMultVal(prog: List.tList; a: tSer; val: LONGREAL;
                     VAR res: tSer);
(* An 'prog' wird ein Programmstueck angehaengt, das alle Komponen-
   ten von 'a', mit 'val' multipliziert. Der konstante Term von 'a'
   wird nur dann mit 'val' multipliziert, wenn er ungleich Null ist.
   Das Ergebnis wird in 'res' zurueckgegeben. Die Zuweisung an 'res'
   erfolgt wie bei 'SerAssign'.
*)
VAR ConstOp, MultOp, aOp: tOp;
    r: tSer;
    i, start: LONGCARD;
BEGIN
    ConstOp:= Type.NewI(OpId);
    ConstOp^.ConstVal:= val;
    MultOp:= NIL;
    TypeNewSerI(r);

    aOp:= Frag.GetItem(a, 0);
    IF aOp^.vOpType = constant THEN
        IF aOp^.ConstVal # 0.0 THEN
            OpMult(prog, aOp, ConstOp, MultOp);
            Frag.SetItem(r, 0, MultOp)
        END
    END;

    FOR i:= 1 TO MaxDeg() DO
        MultOp:= NIL;
        aOp:= Frag.GetItem(a, i);
        OpMult(prog, aOp, ConstOp, MultOp);
        Frag.SetItem(r, i, MultOp)
    END;

    Type.DelI(OpId, ConstOp);
    TypeDelSerI(res);
    res:= r
END SerMultVal;

PROCEDURE Divide(prog: List.tList; a: tSer; VAR res: tSer);
(* An das Ende von 'prog' wird ein Programmstueck angehaengt, dass
   das multiplikative Inverse von 'a' berechnet. Es wird in 'res'
```

```
   zurueckgegeben. Die Zuweisung an 'res' erfolgt wie bei
   'SerAssign'.
   In der Prozedur wird davon ausgegangen, dass 'a' die Form '1-g'
   besitzt, wobei 'g' eine Potenzreihe mit Null als konstantem Term
   ist.
   Es werden nur die homogenen Komponenten bis zum Grad 'MaxDeg()'
   beachtet.
*)
VAR power: tSer;
        (* Potenzen von:
           'ser' mit konstantem Term Null und invertiertem
           Vorzeichen *)
    divisor: tSer;
        (* erste Potenz fuer 'Power' *)
    HilfRes: tSer;
    i: LONGCARD;
BEGIN
    divisor:= tSer(NIL);
    SerAssign(a, divisor);
    SerSetConst(divisor, 0, 0.0);
    SerMultVal(prog, divisor, -1.0, divisor);

    power:= tSer(NIL);
    SerAssign(divisor, power);

    TypeNewSerI(HilfRes);
    SerSetConst(HilfRes, 0, 1.0);

    SerAdd(prog, HilfRes, power, HilfRes);

    FOR i:= 2 TO MaxDeg() DO
        SerMult(prog, power, divisor, power);
        SerAdd(prog, HilfRes, power, HilfRes)
    END;

    TypeDelSerI(divisor);
    TypeDelSerI(power);
    TypeDelSerI(res);
    res:= HilfRes
END Divide;

(* ------------------------------------------------------------- *)
(* Handhabung der Programmatrix (Matrix der Zwischenergebnisse): *)

PROCEDURE InitProgMat(ProgMat: Mat.tMat; a: tMat);
(* Die Programmatrix 'ProgMat' wird anhand der Matrix 'a', deren
   Determinante zu berechnen ist, initialisiert.
   Die Programmatrix gibt an, welche Zwischenergebnisse fuer
   Matrizenelemente durch das bisher erzeugte Programm bereits
   berechnet werden. Die Elemente der Programmatrix sind Potenzreihen
   (Typ 'tSer').
*)
VAR WorkSer: tSer;
    Op: tOp; (* naechstes zu initialisierendes Element von
                'WorkSer' *)
    n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    i,j: LONGCARD; (* Schleifenzaehler *)
BEGIN
    n:= Rows(a);
    FOR i:= 1 TO n DO
        FOR j:= 1 TO n DO
            TypeNewSerI(WorkSer);
            Frag.SetRange(WorkSer, 0, n);

            Op:= Type.NewI(OpId);
```

```
            Op^.ConstVal:= 1.0;
            Frag.SetItem(WorkSer, 0, Op);

            Op:= Type.NewI(OpId);
            Op^.vOpType:= indeterminate;
            Op^.IndetVal:= - Elem(a, i, j);
            Frag.SetItem(WorkSer, 1, Op);

            Mat.Set(ProgMat, i, j, tPOINTER(WorkSer))
        END
    END
END InitProgMat;

(* ------------------------------------------------ *)
(* Algorithmusteil 'Anlegen des Berechnungsprogramms' *)
(* ('BuildProgram' und zugehoerige Prozeduren) :      *)

PROCEDURE SubLine(prog: List.tList; ProgMat: Mat.tMat; ThisLine,
                  SubFromLine: LONGCARD; divisor: tSer);
(* In 'ProgMat' wird ein Vielfaches von Zeile 'ThisLine' so zu Zeile
   'SubFromLine' addiert ('SubFromLine' muss groesser sein als
   'ThisLine'), dass in Spalte 'ThisLine' unterhalb der Hauptdiago-
   nalen Nullen entstehen. In 'divisor' muss das inverse Element
   bzgl. der Multiplikation von 'ProgMat_{thisline, thisline}' ueber-
   geben werden.
*)
VAR j, n: LONGCARD;
    accu: tSer;
BEGIN
    n:= Mat.Rows(ProgMat);
    accu:= tSer(NIL);

    FOR j:= ThisLine TO n DO
        SerMult(prog, tSer(Mat.Elem(ProgMat, ThisLine, j)),
                      tSer(Mat.Elem(ProgMat, SubFromLine, j)), accu);
        SerMult(prog, accu, divisor, accu);
        SerMultVal(prog, accu, -1.0, accu);
        Mat.Set(ProgMat, SubFromLine, j, tPOINTER(accu));
        accu:= tSer(NIL)
    END
END SubLine;

PROCEDURE ZerosInColumn(prog: List.tList; ProgMat: Mat.tMat;
                        column: LONGCARD);
(* An das Ende von 'prog' wird ein Programmstueck angehaengt, dass
   durch Subtraktion eines Vielfachen von Zeile 'column' von allen
   folgenden Zeilen in 'ProgMat' in Spalte 'column' unterhalb der
   Hauptdiagonalen Nullen erzeugt.
*)
VAR divisor: tSer;
        (* inverses Element bzgl. der Multiplikation von
           'ProgMat_{column, column}' *)
    i: LONGCARD;
BEGIN
    divisor:= tSer(NIL);
    Divide(prog, tSer(Mat.Elem(ProgMat, column, column)), divisor);
    FOR i:= column+1 TO Mat.Rows(ProgMat) DO
        SubLine(prog, ProgMat, column, i, divisor)
    END
END ZerosInColumn;

PROCEDURE MultMainDiag(prog: List.tList; ProgMat: Mat.tMat;
                       VAR res: tSer);
(* An 'prog' wird ein Programmstueck angehaengt, das die Elemente
   der Hauptdiagonalen von 'ProgMat' miteinander multipliziert.
```

```
   Das Ergebnis wird in 'res' zurueckgegeben. Die Zuweisung an 'res'
   erfolgt wie bei 'SerAssign'.
*)
VAR prod: tSer;
    i, n: LONGCARD;
BEGIN
    prod:= tSer(NIL);
    SerAssign(tSer(Mat.Elem(ProgMat, 1, 1)), prod);

    n:= Mat.Rows(ProgMat);
    FOR i:= 2 TO n DO
        SerMult(prog, prod, tSer(Mat.Elem(ProgMat, i, i)), prod)
    END;

    TypeDelSerI(res);
    res:= prod
END MultMainDiag;

PROCEDURE AddComponents(prog: List.tList; s: tSer);
(* An 'prog' wird ein Programmstueck angehaengt, das die Summe
   der Komponenten von 's' berechnet. *)
VAR sum: tOp;
    i: LONGCARD;
BEGIN
    sum:= NIL;
    OpAssign(Frag.GetItem(s, 0), sum);
    FOR i:= 1 TO MaxDeg() DO
        OpAdd(prog, Frag.GetItem(s, i), sum, sum)
    END;
    Type.DelI(OpId, sum)
END AddComponents;

PROCEDURE BuildProgram(prog: List.tList; a: tMat);
(* 'prog' wird mit den Anweisungen zur Berechnung der Determinante
   von 'a' gefuellt. ( 1. Anweisung am Listenanfang; letzte Anweisung
   ( deren Ergebnis die Determinanten ist) am Listenende )
*)
VAR ProgMat: Mat.tMat; (* Matrix der Zwischenergebnisse der
                          Berechnungen in 'prog' *)
    n, i: LONGCARD;
    det: tSer; (* Determinante von 'a' als Potenzreihe *)
BEGIN
    n:= Rows(a);
    Mat.Use(ProgMat, SerId);
    Mat.SetSize(ProgMat, n, n);
    det:= tSer(NIL);

    InitProgMat(ProgMat, a);
    FOR i:= 1 TO n-1 DO
        ZerosInColumn(prog, ProgMat, i)
    END;
    MultMainDiag(prog, ProgMat, det);
    AddComponents(prog, det);

    TypeDelSerI(det);
    Mat.DontUse(ProgMat)
END BuildProgram;

(* ------------------------------- *)
(* Handhabung von Anweisungsknoten: *)

              (* CURrent *)
PROCEDURE NodeGetCur(prog: List.tList): tNode;
(* Funktionsergebnis ist der aktuelle Anweisungsknoten von 'prog'. *)
BEGIN
```

```
    RETURN tNode(List.Cur(prog))
END NodeGetCur;

PROCEDURE NodeGetRes(node: tNode): LONGREAL;
(* Funktionsergebnis ist das Berechnungsergebnis fuer den Programm-
   knoten 'node'. *)
BEGIN
    IF NOT node^.ResValid THEN
        Error("Det.GetNodeRes",
            "Knotenergebnis wurde noch nicht berechnet")
    END;
    RETURN node^.res
END NodeGetRes;

PROCEDURE NodeExec(prog: List.tList; n: tNode);
(* Fuer den Anweisungsknoten 'n' aus dem Programm 'prog' wird
   das Ergebnis berechnet. Evtl. zu berechnende Ergebnisse der
   Operanden von 'n' muessen bereits bekannt sein.
*)
VAR op1, op2: LONGREAL;
BEGIN
    op1:= OpGetRes(prog, n^.op1);
    op2:= OpGetRes(prog, n^.op2);
    IF n^.add THEN
        n^.res:= op1 + op2
    ELSE
        n^.res:= op1 * op2
    END;
    n^.ResValid:= TRUE
END NodeExec;

PROCEDURE NodeGetDeg(n: tNode): LONGCARD;
(* Funktionsergebnis ist der Grad des angegebenen Knotens. *)
BEGIN
    RETURN n^.degree
END NodeGetDeg;

(* ------------------------------------------------- *)
(* Handhabung der Zwischenergebnisse der Form f(v;w): *)

TYPE tFvwStore = Hash.tHash; (* Speicher fuer Zwischenergebnisse *)
     tFvw = POINTER TO tFvwRec;
     tFvwRec = RECORD (* ein Zwischenergebnis der Form f(v;w)
                         ( siehe Kapitel 'Parallele Berechnung von
                          Termen' ) *)
                  v, w: tNode;
                  val: LONGREAL
               END;
VAR FvwId: Type.Id;

PROCEDURE EquFvwI(a,b: tPOINTER): BOOLEAN;
(* Vergleichsfunktion fuer Modul 'Type' (Typ 'tFvw') *)
VAR A,B: tFvw;
BEGIN
    A:= a; B:= b;
    RETURN (A^.v = B^.v) AND (A^.w = B^.w)
END EquFvwI;

PROCEDURE HashFvwI(a: tPOINTER; size: LONGCARD): LONGCARD;
(* Hash-Funktion fuer Modul 'Type' (Typ 'tFvw') *)
VAR A: tFvw;
BEGIN
    A:= a;
    RETURN ( (LONGCARD(A^.v) MOD size) + (LONGCARD(A^.w) MOD size) )
           MOD size
```

```
END HashFvwI;

PROCEDURE FvwUse(VAR store: tFvwStore);
(* Bevor eine Variable vom Typ 'tFvwStore' benutzt wird, ist fuer
   diese Variable 'FvwUse' zur Initialisierung aufzurufen.
*)
BEGIN
    Hash.Use(store, FvwId, HashSize)
END FvwUse;

PROCEDURE FvwDontUse(VAR store: tFvwStore);
(* Wenn eine Variable vom Typ 'tFvwStore' nicht mehr benutzt werden
   soll, muss 'FvwDontUse' fuer diese Variable aufgerufen werden,
   damit der fuer die Variable angelegte Speicherplatz wieder frei-
   gegeben wird.
*)
BEGIN
    Hash.DontUse(store)
END FvwDontUse;

PROCEDURE FvwEnter(store: tFvwStore; v,w: tNode; val: LONGREAL);
(* Fuer die Knoten 'v' und 'w' wird das Zwischenergebnis 'val' in
   'store' eingetragen.
*)
VAR FvwI: tFvw;
BEGIN
    FvwI:= Type.NewI(FvwId);
    FvwI^.v:= v;
    FvwI^.w:= w;
    FvwI^.val:= val;
    Hash.Insert(store, FvwI);
END FvwEnter;

PROCEDURE FvwGet(store: tFvwStore; v,w: tNode): LONGREAL;
(* Funktionsergebnis ist das in 'store' eingetragene Zwischenergebnis
   fuer die Knoten 'v' und 'w'. Falls fuer die beiden Knoten keine
   Eintragung vorhanden ist, wird 0 zurueckgegeben.
*)
VAR FvwI, SearchI: tFvw;
    found: BOOLEAN;
    res: LONGREAL;
BEGIN
    IF v = w THEN RETURN 1.0 END;

    SearchI:= Type.NewI(FvwId);
    SearchI^.v:= v;
    SearchI^.w:= w;
    IF Hash.Stored(store, SearchI, FvwI) THEN
        res:= FvwI^.val
    ELSE
        res:= 0.0
    END;
    Type.DelI(FvwId, SearchI);

    RETURN res
END FvwGet;

PROCEDURE FvwGetOp(prog: List.tList; store: tFvwStore;
                   v: tNode; w: tOp): LONGREAL;
(* ... analog 'FvwGet', jedoch muss 'w' ein Operator sein *)
VAR wNode: tNode;
    res: LONGREAL;
BEGIN
    CASE w^.vOpType OF
        indeterminate, constant:
```

```
            res:= 0.0
      | node:
            wNode:= OpGetNode(prog, w);
            res:= FvwGet(store, v, wNode)
    END;
    RETURN res
END FvwGetOp;

(* ------------------------------------------------- *)
(* Algorithmusteil 'Ausfuehrung des Berechnungsprogramms' *)
(* ('BuildProgram' und zugehoerige Prozeduren) :          *)

TYPE tContext = RECORD (* Berechnungskontext fuer das auszufuehrende
                          Programm *)
                    AllDone: BOOLEAN;
                        (* TRUE: alle Berechnungen sind durchgefuehrt;
                                 der letzte Programmknoten enthaelt
                                 das Gesamtergebnis *)
                    Fvw: tFvwStore;
                END;

PROCEDURE ContextUse(VAR c: tContext);
(* Bevor eine Variable vom Typ 'tContext' benutzt wird, ist fuer
   diese Variable 'ContextUse' zur Initialisierung aufzurufen.
*)
BEGIN
    c.AllDone:= FALSE;
    FvwUse(c.Fvw)
END ContextUse;

PROCEDURE ContextDontUse(VAR c: tContext);
(* Wenn eine Variable vom Typ 'tContext' nicht mehr benutzt werden
   soll, muss 'ContextDontUse' fuer diese Variable aufgerufen werden,
   damit der fuer die Variable angelegte Speicherplatz wieder frei-
   gegeben wird.
*)
BEGIN
    FvwDontUse(c.Fvw);
END ContextDontUse;

PROCEDURE ComputeDegrees(prog: List.tList);
(* Fuer alle Knoten in 'prog' werden deren Grade berechnet und in
   den Datensaetzen der Knoten gespeichert.
*)
VAR node: tNode;
    hilf: tOp;
    deg1, deg2: LONGCARD;
BEGIN
    List.First(prog);
    WHILE List.MoreData(prog) DO
        node:= NodeGetCur(prog);

        deg1:= OpGetDeg(prog, node^.op1);
        deg2:= OpGetDeg(prog, node^.op2);
        IF deg1 < deg2 THEN
            hilf:= node^.op1;
            node^.op1:= node^.op2;
            node^.op2:= hilf
        END;
        IF node^.add THEN
            node^.degree:= Func.MaxLCard(deg1, deg2)
        ELSE
            node^.degree:= deg1 + deg2
        END;
        node^.DegValid:= TRUE;
```

```
        List.Next(prog)
    END
END ComputeDegrees;

PROCEDURE OneRun(prog: List.tList);
(* Um eine bessere Fehlersuche zu ermoeglichen wird in 'OneRun'
   alternativ zum implementierten Algorithmus das in 'prog'
   enthaltene Programm in einem einzigen Listendurchlauf
   interpretiert.
*)
VAR n: tNode;
BEGIN
    Message("Det.(BGH.)OneRun", "Programmlaenge ");
    WriteCard(List.Count(prog), 0); WriteLn;
    List.First(prog);
    WHILE List.MoreData(prog) DO
        NodeExec(prog, NodeGetCur(prog));
        List.Next(prog)
    END;
    Message("Det.(BGH.)OneRun", "Programmergebnis ");
    WriteReal(NodeGetRes(NodeGetCur(prog)), 15, 6); WriteLn;

    List.First(prog);
    WHILE List.MoreData(prog) DO
        n:= NodeGetCur(prog);
        n^.ResValid:= FALSE;
        n^.res:= 0.0;
        List.Next(prog)
    END;
END OneRun;

PROCEDURE GetVa(prog, Va: List.tList; a: LONGCARD);
(* In 'Va' wird die Liste aller Elemente der Menge 'V_a' fuer das
   Programm 'prog' zurueckgegeben.
   (siehe Kapitel 'Parallele Berechnung von Termen')
*)
VAR cur: List.tPos;
    n: tNode;
BEGIN
    cur:= List.GetPos(prog);

    List.Empty(Va);
    List.First(prog);
    WHILE List.MoreData(prog) DO
        n:= NodeGetCur(prog);
        IF NOT n^.add THEN
            IF (NodeGetDeg(n) > a)
            AND (OpGetDeg(prog, n^.op1) <= a) THEN
                List.InsertBehind(Va, List.Cur(prog))
            END;
        END;
        List.Next(prog)
    END;

    List.SetPos(prog, cur);
END GetVa;

PROCEDURE GetVa1(prog, Va1: List.tList; a: LONGCARD);
(* In 'Va1' wird die Liste aller Elemente der Menge  V'_a  fuer das
   Programm 'prog' zurueckgegeben.
   (siehe Kapitel 'Parallele Berechnung von Termen')
*)
VAR cur: List.tPos;
    n: tNode;
```

```
BEGIN
    cur:= List.GetPos(prog);

    List.Empty(Va1);
    List.First(prog);
    WHILE List.MoreData(prog) DO
        n:= NodeGetCur(prog);
        IF n^.add THEN
            IF (NodeGetDeg(n) > a)
            AND (OpGetDeg(prog, n^.op2) <= a) THEN
                List.InsertBehind(Va1, List.Cur(prog))
            END;
        END;
        List.Next(prog)
    END;

    List.SetPos(prog, cur);
END GetVa1;

PROCEDURE ComputeFw(prog: List.tList; FvwStore: tFvwStore;
                    Va, Va1: List.tList);
(* Es wird mit Hilfe der Knotenlisten 'Va' und 'Va1' sowie der
   Zwischenergebnisse 'FvwStore' das Ergebnis des aktuellen
   Knotens von 'prog' berechnet und im Datensatz des Knotens
   in 'prog' gespeichert.
*)
VAR cur: List.tPos;
    w: tNode;
        (* aktuelles Element von 'prog' *)
    res: LONGREAL; (* f(w) *)
    fu1, fu2, fuw: LONGREAL;
        (* zur Berechnung von 'res' benutzte Ergebnisse vorangegangener
           Rechnungen *)
    u: tNode;
        (* aktuelles Element von 'Va' bzw. 'Va1' *)
    AddList: Reli.tReli;
BEGIN
    Reli.Use(AddList);
    cur:= List.GetPos(prog);
    w:= NodeGetCur(prog);

    Pram.ParallelStart("Det.ComputeFw");
    List.First(Va);
    WHILE List.MoreData(Va) DO
        u:= NodeGetCur(Va);

        fuw:= FvwGet(FvwStore, u, w);
        fu1:= OpGetRes(prog, u^.op1);
        fu2:= OpGetRes(prog, u^.op2);
        Reli.InsertBehind(AddList, fu1 * fu2 * fuw);

        Pram.Prozessoren(1);
        Pram.Schritte(2);
        Pram.NaechsterBlock("Det.ComputeFw");
        List.Next(Va)
    END;

    List.First(Va1);
    WHILE List.MoreData(Va1) DO
        u:= NodeGetCur(Va1);

        fuw:= FvwGet(FvwStore, u, w);
        fu2:= OpGetRes(prog, u^.op2);
        Reli.InsertBehind(AddList, fu2 * fuw );
```

```
        Pram.Prozessoren(1);
        Pram.Schritte(1);
        Pram.NaechsterBlock("Det.ComputeFw");
        List.Next(Va1)
    END;
    Pram.ParallelEnde("Det.ComputeFw");

    res:= Pram.AddList(AddList);
        (* Alle Summanden werden nach der Binaerbaummethode addiert.
           Der Aufwand dafuer wird in 'Pram.AddList' gezaehlt. *)

    List.SetPos(prog, cur);
    w^.res:= res;
    w^.ResValid:= TRUE;
    List.DontUse(AddList);
END ComputeFw;

PROCEDURE GetV(vList, prog: List.tList; from, to: LONGCARD);
(* Fuer den aktuellen Knoten 'w' von 'prog' werden alle potentiellen
   Knoten 'v' aus 'prog' herausgesucht, so dass der Grad von f(v;w)
   mindestens 'from' und hoechstens 'to' betraegt. In 'vList' wird eine
   Liste dieser Knoten zurueckgegeben.
   Es wird nicht geprueft, ob fuer einen bestimmten Knoten 'v' der Wert
   von 'f(v;w)' ungleich Null ist.
*)
VAR cur: List.tPos;
    w: tNode;
        (* aktueller Knoten von 'prog' beim Prozeduraufruf *)
    v: tNode;
    deg: LONGCARD;
        (* Grad von f(v;w) *)
BEGIN
    cur:= List.GetPos(prog);
    w:= NodeGetCur(prog);
    v:= NIL;
    List.Empty(vList);

    List.First(prog);
    REPEAT
        v:= NodeGetCur(prog);
        deg:= NodeGetDeg(w) - NodeGetDeg(v);
        IF (from <= deg) AND (deg <= to) THEN
            List.InsertBehind(vList, List.Cur(prog))
        END;
        List.Next(prog);
    UNTIL NOT List.MoreData(prog) OR (v = w);

    List.SetPos(prog, cur);
END GetV;

PROCEDURE ComputeFvw(vList, prog: List.tList; FvwStore: tFvwStore;
                     from: LONGCARD);
(* Der aktuelle Knoten von 'prog' werde mit 'w' bezeichnet. Fuer
   alle Knoten 'v' in 'vList' werden die Zwischenergebnisse der
   Form  f(v;w)  (siehe Kapitel 'Parallele Berechnung von Termen')
   berechnet und in 'FvwStore' abgelegt. Alle bereits vorher berech-
   neten Ergebnisse muessen in 'FvwStore' abgelegt sein. In 'from'
   muss der Grad uebergeben werden, den die 'f(v;w)' mindestens haben
   muessen. (Der maximale Grad wird automatisch bestimmt.)
*)
VAR cur: List.tPos;
    a: LONGCARD;
    Va, Va1: List.tList;
    w: tNode;
        (* aktueller Knoten von 'prog' beim Prozeduraufruf *)
```

```
    v,u: tNode;
    res: LONGREAL; (* f(v;w) *)
    fu2, fvu1, fuw, fvu2: LONGREAL;
        (* Zwischenergebnisse zur Berechnung von 'res' *)
    AddList: Reli.tReli;
BEGIN
    Reli.Use(AddList);
    cur:= List.GetPos(prog);
    List.Use(Va, NodeId); List.AddRef(Va);
    List.Use(Va1, NodeId); List.AddRef(Va1);
    w:= NodeGetCur(prog);

    List.First(vList);
    WHILE List.MoreData(vList) DO
        List.Empty(AddList);
        v:= NodeGetCur(vList);
        a:= NodeGetDeg(v) + from;
        GetVa(prog, Va, a);
        GetVa1(prog, Va1, a);

        Pram.ParallelStart("Det.ComputeFvw");
        List.First(Va);
        WHILE List.MoreData(Va) DO
             u:= NodeGetCur(Va);

             fuw:= FvwGet(FvwStore, u, w);
             fvu1:= FvwGetOp(prog, FvwStore, v, u^.op1);
             IF (fuw # 0.0) AND (fvu1 # 0.0) THEN
                 fu2:= OpGetRes(prog, u^.op2);
                 Reli.InsertBehind(AddList, fu2 * fvu1 * fuw);

                 Pram.Prozessoren(1);
                 Pram.Schritte(2);
                 Pram.NaechsterBlock("Det.ComputeFvw")
             END;

             List.Next(Va)
        END;

        List.First(Va1);
        WHILE List.MoreData(Va1) DO
             u:= NodeGetCur(Va1);

             fvu2:= FvwGetOp(prog, FvwStore, v, u^.op2);
             fuw:= FvwGet(FvwStore, u, w);
             res:= res + (fvu2 * fuw);

             Pram.Prozessoren(1);
             Pram.Schritte(1);
             Pram.NaechsterBlock("Det.ComputeFvw");
             List.Next(Va1)
        END;
        Pram.ParallelEnde("Det.ComputeFvw");

        res:= Pram.AddList(AddList);
        FvwEnter(FvwStore, v, w, res);
        List.Next(vList)
    END;

    List.DontUse(Va);
    List.DontUse(Va1);
    List.SetPos(prog, cur);
END ComputeFvw;

PROCEDURE PerformStage(prog: List.tList; stage: LONGCARD;
```

```
                    VAR context: tContext);
(* Fuer das Program 'prog' wird Berechnungsstufe 'stage' mit dem
   Berechnungskontext 'context' durchgefuehrt. *)
VAR DegFrom, (* kleinster / ... *)
    DegTo (* ... groesster Grad der f(w) und f(v;w), die in Stufe
             'stage' zu berechnen sind *)
        :LONGCARD;
    deg: LONGCARD;
        (* Grad des aktuellen Knotens *)
    vList: List.tList;
        (* Liste aller v fuer aktuellen Knoten w zur Berechnung von
           f(v;w) *)
    a: LONGCARD;
        (* Knotengrad fuer die Listen V_a und V'_a *)
    Va: List.tList;
        (* Liste der Knoten V_a *)
    Va1: List.tList;
        (* Liste der Knoten V'_a *)
BEGIN
    context.AllDone:= TRUE;
    List.Use(vList, NodeId); List.AddRef(vList);
    List.Use(Va, NodeId);    List.AddRef(Va);
    List.Use(Va1, NodeId);   List.AddRef(Va1);

    IF stage = 0 THEN
        DegFrom:= 1;
        DegTo:= 1
    ELSE
        DegFrom:= LReal2LCard(
                      power( 2.0, LCard2LReal(stage - 1) )
                  ) + 1;
        DegTo:= LReal2LCard(  power( 2.0, LCard2LReal(stage) )  )
    END;

    a:= DegFrom;
    GetVa(prog, Va, a);
    GetVa1(prog, Va1, a);

    Pram.ParallelStart("Det.PerformStage");
    List.First(prog);
    WHILE List.MoreData(prog) DO
        deg:= NodeGetDeg(NodeGetCur(prog));
        IF (DegFrom <= deg) AND (deg <= DegTo) THEN
            context.AllDone:= FALSE;
            ComputeFw(prog, context.Fvw, Va, Va1);
            Pram.NaechsterBlock("Det.PerformStage")
        END
    END;
    Pram.ParallelEnde("Det.PerformStage");

    Pram.ParallelStart("Det.PerformStage:2");
    List.First(prog);
    WHILE List.MoreData(prog) DO
        GetV(vList, prog, DegFrom, DegTo);
        IF List.Count(vList) > 0 THEN
            context.AllDone:= FALSE;
            ComputeFvw(vList, prog, context.Fvw, DegFrom);
            Pram.NaechsterBlock("Det.PerformStage:2");
        END;
        List.Next(prog);
    END;
    Pram.ParallelEnde("Det.PerformStage:2");

    List.DontUse(vList);
    List.DontUse(Va);
```

```
    List.DontUse(Va1)
END PerformStage;

PROCEDURE AllDone(VAR context: tContext): BOOLEAN;
(* Falls der Berechnungskontext 'context' angibt, dass alle Berech-
   nungen fuer das zugrunde liegende Programm durchgefuehrt sind,
   ist das Funktionsergebnis TRUE. *)
BEGIN
    RETURN context.AllDone
END AllDone;

PROCEDURE ExecProgram(prog: List.tList): LONGREAL;
(* Das in 'prog' enthaltene Programm wird ausgefuehrt. Das Ergebnis
   der letzten Anweisung des Programms wird als Funktionswert zurueck-
   gegeben.
*)
VAR stage: LONGCARD;
    context: tContext;
BEGIN
    ContextUse(context);
    ComputeDegrees(prog);

    IF BGHDebug THEN
        OneRun(prog)
    END;

    stage:= 0;
    REPEAT
        PerformStage(prog, stage, context);
        INC(stage)
    UNTIL AllDone(context);

    IF BGHDebug THEN
        Message("Det.(BGH.)ExecProgram",
            "Anzahl der Berechnungsstufen ");
        WriteCard(stage, 0); WriteLn
    END;

    ContextDontUse(context);
    List.Last(prog);
    RETURN NodeGetRes(NodeGetCur(prog))
END ExecProgram;

(* ------------------------------------------------ *)
(* exportierte Prozedur zum Aufruf des Algorithmus: *)

PROCEDURE BGH(a: tMat): LONGREAL;
VAR aCopy: Rema.tMat; (* Arbeitskopie von 'a' *)
    det: LONGREAL; (* Determinate von 'a' *)
    m2:  LONGREAL;
        (* Faktor zur Berechung der Determinante der Ursprungs-
           matrix 'a' aus der Determinante einer konvertieren
           Repraesentation von 'a' *)
    prog: List.tList;
        (* Programm (im Sinne des Kapitels 'Parallele Berechnung
           von Termen) zur Berechnung der Determinate von 'a' *)
BEGIN
    CheckSquare(a,"Det.BGH");
    IF Rows(a) = 1 THEN
        RETURN Elem(a,1,1)
    END;
    vMaxDeg:= Rows(a);
    List.Use(prog, NodeId);
    aCopy:= Rema.Copy(a);
```

```
    ConvertMat(aCopy, m2);
    BuildProgram(prog, aCopy);
    det:= ExecProgram(prog) * m2;

    List.DontUse(prog);
    Rema.DontUse(aCopy);

    RETURN det
END BGH;

(************************************)
(***** Algorithmus von Berkowitz: *****)

PROCEDURE EpsilonMult(VAR res: List.tList; start, MultMat: tMat;
              OneStep, maximum: LONGINT; RightMult: BOOLEAN);
(* Die Matrix 'start' wird so oft mit der Matrix 'MultMat'
   multipliziert, wie 'maximum' angibt. Sowohl das Endergebnis
   als auch alle Zwischenergebnisse und 'start' selbst werden
   in der Liste 'res' zurueckgegeben.
   Bei 'RightMult = TRUE' wird 'start' von rechts mit 'MultMat'
   multipliziert, sonst von links.
   'res' wird in 'EplilonMult' angelegt. 'OneStep' muss angeben,
   wieviele Potenzen von 'MultMat' in einem Schleifendurchlauf
   berechnet werden sollen.
*)
VAR
    Y: Mali.tMali; (* Liste der Potenzen von 'MultMat' *)
    X: Mali.tMali; (* Liste neuer Elemente fuer Z *)
    Z: Mali.tMali; (* potentielles Ergebnis fuer 'res' *)
    ToBePowered: tMat;
        (* Potenz von 'MultMat', die ihrerseits im naechsten
           Schleifendurchlauf so oft potenziert wird, wie
           'OneStep' angibt *)
    i: LONGCARD; (* Schleifenzaehler *)
    hilf: LONGCARD;
BEGIN
    IF OneStep < 1 THEN OneStep:= 1 END;

    Mali.Use(Y);
    Mali.Use(X);
    Mali.Use(Z);

    Mali.InsertBehind( Z, Rema.Copy(start) );
    ToBePowered:= Rema.Copy(MultMat);

    WHILE List.Count(Z) < (LONGCARD(maximum + 1)) DO
        (* berechne Vektor 'Y': *)
        MultLadnerFischer(Y, ToBePowered, OneStep);
        (* Der Aufwand wird in 'MultLadnerFischer' gezaehlt. *)

        (* Die letzte Potenz wird fuer den naechsten Durchlauf
           aufgehoben: *)
        List.Last(Y);
        ToBePowered:= Mali.OutCur(Y);

        (* berechne Vektor 'X' aus 'Z' und 'Y': *)
        List.First(Y);
        Pram.ParallelStart("Det.EpsilonMult");
        REPEAT
            List.First(Z);
            Pram.ParallelStart("Det.EpsilonMult:2");
            REPEAT
                IF RightMult THEN
                    Mali.InsertBehind(X,
                        Rema.CreateMult(
```

```
                            Mali.Cur(Z), Mali.Cur(Y)
                        )
                    )
                ELSE
                    Mali.InsertBehind(X,
                        Rema.CreateMult(
                            Mali.Cur(Y), Mali.Cur(Z)
                        )
                    )
                END;
                (* Der Aufwand wird in 'Rema.CreateMult' gezaehlt. *)

                List.Next(Z);
                Pram.NaechsterBlock("Det.EpsilonMult:2");
            UNTIL NOT List.MoreData(Z);
            Pram.ParallelEnde("Det.EpsilonMult:2");

            List.Next(Y);
            Pram.NaechsterBlock("Det.EpsilonMult");
        UNTIL NOT List.MoreData(Y);
        Pram.ParallelEnde("Det.EpsilonMult");
        List.Empty(Y);

        (* haenge 'X' an 'Z': *)
        List.Last(Z);
        List.First(X);
        REPEAT
            Mali.InsertBehind(Z, Mali.OutCur(X))
        UNTIL (List.Count(X) = 0) OR (List.Count(Z) = LONGCARD(maximum + 1))
    END;

    List.DontUse(Y);
    List.DontUse(X);
    res:= Z
END EpsilonMult;

PROCEDURE GetM(a: tMat; i: LONGCARD): tMat;
(* Funktionsergebnis ist 'M_i' *)
VAR M: tMat; (* Funktionsergebnis *)
    n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    r, c: LONGCARD; (* Schleifenzaehler *)
BEGIN
    n:= Rows(a);
    IF i >= n THEN
        Error("Det.GetM", "M_i existiert nicht (i zu gross)")
    END;
    n:= Rows(a);
    Rema.Use(M, LCard2Card(n-i), LCard2Card(n-i));
    FOR r:= i+1 TO n DO
        FOR c:= i+1 TO n DO
            Rema.Set(M, LCard2Card(r-i), LCard2Card(c-i),
                     Rema.Elem(a, LCard2Card(r), LCard2Card(c)))
        END
    END;
    RETURN M
END GetM;

PROCEDURE GetR(a: tMat; i: LONGCARD): tMat;
(* Funktionsergebnis ist 'R_i' *)
VAR R: tMat; (* Funktionsergebnis *)
    n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    count: LONGCARD; (* Schleifenzaehler *)
BEGIN
    n:= Rows(a);
    IF i >= n THEN
```

```
        Error("Det.GetR", "R_i existiert nicht (i zu gross)")
    END;
    n:= Rows(a);
    Rema.Use(R, 1, n-i);
    FOR count:= i+1 TO n DO
        Rema.Set(R, 1, count - i,
                 Rema.Elem(a, i, count))
    END;
    RETURN R
END GetR;

PROCEDURE ComputeU(VAR U: Mali.tMali; i: LONGCARD; a: tMat);
(* In 'a' muss die Matrix uebergeben werden, deren Determinante
   zu berechnen ist.
   Die Prozedur berechnet den Vektor 'U_i' und gibt ihn
   in 'U' zurueck.
   'U' wird in 'ComputeU' angelegt.
*)
VAR n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    m: LONGCARD;
           (* hoechster Exponent von 'M' im zu berechnenden
              Vektor 'T' *)
    mRoot: LONGCARD; (* \lceil m^{0.5} \rceil *)
    mRootM1: LONGCARD;
    mEpsilon: LONGCARD; (* \lceil m^{epsilon} \rceil *)
    M, R: Rema.tMat; (* 'M_i' und 'R_i' *)
BEGIN
    n:= Rema.Rows(a);
    IF n-i-1 >= 0 THEN
        m:= n - i - 1
    ELSE
        m:= 0
    END;
    mRoot:= LCard2Card( Func.Ceil(power(real(m), 0.5)) );
    mEpsilon:= LCard2Card(
                   Func.Ceil(power( real(m), BerkEpsilon ))
               );
    M:= GetM(a, i);
    R:= GetR(a, i);

    IF mRoot > 0 THEN
        mRootM1:= mRoot - 1
    ELSE
        mRootM1:= 0
    END;
    EpsilonMult(U, R, M, mEpsilon, mRootM1, TRUE);

    Rema.DontUse(M);
    Rema.DontUse(R)
END ComputeU;

PROCEDURE ComputeMPower(a: tMat; i: LONGCARD): tMat;
(* Das Funktionsergebnis ist die Startpotenz von 'M_i' fuer
   die Prozedur 'ComputeV'. *)
VAR mat, res: Rema.tMat;
    count: LONGCARD;
    n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    m: LONGCARD;
           (* hoechster Exponent von 'M' im zu berechnenden
              Vektor 'T' *)
    mRoot: LONGCARD; (* \lceil m^{0.5} \rceil *)
    MatList: Mali.tMali;
BEGIN
    n:= Rows(a);
    IF n-i-1 >= 0 THEN
```

```
        m:= n-i-1
    ELSE
        m:= 0
    END;
    IF m > 0 THEN
        Mali.Use(MatList);
        mat:= GetM(a, i);
        mRoot:= LCard2Card( Func.Ceil(power( real(m), 0.5)) );

        FOR count:= 1 TO mRoot DO
            Mali.InsertBehind(MatList, Rema.Copy(mat))
        END;
        MultMatList(res, MatList, FALSE);
            (* Der Aufwand wird in 'MultMatList' gezaehlt. *)

        Rema.DontUse(mat);
        List.DontUse(MatList)
    ELSE
        Rema.Use(res, n, n);
        Rema.Unit(res)
    END;
    RETURN res
END ComputeMPower;

PROCEDURE GetS(a: tMat; i: LONGCARD): tMat;
(* Funktionsergebnis ist 'S_i' *)
VAR S: tMat; (* Funktionsergebnis *)
    n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    count: LONGCARD; (* Schleifenzaehler *)
BEGIN
    n:= Rows(a);
    IF i >= n THEN
        Error("Det.GetS", "S_i existiert nicht (i zu gross)")
    END;
    n:= Rows(a);
    Rema.Use(S, n-i, 1);
    FOR count:= i+1 TO n DO
        Rema.Set(S, count-i, 1, Rema.Elem(a, count, i))
    END;
    RETURN S
END GetS;

PROCEDURE ComputeV(VAR V: Mali.tMali; i: LONGCARD;
                   a: tMat; start: tMat);
(* In 'a' muss die Matrix uebergeben werden, deren Determinante
   zu berechnen ist.
   Die Prozedur berechnet den Vektor 'V_i' und gibt ihn
   in 'V' zurueck. 'V' wird in 'ComputeV' angelegt.
   In 'start' muss die Startpotenz der Matrix 'M_i' ueber-
   geben werden.
*)
VAR n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    m: LONGCARD;
           (* hoechster Exponent von 'M' im zu berechnenden
              Vektor 'T' *)
    mRoot: LONGCARD; (* \lceil m^{0.5} \rceil *)
    mEpsilon: LONGCARD; (* \lceil m^{epsilon} \rceil *)
    S: tMat; (* 'S_i' *)
BEGIN
    n:= Rows(a);
    IF n-i-1 >= 0 THEN
        m:= n - i - 1
    ELSE
        m:= 0
    END;
```

```
    mRoot:= LCard2Card( Func.Ceil(power( real(m), 0.5)) );
    mEpsilon:= LCard2Card(
                   Func.Ceil(power( real(m), BerkEpsilon ))
               );
    S:= GetS(a, i);

    EpsilonMult(V, S, start, mEpsilon, mRoot, FALSE);

    Rema.DontUse(S)
END ComputeV;

PROCEDURE ComputeNextTElem(U,V: Mali.tMali): LONGREAL;
(* Funktionsergebnis ist das durch die internen Zeiger von 'U' und
   'V' bestimmte naechste Element des Vektors 'T'. Die Inhalte von
   'U' und 'V' werden nicht veraendert.
*)
VAR u, v: tMat;
        (* aktuell in Bearbeitung befindliche Elemente aus
           'U' und 'V' *)
    i: LONGCARD;
        (* Schleifenzaehler *)
    length: LONGCARD;
        (* Anzahl der Elemente (bzw. Spalten) von 'u' *)
    res: LONGREAL;
        (* Funktionsergebnis *)
    AddList: Reli.tReli;
BEGIN
    Reli.Use(AddList);
    u:= Mali.Cur(U);
    v:= Mali.Cur(V);
    List.Next(U);
    IF NOT List.MoreData(U) THEN
        List.First(U);
        List.Next(V);
        IF NOT List.MoreData(V) THEN
            Error("Det.ComputeNextTElem",
                "Vektor V enthaelt zu wenig Elemente")
        END
    END;
    length:= Columns(u);

    FOR i:= 1 TO length DO
        Reli.InsertBehind(AddList, Elem(u,1,i) * Elem(v,i,1))
    END;
    (* Die Schleifendurchlaeufe werden parallel durchgefuehrt: *)
    Pram.Prozessoren(length);
    Pram.Schritte(1);

    res:= Pram.AddList(AddList);
        (* Der weitere Aufwand wird in 'Pram.AddList' gezaehlt. *)

    List.DontUse(AddList);
    RETURN res
END ComputeNextTElem;

PROCEDURE ComputeT(VAR T: Reli.tReli; i: LONGCARD; a: tMat);
(* Fuer die Matrix 'a' wird der Vektor T_i berechnet und als
   Liste von LONGREAL-Zahlen in 'T' zurueckgegeben. 'T' wird in
   'ComputeT' angelegt und initialisiert.
*)
VAR
    U, V: Mali.tMali;
           (* Die Vektoren 'U', und 'V' werden als Listen
              von Matrizen implementiert. *)
    n: LONGCARD;
```

```
        (* Anzahl der Zeilen und Spalten von 'a' *)
    k: LONGCARD; (* Schleifenzaehler *)
    mPower: tMat;
BEGIN
    n:= Rows(a);

    Reli.Use(T);

    IF n-i > 0 THEN (* n-i: Anzahl der Elemente von T *)
        (* Es ist nur etwas zu tun, wenn T nicht leer sein soll: *)

        (* berechne Vektor 'U': *)
        Pram.ParallelStart("Det.ComputeT");
            ComputeU(U, i, a);
        Pram.NaechsterBlock("Det.ComputeT");
            mPower:= ComputeMPower(a, i);
        Pram.ParallelEnde("Det.ComputeT");

        (* berechne Vektor 'V': *)
        ComputeV(V, i, a, mPower);
        Rema.DontUse(mPower);

        (* berechne Vektor 'T' aus 'U' und 'V': *)
        List.First(U); List.First(V);
        Pram.ParallelStart("Det.ComputeT:FOR");
        FOR k:= 1 TO n-i DO
            Reli.InsertBehind(T, ComputeNextTElem(U, V) );
            Pram.NaechsterBlock("Det.ComputeT:FOR")
        END;
        Pram.ParallelEnde("Det.ComputeT:FOR");

        List.DontUse(U);
        List.DontUse(V)
    END
END ComputeT;

PROCEDURE CreateC(VAR C: tMat; a: tMat;
                  index: LONGCARD; T: Reli.tReli);
(* Als 'C' wird die Matrix C_{index} zurueckgegeben. Sie wird
   aus 'T' und 'a' zusammengestellt. In 'a' muss die zugrunde-
   liegende Matrix, deren Determinante zu berechnen ist,
   uebergeben werden und in 'T' der zu C_{index} gehoerige
   Vektor T_{index} (implementiert als Liste von LONGREAL-Zahlen).
   'T' wird in 'CreateC' deinitalisiert.
   Bei 'index = Rows(a)' wird kein Vektor 'T' benoetigt. In
   diesem Fall muss 'NIL' als 'T' uebergeben werden. *)
VAR n: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    i: LONGCARD; (* Schleifenzaehler *)
BEGIN
    n:= Rows(a);
    Rema.Use(C, n-index+2, n-index+1);
    Set(C, 1, 1, -1.0);
    Set(C, 2, 1, Elem(a, index, index) );
    IF index < n THEN
        List.First(T);
        FOR i:= 3 TO Rows(C) DO
            IF NOT List.MoreData(T) THEN
                Error("Det.CreateC",
                    "Vektor T enthaelt zu wenig Elemente");
            END;
            Set(C, i, 1, Reli.OutCur(T) )
        END;
        SetToeplitz(C);
        List.DontUse(T)
    ELSE
```

```
        IF T # List.tList(NIL) THEN
            Error("Det.CreateC",
            "Programmfehler: nicht benoetigter Vektor T uebergeben");
        END
    END
END CreateC;

PROCEDURE Berkowitz(a: tMat): LONGREAL;
VAR t, (* Zaehler der Matrizen T *)
    c: (* Zaehler der Matrizen C *)
       LONGCARD;
    size: LONGCARD; (* Anzahl der Zeilen und Spalten von 'a' *)
    Tlist: List.tList; (* Liste der Vektoren T *)
    Clist: Mali.tMali; (* Liste der Matrizen C *)
        (* 'Tlist' und 'Clist' werden zur besseren Uebersichtlichkeit
           getrennt verwaltet. *)
    WorkMat: tMat; (* Arbeitsmatrix *)
    WorkList: List.tList; (* Arbeitsliste *)
    det: LONGREAL; (* Determinante von 'a' *)
BEGIN
    CheckSquare(a,"Det.Berkowitz");
    IF Rows(a) = 1 THEN
        RETURN Elem(a, 1, 1)
    END;

    size:= Rows(a);
    List.Use( Tlist, ListId );
    Mali.Use( Clist );

    Pram.ParallelStart("Det.Berkowitz");
    FOR t:= 1 TO size-1 DO
        ComputeT( WorkList, t, a );
        List.InsertBehind( Tlist, tPOINTER(WorkList) );
        Pram.NaechsterBlock("Det.Berkowitz")
    END;
    Pram.ParallelEnde("Det.Berkowitz");

    (* Baue anhand der Vektoren T die Matrizen C auf
       ( Da nur Zuweisungen durchgefuehrt werden, wird dafuer
         kein Aufwand fuer eine PRAM in Rechnung gestellt. ):
    *)
    List.First(Tlist);
    FOR c:= 1 TO size DO
        IF (c < size) AND NOT List.MoreData(Tlist) THEN
            Error("Det.Berkowitz",
                "Es wurden nicht alle Vektoren T berechnet.");
        END;
        IF List.MoreData(Tlist) THEN
            CreateC( WorkMat, a, c, Reli.tReli(List.OutCur(Tlist)) )
        ELSE
            CreateC( WorkMat, a, c, Reli.tReli(NIL) )
        END;
        Mali.InsertBehind( Clist, WorkMat )
    END;

    MultMatList(WorkMat, Clist, TRUE);
    IF BerkDebug THEN
        WriteString("*** Det.Berkowitz: Ergebnisvektor:"); WriteLn;
        Rema.Write(WorkMat)
    END;
    det:= Elem(WorkMat, size + 1, 1);

    Rema.DontUse(WorkMat);
    List.DontUse(Clist);
    List.DontUse(Tlist);
```

```
    RETURN det
END Berkowitz;

(*******************************)
(***** Algorithmus von Pan: *****)

PROCEDURE GetKrylovVector(n: LONGCARD): tMat;
(* Als Funktionswert wird der fuer den Algorithmus gewaehlte Krylov-Vektor
   zurueckgegeben. In 'n' muss die gewuenschte Laenge des Vektors
   angegeben werden.
*)
VAR i: LONGCARD;
    z: tMat;
BEGIN
    Rema.Use(z, n, 1);
    FOR i:= 1 TO n DO
        Set(z, i, 1, 1.0)
    END;
    RETURN z
END GetKrylovVector;

PROCEDURE BuildMatrixFromVectors(l: Mali.tMali; VAR v: tMat);
(* In 'l' muss eine Liste gleichgrosser Matrizen 'bzw.' Vektoren
   uebergeben werden. Sie werden zu einer einzigen Matrix zusammenge-
   fasst und in 'v' zurueckgegeben.
   Die Vektoren in 'l' werden spaltenweise nebeneinander angeordnet.
*)
VAR r, c: LONGCARD; (* Zeilen bzw. Spalten von 'v' *)
    i, j: LONGCARD;
BEGIN
    List.First(l);
    r:= Rows( Mali.Cur(l) );
    c:= List.Count(l) * Columns(Mali.Cur(l)) ;
    Rema.Use(v, r, c);
    FOR j:= 1 TO c DO
        FOR i:= 1 TO r DO
            Set(v, i, j, Elem(Mali.Cur(l), i, 1) )
        END;
        List.Next(l)
    END
END BuildMatrixFromVectors;

PROCEDURE ComputeKrylovMatrix(a: tMat; VAR kry, vec: tMat);
(* In 'kry' wird die Krylov-Matrix zu 'a' zurueckgegeben und in
   'vec' der zugehoerige 'n+1'-te iterierte Vektor. *)
VAR old, new: Mali.tMali; (* Listen iterierter Vektoren *)
    power: tMat; (* aktuelle Potenz von 'a' *)
    n: LONGCARD;
        (* Anzahl der Zeilen und Spalten von 'a' *)
BEGIN
    n:= Rows(a);

    Mali.Use(old); Mali.Use(new);
    Rema.Use(power, n, n);

    Rema.Assign(a, power);
    Mali.InsertBehind(old, GetKrylovVector(n));

    WHILE List.Count(old) < n+1 DO
        List.First(old);
        Pram.ParallelStart("Det.ComputeKrylovMatrix");
        REPEAT
            Mali.InsertBehind(new, Rema.CreateMult(power, Mali.Cur(old)));
            List.Next(old);
            Pram.NaechsterBlock("Det.ComputeKrylovMatrix")
```

```
        UNTIL NOT List.MoreData(old);
        Pram.ParallelEnde("Det.ComputeKrylovMatrix");
        Rema.Mult(power, power, power);

        List.Last(old);
        List.First(new);
        WHILE List.MoreData(new) DO
            Mali.InsertBehind(old, Mali.OutCur(new))
        END
    END;

    List.Last(old);
    WHILE List.Count(old) > n+1 DO
        List.DelCur(old)
    END;
    vec:= Mali.OutCur(old);
    BuildMatrixFromVectors(old, kry);

    Rema.DontUse(power);
    List.DontUse(old); List.DontUse(new)
END ComputeKrylovMatrix;

PROCEDURE Transpose(a: tMat; VAR b: tMat);
VAR r,c: LONGCARD;
BEGIN
    Rema.Use(b, Columns(a), Rows(a));
    FOR r:= 1 TO Rows(a) DO
        FOR c:= 1 TO Columns(a) DO
            Set(b, c, r, Elem(a, r, c))
        END
    END
END Transpose;

PROCEDURE ColSum(m: tMat; c: LONGCARD): LONGREAL;
VAR k: LONGCARD;
    res: LONGREAL;
BEGIN
    res:= 0.0;
    FOR k:= 1 TO Rows(m) DO
        res:= res + ABS( Elem(m, k, c) )
    END;
    RETURN res
END ColSum;

PROCEDURE Norm1(m: tMat): LONGREAL;
VAR max: LONGREAL;
    k: LONGCARD;
BEGIN
    max:= ColSum(m, 1);
    FOR k:= 2 TO Columns(m) DO
        max:= Func.MaxReal( max, ColSum(m, k) )
    END;
    RETURN max
END Norm1;

PROCEDURE GetApproximatedInv(mat: tMat; VAR AproxInv: tMat);
VAR t: LONGREAL;
    mult: tMat;
BEGIN
    Transpose(mat, AproxInv);
    mult:= Rema.CreateMult(mat, AproxInv);
    t:= 1.0 / Norm1(mult);
    Rema.DontUse(mult);
    Rema.ScalMult(t, AproxInv, AproxInv)
END GetApproximatedInv;
```

```
PROCEDURE InvertIterative(mat: tMat; VAR inv: tMat);
(* In 'inv' wird die iterativ berechnete Inverse von 'mat' zurueck-
   gegeben. 'inv' wird in 'InvertIterative' neu angelegt. *)
VAR AproxInv: tMat;
    k,loops: LONGCARD;
    unit: tMat;
    bWork: tMat;
BEGIN
    GetApproximatedInv(mat, AproxInv);

    loops:= Func.Ceil( ld(  LCard2LReal(Rows(mat))  ) );

    Rema.Use(unit, Rows(mat), Columns(mat));
    Rema.ScalMult(2.0, unit, unit);
    Rema.Use(bWork, Rows(mat), Columns(mat));
    FOR k:= 1 TO loops DO
        Rema.Mult(AproxInv, mat, bWork);
        Rema.Sub(unit, bWork, bWork);
        Rema.Mult(bWork, AproxInv, AproxInv);
    END;
    Rema.DontUse(bWork);
    Rema.DontUse(unit);

    inv:= AproxInv
END InvertIterative;

PROCEDURE RoundReal(r: LONGREAL): LONGREAL;
BEGIN
    IF r >= 0.0 THEN
        RETURN LInt2LReal(Func.Floor(r + 0.5))
    ELSE
        RETURN LInt2LReal(Func.Ceil(r - 0.5))
    END
END RoundReal;

PROCEDURE RoundElements(a: tMat);
VAR r,c: LONGCARD;
BEGIN
    FOR r:= 1 TO Rows(a) DO
        FOR c:= 1 TO Columns(a) DO
            Set(a, r, c, RoundReal(Elem(a, r, c)))
        END
    END
END RoundElements;

PROCEDURE Pan(a: tMat): LONGREAL;
VAR kry: tMat; (* Krylov-Matrix zu 'a' *)
    vec: tMat; (* 'n+1'-ter iterierter Vektor zu 'a' *)
    inv: tMat; (* Inverse zu 'kry' *)
    koeff: tMat; (* Vektor der Koeffizienten des charakteristischen
                    Polynoms von 'a' *)
    n: LONGCARD;
    det: LONGREAL;
BEGIN
    CheckSquare(a, "Det.Pan");
    IF Rows(a) = 1 THEN
        RETURN Elem(a, 1, 1)
    END;

    n:= Rows(a);
    kry:= tMat(NIL);
    vec:= tMat(NIL);
    inv:= tMat(NIL);
    koeff:= tMat(NIL);
```

```
    ComputeKrylovMatrix(a, kry, vec);
    InvertIterative(kry, inv);
    koeff:= Rema.CreateMult(vec, inv);
    RoundElements(koeff);
    det:= Elem(koeff, n, 1);

    Rema.DontUse(koeff);
    Rema.DontUse(inv);
    Rema.DontUse(vec);
    Rema.DontUse(kry);

    RETURN det
END Pan;

BEGIN
    MatId:= Type.GetId("Rema.tMat");
    ListId:= Type.GetId("List.tList");

    NodeId:= Type.New(TSIZE(tNodeRec));
    Type.SetNewProc(NodeId, NewNodeI);
    Type.SetDelProc(NodeId, DelNodeI);

    OpId:= Type.New(TSIZE(tOpRec));
    Type.SetNewProc(OpId, NewOpI);

    SerId:= Type.Copy(Type.GetId("Frag.tFrag"));
    Type.SetNewProc(SerId, NewSerI);
    Type.SetDelProc(SerId, DelSerI);

    FvwId:= Type.New(TSIZE(tFvwRec));
    Type.SetEquProc(FvwId, EquFvwI);
    Type.SetHashProc(FvwId, HashFvwI)
END Det.
```

A.4 Definitionsmodul 'Pram'

```
DEFINITION MODULE Pram;

(* Laufzeitmessung in PRAM-Schritten und PRAM-Prozessoren

   Dieses Modul erlaubt es, durch Aufruf von Zaehlprozeduren
   festzustellen, wieviele Schritte und Prozessoren eine PRAM
   (Parallel Random Access Machine) zur Abarbeitung des
   jeweiligen Algorithmus benoetigt.
*)

IMPORT List, Cali, Reli;
FROM Reli IMPORT tReli;

CONST MaxBlockName = 32;
          (* Maximale Laenge, die eine an 'ParallelStart', 'NaechsterBlock'
             oder 'ParallelEnde' uebergebene Zeichenkette haben darf *)

PROCEDURE Start();
(* Die internen Zaehler werden fuer eine neue Messung
   initialisiert. *)

PROCEDURE Ende();
(* Es wird 'SchrittEnde' aufgerufen und anschlieend werden
   die Staende des Schritt- und des Prozessorzaehlers
```

```
   ausgewertet. *)

PROCEDURE Schritte(wert: LONGCARD);
(* Der Prozessorzaehler wird ausgewertet und auf Null gesetzt.
   Der Schrittzaehler wird um den angegebenen Wert inkremen-
   tiert. *)

PROCEDURE Prozessoren(wert: LONGCARD);
(* Der Prozessorzaehler fuer den laufenden Schritt wird um den
   angegebenen Wert inkrementiert.
   Falls z. B. ein Schritt mit einem Prozessor gezaehlt werden
   soll ist
       Pram.Prozessoren(1);
       Pram.Schritte(1);
   aufzurufen. Die umgekehrte Reihenfolge ergibt einen
   Schritt mit 0 Prozessoren!
*)

PROCEDURE ParallelStart(Blockname: ARRAY OF CHAR);
(* Dieser Prozeduraufruf markiert den Anfang einer Kette von
   Anweisungsblocks, die zueinander parallel durchgefuehrt werden.
   Sie werden durch 'NaechsterBlock'-Aufrufe voneinander getrennt.

   Die Kette der Anweisungsblocks wird durch 'ParallelEnde' beendet.

   Die 'ParallelStart'-'ParallelEnde' Aufrufe koennen geschachtelt werden.

   'Blockname' dient zur Pruefung der korrekten Blockschachtelung.
   An alle zusammgehoerigen 'ParallelStart - NaechsterBlock - ParallelEnde'
   Aufrufe muessen die gleichen Zeichenketten an 'Blockname' uebergeben
   werden. Die Konstante 'MaxBlockName' gibt die maximal erlaubte Laenge
   fuer die an 'Blockname' uebergebene Zeichenkette an.
*)

PROCEDURE NaechsterBlock(Blockname: ARRAY OF CHAR);
(* siehe 'ParallelStart' *)

PROCEDURE ParallelEnde(Blockname: ARRAY OF CHAR);
(* siehe 'ParallelStart' *)

PROCEDURE GezaehlteSchritte(): LONGCARD;
(* Der Funktionswert ist nach einem 'Start'-Aufruf und vor einem
   'Ende'-Aufruf der aktuelle Wert des Schrittzaehlers.
   Nach einem 'Ende'-Aufruf und vor dem naechsten 'Start'-Aufruf
   ist der Funktionswert die Anzahl der Schritte im vorangegangenen
   'Start'-'Ende'-Block. *)

PROCEDURE GezaehlteProzessoren(): LONGCARD;
(* analog 'GezaehlteSchritte', jedoch fuer den Prozessorenzaehler *)

PROCEDURE AddList(l: tReli): LONGREAL;
(* Die Elemente in 'l' werden nach der Binaerbaummethode addiert.
   Das Ergebnis wird zurueckgegeben. Der Aufwand wird durch Aufruf
   der Zaehlprozeduren protokolliert.
*)

END Pram.
```

A.5 Implementierungsmodul 'Pram'

```
IMPLEMENTATION MODULE Pram;

(* Erklaerungen im Definitionsmodul *)

FROM SYSTEM IMPORT TSIZE;
FROM InOut IMPORT WriteLn, WriteString, WriteCard;
IMPORT Sys, Func, Str, Type, List, Cali, Reli;
FROM Sys IMPORT tPOINTER;
FROM Func IMPORT Message, Error;

TYPE tBlockname = ARRAY [1..MaxBlockName] OF CHAR;
     pBlockname = POINTER TO tBlockname;

VAR
   stack: Cali.tCali;
       (* Stapel von Zaehlerwerten fuer ParallelStart-
          ParallelEnde-Bloecke; es werden immer Paare von
          Prozessoranzahl und Schrittanzahl gespeichert;
          zweimal 0 gibt ein Blockende an *)
   names: List.tList;
       (* Stapel der Blocknamen zur Pruefung korrekter
          Schachtelung *)
   NameId: Type.Id;
       (* Typnummer fuer 'tBlockname' *)
   steps  : LONGCARD; (* Schrittzaehler der Pram *)
   pro    : LONGCARD; (* Prozessorenzaehler der Pram *)
   MaxPro : LONGCARD; (* Maximale Anzahl bisher in einem Schritt
                         beschaeftigter Prozessoren *)
   running: BOOLEAN;  (* TRUE: 'Start' wurde aufgerufen, jedoch
                                'Ende' noch nicht *)

PROCEDURE Push(value: LONGCARD);
(* schiebt 'value' auf den Stapel 'stack' *)
BEGIN
    List.First(stack);
    Cali.InsertBefore(stack,value)
END Push;

PROCEDURE Pop():LONGCARD;
(* liest oberstes Stapelelement von 'stack' *)
BEGIN
    IF List.Count(stack) = 0 THEN
        WriteLn;
        WriteString("*** Pram.Pop:"); WriteLn;
        WriteString("*** Zaehlstapel ist leer"); WriteLn;
        HALT
    END;
    List.First(stack);
    RETURN Cali.OutCur(stack)
END Pop;

PROCEDURE Start();
BEGIN
    steps:= 0;
    pro:= 0;
    MaxPro:= 0;
    running:= TRUE;
    List.Empty(stack)
END Start;

PROCEDURE Ende();
VAR block: pBlockname;
BEGIN
    running:= FALSE;
    IF List.Count(names) # 0 THEN
```

```
        WriteLn;
        WriteString("*** Pram.Ende:"); WriteLn;
        WriteString("*** Blockschachtelung fehlerhaft"); WriteLn;
        WriteString("Blockstapel:"); WriteLn;
        WriteString("###BOTTOM###"); WriteLn;
        List.Last(names);
        WHILE List.MoreData(names) DO
            block:= pBlockname(List.Cur(names));
            WriteString(block^); WriteLn;
            List.Prev(names)
        END;
        HALT
    END;
    IF List.Count(stack) > 0 THEN
        Message("Pram.Ende", "Der Zaehlstapel ist nicht leer.");
        WriteString("*** Groesse des Zaehlstapels: ");
        WriteCard( List.Count(stack), 0);
        WriteLn;
        HALT
    END
END Ende;

PROCEDURE Schritte(wert: LONGCARD);
BEGIN
    IF pro = 0 THEN
        Error("Pram.Schritte","Schritte ohne Prozessoren ?")
    END;
    IF wert= 0 THEN
        Error("Pram.Schritte","0 Schritte zaehlen ?")
    END;
    IF pro > MaxPro THEN
        MaxPro := pro
    END;
    pro:= 0;
    steps:= steps + wert
END Schritte;

PROCEDURE Prozessoren(wert: LONGCARD);
BEGIN
    IF wert = 0 THEN
        Error("Pram.Prozessoren","0 Prozessoren zaehlen ?")
    END;
    pro:= pro + wert
END Prozessoren;

PROCEDURE NamePush(l: List.tList; name: ARRAY OF CHAR);
VAR item: pBlockname;
BEGIN
    item:= pBlockname( Type.NewI(NameId) );
    Str.Assign(item^, name);
    List.First(l);
    List.InsertBefore(l, tPOINTER(item))
END NamePush;

PROCEDURE NameCheck(l: List.tList; name: ARRAY OF CHAR);
VAR item: pBlockname;
BEGIN
    IF List.Count(l) = 0 THEN
        Error("Pram.NameCheck", "Es existiert kein offener Block.");
    END;
    List.First(l);
    item:= pBlockname( List.Cur(l) );
    IF NOT Str.Equal(item^, name) THEN
        Message("Pram.NameCheck:", "Die Blockschachtelung ist fehlerhaft.");
        WriteString("Erwarteter Block: ");
```

```
        WriteString(name); WriteLn;
        WriteString("Gefundener Block: ");
        WriteString(item^); WriteLn;
        HALT
    END
END NameCheck;

PROCEDURE NamePop(l: List.tList);
BEGIN
    List.First(l);
    List.DelCur(l)
END NamePop;

PROCEDURE ParallelStart(Blockname: ARRAY OF CHAR);
BEGIN
    NamePush(names, Blockname);

    (* speichere bisher erzielte Ergebnisse: *)
    Push(MaxPro);
    Push(steps);

    (* markiere Blockanfang: *)
    Push(0);
    Push(0);
    MaxPro:= 0;
    steps:= 0;
    pro:= 0;
END ParallelStart;

PROCEDURE NaechsterBlock(Blockname: ARRAY OF CHAR);
BEGIN
    NameCheck(names, Blockname);

    IF pro # 0 THEN
        WriteLn;
        WriteString("*** Pram.NaechsterBlock:"); WriteLn;
        WriteString("*** Prozessoren ohne Schritte ?");
        WriteLn;
        HALT
    END;
    IF (MaxPro # 0) OR (steps # 0) THEN
        Push(MaxPro);
        Push(steps)
    END;
    MaxPro:= 0;
    steps:= 0
END NaechsterBlock;

PROCEDURE ParallelEnde(Blockname: ARRAY OF CHAR);
VAR
    CurSteps, CurPro: LONGCARD;
        (* gerade vom Stapel gelesene Zaehlerstaende *)
    ParSt, ParPro: LONGCARD;
        (* maximale vom Stapel gelesene Zaehlerstaende *)
BEGIN
    (* laufenden Block nicht vergessen auszuwerten: *)
    IF (steps # 0) OR (MaxPro # 0) THEN
        NaechsterBlock(Blockname)
    END;

    (* Blockschachtelung pruefen: *)
    NameCheck(names, Blockname);
    NamePop(names);

    (* initialisiere: *)
```

```
    ParSt:= 0; ParPro:= 0;

    (* werte parallel ausgefuehrte Blocks aus: *)
    CurSteps:= Pop();
    CurPro:= Pop();
    WHILE (CurSteps # 0) AND (CurPro # 0) DO
        IF CurSteps > ParSt THEN
            ParSt:= CurSteps
        END;
        ParPro:= ParPro + CurPro;
        CurSteps:= Pop();
        CurPro:= Pop()
    END;

    (* verknuepfe Ergebnis der parallelen Blocks mit
       zuvor erzielten Ergebnissen: *)
    CurSteps:= Pop();
    CurPro:= Pop();
    steps:= CurSteps + ParSt;
    IF CurPro > ParPro THEN
       MaxPro:= CurPro
    ELSE
       MaxPro:= ParPro
    END
END ParallelEnde;

PROCEDURE GezaehlteSchritte(): LONGCARD;
BEGIN
    RETURN steps
END GezaehlteSchritte;

PROCEDURE GezaehlteProzessoren(): LONGCARD;
BEGIN
    IF running THEN
        RETURN pro
    ELSE
        RETURN MaxPro
    END
END GezaehlteProzessoren;

PROCEDURE AddList(l: Reli.tReli): LONGREAL;
VAR a, b, res: LONGREAL;
    li: ARRAY [1..2] OF Reli.tReli;
    from, to: CARDINAL;
BEGIN
    li[1]:= l;
    Reli.Use(li[2]);
    from:= 2; to:= 1;

    WHILE List.Count(li[to]) > 1 DO
        to:= 3 - to; from:= 3 - from;
        List.First( li[from] );

        ParallelStart("Pram.AddList");
        REPEAT
            a:= Reli.OutCur( li[from] );
            b:= Reli.OutCur( li[from] );
            Reli.InsertBehind( li[to], a+b);
            Prozessoren(1);
            Schritte(1);
            NaechsterBlock("Pram.AddList");
        UNTIL List.Count(li[from]) < 2;
        ParallelEnde("Pram.AddList");

        IF List.Count(li[from]) = 1 THEN
```

```
            List.First(li[from]);
            Reli.InsertBehind(li[to], Reli.OutCur(li[from]))
        END;
    END;

    IF List.Count(li[to]) >= 1 THEN
        List.First(li[to]);
        res:= Reli.OutCur(li[to]);
    ELSE
        res:= 0.0
    END;

    List.DontUse(li[2]);
    RETURN res
END AddList;

BEGIN
    Cali.Use(stack);
    steps:= 0;
    MaxPro:= 0;

    NameId:= Type.New(TSIZE(tBlockname));
    List.Use(names, NameId)
END Pram.
```

A.6 Programmodul 'algtest'

```
MODULE algtest;

(*  Test der Algorithmen fuer 3*3--Matrizen

In diesem Testprogramm wird keine R"ucksicht auf Anforderungen an
die Implementierung genommen. Insbesondere wird die Parallelisierung
nicht beachtet.

*)

FROM InOut IMPORT ReadString, WriteString, WriteLn, ReadLReal, WriteReal,
                  ReadCard, WriteCard;
IMPORT Files, NumberIO, Text;
FROM MathLib0 IMPORT log, power;

CONST n= CARDINAL(3);
             (* Anzahl der Zeilen und Spalten, die die Testmatrix hat *)
      nMax= n+1;
             (* Anzahl der Zeilen und Spalten, die eine Matrix maximal
                haben kann *)
      myfile= "algtest.inf";
             (* Datei, in der die Testmatrix gespeichert wird
                (damit sie nicht staendig neu eingegeben werden muss) *)

      BerkDebug= FALSE;
             (* TRUE: Algorithmus von 'Berkowitz' liefert Testausgaben *)

      (*******************************************************)
      (* Konstanten fuer den Alg. von Borodin, von zur Gathen *)
      (* und Hopcroft:                                        *)

      SerMax = n;
             (* maximaler Grad der homogenen Komponenten, die fuer die
                Potenzreihen betrachtet werden *)
```

```
      cBGHNoDebug= 3;
             (* Debug-Ebene, fuer die der Alg. keine Testausgaben
                liefert *)
      cBGHDebug= 3;
             (* cBGHDebug < cBGHNoDebug:
                    Alg. liefert Testausgaben *)
      BGHRadius= 1.0;
             (* angenommener Konvergenzradius der Potenzreihen;
                theoretisch ist hier nur der Wert 1.0 richtig *)

      (************************************)
      (* Konstanten fuer den Alg. von Pan: *)

      PanDebug= TRUE;
             (* TRUE: Alg. liefert Testausgaben *)
      PanLoops= 15;
             (* Anzahl der Iterationen zur Verbesserung der
                Naeherungsinversen *)

TYPE tMat = RECORD (* Matrix aus Fliesskommazahlen *)
                r, c: CARDINAL; (* Rows, Columns *)
                v: ARRAY [1..nMax],[1..nMax] OF LONGREAL; (* Values *)
            END;
     tSer = ARRAY [0..SerMax] OF LONGREAL; (* eine Potenzreihe *)
     tSerMat = RECORD (* Matrix aus Potenzreihen *)
                  r, c: CARDINAL; (* Rows, Columns *)
                  v: ARRAY [1..nMax],[1..nMax] OF tSer
               END;

VAR a: tMat; (* Matrix, deren Determinante zu Berechnen ist *)
    eingabe: CARDINAL; (* Befehl des Benutzers an das Programm *)

    BGHDebug: CARDINAL;
             (* BGHDebug < cBGHNoDebug:
                    Algorithmus von 'Borodin, von zur Gathen und
                    Hopcroft' liefert Testausgaben; je kleiner
                    BGHDebug, umso mehr Testausgaben werden geliefert
             *)

(* zur Fehlersuche: *)
PROCEDURE Wait;
(* Die Prozedur wartet auf das Betraetigen der RETURN-Taste *)
VAR dummy: ARRAY [1..3] OF CHAR;
BEGIN
    ReadString(dummy);
END Wait;

PROCEDURE l(line: ARRAY OF CHAR);
(* Prozedur zur vereinfachten Schreibweise *)
BEGIN
    WriteString(line); WriteLn
END l;

PROCEDURE Hilfe;
(* Es wird eine Liste der Eingabemoeglichkeiten (Befehle an das Programm)
   ausgegeben. *)
BEGIN
    l("*** Hilfe ***");
    l("0: Ende");
    l("1: Hilfe");
    l("2: Matrix eingeben");
    l("3: Matrix ausgeben");
    l("4: Alg. v. Csanky");
    l("5: Alg. v. Borodin, von zur Gathen und Hopcroft");
    l("6: Alg. v. Berkowitz");
```

```
    l("7: Alg. v. Pan");
    l("8: Trivialmethode")
END Hilfe;

(*************************************************************************)

PROCEDURE WriteFile(a: tMat);
(* Die Matrix 'a' wird in der durch die Konstante 'myfile' angegebenen
   Datei abgelegt. *)
VAR f: Files.File;
    i,j: CARDINAL;
BEGIN
    Files.Create(f, myfile, Files.writeSeqTxt, Files.replaceOld);
    FOR i:= 1 TO n DO
        FOR j:= 1 TO n DO
            NumberIO.WriteReal(f, a.v[i,j],20,10); Text.WriteLn(f)
        END
    END;
    Files.Close(f);
END WriteFile;

PROCEDURE Exist(name: ARRAY OF CHAR): BOOLEAN;
(* Falls die Datei mit dem Namen 'name' existiert, wird TRUE zurueck-
   gegeben, sonst FALSE. *)
VAR f: Files.File;
    IsPresent: BOOLEAN;
BEGIN
    Files.Open(f, name, Files.readSeqTxt);
    IsPresent:= Files.State(f) >= 0;
    IF IsPresent THEN
        Files.Close(f)
    END;
    RETURN IsPresent
END Exist;

PROCEDURE ReadFile(VAR a: tMat);
(* Die Matrix, die in der Datei abgelegt ist, die die Konstante 'myfile'
   angibt, wird eingelesen und in 'a' zurueckgegeben. *)
VAR f: Files.File;
    i,j: CARDINAL;
    dummy: BOOLEAN;
BEGIN
    a.r:= n; a.c:= n;
    IF Exist(myfile) THEN
        Files.Open(f, myfile, Files.readSeqTxt);
        FOR i:= 1 TO n DO
            FOR j:= 1 TO n DO
                NumberIO.ReadLReal(f, a.v[i,j], dummy)
            END
        END;
        Files.Close(f)
    ELSE
        FOR i:= 1 TO n DO
            FOR j:= 1 TO n DO
                a.v[i,j]:= 0.0
            END
        END
    END
END ReadFile;

PROCEDURE ReadMat(VAR a: tMat);
(* Es wird zeilenweise eine Matrix vom Benutzer eingelesen und in 'a'
   zurueckgegeben. *)
VAR i,j: CARDINAL;
BEGIN
```

```
    FOR i:= 1 TO n DO
        WriteString("Zeile "); WriteCard(i,0); WriteString(":"); WriteLn;
        FOR j:= 1 TO n DO
            ReadLReal(a.v[i,j])
        END;
        WriteLn
    END;
    WriteFile(a)
END ReadMat;

PROCEDURE WriteMat(a: tMat);
(* Die Matrix 'a' wird auf den Bildschirm ausgegeben. *)
VAR i,j: CARDINAL;
BEGIN
    FOR i:= 1 TO a.r DO
        FOR j:= 1 TO a.c DO
            WriteReal(a.v[i,j],20,10); WriteString("   ")
        END;
        WriteLn
    END
END WriteMat;

PROCEDURE Check(prop: BOOLEAN);
(* Bei 'prop = FALSE' wird das Programm abgebrochen. *)
BEGIN
    IF NOT prop THEN
        HALT
    END
END Check;

(*************************************************************************)

PROCEDURE Trace(a: tMat): LONGREAL;
(* Es wird die Spur der Matrix 'a' berechnet und als Funktionswert zurueck-
   gegeben. *)
VAR erg: LONGREAL;
    i: CARDINAL;
BEGIN
    FOR i:= 1 TO a.r DO
        erg:= erg + a.v[i,i]
    END;
    RETURN erg
END Trace;

PROCEDURE MatClear(VAR a: tMat; r, c: CARDINAL);
(* Die Matrix 'a' wird initialisiert (mit Nullen gefuellt). Als neue Anzahl
   der Zeilen wird 'r' festgesetzt. Als neue Anzahl der Spalten wird 'c'
   festgesetzt. *)
VAR i,j: CARDINAL;
BEGIN
    a.r:= r; a.c:= c;
    FOR i:= 1 TO nMax DO
        FOR j:= 1 TO nMax DO
            a.v[i,j]:= 0.0
        END
    END
END MatClear;

PROCEDURE MatAssign(a: tMat; VAR b: tMat);
(* In Matrix 'b' wird der Inhalt von Matrix 'a' zurueckgegeben. *)
VAR i,j: CARDINAL;
BEGIN
    FOR i:= 1 TO nMax DO
        FOR j:= 1 TO nMax DO
            b.v[i,j]:= a.v[i,j]
```

```
        END
    END;
    b.r:= a.r;
    b.c:= a.c;
END MatAssign;

PROCEDURE MatAdd(a,b: tMat; VAR c: tMat);
(* In Matrix 'c' wird die Summe der Matrizen 'a' und 'b' zurueckgegeben. *)
VAR i,j: CARDINAL;
BEGIN
    Check( (a.r = b.r) AND (a.c = b.c) );
    FOR i:= 1 TO a.r DO
        FOR j:= 1 TO a.c DO
            c.v[i,j]:= a.v[i,j] + b.v[i,j]
        END
    END;
    c.r:= a.r; c.c:= a.c
END MatAdd;

PROCEDURE MatSub(a,b: tMat; VAR c: tMat);
(* In 'c' wird die Differenz der Matrizen 'a' und 'b' zurueckgegeben.
   (a - b) *)
VAR i,j: CARDINAL;
BEGIN
    Check( (a.r = b.r) AND (a.c = b.c) );
    FOR i:= 1 TO n DO
        FOR j:= 1 TO n DO
            c.v[i,j]:= a.v[i,j] - b.v[i,j]
        END
    END;
    c.r:= a.r; c.c:= a.c
END MatSub;

PROCEDURE MatUnit(VAR a: tMat; r, c: CARDINAL);
(* In 'a' wird eine Matrix mit 'r' Zeilen und 'c' Spalten zurueckgegeben,
   die in der Hauptdiagonalen Einsen und sonst nur Nullen enthaelt. *)
VAR i,j: CARDINAL;
BEGIN
    a.r:= r; a.c:= c;
    FOR i:= 1 TO nMax DO
        FOR j:= 1 TO nMax DO
            IF i=j THEN
                a.v[i,j]:= 1.0
            ELSE
                a.v[i,j]:= 0.0
            END
        END
    END
END MatUnit;

PROCEDURE MatMult(a,b: tMat; VAR c: tMat);
(* In 'c' wird das Produkt der Matrizen 'a' und 'b' zurueckgegeben. *)
VAR i,j,k: CARDINAL;
BEGIN
    Check( a.c = b.r );
    FOR i:= 1 TO a.r DO
        FOR j:= 1 TO b.c DO
            c.v[i,j]:= 0.0;
            FOR k:= 1 TO a.c DO
                c.v[i,j]:= c.v[i,j] + a.v[i,k] * b.v[k,j]
            END
        END
    END;
    c.r:= a.r; c.c:= b.c
END MatMult;
```

```
PROCEDURE MatXMult(a: tMat; x: LONGREAL; VAR c: tMat);
(* In 'c' wird das Produkt der Matrix 'a' mit der Zahl 'x'
   zurueckgegeben. *)
VAR i,j: CARDINAL;
BEGIN
    FOR i:= 1 TO a.r DO
        FOR j:= 1 TO a.c DO
            c.v[i,j]:= a.v[i,j] * x
        END
    END;
    c.r:= a.r; c.c:= a.c
END MatXMult;

(*************************************************************************)

PROCEDURE Csanky(a: tMat): LONGREAL;
(* Als Funktionswert wird die Determiante der Matrix 'a' berechnet nach
   dem Algorithmus von Csanky (Frame) zurueckgegeben. *)
VAR b0: tMat;
    z: CARDINAL;
    aSubUnit, unit: tMat;
BEGIN
    MatUnit(b0,n,n);
    FOR z:= n-1 TO 1 BY -1 DO
        MatAssign(a, aSubUnit);
        MatUnit(unit,n,n);
        MatXMult(unit, 1.0/LFLOAT(z), unit);
        MatXMult(unit, Trace(a), unit);
        MatSub(aSubUnit, unit, aSubUnit);

        MatMult(b0, aSubUnit, b0)
    END;
    MatMult(a, b0, b0);
    RETURN Trace(b0) / LFLOAT(n)
END Csanky;

(*************************************************************************)

PROCEDURE SerClear(VAR a: tSer);
(* Die Potenzreihe 'a' wird geloescht. *)
VAR k: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
        a[k]:= 0.0
    END
END SerClear;

PROCEDURE SerWrite(a: tSer);
(* Die Potenzreihe 'a' wird auf den Bildschirm ausgegeben. *)
VAR k: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
        WriteCard(k,4); WriteString(": ");
        WriteReal(a[k],20,10);
        IF k MOD 3 = 0 THEN
            WriteLn
        END
    END;
END SerWrite;

PROCEDURE SerMatWrite(VAR a: tSerMat);
(* Die Potenzreihenmatrix 'a' wird interaktiv auf den Bildschirm ausgegeben.
   Der Benutzer kann die Position der Potenzreihe in der Matrix bestimmen,
   die ausgegeben werden soll. Bei einer Eingabe von 0 wird die Prozedur
```

```
   verlassen. *)
VAR i,j: CARDINAL;
BEGIN
    WriteString("SerMatWrite (0: Ende)"); WriteLn;
    LOOP
        WriteString("Zeile ? "); ReadCard(i);
        IF i=0 THEN RETURN END;
        WriteString("Spalte? "); ReadCard(j);
        IF j=0 THEN RETURN END;
        IF (i <= a.r) AND (j <= a.c) THEN
            SerWrite(a.v[i,j])
        END
    END
END SerMatWrite;

PROCEDURE SerEval(ser: tSer): LONGREAL;
(* Die einzelnen Komponenten der Potenzreihe 'ser' werden addiert. Das
   Ergebnis wird als Funktionsergebnis zurueckgegeben. *)
VAR k: CARDINAL;
    res: LONGREAL;
BEGIN
    res:= 0.0;
    FOR k:= 0 TO SerMax DO
        res:= res + ser[k]
    END;
    RETURN res
END SerEval;

PROCEDURE SerAssign(a: tSer; VAR b: tSer);
(* In 'b' wird eine Kopie der Potenzreihe 'a' zurueckgegeben. *)
VAR k: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
        b[k]:= a[k]
    END
END SerAssign;

PROCEDURE SerAdd(a,b: tSer; VAR c: tSer);
(* Die Potenzreihen 'a' und 'b' werden addiert. Das Ergebnis wird in 'c'
   zurueckgegeben. *)
VAR k: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
        c[k]:= a[k] + b[k]
    END;

    (* Fehlersuche: *)
    IF BGHDebug = 0 THEN
        WriteString("SerAdd: 1.Summand ausgewertet: ");
        WriteReal(SerEval(a), 10, 5); WriteLn;
        WriteString("    2. Summand ausgewertet: ");
        WriteReal(SerEval(b), 10, 5); WriteLn;
        WriteString("Ergebnis ausgewertet: ");
        WriteReal(SerEval(c), 10, 5); WriteLn;
        WriteString("    Summe d. ausgewerteten Summanden:");
        WriteReal(SerEval(a) + SerEval(b), 10, 5); Wait;
    END;
END SerAdd;

PROCEDURE SerSub(a,b: tSer; VAR c: tSer);
(* Die Potenzreihen 'a' und 'b' werden voneinander subtrahiert (a - b).
   Das Ergebnis wird in 'c' zurueckgegeben. *)
VAR k: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
```

```
        c[k]:= a[k] - b[k]
    END;

    (* Fehlersuche: *)
    IF BGHDebug = 0 THEN
        WriteString("SerSub: Subtrahend ausgewertet: ");
        WriteReal(SerEval(a), 10, 5); WriteLn;
        WriteString("    Minuend ausgewertet: ");
        WriteReal(SerEval(b), 10, 5); WriteLn;
        WriteString("Ergebnis ausgewertet: ");
        WriteReal(SerEval(c), 10, 5); WriteLn;
        WriteString("    Diff. d. ausgewerteten Operanden:");
        WriteReal(SerEval(a) - SerEval(b), 10, 5); Wait;
    END;
END SerSub;

PROCEDURE SerMult(a,b: tSer; VAR c: tSer);
(* Die Potenzreihen 'a' und 'b' werden miteinander multipliziert. Das
   Ergebnis wird in 'c' zurueckgegeben. *)
VAR k,l: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
        c[k]:= 0.0;
        FOR l:= 0 TO k DO
            c[k]:= c[k] + a[l] * b[k-l]
        END
    END;

    (* Fehlersuche: *)
    IF BGHDebug = 0 THEN
        WriteString("SerMult: 1.Faktor ausgewertet: ");
        WriteReal(SerEval(a), 10, 5); WriteLn;
        WriteString("    2.Faktor ausgewertet: ");
        WriteReal(SerEval(b), 10, 5); WriteLn;
        WriteString("Ergebnis ausgewertet: ");
        WriteReal(SerEval(c), 10, 5); WriteLn;
        WriteString("    Mult. d. ausgewerteten Faktoren:");
        WriteReal(SerEval(a) * SerEval(b), 10, 5); Wait;
    END;
END SerMult;

PROCEDURE SerXMult(a: tSer; x: LONGREAL; VAR b: tSer);
(* Die Potenzreihe 'a' wird mit der Zahl 'x' multipliziert. Das Ergebnis
   wird in 'b' zurueckgegeben. *)
VAR k: CARDINAL;
BEGIN
    FOR k:= 0 TO SerMax DO
        b[k]:= a[k] * x
    END;

    (* Fehlersuche: *)
    IF BGHDebug = 0 THEN
        WriteString("SerXMult: Potenzreihe ausgewertet: ");
        WriteReal(SerEval(a), 10, 5); WriteLn;
        WriteString("    Ergebnis ausgewertet: ");
        WriteReal(SerEval(b), 10, 5);
        WriteString("    Mult. mit ausgewerteter Reihe:");
        WriteReal(SerEval(a) * x, 10, 5); Wait;
    END;
END SerXMult;

PROCEDURE SerDiv(a,b: tSer; VAR c: tSer);
(* Die Potenzreihe 'a' wird durch die Potenzreihe 'b' mit Hilfe des
   Satzes von Taylor dividiert. Dazu muss der konstante Term von 'b'
   gleich '1' sein. *)
```

```
VAR factor, b2, power: tSer;
    k: CARDINAL;
BEGIN
    SerClear(c);

    (* Fehlersuche: *)
    IF BGHDebug <= 1 THEN
        WriteString("Dividend (ausgewertet: ");
        WriteReal(SerEval(a),20,10); WriteString("):"); WriteLn;
        SerWrite(a); Wait;

        WriteString("Divisor (ausgewertet: ");
        WriteReal(SerEval(b),20,10); WriteString("):"); WriteLn;
        SerWrite(b); Wait
    END;

    BGHDebug:= cBGHNoDebug;

    SerAssign(b, factor);
    factor[0]:= 0.0;
    SerXMult(factor, -1.0, factor);

    SerClear(b2);
    b2[0]:= 1.0;

    SerAssign(factor, power);
    SerAdd(b2, power, b2);
    FOR k:= 1 TO SerMax-1 DO
        SerMult(power, factor, power);
        SerAdd(b2, power, b2)
    END;
    SerMult(a,b2,c);

    BGHDebug:= cBGHDebug;

    (* Fehlersuche: *)
    IF BGHDebug <= 1 THEN
        WriteString("Ergebnis der Division:"); WriteLn;
        SerWrite(c); Wait;

        WriteString("als Zahl: "); WriteReal(SerEval(c),20,10);
        Wait;

        WriteString("Reihen ausgewertet und dividiert: ");
        WriteReal(SerEval(a) / SerEval(b),20,10); Wait;
    END
END SerDiv;

PROCEDURE Kronecker(a,b: CARDINAL): LONGREAL;
(* Diese Prozedur implementiert das Kronecker-Delta. *)
BEGIN
    IF a=b THEN
        RETURN 1.0
    ELSE
        RETURN 0.0
    END
END Kronecker;

PROCEDURE Ceil(a: LONGREAL): LONGREAL;
BEGIN
    IF LFLOAT(TRUNC(a)) # a THEN
        IF a > 0.0 THEN
            a:= LFLOAT(TRUNC(a + 1.0))
        ELSE
            a:= LFLOAT(TRUNC(a))
```

```
        END
    END;
    RETURN a
END Ceil;

PROCEDURE Transform(a: tMat; VAR aSer: tSerMat);
(* In 'aSer' wird die der Matrix 'a' entsprechende Potenzreihenmatrix
   zurueckgegeben. *)
VAR i,j: CARDINAL;
    max: LONGREAL;
    LogMax: LONGREAL;
    LineFactor: LONGREAL;
BEGIN
    max:= 0.0;
    FOR i:= 1 TO a.r DO
        FOR j:= 1 TO a.c DO
            IF ABS(a.v[i,j]) > max THEN
                max:= ABS(a.v[i,j])
            END
        END
    END;

    aSer.r:= a.r; aSer.c:= a.c;
    FOR i:= 1 TO aSer.r DO
        FOR j:= 1 TO aSer.c DO
            SerClear(aSer.v[i,j]);
            aSer.v[i,j][0]:= Kronecker(i,j);
            aSer.v[i,j][1]:= (-1.0)
                             * (Kronecker(i,j) - a.v[i,j])
        END
    END;

    (* Fehlersuche: *)
    IF BGHDebug < cBGHNoDebug THEN
        WriteString("Transform: Test der Transformation... ");
        FOR i:= 1 TO aSer.r DO
            FOR j:= 1 TO aSer.c DO
                IF (SerEval(aSer.v[i,j]) * LineFactor) - a.v[i,j] > 0.00001
                THEN
                    WriteString("FEHLERHAFT   ( Element[");
                    WriteCard(i,0); WriteString(","); WriteCard(j,0);
                    WriteString("] )"); WriteLn;
                    WriteString("    erwartet:"); WriteReal(a.v[i,j],20,10);
                    WriteLn;
                    WriteString("    bekommen:");
                    WriteReal( SerEval(aSer.v[i,j]) * LineFactor, 10, 5);
                    HALT;
                END
            END
        END;
        WriteString("OK"); WriteLn
    END;
END Transform;

PROCEDURE ZeroesInColumn(col: CARDINAL; VAR aSer: tSerMat);
(* Mit Hilfe des Gauss'schen Eliminationsverfahrens werden in Spalte 'col'
   von 'aSer' unterhalb der Hauptdiagonalen Nullen "produziert". *)
VAR LineFactor: tSer;
    multiple: tSer;
    i,j : CARDINAL;
BEGIN
    FOR i:= col+1 TO aSer.r DO
        SerDiv(aSer.v[i,col], aSer.v[col,col], LineFactor);
        FOR j:= col TO aSer.c DO
            SerMult(aSer.v[col,j], LineFactor, multiple);
```

```
            SerSub(aSer.v[i,j], multiple, aSer.v[i,j])
        END
    END;

    (* Fehlersuche: *)
    IF BGHDebug <= 2 THEN
        WriteString("ZeroesInColumn: "); SerMatWrite(aSer)
    END
END ZeroesInColumn;

PROCEDURE MultDiag(aSer: tSerMat; VAR res: tSer);
(* In 'res' wird die Potenzreihe zurueckgegeben, die durch Multiplikation
   der Elemente der Hauptdiagonalen der Potenzreihenmatrix 'aSer'
   entsteht. *)
VAR k: CARDINAL;
    res2: LONGREAL;
BEGIN
    SerAssign(aSer.v[1,1], res);
    FOR k:= 2 TO aSer.c DO
        SerMult(res, aSer.v[k,k], res)
    END;

    (* Fehlersuche: *)
    IF BGHDebug <= 2 THEN
        WriteString("MultDiag: erst ausgewertet und dann multipliziert:");
        res2:= SerEval(aSer.v[1,1]);
        FOR k:= 2 TO aSer.c DO
            res2:= res2 * SerEval(aSer.v[k,k])
        END;
        WriteReal(res2, 10, 5); WriteLn;
        WriteString("    Ergebisreihe ausgewertet: ");
        WriteReal(SerEval(res), 10, 5); WriteLn;
    END
END MultDiag;

PROCEDURE BGH(a: tMat): LONGREAL;
(* Determiantenberechnung mit Hilfe des Algorithmus von 'Borodin, von zur
   Gathen und Hopcroft *)
VAR aSer: tSerMat;
    SerDet: tSer;
    k: CARDINAL;
BEGIN
    Transform(a, aSer);
    FOR k:= 1 TO aSer.c - 1 DO
        ZeroesInColumn(k, aSer)
    END;
    MultDiag(aSer, SerDet);

    (* Fehlersuche: *)
    IF BGHDebug <= 2 THEN
        WriteString("Determinate als Potenzreihe: "); WriteLn;
        SerWrite(SerDet); Wait;
    END;

    RETURN SerEval(SerDet)
END BGH;

(*****************************************************************************)

PROCEDURE GetR(a: tMat; i: CARDINAL; VAR r: tMat);
(* In 'r' wird der Vektor zurueckgegeben, den man aus den Elementen der
   'i'-ten Zeile oberhalb Hauptdiagonalen der Matrix 'a' erhaelt. *)
VAR k: CARDINAL;
BEGIN
    MatClear(r, 1, n-i);
```

```
    FOR k:= 1 TO r.c DO
        r.v[1,k]:= a.v[i,k+i]
    END;

    (* Fehlersuche: *)
    IF BerkDebug THEN
        WriteString("R"); WriteCard(i,0); WriteString(":"); WriteLn;
        WriteMat(r); Wait
    END
END GetR;

PROCEDURE GetS(a: tMat; i: CARDINAL; VAR s: tMat);
(* In 's' wird der Vektor zurueckgegeben, den man aus den Elementen
   der 'i'-ten Spalte von 'a' unterhalb der Hauptdiagonalen erhaelt. *)
VAR k: CARDINAL;
BEGIN
    MatClear(s, n-i, 1);
    FOR k:= 1 TO s.r DO
        s.v[k,1]:= a.v[k+i,i]
    END;

    (* Fehlersuche: *)
    IF BerkDebug THEN
        WriteString("S"); WriteCard(i,0); WriteString(":"); WriteLn;
        WriteMat(s); Wait;
    END
END GetS;

PROCEDURE GetM(a: tMat; i: CARDINAL; VAR m: tMat);
(* In 'm' wird die Untermatrix von 'a' zurueckgegeben, die durch
   Streichen der ersten 'i' Zeilen und Spalten entsteht *)
VAR k,l : CARDINAL;
BEGIN
    MatClear(m, n-i, n-i);
    FOR k:= 1 TO m.r DO
        FOR l:= 1 TO m.c DO
            m.v[k,l]:= a.v[k+i,l+i]
        END
    END;

    (* Fehlersuche: *)
    IF BerkDebug THEN
        WriteString("M"); WriteCard(i,0); WriteString(":"); WriteLn;
        WriteMat(m); Wait;
    END
END GetM;

PROCEDURE MakeToeplitz(VAR a: tMat);
(* Ausgehend von den Elementen in der ersten Spalte von 'a' wird diese
   Matrix in eine Toepliz-Matrix ueberfuehrt. *)
VAR i,j: CARDINAL;
BEGIN
    FOR j:= 2 TO a.c DO
        FOR i:= 1 TO a.r DO
            IF i=1 THEN
                a.v[i,j]:= 0.0
            ELSE
                a.v[i,j]:= a.v[i-1,j-1]
            END
        END
    END
END MakeToeplitz;

PROCEDURE GetC(a: tMat; i: CARDINAL; VAR c: tMat);
(* In 'c' wir die Matrix C_i (siehe Beschreibung des Algorithmus von
```

```
   Berkowitz) fuer die Matrix 'a' zurueckgegeben. *)
VAR r, m, s, rm, rms: tMat;
    k: CARDINAL;
BEGIN
    c.r:= n-i+2; c.c:= n-i+1;
    c.v[1,1]:= -1.0;
    c.v[2,1]:= a.v[i,i];
    IF n >= i+1 THEN
        GetR(a, i, r);
        GetS(a, i, s);
        GetM(a, i, m);
        MatAssign(r, rm);
        MatMult(rm, s, rms);
        c.v[3,1]:= rms.v[1,1];
        FOR k:= 1 TO n-i-1 DO
            MatMult(rm, m, rm);
            MatMult(rm, s, rms);
            c.v[k+3,1]:= rms.v[1,1]
        END
    END;
    MakeToeplitz(c);

    (* Fehlersuche: *)
    IF BerkDebug THEN
        WriteString("C"); WriteCard(i,0); WriteString(":"); WriteLn;
        WriteMat(c); Wait;
    END
END GetC;

PROCEDURE Berk(a: tMat): LONGREAL;
(* Determinantenberechnung mit Hilfe des Algorithmus von Berkowitz *)
VAR p: tMat;
    res, c: tMat;
    i: CARDINAL;
BEGIN
    GetC(a, 1, res);

    FOR i:= 2 TO n DO
        GetC(a, i, c);
        MatMult(res, c, res);

        (* Fehlersuche: *)
        IF BerkDebug THEN
            WriteString("Produkt von C1 bis C"); WriteCard(i,0);
            WriteString(":"); WriteLn;
            WriteMat(res); Wait
        END
    END;

    (*  Fehlersuche: *)
    IF BerkDebug THEN
        FOR i:= 1 TO n+1 DO
            WriteReal(res.v[i,1], 0, 0); WriteLn;
        END;
    END;

    RETURN res.v[n+1, 1]
END Berk;

(*************************************************************************)

PROCEDURE IssueKrylovVektor(n: CARDINAL; VAR z: tMat);
(* In 'z' wird der fuer das Verfahren von Krylov zu benutzende Krylov-Vektor
   der Laenge 'n' zurueckgegeben. *)
VAR i: CARDINAL;
```

```
BEGIN
    MatClear(z, n, 1);
    FOR i:= 1 TO n DO
        z.v[i,1]:= 1.0
    END
END IssueKrylovVektor;

PROCEDURE MatSetColumn(from, to: CARDINAL; src: tMat;
                       VAR dest: tMat);
(* Spalte 'from' der Matrix 'src' wird nach Spalte 'to' der Matrix 'dest'
   kopiert. *)
VAR i: CARDINAL;
BEGIN
    Check(src.r = dest.r);
    FOR i:= 1 TO src.r DO
        dest.v[i,to]:= src.v[i,from]
    END
END MatSetColumn;

PROCEDURE CreateKrylovMatrix(a: tMat; VAR kry, zn: tMat);
(* Fuer die Matrix 'a' wird in 'kry' die zugehoerige Krylov-Matrix
   zurueckgegeben. In 'zn' wird der zugehoerige 'n'-te ('a' besitze n
   Zeilen und Spalten) iterierte Vektor zurueckgegeben. *)
VAR z: tMat;
    k: CARDINAL;
BEGIN
    IssueKrylovVektor(a.r, z);
    MatClear(kry, a.r, a.c);
    MatSetColumn(1, 1, z, kry);
    FOR k:= 2 TO a.r DO
        MatMult(a, z, z);
        MatSetColumn(1, k, z, kry)
    END;
    MatMult(a, z, zn);
    IF PanDebug THEN
        WriteString("Krylov-Matrix:"); WriteLn;
        WriteMat(kry); WriteLn;
        WriteString("z_n:"); WriteLn;
        WriteMat(zn); WriteLn;
        Wait
    END
END CreateKrylovMatrix;

PROCEDURE MatTranspose(a: tMat; VAR b: tMat);
(* In 'b' wird die Transponierte der Matrix 'a' zurueckgegeben. *)
VAR i,j: CARDINAL;
BEGIN
    MatClear(b, a.r, a.c);
    FOR i:= 1 TO b.r DO
        FOR j:= 1 TO b.c DO
            b.v[i,j]:= a.v[j,i]
        END
    END
END MatTranspose;

PROCEDURE MatRowSum(m: tMat; r: CARDINAL): LONGREAL;
(* Als Funktionswert wird die Summe der Elemente in Zeile 'r' der Matrix
   'm' zurueckgegeben. *)
VAR j: CARDINAL;
    res: LONGREAL;
BEGIN
    res:= 0.0;
    FOR j:= 1 TO m.c DO
        res:= res + ABS(m.v[r,j])
    END;
```

```
    RETURN res
END MatRowSum;

PROCEDURE MatColSum(m: tMat; c: CARDINAL): LONGREAL;
(* Als Funktionswert wird die Summe der Elemente in Spalte 'c' der Matrix
   'm' zurueckgegeben. *)
VAR i: CARDINAL;
    res: LONGREAL;
BEGIN
    res:= 0.0;
    FOR i:= 1 TO m.r DO
        res:= res + ABS(m.v[i,c])
    END;
    RETURN res
END MatColSum;

PROCEDURE Max(a, b: LONGREAL): LONGREAL;
(* Es wird das Maximum von 'a' und 'b' zurueckgegeben. *)
BEGIN
    IF a > b THEN
        RETURN a
    ELSE
        RETURN b
    END
END Max;

PROCEDURE Mat1Norm(a: tMat): LONGREAL;
(* Funktionsergebnis ist die 1-Norm (maximale Betrags-Spaltensumme)
   der Matrix 'a'. *)
VAR k: CARDINAL;
    cMax: LONGREAL;
BEGIN
    cMax:= MatColSum(a, 1);
    FOR k:= 2 TO a.c DO
        cMax:= Max(cMax, MatColSum(a, k))
    END;
    RETURN cMax
END Mat1Norm;

PROCEDURE MatInftyNorm(a: tMat): LONGREAL;
(* Funktionsergebnis ist die Unendlich-Norm (maximale Betrags-Zeilensumme)
   der Matrix 'a'. *)
VAR k: CARDINAL;
    rMax: LONGREAL;
BEGIN
    rMax:= MatRowSum(a, 1);
    FOR k:= 2 TO a.r DO
        rMax:= Max(rMax, MatRowSum(a, k))
    END;
    RETURN rMax
END MatInftyNorm;

PROCEDURE ErrorMat(a, b: tMat; VAR error: tMat);
(* Fuer die Matrix 'a' und ihre Naeherungsinverse 'b' wird die Fehlermatrix
   berechnet und in 'error' zurueckgegeben. *)
VAR
    unit, ErrMult: tMat;
BEGIN
    MatUnit(unit, b.r, b.c);
    MatMult(b, a, ErrMult);
    MatSub(unit, ErrMult, error)
END ErrorMat;

PROCEDURE GuessInv(m: tMat; VAR inv: tMat);
(* Fuer die Matrix 'm' wird eine Naeherungsinverse berechnet und in 'inv'
```

```
   zurueckgegeben. *)
VAR t: LONGREAL;
    mMult, mTrans, error: tMat;
BEGIN
    MatTranspose(m, mTrans);
    MatMult(mTrans, m, mMult);

    t:= 1.0 / Mat1Norm(mMult);

    MatXMult(mMult, t, inv);
    IF PanDebug THEN
        WriteString("Naeherungsinverse: "); WriteLn;
        WriteMat(inv); WriteLn;

        WriteString("Fehlermatrix: "); WriteLn;
        ErrorMat(m, inv, error);
        WriteMat(error);
        WriteString("1-Norm der Fehlermatrix: ");
        WriteReal(Mat1Norm(error), 20, 10);
        Wait
    END
END GuessInv;

PROCEDURE ImproveGuessed(a, b: tMat; VAR bStern: tMat);
(* Die Naeherungsinverse 'b' der Matrix 'a' wird iterativ verbessert.
   Das Ergebnis wird in 'bStern' zurueckgegeben. *)
VAR k: CARDINAL;
    hilf, unit, error: tMat;
BEGIN
    Check(a.r = a.c);
    MatUnit(unit, a.r, a.c);
    MatXMult(unit, 2.0, unit);
    MatAssign(b, bStern);
    FOR k:= 1 TO PanLoops DO
        (* B_i := (2E_n - B_{i-1}A)B_{i-1} *)
        MatMult(bStern, a, hilf);
        MatSub(unit, hilf, hilf);
        MatMult(hilf, bStern, bStern);
        IF PanDebug THEN
            WriteString("ImproveGuessed: 1-Norm der Fehlermatrix: ");
            ErrorMat(a, bStern, error);
            WriteReal(Mat1Norm(error), 20, 10);
            WriteLn
        END
    END;
END ImproveGuessed;

PROCEDURE Pan(a: tMat): LONGREAL;
(* Determinantenberechnung mit Hilfe des Algorithmus von Pan *)
VAR kry, InvKry, ImprovedKry, zn, c: tMat;
BEGIN
    CreateKrylovMatrix(a, kry, zn);
    GuessInv(kry, InvKry);
    ImproveGuessed(a, InvKry, ImprovedKry);
    MatMult(ImprovedKry, zn, c);
    IF PanDebug THEN
        WriteString("Ergebnisvektor:"); WriteLn;
        WriteMat(c); WriteLn;
    END;
    RETURN c.v[n+1,1]
END Pan;

(*************************************************************************)

PROCEDURE Trivial(a: tMat): LONGREAL;
```

```
(* Es wird die Determinante von 'a' durch Betrachtung aller
   Index-Permutationen berechnet. *)
BEGIN
    CASE n OF
      2: RETURN a.v[1,1] * a.v[2,2] - a.v[2,1] * a.v[1,2]
    | 3: RETURN   a.v[1,1] * a.v[2,2] * a.v[3,3]
                + a.v[1,2] * a.v[2,3] * a.v[3,1]
                + a.v[1,3] * a.v[2,1] * a.v[3,2]
                - a.v[1,1] * a.v[2,3] * a.v[3,2]
                - a.v[1,2] * a.v[2,1] * a.v[3,3]
                - a.v[1,3] * a.v[2,2] * a.v[3,1]
    ELSE
        WriteString("*** Trivial: zu schwierig");
        RETURN 0.0
    END
END Trivial;

(*************************************************************************)

BEGIN
    BGHDebug:= cBGHDebug;

    WriteString("*** Test der Algorithmen fuer 3*3--Matrizen ***");
    ReadFile(a);
    WriteLn;
    REPEAT
        WriteString("Eingabe (0: Ende, 1: Hilfe)? ");
        ReadCard(eingabe); WriteLn;
        CASE eingabe OF
            0: (* tue nichts *)
          | 1: Hilfe
          | 2: ReadMat(a)
          | 3: WriteMat(a)

          | 4: WriteString("*** Alg. v. Csanky ***"); WriteLn;
               WriteReal(Csanky(a),20,10); WriteLn;

          | 5: WriteString("*** Alg. v. Borodin, von zur Gathen");
               WriteString(" und Hopcroft ***"); WriteLn;
               WriteReal(BGH(a),20,10); WriteLn;

          | 6: WriteString("*** Alg. v. Berkowitz ***"); WriteLn;
               WriteReal(Berk(a),20,10); WriteLn;

          | 7: WriteString("*** Alg. v. Pan ***"); WriteLn;
               WriteReal(Pan(a),20,10); WriteLn;

          | 8: WriteString("*** Trivialmethode ***"); WriteLn;
               WriteReal(Trivial(a),20,10); WriteLn;
        ELSE
            WriteString("Befehl unbekannt")
        END
    UNTIL eingabe = 0
END algtest.
```

Anhang B

Unterstützungsmodule

In diesem Kapitel sind alle Module alphabetisch sortiert gesammelt, denen nicht das Hauptinteresse der Implementierung gilt, die jedoch zur Implementierung der Module in Kapitel A erforderlich sind.

B.1 Definitionsmodul 'Cali'

```
DEFINITION MODULE Cali; (* CArdLIst*)

(*
   Listen von ganzen Zahlen (Typ LONGCARD) unter Verwendung
   des Moduls 'List'
*)

IMPORT Sys, Type, Simptype, List;

TYPE tCali= List.tList;

(* Die folgenden Prozeduren entsprechen in ihrer Bedeutung den
   gleichnamigen im Modul 'List', jedoch angepasst auf den
   Typ CARDINAL als Listenelemente. *)

PROCEDURE Use(VAR list: tCali);

PROCEDURE InsertBefore(list: tCali; item: LONGCARD);

PROCEDURE InsertBehind(list: tCali; item: LONGCARD);

PROCEDURE Insert(list: tCali; item: LONGCARD);

PROCEDURE Cur(list: tCali): LONGCARD;

PROCEDURE OutCur(list: tCali): LONGCARD;

(* Fuer 'LONGCARD-Listen' koennen weiterhin die Prozeduren/Funk-
   tionen aus dem Modul 'List' verwendet werden, die nicht
   namensgleich zu den obigen sind.
*)

END Cali.
```

B.2 Implementierungsmodul 'Cali'

```
IMPLEMENTATION MODULE Cali; (* CArdLIst *)

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, Type, Simptype, List;
FROM Sys IMPORT tPOINTER;
FROM Simptype IMPORT CardId, NewCard, DelCard, pCard;

PROCEDURE Use(VAR list: tCali);
BEGIN
    List.Use(list, CardId())
END Use;

PROCEDURE InsertBefore(list: tCali; item: LONGCARD);
VAR ListItem: pCard;
BEGIN
    ListItem:= NewCard(item);
    ListItem^:= item;
    List.InsertBefore(list,ListItem)
END InsertBefore;

PROCEDURE InsertBehind(list: tCali; item: LONGCARD);
VAR ListItem: pCard;
BEGIN
    ListItem:= NewCard(item);
    ListItem^:= item;
    List.InsertBehind(list, ListItem)
END InsertBehind;

PROCEDURE Insert(list: tCali; item: LONGCARD);
BEGIN
    InsertBehind(list, item)
END Insert;

PROCEDURE Cur(list: tCali): LONGCARD;
VAR ListItem: pCard;
BEGIN
    ListItem:= List.Cur(list);
    RETURN ListItem^
END Cur;

PROCEDURE OutCur(list: tCali): LONGCARD;
VAR ListItem: pCard;
    erg     : LONGCARD;
BEGIN
    ListItem:= List.OutCur(list);
    erg     := ListItem^;
    DelCard(ListItem);
    RETURN erg
END OutCur;

END Cali.
```

B.3 Definitionsmodul 'Data'

```
DEFINITION MODULE Data;

(*   Verwaltung der Testdaten der Diplomarbeit
```

```
     Dieses Modul ermoeglicht die Handhabung aller anfallenden
     Testdaten.
     Es wird dabei nicht zwischen im Hauptspeicher stehenden und
     im Hintergrundspeicher (Festplatte) stehenden Daten unter-
     schieden.
     Variablen vom Typ 'Id' (s. u.) koennen bedenkenlos neu
     gesetzt werden. Etwaige Referenzen werden im Modul zusaetz-
     lich verwaltet, so dass auf diese Weise Daten nicht
     verloren gehen.
*)

IMPORT Sys, Str, Type, List, Rema;
FROM Rema IMPORT tMat;

TYPE Id; (* Dies ist der abstrakte Datentyp, mit dessen Hilfe
            die Testdaten bearbeitet werden. Es muss eine
            Variable dieses Typs deklariert werden, die dann
            mit den Prozeduren dieses Moduls bearbeitet
            werden kann. *)

    tAlg = ( laplace, csanky, bgh, berk, pan );
         (* Aufzaehlung der Algorithmen, fuer die Daten
            verwaltet werden. *)

(*************************)
(* allgemeine Verwaltung: *)

PROCEDURE End;
(* Vor dem Beenden des Programms muss diese Prozedur aufgerufen
   werden, sonst kommt es zu Datenverlusten. *)

PROCEDURE Find(VAR dat: Id; name: ARRAY OF CHAR);
(* Diese Prozedur sucht den zu 'name' gehoerigen Datensatz und
   liefert in 'dat' den zugehoerigen Identifikator zurueck.
   Es wird auf jeden Fall ein Datensatz angelegt. Ob er
   neu angelegt wurde kann mit 'NeverUsed' festgestellt werden. *)

PROCEDURE Del(VAR dat: Id); (* DELete *)
(* Der Datensatz 'dat' wird geloescht. *)

PROCEDURE Flush();
(* Diese Prozedur dient zur Steigerung der Datensicherheit.
   Sie bewirkt, dass alle evtl. im Hauptspeicher vorgenommenen
   Aenderungen an Datensaetzen auf den Hintergrundspeicher
   uebertragen werden. *)

PROCEDURE FlushOnly(dat: Id);
(* ... analog 'Flush', jedoch nur fuer den Datensatz 'dat' *)

PROCEDURE HasChanged(dat: Id);
(* Der Datensatz 'dat' wird als veraendert markiert. Mit Hilfe
   dieser Prozedur kann man das Modul ueber Veraenderungen
   an der Matrix des Datensatzes informieren. *)

PROCEDURE ListNames();
(* Vor der Auflistung der gegenwaertig in Bearbeitung befindlichen
   Datensatze mit Hilfe von 'NextName' muss 'ListNames' zur
   Initialisierung der Auflistung aufgerufen werden. *)

PROCEDURE NextName(VAR name: ARRAY OF CHAR): BOOLEAN;
(* Nachdem mit 'ListNames' die Auflistung der Datensatzname
   initialisiert wurde, kann durch wiederholtes Aufrufen von
   'NextName' eine Liste aller gegenwaertig in Bearbeitung
   befindlichen Datensaetze erstellt werden. In 'name' wird
   jeweils der Name des Datensatzes zurueckgegeben. Das
```

```
   Funktionsergebnis ist TRUE, falls ein Datensatz gefunden
   wurde; in diesem Fall wird in 'name' der zugehoerige
   Name zurueckgegeben.
   Die Auflistung erfolgt sortiert.*)


(*****************************)
(* Handhabung der Datensaetze: *)

PROCEDURE Write(dat: Id);
(* Der Datensatz 'dat' wird in die Standardausgabe ausgegeben. *)

PROCEDURE GetMat(dat: Id): tMat;
(* Das Funktionsergebnis ist die Matrix, des Datensatzes 'dat'.
   Sie wird beim Anlegen eines neuen Datensatzes automatisch
   mit angelegt. *)

PROCEDURE SetAlg(dat: Id; alg: tAlg; pdet: LONGREAL;
                 pp, pst: LONGCARD);
(* Im Datensatz 'dat' werden fuer den Algorithmus 'alg' die
   Determinante 'pdet', die Anzahl der Prozessoren 'pp' und
   die Anzahl der Schritte 'pst' gespeichert. *)

PROCEDURE GetAlg(dat: Id; alg: tAlg; VAR pdet: LONGREAL;
                 VAR pp, pst: LONGCARD): BOOLEAN;
(* ... zu 'SetAlg' gehoerige Prozedur zum Lesen aus dem
   Datensatz. Das Funktionsergebnis ist TRUE, falls vorher
   schon mit 'SetAlg' Daten fuer den Algorithmus
   gespeichert wurden. *)

END Data.
```

B.4 Implementierungsmodul 'Data'

```
IMPLEMENTATION MODULE Data;

(*   Verwaltung der Testdaten der Diplomarbeit

     (Erklaerungen im Definitionsmodul)
*)

FROM SYSTEM IMPORT TSIZE;
FROM InOut IMPORT WriteLn, WriteString, WriteReal, WriteCard,
                  ReadLn, ReadString, ReadReal, ReadLReal,
                  ReadCard, ReadLCard;
IMPORT Sys, Str, Type, List, Rema;
FROM Rema IMPORT tMat;
FROM Sys IMPORT tPOINTER;

TYPE Id = POINTER TO tIdRecord;
     tIdRecord = RECORD
                    name: ARRAY [1..16] OF CHAR;
                        (* Name, unter dem der Datensatz
                           gefuehrt wird *)
                    mat: tMat;
                        (* Matrix, fuer die die Determinante
                           berechnet werden soll *)
                    HasChanged: BOOLEAN;
                        (* TRUE: am Datensatz wurden Veraenderungen
                               vorgenommen, die auf den Hinter-
                               grundspeicher zu uebertragen sind *)
                    AlgDat: ARRAY tAlg OF
```

```
                        RECORD  (* Datensatz fuer jeden Alg.: *)
                            det: LONGREAL; (* Determinante *)
                            p, st: LONGCARD;
                                (* Schritte und Prozessoren *)
                            IsSet: BOOLEAN
                                (* TRUE: es wurden Werte in
                                   det, p und st gespeichert *)
                        END
                END;

VAR
    DatList: List.tList;
        (* Liste der sich im Speicher befindlichen Datensaetze *)
    TypeId: Type.Id;
        (* Typidentifikation fuer 'tIdRecord' *)

PROCEDURE NewI(hilf: tPOINTER);
(* Anlege-Prozedur fuer Modul 'Type' *)
VAR
    i: tAlg;
    dat: Id;
BEGIN
    dat:= hilf;
    Str.Empty(dat^.name);
    Rema.Use(dat^.mat, 1, 1);
    dat^.HasChanged:= FALSE;
    FOR i:= laplace TO pan DO
        WITH dat^.AlgDat[i] DO
            det  := 0.0;
            p    := 0;
            st   := 0;
            IsSet:= FALSE
        END
    END
END NewI;

PROCEDURE DelI(hilf: tPOINTER);
(* Loesch-Prozedur fuer Modul 'Type' *)
VAR
    dat: Id;
BEGIN
    dat:= hilf;
    Rema.DontUse(dat^.mat)
END DelI;

PROCEDURE Write(dat: Id);
(* Der Datensatz 'dat' wird auf die Standardausgabe schrieben.
   Die Matrix des Datensatzes wird dabei jedoch nicht geschrieben. *)
VAR i: tAlg;
BEGIN
    WriteString("Datensatzname: "); WriteLn;
    WriteString(dat^.name); WriteLn;
    WriteString("Algorithmus   Determinante    Schritte");
    WriteString("    Prozessoren"); WriteLn;
    FOR i:= laplace TO pan DO
        CASE i OF
            laplace: WriteString("Laplace  ");
          | csanky : WriteString("Csanky   ");
          | bgh    : WriteString("BGH      ");
          | berk   : WriteString("Berkowitz");
          | pan    : WriteString("Pan      ")
        END;
        WriteLn; WriteString("              ");
        WITH dat^.AlgDat[i] DO
            WriteReal(det,12,4);
```

```
            WriteCard(st,12);
            WriteCard(p,15)
        END;
        WriteLn
    END
END Write;

PROCEDURE WriteF(VAR f: Sys.File; dat: Id);
(* ... analog 'Write', jedoch erfolgt die Ausgabe in die
   Datei 'f' *)
VAR
    i: tAlg;
BEGIN
    Sys.WriteString(f, "Datensatzname: "); Sys.WriteLn(f);
    Sys.WriteString(f, dat^.name); Sys.WriteLn(f);
    Sys.WriteString(f, "Algorithmus   Determinante    Schritte");
    Sys.WriteString(f, "    Prozessoren"); Sys.WriteLn(f);
    FOR i:= laplace TO pan DO
        CASE i OF
            laplace: Sys.WriteString(f, "Laplace  ");
          | csanky : Sys.WriteString(f, "Csanky   ");
          | bgh    : Sys.WriteString(f, "BGH      ");
          | berk   : Sys.WriteString(f, "Berkowitz");
          | pan    : Sys.WriteString(f, "Pan      ")
        END;
        Sys.WriteLn(f); Sys.WriteString(f,"              ");
        WITH dat^.AlgDat[i] DO
            Sys.WriteReal(f,det,12,4);
            Sys.WriteCard(f,st,12);
            Sys.WriteCard(f,p,15);
        END;
        Sys.WriteLn(f)
    END
END WriteF;

PROCEDURE ReadF(VAR f: Sys.File; dat: Id);
(* ... analog 'Read', jedoch wird aus 'f' gelesen *)
VAR
    i: tAlg;
BEGIN
    Sys.ReadLn(f);
    Sys.ReadString(f, dat^.name);
    Sys.ReadLn(f);
    FOR i:= laplace TO pan DO
        Sys.ReadLn(f);
        WITH dat^.AlgDat[i] DO
            Sys.ReadReal(f, det);
            Sys.ReadLCard(f, st);
            Sys.ReadLCard(f, p)
        END;
    END
END ReadF;

PROCEDURE End;
BEGIN
    Flush
END End;

PROCEDURE Insert(dat: Id);
(* Der Datensatz 'dat' wird in 'DatList' alphabetisch nach 'name'
   sortiert eingefuegt. *)
VAR
    cur: Id;
    ordered: BOOLEAN;
    inserted: BOOLEAN;
```

```
BEGIN
    IF List.Count(DatList) = 0 THEN
        List.InsertBefore(DatList, dat)
    ELSE
        List.First(DatList);
        inserted:= FALSE;
        REPEAT
            cur:= List.Cur(DatList);
            ordered:= Str.Ordered(cur^.name, dat^.name);
            IF NOT ordered THEN
                List.InsertBefore(DatList, dat);
                inserted:= TRUE
            ELSE
                IF List.AtLast(DatList) THEN
                    List.InsertBehind(DatList, dat);
                    inserted:= TRUE
                END
            END;
            List.Next(DatList)
        UNTIL inserted;
    END
END Insert;

PROCEDURE Search(VAR dat: Id; name: ARRAY OF CHAR): BOOLEAN;
(* Der Datensatz mit dem Namen 'name' wird in 'DatList'
   gesucht. Falls er gefunden wird, ist das Funktionsergebnis
   TRUE und in 'dat' wird der Datensatz zurueckgegeben.
   Wird er nicht gefunden, ist das Funktionsergebnis FALSE und
   'dat' ist undefiniert. *)
VAR
    found: BOOLEAN;
BEGIN
    IF List.Count(DatList) = 0 THEN
        RETURN FALSE
    END;
    List.First(DatList);
    REPEAT
        dat:= List.Cur(DatList);
        found:= Str.Equal(dat^.name, name);
        IF NOT found THEN
            List.Next(DatList)
        END
    UNTIL found OR (List.Cur(DatList) = dat);
    RETURN found
END Search;

PROCEDURE Find(VAR dat: Id; name: ARRAY OF CHAR);
VAR
    f: Sys.File;
BEGIN
    IF NOT Search(dat, name) THEN
        dat:= Type.NewI(TypeId);
        Str.Assign(dat^.name, name);
        IF Sys.Exist(name) THEN
            Sys.OpenRead(f, name);
            ReadF(f, dat);
            Rema.ReadF(f, dat^.mat);
            Sys.Close(f)
        END;
        Insert(dat)
    END
END Find;

PROCEDURE Del(VAR dat: Id); (* DELete *)
VAR
```

```
    ldat: Id;
BEGIN
    IF Search(ldat, dat^.name) THEN
        List.DelCur(DatList)
    ELSE
        WriteString("*** Data.Del:");
        WriteString("*** Datensatz nicht in Datensatzliste;");
        WriteString("*** --> Programmfehler");
        HALT
    END;
    Sys.Delete(dat^.name);
    dat:= NIL
END Del;

PROCEDURE CallFlushOnly( hilf: tPOINTER );
VAR
    dat: Id;
BEGIN
    dat:= hilf;
    FlushOnly(dat)
END CallFlushOnly;

PROCEDURE Flush();
BEGIN
    IF List.Count(DatList) > 0 THEN
        List.Scan(DatList, CallFlushOnly)
    END
END Flush;

PROCEDURE FlushOnly(dat: Id);
VAR
    f: Sys.File;
BEGIN
    IF dat^.HasChanged THEN
        Sys.OpenWrite(f, dat^.name);
        WriteF(f, dat);
        Rema.WriteF(f, dat^.mat);
        dat^.HasChanged:= FALSE;
        Sys.Close(f)
    END
END FlushOnly;

PROCEDURE HasChanged(dat: Id);
BEGIN
    dat^.HasChanged:= TRUE
END HasChanged;

VAR NextPresent: BOOLEAN;

PROCEDURE ListNames();
BEGIN
    List.First(DatList);
    NextPresent:= List.Count(DatList) > 0
END ListNames;

PROCEDURE NextName(VAR name: ARRAY OF CHAR): BOOLEAN;
VAR
    dat: Id;
BEGIN
    IF NextPresent THEN
        dat:= List.Cur(DatList);
        Str.Assign(name, dat^.name);
        List.Next(DatList);
        NextPresent:= List.Cur(DatList) # dat;
        RETURN TRUE
```

```
    ELSE
        RETURN FALSE
    END
END NextName;

(*****************************)
(* Handhabung der Datensaetze: *)

PROCEDURE GetMat(dat: Id): tMat;
BEGIN
    RETURN dat^.mat
END GetMat;

PROCEDURE SetAlg(dat: Id; alg: tAlg; pdet: LONGREAL;
                 pp, pst: LONGCARD);
BEGIN
    WITH dat^.AlgDat[alg] DO
        det:= pdet;
        p:= pp;
        st:= pst;
        IsSet:= TRUE
    END;
    dat^.HasChanged:= TRUE
END SetAlg;

PROCEDURE GetAlg(dat: Id; alg: tAlg; VAR pdet: LONGREAL;
                 VAR pp, pst: LONGCARD): BOOLEAN;
BEGIN
    WITH dat^.AlgDat[alg] DO
        pdet:= det;
        pp:= p;
        pst:= st;
        RETURN IsSet
    END
END GetAlg;

BEGIN
    TypeId:= Type.New(TSIZE(tIdRecord));
    Type.SetName(TypeId,"Data.Id");
    Type.SetNewProc(TypeId,NewI);
    Type.SetDelProc(TypeId,DelI);

    List.Use(DatList, TypeId)
END Data.
```

B.5 Definitionsmodul 'Frag'

```
DEFINITION MODULE Frag; (* array FRAGments *)

(*       Eindimensionale Felder beliebiger Laenge

   Mit diesem Modul koennen Felder veraenderlicher Laenge
   verwaltet werden.

*)

IMPORT Sys, Func, Type;
FROM Sys IMPORT tPOINTER;

TYPE tFrag;
```

```
PROCEDURE Use(VAR f: tFrag; type: Type.Id; low, high: LONGCARD);
(* Vor der Benutzung einer Variablen vom Typ 'tFrag' muss diese
   Prozedur einmal fuer diese Variable aufgerufen werden.
   Die Elemente von 'f' sind vom durch angegebenen Typ 'type',
   der vorher mit Hilfe des Moduls 'Type' vereinbart werden
   muss.
   Die weiteren Parameter entsprechen denen von 'SetRange' und
   'SetType'.
*)

PROCEDURE DontUse(VAR f: tFrag);
(* Wenn eine Variable vom Typ 'tFrag' nie wieder benutzt werden soll
   (besonders bei lokalen Variablen am Ende von Prozeduren, da dann
   der zugehoerige Speicherplatz automatisch freigegeben wird) muss
   diese Prozedur fuer diese Variable einmal aufgerufen werden.
*)

PROCEDURE SetRange(f: tFrag; low, high: LONGCARD);
(* Fuer 'f' wird der Indexbereich neue festgelegt. 'low' bestimmt
   die untere neue Grenze und 'high' die obere. Alle Elemente von
   'f', die aus dem neuen Indexbereich herausfallen werden auto-
   matisch geloescht. (siehe auch 'AddRef')
*)

PROCEDURE SetType(f: tFrag; type: Type.Id);
(* Der Typ der in 'f' zu speichernden Elemente wird festgesetzt.
   'f' wird geloescht.
*)
       (* ADDition REFerences *)
PROCEDURE AddRef(f: tFrag; HasRef: BOOLEAN);
(* Es wird festgelegt, ob auf die Elemente von 'tFrag' zusaetzliche
   Referenzen vorhanden sind und somit beim Loeschen der Elemente
   deren Speicherplatz mit durch Aufruf von 'Type.DelI' freigegeben
   werden soll.
   Bei 'HasRef = FALSE' wird 'Type.DelI' aufgerufen, sonst nicht.
   Voreingestellt ist 'HasRef = FALSE' fuer alle neu angelegten
   Variablen vom Typ 'tFrag'.
*)

PROCEDURE Empty(f: tFrag);
(* Alle Elemente in 'f' werden geloescht. *)

PROCEDURE GetLow(f: tFrag): LONGCARD;
(* Funktionsergebnis ist die untere Indexgrenze von 'f'. *)

PROCEDURE GetHigh(f: tFrag): LONGCARD;
(* Funktionsergebnis ist die obere Indexgrenze von 'f'. *)

PROCEDURE GetType(f: tFrag): Type.Id;
(* Funktionsergebnis ist der Typ der Elemente von 'f'. *)

PROCEDURE GetItem(f: tFrag; index: LONGCARD): tPOINTER;
(* Das Funktionsergebnis ist das im Feld 'f' unter dem angegebenen
   Index gespeicherte Feldelement. *)

PROCEDURE SetItem(f: tFrag; index: LONGCARD; item: tPOINTER);
(* 'SetItem' ist die zu 'GetItem' gehoerige Prozedur zum Setzen
   des Feldinhalts. Ein vorhandenes Feldelement wird automatisch
   geloescht. (siehe auch 'AddRef')
*)

PROCEDURE Transfer(from, to: tFrag);
(* Alle Feldelement von 'from' werden nach 'to' uebertragen. Dazu muss
   der Indexbereich von 'from' innerhalb des Bereiches von 'to'
   liegen. In 'to' bereits vorhandene Feldelemente werden automatisch
```

```
   geloescht. (siehe auch 'AddRef')
*)

END Frag.
```

B.6 Implementierungsmodul 'Frag'

```
IMPLEMENTATION MODULE Frag; (* array FRAGments *)

(*    Eindimensionale Felder beliebiger Laenge

        ( Erklaerungen im Definitionsmodul )
*)

FROM SYSTEM IMPORT ADR, TSIZE;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
IMPORT Sys, Func, Type;
FROM Sys IMPORT tPOINTER;
FROM Func IMPORT Error;

TYPE tFrag = POINTER TO tFragRec;
     tFragRec = RECORD
                    type: Type.Id;
                        (* Typ der Feldelemente *)
                    low, high: LONGCARD;
                        (* untere bzw. obere Indexgrenze *)
                    AddRef: BOOLEAN;
                        (* TRUE: beim Loeschen von Feldelementen
                                 'Type.DelI' nicht aufrufen *)
                    items: tPOINTER
                        (* Zeiger auf Speicherbereich
                           der Groesse
                           '( high-low+1 ) * TSIZE(tPOINTER) )'
                           mit den Feldelementen
                        *)
                END;

VAR FragId: Type.Id;

PROCEDURE ComputeFragBytes(low, high: LONGCARD): LONGCARD;
(* Das Funktionsergebnis ist die Groesse des Speicherbereiches,
   der noetig ist, um Feldelemente (also Zeiger) fuer den
   Indexbereich mit der unteren Grenze 'low' und der oberen
   Grenze 'high' zu speichern.
*)
BEGIN
    RETURN (high - low + 1) * TSIZE(tPOINTER)
END ComputeFragBytes;

PROCEDURE ComputeLoc(p: tPOINTER; dist: LONGCARD): tPOINTER;
(* 'p' wird als Zeiger auf eine Folge hintereinander gespeicherter
   Zeiger betrachtet. Das Funktionsergebnis ist ein Zeiger auf das
   Element dieser Folge, dessen Nummer 'dist' angibt ( Zaehlung be-
   ginnt bei 0; d. h. 'ComputeLoc(p,0) = p' ) .
*)
BEGIN
    RETURN p + TSIZE(tPOINTER) * dist
END ComputeLoc;

PROCEDURE CheckIndices(proc: ARRAY OF CHAR; low, high: LONGCARD);
```

```
(* Die Indizes werden auf Plausibilitaet geprueft. Falls eine
   Fehlermeldung ausgegeben wird, erscheint 'proc' als ausloesende
   Prozedur.
*)
BEGIN
    IF low > high THEN
        Error(proc,
            "untere Indexgrenze groesser als obere Indexgrenze")
    END
END CheckIndices;

PROCEDURE CheckRange(proc: ARRAY OF CHAR;
                     index, low, high: LONGCARD);
BEGIN
    IF (index < low) OR (high < index) THEN
        Error(proc,
    "Der angegebene Index liegt ausserhalb des erlaubten Bereiches.")
    END
END CheckRange;

PROCEDURE Use(VAR frag: tFrag; type: Type.Id; low, high: LONGCARD);
VAR i: LONGCARD;
BEGIN
    CheckIndices("Frag.Use", low, high);
    frag:= Type.NewI(FragId);
    frag^.type:= type;
    frag^.low:= low;
    frag^.high:= high;
    frag^.AddRef:= TRUE;
    ALLOCATE( frag^.items, ComputeFragBytes(low, high) );
    FOR i:= frag^.low TO frag^.high DO
        SetItem(frag, i, NIL)
    END;
    frag^.AddRef:= FALSE
END Use;

PROCEDURE DontUse(VAR f: tFrag);
BEGIN
    Type.DelI(FragId, f);
END DontUse;

PROCEDURE AddRef(f: tFrag; HasRef: BOOLEAN);
BEGIN
    f^.AddRef:= HasRef
END AddRef;

PROCEDURE Empty(f: tFrag);
VAR
    WorkIndex: LONGCARD;
    WorkPointer: POINTER TO tPOINTER;
BEGIN
    CheckIndices("Frac.Empty", f^.low, f^.high);
    WorkIndex:= f^.low;
    WorkPointer:= f^.items;
    REPEAT
        IF NOT f^.AddRef THEN
            Type.DelI(f^.type, WorkPointer^)
                (* 'Type.DelI' achtet selbst auf NIL-Zeiger.
                   Deshalb wird dies hier nicht geprueft. *)
        ELSE
            WorkPointer^:= NIL
        END;
        WorkPointer:= ComputeLoc(WorkPointer, 1);
        INC(WorkIndex)
    UNTIL WorkIndex > f^.high
```

```
END Empty;

PROCEDURE Move(from, to: tFrag);
(* Der Inhalt des Datensatzes von 'from' wird ohne irgendwelche
   Pruefungen an den Datensatz von 'to' zugewiesen. *)
BEGIN
    to^.type:= from^.type;
    to^.low:= from^.low;
    to^.high:= from^.high;
    to^.AddRef:= from^.AddRef;
    to^.items:= from^.items
END Move;

PROCEDURE Swap(a, b: tFrag);
VAR hilf: tFragRec;
    pHilf: tFrag;
BEGIN
    pHilf:= ADR(hilf);
    Move(b, pHilf);
    Move(a, b);
    Move(pHilf, a)
END Swap;

PROCEDURE SetRange(f: tFrag; low, high: LONGCARD);
VAR NewF: tFragRec;
    pNewF: tFrag;
    i: LONGCARD;
    cHilf: LONGCARD;
    pHilf: POINTER TO tPOINTER;
BEGIN
     IF (f^.low # low) OR (f^.high # high) THEN
         (* neues Feld anlegen: *)
         pNewF:= ADR(NewF);
         Move(f, pNewF);
         NewF.AddRef:= TRUE;
         NewF.low:= low;
         NewF.high:= high;
         NewF.items:= NIL;
         ALLOCATE(NewF.items, ComputeFragBytes(low, high));
         FOR i:= low TO high DO
             SetItem(pNewF, i, NIL)
         END;

         (* Inhalt des alten Feldes in das neue kopieren: *)
         Transfer(f, pNewF);

         (* altes Feld beseitigen: *)
         DEALLOCATE(f^.items, ComputeFragBytes(f^.low, f^.high));

         (* Arbeitsdatensatz in uebergebenen Datensatz kopieren: *)
         Move(pNewF, f)
     END;
END SetRange;

PROCEDURE SetType(f: tFrag; type: Type.Id);
BEGIN
    Empty(f);
    f^.type:= type
END SetType;

PROCEDURE GetLow(f: tFrag): LONGCARD;
BEGIN
    RETURN f^.low
END GetLow;
```

```
PROCEDURE GetHigh(f: tFrag): LONGCARD;
BEGIN
    RETURN f^.high
END GetHigh;

PROCEDURE GetType(f: tFrag): Type.Id;
BEGIN
    RETURN f^.type
END GetType;

PROCEDURE GetItem(frag: tFrag; index: LONGCARD): tPOINTER;
VAR loc: POINTER TO tPOINTER;
BEGIN
    CheckRange("Frag.GetItem", index, frag^.low, frag^.high);
    loc:= ComputeLoc(frag^.items, index - frag^.low);
    RETURN loc^
END GetItem;

PROCEDURE SetItem(frag: tFrag; index: LONGCARD; item: tPOINTER);
VAR loc: POINTER TO tPOINTER;
BEGIN
    CheckRange("Frag.SetItem", index, frag^.low, frag^.high);

    loc:= ComputeLoc(frag^.items, index - frag^.low);
    IF frag^.AddRef THEN
        loc^:= NIL
    ELSE
        (* Type.DelI  prueft selbst auf NIL; deshalb wird hier
           loc^ # NIL nicht geprueft *)
        Type.DelI(frag^.type, loc^)
    END;
    loc^:= item
END SetItem;

PROCEDURE Transfer(from, to: tFrag);
VAR i: LONGCARD;
BEGIN
    IF (from^.low < to^.low) OR (from^.high > to^.high) THEN
        Error("Frag.Transfer",
            "Die Indexbereiche sind nicht kompatibel.")
    END;
    IF from^.type # to^.type THEN
        Error("Frag,Transfer",
            "Die Typen sind nicht gleich.")
    END;
    FOR i:= from^.low TO from^.high DO
        SetItem(to, i, GetItem(from, i))
    END
END Transfer;

PROCEDURE DelFragI(i: tPOINTER);
VAR I: tFrag;
BEGIN
    I:= i;
    IF I^.low > I^.high THEN
        Error("Frag.DelFragI",
            "untere Indexgrenze groesser als obere Indexgrenze");
    END;
    Empty(I);
    DEALLOCATE( I^.items, ComputeFragBytes(I^.low, I^.high) )
END DelFragI;

BEGIN
    FragId:= Type.New(TSIZE(tFragRec));
    Type.SetName(FragId, "Frag.tFrag");
```

```
    Type.SetDelProc(FragId, DelFragI)
END Frag.
```

B.7 Definitionsmodul 'Func'

```
DEFINITION MODULE Func; (* procedures and FUNCtions *)

(*  Prozeduren und Funktionen fuer diverse Zwecke  *)

IMPORT SysMath;

PROCEDURE Message(name, text: ARRAY OF CHAR);
(* Es wird eine Meldung ausgegeben, in der 'name' als Name der aus-
   loesenden Funktion und 'text' als Text der Meldung erscheint.
*)

PROCEDURE Error(name, text: ARRAY OF CHAR);
(* Diese Prozedur gibt eine Fehlermeldung aus. In der Meldung
   erscheint 'name' als ausloesende Prozedur und 'text' als
   Text der Meldung.
*)

PROCEDURE MaxCard(a,b: CARDINAL): CARDINAL;
(* Das Funktionsergebnis ist der groessere der beiden
   Werte 'a' und 'b'. *)

PROCEDURE MaxLCard(a,b: LONGCARD): LONGCARD;
(* ... analog 'MaxCard', jedoch fuer LONGCARD *)

PROCEDURE MinCard(a,b: CARDINAL): CARDINAL;
(* Das Funktionsergebnis ist der kleinere er beiden
   Werte 'a' und 'b'. *)

PROCEDURE MinLCard(a,b: LONGCARD): LONGCARD;
(* ... analog 'MinCard', jedoch fuer LONGCARD *)

PROCEDURE MaxReal(a,b: LONGREAL): LONGREAL;
(* ... analog 'MaxCard' ... *)

PROCEDURE Ceil(a: LONGREAL): LONGINT;
(* Das Funktionsergebnis ist die kleinste ganze Zahl 'b', fuer
   die gilt 'b >= a'. *)

PROCEDURE Floor(a: LONGREAL): LONGINT;
(* Das Funktionsergebnis ist die groesste ganze Zahl 'b', fuer
   die gilt 'b <= a'. *)

PROCEDURE ModReal(a,b: LONGREAL): LONGREAL;
(* Das Funktionsergebnis ist der Rest der Division von
   a durch b:
       a - b * real(entier(a/b))  *)

END Func.
```

B.8 Implementierungsmodul 'Func'

```
IMPLEMENTATION MODULE Func;

(*  Prozeduren und Funktionen fuer diverse Zwecke

       ( Erklaerungen im Definitionsmodul )
*)

FROM InOut IMPORT WriteString, WriteLn;
IMPORT SysMath;
FROM SysMath IMPORT LReal2LInt, LInt2LReal;

PROCEDURE Message(name, text: ARRAY OF CHAR);
(* Es wird eine Meldung ausgegeben, in der 'name' als Name der aus-
   loesenden Funktion und 'text' als Text der Meldung erscheint.
*)
BEGIN
    WriteString("*** "); WriteString(name);
    WriteString(":"); WriteLn;
    WriteString("*** "); WriteString(text);
END Message;

PROCEDURE Error(name, text: ARRAY OF CHAR);
(* Diese Prozedur gibt eine Fehlermeldung aus. In der Meldung
   erscheint 'name' als ausloesende Prozedur und 'text' als
   Text der Meldung.
*)
BEGIN
    Message(name, text); WriteLn;
    HALT
END Error;

PROCEDURE MaxCard(a,b: CARDINAL): CARDINAL;
BEGIN
    IF a>b THEN
        RETURN a
    END;
    RETURN b
END MaxCard;

PROCEDURE MaxLCard(a,b: LONGCARD): LONGCARD;
BEGIN
    IF a>b THEN
        RETURN a
    END;
    RETURN b
END MaxLCard;

PROCEDURE MinCard(a,b: CARDINAL): CARDINAL;
BEGIN
    IF a>b THEN
        RETURN b
    END;
    RETURN a
END MinCard;

PROCEDURE MinLCard(a,b: LONGCARD): LONGCARD;
BEGIN
    IF a>b THEN
        RETURN b
    END;
    RETURN a
END MinLCard;

PROCEDURE MaxReal(a,b: LONGREAL): LONGREAL;
```

```
BEGIN
    IF a>b THEN
        RETURN a
    END;
    RETURN b
END MaxReal;

PROCEDURE Ceil(a: LONGREAL): LONGINT;
BEGIN
    IF (a < 0.0) OR (a = LInt2LReal(LReal2LInt(a))) THEN
        RETURN LReal2LInt(a)
    END;
    RETURN LReal2LInt(a + 1.0)
END Ceil;

PROCEDURE Floor(a: LONGREAL): LONGINT;
BEGIN
    IF (a >= 0.0) OR (a = LInt2LReal(LReal2LInt(a))) THEN
        RETURN LReal2LInt(a)
    END;
    RETURN LReal2LInt(a - 1.0)
END Floor;

PROCEDURE ModReal(a,b: LONGREAL): LONGREAL;
BEGIN
    RETURN a - b * LInt2LReal(LReal2LInt(a/b))
END ModReal;

END Func.
```

B.9 Definitionsmodul 'Hash'

```
DEFINITION MODULE Hash;

(*              Hashing

    ( auch bekannt als 'Streuspeicherung' oder
      'Schluesseltransformation';
      siehe z. B.
          N. Wirth,
          Algorithmen und Datenstrukturen mit Modula-2,
          Teubner Verlag, Stuttgart,
          S. 277 ff.
    )

    Das Modul 'Hash' arbeitet mit dem Modul 'Type' zusammen.
    Es koennen Zeiger auf Objekte gespeichert werden, deren Typ
    in 'Type' definiert worden ist.
*)

IMPORT Sys, SysMath, Func, Type, List, Cali, Frag;
FROM Sys IMPORT tPOINTER;

TYPE tHash; (* Ein Speicher, in dem Daten mit dem Modul 'Hash' ge-
               speichert werden sollen, muss von diesem Typ sein. *)

PROCEDURE Use(VAR h: tHash; type: Type.Id; size: LONGCARD);
(* Vor der Benutzung einer Variablen vom Typ 'tFrag' muss diese
   Prozedur einmal fuer diese Variable aufgerufen werden.
   Die Elemente von 'f' sind vom durch angegebenen Typ 'type',
```

```
   der vorher mit Hilfe des Moduls 'Type' vereinbart werden
   muss.
   Fuer den Typ 'type' muessen dem Modul 'Type' eine Hash-Funktion
   ( 'tHashProc' ) und eine Vergleichsfunktion ( 'tEquProc' )
   bekannt sein.
   In 'size' ist die Kapazitaet von 'h' gemessen in Anzahl der
   gespeicherten Elemente anzugeben. Falls der Platz nicht ausreichen
   sollte, wird er automatisch in Schritte der Groesse 'size'
   erweitert.
*)

PROCEDURE DontUse(VAR h: tHash);
(* Wenn eine Variable vom Typ 'tFrag' nie wieder benutzt werden soll
   (besonders bei lokalen Variablen am Ende von Prozeduren, da dann
   der zugehoerige Speicherplatz automatisch freigegeben wird) muss
   diese Prozedur fuer diese Variable einmal aufgerufen werden.
*)

PROCEDURE Empty(h: tHash);
(* Alle Elemente in 'h' werden geloescht. *)

PROCEDURE CallDelete(h: tHash; call: BOOLEAN);
(* Es wird festgelegt, ob auf die Elemente von 'tFrag' zusaetzliche
   Referenzen vorhanden sind und somit beim Loeschen der Elemente
   deren Speicherplatz mit durch Aufruf von 'Type.DelI' freigegeben
   werden soll.
   Bei 'call = TRUE' wird 'Type.DelI' aufgerufen, sonst nicht.
   Voreingestellt ist 'HasRef = TRUE' fuer alle neu angelegten
   Variablen vom Typ 'tFrag'.
*)

PROCEDURE Insert(h: tHash; item: tPOINTER);
(* 'item' wird im Hash-Speicher 'h' abgelegt. *)

PROCEDURE Stored(h: tHash; item: tPOINTER;
                 VAR FoundItem: tPOINTER): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls 'item' im Hash-Speicher 'h'
   abgelegt ist. Die Gleichheit wird mit 'Type.EquI' festgestellt.
   Falls das Funktionsergebnis TRUE ist, wird das gefundene Element
   als aktuell markiert und kann anschliessend mit 'DelCur' aus
   dem Hash-Speicher entfernt werden. Es wird in 'FoundItem' zurueck-
   gegeben. *)

       (* DELete CURrent *)
PROCEDURE DelCur(h: tHash);
(* Das aktuelle Element von 'h' (siehe 'Stored') wird geloescht. *)

END Hash.
```

B.10 Implementierungsmodul 'Hash'

```
IMPLEMENTATION MODULE Hash;

(*              Hashing

    ( Erklaerungen im Definitionsmodul )
*)

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, SysMath, Func, Type, List, Cali, Frag;
```

```
FROM SysMath IMPORT sqrt, real;
FROM Sys IMPORT tPOINTER;
FROM Func IMPORT Error;

CONST PrimFile = "hash.inf";
          (* In dieser Datei werden berechnete Primzahlen
             gespeichert. *)
      HashFactor = 10L;
          (* Prozentsatz des Hash-Speichers, der nach dem Reorgani-
             sieren durch 'Reorg' belegt sein soll (muss auf jeden
             Fall unter 50 liegen, sonst ist nicht garantiert, dass
             'Reorg' fuer das naechste neue Element einen freien
             Speicherplatz schafft).
          *)
      HashDiff = 10L;
          (* Beim Einfuegen werden mindestens
             ' HashFactor + HashDiff ' Prozent des Speichers durch
             'Insert' durchsucht, bevor der Speicher neu organisiert
             wird. 'HashDiff' muss groesser oder gleich 1 sein.
             ' HashFactor + HashDiff ' muss kleiner als 50 sein
             um einen Effekt zu erzielen.
          *)

TYPE tHash = POINTER TO tHashRec;
      tHashRec = RECORD
                     current: LONGCARD;
                         (* Index des aktuellen Elements von 'items' *)
                     InsertCount: LONGCARD;
                         (* Anzahl der mit 'Insert' eingefuegten
                            Elemente (einschliesslich der bereits
                            wieder geloeschten) *)
                     MinSize: LONGCARD;
                         (* Mindestgroesse von 'items'
                            (ist nach unten durch 100 begrenzt) *)
                     deleted: Cali.tCali;
                         (* Indizes der geloeschten Elemente *)
                     items: Frag.tFrag
                 END;
VAR
    HashId: Type.Id;
        (* Typidentifikator fuer 'tHashRec' *)
    PrimList: Cali.tCali;
        (* Liste von Primzahlen *)

PROCEDURE WritePrim();
(* 'PrimList' wird in 'PrimFile' gespeichert. *)
VAR f: Sys.File;
BEGIN
    Sys.OpenWrite(f, PrimFile);

    List.First(PrimList);
    REPEAT
        Sys.WriteCard(f, Cali.Cur(PrimList), 0);
        List.Next(PrimList)
    UNTIL NOT List.MoreData(PrimList);

    Sys.Close(f)
END WritePrim;

PROCEDURE ReadPrim();
(* 'PrimList' wird aus 'PrimFile' gelesen. *)
VAR f: Sys.File;
    val: LONGCARD;
BEGIN
    Sys.OpenRead(f, PrimFile);
```

```
    List.Empty(PrimList);
    REPEAT
        Sys.ReadLCard(f, val);
        List.InsertBehind(PrimList, val)
    UNTIL Sys.EOF(f);

    Sys.Close(f)
END ReadPrim;

PROCEDURE NewPrim(z: LONGCARD);
(* Es wird sichergestellt, dass 'PrimList' eine Primzahl enthaelt, die
   groesser oder gleich 'z' ist (ggf. werden neue berechnet). *)
VAR i, end, SqrtEnd: LONGCARD;
    cur: List.tPos; CurVal: LONGCARD;
BEGIN
    List.Last(PrimList);
    IF Cali.Cur(PrimList) < z THEN
        REPEAT
            end:= Cali.Cur(PrimList) + z;
            SqrtEnd:= Func.Ceil( sqrt(real(end)) );
            FOR i:= Cali.Cur(PrimList) + 1 TO end DO
                IF (( i MOD 2 ) # 0) AND ((i MOD 3) # 0) THEN
                    Cali.InsertBehind(PrimList, i)
                END
            END;

            List.First(PrimList);
            List.Next(PrimList); List.Next(PrimList);
            REPEAT
                cur:= List.GetPos(PrimList);
                CurVal:= Cali.Cur(PrimList);
                List.Next(PrimList);
                WHILE List.MoreData(PrimList) DO
                    IF (Cali.Cur(PrimList) DIV CurVal) = 0 THEN
                        List.DelCur(PrimList)
                    ELSE
                        List.Next(PrimList)
                    END
                END;
                List.SetPos(PrimList, cur);
                List.Next(PrimList)
            UNTIL List.AtLast(PrimList)
                      OR (Cali.Cur(PrimList) > SqrtEnd)
        UNTIL Cali.Cur(PrimList) >= z;
        WritePrim
    END
END NewPrim;

PROCEDURE GetPrim(z: LONGCARD): LONGCARD;
(* Funktionsergebnis ist die zu 'z' naechst groessere Primzahl.
   Falls 'z' eine Primzahl ist, wird sie als Ergebnis zurueck-
   gegeben.
   Zur Berechnung der Primzahlen wird das Sieb des Eratostenes
   verwendet.
*)
BEGIN
    NewPrim(z);

    List.Last(PrimList);
    WHILE (Cali.Cur(PrimList) > z) AND List.MoreData(PrimList) DO
        List.Prev(PrimList)
    END;
    IF Cali.Cur(PrimList) = z THEN
        RETURN z
```

```
    ELSE
        IF Cali.Cur(PrimList) < z THEN
            List.Next(PrimList)
        END;
        RETURN Cali.Cur(PrimList)
    END
END GetPrim;

PROCEDURE Use(VAR h: tHash; type: Type.Id; size: LONGCARD);
BEGIN
    IF size < 100 THEN
        size:= 100
    END;

    h:= Type.NewI(HashId);
    Frag.SetType(h^.items, type);
    Frag.SetRange(h^.items, 0, GetPrim(size));
    h^.MinSize:= Frag.GetHigh(h^.items)
END Use;

PROCEDURE DontUse(VAR h: tHash);
BEGIN
    Type.DelI(HashId, h)
END DontUse;

PROCEDURE Empty(h: tHash);
BEGIN
    List.Empty(h^.deleted);
    Frag.Empty(h^.items);
    h^.InsertCount:= 0;
    h^.current:= 0
END Empty;

PROCEDURE CallDelete(h: tHash; call: BOOLEAN);
BEGIN
    Frag.AddRef(h^.items, NOT call)
END CallDelete;

PROCEDURE Transfer(from, to: tHash);
(* Alle nicht als geloescht markierten Elemente in 'from' werden
   nach 'to' uebertragen.
   Nach dem Aufruf von 'Transfer' ist 'from' vollstaendig leer.
*)
VAR i: LONGCARD;
BEGIN
     IF Frag.GetType(from^.items) # Frag.GetType(to^.items) THEN
         Error("Hash.Transfer",
             "Die Elementtypen der Hash-Speicher sind verschieden.")
     END;
     List.First(from^.deleted);
     WHILE List.MoreData(from^.deleted) DO
         Frag.SetItem(from^.items, Cali.Cur(from^.deleted), NIL);
         List.Next(from^.deleted)
     END;
     FOR i:= 0 TO Frag.GetHigh(from^.items) DO
         IF Frag.GetItem(from^.items, i) # NIL THEN
             Insert(to, Frag.GetItem(from^.items, i))
         END
     END;
     Empty(from)
END Transfer;

PROCEDURE Assign(from, to: tHash);
(* Die Verwaltungsdaten von 'from' werden 'to' zugewiesen.
   Die Elementtypen von 'h1' und 'h2' muessen gleich sein. *)
```

```
BEGIN
    to^.current:= from^.current;
    to^.InsertCount:= from^.InsertCount;
    to^.MinSize:= from^.MinSize;
    to^.deleted:= from^.deleted;
    to^.items:= from^.items
END Assign;

PROCEDURE Swap(h1, h2: tHash);
(* Die Verwaltungsdaten von 'h1' und 'h2' werden vertauscht.
   Die Elementtypen von 'h1' und 'h2' muessen gleich sein. *)
VAR tmp: tHash;
BEGIN
    Use(tmp, Frag.GetType(h1^.items), Frag.GetHigh(h1^.items));
    Assign(h1, tmp);
    Assign(h2, h1);
    Assign(tmp, h2);
    DontUse(tmp)
END Swap;

PROCEDURE Reorg(h: tHash);
(* Die Element in 'h' werden neu organisiert, so dass maximal soviel
   Prozent des Hash-Speichers belegt sind, wie 'HashFactor' angibt. *)
VAR NewSize: LONGCARD;
    NewHash: tHash;
BEGIN
    NewSize:= Func.MaxLCard(
                  (h^.InsertCount - List.Count(h^.deleted))
                      DIV HashFactor + 1 * 100
                , h^.MinSize
              );

    Use(NewHash, Frag.GetType(h^.items), NewSize);
    Transfer(h, NewHash);
    Swap(h, NewHash);
    DontUse(NewHash)
END Reorg;

VAR level: CARDINAL;
        (* Variable zum Test der Rekursion *)

PROCEDURE Insert(h: tHash; item: tPOINTER);
VAR index, FirstIndex, delta: LONGCARD;
    end: BOOLEAN;
    MaxTries: LONGCARD;
        (* Anzahl der Einfuege-Versuche, bevor 'Reorg'
           angestossen wird *)
    tries: LONGCARD;
        (* Anzahl der bereits durchgefuehrten Versuche *)
BEGIN
    INC(level);
    tries:= 0;
    MaxTries:= Frag.GetHigh(h^.items) DIV 100 + 1
               * (HashFactor + HashDiff);

    index:= Type.HashI( Frag.GetType(h^.items), item,
                        Frag.GetHigh(h^.items)
                      );
    FirstIndex:= index;
    delta:= 1;
    REPEAT
        end:= ( Frag.GetItem(h^.items, index) = NIL );
        IF NOT end THEN
            index:= ( index + delta ) MOD Frag.GetHigh(h^.items);
            delta:= ( delta + 2 ) MOD Frag.GetHigh(h^.items);
```

```
            INC(tries);
        ELSE
            Frag.SetItem(h^.items, index, item);
            INC(h^.InsertCount)
        END
    UNTIL end OR (index = FirstIndex) OR (tries > MaxTries);
    IF NOT end THEN
        IF level > 1 THEN
            Error("Hash.Insert",
            "'Garbage Collection' schafft keinen freien Speicher");
        ELSE
            Reorg(h);
            Insert(h, item)
        END
    END;

    DEC(level)
END Insert;

PROCEDURE Stored(h: tHash; item: tPOINTER;
                 VAR FoundItem: tPOINTER): BOOLEAN;
VAR index, FirstIndex, delta: LONGCARD;
    end, res: BOOLEAN;
BEGIN
    index:= Type.HashI(Frag.GetType(h^.items), item,
                       Frag.GetHigh(h^.items));
    FirstIndex:= index;
    delta:= 1;
    REPEAT
        end:= ( Frag.GetItem(h^.items, index) = NIL );
        IF NOT end THEN
            index:= ( index + delta ) MOD Frag.GetHigh(h^.items);
            delta:= ( delta + 2 ) MOD Frag.GetHigh(h^.items)
        END
    UNTIL end OR (index = FirstIndex);
    IF end THEN
        FoundItem:= Frag.GetItem(h^.items, index)
    ELSE
        FoundItem:= NIL
    END;
    RETURN end
END Stored;

       (* DELete CURrent *)
PROCEDURE DelCur(h: tHash);
BEGIN
    IF h^.current = 0 THEN
        Error("Hash.DelCur",
            "Das aktuelle Element ist undefiniert.")
    END;
    Cali.Insert(h^.deleted, h^.current)
END DelCur;

PROCEDURE NewHash(h: tPOINTER);
VAR H: tHash;
BEGIN
    H:= h;

    H^.current:= 0;
    H^.InsertCount:= 0;
    H^.MinSize:= 1;
    Cali.Use(H^.deleted);
    Frag.Use(H^.items, Type.NoType(), 1, 1)
END NewHash;
```

```
PROCEDURE DelHash(h: tPOINTER);
VAR H: tHash;
BEGIN
    H:= h;
    List.DontUse(H^.deleted);
    Frag.DontUse(H^.items)
END DelHash;

BEGIN
    HashId:= Type.New(TSIZE( tHashRec ) );
    Type.SetName(HashId, "Hash.tHash");
    Type.SetNewProc(HashId, NewHash);
    Type.SetDelProc(HashId, DelHash);

    Cali.Use(PrimList);
    IF Sys.Exist(PrimFile) THEN
        ReadPrim
    ELSE
        Cali.InsertBehind(PrimList, 2);
        Cali.InsertBehind(PrimList, 3)
    END;

    level:= 0;
END Hash.
```

B.11 Definitionsmodul 'Inli'

```
DEFINITION MODULE Inli;

(*
   Listen von ganzen Zahlen (Typ LONGINT) unter Verwendung
   des Moduls 'list'
*)

IMPORT Sys, Type, Simptype, List;

TYPE tInli= List.tList;

(* Die folgenden Prozeduren entsprechen in ihrer Bedeutung den
   gleichnamigen im Modul 'List', jedoch angepasst auf den
   Typ LONGINT als Listenelemente. *)

PROCEDURE Use(VAR list: tInli);

PROCEDURE InsertBefore(list: tInli; item: LONGINT);

PROCEDURE InsertBehind(list: tInli; item: LONGINT);

PROCEDURE Insert(list: tInli; item: LONGINT);

PROCEDURE Cur(list: tInli): LONGINT;

PROCEDURE OutCur(list: tInli): LONGINT;

(* Fuer 'LONGINT-Listen' koennen weiterhin die Prozeduren/Funk-
   tionen aus dem Modul 'List' verwendet werden, die nicht
   namensgleich zu den obigen sind.
*)

END Inli.
```

B.12 Implementierungsmodul 'Inli'

```
IMPLEMENTATION MODULE Inli;

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, Type, Simptype, List;
FROM Sys IMPORT tPOINTER;
FROM Simptype IMPORT IntId, NewInt, DelInt, pInt;

PROCEDURE Use(VAR list: tInli);
BEGIN
    List.Use(list, IntId())
END Use;

PROCEDURE InsertBefore(list: tInli; item: LONGINT);
VAR ListItem: pInt;
BEGIN
    ListItem:= NewInt(item);
    ListItem^:= item;
    List.InsertBefore(list,ListItem)
END InsertBefore;

PROCEDURE InsertBehind(list: tInli; item: LONGINT);
VAR ListItem: pInt;
BEGIN
    ListItem:= NewInt(item);
    ListItem^:= item;
    List.InsertBehind(list,ListItem)
END InsertBehind;

PROCEDURE Insert(list: tInli; item: LONGINT);
BEGIN
    InsertBehind(list,item)
END Insert;

PROCEDURE Cur(list: tInli): LONGINT;
VAR ListItem: pInt;
BEGIN
    ListItem:= List.Cur(list);
    RETURN ListItem^
END Cur;

PROCEDURE OutCur(list: tInli): LONGINT;
VAR ListItem: pInt;
    erg     : LONGINT;
BEGIN
    ListItem:= List.OutCur(list);
    erg     := ListItem^;
    DelInt(ListItem);
    RETURN erg
END OutCur;

END Inli.
```

B.13 Definitionsmodul 'List'

```
DEFINITION MODULE List;

(*         Lineare Listen

   Zur Handhabung einzelner Elemente einer Liste wird
   das Modul 'Types' verwendet. D. h. fuer jeden Art
   von Elementen, die in einer Liste verwaltet werden
   sollen muss mit Hilfe von 'Types' ein entsprechender
   Type vereinbart werden.

   Jede Liste kann nur Elemente eines Datentyps enthalten.

   Fuer jeden Datentyp muessen Prozeduren vereinbart werden,
   die die Behandlung von Elementen dieses Typs ermoeglichen.
   Die erforderlichen Operationen sind:
       Anlegen, Loeschen, auf Gleichheit pruefen, auf Erfuellung
       einer Ordnungsrelation pruefen
   Die Angabe der Prozeduren fuer die Operationen ist optional,
   in Abhaengigkeit davon, wie weit die Moeglichkeiten des Moduls
   genutzt werden.

   Der Zugriff auf Listenelemente erfolgt grundsaetzlich ueber die
   zugehoerigen Adressen.

   In der Liste gibt es immer ein (ggf. undefiniertes) 'aktuelles
   Listenelement', auf das in verschiedenen Prozeduren Bezug genommen
   wird.

   Im Modul 'Type' werden Listen unter dem Typnamen 'List' gefuehrt.

   Listen von Listen koennen durch
       List.Use(ListListVar, Type.GetId("List"));
   vereinbart werden. Alle Listenprozeduren koennen unveraendert
   benutzt werden. Zu beachten ist, dass beim Einfuegen oder
   Entfernen keine Kopien von Listen angelegt werden.

   Listen von Matrizen (Modul 'Mat') koennen durch
       List.Use(MatListVar, Type.GetId("Matrix"));
   vereinbart werden. Es gelten alle Bemerkungen analog zu
   'Listen von Listen'.
*)

IMPORT Sys, Str, Type;
FROM Sys IMPORT tPOINTER;

TYPE tList; (* Dieser Typ ist fuer Variablen zu benutzen, die
               Listen darstellen sollen. *)
     tPos;  (* Dieser Typ ist in Verbindung mit den Prozeduren
               'GetPos' und 'SetPos' zu benutzen. *)

(************************************)
(* allgemeine Verwaltungsprozeduren: *)

PROCEDURE Use(VAR list: tList; type: Type.Id);
(* Vor der Benutzung einer Variablen vom Typ 'tList' muss diese
   Prozedur einmal fuer diese Variable aufgerufen werden.
   Die Elemente in 'list' sind vom durch angegebenen Typ 'type',
   der vorher mit Hilfe des Moduls 'Type' vereinbart werden
   muss.
*)

PROCEDURE DontUse(VAR list: tList);
(* Wenn eine Variable vom Typ 'tList' nie wieder benutzt werden soll
   (besonders bei lokalen Variablen am Ende von Prozeduren, da dann
```

```
   der zugehoerige Speicherplatz automatisch freigegeben wird) muss
   diese Prozedur fuer diese Variable einmal aufgerufen werden.
*)

PROCEDURE Empty(list: tList);
(* Die angegebene Liste wird geleert. *)

(*****************************************************************)
(* Prozeduren, die sich auf das aktuelle Listenelement beziehen: *)

PROCEDURE DelCur(list: tList); (* DELete CURrent *)
(* Falls mit 'AddRef' vereinbart wurde, dass weitere Referenzen
   auf Elemente in der angegebenen Liste existieren, wird
   das aktuelle Listenelement mit Hilfe von 'OutCur' aus
   der Liste entfernt, ansonsten durch 'Type.DelI'.

   Es wird das Element zum 'aktuellen Listenelement', welches
   auf das geloeschte folgt. Falls kein weiteres Element folgt,
   wird das vorhergehende Element zum aktuellen.
*)

PROCEDURE InsertBefore(list: tList; item: tPOINTER);
(* 'item' wird in 'list' vor dem 'aktuellen Element'
   eingefuegt. Falls das 'aktuelle Element' undefiniert ist, wird
   'item' am Anfang eingefuegt. 'item' wird zum neuen 'aktuellen
   Element'.
*)

PROCEDURE InsertBehind(list: tList; item: tPOINTER);
(* 'item' wird in 'list' hinter dem 'aktuellen Element'
   eingefuegt. Falls das 'aktuelle Element' undefiniert ist, wird
   'item' am Ende eingefuegt. 'item' wird zum neuen 'aktuellen
   Element'.
*)

PROCEDURE Insert(list: tList; item: tPOINTER);
(* 'item' wird in 'list' eingefuegt. Die Stelle, an der die
   Einfuegung vorgenommen wird, ist undefiniert.
*)

PROCEDURE First(list: tList);
(* In 'list' wird das erste Listenelement zum aktuellen. Falls
   die Liste keine Elemente enthaelt, ist nach diesem Aufruf
   das 'aktuelle Listenelement' undefiniert.
*)

PROCEDURE Last(list: tList);
(* In 'list' wird das letzte Listenelement zum aktuellen. Falls
   die Liste keine Elemente enthaelt, ist nach diesem Aufruf
   das 'aktuelle Listenelement' undefiniert.
*)

PROCEDURE Prev(list: tList); (* PREVious *)
(* Das vor dem aktuellen Listenelement stehende Element wird zum
   neuen aktuellen.
   Falls vor dem Aktuellen kein Element existiert bleibt das
   erste Element das Aktuelle.
   Falls das aktuelle Listenelement undefiniert war, bleibt es
   undefiniert.
   Der Funktionswert ist TRUE, falls das aktuelle Listenelement
   definiert war und vor ihm ein weiteres existierte.
*)

PROCEDURE AtFirst(list: tList): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls das erste
```

```
   Listenelement das Aktuelle ist. In allen anderen
   Faellen (auch wenn das aktuelle Element undefiniert ist)
   lautet das Ergebnis FALSE.
*)

PROCEDURE AtLast(list: tList): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls das letzte
   Listenelement das Aktuelle ist. In allen anderen
   Faellen (auch wenn das aktuelle Element undefiniert ist)
   lautet das Ergebnis FALSE.
*)

PROCEDURE Next(list: tList);
(* Das hinter dem aktuellen Listenelement stehende Element wird zum
   neuen aktuellen.
   Falls hinter dem Aktuellen kein Element existiert, bleibt das
   letzte Element das Aktuelle.
   Falls das aktuelle Listenelement undefiniert war, bleibt es
   undefiniert.
   Der Funktionswert ist TRUE, falls das aktuelle Listenelement
   definiert war und hinter ihm ein weiteres existierte.
*)

PROCEDURE MoreData(list: tList): BOOLEAN;
(* Nach dem Aufruf der Prozeduren 'First', 'Last', 'Next'
   und 'Prev' kann mit dieser Funktion festgestellt werden,
   ob die Prozeduren das aktuelle Listenelement neu definiert
   haben. Das Funktionsergebnis ist TRUE, falls dies der Fall
   ist, sonst FALSE.
   Durch diese Funktion kann eine Liste nicht nur mit 'Scan',
   sondern auch mit
       List.First(mylist);
       WHILE List.MoreData(mylist) DO
           ...
           List.Next(mylist)
       END
   durchlaufen werden.
   Die Verwendung von 'MoreData' in Verbindung mit Prozeduren
   ausser den oben genannten ist nicht sinnvoll.
*)

PROCEDURE Cur(list: tList): tPOINTER; (* CURrent *)
(* Das Funktionsergebnis ist das aktuelle Listenelement. *)

PROCEDURE GetPos(list: tList): tPos; (* GET POSition *)
(* Das Funktionsergebnis ist die Position des aktuellen
   Listenelements innerhalb der Liste. Dies ist KEINE
   Ordnungszahl fuer die Listenposition. Aus dem Wert kann
   keinerlei weitergehende Information entnommen werden.
   Er kann nur in Verbindung mit 'SetPos' weiter verwendet
   werden.
   Auf diese Weise ist es moeglich, auf bestimmte Listenelement
   sofort zuzugreifen, ohne sie erst die Liste durchsuchen zu
   muessen. *)

PROCEDURE SetPos(list: tList; pos: tPos); (* SET POSition *)
(* Diese Prozedur ist das Gegenstueck zu 'GetPos' (s. o.).
   Das Listenelement an der angegebenen Position wird zum neuen
   aktuellen Listenelement. Fuer 'pos' duerfen als Werte NUR
   Funktionswerte von 'GetPos' verwendet werden. Insbesondere
   darf bei 'SetPos' keine andere Liste als bei 'GetPos' angegeben
   werden. Dies wird nicht ueberprueft und fuehrt zu unvor-
   hersehbaren Reaktionen. *)

PROCEDURE OutCur(list: tList): tPOINTER; (* OUT CURrent *)
```

```
(* Das aktuelle Listenelement wird aus der Liste entfernt, jedoch
   nicht geloescht. Es wird als Funktionswert zurueckgegeben.
   Das auf dieses Element folgende wird zum neuen aktuellen Element.
   Falls kein weiteres Element existiert, wird das vorhergehende
   Element zum aktuellen.
*)

(*************)
(* Diverses: *)

TYPE tScanProc= PROCEDURE(tPOINTER);

PROCEDURE Scan(list: tList; proc: tScanProc);
(* 'proc' wird mit jedem Element in 'list' als Parameter
   (beginnend beim ersten Element) genau einmal aufgerufen.
   Anschliessend ist das erste Listenelement das Aktuelle.
*)

PROCEDURE AddRef(list: tList); (* ADDitional REFerences *)
(* Nachdem diese Prozedur aufgerufen wurde, erfolgt das Loeschen
   von Elementen aus dieser Liste nur durch Aufloesung von
   Referenzen und NICHT MEHR durch den Aufruf von 'Types.DelI'.
*)

PROCEDURE GetType(list: tList): Type.Id;
(* Das Funktionsergebnis ist der Typ der Elemente in der
   angegebenen Liste.
*)

PROCEDURE Count(list: tList): LONGCARD;
(* Das Ergebnis dieser Funktion ist die Anzahl der Elemente in
   'list'.
*)

PROCEDURE CheckStructure(list: tList);
(* Diese Prozedur dient zur Fehlersuche. Sie prueft die interne
   Verwaltungsstruktur von 'list' und gibt Meldungen in die
   Standardausgabe aus, falls sie Strukturfehler entdeckt. *)

END List.
```

B.14 Implementierungsmodul 'List'

```
IMPLEMENTATION MODULE List;

(* Verwaltung von Listen

   (Erklaerungen im Definitionsmodul)
*)

FROM SYSTEM IMPORT TSIZE, ADR;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM InOut IMPORT WriteString, WriteLn;
IMPORT Sys, Str, Type;
FROM Sys IMPORT tPOINTER;

TYPE tPoiListItem = POINTER TO tListItem;
     tPos         = tPoiListItem;
     tListItem    =
         RECORD
```

```
            value    : tPOINTER;
            next,prev: tPoiListItem;
                (* doppelt verkettet *)
        END;
    tList        = POINTER TO tListRecord;
    tListRecord  =
        RECORD (* Ringliste *)
            first  : tPoiListItem;
                     (* erstes Listenelement *)
            current: tPoiListItem;
                     (* aktuelles Listenelement;
                        vgl. 'First', 'Next', 'Previous',
                        'Current' *)
            OldCurrent: tPoiListItem;
                     (* vorangegangener Wert von 'current'
                        (wichtig fuer 'MoreData' *)
            count  : LONGCARD;
                     (* Anzahl der Elemente in der Liste *)
            type   : Type.Id;
                     (* Typ der Elemente der Liste *)
            addref : BOOLEAN (* ADDitional REFerences *)
                     (* TRUE: beim Loeschen von Elementen dieser
                              Liste werden lediglich Referenzen
                              innerhalb der Liste aufgeloest;
                              'Type.DelI' wird nicht aufgerufen *)
        END;
VAR
    i: CARDINAL;
    ListId: Type.Id;

(************************************)
(* allgemeine Verwaltungsprozeduren: *)

PROCEDURE ListNewI(hilf: tPOINTER);
VAR
    list: tList;
BEGIN
    list:= hilf;
    list^.first:= NIL;
    list^.current:= NIL;
    list^.OldCurrent:= NIL;
    list^.count:= 0;
    list^.type:= Type.NoType();
    list^.addref:= FALSE;
END ListNewI;

PROCEDURE Use(VAR list: tList; type: Type.Id);
BEGIN
    list:= Type.NewI(ListId);
    list^.type:= type
END Use;

PROCEDURE ListDelI(hilf: tPOINTER);
VAR
    list: tList;
BEGIN
    list:= hilf;
    Empty(list)
END ListDelI;

PROCEDURE DontUse(VAR list: tList);
BEGIN
    Type.DelI(ListId, list)
END DontUse;
```

```
PROCEDURE Empty(list: tList);
BEGIN
    First(list);
    WHILE Count(list)>0 DO
        DelCur(list)
    END
END Empty;

(*****************************************************************)
(* Prozeduren, die sich auf das aktuelle Listenelement beziehen: *)

PROCEDURE DelCur(list: tList); (* DELete CURrent *)
VAR
    OldCur: tPOINTER;
BEGIN
    IF list^.current=NIL THEN
        WriteLn;
        WriteString("*** List.DeleteCur: kein aktuelles Element ***");
        WriteLn;
        HALT;
        RETURN
    END;

    OldCur:= OutCur(list);
    IF OldCur=NIL THEN RETURN END;
    IF NOT list^.addref THEN
        Type.DelI(list^.type, OldCur)
    END
END DelCur;

       (* INSert *)
PROCEDURE Ins(list: tList; value: tPOINTER; before: BOOLEAN);
(* 'value' wird in 'list' eingefuegt.
   Wenn 'before=TRUE', dann wird vor dem aktuellen Element ein-
   gefuegt, sonst dahinter.
   Die Liste darf leer sein. Das aktuelle Element darf undefiniert
   (=NIL) sein. *)
VAR
    JustCreated: tPoiListItem;
BEGIN
    ALLOCATE(JustCreated, TSIZE(tListItem));
    JustCreated^.value:= value;

    IF Count(list) = 0 THEN
        JustCreated^.prev:= JustCreated;
        JustCreated^.next:= JustCreated;
        list^.first:= JustCreated
    ELSE
        IF list^.current = NIL THEN
            (* fuege ganz am Anfang oder ganz am Ende ein durch
               Einfuegen zwischen Anfang und Ende und korrektes
               Setzen von 'first': *)
            list^.current:= list^.first^.prev;
            IF before THEN
                list^.first:= JustCreated
            END
        ELSE
            IF before THEN
                (* Sonderfall: aktuelles Element ist erstes
                               Element *)
                IF list^.current = list^.first THEN
                    list^.first:= JustCreated
                END;

                (* setze 'current' um ein Element nach vorne, damit
```

```
                korrekt eingefuegt wird: *)
            list^.current:= list^.current^.prev;
        END
    END;

    (* fuege hinter 'current' ein *)
    JustCreated^.next:= list^.current^.next;
    JustCreated^.prev:= list^.current;
    JustCreated^.prev^.next:= JustCreated;
    JustCreated^.next^.prev:= JustCreated
END;

list^.current:= JustCreated;
INC(list^.count)

END Ins;

PROCEDURE InsertBefore(list: tList; item: tPOINTER);
BEGIN
    Ins(list,item,TRUE)
END InsertBefore;

PROCEDURE InsertBehind(list: tList; item: tPOINTER);
BEGIN
    Ins(list,item,FALSE)
END InsertBehind;

PROCEDURE Insert(list: tList; item: tPOINTER);
BEGIN
    Ins(list,item,FALSE)
END Insert;

PROCEDURE First(list: tList);
BEGIN
    WITH list^ DO
        IF current # first THEN
            OldCurrent:= current;
            current:= first
        ELSE
            (* fuer den Sonderfall: 'First' wenn das erste
               Element bereits aktuell ist;
               in diesem Fall soll 'MoreData' trotzdem TRUE
               liefert *)
            OldCurrent:= NIL
        END
    END
END First;

PROCEDURE Last(list: tList);
BEGIN
    IF list^.first # NIL THEN
        IF list^.current # list^.first^.prev THEN
            list^.OldCurrent:= list^.current;
            list^.current:= list^.first^.prev
        ELSE
            (* fuer den Sonderfall: 'Last' wenn das letzte
               Element bereits aktuell ist;
               in diesem Fall soll 'MoreData' trotzdem TRUE
               liefert *)
            list^.OldCurrent:= NIL
        END
    ELSE
        IF list^.current # NIL THEN
            WriteString("*** List.Last:");
            WriteString(" aktuelles Listenelement ausserhalb");
```

```
            WriteString(" der Liste"); WriteLn;
            HALT
        END
    END
END Last;

PROCEDURE Prev(list: tList); (* PREVious *)
BEGIN
    IF list^.current <> NIL THEN
        IF list^.current <> list^.first THEN
            list^.OldCurrent:= list^.current;
            list^.current:= list^.current^.prev
        END
    END
END Prev;

PROCEDURE Next(list: tList);
BEGIN
    IF list^.current <> NIL THEN
        IF list^.current^.next <> list^.first THEN
            list^.OldCurrent:= list^.current;
            list^.current:= list^.current^.next
        END
    END
END Next;

PROCEDURE MoreData(list: tList): BOOLEAN;
BEGIN
    RETURN list^.current # list^.OldCurrent
END MoreData;

PROCEDURE AtFirst(list: tList): BOOLEAN;
BEGIN
    IF list^.current <> NIL THEN
        IF list^.current = list^.first THEN
            RETURN TRUE
        END
    END;
    RETURN FALSE;
END AtFirst;

PROCEDURE AtLast(list: tList): BOOLEAN;
BEGIN
    IF list^.current <> NIL THEN
        IF list^.current = list^.first^.prev THEN
            RETURN TRUE
        END
    END;
    RETURN FALSE;
END AtLast;

PROCEDURE Cur(list: tList): tPOINTER; (* CURrent *)
BEGIN
    RETURN list^.current^.value
END Cur;

PROCEDURE GetPos(list: tList): tPos; (* GET POSition *)
BEGIN
    RETURN list^.current
END GetPos;

PROCEDURE SetPos(list: tList; pos: tPos); (* SET POSition *)
BEGIN
    list^.current:= pos
END SetPos;
```

```
PROCEDURE OutCur(list: tList): tPOINTER; (* OUT CURrent *)
VAR
    erg: tPOINTER;
    item: tPoiListItem;
    OldFirst: tPoiListItem;
BEGIN
    IF list^.current = NIL THEN
        WriteLn;
        WriteString(
            "*** List.OutCur: kein aktuelles Listenelement ***");
        WriteLn;
        HALT;
        RETURN NIL
    END;

    (* Liste enthaelt mindestens ein Element: *)

    item:= list^.current;
    erg:= item^.value;

    IF Count(list) = 1 THEN
        IF list^.current<>list^.first THEN
            WriteLn;
            WriteString(
               "*** List.OutCur: Fehler in Listenverwaltung;  ***");
            WriteLn;
            WriteString(
               "*** aktuelles Listenelement in falscher Liste ***");
            WriteLn;
            HALT
        END;
        list^.first:= NIL;
        list^.current:= NIL;
        list^.OldCurrent:= NIL
    ELSE
        (* Sonderfall: erstes Element ist aktuelles *)
        OldFirst:= list^.first;
        IF list^.current = OldFirst THEN
            list^.first:= list^.first^.next
        END;

        IF list^.current^.next = OldFirst THEN
            list^.current:= list^.current^.prev
        ELSE
            list^.current:= list^.current^.next
        END;
        item^.prev^.next:= item^.next;
        item^.next^.prev:= item^.prev
    END;

    DEALLOCATE(item,TSIZE(tListItem));
    DEC(list^.count);

    RETURN erg
END OutCur;

(*************)
(* Diverses: *)

PROCEDURE Scan(list: tList; proc: tScanProc);
VAR i: LONGCARD;
BEGIN
    IF Count(list) = 0 THEN RETURN END;
    First(list);
```

```
    FOR i:= 1 TO Count(list) DO
        proc(Cur(list));
        Next(list)
    END;
    First(list)
END Scan;

PROCEDURE AddRef(list: tList);
BEGIN
    list^.addref:= TRUE
END AddRef;

PROCEDURE GetType(list: tList): Type.Id;
BEGIN
    RETURN list^.type
END GetType;

PROCEDURE Count(list: tList): LONGCARD;
BEGIN
    RETURN list^.count
END Count;

PROCEDURE StructureError(t1,t2: ARRAY OF CHAR);
(* Es wird eine Fehlermeldung fuer die Prozedur 'CheckStructure'
   ausgegeben. *)
BEGIN
    WriteString("*** List.CheckStructure:"); WriteLn;
    WriteString("*** "); WriteString(t1); WriteLn;
    WriteString("*** "); WriteString(t2); WriteLn;
    HALT
END StructureError;

PROCEDURE CheckChain(list: tList; UseNext: BOOLEAN);
(* Es wird die Verzeigerung von 'l' geprueft. Bei Strukturfehlern
   werden mit Hilfe von 'StructureError' Fehlermeldungen
   ausgegeben. Es wird   l^.first # NIL   vorausgesetzt.
   Bei 'UseNext = TRUE' wird die Verzeigerung entlang der
   'next'-Zeiger verfolgt, sonst entlang der 'prev'-Zeiger. *)
VAR
    CurrentFound: BOOLEAN;
        (* TRUE: list^.current  kann von  list^.first  aus
                 entlang der Verzeigerung erreicht werden
        *)
    MyCount: LONGCARD;
        (* Anzahl der Listenelemente durch Verfolgung der
           Verzeigerung festgestellt *)
    MyItem: tPoiListItem;
        (* Zeiger zur Verfolgung der Verzeigerung *)
    ErrorText: ARRAY [1..50] OF CHAR;
BEGIN
    IF UseNext THEN
        Str.Assign(ErrorText,
            "Verfolgung entlang  list^.first^.next^.next^. ...")
    ELSE
        Str.Assign(ErrorText,
            "Verfolgung entlang  list^.first^.prev^.prev^. ...")
    END;
    MyCount:= 0;
    MyItem:= list^.first;
    REPEAT
        INC(MyCount);
        CurrentFound:=  CurrentFound
                        OR (MyItem = list^.current);
        IF UseNext THEN
            MyItem:= MyItem^.next
```

```
        ELSE
            MyItem:= MyItem^.prev
        END
    UNTIL (MyItem = list^.first) OR (MyCount > list^.count)
          OR (MyItem = NIL);
    IF MyItem = NIL THEN
        StructureError( ErrorText,
        "ergibt keine geschlossene Ringliste")
    END;
    IF MyCount > list^.count THEN
        StructureError( ErrorText,
        "ergibt MEHR Listenelemente als  list^.count  angibt")
    END;
    IF MyCount < list^.count THEN
        StructureError( ErrorText,
        "ergibt WENIGER Listenelemente als  list^.count angibt")
    END;
    IF (list^.current # NIL) AND NOT CurrentFound THEN
        StructureError( ErrorText,
        "fuehrt nicht zum Element  list^.current")
    END
END CheckChain;

PROCEDURE CheckStructure(list: tList);
BEGIN
    IF list^.count = 0 THEN
        IF list^.current # NIL THEN
            StructureError(
            "list^.count = 0    jedoch   list^.current # NIL",
            "")
        END;
        IF list^.first # NIL THEN
            StructureError(
            "list^.count = 0    jedoch   list^.first # NIL",
            "")
        END
    ELSE
        IF list^.first = NIL THEN
            StructureError(
            "list^.count # 0    jedoch   list^.first = NIL",
            "")
        END;
        CheckChain(list,TRUE);
        CheckChain(list,FALSE)
    END
END CheckStructure;

BEGIN
    ListId:= Type.New(TSIZE(tListRecord));
    Type.SetName(ListId,"List.tList");
    Type.SetNewProc(ListId, ListNewI);
    Type.SetDelProc(ListId, ListDelI)
END List.
```

B.15 Definitionsmodul 'Mali'

```
DEFINITION MODULE Mali; (* MAtrix LIst *)

(*
   Listen von Matrizen (Typ Rema.tMat) unter Verwendung
   des Moduls 'list'
```

```
*)

IMPORT Sys, List, Type, Rema;
FROM Rema IMPORT tMat;

TYPE tMali= List.tList;

(* Die folgenden Prozeduren entsprechen in ihrer Bedeutung den
   gleichnamigen im Modul 'List', jedoch angepasst auf den
   Typ 'Rema.tMat' als Listenelemente.
*)

PROCEDURE Use(VAR list: tMali);

PROCEDURE InsertBefore(list: tMali; item: tMat);

PROCEDURE InsertBehind(list: tMali; item: tMat);

PROCEDURE Insert(list: tMali; item: tMat);

PROCEDURE Cur(list: tMali): tMat;

PROCEDURE OutCur(list: tMali): tMat;

(* Fuer 'tMat-Listen' koennen weiterhin die Prozeduren/Funk-
   tionen aus dem Modul 'List' verwendet werden, die nicht
   namensgleich zu den obigen sind.
*)

END Mali.
```

B.16 Implementierungsmodul 'Mali'

```
IMPLEMENTATION MODULE Mali;  (* MAtrix LIst *)

(*
   Listen von Matrizen (Typ 'Rema.tMat') unter Verwendung
   des Moduls 'list'

     (Erklaerungen im Definitionsmodul)
*)

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, List, Type, Rema;
FROM Sys IMPORT tPOINTER;
FROM Rema IMPORT tMat;

VAR MatId: Type.Id; (* Typnummer fuer 'Rema.tMat' *)

PROCEDURE Use(VAR list: tMali);
BEGIN
    List.Use(list, MatId)
END Use;

PROCEDURE InsertBefore(list: tMali; item: tMat);
BEGIN
    List.InsertBefore(list, tPOINTER(item))
END InsertBefore;

PROCEDURE InsertBehind(list: tMali; item: tMat);
BEGIN
```

```
    List.InsertBehind(list, tPOINTER(item))
END InsertBehind;

PROCEDURE Insert(list: tMali; item: tMat);
BEGIN
    List.InsertBehind(list, tPOINTER(item))
END Insert;

PROCEDURE Cur(list: tMali): tMat;
VAR ListItem: tMat;
BEGIN
    ListItem:= tMat( List.Cur(list) );
    RETURN ListItem
END Cur;

PROCEDURE OutCur(list: tMali): tMat;
VAR ListItem: tMat;
BEGIN
    ListItem:= tMat( List.OutCur(list) );
    RETURN ListItem
END OutCur;

BEGIN
    MatId:= Type.GetId("Rema.tMat")
END Mali.
```

B.17 Definitionsmodul 'Mat'

```
DEFINITION MODULE Mat;

(*           2-dimensionale Matrizen

   Dieses Modul erlaubt die Verwaltung von Matrizen beliebiger
   2-dimensionaler Matrizen in Verbindung mit dem Modul 'Type'.
   Als Matrizenelemente sind ausschliesslich Zeiger
   (Typ 'Sys.tPOINTER') erlaubt.
*)

IMPORT Sys, SysMath, Func, Type, Frag;
FROM Sys IMPORT tPOINTER;

TYPE tMat;

PROCEDURE Use(VAR mat: tMat; type: Type.Id);
(* Vor der Benutzung einer Variablen vom Typ 'tMat' muss diese
   Prozedur einmal fuer diese Variable aufgerufen werden.
   Die Elemente von 'mat' sind vom durch angegebenen Typ 'type',
   der vorher mit Hilfe des Moduls 'Type' vereinbart werden
   muss.
*)

PROCEDURE DontUse(VAR mat: tMat);
(* Wenn eine Variable vom Typ 'tMat' nie wieder benutzt werden soll
   (besonders bei lokalen Variablen am Ende von Prozeduren, da dann
   der zugehoerige Speicherplatz automatisch freigegeben wird) muss
   diese Prozedur fuer diese Variable einmal aufgerufen werden.
*)

PROCEDURE SetSize(mat: tMat; row, col: LONGCARD);
(* Durch den Aufruf von 'SetSize' wird die Matrix 'mat' geloescht
   und ihre Groesse auf 'row' Zeilen und 'col' Spalten beschraenkt
```

```
   (Zaehlung der Zeilen und Spalten beginnt bei 1).
   Ausserdem wird der Zugriff auf die Listenelemente beschleunigt.
*)

PROCEDURE Set(mat: tMat; row, col: LONGCARD; item: tPOINTER);
(* 'item' wird in 'mat' in Zeile 'row' und Spalte 'col' gespeichert.
   Ein evtl. bereits vorhandenes Element wird geloescht.
*)

PROCEDURE Elem(mat: tMat; row, col: LONGCARD): tPOINTER;
(* Funktionsergebnis ist das Element von 'mat' in Zeile 'row' und
   Spalte 'col'.
*)

PROCEDURE Rows(mat: tMat): LONGCARD;
(* Falls vorher 'Set' fuer 'mat' aufgerufen wurde, ist das Funktions-
   ergebnis die dort festgelegte Anzahl von Zeilen von 'mat'.
   Andernfalls ist das Funktionsergebnis der hoechste Zeilenindex,
   auf den bisher seit dem letzten 'Use'- oder 'Empty'-Aufruf
   mit 'Set' zugegriffen worden ist.
*)

PROCEDURE Columns(mat: tMat): LONGCARD;
(* ... analog 'Rows', jedoch fuer den Spaltenindex *)

PROCEDURE Empty(mat: tMat);
(* Alle Feldelemente von 'mat' werden geloescht (mit 'Type.DelI'). *)

END Mat.
```

B.18 Implementierungsmodul 'Mat'

```
IMPLEMENTATION MODULE Mat;

(*        2-dimensionale Matrizen

    ( Erklaerungen im Definitionsmodul )
*)

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, SysMath, Func, Type, Frag;
FROM Sys IMPORT tPOINTER;
FROM Func IMPORT Error;

TYPE tMat = POINTER TO tMatRecord;
     tMatRecord = RECORD
                      type: Type.Id; (* Typ der Matrizenelemente *)
                      fix: BOOLEAN;
                          (* TRUE: die Groesse der Matrix ist auf
                                   die in 'r' und 'c' angegebenen
                                   Werte beschraenkt;
                             FALSE: die Groesse der Matrix ist un-
                                   beschraenkt; 'r' und 'c' geben
                                   die maximalen Zeilen- bzw. Spal-
                                   tenindizes an auf die bisher mit
                                   'Set' zugegriffen wurde *)
                      r, c: LONGCARD;
                          (* siehe Erklaerung zur Komponente 'fix' *)
                      rows: Frag.tFrag
                          (* Feld der Matrix-Zeilen *)
                  END;
```

```
VAR MatId, FragId: Type.Id;

PROCEDURE CheckIndices(mat: tMat; row, col: LONGCARD;
                       proc: ARRAY OF CHAR);
(* Es wird eine Fehlermeldung ausgegeben, wenn der Zeilenindex
   'row' oder der Spaltenindex 'col' fuer einen Zugriff auf 'mat'
   ausserhalb des erlaubten Bereiches liegt.
*)
BEGIN
    IF mat^.fix THEN
        IF (row > mat^.r) OR (col > mat^.c) THEN
            Error(proc,
                "Zugriff auf nicht existierendes Matrizenelement");
        END
    END
END CheckIndices;

PROCEDURE Use(VAR mat: tMat; type: Type.Id);
BEGIN
    mat:= Type.NewI(MatId);
    mat^.type:= type;
    mat^.fix:= FALSE;
    mat^.r:= 0;
    mat^.c:= 0
END Use;

PROCEDURE DontUse(VAR mat: tMat);
BEGIN
    Type.DelI(MatId, mat)
END DontUse;

PROCEDURE SetSize(mat: tMat; rows, columns: LONGCARD);
VAR WorkRow: Frag.tFrag;
    k: LONGCARD;
BEGIN
    IF mat^.rows # Frag.tFrag(NIL) THEN
        Frag.DontUse(mat^.rows)
    END;
    Frag.Use(mat^.rows, FragId, 1, rows);
    mat^.fix:= TRUE;
    mat^.r:= rows;
    mat^.c:= columns;
    FOR k:= 1 TO rows DO
        WorkRow:= Frag.tFrag(
                    Frag.GetItem(mat^.rows, k)
                );
        IF WorkRow = Frag.tFrag(NIL) THEN
            Frag.Use(WorkRow, mat^.type, 1, columns);
            Frag.SetItem(mat^.rows, k, tPOINTER(WorkRow))
        ELSE
            Error("Mat.SetSize","bereits initialisierte Zeilen vorhanden");
        END
    END
END SetSize;

PROCEDURE Set(mat: tMat; row, col: LONGCARD; item: tPOINTER);
VAR WorkRow: Frag.tFrag;
BEGIN
    CheckIndices(mat, row, col, "Mat.Set");
    IF mat^.fix THEN
        WorkRow:= Frag.tFrag(
                    Frag.GetItem(mat^.rows, row)
                );
        IF WorkRow = Frag.tFrag(NIL) THEN
```

```
            Frag.Use(WorkRow, mat^.type, 1, mat^.c);
            Frag.SetItem(mat^.rows, row, tPOINTER(WorkRow))
        END;
    ELSE
        Error("Mat.Set","Behandlung von  fix=FALSE  nicht implementiert")
    END;
    Frag.SetItem(WorkRow, col, item)
END Set;

PROCEDURE Elem(mat: tMat; row, col: LONGCARD): tPOINTER;
VAR WorkRow: Frag.tFrag;
BEGIN
    CheckIndices(mat, row, col, "Mat.Elem");
    WorkRow:= Frag.tFrag( Frag.GetItem(mat^.rows, row) );
    IF WorkRow # Frag.tFrag(NIL) THEN
        RETURN Frag.GetItem(WorkRow, col)
    ELSE
        RETURN NIL
    END
END Elem;

PROCEDURE Rows(mat: tMat): LONGCARD;
BEGIN
    RETURN mat^.r
END Rows;

PROCEDURE Columns(mat: tMat): LONGCARD;
BEGIN
    RETURN mat^.c
END Columns;

PROCEDURE Empty(mat: tMat);
BEGIN
    Frag.Empty(mat^.rows);
    IF NOT mat^.fix THEN
        mat^.r:= 0;
        mat^.c:= 0
    END
END Empty;

PROCEDURE MatNewI(m: tPOINTER);
VAR M: tMat;
BEGIN
    M:= m;
    M^.rows:= Frag.tFrag(NIL);
END MatNewI;

PROCEDURE MatDelI(m: tPOINTER);
VAR M: tMat;
BEGIN
    M:= m;
    Frag.DontUse(M^.rows)
END MatDelI;

BEGIN
    MatId:= Type.New(TSIZE(tMatRecord));
    Type.SetName(MatId, "Mat.tMat");
    Type.SetNewProc(MatId, MatNewI);
    Type.SetDelProc(MatId, MatDelI);

    FragId:= Type.GetId("Frag.tFrag")
END Mat.
```

B.19 Definitionsmodul 'Reli'

```
DEFINITION MODULE Reli; (* REalLIst *)

(*
   Listen von Fliesskommazahlen (Typ LONGREAL) unter Verwendung
   des Moduls 'list'
*)

IMPORT Sys, Type, Simptype, List;

TYPE tReli= List.tList;

(* Die folgenden Prozeduren entsprechen in ihrer Bedeutung den
   gleichnamigen im Modul 'List', jedoch angepasst auf den
   Typ LONGREAL als Listenelemente. *)

PROCEDURE Use(VAR list: tReli);

PROCEDURE InsertBefore(list: tReli; item: LONGREAL);

PROCEDURE InsertBehind(list: tReli; item: LONGREAL);

PROCEDURE Insert(list: tReli; item: LONGREAL);

PROCEDURE Cur(list: tReli): LONGREAL;

PROCEDURE OutCur(list: tReli): LONGREAL;

(* Fuer 'LONGREAL-Listen' koennen weiterhin die Prozeduren/Funk-
   tionen aus dem Modul 'List' verwendet werden, die nicht
   namensgleich zu den obigen sind.
*)

END Reli.
```

B.20 Implementierungsmodul 'Reli'

```
IMPLEMENTATION MODULE Reli;  (* REalLIst *)

(*
   Listen von Fliesskommazahlen (Typ LONGREAL) unter Verwendung
   des Moduls 'list'

     (Erklaerungen im Definitionsmodul)
*)

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, Type, Simptype, List;
FROM Sys IMPORT tPOINTER;
FROM Simptype IMPORT RealId, NewReal, DelReal, pReal;

PROCEDURE Use(VAR list: tReli);
BEGIN
    List.Use(list, RealId())
END Use;

PROCEDURE InsertBefore(list: tReli; item: LONGREAL);
```

```
VAR ListItem: pReal;
BEGIN
    ListItem:= NewReal(item);
    ListItem^:= item;
    List.InsertBefore(list,ListItem)
END InsertBefore;

PROCEDURE InsertBehind(list: tReli; item: LONGREAL);
VAR ListItem: pReal;
BEGIN
    ListItem:= NewReal(item);
    ListItem^:= item;
    List.InsertBehind(list,ListItem)
END InsertBehind;

PROCEDURE Insert(list: tReli; item: LONGREAL);
BEGIN
    InsertBehind(list,item)
END Insert;

PROCEDURE Cur(list: tReli): LONGREAL;
VAR ListItem: pReal;
BEGIN
    ListItem:= List.Cur(list);
    RETURN ListItem^
END Cur;

PROCEDURE OutCur(list: tReli): LONGREAL;
VAR ListItem: pReal;
    erg     : LONGREAL;
BEGIN
    ListItem:= List.OutCur(list);
    erg     := ListItem^;
    DelReal(ListItem);
    RETURN erg
END OutCur;

END Reli.
```

B.21 Definitionsmodul 'Rama'

```
DEFINITION MODULE Rema;  (* REal MAtrix *)

(*       2-dimensionale LONGREAL-Matrizen

    Im Modul 'Type' werden Matrizen unter dem Typnamen
    'Matrix' gefuehrt.
    (d. h.
        Type.GetId("Matrix");
     ergibt die zugehoerige Typnummer)

    Verbesserungsmoeglichkeiten:
    - Angabe der gewuenschten Eigenwerte fuer 'Randomize'
    - Ermoeglichung nichtganzzahliger Eigenwerte
    - Abfrage der Parameter fuer 'Randomize'
      (bisher nur Setzen moeglich)
    - Fehlermeldungen bei widerspruechlichen Parametern
      fuer 'Randomize'
    - Erzeugung nichtdiagonalisierbarer Matrizen durch 'Randomize'
    - 'Randomize' auch fuer nichtquadratische Matrizen
```

```
    - Angabe eines Intervalls, indem die Elemente der durch
      'Randomize' erzeugten Matrizen liegen
    - automatische Erkennung der Version der Datenstruktur
      durch 'Write', 'WriteF', 'Read' und 'ReadF'
*)

IMPORT Sys, SysMath, Func, Rnd, Type, Frag, List, Simptype,
       Inli, Cali, Reli, Mat, Pram;
FROM Sys IMPORT File;

TYPE tMat;
         (* Dieser Typ ist fuer Variablen vom Typ 'Matrix'
            zu benutzen. *)

CONST MaxIoRow = 6; (* gibt an, wieviele Matrizenelemente von
                       'WriteReal' maximal in eine Zeile
                       ausgegeben werden *)

PROCEDURE Use(VAR mat: tMat; row, col: LONGCARD);
(* Vor der Benutzung einer Variablen vom Typ 'tMat' muss diese
   Prozedur einmal fuer diese Variable aufgerufen
   werden.
   Eine Ausnahme hierzu bilden Variablen, denen eine Matrix mit
   Hilfe von 'Copy' oder 'CreateMult' zugewiesen wird. Diese
   Variablen duerfen nicht durch 'Use' initialisiert worden
   sein.
   Nach der Grundinitialisierung wird automatisch 'Init(mat,row,col)'
   aufgerufen.
*)

PROCEDURE DontUse(mat: tMat);
(* Wenn eine Variable vom Typ 'tMat' nie wieder benutzt werden soll
   (besonders bei lokalen Variablen am Ende von Prozeduren, da dann der
   zugehoerige Speicherplatz automatisch freigegeben wird) muss diese
   Prozedur fuer diese Variable einmal aufgerufen werden.
*)

PROCEDURE Empty(mat: tMat);
(* Alle Elemente der angegebenen Matrix werden mit 0 belegt. *)

PROCEDURE Unit(mat: tMat);
(* Die Elemente der Hauptdiagonalen von 'mat' werden auf 1
   gesetzt, alle anderen Elemente auf 0 *)

PROCEDURE Assign(mat1, mat2: tMat);
(* Diese Prozedur weist die Elemente von Matrix 'mat1' den
   Elementen von Matrix 'mat2' zu. Dazu muessen die Matrizen die
   gleich Anzahl von Zeilen und Spalten besitzen.
*)

PROCEDURE Copy(mat1: tMat): tMat;
(* Es wird eine Kopie der Matrix 'mat1' angelegt und als Funktions-
   wert zurueckgegeben. Die Variable, der dieser Funktionswert
   zugewiesen wird, darf vorher nicht mit 'Use' initialisiert
   worden sein.
*)

PROCEDURE Elem(mat: tMat; row, col: LONGCARD): LONGREAL;
(* Diese Funktion ergibt den Wert des Elementes an der angegebenen
   Position von Matrix 'mat'.
*)

PROCEDURE Set(mat: tMat; row,col: LONGCARD; val: LONGREAL);
(* Diese Prozedur belegt das Element an der angegebenen Position
   der Matrix 'mat' mit dem in 'val' angegebenen Wert.
```

```
*)

PROCEDURE Rows(mat: tMat): LONGCARD;
(* Diese Funktion ergibt die Anzahl der Zeilen der
   Matrix 'mat'.
*)

PROCEDURE Columns(mat: tMat): LONGCARD;
(* Diese Funktion liefert die Anzahl der Spalten der
   Matrix 'mat'.
*)

PROCEDURE SetSize(mat: tMat; r,c: LONGCARD);
(* Fuer die Matrix 'mat' wird festgelegt, dass sie 'r' Zeilen
   und 'c' Spalten besitzt. Durch den Aufruf dieser Prozedur
   werden alle evtl. in 'mat' gespeicherten Daten geloescht.
*)

PROCEDURE SetReal(mat: tMat; real: BOOLEAN);
(* Es wird festgelegt, ob die durch 'Randomize' zu erzeugende
   Matrix 'mat' Elemente mit Nachkommastellen besitzen darf.
   'real' gibt an, ob Nachkommastellen erlaubt sind:
       TRUE :  erlaubt
       FALSE:  verboten
*)

PROCEDURE SetRank(mat: tMat; rank: LONGCARD);
(* Fuer 'mat' wird der gewuenschte Rang 'rank' zur Benutzung durch
   'Randomize' festgelegt.
*)

PROCEDURE SetMultiplicity(mat: tMat; mult: LONGCARD);
(* Fuer 'mat' wird zur Benutzung durch 'Randomize' festgelegt, dass
   ein Eigenwert die Vielfachheit 'mult' besitzt. Falls mit
   'SetMultiplicity keine Vielfachheiten festgesetzt werden, wird
   fuer jeden Eigenwert (ungleich 0) die Vielfachheit 1 angenommen.
*)

PROCEDURE Randomize(mat: tMat);
(* 'mat' wird anhand der mit 'SetIntervall', 'SetRank' und
   'SetMultiplicity' festgesetzten Parameter mit zufaelligen Werten
   belegt. Die Vielfachheiten (mit 'SetMultiplicity' festgelegt)
   werden nur beachtet, soweit es Matrixgroesse und Rang (mit
   'SetRank' festgelegt) zulassen.
   'mat' muss eine quadratische Matrix sein.
*)

PROCEDURE Det(mat: tMat): LONGREAL;
(* Fuer eine durch 'Randomize' generierte Matrix kann mit dieser
   Funktion ihre Determinante festgestellt werden.
   ('Abfallprodukt' des Generierungsvorganges)
*)

PROCEDURE Write(mat: tMat);
(* Diese Prozedur schreibt den Inhalt der Matrix 'mat' in den
   Standardausgabekanal.
*)

PROCEDURE WriteF(VAR f: File; mat: tMat);
(* ... analog 'Write', jedoch erfolgt die Ausgabe in die Datei 'f' *)

PROCEDURE Read(mat: tMat);
(* Diese Prozedur liest die Werte fuer die Elemente der Matrix
   'mat' aus dem Standard-Eingabekanal. Mit 'Write' ausgegebene
   Matrizen koennen mit dieser Prozedur wieder eingelesen werden.
```

```
   Die Anzahlen der Zeilen und Spalten der einzulesenden Matrix
   duerfen die entsprechenden Werte von 'mat' nicht uebersteigen.
   Evtl. ueberzaehlige Zeilen und Spalten von 'mat' werden mit
   Nullen belegt.
*)

PROCEDURE ReadF(VAR f: File; mat: tMat);
(* ... analog 'Read', jedoch erfolgt die Ausgabe in die Datei 'f' *)

PROCEDURE Add(mat1, mat2, res: tMat);
(* Die Matrizen 'mat1' und 'mat2' werden addiert. Das Ergebnis
   wird in 'res' gespeichert. 'mat1' und 'mat2' muessen die
   gleiche Groesse besitzen. 'mat1', 'mat2' und 'res' koennen
   dieselben Matrizen sein.
   Es werden Zaehlprozeduren des Moduls 'Pram' zur Protokollierung
   des Berechnungsaufwandes aufgerufen.
*)

PROCEDURE Sub(mat1, mat2, res: tMat);
(* Matrix 'mat2' wird von 'mat1' elementweise subtrahiert. Alles
   weitere ist gleich zu 'Add'.
*)

PROCEDURE Mult(mat1, mat2, res: tMat);
(* Die 'a*b'-Matrix 'mat1' wird mit der 'b*c'-Matrix 'mat2'
   multipliziert. Das Ergebnis wird in der 'a*c'-Matrix 'res'
   gespeichert. Die Zeilenanzahl 'a' und die Spaltenanzahl 'c'
   koennen zwischen 1 und 'ArrayMax' liegen. 'mat1', 'mat2' und
   'res' koennen dieselben Matrizen sein.
   Es werden Zaehlprozeduren des Moduls 'Pram' zur Protokollierung
   des Berechnungsaufwandes aufgerufen.
*)

PROCEDURE CreateMult(mat1, mat2: tMat): tMat;
(* 'CreateMult' arbeitet wie 'Mult', jedoch wird fuer das
   Ergebnis eine Matrix passender Groesse neue angelegt und als
   Funktionswert zurueckgegeben. Die Variable, der dieser Funktions-
   wert zugewiesen wird, darf vorher nicht mit 'Use' initialisiert
   worden sein.
*)

(* SCALar MULTiplication*)
PROCEDURE ScalMult(num: LONGREAL; mat1, res: tMat);
(* Die Matrix 'mat1' wird mit 'num' multipliziert. Das
   Ergebnis wird in 'res' gespeichert. 'mat1' und 'res'
   duerfen dieselben Matrizen sein.
   Es werden Zaehlprozeduren des Moduls 'Pram' zur Protokollierung
   des Berechnungsaufwandes aufgerufen.
*)

PROCEDURE Trace(mat: tMat): LONGREAL;
(* Das Funktionsergebnis ist die Summe der Elemente der
   Hauptdiagonalen (Spur) von 'mat'.
   Es werden Zaehlprozeduren des Moduls 'Pram' zur Protokollierung
   des Berechnungsaufwandes aufgerufen.
*)

END Rema.
```

B.22 Implementierungsmodul 'Rema'

```
IMPLEMENTATION MODULE Rema; (* REal MAtrix *)

(*          2-dimensionale LONGREAL-Matrizen

   (Erlaeuterungen im Definitionsmodul)
*)

FROM SYSTEM IMPORT TSIZE;
FROM InOut IMPORT WriteLn, WriteCard, ReadLCard,
                  ReadLReal, WriteReal, WriteString, ReadLn,
                  ReadString;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
IMPORT Sys, SysMath, Func, Rnd, Type, Frag, List, Simptype,
       Inli, Cali, Reli, Mat, Pram;
FROM Sys IMPORT tPOINTER;
FROM SysMath IMPORT LInt2LReal, LInt2Int, Int2LInt,
                    LReal2LInt,
                    Card2LCard, LCard2Card;
FROM Simptype IMPORT RealId, NewReal, pReal,
                     CardId, NewCard, pCard;
FROM Func IMPORT Error;

CONST RndCharSize= 50;
          (* maximale Groesse eines Elements der Matrix 'a'
             in 'Randomize' *)
      RndMixSize = 9.0;
          (* maximale Groessen von Elementen der Matrizen
             's' und 'si' in 'Randomize' *)

TYPE
    tMat= POINTER TO tMatRecord;
    tMatRecord= RECORD
                    longreal: BOOLEAN;
                        (* TRUE: die Matrizenelemente duerfen
                                 Nachkommastellen besitzen *)
                    rank: LONGCARD;
                        (* mit Hilfe von 'SetRank'
                           festgelegter Wert *)
                    mult: Cali.tCali;
                        (* LONGCARD-Liste mit gewuenschten
                           Vielfachheiten von Eigenwerten beim
                           Erzeugen durch 'Randomize' *)
                    det: LONGREAL;
                        (* Nachdem eine Matrix durch 'Randomize'
                           generiert wurde, steht in dieser
                           Komponente die Determinante der
                           generierten Matrix. *)
                    values : Mat.tMat
                END;
(* Prozeduren, die an die fuer 'values' verwendete
   Datenstruktur angepasst werden muessen:
       Use, DontUse, Elem und Set
*)

VAR ElemGen: Rnd.tGen;
        (* Zufallsgenerator fuer Matrizenelemente *)
    IntGen: Rnd.tGen;
        (* Zufallsgenerator zur Erzeugung von zu kombinierenden
           Zeilenindizes *)
    SignGen: Rnd.tGen;
        (* Zufallsgenerator zur Erzeugung von Vorzeichen *)
    MatId: Type.Id;
        (* Typnummer 'Rema.tMat' *)
```

```
PROCEDURE RemaNewI(hilf: tPOINTER);
VAR
    mat: tMat;
BEGIN
    mat:= hilf;
    Cali.Use(mat^.mult);
    Mat.Use(mat^.values, RealId())
END RemaNewI;

PROCEDURE Use(VAR mat: tMat; row, col: LONGCARD);
BEGIN
    mat:= tMat( Type.NewI(MatId) );
    SetSize(mat, row, col)
END Use;

PROCEDURE RemaDelI(hilf: tPOINTER);
VAR
    mat: tMat;
BEGIN
    mat:= hilf;
    List.DontUse(mat^.mult);
    Mat.DontUse(mat^.values);
END RemaDelI;

PROCEDURE DontUse(mat: tMat);
BEGIN
    Type.DelI(MatId, mat)
END DontUse;

PROCEDURE Empty(mat: tMat);
BEGIN
    Mat.Empty(mat^.values)
END Empty;

PROCEDURE Unit(mat: tMat);
VAR i: LONGCARD;
BEGIN
    Empty(mat);
    FOR i:= 1 TO Func.MinLCard(Rows(mat), Columns(mat)) DO
        Set(mat,i,i,1.0)
    END
END Unit;

PROCEDURE Assign(mat1, mat2: tMat);
VAR
    i,j: LONGCARD;
BEGIN
    IF (Rows(mat1) # Rows(mat2)) OR
       (Columns(mat1) # Columns(mat2))
    THEN
        Error("Mat.Assign",
            "Die Matrizen besitzen nicht die gleiche Groesse.")
    END;
    FOR i:= 1 TO Rows(mat1) DO
        FOR j:= 1 TO Columns(mat2) DO
            Set(mat2,i,j,Elem(mat1,i,j))
        END
    END
END Assign;

PROCEDURE Copy(mat1: tMat): tMat;
VAR res: tMat;
BEGIN
    Use(res, Rows(mat1), Columns(mat1));
    Assign(mat1, res);
```

```
    RETURN res
END Copy;

PROCEDURE Elem(mat: tMat; row, col: LONGCARD): LONGREAL;
VAR erg: pReal;
BEGIN
    IF (row > Rows(mat) ) OR
       (col > Columns(mat) )
    THEN
        Error("Mat.Elem",
            "Der verwendete Index ist zu gross.")
    END;
    erg:= pReal( Mat.Elem(mat^.values, row, col) );
    IF erg # pReal(NIL) THEN
        RETURN erg^
    ELSE
        RETURN 0.0
    END
END Elem;

PROCEDURE Set(mat: tMat; row, col: LONGCARD; val: LONGREAL);
VAR item: pReal;
BEGIN
    item:= NewReal(val);
    Mat.Set(mat^.values, row, col, item)
END Set;

PROCEDURE Rows(mat: tMat): LONGCARD;
BEGIN
    RETURN Mat.Rows(mat^.values)
END Rows;

PROCEDURE Columns(mat: tMat): LONGCARD;
BEGIN
    RETURN Mat.Columns(mat^.values)
END Columns;

PROCEDURE SetSize(mat: tMat; r,c: LONGCARD);
BEGIN
    Mat.SetSize(mat^.values, r, c);
    List.Empty(mat^.mult);
    Cali.InsertBefore(mat^.mult, 0);
    mat^.longreal:= FALSE;
    mat^.rank:= Func.MinLCard(r,c);
    mat^.det:= 0.0
END SetSize;

PROCEDURE SetReal(mat: tMat; real: BOOLEAN);
BEGIN
    mat^.longreal:= real
END SetReal;

PROCEDURE SetRank(mat: tMat; rank: LONGCARD);
BEGIN
    IF (rank <= 0) OR
       (Func.MinLCard( Rows(mat), Columns(mat) ) < rank)
    THEN
        Error("Mat.SetRank",
            "Der gewuenschte Rang ist zu gross.")
    END;
    mat^.rank:= rank
END SetRank;

PROCEDURE SetMultiplicity(mat: tMat; mult: LONGCARD);
(* Verbesserungsmoeglichkeit:
```

```
   Pruefung der Liste der Vielfachheiten auf Plausibilitaet *)
BEGIN
    List.First(mat^.mult);
    Cali.InsertBefore(mat^.mult,mult);
END SetMultiplicity;

PROCEDURE GetRndDiagonal(
              a: tMat; rank: LONGCARD;
              MultList: Cali.tCali; max: LONGCARD; decimal: BOOLEAN
         );
(* In 'a' wird eine zufaellig bestimmte Diagonalmatrix mit dem
   Rang 'rank' und Eigenwerten der in 'MultList' angegebenen
   Vielfachheiten. Die Elemente von 'a' besitzen hoechstens
   die Groesse 'max'. Falls in 'decimal' der Wert TRUE uebergeben
   wird, besitzen die Elemente von 'a' Nachkommastellen. *)
VAR
    num: LONGCARD;
        (* Nummer des Elements der Hauptdiagonalen, dass
           zu erzeugen ist *)
    mult: LONGCARD;
        (* aktuell zu verarbeitende Vielfachheit *)
    CharVal: LONGREAL;
        (* naechster zu verwendender Eigenwert zum Erzeugen
           der Diagonalmatrix *)
    size: LONGCARD;
        (* Minimum von Anzahl der Zeilen und Anzahl der Spalten
           von 'a' *)
BEGIN
    size:= Rows(a); (* Die Matrizen sind quadratisch. *)
    Empty(a);
    mult:= 0;
    List.First(MultList);
    num:= 1;
    IF decimal THEN
        Rnd.Range(ElemGen, 0.0,
            (LInt2LReal(max) / LInt2LReal(size)) );
        CharVal:= Rnd.LongReal(ElemGen)
    ELSE
        Rnd.Range(ElemGen, 1.0,
            LInt2LReal( max DIV size ));
        CharVal:= LInt2LReal(Rnd.Int(ElemGen))
    END;
    WHILE num <= rank DO
        IF mult <= 0 THEN
            Set(a, num, num, CharVal);
            IF decimal THEN
                CharVal:= CharVal + Rnd.LongReal(ElemGen)
            ELSE
                CharVal:= CharVal + LInt2LReal(Rnd.Int(ElemGen))
            END;
            IF NOT List.AtLast(MultList) THEN
                mult:= Cali.Cur(MultList);
                List.Next(MultList)
            ELSE
                mult:= 1
            END
        ELSE
            Set(a, num, num, Elem(a,num-1,num-1))
        END;
        INC(num);
        DEC(mult)
    END
END GetRndDiagonal;

PROCEDURE RndMixIndices(list: Cali.tCali; max: LONGCARD);
```

```
(* Diese Prozedur liefert in 'list' die Zahlen von 1 bis 'max'
   in zufaelliger Reihenfolge. *)
VAR
    MixArray: Frag.tFrag;
        (* Feld zum Mischen der Zeilenindizes *)
    i: LONGCARD; (* Zaehler *)
    SwapCount: LONGCARD;
        (* Anzahl der Vertauschungen von Elementen von
           'MixArray', die zur Produktion von Unordnung
           durchzufuehren sind *)
    hilf: tPOINTER;
        (* Hilfsvar. zum Vertauschen von Elementen in
           'MixArray' *)
    hilf2: pCard;
    i1, i2: LONGCARD;
        (* Indizes der zu vertauschenden Elemente in
           'MixArray' *)
BEGIN
    Frag.Use(MixArray, CardId(), 1, max);
    Rnd.Range(IntGen, 1.0, LInt2LReal(max) );
    SwapCount:= 3 * max;
    FOR i:= 1 TO max DO
        Frag.SetItem(MixArray, i, tPOINTER(NewCard(i)))
    END;
    Frag.AddRef(MixArray, TRUE);
    FOR i:= 1 TO SwapCount DO
        i1:= Rnd.Int(IntGen);
        i2:= Rnd.Int(IntGen);
        hilf:= Frag.GetItem(MixArray, i1);
        Frag.SetItem(MixArray, i1, Frag.GetItem(MixArray, i2));
        Frag.SetItem(MixArray, i2, hilf)
    END;
    Frag.AddRef(MixArray, FALSE);
    List.Empty(list);
    FOR i:= 1 TO max DO
        hilf2:= pCard( Frag.GetItem(MixArray, i) );
        Cali.InsertBehind(list, hilf2^)
    END;
    Frag.DontUse(MixArray)
END RndMixIndices;

PROCEDURE GetRndInverse(s,si: tMat; max: LONGREAL);
(* In 's' wird eine zufaellige invertierbare Matrix und in 'si'
   ihre Inverse zurueckgegeben. Die Elemente beider Matrizen sind
   ganzzahlig und ihr maximaler Betrag ist 'max'. *)
VAR
    MixedList: Cali.tCali;
        (* Liste der gemischten Zeilenindizes *)
    AddedList: Inli.tInli;
        (* Liste der zueinander addierten / voneinander
           subtrahierten Zeilen in 's'; die gleichen
           Zeilenoperationen werden in umgekehrter
           Reihenfolge verwendet, um 'si' zu
           berechnen; Bedeutung der Listenelemente:
           1.: Index der Zeile die addiert wird,
           2.: Index der Zeile zu der addiert wird,
           3.: verwendetes Vorzeichen
             ...
         *)
    In1, In2: LONGCARD;
        (* zu kombinierende Zeilen bzw. Spalten *)
    In: LONGCARD;
        (* aktueller Index beim Verknuepfen von Zeilen
           bzw. Spalten *)
    MaxElem: LONGREAL;
```

```
        (* maximaler Betrag eines Matrizenelements *)
    sign: LONGREAL;
        (* -1.0 : Zeilen in 's' werden voneinander abgezogen;
            1.0 : Zeilen in 's' werden zueinander addiert *)
    size: LONGCARD;
        (* Groesse von 's' und 'si' *)
BEGIN
    Cali.Use(MixedList);
    Inli.Use(AddedList);
    size:= Rows(s); (* Die Matrizen sind quadratisch. *)

    (* berechne 's': *)
    Unit(s);
    MaxElem:= 0.0;
    sign:= 1.0;
    REPEAT
        RndMixIndices(MixedList, size);
        List.First(MixedList);
        REPEAT
            In1:= LCard2Card( Cali.OutCur(MixedList) );
            In2:= LCard2Card( Cali.OutCur(MixedList) );
            FOR In:= 1 TO size DO
                Set(s, In2, In,
                        Elem(s, In2, In)
                        + Elem(s, In1, In) * sign
                );
                MaxElem:= Func.MaxReal(
                             MaxElem, ABS( Elem(s, In2, In) )
                          )
            END;
            Inli.InsertBefore(AddedList,
                LInt2Int( LReal2LInt(sign) )
            );
            Inli.InsertBefore(AddedList, In2);
            Inli.InsertBefore(AddedList, In1);
            sign:= LInt2LReal(Rnd.Int(SignGen)) * 2.0 - 1.0
                (* ergibt 1.0 oder -1.0 (!) *)
        UNTIL List.Count(MixedList) <= 1
    UNTIL MaxElem > (max / 2.0);

    (* berechne 'si': *)
    List.First(AddedList);
    Unit(si);
    WHILE List.Count(AddedList) > 0 DO
        In1:= Inli.OutCur(AddedList);
        In2:= Inli.OutCur(AddedList);
        sign:= LInt2LReal( Inli.OutCur(AddedList) );
        FOR In:= 1 TO size DO
            Set(si, In2, In, Elem(si, In2, In)
                            + Elem(si, In1, In) * (- sign)
            )
        END
    END;
    List.DontUse(MixedList);
    List.DontUse(AddedList)
END GetRndInverse;

PROCEDURE TestInverse(a,b: tMat);
(* Diese Prozedur dient zur Fehlersuche.
   Falls 'a' nicht die Inverse von 'b' ist, wird eine Fehlermeldung
   ausgegeben. *)
VAR
    res: tMat;
    i,j: LONGCARD;
    fehler: BOOLEAN;
```

```
    input: ARRAY [1..2] OF CHAR;
BEGIN
    Use(res, Rows(a), Columns(b));
    Mult(a, b, res);
    FOR i:= 1 TO Rows(res) DO
        FOR j:= 1 TO Columns(res) DO
            IF i = j THEN
                fehler:= Elem(res, i, j) # 1.0
            ELSE
                fehler:= Elem(res, i, j) # 0.0
            END;
            IF fehler THEN
                WriteString("*** Mat.TestInverse:"); WriteLn;
                WriteString("*** Matrizen nicht invers");
                WriteString(" zueinander"); WriteLn;
                WriteString("*** sollte Einheitsmatrix sein:");
                WriteLn;
                Write(res);
                WriteString("   <RETURN> ..."); ReadString(input);
                Write(a);
                WriteString("   <RETURN> ..."); ReadString(input);
                Write(b);
                HALT;
                RETURN
            END
        END
    END;
    WriteString("*** Mat.TestInverse: OK");
    WriteLn
END TestInverse;

PROCEDURE Randomize(mat: tMat);
VAR
    a, s, si, sa: tMat;
        (* a : zufaellig zu erzeugende Diagonalmatrix
           s : zufaellige invertierbare Matrix
           si: Inverse von 's'
           sa:= s * a

           Bedingungen fuer die Matrizengroesse (Indizes: i,j,k):
           - Voraussetzungen: mat: i*j, a: i*j
           - sa:= s * a  ==> s: k*i, sa: k*j
           - mat:= sa * si ==> si: j*j, k=i
           - si Inverse von s (damit 'mat' und 'a' die gleichen
             Eigenwerte besitzen) ==> i=j
        *)
    size: LONGCARD;
        (* Anzahl der Zeilen und Spalten der zu verarbeitenden
           Matrizen *)
BEGIN
    IF Rows(mat) # Columns(mat) THEN
        WriteLn;
        WriteString("*** Mat.Randomize:"); WriteLn;
        WriteString("*** Die zu erzeugende Matrix ist ");
        WriteString("nicht quadratisch."); WriteLn;
        HALT
    END;
    size:= Rows(mat);
    Use(a,size,size); Use(s,size,size);
    Use(sa,size,size); Use(si,size,size);

    (* erzeuge zufaellige Diagonalmatrix 'a': *)
    GetRndDiagonal(a, mat^.rank, mat^.mult,
                   Func.MaxLCard(size + 10, size * 2),
                   mat^.longreal);
```

```
    mat^.det:= Trace(a);

    (* berechne zufaellige invertierbare Matrix 's' sowie deren
       Inverse 'si' : *)
    GetRndInverse(s, si, RndMixSize);

    (* zur Fehlersuche: *)
    TestInverse(s, si);

    (* mat := s * a * si
       ('mat' hat dieselben Eigenwerte wie 'a'): *)
    Mult(s,a,sa);
    Mult(sa,si,mat);

    DontUse(a); DontUse(s); DontUse(si); DontUse(sa)
END Randomize;

PROCEDURE Det(mat: tMat): LONGREAL;
BEGIN
    RETURN mat^.det
END Det;

PROCEDURE Write(mat: tMat);
VAR
    row,col: LONGCARD;
    i, j: LONGCARD;
BEGIN
    row:= Rows(mat);
    col:= Columns(mat);
    WriteString("Zeilen: "); WriteLn;
    WriteString("    "); WriteCard(row,0); WriteLn;
    WriteString("Spalten: "); WriteLn;
    WriteString("    "); WriteCard(col,0); WriteLn;
    WriteString("Rang: "); WriteLn;
    WriteString("    "); WriteCard(mat^.rank, 0); WriteLn;
    WriteString("Determinante: "); WriteLn;
    WriteString("    "); WriteReal(mat^.det,12,4); WriteLn;
    WriteString("Nachkommastellen (1: ja, 0: nein): ");
    WriteLn; WriteString("    ");
    IF mat^.longreal THEN
        WriteCard(1,0)
    ELSE
        WriteCard(0,0)
    END;
    WriteLn;
    WriteString("Vielfachheiten ( >1 ) von Eigenwerten");
    WriteString(" (Anzahl, Wert 1, Wert 2, ...):");
    WriteLn;
    WriteCard(List.Count(mat^.mult) - 1, 5);
    List.First(mat^.mult);
    WHILE NOT List.AtLast(mat^.mult) DO
        WriteCard(Cali.Cur(mat^.mult),5);
        List.Next(mat^.mult)
    END;
    WriteLn;

    FOR i:= 1 TO row DO
        FOR j:= 1 TO col DO
            WriteReal(Elem(mat,i,j), 12, 4);
            WriteString(" ")
        END;
        IF col<=MaxIoRow THEN WriteLn END
    END
END Write;
```

```
PROCEDURE WriteF(VAR f: Sys.File; mat: tMat);
VAR
    row,col: LONGCARD;
    i, j: LONGCARD;
BEGIN
    row:= Rows(mat);
    col:= Columns(mat);
    Sys.WriteString(f, "Zeilen: "); Sys.WriteLn(f);
    Sys.WriteString(f,"    ");
    Sys.WriteCard(f, row, 0); Sys.WriteLn(f);

    Sys.WriteString(f, "Spalten: "); Sys.WriteLn(f);
    Sys.WriteString(f,"    ");
    Sys.WriteCard(f, col, 0); Sys.WriteLn(f);

    Sys.WriteString(f, "Rang: "); Sys.WriteLn(f);
    Sys.WriteString(f,"    ");
    Sys.WriteCard(f, mat^.rank, 0); Sys.WriteLn(f);

    Sys.WriteString(f, "Determinante: "); Sys.WriteLn(f);
    Sys.WriteString(f, "    ");
    Sys.WriteReal(f, mat^.det,12,4); Sys.WriteLn(f);

    Sys.WriteString(f, "Nachkommastellen (1: ja, 0: nein): ");
    Sys.WriteLn(f); Sys.WriteString(f,"    ");
    IF mat^.longreal THEN
        Sys.WriteCard(f, 1, 0)
    ELSE
        Sys.WriteCard(f, 0, 0)
    END;
    Sys.WriteLn(f);
    Sys.WriteString(f,
        "Vielfachheiten (groesser als 1) von Eigenwerten");
    Sys.WriteString(f, " (Anzahl, Wert 1, Wert 2, ...):");
    Sys.WriteLn(f);
    Sys.WriteCard(f, List.Count(mat^.mult) - 1, 5);
    List.First(mat^.mult);
    WHILE NOT List.AtLast(mat^.mult) DO
        Sys.WriteCard(f, LCard2Card( Cali.Cur(mat^.mult) ), 5);
        List.Next(mat^.mult)
    END;
    Sys.WriteLn(f);

    FOR i:= 1 TO row DO
        FOR j:= 1 TO col DO
            Sys.WriteReal(f, Elem(mat,i,j), 12, 4);
            IF j < col THEN
                Sys.WriteString(f, " ")
            END
        END;
        IF col<=MaxIoRow THEN Sys.WriteLn(f) END
    END
END WriteF;

PROCEDURE Read(mat: tMat);
VAR
    row,col: LONGCARD;
    i, j   : LONGCARD;
    hilf   : LONGCARD;
    num    : LONGREAL;
BEGIN
    ReadLn; ReadLCard(row);
    ReadLn; ReadLCard(col);
    SetSize(mat, row, col);
    ReadLn; ReadLCard(mat^.rank);
```

```
    ReadLn; ReadLReal(mat^.det);
    ReadLn; ReadLCard(hilf);
    mat^.longreal:=  hilf = 1;
    ReadLn; ReadLn;
    ReadLCard(hilf);
    FOR i:= 1 TO hilf DO
        ReadLCard(j);
        Cali.InsertBefore(mat^.mult, j)
    END;
    ReadLn;

    FOR i:= 1 TO row DO
        FOR j:= 1 TO col DO
            ReadLReal(num);
            Set(mat,i,j,num)
        END
    END
END Read;

PROCEDURE ReadF(VAR f: Sys.File; mat: tMat);
VAR
    row,col: LONGCARD;
    i, j   : LONGCARD;
    hilf: LONGCARD;
    num    : LONGREAL;
    dummy  : ARRAY [1..80] OF CHAR;
BEGIN
    Sys.ReadLn(f); Sys.ReadLCard(f,row);
    Sys.ReadLn(f); Sys.ReadLCard(f,col);
    SetSize(mat, row, col);
    Sys.ReadLn(f); Sys.ReadLCard(f,mat^.rank);
    Sys.ReadLn(f); Sys.ReadReal(f,mat^.det);
    Sys.ReadLn(f); Sys.ReadLCard(f,hilf);
    mat^.longreal:=  hilf = 1;
    Sys.ReadLn(f);
    Sys.ReadLCard(f, hilf);
    FOR i:= 1 TO hilf DO
        Sys.ReadLCard(f, j);
        Cali.InsertBefore(mat^.mult,j)
    END;

    FOR i:= 1 TO row DO
        FOR j:= 1 TO col DO
            Sys.ReadReal(f,num);
            Set(mat, i, j, num)
        END
    END
END ReadF;

PROCEDURE CheckEqualSize(a,b: tMat; proc: ARRAY OF CHAR);
(* Es wird eine Fehlermeldung ausgegeben, falls die Matrizen
   'a' und 'b' nicht die gleiche Groesse besitzen. *)
BEGIN
    IF (Rows(a) # Rows(b)) OR (Columns(a) # Columns(b)) THEN
        Error(proc,
            "Die Matrizen sind verschieden gross.");
    END
END CheckEqualSize;

PROCEDURE Add(mat1, mat2, res: tMat);
VAR
    i,j: LONGCARD;
    hilf: tMat;
    RowsMat1, ColumnsMat1: LONGCARD;
BEGIN
```

```
    Use(hilf,Rows(res),Columns(res));
    CheckEqualSize(mat1, mat2, "Mat.Add");
    CheckEqualSize(mat2, res, "Mat.Add");

    RowsMat1:= Rows(mat1); ColumnsMat1:= Columns(mat1);
    FOR i:= 1 TO RowsMat1 DO
        FOR j:= 1 TO ColumnsMat1 DO
            Set(hilf, i, j, Elem(mat1, i, j) + Elem(mat2, i, j))
        END
    END;
    (* Alle Schleifendurchlaeufe werden parallel durchgefuehrt: *)
    Pram.Prozessoren(RowsMat1 * ColumnsMat1);
    Pram.Schritte(1);

    Assign(hilf,res);
    DontUse(hilf)
END Add;

PROCEDURE Sub(mat1, mat2, res: tMat);
VAR
    i,j: LONGCARD;
    hilf: tMat;
    RowsMat1, ColumnsMat1: LONGCARD;
BEGIN
    Use(hilf,Rows(res),Columns(res));
    CheckEqualSize(mat1, mat2, "Mat.Sub");
    CheckEqualSize(mat2, res, "Mat.Sub");

    RowsMat1:= Rows(mat1); ColumnsMat1:= Columns(mat1);
    FOR i:= 1 TO RowsMat1 DO
        FOR j:= 1 TO ColumnsMat1 DO
            Set(hilf, i, j, Elem(mat1, i, j) - Elem(mat2, i, j))
        END
    END;
    (* Alle Schleifendurchlaeufe werden parallel durchgefuehrt: *)
    Pram.Prozessoren(RowsMat1 * ColumnsMat1);
    Pram.Schritte(1);

    Assign(hilf,res);
    DontUse(hilf)
END Sub;
            (* a*b, b*c, a*c *)
PROCEDURE Mult(mat1,mat2,res: tMat);
VAR
    i,j,k: LONGCARD;
    hilf: tMat;
    RowsRes, ColumnsRes, ColumnsMat1: LONGCARD;
    AddList: Reli.tReli;
BEGIN
    Reli.Use(AddList);
    Use(hilf, Rows(res), Columns(res));
    IF (Columns(mat1) # Rows(mat2)) OR
       (Rows(mat1) # Rows(res)) OR
       (Columns(mat2) # Columns(res))
    THEN
        Error("Mat.Mult",
            "Die Groessen der Matrizen passen nicht zusammen.")
    END;
    RowsRes:= Rows(res); ColumnsRes:= Columns(res);
    ColumnsMat1:= Columns(mat1);

    Pram.ParallelStart("Rema.Mult");
    FOR i:= 1 TO RowsRes DO
        Pram.ParallelStart("Rema.Mult:Zeile");
        FOR j:= 1 TO ColumnsRes DO
```

```
            List.Empty(AddList);

            FOR k:= 1 TO ColumnsMat1 DO
                Reli.InsertBehind(AddList,
                     Elem(mat1, i, k)*Elem(mat2, k, j)
                );
            END;
            (* Die Durchlaeufe der Schleife werden parallel durchge-
               fuehrt: *)
            Pram.Prozessoren( ColumnsMat1 );
            Pram.Schritte( 1 );

            Set( hilf, i, j, Pram.AddList(AddList) );

            Pram.NaechsterBlock("Rema.Mult:Zeile");
        END;
        Pram.ParallelEnde("Rema.Mult:Zeile");

        Pram.NaechsterBlock("Rema.Mult");
    END;
    Pram.ParallelEnde("Rema.Mult");

    Assign(hilf,res);
    DontUse(hilf);
    List.DontUse(AddList)
END Mult;

PROCEDURE CreateMult(mat1, mat2: tMat): tMat;
VAR res: tMat;
BEGIN
    Use(res, Rows(mat1), Columns(mat2));
    Mult(mat1, mat2, res);
    RETURN res
END CreateMult;

PROCEDURE ScalMult(num: LONGREAL; mat1,res: tMat);
VAR
    i,j: LONGCARD;
    RowsRes, ColumnsRes: LONGCARD;
BEGIN
    RowsRes:= Rows(res); ColumnsRes:= Columns(res);
    FOR i:= 1 TO RowsRes DO
        FOR j:= 1 TO ColumnsRes DO
            Set(res,i,j,num*Elem(mat1,i,j))
        END
    END;
    (* Alle Schleifendurchlaeufe werden parallel durchgefuehrt: *)
    Pram.Prozessoren( RowsRes * ColumnsRes );
    Pram.Schritte( 1 )
END ScalMult;

PROCEDURE Trace(mat: tMat): LONGREAL;
(* Das Funktionsergebnis ist die Summe der Elemente der
   Hauptdiagonalen (Spur) der angegebenen Matrix. *)
VAR
    res: LONGREAL;  (* Funktionsergebnis *)
    i: LONGCARD;
    size: LONGCARD; (* Groesse von 'mat' *)
    AddList: Reli.tReli;
BEGIN
    Reli.Use(AddList);
    size:= Func.MinLCard( Rows(mat), Columns(mat) );

    FOR i:= 1 TO size DO
        Reli.InsertBehind( AddList, Elem(mat, i, i) )
```

```
    END;
    res:= Pram.AddList(AddList);

    List.DontUse(AddList);
    RETURN res
END Trace;

BEGIN
    (* initialisiere Zufallsgeneratoren: *)
    Rnd.Use(ElemGen);
    Rnd.Use(IntGen);
    Rnd.Use(SignGen);
    Rnd.Range(SignGen, 0.0, 1.0);

    MatId:= Type.New(TSIZE(tMatRecord));
    Type.SetName(MatId, "Rema.tMat");
    Type.SetNewProc(MatId, RemaNewI);
    Type.SetDelProc(MatId, RemaDelI)
END Rema.
```

B.23 Definitionsmodul 'Rnd'

```
DEFINITION MODULE Rnd;

(*        Zufallszahlengeneratoren

   Mit Hilfe der linearen Kongruenzmethode werden gleich-
   oder normalverteilte Zufallszahlen erzeugt.

   Das Modul verwaltet die Erzeugung von Zufallszahlen objekt-
   orientiert. D. h. es koennen Zufallszahlengeneratoren unter-
   schiedlicher Charakterististiken vereinbart und nebenein-
   ander benutzt werden.
*)

IMPORT SysMath, Func, Sys;

TYPE tGen; (* Dieser Typ muss fuer eine Variable vom Typ
              'Zufallsgenerator' benutzt werden. *)

PROCEDURE Use(VAR gen: tGen);
(* Bevor ein Zufallsgenerator benutzt werden soll, muss 'Use'
   fuer die entsprechende Variable aufgerufen werden. *)

PROCEDURE DontUse(VAR gen: tGen);
(* Falls ein Zufallsgenerator nicht mehr benutzt werden soll,
   insbesondere, wenn der aktuelle Parameter fuer 'gen' eine
   lokale Variable ist, muss 'DontUse' fuer diese Variable
   aufgerufen werden. *)

PROCEDURE Start(gen: tGen; num: LONGCARD);
(* 'num' wird als neuer Startwert fuer den Generator 'gen'
   verwendet.
   Fuer alle Generatoren werden Startwerte voreingestellt,
   um neue Folgen von Zufallszahlen zu erhalten sollte jedoch
   zumindest fuer den ersten mit 'Use' angelegten Generator
   ein jeweils anderer Startwert festgelegt werden.
   'num' muss ein Wert im Bereich von 0 bis 199017 sein.
*)
```

```
PROCEDURE Range(gen: tGen; bot,top: LONGREAL);
(* F"ur 'gen' wird der Bereich festgelegt, in dem die Zufalls-
   zahlen liegen sollen. Voreingestellt ist 'bot=0' und 'top=1'.
   Durch ganzzahlige Zufallszahlen koennen diese Randwerte
   erreicht werden.
   Die Grenzen duerfen auch nachtraeglich veraendert werden,
   ohne dass dies zu Problemen im Modul fuehrt.
*)

PROCEDURE Int(gen: tGen):LONGINT;
(* Mit Hilfe des Generators 'gen' wird als Funktionswert eine
   Zufallszahl entsprechend der vorher mit den anderen Funktionen
   festgelegten Parameter erzeugt.
   Die Zahlen sind im mit 'Range' festgelegten Bereich gleichmaessig
   verteilt.
*)

PROCEDURE LongReal(gen: tGen): LONGREAL;
(* analog 'Int', jedoch wird eine Zahl mit Nachkommastellen
   erzeugt *)

PROCEDURE St(gen: tGen; num: LONGREAL);
(* Fuer den Generator 'gen' wird 'num' als Standardabweichung
   fuer die Erzeugung von Zufallszahlen mit Hilfe von 'Norm'
   festgelegt. Voreingestellt ist 1.
*)

PROCEDURE Mid(gen: tGen; num: LONGREAL);
(* Fuer den Generator 'gen' wird 'num' als Mittelwert fuer die
   Erzeugung von Zufallszahlen mit Hilfe von 'Norm' festgelegt.
   Voreingestellt ist 0.
*)

PROCEDURE Norm(gen: tGen): LONGREAL;
(* Das Funktionsergebnis ist eine mit Hilfe von 'gen' erzeugte
   Zufallszahl entsprechend der vorher mit 'St' und 'Mid'
   eingestellten Parameter.
*)

END Rnd.
```

B.24 Implementierungsmodul 'Rnd'

```
IMPLEMENTATION MODULE Rnd;

(*  Erzeugung von Zufallszahlen

   ( Erklaerungen im Definitionsmodul )
*)

FROM SYSTEM IMPORT TSIZE;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM InOut IMPORT WriteLn, WriteString;
IMPORT SysMath, Func, Sys;
FROM SysMath IMPORT sqrt, cos, ln, entier, real;

CONST
    RndInfoFile= "rnd.inf";
        (* Datei mit Anfangszahl fuer Zufallsgenerator *)

    (* Konstanten zur Erzeugung von Zufallszahlen
```

```
       (Es muss mit LONGREAL-Zahlen gerechnet werden,
        da der Wertebereich von 'LONGCARD' nicht
        ausreicht) : *)
    a= 24298.0;
    c= 99991.0;
    m= 199017.0;
    pi= 3.1415926;
    (* Konstanten zur Voreinstellung von Generatorparametern: *)
    PreX= 42.0; (* Dieser Wert wird als Startwert fuer das
                   Modul benutzt, falls 'RndInfoFile' nicht
                   existiert. *)
    PreNewX = 7.0; (* Um diesen Wert wird der Startwert fuer den
                      letzten neu angelegten Generator inkre-
                      mentiert (modulo 'm'), um den Startwert fuer
                      den naechsten Generator voreinzustellen. *)
    PreNewNextX = 11.0;
        (* Um diesen Wert wird der aus 'RndInfoFile' gelesene
           Startwert inkrementiert (modulo 'm') um den Wert
           fuer den naechsten Programmstart zu erhalten. *)
    PreHi= 1.0; (* untere Grenze *)
    PreLo= 0.0; (* obere Grenze *)
    PreMid = 0.0; (* Mittelwert *)
    PreSt= 1.0; (* Standardabweichung *)

TYPE
   tGen = POINTER TO tGenRec;
   tGenRec = RECORD (* einzelner Generator *)
                x: LONGREAL;
                    (* letzte erzeugte Zufallszahl *)
                lo,hi: LONGREAL;
                    (* untere bzw. obere Grenze des Bereichs,
                       in dem die Zahlen liegen sollen *)
                mid, st: LONGREAL;
                    (* Mittelwert, Varianz *)
             END;

VAR
   NextX: LONGREAL; (* naechster fuer einen Generator zur
                       Voreinstellung zu benutzt Startwert *)

PROCEDURE Use(VAR gen: tGen);
BEGIN
    ALLOCATE(gen, TSIZE(tGenRec));
    WITH gen^ DO
        x  := NextX;
        lo := PreLo;
        hi := PreHi;
        mid:= PreMid;
        st := PreSt
    END;
    NextX:= Func.ModReal( NextX + PreNewX , m )
END Use;

PROCEDURE DontUse(VAR gen: tGen);
BEGIN
    DEALLOCATE(gen,TSIZE(tGenRec));
    gen:= NIL
END DontUse;

PROCEDURE Start(gen: tGen; num: LONGCARD);
BEGIN
    gen^.x:= real(num);
    NextX:= Func.ModReal( real(num) + PreNewX , m )
END Start;
```

```
PROCEDURE Range(gen: tGen; bot,top: LONGREAL);
BEGIN
    IF bot >= top THEN
        WriteLn;
        WriteString("*** Rnd.Range:"); WriteLn;
        WriteString("*** Die untere Intervallgrenze ist ");
        WriteLn;
        WriteString("groesser als die obere.");
        WriteLn;
        HALT
    END;
    gen^.lo:= bot;
    gen^.hi:= top
END Range;

PROCEDURE New(gen: tGen);
BEGIN
    gen^.x:= Func.ModReal( a * gen^.x + c , m )
END New;

PROCEDURE Int(gen: tGen): LONGINT;
BEGIN
    New(gen);
    RETURN entier( ( gen^.x / m )
                   * (gen^.hi + 1.0 - gen^.lo)
                   + gen^.lo
                 )
END Int;

PROCEDURE LongReal(gen: tGen): LONGREAL;
BEGIN
    New(gen);
    RETURN ( gen^.x / m )
           * (gen^.hi-gen^.lo)
           + gen^.lo
END LongReal;

PROCEDURE St(gen: tGen; num: LONGREAL);
BEGIN
    gen^.st:= num
END St;

PROCEDURE Mid(gen: tGen; num: LONGREAL);
BEGIN
    gen^.mid:= num
END Mid;

PROCEDURE Norm(gen: tGen): LONGREAL;
BEGIN
    RETURN gen^.st*(
                  sqrt( -2.0*ln( LongReal(gen) ) )
                  * cos( 2.0*pi*LongReal(gen) )
              ) + gen^.mid
END Norm;

VAR f: Sys.File;

BEGIN
    IF Sys.Exist(RndInfoFile) THEN
        Sys.OpenRead(f,RndInfoFile);
        Sys.ReadReal(f,NextX);
        Sys.Close(f)
    ELSE
        NextX:= PreX
    END;
```

```
    NextX:= Func.ModReal( NextX + PreNewNextX, m );
    Sys.OpenWrite(f,RndInfoFile);
    Sys.WriteReal(f,NextX,8,0);
    Sys.Close(f)
END Rnd.
```

B.25 Definitionsmodul 'Simptype'

```
DEFINITION MODULE Simptype; (* SIMPle TYPEs *)

(*    einfache Datentypen fuer das Modul 'Type'

   'Simptype' definiert die Typen 'LONGCARD', 'LONGINT' und
   'LONGREAL' fuer das Modul 'Type'

*)

IMPORT Sys, Type;
FROM Sys IMPORT tPOINTER;

TYPE pCard = POINTER TO LONGCARD;
     pInt  = POINTER TO LONGINT;
     pReal = POINTER TO REAL;
     pPoint = POINTER TO tPOINTER;

PROCEDURE CardId(): Type.Id;
(* Funktionsergebnis ist der Typindetifikator fuer 'LONGCARD' *)

PROCEDURE NewCard(val: LONGCARD): pCard;
(* Funktionsergebnis ist ein Zeiger auf eine Zahl vom Typ LONGCARD
   mit dem Wert 'val' *)

PROCEDURE DelCard(VAR val: pCard);
(* Der Speicherbereich fuer die Zahl vom Typ LONGCARD, auf die 'val'
   zeigt, wird freigegeben. *)

PROCEDURE IntId(): Type.Id;
(* ... analog 'CardId', jedoch fuer LONGINT *)

PROCEDURE NewInt(val: LONGINT): pInt;
(* ... analog 'NewCard', jedoch fuer LONGINT *)

PROCEDURE DelInt(VAR val: pInt);
(* ... analog 'DelCard', jedoch fuer LONGINT *)

PROCEDURE RealId(): Type.Id;
(* ... analog 'CardId', jedoch fuer LONGREAL *)

PROCEDURE NewReal(val: LONGREAL): pReal;
(* ... analog 'NewCard', jedoch fuer LONGREAL *)

PROCEDURE DelReal(VAR val: pReal);
(* ... analog 'DelCard', jedoch fuer LONGREAL *)

PROCEDURE PointId(): Type.Id;
(* ... analog 'CardId', jedoch fuer pPoint *)

PROCEDURE NewPoint(val: pPoint): pPoint;
(* ... analog 'NewCard', jedoch fuer pPoint *)
```

```
PROCEDURE DelPoint(VAR val: pPoint);
(* ... analog 'DelCard', jedoch fuer pPoint *)

END Simptype.
```

B.26 Implementierungsmodul 'Simptype'

```
IMPLEMENTATION MODULE Simptype;

(*   einfache Datentypen fuer das Modul 'Type'

       ( Erklaerungen im Definitionsmodul )
*)

FROM SYSTEM IMPORT TSIZE;
IMPORT Sys, Type;
FROM Sys IMPORT tPOINTER;

VAR vCardId, vIntId, vRealId, vPointId: Type.Id;
        (* Typidentifikatoren fuer Modul 'Type' *)

PROCEDURE CardId(): Type.Id;
BEGIN
    RETURN vCardId
END CardId;

PROCEDURE NewCard(val: LONGCARD): pCard;
VAR res: pCard;
BEGIN
    res:= Type.NewI(vCardId);
    res^:= val;
    RETURN res
END NewCard;

PROCEDURE DelCard(VAR val: pCard);
BEGIN
    Type.DelI(vCardId, val)
END DelCard;

PROCEDURE IntId(): Type.Id;
BEGIN
    RETURN vIntId
END IntId;

PROCEDURE NewInt(val: LONGINT): pInt;
VAR res: pInt;
BEGIN
    res:= Type.NewI(vIntId);
    res^:= val;
    RETURN res
END NewInt;

PROCEDURE DelInt(VAR val: pInt);
BEGIN
    Type.DelI(vIntId, val)
END DelInt;

PROCEDURE RealId(): Type.Id;
BEGIN
    RETURN vRealId
```

```
END RealId;

PROCEDURE NewReal(val: LONGREAL): pReal;
VAR res: pReal;
BEGIN
    res:= Type.NewI(vRealId);
    res^:= val;
    RETURN res
END NewReal;

PROCEDURE DelReal(VAR val: pReal);
BEGIN
    Type.DelI(vRealId, val)
END DelReal;

PROCEDURE PointId(): Type.Id;
BEGIN
    RETURN vPointId
END PointId;

PROCEDURE NewPoint(val: pPoint): pPoint;
VAR res: pPoint;
BEGIN
    res:= Type.NewI(vPointId);
    res^:= val;
    RETURN res
END NewPoint;

PROCEDURE DelPoint(VAR val: pPoint);
BEGIN
    Type.DelI(vPointId, val)
END DelPoint;

(*****************************************)
(* Operationsprozeduren fuer Modul 'Type': *)

PROCEDURE EquCard(ahilf,bhilf:tPOINTER): BOOLEAN;
VAR
    a, b: pCard;
BEGIN
    a:= ahilf; b:= bhilf;
    RETURN a^ = b^
END EquCard;

PROCEDURE OrdCard(ahilf,bhilf: tPOINTER): BOOLEAN;
VAR
    a, b: pCard;
BEGIN
    a:= ahilf; b:= bhilf;
    RETURN a^ < b^
END OrdCard;

PROCEDURE EquInt(a,b: tPOINTER): BOOLEAN;
VAR A, B: pInt;
BEGIN
    A:= a; B:= b;
    RETURN A^ = B^
END EquInt;

PROCEDURE OrdInt(a,b: tPOINTER): BOOLEAN;
VAR A, B: pInt;
BEGIN
    A:= a; B:= b;
    RETURN A^ < B^
END OrdInt;
```

```
PROCEDURE EquReal(a,b: tPOINTER): BOOLEAN;
VAR A,B: pReal;
BEGIN
    A:= a; B:= b;
    RETURN A^ = B^
END EquReal;

PROCEDURE OrdReal(a,b: tPOINTER): BOOLEAN;
VAR A,B: pReal;
BEGIN
    A:= a; B:= b;
    RETURN A^ < B^
END OrdReal;

PROCEDURE EquPoint(a,b: tPOINTER): BOOLEAN;
VAR A,B: pPoint;
BEGIN
    A:= a; B:= b;
    RETURN A^ = B^
END EquPoint;

BEGIN
    vCardId:= Type.New(TSIZE(LONGCARD));
    Type.SetName(vCardId,"LONGCARD");
    Type.SetEquProc(vCardId,EquCard);
    Type.SetOrdProc(vCardId,OrdCard);

    vIntId:= Type.New(TSIZE(LONGINT));
    Type.SetName(vIntId,"LONGINT");
    Type.SetEquProc(vIntId,EquInt);
    Type.SetOrdProc(vIntId,OrdInt);

    vRealId:= Type.New(TSIZE(LONGREAL));
    Type.SetName(vRealId,"LONGREAL");
    Type.SetEquProc(vRealId,EquReal);
    Type.SetOrdProc(vRealId,OrdReal);

    vPointId:= Type.New(TSIZE(pPoint));
    Type.SetName(vPointId,"pPoint");
    Type.SetEquProc(vPointId,EquPoint)
END Simptype.
```

B.27 Definitionsmodul 'Str'

```
DEFINITION MODULE Str;

(*    Handhabung von Zeichenketten

   Falls nicht anders angegeben wird 0 als erster Index
   innerhalb einer Zeichenkette betrachtet.
   Ein Null-Byte (0C) wird als einer Zeichenkette
   betrachtet.
   Zur Steigerung der Effizienz sind bei allen Prozeduren auch
   die Eingabeparameter als Variablenparameter deklariert.
*)

IMPORT Func;

CONST
```

```
    MaxString=80;

TYPE
    tStr= ARRAY [0..MaxString-1] OF CHAR;

PROCEDURE Empty(VAR str: ARRAY OF CHAR);
(* 'str' wird geloescht (mit 0C gefuellt). *)

PROCEDURE Assign(VAR dst: ARRAY OF CHAR; src: ARRAY OF CHAR);
(* 'dst' wird an 'src' zugewiesen
   (ggf. wird 'dst' mit 0C aufgefuellt).
*)

PROCEDURE Append(VAR dest: ARRAY OF CHAR;
                 suffix: ARRAY OF CHAR);
(* 'suffix' wird an 'dest' angehaengt, soweit in 'dest'
   noch Platz ist.
*)

PROCEDURE Equal(VAR x: ARRAY OF CHAR;
                    y: ARRAY OF CHAR): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls die Zeichenketten
   'x' und 'y' gleich sind, sonst FALSE. Leerstellen sind
   relevant! Angehaengte Null-Bytes werden nicht beachtet.
*)

PROCEDURE Ordered(VAR x: ARRAY OF CHAR;
                      y: ARRAY OF CHAR): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls 'a' nach lexikalischer
   Ordnung kleiner als 'b' ist. Andernfalls ist das
   Funktionsergebnis FALSE. *)

PROCEDURE Length(VAR str: ARRAY OF CHAR): CARDINAL;
(* Das Funktionsergebnis ist die Laenge von 'str'
   (Index des ersten unbenutzten Elements).
   Angehaengte Null-Bytes werden nicht beachtet.
*)

PROCEDURE Insert(VAR substr, str: ARRAY OF CHAR;
                 inx: CARDINAL);
(* 'substr' wird in 'str' eingefuegt, beginnend bei 'str[inx]'.
   'inx' darf keine Position hinter der in 'str' gespeicherten
   Zeichenkette bezeichnen.
*)

PROCEDURE Delete(VAR str: ARRAY OF CHAR; inx,len: CARDINAL);
(* Beginnend bei 'str[inx]' werden 'len' Zeichen aus 'str'
   geloescht.
*)

PROCEDURE In(VAR substr, str: ARRAY OF CHAR;
             VAR pos: CARDINAL): BOOLEAN;
(* Falls 'substr' in 'str' enthalten ist, wird in 'pos' der Index
   des ersten Auftretens zurueckgegeben und der Funktionswert ist
   TRUE. Ansonsten ist der Funktionswert FALSE und 'pos'
   undefiniert.
*)

PROCEDURE NewSub(switch: BOOLEAN);
(* Es wird die Initialisierung der Suche von 'substr' in der Prozedur
   'In' ein- oder ausgeschaltet. Bei ausgeschalteter Initialisierung
   verlaeuft die Suche u. U. deutlich schneller. In diesem Fall muss
   jedoch bei jedem Aufruf von 'In' die gleiche Zeichenkette an 'substr'
   uebergeben werden.
   'switch = TRUE' schaltet die Initialisierung ein und 'switch = FALSE'
```

```
   schaltet sie aus.
*)

PROCEDURE Lower(VAR s: ARRAY OF CHAR);
(* Alle in 's' auftretenden Grossbuchstaben werden durch die
   zugehoerigen Kleinbuchstaben ersetzt. Ein 0-Byte wird als
   Ende der Zeichenkette betrachtet. *)

END Str.
```

B.28 Implementierungsmodul 'Str'

```
IMPLEMENTATION MODULE Str;

(*    Handhabung von Zeichenketten

   (Erklaerungen im Definitionsmodul)
*)

FROM InOut IMPORT WriteLn, WriteString;
IMPORT Func;

PROCEDURE Empty(VAR str: ARRAY OF CHAR);
VAR
    i,h: CARDINAL;
BEGIN
    h:= HIGH(str);
    FOR i:= 0 TO h DO
        str[i]:= 0C;
    END
END Empty;

PROCEDURE Assign(VAR dst: ARRAY OF CHAR; src: ARRAY OF CHAR);
VAR
    i,highd,highs: CARDINAL;
BEGIN
    highd:= HIGH(dst);
    highs:= HIGH(src);
    FOR i:= 0 TO highd DO
        IF i<= highs THEN
            dst[i]:= src[i]
        ELSE
            dst[i]:= 0C;
        END
    END
END Assign;

PROCEDURE Append(VAR dest: ARRAY OF CHAR;
                 suffix: ARRAY OF CHAR);
VAR
    InDest,InSuffix,highd,highs: CARDINAL;
BEGIN
    highd:= HIGH(dest);
    highs:= HIGH(suffix);
    InDest:= Length(dest);
    InSuffix:= 0;
    WHILE (InDest <= highd) AND (InSuffix <= highs) DO
        dest[InDest]:= suffix[InSuffix];
        INC(InDest); INC(InSuffix)
    END
```

```
END Append;

PROCEDURE Equal(VAR x: ARRAY OF CHAR;
                y: ARRAY OF CHAR): BOOLEAN;
VAR
    num, max: CARDINAL;
    equal, end: BOOLEAN;
BEGIN
    num:= 0; max:= Func.MinCard(HIGH(x), HIGH(y));
    REPEAT
        equal:= x[num] = y[num];
        end:= (x[num] = 0C) OR (y[num] = 0C) OR (num = max);
        INC(num)
    UNTIL end OR NOT equal;
    IF equal AND (num > max) THEN
        IF (x[num-1] # 0C) AND (HIGH(x) # HIGH(y)) THEN
            IF HIGH(x) < HIGH(y) THEN
                equal:= y[num] # 0C
            ELSE
                equal:= x[num] # 0C
            END
        END
    END;
    RETURN equal
END Equal;

PROCEDURE Ordered(VAR x: ARRAY OF CHAR;
                      y: ARRAY OF CHAR): BOOLEAN;
VAR
    num, max: CARDINAL;
    equal, end, order: BOOLEAN;
BEGIN
    num:= 0; max:= Func.MinCard(HIGH(x), HIGH(y));
    REPEAT
        equal:= x[num] = y[num];
        end:= (x[num] = 0C) OR (y[num] = 0C) OR (num = max);
        INC(num)
    UNTIL end OR NOT equal;
    IF equal THEN
        (* hier gilt: (x[num-1] = y[num-1]) AND (num > max) *)
        IF x[num-1] = 0C THEN
            order:= TRUE
        ELSE
            order:= HIGH(x) <= HIGH(y)
        END
    ELSE
        order:= x[num-1] < y[num-1]
    END;
    RETURN order
END Ordered;

PROCEDURE Length(VAR str: ARRAY OF CHAR): CARDINAL;
(* Verbesserungsmoeglichkeit:
       Suche nach der Intervallhalbierungsmethode
*)
VAR
    i,high: CARDINAL;
    cont: BOOLEAN;
BEGIN
    i:= 0;
    high:= HIGH(str);
    WHILE (str[i] # 0C) AND (i < high) DO
        INC(i)
    END;
    IF str[i] = 0C THEN
```

```
        RETURN i
    END;
    RETURN i+1
END Length;

PROCEDURE Insert(VAR substr, str: ARRAY OF CHAR;
                 inx: CARDINAL);
VAR
    i, j, SubLength, StrEnd, SubHigh, StrHigh,
    NewLastChar: CARDINAL;
BEGIN
    StrEnd:= Length(str);
    IF inx>StrEnd THEN
        WriteLn;
        WriteString("*** Str.Insert:                      ***");
        WriteLn;
        WriteString("***     angegebener Index groesser  ***");
        WriteLn;
        WriteString("***     als Laenge der Zeichenkette ***");
        WriteLn;
        HALT
    END;

    SubLength:= Length(substr);
    SubHigh:= HIGH(substr); StrHigh:= HIGH(str);

    (* in 'str' Platz fuer 'substr' schaffen: *)
    NewLastChar:= (StrEnd-1) + SubLength;
    i:= NewLastChar; (* i: Ziel der Verschiebung *)
    FOR j:= StrEnd-1-Func.MaxCard(NewLastChar-StrHigh, 0)
    TO inx BY -1 DO
        str[i]:= str[j];
        INC(i)
    END;

    (* 'substr' einfuegen: *)
    j:= 0; (* j: Position in 'substr' *)
    FOR i:= inx TO Func.MaxCard(inx+SubLength-1, StrHigh) DO
        str[i]:= substr[j];
        INC(j)
    END
END Insert;

PROCEDURE Delete(VAR str: ARRAY OF CHAR; inx,len: CARDINAL);
VAR
    i, j, StrEnd, StrHigh, NewLastChar: CARDINAL;
    cont: BOOLEAN;
BEGIN
    StrEnd:= Length(str);
    IF inx>StrEnd THEN
        WriteLn;
        WriteString("*** Str.Delete:                      ***");
        WriteLn;
        WriteString("***     angegebener Index groesser  ***");
        WriteLn;
        WriteString("***     als Laenge der Zeichenkette ***");
        WriteLn;
        HALT
    END;

    StrHigh:= HIGH(str);

    FOR i:= inx TO StrEnd DO
        IF (i<inx+len) AND (i+len<=StrHigh) THEN
            str[i]:= str[i+len]
```

```
        ELSE
            str[i]:= 0C
        END
    END
END Delete;

VAR
    d: ARRAY [0C..255C] OF INTEGER;
        (* 'Vorrueckwerte' fuer jedes Zeichen im Zeichensatz *)
    changed: ARRAY [0..255] OF CHAR;
        (* Liste der Indizes in 'd', die veraendert wurden *)
    top: INTEGER;
        (* erstes unbenutztes Element in 'changed' *)
    newsub: BOOLEAN;
        (* FALSE: es wird angenommen, dass alle 'substr'-Parameter,
                  die an 'In' uebergeben werden, gleich sind und
                  die Suche nicht initialisiert werden muss
                  (-> schneller)
        *)
    SubLen: INTEGER;
        (* Laenge der zu suchenden Zeichenkette *)

PROCEDURE InitSearch(VAR substr: ARRAY OF CHAR);
(* 'substr' wird als zu suchende Zeichenkette betrachtet. Strukturinfor-
   mation fuer die spaetere Suche durch 'In' werden in 'd' gespeichert.
*)
VAR i, InSub: INTEGER;
BEGIN
    SubLen:= Length(substr);
    FOR i:= 0 TO top-1 DO
        d[changed[i]] := -1
    END;
    top:= 0;
    FOR InSub:= 0 TO SubLen-2 DO
        IF d[substr[InSub]] = -1 THEN
            changed[top]:= substr[InSub];
            INC(top)
        END;
        d[substr[InSub]]:= SubLen-InSub-1
    END
END InitSearch;

PROCEDURE In(VAR substr, str: ARRAY OF CHAR;
             VAR pos: CARDINAL): BOOLEAN;
(* Es wird der Boyer-Moore Algorithmus aus
       N. Wirth, Algorithmen und Datenstrukturen mit Modula-2,
       Teubner Verlag, S. 67 ff.
   verwendet.
*)
VAR
    StrLen, InStr, InSub, WorkInStr: INTEGER;
    stop: BOOLEAN;
    inequal: BOOLEAN;
    i: INTEGER;
BEGIN
    StrLen:= Length(str);

    IF newsub THEN
        InitSearch(substr);
        IF (SubLen < 1)  OR (SubLen > StrLen) THEN RETURN FALSE END
    ELSE
        IF SubLen > StrLen THEN RETURN FALSE END
    END;

    InStr:= SubLen-1;
```

```
    REPEAT
        WorkInStr:= InStr; InSub:= SubLen-1;
        REPEAT
            inequal:= substr[InSub] # str[WorkInStr];
            DEC(WorkInStr); DEC(InSub)
        UNTIL (InSub < 0) OR inequal;
        IF d[str[InStr]] = -1 THEN
            InStr:= InStr + SubLen
        ELSE
            InStr:= InStr + d[str[InStr]]
        END
    UNTIL ((InSub < 0) AND NOT inequal) OR (InStr >= StrLen);

    pos:= WorkInStr+1;
    RETURN (InSub < 0) AND NOT inequal
END In;

PROCEDURE NewSub(switch: BOOLEAN);
BEGIN
    newsub:= switch
END NewSub;

PROCEDURE Lower(VAR s: ARRAY OF CHAR);
VAR i: CARDINAL;
    stop: BOOLEAN;
BEGIN
    stop:= FALSE;
    i:= 0;
    REPEAT
        IF ('A' <= s[i]) AND (s[i] <= 'Z') THEN
            s[i]:= CHR(ORD('a') + ORD(s[i]) - ORD('A'))
        END;
        stop:= (s[i] = CHR(0)) OR (i+1 > HIGH(s));
        INC(i)
    UNTIL stop
END Lower;

VAR
    ch: CHAR;

BEGIN
    FOR ch:= 0C TO 255C DO d[ch]:= -1 END;
    top:= 0;
    newsub:= TRUE
END Str.
```

B.29 Definitionsmodul 'Sys'

```
DEFINITION MODULE Sys;

(*      Systemabhaengige nichtmathematische Prozeduren
        (Atari ST, TOS 2.06, Megamax Modula-2 Compiler V4.2)

    In diesem Modul sind alle nichtmathematischen Prozeduren
    gesammelt, deren Implementierung von dem System abhaengt,
    auf dem das Programm uebersetzt wurde.

*)

FROM SYSTEM IMPORT ADDRESS;
```

```
TYPE File;
     tPOINTER= ADDRESS;
         (* 'tPOINTER' ist zuweisungskompatibel zu allen
            Zeigern *)

PROCEDURE OpenRead(VAR f: File; name: ARRAY OF CHAR);
(* Die Datei 'f' wird unter dem in 'name' angegebenen Namen zum
   Lesen geoeffnet. *)

PROCEDURE OpenWrite(VAR f: File; name: ARRAY OF CHAR);
(* Die Datei 'f' wird unter dem in 'name' angegebenen Namen zum
   Schreiben geoeffnet. Ein bereits existierende Datei wird
   geloescht. *)

PROCEDURE Close(VAR f: File);
(* Die Datei 'f' wird geschlossen. *)

PROCEDURE Exist(name: ARRAY OF CHAR): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls eine Datei mit dem in 'name'
   angegebenen Namen existiert. *)

PROCEDURE Delete(name: ARRAY OF CHAR);
(* Die Datei mit dem angegebenen Namen wird geloescht. *)

PROCEDURE EOF(VAR f: File): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls das Ende der Datei 'f'
   erreicht ist. *)

PROCEDURE State(VAR f: File): INTEGER;
(* Das Funktionsergebnis ist ein nichtnegativer Wert, falls die
   letzte Operation auf 'f' erfolgreich war. *)

PROCEDURE WriteLn(VAR f: File);
(* Die Ausgabe der laufenden Zeile in die Datei 'f' wird beendet. *)

PROCEDURE WriteCard(VAR f: File; val: LONGCARD; length: CARDINAL);
(* 'val' wird in die Datei 'f' ausgegeben. Die Ausgabe hat eine
   Laenge von 'length' Zeichen. *)

PROCEDURE WriteReal(VAR f: File; val: LONGREAL;
                    length, dec: CARDINAL);
(* ... analog 'WriteCard'. Es werden 'dec' Nachkommastellen ausge-
   geben (also  length <= dec+1).  *)

PROCEDURE WriteString(VAR f: File; val: ARRAY OF CHAR);
(* ... analog 'WriteCard' *)

PROCEDURE ReadCard(VAR f: File; VAR val: CARDINAL);
(* Eine Zahl von Typ CARDINAL wird aus 'f' gelesen und in 'val'
   zurueckgegeben. *)

PROCEDURE ReadLCard(VAR f: File; VAR val: LONGCARD);
(* ... analog 'ReadCard', jedoch fuer Typ 'LONGCARD' *)

PROCEDURE ReadReal(VAR f: File; VAR val: LONGREAL);
(* Eine Zahl von Typ LONGREAL wird aus 'f' gelesen und in 'val'
   zurueckgegeben. *)

PROCEDURE ReadString(VAR f: File; VAR val: ARRAY OF CHAR);
(* Aus der Datei 'f' wird eine Zeichenkette gelesen und in 'val'
   zurueckgegeben. Fuehrende Leerstellen werden ignoriert.
   Das Lesen wird am Zeilenende oder bei der ersten Leerstelle
   nach der Zeichenkette abgebrochen. *)
```

```
PROCEDURE ReadLine(VAR f: File; VAR val: ARRAY OF CHAR);
(* Die aktuelle Zeile wird bis zum Zeilenende gelesen und in
   'val' zurueckgegeben. Nachfolgende Leseversuche werden
   in der naechsten Zeile fortgesetzt. *)

PROCEDURE ReadLn(VAR f: File);
(* Die aktuelle Zeile wird bis zu ihrem Ende verworfen. Das
   Lesen wird mit der naechsten Zeile fortgesetzt. *)

END Sys.
```

B.30 Implementierungsmodul 'Sys'

```
IMPLEMENTATION MODULE Sys;

(*      Systemabhaengige nichtmathematische Prozeduren
        (Atari ST, TOS 2.06, Megamax Modula-2 Compiler V4.2)

          (Erklaerungen im Definitionsmodul)
*)

IMPORT Files, Text, NumberIO;
FROM Files IMPORT Access, ReplaceMode, Open, Create;

TYPE
     File= Files.File;

PROCEDURE OpenRead(VAR f: File; name: ARRAY OF CHAR);
BEGIN
    Open(f, name, Access(readSeqTxt))
END OpenRead;

PROCEDURE OpenWrite(VAR f: File; name: ARRAY OF CHAR);
BEGIN
    Create(f, name, Access(writeSeqTxt), ReplaceMode(replaceOld))
END OpenWrite;

PROCEDURE Close(VAR f: File);
BEGIN
    Files.Close(f)
END Close;

PROCEDURE Exist(name: ARRAY OF CHAR): BOOLEAN;
VAR
    f: File;
BEGIN
    Open(f, name, Access(readOnly) );
    IF Files.State(f) < 0 THEN
        RETURN FALSE
    END;
    Close(f);
    RETURN TRUE
END Exist;

PROCEDURE Delete(name: ARRAY OF CHAR);
VAR
    f: File;
BEGIN
    IF Exist(name) THEN
        OpenWrite(f, name);
```

```
        Files.Remove(f)
    END
END Delete;

PROCEDURE EOF(VAR f: File): BOOLEAN;
BEGIN
    RETURN Files.EOF(f)
END EOF;

PROCEDURE State(VAR f: File): INTEGER;
BEGIN
    RETURN Files.State(f)
END State;

PROCEDURE WriteLn(VAR f: File);
BEGIN
    Text.WriteLn(f)
END WriteLn;

PROCEDURE WriteCard(VAR f: File; val: LONGCARD; length: CARDINAL);
BEGIN
    NumberIO.WriteCard(f, val, length)
END WriteCard;

PROCEDURE WriteReal(VAR f: File; val: LONGREAL;
                    length,dec: CARDINAL);
BEGIN
    NumberIO.WriteReal(f, val, length, dec)
END WriteReal;

PROCEDURE WriteString(VAR f: File; val: ARRAY OF CHAR);
BEGIN
    Text.WriteString(f, val)
END WriteString;

PROCEDURE ReadCard(VAR f: File; VAR val: CARDINAL);
VAR
    dummy: BOOLEAN;
BEGIN
    NumberIO.ReadCard(f, val, dummy)
END ReadCard;

PROCEDURE ReadLCard(VAR f: File; VAR val: LONGCARD);
VAR
    dummy: BOOLEAN;
BEGIN
    NumberIO.ReadLCard(f, val, dummy)
END ReadLCard;

PROCEDURE ReadReal(VAR f: File; VAR val: LONGREAL);
VAR
    dummy: BOOLEAN;
BEGIN
    NumberIO.ReadLReal(f, val, dummy)
END ReadReal;

PROCEDURE ReadString(VAR f: File; VAR val: ARRAY OF CHAR);
BEGIN
    Text.ReadToken(f, val)
END ReadString;

PROCEDURE ReadLine(VAR f: File; VAR val: ARRAY OF CHAR);
BEGIN
    Text.ReadFromLine(f, val)
END ReadLine;
```

```
PROCEDURE ReadLn(VAR f: File);
VAR
    dummy: ARRAY [1..160] OF CHAR;
BEGIN
    Text.ReadFromLine(f,dummy)
END ReadLn;

END Sys.
```

B.31 Definitionsmodul 'SysMath'

```
DEFINITION MODULE SysMath;

(*          Systemabhaengige mathematische Prozeduren
       (Atari ST, TOS 2.06, Megamax Modula-2 Compiler V4.2)

    In diesem Modul sind alle mathematischen Prozeduren gesammelt,
    um eine Portierung auf andere Rechner/Compiler zu vereinfachen.
*)

(*******************************)
(***** Typkonvertierungen: *****)

(* zu den Funktionen sind nur Erklaerungen angegeben, falls sie
   neben der Typkonvertierung noch weitere Effekte auf den
   jeweiligen Wert besitzen *)

PROCEDURE LCard2Card(x: LONGCARD): CARDINAL;

PROCEDURE LCard2LReal(x: LONGCARD): LONGREAL;

PROCEDURE Card2LCard(x: CARDINAL): LONGCARD;

PROCEDURE Card2LReal(x: CARDINAL): LONGREAL;

PROCEDURE LReal2Card(x: LONGREAL): CARDINAL;
(* Nachkommastellen werden abgeschnitten *)

PROCEDURE LReal2LCard(x: LONGREAL): LONGCARD;
(* Nachkommastellen werden abgeschnitten *)

PROCEDURE LReal2LInt(x: LONGREAL): LONGINT;
(* Nachkommastellen werden abgeschnitten *)

PROCEDURE entier(x: LONGREAL): LONGINT;
(* identisch zu 'LReal2LInt'; zur Erhoehung der Lesbarkeit des
   Quelltextes *)

PROCEDURE LInt2Int(x: LONGINT): INTEGER;

PROCEDURE LInt2LReal(x: LONGINT): LONGREAL;

PROCEDURE real(x: LONGINT): LONGREAL;
(* identisch zu 'LInt2LReal'; zur Erhoehung der Lesbarkeit des
   Quelltextes *)

PROCEDURE Int2LInt(x: INTEGER): LONGINT;

(************************)
```

```
(***** Berechnungen: *****)

PROCEDURE cos(x: LONGREAL): LONGREAL;
(* Cosinus von x (Bogenmass) *)

PROCEDURE sin(x: LONGREAL): LONGREAL;
(* Sinus von x (Bogenmass) *)

PROCEDURE ln(x: LONGREAL): LONGREAL;
(* Logarithmus zur Basis e von x *)

PROCEDURE ld(x: LONGREAL): LONGREAL;
(* Logarithmus zur Basis 2 von x *)

PROCEDURE lg(x: LONGREAL): LONGREAL;
(* Logarithmus zur Basis 10 von x *)

PROCEDURE power(b, x: LONGREAL): LONGREAL;
(* b^x *)

PROCEDURE sqrt(x: LONGREAL): LONGREAL;
(* Wurzel von x *)

END SysMath.
```

B.32 Implementierungsmodul 'SysMath'

```
IMPLEMENTATION MODULE SysMath;

(*          Systemabhaengige mathematische Prozeduren
       (Atari ST, TOS 2.06, Megamax Modula-2 Compiler V4.2)

            (Erklaerungen im Definitionsmodul)
*)

IMPORT MathLib0;

(*****************************)
(***** Typkonvertierungen: *****)

(* zu den Funktionen sind nur Erklaerungen angegeben, falls sie
   neben der Typkonvertierung noch weitere Effekte auf den
   jeweiligen Wert besitzen *)

PROCEDURE LCard2Card(x: LONGCARD): CARDINAL;
BEGIN
    RETURN SHORT(x)
END LCard2Card;

PROCEDURE LCard2LReal(x: LONGCARD): LONGREAL;
BEGIN
    RETURN LInt2LReal(x)
END LCard2LReal;

PROCEDURE Card2LCard(x: CARDINAL): LONGCARD;
BEGIN
    RETURN LONG(x)
END Card2LCard;

PROCEDURE Card2LReal(x: CARDINAL): LONGREAL;
```

```
BEGIN
    RETURN LInt2LReal(LONG(x))
END Card2LReal;

PROCEDURE LReal2Card(x: LONGREAL): CARDINAL;
BEGIN
    RETURN LInt2Int(LReal2LInt(x))
END LReal2Card;

PROCEDURE LReal2LCard(x: LONGREAL): LONGCARD;
BEGIN
    RETURN LReal2LInt(x)
END LReal2LCard;

PROCEDURE LReal2LInt(x: LONGREAL): LONGINT;
BEGIN
    RETURN MathLib0.entier(x)
END LReal2LInt;

PROCEDURE entier(x: LONGREAL): LONGINT;
BEGIN
    RETURN LReal2LInt(x)
END entier;

PROCEDURE LInt2Int(x: LONGINT): INTEGER;
BEGIN
    RETURN SHORT(x)
END LInt2Int;

PROCEDURE LInt2LReal(x: LONGINT): LONGREAL;
BEGIN
    RETURN MathLib0.real(x)
END LInt2LReal;

PROCEDURE real(x: LONGINT): LONGREAL;
BEGIN
    RETURN LInt2LReal(x)
END real;

PROCEDURE Int2LInt(x: INTEGER): LONGINT;
BEGIN
    RETURN LONG(x)
END Int2LInt;

(*************************)
(***** Berechnungen: *****)

PROCEDURE cos(x: LONGREAL): LONGREAL;
BEGIN
    RETURN MathLib0.cos(x)
END cos;

PROCEDURE sin(x: LONGREAL): LONGREAL;
BEGIN
    RETURN MathLib0.sin(x)
END sin;

PROCEDURE ln(x: LONGREAL): LONGREAL;
BEGIN
    RETURN MathLib0.ln(x)
END ln;

PROCEDURE ld(x: LONGREAL): LONGREAL;
(* Logarithmus zur Basis 2 von x *)
BEGIN
```

```
    RETURN MathLib0.ld(x)
END ld;

PROCEDURE lg(x: LONGREAL): LONGREAL;
BEGIN
    RETURN MathLib0.log(x)
END lg;

PROCEDURE power(b, x: LONGREAL): LONGREAL;
BEGIN
    RETURN MathLib0.power(b, x)
END power;

PROCEDURE sqrt(x: LONGREAL): LONGREAL;
BEGIN
    RETURN MathLib0.sqrt(x)
END sqrt;

END SysMath.
```

B.33 Definitionsmodul 'Type'

```
DEFINITION MODULE Type;

(* Verwaltung von Elementen selbst definierter Typen fuer die
   Benutzung in Verbindung mit komplexen Datenstrukturen
   (z. B. Listen, Baeume)

   Anstatt mit den Elementen der verschiedenen Typen wird mit
   deren Adressen gearbeitet. Dieses Modul dient dazu, den
   Umgang mit diesen Adressen zu uebernehmen.

   Fuer jeden Datentyp muessen Prozeduren vereinbart werden,
   die die Behandlung von Elementen dieses Typs ermoeglichen.
   Die erforderlichen Operationen sind:
       1) Anlegen
       2) Loeschen
       3) auf Gleichheit pruefen
       4) auf Erfuellung einer Ordnungsrelation pruefen
       5) Berechnung einer Hash-Funktion
   Die Angabe der Prozeduren fuer die Operationen ist optional.
*)

IMPORT Sys, Str;
FROM Sys IMPORT tPOINTER;

TYPE Id; (* Von diesem Typ sind Identifikatoren fuer
                 Typen, die mit diesem Modul verwaltet
                 werden. *)

     (* Operationsprozeduren zum Behandeln von Elementen
        selbst definierter Typen: *)

     tNewProc= PROCEDURE(tPOINTER);
         (* 'Anlegen':
            Die entsprechende Prozedur wird zum Initialisieren
            von Listenelementen benutzt. Dies ist z. B. besonders
            nuetzlich bei 'Listen von Listen' um die Tochterliste
            gleich mit zu initialisieren. Das Modul 'Type' ruft
            'Allocate' selbst auf! *)
```

```
     tDelProc= PROCEDURE(tPOINTER);
         (* 'Loeschen':
            Die entsprechende Prozedur wird zum Loeschen
            von Listenelementen benutzt. Dies ist z. B. besonders
            nuetzlich bei 'Listen von Listen' um die Tochterliste
            gleich mit zu loeschen. Das Modul 'Type' ruft
            'Deallocate' selbst auf! *)

     tEquProc= PROCEDURE(tPOINTER,tPOINTER): BOOLEAN;
         (* 'Pruefen auf Gleichheit' *)

     tOrdProc= PROCEDURE(tPOINTER,tPOINTER): BOOLEAN;
         (* 'Pruefen auf Erfuellung einer Ordnungsrelation':
            Die Prozedur muss TRUE ergeben, falls die
            angegebenen Elemente die Relation erfuellen *)

     tHashProc= PROCEDURE(tPOINTER, LONGCARD): LONGCARD;

PROCEDURE New(size: LONGCARD):Id;
(* Das Funktionsergebnis ist eine noch nicht benutzte
   Typnummer zur Verwendung fuer alle Prozeduren, an die
   Parameter vom Typ 'Id' uebergeben werden muessen.
   'size' gibt die Groesse der Elemente von dem neuen
   Typ an (mit TSIZE festgestellt). *)

PROCEDURE Copy(type: Id): Id;
(* 'Copy' arbeitet analog zu 'New', jedoch mit dem Unterschied,
   dass der neue Typ eine Kopie des angegebenen Typs ist.
   Abgesehen von einem mit 'SetName' festgelegten Namen werden
   alle Einstellungen uebernommen.
*)

PROCEDURE SetName(type: Id; name: ARRAY OF CHAR);
(* Fuer den angegebenen Typ wird die angegebene Zeichenkette
   als Name vereinbart. Vom Namen werden nur die ersten 16
   Zeichen beruecksichtigt. Bei weniger als 16 Zeichen wird
   der Rest mit Leerstellen aufgefuellt. *)

PROCEDURE GetName(type: Id; VAR name: ARRAY OF CHAR);
(* In 'name' wird der mit 'SetName' fuer den Typ vereinbarte
   Name zurueckgegeben. Falls kein Name vereinbart wurde,
   werden Leerstellen zurueckgegeben. Die an 'name'
   zurueckgegebene Zeichenkette wird ggf. abgeschnitten oder
   mit Leerstellen aufgefuellt.
*)

PROCEDURE GetId(name: ARRAY OF CHAR): Id;
(* Das Funktionsergebnis ist der Identifikator des Types,
   fuer den 'name' mit Hilfe von 'SetName' als Name
   vereinbart wurde. Falls der gesuchte Typ nicht bekannt
   ist, wird eine Fehlermeldung ausgegeben. *)

PROCEDURE NoType(): Id;
(* Ergibt eine Konstante zur Benutztung in Verbindung mit
   'Equal'. Diese Konstante wird ggf. von Prozeduren
   zurueckgegeben. Mit Hilfe von 'NoType' und 'Equal'
   kann so auf Nichtvorhandensein eines Typs geprueft
   werden. *)

PROCEDURE Equal(t1,t2: Id): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls 't1' und 't2' denselben
   Typ bezeichnen. *)

PROCEDURE SetNewProc(type: Id; NewProc: tNewProc);
```

```
PROCEDURE SetDelProc(type: Id; DelProc: tDelProc);
PROCEDURE SetEquProc(type: Id; EquProc: tEquProc);
PROCEDURE SetOrdProc(type: Id; OrdProc: tOrdProc);
PROCEDURE SetHashProc(type: Id; HashProc: tHashProc);
(* Fuer den angegebenen Typ wird die entsprechende Operations-
   prozedur aufgerufen. *)

PROCEDURE OrdProcDefined(type: Id): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls fuer den angegebenen
   Typ mit 'SetOrdProc' eine Prozedur vereinbart wurde. *)

PROCEDURE NewI(type: Id): tPOINTER; (*NEW Item*)
(* Der Funktionswert ist ein Zeiger auf ein initialisiertes
   Element des angegebenen Typs. Nach 'Allocate' wird 'NewProc'
   fuer dieses Element aufgerufen. *)

PROCEDURE DelI(type: Id; VAR item: tPOINTER); (* DELete Item*)
(* Das Element 'item' des Typs 'type' wird durch Aufruf von
   'DelProc' und 'Deallocate' geloescht. Es duerfen keine
   Referenzen auf das Element mehr vorhanden sein.
   Nach dem Aufruf von 'DelI' gilt 'item = NIL' .
*)

PROCEDURE EquI(type: Id; a,b: tPOINTER): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls die angegebenen
   Elemente des angegebenen Typs gleich sind.
   (benutzt die mit 'SetEquProc' definierte Prozedur)
*)

PROCEDURE OrdI(type: Id; a,b: tPOINTER): BOOLEAN;
(* Das Funktionsergebnis ist TRUE, falls die angegebenen
   Elemente des angegebenen Typs die Ordnungsrelation
   erfuellen.
   (benutzt die mit 'SetOrdProc' definierte Prozedur)
*)

PROCEDURE HashI(type: Id; a: tPOINTER; max: LONGCARD): LONGCARD;
(* Zurueckgegeben wird das Ergebnis der Hash-Funktion fuer das
   angegebene Element des angegebenen Typs. In 'max' muss die
   obere Grenze des zulaessigen Funktionswerts angegeben werden.
   Fuer den Funktionswert 'f' muss gelten ' 0 <= f < max'.
*)

END Type.
```

B.34 Implementierungsmodul 'Type'

```
IMPLEMENTATION MODULE Type;

(* Verwaltung von Elemente selbst definierte Datentypen

   ( Erklaerungen im Definitionsmodul )
*)
(* Verbesserungsmoeglichkeit:
    - Verwaltung von Initialisierungswerten fuer 'NewProcs'
      ( - Export von zusaetzlichen Prozeduren zur Verwaltung der
          Werte
        - Implementierung durch Stapel fuer Initialisierungs-
          werte
        - Unterscheidung zwischen Standardinitialisierungswerten
```

```
          und variablen Initialisierungswerten
      )
*)

FROM SYSTEM IMPORT TSIZE, ADR;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM InOut IMPORT WriteString, WriteLn;
IMPORT Sys, Str;
FROM Sys IMPORT tPOINTER;

CONST MaxTypes= 30;
          (* maximale Anzahl von Typen, die verwaltet werden
             koennen *)
      MaxName= 32;
          (* maximale Laenge eines Namens fuer eine Typ *)

TYPE tProcs       = RECORD
                        (* Operationsprozeduren fuer Listen-
                           element-Typen *)
                        NewProc: tNewProc;
                        DelProc: tDelProc;
                        EquProc: tEquProc;
                        OrdProc: tOrdProc;
                        orddef : BOOLEAN; (* ORDer DEFined *)
                                 (* TRUE: es wurde eine Prozedur
                                    fuer 'OrdProc' angegeben *)
                        HashProc: tHashProc
                    END;
     tPoiTypeRec = POINTER TO tTypeRec;
     Id= tPoiTypeRec;
     tTypeRec    = RECORD
                       InUse  : BOOLEAN;
                           (* TRUE: dieser Datensatz wird bereits
                              fuer die Beschreibung eines Typs
                              verwendet *)
                       name : ARRAY [1..MaxName] OF CHAR;
                       size : LONGINT;
                           (* Groesse eines Elementes des Typs *)
                       procs: tProcs
                   END;
     tTypeArray  = ARRAY [1..MaxTypes] OF tTypeRec;
VAR
    TypeArray: tTypeArray;

PROCEDURE DefaultNewProc(adr: tPOINTER);
BEGIN
    (* tue nichts *)
END DefaultNewProc;

PROCEDURE DefaultDelProc(adr: tPOINTER);
BEGIN
    (* tue nichts *)
END DefaultDelProc;

PROCEDURE DefaultEquProc(adr1,adr2: tPOINTER): BOOLEAN;
BEGIN
    WriteLn;
    WriteString("*** Type.DefaultEquProc:"); WriteLn;
    WriteString("*** keine Vergleichsprozedur vereinbart");
    WriteLn;
    HALT;
    RETURN FALSE;
END DefaultEquProc;

PROCEDURE DefaultOrdProc(adr1,adr2: tPOINTER): BOOLEAN;
```

```
BEGIN
    WriteLn;
    WriteString("*** Type.DefaultOrdProc:"); WriteLn;
    WriteString("*** keine Ordnungsprozedur vereinbart");
    WriteLn;
    HALT;
    RETURN TRUE;
END DefaultOrdProc;

PROCEDURE DefaultHashProc(adr1: tPOINTER; size: LONGCARD): LONGCARD;
BEGIN
    WriteLn;
    WriteString("*** Type.DefaultHashProc:"); WriteLn;
    WriteString("*** keine Hash-Prozedur vereinbart");
    WriteLn;
    HALT;
    RETURN 0;
END DefaultHashProc;

PROCEDURE New(size: LONGCARD):Id;
VAR i,j: CARDINAL;
BEGIN
    i:= 1;
    LOOP
        IF NOT TypeArray[i].InUse THEN EXIT END;
        INC(i);
        IF i>MaxTypes THEN EXIT END
    END;
    IF i>MaxTypes THEN
        WriteLn;
        WriteString("*** Type.New:                          ***");
        WriteLn;
        WriteString("***     zu viele Typen;                ***");
        WriteLn;
        WriteString("***     Konstante 'MaxTypes' erhoehen ***");
        WriteLn;
        HALT
    END;
    FOR j:= 1 TO MaxName DO
        TypeArray[i].name[j]:= ' '
    END;
    TypeArray[i].InUse:= TRUE;
    TypeArray[i].size:= size;
    TypeArray[i].procs.NewProc:= DefaultNewProc;
    TypeArray[i].procs.DelProc:= DefaultDelProc;
    TypeArray[i].procs.EquProc:= DefaultEquProc;
    TypeArray[i].procs.OrdProc:= DefaultOrdProc;
    TypeArray[i].procs.orddef:= FALSE;
    TypeArray[i].procs.HashProc:= DefaultHashProc;
    RETURN ADR(TypeArray[i])
END New;

PROCEDURE Copy(type: Id): Id;
VAR NewType: Id;
BEGIN
    NewType:= New(type^.size);
    NewType^.procs.NewProc:= type^.procs.NewProc;
    NewType^.procs.DelProc:= type^.procs.DelProc;
    NewType^.procs.EquProc:= type^.procs.EquProc;
    NewType^.procs.OrdProc:= type^.procs.OrdProc;
    NewType^.procs.orddef:= type^.procs.orddef;
    NewType^.procs.HashProc:= type^.procs.HashProc;
    RETURN NewType
END Copy;
```

```
PROCEDURE SetName(type: Id; name: ARRAY OF CHAR);
BEGIN
    Str.Assign(type^.name, name)
END SetName;

PROCEDURE GetName(type: Id; VAR name: ARRAY OF CHAR);
BEGIN
    Str.Assign(name, type^.name)
END GetName;

PROCEDURE GetId(name: ARRAY OF CHAR): Id;
VAR i: CARDINAL;
    res: Id;
BEGIN
    i:= 1; res:= Id(NIL);
    LOOP
        IF TypeArray[i].InUse THEN
            IF Str.Equal(name,TypeArray[i].name) THEN
                res:= ADR(TypeArray[i]);
                EXIT
            END
        END;
        INC(i);
        IF i > MaxTypes THEN
            WriteString("*** Type.GetId:"); WriteLn;
            WriteString("*** Typ nicht gefunden"); WriteLn;
            HALT
        END
    END;
    RETURN res
END GetId;

PROCEDURE NoType(): Id;
BEGIN
    RETURN Id(NIL)
END NoType;

PROCEDURE Equal(t1,t2: Id): BOOLEAN;
BEGIN
    RETURN t1=t2
END Equal;

PROCEDURE SetNewProc(type: Id; NewProc: tNewProc);
BEGIN
    type^.procs.NewProc:= NewProc
END SetNewProc;

PROCEDURE SetDelProc(type: Id; DelProc: tDelProc);
BEGIN
    type^.procs.DelProc:= DelProc
END SetDelProc;

PROCEDURE SetEquProc(type: Id; EquProc: tEquProc);
BEGIN
    type^.procs.EquProc:= EquProc
END SetEquProc;

PROCEDURE SetOrdProc(type: Id; OrdProc: tOrdProc);
BEGIN
    type^.procs.OrdProc:= OrdProc;
    type^.procs.orddef:= TRUE
END SetOrdProc;

PROCEDURE SetHashProc(type: Id; HashProc: tHashProc);
BEGIN
```

```
    type^.procs.HashProc:= HashProc;
    type^.procs.orddef:= TRUE
END SetHashProc;

PROCEDURE OrdProcDefined(type: Id): BOOLEAN;
BEGIN
    RETURN type^.procs.orddef
END OrdProcDefined;

PROCEDURE NewI(type: Id): tPOINTER; (* NEW Item *)
VAR NewMem: tPOINTER;
BEGIN
    ALLOCATE(NewMem, type^.size);
    type^.procs.NewProc(NewMem);
    RETURN NewMem
END NewI;
       (* DELete Item *)
PROCEDURE DelI(type: Id; VAR item: tPOINTER);
BEGIN
    IF (type # Id(NIL)) AND (item # NIL) THEN
        type^.procs.DelProc(item);
        DEALLOCATE(item, type^.size);
        item:= NIL
    END
END DelI;

PROCEDURE EquI(type: Id; a,b: tPOINTER): BOOLEAN; (* EQUal Items *)
BEGIN
    RETURN type^.procs.EquProc(a,b)
END EquI;

PROCEDURE OrdI(type: Id; a,b: tPOINTER): BOOLEAN; (* ORDered Items *)
BEGIN
    RETURN type^.procs.OrdProc(a,b)
END OrdI;

PROCEDURE HashI(type: Id; a: tPOINTER; size: LONGCARD): LONGCARD;
BEGIN
    RETURN type^.procs.HashProc(a, size)
END HashI;

VAR j: CARDINAL;

BEGIN
    FOR j:= 1 TO MaxTypes DO
        TypeArray[j].InUse:= FALSE
    END
END Type.
```

Anhang C

Testprogramme

In diesem Kapitel sind alle Programmodule gesammelt, die zum Test von Teilen der Gesamtimplementierung erforderlich sind.

C.1 Programmodul 'listtest'

```
MODULE listtest;

(*
     Test der Module 'List' und 'Cali'
*)

IMPORT Debug;
FROM InOut IMPORT WriteCard, WriteString,
                  ReadCard, ReadString, WriteLn;
IMPORT Sys, Type, Simptype, List, Cali;
FROM Sys IMPORT tPOINTER;
FROM Simptype IMPORT pCard;

VAR
    eingabe: CARDINAL;
    wert: CARDINAL;
    l: Cali.tCali;

    pos: List.tPos;

PROCEDURE PrintItem(item: tPOINTER);
VAR
    p: pCard;
BEGIN
    p:= item;
    WriteCard(p^,5);
    IF List.GetPos(l) = pos THEN
        WriteString("<-- ")
    END
END PrintItem;

PROCEDURE PrintList(l: Cali.tCali);
(* Gibt die Elemente von 'l' auf der Standardausgabe aus. *)
BEGIN
    pos:= List.GetPos(l);
    List.Scan(l,PrintItem);
    List.SetPos(l,pos)
```

```
END PrintList;

PROCEDURE WriteBool(v: BOOLEAN);
(* Gibt 'v' auf der Standardausgabe aus. *)
BEGIN
    IF v THEN
        WriteString("TRUE")
    ELSE
        WriteString("FALSE")
    END
END WriteBool;

BEGIN
    WriteLn;
    WriteString(">>> Test der Module 'List' und 'Cali' <<<");
    WriteLn;
    Cali.Use(l);
    REPEAT
        WriteString(" 0 - Ende"); WriteLn;
        WriteString(" 1 - Empty"); WriteLn;
        WriteString(" 2 - Next"); WriteLn;
        WriteString(" 3 - Prev"); WriteLn;
        WriteString(" 4 - InsertBefore"); WriteLn;
        WriteString(" 5 - InsertBehind"); WriteLn;
        WriteString(" 6 - DelCur"); WriteLn;
        WriteString(" 7 - AtFirst"); WriteLn;
        WriteString(" 8 - AtLast"); WriteLn;
        WriteString(" 9 - Count"); WriteLn;
        WriteString(" Liste: ");
        PrintList(l); WriteLn;
        WriteString("Auswahl? "); ReadCard(eingabe); WriteLn;
        CASE eingabe OF
            0: (* tue nichts *)
          | 1: List.Empty(l)
          | 2: List.Next(l)
          | 3: List.Prev(l)
          | 4: WriteString(" Wert? "); ReadCard(wert); WriteLn;
               Cali.InsertBefore(l,wert)
          | 5: WriteString(" Wert? "); ReadCard(wert); WriteLn;
               Cali.InsertBehind(l,wert)
          | 6: List.DelCur(l)
          | 7: WriteBool(List.AtFirst(l)); WriteLn
          | 8: WriteBool(List.AtLast(l)); WriteLn
          | 9: WriteString(" Laenge der Liste: ");
               WriteCard(List.Count(l),4); WriteLn
        ELSE
            WriteString(" Eingabefehler"); WriteLn
        END;
        List.CheckStructure(l)
    UNTIL eingabe=0;
END listtest.
```

C.2 Programmodul 'pramtest'

```
MODULE pramtest;

(*
     Test des Moduls 'Pram'
*)

IMPORT Debug;
```

```
FROM InOut IMPORT WriteCard, ReadLCard, ReadCard,
                  WriteString, ReadString, WriteLn;
FROM Sys IMPORT tPOINTER;
IMPORT Pram;

VAR
    eingabe: CARDINAL;
    wert: LONGCARD;

BEGIN
    WriteLn;
    WriteString(">>> Test des Moduls 'Pram' <<<");
    WriteLn;
    REPEAT
        WriteString(" 0 - Ende"); WriteLn;
        WriteString(" 1 - Pram.Start"); WriteLn;
        WriteString(" 2 - Pram.Ende"); WriteLn;
        WriteString(" 3 - Prozessoren eingeben"); WriteLn;
        WriteString(" 4 - Schritte eingeben"); WriteLn;
        WriteString(" 5 - ParallelStart"); WriteLn;
        WriteString(" 6 - NaechsterBlock"); WriteLn;
        WriteString(" 7 - ParallelEnde"); WriteLn;
        WriteString(" GezaehlteSchritte() = ");
        WriteCard(Pram.GezaehlteSchritte(), 4);
        WriteString(";   GezaehlteProzessoren() = ");
        WriteCard(Pram.GezaehlteProzessoren(), 4);
        WriteLn;
        WriteString("Auswahl? "); ReadCard(eingabe); WriteLn;
        CASE eingabe OF
            0: (* tue nichts *)
          | 1: Pram.Start
          | 2: Pram.Ende
          | 3: WriteString("   Prozessoren? ");
               ReadLCard(wert); WriteLn;
               Pram.Prozessoren(wert);
          | 4: WriteString("   Schritte? ");
               ReadLCard(wert); WriteLn;
               Pram.Schritte(wert);
          | 5: Pram.ParallelStart("pramtest")
          | 6: Pram.NaechsterBlock("pramtest")
          | 7: Pram.ParallelEnde("pramtest")
        ELSE
            WriteString(" Eingabefehler"); WriteLn
        END
    UNTIL eingabe=0;
END pramtest.
```

C.3 Programmodul 'rndtest'

```
MODULE rndtest;

(*   Test des Moduls 'Rnd'  *)

FROM InOut IMPORT WriteString, WriteLn, ReadString, ReadCard,
                  WriteCard;
IMPORT Rnd;

VAR
    w: ARRAY [1..6] OF LONGCARD;
    Wurfzahl,i : CARDINAL;
```

```
    input, input2: CARDINAL;
    gen: Rnd.tGen;

BEGIN
    WriteString(">>> Test des Moduls 'rnd' <<<"); WriteLn;
    Rnd.Use(gen);
    WriteString(" Anfangszahl? "); ReadCard(input);
    WriteLn;
    Rnd.Start(gen, input);
    REPEAT
        WriteString(" 0 - Ende "); WriteLn;
        WriteString(" 1 - neue Anfangszahl"); WriteLn;
        WriteString(" 2 - neuer Generator"); WriteLn;
        WriteString(" 3 - Wuerfeltest"); WriteLn;
        WriteString(" Auswahl? "); ReadCard(input);
        CASE input OF
            0 : (* tue nichts *)
          | 1 : WriteString(" Anfangszahl? "); ReadCard(input2);
                WriteLn;
                Rnd.Start(gen, input2)
          | 2 : Rnd.DontUse(gen);
                Rnd.Use(gen)
          | 3 : WriteString(" Anzahl der Wuerfe? ");
                ReadCard(Wurfzahl);
                Rnd.Range(gen,1.0,6.0);
                FOR i:= 1 TO 6 DO
                    w[i]:= 0;
                END;
                FOR i:= 1 TO Wurfzahl DO
                    INC( w[ SHORT(Rnd.Int(gen)) ] )
                END;
                FOR i:= 1 TO 6 DO
                    WriteCard(i,2); WriteString(": ");
                    WriteCard(w[i],0);
                    WriteLn
                END
        ELSE
            WriteString(" Eingabefehler")
        END
    UNTIL input = 0
END rndtest.
```

C.4 Programmodul 'strtest'

```
MODULE strtest;

(*   Test des Moduls 'str'  *)

FROM InOut IMPORT WriteString, WriteLn, ReadString, ReadCard,
                  WriteCard;
IMPORT Str, Debug;

VAR
    s1, s2: ARRAY [1..40] OF CHAR;
    input: CARDINAL;
    pos: CARDINAL;
BEGIN
    Str.Empty(s1); Str.Empty(s2);
    WriteString(">>> Test des Moduls 'str' <<<"); WriteLn;
    REPEAT
```

```
        WriteString(" 0 - Ende "); WriteLn;
        WriteString(" 1 - string1  eingeben"); WriteLn;
        WriteString(" 2 - string2  eingeben"); WriteLn;
        WriteString(" 3 - Str.Equal(string1, string2) "); WriteLn;
        WriteString(" 4 - Str.Ordered(string1, string2)"); WriteLn;
        WriteString(" 5 - Str.Append(string1, string2)"); WriteLn;
        WriteString(" 6 - Str.In(string2, string1, pos)"); WriteLn;
        WriteString(" string1 = "); WriteString(s1); WriteLn;
        WriteString(" string2 = "); WriteString(s2); WriteLn;
        WriteString(" Auswahl? "); ReadCard(input);
        CASE input OF
            0 : (* tue nichts *)
          | 1 : WriteString(" string1? ");
                ReadString(s1);
          | 2 : WriteString(" string2? ");
                ReadString(s2);
          | 3 : IF Str.Equal(s1, s2) THEN
                    WriteString(" TRUE")
                ELSE
                    WriteString(" FALSE")
                END;
                WriteLn
          | 4 : IF Str.Ordered(s1, s2) THEN
                    WriteString(" TRUE")
                ELSE
                    WriteString(" FALSE")
                END;
                WriteLn
          | 5 : Str.Append(s1, s2)
          | 6 : IF Str.In(s2, s1, pos) THEN
                    WriteString(" gefunden:  pos = ");
                    WriteCard(pos,5)
                ELSE
                    WriteString(" NICHT gefunden")
                END;
                WriteLn
        ELSE
            WriteString(" Eingabefehler")
        END
    UNTIL input = 0
END strtest.
```

C.5 Programmodul 'typetest'

```
MODULE typetest;

(*
     Test des Moduls 'Type'
*)

IMPORT Debug;
FROM InOut IMPORT WriteCard, WriteString,
                  ReadCard, ReadLCard, ReadString, WriteLn;
FROM Sys IMPORT tPOINTER;
IMPORT Type;

TYPE tTestRec = RECORD
                   id: Type.Id;
                   size: LONGCARD;
                   val: ARRAY [1..4] OF tPOINTER
```

```
            END;
     tTestArray = ARRAY [1..4] OF tTestRec;

VAR
    eingabe: CARDINAL;
    wert: CARDINAL;
    TestArray: tTestArray;
    i: CARDINAL;
    cur: CARDINAL;
    name: ARRAY [1..40] OF CHAR;
    FoundId: Type.Id;
    dummy: BOOLEAN;

PROCEDURE NewProc1(h: tPOINTER);
BEGIN
    WriteString("Hier ist NewProc1 ."); WriteLn
END NewProc1;

PROCEDURE DelProc1(h: tPOINTER);
BEGIN
    WriteString("Hier ist DelProc1 ."); WriteLn
END DelProc1;

PROCEDURE EquProc1(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc1 ."); WriteLn;
    RETURN TRUE
END EquProc1;

PROCEDURE OrdProc1(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc1 ."); WriteLn;
    RETURN TRUE
END OrdProc1;

PROCEDURE NewProc2(h: tPOINTER);
BEGIN
    WriteString("Hier ist NewProc2 ."); WriteLn
END NewProc2;

PROCEDURE DelProc2(h: tPOINTER);
BEGIN
    WriteString("Hier ist DelProc2 ."); WriteLn
END DelProc2;

PROCEDURE EquProc2(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc2 ."); WriteLn;
    RETURN TRUE
END EquProc2;

PROCEDURE OrdProc2(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc2 ."); WriteLn;
    RETURN TRUE
END OrdProc2;

PROCEDURE NewProc3(h: tPOINTER);
BEGIN
    WriteString("Hier ist NewProc3 ."); WriteLn
END NewProc3;

PROCEDURE DelProc3(h: tPOINTER);
BEGIN
    WriteString("Hier ist DelProc3 ."); WriteLn
```

```
END DelProc3;

PROCEDURE EquProc3(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc3 ."); WriteLn;
    RETURN TRUE
END EquProc3;

PROCEDURE OrdProc3(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc3 ."); WriteLn;
    RETURN TRUE
END OrdProc3;

PROCEDURE NewProc4(h: tPOINTER);
BEGIN
    WriteString("Hier ist NewProc4 ."); WriteLn
END NewProc4;

PROCEDURE DelProc4(h: tPOINTER);
BEGIN
    WriteString("Hier ist DelProc4 ."); WriteLn
END DelProc4;

PROCEDURE EquProc4(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc4 ."); WriteLn;
    RETURN TRUE
END EquProc4;

PROCEDURE OrdProc4(h1, h2: tPOINTER): BOOLEAN;
BEGIN
    WriteString("Hier ist EquProc4 ."); WriteLn;
    RETURN TRUE
END OrdProc4;

BEGIN
    cur:= 4;
    WriteLn;
    WriteString(">>> Test des Moduls 'Type' <<<");
    WriteLn;
    REPEAT
        WriteString(" 0 - Ende"); WriteLn;
        WriteString(" 1 - Typ waehlen"); WriteLn;
        WriteString(" 2 - New"); WriteLn;
        WriteString(" 3 - SetName"); WriteLn;
        WriteString(" 4 - GetId"); WriteLn;
        WriteString(" 5 - NewI"); WriteLn;
        WriteString(" 6 - DelI"); WriteLn;
        WriteString(" 7 - Proc's"); WriteLn;
        WriteString(" aktueller Typ: ");
        WriteCard(cur,5); WriteLn;
        WriteString("Auswahl? "); ReadCard(eingabe); WriteLn;
        WITH TestArray[cur] DO
            CASE eingabe OF
                0: (* tue nichts *)
              | 1: WriteString(" Typ [1..4]? ");
                   ReadCard(cur); WriteLn;
              | 2: WriteString(" Groesse? ");
                   ReadLCard(size);
                   id:= Type.New(size);
                   CASE cur OF
                       1:
                          Type.SetNewProc(id, NewProc1);
                          Type.SetDelProc(id, DelProc1);
```

```
                        Type.SetEquProc(id, EquProc1);
                        Type.SetOrdProc(id, OrdProc1)
                    | 2:
                        Type.SetNewProc(id, NewProc2);
                        Type.SetDelProc(id, DelProc2);
                        Type.SetEquProc(id, EquProc2);
                        Type.SetOrdProc(id, OrdProc2)
                    | 3:
                        Type.SetNewProc(id, NewProc3);
                        Type.SetDelProc(id, DelProc3);
                        Type.SetEquProc(id, EquProc3);
                        Type.SetOrdProc(id, OrdProc3)
                    | 4:
                        Type.SetNewProc(id, NewProc4);
                        Type.SetDelProc(id, DelProc4);
                        Type.SetEquProc(id, EquProc4);
                        Type.SetOrdProc(id, OrdProc4)
                END;
            | 3: WriteString(" Name? ");
                ReadString(name);
                Type.SetName(id, name);
            | 4: WriteString(" Name? ");
                ReadString(name);
                WriteString("Id: ");
                WriteCard(LONGCARD(Type.GetId(name)), 5);
                WriteLn;
            | 5: WriteString(" Nummer [1..4]? ");
                ReadCard(wert);
                val[wert]:= Type.NewI(id)
            | 6: WriteString(" Nummer [1..4]? ");
                ReadCard(wert);
                Type.DelI(id, val[wert])
            | 7: WriteString(" EquProc: "); WriteLn;
                dummy:= Type.EquI(id, val[1], val[2]);
                WriteString(" OrdProc: "); WriteLn;
                dummy:= Type.OrdI(id, val[1], val[2])
        ELSE
            WriteString(" Eingabefehler"); WriteLn
        END;
    END;
  UNTIL eingabe=0;
END typetest.
```