Technische Informatik

Rothe · Steinbach

BASIC-Programmbausteine

VEB Verlag Technik Berlin

Rothe · Steinbach

BASIC-Programmbausteine

Technische Informatik

Herausgegeben von
Dr.-Ing. Ludwig Claßen
Dr.-Ing. Dieter Nedo
Dr. rer. nat. Gerhard Paulin

BASIC-Programmbausteine

Lineare Gleichungssysteme
Nichtlineare Gleichungen
Differentialgleichungen

Dr. sc. techn. Hendrik Rothe
Dr.-Ing. Manfred Steinbach

VEB VERLAG TECHNIK BERLIN

Trotz aller Sorgfalt der Autoren und des Verlages bei der Erarbeitung und Zusammenstellung der Programme sind Fehler nicht völlig auszuschließen. Bei der Nutzung sollte sehr umsichtig und gewissenhaft vorgegangen und bei jeder Unklarheit fachlicher Rat eingeholt werden. Garantieansprüche und Forderungen jeglicher Art aus der Benutzung der Programme sind ausgeschlossen.

Rothe, Hendrik:
BASIC-Programmbausteine : lineare Gleichungssysteme,
nichtlineare Gleichungen, Differentialgleichungen /
Hendrik Rothe ; Manfred Steinbach. – 1. Aufl. –
Berlin : Verl. Technik, 1989. – 256 S. : 39 Bilder,
9 Taf. – (Technische Informatik)
ISBN 3-341-00621-4
NE: Steinbach, Manfred:

ISBN 3-341-00621-4
ISSN 0863-0860 Technische Informatik

1. Auflage

Lizenz 201 · 370/2/89
Printed in the German Democratic Republic
Gesamtherstellung:
(52) Nationales Druckhaus, VOB National, Berlin
Lektor: Dipl.-Ing. *Uta-Dorothé Hart*
Einbandgestaltung: *Gabriele Schwesinger*
LSV 3054 · VT 3/5927-1
Bestellnummer: 554 003 5
02500

Vorwort

Mikrorechner sind so leistungsfähig geworden, daß nur zur Lösung von wenigen wissenschaftlichen oder technischen Problemen Großrechner benötigt werden. Sie sind heute so verbreitet, daß ein bedeutender Teil des gesellschaftlichen Arbeitsvermögens durch sie beeinflußt wird. Das vorliegende Buch soll einen Beitrag zur Effektivitätssteigerung beim Mikrorechnereinsatz geben.
Ziel des Buches ist, für die Realisierung von Anwenderprogrammen Programmbausteine anzubieten, mit denen Standardaufgaben der Numerik einfach und sicher lösbar sind.
In einem einführenden Abschnitt werden die Voraussetzungen für die Realisierung eines derartigen Bausteinsystems abgehandelt. In den sich anschließenden Abschnitten werden solche Programmbausteine für folgende Gebiete der numerischen Mathematik gegeben:

- Matrizenoperationen und lineare Gleichungssysteme
- nichtlineare Gleichungen und Gleichungssysteme
- gewöhnliche Differentialgleichungen.

Die Programmbausteine sind in der Programmiersprache BASIC geschrieben. BASIC ist die derzeit verbreitetste Programmiersprache. Sie hat für Probleme kleinen bis mittleren Umfangs wesentliche Vorteile. Einem Bausteinkonzept setzt diese Sprache einige Schwierigkeiten entgegen. Deshalb sind in der Literatur bisher nur wenige Versuche dieser Art beschrieben worden. Die beispielsweise von DEC herausgebrachten "Scientific Subroutines" für die Sprache FORTRAN entstanden offenbar aus ähnlichen Überlegungen, sind aber für die einzelnen Gebiete nicht so umfassend und konnten auch in mehreren Fällen durch günstigere Abläufe ersetzt werden. Trotzdem gehen einige unserer Programme (Matrixinversion und Determinanten) auf diese Programmsammlung zurück.
Unser Ansatz zur Überwindung der BASIC-Probleme beruht auf den folgenden, im ersten Abschnitt erläuterten Prinzipien:

- Festlegung eines BASIC-Subset, das unabhängig von BASIC-Dialekten und maschinenspezifischen Besonderheiten auf nahezu jedem Rechner lauffähig ist
- Festlegung von Standardprozeduren der strukturierten BASIC-Programmierung
- Festlegung von Regeln für die Vergabe von Programmzeilennummern und Variablennamen, um damit dem Problem des Fehlens lokaler Variabler zu begegnen.

Wir haben versucht, die Programmbausteine so zu gestalten, daß sie sich mühelos auch in FORTRAN, PASCAL oder ähnlichen Sprachen kodieren lassen. Für alle Programmbausteine sind Textbeispiele angegeben und für einen Teil der Programme Anwendungsbeispiele aus der technischen Mechanik und der Regelungstechnik.
Ein Problem für uns bestand darin, ein vernünftiges Gleichgewicht zwischen kochbuchartiger Beschreibung des Inhalts ohne mathematische Begründung und mathematisch strenger Behandlung zu finden. Beide Extreme schienen uns nicht angebracht. Deshalb haben wir die Abläufe ausreichend erklärt, jedoch auf strenge mathematische Beweisführung verzichtet. Der ingenieurmäßigen Praxis folgend sind Voraussetzungen und Beweise erwähnt, die auch vom Nichtmathematiker nachempfunden werden können. Zum Teil werden numerische Experimente für die Qualitätsbeurteilung von Programmen herangezogen.
In dem Buch sind etwa 150 KByte BASIC-Quelltext vorhanden. Alle Softwaremodule wurden sorgfältig getestet und – um Setzfehler auszuschalten – als Reproduktion wiedergegeben. Die Programme sind in der Schriftart OCR-B gedruckt. Damit besteht die Möglichkeit der

Maschinenlesbarkeit durch einen Klarschriftleser. Darüber hinaus stehen die Programme in den gebräuchlichsten Datenformaten auf Disketten und Magnetbändern zur Verfügung und können von den Autoren unter folgender Adresse abgefordert werden:

Friedrich-Schiller-Universität
Sektion Technologie, Technikum LAURA
Schloßgasse, Jena, 6900

Fehler und Unvollständigkeiten sind in einem derartigen Buch nicht vollständig auszuschließen. Die Autoren sind deshalb für Hinweise dankbar.

Für mannigfaltige Hilfe und Unterstützung bei der Schaffung der rechentechnischen Voraussetzungen danken wir Herrn *Reinhard Lorenz*, Herrn *Jochen Rose*, Frau *Helene Schüler* und Herrn *Rolf Tietsch*. Den Lektoren, Herrn *Hartmut Heinrich* und Frau *Uta-Dorothé Hart*, danken wir für die gute Zusammenarbeit.

Hendrik Rothe *Manfred Steinbach*

Inhaltsverzeichnis

Programmübersicht

1. Grundlagen der Programmierung

Wir wollen es als Standardfall ansehen, daß man eine wissenschaftlich-technische Aufgabe kleinen bis mittleren Umfangs mit rechentechnischen Mitteln zu lösen hat. Ein Rechner steht zur Verfügung; seine Leistung ist vorgegeben. Um die Aufgabe mit Hilfe des Rechners zu lösen, ist das Vorhandensein eines Rechenprogramms unabdingbare Voraussetzung. Das Programm kann als Interface zwischen dem Computer und dem Benutzer mit seiner Aufgabe verstanden werden. Da das Problem neuartig sei, steht ein geeignetes Programm nicht oder höchstens in Teilen zur Verfügung. Welche Forderungen wird man an die bevorstehende Interaktion mit dem Computer stellen; wie muß das Programm beschaffen sein, und welche Eigenschaften sollte der Computer haben?

Korrektheit

Der Rechner führt fehlerlos und vollständig aus, was zur Lösung der Aufgabe nötig ist, nicht mehr und nicht weniger. Es werden alle erforderlichen Eingabe- und Ausgabedaten und nur diese berücksichtigt. Fehlerhafte Abläufe werden zumindest signalisiert. Die Richtigkeit der Aktionen ist testbar. Dokumentationen stimmen mit der Realität überein. Die Abläufe geschehen auch bei geringfügigen Änderungen der vorausgesetzten Daten und Umstände noch fehlerfrei – man spricht in diesem Falle von "Robustheit" der Programme –, z. B. darf das Ergebnis einer zeitraubenden Operation nicht letztlich durch die Eingabe einer falschen Zahl irreversibel beeinträchtigt werden.

Genauigkeit

Die begrenzte Genauigkeit bei der Ausführung numerischer Berechnungen führt nicht zu unzulässigen Ergebnisfehlern. Die Überschreitung von Toleranzgrenzen wird signalisiert oder geeignet berücksichtigt.

Verständlichkeit

Die Abläufe sind verständlich; das zugrunde liegende Programm besitzt dem Inhalt und der Form nach eine einleuchtende Struktur. Beim Betrachten des Programmlistings wird man zunächst mit Funktionsblöcken konfrontiert, die hierarchisch aus kleineren Funktionsblöcken zusammengesetzt sind. Ist diese Forderung erfüllt, so spricht man von strukturierten Programmen. Das Programm ist ausreichend kommentiert; es existiert eine Dokumentation. Das Niveau von Kommentaren und Dokumentation ist innerhalb des gesamten Pakets einigermaßen einheitlich. Programmierstil, Daten- und Programmstrukturen, Terminologie und Spezifikationen sind konsistent. Ähnliche Probleme werden mit ähnlichen Mitteln behandelt. Formate für Ein- und Ausgabe sind gleich bzw. ohne Willkür dem Problem angepaßt. Es sind keine schwer nachvollziehbaren und unkommentierten Programmierkniffe angewendet.

Flexibilität

Geringfügig veränderte Aufgabenstellungen können durch geringfügige Veränderungen am Programm behandelt werden. Das Programm kann um zusätzliche Funktionen erweitert werden. Möglichst viele Funktionsblöcke sind von allgemeinem Wert und bei der Lösung ähnlicher Probleme wiederverwendbar.

Portabilität

Das Programm ist mit möglichst wenigen Änderungen auch auf anderen Rechnern und mit anderer Peripherie lauffähig. Auf spezielle Hardware oder Systemsoftware zugeschnittene Anweisungen sind in wenigen Programmteilen konzentriert und nicht im gesamten Programm verstreut.

Anwenderfreundlichkeit

Ein- und Ausgabe sind ausreichend erläutert und verständlich. Geringfügiges Fehlverhalten des Benutzers wird toleriert oder ist korrigierbar. Unvollständigkeiten bei der Bereitstellung der Peripherie führen nicht zum Programmabsturz, sondern werden mit entsprechender Meldung moniert. Die bei der Benutzung geforderten Aktivitäten beschränken sich auf das Nötige. Bei der Bedienung des Rechners wird man über notwendige Aktivitäten im erforderlichen Umfange informiert, ohne daß eine ermüdende und informationsarme Textflut ausgegeben wird. Notwendige Eingaben werden mit einem Stichwort versehen und nicht nur mit Fragezeichen angefordert. Wo es sinnvoll ist, wird der Bedienende hin und wieder über den Stand der Programmabarbeitung informiert. Zeit und Arbeitskraft des Benutzers werden nicht vergeudet; bei den hier zu betrachtenden Rechnern ist die Arbeitszeit in jedem Falle wertvoller als die Rechenzeit. Lang laufende Programme können ohne Zutun des Benutzers ablaufen. Fehlermeldungen werden in sachlicher Form gegeben. Voraussehbare Frustrationen werden vermieden.

Effizienz

Alle Computerressourcen, wie Rechenzeit, Speicher, Leistungen des Betriebssystems, Peripheriegeräte und Verbrauchsmaterial, werden möglichst sparsam genutzt (Recheneffizienz). Das Programm ist für den Benutzer bzw. den Bediener effizient, indem Belastungen bei der Vorbereitung, Durchführung und Nachbereitung gering bleiben (Nutzereffizienz). Der Aufwand für die Ausarbeitung des Programms soll ebenfalls so gering wie möglich sein (Programmiereffizienz). Hierbei sollte die Frage der Recheneffizienz nicht stärker als unbedingt notwendig bewertet werden; denn hohe Recheneffizienz steht fast allen anderen Qualitätsparametern entgegen.
Voraussetzung für optimal ausgewogene Qualitätsparameter ist die rechtzeitige Abwägung der voraussichtlichen Dauer und Häufigkeit entsprechender Rechenaufgaben. Man überlege zu allem Anfang, ob überhaupt die Aufgabe gelöst werden muß. Die geringste Effektivität erreicht man in dem nicht seltenen Fall, daß sich nach Abschluß der Arbeiten deren Unnötigkeit herausstellt.
Die Realisierung der soeben genannten Anforderungen an die Interaktion mit dem Computer setzt seitens des Programms folgendes voraus:

- Verwendung einer geeigneten Programmiersprache
- Berücksichtigung der begrenzten Rechengenauigkeit
- Anwendung der strukturierten Programmierung
- angemessene Dokumentation und Kommentierung
- ausreichende Programmtestung und -testmöglichkeit
- rechenzeitoptimale Programmierung
- speicherplatzoptimale Programmierung.

Diese Gesichtspunkte werden nachfolgend eingehender beleuchtet.

1.1. Programmieren in BASIC

BASIC ist ein Akronym für "Beginner's All Purpose Symbolic Instruction Code". Die Sprache ist Anfang der sechziger Jahre in den USA im Dartmouth College von *John G. Kemeny* und *Thomas E. Kurtz* entwickelt worden und hat sich seitdem ungewöhnlich stark verbreitet. Der Käufer dieses Buches hat bezüglich der Anwendung dieser Sprache seine Wahl bereits getroffen; wir wollen dennoch auf ihre Vor- und Nachteile hinweisen.
Als Vorteile sind anzusehen:

- Die Sprache gestattet einen besonders komfortablen Dialog mit dem Computer bei in der Entwicklung befindlichen Problemen, deren Algorithmierung erst im Laufe und im Zusammenhang mit Programmierung und Berechnung erarbeitet werden muß.
- Die Sprache ist sehr schnell erlernbar. Bei Kenntnis von etwa 50 Schlüsselwörtern ist mäßig anspruchsvolles Programmieren möglich. Kleine Probleme können bereits bei Kenntnis einer Teilmenge der Sprache bearbeitet werden. Kenntnisse spezieller Eigenschaften der Hardware sind nicht erforderlich. Die Verwendung des dezimalen Zahlensystems ist für fast alle Probleme angebracht und ausreichend.
- Die Sprache ist flexibel; sie ist für kleine wie für mäßig große Probleme und für Aufgaben aus den verschiedensten Wissensbereichen geeignet. Anpassungen an veränderte Aufgabenstellungen sind auf einfache Weise realisierbar.
- BASIC kommt der Forderung nach Portabilität entgegen. In dieser Sprache geschriebene Programme sind wegen ihrer geringen Maschinennähe bei zweckmäßiger Anwendung leicht von einem Rechner auf einen anderen zu übertragen.
- In BASIC geschriebene Programme sind bei sinnvoller Verwendung der Sprachelemente leicht verständlich oder, wie man auch sagt, transparent. Das hat seine Ursache insbesondere darin, daß alle Schlüsselwörter der englischen Umgangssprache entnommen sind. Die Einfügung von Kommentaren ist problemlos möglich.
- BASIC ist die verbreitetste Computersprache. Sie ist auf praktisch allen gängigen Computern implementiert. Dadurch ist eine Vielzahl von fertigen Programmen leicht beschaffbar.
- Die Zeichenkettenbehandlung ist in BASIC besonders elegant gelöst.
- Die Fehlersuche und -beseitigung, das "debugging" (auf deutsch das "Entlausen"), gestaltet sich in BASIC besonders einfach. Das ist insofern ein wichtiger Gesichtspunkt, als bei größeren Programmen das Debugging immer einen sehr wesentlichen Teil des gesamten Programmentwicklungsaufwands ausmacht.
- BASIC kann als Interpreter- und als Compilersprache genutzt werden. In der Version als Interpretersprache, bei der jede Programmzeile vor jeder Abarbeitung in eine maschinenverständliche Form gebracht wird, ist ein Programm zwar langsam, kann aber leicht verändert und berichtigt werden, und es kann mitten im Programmlauf angehalten und der aktuelle Stand der Variablen und Felder beobachtet werden. Nach Erreichung ausreichender Fehlerfreiheit kann dann das Programm compiliert werden, womit erheblicher Geschwindigkeitsgewinn verbunden ist.

Demgegenüber haben zumindest einfache BASIC-Versionen auch einige wesentliche Nachteile:

- BASIC verletzt einen wichtigen Grundsatz der strukturierten Programmierung, daß nämlich einzelne Programmodule bzw. einzelne Programmteile sich nicht unbeabsichtigt gegenseitig beeinflussen dürfen. Da es eine Unterscheidung zwischen lokalen und globalen Variablen bei den einfachen, hier vorausgesetzten Dialekten nicht gibt, kann bei unachtsamer Programmierung eine Variable eines bestimmten Namens ungewollt in entfernten Teilen des Programms verändert werden. Dieser erheblichen Gefahr ist nur durch sehr diszipliniertes Programmieren auszuweichen.

- BASIC erzwingt nicht die Anwendung der strukturierten Programmierung. Sinnvolle Strukturierung ist möglich, erfordert aber Disziplin und entsprechende Kenntnisse des Programmierers.
- Die Orientierung im Programm auf der Basis von Zeilennummern ist als Komplikation anzusehen. Der Aufruf von Programmteilen mit einem Namen, der nichts mit der Position im Programm zu tun hat, muß als eine bei weitem bessere Lösung bezeichnet werden. Es ist ebenfalls ungünstig, daß im Laufe der Programmentwicklung auf noch nicht vorhandene und damit nicht bekannte Zeilennummern verzweigt werden muß.
- Der Vorteil der Portabilität und Transparenz des ursprünglichen BASIC ist durch den Wildwuchs einer Vielzahl von Sprachdialekten sehr beeinträchtigt. Eine Standardisierung der Sprache hat zu lange auf sich warten lassen [ISO-84].
- Die Sprache ist nicht recheneffektiv. In der Version als Interpretersprache wird der größte Teil der Zeit für die vielfachen Übersetzungen verbraucht. Sprachversionen mit Compilern übersetzen das gesamte Programm vor der Ausführung in Maschinensprache oder eine maschinenspracheähnliche Form. Sie erreichen wesentlich größere Geschwindigkeiten, sind aber dennoch langsamer und speicherplatzaufwendiger als optimal entwickelte Maschinenprogramme oder Programme in maschinennahen Sprachen.

Aus den genannten Nachteilen erkennt man leicht die Grenzen für die Anwendung von BASIC: Die Sprache sollte nicht angewendet werden für Probleme mit sehr großem Rechenaufwand, bei denen die Rechenzeit zum wesentlichen Faktor wird. Die Anwendung ist ebenfalls nicht sinnvoll für große Aufgaben, an denen mehrere Programmierer arbeiten. Aufgaben, die direkten und effektiven Zugriff auf die Hardware erfordern, also etwa betriebssystemähnliche Programme, kann man nicht in BASIC schreiben. Ebenso sind Probleme, die lange und intensive Interaktion des Bedieners mit dem Rechner erfordern, also etwa Datei- oder Textverarbeitungsprogramme, keine geeigneten Objekte für eine BASIC-Programmierung. Sie würden für den Bediener zur Geduldsprobe.
Wir wollen in diesem vorangestellten Abschnitt einige Ausführungen machen, die das Wirksamwerden der Nachteile von BASIC verhindern helfen. Zunächst muß ein auf fast allen Rechnern lauffähiges Subset der Sprache angegeben werden, indem man sich auf eine Reihe grundlegender Sprachelemente beschränkt. Werden diese Elemente vom betreffenden Rechner realisiert, so können die in den weiteren Abschnitten folgenden Programme bedenkenlos übernommen werden. Fehlen einige Elemente, so wird es in vielen Fällen möglich sein, sie durch geeignete Konstrukte aus weniger komplexen Elementen zu ersetzen. Komfortablere BASIC-Dialekte werden von dem angegebenen Subset in ihrer Leistungsfähigkeit natürlich nicht voll ausgenutzt.

1.1.1. Verwendete Sprachelemente

Jedes BASIC-Programm besteht aus numerierten Zeilen (program lines) und aus Anweisungen (statements). Jede Zeile wird beendet durch eine Zeilenbegrenzung (line terminator). Die Zeilenbegrenzung wird durch die Wagenrücklauftaste (return oder enter) veranlaßt. Die Anweisungen bestehen i. allg. aus dem Anweisungsschlüsselwort (statement keyword) und dem Anweisungskörper (statement body). Der Anweisungskörper besteht seinerseits aus Ausdrücken (expressions), Funktionen, Variablen, Konstanten und Operatoren. Bei den Anweisungen unterscheidet man zwischen ausführbaren Anweisungen (executable statements) und nicht ausführbaren oder deklarierenden Anweisungen (non-executable statements). Als nicht ausführbare Anweisungen werden in unserem Subset nur die Anweisungen **rem** und **data** verwendet werden.

Eine BASIC-Programmzeile hat demnach beispielsweise folgende Form:

Zeilen-nummer	Anweisungs-schlüsselwort	Anweisungs-körper	Zeilen-begrenzer
1010	**print**	**sqr**(a*x + **exp**(x↑2))	RETURN

In unserem Subset werden keine Zeilen mit Mehrfachanweisungen verwendet. Lediglich die für den Programmablauf unerheblichen Kommentare (remarks, **rem**-Anweisungen) werden zuweilen nach einer ersten Anweisung noch auf der gleichen Programmzeile untergebracht. Zwischen den beiden Anweisungen wird dann als Trennzeichen der Doppelpunkt verwendet.
Als Zeilennummern werden Zahlen aus dem Bereich 0 bis 63 999 verwendet. Einige Rechner sind nicht in der Lage, so hohe Zeilennummern zu akzeptieren. In diesem Falle ist entsprechend umzunumierieren. Die maximale Zeilenlänge wird mit 80 Zeichen (einschließlich Zeilennummer) angenommen. Auch hier sind einige wenige Rechner auf eine Verringerung angewiesen. Die meisten der nachfolgend angegebenen Programmlistings enthalten nur gerade Zeilennummern. Dadurch hat man ggf. die Möglichkeit, lange Zeilen ohne völlige Umnumerierung in zwei kürzere aufzuteilen.
Elementarer Bestandteil jedes BASIC-Programms ist der Zeichensatz (character set), der aus Ziffern, Buchstaben und Spezialzeichen besteht. Der in diesem Buch verwendete Zeichensatz ist eingeschränkt. Eine Besonderheit von BASIC besteht darin, daß Leerzeichen (spaces) außer in Kommentaren und Zeichenketten (strings) keinerlei Bedeutung haben.
Neben den aufgezählten Sprachelementen, die nachfolgend detaillierter behandelt werden, sind für den Rechenbetrieb die sog. Kommandos (commands) von besonderer Bedeutung. Sie veranlassen den Rechner zur Ausführung von Aktionen, erscheinen aber nicht im Programmlisting (z. B. LIST, RUN, MERGE, RENUMBER u. a.) Entsprechend der Zielstellung dieses Buches unterbleibt die Behandlung der Kommandos; die Gerätedokumentationen sollten hierzu ausreichende Informationen geben.

1.1.2. BASIC- Zeichensatz

Es wird von der in BASIC üblichen Verwendung des ASCII-Zeichensatzes ausgegangen. ASCII bedeutet "American Standard Code for Information Interchange". Die festgelegten Kodenummern ordnen den vom Rechner nicht direkt verarbeitbaren Zeichen numerische Äquivalente zu. Die ASCII-Kodes des von uns verwendeten eingeschränkten Zeichensatzes sind im **Anhang 1** mit ihren dezimalen Werten angegeben.

1.1.3. Konstanten

Wir unterscheiden Gleitpunktkonstanten (real constants), Ganzzahl- oder Integerkonstanten (integer constants) und Zeichenkettenkonstanten oder Textkonstanten (string constants). Konstanten erscheinen i. allg. mit allen sie repräsentierenden Ziffern und Zeichen bzw. Buchstaben im Programmlisting oder bei der Datenein- oder Datenausgabe. Ihr Name besagt, daß sie innerhalb des Programms unveränderlich sind.
Gleitpunktkonstanten repräsentieren gebrochene Zahlen mit Dezimalteilen. In Mikrorechnern, für die wir im folgenden 8-Bit-Architektur annehmen, decken sie üblicherweise den Wertebereich 2^{-128} bis 2^{127} bzw. 2.93873588 * 10E-39 bis 1.70141183E + 38 ab.
Wertebereich und Genauigkeit der Gleitpunktkonstanten werden verständlich durch die Tatsache, daß der Prozessor des Rechners intern mit binärer Gleitpunktarithmetik arbeitet. Intern ist jede Gleitpunktkonstante demnach dargestellt durch den Ausdruck

$$z = a \cdot 2^b \qquad (1.1)$$

a ist die Mantisse; sie belegt üblicherweise 3 bis 6 Byte, d. h. 3 * 8 = 24 bis 6 * 8 = 48 Binärstellen. Der Wertebereich ist $1/2 \leqq a < 1$.
b ist der Exponent; hierfür ist die Verwendung von 1 Byte üblich. Gespeichert wird immer der Wert b + 128. Da ein 8-Bit-Byte den Wertebereich 0 bis 255 abdeckt, gilt

$$-128 \leqq b + 128 \leqq 127. \tag{1.2}$$

Zurück zur Mantisse. Belegt sie n Bytes, so ist ihr Wert darstellbar durch den Ausdruck

$$a = c_1/256 + c_2/256^2 + c_3/253^3 + \ldots + c_n/256^n \tag{1.3}$$

(s. **Bild 1.1**). Da a zwischen 0 und 1 liegt, ist die erste Binärstelle, das erste Bit von c_v, immer 1 und braucht nicht gespeichert zu werden. Dieses Bit wird für das Vorzeichen verwendet.

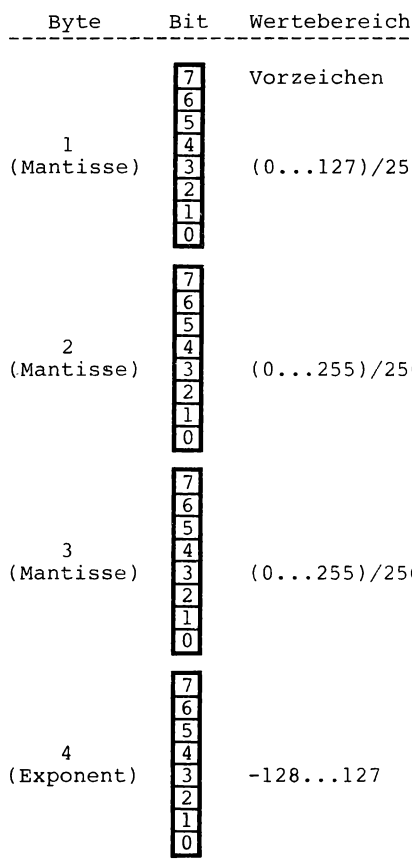


Bild 1.1. Rechnerinterne Organisation der Abspeicherung einer Gleitpunktzahl

Im dargestellten Beispiel werden vier Byte verwendet: drei für die Mantisse und eines für den Exponenten. Das erste Bit des ersten Bytes enthält die Vorzeicheninformation. – Ist kein Mantissenbit gesetzt, so ergibt sich der Wert 1/2, da das erste Bit des ersten Bytes automatisch als gesetzt betrachtet wird. Sind alle Mantissenbits gesetzt, so ergibt sich ein Wert, der zunächst um den Betrag der Maschinenkonstante < 1 ist. Da das letzte Bit des letzten Bytes immer gesetzt wird, erreicht man richtige Rundung. – Der Exponent besteht aus einem Byte, sein Wertebereich liegt also zunächst zwischen 0 und 255. Durch konstante Subtraktion von 128 wird der Bereich 2^{-128} bis 2^{127} realisiert.

Nach Kenntnis der obigen Ausführungen läßt sich n, die Anzahl der für die Mantissenabspeicherung verwendeten Bytes, folgendermaßen bestimmen:
Sind alle Bits des Mantissenbereichs gesetzt, so ist der Maximalwert der Mantisse

$$a_{max} = 255/256 + 255/256^2 + \ldots + 255/256^n = 1 - u. \tag{1.4}$$

Das Restglied dieser Entwicklung beläuft sich auf $u = 1/256^n$. u wird auch als Maschinenkonstante bezeichnet. Im folgenden Programm wird n erhöht, bis der Rechner $a-1$ nicht mehr von Null unterscheiden kann. Das größte n, für das dies zutrifft, ist die Anzahl der Mantissenbytes. Es wird ganzzahliges n vorausgesetzt.

```
30 n = n + 1
40 a = a + 255/256↑n
50 if1 - a > 0 goto 30
60 print "Die Mantisse belegt"; n; "Bytes"
70 print "Die Maschinenkonstante ist"; 1/256↑n
```

Es ist üblich, daß eine Überschreitung des Wertebereichs (overflow) vom Rechner als fataler Fehler (fatal error) angesehen wird, die Programmausführung wird also abgebrochen; bei Unterschreitung des Wertebereichs (underflow) wird der Konstanten der Wert null zugewiesen. Im letztgenannten Fall kommt es nicht zum Programmabbruch.
Integerkonstanten können nur ganze bzw. natürliche Zahlen sein. Wir kennzeichnen die Integerkonstanten hier nicht besonders; sie sind daher erst mit dem Rückruf aus einer Integervariablen eindeutig als solche zu erkennen.
Integerzahlen werden üblicherweise in zwei Bytes abgespeichert und überstreichen damit den Bereich -2^{15} bis $2^{15} - 1$ bzw. -32768 bis $+32767$. Die Verhältnisse sind aus **Bild 1.2** ersichtlich. Das erste Bit im ersten der beiden Bytes ist wiederum für das Vorzeichen verwendet. Angabe des positiven Vorzeichens ist nicht erforderlich. Bereichsüberschreitung ist ein fataler Fehler. Die Verwendung von Integerzahlen spart Speicherplatz und zumeist auch Rechenzeit.

Byte	Bit b	2^b
1 (high)	15	32768
	14	16384
	13	8192
	12	4096
	11	2048
	10	1024
	9	512
	8	256
0 (low)	7	128
	6	64
	5	32
	4	16
	3	8
	2	4
	1	2
	0	1

Bild 1.2. Organisation der Abspeicherung einer Integerzahl im 8-Bit-Rechner

Sind alle Bits gesetzt, so liefert die Summe aller 2^b die Zahl 65535. Durch konstante Subtraktion von 2^{15} wird der Wertebereich -32768 (kein Bit gesetzt) bis 32767 (alle Bits gesetzt) realisiert.

Man testet darauf, ob Integerkonstanten im betreffenden Rechner in zwei oder evtl. mehr Bytes abgelegt werden, indem man den Rechner zum Druck von Zahlen zu veranlassen sucht, die den o. g. Bereich überschreiten.
Stringkonstanten, auch Zeichenketten oder Literale, enthalten eine Anzahl von Zeichen, deren ASCII-Kode zwischen 0 und 127, bei manchen Rechnern auch zwischen 0 und 255 liegt. Die Stringkonstante wird durch Anführungsstriche (Begrenzer, delimiter) als solche kenntlich. Zur Stringkonstanten gehören die Begrenzer nicht, nur deren Inhalt. Demnach werden die Anführungsstriche auch bei der Ausgabe nicht mitgedruckt:

```
100 print "ABC 1.30 # XX"
RUN
ABC 1.30 # XX
```

Probleme können sich ergeben, wenn Anführungszeichen im String enthalten sein sollen. Man kombiniert dann mit Apostroph wie nachfolgend gezeigt:

```
10 print "Paul sagte 'lass mich in Ruhe'"
```

Manche Rechner akzeptieren auch:

```
10 print 'Paul sagte "lass mich in Ruhe"'
```

oder (als sicherste Methode):

```
10 print "Paul sagte" + chr$ (34) + "lass mich in Ruhe" + chr$ (34)
```

Die Anzahl der zulässigen Zeichen im String ist von Rechner zu Rechner verschieden. Sehr verbreitet sind maximal 255 Zeichen. Die nachfolgende Routine verlängert einen String bis zum Abbruch durch Stringfehler. Die zuletzt gedruckte Stringlänge ist die maximal mögliche.

```
30 n % = n % + 1
40 a$ = a$ + "a"
50 print "Stringlänge"; n %; "Zeichen"
60 goto 30
```

1.1.4. Variable

Einfache Variable werden durch Buchstaben oder durch Kombination eines Buchstabens mit einer nachfolgenden Ziffer gekennzeichnet. Das erste Zeichen des Namens muß ein Buchstabe sein. Ein so gewählter Variablenname reserviert im Rechner einen Speicherplatz, dessen Inhalt im Verlauf der Programmabarbeitung mit entsprechenden Konstanten belegt werden kann; die Belegung kann häufig verändert werden. Entsprechend der Art der zuzuordnenden Konstanten sind drei Arten von Variablen zu unterscheiden: Gleitpunkt-, Integer- und Stringvariable. Die Namen für Integer- und Stringvariblen werden ihrem Typ entsprechend gekennzeichnet: Integervariablen hängt ein % an, Stringvariablen ein $. Die Anzahl der reservierten Bytes entspricht der Zahl der Bytes für die zuzuordnenden Konstanten.
Beim Versuch, unpassende Konstanten in Variablen abzuspeichern, ergeben sich folgende Raktionen des Rechners:

	Gleitpunkt-variable	Integer-variable	String-variable
Gleitpunkt-konstante	akzeptiert	Integerzahl im Wertebereich	fataler Fehler
Integer-konstante	akzeptiert	akzeptiert	fataler Fehler
String-konstante	fataler Fehler	fataler Fehler	akzeptiert

Auf den Versuch zur Abspeicherung einer Gleitpunktzahl in einer Integervariablen wird wie angedeutet mit der Abspeicherung einer Integerzahl reagiert, sofern die Zahl im zulässigen Wertebereich liegt. Diese Integerzahl wird durch Weglassen, Abschneiden (truncation) des Nachkommateils gebildet; die Zahl wird nicht gerundet.
Bei den meisten BASIC-Versionen werden beim Start eines Programms die Inhalte aller Variablen auf Null gesetzt. Dieser Fall wird auch in den nachfolgenden Programmen angenommen. Wo das nicht der Fall ist, müssen ggf. bei Beginn des Programms entsprechende Zuweisungen erfolgen. Man testet etwa folgendermaßen:

```
100 print a; b; c; a1; b1; c1;
110 print a %; b %; c %; a1 %; b1 %; c1 %
120 print a$; b$; c$; a1$; b1$; c1$
```

Bei allgemeinem Ergebnis null kann mit dem vorausgesetzten Verhalten gerechnet werden.
Feldvariable (subscripted variables) sind Gleitpunkt-, Integer- oder Stringvariable, die mit einer Anzahl von Indizes (subscripts) versehen sind. Die Indizes sind als Marken oder Zeiger zu verstehen, die auf einen Wert des Feldes zeigen. Die Anzahl der Indizes kann man auch als Dimensionsanzahl verstehen: Ein Index bezeichnet ein eindimensionales Feld (Vektor), zwei Indizes ermöglichen den Zugriff auf ein zweidimensionales Feld (Matrix). Wir beschränken uns auf die Verwendung maximal zweidimensionaler Felder, und wir setzen weiter voraus, daß der Zählbeginn für alle Indizes bei 0 liegt und nicht variabel ist. Die Indizes sind als Gleitpunktkonstanten bzw. -variablen zu schreiben. Dennoch darf der Bereich der Indizes die obere Zahlengrenze für Integerkonstanten nicht überschreiten.
Der vorgesehene Wertebereich ist am Beginn des Programms durch die **dim**-Anweisung zu dimensionieren. Überschreitung des dimensionierten Wertebereichs führt i. allg. zu fatalem Fehler.
Es wird hier nicht vorausgesetzt, daß kleine Feldgrößen automatisch dimensioniert werden. Jedoch wird wie bei den einfachen Variablen angenommen, daß allen Feldvariablen zu Programmbeginn der Wert null zugewiesen wird. Es ist zulässig, daß der gleiche Variablenname für einfache und für Feldvariable verwendet wird; es ist aber i. allg. nicht zulässig, daß der gleiche Variablenname für ein- und für zweidimensionale Feldvariable Verwendung findet.

Beispiel

x	x (j1)
x (j1)	x (j1, j2)
zulässig	nicht zulässig

Den für ein Variablenfeld erforderlichen Speicherplatz bestimmt man folgendermaßen, wobei wieder auf rechnerspezifische Unterschiede hingewiesen sei:

	Gleitpunktfelder	Integerfelder	Stringfelder
Feldname	4 Byte	4 Byte	4 Byte
Jede Dimension	2 Byte	2 Byte	2 Byte
Jedes Element	n Byte	2 Byte	3 Byte + 1 Byte je Zeichen

Man wähle die Variablennamen möglichst sinnvoll, z. B. a für Fläche, f für Kraft, so daß das Lesen des Programms mnemonisch unterstützt wird. In Analogie zu FORTRAN ist es günstig, Integervariable mit i ... n zu benennen.

1.1.5. Ausdrücke

Ausdrücke (expressions) verknüpfen Konstanten oder Variable durch Operatoren. Wir unterscheiden arithmetische, logische und Stringausdrücke.
Arithmetische Ausdrücke enthalten mindestens einen der folgenden Operatoren:

+ Addition
− Subtraktion
* Multiplikation
/ Division
↑ Potenzierung.

Zwei unmittelbar aufeinander folgende Operatoren sind nicht zulässig. Negation einer Zahl ist zu klammern, z. B. z = a* (−b). Die Verarbeitung von Integerkonstanten in arithmetischen Ausdrücken wird in den verschiedenen BASIC-Dialekten sehr uneinheitlich gehandhabt. Wir werden solche Ausdrücke i. allg. vermeiden und ggf. besonders darauf hinweisen. Die Operatoren werden in einem Ausdruck entsprechend ihrer Priorität abgearbeitet, Operatoren gleicher Priorität innerhalb der Programmzeile von links nach rechts. Vor allen anderen Berechnungen werden die Inhalte von Klammern abgearbeitet. Die innerste Klammer hat die höchste Priorität. Die maximal zulässige Anzahl von Klammerebenen ist i. allg. größer als 10, ggf. ist dies durch ein entsprechendes Programm zu testen. Die Hierarchie der Operatoren ist folgende:

↑ höchste Priorität
*/
+− niedrigste Priorität.

Beispiele

(25 + (16 * (9 ↑ 2)))
(25 + (16 * 81)) innere Klammer
(25 + 1296) innere Klammer

Der Ausdruck liefert ohne Klammern das gleiche Ergebnis:

25 + 16 * 9 ↑ 2
25 + 16 * 81 Potenz hat höchste Priorität
25 + 1296 Multiplikation hat höchste Priorität

15 ↑ 2 + 12 ↑ 2 − (35 * 8)
15 ↑ 2 + 12 ↑ 2 − 280 Klammer
225 + 12 ↑ 2 − 280 linke Potenz
225 + 144 − 280 Potenz
369 − 280 gleiche Priorität, v.l.n.r.

18/3 * 8
6 * 8 gleiche Priorität, v.l.n.r.

18/(3 * 8)
18/24 Klammer hat höchste Priorität.

Logische Ausdrücke gestatten die Verwendung folgender Operatoren:

Operator	Beispiel	Bedeutung
=	a = b	a ist gleich b
<	a < b	a ist kleiner als b
>	a > b	a ist größer als b
<=	a < = b	a ist kleiner oder gleich b
>=	a > = b	a ist größer oder gleich b
<>	a < > b	a ist nicht gleich b

Das Ergebnis der Anwendung werden wir hier nur zur Ausführung bedingter Sprünge verwenden.
Die logischen Operatoren können auch auf Stringkonstanten angewendet werden und ermöglichen z. B. das automatische Sortieren. Der Stringvergleich wird Zeichen für Zeichen von links nach rechts vorgenommen; die Entscheidung wird auf der Basis der ASCII-Kodenummern gefällt. Beim Vergleich von Strings unterschiedlicher Länge wird der kürzere String behandelt, als sei er mit angehängten Leerzeichen (**chr$** (0), NUL) bis zur Länge des längeren Strings aufgefüllt.

Beispiel

```
a$ = "BASIC"
if a$ < "BASIS" goto ...
```

Die Bedingung ist erfüllt, der Sprung wird ausgeführt.

Stringausdrücke dürfen darüber hinaus nur den "+"-Operator enthalten. Er ermöglicht die Stringverkettung (concatenation). Stringkonstanten können nicht mit numerischen Konstanten verkettet werden.

1.1.6. Anweisungen und Funktionen

Anweisungen (statements) veranlassen den Rechner zur Ausführung bestimmter Aktivitäten beim Programmablauf.
Funktionen verwandeln numerische oder Stringkonstanten entsprechend einer definierten Vorschrift oder liefern bestimmte Konstanten. Stringfunktionen, die ihrerseits einen String liefern, haben ein Dollarzeichen am Ende des Funktionsnamens; solche, die einen Zahlenwert liefern, tragen kein Dollarzeichen und sehen äußerlich wie numerische Funktionen aus, sind aber durch einen String im Argument gekennzeichent. Wir verwenden die im **Anhang 2** angegebenen Anweisungen und Funktionen. Sie sind dort in alphabetischer Folge der Anweisungsnamen aufgelistet.

1.2. Rechengenauigkeit

Begrenzte Rechengenauigkeit hat ihre Ursache in Abbruchfehlern bei der Gleitkommadarstellung und in der endlichen Approximationsgenauigkeit bei der Realisierung von Funktionen. Wir behandeln zuerst die Abbruchfehler.
Im Abschnitt 1.1 ist die interne Zahlendarstellung erläutert worden; es wurde gezeigt, wie eine Gleitpunktzahl in der Form

$$z = \pm a \cdot 2^b \tag{1.5}$$

gehandhabt wird, wobei die Mantisse a wegen der Abspeicherung in n Bytes mit dem Abbruchfehler

$$\Delta \leqq 2^{-8n} \text{ oder } \Delta \leqq 256^{-n} \tag{1.6}$$

behaftet ist. Die Möglichkeit zur Bestimmung des rechnerspezifischen n wurde im Abschnitt 1.1 beschrieben.
Bei der Verarbeitung von Gleitpunktzahlen im Prozessor wird die gesamte Mantisse nach Abspaltung des Vorzeichens um ein Bit nach links verschoben, und das nun leere Bit wird grundsätzlich gesetzt. Damit ist die Mantisse nunmehr nicht abgebrochen, sondern in der letzten Stelle richtig gerundet. Der Absolutwert des Restfehlers halbiert sich damit und hat jetzt mit gleicher Wahrscheinlichkeit positives oder negatives Vorzeichen:

$$|\Delta| \leqq 1/2 \cdot 2^{-8n}. \tag{1.7}$$

Der Rechner stellt die Zahl $\hat{z}$ demnach dar in der Form

$$\hat{z} = (a + \Delta) \cdot 2^b = z(1 + \varepsilon) \tag{1.8}$$

mit dem Relativfehler

$$\varepsilon = \Delta/a; \quad |\varepsilon| \leqq 1/a \cdot 2^{-8n-1}, \; |\varepsilon|_{max} = 2^{-8n}. \tag{1.9}$$

Bei der Kombination zweier entsprechend fehlerhafter Zahlen mittels der Grundrechenarten ergeben sich folgende Gesamtfehler:

Addition (Vorzeichen der Summanden gleich)
Absolutfehler:

$$\Delta(z_1 + z_2) = \Delta z_1 + \Delta z_2 = z_1\varepsilon_1 + z_2\varepsilon_2 \tag{1.10}$$

Maximaler Absolutfehler:

$$|\Delta(z_1 + z_2)|_{max} = |\Delta z_1| + |\Delta z_2| = (z_1 + z_2) \cdot |\varepsilon|_{max}. \tag{1.11}$$

Relativfehler:

$$\frac{\Delta(z_1 + z_2)}{z_1 + z_2} = \frac{\Delta z_1 + \Delta z_2}{z_1 + z_2} = \frac{z_1\varepsilon_1 + z_2\varepsilon_2}{z_1 + z_2} \tag{1.12}$$

Maximaler Relativfehler:

$$\left|\frac{\Delta(z_1 + z_2)}{z_1 + z_2}\right|_{max} = \frac{z_1|\varepsilon|_{max} + z_2|\varepsilon|_{max}}{z_1 + z_2} = |\varepsilon|_{max}. \tag{1.13}$$

Subtraktion:
Absolutfehler:

$$\Delta(z_1 - z_2) = |\Delta z_1| + |\Delta z_2| = (z_1 + z_2) \cdot |\varepsilon|_{max}. \tag{1.14}$$

Man beachte hier und im folgenden, daß die Vorzeichen der z_1 und z_2 unbestimmt sind. Deshalb verwenden wir bei Verknüpfungsoperationen immer das Additionszeichen. Auf diese Weise wird der größte mögliche Fehler betrachtet.

Maximaler Absolutfehler:

$$|\Delta(z_1 - z_2)|_{max} = |\Delta z_1| + |\Delta z_2| = (z_1 + z_2) \cdot |\varepsilon|_{max}. \tag{1.15}$$

Relativfehler:

$$\frac{\Delta(z_1 - z_2)}{z_1 - z_2} = \frac{\Delta z_1 + \Delta z_2}{z_1 - z_2} = \frac{z_1\varepsilon_1 + z_2\varepsilon_2}{z_1 - z_2}. \tag{1.16}$$

Dies ist ein außerordentlich gefährlicher Fehler, der bei der Algorithmierung und Programmierung sorgfältig beachtet werden muß.
Maximaler Relativfehler:

$$\left|\frac{\Delta(z_1 - z_2}{z_1 - z_2}\right|_{max} \to \infty \text{ für } z_1 = z_2. \tag{1.17}$$

Multiplikation
Absolutfehler:

$$\Delta(z_1 * z_2) = \Delta z_1 \cdot z_2 + \Delta z_2 \cdot z_1 + \Delta z_1 \cdot \Delta z_2 \approx \Delta z_1 \cdot z_2 + \Delta z_2 \cdot z_1 \approx z_1 \cdot z_2(\varepsilon_1 + \varepsilon_2). \tag{1.18}$$

Maximaler Absolutfehler:

$$|\Delta(z_1 \cdot z_2)|_{max} = |\Delta z_1| \cdot z_2 + |\Delta z_2| \cdot z_1 = 2z_1z_2|\varepsilon|_{max}. \tag{1.19}$$

Relativfehler:

$$\frac{\Delta(z_1 \cdot z_2)}{z_1 \cdot z_2} = \frac{\Delta z_1 \cdot z_2 + \Delta z_2 \cdot z_1 + \Delta z_1 \cdot \Delta z_2}{z_1 \cdot z_2} \approx \frac{\Delta z_1}{z_1} + \frac{\Delta z_2}{z_2} = \varepsilon_1 + \varepsilon_2. \tag{1.20}$$

Maximaler Relativfehler:

$$\left|\frac{\Delta(z_1 \cdot z_2)}{z_1 \cdot z_2}\right|_{max} = 2|\varepsilon|_{max}. \tag{1.21}$$

Division
Absolutfehler:

$$\Delta\left(\frac{z_1}{z_2}\right) = \frac{\Delta z_1 \cdot z_2 + \Delta z_2 \cdot z_1}{z_2\,(z_2 + \Delta z_2)} \approx \frac{\Delta z_1 \cdot z_1 + \Delta z_2 \cdot z_1}{z_2^2} = \frac{z_1}{z_2}(\varepsilon_1 + \varepsilon_2). \tag{1.22}$$

Maximaler Absolutfehler:

$$\left|\Delta\left(\frac{z_1}{z_2}\right)\right|_{max} = 2\,\frac{z_1}{z_2}|\varepsilon|_{max}. \tag{1.23}$$

Relativfehler:

$$\frac{\Delta\,(z_1 / z_2)}{z_1 / z_2} = \frac{(\Delta z_1 \cdot z_2 + \Delta\, z_2 \cdot z_1) \cdot z_2}{z_1 - z_2^2} = \frac{\Delta z_1}{z_1} + \frac{\Delta z_2}{z_2} = \varepsilon_1 + \varepsilon_2. \tag{1.24}$$

Maximaler Relativfehler:

$$\left|\frac{\Delta\,(z_1 / z_2)}{z_1 / z_2}\right|_{max} = 2\,|\varepsilon|_{max}. \tag{1.25}$$

Für die Fehler arithmetischer Funktionen in der Folge der Ungenauigkeiten der Gleitkommaarithmetik gelten folgende Beziehungen:

$$\Delta f\,(z) = f\,(z + \Delta z) - f\,(z) \approx f'(z) \cdot \Delta z \tag{1.26}$$

$$\frac{\Delta f\,(z)}{f\,(z)} \approx \frac{f'\,(z) \cdot z}{f\,(z)} \cdot \frac{\Delta z}{z}; \tag{1.27}$$

woraus z. B. folgt

$$\frac{\Delta x^n}{x^n} \approx n\varepsilon \tag{1.28}$$

$$\frac{\Delta e^z}{e^z} \approx z \cdot \frac{\Delta z}{z} \approx z\varepsilon \tag{1.29}$$

$$\frac{\Delta \log\,(z)}{\log\,(z)} \approx \frac{1}{\log\,(z)} \cdot \frac{\Delta z}{z} \approx \frac{1}{\log\,(z)} \cdot \varepsilon \tag{1.30}$$

$$\frac{\Delta \sin\,(z)}{\sin\,(z)} \approx z \cdot \cot\,(z) \cdot \frac{\Delta z}{z} \approx z \cdot \cot\,(z) \cdot \varepsilon. \tag{1.31}$$

Das Beispiel des Sinus zeigt, daß der relative Fehler des Funktionswerts in der Nähe der Nullstellen $k\pi$ gefährlich anwachsen kann. Das gilt in gleicher Weise für alle Funktionen, deren Funktionswert null wird an Stellen, an denen das Argument nicht auch entsprechend klein ist.
Es ist möglich, die „Buchführung" über die Rechenfehler dem Rechner selbst zu übertragen. Die Methode wird von der sog. Intervallarithmetik bereitgestellt: Es wird das Intervall, innerhalb dessen die Eingangsgrößen unsicher sind oder schwanken können, in das Intervall des daraus folgenden Ergebnisses umgerechnet. Sind z. B. die Eingangsdaten 1.57 und 1.62, so sind bereits wegen der begrenzten Genauigkeit bei der Eingabe der Zahlenwerte und ohne irgendwelche Beteiligung von Rechnerfehlern die möglichen Intervalle:

$1.57 = \{1.565\ ;\ 1.575\}$
$1.62 = \{1.615\ ;\ 1.625\}$

$1.57 + 1.62 = \{3.18\ ;\ 3.20\}$
$1.57 - 1.62 = \{-.060\ ;\ -.040\}$

1.57 * 1.62 = {2.5275 ; 2.5594}
1.57 / 1.62 = {.9631 ; .9752}

tan (1.57) = {172.521 ; −237.886}

Für die Verfolgung von Fehlern, die aus den Ungenauigkeiten der Gleitkommaarithmetik herrühren, kann das Verfahren in der Weise angewendet werden, daß ein z. B. um den Faktor 8 oder 10 vergrößerter Maschinenfehler angesetzt wird und daß man das erhaltene Ergebnisintervall auf den realen Maschinenfehler umrechnet. Die Intervallgrenzen sind im Programm zweckmäßig in jeder möglichen Kombination zu verwenden, und das Ergebnisintervall ist durch Aussuchen des größten und des kleinsten Ergebnisses zu bestimmen.
Für die Funktionsrealisierung ist die Verwendung von Rationalapproximationen üblich; die gemeinhin bekannten Reihenentwicklungen wären zu rechenzeitaufwendig und würden u. U. Konvergenzprobleme mit sich bringen. Zur Größe der Approximationsfehler können keine allgemeingültigen Angaben gemacht werden; bei guter Systemsoftware gehen sie in den Abbruchfehlern unter. Approximationsmethoden und deren Fehler sind beschrieben in [Cody80] [Hart78] [Hast55] [Luke76].
Wenn man sich ein Bild von der diesbezüglichen Qualität seines Rechners machen will, dann kann man die Werte der integrierten Funktionen den Ergebnissen von Reihenentwicklungen gegenüberstellen. Das **Programm P 1.01 FKTEST** leistet das für die Sinusfunktion. Die Mantissenlänge des Recherns, die mit der Testroutine im Abschnitt 1.1 bestimmt wird, ist auf Anforderung einzugeben. Aus diesem Wert wird das Abbruchkriterium bestimmt. Ein hinreichender Test für die Genauigkeit der BASIC-Funktionen ist das in P 1.01 FKTEST ge-

Programm P 1.01. FKTEST. Beispielprogramm für die Testung der in BASIC enthaltenen Funktionen.

```
1000 rem !-------------------------------------------------!
1002 rem ! P1.01  FKTEST                                    !
1004 rem !-------------------------------------------------!
1006 print "Mantissenlänge"
1008 input n1
1010 m=256^(-n1)
1012 print "Argument","Delta abs.","Delta rel."
1016 for x=.1 to 3.2 step .1
1018   gosub 2000
1020   print x,s1-sin(x),(s1-sin(x))/s1
1022 next x
1024 goto 63999
1026 rem !-------------------------------------------------!
1028 rem !-------------------------------------------------!
1030 rem
1032 rem
2000 rem !-------------------------------------------------!
2002 rem ! SINSER                                           !
2004 rem !    inp:n1,x                                      !
2006 rem !    int:i9,u9                                     !
2008 rem !    out:s1                                        !
2010 rem !-------------------------------------------------!
2012   s1=0
2014   u9=x
2016   for i9=1 to 10000
2018     s1=s1+u9
2020     u9=u9*(-1)*x*x/((2*i9)*(2*i9+1))
2022     if abs(u9/s1)<m goto 2026
2024   next i9
2026 return
2028 rem !-------------------------------------------------!
63999 end
```

Die betreffende Rechnerfunktion wird den Ergebnissen einer Reihenentwicklung gegenübergestellt. Die Reihenentwicklung ist in einer gesonderten Unterroutine zu programmieren. Zur Verwendung beim Abbruchtest ist die Mantissenlänge n1 des Rechners einzugeben (s. Abschn. 1.1). Das Beispiel enthält das Unterprogramm Reihenentwicklung für die Sinusfunktion.

---

zeigte Verfahren nicht, da die Abbruchfehler bei den Reihenentwicklungen erheblich groß werden können. Da indessen das Fehlverhalten der Rationalapproximationen und das der Reihenentwicklungen mit hoher Wahrscheinlichkeit verschieden ist, kann eine Information über das Vorhandensein von Diskrepanzen aus einer größeren Anzahl von Realisierungen gewonnen werden. Für genauere diesbezügliche Untersuchungen vergleiche man mit vielstelligen Tafeln, wie sie z. B. in [Abra68] enthalten sind.
Ein schneller und instruktiver Test für die Genauigkeit der implementierten Funktionen ist der von *Savage* vorgeschlagene Test [Kels87]. In einer Schleife wird die Variable a hochgezählt. Bei jedem Schleifendurchlauf wird a den Operationen Quadrieren, Radizieren, Logarithmieren, Exponentialfunktion, Arcustangensfunktion und Tangensfunktion unterworfen. Bei fehlerfreier Rechnung dürfte a dadurch nicht verändert werden. In der Praxis summieren sich die Fehler auf und werden dadurch sehr deutlich. Gleichzeitig bekommt man eine Information über die Geschwindigkeit der Funktionenbildung (s. Abschn. 1.6). Ergebnisse der Testung verschiedener Rechner und Sprachen findet man im **Anhang 4**.

```
1000 rem Savage-Test
1010 a = 1
1020 t1 = ti
1030 for i = 1 to 2499
1040 a = tan (atn (exp (log(sqr (a * a))))) + 1
1050 next i
1060 print t1, ti, (a - 2500)/2500
1070 end
```

(**ti**: Systemzeit)

1.3. Strukturierte Programmierung und Grundsätze für den Programmaufbau

Strukturierter Programmaufbau beeinflußt – mit Ausnahme evtl. der Recheneffizienz – alle im Abschnitt 1 genannten Forderungen an die Interaktion des Benutzers mit dem Rechner. Die Beachtung der Regeln der strukturierten Programmierung ist unbedingt zu empfehlen.

1.3.1. Strukturierte Programmierung

N. Wirth [Wirt71] schreibt zum systematischen Vorgehen bei der Programmierung:
„Die Entwurfsphase hat bei der Analyse der zu lösenden Aufgabe zu beginnen und hat von ihrer ... späteren Implementierung ... zu abstrahieren. Ausgehend von der Aufgabenstellung wird eine Aufgabe solange in Teilaufgaben aufgelöst, bis die Funktionen vorliegen, die

leicht in ein Programm umzusetzen sind. Daraus resultiert eine baumartige Aufgabenstruktur, für die in der Fachliteratur synonym auch die Bezeichnungen ‚schrittweise Verfeinerung' und ‚hierarchische Programmstruktur' zu finden sind ... Auf diesem Abstraktionskonzept basieren die meisten Konstruktionsmethoden des modernen Softwareentwurfs. Es stellt quasi eine Minimalforderung dar, die zu erfüllen ist, damit ein Programmentwurf das Attribut ‚systematisch' tragen kann."
Die von *Wirth* vorgeschlagene systematische Arbeitsmethode entspricht auch dem neuerdings verwendeten Begriff der Top-down-Programmierung, bei der ebenfalls das komplexe Gesamtproblem (top) in immer kleinere Teilprobleme zerlegt wird, die schließlich elementar lösbar sind (down). Daneben kennt man die Technik der Bottom-up-Programmierung, die davon ausgeht, daß die vorhandenen Funktionen und Anweisungen (bottom) für die Problemlösung zu elementar sind. Man schafft sich daraus wirksamere Anweisungsblöcke und aus diesen noch wirksamere (up), bis ein Block bzw. ein Programm geschaffen ist, das das Problem löst. – In der praktischen Arbeit wird selten allein Top-down oder Bottom-up programmiert, man wird zumeist beide Ansätze in Kombination benutzen. Das Ergebnis solchen Vorgehens sollten auf jeden Fall strukturierte Programme sein, womit folgendes gemeint ist:

- Das Programm besteht aus einzelnen Blöcken oder Modulen, die alle nur einen Eingang und einen Ausgang haben.
- Die Module sollen voneinander unabhängig und austauschbar sein.
- Die Module können ihrerseits in hierarchischer Folge wieder aus Modulen zusammengesetzt sein. Sonst dürfen sie nur hintereinander, in Schleifen, oder bei Verzweigungen auch nebeneinander stehen. Sie dürfen sich nicht überlappen.
- Sprünge von Modul zu Modul dürfen nur über die Aus- und Eingänge und unter Beachtung der Hierarchieverhältnisse erfolgen.
- Unbedingte Sprünge sind nach Möglichkeit zu vermeiden; die Fehlerdichte in Programmen ist nach *Dijkstra* [Dijk68] proportional zur Anzahl der darin vorkommenden GOTO-Anweisungen.
- Für alle Module, besonders aber für das Hauptprogramm, ist streng linearer Aufbau anzustreben. Das ist i. allg. wegen notwendiger Verzweigungen nicht zu erreichen, jedoch sind alle Programmteile möglichst eng um einen linear gedachten Ablauf zu führen. Nach Verzweigungen soll – auch wenn dadurch später weitere Verzweigungen nötig werden – möglichst rasch wieder zu einem gemeinsamen Strang zusammengeführt werden.
- Module sollen nicht länger als ein gleichzeitig zu übersehender Bildschirminhalt sein, höchstens aber den Umfang einer A-4-Seite haben. Das Hauptprogramm soll durch Kommentare so ausführlich erläutert sein, daß dessen innere Logik ohne Schwierigkeit verfolgt werden kann.
- Jeder Modul ist dafür verantwortlich, daß er gültige Daten weitergibt; die Testung auf Gültigkeit erfolgt also nicht bei Programmbeginn. Eingangsdaten sind ggf. zuerst durch Fehlerbehandlungsroutinen zu schicken, um den Gültigkeitsbereich der empfangenen Daten zu vergrößern.

Ein in der beschriebenen Weise strukturiertes Programm kann fast alle der am Beginn dieses Abschnitts erwähnten Qualitätsmerkmale aufweisen. Ein unstrukturiertes Programm ist dagegen in fast allen Qualitätsparametern beeinträchtigt. Der Aufwand für Programmierung und Fehlersuche beim strukturierten Programm ist gering; Lesbarkeit, Verständlichkeit, Wartbarkeit und Änderungsfreundlichkeit sind leicht zu gewährleisten. Die Anwendung für ähnliche Aufgaben und die Wiederverwendbarkeit von Modulen werden optimal erreicht. Die Erarbeitung der Dokumentation kann rationalisiert werden. Es können mehrere Programmierer gleichzeitig an einem Programm arbeiten, und die Erprobung von Teilfunktionen kann bereits vor der Fertigstellung des Gesamtprogramms beginnen. Die Komplexität und der Umfang eines Programms sind an der Anzahl der verwendeten Module und der Anzahl der Hierarchiestufen zu messen, nicht an der Kompliziertheit und Verwickeltheit.

Man sollte sich dennoch nicht vor der Feststellung scheuen, daß ein Programm seine derart definierte Struktur u. U. teilweise verliert, wenn es ohne Rücksicht auf andere Qualitäten konsequent zeit- oder speicherplatzoptimiert wird. Mit zunehmender Leistungssteigerung auf der Hardwareseite sinkt aber die Bedeutung dieser Merkmale, zumal auch aus der Abkehr von der Strukturierung keine bedeutenden Gewinne resultieren können.
Es wird zuweilen behauptet, daß es nicht möglich sei, mit BASIC strukturiert zu programmieren, während z. B. PASCAL quasi automatisch strukturierte Programme liefere. In so krasser Form trifft das nicht zu. Obwohl PASCAL eine Reihe von Anweisungen aufweist, die die strukturierte Programmierung unterstützen, bietet diese Sprache gleichwohl auch alle Möglichkeiten zur Schaffung wilder Spaghettiprogramme; demgegenüber bietet BASIC fast alle Möglichkeiten zur Erreichung gut strukturierter Programme. Der wesentliche Nachteil von BASIC besteht nur darin, daß alle Variablen globalen Charakter haben, d. h., eine bestimmte Variable kann von jedem Programmteil aus verändert werden. Dadurch ist die Forderung nach Unabhängigkeit der Programmodule nicht erfüllt; nur durch Sorgfalt bei der Programmierung kann die gegenseitige Beeinflussung vermieden werden.
Programmiersprachen, die lokale Variablen anbieten (z. B. PASCAL, FORTRAN), ermöglichen die zwangsweise Entkoppelung der Module.

1.3.2. Grundsätze für den Programmaufbau

Um die genannten Qualitätskriterien nach Möglichkeit zu erfüllen, um strukturierte Programmierung zu unterstützen und um möglichst hardwareunabhängig zu sein, wollen wir in diesem Buch folgende Regeln beachten:

- Mit dem Ziel geringer gegenseitiger Beeinflussung der Programmbausteine bzw. Unterprogramme bekommen Variable, die in den Unterprogrammen nur als Hilfsvariable verwendet werden, Namen mit 7, 8 oder 9 als zweitem Zeichen (z. B. c7, f8$, x9%), und zwar vorzugsweise so, daß Unterprogramme der ersten Hierarchieebene (vom Programm direkt aufgerufen) Variablennamen mit der Kennzahl 9 verwenden, Unterprogramme der zweiten Hierarchieebene (von einem Programm der ersten Hierarchieebene aufgerufen) Kennzahl 8 und Unterprogramme der dritten und bei uns letzten vorkommenden Hierarchieebene die Kennzahl 7. Wenn abzusehen ist, daß ein Unterprogramm keine untergeordneten Programme mehr bekommen kann (z. B. Lösung von Gleichungssystemen, Interpolationen usw.), dann wird immer die Kennzahl 7 verwendet.
- Variable, die an ein Unterprogramm übergeben werden, sollen nach Möglichkeit ihre ursprüngliche Größe behalten und nicht im Unterprogramm mit anderen Werten überschrieben oder verändert werden (z. B. durch Umrechnung von Winkeln aus dem Grad- in den Radianmodus).
- Ganzzahlige Schleifenzähler werden vorwiegend durch Variablennamen bezeichnet, die mit i...m beginnen. Wir bezeichnen die Schleifenzähler aber nicht mit Integer (z. B. i8%), da manche BASIC-Versionen damit die Schleifensteuerung nicht ausführen können.
- Um einen Überblick über die in einem Programm bereits verwendeten Variablennamen zu behalten, empfiehlt sich – sofern es sich um größere Programme handelt – die Verwendung einer Variablenliste. Mit dem **Programm P 1.02 VALIST** stellt man eine solche Liste leicht her. Die verschiedenen möglichen Variablenarten werden bei der Benutzung durch unterschiedliche Farben gekennzeichnet. Für sehr große Programme wird für jede Variablenart eine eigene Liste angelegt. Zweckmäßig schreibt man zu jedem Variablennamen die Zeilennummer des ersten Auftretens.
- Die Eingangsgrößen werden in den Modulen nach Möglichkeit nicht verändert.
  Winkelgrößen werden z. B. bei Ein- und Ausgaben in dezimalgeteiltem Altgrad genommen; intern wird in Radian gerechnet. Die Umrechnungen geschehen nur in den Ein- und Ausgabemodulen.

- In den Unterprogrammen können durch bestimmte Datenkonstellationen Fehler entstehen, die das Rechenergebnis unzulässig beeinflussen. (Im Hauptprogramm sollten keine Operationen ausgeführt werden, die solche Situationen zu schaffen in der Lage sind.)

Programm P 1.02. VALIST. Programm zur Erzeugung einer Variablenliste für BASIC-Programme.

```
1000 rem !-------------------------------------------------!
1002 rem ! P1.02 VALIST                                     !
1004 rem !-------------------------------------------------!
1006 open 1,4
1008 for i=1 to 85 : print#1,"-"; : next i : print#1
1010 print#1,"Variablenliste fuer Programm:"
1012 for i=1 to 85 : print#1,"-"; : next i : print#1
1014 for i=65 to 90
1016   print#1,chr$(i);tab(7);
1018   for j=1 to 10
1020     if j<>7goto 1026
1022       print#1,chr$(i)+chr$(j+47)+"    !";tab((j+1)*7);
1024     goto 1028
1026       print#1,chr$(i)+chr$(j+47);tab((j+1)*7);
1028   next j
1030   print#1
1032   print#1,tab(61);"!"
1034   print#1,tab(61);"!"
1036 next i
1038 for i=1 to 85 : print#1,"-"; : next i : print#1
1040 print#1,"Farbencode:        ";
1042 print#1,"-";tab(10);"()";tab(20);"$";tab(30);
1044 print#1,"$()";tab(40);"%";tab(50);"%()"
1046 for i=1 to 85 : print#1,"-"; : next i : print#1
1048 close 1
1050 end
1052 rem !-------------------------------------------------!
```

```
--------------------------------------------------------------------------------
Variablenliste fuer Programm :
--------------------------------------------------------------------------------
a     a0    a1    a2    a3    a4    a5    a6    !    a7    a8    a9
                                                !
b     b0    b1    b2    b3    b4    b5    b6    !    b7    b8    b9
                                                !
.
.
.
.
w     w0    w1    w2    w3    w4    w5    w6    !    w7    w8    w9
                                                !
x     x0    x1    x2    x3    x4    x5    x6    !    x7    x8    x9
                                                !
y     y0    y1    y2    y3    y4    y5    y6    !    y7    y8    y9
                                                !
z     z0    z1    z2    z3    z4    z5    z6    !    z7    z8    z9
                                                !
--------------------------------------------------------------------------------
Farbencode:      -        ()        $        $()       %        %()
--------------------------------------------------------------------------------
```

Durch geeignete Kennzeichnung, z. B. in verschiedenen Farben, kann die Benutzung von Variablennamen vermerkt und eine Doppelbenutzung wirksam verhindert werden. Zweckmäßig schreibt man zu jedem Variablennamen die Zeilennummer des ersten Auftretens. Bei größeren Programmen empfiehlt sich statt der Farbkennzeichnung die Benutzung eigener solcher Listen für jeden Variablentyp.

Fehlersituationen werden durch Fehlerflags signalisiert, die in der Fehlervariablen e1% zusammengefaßt sind. Nach Beendigung eines Unterprogramms sind die Flags zu testen; ggf. sind entsprechende Fehlerbehandlungsmaßnahmen einzuleiten. Wenn Flag 0 gesetzt war, dann soll das zum Programmabbruch führen. Die einzelnen Flags werden als Binärzahlen verwendet, so daß der Wert von e1% folgendermaßen zu interpretieren ist:

e1% = 0	Kein Flag gesetzt. Routine fehlerlos durchlaufen
e1% = 1 = 2↑0	Flag 0 gesetzt (führt zum Programmabbruch)
e1% = 2 = 2↑1	Flag 1 gesetzt (nur Hinweis, kein Abbruch)
e1% = 4 = 2↑2	Flag 2 gesetzt (nur Hinweis, kein Abbruch)

e1% = 16384 = 2↑14 Flag 14 gesetzt (nur Hinweis, kein Abbruch).

Die Auswertung eines e1%, bei dem mehrere Flags gesetzt waren, kann – wo vorhanden – durch die Funktion Binärzahltransformation realisiert werden. Wir setzen das Vorhandensein dieser Funktion nicht voraus und verwenden z. B. das **Programm P1.03 ERRPRG** zur Fehlerbehandlung. Kann der gleiche Fehlerfall z. B. beim Durchlaufen einer Schleife mehrmals auftreten, so darf selbstverständlich das entsprechende Fehlerflag nicht einfach hochgezählt

Programm P 1.03. ERRPRG. Das Programm dient zur Auswertung der verschiedenen Flags in der Fehlervariablen e1%.

```
9000 rem !-------------------------------------------------!
9002 rem ! P1.03  ERRPRG                                   !
9004 rem !    inp:e1%,z1$                                  !
9006 rem !    int:e7%                                      !
9008 rem !    out:-                                        !
9010 rem !-------------------------------------------------!
9012   if e1%=0 goto 9034
9014   print "Fehlerzustand nach Routine ";z1$
9016   print "Fehlernummer ";
9018   e7%=e1%
9020   for i7=14 to 0 step -1
9022     if e7%/2^i7<.99995 goto 9028
9024     print i7;
9026     e7%=e7%-2^i7
9028   next i7
9030   print
9032   stop
9034 return
9036 rem !-------------------------------------------------!
```

Wenn Flag 0 gesetzt ist, wird das als fataler Fehler angesehen. Die Programmabarbeitung wird in diesem Fall unterbrochen.

---

werden. Komfortablere BASIC-Versionen verfügen über den logischen Operator **or**, mittels dessen man z. B. Flag 4 folgendermaßen setzt.:

e1% = e1% **or** 2↑4

Verfügt man über den **or**-Operator nicht, so schreibt man z. B.

if usw. (Fehler) **then if** e9% = 0 **then** e9% = 2↑4

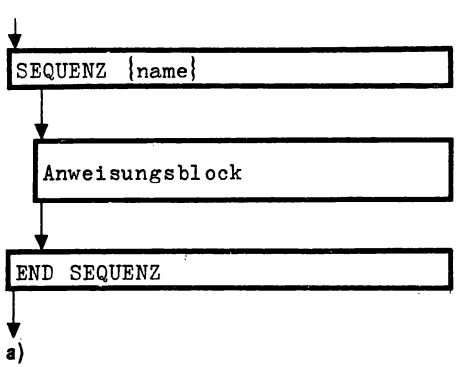
und addiert e9% am Schluß des Unterprogrammes zu e1%.

1.3.3. Programmbausteine

Programmbausteine sind als komplexere funktionelle Einheiten zu verstehen. Jedes Programm kann bei geeignetem Bausteinsatz vollständig aus solchen Bausteinen aufgebaut werden. Die Kombination kann durch Aneinanderreihung (sequentielle Folge) oder durch Schachtelung (nesting) geschehen. Andere Formen der Kombination sollten möglichst nicht verwendet werden. Allgemein ausgedrückt ist die sequentielle Folge die Kombination bei gleicher Hierarchieebene, die Schachtelung dagegen die Kombination mit unterschiedlichen Hierarchieebenen, wobei der untergeordnete Baustein vollständig in dem übergeordneten enthalten sein muß. Wir versuchen, die Hierarchieebenen durch je zwei Leerzeichen vor Zeilenbeginn deutlich sichtbar zu machen.
BASIC stellt im Vergleich zu höher entwickelten Sprachen, wie PASCAL [Paul81] oder COMAL [Chri85], fast nur Bauelemente für solche Bausteine bereit. Wir zeigen nachfolgend, daß alle PASCAL- und COMAL-Bausteine auch in BASIC mit geringem Mehraufwand realisierbar sind. In der **rem**-Zeile am Beginn jedes Bausteins geben wir außer dem Namen des Bausteins auch die verwendeten Variablen an, und zwar nach INP: die in den Baustein eingehenden Variablen; nach OUT: die vom Baustein für weitere Verwendung generierten oder veränderten Variablen; nach INT: die nur im Baustein intern verwendeten Variablen.
Die anschließend gegebene Darstellung der Programmbausteine hat in der konsequenten Form hauptsächlich didaktischen Wert; in der Praxis wird man in der Regel nicht alle angegebenen **rem**-Zeilen benutzen, obwohl dadurch eine ausgezeichnete Programmdokumentation entstünde. Dagegen sollte man konsequent versuchen, Programme nur aus diesen Bausteinen aufzubauen.

1.3.3.1. Linearer Ablauf (Sequenz)

Die Sequenz ist ein linear zu durchlaufender Programmteil. **Bild 1.3** zeigt die Struktur der Sequenz.



a)

```
1ØØØ rem  SEQUENZ {name}
1ØØ2 rem    inp: ...
1ØØ4 rem    int: ...
1ØØ6 rem    out: ...
  .
  .
  .          Anweisungsblock
  .
  .
  .
2ØØØ rem  END SEQUENZ
```

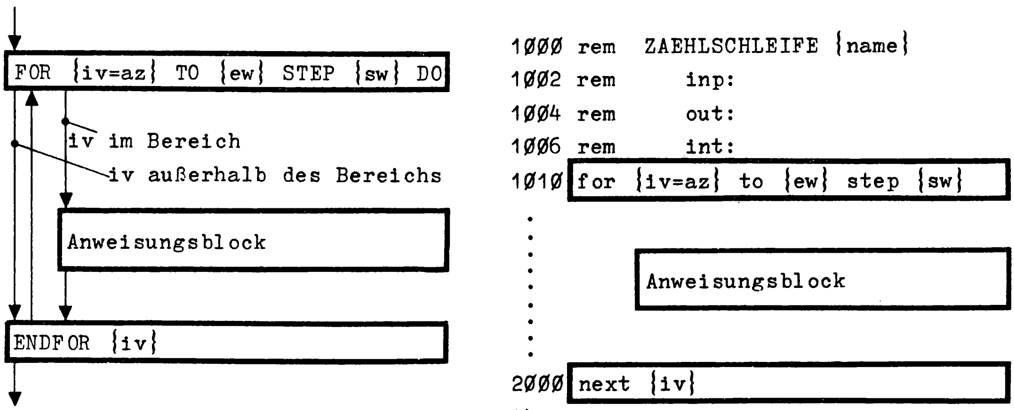
b)

```
1000 rem!---------------------------------!
1002 rem! SEQUENZ INITIALISIERUNG         !
1004 rem!    inp:u1(i9,j9),u2(i9,j9),u3(i9,j9)!
1006 rem!    int:s9,t9                    !
1008 rem!    int:u1(i9,j9),u2(i9,j9),u3(i9,j9)!
1010 rem!---------------------------------!
1012    u3(i9,j9)=u1(i9,j9)-u2(i9,j9)
1014    s9=u1(i9,j9)+u2(i9,j9)
1016    t9=u1(i9,j9)-u2(i9,j9)
1018    u1(i9,j9)=s9
1020    u2(i9,j9)=t9
1022 rem!---------END SEQUENZ-------------!
```
c)

Bild 1.3. Das Prinzip der Programmierung einer Sequenz
a) Version höher entwickelter Sprachen
b) Version in BASIC
c) Beispiel in BASIC

1.3.3.2. Zählschleife (FOR-NEXT-Schleife)

Mittels der Zählschleife wird der Anweisungsblock für jeden Wert ausgeführt, den die Index-Variable einschließlich Anfangs- und Endwert annimmt. Wir setzen voraus, daß der Block nicht ausgeführt wird, wenn der Endwert von vornherein kleiner als der Anfangswert ist. Das ist nicht bei allen BASIC-Interpretern der Fall. Zum Beispiel zeigen einige Commodore-BASIC-Versionen abweichendes Verhalten, was bei der Übernahme von Programmen zu schwer auffindbaren Fehlern führen kann. **Bild 1.4** zeigt die Struktur der Zählschleife.



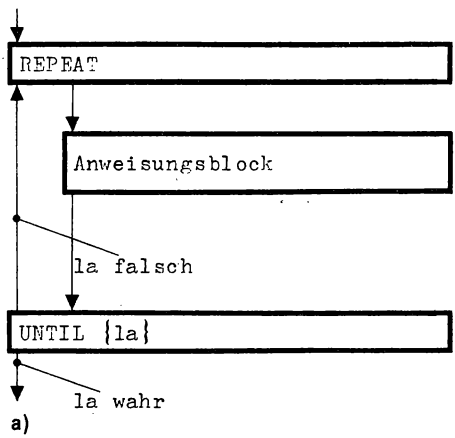
a) b)

```
1000 rem!---------------------------------!
1002 rem! ZAEHLSCHLEIFE SINUS             !
1004 rem!     inp:s1,u9,x                 !
1006 rem!     out:s1                      !
1008 rem!     int:i9                      !
1010 rem!---------------------------------!
1012 for i9=1 to 100
1014    s1=s1+u9
1016    u9=u9*(-1)*x*x/((2*i9)*(2*i9+1))
1018 next i9
1020 rem!---------END ZAEHLSCHLEIFE-------!
```
c)

Bild 1.4. Das Prinzip der Programmierung einer Zählschleife (FOR-NEXT-Schleife)
iv Indexvariable; az Anfangszuweisung; ew Endwert; sw Schrittweite
a) Version höher entwickelter Sprachen
b) Version in BASIC
c) Beispiel in BASIC

1.3.3.3. REPEAT-UNTIL-Schleife

Der Anweisungsblock wird zunächst ausgeführt; danach wird entschieden, ob der Block wiederholt wird. In der vorgeschlagenen BASIC-Variante ist der logische Ausdruck gegenüber der PASCAL-Variante zu negieren. Es empfiehlt sich, die REPEAT-UNTIL-Schleife in eine Zählschleife mit ausreichend großer Wiederholungszahl einzuschließen; das vermeidet zeitaufwendige Rücksprünge, außerdem wird die Gefahr unendlicher Loops verringert. Im **Bild 1.5** ist der Aufbau von REPEAT-UNTIL-Schleifen dargestellt.



a)

```
1000 rem  REPEAT {name}
1002 rem     inp:
1004 rem     out:
1006 rem     int:
      .
      .
      .
      .
      .
      .
      .
      .
1980 if {la} goto 1000
1990 rem  END REPEAT
```

b)

```
1000 rem!-----------------------------------!
1002 rem! REPEAT STRING                     !
1004 rem!    inp:b$                         !
1006 rem!    out:a$                         !
1008 rem!    int:                           !
1010 rem!-----------------------------------!
1012   input a$
1014   b$=right$(a$,5)
1016   if a$>b$ goto 1012
1018 rem!---------END REPEAT-----------------!
```

c)

```
1000 rem!-----------------------------------!
1002 rem! REPEAT STRING                     !
1004 rem!   inp:b$                         !
1006 rem!   int:i9                         !
1008 rem!   out:a$                         !
1010 rem!-----------------------------------!
1012 for i9=1 to 10000
1014   input a$
1016   b$=right$(a$,5)
1018   if a$>b$ goto 1014
1020 next i9
1022 rem!---------END REPEAT-----------------!
```

d)

Bild 1.5. Das Prinzip der Programmierung einer REPEAT-UNTIL-Schleife

$\overline{la}$ negierter logischer Ausdruck

a) Version höher entwickelter Sprachen

b) Version in BASIC

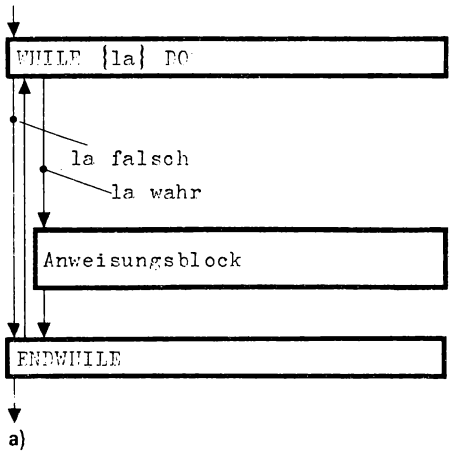
c) Beispiel in BASIC. Die Routine soll sichern, daß mit input ein String eingegeben wird, von dem fünf Zeichen wesentlich sind und der im Alphabet vor dem vorhandenen String b$ liegt.

d) Beispiel wie im Bild 1.5 c, aber mit Einschluß in eine Zählschleife.

1.3.3.4. WHILE-Schleife

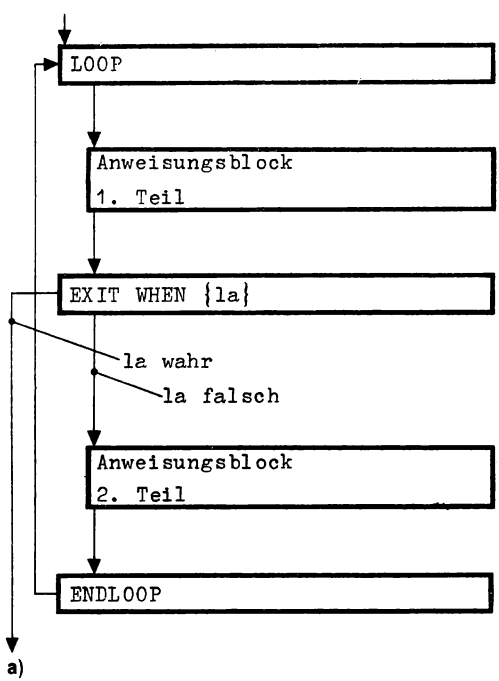
Solange (while) der logische Ausdruck wahr ist, wird der Block ausgeführt. Ist er falsch, so wird mit der auf die Schleife folgenden Zeile fortgefahren. In der vorgeschlagenen BASIC-Variante ist der logische Ausdruck gegenüber der PASCAL-Variante zu negieren. Es emp-

fiehlt sich auch hier, die Schleife in eine Zählschleife mit ausreichend großer Wiederholungszahl einzuschließen; das vermeidet zeitaufwendige Rücksprünge, außerdem wird die Gefahr unendlicher Loops verringert. **Bild 1.6** zeigt die Struktur von WHILE-Schleifen.




```
1ØØØ rem  WHILE {name}
1ØØ2 rem     inp:
1ØØ4 rem     int:
1ØØ6 rem     out:
1Ø1Ø if {la} goto 2ØØØ
  .
  .
  .          Anweisungsblock
  .
  .
  .
2ØØØ rem  END WHILE
```
b)

Bild 1.6. Das Prinzip der Programmierung einer WHILE-Schleife
la logischer Ausdruck; $\overline{\text{la}}$ negierter logischer Ausdruck
a) Version höher entwickelter Sprachen
b) Version in BASIC




```
1ØØØ rem  LOOP {name}
1ØØ2 rem     inp:
1ØØ4 rem     int:
1ØØ6 rem     out:
  .
  .
  .          Anweisungsblock
  .          1. Teil
  .
  .
  .
11ØØ if {la} goto 2ØØØ
  .
  .
  .          Anweisungsblock
  .          2. Teil
  .
  .
  .
19ØØ goto 1ØØØ
2ØØØ rem  END LOOP
```
b)

Bild 1.7. Das Prinzip der Programmierung einer LOOP-Schleife
la logischer Ausdruck

a) Version höher entwickelter Sprachen
b) Version in BASIC

```
1ØØØ rem!-----------------------------------!
1ØØ2 rem! LOOP ANTWORTPRUEFUNG              !
1ØØ4 rem!    inp:a                          !
1ØØ6 rem!    int:                           !
1ØØ8 rem!    out:a$                         !
1Ø1Ø rem!-----------------------------------!
1Ø12   print "ist z   ";a
1Ø14   input "JA oder NEIN",a$
1Ø16   if a$ = "JA" goto 1Ø22
1Ø18   a=a+1
1Ø2Ø goto 1Ø12
1Ø22 rem!---------END LOOP-------------------!
```

c)

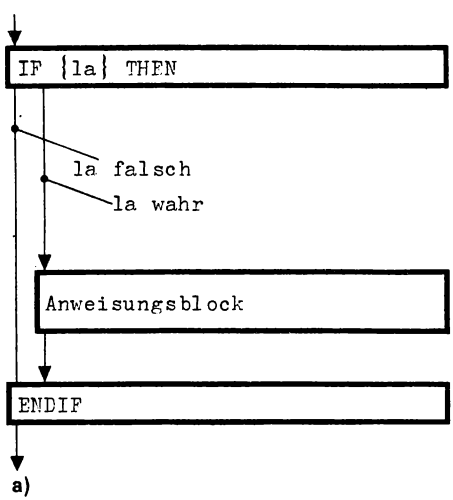
Bild 1.7.
c) Beispiel in BASIC

1.3.3.5. LOOP-Schleife

Die LOOP-Schleife wird verlassen, sofern der logische Ausdruck den Wert "wahr" hat. Die Abbruchbedingung steht zwischen zwei Teilen eines Anweisungsblocks. Auch hier empfiehlt sich u. U. der Einschluß in eine Zählschleife. **Bild 1.7** verdeutlicht den Aufbau einer LOOP-Schleife.

1.3.3.6. Abzweigung (IF-GOTO)

Wenn der logische Ausdruck wahr ist, so wird der Anweisungsblock ausgeführt; sonst wird er übersprungen. In der vorgeschlagenen BASIC-Variante ist der logische Ausdruck gegenüber der PASCAL-Variante zu negieren. **Bild 1.8** zeigt, wie eine solche Abzweigung aufgebaut ist.



a)

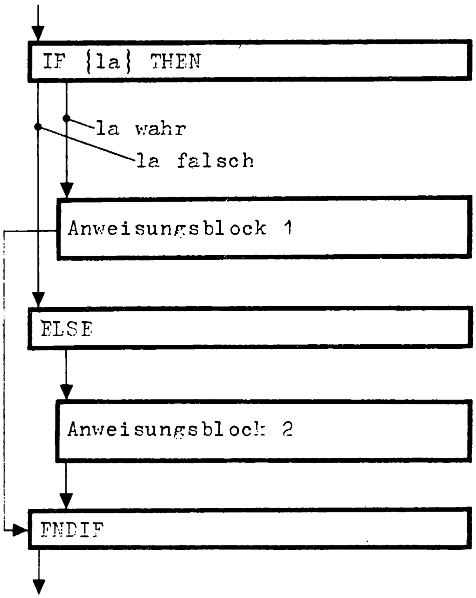
```
1ØØØ rem  IF{name}
1ØØ2 rem      inp:
1ØØ4 rem      int:
1ØØ6 rem      out:
1Ø1Ø if {la} goto {zn von ENDIF}
  .
  .
  .          Anweisungsblock
  .
  .
  .
2ØØØ rem  ENDIF
```

b)

Bild 1.8. Das Prinzip der Programmierung einer Abzweigung (IF-GOTO)
la logischer Ausdruck; $\overline{la}$ negierter logischer Ausdruck; zn Zeilennummer
a) Version höher entwickelter Sprachen
b) Version in BASIC

1.3.3.7. Gabelung (IF-THEN-ELSE)

Wenn der logische Ausdruck wahr ist, so wird der erste Anweisungsblock ausgeführt. Ist er falsch, so wird der zweite Block ausgeführt. In der vorgeschlagenen BASIC-Variante ist der logische Ausdruck gegenüber der PASCAL-Variante zu negieren. **Bild 1.9** zeigt den Aufbau einer Gabelung. Es ist ein Nachteil des vereinbarten einfachen BASIC, daß für die Gabelung mehr als eine Programmzeile benötigt wird.



a)

```
1000 rem   IF/THEN/ELSE {name}
1002 rem        inp:
1004 rem        int:
1006 rem        out:
1010 if {¬la} goto {zn von ELSE}
 :
 :        Anweisungsblock 1
 :
1500 goto {zn von END IF/THEN/ELSE}
1510 rem   ELSE
 :
 :        Anweisungsblock 2
 :
2000 rem   END IF/THEN/ELSE
```

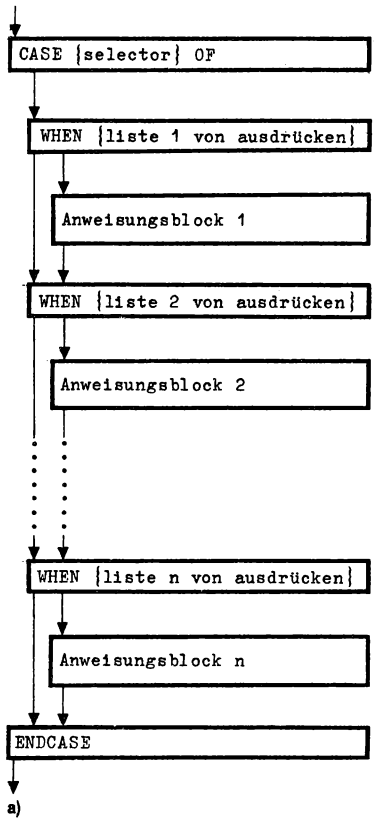
b)

Bild 1.9. Das Prinzip der Programmierung einer Gabelung (IF-THEN-ELSE)

la logischer Ausdruck; $\overline{la}$ negierter logischer Ausdruck; zn Zeilennummer
a) Version höher entwickelter Sprachen
b) Version in BASIC

1.3.3.8. Mehrfachverzweigung (CASE)

In Abhängigkeit von der Größe einer Steuerzahl (selector) werden ein oder mehrere Anweisungsblöcke zur Ausführung ausgewählt. Die volle Allgemeinheit der COMAL-Anweisung läßt sich in diesem Falle mit BASIC nicht vergleichsweise einfach simulieren; der Selektor muß in BASIC vor Einsprung in den Baustein erst aufbereitet werden, und die Abarbeitung mehrerer Blöcke erfordert i. allg. den Einbau des gesamten Bausteins in eine übergeordnete Schleife oder die Verwendung weiterer ON...GOTO...-Anweisungen statt der einfachen Verweise auf ENDCASE (GOTO 9000 in der ersten BASIC-Variante).
Ferner kann in BASIC nicht die Auswahl aus einer Liste mit mehreren Möglichkeiten unkompliziert ausgeführt werden.
In BASIC sind zwei Varianten möglich; man kann mit ON...GOTO und mit ON...GOSUB arbeiten. **Bild 1.10** zeigt die Realisierung solcher CASE-Konstrukte.



a)

```
1000 rem  CASE/GOTO {name}
1002 rem    inp:
1004 rem    int:
1006 rem    out:
1010 on i9 goto 2000,3000,5000,...
2000 rem WHEN i9=1
        .
        .
        .
        .
        .
        .
        .
                Anweisungsblock 1
2090 goto 9000
3000 rem  WHEN i9=2
        .
        .
        .
        .
        .
        .
        .
                Anweisungsblock 2
3090 goto 9000
5000 rem  WHEN I9=3
        .
        .
        .
        .
        .
        .
        .
                Anweisungsblock 3
5090 goto 9000
        .
        .
        .
        .
        .
        .
        .
9000 rem  ENDCASE
```

b)

Bild 1.10. Das Prinzip der Programmierung einer Mehrfachverzweigung (CASE)

le Listenelement; zn Zeilennummer
a) Version höher entwickelter Sprachen
b) Version in BASIC mit on. . .goto. . .
c) Version in BASIC mit on. . .gosub . . .
d) Version in BASIC, die der Version höher entwickelter Sprachen vollständig entspricht.
Es wird nicht nur eine einzige Möglichkeit ausgewählt und ausgeführt, sondern vor jedem Anweisungsblock wird entschieden, ob der Selektor irgendeinem Listenelement der zu dem betreffenden Block gehörenden Liste entspricht. Im allgemeinen werden also mehrere Anweisungsblöcke ausgeführt. In diesem Falle bieten PASCAL und COMAL, wie man sieht, erhebliche Vorteile gegenüber BASIC.

```
1000 rem  CASE/GOSUB {name}
1002 rem      inp:
1004 rem      int:
1006 rem      out:
1008 on i9 gosub 2000,3000,5000,...
1010 goto 5110
1012 rem  END CASE/GOSUB
1014 rem ----------------------------
1016 rem
2000 rem ----------------------------
2002 rem  UNTERPROGRAMM 1
  .
  .
  .        Anweisungsblock 1
  .
  .
2900 return
2910 rem ----------------------------
3000 rem ----------------------------
3002 rem  UNTERPROGRAMM 2
  .
  .
  .        Anweisungsblock 2
  .
  .
3080 return
3090 rem ----------------------------
4090 rem ----------------------------
5000 rem  UNTERPROGRAMM 3
  .
  .
  .        Anweisungsblock 3
  .
  .
5100 return
5110 rem ----------------------------
```

c)

```
1000 rem  CASE {name}{selector} OF
1002 rem      inp:
1004 rem      int:
1006 rem      out:
1020 rem    WHEN 1
1022  q9%=0
1024  data {liste 1 von ausdrücken}
1026  for i9=1 to  {m1}
1028    read  {le}
1030    if {le} = {selector} then q1%=1
1032  next i9
1034  if q9%=0 goto {zn von WHEN 2}
  .
  .
  .        Anweisungsblock 1
  .
  .
1120 rem    WHEN 2
1122  q9%=0
1124  data {liste 2 von ausdrücken}
1126  for i9=1 to  {m2}
1128    read  {le}
1130    if {le} = {selector} then q1%=1
1132  next i9
1134  if q9%=0 goto {zn von WHEN 3}
  .
  .
  .        Anweisungsblock 2
  .
  .
1220 rem    WHEN 3
  .
  .
1920 rem    WHEN n
1922  q9%=0
1924  data {liste n von ausdrücken}
1926  for i9=1 to  {mn}
1928    read  {le}
1930    if {le} = {selector} then q1%=1
1932  next i9
1934  if q9%=0 goto {zn von ENDCASE}
  .
  .
  .        Anweisungsblock n
  .
  .
2000 rem  ENDCASE
```

d)

1.3.3.9. Unterprogrammsprung

Beim Unterprogrammsprung ergeben sich keine bedeutenden Unterschiede zwischen den Sprachen. Wir fügen die Schemata der Vollständigkeit und der Kommentierung wegen bei. Im **Bild 1.11** wird das Schema für den Unterprogrammsprung gezeigt.

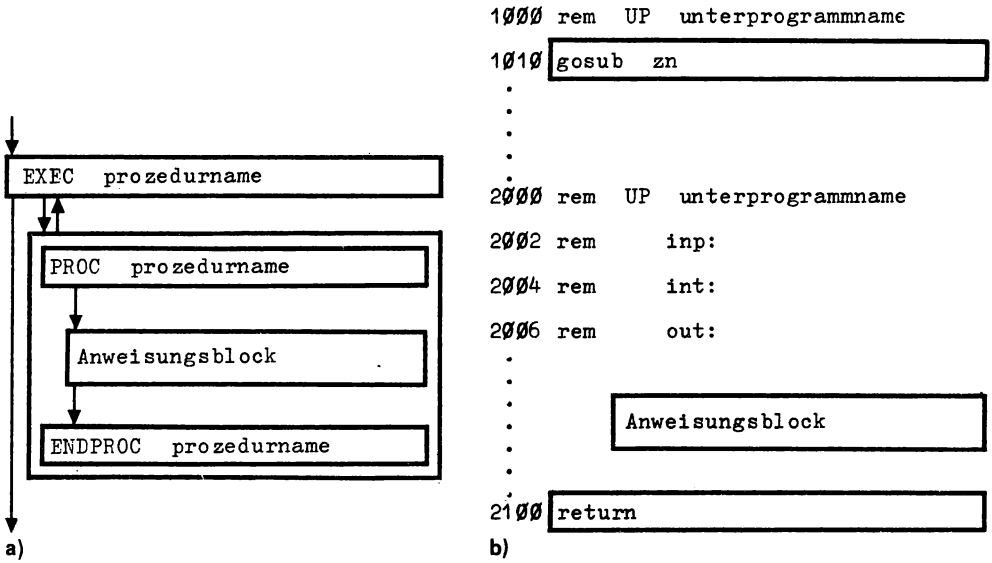


Bild 1.11. Das Prinzip der Programmierung eines Unterprogrammsprungs
zn Zeilennummer
a) Version höher entwickelter Sprachen
b) Version in BASIC

1.3.4. Aufbau der Hauptprogramme

Sobald ein Programm insgesamt länger als etwa 100 Programmzeilen ist, sollte es aus einem möglichst kurzen Hauptprogramm und mehreren Unterprogrammen bestehen. Im Hauptprogramm sollen keine rechentechnischen Kunststücke stattfinden; seine Haupttugend muß die Verständlichkeit sein. Nach einem erklärenden Programmkopf werden im Hauptprogramm zunächst alle Felder dimensioniert. Bei Programmen kleiner und mittlerer Größe folgen maschinen- und peripheriespezifische Initialisierungen, ggf. Abfragen zur Übernahme von Steuerparametern. Bei größeren Programmen kann dieser Komplex in einem eigenen Unterprogramm behandelt werden. Danach wird die Folge des Aufrufs der Unterprogramme gesteuert, wobei im Hauptprogramm einfache Strukturen, wie Sequenzen und Zählschleifen, anzustreben sind. Nach Abschluß dieses Teils werden ggf. geöffnete Datenkanäle geschlossen, und das Programm wird mit **end** beendet. Für Dialekte, die die **end**-Anweisung nur in der zahlenmäßig letzten Zeile vertragen, hat man hier sinngemäß einen Sprung zu dieser letzten Zeile zu programmieren.
Folgendes Schema könnte demnach als Richtschnur dienen:

- Programmname, ggf. kurze Funktionserklärung
- Name des Programmfile und eventueller Hilfsfiles
- Versionsnummer oder Datum der letzten Aktualisierung

```
1000 rem!-------------------------------------------------!
1002 rem!  Polynomregression V1.1                          !
1004 rem!                                                 !
1006 rem!    Disk: NUMALG 3V   File: POLREG2              !
1008 rem!    Stand: 10-Jul-87                             !
1010 rem!-------------------------------------------------!
1018 rem
1020 rem  .............  Felddimensionierung
1022 rem
1024 dim u(9,9),v(9),w(9)
1026 dim x9%(9),y9%(9)
1028 dim x(200),y(200),z(200)
1030 rem
1032 rem
1034 rem
1040 rem  ..............  Geräteeinstellung  ..............
1042 rem
1044 poke 53281,0 : poke646,1 : rem     Bildschirmfarbe
1046 poke 53272,23 : rem                Kleinschrift
1048 print chr$(147) : rem              Bildschirm löschen
1050 print "Drucker eingeschaltet ?"
1052 print "Schalter 1,5,7 on ?"
1054 print "ja/nein ?"
1056 get a1$ : if a1$="" goto 1056
1058 if a1$="j" goto 1080
1060   goto 1160
1062 rem
1064 rem
1066 rem
1080 rem                   Filename erfragen
1082 rem
1084 print "Name des Datenfile ?"
1086 input d1$
1088 rem
1090 rem
1092 rem
1100 rem                Beginn der Berechnung  .............
1102 rem
1104 gosub 29000 : rem                  DATLES
1106 gosub 27000 : rem                  MIWERT
1108 gosub 21000 : rem                  MINMAX
1110 gosub 20000 : rem                  GRANGE
1112 open 1,4,7
1114 for n=1 to 9
1116   if z0-n =1 goto 1160
1118   gosub 26000 : rem                MATGEN
1120   gosub 10500 : rem                MATINM
1122   gosub 25000 : rem                MATMUL
1124   gosub 23000 : rem                RESERR
1126   m1%=z0-n : q1=.9 : rem Parameter der t-Verteilung
1128   gosub 34100 : rem                STUQUA
1130   gosub 22000 : rem                PRIKOF
1132   gosub 39000 : rem                GRAFIK
1134 next n
1136 rem
1138 rem
1140 rem
1150 rem                    Programm beenden
1152 rem
1154 print "Fehlerzustand ";e1%
1156 close 1
1160 end
1162 rem!-------------------------------------------------!
```

Bild 1.12. Beispiel für ein Hauptprogramm

Im Hauptprogramm werden die Felder dimensioniert; ggf. werden auch einige gerätespezifische Einstellungen vorgenommen. Danach sollten nur noch Unterprogrammaufrufe in einfachen Strukturen der soeben geschilderten Art erfolgen, vorzugsweise in Sequenzen oder Zählschleifen. Im hier gezeigten Beispiel handelt es sich um ein Programm zur Polynomregression, das insgesamt etwa 500 Programmzeilen umfaßt. Dennoch ist das Hauptprogramm sehr leicht lesbar.

- Felddimensionierung
- Abfrage des Vorhandenseins und des Einschaltzustands der Peripherie
- Abfrage von Steuerparametern
- Steuerung des eigentlichen Programmablaufs, Aufruf der Unterprogramme
- offene Datenkanäle schließen
- **end**-Anweisung.

Bild 1.12 soll als Beispiel für entsprechendes Vorgehen bei einem mittelgroßen Programm dienen.

1.4. Programmdokumentation

Dokumentation und Kommentar sind überflüssiger Aufwand, wenn ein Programm an einem Tage entwickelt, benutzt und wieder gelöscht wird. Aber nur dann! Die Verwendung eines unbekannten Programms ohne Dokumentation und Kommentierung macht fast immer mehr Arbeit als eine Neuentwicklung. Man sollte solche Software nur aufheben, wenn die Aussicht auf Erhalt der Dokumentationen noch besteht; sonst betrachte man sie getrost als Müll. BASIC-Programme werden zumeist nicht für sehr lange Lebensdauer ausgelegt; deshalb wird oft weniger Aufwand für die Dokumentation getrieben. Es ist Sache der Erfahrung, das Optimum bezüglich des insgesamt geringsten Aufwands für Programmierung und Benutzung zu finden. Der Anfänger wird i. allg. die Wichtigkeit der Dokumentation unterschätzen. Beschwörungen an dieser Stelle würden nicht viel nützen. Wir werden uns kurz fassen.
Es ist eine verbreitete Arbeitsweise, zuerst das Programm zum Laufen zu bringen und, wenn alles in Ordnung ist, auch noch die Dokumentation anzufertigen. Solch ein Vorhaben ist aussichtslos. Das fertige Programm zu dokumentieren macht den meisten Programmierern nicht viel Spaß. Wenn es keinen Spaß macht, dann wird man ohne viel Überlegung Gründe finden, dafür keine Zeit zu haben. *Schulz* schreibt treffend [Schu 82]:
"Jahrelange Erfahrungen haben gezeigt, daß eine Programmdokumentation nur dann mit genügender Sorgfalt erstellt wird, wenn sie quasi als Abfallprodukt beim Programmentwurf entsteht. Jede nachträgliche Dokumentation führen die Programmierer schon aus Zeitgründen entweder gar nicht oder nur sehr oberflächlich aus. Oder anders ausgedrückt, der Programmentwurf muß so beschaffen sein, daß er selbstdokumentierend wirkt. Auf die Bedeutung der Dokumentation von Programmprodukten braucht man nicht besonders hinzuweisen. Es sei nur der Aspekt der Änderungsfreundlichkeit genannt, der inhärent mit einer guten Dokumentation verbunden ist."
Programme und Programmbausteine sollten mindestens in folgender Weise dokumentiert werden (s. z. B. [Wals 72]:

- Name des Programmfile
- Kurzbezeichnung des Unterprogramms
- Programmzeilenbereich
- Programmbeschreibung, Algorithmen, Einschränkungen, Genauigkeit
- Voraussetzungen für Ablauf (Kanäle öffnen, Dimensionierung usw.)
- relative Laufzeit
- Variablenimport vom aufrufenden Programm (Inp:)
- Variablenexport an das aufrufende Programm (Out:)
- intern benutzte Hilfsvariable (Int:)
- Erfordernisse nach dem Rücksprung (z. B. Kanäle schließen)
- Beispiel
- Literatur
- Programmlisting.

Strukturierte Programmierung erleichtert die Dokumentation, indem Programmblöcke mit Beschreibungen und Kommentaren wiederverwendet werden können. Die auf diese Weise erreichbare bessere Lesbarkeit kommt der Dokumentation entgegen.
Zur Frage der Lesbarkeit sei bemerkt, daß BASIC besonders leicht zu lesende Programme ermöglicht, weil es Elemente der normalen englischen Sprache benutzt. Man bemühe sich, die Lesbarkeit auch durch Verwendung von Elementen der deutschen Hochsprache in den Programmbeschreibungen und Dokumentationen zu unterstützen.
Um einen Überblick über die in einem Programm bereits verwendeten Variablennamen zu behalten, empfiehlt sich – sofern es sich um größere Programme handelt – die Verwendung einer Variablenliste, die man mit dem P 1.02 VALIST leicht herstellt und die bei längeren BASIC-Programmen unverzichtbarer Teil der Dokumentation wird.
Gewarnt sei vor komplizierten Programmiertricks zur Effektivitätsverbesserung oder zur Erhöhung der Eleganz. Sind solche Stellen nicht ausreichend dokumentiert, so kann man in die Lage kommen, nach Wochen das gesamte Programm noch einmal anzufertigen.
Neben der Programmdokumentation gibt es die Möglichkeit der Kommentierung unter Zuhilfenahme der **rem**-Anweisungen. Für kleine Programme kann die alleinige Verwendung dieser Art von Dokumentation ausreichend sein. Es wird empfohlen, alle Programmblöcke auf diese Weise mit einem Kopf zu versehen, der zumindest den Namen des Moduls enthält sowie die Namen der eingehenden, der intern verwendeten sowie der ausgehenden Variablen entsprechend dem folgenden Schema:

Beispiel

```
1000 rem
1002 rem     UP Polynomberechnung
1004 rem        Inp: [[variablenname]]
1006 rem        Int: [[variablenname]]
1008 rem        Out: [[variablenname]]
1020 rem
1040 . . .
```

In unserem Subset werden keine Zeilen mit Mehrfachanweisungen verwendet. Lediglich die für den Programmablauf unerheblichen Kommentare (remarks, **rem**-Anweisungen) werden zuweilen nach einer ersten Anweisung noch auf der gleichen Programmzeile untergebracht. Zwischen den beiden Anweisungen wird dann als Trennzeichen der Doppelpunkt verwendet.
Wo in besonderen Fällen aus Mangel an Platz im Programmspeicher oder wegen der Laufzeitproblematik der Komfort derartiger Kommentierung nicht möglich ist, sollte wenigstens in Kommentaren auf die entsprechenden Stellen in der Dokumentation verwiesen werden. Akzeptabel ist auch das Vorgehen, bei dem zwei Programmversionen gespeichert werden, eine mit Kommentierung für die Programmaktualisierung und eine bereinigte Version für die Ausführung der Berechnungen. Bei Verwendung von Compilern erübrigt sich solch ein Vorgehen; bei der Compilation werden ohnehin alle Kommentare eliminiert.

1.5. Fehlersuche und Fehlerbeseitigung

Anfänger auf allen Gebieten schöpferischer wissenschaftlicher oder technischer Tätigkeit unterschätzen regelmäßig den Zeitaufwand für Fehler- und Mängelbeseitigung am Produkt. Wenn eine neue Maschine oder ein neues Programm den ersten Probelauf absolviert hat, dann ist höchstens die Hälfte der notwendigen Arbeit getan. Manchmal sind es erst wenige Prozent. Der Probelauf eines Programms ist erst möglich, wenn die formalen Fehler, wie Schreibfehler und Syntaxfehler, beseitigt sind. Dieser Arbeitsgang ist relativ schnell erledigt, da BASIC solche Mängel mit Fehlermeldungen moniert. Die schwerwiegenden und schwer

zu findenden Fehler sind die logischen Fehler, die dem Programmlauf ungewollt und unbemerkt einen falschen Weg geben, oder aber Fehler in den zugrunde liegenden Algorithmen. Solche Fehler können auch durch Rechenungenauigkeiten des Computers verursacht sein; Iterationen konvergieren nicht, oder Pole und Nullstellen verursachen Komplikationen. Der Mathematiker mag solche Mängel als Ergebnis schlampiger Algorithmierung bezeichnen; für den Physiker und den Ingenieur ist indessen die Problematik oft zu komplex, als daß in der täglichen Praxis eine unanfechtbare theoretische Vorklärung aller Eventualitäten erfolgen könnte. Damit wird auch diese Problematik im Arbeitsgang Fehlersuche zu behandeln sein.
Wichtige Voraussetzungen für fehlerarme Programmierung sind ein strukturierter Aufbau und die Fertigstellung des logischen Konzepts vor Beginn der Eingabe (Kodierung) des Programms. Allerdings ist die letztgenannte Forderung nicht mehr so wörtlich zu nehmen wie zu den Zeiten, als Programme noch auf Lochkarten gestanzt wurden. Moderne Personalcomputer kann man unter bestimmten Umständen (z. B. Vorhandensein eines leistungsfähigen Texteditors) sehr wohl schon in der Phase der Vorüberlegungen einsetzen.
Man unterscheide streng zwischen Fehlerbeseitigung und Programmoptimierung und beginne keinesfalls mit Optimierungen, bevor das ursprünglich geplante Programm fehlerfrei läuft. Programmoptimierungen sollten überhaupt nur in Angriff genommen werden, wenn sie unumgänglich notwendig sind, da sie häufig Quelle neuer Programmfehler werden [Jack 79]. Vielfache Optimierungen beeinträchtigen auch die Lesbarkeit.
Bei der Fehlersuche sollten keinesfalls mehrere vermutete Fehler gleichzeitig korrigiert werden. Das würde nach mehreren derartigen Operationen zum völligen Chaos führen. Eine einzige vermutete Fehlerursache muß durch Korrektur versuchsweise beseitigt werden, und diese Korrektur sollte z. B. durch eingefügtes **rem******* gekennzeichnet werden. Durch einen Probelauf ist zu testen, ob sich die Vermutung bestätigt. Danach ist über den Verbleib der Korrektur zu entscheiden, bevor eine weitere Korrektur vorgenommen wird.
Eine große Gefahr besteht ferner darin, daß beim Verbessern eines Fehlers ein neuer Fehler erzeugt wird (z. B. durch Umbenennen von Variablen). Nach kritischen Änderungen ist deshalb durch Probelauf zu testen, ob der vorher erreichte Zustand noch vorhanden ist.
Sehr häufig sind Fehler an den Schnittstellen zwischen Programmteilen oder zwischen Programm und Anwender. Bei der Entgegennahme von Daten ist es zweckmäßig, z. B. alle Daten zunächst als Strings entgegenzunehmen und eine geeignete Fehlerbehandlungsroutine anzuschließen, mit der z. B. falsche Zeichen erkannt und ggf. eliminiert werden. Das Aufwand-Nutzen-Verhältnis ist in diesem Falle vorher abzuschätzen.
Übliche Technik bei der Fehlersuche ist der Einbau von **print**- und **stop**-Anweisungen. Solche Anweisungen sind immer auffällig zu kennzeichnen, damit sie später leicht wieder eliminiert werden können, z. B.

```
print "*****", a, b, c
stop : rem *****
```

Manche Interpreter verfügen über das Kommando VARIABLE o. ä., mit dem nach **stop** alle Variablen mit ihren aktuellen Werten gelistet werden können. Wo ein solches Kommando nicht vorhanden ist, kann ein entsprechendes Unterprogramm verwendet werden, das z. B. nach **stop** mit **goto** . . . aufgerufen wird (in diesem Falle vor **return** den Befehl **stop** einfügen).
Wichtig ist, daß beim Probelauf jeder mögliche Programmzweig mindestens einmal durchlaufen wird. Bei großen Programmen kann das mit Sicherheit natürlich nicht erreicht werden, so daß die Testung auf eine statistische Testung hinausläuft. Damit ist bereits klar, daß die Aussage über die Fehlerfreiheit eines Programms nur statistischen Charakter haben kann. Garantie für völlige Fehlerfreiheit kann für große Programme nie gegeben werden.
Wo es möglich ist, sollte an Beispielen getestet werden, deren Ergebnisse zuverlässig bekannt sind. Für viele Programmodule wird sich das erreichen lassen. Auch Läufe des glei-

chen Programms auf einem anderen Rechner (z. B. mit anderer Wortlänge) können zur Fehlerfindung dienen. Taschenrechner verfügen über eine im Vergleich zu Rechnern mit einfacher Wortlänge ausgezeichnet zuverlässige Arithmetik. Ihre Benutzung für solche Testzwecke kann empfohlen werden. Ferner sei an die Möglichkeit der Simulation erinnert. Hat man z. B. den Fehlerausgleich in einem geodätischen Netz zu programmieren, dann kann man zunächst ein fiktives fehlerfreies Netz aufbauen. Danach bringt man an den Knotenkoordinaten definierte Fehlerwerte an und führt die Ausgleichung durch. Im Ergebnis müssen sich wieder die eingegebenen Fehlerwerte ergeben. Zweckmäßig sind außerdem Testungen auf das Verhalten bei Eingabe unzulässiger oder gar unsinniger Werte (Testung auf Robustheit) sowie die Eingabe von zufälligen Werten.
Die folgenden Fehlerdetails sind übrigens häufig zu beobachten:

- Es werden beim Schreiben ähnlich aussehende Zeichen verwechselt, insbesondere sind das:
  : ;
  . ,
  O 0
  l 1
- Formelausdrücke sind in Analogie zur mathematischen Schreibweise und damit fälschlich ohne Multiplikationszeichen * geschrieben.
- Oft wird die Variableninitialisierung vor Beginn eines Summationszyklus vergessen. Während sich beim ersten Durchlauf keine Komplikationen ergeben, weil der Rechner das Nullsetzen nach RUN jedenfalls ausführt, wird bei erneuter Ausführung zu der vorangegangenen Summe addiert.
- Gleiche Variablennamen sind versehentlich mehrfach benutzt.
- Schleifen werden versehentlich einmal zuwenig oder einmal zu oft abgearbeitet. Ursache dafür ist meist die fehlende Unterscheidung zwischen REPEAT-UNTIL- und WHILE-Schleifen, und darauf wiederum haben u. U. Unterschiede zwischen den BASIC-Interpretern Einfluß.
- Es wird zu Zeilen verzweigt, die nicht existieren. Das passiert leicht bei Sprüngen zu nachträglich beseitigten **rem**-Zeilen.
- Fehlende abschließende Anführungszeichen bei Strings führen i. allg. dazu, daß die " ersatzweise am Zeilenende angenommen werden. Damit werden die ggf. dem String folgenden Anweisungen als Teil des Strings interpretiert und nicht ausgeführt. Da in diesem Fall keine Fehlermeldung erfolgt, ist der Fehler schwer zu finden.
- Wenn bei Mehrfachverzweigungen keine der Bedingungen erfüllt ist, dann kann das Programm einen ungewollten Ablauf nehmen. Auch hier wird u. U. keine Fehlermeldung angezeigt.
- Bei Anweisungen mit zahlreichen Klammern ergeben sich oft Fehler. Prüfen Sie beim Lesen das Programm, indem Sie fortlaufend zählen mit + 1 für öffnende und − 1 für schließende Klammer! Das Ergebnis muß 0 sein.

Wenn eine Fehlerursache nicht gefunden wird, dann ist der Anfänger leicht geneigt, dem Rechner die Schuld "in die Chips zu schieben". Solche Vermutungen bestätigen sich aber äußerst selten. Man verwende als Neuling keine Zeit für die Bestätigung solcher Spekulationen.

1.6. Rechenzeitbedarf

BASIC als interpretierende Sprache ist ihrer Natur nach langsam: Eine Programmzeile muß bei jedem Aufruf zuerst in eine prozessorverständliche Form transformiert werden, und das benötigt viel mehr Zeit, als die eigentliche Rechenzeit ausmacht. Um so mehr ist es von

Wichtigkeit, daß lange Programme bezüglich des Zeitbedarfs optimal aufgebaut werden. Hierfür lassen sich einige Regeln angeben. Die angegebenen Exekutionszeiten für die Beispiele sind mit dem RT-11-BASIC bei 7 Bytes Mantissenlänge bestimmt.

- Addition ist schneller als Multiplikation; Multiplikation ist schneller als Division und als Potenzbildung. Langsamere Operationen sollten demnach, wo möglich, ersetzt werden.

```
z = x+y          2.4 ms
z = x*y          7.5 ms
z = x/y          8.1 ms
z = x↑2         15.0 ms
z = x*x          7.5 ms,
```

  Der Variablenaufruf allein:

```
z = y            1.2 ms
```

  Der Zeitbedarf hängt sehr von der Größe der Operanden ab. In den Beispielen wurde verwendet:

```
x = 1.771        y = 2.17
```

- Die Übernahme von Konstanten aus Programmzeilen ist zeitaufwendig. Man ordne Konstanten vor der Verwendung in vielfach zu durchlaufenden Schleifen entsprechenden Variablen zu.

```
320 z = 1.771*2.17     8.4 ms
320 z = x*y            7.5 ms
```

- In häufig durchlaufenen Schleifen sind nur die unbedingt notwendigen Rechenoperationen durchzuführen.

```
10 for i = 1 to 1000
20 y = sin(10*pi/180)+sqr(i)
30 next i
```

  Günstiger ist:

```
10 z = sin(10*pi/180)
20 for i = 1 to 1000
30 y = z+sqr(i)
40 next i
```

- Bei klein bleibenden Argumenten ersetzt man Funktionen zweckmäßig durch die ersten Glieder ihrer Reihenentwicklung.

```
y = sin(x)          78.0 ms
y = x                1.2 ms
y = x-x*x*x/6       24.3 ms
```

- Bei **goto** und **gosub** zu höheren Zeilennummern sucht der Rechner in aufsteigender Folge die Zeilennummern bis zur Zieladresse durch. Das Auffinden weit entfernter Zeilennummern dauert entsprechend lange. Beim Sprung zu kleineren Zeilennummern wird vom Programmanfang an gesucht. Der Sprung zu der rückwärtig benachbarten Zeilennummer ist demnach die zeitaufwendigste Lösung. In entsprechend gelagerten Fällen kann die Verwendung von Schleifen vorteilhaft sein; die Positionen der **for**-Anweisungen werden nicht in der geschilderten Weise gesucht, sondern für die Dauer der Schleifenöffnung in einem speziellen Speicher bereitgehalten. Aus diesem Grunde sind häufig benötigte Unterprogramme entweder vorwärts möglichst benachbart oder rückwärts am Programmanfang zu positionieren.

- Es ist bereits erwähnt worden, daß Variablennamen bestimmten Speicherplätzen im Rechner entsprechen. Die richtige Zuordnung von Variablennamen und Speicherplatz wird durch eine Symboltabelle gewährleistet, in die der Interpreter die Variablennamen in der Reihenfolge ihres Auftretens bei der Programmabarbeitung einträgt. Für spät eingetragene Variable braucht der Interpreter länger, sie bei erneutem Aufruf zu finden. Sehr häufig gebrauchte Variable sind daher am Programmanfang bereits einmal zu erwähnen, z. B. indem man ihnen den Wert null zuweist, und man sollte, wenn es auf geringen Zeitbedarf ankommt, nicht mehr gebrauchte Variable erneut einsetzen, statt immer wieder neue Variablennamen zu kreieren.
- Zeichenketten werden im Stringspeicherbereich der Reihe nach abgelegt. Wird eine Stringvariable vom Programm verändert, dann wird grundsätzlich der ursprüngliche Speicherplatz aufgegeben, und die veränderte Stringvariable wird unter Beibehaltung ihres Namens an das momentane Ende des Stringspeicherbereichs geschrieben. Das kann dazu führen, daß der Stringspeicherbereich trotz an sich gähnender Leere als voll erkannt wird, und das veranlaßt das Betriebssystem, den gesamten Stringspeicherbereich aufzuräumen und wieder dicht zu packen, also eine "Müllsammlung" (garbage collection) auszuführen. Dies verbraucht sehr viel Rechenzeit, und man muß sorgen, daß die „garbage collection" kaum nötig wird, weil man Strings selten verändert. In besonders gelagerten Fällen kann es zweckmäßig sein, die Strings durch Integerzahlen auszudrücken, was aber einen besonderen Vorteil des BASIC, die sehr komfortable Stringbehandlung, ganz zunichte macht.
- **print**-Anweisungen sind ebenso wie der Verkehr mit anderen Peripheriegeräten vergleichsweise sehr zeitaufwendige Aktionen. Nach Abschluß der Fehlersuche sollten nicht unbedingt erforderliche Ausgaben von Zwischenergebnissen aus dem Programm beseitigt werden.

Naturgemäß sind Rechengeschwindigkeit, Speicherplatzbedarf und Übersichtlichkeit Programmeigenschaften, die sich z. T. gegenseitig ausschließen. Der zweckmäßige Kompromiß hängt von der Art der Aufgabe ab.

Zeitbedarfsmessungen führt man aus, indem man die zu untersuchende Operationsfolge in eine Schleife schreibt und vielfach durchlaufen läßt. Die Differenz aus dem Zeitbedarf für die Schleife mit und ohne die Operationenfolge geteilt durch die Zahl der Schleifendurchläufe liefert das gewünschte Ergebnis. Das **Programm P1.04 ZEITMS** arbeitet in der geschilderten Weise. Für Vergleiche der Geschwindigkeit verschiedener Rechner und BASIC-Versionen sind im Anhang 3 eine Reihe von Zeiten für die Ausführung elementarer Funktionen und Anweisungen angegeben. Für Messungen am eigenen Rechner sind die entsprechenden Anweisungen in die o. g. Routine einzusetzen. Wo die Funktion **ti** nicht zur Verfügung steht, kann unter Zuhilfenahme einer Stoppuhr gemessen werden. Statt der **ti**-Zuweisung schreibt man bei Schleifenanfang und -ende eine Meldung auf den Bildschirm oder läßt ein akustisches Signal ertönen.

Ähnliche Informationen liefert das im Abschnitt 1.2 angegebene Testprogramm nach *Savage*, dessen Ergebnisse im Anhang 4 zusammengestellt sind.

Manchmal wird es möglich sein, innerlich baugleiche Rechner verschiedenen Fabrikats durch derartige Tests als solche zu erkennen, und bei lediglich verschiedenen Prozessortaktzeiten kann noch immer aus der Gleichheit der Zeitbedarfsverläufe auf entsprechende Übereinstimmungen geschlossen werden.

Zum Abschluß der Ausführungen über den Rechenzeitbedarf ist darauf hinzuweisen, daß es für eine Reihe von Rechnern BASIC-Compiler gibt, mit denen BASIC-Programme in maschinennahe Form umgewandelt werden können. Dadurch ist ein erheblicher Geschwindigkeitsgewinn möglich, und es vereint sich der Vorteil der im BASIC einfachen Fehlersuche mit dem Geschwindigkeitsvorteil compilierender Sprachen. Der volle Nutzen wird sich meist erst bei längeren Programmen zeigen; kleine Programme werden durch die Compilation relativ stark mit organisatorischen Anweisungen belastet.

Programm P 1.04. ZEITMS. Routine zur Zeitbedarfsmessung an Programmen, Funktionen und Anweisungen.

```
1000 rem !-----------------------------------------------!
1002 rem ! P1.04  ZEITMS                                 !
1004 rem !-----------------------------------------------!
1006 s1=3333
1008 t1=ti
1010 for i=1 to s1
1012 next i
1014 t2=ti
1016 for i=1 to s1
1018   rem !-----------------------------------------------!
1020   rem ! zu untersuchender Programmteil               !
1030   rem !-----------------------------------------------!
1032 next i
1034 t3=ti
1036 print "Zeitbedarf";(t3-t2-t2+t1)*1000/s1;"Millisek."
1038 end
```

Die Differenz aus Zeitbedarf für Schleife mit Prüfling und für leere Schleife dividiert durch die Anzahl der Schleifendurchläufe ergibt den Zeitbedarf für die Ausführung des zu prüfenden Programmteils.

1.7. Speicherplatzbedarf

Für die Bestimmung des Speicherplatzes wurden bereits im Abschnitt 1.1 einige Informationen gegeben. Hier folgt noch eine Liste von Maßnahmen, die zur Komprimierung (crunching) von Programmen geeignet sind. Die Wirksamkeit der Maßnahmen ist z. T. rechnerspezifisch sehr unterschiedlich.

- Optionale Anweisungen, wie **rem**, **let**, **then goto** usw., beseitigen. Da durch die Beseitigung speziell der **rem**-Anweisungen die Übersichtlichkeit des Programms leiden wird, sollte das Programm vorher gelistet oder in der originalen Form abgespeichert werden.
- Lange Texte in **print**-Anweisungen beseitigen oder verkürzen.
- Mehrfachanweisungszeilen anwenden, sofern das der verwendete Rechner ermöglicht.
- Mehrfachverzweigungen, wie **on** a **goto** . . . und **on** a **gosub** . . ., verwenden, sofern das möglich ist. In manchen Fällen bringt diese Maßnahme zusätzlich einen Geschwindigkeitsgewinn.
- Programmzeilennummern verkleinern, da im allgemeinen für jede Ziffer ein Byte verbraucht wird.
- Mehrfach benötigte Konstanten in Variablen ablegen.
- Nicht benötigte Leerzeichen eliminieren.
- Ersatz von Real- und ggf. Stringvariablen und -feldern durch Integervariable und -felder.
- Variable und Felder neu belegen, wenn sie für ihren anfänglichen Zweck nicht mehr benötigt werden.
- Mit **dim**-Anweisungen dimensionierte Felder auf ihre notwendige Größe hin untersuchen. Prüfen Sie, ob alle Feldvariablen als solche nötig sind oder ob z. B. durch sequentielle vollständige Abarbeitung auch mit einfachen Variablen gearbeitet werden kann. Bei der Berechnung der Standardabweichung ist es aus der Sicht der Speicherplatzoptimalität unzweckmäßig, erst den Mittelwert und dann die Summe aus Meßwerten minus Mittelwert zu bilden; die Berechnung aus der Summe der Meßwerte und der Summe der Meßwertquadrate erübrigt die Abspeicherung des gesamten Meßwertkollektivs. Zweckmäßig wer-

den alle Felder dimensioniert, auch kleine, die von vielen Rechnern automatisch dimensioniert würden. Für ein im Programmtext nicht dimensioniertes zweidimensionales Feld werden sofort 121 Plätze reserviert; das ist häufig viel mehr als gebraucht wird.

- Daten nicht in **data**-Feldern oder in Feldvariablen abspeichern, sondern aus Massenspeichern übernehmen und sofort sequentiell verarbeiten.
- Daten mit gesonderten Programmen vorverdichten.
- Schleifen und Unterprogramme effektiv anwenden und dadurch mehrfach auftretende gleiche oder ähnliche Programmteile eliminieren.
- Benutzerdefinierte Funktionen anwenden. Diese zumeist mit FN . . . bezeichneten Funktionen sind im Basicode- und im Tiny-Vokabular nicht enthalten und werden hier auch sonst nicht verwendet.
- Ersatz großer Felder im Rechnerspeicher durch virtuelle Dateien auf dem Massenspeicher.
- Reduktion der Anzahl gleichzeitig geöffneter Dateien.
- Das Programm in kleinere Teile aufteilen und mit OVERLAY, CHAIN o. ä. jeweils erst bei Gebrauch vom Massenspeicher einlesen.
- Das Programm in Teile aufteilen, die jeweils zu Zwischenergebnissen mit geringen Datenraten führen. Die anfallenden Daten sind zwischenzuspeichern.

2. Lineare Algebra und Matrizenrechnung

2.1. Matrizenoperationen

Eine Matrix habe m Zeilen und n Spalten. Jedes Element soll – an diese Konvention wollen wir uns im folgenden halten – zuerst durch den Zeilen- und dann durch den Spaltenindex gekennzeichnet sein. Bei Schleifendurchläufen bezeichnen wir i. allg. mit i9 die Zeilennummer und mit j9 die Spaltennummer.

$$\begin{array}{l} \\ i9=1 \\ i9=2 \\ \\ \\ i9=m \end{array} \begin{array}{c} \begin{array}{cccc} j9=1 & j9=2 & j9=3 & j9=n \end{array} \\ \begin{pmatrix} u(1,1) & u(1,2) & u(1,3) & u(1,n) \\ u(2,1) & u(2,2) & u(2,3) & u(1,n) \\ & & & \\ & & & \\ u(m,1) & u(m,2) & u(m,3) & u(m,n) \end{pmatrix} \end{array} \tag{2.1}$$

Wir setzen den Beginn der Zählung für die Felddimensionierung mit 1 an. Das ermöglicht die beste Übersichtlichkeit in den Programmen. Die meisten Rechner beginnen die Zählung bei 0. In diesem Falle wird mit unserer Methode Speicherplatz verschenkt. Ist das unerträglich, so erniedrige man in den Programmen alle Indizes um 1. Bessere BASIC-Versionen verfügen über die Anweisung "OPTION BASE . . .". Hier würde für die Verwendung der Programme in der angegebenen Form (außer bei den Bandmatrizen) "OPTION BASE 1" zu geben sein.
Will oder muß man die Matrix in einem eindimensionalen Feld unterbringen, so wollen wir bis auf Ausnahmen bei schwach besetzten Matrizen, auf die in den Abschnitten 2.3.6 bis 2.3.9 hingewiesen wird, einheitlich spaltenweise vorgehen. Rechteckige Matrizen sind demnach folgendermaßen zu indizieren:

$$\begin{pmatrix} u(1) & u(m+1) & u(2*m+1) & u[(n-1)*m+1] \\ u(2) & u(m+2) & u(2*m+2) & u[(n-1)*m+2] \\ & & & \\ & & & \\ u(m) & u(2*m) & u(3*m) & u(n*m) \end{pmatrix} \tag{2.2}$$

Der Zusammenhang zwischen dem eindimensionalen Index k und den zweidimensionalen i und j gemäß der Anordnung in (2.1) ist demnach

$$k = i + (j-1)*m \tag{2.3}$$

Nachfolgend werden Programme für die Operationen Addition, Subtraktion, Skalarmultiplikation, Kopieren, Drucken, Transponieren und Multiplikation angegeben. Aus Gründen des

geringen Platzverbrauchs sind die sehr einfachen Operationen hier in CASE-Konstrukten konzentriert. Die Programme sind für ein- und zweidimensionale Abspeicherung der Matrizen angegeben.

2.1.1. Matrizenaddition

Berechnung mit dem **Programm P 2.01 MATOP2** oder **P 2.02 MATOP1**. Steuerflag c1%=1 bewirkt, daß die Summe der Matrizen U_1 und U_2 in der Matrix U_3 abgelegt wird. c1%=3 bewirkt, daß Matrix U_1 mit der Summe $U_1 + U_2$ überschrieben wird.

Programm P 2.01. MATOP2. Führt in Abhängigkeit vom Wert eines Steuerflags die Operationen Matrizenaddition und -subtraktion, Skalarmultiplikation, Kopieren, Drucken und Transponieren aus.

```
10000 rem !-----------------------------------------------!
10002 rem ! P2.01  MATOP2                                 !
10004 rem !        inp:a,c1%,m,n,u1(n,m),u2(n,m),u3(n,m)  !
10006 rem !        int:i9,j9,s9,t9                        !
10008 rem !        out:u3(n,m),(( u1(n,m),u2(n,m) ))      !
10010 rem !-----------------------------------------------!
10012   for i9=1 to m
10014     for j9=1 to n
10016       on c1% goto 10018,10022,10026,10036,10040,
            10044,10048
10018       u3(i9,j9)=u1(i9,j9)+u2(i9,j9)                 ! addieren
10020       goto 10050
10022       u3(i9,j9)=u1(i9,j9)-u2(i9,j9)                 ! subtrahieren
10024       goto 10050
10026       s9=u1(i9,j9)+u2(i9,j9)                        ! Addition und
10028       t9=u1(i9,j9)-u2(i9,j9)                        ! Subtraktion
10030       u1(i9,j9)=s9
10032       u2(i9,j9)=t9
10034       goto 10050
10036       u1(i9,j9)=a*u1(i9,j9)                         ! Skalarmultipli-
10038       goto 10050                                    ! kation
10040       u3(i9,j9)=u1(i9,j9)                           ! kopieren
10042       goto 10050
10044       print u1(i9,j9);                              ! drucken
10046       goto 10050
10048       u3(i9,j9)=u1(j9,i9)                           ! transponieren
10050     next j9
10052     if c1%=6 then print
10054   next i9
10056   if c1%<>7 goto 10064
10058     s9=m
10060     m=n
10062     n=s9
10064 return
10066 rem !-----------------------------------------------!
```

Das Programm setzt voraus, daß alle beteiligten Matrizen die gleiche Dimension haben.

Bedeutung der Variablennamen:

u1(n,m), u2(n,m), u3(n,m)=Matrizen. u3(n,m) ist i. allg. die Ergebnismatrix. m Zeilenzahl; n Spaltenzahl; c1% Steuerflag.

c1%=1 realisiert Addition von $U_3 = U_1 + U_2$

c1%=2 realisiert Subtraktion $U_3 = U_1 - U_2$

c1%=3 realisiert Addition und Subtraktion und überschreibt die ursprünglichen Matrizen, also $U_1 = U_1 + U_2$ und $U_2 = U_1 - U_2$

c1%=4 führt die Skalarmultiplikation $U_1 = a * U_1$ aus
c1%=5 kopiert die Matrix U_1 nach U_3
c1%=6 druckt die Matrix U_1 zeilenweise, führt am Zeilenende Return aus
c1%=7 transponiert die Matrix U_1 und legt das Ergebnis in U_3 ab, also $U_3 = U_1^T$. Vertauscht anschließend m und n.

Programm P 2.02. MATOP1. Führt in Abhängigkeit vom Wert eines Steuerflags die Operationen Matrizenaddition und -subtraktion, Skalarmultiplikation, Kopieren, Drucken und Transponieren aus.

```
10070 rem !-----------------------------------------------!
10072 rem ! P2.02  MATOP1                                 !
10074 rem !        inp:a,c1%,m,n,u1(n*m),u2(n*m),u3(n*m)  !
10076 rem !        int:i9,j9,l9,s9,t9                     !
10078 rem !        out:u3(n*m),(( u1(n*m),u2(n*m) ))      !
10080 rem !-----------------------------------------------!
10082   for i9=1 to m*n
10084     on c1% goto 10086,10090,10094,10104,10108,10112,
          10120
10086     u3(i9)=u1(i9)+u2(i9)                            ! addieren
10088     goto 10126
10090     u3(i9)=u1(i9)-u2(i9)                            ! subtrahieren
10092     goto 10126
10094     s9=u1(i9)+u2(i9)                                ! Addition und
10096     t9=u1(i9)-u2(i9)                                ! Subtraktion
10098     u1(i9)=s9
10100     u2(i9)=t9
10102     goto 10126
10104     u1(i9)=a*u1(i9)                                 ! Skalarmultipli-
10106     goto 10126                                      ! kation
10108     u3(i9)=u1(i9)
10110     goto 10126
10112     l9=int((i9-1)/n)*(1-n*m)+m*i9-m+1
10114     print u3(l9);                                   ! drucken
10116     if i9/n=int(i9/n) then print
10118     goto 10126
10120     l9=int((i9-1)/n)*(1-n*m)+m*i9-m+1
10122     u3(i9)=u1(l9)                                   ! transponieren
10124     goto 10126
10126   next i9
10128   if c1%<>7 goto 10136
10130     s9=m                                            ! Rücktaussch m,n
10132     m=n
10134     n=s9
10136 return
10138 rem !-----------------------------------------------!
```

Das Programm setzt voraus, daß alle beteiligten Matrizen die gleiche Dimension haben und in eindimensionalen Feldern spaltenweise abgespeichert sind.

Bedeutung der Variablennamen:
u1(m*n), u2(m*n), u3(m*n)=Matrizen. u3(m*n) ist i. allg. die Ergebnismatrix. m Zeilenzahl; n Spaltenzahl; c1% Steuerflag. Funktionen in Abhängigkeit vom Wert des Steuerflags s. Bildunterschrift zu P 2.01.

2.1.2. Matrizensubtraktion

Berechnung mit P 2.01 MATOP2 oder P 2.02 MATOP1. Steuerflag c1%=2 bewirkt, daß die Differenz $U_1 - U_2$ in U_3 abgelegt wird. c1%=3 bewirkt, daß U_2 mit der Differenz $U_1 - U_2$ überschrieben wird.

2.1.3. Skalarmultiplikation

Berechnung mit P 2.01 MATOP2 oder P 2.02 MATOP1. Steuerflag c1%=4 bewirkt, daß Matrix U_1 mit dem Faktor a skalar multipliziert wird.

2.1.4. Kopieren

Realisierung mit P 2.01 MATOP2 oder P 2.02 MATOP1. Steuerflag c1%=5 bewirkt, daß Matrix U_1 in die Matrix U_2 kopiert wird.

2.1.5. Drucken

Realisierung mit P 2.01 MATOP2 oder P 2.02 MATOP1. Steuerflag c1%=6 bewirkt, daß die Matrix in zweidimensionaler Form gedruckt wird. Die Formatierung und die Sicherung gegen unerwünschte Zeilenüberläufe sind rechnerspezifisch hinzuzufügen.

Programm P 2.03. MATTR2. Transponiert eine Matrix innerhalb eines quadratischen Feldes und setzt den nicht von der transponierten Matrix belegten Feldbereich null.

```
10142 rem !-----------------------------------------------!
10144 rem ! P2.03  MATTR2                                 !
10146 rem !        inp:m,n,u(m,n)                         !
10148 rem !        int:i9,j9,p9,s9                        !
10150 rem !        out:u(n,m)                             !
10152 rem !-----------------------------------------------!
10154   p9=m                                              ! Suche nach größter
10156   if n>m then p9=n                                  ! Dimension
10158   for i9=1 to p9                                    ! transponieren
10159     if i9+1>p9 goto 10170
10160     for j9=i9+1 to p9
10162       s9=u(i9,j9)
10164       u(i9,j9)=u(j9,i9)
10166       u(j9,i9)=s9
10168     next j9
10170   next i9
10172   for i9=1 to p9                                    ! Nullen des freien
10174     for j9=1 to p9                                  ! Bereichs
10176       if i9<=n then if j9<=m goto 10180
10178       u(i9,j9)=0
10180     next j9
10182   next i9
10184   s9=m                                              ! Tausch von n und m
10186   m=n
10188   n=s9
10190 return
10192 rem !-----------------------------------------------!
```

Es wird ein zweidimensionales Feld benutzt. m und n werden im Anschluß an die Operation vertauscht. Das Programm setzt voraus, daß die erste i9-Schleife abweisend ist, d. h. nicht durchlaufen wird, wenn der Anfangswert größer als der Endwert ist. Ggf. ist eine bedingte Sprunganweisung vorzuschalten.

Bedeutung der Variablennamen:
m Zeilenzahl der Originalmatrix; n Spaltenzahl der Originalmatrix; p9=sup(m,n) Dimension des benötigten quadratischen Speicherbereichs; u(p9,p9) benötigter Speicherbereich; u(m,n) Originalmatrix; u(n,m) transponierte Matrix.

2.1.6. Transponieren

Mit P 2.01 MATOP2 oder P 2.02 MATOP1 kann die transponierte Matrix $\boldsymbol{U}_1^T$ gebildet und in $\boldsymbol{U}_3$ abgelegt werden. Die transponierte Matrix ist in umgekehrter Reihenfolge zu dimensionieren. Ist also **dim** u1(n,m), so muß **dim** u2(m,n) gewählt werden. Soll die ursprüngliche Matrix mit der transponierten überschrieben werden, so ist das Feld, in der die beiden Matrizen nacheinander gespeichert werden, mit dim u1(p,p) als quadratisches Feld zu dimensionieren, wobei p=sup(n,m) also der größere der beiden Werte n,m ist. Die **Programme P 2.03 MATTR2** und **P 2.04 MATTR1** führen diese Operation aus. MATTR2 füllt die nicht benötigten Elemente der p*p-Matrix mit Nullen auf. Nach Ausführung der Operation werden n und m bezüglich ihres Wertes vertauscht.
In der Praxis ist zu überlegen, ob die Ausführung der Operation nicht auch durch Indexrechnung in einer vor- oder nachgelagerten Routine möglich ist.

Programm P 2.04. MATTR1. Transponiert eine Matrix innerhalb ein und desselben Feldbereichs.

```
10200 rem !-----------------------------------------------!
10202 rem ! P2.04   MATTR1                                !
10204 rem !         inp:m,n,u(m*n)                        !
10206 rem !         int:i9,j9,s9                          !
10208 rem !         out:u(n*m)                            !
10210 rem !-----------------------------------------------!
10212   for i9=1 to m
10214     for j9=i9+1 to n
10216       s9=u((j9-1)*m+i9)
10218       u((j9-1)*m+i9)=u((i9-1)*m+i9)
10220       u((i9-1)*m+j9)=s9
10222     next j9
10224   next i9
10226   s9=m                                    ! Tausch von n und m
10228   m=n
10230   n=s9
10232 return
10234 rem !-----------------------------------------------!
```

Es wird ein eindimensionales Feld benutzt. m und n werden nach Abschluß der Operation vertauscht.

Bedeutung der Variablennamen:
m Zeilenzahl der Originalmatrix bzw. Spaltenzahl der transponierten Matrix; n Spaltenzahl der Originalmatrix bzw. Zeilenzahl der transponierten Matrix; u(m*n) Matrix.

2.1.7. Matrizenmultiplikation

Die Multiplikation setzt voraus, daß die Anzahl der Spalten in der ersten Matrix mit der Anzahl der Zeilen in der zweiten Matrix übereinstimmt. Die Dimension der Produktmatrix ist:
Zeilenzahl = Zeilenzahl der ersten Matrix
Spaltenzahl = Spaltenzahl der zweiten Matrix **(Bild 2.1)**.

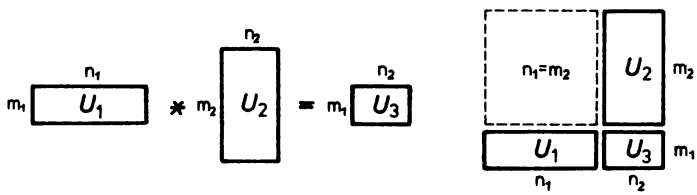


Bild 2.1. Schema der Matrizenmultiplikation
Rechts das Schema von *Falk*, aus dem die Bedingungen für die Multiplizierbarkeit anschaulich hervorgehen. Die gestrichelte Fläche muß quadratisch sein.

Die Elemente der Produktmatrix werden nach folgender Formel gebildet:

$$w_{ij} = u_{i1}v_{1j} + u_{i2}v_{2j} + \ldots + u_{in_1}v_{m_2j} \tag{2.4}$$

Das Matrizenprodukt hängt von der Reihenfolge der zu multiplizierenden Matrizen ab. Auch bei quadratischen Matrizen ist i. allg. die Produktmatrix von der Multiplikationsreihenfolge abhängig. Vereinfachungen würden sich bei Diagonalmatrizen ergeben: Das Produkt zweier

Programm P 2.05. MATPR2. Bildet das Matrizenprodukt.

```
10240 rem !-------------------------------------------------!
10242 rem ! P2.05  MATPR2                                   !
10244 rem !        inp:m1,m2,n1,n2,u1(m1,n1),u2(m2,n2)      !
10246 rem !        int:i9,j9,k9                             !
10248 rem !        out:e1%,u3(m1,n2)                        !
10250 rem !-------------------------------------------------!
10252   e1%=0
10254   if n1<>m2 goto 10274
10256   for i9=1 to m1
10258     for j9=1 to n2
10260       u3(i9,j9)=0
10262       for k9=1 to n1
10264         u3(i9,j9)=u3(i9,j9)+u1(i9,k9)*u2(k9,j9)
10266       next k9
10268     next j9
10270   next i9
10272   goto 10276
10274     e1%=1
10276 return
10278 rem !-------------------------------------------------!
```

Die drei Matrizen befinden sich in zweidimensionalen Feldern.

Bedeutung der Variablennamen:
u1(m1,n1) erste Faktormatrix; u2(m2,n2) zweite Faktormatrix; u3(m1,n2) Produktmatrix; m1,m2 Zeilenzahl der ersten bzw. zweiten Faktormatrix; n1,n2 Spaltenzahl der ersten bzw. zweiten Faktormatrix; e1% Fehlerflag; e1%=1 bedeutet n1 ungleich m2, d. h. nicht verkettete Faktormatrizen.

Diagonalmatrizen ist wieder eine Diagonalmatrix. Dieser Spezialfall ist in den folgenden Programmen jedoch nicht als Vereinfachung besonders berücksichtigt. Das **Programm P 2.05 MATPR2** gibt das Matrizenprodukt für den Fall zweidimensionaler Abspeicherung, **P 2.06 MATPR1** für den Fall eindimensionaler Abspeicherung der Matrix, **P 2.07 MATPRK** multipliziert Matrizen mit komplexen Elementen. Dabei sind die komplexen und die imaginären Anteile der Elemente in getrennten zweidimensionalen Feldern gespeichert. Die folgenden Beispiele können zur Programmtestung dienen:

m1=2 n1=3 m2=3 n2=4

$$\begin{pmatrix} 1 & 2 & -3 \\ -4 & 5 & 6 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 5 & 0 \\ 0 & 4 & -2 & 0 \\ 2 & 3 & 7 & 1 \end{pmatrix} = \begin{pmatrix} -5 & 0 & -20 & -3 \\ 8 & 34 & 12 & 6 \end{pmatrix}$$

m1=2 n1=2 m2=2 n2=3

$$\begin{pmatrix} 1-6i & 2+5i \\ -4-3i & 5+2i \end{pmatrix} * \begin{pmatrix} 1+2i & 1-2i & 5-i \\ 0+4i & 4+i & -2+3i \end{pmatrix} = \begin{pmatrix} -31+16i & 16+26i & -8+25i \\ -6+9i & 8+18i & -39+0i \end{pmatrix}$$

Programm P 2.06. MATPR1. Bildet das Matrizenprodukt.

```
10282 rem !-----------------------------------------------!
10284 rem ! P2.06  MATPR1                                 !
10286 rem !        inp:m1,m2,n1,n2,u1(m1*n1),u2(m2*n2)    !
10288 rem !        int:i9,j9,k9,l9                        !
10290 rem !        out:e1%,u3(m1*n2)                      !
10292 rem !-----------------------------------------------!
10294   e1%=0
10296   if n1<>m2 goto 10318
10298   for i9=1 to m1
10300     for j9=1 to n2
10302       l9=i9+(j9-1)*m1
10304       u3(l9)=0
10306       for k9=1 to n1
10308         u3(l9)=u3(l9)+u1(i9+(k9-1)*m1)*u2(k9+(j9-1)*m2)
10310       next k9
10312     next j9
10314   next i9
10316   goto 10320
10318     e1%=1
10320 return
10322 rem !-----------------------------------------------!
```

Die drei Matrizen befinden sich in eindimensionalen Feldern.

Bedeutung der Variablennamen:
u1(m1,n1) erste Faktormatrix; u2(m2,n2) zweite Faktormatrix; u3(m1,n2) Produktmatrix; m1,m2 Spaltenzahl der ersten bzw. zweiten Faktormatrix; e1% Fehlerflag; e1%=1 bedeutet n1 ungleich m2, d. h. nicht verkettete Faktormatrizen.

Programm P 2.07. MATPRK. Bildet das Produkt zweier Matrizen mit komplexen Elementen.

```
10390 rem !-----------------------------------------------!
10392 rem ! P2.07  MATPRK                                 !
10394 rem !        inp:m1,m2,n1,n2,u1(m1,n1),u2(m2,n2),   !
      !            v1(m1,n1),v2(m2,n2)                   !
10396 rem !        int:i9,j9,k9                           !
10398 rem !        out:e1%,u3(m1,n2),v3(m1,n2)            !
10400 rem !-----------------------------------------------!
10402   e1%=0
10404   if n1<>m2 goto 10428
10406   for i9=1 to m1
10408     for j9=1 to n2
10410       u3(i9,j9)=0
10412       v3(i9,j9)=0
10414       for k9=1 to n1
10416         u3(i9,j9)=u3(i9,j9)+u1(i9,k9)*u2(k9,j9)-
              -v1(i9,k9)*v2(k9,j9)
10418         v3(i9,j9)=v3(i9,j9)+u1(i9,k9)*v2(k9,j9)+
              +v1(i9,k9)*u2(k9,j9)
10420       next k9
10422     next j9
10424   next i9
10426   goto 10430
10428     e1%=1
10430 return
10432 rem !-----------------------------------------------!
```

Real- und Imaginärteile sind in getrennten zweidimensionalen Feldern gespeichert.

Bedeutung der Variablennamen:
u1(m1,n1), v1(m1,n1) Real- und Imaginärteil der ersten Faktormatrix; u2(m2,n2), v2(m2,n2) Real- und Imaginärteil der zweiten Faktormatrix; u3(m1,n2), v3(m1,n2) Real- und Imaginärteil der Produktmatrix; m1, m2 Zeilenzahl der ersten bzw. zweiten Faktormatrix; n1,n2 Spaltenzahl der ersten bzw. zweiten Faktormatrix; e1% Fehlerflag; e1%=1 bedeutet n1 ungleich m2, d. h. nicht verkettete Faktormatrizen.

2.1.8. Matrixinversion

Ziel der Matrixinversion ist die Berechnung der zu einer gegebenen quadratischen Matrix $\boldsymbol{U}$ inversen Matrix $\boldsymbol{U}^{-1}$, so daß $\boldsymbol{U} \cdot \boldsymbol{U}^{-1} = \boldsymbol{E}$ gilt. Dabei ist $\boldsymbol{E}$ die Einheitsmatrix, eine nur mit dem Wert 1 auf der Hauptdiagonalen besetzte Matrix. Als Methoden zur praktischen Berechnung eignen sich die für die Lösung von Gleichungssystemen üblichen, also z. B. Gauß-Verfahren, Crout-Verfahren und Gauß-Jordan-Verfahren; u. a. bei *Gastinel* [Gast 72] sind diese Verfahren beschrieben und die Anzahl notwendiger Rechenoperationen gegenübergestellt. Es zeigt sich, daß die Unterschiede bezüglich der Operationenzahl nicht erheblich sind. Wir beschreiben hier das sog. Austauschverfahren, das in der dargelegten Form dem Gauß-Jordan-Verfahren sehr ähnlich ist.

Neben die zu invertierende Matrix wird – das muß im Rechner nicht notwendig so ausgeführt werden – eine Einheitsmatrix $\boldsymbol{E}$ gleicher Größe geschrieben. Die Matrix $\boldsymbol{U}$ wird wie beim Gauß-Jordan-Verfahren allmählich in die Einheitsmatrix übergeführt. Die dazu notwendigen Operationen werden in entsprechender Weise auf beide Matrizen angewendet.

Nach Ausführung aller Rechenschritte ist aus der Einheitsmatrix die invertierte Matrix $\boldsymbol{U}^{-1}$ geworden und aus der ursprünglichen Matrix $\boldsymbol{U}$ eine Einheitsmatrix. Die Inversion einer 3*3-Matrix gestaltet sich folgendermaßen:

- Matrix und entsprechende Einheitsmatrix bereitstellen:

$$\left(\begin{array}{ccc|ccc} u_{11} & u_{12} & u_{13} & 1 & 0 & 0 \\ u_{21} & u_{22} & u_{23} & 0 & 1 & 0 \\ u_{31} & u_{32} & u_{33} & 0 & 0 & 1 \end{array}\right)$$

- u_{11} wird als sog. Pivotelement verwendet. Die Pivotierung ist ausführlicher im Abschn. 2.3.2. erläutert. Die erste Zeile (Pivotzeile) beider Matrizen wird durch das Pivotelement geteilt. Danach wird die erste Zeile nach Multiplikation mit u_{21} von der zweiten und nach Multiplikation mit u_{31} von der dritten Zeile subtrahiert:

$$\left(\begin{array}{ccc|ccc} \frac{u_{11}}{u_{11}} & \frac{u_{12}}{u_{11}} & \frac{u_{13}}{u_{11}} & \frac{1}{u_{11}} & 0 & 0 \\ u_{21}-u_{21}\cdot\frac{u_{11}}{u_{11}} & u_{22}-u_{21}\cdot\frac{u_{12}}{u_{11}} & u_{23}-u_{21}\cdot\frac{u_{13}}{u_{11}} & -\frac{u_{21}}{u_{11}} & 1 & 0 \\ u_{31}-u_{31}\cdot\frac{u_{11}}{u_{11}} & u_{32}-u_{31}\cdot\frac{u_{12}}{u_{11}} & u_{33}-u_{31}\cdot\frac{u_{13}}{u_{11}} & -\frac{u_{31}}{u_{11}} & 0 & 1 \end{array}\right) = \left(\begin{array}{ccc|ccc} 1 & u_{12}^{*} & u_{13}^{*} & v_{11}^{*} & 0 & 0 \\ 0 & u_{22}^{*} & u_{23}^{*} & v_{21}^{*} & 1 & 0 \\ 0 & u_{32}^{*} & u_{33}^{*} & v_{31}^{*} & 0 & 1 \end{array}\right)$$

- u_{22}^{*} sei Pivotelement. Die zweite Zeile (Pivotzeile) beider Matrizen wird durch das Pivotelement geteilt. Danach wird die zweite Zeile nach Multiplikation mit u_{12}^{*} von der ersten Zeile und nach Multiplikation mit u_{32}^{*} von der dritten Zeile subtrahiert:

$$\left(\begin{array}{ccc|ccc} 1 & u_{12}^{*}-u_{12}^{*}\cdot\frac{u_{22}^{*}}{u_{22}^{*}} & u_{13}^{*}-u_{12}^{*}\cdot\frac{u_{23}^{*}}{u_{22}^{*}} & v_{11}-u_{12}^{*}\cdot\frac{v_{21}^{*}}{u_{22}^{*}} & -\frac{u_{12}^{*}}{u_{22}^{*}} & 0 \\ 0 & \frac{u_{22}^{*}}{u_{22}^{*}} & \frac{u_{23}^{*}}{u_{22}^{*}} & \frac{v_{21}^{*}}{u_{22}^{*}} & \frac{1}{u_{22}^{*}} & 0 \\ 0 & u_{32}^{*}-u_{32}^{*}\cdot\frac{u_{22}^{*}}{u_{22}^{*}} & u_{33}^{*}-u_{32}^{*}\cdot\frac{u_{23}^{*}}{u_{22}^{*}} & v_{31}^{*}-u_{32}^{*}\cdot\frac{v_{21}^{*}}{u_{22}^{*}} & -\frac{u_{32}^{*}}{u_{22}^{*}} & 1 \end{array}\right) = \left(\begin{array}{ccc|ccc} 1 & 0 & u_{13}^{**} & v_{11}^{**} & v_{12}^{**} \\ 0 & 1 & u_{23}^{**} & v_{21}^{**} & v_{22}^{**} \\ 0 & 0 & u_{33}^{**} & v_{31}^{**} & v_{32}^{**} \end{array}\right)$$

- Als letztes Pivotelement wird u_{33}^{**} verwendet. Die Pivotzeile beider Matrizen wird durch das Pivotelement dividiert. Nach Multiplikation mit u_{23}^{**} wird die Pivotzeile von der zweiten Zeile beider Matrizen und nach Multiplikation mit u_{13}^{**} von der ersten Zeile beider Matrizen subtrahiert:

$$\left(\begin{array}{ccc|ccc} 1 & 0 & u_{13}^{**}-u_{13}^{**}\cdot\frac{u_{33}^{**}}{u_{33}^{**}} & v_{11}^{**}-u_{13}^{**}\cdot\frac{v_{31}^{**}}{u_{33}^{**}} & v_{12}^{**}-u_{13}^{**}\cdot\frac{v_{32}^{**}}{u_{33}^{**}} & -\frac{u_{13}^{**}}{u_{33}^{**}} \\ 0 & 1 & u_{23}^{**}-u_{23}^{**}\cdot\frac{u_{33}^{**}}{u_{33}^{**}} & v_{21}^{**}-u_{23}^{**}\cdot\frac{v_{31}^{**}}{u_{33}^{**}} & v_{22}^{**}-u_{23}^{**}\cdot\frac{v_{32}^{**}}{u_{33}^{**}} & -\frac{u_{23}^{**}}{u_{33}^{**}} \\ 0 & 0 & \frac{u_{33}^{**}}{u_{33}^{**}} & \frac{v_{31}^{**}}{u_{33}^{**}} & \frac{v_{32}^{**}}{u_{33}^{**}} & \frac{1}{u_{33}^{**}} \end{array}\right)$$

$$= \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & v_{11}^{***} & v_{12}^{***} & v_{13}^{***} \\ 0 & 1 & 0 & v_{21}^{***} & v_{22}^{***} & v_{23}^{***} \\ 0 & 0 & 1 & v_{31}^{***} & v_{32}^{***} & v_{33}^{***} \end{array}\right)$$

Rechts steht die inverse Matrix $\boldsymbol{V}=\boldsymbol{U}^{-1}$ und links nunmehr die Einheitsmatrix.

Bei Verfolgung des Rechengangs stellt man fest, daß in der rechten Matrix spaltenweise gerade in dem Maße signifikante Information zuwächst wie in der linken Matrix Plätze frei werden. Es ist deshalb auf einfache Weise möglich, mit dem Speicherplatz für eine einzige Matrix auszukommen, indem die neu entstehenden Elemente v_{ij} auf die Plätze der links entstehenden Einheitsmatrix geschrieben werden.

Programm P 2.08. MATINO. Matrizeninversion ohne Pivotierung.

```
10440 rem !-----------------------------------------------!
10442 rem ! P2.08  MATINO                                 !
10444 rem !        inp:d1,n,u(n,n)                        !
10446 rem !        int:f9,i9,j9,k9,p9,                    !
10448 rem !        out:e1%,u(n,n)                         !
10450 rem !-----------------------------------------------!
10452   e1%=0
10454   for i9=1 to n
10456     p9=u(i9,i9)
10458     if abs(p9)<d1 goto 10488
10460     for j9=1 to n
10462       if j9=i9 goto 10474
10464       f9=u(j9,i9)/p9
10466       for k9=1 to n
10468         u(j9,k9)=u(j9,k9)-u(i9,k9)*f9
10470       next k9
10472       u(j9,i9)=-f9
10474     next j9
10476     for j9=1 to n
10478       u(i9,j9)=u(i9,j9)/p9
10480     next j9
10482     u(i9,i9)=1/p9
10484   next i9
10486   goto 10490
10488   e1%=1
10490 return
10492 rem !-----------------------------------------------!
```

Die ursprüngliche Matrix wird von der invertierten überschrieben.

Bedeutung der Variablennamen:
u(n,n) Matrix; n Dimension der Matrix; e1% Fehlerflag; e1%=1 singuläre Matrix.

Die programmtechnische Realisierung ist relativ einfach und wird im **Programm P 2.08 MATINO** ausgeführt. Dieses Programm hat keine Sicherungen gegen ungünstigen Matrixaufbau und ist deshalb nur bei unkritischer Kondition geeignet. Im **Programm P 2.09 MATINM** ist schließlich zusätzlich das Verfahren der totalen Pivotsuche angewendet, (s. auch Abschn. 2.3.2), d. h., in der gesamten Matrix wird nach dem größten verwendbaren Pivotelement gesucht. Die Position des so gefundenen Pivotelements wird in den beiden eindimensionalen Hilfsfeldern x9%(n) und y9%(n) bezüglich Zeile und Spalte festgehalten. Diese Hilfsfelder sind im Hauptprogramm zu dimensionieren. Nach Abschluß der eigentlichen Inversion werden die durch die Pivotierung zustande gekommenen Zeilen- und Spaltenvertauschungen auf der Basis der Informationen in den Hilfsfeldern wieder rückgängig gemacht. Die totale Pivotierung ist in ähnlicher Form enthalten im Programm von *C. Bau* [Enge 85] bzw. in den "Scientific Subroutines" von DEC und im Standardpaket von [Hewl 79].
Eine Information über die Genauigkeit der Lösung liefert die Testgröße t1, die bei Differenzbildung den kleinsten aufgetretenen Relativwert der Differenz angibt, also

$$t_1 = \frac{v_{ij} - u_{ik} \cdot v_{kj}/u_{kk}}{v_{ij}}. \qquad (2.5)$$

Liegt dieser Wert in der Größe der Maschinenkonstante 2^{-8m} (m Wortlänge der Mantisse in Byte), so war die zu invertierende Matrix nahezu singulär, und die invertierte Matrix ist entsprechend ungenau. Außerdem kann ohne merklichen Mehraufwand die Determinante berechnet werden; ihre Größe gibt ebenfalls einen Hinweis auf die Kondition der Matrix.
Es muß darauf hingewiesen werden, daß die Matrizeninversion nur dann benutzt werden sollte, wenn die invertierte Matrix explizit benötigt wird. Insofern handelt es sich um ein in der Numerik vergleichsweise selten benötigtes Verfahren. Zur Lösung von Gleichungssystemen mit verschiedenen Spaltenvektoren ist die Matrizeninversion nicht zu empfehlen; der Aufwand an Rechenzeit ist wesentlich größer als bei Benutzung z. B. des Banachiewicz-Algorithmus mit Anwendung auf mehrere rechte Seiten [Enge 85]; s. hierzu Abschnitt 2.3.
Jedes der angegebenen Programme überschreibt die Originalmatrix. Wenn beide Matrizen weiter benötigt werden, dann ist die Originalmatrix vor Beginn der Inversion in ein anderes Feld zu kopieren.
Wegen der Verwendung der totalen Pivotierung ist MATINM auch zur Rangbestimmung an quadratischen Matrizen gut geeignet. Die Größe r1 kann wahlweise berechnet und abgefragt werden. Benötigt man nur den Rang, so kann der Teil Rückvertauschung (dritte i9-Schleife) entfallen.
Ein spezielles Programm für Matrizen in eindimensionalen Feldern geben wir hier nicht an. Auf Kosten eines etwas vergrößerten Zeitaufwands für die Indexrechnung kann man die angegebenen Programme sehr einfach modifizieren. Wir setzen voraus, daß wie in (2.2) indiziert wird und daß alle Felder mit 1 beginnend gezählt werden. Dann gilt für den Index k des eindimensionalen Feldes (2.3).
Beispiele für die Programmtestung finden sich am Ende des Abschnitts 2.3.

Programm P 2.09. MATINM. Matrizeninversion mit totaler Pivotierung.

```
10500 rem !-----------------------------------------------!
10502 rem ! P2.09  MATINM                                 !
10504 rem !        inp:d1,n,u(n,n)                        !
10506 rem !        int:f9,h9,i9,j9,k9,m9,p9,x9,y9,        !
      !              x9(n),y9(n)                         !
10508 rem !        out:e1%,d,r1,t1,u(n,n)                 !
10510 rem !-----------------------------------------------!
10512   e1%=0
10514   d=1
10516   t1=1e38
10518   for i9=1 to n                                     ! Permutations-
10520     x9(i9)=0                                        ! vektoren nullen
10522     y9(i9)=0
10524   next i9
10526   for i9=1 to n                                     ! Pivotsuche
10528     p9=0
10530     for j9=1 to n
10532       if x9(j9)<>0 goto 10548
10534       for k9=1 to n
10536         if y9(k9)<>0 goto 10546
10538         if abs(u(j9,k9))<=p9 goto 10546
10540         p9=abs(u(j9,k9))
10542         x9=j9
10544         y9=k9
10546       next k9
10548     next j9
10550     if abs(p9)>d1 goto 10560                        ! Aussprung bei
10552     d=0                                             ! Singularität
10554     e1%=1
10556     r1=i9-1                                         ! Rangbestimmung
10558     goto 10650
10560     r1=i9
10562     x9(x9)=y9
```

```
10564      y9(y9)=x9
10566      p9=u(x9,y9)
10568      for j9=1 to n                                   ! Inversion
10570        if j9=x9 goto 10592
10572        f9=u(j9,y9)/p9
10574        for k9=1 to n
10576          t9=u(j9,k9)
10578          u(j9,k9)=u(j9,k9)-u(x9,k9)*f9               ! Vielfaches der
10580          if y9=k9 goto 10588                         ! Pivotzeile subtra-
10582          if t9=0 goto 10588                          ! hieren
10584          t9=abs(u(j9,k9)/t9)                         ! Testgröße bilden
10586          if t9<t1 then t1=t9
10588        next k9
10590        u(j9,y9)=-f9                                  ! Pivotspalte der in-
10592      next j9                                         ! vertierten Matrix
10594      for j9=1 to n                                   ! besetzen
10596        u(x9,j9)=u(x9,j9)/p9
10598      next j9
10600      u(x9,y9)=1/p9                                   ! auf Pivot das Ele-
                                                           ! ment der invertier-
                                                           ! ten Matrix setzen
10602      d=d*p9                                          ! Determinante
10604    next i9
10606    for i9=1 to n                                     ! Rückordnung
10608      m9=x9(i9)
10610      if m9=i9 goto 10628
10612      for j9=1 to n
10614        h9=u(i9,j9)
10616        u(i9,j9)=u(m9,j9)
10618        u(m9,j9)=h9
10620      next j9
10622      x9(i9)=x9(m9)
10624      x9(m9)=m9
10626      goto 10608
10628      m9=y9(i9)
10630      if m9=i9 goto 10650
10632      for j9=1 to n
10634        h9=u(j9,i9)
10636        u(j9,i9)=u(j9,m9)
10638        u(j9,m9)=h9
10640      next j9
10642      y9(i9)=y9(m9)
10644      y9(m9)=m9
10646      goto 10628
10648    next i9
10650  return
10652  rem !-------------------------------------------------!
```

Die ursprüngliche Matrix wird von der invertierten überschrieben. Das Programm liefert auch den Rang der Matrix sowie den kleinsten Wert der im Laufe der Berechnung aufgetretenen relativen Differenz als Maß für die Genauigkeit der Inversion. – Auf der Basis der in den Hilfsfeldern gespeicherten Information wird die Matrix im Anschluß an die Inversion wieder umgeordnet und damit die dem Originalzustand entsprechende Anordnung der Elemente der invertierten Matrix wiederhergestellt.

Bedeutung der Variablennamen:
u(n,n) Matrix; n Dimension der Matrix; e1% Fehlerflag; e1%=1 singuläre Matrix; d1 kleinste zulässige Größe des Pivotelements; r1 Rang der ursprünglichen Matrix; t1 kleinster Wert der bei der Umrechnung aufgetretenen relativen Differenz zweier Elemente; x9(n), y9(n) Hilfsfelder für die Pivotierung. Diese Felder sind im Hauptprogramm zu dimensionieren.

2.2. Matrizeneigenschaften und -kennzahlen

Eine Reihe von Matrizeneigenschaften ist durch Zahlenwerte gekennzeichnet. Solche Eigenschaftskennzahlen können bestimmt sein durch die Elemente auf der Hauptdiagonalen (Spur der Matrix), durch Elementkombinationen in Zeilen oder Spalten und in der gesamten Matrix. Wir fassen die einfacheren Berechnungen hier aus Gründen der Platzersparnis in die drei CASE-Konstrukten P 2.10 ZSNORM, P 2.11 SUNORM und P 2.12 MMNORM zusammen. Ggf. kann man leicht auf die Berechnung nur einer Kennzahl abrüsten.

2.2.1. Spur der Matrix

Die Spur einer quadratischen Matrix ist gegeben durch die Summe der Elemente auf der Hauptdiagonalen [Zurm50]:

$$s = a_{11} + a_{22} + \ldots + a_{nn}. \tag{2.6}$$

Berechnung mit **Programm P 2.10 ZSNORM**. Steuerflag c1%=4. Bedingung: m=n.

2.2.2. Zeilennorm

Die Zeilennorm ist die maximal vorkommende Summe der Absolutwerte der Elemente in einer Zeile. b1 gibt diesen größten Absolutwert an, a1 die Nummer der Zeile, in der der Absolutwert seinen größten Betrag annimmt. Sind zwei Zeilensummen gleich, so wird die kleinere Zeilennummer ausgegeben. Ist die Matrix leer, dann gibt a1 den Wert 1 aus. Berechnung mit P 2.10 ZSNORM. Steuerflag c1%=1.

Programm P 2.10. ZSNORM. Berechnet in Abhängigkeit vom Wert eines Steuerflags Zeilen- und Spaltennorm sowie Spur einer Matrix.

```
10660 rem !-----------------------------------------------!
10662 rem ! P2.10  ZSNORM                                 !
10664 rem !        inp:c1%,m,n,u(n,m)                     !
10666 rem !        int:i9,j9,t9                           !
10668 rem !        out:a1,b1,e1%                          !
10670 rem !-----------------------------------------------!
10672   e1%=0
10674   if c1%=3 goto 10678
10676   if n<>m goto 10710
10678   a1=1
10680   b1=0
10682   for i9=1 to n
10684     if c1%<>3 goto 10690
10686       b1=b1+u(i9,i9)
10688       goto 10706
10690     t9=0
10692     for j9=1 to m
10694       if c1%=1 then t9=t9+abs(u(i9,j9))
10696       if c1%=2 then t9=t9+abs(u(j9,i9))
10698     next j9
10700     if t9<=b1 goto 10706
10702       a1=i9
10704       b1=t9
10706   next i9
10708  goto 10712
10710     e1%=1
10712 return
10714 rem !-----------------------------------------------!
```

Bedeutung der Variablennamen:
u(n,m,) Matrix; m Zeilenzahl; n Spaltenzahl; c1% Steuerflag. c1%=1 liefert die Zeilennorm, c1%=2 die Spaltennorm und c1%=3 die Spur einer quadratischen n*n-Matrix.
a1 Zeile oder Spalte mit größter Absolutwertsumme; b1 Zeilen- oder Spaltennorm bzw. Spur.

2.2.3. Spaltennorm

Die Spaltennorm ist die maximal vorkommende Summe der Absolutwerte der Elemente in einer Spalte. b2 gibt diesen größten Absolutwert an, a2 die Nummer der Zeile, in der der Absolutwert seinen größten Betrag annimmt. Sind zwei Spaltensummen gleich, so wird die kleinere Spaltennummer ausgegeben. Ist die Matrix leer, so gibt a2 den Wert 1 aus. Berechnung mit P 2.10 ZSNORM. Steuerflag c1%=2.

2.2.4. Euklidische Norm

Die Euklidische Norm, auch Frobenius-Norm oder Betrag der Matrix genannt, ist die Wurzel aus der Summe der Quadrate sämtlicher Elemente u_{ij} einer Matrix. Berechnung mit **Programm P 2.11 SUNORM**. Steuerflag c1%=1.

Programm P 2.11. SUNORM. Bestimmt in Abhängigkeit vom Wert eines Steuerflags die euklidische Norm einer Matrix, die Betragssummennorm oder die Summe aller Matrixelemente.

```
10720 rem !-----------------------------------------------!
10722 rem ! P2.11  SUNORM                                 !
10724 rem !        inp:c1%,m,n,u(n,m)                     !
10726 rem !        int:i9,j9                              !
10728 rem !        out:b1                                 !
10730 rem !-----------------------------------------------!
10732   b1=0
10734   for i9=1 to n
10736     for j9=1 to m
10738       if c1%=1 then b1=b1+u(i9,j9)^2
10740       if c1%=2 then b1=b1+abs(u(i9,j9))
10742       if c1%=3 then b1=b1+u(i9,j9)
10744     next j9
10746   next i9
10748   if c1%=1 then b1=sqr(b1)
10750 return
10752 rem !-----------------------------------------------!
```

Bedeutung der Variablennamen:
u(n,m)Matrix; m Zeilenzahl; n Spaltenzahl; c1% Steuerflag. c1%=1 liefert die euklidische Norm, c1%=2 die Betragssummennorm und c1%=3 die Summe aller Elemente; b1 enthält die Norm entsprechend dem Wert von c1%.

2.2.5. Betragssummennorm

Die Betragssummennorm ist die Summe der Absolutbeträge aller Matrixelemente. Berechnung mit P 2.11 SUNORM. Steuerflag c1%=2.

2.2.6. Elementsummennorm

.Die Elementsummennorm ist die Summe aller Matrixelemente. Berechnung mit P 2.11 SUNORM. Steuerflag c1%=4.

2.2.7. Größtes Matrixelement

Bestimmung mit **Programm P 2.12 MMNORM.** Steuerflag c1%=1.
a1 und a2 liefern Zeile und Spalte des größten Elements, b1 den Wert dieses Elements. Kommen mehrere Elemente für den Größtwert in Frage, so liefern a1 und a2 die kleineren Werte. Bei leerer Matrix sind a1=a2=0.

Programm P 2.12. MMNORM. Bestimmt in Abhängigkeit vom Wert eines Steuerflags aus einer Matrix das größte und das kleinste Element sowie das Element größten und kleinsten absoluten Betrags.

```
10760 rem !-----------------------------------------------!
10762 rem ! P2.12  MMNORM                                 !
10764 rem !        inp:c1%,m,n,u(n,m)                     !
10766 rem !        int:i9,j9                              !
10768 rem !        out:a1,b1                              !
10770 rem !-----------------------------------------------!
10772   a1=1
10774   a2=1
10776   if c1%=1 then b1=-1.7e38
10778   if c1%=2 then b1=+1.7e38
10780   if c1%=3 then b1=0
10782   if c1%=4 then b1=+1.7e38
10784   for i9=1 to n
10786     for j9=1 to m
10788       if c1%=1 then if u(i9,j9)>b1 then gosub 10804
10790       if c1%=2 then if u(i9,j9)<b1 then gosub 10804
10792       if c1%=3 then if abs(u(i9,j9))>abs(b1) then
            gosub 10804
10794       if c1%=4 then if abs(u(i9,j9))<abs(b1) then
            gosub 10804
10796     next j9
10798   next i9
10800   goto 10812
10802   rem uup elementmarkierung
10804     a1=i9
10806     a2=j9
10808     b1=u(i9,j9)
10810   return
10812 return
10814 rem !-----------------------------------------------!
```

Bedeutung der Variablennamen:
u(n,m) Matrix; m Zeilenzahl; n Spaltenzahl; c1% Steuerflag. c1%=1 liefert das größte, c1%=2 das kleinste Matrixelement; c1%=4 liefert das Element größten Absolutbetrags, c1%=8 das Element kleinsten Absolutbetrags.
a1, a2 Zeilen- bzw. Spaltennummer des extremalen Elements; b1 Wert des extremalen Elements.

2.2.8. Kleinstes Matrixelement

Bestimmung wie für größtes Element. Steuerflag c1%=2.

2.2.9. Absolut größtes Matrixelement

Bestimmung wie für größtes Element. Steuerflag c1%=3.

2.2.10. Absolut kleinstes Matrixelement

Bestimmung wie für größtes Element. Steuerflag c1%=4.

2.2.11. Symmetrie

Es wird auf Symmetrie zur Hauptdiagonale geprüft. Man verwende **Programm P 2.13 SYMMET.** Voraussetzung ist eine quadratische Matrix.

Programm P 2.13. SYMMET. Stellt fest, ob eine Matrix symmetrisch zur Hauptdiagonalen ist bzw. ob die Abweichungen von der Symmetrie eine vorgegebene Fehlerschranke unterschreiten.

```
10820 rem !-----------------------------------------------!
10822 rem ! P2.13  SYMMET                                 !
10824 rem !        inp:d1,n,u(n,n)                        !
10826 rem !        int:i9,j9                              !
10828 rem !        out:e1%                                !
10830 rem !-----------------------------------------------!
10832   e1%=0
10834   for i9=1 to n-1
10836     for j9=i+1 to n
10838       if abs(u(i9,j9)/u(j9,i9)-1)>d1 goto 10846
10840     next j9
10842   next i9
10844   goto 10848
10846     e1%=2
10848 return
10850 rem !-----------------------------------------------!
```

Bedeutung der Variablennamen:
u(n,n) quadratische Matrix; n Zeilen- bzw. Spaltenzahl; d1 Toleranz für Abweichung von der Symmetrie. Die Toleranz wird als relative Differenz der quasisymmetrischen Elemente verstanden; d1 muß positiv sein. e1% Fehlerflag. e1%=0 bedeutet, daß die Matrix innerhalb der vorgegebenen Toleranz symmetrisch ist; e1%=2 bedeutet Unsymmetrie.

2.2.12. Determinante

Für die Berechnung des Wertes einer Determinante bieten sich zwei Verfahren an: die Berechnung nach dem Entwicklungssatz von *Laplace* mit schrittweiser Verkleinerung der Größe der Unterdeterminanten sowie die Berechnung mit Hilfe des sog. verketteten Algorithmus. Wegen größerer Geschwindigkeit behandeln wir hier nur das letztgenannte Verfahren. Der verkettete Algorithmus, d. h. die Zerlegung in Dreiecksmatrizen, ist in der Literatur vielfach beschrieben worden (s. z. B. [Hilb77] [Fors71] [Bach82]).
Programm P 2.14 DETERO und **P 2.15 DETERM** berechnen den Wert der Determinante nach diesem Verfahren. DETERO arbeitet ohne, DETERM mit Pivotierung. Im mittleren Teil des Programms DETERM wird die partielle Pivotierung und eventuelle Zeilenvertauschung aus-

geführt. Danach folgt die Dreieckszerlegung und in der letzten Schleife die Multiplikation der Diagonalelemente.

Programm P 2.14. DETERO. Berechnet die Determinante einer Matrix.

```
10860 rem !-------------------------------------------------!
10862 rem ! P2.14  DETERO                                    !
10864 rem !        inp:n,u(n,n)                              !
10866 rem !        int:i9,j9,k9                              !
10868 rem !        out:d                                     !
10870 rem !-------------------------------------------------!
10871   if n=1 goto 10904
10872   for i9=1 to n-1                                      ! Dreieckszerlegung
10874     for j9=i9+1 to n
10876       u(j9,i9)=u(j9,i9)/u(i9,i9)
10878     next j9
10880     if i9=1 goto 10892
10882     for j9=i9+1 to n
10884       for k9=1 to i9-1
10886         u(i9,j9)=u(i9,j9)-u(i9,k9)*u(k9,j9)
10888       next k9
10890     next j9
10892     for j9=i9+1 to n
10894       for k9=1 to i9
10896         u(j9,i9+1)=u(j9,i9+1)-u(j9,k9)*u(k9,i9+1)
10898       next k9
10900     next j9
10902   next i9
10904   d=1
10906   for i9=1 to n                                        ! Produkt der
10908     d=d*u(i9,i9)                                       ! Diagonalelemente
10910   next i9
10912 return
10914 rem !-------------------------------------------------!
```

Es erfolgt keine Pivotierung und keine Sicherung gegen Singularität.

Bedeutung der Variablennamen:
u(n,n) Matrix; n Zeilen- bzw. Spaltenzahl der Matrix; d Wert der Determinante.

Bei größeren Systemen (n > etwa 20) kann der Wert der Determinante auch dann sehr groß werden, wenn die Matrixelemente von der Größenordnung 1 sind [Fors71]. Man muß in solchen Fällen das Programm modifizieren, z. B. in der Weise, daß das Produkt der Diagonalelemente als Summe der Logarithmen von deren absoluten Beträgen gebildet wird. Das Vorzeichen der Determinante ist in diesem Fall als Produkt der SGN-Funktionen der Faktoren zu bilden.

Beispiele zur Testung der Unterprogramme P 2.14 DETERO und P 2.15 DETERM sind im Abschnitt 2.3 angegeben.

Für den praktisch häufigen Fall n=3 wird man statt mit P 2.14 DETERO bzw. P 2.15 DETERM zweckmäßiger mit der folgenden Formel von *Sarrus* rechnen:

$$\left.\begin{aligned} d = {} & u(1,1)*u(2,2)*u(3,3) + u(2,1)*u(3,2)*u(1,3) \\ & + u(3,1)*u(1,2)*u(2,3) - u(3,1)*u(2,2)*u(1,3) \\ & - u(2,1)*u(1,2)*u(3,3) - u(1,1)*u(3,2)*u(2,3) \end{aligned}\right\} \qquad (2.7)$$

sowie in dem sehr einfachen Fall n=2 mit

$$d = u(1,1)*u(2,2) - u(1,2)*u(2,1) \qquad (2.8)$$

Die Berechnung des Wertes einer Determinante mit komplexen Elementen gestaltet sich in entsprechender Weise. **Programm P 2.16 DETERK** ist ein Programm, das in diesem Fall ein-

Programm P 2.15. DETERM. Berechnet die Determinante einer Matrix und verwendet die partielle Pivotierung.

```
10920 rem !-----------------------------------------------!
10922 rem ! P2.15  DETERM                                 !
10924 rem !        inp:d1,n,u(n,n)                        !
10926 rem !        int:i9,j9,k9,l9,s9                     !
10928 rem !        out:d,e1%,r1                           !
10930 rem !-----------------------------------------------!
10932   e1%=0
10934   for i9=1 to n-1
10936     l9=i9                                           ! Suche nach größtem
10938     s9=abs(u(i9,i9))                                ! Element der
10940     for j9=i9+1 to n                                ! i9-ten Spalte
10942       if s9>abs(u(j9,i9)) goto 10948
10944       s9=abs(u(j9,i9))
10946       l9=j9
10948     next j9
10950     if abs(s9)<d1 goto 11022                        ! Pivot zu klein
10952     if l9=i9 goto 10964
10954     for j9=1 to n
10956       s9=-u(i9,j9)
10958       u(i9,j9)=u(l9,j9)
10960       u(l9,j9)=s9
10962     next j9
10964     for j9=i9+1 to n
10966       u(j9,i9)=u(j9,i9)/u(i9,i9)
10968     next j9
10970     if i9=1 goto 10982
10972     for j9=i9+1 to n
10974       for k9=1 to i9-1
10976         u(i9,j9)=u(i9,j9)-u(i9,k9)*u(k9,j9)
10978       next k9
10980     next j9
10982     for j9=i9+1 to n
10984       for k9=1 to i9
10986         u(j9,i9+1)=u(j9,i9+1)-u(j9,k9)*u(k9,i9+1)
10988       next k9
10990     next j9
10992   next i9
10994   d=1                                               ! Produkt der
10996   for i9=1 to n                                     ! Diagonalelemente
10998     s9=0
11000     for j9=1 to n-i9+1                              ! Rückordnung für
11002       if abs(u(j9,j9))<=s9 goto 11008               ! ggf. notwendige
11004         s9=u(j9,j9)                                 ! Rangbestimmung
11006         l9=j9
11008     next j9
11010     if abs(s9)<d1 goto 11022                        ! Faktor zu klein
11012       d=d*s9
11014       u(l9,l9)=u(n-i9+1,n-i9+1)
11016   next i9
11018   r1=n
11020   goto 11028
11022     r1=i9-1                                         ! Rangbestimmung
11024     d=0
11026     e1%=2
11028 return
11030 rem !-----------------------------------------------!
```

Wenn der Rang der Matrix kleiner als n ist, wird bei Angabe des Ranges und eines Fehlerflags abgebrochen.

Bedeutung der Variablennamen:
u(n,n) Matrix; n Zeilen- bzw. Spaltenzahl der Matrix; d1 kleinster zulässiger Faktor für die Bildung der Matrix; d Wert der Determinante; r Rang der Matrix; e1% Fehlerflag. e1%=2 bedeutet, daß ein Faktor für die Bildung der Matrix den Minimalwert d1 unterschritten hat.

gesetzt werden kann [Bach82] [Herr85]. Zur Testung kann folgendes Beispiel verwendet werden:

$$U+i*V=\left(\begin{array}{ccc|c} 10-8*i & 12+3*i & -3+2*i & 5-7*i \\ 12-3*i & 10+8*i & 4-6*i & 10+10*i \\ \hline 1-2*i & 8-4*i & 10+i & 3-8*i \\ \hline 7-5*i & 2+10*i & 3-4*i & 9-2*i \end{array}\right)$$

n = 2: $d = \det(U + i * V) = 11 + 0 * i$
n = 3: $d = \det(U + i * V) = 658 + 1267 * i$
n = 4: $d = \det(U + i * V) = -12206.276 + 47124.2753 * i$

Programm P 2.16. DETERK. Berechnet den i. allg. komplexen Wert einer Determinante mit komplexen Elementen.

```
11040 rem !-----------------------------------------------!
11042 rem ! P2.16  DETERK                                 !
11044 rem !        inp:n,u(n,n),v(n,n)                    !
11046 rem !        int:i9,j9,k9,l9,r9,s9,t9               !
11048 rem !        out:c,d                                !
11050 rem !-----------------------------------------------!
11051   if n=1 goto 11134
11052   for i9=1 to n-1
11054     l9=i9                                           ! Suche nach größtem
11056     r9=abs(u(i9,i9))+abs(v(i9,i9))                  ! Modul der i9-ten
11058     for j9=i9+1 to n                                ! Spalte
11060       t9=abs(u(j9,i9))+abs(v(j9,i9))
11062       if r9>=t9 then 11068
11064       l9=j9
11066       r9=t9
11068     next j9
11070     if l9=i9 then 11088
11072     for j9=1 to n                                   ! Zeilenvertauschung
11074       r9=-u(i9,j9)
11076       u(i9,j9)=u(l9,j9)
11078       u(l9,j9)=r9
11080       s9=-v(i9,j9)
11082       v(i9,j9)=v(l9,j9)
11084       v(l9,j9)=s9
11086     next j9
11088     for j9=i9+1 to n                                ! Dreieckszerlegung
11090       s9=u(i9,i9)*u(i9,i9)+v(i9,i9)*v(i9,i9)
11092       r9=(u(j9,i9)*u(i9,i9)+v(j9,i9)*v(i9,i9))/s9
11094       v(j9,i9)=(u(i9,i9)*v(j9,i9)-u(j9,i9)*v(i9,i9))/s9
11096       u(j9,i9)=r9
11098     next j9
11100     if i9=1 then 11114
11102     for j9=i9+1 to n
11104       for k9=1 to i9-1
11106         u(i9,j9)=u(i9,j9)-u(i9,k9)*u(k9,j9)+v(i9,k9)*v(k9,j9)
11108         v(i9,j9)=v(i9,j9)-v(i9,k9)*u(k9,j9)-u(i9,k9)*v(k9,j9)
11110       next k9
11112     next j9
11114     for j9=i9+1 to n
11116       for k9=1 to i9
11118         u(j9,i9+1)=u(j9,i9+1)-u(j9,k9)*u(k9,i9+1)+v(j9,k9)*v(k9,i9+1)
11120         v(j9,i9+1)=v(j9,i9+1)-v(j9,k9)*u(k9,i9+1)-u(j9,k9)*v(k9,i9+1)
11122       next k9
11124     next j9
11126   next i9
11128   d=1                                               ! Produkt der
11130   c=0                                               ! Diagonalelemente
11132   if n=2*int(n/2) then 11138
```

```
11134   d=u(n,n)
11136   c=v(n,n)
11137   if n=1 goto 11154
11138   for i9=1 to int(n/2)
11140     l9=2*int(n/2)
11142     r9=u(i9,i9)*u(l9,l9)-v(i9,i9)*v(l9,l9)
11144     s9=u(i9,i9)*v(l9,l9)+u(l9,l9)*v(i9,i9)
11146     t9=d*r9-c*s9
11148     c=c*r9+d*s9
11150     d=t9
11152   next i9
11154 return
11156 rem !-----------------------------------------------!
```

Bedeutung der Variablennamen:
u(n,n) reelle Anteile der Elemente; v(n,n) imaginäre Anteile der Elemente; n Dimension der Matrix; c Imaginärteil der Determinante; d Realteil der Determinante.

2.2.13. Rang

Der Rang einer Matrix ist die Maximalzahl linear unabhängiger Spalten- oder Zeilenvektoren, aus denen die Matrix ***U*** besteht. Wir betrachten hier nur die Rangbestimmung an quadratischen Matrizen. Die Rangbestimmung ist im P 2.15 DETERM mit enthalten.

Programm P 2.17. DEFINI. Stellt mittels der Cholesky-Zerlegung fest, ob eine gegebene quadratische Matrix positiv-definit ist.

```
11160 rem !-----------------------------------------------!
11162 rem ! P2.17  DEFINI                                 !
11164 rem !        inp:n,u(n,n)                           !
11166 rem !        int:i9,j9,k9,u9(n,n)                   !
11168 rem !        out:e1%                                !
11170 rem !-----------------------------------------------!
11172   e1%=0
11174   for i9=1 to n
11176     if u(i9,i9)>0 goto 11182
11178       e1%=2
11180       goto 11204
11182     u9(i9,i9)=sqr(u(i9,i9))
11184     if i9=n goto 11202
11186     for j9=i9+1 to n
11188       u9(i9,j9)=u(i9,j9)/u9(i9,i9)
11190     next j9
11192     for j9=i9+1 to n
11194       for k9=i9 to n
11196         u(j9,k9)=u(j9,k9)-u9(i9,j9)*u9(i9,k9)
11198       next k9
11200     next j9
11202   next i9
11204 return
11206 rem !-----------------------------------------------!
```

Die zu prüfende Matrix wird innerhalb des Unterprogramms in das Feld u9(n,n) kopiert, damit die ursprüngliche Matrix nicht zerstört wird. u9(n,n) ist im Hauptprogramm zu dimensionieren.

Bedeutung der Variablennamen:
u(n,n) Matrix; n Zeilen- bzw. Spaltenzahl; u9(n,n) Hilfsmatrix; e1% Flag. e1%=0 bedeutet, u(n,n) ist positiv-definit; e1%=2 bedeutet, u(n,n) ist nicht positiv-definit.

2.2.14. Positiv-Definitheit

Positiv-Definitheit einer quadratischen Matrix ist beispielsweise gegeben, wenn alle Hauptunterdeterminanten positiv sind, also

$$\begin{aligned} &|a(1,1)| > 0 \\ &\begin{vmatrix} a(1,1) & a(1,2) \\ a(2,1) & a(2,2) \end{vmatrix} > 0 \\ &\begin{vmatrix} a(1,1) & a(1,2) & a(1,3) \\ a(2,1) & a(2,2) & a(2,3) \\ a(3,1) & a(3,2) & a(3,3) \end{vmatrix} > 0 \\ &\text{usw.} \end{aligned} \tag{2.9}$$

Weitere Kriterien s. [Schw68].
Das Cholesky-Verfahren setzt die Positiv-Definitheit voraus; wenn alle Diagonalelemente der nach *Cholesky* reduzierten Matrix, d. h. die Radikanden, positiv sind, dann ist das ein hinreichendes Kriterium für Positiv-Definitheit. **Programm P 2.17 DEFINI** ist ein kurzes Programm, das die Cholesky-Zerlegung ausführt und dabei auf Positiv-Definitheit testet. Um die ursprüngliche Matrix nicht zu zerstören, wird im Hilfsfeld u9(n,n) die choleskyzerlegte Matrix aufgebaut. u9(n,n) ist im Hauptprogramm zu dimensionieren.

2.3. Lineare Gleichungssysteme

Ein lineares Gleichungssystem mit n Unbekannten und n Gleichungen liege in folgender Form vor:

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \ldots + u_{1n}x_n &= v_1 \\ u_{21}x_1 + u_{22}x_2 + \ldots + u_{2n}x_n &= v_2 \\ \vdots \quad\quad \vdots \quad\quad\quad\quad \vdots \quad\quad \vdots & \\ u_{n1}x_1 + u_{n2}x_2 + \ldots + u_{nn}x_n &= v_n \end{aligned} \tag{2.10}$$

Dieses System wird mit Vorteil in Matrizenschreibweise dargestellt und lautet dann:

$$\boldsymbol{Ux} = \boldsymbol{v}$$

$$\boldsymbol{U} = \begin{pmatrix} u_{11} & u_{12} & & u_{1n} \\ u_{21} & u_{22} & & u_{2n} \\ \cdot & & & \\ & & & \\ u_{n1} & u_{n2} & & u_{nn} \end{pmatrix} \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \\ \\ x_n \end{pmatrix} \quad \boldsymbol{v} = \begin{pmatrix} v_1 \\ v_2 \\ \\ \\ v_n \end{pmatrix} \tag{2.11}$$

U heißt Koeffizientenmatrix, ***v*** Spaltenvektor und ***x*** Lösungsvektor. Durch die auszuführende Berechnung wird der Lösungsvektor gesucht, d. h. derjenige Satz von Werten x(i), der die Gleichungen (2.10) zu Identitäten macht.
Bei der rechentechnischen Realisierung kann man Koeffizientenmatrix und Spaltenvektor zu einer einzigen sog. erweiterten Matrix zusammenfassen. Es genügt dadurch die Verwendung nur eines Variablennamens und die Dimensionierung nur eines Feldes mit **dim** a(n,n+1):

$$\left(\begin{array}{cccc|c} u_{11} & u_{12} & & u_{1n} & u_{1\,n+1} \\ u_{21} & u_{22} & & u_{2n} & u_{2\,n+1} \\ & & & & \\ & & & & \\ u_{n1} & u_{n2} & & u_{nn} & u_{n\,n+1} \end{array}\right) \qquad (2.12)$$

Die Maßnahme verkürzt u. U. die Programme, indem z. B. bei zeilenweiser Verarbeitung auch der Spaltenvektor mit in der gleichen Schleife erfaßt werden kann. In einigen Fällen erschien diese Anordnung aus Gründen der Übersichtlichkeit nicht sinnvoll, und zwar bei eindimensionaler Abspeicherung der Matrizen – dann würde der Spaltenvektor einen hohen Index tragen und schwer von der Matrix zu unterscheiden sein – sowie auch in Fällen, bei denen mit einer Matrix und mehreren Spaltenvektoren gerechnet wird. In diesen Fällen nennen wir den Spalten- und späteren Lösungsvektor u1(n).
Zur Bestimmung des Lösungsvektors kennt man direkte und iterative Methoden. Direkte Methoden liefern im Ergebnis eines abgeschlossenen Rechengangs das lediglich durch rechnerbedingte Abbruchfehler beeinflußte Ergebnis. Iterative Methoden setzen das Vorhandensein einer Näherungslösung (Startvektor) voraus. Diese Näherung wird schrittweise verbessert.
Vereinfachungen der Rechnungen können sich ergeben, wenn die Koeffizientenmatrizen besondere Eigenschaften haben. Wir behandeln deshalb hier folgende Formen von Koeffizientenmatrizen:

- allgemeine Matrizen. Für Größe und Anordnung der Matrixelemente gelten keine Ordnungsprinzipien oder Einschränkungen
- symmetrische Matrizen, d. h. Matrizen, die symmetrisch zur Hauptdiagonalen sind
- Bandmatrizen, d. h. Matrizen, bei denen nur die Hauptdiagonale und eine Anzahl dieser benachbarte Nebendiagonalen besetzt sind. Die übrigen Elemente sind null.

Gleichungssysteme mit komplexen Koeffizienten, komplexen Spaltenvektoren und komplexen Unbekannten können formal mit den gleichen Methoden wie reelle Gleichungssysteme behandelt werden, jedoch sind dann alle Berechnungen mit komplexen Zahlen auszuführen. Auf Kosten vergrößerter Matrizen (die Koeffizientenmatrix muß zweifach gespeichert werden) kann die komplexe Berechnung ganz vermieden werden, wenn man reelle und imaginäre Anteile wie unabhängige Unbekannte behandelt [Zurm 50].
Das Gleichungssystem in Matrizenschreibweise lautet demnach:

$$\boldsymbol{W} * \boldsymbol{z} = \boldsymbol{a} \qquad (2.13)$$

w(j,k)	= u(j,k) + i∗v(j,k)	komplexe Koeffizientenmatrix
a(j)	= b(j) + i∗c(j)	komplexer Spaltenvektor
z(j)	= x(j) + i∗y(j)	komplexer Lösungsvektor.

Multipliziert man aus, so ergeben sich zwei Gleichungssysteme, in denen nur reelle Größen vorkommen:

$$\left.\begin{array}{l} \boldsymbol{U}*\boldsymbol{x} - \boldsymbol{V}*\boldsymbol{y} = \boldsymbol{b} \\ \boldsymbol{V}*\boldsymbol{x} + \boldsymbol{U}*\boldsymbol{y} = \boldsymbol{c} \end{array}\right\} \qquad (2.14)$$

oder als Hypermatrix geschrieben:

$$\begin{pmatrix} \boldsymbol{U} & -\boldsymbol{V} \\ \boldsymbol{V} & \boldsymbol{U} \end{pmatrix} \begin{pmatrix} \boldsymbol{x} \\ \boldsymbol{y} \end{pmatrix} = \begin{pmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{pmatrix} \tag{2.15}$$

Für den Fall von 2*2-Matrizen würde die Matrizengleichung in ausgeschriebener Form lauten:

$$\left(\begin{array}{cc|cc} u_{11} & u_{12} & -v_{11} & -v_{12} \\ u_{21} & u_{22} & -v_{21} & -v_{22} \\ \hline v_{11} & v_{12} & u_{11} & u_{12} \\ v_{21} & v_{22} & u_{21} & u_{22} \end{array}\right) \left(\begin{array}{c} x_1 \\ x_2 \\ \hline y_1 \\ y_2 \end{array}\right) = \left(\begin{array}{c} b_1 \\ b_2 \\ \hline c_1 \\ c_2 \end{array}\right) \tag{2.16}$$

2.3.1. Gauß-Jordan-Verfahren

Zeilen werden nach Multiplikation mit geeigneten Konstanten fortlaufend voneinander subtrahiert. Im Ergebnis wird aus der Koeffizientenmatrix eine Diagonalmatrix, d. h. eine Matrix, in der nur die Hauptdiagonale besetzt ist. Der Spaltenvektor wird mit dem Lösungsvektor überschrieben. Der Ablauf bei einem System mit drei Unbekannten ist folgender:

- Die erste Zeile wird von den übrigen subtrahiert. Damit die Elemente der ersten Spalte wegfallen, wird die erste Zeile vor Ausführung der Subtraktion mit u_{21}/u_{11} bzw. u_{31}/u_{11} multipliziert:

$$\left(\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21} - u_{11} \cdot \frac{u_{21}}{u_{11}} & u_{22} - u_{12} \cdot \frac{u_{21}}{u_{11}} & u_{23} - u_{13} \cdot \frac{u_{21}}{u_{11}} & u_{24} - u_{14} \cdot \frac{u_{21}}{u_{11}} \\ u_{31} - u_{11} \cdot \frac{u_{31}}{u_{11}} & u_{32} - u_{12} \cdot \frac{u_{31}}{u_{11}} & u_{33} - u_{13} \cdot \frac{u_{31}}{u_{11}} & u_{34} - u_{14} \cdot \frac{u_{31}}{u_{11}} \end{array}\right) = \left(\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22}^* & u_{23}^* & u_{24}^* \\ 0 & u_{32}^* & u_{33}^* & u_{34}^* \end{array}\right)$$

- Die zweite Zeile wird von den übrigen subtrahiert. Ihre Elemente werden vor Ausführung der Subtraktion mit u_{12}/u_{22}^* bzw. u_{32}^*/u_{22}^* multipliziert:

$$\left(\begin{array}{ccc|c} u_{11} & u_{12} - u_{22}^* \cdot \frac{u_{12}}{u_{22}^*} & u_{13} - u_{23}^* \cdot \frac{u_{12}}{u_{22}^*} & u_{14} - u_{24}^* \cdot \frac{u_{12}}{u_{22}^*} \\ 0 & u_{22}^* & u_{23}^* & u_{24}^* \\ 0 & u_{32}^* - u_{22}^* \cdot \frac{u_{32}^*}{u_{22}^*} & u_{33}^* - u_{23}^* \cdot \frac{u_{32}^*}{u_{22}^*} & u_{34}^* - u_{24}^* \cdot \frac{u_{32}^*}{u_{22}^*} \end{array}\right) = \left(\begin{array}{ccc|c} u_{11} & 0 & u_{13}^* & u_{14}^* \\ 0 & u_{22}^* & u_{23}^* & u_{24}^* \\ 0 & 0 & u_{33}^{**} & u_{34}^{**} \end{array}\right)$$

- Die dritte Zeile wird von den übrigen subtrahiert. Ihre Elemente werden vor Ausführung der Subtraktion mit u_{13}^*/u_{33}^{**} bzw. u_{23}^*/u_{33}^{**} multipliziert:

$$\left(\begin{array}{ccc|c} u_{11} & 0 & u_{13}^* - u_{33}^{**} \cdot \frac{u_{13}^*}{u_{33}^{**}} & u_{14}^* - u_{34}^{**} \cdot \frac{u_{13}^*}{u_{33}^{**}} \\ 0 & u_{22}^* & u_{23}^* - u_{33}^{**} \cdot \frac{u_{23}^*}{u_{33}^{**}} & u_{24}^* - u_{34}^{**} \cdot \frac{u_{23}^*}{u_{33}^{**}} \\ 0 & 0 & u_{33}^{**} & u_{34}^{**} \end{array}\right) = \left(\begin{array}{ccc|c} u_{11} & 0 & 0 & u_{14}^{**} \\ 0 & u_{22}^* & 0 & u_{24}^{**} \\ 0 & 0 & u_{33}^{**} & u_{34}^{**} \end{array}\right)$$

In einem anschließenden Prozeß werden die Elemente des nunmehrigen Spaltenvektors durch die Hauptdiagonalelemente dividiert und sind damit Elemente des Lösungsvektors. Die Diagonalelemente behalten ihre zufälligen Werte. Der geschilderte Ablauf entspricht nicht dem üblichen Rechengang, bei dem die Spalten sofort durch die Hauptdiagonalelemente dividiert werden. Die geschilderte Methode erfordert aber weniger Divisionen und ist dadurch schneller. Ebenso ist es nicht erforderlich, die Nullelemente wirklich zu bilden; wir lassen dort im Interesse kürzerer Rechenzeit zufällige Zwischenergebnisse stehen. Schreibt man im **Programm P 2.18 GAJOST** auf Zeile 11240 für die untere Schleifengrenze i9 statt wie angegeben i9+1, so erscheinen in der transformierten Matrix wieder Nullen.
Will man den Spalten- bzw. Lösungsvektor mit einem anderen Variablennamen versehen, z. B. v(n), so hat man die k9-Schleife im Zeilenbereich 11240 . . . 11244 nur bis n laufen zu lassen und als Zeile 11239 einzufügen:

v(j9) = v(j9) − v(i9)*b9.

Das Gauß-Jordan-Verfahren wird bezüglich der Rechengeschwindigkeit und der Rechengenauigkeit von anderen Methoden übertroffen (s. Abschn. 2.3.12.2). Für ungefährlich konditionierte Systeme mit wenigen Unbekannten (n < 6) kann es indessen wegen der Kürze des Programms empfohlen werden. Programmtechnischer Aufwand für Pivotierung u. ä. lohnt sich im Normalfall nicht.

Programm P 2.18. GAJOST. Lösung eines linearen Gleichungssystems mit n Unbekannten nach der Methode *Gauß-Jordan.*

```
11220 rem !-----------------------------------------------!
11222 rem ! P2.18  GAJOST                                 !
11224 rem !        inp:n,u(n,n+1)                         !
11226 rem !        int:b9,i9,j9,k9                        !
11228 rem !        out:u(i,n+1)                           !
11230 rem !-----------------------------------------------!
11232   for i9=1 to n
11234     for j9=1 to n
11236       if i9=j9 goto 11246
11238       b9=u(j9,i9)/u(i9,i9)
11240       for k9=i9+1 to n+1
11242         u(j9,k9)=u(j9,k9)-u(i9,k9)*b9
11244       next k9
11246     next j9
11248   next i9
11250   for i9=1 to n
11252     u(i9,n+1)=u(i9,n+1)/u(i9,i9)
11254   next i9
11256 return
11258 rem !-----------------------------------------------!
```

Es handelt sich um eine Variante des gängigen Verfahrens, die bezüglich der Anzahl notwendiger Rechenoperationen optimiert ist. Die Division durch die Diagonalelemente erfolgt erst am Ende des Rechengangs, und die Elemente außerhalb der Hauptdiagonalen bleiben mit zufälligen Zwischenergebnissen besetzt. Das Programm arbeitet ohne Pivotierung und ohne Sicherung gegen Singularität. Bei Verwendung der Schleifenanweisung **for** k9=i9 **to** n+1 werden die Elemente außerhalb der Hauptdiagonalen mit Nullen aufgefüllt. In der angegebenen Form mit **for** k9=i9+1 **to** n+1 bleiben zufällige Zwischenergebnisse stehen.
Bedeutung der Variablennamen:
u(i,i+1) erweiterte Matrix; n Anzahl der Unbekannten; u(i,n+1) Spaltenvektor, wird vom Lösungsvektor überschrieben; $1 \leqq i \leqq n$.

2.3.2. Gauß-Elimination

Eine praktisch sehr wichtige Methode zur Lösung von Gleichungssystemen besteht darin, daß das Gleichungssystem in eine Dreiecksform gebracht wird. Durch Addition oder Subtraktion von Zeilen zu anderen nach Multiplikation mit geeigneten Faktoren wird erreicht, daß alle Elemente unterhalb der Hauptdiagonalen zu Null werden. Nachdem diese Situation erreicht ist, wird durch Substitution, beginnend mit der letzten Zeile, der Lösungsvektor bestimmt.
Für ein System mit drei Unbekannten ergibt sich folgender Lösungsweg:

- Subtraktion der ersten Zeile nach Multiplikation mit u_{21}/u_{11} von der zweiten Zeile und nach Multiplikation mit u_{31}/u_{11} von der dritten Zeile:

$$\left(\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21}-u_{21}\cdot\frac{u_{11}}{u_{11}} & u_{22}-u_{21}\cdot\frac{u_{12}}{u_{11}} & u_{23}-u_{21}\cdot\frac{u_{13}}{u_{11}} & u_{24}-u_{21}\cdot\frac{u_{14}}{u_{11}} \\ u_{31}-u_{31}\cdot\frac{u_{11}}{u_{11}} & u_{32}-u_{31}\cdot\frac{u_{12}}{u_{11}} & u_{33}-u_{31}\cdot\frac{u_{13}}{u_{11}} & u_{34}-u_{31}\cdot\frac{u_{14}}{u_{11}} \end{array}\right) = \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22}^* & u_{23}^* & u_{24}^* \\ 0 & u_{32}^* & u_{33}^* & u_{34}^* \end{pmatrix}$$

Bei der rechentechnischen Realisierung brauchen die Operationen in der ersten Spalte nicht ausgeführt zu werden. Die Nullen in der Matrix werden für die weitere Rechnung nicht benötigt, aber man spart etwas Rechenzeit.

- Subtraktion in der zweiten Zeile nach Multiplikation mit u_{32}^*/u_{22}^* von der dritten Zeile:

$$\left(\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22}^* & u_{23}^* & u_{24}^* \\ 0 & u_{32}^*-u_{32}^*\cdot\frac{u_{22}^*}{u_{22}^*} & u_{33}^*-u_{32}^*\cdot\frac{u_{23}^*}{u_{22}^*} & u_{34}^*-u_{32}^*\cdot\frac{u_{24}^*}{u_{22}^*} \end{array}\right) = \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22}^* & u_{23}^* & u_{24}^* \\ 0 & 0 & u_{33}^{**} & u_{34}^{**} \end{pmatrix}$$

Auch hier braucht wiederum die Bildung der Null nicht ausgeführt zu werden.

- Bestimmung des Lösungsvektors durch Rückwärtseinsetzen (back substitution).

Wenn wir zur Erläuterung der Rücksubstitution statt der Matrix das nunmehrige Gleichungssystem noch einmal vollständig aufschreiben, so ergibt sich nach Weglassen der Sterne folgende Form:

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 &= u_{14} \\ u_{22}x_2 + u_{23}x_3 &= u_{24} \\ u_{33}x_3 &= u_{34} \end{aligned}$$

$x_3 = u_{34}/u_{33}$ ist bereits die erste Lösung, die weiteren Lösungen erhält man, wenn man nach oben hin einsetzt:

$$\begin{aligned} x_2 &= (u_{24}-u_{23}x_3)/u_{22} \\ x_1 &= (u_{14}-u_{13}x_3-u_{12}x_2)/u_{11} \end{aligned}$$

Dieses Rechenschema wird vom **Programm P 2.19 GAOP2D** realisiert. – Bei ungünstigen Größen der Matrixelemente ist erheblicher Genauigkeitsverlust möglich, wenn nicht gar Abbruch durch Divisionsfehler in dem Falle, daß eines der u_{ii} Null ist. Zeilen- und Spaltenvertauschungen verändern das Ergebnis nicht. Wir können daher vor Ausführung jedes Reduktionsschritts die Matrix so umordnen, daß das jeweils größte mögliche Element auf der Diagonalen steht. Zieht man zur Suche des größten Diagonalelements Zeilen- und Spaltenvertauschung in Betracht, so spricht man von "vollständiger Pivotierung" (total pivoting).

Programm P 2.19. GAOP2D. Lösung eines linearen Gleichungssystems mit n Unbekannten mittels der Gauß-Elimination.

```
11260 rem !-----------------------------------------------!
11262 rem ! P2.19  GAOP2D                                 !
11264 rem !        inp:n,u(n,n+1)                         !
11266 rem !        int:i9,j9,k9,l9                        !
11268 rem !        out:u(i,n+1)                           !
11270 rem !-----------------------------------------------!
11272   for i9=1 to n-1
11274     for j9=i9+1 to n
11276       l9=u(j9,i9)/u(i9,i9)
11278       for k9=i9+1 to n+1
11280         u(j9,k9)=u(j9,k9)-l9*u(i9,k9)
11282       next k9
11284     next j9
11286   next i9
11288   for i9=n to 1 step-1
11290     l9=u(i9,n+1)
11292     if i9=n goto 11300
11294     for j9=i9+1 to n
11296       l9=l9-u(i9,j9)*u(j9,n+1)
11298     next j9
11300     u(i9,n+1)=l9/u(i9,i9)
11302   next i9
11304 return
11306 rem !-----------------------------------------------!
```

Das Programm arbeitet ohne Pivotierung und ohne Sicherung gegen Singularität. Matrix und Spaltenvektor sind in einem zweidimensionalen Feld gespeichert.

Bedeutung der Variablennamen:
u(i,j) erweiterte Matrix; n Anzahl der Unbekannten; u(i,n+1) Spaltenvektor, wird vom Lösungsvektor überschrieben; $1 \leqq i \leqq n$; $1 \leqq j \leqq n+1$.

Wählt man dagegen nur in der Spalte des Diagonalelements das größte Element aus und bringt es durch Zeilenvertauschung in die Diagonalelementposition, so nennt man dieses Verfahren "teilweise Pivotierung" (partial pivoting). Einige Autoren sprechen von "teilweiser Pivotierung", wenn eine Zeilenvertauschung nur im Falle der Unterschreitung eines vorgegebenen Minimalwerts durch das Diagonalelement ausgeführt wird. Die Suche des größten Elements in der jeweiligen Diagonalspalte wird demgegenüber als "vollständige Pivotierung" bezeichnet. Wir werden die zuerst genannte Definition benutzen. – Im **Programm P 2.20 GAMP2D** wurde die teilweise Pivotierung angewendet.

Die Notwendigkeit der Pivotierung wird ausführlich in [Fors71] behandelt. In der gleichen Literaturstelle wird darauf hingewiesen, daß für praktische Anwendungen die "teilweise Pivotierung" vollständig ausreichend sei. Wir beschränken uns daher auf die Programmierung dieses Verfahrens. Ein BASIC-Programm für vollständige Pivotierung findet man bei [Scra84], ebenso im Abschnitt 2.1.8. Wegen des zusätzlichen Rechenzeitaufwands sollte man die Pivotierung nur verwenden, wenn die Werte der Matrixelemente tatsächlich in großen Bereichen schwanken. Auf die Hilbert-Matrix angewendet bringt beispielsweise die Pivotsuche keinerlei Gewinn.

Die Verwendung des Programms für mehrere rechte Seiten ist grundsätzlich möglich. Wir verweisen jedoch auf die unter LR-Zerlegung (s. Abschn. 2.3.3) aufgeführten Programme, bei denen die Auftrennung bereits realisiert ist.

Beispiele zur Testung der angegebenen Programme finden sich am Ende des Abschnitts 2.3.

Programm P 2.20. GAMP2D. Lösung eines linearen Gleichungssystems mit n Unbekannten mittels der Gauß-Elimination.

```
11310 rem !-----------------------------------------------!
11312 rem ! P2.20  GAMP2D                                 !
11314 rem !        inp:d1,n,u(n,n+1)                      !
11316 rem !        int:i9,j9,k9,l9,s9                     !
11318 rem !        out:e1%,u(i,n+1)                       !
11320 rem !-----------------------------------------------!
11322   e1%=0
11324   for i9=1 to n-1
11326     l9=i9                                          ! Pivotierung
11328     s9=abs(u(i9,i9))
11330     for j9=i9+1 to n
11332       if s9>abs(u(j9,i9)) goto 11338               ! Suche nach größtem
11334         s9=abs(u(j9,i9))                           ! Pivotelement
11336         l9=j9
11338     next j9
11340     if abs(s9)<d1 goto 11386                       ! Aussprung bei
                                                         ! Singularität
11342     if l9=i9 goto 11354
11344     for j9=i9 to n+1                               ! Zeilenvertauschung
11346       s9=u(i9,j9)
11348       u(i9,j9)=u(l9,j9)
11350       u(l9,j9)=s9
11352     next j9
11354     for j9=i9+1 to n                               ! Gauß-Elimination
11356       l9=u(j9,i9)/u(i9,i9)
11358       for k9=i9+1 to n+1
11360         u(j9,k9)=u(j9,k9)-l9*u(i9,k9)
11362       next k9
11364     next j9
11366   next i9
11368   for i9=n to 1 step-1                           ! Rücksubstitution
11370     l9=u(i9,n+1)
11372     if i9=n goto 11388
11374     for j9=i9+1 to n
11376       l9=l9-u(i9,j9)*u(j9,n+1)
11378     next j9
11380     u(i9,n+1)=l9/u(i9,i9)
11382   next i9
11384   goto 11388
11386     e1%=1
11388 return
11390 rem !-----------------------------------------------!
```

Das Programm verwendet partielle Pivotierung und testet auf Singularität. Matrix und Spaltenvektor sind in einem zweidimensionalen Feld gespeichert.

Bedeutung der Variablennamen:
u(i,i+1) erweiterte Matrix; n Anzahl der Unbekannten; u(i,n+1) Spaltenvektor, wird vom Lösungsvektor überschrieben; e1% Fehlerflag. e1%=0 bedeutet fehlerfreie Abarbeitung; e1%=1 bedeutet singuläre Matrix, $1 \leqq i \leqq n$.

Das folgende **Programm P 2.21 GAMP1D** arbeitet in gleicher Weise wie P 2.20 GAMP2D, jedoch sind hier die Matrixelemente in einem eindimensionalen Feld u(n*n) untergebracht. Dadurch ist eine Reihe von Indexrechnungen notwendig, die das Programm geringfügig verlangsamen. Indessen kann es in Fällen nützlich sein, in denen ein einmal dimensioniertes Feld wiederverwendet werden muß. Die Indexzählung geschieht in Spaltenrichtung entsprechend dem in (2.3) angegebenen Schema, wobei jedoch der Spaltenvektor mit u1(i) bezeichnet ist. Das folgende Beispiel zeigt die Indizierung für ein 3*3-System:

$$\left(\begin{array}{ccc|c} u(1) & u(4) & u(7) & u1(1) \\ u(2) & u(5) & u(8) & u1(2) \\ u(3) & u(6) & u(9) & u1(3) \end{array}\right)$$

Dem Programm liegt ein FORTRAN-Programm aus der Programmsammlung "Scientific Subroutines" von Digital Equipment Corp. zugrunde.

Programm P 2.21. GAMP1D. Lösung eines linearen Gleichungssystems mit n Unbekannten mittels der Gauß-Elimination.

```
11400 rem !-----------------------------------------------!
11402 rem ! P2.21  GAMP1D                                 !
11404 rem !        inp:d1,n,u(n*n),u1(n)                  !
11406 rem !        int:b9,c9,h9,i9,j9,k9,l9,m9,s9         !
11408 rem !        out:e1%,u1(n)                          !
11410 rem !-----------------------------------------------!
11412   e1%=0
11414   l9=-n
11416   for i9=1 to n
11418     l9=l9+n+1                                       ! Pivotierung
11420     b9=0
11422     for j9=i9 to n
11424       if abs(b9)>=abs(u(l9-i9+j9)) goto 11430        ! Suche nach größtem
11426       b9=u(l9-i9+j9)                                ! Pivotelement
11428       m9=j9
11430     next j9
11432     if abs(b9)>d1 goto 11438                        ! Aussprung bei
11434     e1%=1                                           ! Singularität
11436     goto 11500
11438     c9=i9+n*(i9-2)
11440     for j9=i9 to n                                  ! Zeilenvertauschung
11442       c9=c9+n
11444       s9=u(c9)
11446       u(c9)=u(c9+m9-i9)
11448       u(c9+m9-i9)=s9
11450       u(c9)=u(c9)/b9
11452     next j9
11454     s9=u1(m9)                                       ! Gauß-Elimination
11456     u1(m9)=u1(i9)
11458     u1(i9)=s9/b9
11460     if i9=n goto 11482
11462     c9=n*(i9-1)
11464     for j9=i9+1 to n
11466       b9=c9+j9
11468       for k9=i9+1 to n
11470         h9=n*(k9-1)+j9
11472         u(h9)=u(h9)-u(b9)*u(h9+i9-j9)
11474       next k9
11476       u1(j9)=u1(j9)-u1(i9)*u(b9)
11478     next j9
11480   next i9
11482   for i9=1 to n-1                                   ! Rücksubstitution
11484     b9=n*n-i9
11486     c9=n
11488     for j9=1 to i9
11490       u1(n-i9)=u1(n-i9)-u(b9)*u1(c9)
11492       b9=b9-n
11494       c9=c9-1
11496     next j9
11498   next i9
11500 return
11502 rem !-----------------------------------------------!
```

Das Programm verwendet partielle Pivotierung und testet auf Singularität. Matrix und Spaltenvektor sind in je einem eindimensionalen Feld gespeichert.

Bedeutung der Variablennamen:
u(i,j) Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; e1% Fehlerflag. e1%=0 bedeutet fehlerfreie Abarbeitung; e1%=1 bedeutet singuläre Matrix. $1 \leqq i \leqq n$; $1 \leqq j \leqq n$.

2.3.3. LR-Zerlegung

Bei der LR-Zerlegung wird die ursprüngliche quadratische Matrix ***U*** in das Produkt aus zwei Dreiecksmatrizen zerlegt, einer linken ***L*** und einer rechten ***R***. Im englischen Schrifttum ist auch die Bezeichnung LU-Zerlegung üblich (lower and upper matrix). Das Verfahren ist der Gauß-Elimination ähnlich und liefert gleiche Ergebnisse, erfordert aber weniger Speicherzugriffe und ist daher etwas schneller als jenes. Die Matrizenmultiplikation ***L * R*** liefert die ursprüngliche Matrix ***U = L * R***. Im Schema von *Falk* (Bild 2.1) ergibt sich für eine 4*4-Matrix die im **Bild 2.2** gezeigte Anordnung [Böhm 85].

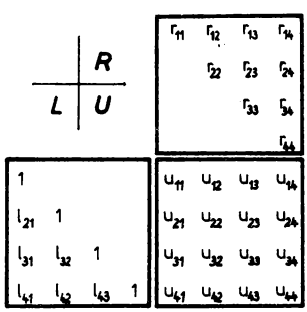


Bild 2.2. Multiplikationsschema für die LR-Zerlegung
Die linke Dreiecksmatrix *L* multipliziert mit der rechten Dreicksmatrix *R* ergibt die urpsprüngliche Matrix *U*.

Zur Lösung von Gleichungssystemen geht man so vor, daß das ursprüngliche System ***U x = v*** in das äquivalente System ***R x = z*** mit ***L z = v*** übergeführt wird, was durch Umstellen dieser drei einfachen Matrizengleichungen leicht verifiziert werden kann.
Da die Zahlenwerte auf der Hauptdiagonalen der Matrix ***L*** keine signifikante Information enthalten, brauchen sie nicht gespeichert zu werden. Damit können aber auch die Matrizen ***L*** und ***R*** auf dem gleichen Speicherbereich wie die ursprüngliche Matrix ***U*** untergebracht werden.
Schreibt man die auszuführende Matrizenmultiplikation explizit auf, so ergibt sich bei einer 4*4-Matrix folgendes System von Gleichungen, aus denen sich die Koeffizienten l_{ij} und r_{ij} bestimmen lassen:

$$u_{11} = r_{11}$$
$$u_{12} = r_{12}$$
$$u_{13} = r_{13}$$
$$u_{14} = r_{14}$$

$$u_{21} = l_{21}r_{11}$$
$$u_{22} = l_{21}r_{12} + r_{22}$$
$$u_{23} = l_{21}r_{13} + r_{23}$$
$$u_{24} = l_{21}r_{14} + r_{24}$$

$$u_{31} = l_{31}r_{11}$$
$$u_{32} = l_{31}r_{12} + l_{32}r_{22}$$
$$u_{33} = l_{31}r_{13} + l_{32}r_{23} + r_{33}$$
$$u_{34} = l_{31}r_{14} + l_{32}r_{24} + r_{34}$$

$$u_{41} = l_{41}r_{11}$$
$$u_{42} = l_{41}r_{12} + l_{42}r_{22}$$
$$u_{43} = l_{41}r_{13} + l_{42}r_{23} + l_{43}r_{33}$$
$$u_{44} = l_{41}r_{14} + l_{42}r_{24} + l_{43}r_{34} + r_{44}$$

Für das Überschreiben der ursprünglichen Matrixelemente mit den r_{ij}, l_{ij} gibt es verschiedene Schemata („Parkettierungen"): Überschreibt man Zeile für Zeile, Spalte für Spalte oder abwechselnd zeilen- und spaltenweise, so spricht man von Parkettierungen nach *Gauß*, nach *Banachiewicz* oder nach *Crout* **(Bild 2.3).**

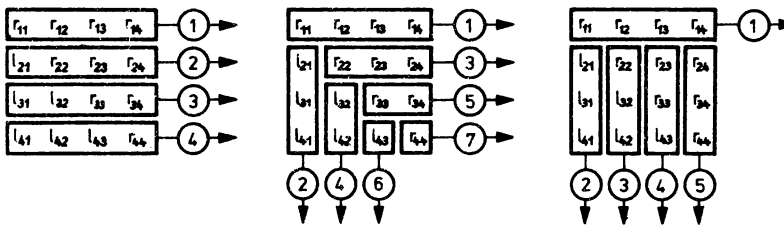


Bild 2.3. Parkettierungsmöglichkeiten für den Aufbau der LR-Matrizen

Von links nach rechts Methoden nach *Gauß*, nach *Crout* und nach *Banachiewicz*. In den Kreisen ist die Abarbeitungsreihenfolge angegeben.

In den nachfolgenden Programmen verwenden wir der relativ einfachen Pivotierungsmöglichkeit wegen das Verfahren von *Banachiewicz*. Demnach speichern wir eine dreieckszerlegte 4*4-Matrix in der folgenden Reihenfolge ab:

$$r_{11} := u_{11}$$
$$r_{12} := u_{12}$$
$$r_{13} := u_{13}$$
$$r_{14} := u_{14}$$

$$l_{21} := u_{21}/r_{11}$$
$$l_{31} := u_{31}/r_{11}$$
$$l_{41} := u_{41}/r_{11}$$

$$r_{22} := u_{22} - l_{21}r_{12}$$
$$l_{32} := (u_{32} - l_{31}r_{12})/r_{22}$$
$$l_{42} := (u_{42} - l_{41}r_{12})/r_{22}$$

$$r_{23} := u_{23} - l_{21}r_{13}$$
$$r_{33} := u_{33} - l_{31}r_{13} - l_{32}r_{23}$$
$$l_{43} := (u_{43} - l_{41}r_{13} - l_{42}r_{23})/r_{33}$$

$$r_{24} := u_{24} - l_{21}r_{14}$$
$$r_{34} := u_{34} - l_{31}r_{14} - l_{32}r_{24}$$
$$r_{44} := u_{44} - l_{41}r_{14} - l_{42}r_{24} - l_{43}r_{34}$$

Das **Programm P 2.22 BANAOP** führt die Dreieckszerlegung in der geschilderten Weise durch. Es arbeitet ohne Pivotierung und ohne Sicherung gegen Singularität. Zur Lösung von Gleichungssystemen ist anschließend das **Programm P 2.23 SUBST1** aufzurufen. Mit letzterem können unter Verwendung der einmal dreieckszerlegten Matrix für mehrere Spaltenvektoren die Lösungsvektoren bestimmt werden. Die Lösungsvektoren überschreiben die ursprünglichen Spaltenvektoren. In dieser Form verläuft die Rechnung wesentlich ökonomischer als bei Verwendung der Matrixinversion.

Programm P 2.22. BANAOP. Führt die LR-Zerlegung einer Matrix durch.

```
11510 rem !-----------------------------------------------!
11512 rem ! P2.22  BANAOP   /folgt:SUBST1/                !
11514 rem !        inp:n,u(n,n)                           !
11616 rem !        int:i9,j9,k9                           !
11618 rem !        out:u(n,n)                             !
11520 rem !-----------------------------------------------!
11522   for i9=1 to n
11524     if i9<3goto 11536
11526     for j9=2 to i9-1
11528       for k9=1 to j9-1
11530         u(j9,i9)=u(j9,i9)-u(j9,k9)*u(k9,i9)
11532       next k9
11534     next j9
11536     for j9=i9 to n
11538     if i9<2 goto 11546
11540       for k9=1 to i9-1
11542         u(j9,i9)=u(j9,i9)-u(j9,k9)*u(k9,i9)
11544       next k9
11546       if j9>=i9+1 then u(j9,i9)=u(j9,i9)/u(i9,i9)
11548     next j9
11550   next i9
11552 return
11554 rem !-----------------------------------------------!
```

Das Programm arbeitet ohne Pivotierung und ohne Sicherung gegen Singularität und ist daher nur für kleinere, unkritische Anwendungen geeignet oder aber für Matrizen, deren Elemente regulär variieren. Man achte auf abweisende Schleifensteuerung und verwende notfalls bedingte Sprunganweisungen vor den Schleifenanweisungen. Zur Lösung von Gleichungssystemen ist im Anschluß P 2.23 SUBST1 aufzurufen.

Bedeutung der Variablennamen:
u(i,j) Matrix, wird von der LR-zerlegten Matrix überschrieben; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben, jedoch erst von SUBST1 benötigt; n Anzahl der Unbekannten; $1 \leqq i \leqq n$; $1 \leqq j \leqq n$.

Programm P 2.23. SUBST1. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor u1(i) durch und setzt das Vorhandensein einer mittels LR-Zerlegung erzeugten oberen Dreiecksmatrix u(i,j) voraus.

```
11560 rem !-----------------------------------------------!
11562 rem ! P2.23  SUBST1  /folgt auf:BANAOP/              !
11564 rem !        inp:n,u(n,n),u1(n)                     !
11566 rem !        int:i9,j9                              !
11568 rem !        out:u1(n)                              !
11570 rem !-----------------------------------------------!
11572   for i9=2 to n                                     ! Vorwärtssubst.
11574     for j9=1 to i9-1
11576         u1(i9)=u1(i9)-u(i9,j9)*u1(j9)
11578     next j9
11580   next i9
11581   u1(n)=u1(n)/u(n,n) : if n=1 goto 11594
11582   for i9=n-1 to 1 step -1                           ! Rückwärtssubst.
11584     for j9=n to i9+1 step -1
11586       u1(i9)=u1(i9)-u(i9,j9)*u1(j9)
11588     next j9
11590     u1(i9)=u1(i9)/u(i9,i9)
11592   next i9
11594 return
11596 rem !-----------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten. Der Spaltenvektor wird vom Lösungsvektor überschrieben.

Bedeutung der Variablennamen:
u1(i) Spaltenvektor; n Anzahl der Unbekannten; $1 \leqq i \leqq n$; $1 \leqq j \leqq n$.

Wenn zu befürchten ist, daß ein r_{ii} zu Null wird, dann ist Pivotierung und Sicherung gegen Singularität angebracht. Das **Programm P 2.24 BANAMP** enthält einen Algorithmus zur partiellen Pivotierung. Die dabei vorgenommenen Zeilenvertauschungen werden in dem Vektor x9%(n) (Permutationsvektor) gespeichert und am Ende der Berechnung rückgängig gemacht.

Programm P 2.24. BANAMP. Führt die LR-Zerlegung einer Matrix durch.

```
11600 rem !-----------------------------------------------!
11602 rem ! P2.24  BANAMP    /folgt:SUBST2/                !
11604 rem !        inp:d1,n,u(n,n)                         !
11606 rem !        int:i9,j9,k9,p9,s9,t9                   !
11608 rem !        out:e1%,s1(n),u(n,n)                    !
11610 rem !-----------------------------------------------!
11612   e1%=0
11614   for i9=1 to n
11616     s1(i9)=i9
11618   next i9
11620   for i9=1 ton
11622     if i9=1 goto 11644
11623     if i9=2 goto 11634
11624     for j9=2 to i9-1
11626       for k9=1 to j9-1
11628         u(j9,i9)=u(j9,i9)-u(j9,k9)*u(k9,i9)
11630       next k9
11632     next j9
11634     for j9=i9 to n
11636       for k9=1 to i9-1
11638         u(j9,i9)=u(j9,i9)-u(j9,k9)*u(k9,i9)
11640       next k9
11642     next j9
11644     if i9=n goto 11676
11646     p9=i9
11648     t9=abs(u(i9,i9))
11650     for j9=i9+1 to n
11652       if abs(u(j9,i9))<=t9 goto 11658
11654         t9=abs(u(j9,i9))
11656         p9=j9
11658     next j9
11660     for j9=1 to n
11662       s9=u(i9,j9)
11664       u(i9,j9)=u(p9,j9)
11666       u(p9,j9)=s9
11668     next j9
11670     s9=s1(i9)
11672     s1(i9)=s1(p9)
11674     s1(p9)=s9
11676     if abs(u(i9,i9))<d1 goto 11688
11677     if i9=n goto 11684
11678     for j9=i9+1 to n
11680       u(j9,i9)=u(j9,i9)/u(i9,i9)
11682     next j9
11684   next i9
11686   goto 11690
11688     e1%=1
11690 return
11692 rem !-----------------------------------------------!
```

Das Programm arbeitet mit partieller Pivotierung. Der Vertauschungszustand wird in einem sog. Permutationsvektor gespeichert und am Ende der Berechnung rückgängig gemacht. Man achte auf abweisende Schleifensteuerung und verwende notfalls bedingte Sprunganweisungen vor den Schleifenanweisungen. Zur Lösung von Gleichungssystemen ist im Anschluß das P 2.25 SUBST2 aufzurufen.

Bedeutung der Variablennamen:
u(i,j) Matrix, wird von der LR-zerlegten Matrix überschrieben; n Anzahl der Unbekannten; x9%(i) Permutationsvektor, ist im Hauptprogramm zu dimensionieren; e1% Fehlerflag. e1%=0 bedeutet ordnungsgemäße Abarbeitung; e1%=1 bedeutet singuläre Matrix. $1 \leqq i \leqq n$; $1 \leqq j \leqq n$.

Bei Verwendung mit nur einem Spaltenvektor ist es zeitgünstiger, die Vertauschungen am Spaltenvektor durchzuführen. Das würde dann aber Eingriffe am Hauptprogramm oder dem hier ebenfalls anzuschließenden **Programm P 2.25 SUBST2** erfordern, die wir der Einheitlichkeit und Übersichtlichkeit der Programmstrukturen wegen vermieden haben.

Programm P 2.25. SUBST2. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor u1(i) durch.

```
11700 rem !-----------------------------------------------!
11702 rem ! P2.25  SUBST2   /folgt auf:BANAMP/             !
11704 rem !        inp:n,s1(n),u(n,n),u1(n)               !
11706 rem !        int:i9,j9                              !
11708 rem !        out:u1(s1(n))                          !
11710 rem !-----------------------------------------------!
11712   for i9=2 to n
11714     for j9=1 to i9-1
11716       u1(s1(i9))=u1(s1(i9))-u(i9,j9)*u1(s1(j9))
11718     next j9
11720   next i9
11722   for i9=n to1 step -1
11723     if i9=n goto 11730
11724     for j9=n to i9+1 step -1
11726       u1(s1(i9))=u1(s1(i9))-u(i9,j9)*u1(s1(j9))
11728     next j9
11730     u1(s1(i9))=u1(s1(i9))/u(i9,i9)
11732   next i9
11734 return
11736 rem !-----------------------------------------------!
```

Es setzt das Vorhandensein einer mit Hilfe der LR-Zerlegung erzeugten oberen Dreiecksmatrix u(i,j) voraus; dabei wird die Zeilenvertauschung bei der Pivotierung berücksichtigt. Dazu ist dem Programm ein Permutationsvektor zu übergeben, der in BANAMP erzeugt und im Hauptprogramm dimensioniert wurde. Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten. Werden die Elemente des Lösungsvektors in der Reihenfolge gewünscht, in der sie ohne Pivotierung erschienen wären, so ist unter Berücksichtigung des Permutationsvektors auszulesen, also z. B.

for i=1 **to** n : x(i)=u1(s1(i)) : **next** i

Bedeutung der Variablennamen:
u(i,j) LR-zerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; s(i) Permutationsvektor; n Anzahl der Unbekannten; $1 \leqq i \leqq n$; $1 \leqq j \leqq n$.

2.3.4. Cholesky-Verfahren

Die Anwendung des Cholesky-Verfahrens setzt voraus, daß die Matrix des Gleichungssystems symmetrisch und positiv-definit ist. Als grobe Faustregel kann man, wenn man weitere Untersuchungen nicht angestellt hat, folgendes annehmen: Eine Matrix ist jedenfalls dann positiv-definit, wenn sie als Produkt einer Matrix mit ihrer Transponierten gewonnen wurde (Gaußsche Normalform). Normalgleichungssysteme der Ausgleichsrechnung befinden sich immer in der Gaußschen Normalform. Eine irgendwie anders gewonnene symmetrische Matrix ist zunächst mit hoher Wahrscheinlichkeit nicht positiv-definit.
Die Eigenschaft der Symmetrie kann ggf. mit P 2.13 SYMMET geprüft werden. Die Prüfung auf Positiv-Definitheit geschieht zweckmäßig durch die folgenden Programme selbst oder aber mittels P 2.17 DEFINI. Will man an einer Matrix nur die Prüfung auf Positiv-Definitheit vornehmen, so hat man die Matrix vor Ausführung des Programms zu kopieren; denn die Prüfung verändert die Matrix.
Der Lösungsansatz beim Cholesky-Verfahren ähnelt dem bei der LR-Zerlegung. Statt des Ansatzes $\boldsymbol{U} = \boldsymbol{L} * \boldsymbol{R}$ wird jedoch wegen der vorausgesetzten Symmetrie der Ansatz

$$\boldsymbol{U} = \boldsymbol{L}^T * \boldsymbol{L} \tag{2.17}$$

gewählt, also transponierte Dreiecksmatrix $\boldsymbol{L}^T$ mal Dreiecksmatrix $\boldsymbol{L}$, was in jedem Falle zu einer symmetrischen Matrix führt. Analog zur Darstellung im Bild 2.2 ergibt sich damit die Anordnung gemäß **Bild 2.4.**

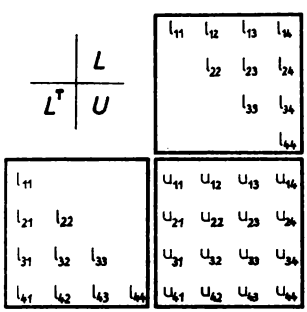


Bild 2.4. Multiplikationsschema für den Algorithmus von *Cholesky*
Da die Matrix L^T die gleichen Elemente wie die Matrix L enthält, genügt die Abspeicherung nur einer Dreiecksmatrix.

Man kann auch hier in verschiedener Weise parkettieren; die Vorgehensweise ist unerheblich, da eine Pivotierung bei diesem Verfahren keinen Effekt ergäbe [Böhm 85].
Die ursprüngliche Matrix wird wie beim LR-Verfahren überschrieben. Die Diagonalelemente werden durch Radizieren gewonnen. Wird dabei ein Radikand negativ, so war die Matrix nicht positiv definit und die Anwendung des Verfahrens war nicht gerechtfertigt. Man beachte, daß ein nicht positiv-definites Gleichungssystem durchaus lösbar sein kann, nur eben nicht mit dem Cholesky-Verfahren. Man werte das Fehlerflag aus, sonst kann es u. U. unbemerkt zu falschen Lösungen kommen.
Die Bearbeitung der Elemente unterhalb der Hauptdiagonalen ist nicht erforderlich; dieser Teil des Speicherbereichs kann während des gesamten Rechenganges unbesetzt bleiben. –
Die explizite Berechnung der Matrizengleichung (2.17) liefert für eine 4 * 4-Matrix folgendes System von Gleichungen zur Berechnung der Matrix $\boldsymbol{L}$, durch deren Elemente die ursprüngliche Matrix überschrieben wird (s. Bild 2.4):

$$u_{11} = l_{11}^2$$
$$u_{12} = l_{11} l_{12}$$
$$u_{13} = l_{11} l_{13}$$
$$u_{14} = l_{11} l_{14}$$

$[u_{21} = l_{12}l_{11}]$
$u_{22} = l_{12}^2 + l_{22}^2$
$u_{23} = l_{12}l_{13} + l_{22}l_{23}$
$u_{24} = l_{12}l_{14} + l_{22}l_{24}$

$[u_{31} = l_{13}l_{11}]$
$[u_{32} = l_{13}l_{12} + l_{23}l_{22}]$
$u_{33} = l_{13}^2 + l_{23}^2 + l_{33}^2$
$u_{34} = l_{13}l_{14} + l_{23}l_{24} + l_{33}l_{34}$

$[u_{41} = l_{11}l_{14}]$
$[u_{42} = l_{12}l_{14} + l_{24}l_{22}]$
$[u_{43} = l_{13}l_{14} + l_{24}l_{23} + l_{34}l_{33}]$
$u_{44} = l_{14}^2 + l_{24}^2 + l_{34}^2 + l_{44}^2$

Die in Klammern stehenden Gleichungen brauchen der Symmetrie wegen nicht berücksichtigt zu werden.
Das **Programm P 2.26 CHOL2D** realisiert die beschriebene Dreieckszerlegung. Zur Lösung von Gleichungssystemen ist **Programm P 2.27 SUBST3** anzuschließen.

Programm P 2.26. CHOL2D. Führt die Dreieckszerlegung an einer symmetrischen, positiv-definiten Matrix aus.

```
11750 rem !-------------------------------------------------!
11752 rem ! P2.26  CHOL2D   /folgt:SUBST3/                  !
11754 rem !        inp:n,u(n,n)                             !
11756 rem !        int:i9,j9,k9,s9                          !
11758 rem !        out:e1%,u(n,n)                           !
11760 rem !-------------------------------------------------!
11762   e1%=0
11764   for i9=1 to n
11766     s9=u(i9,i9)
11768     if i9=1 goto 11776
11770     for j9=1 to i9-1
11772       s9=s9-u(j9,i9)*u(j9,i9)
11770     next j9
11776     if s9<=0 goto 11802
11778     u(i9,i9)=sqr(s9)
11780     if i9=n goto 11798
11782     for j9=i9+1 to n
11784       s9=u(i9,j9)
11786       if i9=1 goto 11794
11788       for k9=1 to i9-1
11790         s9=s9-u(k9,i9)*u(k9,j9)
11792       next k9
11794       u(i9,j9)=s9/u(i9,i9)
11796     next j9
11798   next i9
11800   goto 11804
11802     e1%=1
11804 return
11806 rem !-------------------------------------------------!
```

Es genügt, wenn nur die obere Dreiecksmatrix besetzt wird, auf die Elemente unterhalb der Hauptdiagonalen wird nicht zugegriffen. Die Prüfung auf Symmetrie ist ggf. vor der Anwendung mit P 2.13 SYMMET zu testen. Die Prüfung auf Positiv-Definitheit geschieht durch P 2.17 DEFINI oder durch das vorliegende Programm selbst. Sobald zu radizierende Elemente negativ werden, ist die Bedingung der Positiv-Definitheit nicht erfüllt. Man beachte das Fehlerflag. Zur Lösung von Gleichungssystemen ist im Anschluß an die Ausführung dieses Programms P 2.27 SUBST3 aufzurufen.

Bedeutung der Variablennamen:
u(i,j) Matrix; n Zeilen- bzw. Spaltenzahl der Matrix; $1 \leqq i \leqq n$, $1 \leqq j \leqq n$.

Programm P 2.27. SUBST3. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor aus und setzt das Vorhandensein einer mittels Cholesky-Verfahrens zerlegten oberen Dreiecksmatrix voraus.

```
11810 rem !-------------------------------------------------!
11812 rem ! P2.27  SUBST3   /folgt auf:CHOL2D/              !
11814 rem !        inp:n,u(n,n),u1(n)                       !
11816 rem !        int:i9,j9,s9                             !
11818 rem !        out:u1(n)                                !
11820 rem !-------------------------------------------------!
11822   for i9=1 to n
11824     s9=u1(i9)
11826     if i9=1 goto 11834
11828     for j9=1 to i9-1
11830       s9=s9-u(j9,i9)*u1(j9)
11832     next j9
11834     u1(i9)=s9/u(i9,i9)
11836   next i9
11838   for i9=n to 1 step -1
11840     s9=u1(i9)
11842     if i9=n goto 11850
11844     for j9=i9+1 to n
11846       s9=s9-u(i9,j9)*u1(j9)
11848     next j9
11850     u1(i9)=s9/u(i9,i9)
11852   next i9
11854 return
11856 rem !-------------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Matrix nacheinander mehrere Spaltenvektoren verarbeiten.

Bedeutung der Variablennamen:
u(i,j) choleskyzerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $1 \leqq j \leqq n$.

Das Cholesky-Verfahren setzt zwar die Symmetrie der Matrizen voraus, benötigt aber die unter der Hauptdiagonalen gelegenen Elemente an keiner Stelle des Rechengangs. Um die Speicherkapazität des Rechners voll auszunutzen, kann deshalb die Matrix mit besonderem Vorteil in einem eindimensionalen Feld abgespeichert werden. Wir verwenden folgende Numerierung für die Elemente der eindimensional gespeicherten Matrix:

$$\begin{pmatrix} u(1) & u(2) & u(4) & u(7) \\ & u(3) & u(5) & u(8) \\ & & u(6) & u(9) \\ & & & u(10) \end{pmatrix}$$

Die Zahl z der Elemente einer n*n-Matrix ohne die unter der Hauptdiagonalen gelegenen Elemente ergibt sich als

$$z = \tfrac{1}{2}(n \cdot [n + 1]). \tag{2.18}$$

Tafel 2.1. Speicherplatzbegrenzung für Gleichungsanzahl

Maximal mögliche Anzahl von Gleichungen, die bei gegebener Speicherkapazität des Rechners und Wortlänge der Gleitpunktzahlen verarbeitet werden können

Rechner-speicher	Allg. Matrix + Spaltenvektor Mantissenlänge					Symmetrische Matrix + Spaltenvektor Mantissenlänge				
	3 Byte	4 Byte	5 Byte	6 Byte	7 Byte	3 Byte	4 Byte	5 Byte	6 Byte	7 Byte
4 K	31	28	25	23	23	43	39	35	32	30
8 K	44	39	36	33	31	62	55	50	46	43
16 K	63	56	51	47	44	89	79	72	66	62
32 K	90	80	73	67	63	126	112	103	95	89
64 K	127	113	104	96	90	179	160	146	135	126
128 K	180	161	147	136	127	254	227	207	192	179
256 K	255	228	208	193	180	360	322	294	272	254
512 K	361	323	295	273	255	510	456	416	385	360

Der linke Teil der Tafel enthält die Werte, denen eine vollständige quadratische Matrix und der zugehörige Spaltenvektor zugrunde liegen, der rechte Tafelteil die entsprechenden Werte für symmetrische Matrix und die vorgeschlagene Abspeicherung ohne den Teil unterhalb der Hauptdiagonalen.

Programm P 2.28. CHOL1D. Führt die Dreieckszerlegung an einer symmetrischen, positiv-definiten Matrix aus.

```
11860 rem !-----------------------------------------------!
11862 rem ! P2.28  CHOL1D   /folgt:SUBST4/                !
11864 rem !        inp:n,u(n*(n+1)/2)                     !
11866 rem !        int:i9,j9,k9                           !
11868 rem !        out:e1%,u(n*(n+1)/2)                   !
11870 rem !-----------------------------------------------!
11872   e1%=0
11874   for i9=1 to n
11876     l9=(i9*i9+i9)/2
11878     s9=u(l9)
11880     if i9=1 goto 11888
11882     for j9=1 to i9-1
11884       s9=s9-u((i9*i9-i9)/2+j9)^2
11886     next j9
11888     if s9<=0 goto 11914
11890     u(l9)=sqr(s9)
11892     if i9=n goto 11910
11894     for j9=i9+1 to n
11896       s9=u((j9*j9-j9)/2+i9)
11898       if i9=1 goto 11906
11900       for k9=1 to i9-1
11902         s9=s9-u((i9*i9-i9)/2+k9)*u((j9*j9-j9)/2+k9)
11904       next k9
11906       u((j9*j9-j9)/2+i9)=s9-u(l9)
11908     next j9
11910   next i9
11912   goto 11916
11914     e1%=1
11916 return
11918 rem !-----------------------------------------------!
```

Die Matrix wird in einem eindimensionalen Feld spaltenweise abgespeichert. Der Teil unterhalb der Hauptdiagonale wird nicht gespeichert; dadurch spart diese eindimensionale Variante fast die Hälfte des Speicherplatzes gegenüber P 2.26 CHOL2D ein. Zur Lösung von Gleichungssystemen ist im Anschluß an die Ausführung dieses Unterprogramms das P 2.29 SUBST4 aufzurufen. Die ursprüngliche Matrix wird von der dreieckszerlegten Matrix überschrieben.

Bedeutung der Variablennamen:
u(i) Matrix; n Zeilen- bzw. Spaltenzahl einer gedachten umschließenden quadratischen Matrix; $1 \leqq i \leqq (n*(n+1)/2$.

Programm P 2.29. SUBST4. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mittels Cholesky-Verfahrens erzeugten oberen Dreiecksmatrix u(j) in eindimensionaler Abspeicherung voraus.

```
11930 rem !-----------------------------------------------!
11932 rem ! P2.29  SUBST4   /folgt auf:CHOL1D/            !
11934 rem !        inp:n,u(n*(n+1)/2),u1(n)               !
11936 rem !        int:i9,j9,s9                           !
11938 rem !        out:u1(n)                              !
11940 rem !-----------------------------------------------!
11942   for i9=1 to n
11944     s9=u1(i9)
11946     if i9=1 goto 11954
11948     for j9=1 to i9-1
11950       s9=s9-u((i9*i9-i9)/2+j9)*u1(j9)
11952     next j9
11954     u1(i9)=s9/u((i9*i9+i9)/2)
11956   next i9
11958   for i9=n to 1 step -1
11960     s9=u1(i9)
11962     if i9=n goto 11970
11964     for j9=i9+1 to n
11966       s9=s9-u((j9*j9-j9)/2+i9)*u1(j9)
11968     next j9
11970     u1(i9)=s9/u((i9*i9+i9)/2)
11972   next i9
11974 return
11976 rem !-----------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten. Der Spaltenvektor wird vom Lösungsvektor überschrieben.

Bedeutung der Variablennamen:
u1(i) Spaltenvektor; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $1 \leqq j \leqq (n*(n+1)/2)$.

Die Gesamtzahl der benötigten Feldelemente vergrößert sich gegenüber z aus (2.18) um n für den Spaltenvektor, bei Verfügbarkeit von z Feldelementen können also maximal

$$n_{max} = \tfrac{1}{2}(\sqrt{9 + 8z} - 3) \tag{2.19}$$

Gleichungen mit ebensovielen Unbekannten behandelt werden. Demgegenüber werden für eine voll besetzte quadratische Matrix plus Spaltenvektor $n*n+n$ Zahlen benötigt. **Tafel 2.1** zeigt den Zusammenhang zwischen Speicherkapazität, Wortlänge der Gleitpunktzahlen und Anzahl möglicher Gleichungen für normale, voll besetzte Matrizen und für symme-

trische Matrizen in eindimensionaler Abspeicherung. Die eindimensionale Abspeicherung ermöglicht bei gegebenem Speicherplatz die Abarbeitung von nahezu der sqr(2)-fachen Anzahl von Gleichungen.
Die Zuordnung der Indizes i,j in der zweidimensionalen Darstellung zum Index k in der eindimensionalen Darstellung geschieht durch die Formel

$$k = \tfrac{1}{2}(j^2 + j) + i. \tag{2.20}$$

Das **Programm P 2.28 CHOL1** arbeitet mit der geschilderten eindimensionalen Abspeicherung. Zur Lösung von Gleichungssystemen ist ihm das **Programm P 2.29 SUBST4** anzuschließen.

2.3.5. QR-Zerlegung

Bei den bisher behandelten Methoden zur Lösung von Gleichungssystemen kann sich die Kondition des Systems im Laufe der Berechnung verschlechtern. Die von *Householder* [Hous 64] angegebene sog. QR-Zerlegung vermeidet das und bietet gleichzeitig die Möglichkeit zur Behandlung auch überbestimmter Gleichungssysteme. **Bild 2.5** zeigt das Prinzip der Methode: Die n * m-Matrix $\boldsymbol{U}$ wird durch das Householder-Verfahren in das Produkt aus einer quadratischen n * n-Matrix $\boldsymbol{P}$ und einer oberen m * m-Matrix $\boldsymbol{R}$ zerlegt.
Die Matrix $\boldsymbol{P}$ muß hermitesch ($\boldsymbol{P}^T = \boldsymbol{P}$), unitär ($\boldsymbol{P}^T\boldsymbol{P} = \boldsymbol{E}$) und damit auch involutorisch sein ($\boldsymbol{P} \cdot \boldsymbol{P} = \boldsymbol{E}$).
Die Vorgehensweise für die Transformation von $\boldsymbol{U}$ in die Matrizen $\boldsymbol{P}$ und $\boldsymbol{R}$ skizzieren wir hier nur kurz und verweisen auf die ausführliche Darstellung in *Stoer* [Stoe 83].
Zwei Vektoren $\boldsymbol{x},\boldsymbol{y}$ sollen durch die Matrix $\boldsymbol{P}$ ineinander transformiert werden:

$$\boldsymbol{y} = \boldsymbol{P}\,\boldsymbol{x}. \tag{2.21}$$

Für $\boldsymbol{P}$ wird nach *Householder* der Ansatz verwendet:

$$\boldsymbol{P} = \boldsymbol{E} - 2\,\boldsymbol{w}\,\boldsymbol{w}^T \text{ mit } \boldsymbol{w}^T\boldsymbol{w} = 1. \tag{2.22}$$

Es wird nun versucht, einen Vektor $\boldsymbol{w}$ und damit $\boldsymbol{P}$ so zu bestimmen, daß ein gegebener Vektor, d. h. eine Spalte der ursprünglichen Matrix

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \\ \\ x_n \end{pmatrix} \tag{2.23}$$

in ein zunächst unbekanntes Vielfaches des ersten Achsenvektors

$$\boldsymbol{e}_1 = \begin{pmatrix} 1 \\ 0 \\ \\ \\ 0 \end{pmatrix} \tag{2.24}$$

transformiert wird:

$$s \cdot \boldsymbol{e}_1 = \boldsymbol{P} \cdot \boldsymbol{x} \tag{2.25}$$

s ergibt sich als Euklidnorm des Vektors $\boldsymbol{x}$:

$$\sigma = \boldsymbol{x}^T\boldsymbol{x} = \sum_{i=1}^{n} x_i^2 \qquad s = \text{sgn} \cdot \sqrt{\sigma} \tag{2.26}$$

Das Vorzeichen sgn ist zunächst frei wählbar; man wählt es so, daß Parallelität der Vektoren $\boldsymbol{e}_1$ und $\boldsymbol{x}$ vermieden wird. Aus der Gleichung

$$\boldsymbol{P}\,\boldsymbol{x} = \boldsymbol{x} - 2\,(\boldsymbol{w}^T\boldsymbol{x})\,\boldsymbol{w} = s\,\boldsymbol{e}_1 \text{ und } \boldsymbol{w}^T\boldsymbol{w} = 1 \tag{2.27}$$

folgt für den Vektor $\boldsymbol{w}$

$$\boldsymbol{w} = \frac{\boldsymbol{x} - s\boldsymbol{e}_1}{\|\boldsymbol{x} - s\boldsymbol{e}_1\|} \tag{2.28}$$

Die Transformationsmatrix wird damit

$$\boldsymbol{P} = \boldsymbol{E} - 2\,\boldsymbol{w}\,\boldsymbol{w}^T = \boldsymbol{E} - \beta\,\boldsymbol{u}\,\boldsymbol{u}^T$$

$$\boldsymbol{u} = \boldsymbol{x} - s\,\boldsymbol{e}_1 = \begin{pmatrix} x_1 + \sigma \\ x_2 \\ \\ \\ x_n \end{pmatrix} \tag{2.29}$$

$$\beta = \frac{1}{sx_1 - \sigma}$$

Mit dieser sog. Householder-Transformation wird die ursprüngliche Matrix schrittweise in eine obere Dreiecksmatrix transformiert. Die untere Dreiecksmatrix wird benutzt, um die Elemente der Transformationsmatrix $\boldsymbol{P}$ zu speichern (s. Bild 2.5.).

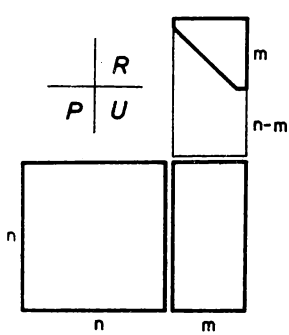


Bild 2.5. Multiplikationsschema für den Algorithmus von *Householder*
m Anzahl der Unbekannten; n Zahl der Gleichungen. Die Matrix *R* kann auf dem gleichen Speicherplatz wie die Matrix *P* abgespeichert werden.

Das **Programm P 2.30 HOUSEH**, das sich an ein ALGOL-Programm in [Stoe 83] anlehnt, realisiert die Berechnung des oben geschilderten Verfahrens. Es ist darüber hinaus zur Lösung überbestimmter Systeme geeignet und gibt für den Fall, daß $n < m$ ist, die Quadratsumme s_2 des Residuums aus (s. auch [Kahm 85]).

Programm P 2.30. HOUSEH. Führt die QR-Zerlegung nach *Householder* aus und erzeugt aus der ursprünglichen Matrix eine obere Dreiecksmatrix sowie den Lösungsvektor.

```
11990 rem !-----------------------------------------------!
11992 rem ! P2.30  HOUSEH                                 !
11994 rem !        inp:d1,m,n,u(n,m+1)                    !
11996 rem !        int:b8,b9,i9,j9,k9,s8,s9               !
11998 rem !        out:e1%,s2,u(i,m+1)                    !
12000 rem !-----------------------------------------------!
12002   e1%=0
12004   for i9=1 to m
12006     b9=0
12008     for j9=i9 to n
12010       b9=b9+u(j9,i9)^2
12012     next j9
12014     if abs(b9)>d1 goto 12020
12016       e1%=1
12018       goto 12084
12020     s8=-sqr(b9)
12022     if u(i9,i9)<>0 then s8=s8*sgn(u(i9,i9))
12024     b8=1/(b9-s8*u(i9,i9))
12026     u(i9,i9)=u(i9,i9)-s8
12028     for j9=i9+1 to m+1
12030       s9=0
12032       for k9=i9 to n
12034         s9=s9+u(k9,i9)*u(k9,j9)
12036       next k9
12038       s9=s9*b8
12040       for k9=i9 to n
12042         u(k9,j9)=u(k9,j9)-u(k9,i9)*s9
12044       next k9
12046     next j9
12048     u(i9,i9)=s8
12050   next i9
12052   if abs(u(m,m))>d1 goto 12058
12054     e1%=1
12056     goto 12084
12058   for i9=m to 1 step -1
12060     if i9=m goto 12068
12062     for j9=m to i9+1 step -1
12064       u(i9,m+1)=u(i9,m+1)-u(i9,j9)*u(j9,m+1)
12066     next j9
12068     u(i9,m+1)=u(i9,m+1)/u(i9,i9)
12070   next i9
12072   if n=m goto 12084
12074   s2=0
12076   for i9=m+1 to n
12078     s2=s2+u(i9,m+1)^2
12080   next i9
12082   s2=sqr(s2)
12084 return
12086 rem !-----------------------------------------------!
```

In den Elementen der unteren Dreiecksmatrix verbleiben zufällige Zwischenergebnisse. Die Zeilenzahl kann größer als die Spaltenzahl sein, d. h., es können überbestimmte Gleichungssysteme behandelt werden. Die Matrix ***U*** muß dazu jedoch quadratisch dimensioniert werden, also Zeilenzahl mal Zeilenzahl.

Bedeutung der Variablennamen:
m Spaltenzahl = Anzahl der Unbekannten; n Zeilenzahl = Anzahl der Gleichungen; u(m,m) Matrix, u(i,m+1) Lösungsvektor; e1% Fehlerflag. e1%=0 bedeutet fehlerfreie Abarbeitung; e1%=1 bedeutet singuläres System. s2 Quadratsumme der Residuen bei überbestimmtem System.

2.3.6. Systeme mit tridiagonalen Matrizen

Systeme mit tridiagonalen Matrizen spielen z. B. bei Splineapproximationen eine wichtige Rolle.
Das System $\boldsymbol{Ux = u}$ wird wie bei der LR-Zerlegung in eine untere und eine obere Dreiecksmatrix zerlegt. In der $\boldsymbol{L}$-Matrix sind dann nur die Hauptdiagonale und die darunter liegende Nebendiagonale mit von null verschiedenen Elementen besetzt. In der $\boldsymbol{R}$-Matrix sind die Hauptdiagonale und die darüber liegende Nebendiagonale besetzt; alle übrigen Elemente sind null. Die $\boldsymbol{R}$-Matrix wird so normiert, daß die Hauptdiagonalelemente zu eins werden; diese Elemente erfordern dann keine Speicherung, und der für die ursprüngliche Matrix erforderliche Speicherplatz genügt für die weitere Bearbeitung. Die Elemente des umgestellten Systems ergeben sich aus

$$\begin{aligned} \boldsymbol{U &= L\,R} \\ \boldsymbol{u &= L\,v.} \end{aligned} \tag{2.30}$$

Die Berechnung geschieht analog dem Verfahren bei der LR-Zerlegung (s. Abschn. 2.2.4). Die Elemente der Matrizen $\boldsymbol{L}$ und $\boldsymbol{R}$ werden wie dort durch Koeffizientenvergleich aus den Matrizenprodukten gewonnen. Wir bezeichnen mit d_i die Diagonalelemente, mit l_i die Elemente der linken und mit r_i die Elemente der rechten Nebendiagonalen sowie mit u_i die Elemente des Spaltenvektors.

$$U = \begin{pmatrix} d_1 & r_1 & & & & \\ l_2 & d_2 & r_2 & & & \\ & l_3 & d_3 & r_3 & & \\ & & & & r_{n-1} & \\ & & & l_n & d_n & \end{pmatrix} \qquad u = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \\ u_n \end{pmatrix} \tag{2.31}$$

Die Elemente der ursprünglichen Matrix werden folgendermaßen verändert, um schließlich in $\boldsymbol{L}$ – Matrix, $\boldsymbol{R}$ – Matrix und Lösungsvektor überzugehen:

$[d_1 := d_1]$ $r_1 := r_1/d_1$		LR-Zerlegung erste Zeile P 2.31 TRIDIA
$d_i := d_i - l_i * r_{i-1}$ $r_i := r_i/d_i$	**for** i = 2 **to** n **step** 1 **for** i = 2 **to** n **step** 1	LR-Zerlegung Rest P 2.31 TRIDIA
$u_i := u_i/d_i$ $u_i := (u_i - l_i * u_{i-1})/d_i$	**for** i = 2 **to** n **step** 1	Vorwärtssubstitution P 2.32 SUBST 5
$[u_n := u_n]$ $u_i := u_i - r_i * u_{i+1}$	**for** i = n-1 **to** 1 **step** − 1	Rückwärtssubstitution P 2.32 SUBST 5

(2.32)

Wir verwenden zur Abspeicherung der Matrix ein zweidimensionales Feld u(m,n), in dem der zweite Index die Zeile in der Matrix angibt. Der erste Index bezeichnet die jeweilige Pa-

rallele zur Hauptdiagonalen. u(0,i) ist die Hauptdiagonale selbst, u(1,i) die rechte (obere) Nebendiagonale, u(2,i) die linke (untere) Nebendiagonale. Die Abspeicherung geschieht folgendermaßen:

$$[u(2,1) = 0]\begin{pmatrix} u(0,1) & u(1,1) & & & \\ u(2,2) & u(0,2) & u(1,2) & & \\ & & & & u(1,n-1) \\ & & & u(2,n) & u(n) \end{pmatrix}[u(1,n) = 0]$$

Alle nicht bezeichneten Elemente haben den Wert null. Man hat demnach zu dimensionieren: **dim** u(2,n), **dim** u1(n). Die Elemente u(2,1) und u(1,n) werden nicht benutzt. Der Ablauf wird durch das **Programm P 2.31 TRIDIA** realisiert. Die anschließende Substitution geschieht mit **Programm P 2.32. SUBST5.** Beispiele zur Programmkontrolle finden sich am Ende des Abschnitts 2.3.

Programm P 2.31. TRIDIA. Zerlegt eine tridiagonale Matrix in eine untere und eine obere bidiagonale Matrix und löst das Gleichungssystem.

```
12100 rem !-----------------------------------------------!
12102 rem ! P2.31  TRIDIA   /folgt:SUBST5/                !
12104 rem !        inp:d1,n,u(2,n)                        !
12106 rem !        int:i9                                 !
12108 rem !        out:e1%,u(2,n)                         !
12110 rem !-----------------------------------------------!
12112   e1%=0
12114   if abs(u(0,1))<d1 goto 12132
12116   u(1,1)=u(1,1)/u(0,1)
12118   if n=1 goto 12134
12120   for i9=2 to n
12122     u(0,i9)=u(0,i9)-u(2,i9)*u(1,i9-1)
12124     if abs(u(0,i9))<d1 goto 12132
12126     u(1,i9)=u(1,i9)/u(0,i9)
12128   next i9
12130   goto 12134
12132     e1%=1
12134 return
12136 rem !-----------------------------------------------!
```

Die Matrix wird in einem zweidimensionalen Feld folgendermaßen abgespeichert: u(0,i) mit i=1 . . . n enthält die Elemente der Hauptdiagonalen, also u(i,i), wenn zweidimensional gespeichert würde; u(1,i) mit i=1...n enthält die Elemente der rechten Nebendiagonalen (u(i,i+1) im zweidimensionalen Fall), wobei u(1,n) nicht verwendet wird; u(2,i) mit i=1 . . . n enthält die Elemente der linken Nebendiagonalen, also u(i,i−1), wenn zweidimensional gespeichert würde, wobei u(2,1) nicht verwendet wird. Es wird nur die LR-Zerlegung der Matrix ausgeführt. Zur Bestimmung des Lösungsvektors ist anschließend P 2.32 SUBST5 aufzurufen.

Bedeutung der Variablennamen:
n Anzahl der Gleichungen = Anzahl der Unbekannten; u(j,i) Matrix, wird von den LR-zerlegten Bidiagonalmatrizen überschrieben; d1 kleinster zulässiger Wert für ein Diagonalelement; e1% Fehlerflag. e1%=1 bedeutet singuläre Matrix bzw. Diagonalelement kleiner als d1. $0 \leqq j \leqq 2$, $1 \leqq i \leqq n$.

Programm P 2.32. SUBST5. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mittels P 2.31. TRIDIA erzeugten oberen Dreiecksmatrix u(j,i) voraus.

```
12132 rem !-------------------------------------------------!
12134 rem ! P2.32  SUBST5   /folgt auf:TRIDIA/               !
12136 rem !         inp:n,u(2,n),u1(n)                       !
12138 rem !         int:i9                                   !
12140 rem !         out:u1(n)                                !
12142 rem !-------------------------------------------------!
12144   u1(1)=u1(1)/u(0,1)
12145   if n=1 goto 12158
12146   for i9=2 to n
12148     u1(i9)=(u1(i9)-u(2,i9)*u1(i9-1))/u(0,i9)
12150   next i9
12152   for i9=n-1 to 1 step -1
12154     u1(i9)=u1(i9)-u(1,i9)*u1(i9+1)
12156   next i9
12158 return
12160 rem !-------------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten.

Bedeutung der Variablennamen:
u(j,i) LR-zerlegte Matrix; u1(i)=Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $0 \leqq j \leqq 2$.

2.3.7. Systeme mit zyklisch-tridiagonalen Matrizen

Die Gleichungssysteme mit zyklisch-tridiagonalen Matrizen werden in erster Linie für die Approximation in sich geschlossener Kurven benötigt, also z. B. für Splineapproximationen entsprechender Graphen. Die zyklisch-tridiagonalen Systeme sichern, daß sich der Lösungsvektor periodisch fortsetzt.

Der Aufbau der Gleichungssysteme ähnelt dem der einfach tridiagonalen Systeme; lediglich die bei jenen fehlenden Elemente l_1 und r_n werden hier in der Matrix berücksichtigt, was die periodische Fortsetzung sichert:

$$U = \begin{pmatrix} d_1 & r_1 & & & & & \\ l_2 & d_2 & r_2 & & & & \\ & l_3 & d_3 & r_3 & & & \\ & & & & & & \\ & & & & & r_{n-1} & \\ r_n & & & & l_n & d_n & \end{pmatrix} \quad u = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \\ \\ u_n \end{pmatrix} \tag{2.33}$$

Alle nicht bezeichneten Elemente haben den Wert null.

Die Dreieckszerlegung $\boldsymbol{U = L\,R}$ erfolgt wie bei den einfach tridiagonalen Matrizen. Allerdings ergibt sich ein komplizierterer Aufbau der Matrizen $\boldsymbol{L}$ und $\boldsymbol{R}$. Bedingt durch das Vorhandensein der Elemente l_1 und r_n außerhalb des Dreierstreifens liefert die Dreieckszerlegung zusätzlich eine untere Zeile in der $\boldsymbol{L}$-Matrix und eine rechte Spalte in der $\boldsymbol{R}$-Matrix. Das spielt bei der Dimensionierung des benötigten Feldes eine Rolle: Während einfach tridiagonale Matrizen in einem Feld der Größe $3*n$ speicherbar sind, erfordern zyklisch-tridiagonale Systeme Felder der Größe von fast $5*n$.

Selbstverständlich können zyklisch-tridiagonale Systeme auch mit den oben behandelten Verfahren für allgemeine Matrizen gelöst werden. Die Ausnutzung der speziellen Matrixstruktur ermöglicht aber sehr viel schnellere und auch fehlerärmere Matrixzerlegung. Bei geeigneter Besetzung der Felder kann auch der benötigte Speicherplatz beträchtlich verringert werden.
Zur Erläuterung des Algorithmus bezeichnen wir die Elemente der Matrizen entsprechend (2.31).
Die Elemente d_i sind die Hauptdiagonalelemente, die l_i die Elemente der linken und die r_i die der rechten Nebendiagonalen. u_i sind die Elemente des Spaltenvektors. Der Spaltenvektor wird wie üblich vom Lösungsvektor überschrieben. Die LR-Zerlegung liefert:

$$\boldsymbol{R} = \begin{pmatrix} 1 & r_1 & & & s_1 \\ & 1 & r_2 & & s_2 \\ & & & & \\ & & & s_{n-2} & \\ & & & r_{n-1} & \\ & & & 1 & \end{pmatrix} \quad \boldsymbol{L} = \begin{pmatrix} d_1 & & & & \\ l_2 & d_2 & & & \\ & l_2 & & & \\ & & & & \\ w_1 & w_2 & & w_{n-2} & l_n & d_n \end{pmatrix} \tag{2.34}$$

s_i sind die Elemente der zusätzlich auftretenden rechten Spalte und w_i die der zusätzlichen waagerechten Zeile in der Matrix $\boldsymbol{L}$. Alle nicht bezeichneten Elemente haben den Wert null, werden aber bei der folgenden Berechnung nicht benötigt. Durch geeignete Speicherorganisation werden sie im **Programm P2.33 TRIDIZ** nicht wirksam. Der folgende Algorithmus arbeitet in der Weise, daß die ursprüngliche Matrix $\boldsymbol{U}$ von den beiden Matrizen $\boldsymbol{L}$ und $\boldsymbol{R}$ überschrieben wird. Die Werte eins auf der Hauptdiagonalen der Matrix $\boldsymbol{L}$ werden nicht gespeichert.

Programm P 2.33. TRIDIZ. Das Programm führt die LR-Zerlegung an zyklisch-tridiagonalen Matrizen aus und nutzt deren schwache Besetzung.

```
12260 rem !-----------------------------------------------!
12262 rem ! P2.33  TRIDIZ   /folgt:SUBST6/                !
12264 rem !         inp:d1,n,u(4,n)                       !
12266 rem !         int:i9,s9                             !
12268 rem !         out:d,e1%,u(4,n)                      !
12270 rem !-----------------------------------------------!
12272   e1%=0
12274   if n=1 goto 12322
12276   if abs(u(0,1))<d1 goto 12332
12278   u(1,1)=u(1,1)/u(0,1)                              ! Anfangselemente
12280   if n>2 goto 12286
12282   u(0,2)=u(0,2)-u(2,2)*u(1,1)
12284   goto 12322
12286   u(3,1)=u(2,1)/u(0,1)
12288   u(4,1)=u(1,n)                                     ! allg. Elemente
12290   for i9=2 to n-1
12292     u(0,i9)=u(0,i9)-u(2,i9)*u(1,i9-1)
12294     if abs(u(0,i9))<d1 goto 12332
12296     if i9=n-1 goto 12304
12298     u(1,i9)=u(1,i9)/u(0,i9)
12300     u(3,i9)=-(u(2,i9)*u(3,i9-1))/u(0,i9)
12302     u(4,i9)=-u(4,i9-1)*u(1,i9-1)
12304   next i9
12306   u(2,n)=u(2,n)-u(4,n-2)*u(1,n-2)                   ! Endelemente
12308   u(1,n-1)=(u(1,n-1)-u(2,n-1)*u(3,n-2))/u(0,n-1)
12310   s9=0
```

```
12312    for i9=1 to n-2
12314      s9=s9+u(4,i9)*u(3,i9)
12316    next i9
12318    u(0,n)=u(0,n)-u(2,n)*u(1,n-1)-s9
12320    if abs(u(0,n))<d1 goto 12332
12322    d=1                                          ! Determinante
12324    for i9=1 to n
12326      d=d*u(0,i9)
12328    next i9
12330    goto 12334
12332      e1%=1
12334  return
12336  rem !-------------------------------------------------!
```

Die Matrix ist in einem Feld der Größe (4,n) gespeichert. Das Schema der Abspeicherung ist im ausführlichen Text erklärt. Zur Lösung von Gleichungssystemen ist anschließend P 2.34 SUBST6 aufzurufen.

Bedeutung der Variablennamen:
d1 kleinster zulässiger Wert eines Diagonalelements, Unterschreitung wird als Singularität angesehen; n Anzahl der Unbekannten; u(j,i) Matrix des Systems; e1% Fehlerflag. e1%=1 bedeutet Singularität und vorzeitigen Rücksprung aus dem Unterprogramm. $1 \leqq i \leqq n$, $0 \leqq j \leqq 4$.

Die Umrechnung von originaler Matrix in die dreieckszerlegten Matrizen geschieht folgendermaßen [Enge 85]:

Anfangsglieder		
$[d_1 := d_1]$		
$r_1 := r_1/d_1$		P 2.33 TRIDIZ bzw.
$s_1 := l_1/d_1$		P 2.35 TRIDIZ2
$w_1 := r_n$		
$u_1 := u_1/d_1$		P 2.34 SUBST6 bzw P 2.36 SUBST7
Allgemeine Glieder		
$d_i := d_i - l_i\, r_{i-1}$	**for** i=2 **to** n-1	
$r_i := r_i/d_i$	**for** i=2 **to** n-2	P 2.33 TRIDIZ bzw.
$s_i := -(l_i\, s_{i-1})/d_i$	**for** i=2 **to** n-2	P 2.35 TRIDIZ2
$w_i := -w_{i-1}\, l_{i-1}$	**for** i=2 **to** n-2	
$u_i := (u_i - u_{i-1}\, l_i)/d_i$	**for** i=2 **to** n-1	P 2.34 SUBST6 bzw. P 2.36 SUBST7
Endglieder		
$l_n := l_n - w_n\, r_{n-2}$		
$r_{n-1} := (r_{n-1} - l_{n-1}\, s_{n-2})/d_{n-1}$	**for** i=1 **to** n-2	P 2.33 TRIDIZ bzw.
$s := s + w_i\, s_i$		P 2.35 TRIDIZ2
$d_n := d_n - l_n\, r_{n-l} - s$		
$t := t + w_i\, u_i$	**for** i=1 **to** n-2	
$u_n := (u_n - l_n\, u_{n-1} - t)/d_n$		
Rücksubstitution		
$[u_n := u_n]$		P 2.34 SUBST6 bzw.
$u_{n-1} := u_{n-1} - r_{n-1} u_n$		P 2.36 SUBST7
$u_i := u_i - r_i\, u_{i+1} - s_i\, u_n$	**for** i=n-2 **to** 1 step -1	(2.35)

Programm P 2.35. TRIDIZ2. Das Programm führt die LR-Zerlegung an zyklisch-tridiagonalen Matrizen aus.

```
12420 rem !-----------------------------------------------!
12422 rem ! P2.35  TRDIZ2   /folgt:SUBST7/                !
12424 rem !        inp:d1,n,u(n,n)                        !
12426 rem !        int:i9,s9                              !
12428 rem !        out:d,e1%,u(n,n)                       !
12430 rem !-----------------------------------------------!
12432   e1%=0
12434   if n=1 goto 12480
12436   if abs(u(1,1))<d1 goto 12490
12438   u(1,2)=u(1,2)/u(1,1)                              ! Anfangselemente
12440   if n>2 goto 12446
12442   u(2,2)=u(2,2)-u(2,1)*u(1,2)
12444   goto 12480
12446   u(1,n)=u(1,n)/u(1,1)
12448   for i9=2 to n-1                                   ! allg. Elemente
12450     u(i9,i9)=u(i9,i9)-u(i9,i9-1)*u(i9-1,i9)
12452     if abs(u(i9,i9))<d1 goto 12490
12454     if i9=n-1 goto 12462
12456     u(i9,i9+1)=u(i9,i9+1)/u(i9,i9)
12458     u(i9,n)=-(u(i9,i9-1)*u(i9-1,n))/u(i9,i9)
12460     u(n,i9)=-u(n,i9-1)*u(i9-1,i9)
12462   next i9
12464   u(n,n-1)=u(n,n-1)-u(n,n-2)*u(n-2,n-1)             ! Endelemente
12466   u(n-1,n)=(u(n-1,n)-u(n-1,n-2)*u(n-2,n))/u(n-1,n-1)
12468   s9=0
12470   for i9=1 to n-2
12472     s9=s9+u(n,i9)*u(i9,n)
12474   next i9
12476   u(n,n)=u(n,n)-u(n,n-1)*u(n-1,n)-s9
12478   if abs(u(n,n))<d1 goto 12490
12480   d=1                                               ! Determinante
12482   for i9=1 to n
12484     d=d*u(i9,i9)
12486   next i9
12488   goto 12492
12490     e1%=1
12492 return
12494 rem !-----------------------------------------------!
```

Bezüglich der Anzahl notwendiger Rechenoperationen wird die schwache Besetzung ausgenutzt. Die Möglichkeit der kompakten Abspeicherung wird im Gegensatz zu P 2.33 TRIDIZ mit Rücksicht auf evtl. einfachere Organisation nicht genutzt. Die Matrix ist in einem zweidimensionalen Feld der Größe n*n gespeichert; die unbenutzten Elemente werden im Laufe des Rechengangs nicht benötigt und nicht verändert. Für das Schema der Abspeicherung s. den ausführlichen Text. Zur Lösung von Gleichungssystemen ist im Anschluß P 2.36 SUBST7 aufzurufen.

Bedeutung der Variablennamen:
d1 kleinster zulässiger Wert eines Diagonalelements, Unterschreitung wird als Singularität angesehen; n Anzahl der Unbekannten; u(n,n) Matrix des Systems; e1% Fehlerflag. e1%=1 bedeutet Singularität und vorzeitigen Rücksprung aus dem Unterprogramm.

Die in [ ] stehenden Operationen werden nicht ausgeführt; die ursprünglichen Werte bleiben einfach stehen.
Im Programm **P 2.35 TRIDIZ2** wird der o. a. Algorithmus abgearbeitet. Die Matrix **U** befindet sich in einem zweidimensionalen Feld der Größe n*n. Bei zunehmendem n wird viel Spei-

cherplatz vergeudet, trotzdem bleibt der Vorteil der höheren Geschwindigkeit und Genauigkeit erhalten. Die Besetzung geschieht folgendermaßen:

$$\begin{pmatrix} u(1,1) & u(1,2) & & & & u(1,n) \\ u(2,1) & u(2,2) & u(2,3) & & & u(2,n) \\ & u(3,2) & u(3,3) & u(3,4) & & u(3,n) \\ & & & & & \\ & & & & & u(n-1,n) \\ u(n,1) & u(n,2) & u(n,3) & & u(n,n-1) & u(n,n) \end{pmatrix}$$

Alle nicht bezeichneten Elemtente sind unbesetzt bzw. beliebig für andere Zwecke nutzbar. Die Rücksubstitution geschieht mit dem Programm **P 2.36 SUBST7.**

Programm P 2.36. SUBST7. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mittels P 2.35 TRIDIZ2 erzeugten oberen Dreiecksmatrix voraus.

```
12500 rem !-------------------------------------------------!
12502 rem ! P2.36  SUBST7   /folgt auf:TRDIZ2/              !
12504 rem !        inp:n,u(n,n),u1(n)                       !
12506 rem !        int:i9,s9                                !
12508 rem !        out:u1(n)                                !
12510 rem !-------------------------------------------------!
12512   u1(1)=u1(1)/u(1,1)
12514   if n=1 goto 12548
12516   if n>2 goto 12524
12518   u1(2)=(u1(2)-u(2,1)*u1(1))/u(2,2)
12520   u1(1)=u1(1)-u(1,2)*u1(2)
12522   goto 12548
12524   for i9=2 to n-1
12526     u1(i9)=(u1(i9)-u1(i9-1)*u(i9,i9-1))/u(i9,i9)
12528   next i9
12530   s9=0
12532   for i9=1 to n-2
12534     s9=s9+u(n,i9)*u1(i9)
12536   next i9
12538   u1(n)=(u1(n)-u(n,n-1)*u1(n-1)-s9)/u(n,n)
12540   u1(n-1)=u1(n-1)-u(n-1,n)*u1(n)
12542   for i9=n-2 to 1 step -1
12544     u1(i9)=u1(i9)-u(i9,i9+1)*u1(i9+1)-u(i9,n)*u1(n)
12546   next i9
12548 return
12550 rem !-------------------------------------------------!
```

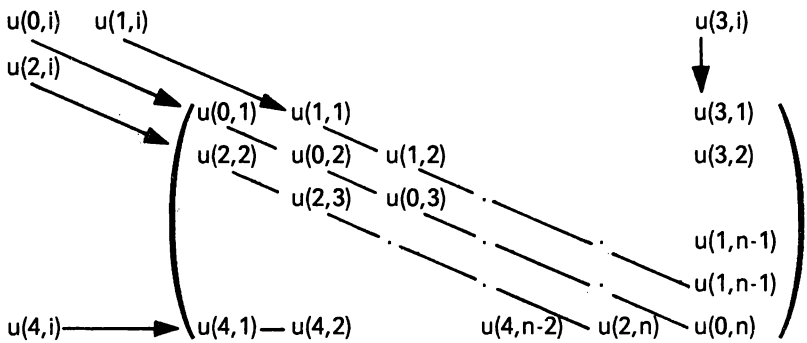
Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten.

Bedeutung der Variablennamen:
u(j,i) LR-zerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $1 \leqq j \leqq n$.

Im P 2.33 TRIDIZ wird analog vorgegangen; es wird jedoch entsprechend zum Vorgehen bei den Tridiagonalmatrizen (s. Abschn. 2.3.6) ein zweidimensionales Feld benutzt, in dem nur die fünf benötigten Vektoren der Reihe nach gespeichert sind. Die Zuordnung ist folgende:

$d_1 \ldots d_n := u(0,1) \ldots u(0,n)$
$r_1 \ldots r_{n-1} := u(1,1) \ldots u(1,n-1)$
$l_2 \ldots l_n := u(2,2) \ldots u(2,n)$
$s_1 \ldots s_{n-2} := u(3,1) \ldots u(3,n-2)$
$w_1 \ldots w_{n-2} := u(4,1) \ldots u(4,n-2)$

bzw. in Matrixform geschrieben:



Die ersten und die letzten Elemente der Vektoren s_i und w_i sind der übersichtlicheren Zuordnung wegen nicht ausgenutzt. Deshalb ist die Anwendung des Programms P 2.33 TRIDIZ erst für $n > 5$ mit Speicherplatzersparnis verbunden. Bei größeren Systemen wird allerdings

Programm P 2.34. SUBST6. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mittels P 2.33. TRIDIZ erzeugten oberen Dreiecksmatrix u(j,i) voraus.

```
12350 rem !-------------------------------------------------!
12352 rem ! P2.34  SUBST6   /folgt auf:TRIDIZ/              !
12354 rem !         inp:n,u(4,n),u1(n)                      !
12356 rem !         int:i9,s9                               !
12358 rem !         out:u1(n)                               !
12360 rem !-------------------------------------------------!
12362   u1(1)=u1(1)/u(0,1)
12364   if n=1 goto 12398
12366   if n>2 goto 12374
12368   u1(2)=(u1(2)-u(2,2)*u1(1))/u(0,2)
12370   u1(1)=u1(1)-u(1,1)*u1(2)
12372   goto 12398
12374   for i9=2 to n-1
12376     u1(i9)=(u1(i9)-u1(i9-1)*u(2,i9))/u(0,i9)
12378   next i9
12380   s9=0
12382   for i9=1 to n-2
12384     s9=s9+u(4,i9)*u1(i9)
12386   next i9
12388   u1(n)=(u1(n)-u(2,n)*u1(n-1)-s9)/u(0,n)
12390   u1(n-1)=u1(n-1)-u(1,n-1)*u1(n)
12392   for i9=n-2 to 1 step -1
12394     u1(i9)=u1(i9)-u(1,i9)*u1(i9+1)-u(3,i9)*u1(n)
12396   next i9
12398 return
12400 rem !-------------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten.
Bedeutung der Variablennamen:
u(j,i) LR-zerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten, $1 \leqq i \leqq n$, $0 \leqq j \leqq 4$.

---

die Einsparung erheblich; der Faktor für die Ersparnis ist asymptotisch n/5. Größere Systeme, also z. B. Splineapproximationen über mehrere hundert Punkte, sind zumeist nur mit derartiger Speicherung behandelbar.
Die Rücksubstitution geschieht mit **Programm P 2.34 SUBST6.** Beispiele zur Programmkontrolle finden sich am Ende des Abschnitts 2.3.

2.3.8. Systeme mit fünfdiagonalen Matrizen

Wir gehen wie bei den tridiagonalen Matrizen vor und zerlegen in eine linke Matrix $\boldsymbol{L}$, die auf der Hauptdiagonalen und auf zwei darunter liegenden Nebendiagonalen besetzt ist, sowie in die rechte Matrix $\boldsymbol{R}$, die auf zwei oberen Nebendiagonalen von null verschiedene Elemente enthält, während ihre Hauptdiagonale durchgehend mit dem Wert eins besetzt ist. Es wird folgendermaßen bezeichnet:

$$\begin{pmatrix} d_1 & e_1 & f_1 & & & & \\ c_2 & d_2 & e_2 & f_2 & & & \\ b_3 & c_3 & d_3 & e_3 & f_3 & & \\ & & & & & f_{n-2} & \\ & & & b_{n-1} & c_{n-1} & d_{n-1} & e_{n-1} \\ & & & & b_n & c_n & d_n \end{pmatrix}$$

Es ergeben sich folgende Zuordnungen (s. [Enge 85]):

Erste Elemente (2.36)
$[d_1 := d_1]$
$e_1 := e_1/d_1$
$f_1 := f_1/d_1$

Zweite Elemente (2.37)
$[c_2 := c_2]$
$d_2 := d_2 - c_2e_1$
$e_2 := (e_2 - c_2f_1)/d_2$
$f_2 := f_2/d_2$

Allgemeine Elemente (2.38)

$c_i := c_i - b_ie_{i-2}$	**for** i=3 **to** n
$d_i := d_i - b_if_{i-2} - c_ie_{i-1}$	**for** i=3 **to** n
$e_i := (e_i - c_if_{i-1})/d_i$	**for** i=3 **to** n−1
$f_i := f_i/d_i$	**for** i=3 **to** n−2

Vorwärtseinsetzen Spaltenvektor (2.39)

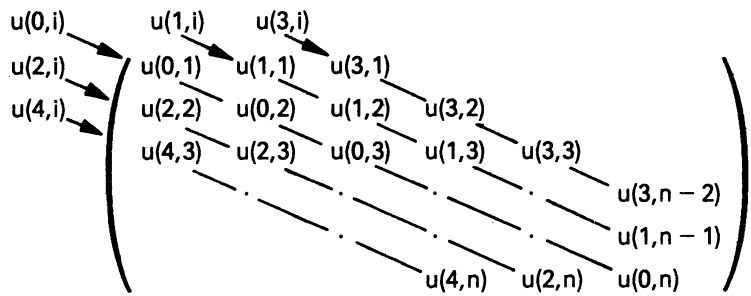
$u_1 := u_1/d_1$
$u_2 := (u_2 - c_2u_1)/d_2$
$u_i := (u_i - b_iu_{i-2} - c_iu_{i-1})/d_i$ **for** i=3 **to** n

---

Rücksubstitution (2.40)

$[u_n := u_n]$
$u_{n-1} := u_{n-1} - e_{n-1}u_n$
$u_i := u_i - e_iu_{i+1} - f_iu_{i+2}$ **for** i=n−2 **to** 1

Die Determinante kann beiläufig als Produkt aller d_i erhalten werden.
Im **Programm P 2.37 PENDIA** wird die Abspeicherung auf einem zweidimensionalen Feld der Größe u(4,n) in der folgenden Weise realisiert:



Programm P 2.37. PENDIA. Das Programm führt die LR-Zerlegung an fünfdiagonalen Matrizen aus bei Ausnutzung der schwachen Besetzung.

```
12570 rem !-----------------------------------------------!
12572 rem ! P2,37  PENDIA   /folgt:SUBST8/                !
12574 rem !        inp:d1,n,u(4,n)                        !
12576 rem !        int:i9                                 !
12578 rem !        out:d,e1%,u(4,n)                       !
12580 rem !-----------------------------------------------!
12582   e1%=0
12584   if abs(u(0,1))<d1 goto 12626
12586   u(1,1)=u(1,1)/u(0,1)                              ! erste Elemente
12588   u(3,1)=u(3,1)/u(0,1)
12590   u(0,2)=u(0,2)-u(2,2)*u(1,1)                       ! zweite Elemente
12592   if abs(u(0,2))<d1 goto 12626
12594   u(1,2)=(u(1,2)-u(2,2)*u(3,1))/u(0,2)
12596   u(3,2)=u(3,2)/u(0,2)
12598   for i9=3 to n                                     ! allg. Elemente
12600     u(2,i9)=u(2,i9)-u(4,i9)*u(1,i9-2)
12602     u(0,i9)=u(0,i9)-u(4,i9)*u(3,i9-2)-u(2,i9)*u(1,i9-1)
12604     if abs(u(0,i9))< d1 goto 12626
12606     if i9>n-1 goto 12614
12608     u(1,i9)=(u(1,i9)-u(2,i9)*u(3,i9-1))/u(0,i9)
12610     if i9>n-2 goto 12614
12612     u(3,i9)=u(3,i9)/u(0,i9)
12614   next i9
12616   d=1                                               ! Determinante
```

```
12618   for i9=1 to n
12620     d=d*u(0,i9)
12622   next i9
12624   goto 12628
12626     e1%=1
12628 return
12630 rem !-------------------------------------------------!
```

Die Matrix ist in einem Feld der Größe (4,n) gespeichert. u(0,i) sind die Hauptdiagonalelemente, u(1,i) die Elemente der ersten rechten Nebendiagonalen, u(2,i) die Elemente der ersten linken Nebendiagonalen, u(3,i) die Elemente der zweiten rechten und u(4,i) die Elemente der zweiten linken Nebendiagonalen. Zur Lösung von Gleichungssystemen ist anschließend P 2.38 SUBST8 aufzurufen.

Bedeutung der Variablennamen:
d1 kleinster zulässiger Wert eines Diagonalelements, Unterschreitung wird als Singularität angesehen; d Determinante der Matrix; n Anzahl der Unbekannten; u(j,i) Matrix des Systems; e1% Fehlerflag. e1%=1 bedeutet Singularität und vorzeitigen Rücksprung aus dem Unterprogramm. $1 \leqq i \leqq n$, $0 \leqq j \leqq 4$.

Programm P 2.38. SUBST8. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mit Hilfe P 2.37 PENDIA erzeugten oberen Dreiecksmatrix voraus.

```
12640 rem !-------------------------------------------------!
12642 rem ! P2.38  SUBST8   /folgt auf:PENDIA/              !
12644 rem !        inp:n,u(4,n),u1(n)!                      !
12646 rem !        int:i9                                   !
12648 rem !        out:u1(n)                                !
12650 rem !-------------------------------------------------!
12652   u1(1)=u1(1)/u(0,1)                                  ! Vorwärtssubstitut.
12654   if n=1 goto 12672
12656   u1(2)=(u1(2)-u(2,2)*u1(1))/u(0,2)
12657   if n=2 goto 12664
12658   for i9=3 to n
12660     u1(i9)=(u1(i9)-u(4,i9)*u1(i9-2)-
                -u(2,i9)*u1(i9-1))/u(0,i9)
12662   next i9
12664   u1(n-1)=u1(n-1)-u(1,n-1)*u1(n)                      ! Rücksubstitution
12665   if n<3 goto 12672
12666   for i9=n-2 to 1 step -1
12668     u1(i9)=u1(i9)-u(1,i9)*u1(i9+1)-u(3,i9)*u1(i9+2)
12670   next i9
12672 return
12674 rem !-------------------------------------------------!
```

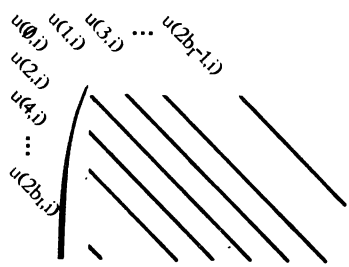
Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten.

Bedeutung der Variablennamen:
u(j,i) LR-zerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $0 \leqq j \leqq 4$.

Einige Plätze bleiben analog zu P 2.31 TRIDIA der besseren Übersichtlichkeit wegen unbesetzt. Die Rücksubstitution erfolgt mit **Programm P 2.38 SUBST8.**

2.3.9. Systeme mit Bandmatrizen

Die Systeme mit Bandmatrizen stellen gegenüber den drei- und fünfdiagonalen Systemen die Erweiterung zu größerer Anzahl von Diagonalen dar. Solche Systeme spielen bei zahlreichen Anwendungen z. B. in der Ausgleichsrechnung oder den Tragwerksberechnungen eine Rolle.
Der rechentechnische Ablauf geschieht wie bei den zuletzt geschilderten Systemen: Die Bandmatrix ***U*** wird in eine linke Matrix ***L*** und eine rechte Matrix ***R*** zerlegt, anschließend erfolgt die Rücksubstitution zur Gewinnung des Lösungsvektors, was nach den vorangegangenen Abschnitten keine Neuigkeiten bietet.
Die hier verwendete Organisation der Speicherung der Matrix entspricht dem für die Drei- und Fünfdiagonalmatrizen verwendeten Prinzip: Es wird ein zweidimensionales Feld verwendet, und dessen erster Index bezeichnet die jeweilige Diagonale. Null bezeichnet die Hauptdiagonale, ungerade Zahlen die rechten (oberen) Nebendiagonalen in zunehmendem Abstand und gerade Zahlen die linken (unteren) Nebendiagonalen:



b1 ist die Anzahl der rechten und der linken Nebendiagonalen. Wenn b1 den Wert n−1 erreicht, dann ist die Matrix voll besetzt; es handelt sich also nicht mehr um eine Bandmatrix. Nichtsdestoweniger funktioniert auch in diesem Falle das angegebene Programm.
Das hier verwendete Prinzip der Speicherung der Matrix ist ungewöhnlich, hat aber den Vorteil, daß man eine gegebene Matrix auf einfache Weise mit verschiedenen Bandbreiten transformieren kann.
Die Berechnung geschieht mit dem **Programm P 2.39 BANDMA.** Für die anschließende Substitution ist **Programm P 2.40 SUBST9** zu verwenden. Beispiele für die Programmtestung sind am Ende des Abschnitts 2.3 angegeben.

Programm P 2.39. BANDMA. Das Programm führt die LR-Zerlegung an Bandmatrizen aus bei Ausnutzung der schwachen Besetzung.

```
12460 rem !-------------------------------------------------!
12462 rem ! P2.39  BANDMA   /folgt:SUBST9/                  !
12464 rem !        inp:b1,d1,n,u(2*b1,n)                    !
12466 rem !        int:b8,b9,i9,j9,k8,k9,l7,l8,l9,s9,t9     !
12468 rem !        out:d,e1%,u(2*b1,n)                      !
12470 rem !-------------------------------------------------!
12472   e1%=0
12474   for i9=1 to n
12476     b9=i9-b1
12478     if b9<1 then b9=1
12480     s9=u(0,i9)
12482     if i9-1<=b9 goto 12492
12484     for j9=b9 to i9-1
12486       k9=i9+i9-j9-j9
12488       s9=s9-u(k9,i9)*u(k9-1,j9)
```

```
12490      next j9
12492      if abs(s9)<d1 goto 12546
12494      u(0,i9)=s9
12496      b8=i9+b1
12498      if b8>n then b8=n
12500      if b8<=i9 goto 12534
12502      for j9=i9+1 to b8
12504        b9=j9-b1
12506        if b9<1 then b9=1
12508        l9=j9+j9-i9-i9
12510        s9=u(l9-1,i9)
12512        t9=u(l9,j9)
12514        if b9>=i9 goto 12528
12516        for k8=b9 to i9-1
12518          l8=i9+i9-k8-k8
12520          l7=j9+j9-k8-k8
12522          s9=s9-u(l8,i9)*u(l7-1,k8)
12524          t9=t9-u(l7,j9)*u(l8-1,k8)
12526        next k8
12528        u(l9-1,i9)=s9
12530        u(l9,j9)=t9/u(0,i9)
12532      next j9
12534    next i9
12536    d=1
12538    for i9=1 to n
12540      d=d*u(0,i9)
12542    next i9
12544    goto 12548
12546      e1%=1
12548  return
12550  rem !-------------------------------------------------!
```

Die Bandbreite ist als Anzahl rechter oder linker Nebendiagonalen an das Programm zu übergeben. Bandbreite null bedeutet ausschließliche Besetzung der Hauptdiagonalen. Die Matrix ist in einem Feld der Größe (2*b1,n) gespeichert. u(0,i) sind die Hauptdiagonalelemente, u(1,i) die Elemente der ersten rechten Nebendiagonalen, u(2,i) die Elemente der ersten linken Nebendiagonalen, u(3,i) die Elemente der zweiten rechten und u(4,i) die Elemente der zweiten linken Nebendiagonalen usw. Zur Lösung von Gleichungssystemen ist anschließend P 2.40 SUBST9 aufzurufen.

Bedeutung der Variablennamen:
b1 Bandbreite der Matrix; d1 kleinster zulässiger Wert eines Diagonalelements, Unterschreitung wird als Singularität angesehen; d Determinante der Matrix; n Anzahl der Unbekannten; u(j,i) Matrix des Systems; e1% Fehlerflag. e1%=1 bedeutet Singularität und vorzeitigen Rücksprung aus dem Unterprogramm. $1 \leqq i \leqq n$, $0 \leqq j \leqq 2*b1$.

Programm P 2.40. SUBST9. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mittels P 2.39. BANDMA erzeugten oberen Dreiecksmatrix voraus.

```
13000 rem !-------------------------------------------------!
13002 rem ! P2.40  SUBST9   /folgt auf:BANDMA/              !
13004 rem !        inp:b1,n,u(2*b1,n),u1(n)                 !
13006 rem !        int:b9,i9,j9,t9                          !
13008 rem !        out:e1%,u1(n)                            !
13010 rem !-------------------------------------------------!
13012   for i9=1 to n
13014     b9=i9-b1
13016     if b9<1 then b9=1
13018     s9=u1(i9)
13020     if b9>=i9 goto 13028
```

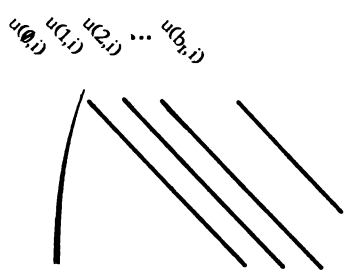
```
13022      for j9=b9 to i9-1
13024        s9=s9-u(i9+i9-j9-j9,i9)*u1(j9)
13026      next j9
13028      u1(i9)=s9
13030    next i9
13032    for i9=n to 1 step -1
13034      b9=i9+b1
13036      if b9>n then b9=n
13038      s9=u1(i9)
13040      if b9<=i9 goto 13048
13042      for j9=i9+1 to b9
13044        s9=s9-u(j9+j9-i9-i9-1,i9)*u1(j9)
13046      next j9
13048      u1(i9)=s9/u(0,i9)
13050    next i9
13052 return
13054 rem !-------------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten.

Bedeutung der Variablennamen:
u(j,i) LR-zerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $0 \leqq j \leqq 4$.

2.3.10. Systeme mit symmetrischen Bandmatrizen

Die Systeme mit symmetrischen Bandmatrizen stellen das Analogon zu den Systemen mit allgemeinen Bandmatrizen dar, setzen aber die Symmetrie zur Hauptdiagonalen sowie Positiv-Definitheit voraus. Damit wird das Cholesky-Verfahren anwendbar, und die Speicherung des unterhalb der Hauptdiagonalen liegenden Matrixteils kann entfallen.
Wir organisieren die Speicherung der Matrix wieder in einem zweidimensionalen Feld. Der erste Index dieses Feldes bezeichnet die Diagonale, der zweite Index die Matrixzeile; null bezeichnet die Hauptdiagonale, eins die erste rechte Nebendiagonale usw. Insgesamt werden b1 Nebendiagonalen verwendet.



Die Realisierung des Verfahrens wird mit **Programm P 2.41 BANDSY** ermöglicht, die Substitution mit **Programm P 2.42 SUBS10.** Beispiele für die Programmtestung finden sich am Ende von Abschnitt 2.3.
Diese Art der Abspeicherung nutzt den verfügbaren Speicherplatz nicht optimal aus, sofern b1 dem Wert von n nahekommt. Sollte sich das in speziellen Fällen als Mangel erweisen, so wäre die Organisation zu ändern; man würde z. B. eindimensional zu speichern haben, was allerdings die Übersichtlichkeit des Programms beeinträchtigt.

Programm P 2.41. BANDSY. Das Programm realisiert die Cholesky-Zerlegung an positiv-definiten Bandmatrizen bei Ausnutzung der schwachen Besetzung.

```
13080 rem !-----------------------------------------------!
13082 rem ! P2.41  BANDSY    /folgt:SUBS10/                !
13084 rem !         inp:b1,d1,n,u(b1,n)                    !
13086 rem !         int:b8,b9,i9,j9,k9                     !
13088 rem !         out:e1%,u(b1,n)                        !
13090 rem !-----------------------------------------------!
13092   e1%=0
13094   for i9=1 to n
13096     b9=i9-b1
13098     if b9<1 then b9=1
13100     s9=u(0,i9)
13101     if i9<=b9 goto 13108
13102     for j9=b9 to i9-1
13104       s9=s9-u(i9-j9,j9)*u(i9-j9,j9)
13106     next j9
13108     if s9<d1 goto 13138
13110     u(0,i9)=sqr(s9)
13112     b8=i9+b1
13114     if b8>n then b8=n
13115     if b8<=i9 goto 13134
13116     for j9=i9+1 to b8
13118       b9=j9-b1
13120       if b9<1 then b9=1
13122       s9=u(j9-i9,i9)
13123       if b9>=i9 goto 13130
13124       for k9=b9 to i9-1
13126         s9=s9-u(i9-k9,k9)*u(j9-k9,k9)
13128       next k9
13130       u(j9-i9,i9)=s9/u(0,i9)
13132     next j9
13134   next i9
13136   goto 13140
13138     e1%=1
13140 return
13142 rem !-----------------------------------------------!
```

Die Bandbreite ist als Anzahl rechter Nebendiagonalen an das Programm zu übergeben. Bandbreite null bedeutet ausschließliche Besetzung der Hauptdiagonalen. Die Matrix ist in einem Feld der Größe (b1,n) oberhalb der Hauptdiagonalen gespeichert. u(0,i) sind die Hauptdiagonalelemente, u(1,i) die Elemente der ersten rechten Nebendiagonalen, u(2,i) die Elemente der zweiten rechten Nebendiagonalen usw. Zur Lösung von Gleichungssystemen ist anschließend P 2.42 SUBS10 aufzurufen.

Bedeutung der Variablennamen:
u(j,i) choleskyzerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; b1 Bandbreite der Matrix; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $0 \leqq j \leqq b1$.

Programm P 2.42. SUBS10. Führt die Vorwärts- und Rückwärtssubstitution für einen Spaltenvektor durch und setzt das Vorhandensein einer mittels P 2.41 BANDSY erzeugten oberen Dreiecksmatrix voraus.

```
13160 rem !-----------------------------------------------!
13162 rem ! P2.42  SUBS10   /folgt auf:BANDSY/             !
13164 rem !        inp:b1,n,u(b1,n),u1(n)                 !
13166 rem !        int:b9,i9,j9                           !
13168 rem !        out:u1(n)                              !
13170 rem !-----------------------------------------------!
13172   for i9=1 to n
13174     b9=i9-b1
13176     if b9<1 then b9=1
13178     s9=u1(i9)
13180     if b9>=i9 goto 13188
13182     for j9=b9 to i9-1
13184       s9=s9-u(i9-j9,j9)*u1(j9)
13186     next j9
13188     u1(i9)=s9/u(0,i9)
13190   next i9
13192   for i9=n to 1 step -1
13194     b9=i9+b1
13196     if b9>n then b9=n
13198     s9=u1(i9)
13200     if b9<=i9 goto 13208
13202     for j9=i9+1 to b9
13204       s9=s9-u(j9-i9,i9)*u1(j9)
13206     next j9
13208     u1(i9)=s9/u(0,i9)
13210   next i9
13212 return
13214 rem !-----------------------------------------------!
```

Das Programm ist vom Hauptprogramm aus gesondert aufzurufen und kann auf der Basis ein und der gleichen Dreiecksmatrix nacheinander mehrere Spaltenvektoren verarbeiten.

Bedeutung der Variablennamen:
u(j,i) choleskyzerlegte Matrix; u1(i) Spaltenvektor, wird vom Lösungsvektor überschrieben; n Anzahl der Unbekannten; $1 \leqq i \leqq n$, $0 \leqq j \leqq b1$.

2.3.11. Nachiteration

Mit zunehmender Größe der Gleichungssysteme werden die Lösungsvektoren durch Abbruchfehler im Rechner immer fehlerhafter (s. Abschn. 2.3.13.1). Hat man für die Berechnung der Lösung bereits optimale Algorithmen verwendet, dann bleibt als Möglichkeit für die weitere Verbesserung der Lösung das Verfahren der Nachiteration. Wir behandeln zwei Verfahren: die Nachiteration in Gesamtschritten sowie die Nachiteration in Einzelschritten, auch Koordinatenrelaxation genannt.

2.3.11.1. Nachiteration in Gesamtschritten

Bei der Nachiteration in Gesamtschritten wird jeweils der gesamte Lösungsvektor verbessert. Der Prozeß wird zyklisch wiederholt bis zum Abbruch wegen ausreichender Genauigkeit oder wegen zu großen Rechenzeitaufwands.
Das ursprüngliche Gleichungssystem laute

$$\boldsymbol{U}\boldsymbol{x} = \boldsymbol{v}. \tag{2.41a}$$

Der Lösungsvektor $\boldsymbol{x}^{(0)}$ wurde mit einem der weiter oben geschilderten Verfahren als nullte Näherung bestimmt und soll nun iterativ verbessert werden. Wegen der Fehlerhaftigkeit des Lösungsvektors wird sich, wenn man die Matrix $\boldsymbol{U}$ mit dem Lösungsvektor $\boldsymbol{x}^{(0)}$ multipliziert, nicht wieder genau der Spaltenvektor $\boldsymbol{v}$ ergeben. Die Abweichungen werden in einem sog. Residuumsvektor r abgelegt:

$$\boldsymbol{r} = \boldsymbol{U}\boldsymbol{x}^{(0)} - \boldsymbol{v} \tag{2.41b}$$

bzw. in ausgeschriebener Form für die Zeile i:

$$r_i = u_{i1}x_1^{(0)} + u_{i2}x_2^{(0)} + \ldots + u_{in}x_n^{(0)} - v_i. \tag{2.42}$$

Löst man das Gleichungssystem

$$\boldsymbol{U}\boldsymbol{k} = \boldsymbol{r}, \tag{2.43}$$

so erhält man als Lösungsvektor $\boldsymbol{k}$ die Korrekturen, die am zuerst bestimmten Lösungsvektor anzubringen sind, damit die erste Näherung $\boldsymbol{x}^{(1)}$ erhalten wird:

$$\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(0)} - \boldsymbol{k} \tag{2.44}$$

bzw.

$$x_i := x_i - k_i \tag{2.45}$$

Mit dem so verbesserten Lösungvektor kann erneut die Operationsfolge entsprechend (2.41), (2.43), (2.44) durchlaufen werden, usw.
Ist die ursprüngliche Matrix bereits LR- oder Cholesky-zerlegt, so braucht zur Berechnung der verbesserten Lösung nur das entsprechende Substitutionsprogramm aufgerufen zu werden. Daher laufen bei Verwendung der in den Abschnitten 2.3.3–2.3.10 angegebenen unterteilten Programme die Nachiterationszyklen wesentlich schneller ab als die erste Lösung des Gleichungssystems.
Es ist darauf hinzuweisen, daß in der geschilderten Form die Nachiteration lediglich dann mit Sicherheit einen Effekt bringt, wenn ihre Berechnung mit erhöhter, also z. B. doppelter Rechnergenauigkeit bzw. Mantissenlänge erfolgt [Fors 71]. Für denjenigen, der die Möglichkeit zur Rechnung mit verschiedenen Genauigkeiten nicht hat, wird nachfolgend ein Verfahren vorgeschlagen, das auch bei Berechnung mit nur einer einzigen Mantissenlänge wesent-

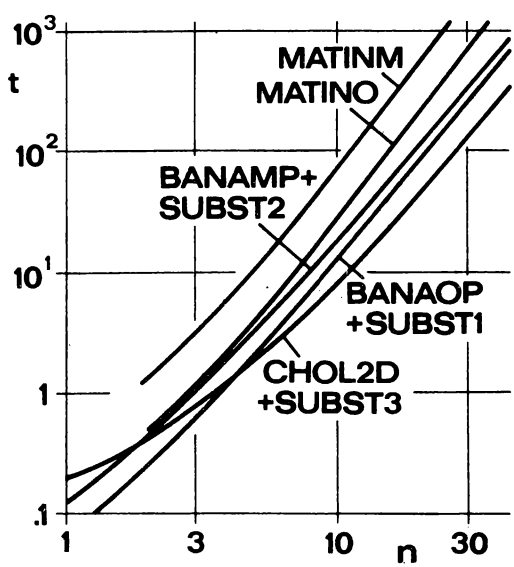


Bild 2.6. Zeitbedarf für verschiedene Verfahren zur Lösung von Gleichungssystemen in Abhängigkeit von der Anzahl n der Unbekannten

Die angegebenen Zeiten in Sekunden gelten für einen 6510-Prozessor mit 1 MHz Taktfrequenz.

BANAOP	LR-Zerlegung nach *Banachiewicz* ohne Pivotierung
BANAMP	LR-Zerlegung nach *Banachiewicz* mit partieller Pivotierung
CHOL2D	Cholesky-Zerlegung
MATINO	Matrixinversion ohne Pivotierung
MATINM	Matrixinversion mit totaler Pivotierung

Programm P 2.43. NACHIG. Nachiteration in Gesamtschritten.

```
13050 rem !-----------------------------------------------!
13052 rem ! P2.43  NACHIG                                 !
13054 rem !         inp: f1,n,u(n,n),u1(n),z1             !
13056 rem !         int: i8,j8,k8,m8                      !
      !                u8(n,n),v8(n),w8(n),x8(n)        !
13058 rem !     out:e1%,m1,u1(n)                          !
13060 rem !-----------------------------------------------!
13062   for i8=1 to n                                     ! System kopieren
13064     for j8=1 to n
13066       u8(i8,j8)=u(i8,j8)
13068     next j8
13070     v8(i8)=u1(i8)
13072   next i8
13074   gosub 11510                                       ! LR-Zerlegung BANAOP
13076   if e1%<>0 goto 13142
13078   gosub 11560                                       ! Substitution SUBST1
13080   m1=1e35
13082   for i8=1 to z1                                    ! Iterationssschleife
13084     for j8=1 to n
13086       w8(j8)=u1(j8)
13088     next j8
13090     m8=0
13092     for j8=1 to n                                   ! Residuum berechnen
13094       u1(j8)=0
13096       for k8=1 to n
13098         u1(j8)=u1(j8)+u8(j8,k8)*w8(k8)
13100       next k8
13102       u1(j8)=u1(j8)-v8(j8)
13104       f8=abs(u1(j8)/v8(j8))                         ! größtes Residuum
13106       if f8>m8 then m8=f8
13108     next j8
13110     if m8=0 then m8=1/256^4
13112     m8=-log(m8)/log(10)
13114     if m8<f1 goto 13136                             ! Ende der Iteration
13116     if m8>=m1 goto 13126
13118     m1=m8
13120     for j8=1 to n
13122       x8(j8)=w8(j8)
13124     next j8
13126     gosub 11560                                     ! Substitution
13128     for j8=1 to n
13130       u1(j8)=w8(j8)-u1(j8)
13132     next j8
13134   next i8
13136   for i8=1 to n
13138     u1(i8)=x8(i8)
13140   next i8
13142 return
13144 rem !-----------------------------------------------!
```

Verbessert eine erste Lösung iterativ und ist auch dann wirksam, wenn die Nachiteration mit der gleichen Rechengenauigkeit wie die Anfangslösung erfolgt.
Die benutzten Felder sind im Hauptprogramm zu dimensionieren. Die Unterprogrammaufrufe zur Matrixzerlegung und zur Substitution sind problemspezifisch einzusetzen. Bei Verwendung von Unterprogrammen mit Abspeicherung der Matrix in eindimensionalen Feldern müßte das Programm sinngemäß abgeändert werden.

Einzugeben sind:

n	Anzahl der Gleichungen bzw. der Unbekannten
u(n,n)	ursprüngliche Matrix, wird zur dreieckszerlegten Matrix
u1(n)	ursprünglicher Spaltenvektor, wird zunächst zum Lösungsvektor und wird zwischenzeitlich als Residuumsvektor und dann als Korrekturvektor benutzt; am Schluß wird der verbesserte Lösungsvektor nach u1(n) kopiert

f1	Genauigkeit, bei deren Erreichung die Iteration abgebrochen wird; f1 wird verglichen mit dem dekadischen Logarithmus des Quotienten aus Residuum und ursprünglichem Spaltenvektorelement, also dem Logarithmus des Relativbetrags des Residuums (z. B. f1=1.0E–8)
z1	Maximalzahl auszuführender Iterationsschritte; nach Ausführung von z1 Iterationsschritten wird unabhängig von der erreichten Genauigkeit abgebrochen
m1	beste bisher erreichte Genauigkeit (log des relativen Größtwertes des Residuums)
m8	aktuell erreichte Genauigkeit (log des relativen Größtwerts des Residuums); damit sich für m8=0 kein Logarithmusfehler ergibt, wird in diesem Falle m8 gleich der Maschinenkonstante gesetzt; die Potenz von 256 ist also rechnerspezifisch zu wählen.

Ferner werden benutzt:

u8(n,n)	Kopie der ursprünglichen Matrix; die Kopie bleibt zur Berechnung der Residuen unverändert erhalten
v8(n)	Kopie des ursprünglichen Spaltenvektors; die Kopie bleibt zur Berechnung der Residuen unverändert erhalten
w8(n)	aktueller Lösungsvektor
x8(n)	bester bisher erreichter Lösungsvektor.

---

lichen Genauigkeitsgewinn bringt – auf Kosten allerdings nicht sehr effektiver Rechenzeitausnutzung. Man läßt dazu den Iterationszyklus wie oben geschildert laufen und stellt nach jedem Durchlauf den Wert m8 des größten Elements im Residuumsvektor fest (Maximumnorm). Ist dieser Wert kleiner als der bis dahin erreichte Kleinstwert, so wird der zugehörige Lösungsvektor in den Bestwertvektor $\boldsymbol{x}$ übernommen – andernfalls behält $\boldsymbol{x}$ seinen bisher innegehabten Wert. Der Ablauf wird vom **Programm P 2.43 NACHIG** realisiert. Im allgemeinen erreicht man bei den mit Zufallszahlen besetzten Testmatrizen (s. Abschn. 2.3.12.2) mit weniger als zehn Iterationszyklen etwa zwei Zehnerpotenzen Genauigkeitsgewinn (**Bild 2.6**).

2.3.11.2. Nachiteration durch Koordinatenrelaxation

Der Vektor der genäherten Lösung wird Gleichung für Gleichung so verändert, daß das Residuum möglichst klein wird. Nach der Behandlung aller n Gleichungen wird von vorn begonnen, sofern das Residuum einen vorgegebenen Schwellwert noch nicht unterschritten hat. Das Verfahren wird auch zyklische Koordinatenrelaxation genannt [Böhm 85][Gast 72]. Für das ursprüngliche System

$$\boldsymbol{U}\boldsymbol{x} = \boldsymbol{v} \tag{2.46}$$

liege eine Näherungslösung $\boldsymbol{x}^{(0)}$ vor. Damit ist der Vektor der Residuen

$$\boldsymbol{r} = \boldsymbol{U}\boldsymbol{x}^{(0)} - \boldsymbol{v} \tag{2.47}$$

bzw. in ausgeschriebener Form

$$r_i = u_{i1}x_1^{(0)} + u_{i2}x_2^{(0)} + \ldots + u_{in}x_n^{(0)} - v_i \tag{2.48}$$

Beim zu schildernden Verfahren wird Gleichung für Gleichung $\boldsymbol{x}^{(0)}$ so verändert, daß der Betrag von $\boldsymbol{r}$ kleiner wird. Für die erste Gleichung des Systems gilt z. B.

$$x^{(1)} = x_1^{(0)} + \Delta x_1 \tag{2.49}$$

$$u_{11}(x_1^{(0)} + \Delta x_1) + u_{12}x_2^{(0)} + \ldots + u_{1n}x_n^{(0)} = v_1, \tag{2.50}$$

wobei also Δx_1 die Korrektur ist, die die Näherung $x_1^{(0)}$ zur Näherung $x_1^{(1)}$ macht.

Löst man die letzte Gleichung nach Δx_1 auf, so ergibt sich

$$\Delta x_1 = \frac{v_1 - u_{11}x_1^{(0)} - u_{12}x_1^{(0)} - \ldots - u_{1n}x_n^{(0)}}{u_{11}}$$

$$= \frac{-r_1}{u_{11}} \tag{2.51}$$

und

$$x_1^{(1)} = x_1^{(0)} - r_1/u_{11} \tag{2.52}$$

Von der Änderung eines einzigen Wertes aus dem Lösungsvektor sind alle Residuen betroffen. Ihre aktualisierten Werte berechnen sich folgendermaßen:

$$r_j^{(1)} = u_{j1}x_1^{(0)} + u_{j1}\Delta x_1 + u_{j2}x_2^{(0)} + \ldots + u_{jn}x_n^{(0)} - v_j$$
$$= r_j^{(0)} + u_{j1}\Delta x_1 = r_j^{(0)} - u_{j1}r_1^{(0)}/u_{11} \tag{2.53}$$

Der Algorithmus wird vom **Programm P 2.44 NACHIE** realisiert. Sobald ein Residuumsvektor erreicht ist, in dem alle $\log_{10}(\text{abs}(r_i/v_i))$ den vorgegebenen Wert f1 unterschreiten, wird die Iteration beendet. Ebenso wird die Rechnung unabhängig von der erreichten Genauigkeit abgebrochen, wenn z1 Iterationszyklen durchlaufen wurden.

Programm P 2.44. NACHIE. Nachiteration in Einzelschritten (Koordinatenrelaxation).

```
13150 rem !-----------------------------------------------!
13152 rem ! P2.44  NACHIE                                 !
13154 rem !        inp:f1,n,u(n,n),u1(n),z1               !
13156 rem !        int:i8,j8,k8,m8,r8(n),s8,u8(n,n),v8(n) !
13158 rem !        out:e1%,m1,u1(n)                       !
13160 rem !-----------------------------------------------!
13162   for i8=1 to n                                     ! Test auf Einhaltung
13164     s8=0                                            ! des Spaltensummen-
13166     for j8=1 to n                                   ! kriteriums
13168       s8=s8+u(i8,j8)
13170     next j8
13172     if s8<2*u(i8,i8) goto 13178
13174       e1%=4
13176       goto 13250
13178   next i8
13180   for i8=1 to n                                     ! System kopieren
13182     for j8=1 to n
13184       u8(i8,j8)=u(i8,j8)
13186     next j8
13188     v8(i8)=u1(i8)
13190   next i8
13192   gosub 11510                                       ! LR-Zerlegung BANAOP
13194   if e1%<>0 goto 13248
13196   gosub 11560                                       ! Substitution SUBST1
13198   for i8=1 to n                                     ! Residuum berechnen
13200     r8(i8)=0
13202     for j8=1 to n
13204       r8(i8)=r8(i8)+u8(i8,j8)*u1(j8)
13206     next j8
13208     r8(i8)=r8(i8)-v8(i8)
13210   next i8
13212   m1=1e35
13214   for i8=1 to z1                                    ! Iterationssschleife
13216     m8=0
13218     for j8=1 to n
13220       f8=abs(r8(j8)/v8(j8))                         ! größtes Residuum
13222       if f8>m8 then m8=f8
13224     next j8
13226     if m8=0 then m8=1/256^4
13228     m8=log(m8)/log(10)
```

```
13230      if m1>m8 then m1=m8
13232      if m8<f1 goto 13250                              ! Ende der Iteration
13234      for j8=1 to n
13236        f8=-r8(j8)/u8(j8,j8)
13238        u1(j8)=u1(j8)+f8                               ! verbesserte Lösung
13240        for k8=1 to n
13242          r8(k8)=r8(k8)+u8(k8,j8)*f8
13244        next k8
13246      next j8
13248    next i8
13250  return
13252  rem !-----------------------------------------------!
```

Verbessert eine genäherte Lösung iterativ, und zwar nacheinander Gleichung für Gleichung. Das Verfahren konvergiert nur dann gegen eine sinnvolle Lösung, wenn die Matrix das Spaltensummenkriterium erfüllt. Auf die Einhaltung dieses Kriteriums wird in einem speziellen Unterprogramm getestet. Ggf. wird Fehlerflag e1%=1 gesetzt und die Abarbeitung beendet. Die benutzten Felder sind im Hauptprogramm zu dimensionieren. Die Unterprogrammaufrufe sind problemspezifisch einzusetzen. Es können alle beschriebenen Programme zur Lösung von Gleichungssystemen verwendet werden, auch solche, die keine getrennte Behandlung der Substitution realisieren. Bei Verwendung von Unterprogrammen mit Abspeicherung der Matrix in eindimensionalen Feldern müßte das vorliegende Programm sinngemäß abgeändert werden, ebenso bei Programmen, die mit einer einzigen erweiterten Matrix arbeiten.

Vom aufrufenden Hauptprogramm sind zu übergeben:

n	Anzahl der Gleichungen bzw. der Unbekannten
u(n,n)	ursprüngliche Matrix, wird i. allg. zur dreieckszerlegten Matrix
u1(n)	ursprünglicher Spaltenvektor, wird zunächst zum Vektor der Näherungslösung und dann zum Vektor der verbesserten Lösungen
f1	Genauigkeit, bei deren Erreichung die Iteration abgebrochen wird; f1 wird verglichen mit dem dekadischen Logarithmus des Quotienten aus Residuum und ursprünglichem Spaltenvektorelement, also dem Logarithmus des Relativbetrags des Residuums (z. B. f1=1.0E−8)
z1	Maximalzahl auszuführender Iterationsschritte; nach Ausführung von z1 Iterationsschritten wird unabhängig von der erreichten Genauigkeit abgebrochen
m1	beste bisher erreichte Genauigkeit (log des relativen Größtwerts des Residuums)
m8	aktuell erreichte Genauigkeit (log des relativen Größtwerts des Residuums); damit sich für m8=0 kein Logarithmusfehler ergibt, wird in diesem Falle m8 gleich der Maschinenkonstante gesetzt; die Potenz von 256 ist demnach rechnerspezifisch zu wählen.

Ferner werden verwendet:

u8(n,n)	Kopie der ursprünglichen Matrix; die Kopie bleibt zur Berechnung der Residuen unverändert erhalten.
v8(n)	Kopie des ursprünglichen Spaltenvektors; die Kopie bleibt zur Berechnung der Residuen unverändert erhalten.

Alle bis zu diesem Kapitel behandelten Verfahren lieferten bei jeder sinnvollen Besetzung der Matrix des Gleichungssystems eine Lösung – bei ungünstiger Besetzung allenfalls von mangelhafter Genauigkeit. Die Relaxationsverfahren konvergieren indessen nur im Falle sog. diagonal-dominanter Matrizen, bei denen also für alle Spalten oder für alle Zeilen gilt

$$|u_{k,k}| > \sum_{\substack{j=1 \\ j \neq k}}^{n} |u_{j,k}|.$$

Je größer $|u_{k,k}|$ gegenüber der Summe der Absolutwerte der übrigen Spaltenelemente ist, um so rascher konvergiert das Verfahren. Ist demgegenüber $|u_{k,k}|$ kleiner als die Absolut-

wertsumme der übrigen Spaltenelemente, so kann nicht damit gerechnet werden, daß ein Relaxationsverfahren gegen eine vernünftige Lösung konvergiert. Der erste Teil von P 2.44 NACHIE testet das Vorhandensein der Diagonaldominanz und setzt ggf. vor dem vorzeitigen Rücksprung das Fehlerflag e1% = 2.
Bei guter Konvergenz kann das Verfahren auch ohne vorherige Verwendung von Programmen zur klassischen Behandlung von Gleichungssystemen verwendet werden. Der Lösungsvektor wird dazu in nullter Näherung mit Nullen oder einer sonstwie plausiblen Näherung besetzt.
Das für große Gleichungssysteme sehr wichtige Gebiet der iterativen Methoden behandeln wir dennoch hier nicht weiter; es ist bezüglich der notwendigen Behandlung der mathematischen Grundlagen relativ aufwendig. Außerdem sind große Systeme, für die solche Verfahren zu bevorzugen wären, nicht sehr zweckmäßig mit BASIC-Programmen zu lösen.

2.3.12. Testung und Bewertung der Programme

Im folgenden Abschnitt werden Beispiele gegeben, mit denen die aufgeführten Programme auf richtigen Lauf getestet werden können. Ursache für Unregelmäßigkeiten könnten Schreibfehler oder Besonderheiten des betreffenden BASIC-Dialekts sein. Danach folgt ein Vergleich der Programme bezüglich ihrer Ausführungsgeschwindigkeit und Genauigkeit.

2.3.12.1. Beispiele zur Programmtestung

Beispiele: allgemeine Systeme
Geeignet für P 2.18 GAJOST, P 2.19 GAOP2D, P 2.20 GAMP2D, P 2.21 GAMP1D, P 2.22 BANAOP + P 2.23 SUBST1, P 2.24 BANAMP + P 2.25 SUBST2, P 2.30 HOUSEH, P 2.14 DETERO, P 2.15 DETERM.

Matrix

$$\begin{pmatrix} 6 & 3 & -2 & 2 & 3 & 1 \\ 1 & 4 & -3 & 4 & 2 & -2 \\ 2 & 3 & -1 & -2 & 9 & 0 \\ 4 & 3 & 0 & 2 & 1 & 2 \\ 3 & 5 & -6 & 6 & 2 & 5 \\ -1 & -2 & 1 & 0 & 3 & 9 \end{pmatrix}$$

Spaltenvektor

$$\begin{pmatrix} 3 \\ 8 \\ -4 \\ 2 \\ -3 \\ 7 \end{pmatrix}$$

Die Lösungen werden für n = 1...6 angegeben. Für n < 6 wird jeweils der links oben beginnende Teil der Matrix und des Spaltenvektors verwendet. Die angegebenen Werte sind die mit der Determinante multiplizierten Werte des Lösungsvektors. Um den Lösungsvektor selbst zu bekommen, hat man also durch die Determinante zu teilen. Das ermöglicht die Angabe der Lösungen ohne Abbruchfehler.

n = 1	n = 2	n = 3	n = 4	n = 5	n = 6
det = 6	det = 21	det = 25	det = 200	det = − 200	det = − 14040
3	− 12	− 5	− 40	40	− 6432
	45	− 95	− 50	4850	62220
		− 195	− 140	− 3060	− 49752
			355	− 5955	− 95586
				− 3200	− 39840
					21000

Beispiele: Matrixinversion
Die angegebenen invertierten Matrizen beziehen sich auf die oben angegebene Beispielmatrix. Die angegebenen Elemente sind wiederum durch die Determinante zu dividieren.

$n = 1 \quad \det = 6 \qquad (1)$

$n = 2 \quad \det = 21 \qquad \begin{pmatrix} 4 & -3 \\ -1 & 6 \end{pmatrix}$

$n = 3 \quad \det = 25 \qquad \begin{pmatrix} 5 & -3 & -1 \\ -5 & -2 & 16 \\ -5 & -12 & 21 \end{pmatrix}$

$n = 4 \quad \det = 200 \qquad \begin{pmatrix} 40 & -24 & -8 & 0 \\ -50 & 20 & 40 & 50 \\ -60 & -24 & -8 & 100 \\ -5 & 18 & -44 & 25 \end{pmatrix}$

$n = 5 \quad \det = -200 \qquad \begin{pmatrix} -40 & 24 & 8 & 0 & 200 \\ 350 & 400 & -100 & -350 & -300 \\ -140 & -256 & 48 & 100 & 200 \\ -345 & -508 & 114 & 325 & 350 \\ -200 & -280 & 40 & 200 & 200 \end{pmatrix}$

$n = 6 \quad \det = -14040 \begin{pmatrix} -3424 & 840 & 720 & 616 & 616 & 88 \\ 6020 & 2640 & -2250 & -6020 & -2510 & 2650 \\ 1176 & -2880 & 540 & -3984 & 3036 & -1572 \\ -2722 & -6180 & 2475 & 1318 & 3073 & -3071 \\ -1720 & -2760 & -360 & 1720 & 1720 & -1760 \\ 1400 & 1920 & -360 & -1400 & -1400 & -200 \end{pmatrix}$

Beispiele: Tridiagonalsysteme
Geeignet für alle Programme zur Lösung von Gleichungssystemen, jedoch speziell für P 2.31 TRIDIA + P 2.32 SUBST5 und P 2.39 BANDMA + P 2.40 SUBST9. Die Beispiele basieren auf der am Anfang des Abschnitts angegebenen Matrix und Spaltenvektor.
Alle Matrixelemente außerhalb des dreidiagonalen Bandes sind Null. Nach Division durch die Determinante sind die angegebenen Vektoren die Lösungsvektoren.

n = 1	n = 2	n = 3	n = 4	n = 5	n = 6
det = 6	det = 21	det = 33	det = 66	det = − 66	det = − 1584
3	− 12	75	114	− 276	− 4824
	45	− 117	− 162	486	8064
		− 219	− 354	732	13368
			66	231	2244
				− 594	− 7656

Beispiele: zyklisch-tridiagonale Systeme
Geeignet für alle allgemeinen Programme, jedoch speziell für P 2.33 TRIDIZ + P 2.34 SUBST6 und P 2.35 TRIDIZ2 + P 2.36 SUBST7. Die Beispiele basieren auf der zuerst angegebenen Matrix und dem dort angegebenen Spaltenvektor. Alle außerhalb der zyklisch-tridiagonalen Matrix liegenden Elemente sind Null. Nach Division durch die Determinante sind die angegebenen Vektoren die Lösungsvektoren.

n = 1	n = 2	n = 3	n = 4	n = 5	n = 6
det = 6	det = 21	det = 25	det = − 46	det = − 102	det = − 1504
3	− 12	− 5	94	− 6	− 5024
	45	− 95	− 78	846	8196
		− 195	50	1398	13264
			− 234	366	2654
				− 936	− 8316
					1044

Beispiele: fünfdiagonale Systeme
Geeignet für alle allgemeinen Programme, jedoch speziell für P 2.37 PENDIA + P 2.38 SUBST8. Die Beispiele basieren auf der zuerst angegebenen Matrix und dem dort angegebenen Spaltenvektor. Alle außerhalb der fünfdiagonalen Matrix liegenden Elemente sind Null. Nach Division durch die Determinante sind die angegebenen Vektoren die Lösungsvektoren.

n = 1	n = 2	n = 3	n = 4	n = 5	n = 6
det = 6	det = 21	det = 25	det = 170	det = 1702	det = 10428
3	− 12	− 5	100	− 3268	− 49868
	45	− 95	− 110	5150	61348
		− 195	− 120	− 4632	− 73224
			335	− 4403	− 82943
				− 3240	− 40570
					21634

Beispiele: siebendiagonale Bandmatrizen
Geeignet für alle allgemeinen Programme, jedoch speziell für P 2.39 BANDMA + P 2.40 SUBST9. Die Beispiele basieren auf der zuerst angegebenen Matrix und dem dort angegebenen Spaltenvektor. Alle außerhalb der siebendiagonalen Matrix liegenden Elemente sind Null. Nach Division durch die Determinante sind die angegebenen Vektoren die Lösungsvektoren.

n = 1	n = 2	n = 3	n = 4	n = 5	n = 6
det = 6	det = 21	det = 25	det = 200	det = 40	det = − 4020
3	− 12	− 5	− 40	− 2000	− 21600
	45	− 95	− 50	7460	79720
		− 195	− 140	− 360	− 11520
			355	− 5490	− 72330
				− 3320	− 37340
					10600

Beispiele: für symmetrische, positiv-definite Systeme
Speziell geeignet für P 2.26 CHOL2D + P 2.27 SUBST3 und für P 2.28 CHOL1D + P 2.29 SUBST4. Wir verwenden die Pascal-Matrix, deren Elemente in leicht zu übersehender Weise wie die Elemente des Pascalschen Dreiecks gebildet werden. Diese Matrix ist symmetrisch und positiv-definit. Ihre Determinante ist für jedes n eins. Die angegebenen Werte sind damit unmittelbar Lösungsvektoren.

Matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 6 & 10 & 15 & 21 \\ 1 & 4 & 10 & 20 & 35 & 56 \\ 1 & 5 & 15 & 35 & 70 & 126 \\ 1 & 6 & 21 & 56 & 126 & 252 \end{pmatrix}$$

Spaltenvektor

$$\begin{pmatrix} 3 \\ 8 \\ -4 \\ 2 \\ -3 \\ 7 \end{pmatrix}$$

n = 1	n = 2	n = 3	n = 4	n = 5	n = 6
3	−2	−19	−54	−118	−237
	5	39	144	400	995
		−17	−122	−506	−1696
			35	291	1481
				−64	−659
					119

Beispiele: Systeme mit symmetrischen, positiv-definiten Bandmatrizen
Speziell geeignet für P 2.41 BANDSY + P 2.42 SUBS10, daneben für alle Programme außer denen für zyklisch-tridiagonale Systeme. Die Matrizen sind aus Bandmatrizen mit nur rechten Nebendiagonalen durch Multiplikation mit der entsprechend transponierten Matrix gewonnen. Die Ursprungsmatrix für die dreidiagonalen Matrizen lautet:

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & \dots \\ 0 & 2 & 3 & 0 & 0 & \dots \\ 0 & 0 & 3 & 4 & 0 & \\ 0 & 0 & 0 & 4 & 5 & \\ & & & & & \dots \\ . & & & & & \dots \end{pmatrix}$$

Der Spaltenvektor wurde auf allen Positionen mit dem Wert eins besetzt. Nachfolgend wird jeweils die Matrix und der mit der Determinante multiplizierte Lösungsvektor angegeben. b1 ist die Anzahl der Nebendiagonalen gemäß der Variablenvereinbarung in den Programmen.

n = 1 b1 = 1 det = 1

$$(1) \qquad (1)$$

n = 2 b1 = 1 det = $4 = (2!)^2$

$$\begin{pmatrix} 4 & 4 \\ 4 & 4 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

n = 3 b1 = 1 det = $36 = (3!)^2$

$$\begin{pmatrix} 5 & 4 & 0 \\ 4 & 13 & 9 \\ 0 & 9 & 9 \end{pmatrix} \qquad \begin{pmatrix} 36 \\ -36 \\ 40 \end{pmatrix}$$

n = 4 b1 = 1 det = $576 = (4!)^2$

$$\begin{pmatrix} 5 & 4 & 0 & 0 \\ 4 & 13 & 9 & 0 \\ 0 & 9 & 25 & 16 \\ 0 & 0 & 16 & 16 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 144 \\ -144 \\ 180 \end{pmatrix}$$

n = 5 b1 = 1 det = $14400 = (5!)^2$

$$\begin{pmatrix} 5 & 4 & 0 & 0 & 0 \\ 4 & 13 & 9 & 0 & 0 \\ 0 & 9 & 25 & 16 & 0 \\ 0 & 0 & 16 & 41 & 25 \\ 0 & 0 & 0 & 15 & 25 \end{pmatrix} \qquad \begin{pmatrix} 14400 \\ -14400 \\ 16000 \\ -16000 \\ 16576 \end{pmatrix}$$

$n = 6 \quad b1 = 1 \quad \det = 518400 = (6!)^2$

$$\begin{pmatrix} 5 & 4 & 0 & 0 & 0 & 0 \\ 4 & 13 & 9 & 0 & 0 & 0 \\ 0 & 9 & 25 & 16 & 0 & 0 \\ 0 & 0 & 16 & 41 & 25 & 0 \\ 0 & 0 & 0 & 25 & 61 & 36 \\ 0 & 0 & 0 & 0 & 36 & 36 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 129600 \\ -129600 \\ 162000 \\ -162000 \\ 176400 \end{pmatrix}$$

$n = 4 \quad b1 = 2 \quad \det = 576 = (4!)^2$

$$\begin{pmatrix} 14 & 13 & 9 & 0 \\ 13 & 29 & 25 & 16 \\ 9 & 25 & 25 & 16 \\ 0 & 16 & 16 & 16 \end{pmatrix} \qquad \begin{pmatrix} 576 \\ -576 \\ 0 \\ 612 \end{pmatrix}$$

$n = 5 \quad b1 = 2 \quad \det = 14400 = (5!)^2$

$$\begin{pmatrix} 14 & 13 & 9 & 0 & 0 \\ 13 & 29 & 25 & 16 & 0 \\ 9 & 25 & 50 & 41 & 25 \\ 0 & 16 & 41 & 41 & 25 \\ 0 & 0 & 25 & 25 & 25 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 3600 \\ -3600 \\ 0 \\ 4176 \end{pmatrix}$$

$n = 6 \quad b1 = 2 \quad t = 518400 = (6!)^2$

$$\begin{pmatrix} 14 & 13 & 9 & 0 & 0 & 0 \\ 13 & 29 & 25 & 16 & 0 & 0 \\ 9 & 25 & 50 & 41 & 25 & 0 \\ 0 & 16 & 41 & 77 & 61 & 36 \\ 0 & 0 & 25 & 61 & 61 & 36 \\ 0 & 0 & 0 & 36 & 36 & 36 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 0 \\ 57600 \\ -57600 \\ 0 \\ 72000 \end{pmatrix}$$

2.3.12.2. Bewertung der Programme

Folgende Programmeigenschaften sollen betrachtet werden: Programmlänge, Speicherplatzbedarf, Ausführungsgeschwindigkeit und Einfluß von Rechenungenauigkeiten.

Programmlänge. Die geringste Programmlänge benötigt das Gauß-Jordan-Verfahren P 2.18 GAJOST mit 12 Zeilen, gefolgt von der Gauß-Elimination ohne Pivotierung P 2.19 GAOP2D mit 16 Zeilen. Die komfortableren Programme benötigen einschließlich Substitution 35 bis 50 Programmzeilen. Das Tridiagonalverfahren braucht 19 Zeilen, die übrigen Verfahren mit Bandmatrizen um 50 Zeilen.

Speicherplatz. Der Speicherplatz bei n Gleichungen beträgt i. allg. für Matrix plus Spaltenvektor $n(n+1)$ Plätze. Wenn, wie es hier geschehen ist, die Feldelemente mit dem Index null nicht verwendet werden, dann sind $(n+1)(n+2)$ Plätze nötig. Eine Ausnahme bildet das Cholesky-Verfahren mit eindimensionaler Abspeicherung P 2.28 CHOL1D, bei dem der Speicherplatz auf die reichliche Hälfte gesenkt ist (s. Taf. 2.1). Für die Bandmatrizen einschließlich Spaltenvektor werden benötigt:

P 2.31. TRIDIA 4n
P 2.33. TRIDIZ 6n
P 2.35. TRDIZ2 (n+1)(n+2)
P 2.37. PENDIA 6n
P 2.39. BANDMA 2n(b1+1)
P 2.41. BANDSY n(b1+2).

Die Feldelemente der Bandmatrizen beginnen, außer bei P 2.35 TRDIZ2, mit dem Index Null.

Ausführungsgeschwindigkeit. Die Ausführungsgeschwindigkeit kann über die Anzahl der Operationen für die Behandlung eines Gleichungssystems ermittelt werden. Hierfür sind in der Fachliteratur formelmäßige Zusammenhänge angegeben (z. B. [Enge85] [Gast72]). Da für die unterschiedlichen Operationen sehr verschiedene Ausführungszeiten benötigt werden, müßten diese Formeln zur Bestimmung praxisgerechter Kennzahlen verfeinert werden. Es wurde hier demgegenüber die Ausführungszeit am realen Prozeß gemessen, womit sich leicht vergleichbare, wenn auch in den Absolutwerten des Zeitbedarfs rechnerspezifische

Tafel 2.2. Zeitbedarf für Lösung von Gleichungssystemen

Programm	a_1	a_2	a_3
GAJOST	.0	.79921e − 1	.13822e − 1
GAOP2D	.0	.25845e − 1	.81320e − 2
GAMP2D	.20615e − 1	.50096e − 1	.82613e − 2
BANAOP	.0	.11194e − 1	.96422e − 2
SUBST1	.13318e − 1	.21692e − 1	.0
BANAMP	.0	.56964e − 1	.10011e − 1
SUBST2	.16642e − 1	.30675e − 1	.0
CHOL2D	.62325e − 1	.19768e − 1	.34441e − 2
SUBST3	.47120e − 1	.18186e − 1	.0
HOUSEH	.16684e + 0	.89397e − 1	.15266e − 1
MATINO	.0	.54616e − 1	.23697e − 1
MATINM	.0	.25068e − 1	.67354e − 1
Mat. Mult.	.0	.18398e − 1	.0
TRIDIA	.58960e − 1	−.12991e − 4	.0
SUBST5	.55434e − 1	−.12909e − 4	.0
TRIDIZ	.12941e + 0	.27410e − 3	.0
SUBST6	.75500e − 1	.69796e − 4	.0
PENDIA	.13711e + 0	.68681e − 3	.0
SUBST8	.78720e − 1	.12100e − 3	.0
BANDMA	.44821e − 1	.21040e − 1	.10824e − 1
SUBST9	.50841e − 1	.23952e − 1	.0

Zeitbedarf $a_1 n + a_2 n^2 + a_3 n^3$
Die Ablaufzeiten wurden im Bereich n = 1...36 experimentell mit einem 6510-Prozessor bei 1 MHz Taktfrequenz ermittelt und durch kubische Polynome ausgeglichen. Angegeben sind die Konstanten dieser Polynome.

Kennzahlen ergeben. Der Test geschah mit einem 6510-Prozessor (C64); eine ungefähre Umrechnung kann auf der Grundlage der Tafel 1.2 ausgeführt werden.
Da die Anzahl der Operationen bei den verwendeten Verfahren durch Polynome mit linearen, quadratischen und kubischen Potenzen von n exakt beschreibbar ist, konnten auch die experimentellen Ergebnisse mit guter Genauigkeit durch solche Polynome approximiert werden. Die Polynomkoeffizienten sind in **Tafel 2.2** angegeben.
Für große Systeme ($n > 20$) spielt nur das kubische Glied noch eine merkliche Rolle. Es ist beim Gauß-Jordan- und Householder-Verfahren am größten, bei den Gauß- und Banachiewicz-Verfahren etwa 50 % kleiner. Sehr günstig ist bei größeren Systemen das Cholesky-Verfahren; es ist in diesen Fällen etwa 4.5mal schneller als P 2.30 HOUSEH und etwa 2.5mal schneller als P 2.20 GAMP2D (Bild 2.6).
Wenn nur ein einziges System zu behandeln ist, dann ist die Gauß-Elimination schneller als das Banachiewicz-Verfahren **(Bild 2.7)**. Allerdings kann das rechnerabhängig auch anders sein, und zwar, wenn die Zugriffe auf Feldelemente relativ zeitaufwendiger sind, was z. B. beim Lesen der Feldelemente von Massenspeichern der Fall ist. Anhand des Bildes 2.6 soll auch noch einmal darauf hingewiesen werden, daß die Verwendung der Matrixinversion und anschließende Matrizenmultiplikation für die Lösung von Gleichungssystemen mit mehreren Spaltenvektoren im Zeitaufwand sehr ungünstig ist. Das Verfahren ist etwa 12mal langsamer als P 2.26 CHOL2D und etwa 6mal langsamer als P 2.22 BANAOP. Die Inversion ist also nur zu vertreten, wenn die invertierte Matrix im weiteren Verlauf des Programms noch für andere Zwecke verwendet werden soll.

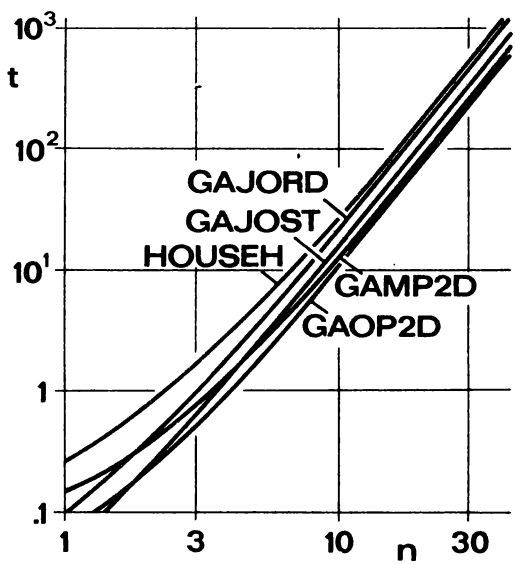


Bild 2.7. Zeitbedarf für verschiedene Verfahren zur Lösung von Gleichungssystemen in Abhängigkeit von der Anzahl n der Unbekannten

Die angegebenen Zeiten in Sekunden gelten für einen 6510-Prozessor mit 1 MHz Taktfrequenz.
GAJORD Gauß-Jordan-Verfahren
GAJOST modifiziertes Gauß-Jordan-Verfahren
GAOP2D Gauß-Elimination ohne Pivotierung
GAMP2D Gauß-Elimination mit partieller Pivotierung
HOUSEH LU-Zerlegung nach *Householder*

Die partielle Pivotierung schlägt sich nur in den Koeffizienten der quadratischen Potenzen nieder. Bei größeren Systemen wirkt sie daher nicht wesentlich verlangsamend. Dagegen zeigen sowohl Bild 2.6 als auch die Zahlenwerte der Koeffizienten in Tafel 2.2, daß die in MATINM verwendete totale Pivotierung gegenüber der partiellen Pivotierung sehr viel Zeit braucht.
Die drei- und fünfdiagonalen Systeme ergeben beim Polynomausgleich des Zeitbedarfs nur lineare Glieder. Die sehr kleinen negativen quadratischen Glieder sind praktisch vernachlässigbar **(Bild 2.8)**. Bei den Bandmatrizen steigt die Ausführungszeit bis zu dem der vollen Besetzung entsprechenden n mit der dritten Potenz an; danach wächst die Ausführungszeit nur noch linear **(Bild 2.9)**.

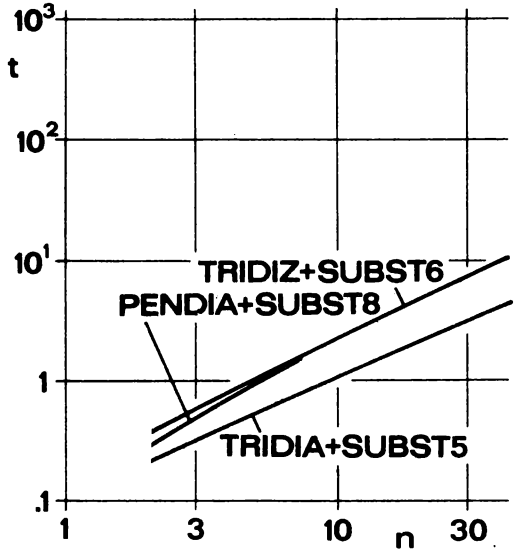


Bild 2.8. Zeitbedarf für Verfahren zur Lösung von Gleichungssystemen mit drei- und fünfdiagonalen Matrizen in Abhängigkeit von der Anzahl n der Unbekannten

Die angegebenen Zeiten in Sekunden gelten für einen 6510-Prozessor mit 1 MHz Taktfrequenz. Dargestellt sind die Gesamtzeiten für Dreieckszerlegung und Substitution. Zerlegung und Substitution sind in allen drei Fällen ziemlich genau mit je 50% am Gesamtzeitbedarf beteiligt.

TRIDIA Systeme mit tridiagonalen Matrizen
TRIDIZ Systeme mit zyklisch-tridiagonalen Matrizen
PENDIA Systeme mit fünfdiagonalen Matrizen

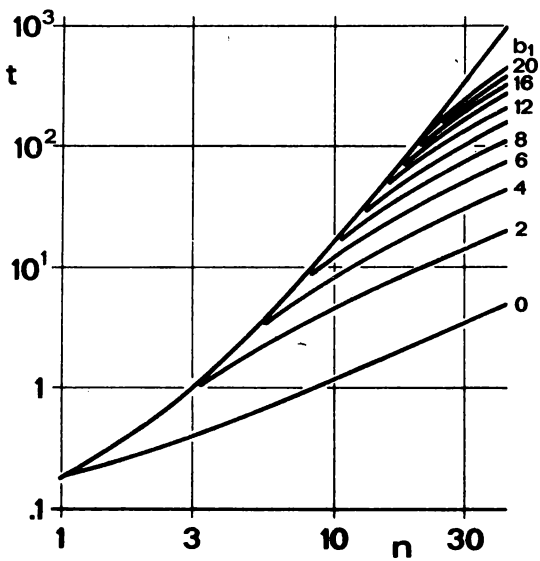


Bild 2.9. Zeitbedarf für die Lösung von Gleichungssystemen mit Bandmatrizen mittels des Unterprogramms BANDMA in Abhängigkeit von der Anzahl n der Unbekannten und der Bandbreite b1

Die angegebenen Zeiten in Sekunden gelten für einen 6510-Prozessor mit 1 MHz Taktfrequenz. Die oberste Kurve ist die Grenzkurve für volle Besetzung. Eine Matrix der Bandbreite b1 ist voll besetzt, wenn sie die Größe (b1+1) (b1+1) hat.

Fehlerverhalten. Das Fehlerverhalten der verschiedenen Verfahren wurde ebenfalls mit Hilfe numerischer Experimente untersucht. Als Testmatrizen wurden verwendet:

- Zufallsmatrizen, bei denen Matrix und Spaltenvektor mit zwischen -0.5 und 0.5 gleichverteilten Zufallszahlen besetzt sind
- Hilbert-Matrizen mit Spaltenvektorelementen = eins.

Die Hilbert-Matrix besteht aus Elementen mit dem Wert
$e_{i,j} = 1/(i+j-1)$, also

$$\begin{pmatrix} 1/1 & 1/2 & 1/3 & & 1/n \\ 1/2 & 1/3 & 1/4 & & \\ . & & & & \\ . & & & & \\ 1/n & & & & \end{pmatrix}$$

Diese Matrix ist relativ schlecht konditioniert und ermöglicht einen strengen Test für die Genauigkeit der Lösung von Gleichungssystemen.
Bei der Testung wurde wiederum mit einem 6510-Prozessor gearbeitet. Das verwendete BASIC hatte eine Mantissenlänge von 4 Bytes; der dekadische Logarithmus des kleinsten möglichen relativen Fehlers betrug damit −9.63295986. Untersucht wurden Gleichungssysteme mit $1 \leqq n \leqq 70$. Die Programme zur Bestimmung der Fehler arbeiteten in folgender Weise: Matrix und Spaltenvektor wurden in Hilfsfelder kopiert; danach wurde jeweils das Gleichungssystem gelöst. Der berechnete Lösungsvektor wurde mit der ursprünglichen Matrix multipliziert. Damit hätte sich der ursprüngliche Spaltenvektor ergeben müssen. Der Größtwert des dekadischen Logarithmus des Absolutbetrags der relativen Abweichung zwischen den beiden Vektoren (Maximumnorm) wurde als Maß für die Qualität des Lösungsverfahrens benutzt. Sein Wert wird mit f bezeichnet und ist in Abhängigkeit von n bestimmt worden. Naturgemäß ergaben sich starke Schwankungen, aber es zeigte sich bei den Hilbert-Matrizen ein deutliches "Sättigungsverhalten"; von einem gewissen n an nehmen die Fehler nicht mehr ersichtlich zu **(Bild 2.10)**. Die f(n) wurden daher durch folgende empirische Funktion approximiert:

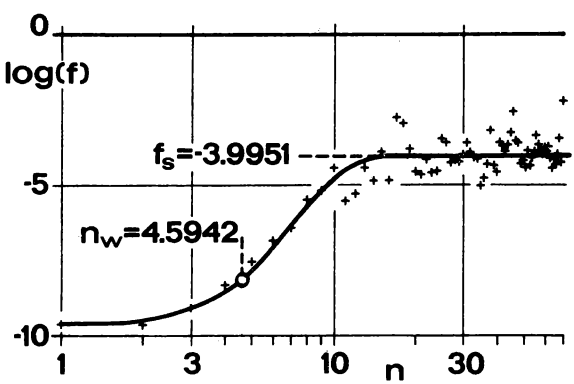


Bild 2.10. Fehlerverhalten bei der Lösung von Gleichungssystemen mit Hilbert-Matrizen und Spaltenvektorelementen = 1

Dargestellt ist die Maximumnorm des Vektors der Relativfehler. Die Werte gelten für Rechnung mit 4 Byte Mantissenlänge. Das verwendete Rechenverfahren ist BANAOP+SUBST1. Die verwendete empirische Approximationsfunktion ist das Integral über die Rayleigh-Rice-Verteilung.

$$f(n) = f_s - (f_s - f_a)\exp(-.5(n-1)^2/n_w^2). \tag{2.54}$$

Darin sind
- f(n) approximierter Fehlerwert als Funktion von n
- f_a kleinster möglicher Fehler (Anfangswert) = −9.63295986
- f_s größter mittlerer Fehler (Sättigungswert) – durch Ausgleichung zu bestimmender Parameter
- n_w Wendepunktsabszisse = Maß für die Geschwindigkeit des Anstiegs der Fehler – durch Ausgleichung zu bestimmender Parameter.

Die durch nichtlineare Ausgleichung bestimmten Parameter sind in **Tafel 2.3** aufgelistet. Es zeigt sich, daß selbstverständlich bei den Hilbert-Matrizen die Pivotierung wegen der bereits gegebenen Regularität keine Vorteile bringt. Auch das diesbezüglich optimal arbeitende Householder-Verfahren erweist sich eher als ungünstiger. Ganz ungünstig verhält sich das Gauß-Jordan-Verfahren gegenüber den Hilbert-Matrizen. Etwa von n = 10 an sind hier die Fehler fast so groß wie die Ergebnisse.

Tafel 2.3. Genauigkeitsparameter für Lösung von Gleichungssystemen

Genauigkeit verschiedener Methoden zur Lösung von Gleichungssystemen mit Hilbert-Matrizen und Spaltenvektorelementen = 1

Programm	n_w	$\log(f_s)$	s
GAJOST	5.8676	−0.8650	0.7379
GAOP2D	4.4875	−4.1431	0.5867
GAMP2D	4.1809	−4.3535	0.3800
BANAOP	4.4360	−4.1271	0.5341
BANAMP	4.1211	−4.3700	0.3293
CHOL2D	3.6665	−6.4545	1.5286
HOUSEH	3.6975	−3.4415	0.5101

Angegeben sind die Konstanten n_w und f_s einer für die Approximation verwendeten Exponentialfunktion (s. Text) sowie die Reststandardabweichung s gegenüber der ausgleichenden Funktion. Die Werte wurden im Bereich n = 1...70 experimentell ermittelt. Die Mantissenlänge betrug 4 Byte, der dekadische Logarithmus der äußersten erreichbaren Genauigkeit ist damit −9.6329568.

Bei den Zufallsmatrizen ist ein Sättigungsverhalten nicht zu beobachten. Die Fehler nehmen mit der Anzahl n der Gleichungen stetig zu. Folgende empirische Funktion erwies sich für die Approximation des Fehlerverhaltens als geeignet:

$$\log_{10}(f) = -9.63295986 + a\,[\log_{10}(n)]^{1.4}. \tag{2.55}$$

a ist der durch Ausgleichung zu bestimmende Parameter. **Bild 2.11** zeigt Fehlerwerte und approximierende Funktion für das Verfahren P 2.22 BANAOP + P 2.23 SUBST1. Der Zahlen-

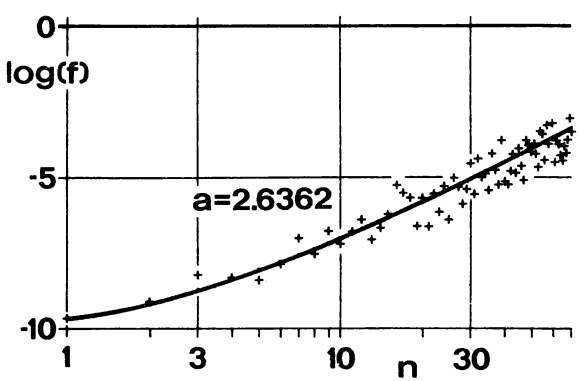


Bild 2.11. Fehlerverhalten bei der Lösung von Gleichungssystemen mit Zufallsmatrizen

Matrix und Spaltenvektor sind mit zwischen −0.5 und 0.5 gleichverteilten Zufallszahlen besetzt. Dargestellt ist die Maximumnorm des Vektors der Relativfehler. Die Werte gelten für Rechnung mit 4 Byte Mantissenlänge. Das verwendete Rechenverfahren ist BANAOP+SUBST1. Die in diesem Bild gezeigten Meßpunkte sind Mittelwerte aus zwölf Durchläufen. Die verwendete empirische Approximationsfunktion basiert auf dem dekadischen Logarithmus von n zur Potenz 1.4.

wert von a ist anschaulich leicht zu deuten: Es ist der dekadische Logarithmus des Faktors, um den der mittlere Fehler bei einem System mit n = 10 Unbekannten größer ist als die Grenzgenauigkeit des Rechners. **Tafel 2.4** enthält die Parameter a für die verschiedenen untersuchten Verfahren zur Lösung von Gleichungssystemen. Gegenüber der Situation bei den Hilbert-Matrizen zeigt sich hier der Vorteil der Pivotierung deutlich; die Pivotierung bringt bei n = 10 einen Genauigkeitsgewinn von ungefähr einer Zehnerpotenz. Das Householder-Verfahren verhält sich hier zwar gut, aber nicht sehr gut. Auch in Anbetracht seiner schlechten Laufzeiteigenschaften sollte es deshalb nur für überbestimmte Systeme angewendet werden. Das Gauß-Jordan-Verfahren ist offenbar nur bei den sehr schlecht konditionierten Hilbert-Matrizen so ungünstig. Bei den normal konditionierten Zufallsmatrizen entspricht seine Genauigkeit der der Gauß- und Banachiewicz-Verfahren.
Das Cholesky-Verfahren konnte mit Hilbert-Matrizen nicht geprüft werden. Aufgrund von Rechenungenauigkeiten wurden Hilbert-Matrizen mit n > 10 als nicht positiv-definit zurück-

Tafel 2.4. Fehlerparameter für Lösung von Gleichungssystemen

Parameter a einer empirischen Fehlerfunktion für die Restfehler bei der Lösung von Gleichungssystemen mit zwischen −0.5 und +0.5 gleichverteilten Zufallszahlen in Matrix und Spaltenvektor

Programm	a	s
GAJOST	2.641	0.996
GAOP2D	2.749	1.051
GAMP2D	1.792	0.528
BANAOP		
Spv=RND	2.636	1.446
Spv=1.0	1.971	0.838 [1]
BANAMP	1.724	0.801
CHOL2D	2.500	1.492 [2]
BANAOP	2.255	1.469 [2]
HOUSEH	2.055	1.003
TRIDIA	1.476	0.984
TRIDIZ	1.885	1.241
PENDIA	1.952	1.127
NACHIG		[3]
z=1	2.636	1.446
z=3	1.480	0.961
z=10	1.230	0.879
z=100	1.018	0.752
BANDMA		
b1=0	0.057	0.210
b1=1	1.798	1.334
b1=2	1.995	1.258
b1=4	2.213	1.255
b1=6	2.352	1.239
b1=8	2.455	1.286
b1=10	2.539	1.404
b1=12	2.609	1.086
b1=14	2.670	1.430
b1=16	2.724	1.347
b1=18	2.772	1.180
b1=20	2.816	1.569

a ist der Logarithmus des Faktors, um den der mittlere Fehler bei einem System mit zehn Unbekannten größer ist als die Grenzgenauigkeit des Rechners. s ist die Standardabweichung der Meßwerte gegenüber der Approximation.

[1] Spaltenvektorelemente im Gegensatz zu allen anderen Fällen = 1

[2] positiv-definite Zufallsmatrix

[3] Matrixzerlegung mit BANAOP

gewiesen. Zufallsmatrizen sind demgegenüber von vornherein nicht positiv-definit. Es wurden deshalb Zufallsmatrizen mit ihren Transponierten multipliziert. Zum Vergleich wurden diese nun positiv-definiten Zufallsmatrizen mit P 2.22 BANAOP verarbeitet. Es zeigt sich kein deutlicher Unterschied in der Genauigkeit, was wegen des grundsätzlich gleichen Ablaufs der Matrixzerlegung auch nicht zu erwarten war.

Systeme mit Bandmatrizen werden mit vergleichsweise hoher Genauigkeit abgearbeitet, be-

dingt auch durch die geringere Anzahl beteiligter Zufallselemente. Mit zunehmender Bandbreite sinkt die Genauigkeit auf die Werte, die auch von der Gauß-Zerlegung erreicht werden.

Unter P 2.43 NACHIG sind in Tafel 2.4 die Genauigkeitswerte aufgeführt, die bei Nachiteration in Gesamtschritten erreicht werden. Die Angabe bezieht sich auf das geschilderte Verfahren mit mehrfachem Durchlauf und Auswahl der lösungsverbessernden Korrekturen. Die Matrix wurde mit P 2.22 BANAOP zerlegt. Der Parameter z ist die Zahl der Durchläufe durch das Substitutionsprogramm. $z = 1$ bedeutet einen Durchlauf, also normalen Ablauf ohne Nachiteration. Man erkennt, daß der Übergang von einem auf drei Substitutionsläufe bei $n = 10$ im Mittel mehr als eine Zehnerpotenz Genauigkeitsgewinn bringt. Die weitere Vergrößerung der Zahl der Substitutionsläufe verbessert die Genauigkeit dann nur noch vergleichsweise wenig.

In diesem Zusammenhang ist eine Bemerkung zum Wert der Konditionsmaße für Marizen [Enge 85] bei der Behandlung von Gleichungssystemen angebracht. Zunächst ist die Bestimmung der Konditionsmaße rechentechnisch aufwendiger als die Bestimmung des Lösungsvektors für ein Gleichungssystem, indem z. B. Matrizen zu invertieren oder Spektralnormen zu berechnen sind. Dann zeigt sich aber auch, daß diese Konditionsmaße nur im statistischen Mittel ein Maß für die Richtigkeit der Lösung von Gleichungssystemen sind; u. U. ergeben sich trotz günstiger Konditionsmaße sehr fehlerhafte Lösungen. Die Konditionsmaße können also für den Fall von irgendwie gearteten Matrizen nur die statistische Sicherheit für die Behauptung vergrößern, die Lösung sei ausreichend gut. Programme wie P 2.43 NACHIG weisen dagegen jede zu ungenaue Lösung mit Sicherheit zurück – allerdings auf Kosten verdoppelten Speicherplatzbedarfs.

Die hier angegebenen Ergebnisse numerischer Experimente können selbstverständlich nur ungefähre Abschätzungen ermöglichen; die angegebenen Zahlenwerte sind nicht beliebig verallgemeinerungsfähig, weil sie an spezielle Matrizen gebunden sind. Bezüglich allgemeinerer Aussagen sei auf die Monografie von *Wilkinson* [Wilk 69] verwiesen.

Im Ergebnis der Fehleruntersuchung muß festgestellt werden, daß Gleichungssysteme mit Matrizen, über die weiter nichts bekannt ist, immer für Überraschungen sorgen. Es kommen mit endlichen Wahrscheinlichkeiten Ergebnisse vor, die vom wahren Wert um ein vielfaches abweichen. Im Maschinenbau gilt der Grundsatz, daß man nur fertigen kann, was man auch prüfen kann. In unserem Zusammenhang sollte man die Regel akzeptieren, daß man nur Systeme gelöst hat, deren Lösung man auch auf Richtigkeit geprüft hat. Deshalb muß ein vollständiges Programm die ursprünglichen Matrizen speichern und im Anschluß an die Lösung die Kontrolle auf Restfehler ausführen. Das Programm P 2.43 NACHIG ist für ein solches Vorgehen geeignet. Als Unterprogramme für die eigentliche Behandlung der Gleichungssysteme sind alle zweiteiligen Programme geeignet, also P 2.22 BANAOP + P 2.23 SUBST1, P 2.24 BANAMP + P 2.25 SUBST2 sowie P 2.26 CHOL2D + P 2.27 SUBST3. Bei BANAMP ist das Auslesen des Lösungsvektors unter Berücksichtigung des Permutationsvektors zu realisieren; ferner sind die Unterprogrammsprünge anzupassen, wenn man nicht bei P 2.22 BANAOP bleibt. Die Verwendung mit den Bandsystemen ist ebenfalls möglich; hier muß indessen die veränderte Matrixstruktur berücksichtigt werden.

Als Parameter für den Ablauf von P 2.43 NACHIG sind anzugeben die maximale Anzahl zu durchlaufender Substitutionszyklen sowie der maximal zulässige Fehler als dekadischer Logarithmus des Absolutwerts des maximalen relativen Fehlers. Kann der vorgeschriebene Fehler bei der angegebenen Anzahl zu durchlaufender Zyklen nicht unterboten werden, so wird im Fehlerflag e1% der Wert zwei gesetzt, und man hat dann ggf. manuell oder vom Hauptprogramm aus einzugreifen.

Bei der Anwendung in iterierenden Programmen ist der Einsatz eines so aufwendigen Unterprogramms für die Lösung der Gleichungssysteme in der Regel nicht erforderlich. Bei der Iteration wird die Kondition des Gleichungssystems im normalen Fall ständig verbessert. Damit ist auch die Konvergenz gegen die richtige Lösung gewährleistet.

3. Nichtlineare Gleichungen und Gleichungssysteme

Es sei f eine im abgeschlossenen Gebiet $G \subset R^1$ stetige und reellwertige Funktion.
Die Zahl $x_0 \subset R^1$ heißt *Nullstelle* der Funktion f, falls gilt

$$f(x) = 0 \tag{3.1}$$

mit $x = x_0$. Man sagt auch, daß x_0 eine Lösung der Gleichung $f(x) = 0$ ist.
Die spezielle Form von f

$$f(x) = \sum_{i=o}^{n} a_i x^i \; (a_i \text{reell}) \tag{3.2}$$

nennt man ein *algebraisches Polynom.*
Die Zahl n heißt der *Grad* des algebraischen Polynoms.
Der Fundamentalsatz der Algebra besagt, daß ein algebraisches Polynom (3.2) vom Grad n genau n Lösungen aufweist. Diese können komplex und zuweilen auch identisch sein. Identische Nullstellen eines algebraischen Polynoms nennt man auch mehrfache Nullstellen.
Ist der Grad des algebraischen Polynoms kleiner oder gleich vier, dann existieren geschlossene Lösungsformeln, die aus den Koeffizienten a_i die Nullstellen von (3.2) bestimmen. Für $n \geqq 5$ müssen die Lösungen numerisch bestimmt werden. Aber selbst bei einem Polynom vom Grad $n = 3$ ist die Berechnung der Lösungen mit den geschlossenen Lösungsformeln wegen der notwendigen Fallunterscheidungen ein sehr mühsamer Prozeß. Deswegen sollte bereits in diesem Fall eine numerische Lösung in Erwägung gezogen werden.
Falls f(x) sich nicht in der Form (3.2) darstellen läßt, ist (3.1) eine *transzendente* Gleichung.
Von einem *System nichtlinearer Gleichungen* spricht man bei folgender Problemstellung:

$$f_i(x_1, x_2, \ldots, x_n) = 0. \tag{3.3}$$

Der Definitionsbereich der Funktion D_i ist endlich und abgeschlossen im R^n.
Die Funktionen f_i; $i = 1(1)n$ seien stetig und reellwertig in ihrem jeweiligen Definitionsbereich.
Das Auffinden von Lösungen des Systems (3.3) ist ein schwieriges Problem, da a priori nicht bekannt ist, ob das System lösbar ist und wie viele Lösungen es aufweist. Nichtlineare Gleichungssysteme treten vor allem in der nichtlinearen Optimierung und bei der Lösung von Randwertaufgaben bei Differentialgleichungen auf.

3.1. Algebraische Gleichungen

Zum Auffinden der Nullstellen von (3.2) sind eine Reihe numerischer Verfahren in Gebrauch, z. B.

- klassische Iterationsverfahren (Fixpunktsatz)
  Newton-Verfahren für einfache und mehrfache Nullstellen
  Regula Falsi für einfache und mehrfache Nullstellen

Verfahren von *Steffensen* für einfache und mehrfache Nullstellen
Bisektionsverfahren in den verschiedensten Modifikationen

- Kombination des Newton-Verfahrens mit dem Horner-Schema zur systematischen Deflation von Nullstellen
- QD-Verfahren
- Muller-Verfahren
- Verfahren von *Bairstow*.

Für praktische Anwendungen benötigt man ein Verfahren, das ohne die Vorgabe von Näherungswerten für die Nullstellen möglichst schnell alle reellen und komplexen Nullstellen von (3.2) mit hinreichender Genauigkeit bestimmt. Diese Forderungen können sowohl durch die klassischen Iterationsverfahren als auch durch die Kombination des Newton-Verfahrens mit dem Horner-Schema nicht erfüllt werden. Ein Vergleich der anderen angeführten Lösungsmethoden (QD-Verfahren, Muller-Verfahren, Verfahren von *Bairstow*) ergab, daß das Muller-Verfahren für die Anwendung auf Mikrorechnern am besten geeignet ist. Es weist geringe Rundungsfehler auf und findet alle Nullstellen eines algebraischen Polynoms ohne die Vorgabe von Startwerten in einer vergleichsweise geringen Rechenzeit. Die Anwendung des Muller-Verfahrens kann deshalb für die Lösung von algebraischen Gleichungen mit $n > 2$ empfohlen werden.
Die nun folgende Beschreibung des Muller-Algorithmus ([Mull56]) basiert auf den Ausführungen in [Enge85]. Betrachtet wird das Polynom $P_n(x)$ mit

$$P_n(x) = \sum_{i=1}^{n} a_i x^i;\ a_i \in R^1, a_n \neq 0. \tag{3.4}$$

Das Grundprinzip des Muller-Verfahrens basiert auf der sukzessiven *Deflation von Nullstellen* und damit der Erniedrigung des Grades von $P_n(x)$. Die Deflation wird so lange wiederholt, bis alle Nullstellen gefunden sind. Die Berechnung der Deflationspolynome erfolgt mit dem Horner-Schema, während die Ermittlung der Näherungswerte für die Nullstellen mit der Iterationsvorschrift nach *Muller* vorgenommen wird.
Der Algorithmus des Muller-Verfahrens lautet:

- Bestimmung eines Näherungswerts $\overline{x_1}$ für die betragskleinste Nullstelle x_1 von $P_n(x)$.
- Ermittlung des Deflationspolynoms $P_{n-1}(x)$ durch Abdividieren der Nullstelle $\overline{x_1}$:

  $$P_{n-1}(x) \approx P_n(x)/(x - \overline{x_1}).$$

  Der entstehende Fehler wird vernachlässigt. $P_{n-1}(x)$ wird im Muller-Algorithmus gleich $P_n(x)$ gesetzt.
- Bestimmung der betragskleinsten Nullstelle x_2 des Deflationspolynoms $P_{n-1}(x)$ durch Muller-Iteration und Ermittlung von $P_{n-2}(x)$:

  $$P_{n-2}(x) = P_{n-1}(x)/(x - \overline{x_2}).$$

  $P_{n-2}(x)$ wird im Muller-Algorithmus gleich $P_n(x)$ gesetzt.
- Fortführung des Algorithmus, bis alle Nullstellen ermittelt sind. Meist gilt:

  $$|\overline{x_1}| \leqq |\overline{x_2}| \leqq \ldots \leqq |\overline{x_n}|. \tag{3.5}$$

Das Kernstück des Muller-Verfahrens ist die Muller-Iteration. Sie wird durch die folgende Vorgehensweise realisiert:

- Zu je drei Wertepaaren $(x^k, f(x^k)); k = l-2, l-1, l$ werden das zugehörige quadratische Interpolationspolynom und dessen Nullstellen bestimmt.
- Eine der Nullstellen wird als Näherung x^{l+1} für die gesuchte betragskleinste Nullstelle x_1 von $P_n(x)$ gewählt.

Die entsprechenden Gleichungen lauten:

$$x^{l+1} = x^l + h_l q_{l+1};\ l = 2, 3, \ldots \tag{3.6}$$

$$q_{l+1} = -2C_l/(\max(\text{abs}(B_l + \text{sqr}(B_l B_l - 4A_l C_l)) * \text{sign}(\max(B_l + \text{sqr}(B_l B_l - 4A_l C_l))) \tag{3.7}$$

(für $f(x^l) \neq f(x^{l-1}) \neq f(x^{l-2})$, sonst gilt $q_{l+1} = 1$)

$$h_l = x^l - x^{l-1} \tag{3.8}$$

$$q_l = h_l/(h_l - 1) \tag{3.9}$$

$$A_l = q_l f(x^l) - q_l(1 + q_l) f(x^{l-1}) + q_l^2 f(x^{l-2}) \tag{3.10}$$

$$B_l = (2q_l + 1) f(x^l) - (1 - q_l)^2 f(x^{l-1}) + q_l f(x^{l-2}) \tag{3.11}$$

$$C_l = (1 + q_l) f(x^l). \tag{3.12}$$

Als Startwerte für die Muller-Iteration werden die folgenden Werte fest vorgegeben:

$$(x^0 = -1,\ f(x^0) = a_0 - a_1 + a_2) \tag{3.13}$$

$$(x^1 = 1,\ f(x^1) = a_0 + a_1 + a_2) \tag{3.14}$$

$$(x^2 = 0,\ f(x^2) = a_0). \tag{3.15}$$

Die Werte a_0, a_1, a_2 sind die entsprechenden Koeffizienten des Polynoms $P_n(x)$.
Die Muller-Iteration wird beendet, falls gilt

$$|x^{l+1} - x^l|/|x^{l+1}| < \varepsilon;\ \varepsilon > 0. \tag{3.16}$$

Ist (3.16) erfüllt, so ist x^{l+1} die gesuchte Näherung $\bar{x}_1$ für die betragskleinste Nullstelle x_1 von $P_n(x)$.
Falls der Radikand der Wurzel in (3.7) negativ wird, können zwei Fälle auftreten: Zum einen

Programm P 3.01. POLQ.TEST. Testbeispiel für die Lösung quadratischer Gleichungen der Form $f(x)=ax^2+bx+x=0$.

```
 10 rem  POLQ
 20 print "loesung quadratischer gleichungen"
 30 a=1:b=2:c=3
 40 e1%=1:gosub 22000
 50 if e1%=2 then print "reelle wurzeln"
 60 if e1%=4 then print "komplexe wurzeln"
 70 if e1%>4 then print "fehlerflag=";e1%
 80 if e1%>4 then 130
 90 print
100 print x1;:if e1%=4 then print ,"(realteil)"
110 print
120 print x2;:if e1%=4 then print ,"(imaginaerteil)"
130 end
```

Globale Variable sind die Koeffizienten a,b,c, das Fehlerflag e1% sowie x1, x2 als die Lösungen der quadratischen Gleichung.

Das Fehlerflag hat folgende Bedeutungen:

e1%=1 Wert vor Initialisierung
e1%=2 reelle Wurzeln
e1%=4 komplexe Wurzeln
e1%=8 Gleichung schlecht konditioniert, Rundungsfehler können auftreten.

Programm P 3.02. POLQ.PROG. Modul zur Lösung quadratischer Gleichungen der Form $ax^2+bx+c=0$.

```
21999 rem !-----------------------------------------------!
22000 rem !                      POLQ                     !
22001 rem !-----------------------------------------------!
22002  if e1%<>1 then e1%=16
22004  if e1%=16 then 22040
22006  u=1
22008  if 1+u/2<>1 then u=u/2
22010  if 1+u/2<>1 then 22008
22012  if abs(a)<=4*u then e1%=8
22014  if e1%=8 then 22040
22016  s7=b*b-4*a*c
22018  r7=sqr(abs(s7))
22020  if s7<0 then 22034
22022  if abs(1-abs(b/r7))<333*u then e1%=8
22024  if e1%=8 then 22040
22026  e1%=2
22028  x1=(-b-r7)/(2*a)
22030  x2=(-b+r7)/(2*a)
22032  goto 22040
22034  e1%=4
22036  x1=-b/(2*a)
22038  x2=r7/(2*a)
22040 return
22041 rem !-----------------------------------------------!
```

Ist der Betrag des Koeffizienten a kleiner als eine durch die Maschinenkonstante bestimmte untere Grenze, so wird die Routine mit dem Fehlerflag e1%=8 verlassen. Die in diesem Fall auftretenden Rundungsfehler würden das Ergebnis zu stark verfälschen. Die Lösungen der Gleichung sind auf den Variablen x1, x2 gespeichert. Der Wert 2 für das Fehlerflag e1% signalisiert zwei reelle Lösungen, e1%=4 bedeutet eine konjugiert komplexe Wurzel.

Programm P 3.03. POLM.TEST. Testbeispiel zum Muller-Verfahren für Polynome.

```
 10 rem Test Muller-Verfahren
 20 n=7:rem Grad des Polynoms
 30 data -480,0,0,0,28,0,0,1:rem f(x)=x^7+28*x^4-480
 40 f2=1e-4:f1=1e-4:rem rel./abs. Genauigkeit
 50 n1=100:rem max. Anzahl der Iterationsschritte
 60 e1%=1:gosub 22100:rem Initialisierung
 70 if e1%<>32 then 140
 80 gosub 22200:rem Muller-Verfahren
 90 if e1%<>0 then 140
100 print "nst-nr.";tab(10);"realteil";tab(25);"imag.teil"
110 for i=1 to n
120 print i;tab(10);r(i+1);tab(25);i(i+1)
130 next i
140 print "fehlerflag=";e1%
150 end
```

Gesucht werden alle Wurzeln der algebraischen Gleichung

$$f(x)=x^7+28x^4-480=0.$$

Der Grad n des zu untersuchenden Polynoms wird in Zeile 20 definiert. Die Koeffizienten des Polynoms sind in Zeile 30 als **data**-Werte gespeichert. Zu beachten ist, daß die Koeffizienten beginnend mit x^0 in aufsteigender Reihenfolge bis x^n anzugeben sind. Mit f1 kann die

gewünschte relative Genauigkeit des Ergebnisses vorgegeben werden, f2 bestimmt die absolute Iterationsgenauigkeit. Um ein "Festfahren" der Iteration zu vermeiden, kann durch n1 die maximale Anzahl von Iterationsschritten festgesetzt werden.
Die Lösungen stehen auf den Feldern r(n+1) – Realteil – und i(n+1) – Imaginärteil. Zu berücksichtigen ist, daß die erste Nullstelle auf r(2), i(2) steht, die n-te demzufolge auf r(n+1), i(n+1).

Die Variable e1% ist das Fehlerflag:

e1%=0 Nullstellensuche ohne Fehler abgelaufen
e1%=1 Wert vor Aufruf der Initroutine
e1%=2 Fehler während der Initialisierungssequenz
e1%=4 Newton-Verfahren konvergiert nicht, f1 und/oder f2 vergrößern
e1%=16 POLM.PROG mit fehlerhaftem Init aufgerufen
e1%=32 Initialisierung ordnungsgemäß abgelaufen.

---

kann eine reelle Lösung von $P_n(x)=0$ aufgrund von Rundungsfehlern durch eine Folge komplexer Zahlen x^l approximiert worden sein; die Imaginärteile der x^l sind in diesem Fall sehr klein und streben gegen Null. Zum anderen kann es sein, daß die Muller-Iteration gegen eine komplexe Nullstelle x_1 konvergiert; dann ist auch der konjugiert komplexe Wert $\bar{x}_1$ Nullstelle von $P_n(x)$.
Somit kann der aktuelle Grad von $P_n(x)$ um zwei erniedrigt werden:

$$P_{n-2}(x) = P_n(x)/(x-\bar{x}_1)(x-\tilde{x}_1). \tag{3.17}$$

Die weiteren Nullstellen des so ermittelten Deflationspolynoms werden durch wiederholte Anwendung der Muller-Iteration bestimmt.

Programm P 3.04. POLM.INIT. Initialisierung des Muller-Verfahrens für Polynome.

```
22099 rem !-----------------------------------------------!
22100 rem !       Init Muller-Verfahren für Polynome      !
22101 rem !-----------------------------------------------!
22102   if e1%<>1 then 22140
22104   dim a(n+1)
22106   dim r(n+1)
22108   dim i(n+1)
22110   for i7=1 to n+1
22112     rem Matrixelemente einlesen
22114     read a(i7)
22116   next i7
22118   if n=0 then 22140
22120   if n<0 or n1<1 then 22140
22122   if f2<0 or f1<0 or f2+f1=0 then 22140
22124   if a(n+1)=0 then 22140
22126   h7=a(n+1)
22128   for i7=0 to n
22130     r(i7+1)=a(i7+1)/h7
22132     i(i7+1)=0
22134   next i7
22136   e1%=32
22138   goto 22142
22140   e1%=2
22142 return
22143 rem !-----------------------------------------------!
```

Die benötigten Felder a(n+1), r(n+1), i(n+1) werden dimensioniert. Die Koeffizienten des zu untersuchenden Polynoms werden aus der **data**-Zeile in das Feld a( ) eingelesen. In den Zeilen 22118 bis 22124 werden die globalen Eingabeparameter n, n1, f1, f2, a(n+1) auf Zulässigkeit geprüft. Ergeben sich Fehler, so wird die Initialisierungssequenz mit dem Fehlerkode e1%=2 abgebrochen.

Programm 3.05. POLM.PROG. Alle Nullstellen des Polynoms mit den auf a(2) bis a(n+1) gespeicherten Koeffizienten werden durch das Verfahren von Muller gesucht.

```
22199 rem !-------------------------------------------------!
22200 rem !NST i7 suchen;mit NEWTON verbessern;abdividieren!
22201 rem !-------------------------------------------------!
22202   if e1%<>32 then e1%=16
22204   if e1%=16 then 22402
22206   if e1%<>16 then e1%=0
22208   for i7=n to 1 step -1
22210     if i7=1 then z7=-r(1)
22212     if i7=1 then z8=-i(1)
22214     if i7=1 then 22234
22216     if i7>2 then gosub 22252
22218     if i7>2 then 22234
22220     a7=1
22222     a8=0
22224     b7=r(2)
22226     b8=i(2)
22228     c7=r(1)
22230     c8=i(1)
22232     gosub 22406
22234     gosub 22454
22236     if e1%<>0 then e1%=4
22238     if e1%<>0 then 22402
22240     gosub 22532
22242     r(i7+1)=z7
22244     i(i7+1)=z8
22246   next i7
22248 return
22249 rem !-------------------------------------------------!
22250 rem !                 Muller-Iteration                !
22251 rem !-------------------------------------------------!
22252   x7=-1
22254   x8=0
22256   y7=0
22258   y8=0
22260   z7=1
22262   z8=0
22264   v7=r(1)
22266   v8=i(1)
22268   u7=1
22270   u8=0
22272   w7=1
22274   w8=0
22276   for j7=i7-1 to 0 step -1
22278     u7=r(j7+1)-u7
22280     u8=i(j7+1)-u8
22282     w7=r(j7+1)+w7
22284     w8=i(j7+1)+w8
22286   next j7
22288   if w7*w7+w8*w8<=f1*f1 then 22402
22290   if v7*v7+v8*v8<=f1*f1 then z7=0
22292   if v7*v7+v8*v8<=f1*f1 then 22402
22294   if u7*u7+u8*u8<=f1*f1 then z7=-1
22296   if u7*u7+u8*u8<=f1*f1 then 22402
22298   d7=1
```

```
22300   for n7=1 to n1
22302     p7=x7-z7
22304     p8=x8-z8
22306     q7=y7-z7
22308     q8=y8-z8
22310     h7=p7*p7+p8*p8
22312     r7=((u7-w7)*p7+(u8-w8)*p8)/h7
22314     r8=((u8-w8)*p7-(u7-w7)*p8)/h7
22316     h7=q7*q7+q8*q8
22318     s7=((v7-w7)*q7+(v8-w8)*q8)/h7
22320     s8=((v8-w8)*q7-(v7-w7)*q8)/h7
22322     a7=r7-s7
22324     a8=r8-s8
22326     b7=(p7*s7-p8*s8)-(q7*r7-q7*r8)
22328     b8=(p7*s8+p8*s7)-(q7*r8+q8*r7)
22330     c7=(x7-y7)*w7-(x8-y8)*w8
22332     c8=(x8-y8)*w7+(x7-y7)*w8
22334     x7=y7
22336     x8=y8
22338     u7=v7
22340     u8=v8
22342     y7=z7
22344     y8=z8
22346     v7=w7
22348     v8=w8
22350     if a7<>0 or a(i7+1)<>0 then gosub 22406
22352     if a7<>0 or a(i7+1)<>0 then 22368
22354     if b7=0 and b8=0 then 22364
22356     h7=b7*b7+b8*b8
22358     z7=-(c7*b7+c8*b8)/h7
22360     z8=-(c8*b7-c7*b8)/h7
22362     goto 22368
22364     z7=.629*q7
22366     z8=.629*q8
22368     h7=sqr(z7*z7+z8*z8)
22370     if h7<=3*d7 the.. 22376
22372     z7=z7*3*d7/h7
22374     z8=z8*3*d7/h7
22376     d7=h7
22378     z7=z7+y7
22380     z8=z8+y8
22382     if d7<f2 then 22402
22384     w7=z7+r(i7)
22386     w8=z8+i(i7)
22388     for j7=i7-2 to 0 step -1
22390       h7=w7
22392       w7=w7*z7-w8*z8+r(j7+1)
22394       w8=h7*z8+w8*z7+i(j7+1)
22396     next j7
22398     if w7*w7+w8*w8<f1*f1 then 22402
22400   next n7
22402 return
22403 rem !-------------------------------------------------!
22404 rem ! betragskleinste NST des quadr. Polynoms suchen !
22405 rem !-------------------------------------------------!
22406   p7=b7*b7-b8*b8
22408   p8=2*b7*b8
22410   q7=a7*c7-a8*c8
22412   q8=a7*c8+a8*c7
22414   p7=p7-4*q7
22416   p8=p8-4*q8
22418   h7=sqr(p7*p7+p8*p8)
22420   q7=h7+p7
22422   if q7<0 then q7=0
22424   q8=h7-p7
22426   if q8<0 then q8=0
22428   q7=sqr(q7/2)
```

```
22430   q8=sqr(q8/2)
22432   if p8<0 then q8=-q8
22434   h7=q7*b7+q8*b8
22436   if h7>0 then q7=-q7
22438   if h7>0 then q8=-q8
22440   p7=q7-b7
22442   p8=q8-b8
22444   h7=p7*p7+p8*p8
22446   z7=2*(c7*p7+c8*p8)/h7
22448   z8=2*(c8*p7-c7*p8)/h7
22450 return
22451 rem !-------------------------------------------!
22452 rem !                Newton-Verfahren           !
22453 rem !-------------------------------------------!
22454   gosub 22488
22456   d7=u7*u7+u8*u8
22458   x7=z7
22460   x8=z8
22462   z7=z7-w7
22464   z8=z8-w8
22466   if x7=z7 and x8=z8 then 22484
22468   gosub 22488
22470   h7=u7*u7+u8*u8
22472   if h7>=d7 then 22478
22474   d7=h7
22476   goto 22458
22478   z7=x7
22480   z8=x8
22482   if w7*w7+w8*w8>f2*f2 and d7>f1*f1 then e1%=4
22484 return
22485 rem !-------------------------------------------!
22486 rem !       Hilfsprogramm für Newton-Iteration  !
22487 rem !-------------------------------------------!
22488   v7=a(n+1)
22490   v8=0
22492   u7=z7*a(n+1)+a(n)
22494   u8=z8*a(n+1)
22496   if n=1 then 22514
22498   for j7=n-2 to 0 step -1
22500     h7=v7
22502     v7=v7*z7-v8*z8+u7
22504     v8=h7*z8+v8*z7+u8
22506     h7=u7
22508     u7=u7*z7-u8*z8+a(j7+1)
22510     u8=h7*z8+u8*z7
22512   next j7
22514   h7=v7*v7+v8*v8
22516   if h7<>0 then 22524
22518   w7=u7
22520   w8=u8
22522   goto 22528
22524   w7=(u7*v7+u8*v8)/h7
22526   w8=(u8*v7-u7*v8)/h7
22528 return
22529 rem !-------------------------------------------!
22530 rem !              Abdividieren von NST         !
22531 rem !-------------------------------------------!
22532   print i7
22534   for j7=i7-1 to 0 step -1
22536     r(j7+1)=r(j7+1)+(z7*r(j7+2)-z8*i(j7+2))
22538     i(j7+1)=i(j7+1)+(z8*r(j7+2)+z7*i(j7+2))
22540   next j7
22542   for j7=0 to i7-1
22544     r(j7+1)=r(j7+2)
22546     i(j7+1)=i(j7+2)
22548   next j7
22550 return
22551 rem !-------------------------------------------!
```

Falls die geforderte relative (f1) bzw. absolute (f2) Genauigkeit nicht erreicht werden kann, so wird die Routine mit e1%=4 verlassen. In diesem Fall empfiehlt sich ein Neustart mit vergrößerten Werten f1, f2.
Normale Beendigung der Routine liefert e1%=0. Wird POLM. PROG mit fehlerhafter Initialisierung aufgerufen, so ist der Rückkehrkode e1%=16.
Falls auf Rechnern mit $u \approx 2.328.10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

Die globale Konvergenz des Muller-Verfahrens konnte bisher nicht nachgewiesen werden. Da es allerdings bei keinem der gerechneten Testbeispiele versagte, kann seine Anwendung nachhaltig empfohlen werden.
Die Software zur Lösung algebraischer Gleichungen besteht aus den Programmen POLQ **(P 3.01 POLQ.TEST, P 3.02 POLQ.PROG)** und POLM **(P 3.03 POLM.TEST, P 3.04 POLM. INIT, P 3.05 POLM.PROG)**.
Das Programm POLQ ist speziell für quadratische Gleichungen geschrieben (n = 2), weil in diesem Fall der Einsatz des Muller-Verfahrens zu aufwendig wäre.
Im Programm POLQ wird die quadratische Gleichung $f(x) = ax^2 + bx + c = 0$ gelöst. Dabei wird untersucht, ob Rundungsfehler Einfluß auf das Ergebnis nehmen können. Dies ist besonders dann möglich, wenn der Betrag des Koeffizienten a die Größenordnung der Maschinenkonstante u aufweist oder Terme in der Lösungsformel voneinander subtrahiert werden, deren Betrag fast gleich ist (Auslöschung geltender Ziffern).
In diesen Fällen wird die Routine P 3.02 POLQ.PROG ohne Berechnung der Nullstellen von f(x) mit e1 % = 8 verlassen.
Das Programm POLM mit den Modulen P 3.03 POLM.TEST, P 3.04 POLM.INIT und P 3.05 POLM.PROG realisiert das durch (3.6) bis (3.17) gegebene Muller-Verfahren für Polynome. Es bestimmt alle reellen und komplexen Nullstellen von (3.4).
In der Routine P 3.04 POLM.INIT (22100 bis 22142) werden die benötigten Felder dimensioniert und mit den entsprechenden Startwerten versehen. Es ist zu beachten, daß der Grad n und die Koeffizienten a_i des zu untersuchenden Polynoms in der Routine P 3.03 POLM.TEST in **data**-Zeilen vorzugeben sind. Die Reihenfolge $n, a_0, a_1, \ldots, a_n$ muß dabei unbedingt eingehalten werden.

Falls gilt

$a_i = 0 \quad (i = 0(1)n - 1)$
$a_n \neq 0,$

wird von P 3.04 POLM.INIT kein Fehler angezeigt, und P 3.05 POLM.PROG kann aufgerufen werden. Das Polynom (3.4) hat die n-fache Nullstelle x = 0. Diese kann aber vom Algorithmus nicht ordnungsgemäß ermittelt werden. Man erhält entweder einen OVERFLOW ERROR oder aber einen DIVISION BY ZERO ERROR. Die Bearbeitung einer algebraischen Gleichung der Form $f(x) = x^n = 0$ ist also zu vermeiden.
Die für die Arbeit des Programms POLM notwendigen globalen Variablen sind in der Legende zu P 3.03 POLM.TEST aufgeführt.
Der Modul P 3.05 POLM.PROG (22200 bis 22550), der das Muller-Verfahren realisiert, besteht aus folgenden Routinen:

- Hauptsteuerschleife (22200 bis 22248)
- Muller-Iteration (22250 bis 22402)
- Suchen der betragskleinsten Nullstelle eines quadratischen Polynoms (22404 bis 22450)
- Newton-Verfahren zur Konvergenzbeschleunigung (22452 bis 22528)
- Abspalten von Nullstellen durch Division nach *Horner* (22530 bis 22550).

Der Ablauf der Berechnungen wird durch die Hauptsteuerschleife in den Zeilen 22200 bis 22248 kontrolliert. Dabei werden drei Fälle unterschieden:

$i7 > 2$
$i7 = 2$
$i7 = 1$.

Im Fall $i7 > 2$ wird zunächst die Muller-Iteration aufgerufen, die wiederum die Routine zur Ermittlung der betragskleinsten Nullstelle eines quadratischen Polynoms aufruft. Die gefundene Nullstelle wird mit dem Newton-Verfahren verbessert. Falls das Newton-Verfahren nicht konvergiert, wird die Routine P 3.05 POLM.PROG mit e1 % = 4 verlassen.
Anderenfalls wird die gefundene Nullstelle abdividiert. Der Realteil der Nullstelle wird auf r(i7 + 1) gespeichert; der Imaginärteil wird auf i(i7 + 1) abgelegt. Die erste Nullstelle befindet sich somit auf r(2), i(2), die letzte auf r(i7 + 1), i(i7 + 1).
Im Fall $i7 = 2$ wird zuerst die Routine zur Ermittlung der betragskleinsten Nullstelle eines quadratischen Polynoms aufgerufen. Die gefundene Näherung für die Nullstelle wird dann mit Hilfe des Newton-Verfahrens verbessert und schließlich noch abdividiert. Die Ergebnisse werden wiederum auf den Feldelementen r(i7 + 1), i(i7 + 1) abgespeichert.
Ist $i7 = 1$, dann wird zuerst eine Näherung für die letzte zu bestimmende Nullstelle auf die lokalen Variablen z7 (Realteil) und z8 (Imaginärteil) gespeichert. Danach wird das Newton-Verfahren zur Verbesserung dieser Näherungswerte aktiviert. Die verbesserten Näherungswerte werden dann wiederum abdividiert.
Im P 3.03 POLM.TEST wird die Arbeitsweise des Muller-Verfahrens anhand der Lösung der algebraischen Gleichung

$$f(x) = x^7 + 28x^4 - 480 = 0$$

gezeigt. Nach 52.7 s/Rechenzeit (6510-Prozessor, 1 MHz Taktfrequenz) werden folgende Nullstellen ausgegeben:

Nullstelle-Nr.	Realteil	Imaginärteil
1	1.68431572	−2.66379119
2	1.68431572	2.66379119
3	−2.5778039	$2.74003578 \cdot 10^{-24}$
4	−2.45808917	$-1.62439506 \cdot 10^{-12}$
5	−0.127811265	−1.98742322
6	−0.127811265	1.98742322
7	1.92288415	0

Diese Lösungen sind im Rahmen der 5-Byte-Gleitkommaarithmetik als exakt anzusehen. Die Bestimmung der Nullstellen dieses Polynoms mit dem Bairstow-Verfahren lieferte wesentlich ungenauere Ergebnisse. Die Nullstellen 1, 2, 5, 6 sind konjugiert komplexe Nullstellen. Dagegen sind die Nullstellen 3, 4, 7 reelle Nullstellen, obwohl bei den Nullstellen 3 und 4 Imaginärteile ungleich Null angegeben sind.
Da komplexe Nullstellen aber stets paarweise auftreten müssen, sind in diesem Fall keine Fehlinterpretationen möglich.

3.2. Transzendente Gleichungen

3.2.1. Iterationsverfahren

Anstelle der nichtlinearen Gleichung (3.1) wird nunmehr eine Gleichung der Form

$$x = F(x) \tag{3.18}$$

betrachtet. Die Funktion F sei in einem abgeschlossenen Intervall I stetig und reellwertig. Die Größe x_0 heißt Lösung von (3.18), wenn $x_0 = F(x_0)$ ist. Gleichungen der Form (3.18) sind für die weiteren Betrachtungen deshalb interessant, weil gilt

$$x_0 = F(x_0) \Leftrightarrow f(x_0) = 0. \tag{3.19}$$

Die Gleichungen $f(x) = 0$ und $x = F(x)$ weisen somit im Intervall I die gleichen Lösungen auf. Mit Hilfe eines Startwerts $x^{(0)}$ wird eine Zahlenfolge $\{x^{(i)}\}$ nach der Vorschrift

$$x^{(i+1)} = F(x^{(i)}); \; i = 0, 1, 2, 3 \ldots \tag{3.20}$$

konstruiert. Diese Zahlenfolge konvergiert nur dann, wenn gilt

$$x^{(i+1)} = F(x^{(i)}) \in I. \tag{3.21}$$

Ihr Grenzwert ist

$$\lim_{i \to \infty} x^{(i)} = x_0. \tag{3.22}$$

Die Größe x_0 ist sowohl Lösung von (3.1) als auch von (3.18). In diesem Fall ergeben sich die folgenden Beziehungen:

$$x_0 = \lim_{i \to \infty} x^{(i)} = \lim_{i \to \infty} x^{(i+1)} = \lim_{i \to \infty} F(x^{(i)}) = F(\lim_{i \to \infty} x^{(i)}) = F(x_0). \tag{3.23}$$

Die durch (3.23) beschriebene Vorgehensweise wird als *Iterationsverfahren* oder Verfahren der schrittweisen Annäherung bezeichnet.
Gleichung (3.20) heißt *Iterationsvorschrift*. Für jeden Wert i stellt (3.20) einen *Iterationsschritt* dar.

Die Funktion F nennt man *Schrittfunktion*; die Folge $\{x^{(i)}\}$ heißt *Iterationsfolge*.
Der für ein Iterationsverfahren charakteristische Algorithmus hat somit die folgende Form:

$x^{(0)}$: Startwert
$x^{(1)} = F(x^{(0)})$
$x^{(2)} = F(x^{(1)})$

$$x^{(i+1)} = F(x^{(i)}) \tag{3.24}$$

$$x^{(N+1)} = F(x^{(N)}).$$

Wenn dieser Algorithmus konvergiert, ergibt sich im Rahmen der Rechengenauigkeit des verwendeten Rechners:

$$x^{(N+1)} \approx x^{(N)} \approx x_0. \tag{3.25}$$

Zunächst soll die Existenz von Lösungen von (3.18) im endlichen und abgeschlossenen Intervall I untersucht werden. Dafür gilt folgender Satz:

(3.18) besitzt in I mindestens eine Lösung x_0, falls F stetig in I ist und außerdem alle iterierten Werte F(x*) in I liegen ($x^* \in I$).
Darüber hinaus ist die iterierte Lösung eindeutig, wenn eine sog. *Lipschitz-Bedingung* erfüllt wird:

$$|F(x) - F(x^*)| \leqq L|x - x^*|. \tag{3.26}$$

Falls die Bedingung (3.26) für die untersuchte Funktion F gilt, nennt man F *lipschitzbeschränkt*. Für lipschitzbeschränkte Funktionen F gilt der Eindeutigkeitssatz:

■ Die Gleichung $x = F(x)$ besitzt höchstens eine Lösung $x_0 \in I$, wenn F im Intervall I einer Lipschitz-Bedingung (3.26) genügt.

Die Funktion F ist sicher lipschitzbeschränkt, falls gilt

$$|F'(x)| \leqq L < 1. \tag{3.27}$$

Die Funktion F besitzt also genau dann eine Lösung $x_0 = F(x_0)$ im endlichen und abgeschlossenen Intervall I, wenn der Existenzsatz erfüllt ist und entweder (3.26) oder (3.27) gelten.
Außerdem ist zu klären, unter welchen Bedingungen das Iterationsverfahren (3.20) gegen x_0 konvergiert.
Zur Beantwortung dieser Frage soll vorerst angenommen werden, daß der iterierte Wert $x^{(i)}$ sich in einer Umgebung der Lösung von (3.18) befindet:

$$x^{(i)} = x_0 + \varepsilon^{(i)}. \tag{3.28}$$

Die Größe $\varepsilon^{(i)}$ soll dabei betragsmäßig klein gegenüber x_0 sein.
Wenn $\varepsilon^{(i)}$ mit wachsendem i immer kleiner wird, dann konvergiert der Iterationsprozeß gegen x_0. Anderenfalls ist der Prozeß divergent.
Zur Untersuchung der Konvergenz des Iterationsprozesses empfiehlt sich die Entwicklung von F(x) in eine Taylor-Reihe um die gesuchte Lösung x_0:

$$\begin{aligned} x^{(i+1)} &= F(x^{(i)}) \\ &= F(x_0 + \varepsilon^{(i)}) \\ &= F(x_0) + \varepsilon^{(i)}F'(x_0) + 0.5\varepsilon^{(i)2}F''(x_0) + \ldots \end{aligned} \tag{3.29}$$

Unter Vernachlässigung der höheren Potenzen von $\varepsilon^{(i)}$ erhält man

$$x^{(i+1)} \approx x_0 + \varepsilon^{(i)}F'(x_0) \tag{3.30}$$

oder

$$\varepsilon^{(i+1)} \approx \varepsilon^{(i)}F'(x_0). \tag{3.31}$$

Aus (3.31) ist ersichtlich, daß $\varepsilon^{(i)}$ genau dann bei wachsendem i abnimmt, wenn $|F'(x_0)| < 1$ ist.
Wenn diese Bedingung in einer ε-Umgebung von x_0 erfüllt ist, dann konvergiert die Iteration.
An dieser Stelle wird ein schwerwiegender Nachteil für die praktische Anwendung des Iterationsverfahrens sichtbar:

■ Ein Iterationsprozeß konvergiert genau dann, wenn

$|F'(x_0)| < 1$ ist.

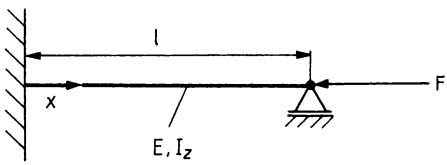
Durch die Iteration soll aber erst die Lösung x_0 gesucht werden, so daß von vornherein nichts über $|F'(x_0)|$ ausgesagt werden kann.

Es ist somit zu fordern, daß $|F'(x)|$ im gesamten Suchintervall I kleiner als Eins ist.
Von dieser Tatsache muß man sich vor dem Beginn der Berechnungen überzeugen. Ist die Konvergenzbedingung (3.27) für das Suchintervall I nicht erfüllt, so ist ein neues Intervall I* festzulegen, in dem (3.27) erfüllt ist.

Als Beispiel für die Anwendung eines iterativen Prozesses betrachten wir die Lösung der transzendenten Gleichung

$$f(x) = x - \tan(x) = 0. \tag{3.32}$$

Diese Gleichung tritt in der technischen Mechanik bei der Behandlung der *Eulerschen Knickfälle* auf ([Rued65]). Das durch (3.32) modellierte Stabilitätsproblem ist im **Bild 3.1** dargestellt.



E Elastizitätsmodul
I_z Biegesteifigkeit
l Länge

Bild 3.1. Eulerscher Knickfall

Die für die Iterationen notwendige Umformung von (3.32) läßt sich z. B. erreichen, indem man schreibt

$$x = \tan(x) \tag{3.33}$$

mit $F(x)=\tan(x)$.
Aus

$$F'(x) = 1 + \tan^2(x) \geqq 1 \; \forall \, x \in R \tag{3.34}$$

kann man erkennen, daß der durch (3.33) gegebene Iterationsprozeß für beliebige Startwerte $x_0 > 0$ divergent ist.
Dies kann durch das folgende kurze BASIC-Programm veranschaulicht werden:

```
10 input x
20 x=tan(x)
30 print x
40 goto 20
```

Eine andere Möglichkeit der Umformung von (3.32) ist

$$x = \arctan(x) + k\pi; \; k = \pm 1,2,3,\ldots \tag{3.35}$$

Mit

$$F'(x) = \frac{1}{1+x^2} \tag{3.36}$$

ergibt sich Konvergenz für alle Startwerte x_0 bei $|x| \neq 0$.
Das sich für $k=1$ ergebende Verhalten des Iterationsverfahrens kann durch das folgende BASIC-Programm demonstriert werden:

```
10 input x
20 x=atn(x)+pi
30 print x
40 goto 20
```

Für verschiedene Startwerte x_0 liefert es schnell konvergierende Folgen $\{x^{(i)}\}$.

Startwerte x_0	0	4	5
$x^{(1)}$	3.14159265	4.46741032	4.51499342
$x^{(2)}$	4.40421991	4.49217574	4.49442337
$x^{(3)}$	4.48911945	4.49335123	4.4934573
$x^{(4)}$	4.49320683	4.49340671	4.49341172
$x^{(5)}$	4.4933999	4.49340933	4.49340956
$x^{(6)}$	4.49340901	4.49340945(*)	4.49340946
$x^{(7)}$	4.49340944		4.49340945(*)
$x^{(8)}$	4.49340945(*)		

Nach maximal acht Iterationen ist im Rahmen der 5-Byte-Gleitkommaarithmetik die erste nichttriviale Lösung (*) von (3.32) gefunden.
Bereits an dem einfachen Beispiel (3.32) kann man erkennen, daß die Anwendung eines Iterationsverfahrens nicht immer problemlos möglich ist. Es sind stets Überlegungen zur Konvergenz des Prozesses notwendig.
Aus (3.30) ist ersichtlich, daß die Konvergenzgeschwindigkeit von $|F'(x_0)|$ abhängt. Je kleiner der Betrag der Ableitung ist, um so schneller konvergiert das Iterationsverfahren. Ein Maß für die Schnelligkeit der Konvergenz eines Iterationsprozesses kann aus (3.29) abgeleitet werden. Falls z. B. $F'(x_0)$ gleich Null ist, kann man schreiben

$$x^{(i+1)} = F(x_0) + 0.5\ \varepsilon^{(i)2} F''(x_0) + \ldots; \tag{3.37}$$

und daraus folgt

$$\varepsilon^{(i+1)} = 0.5\ \varepsilon^{(i)2} F''(x_0). \tag{3.38}$$

In diesem Fall ist $\varepsilon^{(i+1)}$ ein Vielfaches des Quadrats von $\varepsilon^{(i)}$. Wenn also $\varepsilon^{(i)}$ klein ist, dann ist $\varepsilon^{(i+1)}$ noch wesentlich kleiner. In der Tat verdoppelt sich bei einem Iterationsprozeß der Form (3.37) etwa die Anzahl der gültigen Stellen je Iterationsschritt.

Falls gilt

$$\varepsilon^{(i+1)} = A\ \varepsilon^{(i)^n}; A = \text{const}, A \neq 0, \tag{3.39}$$

sagt man, daß der entsprechende Iterationsprozeß die Konvergenzordnung n aufweist.
Folgende praktisch wichtige Spezialfälle seien genannt:

- Konvergenzordnung 1 (einfaches Iterationsverfahren)
  $F'(x_0) \neq 0$, Konvergenz bei $|F'(x_0)| < 1$
- Konvergenzordnung 2 (Newton-Verfahren)
  $F'(x_0) = 0$, $F''(x_0) \neq 0$
- Konvergenzordnung 3 (speziell konstruierte Verfahren)
  $F'(x_0) = 0$, $F''(x_0) = 0$, $F'''(x_0) \neq 0$.

Iterationsprozesse mit einer Konvergenzordnung $n > 1$ konvergieren immer, wenn man ausreichend nahe an der gesuchten Lösung x_0 mit der Iteration beginnt.
Das Problem besteht darin, eine „ausreichend" genaue Näherung für die gesuchte Lösung x_0 zu finden.
Die nächste für praktische Anwendungen interessante Frage ist die nach der Genauigkeit der durch ein Iterationsverfahren generierten Näherungslösungen. In [Enge85] sind dazu zwei Fehlerschätzungsformeln angegeben. Sie gelten unter der Voraussetzung, daß der Rechnungsfehler vernachlässigbar klein ist.

Die erste Abschätzung ist eine *A-priori-Fehlerabschätzung*. Sie kann bereits nach dem ersten Iterationsschritt ausgewertet werden:

$$|e^{(i)}| = |x^{(i)} - x_0| \leqq \frac{L^i}{1-L}|x^{(1)} - x^{(0)}|. \tag{3.40}$$

Die zweite Abschätzung ist eine *A-posteriori-Abschätzung*. Diese ist nach Abschluß der Iterationen möglich:

$$|e^{(i)}| = |x^{(i)} - x_0| \leqq \frac{L}{1-L}|x^{(i)} - x^{(i-1)}|. \tag{3.41}$$

Sie liefert eine bessere Abschätzung des Iterationsfehlers als (3.40).
Da die Bestimmung der Lipschitz-Konstante meist problematisch ist, kann folgende Näherung empfohlen werden:

$$L \leqq \max|F'(x)|;\ x \in I. \tag{3.42}$$

Bei der numerischen Realisierung eines Iterationsprozesses treten aber nicht nur Verfahrensfehler, sondern auch die durch die endliche Stellenzahl in der Maschine bedingten Rechnungsfehler auf.
Für den *akkumulierten Rechnungsfehler* im i-ten Iterationsschritt kann folgende Näherung angegeben werden:

$$|r^{(i)}| \leqq \frac{e_r}{1-L};\ 0 \leqq L < 1. \tag{3.43}$$

Diese Abschätzung zeigt, daß der Fehler unabhängig von der Anzahl der Schritte ist. Der Iterationsprozeß arbeitet somit stabil.

Numerische Verfahren arbeiten dann besonders effektiv, wenn der Verfahrensfehler etwa gleich dem Rechnungsfehler ist. Die Gleichsetzung von (3.40) und (3.43) ergibt

$$\frac{L^i}{1-L}\,|\,x^{(1)} - x^{(0)}\,| \approx \frac{e_r}{1-L}. \tag{3.44}$$

Mit $e_r \approx u$ (Maschinenkonstante) und unter Berücksichtigung von (3.42) kann man die Anzahl der notwendigen Iterationsschritte i_{max} bis zur Konvergenz des Prozesses angeben:

$$i_{max} \approx \text{int}(\log(u/\text{abs}(x^{(1)} - x^{(0)}))/\log(\max(\text{abs}(F'(x))));\ x \in I. \tag{3.45}$$

Betrachten wir nunmehr das Beispiel (3.35) mit $x^{(0)} = 4$ und $I = [4,5]$. Man erhält:

$$\begin{aligned} x^{(1)} &= 4.46741032 \\ F'(4) &= 0.0588235294 \\ F'(x^{(0)}) &= 0.0477150344 \\ F'(5) &= 0.0384615383. \end{aligned}$$

Daraus folgt mit $u = 2.328 \cdot 10^{-10}$ und unter der Annahme, daß $F'(4) = \max(\text{abs}(F'(x)));\ x \in I$ ist:

$$i_{max} = 7.$$

Dies ist eine recht genaue Abschätzung, da für das betrachtete Beispiel das Iterationsverfahren für $x \in [4,5]$ bereits nach sechs Schritten konvergierte.
Für möglichst rasche Konvergenz ist also zu fordern:

$$\begin{aligned} &\text{abs}(x^{(1)} - x^{(0)}) \ll 1 \\ &L \ll 1. \end{aligned} \tag{3.46}$$

Je besser diese beiden Forderungen bei festem u erfüllt werden, um so schneller konvergiert der betrachtete Iterationsprozeß.

Es lohnt sich also durchaus, die Funktion F(x) so zu konstruieren, daß sie für ein Iterationsverfahren besonders geeignet ist.
Das ist jedoch meist nicht ohne größeren Rechenaufwand möglich. Außerdem muß man vor Beginn der Berechnungen ein Suchintervall I so festlegen, daß (3.26) gültig ist. Die Erfüllung dieser Vorbedingungen kann meist nur bei einfach strukturierten Funktionen f(x) gewährleistet werden. Deshalb sollte man Iterationsverfahren auch nur in diesen Fällen einsetzen.
Das im folgenden angegebene BASIC-Programm geht auf [Scra84, S. 19] zurück. Es arbeitet mit einer Formel zur Konvergenzbeschleunigung (Zeile 230) und erreicht so die Konvergenzordnung zwei.
Für den Startwert x0 wird überprüft, ob der Betrag der Ableitung der Schrittfunktion kleiner als Eins ist. Dies erfolgt durch numerische Differentiation der Schrittfunktion in den Zeilen 310 bis 400.
Der Test auf Einhaltung der Lipschitzbedingung (3.27) wird in Zeile 380 ausgeführt. Falls (3.27) nicht erfüllt ist, wird das Programm mit einer entsprechenden Fehlermeldung beendet. Anderenfalls wird der Algorithmus des Iterationsverfahrens in den Zeilen 100 bis 270 abgearbeitet.
Mit dem Startwert x0 = 4 und der Fehlerschranke eps = 1e − 9 konvergiert der Algorithmus für die in Zeile 290 definierte Schrittfunktion (3.35) (k = 1) bereits nach zwei Iterationen. Das Ergebnis ist im Rahmen der 5-Byte-Gleitkommaarithmetik als exakt anzusehen.

```
 10 rem iterationsverfahren mit konvergenzbeschleunigung
 20 rem x0: startwert f: funktionswert
 30 rem x: aktueller x-wert
 40 rem eps: abbruchschranke
 50 rem r: anzahl funktionswertaufrufe
 60 rem f in zeile 290 definieren
 70 input "x0"; x0
 80 input"eps"; eps
 90 gosub 310
100 r=0
110 print
120 print r, x0
130 x=x0
140 gosub 280
150 b=f
160 print r+1, b
170 x=b
180 gosub 280
190 c=f
200 print r+2, c
210 q=c-2*b+x0
220 if abs(q) <eps then 270
230 x0=c-(c-b)*(c-b)/(c-2*b+x0)
240 r=r+3
250 print"----------------"
260 goto 120
270 end
280 rem schrittfunktion
290 f=atn(x)+π
300 return
310 rem f'(x) überprüfen
320 x=x0
330 gosub 280
```

```
340 b = f
350 x = x + .001
360 gosub 280
370 b = (f - b)/.001
380 if abs(b) < 1 then return
390 print"abs(f'(x0)) > 1)"
400 end
```

x0 = 4
eps = 1e − 9

Auswertungen x von f	
1	4.46741032
2	4.49217574
3	4.49356134 (*)
4	4.49341663
5	4.4934098
6	4.49340945 (*)
7	4.49340945
8	4.49340945

(*) iterierter Wert

3.2.2. Newton-Verfahren

Das Newton-Verfahren, manchmal auch Newton-Raphson-Verfahren genannt, ist ein spezielles Iterationsverfahren. Durch geeignete Umformung der Schrittfunktion F(x) wird erreicht, daß $F'(x_0)$ in der Umgebung von x_0 verschwindet. Damit erreicht man quadratische Konvergenz in einer ε-Umgebung von x_0.
Zur Herleitung des Newton-Verfahrens gehen wir davon aus, daß eine Nullstelle x_0 der Funktion f(x) durch ein Iterationsverfahren bestimmt werden soll.
Dazu ist die Gleichung $f(x) = 0$ in eine iterationsfähige Form zu bringen: $x = F(x)$. Strebt f(x) in der Umgebung von x_0 gegen Null, kann (3.18) umgeformt werden:

$$x = F(x) = x - \frac{f(x)}{f'(x)};\ x \approx x_0,\ f'(x_0) \neq 0, \text{ einfache Nullstelle.} \tag{3.47}$$

Die erste Ableitung der Schrittfunktion kann mit Hilfe der Quotientenregel berechnet werden:

$$F'(x) = 1 - \frac{f'^2(x) - f(x) \cdot f''(x)}{f'^2(x)} \tag{3.48}$$

$$F'(x) = \frac{f(x) \cdot f''(x)}{f'^2(x)}.$$

Unter Berücksichtigung von (3.26) und (3.27) ergibt sich aus (3.48) eine notwendige Bedingung für die Konvergenz des Newton-Verfahrens.
Das Newton-Verfahren konvergiert genau dann, wenn für das Suchintervall I mit $x \in [x_0 - \varepsilon, x_0 + \varepsilon]$; ε klein, $\varepsilon > 0$ gilt

$$\Lambda = \left| \frac{f(x) \cdot f''(x)}{f'^2(x)} \right| \leqq L < 1; \tag{3.49}$$

$$f'(x) \neq 0 \forall x \in [x_0 - \varepsilon, x_0 + \varepsilon].$$

Aus (3.48) ist außerdem ersichtlich, daß F'(x) für $x = x_0$ den Wert null annimmt, falls $f'(x_0)$ ungleich Null ist. Damit ist die quadratische Konvergenz des Newton-Verfahrens gewährleistet. Die Herleitung des Newton-Verfahrens ist auch anschaulich möglich. Betrachten wir dazu **Bild 3.2.**

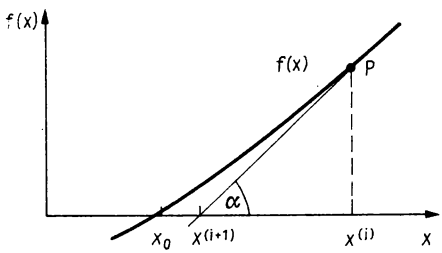


Bild 3.2. Herleitung des Newton-Verfahrens

Die Funktion f(x) weist eine Nullstelle bei $x = x_0$ auf. Der Iterationsprozeß sei bei $x^{(i)}$ angelangt. Eine bessere Näherung $x^{(i+1)}$ läßt sich offensichtlich gewinnen, indem im Punkt P eine Tangente an f(x) gelegt wird. Der Schnittpunkt dieser Tangente mit der x-Achse ergibt dann das nächste Glied $x^{(i+1)}$ der Iterationsfolge.
Diese Vorgehensweise kann natürlich nur dann zum Ziel führen, wenn zwischen $x^{(i)}$ und x_0 f(x) nicht zu stark gekrümmt ist.
Der Steigungswinkel α der Tangente an f(x) im Punkt P ist durch folgende Gleichung gegeben:

$$\tan \alpha = \frac{f(x^{(i)})}{x^{(i)} - x^{(i+1)}}. \tag{3.50}$$

Aus der analytischen Geometrie ist bekannt, daß folgende Beziehung gilt:

$$\tan \alpha = f'(x^{(i)}). \tag{3.51}$$

Aus (3.50) und (3.51) ergibt sich nunmehr die Schrittfunktion des Newton-Verfahrens:

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}. \tag{3.52}$$

Anhand der bereits im Abschnitt 3.2.1 betrachteten Gleichungen $f(x) = x - \tan(x) = 0$ und $g(x) = x - \arctan(x) - \pi = 0$ soll die Konvergenz des Newton-Verfahrens dargestellt werden. Die Bestimmung der Lösung von g(x) zwischen π und 2π ist z. B. mit folgendem kurzem BASIC-Programm möglich:

```
10 input x
20 if x = 0 or x = 1 then 10
30 f = x - atn(x) - pi
40 fs = 1 - 1/(1 + x*x)
50 xn = x - f/fs
60 print xn, f, fs
70 if abs(x - xn) > 1e - 10 then x = xn: goto 30
80 end
```

Für den Eingabewert $x^{(0)} = x = 4$ ergeben sich die folgenden Werte $x^{(i)}$:

i	$x^{(i)}$
1	4.498571
2	4.4933971
3	4.49340949
4	4.49340945.

(5-Byte-Gleitkommaarithmetik)

Das Newton-Verfahren konvergiert somit deutlich schneller als das im Abschnitt 3.2.1 behandelte Iterationsverfahren. Außerdem ist bemerkenswert, daß das Newton-Verfahren für g(x) mit jedem Startwert $x^{(0)}$ mit $x^{(0)} \neq 0$ in maximal sechs Schritten gegen die gesuchte Lösung konvergiert.
Dieses Verhalten kann aus **Bild 3.3** abgelesen werden, das die Funktion $\Lambda(f(x), f'(x), f''(x))$ darstellt.

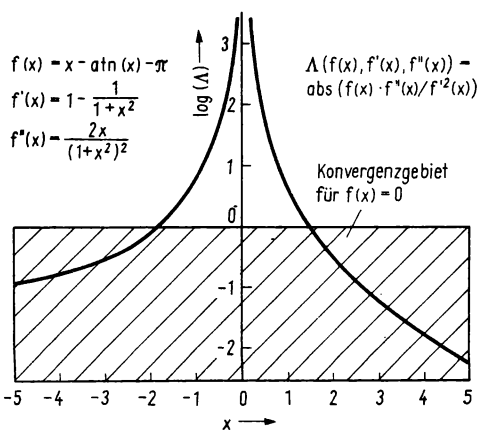


Bild 3.3. Konvergenzgebiet des Newton-Verfahrens für $f(x)=x-\text{atn}(x)-\pi=0$

Die Iteration kann mit einem beliebigen Wert $x^{(0)}$; $|x^{(0)}| > 1$ begonnen werden. Für den iterierten Wert wird Λ stets wesentlich kleiner als Eins sein, so daß die notwendige Bedingung für die Konvergenz des Newton-Verfahrens erfüllt ist. Zur Tabellierung des Konvergenzgebiets wurde das folgende kleine BASIC-Programm geschrieben:

```
10 input x0: rem Start der Tabellierung
20 input x1: rem Ende der Tabellierung
30 input d: rem Schrittweite der Tabellierung
40 for x := x0 to x1 step d
50 f = atn(x) - pi
60 fs = 1 - 1/(1 + x*x)
70 f2 = 2*x/((1 + x*x)↑2)
80 la = abs(f*f2/fs/fs)
90 print x, la
100 next i
110 end
```

Falls beim verwendeten Rechner pi nicht als Konstante fest vereinbart ist, muß zwischen die Zeilen 30 und 40 noch eine Zeile mit der Anweisung pi = 3.14159265 eingefügt werden.
Nunmehr soll die anfängliche Aufgabenstellung (3.32) nochmals betrachtet werden. Gesucht war die erste von Null verschiedene Lösung der transzendenten Gleichung $f(x) = x - \tan(x)$.

Beginnt man die Iterationen mit dem Startwert $x^{(0)} = 4$, erhält man folgende Ergebnisse:

i	$x^{(i)}$
1	6.12015848
2	238.404256
3	1957.05366
4	78504.619

Nach einigen weiteren Iterationen meldet der Rechner einen OVERFLOW ERROR.
Dieses Verhalten des Iterationsprozesses kann aus **Bild 3.4** entnommen werden.
Falls $x^{(0)} < 1$ ist, konvergiert das Newton-Verfahren gegen die triviale Lösung $x_0 = 0$.
Lediglich für $x \in [4.4, 4.6]$ liefert das Newton-Verfahren nach fünf Iterationsschritten eine hinreichend genaue Approximation der gesuchten Lösung x_0.

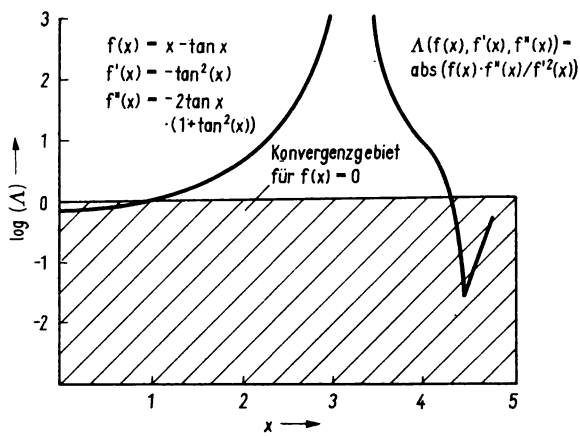


Bild 3.4. Konvergenzgebiet des Newton-Verfahrens für f(x)=x−tan(x)=0

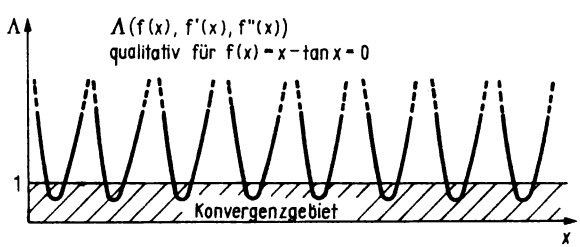


Bild 3.5. Globales Konvergenzverhalten des Newton-Verfahrens für f(x)=x−tan(x)=0

Dieses Verhalten stellt im Hinblick auf die praktische Anwendung des Newton-Verfahrens eine starke Einschränkung dar.
Bild 3.5 zeigt eine qualitative Darstellung des globalen Konvergenzverhaltens des Newton-Verfahrens für $f(x) = x - \tan(x) = 0$. Nur wenn man ein $x^{(0)}$ aus den schraffierten Intervallen wählt, kann man Konvergenz gegen eine der unendlich vielen Lösungen von (3.32) erwarten.
In allen anderen Fällen divergiert das Verfahren.
Die Schwierigkeit bei der Anwendung des Newton-Verfahrens besteht also wiederum in der

geschickten Wahl bzw. Modifikation von f(x) sowie in der Bestimmung eines Startwerts $x^{(0)}$, der hinreichend nahe an der gesuchten Lösung liegt.
In vielen praktischen Fällen läßt sich dieses "hinreichend nahe" nicht genau quantifizieren, so daß man von vornherein nichts über die Konvergenz des Verfahrens aussagen kann.
Außerdem ist zu bemerken, daß bei vielen wissenschaftlichen und technischen Problemen, die die Lösung einer transzendenten Gleichung erfordern, f'(x) nicht explizit angegeben werden kann. Meist erhält man die erste Ableitung lediglich numerisch als Ergebnis einer mehr oder minder komplizierten Subroutine.
In diesem Fall kann die Anwendung der *Sekantenregel* empfohlen werden.
Bei dieser Methode können zwar die gleichen Konvergenzprobleme wie beim Newton-Verfahren auftreten, aber f'(x) muß nicht berechnet werden. Das führt sowohl zur Einsparung von Rechenzeit als auch von Programmspeicherplatz.

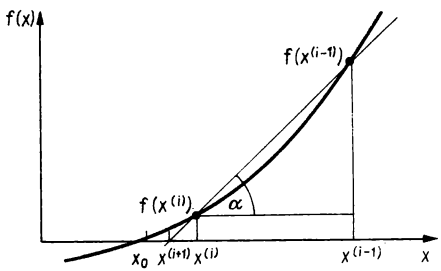


Bild 3.6. Herleitung des Sekantenverfahrens

3.2.3. Sekantenverfahren

Betrachten wir zur Herleitung des Sekantenverfahrens **Bild 3.6.**
Der Steigungswinkel der Sekante kann durch die folgende einfache Relation ermittelt werden:

$$\tan \alpha = \frac{f(x^{(i)})}{x^{(i)} - x^{(i+1)}} = \frac{f(x^{(i-1)}) - f(x^{(i)})}{x^{(i-1)} - x^{(i)}}. \tag{3.53}$$

Aus (3.53) kann unmittelbar der verbesserte Wert $x^{(i+1)}$ ermittelt werden:

$$x^{(i+1)} = x^{(i)} - \frac{(x^{(i-1)} - x^{(i)})}{(f(x^{(i-1)}) - f(x^{(i)}))} f(x^{(i)}). \tag{3.54}$$

Ein wesentlicher Nachteil des Sekantenverfahrens gegenüber dem Newton-Verfahren ist die Notwendigkeit der Kenntnis von zwei Startwerten $x^{(0)}$, $x^{(1)}$, die (3.49) erfüllen müssen.
Außerdem konvergiert das Sekantenverfahren langsamer als das Newton-Verfahren. Dies ist dadurch bedingt, daß die Tangente an f(x) für $x = x^{(i)}$ durch die Sekante ersetzt wird.

Falls zwischen $x^{(i)}$ und $x^{(i+1)}$ f(x) zu stark gekrümmt ist, kann die Sekante die Ableitung $f'(x^{(i)})$ nicht hinreichend genau approximieren.
Der dann mit (3.54) iterierte Wert $x^{(i+1)}$ wird voraussichtlich einen größeren Fehler aufweisen, als ein mit dem Newton-Verfahren berechneter Wert $x^{(i+1)}$.

```
10 input"x0, x1, eps"; x0, x1, eps
20 r = 1
30 x = x0
40 gosub 200
50 f0 = f
60 x = x1
```

```
 70 gosub 200
 80 f1 = f
 90 r = r + 1
100 xn = x1 - (x0 - x1)/(f0 - f1)*f1
110 x0 = x1
120 x1 = xn
130 f0 = f1
140 x = x1
150 gosub 200
160 f1 = f
170 print r, x1
180 if abs(x0-x1) >eps then 90
190 end
200 rem funktion
210 f = x - atn(x) - π
220 return

x0 = 4
x1 = 5
eps = 1e - 9
```

Anzahl Funktions-auswertungen	x-Wert
1	4.49076231
2	4.49339678
3	4.49340945

Bereits nach drei Iterationsschritten ist die im Rahmen der 5-Byte-Gleitkommaarithmetik exakte Lösung gefunden.
In diesem Fall konvergiert das Sekantenverfahren offensichtlich schneller als das Newton-Verfahren.
Für $f(x) = \tan(x) - x$ konvergiert die Sekantenregel nach acht Iterationsschritten gegen die gesuchte Lösung x_0, falls die Startwerte $x^{(0)}$, $x^{(1)}$ im Intervall [4.4, 4.6] lagen.
Die bisherigen Ausführungen bezogen sich lediglich auf die Ermittlung einfacher Nullstellen von f(x). Für die Bestimmung mehrfacher Nullstellen existieren spezielle Verfahren, die allerdings an dieser Stelle nicht besprochen werden können. Dem interessierten Leser sei deshalb [Enge85] empfohlen.

3.2.4. Bisektionsverfahren

Aus den Betrachtungen in den Abschnitten 3.2.1, 3.2.2 und 3.2.3 folgt, daß die Bestimmung der Nullstellen transzendenter Funktionen ein u. U. recht schwieriges und zeitraubendes Problem sein kann.
Für praktische Belange benötigt man eine Methode, die stets konvergiert, besonders dann, wenn nur grobe Abschätzungen über die Lage von x_0 im Suchintervall $[x^{(0)}, x^{(1)}]$ möglich sind.
Außerdem sollte diese Methode möglichst ohne die analytische oder numerische Berechnung von Ableitungen der Funktion f(x) auskommen.
Eine solche Methode existiert. Sie basiert auf folgendem Satz:

- Ist f(x) eine stetige Funktion auf dem abgeschlossenen Intervall $[x^{(0)}, x^{(1)}]$ und gilt $f(x^{(0)}) \cdot f(x^{(1)}) < 0$, dann existiert in $(x^{(0)}, x^{(1)})$ mindestens eine Nullstelle ungerader Ordnung.

Die auf der Anwendung dieses Satzes basierenden numerischen Verfahren nennt man *Bisektionsverfahren.*
Der Grundalgorithmus dieser Verfahren läßt sich am besten durch das folgende BASIC-Programm darstellen:

```
 10 input x0, x1, eps
 20 x = x0
 30 gosub 200
 40 f1 = f
 50 x = (x0 + x1)/2
 60 gosub 200
 70 if f1*f < 0 then 100
 80 x0 = x
 90 goto 110
100 x1 = x
110 print x0, x1
120 if abs (x0 - x1) > eps then 50
130 end
200 rem funktion
210 f = x - atn (x)- π
220 return
```

Für die Startwerte $x^{(0)}$, $x^{(1)} \sim 4.4, 4.5$ erhält man die folgenden *Einschließungsintervalle:*

```
x0 = 4.4
x1 = 4.5
eps = 1e - 8
```

x0	x1
4.45	4.5
4.475	4.5
4.4875	4.5
4.4875	4.49375
4.490625	4.49375
4.4921875	4.49375
4.49296875	4.49375
4.49335938	4.49375
4.49335938	4.49355469
4.49335938	4.49345703
4.4934082	4.49345703
4.4934082	4.49343262
4.4934082	4.49342042
4.4934082	4.49341431
4.4934082	4.49341126
4.4934082	4.49340973
4.49340897	4.49340973
4.49340935	4.49340973
4.49340935	4.49340955
4.49340945	4.49340955
4.49340945	4.4934095
4.49340945	4.49340947
4.49340945	4.49340946
4.49340945	4.49340946

Bemerkenswert ist, daß sich diese Einschließungsintervalle sowohl für f(x) = x − tan (x) als auch für f(x) = x − atn (x) − π ergeben.
Die Konvergenz eines Bisektionsverfahrens kann beschleunigt werden, indem die durch die Berechnung der Funktionswerte gewonnenen Informationen zur Ermittlung der Einschließungsintervalle herangezogen werden.
Beispiele hierfür sind das in [Enge85] angegebene *"Pegasusverfahren"* sowie das in den **Programmen P 3.06 ROOT.TEST** und **P 3.07 ROOT.PROG** numerisch realisierte *Bisektionsverfahren nach Shampine* [Sham73].

Programm P 3.06. ROOT.TEST. Testbeispiel für die Lösung einer nichtlinearen Gleichung f(x)=0, die reelle Wurzeln im Intervall x0, x1 aufweist.

```
 10 rem Testbeispiel ROOT
 20 u=1
 30 if 1+u/2<>1 then u=u/2
 40 if 1+u/2<>1 then 30
 50 print :print
 60 print "   wurzeln nichtlinearer reellwertiger":print "funktionen"
 70 print :print
 80 input "ist f(x) definiert j/n";a$
 90 if a$<>"j" and a$<>"n" then 80
100 print :print
110 if a$="n" then print "definiere funktion f(x) in zeile 22764 !":stop
120 input "relativer fehler";f1
130 input "absoluter fehler";f2
140 print
150 print "intervallgrenzen"
160 print
170 input "untere grenze (x0)";x0
180 input "obere  grenze (x1)";x1
190 print
200 e1%=1
210 for k=1 to 10000
220 gosub 22600:rem ROOT-Steuerprogramm
230 if e1%>0 then 270
240 gosub 22764:rem Funktionsauswertung
250 f7=f
260 next k
270 print :print
280 print "          e r g e b n i s s e"
290 print "          ********************"
300 print
310 if e1%=1 then print "abbruchkriterium ist erfuellt"
320 if e1%=1 then print "x=";x,"f=";f
330 if e1%=2 then print "f(x0) ist genau 0."
340 if e1%=2 then print "das intervall erfuellt das abbruchkri-"
350 if e1%=2 then print "terium eventuell nicht."
360 if e1%=3 then print "abs f(x1) wird sehr gross. probleme mit
    zahlueberlauf."
370 if e1%=3 then print "x0 liegt in der naehe eines poles von f(x)."
380 if e1%=4 then print "keine wurzel im reellen gefunden."
390 if e1%=4 then print "eventuell liegt zwischen x0 und x1 ein  lokales
    minimum."
400 if e1%=5 then print "500 funktionsauswertungen bis jetzt benoetigt."
410 print :print "x0=";x0,"x1=";x1:print "f=";f
420 print
430 input "weiter j/n";a$
440 print
450 if a$<>"j" and a$<>"n" then 430
460 if a$="j" then goto 210
470 end
```

Die zu lösende Gleichung ist in der Zeile 22764 zu definieren. Die geforderte relative bzw. absolute Genauigkeit der Iteration kann durch die globalen Variablen f1, f2 vorgegeben werden. Die iterierte Lösung steht auf der Variablen x, der korrespondierende Funktionswert auf f.

Die Variable e1% ist das Fehlerflag:

e1%=1 a) Wert vor Aufruf von ROOT.PROG
b) Abbruchkriterium ist erfüllt

e1%=2 f(x0) ist genau 0; das Intervall erfüllt das Abbruchkriterium evtl. nicht

e1%=3 abs(f(x1)) wird sehr groß, es kann zu Zahlenbereichsüberschreitungen kommen; die Variable x0 liegt in der Nähe eines Poles von f(x)

e1%=4 keine Wurzel im Reellen gefunden; zwischen x0, x1 kann ein lokales Minimum liegen

e1%=5 bisher 500 Auswertungen von f(x) benötigt.

Programm P 3.07. ROOT.PROG. Bestimmung von reellen Nullstellen einer in Zeile 22764 definierten Funktion f(x) im Intervall x0, x1 mit vorgebbarer relativer (f1) bzw. absoluter (f2) Genauigkeit.

Falls auf Rechnern mit $u \approx 2.328.10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

```
22599 rem !-------------------------------------------------!
22600 rem !               Root-Steuerprogramm               !
22601 rem !-------------------------------------------------!
22602   if e1%>=0 then 22612
22604   e1%=abs(e1%)
22606   if e1%=1 then 22630
22608   if e1%=2 then 22640
22610   if e1%=3 then 22720
22612   e7=u
22614   if f1>e7 then e7=f1
22616   e8=abs(f2)
22618   i7=0
22620   b7=abs(x0-x1)
22622   h7=x1
22624   x=h7
22626   e1%=-1
22628   goto 22636
22630   f8=f7
22632   x=x0
22634   e1%=-2
22636 return
22637 rem !-------------------------------------------------!
22638 rem !                  Hauptroutine                   !
22639 rem !-------------------------------------------------!
22640   f9=f7
22642   g7=f8
22644   j7=2
22646   g8=abs(f9)
22648   if abs(g7)>g8 then g8=abs(g7)
22650   rem Iterationszyklus
22652   if abs(g7)>=abs(f9) then 22666
22654   h7=x0
22656   f8=f9
22658   x0=x1
```

```
22660   f9=g7
22662   x1=h7
22664   g7=f8
22666   c7=.5*(x1-x0)
22668   a7=abs(c7)
22670   t7=e7*abs(x0+e8)
22672   if a7<=t7 then 22736
22674   if j7>500 then 22758
22676   p7=(x0-h7)*f9
22678   q7=f8-f9
22680   if p7>=0 then 22686
22682   p7=-p7
22684   q7=-q7
22686   h7=x0
22688   f8=f9
22690   i7=i7+1
22692   if i7<4 then 22700
22694   if 8*a7>=b7 then 22712
22696   i7=0
22698   a7=b7
22700   if p7>abs(q7*t7) then 22706
22702   x0=x0+abs(t7*sgn(c7))
22704   goto 22714
22706   if p7>=c7*q7 then 22712
22708   x0=x0+p7/q7
22710   goto 22714
22712   x0=.5*(x1+x0)
22714   x=x0
22716   e1%=-3
22718   goto 22760
22720   f9=f7
22722   if f9=0 then 22746
22724   j7=j7+1
22726   if sgn(f9)<>sgn(g7) then 22652
22728   x1=h7
22730   g7=f8
22732   goto 22638
22734   rem Test auf erfolgreiche Beendigung der Routine
22736   if sgn(f9)=sgn(g7) then 22754
22738   if abs(f9)>g8 then 22750
22740   e1%=1
22742   goto 22760
22744   rem Setzen der Rückkehrcodes<>1
22746   e1%=2
22748   goto 22760
22750   e1%=3
22752   goto 22760
22754   e1%=4
22756   goto 22760
22758   e1%=5
22760 return
22761 rem !-------------------------------------------------!
22762 rem !                        f(.)                     !
22763 rem !-------------------------------------------------!
22764   f=((abs(sin(x)-1))^.001)*1e5
22766 return
22767 rem !-------------------------------------------------!
```

Werden mit vorgegebenen Werten f1, f2 keine Lösungen gefunden, so besagt dies nicht notwendig, daß sich im Intervall x0, x1 keine Lösungen befinden. Die Berechnungen sollten in diesem Fall mit variierten Werten f1, f2 wiederholt werden.

Der Modul P 3.06 ROOT.TEST ist im Scratchbereich angeordnet. Er umfaßt den Zeilenbereich von 10 bis 470. Die globalen Eingabeparameter
f1 zulässiger relativer Fehler der Lösung
f2 zulässiger absoluter Fehler der Lösung
x0 untere Grenze des Suchgebiets
x1 obere Grenze des Suchgebiets
werden interaktiv (Zeilen 120 bis 180) eingegeben.
Die vom Modul P 3.07 ROOT.PROG gelieferten Fehlerkodes e1% werden von P 3.06 ROOT.TEST ausgewertet. Ihre Bedeutung wird im Klartext ausgegeben (Zeilen 310 bis 400).
In Zeile 430 wird abgefragt, ob der Nutzer mit den bisher gelieferten Ergebnissen zufrieden ist oder ob weiter iteriert werden soll. Diese Abfrage kann so lange mit "j" für "ja" beantwortet werden, bis an den Ergebnissen x0, x1 und dem Funktionswert f(x0) keine Veränderungen mehr festgestellt werden können.
Meist kann durch diese Vorgehensweise eine Verbesserung der relativen und absoluten Genauigkeit des Ergebnisses gegenüber den Vorgabewerten f1, f2 erreicht werden.
Der Aufruf des Bisektionsverfahrens nach *Shampine* erfolgt in der **for-next**-Schleife in den Zeilen 210 bis 260.
Die Anweisungen innerhalb dieser Schleife werden abgearbeitet, bis P 3.07 ROOT.PROG einen Fehlercode e1% ≠ 0 generiert.
Dann erfolgt ein Sprung zur Zeile 270, und der durch den Wert von e1% definierte Klartext wird ausgegeben.
Die interaktive Arbeitsweise von P 3.06 ROOT.TEST ist notwendig, um ohne A-priori-Informationen über f(x) möglichst schnell die gesuchten Nullstellen lokalisieren zu können.
So ist es z. B. ohne Änderungen der Software möglich, das Suchgebiet in sich überlappende Teilsuchgebiete einzuteilen und durch P 3.07 ROOT.PROG abarbeiten zu lassen.
Wird in einem Teilsuchgebiet keine Nullstelle gefunden, ist die Abfrage in Zeile 430 mit "n" für "nein" zu beantworten.
Danach startet man P 3.06 ROOT.TEST erneut mit anderen Vorgabewerten x0, x1.
Hat man keine genauen Vorstellungen von der Lage einer Nullstelle, sollten f1 und f2 anfänglich nicht zu klein gewählt werden. In diesem Fall sind Werte für f1, f2 um 10^{-4} angebracht.
Das Programm ROOT kann auch zur Überprüfung der Stetigkeit von f(x) im Suchintervall herangezogen werden. Liegt zwischen x0 und x1 ein Pol von f(x), wird dies durch den Rückkehrkode e1% = 3 signalisiert.
Der Nutzer des Programms ROOT wird somit weitgehend von vorbereitenden Routinearbeiten entlastet.
Falls das Programm ROOT einen Pol von f(x) erkennt, muß die Umgebung des Poles aus dem Suchgebiet ausgeklammert werden. Man bildet dann jeweils ein neues Suchgebiet links und rechts vom Pol. Dieser Fall tritt bei dem schon mehrfach diskutierten Beispiel f(x) = x − tan(x) = 0 auf. Die Funktion f(x) weist für x = 1.5 π einen Pol auf. Wird ein Suchgebiet x0, x1 ~ 4,5 vorgegeben, dann erhält man nach einigen Sekunden vom Programm die entsprechende verbale Rückmeldung verbunden mit der Angabe des vermutlichen Poles.
Ein Neustart des Programms mit den beiden Suchgebieten x0, x1 ~ 4,4.7 und x0, x1 ~ 4.72,5 liefert im ersten Suchintervall die Nullstelle x_0 = 4.49340943. Nach Bearbeitung des zweiten Intervalls wird eine Fehlmeldung ausgegeben.
Das eigentliche Bisektionsverfahren ist im Modul P 3.07 ROOT.PROG realisiert.
Der Modul besteht aus dem Steuerprogramm in den Zeilen 22600 bis 22636, der Hauptroutine (22638 bis 22760) sowie dem Unterprogramm für f(x) in den Zeilen 22762 bis 22766.
Das Steuerprogramm ruft je nach Wert des Fehlerflags e1% die einzelnen Teile der Hauptroutine auf.
In der Hauptroutine von P 3.07 ROOT.PROG ist das Bisektionsverfahren nach *Shampine* rea-

lisiert, das aus Platzgründen nicht weiter erläutert werden soll. Der interessierte Leser wird auf die angegebene Literatur [Sham73] verwiesen.
Als eines von vielen Testbeispielen für das Programm ROOT wurde die in Zeile 22764 programmierte Funktion:
f(x) = (abs(sin(x) − 1)^(0.001)) * 1e5
herangezogen. Die erste Nullstelle für $x > 0$ liegt bei $x_0 = 0.5\,\pi$.
Dieses Ergebnis liefert das Programm ROOT bei Vorgabe des Suchintervalls x0, x1 ~ 1,2 nach etwa 5 s (6510-Prozessor).

3.3. Nichtlineare Gleichungssysteme

3.3.1. Iterationsverfahren

In Analogie zum eindimensionalen Fall wird ein zu (3.3) äquivalentes System

$$x_i = F_i(x_1,x_2,\ldots,x_i,\ldots,x_n);\ n \in N,\ n \geqq 2 \tag{3.55}$$

gebildet. Der Vektor $\bar{x} = (x_1,x_2,x_3,\ldots x_n)^T$ heißt Fixpunkt von (3.3), falls gilt $\bar{x} = \bar{F}(\bar{x})$.
Mit Hilfe eines geeignet zu bestimmenden Startvektors $\bar{x} \in D_F$ (Definitionsbereich von F) wird eine Iterationsfolge $\{\bar{x}^{(i)}\}$ nach der Vorschrift

$$\bar{x}^{(i+1)} = \bar{F}(\bar{x}^{(i)});\ i = 0, 1, 2, 3\ldots \tag{3.56}$$

konstruiert. Die Vektorfunktion $\bar{F}$ heißt wiederum Schrittfunktion, (3.56) in Analogie zu (3.20) Iterationsvorschrift.
Es lassen sich notwendige Bedingungen für die Konvergenz des Iterationsverfahrens (3.56) formulieren, die denen des eindimensionalen Falles entsprechen.
Das Äquivalent zu $F'(x)$ im eindimensionalen Fall ist die *Funktionalmatrix* **J** (*Jacobi-Matrix*) für $n > 1$. Sie kann nur berechnet werden, wenn die F_i in D_F stetige partielle Ableitungen aufweisen.
Es gilt:

$$\mathbf{J} = \left(\frac{\delta F_i}{\delta x_j}\right) = \begin{pmatrix} \frac{\delta F_1}{\delta x_1} & \frac{\delta F_2}{\delta x_2} & \frac{\delta F_1}{\delta x_n} \\ & & \\ \frac{\delta F_n}{\delta x_1} & \frac{\delta F_n}{\delta x_2} & \frac{\delta F_n}{\delta x_n} \end{pmatrix} \tag{3.57}$$

Die für die Konvergenz der Iteration (3.56) notwendige Lipschitz-Bedingung lautet:

$$\| \mathbf{J} \| \leqq L < 1, \tag{3.58}$$

sofern D_F konvex und abgeschlossen ist.
Der Nachweis der Konvergenz eines Iterationsverfahrens gestaltet sich im mehrdimensionalen Fall noch weitaus komplizierter als im eindimensionalen Fall. Deshalb sollen an dieser Stelle keine weiteren Betrachtungen zu dieser Problematik folgen. Der interessierte Leser wird auf die Literatur [Enge85] verwiesen.

3.3.2. Newton-Verfahren für Systeme

Gegeben sei ein System nichtlinearer Gleichungen (3.3) mit einer Lösung im Definitionsbereich D_f.
Die Funktionen f_i müssen in D_f stetige partielle erste und zweite Ableitungen aufweisen.
Außerdem muß für die Jacobi-Matrix

$$\mathbf{J} = \left(\frac{\delta f_i}{\delta x_j}\right);\ i = 1\,(1)\,n;\ j = 1\,(1)\,n \tag{3.59}$$

gelten

$$\det(\mathbf{J}) \neq 0. \tag{3.60}$$

Dann existiert in Analogie zum eindimensionalen Fall immer eine Umgebung $D_0 \in D_f$ so, daß die Schrittfunktion des Newton-Verfahrens

$$\bar{x}^{(i+1)} = \bar{x}^{(i)} - \mathbf{J}^{-1}(\bar{x}^{(i)}) \cdot \bar{f}(\bar{x}^{(i)}) \tag{3.61}$$

gegen die Lösung $\bar{x}_0 : \bar{f}(\bar{x}_0) = \bar{0}$ des Systems (3.3) konvergiert.
Je Iterationsschritt (3.60) ist eine Matrixinversion bzw. die Lösung eines linearen Gleichungssystems erforderlich. Das bedingt für größere Systeme einen nicht unerheblichen Rechenaufwand.
Deshalb kann man z. B. so vorgehen, daß man eine bestimmte Anzahl von Schritten mit fester Jacobi-Matrix rechnet.
Außerdem ist es empfehlenswert, den Anteil $\mathbf{J}^{-1}(\bar{x}^{(i)}) \cdot \bar{f}(\bar{x}^{(i)})$ in (3.60) mit einem Faktor $p < 1$ so lange zu multiplizieren, bis gilt

$$\| \bar{f}(\bar{x}^{(i+1)}) \| < \| \bar{f}(\bar{x}^{(i)}) \|. \tag{3.62}$$

Diese Vorgehensweise dient der Verhinderung oszillierender Lösungen.
Ersetzt man den Differentialquotienten in (3.60) durch den zentralen Differenzenquotienten, gelangt man zur Regula Falsi für Systeme.
Die Anwendung der Regula Falsi ist dann notwendig, wenn die Jacobi-Matrix nicht explizit berechenbar ist.
Die Konvergenz des Newton-Verfahrens bzw. der Regula Falsi kann nur gewährleistet werden, wenn der Startvektor $\bar{x}^{(0)}$ nahe genug bei der Lösung $\bar{x}_0$ des Systems liegt. Es treten also die gleichen Probleme wie im eindimensionalen Fall auf.
Hat man allerdings einen Vektor $\bar{x}^{(0)}$ gefunden, der diese Bedingung erfüllt, dann konvergiert das Newton-Verfahren quadratisch, d. h. mit der Konvergenzordnung zwei. Die Konvergenzordnung der Regula Falsi beträgt etwa 1.3.

3.3.3. Gradientenverfahren

Die Lösung des Systems (3.3) wird auf die Lösung von

$$\bar{Q}(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} f_i^2\,(x_1, x_2, \ldots x_n) = 0 \tag{3.63}$$

zurückgeführt. Es ist einzusehen, daß gilt

$$\bar{Q}(\bar{x}) = 0 \Leftrightarrow \bar{f}(\bar{x}) = 0. \tag{3.64}$$

Mit

$$\operatorname{grad}(\bar{Q}(\bar{x})) \doteq \left(\frac{\delta \bar{Q}}{\delta x_1}, \frac{\delta \bar{Q}}{\delta x_2}, \ldots, \frac{\delta \bar{Q}}{\delta x_n}\right)^T \tag{3.65}$$

ergibt sich die *Schrittfunktion des Gradientenverfahrens* für Systeme zu

$$\bar{x}^{(i+1)} = \bar{x}^{(i)} - \frac{\overline{Q}\,(\bar{x}^{(i)})}{(\text{grad}\,(\overline{Q}(\bar{x}^{(i)})))^2}\,\text{grad}\,(\overline{Q}(\bar{x}^{(i)})). \tag{3.66}$$

Für das Gradientenverfahren gelten die gleichen Konvergenzaussagen wie für das Newton-Verfahren. Es hat sich allerdings gezeigt, daß man beim Gradientenverfahren mit gröberen Ausgangsnäherungen $\bar{x}^{(0)}$ als beim Newton-Verfahren starten kann.
Allerdings konvergiert das Gradientenverfahren nur linear. Das Gradientenverfahren weist noch einen bemerkenswerten Nachteil gegenüber dem Newton-Verfahren auf. Es konvergiert nicht nur gegen Vektoren $\bar{x}_0$, für die $\bar{f}(\bar{x}_0) = 0$ gilt, sondern auch gegen solche Vektoren $\bar{x}_0$, für die der Gradient von $\overline{Q}(\bar{x})$ verschwindet (*Nichtnullminima*).
Für praktische Anwendungen ist es also naheliegend, das Gradientenverfahren (großer Konvergenzbereich) mit dem Newton-Verfahren (schnelle Konvergenz) zu kombinieren. Somit könnte man die Vorteile beider Verfahren nutzen und ihre Nachteile weitestgehend eliminieren.

3.3.4. Such-Weg-Verfahren

Das *Such-Weg-Verfahren* ist eine Kombination des Gradientenverfahrens mit dem Newton-Verfahren [Enge85]. Es kommt mit sehr groben Anfangsnäherungen aus und findet trotzdem in vergleichsweise wenigen Rechenschritten eine Lösung des Systems (3.3.).
Durch das Gradientenverfahren werden die Ausgangsnäherungen $\bar{x}^{(0)}$ soweit verbessert, daß sie als Startnäherungen für das quadratisch konvergente Newton-Verfahren dienen können.
Dabei wird von der Problemformulierung (3.63) ausgegangen. Im Suchschritt des Verfahrens wird eine Näherung $\bar{x}^{(1)}$ nach der Vorschrift

$$\bar{x}^{(1)} = \bar{x}^{(0)} + \alpha\Delta\,\bar{x}^{(0)} \tag{3.67}$$

mit

$$\bar{x}^{(0)} = -\frac{\overline{Q}\,(\bar{x}^{(0)})\,\text{grad}\,(\overline{Q}\,(\bar{x}^{(0)}))}{(\text{grad}\,(\overline{Q}\,(\bar{x}^{(0)})))^2} \tag{3.68}$$

$$\alpha = \frac{3\,P_0 - 4\,P_1 + P_2}{4\,(P_0 - 2\,P_1 + P_2)};\ \alpha \in [0,1] \tag{3.69}$$

berechnet.
Dabei gilt

$$P_0 = |\,\text{grad}\,(\overline{Q}\,(\bar{x}^{(0)}))\,| \tag{3.70}$$

$$P_1 = |\,\text{grad}\,(\overline{Q}\,(\bar{x}^{(0)} + 0{,}5\,\Delta\bar{x}^{(0)}))\,| \tag{3.71}$$

$$P_2 = |\,\text{grad}\,(\overline{Q}\,(\bar{x}^{(0)} + \Delta\bar{x}^{(0)}))\,|. \tag{3.72}$$

Im "Weg-Schritt" wird der Vektor $\bar{x}^{(2)}$ nach folgender Vorschrift ermittelt:

$$\bar{x}^{(2)} = \bar{x}^{(1)} + \beta\Delta\bar{x}^{(1)}. \tag{3.73}$$

Der Vektor $\Delta\bar{x}^{(1)}$ wird nach (3.68) berechnet, wobei alle Indizes (0) durch (1) zu ersetzen sind.
Der skalare Faktor β ergibt sich zu

$$\beta = \frac{3\,Q_0 - 4\,Q_1 + Q_2}{4\,(Q_0 - 2\,Q_1 + Q_2)} \tag{3.74}$$

mit $\overline{Q}_0 = \overline{Q}\,(\bar{x}^{(0)})$, $\overline{Q}_1 = \overline{Q}\,(\bar{x}^{(0)} + 0{,}5\,\Delta\bar{x}^{(1)})$, $\overline{Q}_2 = \overline{Q}\,(\bar{x}^{(0)} + \Delta\bar{x}^{(1)})$.

Zur Steuerung des Verfahrens wird die Größe H herangezogen:

$$H = \max_{j = 1\,(1)\,n} | \bar{x}_j^{(i+1)} - \bar{x}_j^{(i)} | \; / \; \max_{j = 1\,(1)\,n} | \bar{x}_j^{(i)} - \bar{x}_j^{(i-1)} |. \tag{3.75}$$

Nach jeweils fünf Such-Weg-Schritten wird die erreichte Näherung $x^{(10)}$ als Startwert für das Newton-Verfahren verwendet. Wenn die H-Werte aufeinanderfolgender Newton-Iterationen abnehmen, kann man davon ausgehen, daß das Newton-Verfahren konvergiert. In diesem Fall wird so lange mit dem Newton-Verfahren weitergerechnet, bis ein Abbruchkriterium erfüllt ist oder durch wachsende H-Werte (H > 2) ein Nichtnullminimum erkannt wird. In diesem Fall werden wiederum fünf Such-Weg-Schritte ausgeführt, d. h., der Iterationszyklus beginnt von neuem. Dieser Algorithmus ist im **Programm NLGS** numerisch realisiert. Das recht umfangreiche Programm besteht aus den Modulen **P 3.09 NLGS.INIT, P 3.10 NLGS.PROG, P 3.11 NLGS.HELP** und **P 3.12 NLGS.EQNS.**
Wichtige Voraussetzung für die Anwendbarkeit des Programms ist die explizite Kenntnis der Jacobi-Matrix (3.59), die im Suchbereich des Verfahrens nicht singulär werden darf.

Programm P 3.09. NLGS.INIT. Initialisierung des Verfahrens zur Lösung nichtlinearer Gleichungssysteme.

```
22799 rem !-------------------------------------------------!
22800 rem !                    Init NLGS                    !
22801 rem !-------------------------------------------------!
22802   if e1%<>1 then 22862
22804   u=1
22806   if 1+u/2<>1 then u=u/2
22808   if 1+u/2<>1 then 22806
22810   if n<1 or dx<=u or df<=u then 22862
22812   dim a(n,n)
22814   dim b(n,4)
22816   dim c(n,4)
22818   dim d(n,n,4)
22820   dim e(n,n)
22822   dim f(n)
22824   dim g(n,4)
22826   dim h(n)
22828   dim j(4)
22830   dim k(4)
22832   dim l(4)
22834   dim m(4)
22836   dim o(n)
22838   dim x(n)
22840   z7=0
22842   e7=sqr(df)
22844   for i7=1 to 4
22846     k(i7)=1
22848     l(i7)=1
22850   next i7
22852   for i7=1 to n
22854     b(i7,1)=i(i7)
22856   next i7
22858   e1%=32
22860   goto 22864
22862   e1%=2
22864 return
22865 rem !-------------------------------------------------!
```

Geprüft werden die Vorgabewerte n, dx, df. Die benötigten Felder werden dimensioniert bzw. initialisiert. Treten Fehler auf, so wird die Routine mit e1%=2 verlassen, sonst mit e1%=32.

Programm P 3.10. NLGS.PROG. Such-Weg-Verfahren zur Lösung nichtlinearer Gleichungssysteme der Form
$f_i(x_1,x_2,\ldots,x_n)=0;\ i=1,2,\ldots n.$

```
22899 rem !-------------------------------------------------!
22900 rem !                        NLGS                     !
22901 rem !-------------------------------------------------!
22904   i7=1
22906   gosub 23374
22908   if m(1)=0 then e1%=0
22910   if m(1)=0 then 23356
22912   rem Gradientenverfahren falls Gradient <> 0
22914   i7=1
22916   gosub 23398
22918   q7=m(1)
22920   i7=1
22922   gosub 23482
22924   if g7<>0 then 22974
22926   rem Newton-Verfahren
22928   gosub 23420
22930   if e1%<>0 then e1%=4
22932   if e1%<>0 then 23356
22934   s7=l(2)/l(1)
22936   if s7>1e+10 then e1%=4
22938   if s7>1e+10 then 23356
22940   h7=.5
22942   if s7>5 then h7=5/s7
22944   if m(2)>m(1) then 22960
22946   i7=2
22948   j7=1
22950   gosub 23506
22952   if l(1)<dx then 23350
22954   i7=1
22956   gosub 23398
22958   goto 22928
22960   if l(2)<dx then 23350
22962   i7=2
22964   j7=2
22966   gosub 23522
22968   l(2)=h7*l(2)
22970   goto 22944
22972   rem Gradientenverfahren
22974   gosub 23462
22976   k(2)=h8
22978   h7=5*k(1)/k(2)
22980   if h7<1 then 22990
22982   i7=2
22984   gosub 23374
22986   if m(2)<=m(1) then 23004
22988   h7=.5
22990   i7=2
22992   j7=2
22994   gosub 23522
22996   k(2)=h7*k(2)
22998   if k(2)<=dx then 22928
23000   if m(2)>m(1) then h7=7.5
23002   if m(2)>m(1) then 22990
23004   i7=3
23006   j7=2
23008   h7=.5
23010   gosub 23522
23012   k(3)=.5*k(2)
23014   i7=2
23016   gosub 23398
```

```
23018   gosub 23482
23020   i7=3
23022   gosub 23398
23024   gosub 23482
23026   h7=j(1)-2*j(3)+j(2)
23028   if h7<=0 then 23318
23030   h7=(3*j(1)-4*j(3)+j(2))/(2*h7)
23032   if h7<=0 then 23318
23034   if h7>3 then h7=3
23036   i7=4
23038   j7=3
23040   gosub 23522
23042   k(4)=h7*k(3)
23044   if m(4)>m(1) then 23318
23046   i7=4
23048   gosub 23398
23050   gosub 23482
23052   if j(4)>j(2) or j(4)>j(1) then 23318
23054   l7=3
23056   if j(4)<=j(3) then l7=4
23058   for j7=1 to n
23060     c(j7,1)=c(j7,l7)
23062     b(j7,1)=b(j7,l7)
23064     for k7=1 to n
23066       d(j7,k7,1)=d(j7,k7,l7)
23068     next k7
23070   next j7
23072   m(1)=m(l7)
23074   k(1)=k(l7)
23076   m7=0
23078   i7=1
23080   gosub 23482
23082   if g7=0 then 22928
23084   gosub 23462
23086   l(2)=h8
23088   h7=2*l(1)/l(2)
23090   if h7<1 then 23108
23092   i7=2
23094   gosub 23374
23096   if m(2)<=m(1) then 23136
23098   h7=.5
23100   m7=1
23102   i7=2
23104   j7=3
23106   gosub 23506
23108   i7=2
23110   j7=2
23112   gosub 23522
23114   l(2)=h7*l(2)
23116   if l(2)<=dx then 22928
23118   i7=2
23120   gosub 23374
23122   if m(2)<=m(1) then 23136
23124   h7=.5
23126   m7=1
23128   i7=2
23130   j7=3
23132   gosub 23506
23134   goto 23108
23136   if m7<>0 then 23148
23138   i7=3
23140   if m7=0 then j7=2
23142   h7=2
23144   gosub 23522
23146   l(3)=2*l(2)
23148   h7=m(1)-2*m(2)+m(3)
```

```
23150   if h7=0 then 23156
23152   h7=(3*m(1)-4*m(2)+m(3))/(2*h7)
23154   if h7>3 then h7=3
23156   if h7<=0 then h7=2
23158   i7=4
23160   j7=2
23162   gosub 23522
23164   l(4)=h7*l(2)
23166   if h7<=2 then 23180
23168   if m(3)>=m(4) then i7=4
23170   if m(3)>=m(4) then 23200
23172   h7=h7/2-1
23174   if h7<2.2 then i7=3
23176   if h7<2.2 then 23200
23178   goto 23158
23180   if m(4)<=m(1) then 23194
23182   i7=2
23184   j7=3
23186   gosub 23506
23188   h7=.5
23190   m7=1
23192   goto 23108
23194   i7=2
23196   if m(i7)>m(3) then i7=3
23198   if m(i7)>m(4) then i7=4
23200   j7=1
23202   gosub 23506
23204   ifabs(q7-m(1))/m(1)>=e7 then 23212
23206   i7=1
23208   gosub 23398
23210   goto 22928
23212   z7=z7+1
23214   if m(1)>df then 23226
23216   if l(1)<dx then e1%=0
23218   if l(1)<dx then 23356
23220   i7=1
23222   gosub 23398
23224   goto 22928
23226   if l(1)>e7 then 23234
23228   i7=1
23230   gosub 23398
23232   goto 22928
23234   if (z7/5)=int(z7/5) then 23242
23236   if z7<100 then 22914
23238   e1%=4
23240   goto 23356
23242   i7=1
23244   j7=3
23246   gosub 23506
23248   k(3)=k(1)
23250   i7=1
23252   gosub 23398
23254   gosub 23420
23256   if e1%<>0 then e1%=4
23258   if e1%<>0 then 23356
23260   s=l(2)/l(1)
23262   if s7>1e+10 then e1%=4
23264   if s7>1e+10 then 23356
23266   if m(2)<=m(1) then 23180
23268   i7=3
23270   j7=1
23272   gosub 23506
23274   k(1)=k(3)
23276   goto 22914
23278   if s7>5 then 23296
23280   i7=2
```

```
23282   j7=1
23284   gosub 23506
23286   if m(1)>df then 23250
23288   if l(1)<dx then 23350
23290   i7=1
23292   gosub 23398
23294   goto 22928
23296   h7=5/s7
23298   i7=2
23300   j7=2
23302   gosub 23522
23304   l(2)=h7*l(2)
23306   if m(2)<=m(1) then 23280
23308   i7=3
23310   j7=1
23312   gosub 23506
23314   k(1)=k(3)
23316   goto 22914
23318   i7=2
23320   j7=4
23322   gosub 23506
23324   i7=3
23326   j7=2
23328   gosub 23506
23330   i7=4
23332   j7=3
23334   gosub 23506
23336   l(2)=k(3)
23338   l(3)=k(2)
23340   for j7=1 to n
23342     g(j7,2)=g(j7,3)
23344   next j7
23346   goto 23148
23348   rem  Programmende: Schrittweite < dx
23350   e1%=0
23352   if m(1)>df then e1%=4
23354   rem  Programmende: Werte umspeichern
23356   for j7=1 to n
23358     o(j7)=b(j7,1)
23360   next j7
23362   f=m(1)
23364 return
23365 rem !-----------------------------------------------!
```

Die Lösung wird sowohl mit dem Newton-Verfahren als auch mit einem Gradientenverfahren gesucht, um schnellstmögliche Konvergenz in einem großen Suchbereich zu gewährleisten. Treten mit den gewählten Parametern Konvergenzprobleme auf, so wird die Routine mit e1%=4 verlassen. In diesem Fall sollten die Berechnungen mit größeren Werten für die Variablen df und dx bzw. genaueren Schätzwerten i(n) wiederholt werden. Im Normalfall meldet sich die Routine mit e1%=0 zurück. Die Lösungen des Systems stehen dann auf dem Feld o(n).Falls auf Rechnern mit $u \approx 2.328.10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z=max(abs(x),abs(y)).

Programm P 3.11. NLGS.HELP. Hilfsroutinen für P 3.10 NLGS.PROG.

```
23367 rem !-------------------------------------------------!
23368 rem !                  Hilfsprogramme                 !
23369 rem !-------------------------------------------------!
23372   rem Routine 1
23374   for j7=1 to n
23376     x(j7)=b(j7,i7)
23378   next j7
23380   gosub 23622:rem nichtlineares Gleichungssystem
23382   f=0
23384   for j7=1 to n
23386     c(j7,i7)=f(j7)
23388     f=f+f(j7)*f(j7)
23390   next j7
23392   m(i7)=f
23394   goto 23532
23396   rem Routine 2
23398   for j7=1 to n
23400     x(j7)=b(j7,i7)
23402   next j7
23404   gosub 23634:rem part. Ableitungen des Systems
23406   for j7=1 to n
23408     for k7=1 to n
23410       d(j7,k7,i7)=e(j7,k7)
23412     next k7
23414   next j7
23416   goto 23532
23418   rem Routine 3
23420   for j7=1 to n
23422     x(j7)=c(j7,1)
23424 next j7
23426   for j7=1 to n
23428     for k7=1 to n
23430       a(j7,k7)=d(j7,k7,l7)
23432     next k7
23434   next j7
23436   gosub 23534
23438   if e1%<>0 then 23532
23440   h7=0
23442   for j7=1 to n
23444     g(j7,2)=-x(j7)
23446     b(j7,2)=b(j7,1)-x(j7)
23448     h7=h7+x(j7)*x(j7)
23450   next j7
23452   l(2)=sqr(h7)
23454   i7=2
23456   gosub 23374
23458   goto 23532
23460   rem Routine 4
23462   h7=m(1)/g7
23464   h8=0
23466   for j7=1 to n
23468     g(j7,2)=-h7*h(j7)
23470     b(j7,2)=b(j7,1)+g(j7,2)
23472     h8=h8+g(j7,2)*g(j7,2)
23474   next j7
23476   h8=sqr(n)
23478   goto 23532
23480   rem Routine 5
23482   g7=0
23484   for j7=1 to n
23486     h7=0
23488     for k7=1 to n
23490       h7=h7+c(k7,i7)*d(k7,j7,i7)
```

```
23492     next k7
23494     h(j7)=2*h7
23496     g7=g7+4*h7*h7
23498   next j7
23500   j(i7)=sqr(g7)
23502   goto 23532
23504   rem Routine 6
23506   for k7=1 to n
23508     b(k7,j7)=b(k7,i7)
23510     c(k7,j7)=c(k7,i7)
23512   next k7
23514   l(j7)=l(i7)
23516   m(j7)=m(i7)
23518   goto 23532
23520   rem Routine 7
23522   for k7=1 to n
23524     g(k7,i7)=h7*g(k7,j7)
23526     b(k7,i7)=b(k7,1)+g(k7,i7)
23528   next k7
23530   gosub 23374
23532 return
23533 rem !-----------------------------------------------!
23534 rem !Funktionalmatrix invertieren;neue x(i) iterieren!
23535 rem !-----------------------------------------------!
23536   if n=1 then e1%=0
23538   if n=1 then 23614
23540   for i7=1 to n
23542     s7=a(i7,i7)
23544     if i7=1 then 23552
23546     for k7=1 to i7-1
23548       s7=s7-a(i7,k7)*a(k7,i7)
23550     next k7
23552     if s7=0 then e1%=4
23554     if s7=0 then 23614
23556     a(i7,i7)=s7
23558     if i7=n then 23578
23560     for j7=i7+1 to n
23562       s7=a(i7,j7)
23563       t7=a(j7,i7)
23564       if i7=1 then 23574
23566       for k7=1 to i7-1
23568         s7=s7-a(i7,k7)*a(k7,j7)
23570         t7=t7-a(j7,k7)*a(k7,i7)
23572       next k7
23574       a(i7,j7)=s7
23575       a(j7,i7)=t7/a(i7,i7)
23576     next j7
23578   next i7
23580   for i7=1 to n
23582     s7=x(i7)
23584     if i7=1 then 23592
23586     for k7=1 to i7-1
23588       s7=s7-a(i7,k7)*x(k7)
23590     next k7
23592     x(i7)=s7
23594   next i7
23596   for i7=n to 1 step -1
23598     s7=x(i7)
23600     if i7=n then 23608
23602     for k7=i7+1 to n
23604       s7=s7-a(i7,k7)*x(k7)
23606     next k7
23608     x(i7)=s7/a(i7,i7)
23610   next i7
23612   e1%=0
23614 return
23615 rem !-----------------------------------------------!
```

Vor allem werden in diesem Programmteil oft benötigte Umspeicherungen vorgenommen. Außerdem wird die Jacobi-Matrix (Matrix der partiellen Ableitungen) invertiert. Treten Fehler während der Inversion auf, ist der Rückkehrkode e1%=4; treten keine Probleme auf, gilt e1%=0.
Falls auf Rechnern mit $u \approx 2.328.10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z=max(abs(x),abs(y)).

Programm P 3.12. NLGS. EQNS. Diese Routine enthält das zu lösende nichtlineare Gleichungssystem
$f_i(x_1,x_2,\ldots,x_n)=0$; i=1,2,. . .,n
sowie die expliziten partiellen Ableitungen des Systems (Jacobi-Matrix) $\delta f_i/\delta x_j$.

```
23617 rem !-------------------------------------------------!
23618 rem !      Nichtlineares Gleichungssystem f(i)= 0     !
23619 rem !-------------------------------------------------!
23622   f(1)=tan(x(1))+cos(x(2))
23624   f(2)=4*x(1)-x(2)
23626 return
23628 rem !-------------------------------------------------!
23630 rem !        Partielle Ableitungen df(i)/dx(j)        !
23632 rem !-------------------------------------------------!
23634   e(1,1)=2/(1+cos(2*x(1)))
23636   e(1,2)=-sin(x(2))
23638   e(2,1)=4
23640   e(2,2)=-1
23642 return
23643 rem !-------------------------------------------------!
```

Im Modul **P 3.08 NLGS.TEST** werden die für die Berechnungen benötigten globalen Variablen vereinbart und initialisiert:

n Anzahl der Unbekannten x_i bzw. der rechten Seiten f_i

i(n) Inputwerte, d. h. Vektor der Startnäherungen $\bar{x}^{(0)}$

df Abbruchschranke für Funktionswertänderungen:

$$df = \sum_{i=1}^{n} df_i^2$$

dx Abbruchschranke für die Änderungen der Variablen:

$$dx = \sum_{i=1}^{n} dx_i^2$$

f Funktionswert: $f = \sum_{i=1}^{n} f_i^2$

o(n) Outputwerte, d. h. als Lösungen des Systems akzeptierte Werte x_i, die die vorgegebenen Abbruchbedingungen erfüllen.

Die Fehlerkodes e1% haben die in der Legende zu P 3.08 NLGS.TEST angegebenen Bedeutungen.
Durch den Modul P 3.08 NLGS.TEST wird die Lösung des in P 3.12 NLGS.EQNS vereinbarten nichtlinearen Gleichungssystems

$$f_1 = \tan(x_1) + \cos(x_2)$$

$$f_2 = 4x_1 - x_2$$

vorbereitet. Ein Lösungsvektor dieses Systems ist offensichtlich, da es sich um ein konstruiertes Beispiel handelt. Er lautet:

$$x_0 = (x_1, x_2) = (\pi/4, \pi).$$

Programm P 3.08. NLGS.TEST. Testbeispiel zur Lösung nichtlinearer Gleichungssysteme der Form
$f_i(x_1, x_2, \ldots, x_i, \ldots, x_n) = 0$; mit $i = 1,2,\ldots,n$
durch das Such-Weg-Verfahren.

```
 10 rem Beispiel NLGS
 20 n=2:dim i(n)
 30 df=.1e-7
 40 dx=.1e-7
 50 i(1)=.5
 60 i(2)=4
 70 e1%=1:gosub 22800:rem Init
 80 gosub 22900:rem NLGS
 90 if e1%<>0 then print "keine loesung gefunden":print
    "fehlerflag=";e1%:goto 130
100 print :print "x(1)=";o(1);" x(2)=";o(2):print
110 print "df=";df;" dx=";dx;" e1%=";e1%
120 print :print "f=";f
130 end
```

Die Jacobi-Matrix des Systems muß explizit bekannt sein.
Globale Variable sind

n	Anzahl der Unbekannten bzw. Gleichungen
i(n)	Inputwerte für P 3.10 NLGS.PROG, d. h. Schätzungen für die Lösungen des Systems
df	Abbruchschranke für Funktionswertänderungen: $df = df_1 \cdot df_1 + df_2 \cdot df_2 + \ldots + df_n \cdot df_n$
dx	Abbruchschranke für Änderungen der Variablen: $dx = dx_1 \cdot dx_1 + dx_2 \cdot dx_2 + \ldots + dx_n \cdot dx_n$
f	Funktionswert $f = f_1 \cdot f_1 + f_2 \cdot f_2 + \ldots + f_n \cdot f_n$
o(n)	Outputwerte, d. h. von P 3.10 NLGS.PROG iterierte Lösungen des nichtlinearen Gleichungssystems.

Die Variable e1% ist das Fehlerflag:

e1%=0	P 3.10 NLGS.PROG ordnungsgemäß abgeschlossen
e1%=1	Wert vor Aufruf der Initroutine
e1%=2	Fehler während der Initialisierung
e1%=4	Newton-Verfahren bzw. Gradientenverfahren konvergieren nicht, bzw. geforderte Genauigkeit kann nicht erreicht werden
e1%=16	P 3.10 NLGS.PROG mit fehlerhafter Initialisierung aufgerufen
e1%=32	Initialisierung ordnungsgemäß abgeschlossen.

Nach wenigen Sekunden Rechenzeit wurden diese Ergebnisse vom Programm NLGS auf neun Stellen genau geliefert (5-Byte-Gleitkommaarithmetik).
Auch die im Abschnitt 3.2 mehrfach erwähnte transzendente Gleichung $f(x) = x - \tan(x) = 0$ kann mit dem Programm NLGS gelöst werden. Dabei sind recht grobe Anfangsnäherungen für $\bar{x}_0$ zulässig.
Der Modul P 3.09 NLGS.INIT (22800 bis 22864) überprüft die Vorgabewerte n, df und dx auf Zulässigkeit.
Außerdem werden die benötigten Felder dimensioniert und einige Initialisierungen vorgenommen.
Kernstück des Programms NLGS sind die Routinen P 3.10 NLGS.PROG (22900 bis 23364) und P 3.11 NLGS.HELP (22368 bis 23614).
In der Routine P 3.10 NLGS.PROG wird das durch (3.67) bis (3.70) gegebene Such-Weg-Verfahren realisiert. Außerdem ist das Newton-Verfahren für Systeme implementiert.
Die Routine NLGS.HELP (P 3.11) unterstützt diesen Vorgang durch die Ausführung benötigter Hilfsoperationen (Aufbau und Inversion der Jacobi-Matrix, Iteration verbesserter x_i). Das Programm NLGS wurde zur Lösung einer Vielzahl von Testaufgaben herangezogen. Dabei stellte es stets seine sehr guten Leistungsparameter unter Beweis.
Es kann deshalb selbst zur Lösung komplexer Probleme ($n \leqq 20$) herangezogen werden.

4. Gewöhnliche Differentialgleichungen

4.1. Grundlagen

Viele Prozesse in Natur und Technik können durch *Differentialgleichungen* beschrieben werden.
In diesen Gleichungen treten neben den unabhängigen Veränderlichen und einer oder mehreren Funktionen auch noch die Ableitungen dieser Funktionen auf.
Wenn die in den Differentialgleichungen auftretenden Funktionen nur von einer Variablen abhängen, spricht man von *gewöhnlichen Differentialgleichungen* (GDGln.), andernfalls von *partiellen Differentialgleichungen* (PDGln.).

4.1.1. Mathematische Formulierung der Problemstellung

In dynamischen Prozessen des Maschinenbaus, der Elektrotechnik und auch der Gerätetechnik ist die unabhängige Veränderliche in einer gewöhnlichen Differentialgleichung die Zeit t, so daß man schreiben kann:

$$F(t, \dot{x}, \ddot{x}, \dddot{x}, \ldots, x^{(n)}) = 0. \tag{4.1}$$

Ein Punkt bedeutet die einmalige Ableitung der unbekannten Funktion x nach der Zeit t.
Außerdem wollen wir vereinbaren, daß gilt

$$x^{(i)} = \frac{d^i x}{dt^i}. \tag{4.2}$$

Die durch (4.1) gegebene Differentialgleichung nennt man *n-ter Ordnung*, da n die höchste vorkommende Ableitung charakterisiert.
Ferner bezeichnet man (4.1) als eine Differentialgleichung in *impliziter Form*, da über die Funktion F eine Abhängigkeit zwischen der Zeit t, der gesuchten Funktion x(t) sowie deren zeitlichen Ableitungen hergestellt wird.
Ist F zusätzlich nichtlinear, was oftmals gegeben sein wird, so spricht man von einer *nichtlinearen Differentialgleichung*.
Treten Differentialgleichungen in der Form (4.1) auf, so bereitet die mathematisch-analytische bzw. numerische Behandlung Schwierigkeiten.
Deshalb strebt man eine Darstellung der GDGl. in *Normalform* an. Diese kann stets gefunden werden, falls die Funktion F eindeutig und nichtsingulär ist. Dabei ist es nicht unbedingt notwendig, daß die Gleichung (4.1) analytisch nach $x^{(n)}$ aufgelöst werden kann.
Im Fall der analytischen Auflösbarkeit nach $x^{(n)}$ kann man schreiben:

$$x^{(n)} = f(t, x, \dot{x}, \ldots, x^{(n-1)}), \tag{4.3}$$

und weiter mit

$$\begin{aligned} x &= x_1 \\ \dot{x} &= x_2 \\ \ddot{x} &= x_3 \\ &\vdots \\ x^{(n-1)} &= x_n \end{aligned} \tag{4.4}$$

gelangt man schließlich zur angestrebten Normalform:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ &\vdots \\ \dot{x}_n &= f\,(t, x_1, x_2, \ldots, x_{n-1}). \end{aligned} \tag{4.5}$$

Eine gewöhnliche Differentialgleichung n-ter Ordnung kann auf diese Weise in ein *System* von n verkoppelten Differentialgleichungen 1. Ordnung umgewandelt werden.
Alle im weiteren behandelten numerischen Lösungsverfahren für GDGln. basieren auf der Voraussetzung, daß die Normalform (4.5) der zu integrierenden GDGl. bekannt ist.
Ein Beispiel soll die Zusammenhänge verdeutlichen.
Betrachtet wird die gewöhnliche nichtlineare Differentialgleichung 4. Ordnung in impliziter Form:

$$(t^2 - 2x + 6\dot{x}\ddot{x} - 12\ddot{x} + x^{(4)}) = 0. \tag{4.6}$$

Die Auflösung nach der höchsten Ableitung ist einfach möglich, und man erhält

$$x^{(4)} = -t^2 + 2x - 6\dot{x}\ddot{x} + 12\ddot{x}. \tag{4.7}$$

Mit der Substitution (4.4) ergibt sich schließlich die Normalform zu

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ \dot{x}_3 &= x_4 \\ \dot{x}_4 &= -t^2 + 2x_1 - 6x_2x_3 + 12x_4. \end{aligned} \tag{4.8}$$

Die Notwendigkeit numerischer Lösungsverfahren für die bei gewöhnlichen Differentialgleichungen auftretenden Probleme erklärt sich daraus, daß mathematische Modelle selbst einfachster realer Vorgänge nicht elementar integrierbar sind. Man denke z. B. an ein so einfaches System wie ein mathematisches Pendel mit einem Ausschlag von mehr als 10°.

4.1.2. Anfangswertprobleme

Im folgenden soll das Differentialgleichungssystem (4.5) in der Form weiter verallgemeinert werden, daß n Funktionen f_i ; i = 1(1)n zulässig sind. Somit resultiert das folgende *Anfangswertproblem* [Enge 85]:

$$\begin{aligned} \dot{x}_i &= f_i\,(t, x_1, x_2, \ldots, x_n) \\ x_i\,(t = t_0) &= x_{i0}\ ;\ i = 1(1)n. \end{aligned} \tag{4.9}$$

Aus der unendlichen Anzahl möglicher Lösungen des Differentialgleichungssystems sollen genau diejenigen n Lösungsfunktionen f_i ausgewählt werden, die zum Anfangszeitpunkt $t = t_0$ die Werte x_{i0} aufweisen.
Für *Existenz* und *Eindeutigkeit* der Lösungen von (4.9) müssen die folgenden beiden Voraussetzungen erfüllt sein:

a) Die Funktionen f_i ; i = 1(1)n seien stetig in einem Gebiet G des durch $(t, x_1, x_2, \ldots, x_n)$ aufgespannten (n+1)-dimensionalen Raumes R^{n+1}.
b) Für die Funktionen f_i gelte mit einer beliebigen, aber fixen Lipschitz-Konstanten L:

$$|f_i\,(t, x_1, x_2, \ldots, x_n) - f_i\,(t, x_1^*, x_2^*, \ldots, x_n^*)| \leqq L \sum_{i=1}^{n} |x_i - x_i^*|\ ;\ i = 1(1)n. \tag{4.10}$$

Sind die Voraussetzungen a und b erfüllt, dann existiert in einem Gebiet $G^* \subset G$ des R^{n+1} genau eine Lösung des Anfangswertproblems (4.9), bestehend aus n Funktionen f_i ; i = 1(1)n mit $x_i(t_0) = x_{i0}$.
Die Voraussetzung b ist unter anderem dann erfüllt, wenn die f_i in G beschränkte partielle Ableitungen besitzen.
In diesem Fall existiert die Jacobi-Matrix $(\delta f_i/\delta x_j)$ und ist – unter bestimmten Voraussetzungen – auch nichtsingulär.
Dann gilt

$$L = \max |(\delta f_i/\delta x_j)| \; ; \; 1 \leqq i, j \leqq n$$
$$(t, x_1, x_2, \ldots, x_n) \in G.$$

Vor Beginn der Berechnungen muß man sich also vergewissern, ob die rechten Seiten der GDGln. und deren partielle Ableitungen im vorgegebenen Integrationsintervall Singularitäten aufweisen. Dies wäre z. B. der Fall, wenn Ausdrücke der Form 1/0 oder 0/0 im Integrationsintervall auftreten können. Der Rechner meldet dann nach einigen Rechenschritten unweigerlich einen OVERFLOW ERROR und bricht die Berechnungen ab.
Mit der im Abschnitt 4.1.1 angegebenen Substitution kann jedes Anfangswertproblem für eine GDGl. n-ter Ordnung auf ein System von n GDGln. 1. Ordnung zurückgeführt werden.
Die Transformation der Anfangsbedingungen ist in gleicher Weise möglich.
Dies wird in den Abschnitten 4.2.4 und 4.3.3 ausführlich erläutert.

4.1.3. Randwertprobleme

Von einem *Randwertproblem* wird gesprochen, wenn sowohl eine GDGl. n-ter Ordnung (bzw. das korrespondierende System von n GDGln. 1. Ordnung)

$$x^{(n)} = f(t, x, \dot{x}, \ldots, x^{(n-1)})$$

als auch n Gleichungen

$$B_j(x(t_0), \dot{x}(t_0), \ldots, x^{(n-1)}(t_0), \ldots, x(t_k), \dot{x}(t_k), \ldots, x^{(n-1)}(t_k), \ldots, x(t_l), \dot{x}(t_l), \ldots, x^{(n-1)}(t_l)) = 0 \qquad (4.11)$$

mit j = 1(1)n vorliegen.
Die Gleichungen (4.11) formulieren Bedingungen für die Lösungsfunktion x(t) sowie deren Ableitungen zu l Zeitpunkten t_k. Man spricht deshalb von einem l-Punkt-Randwertproblem. Ist l = 2 und sucht man die Lösung von (4.3) für alle $t \in [t_0, t_1]$, nennt man (4.3) und (4.11) ein *Zweipunktrandwertproblem* (ZPRWP oder TPBVP – Two Point Boundary Value Problem). Als Beispiele für im Bereich der Technik auftretende Zweipunktrandwertprobleme können Bewegungsaufgaben angeführt werden.
Dabei soll ein Objekt von einem Anfangspunkt in einer bestimmten Zeit in einen Endpunkt übergeführt werden. Zu Beginn und am Ende der Bewegung sind gewisse geforderte Werte für x(t) und $\dot{x}(t)$ zu realisieren.
Dieser Aufgabenstellung entsprechen die folgenden Gleichungen:

$$\dot{x}^{(n)} = f(t, x, \dot{x}, \ldots, x^{(n-1)})$$

$$B_j(x(t_0), \dot{x}(t_0), x(t_1), \dot{x}(t_1)) = 0 \qquad (4.12)$$

mit j = 1(1)n.
Aber selbst in dieser vereinfachten Form ist das ZPRWP (4.3), (4.12) noch hinreichend schwierig behandelbar, da keine Voraussetzungen bezüglich (4.3) angegeben worden sind.
Im allgemeinen Fall handelt es sich bei (4.3), (4.12) um ein nur numerisch lösbares nichtlineares ZPRWP.

Die Angabe von notwendigen und hinreichenden Bedingungen für die Existenz von Lösungen des Zweipunktrandwertproblems (4.3), (4.12) gestaltet sich recht kompliziert, so daß an dieser Stelle auf die Literatur [Kell68] [Enge85] verwiesen werden muß.
Für die durch die Problemformulierung (4.3), (4.12) besonders berücksichtigten technischen Belange kann man aber davon ausgehen, daß eine Lösung des Zweipunktrandwertproblems existiert, wenn (4.3) lösbar auf $[t_0, t_1]$ mit den durch (4.12) bedingten Anfangswerten ist. Außerdem ist es erforderlich, daß die Bestandteile des modellierten technischen Systems die durch die Lösungsfunktion x(t) sowie deren Ableitungen implizierten Forderungen (z. B. Brems- und Beschleunigungskräfte) erfüllen können.

4.1.4. Prinzipielle Vorgehensweise bei der numerischen Lösung gewöhnlicher Differentialgleichungen

Für fast alle praktisch relevanten GDGln. ist es nicht möglich, ein Integral in geschlossener Form zu finden. Deshalb geht man bei der numerischen Lösung wie folgt vor:

Das Integrationsintervall $[t_0, t_1]$ wird diskretisiert. Im Ergebnis findet man m – nicht notwendig äquidistante – Stützpunkte t^k, an denen dann Näherungswerte $x_k^*(t^k)$ für die gesuchten $x_k(t^k)$ durch eine geeignete Methode so bestimmt werden, daß gilt

$$x_k^*(t^k) \approx x_k(t^k) \approx x_k \; ; \; k = 1(1)m \tag{4.13}$$

$(t^0 = t_0, \; t^m = t_1)$.

Hat man einen Näherungswert x_k^* im Stützpunkt t^k bestimmt, so kann durch Integration die Lösung im nächsten Stützpunkt t^{k+1} bestimmt werden:

$$x_{k+1}(t^{k+1}) = \int_{t_k}^{t_{k+1}} f\,(t,x(t))dt \; ; \quad k = 0(1)m-1. \tag{4.14}$$

Hierbei wird von einer Darstellung des Differentialgleichungssystems in Normalform ausgegangen.
Man kann auch sagen, daß, ausgehend von den bekannten Anfangswerten $x_0(t_0)$, $\dot{x}_0(t_0)$, $\ddot{x}_0(t_0)$,..., $x^{(n-1)}(t_0)$ bzw. den $x_{i0}(t_0)$; i = 1(1)n, schrittweise eine Folge von Werten $x^*(t^k)$ konstruiert wird, die die gesuchte Lösung x(t) möglichst gut approximiert.
Wie das konkret zu erfolgen hat, soll am Beispiel der numerischen Lösung einer gewöhnlichen Differentialgleichung 1. Ordnung erläutert werden.
Da die prinzipielle Vorgehensweise für viele numerische Verfahren gleichartig ist, sollte der Leser die folgenden Seiten nicht überschlagen, sondern zu verstehen suchen.
Betrachten wir **Bild 4.1** (s. auch [Scra84] [Albr79] [Lamb73]).

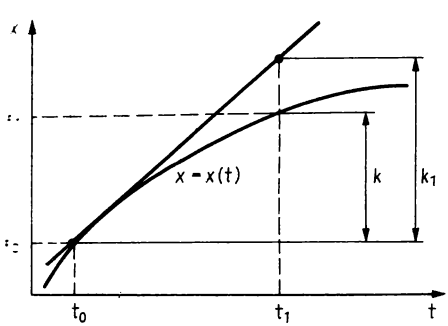


Bild 4.1. Vorgehensweise bei der numerischen Lösung einer GDGl.

Durch $x = x(t)$ ist die Lösung der GDGl. 1. Ordnung

$$\dot{x} = f(x, t) \tag{4.15}$$

mit der Anfangsbedingung $x(t_0) = x_0$ gegeben.
Gesucht ist die Lösung von (4.15) zum Zeitpunkt t_1, d. h., gesucht ist $x(t_1) = x_1$.
Da die Steigung von $x(t)$ im Punkt (t_0, x_0) über die GDGl. (4.15) bekannt ist, kann eine erste Näherung für die gesuchte Lösung $x(t_1) = x_1$ sofort angegeben werden:

$$x_1 \approx x_1^* = x_0 + (t_1 - t_0) \cdot f(t_0, x_0). \tag{4.16}$$

Mit $h = t_1 - t_0$ und

$$k_1 = h \cdot f(t_0, x_0) \tag{4.17}$$

kann man schreiben:

$$x_1 \approx x_1^* = x_0 + h \cdot k_1. \tag{4.18}$$

Aus Bild 4.1 ist ersichtlich, daß (4.18) die gesuchte Lösung x_1 um so besser approximiert, je kleiner h gewählt wird.
Es ist aber ebenso offensichtlich, daß von dieser einfachen Vorgehensweise keine allzu genauen Ergebnisse erwartet werden können.
Eine bessere Approximation von x_1 kann erhalten werden, wenn ein – vorläufig noch nicht näher bekannter – Zwischenpunkt im Intervall $[t_0, t_1]$ in die Betrachtungen einbezogen wird

$$k_2 = h \cdot f(t_0 + ah, x_0 + bk_1) \tag{4.19}$$

und dann k_1 und k_2 durch geeignete Gewichtsfaktoren G_1, G_2 so verknüpft werden, daß sich die gesuchte Größe k (Zuwachs von $x(t)$ im Intervall $[t_0, t_1]$) ergibt:

$$k = G_1k_1 + G_2k_2. \tag{4.20}$$

Aus Bild 4.1 kann man keinerlei Schlüsse über die Größen a, b, G_1, G_2 ziehen, so daß einige analytische Betrachtungen notwendig werden.
Ziel dieser Betrachtungen muß es sein, einen Zusammenhang zwischen der GDGl. (4.15), dem heuristisch formulierten Lösungsansatz (4.17) und (4.19) sowie der Gleichung (4.20) herzustellen, um so die unbekannten Größen a, b, G_1, G_2 bestimmen zu können. Problematisch hierbei wird die Auswertung der Funktion $f(t, x)$, über die nichts außer ihrer Stetigkeit und stetigen Differenzierbarkeit auf $[t_0, t_1]$ bekannt ist.
Die Funktion $f(t, x)$ soll deshalb durch ihre Taylor-Entwicklung um (t_0, x_0) ersetzt werden. Damit ergibt sich (4.19) bei Abbruch der Entwicklung nach dem ersten Glied zu

$$k_2 = h \cdot (f(t_0, x_0) + ah\,\delta f/\delta t + bk_1\,\delta f/\delta x + O(h^2)), \tag{4.21}$$

wobei $O(h^2)$ einen Fehler der Entwicklung in der Größenordnung von h^2 symbolisieren soll.
Außerdem erhält man durch einfaches Ausrechnen

$$k_2 = h \cdot f(t_0, x_0) + ah^2\,\delta f/\delta t + bh^2 f(t_0, x_0)\,\delta f/\delta x + O(h^3). \tag{4.22}$$

Damit ergibt sich k bei Vernachlässigung des Fehlerterms zu

$$k = G_1hf(t_0, x_0) + G_2hf(t_0, x_0) + G_2ah^2\,\delta f/\delta t + G_2bh^2 f(t_0, x_0)\,\delta f/\delta x, \tag{4.23}$$

und schließlich resultiert daraus

$$k = (G_1 + G_2)\,hf(t_0, x_0) + G_2ah^2\,\delta f/\delta t + G_2bh^2 f(t_0, x_0)\,\delta f/\delta x. \tag{4.24}$$

Betrachten wir nun die Entwicklung der Lösung von (4.15) um (t_0, x_0):

$$x_1 = x(t_0 + h) \approx x_0 + h\dot{x}_0 + \frac{1}{2}h^2\ddot{x}_0 + \ldots \tag{4.25}$$

Unter Beachtung der Tatsache, daß k der Zuwachs von x(t) im Intervall $[t_0, t_1]$ ist, kann man schreiben:

$$k = h\dot{x}_0 + \frac{1}{2}h^2\ddot{x}_0 + \ldots \tag{4.26}$$

Die Größe $\dot{x}_0$ ist bekannt:

$$\dot{x}_0 = f(t_0, x_0). \tag{4.27}$$

Somit muß nur noch $\ddot{x}_0$ bestimmt werden. Dazu ist (4.15) einmal total nach der Zeit zu differenzieren:

$$\ddot{x} = \frac{d}{dt}\dot{x} = \frac{d}{dt}f(t, x) = \frac{\delta f}{\delta t} \cdot \frac{dt}{dt} + \frac{\delta f}{\delta x} \cdot \frac{dx}{dt}. \tag{4.28}$$

Da die zweite Ableitung zur Zeit $t = t_0$ gesucht wird, kann man schreiben:

$$\ddot{x}_0 = \delta f/\delta t + \delta f/\delta x \cdot f(t_0, x_0). \tag{4.29}$$

Somit ist k angebbar zu

$$k = hf(t_0, x_0) + \frac{1}{2}h^2(\delta f/\delta t + \delta f/\delta x \cdot f(t_0, x_0)). \tag{4.30}$$

Vergleicht man nunmehr (4.24) und (4.30), so ergibt sich

$$\begin{aligned} G_1 + G_2 &= 1 \\ aG_2 &= 0.5 \\ bG_2 &= 0.5. \end{aligned} \tag{4.31}$$

Dies ist ein Gleichungssystem mit drei Gleichungen und vier Unbekannten. Es hat mithin unendlich viele Lösungen.
Wählt man z. B. $a = b = 1$, so erhält man $G_1 = G_2 = 0.5$. Es ergibt sich ein Runge-Kutta-Verfahren 2. Ordnung (genau bis einschließlich der Glieder 2. Ordnung, der Fehler des Verfahrens ist in der Größenordnung von h^3; s. auch Abschn. 4.1.5 und 4.1.6):

$$\begin{aligned} k_1 &= hf(t_0, x_0) \\ k_2 &= hf(t_0 + h, x_0 + k_1) \\ k &= \frac{1}{2}(k_1 + k_2) \\ x_1 &\approx x_1^* = x_0 + k + O(h^3). \end{aligned} \tag{4.32}$$

Wählt man $a = 0$, so erhält man das modifizierte *Euler-Verfahren*; mit $a = {}^2/_3$ ergibt sich das Verfahren von *Heun* [Albr79].

4.1.5. Einteilung der Verfahren

Die numerischen Verfahren zur Lösung des Anfangswertproblems (4.9) können prinzipiell wie folgt eingeteilt werden [Enge85] [Albr79]:

- *Einschrittverfahren*
- *Mehrschrittverfahren*
- *Extrapolationsverfahren.*

Einschrittverfahren verwenden zur Ermittlung von x_{k+1} nur den vorhergehenden Wert x_k.
Mehrschrittverfahren bestimmen den gesuchten Wert x_{k+1} mit Hilfe von $p+1$; $p>1$ vorangehenden Werten $x_{k-p}, x_{k-p+1}, \ldots, x_{k-1}, x_k$.
Extrapolationsverfahren sind eine Übertragung des Verfahrens von *Romberg* auf Differentialgleichungsprobleme.

Bei Ein- bzw. Mehrschrittverfahren unterscheidet man *implizite* und *explizite* Verfahren. Bei expliziten Verfahren tritt der zu bestimmende Wert x_{k+1} nicht in der Integrationsformel auf; bei impliziten Verfahren ist x_{k+1} in der Formel vorhanden. Aus diesem Grund ist bei impliziten Verfahren ein Iterationsalgorithmus zur Bestimmung von x_{k+1} notwendig.
Eine weitergehende Klassifikation der Verfahren zur numerischen Behandlung gewöhnlicher Differentialgleichungen findet der interessierte Leser in [Albr79].

4.1.6. Fehler in der numerischen Lösung

Bei der numerischen Lösung von GDGln. können zwei Fehlerarten auftreten:

- *Verfahrensfehler* (Abbruchfehler, truncation errors)
- *Rundungsfehler* (rounding errors).

Verfahrensfehler treten deshalb auf, weil die Taylor-Entwicklung von f(t,x) (s. Abschnitt 4.1.4) nach einer bestimmten Anzahl von Gliedern abgebrochen wird. Somit kann f(t,x) durch den Lösungsansatz nicht vollständig erfaßt werden.
Bei einer Entwicklung bis einschließlich der n-ten Ableitung sind die auftretenden Fehler in der Größenordnung von h^{n+1}; dies wurde im Abschnitt 4.1.5 durch $O(h^{n+1})$ symbolisiert.
Das ist aber nur der *lokale Verfahrensfehler,* der bei Integration von x_k bis x_{k+1} auftritt.
Der *globale Verfahrensfehler,* der alle durch das numerische Verfahren im Integrationsintervall bewirkten Fehler erfaßt, wird i. allg. größer sein.
In diesem Zusammenhang spricht man auch von der *Konvergenzordnung* q eines numerischen Verfahrens.
Dies ist das Pendant zur Fehlerordnung und somit das "Gütesiegel" eines Verfahrens.
Hat ein Verfahren lokale Fehler in der Größenordnung $O(h^m)$, so hat es die lokale Konvergenzordnung $q_l = m$.
Die Ermittlung der lokalen Fehlerordnung ist meist einfach möglich, weil sie aus der Herleitung des Verfahrens folgt. Schwieriger ist die Bestimmung der globalen Konvergenzordnung q_g.
Für die oft benutzten expliziten Einschrittverfahren (z. B. klassische Runge-Kutta-Verfahren) gilt

$$q_g = q_l - 1. \tag{4.33}$$

Allgemeine Richtlinie bei der Auswahl eines Verfahrens ist das Anstreben einer möglichst hohen globalen Konvergenzordnung q_g, da dann mit großen Schrittweiten h integriert werden kann (geringe Rechenzeit).
Dies gilt jedoch nur für „gutartige" GDGln. (s. auch Abschn. 4.1.8 und 4.1.9).
Der Verfahrensfehler kann verringert werden, indem h verkleinert wird. Er strebt dann mit h^{q_g} gegen Null. Dabei wächst jedoch die Anzahl der auszuführenden Rechenoperationen an, und der Rundungsfehler tritt stärker in Erscheinung.
Der Gesamtfehler als Summe aus Verfahrensfehler und Rundungsfehler kann also nicht beliebig klein gehalten werden. Die Schrittweite h sollte deshalb so gewählt werden, daß Verfahrensfehler und Rundungsfehler von der gleichen Größenordnung sind.
Für explizite Einschrittverfahren kann der globale Rechnungsfehler $r_{k,h}$ wie folgt angegeben werden [Enge85]:

$$r_{k,h} \begin{cases} \dfrac{r_{max}}{h}\,(x_k - x_0) \text{ für } C = 0 \\ \dfrac{r_{max}}{h}\,\exp(C(x_k - x_0) - 1) \text{ sonst.} \end{cases} \tag{4.34}$$

Dabei ist r_{max} der je Rechenschritt auftretende maximale Fehler. Er kann bei Kenntnis der Maschinenkonstante u ermittelt werden. Die Konstante C hängt vom verwendeten Verfahren ab.

Für das *Polygonzugverfahren von Euler-Cauchy*

$$x_1 \approx x_1^* = x_0 + hf(t_0,x_0) \tag{4.35}$$

ist C z. B.

$$C = \max_{(t,x)\in G} |\delta f/\delta x| \tag{4.36}$$

Für klassische *Runge-Kutta-Verfahren* (s. Abschn. 4.1.4) kann C wie folgt dargestellt werden:

$$C \sim \frac{p}{h} (p < 1). \tag{4.37}$$

Betrachten wir nun die einfache GDGl.

$$\dot{x} = -2x. \tag{4.38}$$

Sie soll für $t\in [0,1]$ unter der Anfangsbedingung $x(0) = 1$ mit dem Polygonzugverfahren von *Euler-Cauchy* numerisch integriert werden. Die exakte Lösung von (4.38) ist bekannt:

$$x(t) = \exp(-2t).$$

Gerechnet wird auf einer Maschine mit einer 5-Byte-Zahlendarstellung: 4 Byte für die Mantisse und 1 Byte für den Exponenten. Die Maschinenkonstante u ist somit angebbar zu

$$u = 2.3283064 \cdot 10^{-10}. \tag{4.39}$$

Das Programm soll so geschrieben sein, daß gilt

$$r_{max} = 13u. \tag{4.40}$$

Das Maximum von $|\delta f/\delta x|$ in G ist einfach ermittelbar:

$$C = 2, \tag{4.41}$$

und falls der globale Rechnungsfehler für das Integrationsintervall in Abhängigkeit von h interessiert, kann man schreiben:

$$r = \frac{13u}{h} (e^2-1)$$

$$r = 1.9338385 \cdot 10^{-8}/h. \tag{4.42}$$

In diesem speziellen Fall wird man mit $h \approx 0.1$ rechnen, so daß ein globaler Rechenfehler von etwa 10^{-9} zu erwarten ist. Dies stimmt recht gut mit praktischen Ergebnissen überein. Man kann erkennen, daß

$$r_h \sim \frac{1}{h} \tag{4.43}$$

gilt. Bei $h \to 0$ wächst somit der Rechenfehler über alle Grenzen.
Verfahrensfehler und Rechenfehler verhalten sich gegenläufig. Die Ermittlung einer günstigen Schrittweite h in Abhängigkeit von der globalen Konvergenzordnung sowie der Maschinenkonstante u stellt demnach eine nichttriviale Optimierungsaufgabe dar. Diese ist durch die *Implementation* eines Verfahrens (Anpassung an die jeweilige Soft- und Hardwareumgebung) möglichst gut zu lösen.

4.1.7. Optimale Implementation numerischer Verfahren

4.1.7.1. Schrittweitensteuerung durch Richardson-Extrapolation

Wie bereits dargelegt, sind numerische Verfahren zur Lösung von GDGln. durch Verfahrens- und Rundungsfehler charakterisiert.
Eine Schrittweitensteuerung hat die Aufgabe, diese beiden Fehler in vorgegebenen Grenzen zu halten und dabei dem Verfahren einen möglichst geringen Arbeitsaufwand – gemessen an der Zahl der Rechenschritte – aufzuerlegen. Basis hierfür ist die Kenntnis von Fehlerschätzungsformeln für die numerischen Verfahren.
Diese Fehlerschätzungsformeln gehen davon aus, daß eine Integration der GDGln. im Intervall $[t_0,t_1]$ mit zwei unterschiedlichen Schrittweiten h_1 und h_2 (i. allg. gilt $h_1 = ph_2$, $p > 1$) vorgenommen wird. Dann kann der globale Verfahrensfehler für die Schrittweite h_1 und ein Verfahren der globalen Konvergenzordnung q_g näherungsweise angegeben werden [Albr79] [Enge85] zu

$$e^k_{h_1} = x(t^k) - x^*_{h_1}(t^k) \approx \frac{1}{\left(\frac{h_2}{h_1}\right)^{q_g} - 1}\left(x^*_{h_1}(t^k) - x^*_{h_2}(t^k)\right). \tag{4.44}$$

Einen verbesserten Wert für $x(t^k)$ kann man wie folgt aufschreiben:

$$x^*_{h_1}(t^k) = x^*_{h_1}(t^k) + e^k_{h_1} \tag{4.45}$$

$$= \frac{1}{\left(\frac{h_2}{h_1}\right)^{q_g} - 1}\left[\left(\frac{h_2}{h_1}\right)^{q_g} x^*_{h_1}(t^k) - x^*_{h_2}(t^k)\right]$$

Diese Vorgehensweise bezeichnet man als *Richardson-Extrapolation,* und es gilt

$$x(t^k) = x^*_{h_1} + O(h_1^{q_g+1}). \tag{4.46}$$

Die globale Konvergenzordnung des Verfahrens ist von q_g auf $q_g + 1$ erhöht worden. In bestimmten Fällen kann auch eine Erhöhung auf $q_g + 2$ eintreten.
Dies ist z. B. bei dem im Abschnitt 4.2.2 behandelten *Trapezverfahren* der Fall.
Mit Gleichung (4.44) kann somit der Verfahrensfehler näherungsweise ermittelt und zur geeigneten Steuerung der Schrittweite h herangezogen werden.
Hierbei ist allerdings der Rundungsfehler noch nicht erfaßt. Da bei vielen Verfahren nur genäherte Aussagen über den Rundungsfehler gegeben werden können, beginnt an dieser Stelle die „Kunst" bei der Implementation numerischer Verfahren.
Denn nunmehr ist die konkrete Formel, die die Änderung von h in Abhängigkeit von (4.44) realisiert, ausschlaggebend für die Effizienz des Verfahrens.
Für die in diesem Abschnitt behandelten Verfahren wurde die Formel für die Änderung von h sowie die in dieser Formel vorhandenen Parameter durch eine lange Serie von Testrechnungen auf höchstmögliche Leistungsfähigkeit ausgelegt. Basis dafür war eine Maschine mit einer 5-Byte-Zahlendarstellung.
In den entsprechenden Abschnitten wird auf diese Parameter hingewiesen, so daß sie gemäß den Wünschen des Nutzers der Programme variiert werden können.

4.1.7.2. Schrittweitensteuerung durch eingebettete Verfahren

Ähnlich wie bei der Richardson-Extrapolation werden wiederum zwei Werte $x^*(t^k)$ und $\tilde{x}^*(t^k)$ berechnet.
Allerdings bleibt bei dieser Vorgehensweise die Schrittweite h konstant.

Die erste Näherung x* wird mit einem Einschrittverfahren der Konvergenzordnung $q_g = m$ berechnet, während $\tilde{x}^*$ mit einem Verfahren der Ordnung $q_g = m + 1$ ermittelt wird. Dann gilt

$$x(t^k) \approx x^*(t^k) = \tilde{x}^*(t^k) + O(h^{m+2}) \tag{4.47}$$

$$e_h^k \approx \frac{1}{h}(\tilde{x}^* - x^*). \tag{4.48}$$

Diese Methode ist besonders dann weniger aufwendig als die Richardson-Extrapolation, wenn die beiden benutzten Verfahren explizite Einschrittverfahren sind und bestimmte Koeffizienten des Verfahrens der Konvergenzordnung m sämtlich im Verfahren der Ordnung m + 1 enthalten („eingebettet") sind. Dies ist bei vielen der sog. *Runge-Kutta-Fehlberg-Verfahren* der Fall [Fehl69] [Fehl70] [Albr79].
Zwei dieser *RKF-Verfahren* werden im Abschnitt 4.3.3 vorgestellt.
Es soll an dieser Stelle betont werden, daß die Aussagen des Abschnitts 4.1.7.1 über die Erfassung der Rundungsfehler auch hier wieder zutreffend sind.

4.1.8. Stabilitätsprobleme

Instabilitäten bei der Lösung einer GDGl. können sowohl durch die GDGl. selbst als auch durch das zur Lösung benutzte numerische Verfahren hervorgerufen werden. Untersuchen wir zunächst die *Stabilität des Anfangswertproblems*

$$\dot{x} = f(t, x)$$
$$x(t_0) = x_0,\ t \in [t_0, \infty). \tag{4.49}$$

Die Funktion f(t, x) sei stetig und mindestens einmal stetig differenzierbar ([Albr79]).
Die Untersuchungen werden vereinfacht, wenn ein „gestörtes Problem" definiert wird, bei dem eine kleine Perturbation s_0 in der Anfangsbedingung auftritt:

$$\bar{\dot{x}} = f(t, x)$$
$$\bar{x}(t_0) = x_0 + s_0,\ t \in [t_0, \infty). \tag{4.50}$$

Für den Fehler $e(t) = \bar{x}(t) - x(t)$ kann man die folgende GDGl. 1. Ordnung aufschreiben:

$$\dot{e} = f(t, x(t) + e(t)) - f(t, x(t))$$
$$e(t_0) = s_0. \tag{4.51}$$

Das Problem (4.49) heißt stabil, wenn zu jedem $\varepsilon > 0$ eine positive Zahl s derart existiert, daß gilt

$$|e(t, s_0, t_0)| < \varepsilon;\ t \in [t_0, \infty),\ |s_0| < s. \tag{4.52}$$

Andernfalls heißt das Problem (4.49) instabil.
Betrachten wir folgendes Beispiel:

$$\dot{x} = \lambda x$$
$$x(0) = x_0\ (\neq 0). \tag{4.53}$$

Es folgt sofort:

$$x(t) = x_0 \exp(\lambda t) \tag{4.54}$$
$$\dot{e} = \lambda e \text{ mit}$$
$$e(0) = s_0. \tag{4.55}$$

Damit ergibt sich $e(t, s_0, 0)$ zu

$$e(t, s_0, 0) = s_0 \exp(\lambda t). \tag{4.56}$$

Selbst bei einer noch so kleinen Störung s_0 gilt für $\lambda \geqq 0$ ab einem gewissen Zeitpunkt t^*:

$$e(t^*, s_0, 0) > \varepsilon. \tag{4.57}$$

Somit ist (4.53) für $\lambda \geqq 0$ instabil, für $\lambda < 0$ dagegen stabil. Die Instabilität muß bei hinreichend kleinen Integrationsintervallen $[t_0, t_1]$ nicht notwendig zutage treten. Bei großen Intervallen tritt sie jedoch derart in Erscheinung, daß sich der Rechner mit einem OVERFLOW ERROR meldet. Er weist damit nachdrücklich darauf hin, daß der Nutzer sich vor Eintippen von RUN von der prinzipiellen Lösbarkeit des Problems mit Hilfe eines numerischen Verfahrens überzeugt haben sollte. Gegenwärtig ist noch kein numerisches Verfahren bekannt, das instabile Probleme im Bereich einer stark anwachsenden Lösungsfunktion integrieren könnte.
Die hier vorgenommenen Betrachtungen gelten sinngemäß auch für Probleme höherer Ordnung. In diesem Fall müssen alle $e_i(t, s_{0i}, x_i)$ dem Kriterium der eben angegebenen Definition genügen, um die Stabilität des betrachteten Problems zu gewährleisten.
Der andere Teil der Stabilitätsproblematik ist die Frage nach der Stabilität eines bestimmten numerischen Verfahrens. Von einem solchen Verfahren muß zumindest gefordert werden, daß es den prinzipiellen Lösungsverlauf richtig widerspiegelt. Der globale Rechnungsfehler als Summe aus Verfahrens- und Rundungsfehler darf sich also nicht derart akkumulieren, daß das Verhalten der numerischen Lösung von dem der wirklichen Lösung signifikant abweicht.
Das wäre besonders dann unangenehm, wenn die wirkliche Lösung nicht bekannt ist.
Die Definition der Stabilität numerischer Verfahren ist allerdings so kompliziert, daß sie an dieser Stelle nicht ausführlich besprochen werden kann.
Der interessierte Leser muß daher auf die Literatur [Albr79] [Lamb73] verwiesen werden.
Wegen der Wichtigkeit der Problematik soll aber zumindest die prinzipielle Vorgehensweise demonstriert werden.
Als Beispiel wählen wir das im Abschnitt 4.1.4 behandelte Runge-Kutta-Verfahren 2. Ordnung:

$$\begin{aligned} k_1 &= hf(t_0, x_0) \\ k_2 &= hf(t_0 + h, x_0 + k_1) \\ k &= \frac{1}{2}(k_1 + k_2) \\ x_1 &\approx x_1^* = x_0 + k \end{aligned} \tag{4.58}$$

sowie die eben betrachtete GDGl.

$$\begin{aligned} \dot{x} &= \lambda x \\ x(0) &= x_0 (\neq 0) \\ t &\in [t_0, t_1];\ t_0 < t_1. \end{aligned}$$

Vorausgesetzt wird außerdem, daß $\lambda < 0$ ist. Damit wird die Stabilität des Problems (4.53) gewährleistet.
Es soll nun untersucht werden, unter welchen Bedingungen die numerische Lösung das Verhalten der monoton fallenden Lösung

$$x(t) = x_0 \exp(\lambda t) \quad (\lambda < 0)$$

widerspiegelt.
Dazu soll die durch das Verfahren approximierte Lösung x_1 als Funktion von h und λ aufgeschrieben werden:

$$k_1 = h\lambda x_0$$
$$k_2 = h\lambda(x_0 + h\lambda x_0)$$
$$k = \frac{1}{2}(k_1 + k_2) = h\lambda x_0 + \frac{1}{2}h^2\lambda^2 x_0 \tag{4.59}$$
$$x_1 = x_0(1 + h\lambda + \frac{1}{2}h^2\lambda^2).$$

Ausschlaggebend für das Verhalten des Verfahrens ist offensichtlich

$$g(h, \lambda) = (1 + \lambda h + \frac{1}{2}h^2\lambda^2). \tag{4.60}$$

Mit $h > 0$, $\lambda < 0$ ist eine abklingende numerische Lösung x_1, d. h. $g(h,\lambda) < 1$, nur zu erreichen, falls gilt

$$-2 < h\lambda < 0. \tag{4.61}$$

Gleichung (4.61) definiert den *reellen Stabilitätsbereich* eines Runge-Kutta-Verfahrens 2. Ordnung. Eine Schrittweitensteuerung hat stets dafür Sorge zu tragen, daß ein numerisches Verfahren innerhalb seines Stabilitätsbereichs betrieben wird.
Es ist zu beachten, daß die im Beispiel vorgenommene Abschätzung nur für lineare GDGln. gilt. Im allgemeinen Fall ist die Schrittweitensteuerung eines Verfahrens durch Testrechnungen bezüglich ihres Stabilitätsverhaltens zu optimieren.

4.1.9. Steifheit von Differentialgleichungen

Ein Differentialgleichungsproblem wird als *steif* bezeichnet, wenn es sowohl schnell veränderliche Lösungsanteile enthält als auch solche, die nur langsam veränderlich sind. Dabei muß aus der konkreten Lösung des Problems die schnelle Veränderbarkeit nicht immer ersichtlich sein.
Als Beispiel für das Auftreten steifer GDGln. kann die Simulation des Verhaltens von Industrierobotern genannt werden. Industrieroboter weisen als elektromechanische Systeme sowohl große mechanische Zeitkonstanten (s) als auch kleine elektronische Zeitkonstanten (ms, µs) auf.
Betrachten wir zur weiteren Verdeutlichung des Sachverhalts die einfache, analytisch lösbare GDGl.

$$\ddot{x} + a\dot{x} + bx = 0 \tag{4.62}$$

mit $a, b > 0$, $a/b \gg 1$ und $x(0) = x_0$, $\dot{x}(0) = \dot{x}_0(x_0, \dot{x}_0 \neq 0)$.
Die allgemeine Lösung dieser stabilen linearen homogenen GDGl. 2. Ordnung ist mit

$$\lambda_{1/2} = -\frac{a}{2} \pm \sqrt{\left(\frac{a}{2}\right)^2 - b} \tag{4.63}$$

angebbar zu

$$x(t) = \frac{\lambda_2 x_0 - \dot{x}_0}{\lambda_2 - \lambda_1}\exp(\lambda_1 t) + \frac{\dot{x}_0 - \lambda_1 x_0}{\lambda_2 - \lambda_1}\exp(\lambda_2 t). \tag{4.64}$$

Es sei darauf hingewiesen, daß wegen $a, b > 0$ und $a/b \gg 1$ gilt

$$\lambda_1, \lambda_2 < 0 \text{ bzw. } |\lambda_1| \ll |\lambda_2|.$$

Wählen wir nun $\dot{x}_0 = \lambda_1 x_0$, ergibt sich als exakte Lösung von (4.62)

$$x(t) = x_0\exp(\lambda_1 t). \tag{4.65}$$

Vom zweiten Lösungsanteil $\exp(\lambda_2 t)$ in (4.64), der wegen $|\lambda_1| \ll |\lambda_2|$ wesentlich schneller als der Anteil mit $\exp(\lambda_1 t)$ abklingt, ist nichts mehr zu erkennen.
Versucht man nun, die GDGl. (4.62) mit dem bereits mehrfach erwähnten Runge-Kutta-Verfahren 2. Ordnung numerisch zu integrieren, und wählt eine Schrittweite h gemäß der Stabilitätsbedingung

$$-2 < h_1\lambda_1 < 0,$$

so wird sich der Rechner nach einiger Zeit mit einem OVERFLOW ERROR melden.
Ein Zahlenbeispiel möge dies noch weiter verdeutlichen. Mit

$$a = 1000$$
$$b = 999$$
$$x_0 = 1$$

erhält man

$$\lambda_1 = -1$$
$$\lambda_2 = -999$$
$$\dot{x}_0 = \lambda_1 x_0 = -1.$$

Die allgemeine Lösung (4.64) enthält somit Lösungsanteile $\exp(-t)$ und $\exp(-999t)$, so daß für die Stabilität des benutzten numerischen Verfahrens zu fordern ist:

$$-2 < h_1\lambda_1 < 0 \text{ und } -2 < h_2\lambda_2 < 0;\ h = \min(h_1, h_2).$$

Die Stabilität ist nur gewährleistet, wenn $h = h_2 \approx 0.002$ gesetzt wird. Die Schrittweite h_1 ergibt sich etwa tausendfach größer, was mit $h = h_1$ Instabilität bewirken würde. Wenn mit expliziten Einschrittverfahren steife GDGln. integriert werden sollen, dann muß man sehr kleine Schrittweiten h wählen, um im Stabilitätsbereich des Verfahrens zu bleiben.
Das bedingt bei der Abarbeitung eines vorgegebenen Integrationsintervalls die Ausführung von sehr vielen Integrationsschritten.
Damit wächst der Rundungsfehler stark an, und es kann zum Abweichen der numerischen Lösung von der tatsächlichen Lösung kommen.
Man sagt dann, daß das verwendete Verfahren nicht *steifstabil* ist.
Zur Behandlung steifer Probleme wurden besondere Verfahren entwickelt, die einen großen Stabilitätsbereich aufweisen, aber dafür auch einen großen Rechenaufwand erfordern. Der interessierte Leser wird auf das Buch von *Gear* [Gear71] verwiesen, in dem Verfahren und Programme zur Lösung steifer Probleme angegeben sind.
Im Abschnitt 4.2.2 wird ein einfaches Programm zur Lösung von steifen Problemen angeboten.
Es handelt sich hierbei um das einfachste steifstabile Verfahren mit einer relativ niedrigen globalen Konvergenzordnung.
Es sollte nur für die Bearbeitung steifer Probleme verwendet werden, da es bei nichtsteifen Problemen wegen der niedrigen Konvergenzordnung relativ lange Rechenzeiten erfordert.

4.1.10. Unstetige rechte Seiten

Bei der Beschreibung bestimmter realer Prozesse werden die rechten Seiten der Modellgleichungen *unstetig*.
Als Beispiel für diesen Sachverhalt kann die mathematische Modellierung von Prozessen mit harten Beschränkungen oder die Erfassung der Kopplung digitaler und analoger Systeme angeführt werden.

Zur numerischen Integration müssen die unstetigen rechten Seiten der GDGln. in stückweise stetige rechte Seiten umgewandelt werden.
Entsprechend den für die Umschaltung relevanten Systemgrößen sind während der Integration die entsprechenden stetigen rechten Seiten als aktuelle rechte Seiten zu definieren. Dabei können natürlich Sprünge in den $\dot{x}_i$ auftreten.
Es hat sich gezeigt, daß vor allem Mehrschrittverfahren in diesem Fall versagen können. Einschrittverfahren sind zur Behandlung solcher Probleme wesentlich besser geeignet, falls man dafür Sorge trägt, daß die Sprünge in den $\dot{x}_i$ nur zu Beginn eines Integrationsintervalls wirksam werden können.
Da sie außerdem gegenüber den Mehrschrittverfahren einen wesentlich geringeren Implementationsaufwand erfordern und zudem einen größeren Stabilitätsbereich aufweisen, sind sie für die Anwendung auf Mikrorechnern vorzuziehen.

4.2. Übersicht zum Programmsystem

Eine Übersicht der zum Lösen von GDGln. erarbeiteten Software führt **Tafel 4.1** auf.
Das Programmsystem umfaßt fünf Verfahren zur Lösung von Anfangswertproblemen sowie ein Verfahren zur Behandlung von nichtlinearen Zweipunktrandwertproblemen der Form

Tafel 4.1. Software zur Lösung von Differentialgleichungsproblemen

Modul	Programm	Init	Step	Bemerkungen
TRA4	P 4.03 P 4.04 P 4.05	28 000	28 066	Trapezverfahren mit Richardson-Extrapolation; globale Fehlerordnung 4; Schrittweitensteuerung; 1841 Byte (28 068 ... 28 218); 2514 Byte komplett
RKV5	P 4.06 P 4.07 P 4.08	28 300	28 366	Runge-Kutta-Verfahren 4. Ordnung mit Richardson-Extrapolation; Schrittweitensteuerung; globale Fehlerordnung 5; 1744 Byte (28 364 ... 28 500); 2417 Byte komplett
RKF5	P 4.09 P 4.10	28 600	28 642	Runge-Kutta-Fehlberg-Verfahren mit Schrittweitensteuerung (eingebettetes Verfahren); globale Fehlerordnung 5; 2249 Byte (28 640 ... 28 828); 2922 Byte komplett
RKF6	P 4.11 P 4.12 P 4.13	28 900	28 976	Runge-Kutta-Fehlberg-Verfahren (8 Stützstellen); Schrittweitensteuerung (eingeb. Verf.); globale Fehlerordnung 6; 2872 Byte (28 974 ... 29 196); 3545 Byte komplett
EXGS	P 4.14 P 4.15 P 4.16	29 200	29 256	Extrapolation nach *Gragg-Stoer* mit Fehlertest; globale Fehlerordnung nach Tableaugröße der Richardson-Extrapolation; 2156 Byte (29 254 ... 29 432); 2829 Byte komplett
BVPS	P 4.20 P 4.21 P 4.22	29 750	29 806	Einfachschießverfahren zur Lösung von Zweipunktrandwertproblemen; muß mit RKV5 zusammenarbeiten; 2289 Byte (29 804 ... 29 992); 4458 Byte komplett

(4.3), (4.12). Die Programme bestehen jeweils aus einer Initialisierungsroutine sowie den eigentlichen Rechenroutinen. Die Anfangs- und Endzeilennummern der Module sind in Tafel 4.1 angegeben.
In der Spalte für Bemerkungen ist der Speicherplatzbedarf der Module in Byte angegeben. Die Zahl in Klammern gibt den Speicherbedarf für ein lauffähiges Applikationsprogramm unter Zugrundelegung des im Abschnitt 4.3.5 behandelten Testbeispiels an.
Diese Werte gelten für einen Rechner, dessen BASIC-Interpreter die Programmzeilen sofort bei Eingabe auf Schlüsselwörter (**print, gosub** usw.) untersucht und diese in Tokens (Zeichen außerhalb des ASCII-Alphabets) umwandelt. Wird vom benutzten Rechner reiner ASCII-Quelltext abgespeichert, muß mit einem höheren Speicherbedarf gerechnet werden.
Tafel 4.2 zeigt die in den Programmen verwendeten lokalen Variablen und Felder.

Tafel 4.2. Lokale Variable und Felder

Modul	Lokale Variable und Felder	
TRA4	Variable Felder y7 u	c7, h7, h8, h9, **i7**, m7, r7, s8, s9, t7, t8, y7, z7, u d( ), h( ), i( ), j( ), l( ), m( ), n( ) Anzahl der Fuktionsauswertungen Maschinenkonstante (Laufvariable halbfett)
RKV5	Variable Felder	h7, h8, h9, **i7**, m7, r7, s7, s8, s9, t7, t8, y7, z7, u d( ), h( ), i( ), j( ), k( ), l( )
RKF5	Variable Felder	d7, h7, h8, **i7**, **j7**, k7, m7, r7, s7, s8, t8, y7, u l( ), n( )
RKF6	Variable Felder	a7, c7, c8, e7, h7, h8, **i7**, **j7**, **k7**, m7, r7, r8, s7, s8, s9, t7, y7, u p( ), q( ), r( ), s( ), t( ), u( ), v( ), w( ), y( ), z( )
EXGS	Variable Felder	h7, h8, **i7**, **j7**, **k7**, m7, p7, r7, t7, v7, y7, z7, u r( ), s( ), t( ), u( ), v( ), w( ), z( )
BVPS (ohne RKV5)	Variable Felder	i9, **j7**, **k7**, **l7**, n7, n9, **p7**, **q7**, r8 a( ), b( ), c( ), m( ), o( ), r( ), t( ), u( ), y( )

In allen Verfahren zur Lösung von Anfangswertproblemen wird die lokale Variable u zur Bezeichnung der Maschinenkonstante verwendet.
Die lokale Variable y7 dient in allen Programmen zur Speicherung der Anzahl der Auswertungen der rechten Seiten. In bestimmten Fällen kann es sinnvoll sein, auf diese Variable bei der Steuerung des Programmablaufs zurückzugreifen.
Nach Ablauf der Initialisierung kann durch die Anweisung

```
print "u="; u
```

die Ausgabe der Maschinenkonstante u veranlaßt werden.
Tafel 4.3 gibt alle für die Initialisierung und die Arbeit der vorgestellten Software wichtigen globalen Variablen sowie das global zu vereinbarende Ergebnisfeld x( ) an. Die in Spalte 3 der Tafel mit einem x gekennzeichneten Variablen müssen vor dem Aufruf der Initialisierungsroutine vereinbart und mit Anfangswerten versehen worden sein. Anderenfalls kann es zu Fehlerzuständen kommen.
Eine Sonderstellung nimmt das Feld x() ein. Es muß vor Aufruf der Initialisierungsroutine dimensioniert werden: **dim** x(n), wobei n die Ordnung des Differentialgleichungssystems angibt.

Die nicht notwendig zu initialisierenden Variablen werden mit den in Spalte 4 angegebenen Standardwerten versehen. Die Arbeit der Routinen kann jederzeit durch eine Programmanweisung nach einem **return** oder durch die BREAK-Taste unterbrochen werden, um z. B. die gerade aktuelle Zeit t oder die Integrationsergebnisse x() auszugeben.

Tafel 4. 3. Globale Variable und Felder

Variable	Bedeutung	(x)	Standardwert	Bemerkungen
e1 %	Fehlerflag; s. Tafel 4. 4.	x	–	
f1	Relativer Fehler des Ergebnisses	(x)	(0)	f1 oder f2 müssen gesetzt sein
f2	Absoluter Fehler des Ergebnisses	(x)	(0)	f1 oder f2 müssen gesetzt sein
h	Integrationsschrittweite	–	h8	h kann sinnvoll vorgegeben werden
h1	Minimal zulässige Integrationsschrittweite	–	h8	wird im Init ermittelt: h1 > h; falls h1 > h8, wird h1 = h8 gesetzt
m1	Zulässige Anzahl von Auswertungen der rechten Seiten	–	1000	dient zur Vermeidung des "Festfahrens" der Routine; kann definiert werden
n	Ordnung des Differentialgleichungssystems	x	– (12 ist Maximum)	**dim** x(n); die rechten Seiten sind ab Zeile 30 000 einzugeben
t	Anfangszeit, aktuelle Zeit	x	–	t ≠ t1 vor Initroutine: bei RKF5 ist t ≠ t1 nach Rückkehr (t = t1 – h)
t1	Endzeit bzw. Endpunkt der Integration	x	–	s. Bemerkung zu t
x()	Feld der abhängigen Variablen Anfangswerte/Integrationsergebnis	x	0	Die Anfangswerte müssen vor Aufruf der Integrationsroutine gesetzt sein

Ein x in Spalte 3 bedeutet, daß die entsprechende globale Variable vor (!) Aufruf der Init-Routine auf ihren aktuellen Wert gesetzt werden muß.

Bei Unterbrechung der Programmabarbeitung durch die BREAK-Taste kann mit dem Kommando CONT die Abarbeitung des Programms fortgesetzt werden, falls beim Abfragen aktueller Variablenwerte kein Fehler aufgetreten ist.
Dieser würde zu einem CAN'T CONTINUE ERROR führen.
Für diesen Fall notiert man sich die nach dem Betätigen der BREAK-Taste angegebene Zeilennummer: BREAK IN XXXXX, um mit **goto** XXXXX den Rechner doch noch zur Wiederaufnahme der Arbeit zu veranlassen. RUN XXXXX ist in diesem Fall zu vermeiden, da sämtliche Zwischenergebnisse gelöscht werden.
Die globale Variable e1% ist das Fehlerflag des Programmsystems.

Die Bedeutung der Werte von e1% zeigt **Tafel 4.4.** Es wurden Zweierpotenzen zur Signalisierung der einzelnen Fehlerzustände verwendet, um durch **or**-Verknüpfungen mehrere Fehler gleichzeitig anzeigen zu können. So würde e1%=76 darauf hindeuten, daß

- das Integrationsende t1 mit m1 Auswertungen der rechten Seiten nicht erreicht wurde
- Rundungsfehler verstärkt im Ergebnis auftreten können
- der Steifheitstest angesprochen hat.

Tafel 4. 4. Bedeutung der Fehlervariablen e1 %

e1 %	Bedeutung
0	Während der Integration sind keine Fehler aufgetreten; alles in Ordnung
1	Alle Vorbereitungen für den Aufruf der Initialisierungsroutine sind erfolgt; vor Aufruf der Initialisierungsroutine muß e1 % = 1 gesetzt werden, sonst Rückkehr mit e1 % = 2
2	Fehler in der Initialisierungsroutine: n > 12, n ≦ 0, f1 = f2 = 0, t1 − t kleiner als durch Maschinengenauigkeit bedingte zulässige minimale Schrittweite h1; e1 % ≠ 1 vor Aufruf der Routine
4	Integrationsende mit m1 Funktionsauswertungen noch nicht erreicht; e1 % = 32 setzen und Routine nochmals aufrufen
8	Es wurde mindestens einmal mit der minimal zulässigen Schrittweite h1 gerechnet; Rundungsfehler können somit auftreten und das Integrationsergebnis verfälschen
16	Die Integrationsroutine wurde nach Rückkehr aus einer fehlerhaften Initialisierungssequenz aufgerufen; es erfolgt deshalb keine Aktion
32	Die Initialisierungsroutine wurde ordnungsgemäß beendet und der Aufruf der eigentlichen Integrationsroutine kann erfolgen
64	Der Steifheitstest nach *Shampine* ist positiv (nur bei RKF6); ein Verfahren mit einer kleineren globalen Konvergenzordnung sollte zur Bearbeitung des Problems eingesetzt werden

Die rechten Seiten der zu behandelnden GDGln. werden ab Zeile 30000 eingegeben. Dies zeigt **Programm P 4.01GDGL.EQNS.** Für eine rechte Seite sind jeweils 99 Programmzeilen vorgesehen; auf der hundertsten Programmzeile steht das abschließende **return**.
Die aktuellen Werte der rechten Seiten werden jeweils auf die Variable r7 gespeichert und an die aufrufende Routine zurückgegeben.
Zur Programmierung der rechten Seiten sind beliebige numerische Konstanten sowie alle anderweitig noch nicht benutzten Variablen (s. Taf. 4.2 und 4.3) zulässig.
Das Feld x() enthält die Integrationsergebnisse $x_j(t)$ und ist für diesen Zweck in allen Programmen reserviert. Die Startzeilennummer sznr für die Eingabe der rechten Seite j kann wie folgt ermittelt werden:

sznr = 30000 + (j − 1)*100
j = 1(1)n (n ≦ 12).

Das System ist zur Verarbeitung von maximal zwölf GDGln. 1. Ordnung ausgelegt.
Sollen Systeme größerer Ordnung behandelt werden, so ist dies ohne Probleme möglich, wenn die in den Routinen vorhandenen Programmzeilen

. . . **on** i7 **gosub** 30000, 30100, . . .

erweitert und die entsprechenden rechten Seiten zusätzlich eingefügt werden.
Noch eine Bemerkung zur zeitlichen Effizienz des Programmsystems.

Programm P 4.01. GDGL.EQNS. Programmteil zur Eingabe der rechten Seiten der zu integrierenden gewöhnlichen Differentialgleichungen.

```
29997 rem !-----------------------------------------------!
29998 rem !            Rechte Seiten der gDGLn            !
29999 rem !-----------------------------------------------!
30000   r7=x(2)
30098   y7=y7+1                                          ! Zählen d. Funk-
30099 return                                             ! tionsaufrufe
30100   r7=-x(1)
30199 return
30200   r7=0
30299 return
30300   r7=0
30399 return
30400   r7=0
30499 return
30500   r7=0
30599 return
30600   r7=0
30699 return
30700   r7=0
30799 return
30800   r7=0
30899 return
30900   r7=0
30999 return
31000   r7=0
31099 return
31100   r7=0
31199 return
31200 rem !-----------------------------------------------!
```

Vorbereitet ist die Eingabemöglichkeit für zwölf rechte Seiten. Die GDGln. müssen in Normalform eingegeben werden. Die lokale Variable y7 wird bei jedem Aufruf der rechten Seiten inkrementiert.

Gemäß der im Abschnitt 1 getroffenen Vereinbarungen ist ein Applikationsprogramm in Richtung steigender Zeilennummern aus Gründen der Übersichtlichkeit wie folgt aufgebaut:

- Hauptprogramm
- Initialisierung
- Integrationsschritt
- rechte Seiten.

Wegen des Interpretersucheffekts wäre es aus Zeitgründen günstiger, die rechten Seiten möglichst an den Anfang des BASIC-Speicherbereichs zu legen.
Somit könnte man bei Problemen, die eine große Anzahl von Auswertungen der rechten Seiten erfordern, eine nicht unerhebliche Rechenzeitersparnis erzielen.
Zu diesem Zweck sind lediglich die bereits erwähnten Programmzeilen

. . . **on** i7 **gosub** . . .

zu ändern und die rechten Seiten vorzuverlegen.
In P 4.01 GDGL.EQNS ist die Eingabe der analytisch lösbaren GDGl. 2. Ordnung

$$\ddot{x} = -x$$

dargestellt.
Die allgemeine Lösung lautet:

$$x(t) = C_1 \sin(t) + C_2 \cos(t).$$

Die Normalform ist einfach angebbar:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = -x_1.$$

In dieser Form sind die beiden rechten Seiten ($\dot{x}_1, \dot{x}_2$) auch in die Zeilen 30000 und 30100 eingetragen.

Das Programm P 4.02 GDGL.TEST integriert die betrachtete GDGl. im Intervall $[0, 2\pi]$. Dabei gelangt das im Modul RKV5 implementierte Runge-Kutta-Verfahren zur Anwendung. Durch die konkreten Anfangsbedingungen wird aus der allgemeinen Lösung die Sinusfunktion ausgewählt. Bereits aus diesem einfachen Beispiel ist klar ersichtlich, daß zur Lösung einer bestimmten GDGl. bei Anwendung der vorgestellten Module nur eine sehr geringe Anzahl von Anweisungen notwendig ist.

Programm P 4.02. GDGL.TEST. Testbeispiel für die Integration gewöhnlicher Differentialgleichungen.

```
 1 n=2:m1=300000:dim x(2):f1=1e-4:f2=0:t=0:t1=3.141592654
 2 e1%=1:h=1:x(1)=0:x(2)=1
 3 gosub 28300:rem Initialisierung
 4 print "testbeispiel sinusfunktion : loesung mit rkv5"
 5 print :print
 6 c=ti:gosub 28366:c=ti-c:rem Ausführung der Integration
 7 print "x(1)=";x(1),"x(2)=";x(2)
 8 print "rechenzeit : ";c/50;" sekunden
 9 print "fkt.aufrufe: ";y7
10 print "fehlerflag : ";e1%:print:print:print
11 end
```

Globale Variable für alle Anfangswertproblemlöser sind (s. auch Tafel 4.1)

n	Anzahl der zu integrierenden GDGln. 1. Ordnung
m1	zulässige Anzahl von Auswertungen der rechten Seiten
x(n)	Anfangswerte, Integrationsergebnisse
f1	geforderte relative Genauigkeit
f2	geforderte absolute Genauigkeit
h	Schrittweitenvorschlag
t	Startpunkt der Integration
t1	Endpunkt der Integration
e1%	Fehlerflag.

Die verschiedenen möglichen Werte des Fehlerflags e1% haben die in Tafel 4.4 angegebene Bedeutung.

4.3. Verfahren und Programme zur Lösung von Anfangswertproblemen

4.3.1. Steifstabiles Trapezverfahren

Das in den **Programmen P 4.03.TRA4.INIT, P 4.04.TRA4.CONT** und **P 4.05.TRA4.PROG** realisierte steifstabile Trapezverfahren TRA4 ist ein implizites Einschrittverfahren der globalen Konvergenzordnung 4. Eine erste Näherung für den unbekannten Wert x_1^* wird berechnet mit dem *Euler-Cauchy-Prädiktor*

$$x_1 = x_0 + hf(t_0, x_0). \tag{4.66}$$

Programm P 4.03. TRA4.INIT. Initialisierung des steifstabilen Trapezverfahrens P 4.05 TRA4.PROG sowie seiner Steuerung P 4.04 TRA4.CONT.

```
27999 rem !-----------------------------------------------!
28000 rem !               Initialisierung TRA4              !
28001 rem !-----------------------------------------------!
28002   if e1%<>1 then 28062
28004   u=1
28006   if 1+u/2<>1 then u=u/2
28008   if 1+u/2<>1 then 28006
28010   if (n<=0 or n>12) then 28062
28012   if f1<0 or f2<0 or (f1+f2)=0 then 28062
28014   h7=t1-t
28016   h8=abs(t)
28018   if abs(t1)>h8 then h8=abs(t1)
28020   h8=10*u*h8
28022   if abs(h7)<h8 then 28062
28024   if h=0 then h=h8
28026   if h7/h<0 then h=-h
28028   if abs(h)>abs(h7)/2 then h=h7/3
28030   if (h1=0 or abs(h1)>abs(h)) then h1=h8
28032   if m1<=0 then m1=1000
28034   dim d(n)
28036   dim h(n)
28038   dim i(n)
28040   dim j(n)
28042   dim l(n)
28044   dim m(n)
28046   dim n(n)
28048   h7=1/2
28050   h8=1/3
28052   h9=h7*h7*h7
28054   s8=4096
28056   s9=6.5536e-4
28058   e1%=32
28060   goto 28064
28062   e1%=2
28064 return
28065 rem !-----------------------------------------------!
```

Alle globalen Eingangsparameter werden auf Gültigkeit überprüft. Benötigte Felder werden dimensioniert, und einige numerische Konstanten werden vereinbart. Der normale Rückkehrkode ist e1%=32; bei Fehlern wird e1%=2 gesetzt.

Programm P 4.04. TRA4.CONT. Schrittweitensteuerung und Richardson-Extrapolation für das steifstabile Trapezverfahren P 4.05 TRA4.PROG.
Falls auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x >= y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y) >= z * u **then** . . .

mit z = max(abs(x),abs(y)).

```
28065 rem !-----------------------------------------------!
28066 rem ! Schrittweitensteuerung durch R.-Extr. für TRA4 !
28067 rem !-----------------------------------------------!
28068   if e1%<>32 then 28136
28070   e1%=0
28072   y7=0
28074   for i7=1 to n
```

```
28076     i(i7)=x(i7)
28078   next i7
28080   t8=t
28082   h=h/h7
28084   gosub 28152
28086   for i7=1 to n
28088     j(i7)=x(i7)
28090     x(i7)=i(i7)
28092   next i7
28094   h=h*h7
28096   t=t8
28098   gosub 28152
28100   gosub 28152
28102   m7=0
28104   for i7=1 to n
28106     d(i7)=h8*(x(i7)-j(i7))
28108     x(i7)=x(i7)+d(i7)
28110     d(i7)=h9*abs(d(i7))/(f1*abs(x(i7))+f2+u)
28112     if d(i7)>m7 then m7=d(i7)
28114   next i7
28116   if m7<s9 then m7=s9
28118   if m7>1 then gosub 28140
28120   h=h/(sqr(sqr(m7)))
28122   if abs(h)<=abs(h1) then h=h1
28124   if h=h1 then e1%=e1% or 8
28126   if abs(t)>=abs(t1) then 28138
28128   if abs(t1-t)<abs(h/h7) then h=(t1-t)*h7
28130   if y7>=m1 then e1%=e1% or 4
28132   if y7>=m1 then 28138
28134   goto 28074
28136   e1%=e1% or 16
28138 return
28140   if m7>s8 then m7=s8
28142   t=t8
28144   for i7=1 to n
28146     x(i7)=i(i7)
28148   next i7
28150 return
28151 rem !-------------------------------------------------!
```

Folgende Fehlerkodes können durch die Routine generiert werden: 0,4,8,16. Die Bedeutung der Fehlerkodes ist Tafel 4.4 zu entnehmen.

Programm P 4.05. TRA4.PROG. Steifstabiles Trapezverfahren der globalen Konvergenzordnung 2. Die Konvergenzordnung wird durch die Richardson-Extrapolation in P 4.04 TRA4.CONT auf 4 erhöht.
Falls auf Rechnern mit $u \neq 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x > = y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y) > = z * u **then** . . .

mit z = max(abs(x),abs(y)).

```
28151 rem !-------------------------------------------------!
28152 rem !          Trapezverfahren (steifstabil)          !
28153 rem !-------------------------------------------------!
28154   c7=0
28156   for i7=1 to n
28158     h(i7)=x(i7)
28160   next i7
```

```
28162    t7=t
28164    for i7=1 to n
28166      if i7<=6 then on i7  gosub 30000,30100,30200,
           30300,30400,30500
28168      if i7>6  then on i7-6gosub 30600,30700,30800,
           30900,31000,31100
28170      m(i7)=r7
28172    next i7
28174    for i7=1 to n
28176      x(i7)=h(i7)+h*m(i7)
28178    next i7
28180    t=t7+h
28182    c7=c7+1
28184    for i7=1 to n
28186      if i7<=6 then on i7  gosub 30000,30100,30200,
           30300,30400,30500
28188      if i7>6  then on i7-6gosub 30600,30700,30800,
           30900,31000,31100
28190      n(i7)=r7
28192    next i7
28194    for i7=1 to n
28196      x(i7)=h(i7)+.5*(m(i7)+n(i7))*h
28198      if c7=1 then l(i7)=x(i7)
28200    next i7
28202    if c7=1 then 28182
28204    z7=0
28206    for i7=1 to n
28208      z7=z7+abs(l(i7)-x(i7))
28210      l(i7)=x(i7)
28212    next i7
28214    if z7<f2 then 28218
28216    if c7<f3 then 28182
28218  return
28119  rem !-------------------------------------------------!
```

Die Routine integriert die vorgegebenen GDGln. von t bis t+h. Sie generiert keine Fehlerkodes. Aufruf und Steuerung von P 4.05 TRA4.PROG erfolgen durch P 4.04 TRA4.CONT.

Danach erfolgt eine iterative Verbesserung des vorhergesagten Wertes mit der impliziten *Korrekturformel:*

$$x^*_{1,l+1} = x^*_{1,l} + \frac{h}{2}(f(t_0, x_0) + f(t_{1,l}, x^*_{1,l})). \tag{4.67}$$

Diese Iteration konvergiert, falls gilt

$$\frac{h}{2}|\delta f/\delta x| \leqq 1.$$

Das Programm TRA4 besteht aus der Initialisierung von P 4.03 (28000 bis 28064), der Schrittweitensteuerung bzw. Richardson-Extrapolation P 4.04 TRA4.CONT (28066 bis 28150), die die globale Fehlerordnung des Trapezverfahrens von zwei auf vier erhöht, sowie der eigentlichen Integrationsroutine P 4.05 TRA4.PROG (28152 bis 28218).
Betrachten wir zunächst das Trapezverfahren.
In Zeile 28154 wird der Iterationszyklenzähler auf Null gesetzt.
Danach werden die gültigen Integrationsergebnisse des letzten Schrittes auf das Feld h() gerettet, und die aktuelle Zeit t wird auf die lokale Variable t7 gespeichert. Dies geschieht, um bei einem Fehler innerhalb des nächsten auszuführenden Schrittes (z. B. zu große Schrittweite h in bezug auf die vorgegebenen Fehlerschranken) die bereits erzielten gültigen Resultate nicht zu verlieren.

In den Zeilen 28164 bis 28178 wird der Euler-Cauchy-Prädiktor durch die erstmalige Auswertung der rechten Seiten bestimmt.
Im weiteren Ablauf der Berechnungen wird zur aktuellen Zeit t die Schrittweite h addiert, damit bei der folgenden iterativen Verbesserung von x() eine evtl. vorhandene zeitabhängige Funktion in den rechten Seiten der GDGln. das richtige Argument angeboten bekommt.
In Zeile 28182 erfolgt der Eintritt in den Iterationszyklus, der die Programmzeilen bis 28216 belegt.
Nach Inkrementierung des Zyklenzählers c7 werden die rechten Seiten erneut ausgewertet.
In der Laufanweisung in den Zeilen von 28194 bis 28200 ist die eigentliche Trapezregel programmiert. Auf das Feld l() werden die Differenzen der x() aufeinanderfolgender Iterationen gespeichert.
Ist $c7 = 1$, so folgt ein unbedingter Rücksprung zum Beginn der Iteration, während für $c7 > 1$ die Abbruchbedingungen ausgewertet werden.
Dies sind zum einen die Norm der Abweichungen der Ergebnisse aufeinanderfolgender Iterationen (z7) sowie zum anderen die Anzahl der Iterationen selbst.
Bei der Erstellung des Applikationsprogramms ist zu beachten, daß außer dem relativen und dem absoluten Fehler der Integration (f1, f2) ebenfalls die maximale Anzahl f3 der auszuführenden Iterationen vorgegeben werden kann. Meist werden hier Werte kleiner als zehn ausreichend sein. Die Abbruchbedingungen werden in den Zeilen 28214 bzw. 28216 abgefragt.
Je nach Ergebnis wird die Routine beendet, oder es wird eine weitere Iteration ausgeführt.
Die Richardson-Extrapolation (gleichzeitig auch Schrittweitensteuerung) arbeitet wie bereits im Abschnitt 4.1.7.1 beschrieben.
Es werden zwei Schritte mit der Schrittweite h (28088, 28100) bzw. ein Schritt mit 2h (28084) ausgeführt. In Zeile 28108 werden die beiden Resultate zum korrigierten Ergebnis verknüpft.
In den Zeilen 28110 bis 28112 wird aus der Differenz d() der Schritte mit den Schrittweiten h bzw. 2h die Größe m7 als Schätzung für den globalen Rechnungs- und Verfahrensfehler ermittelt.
Die lokale Variable m7 wird anschließend zur Beeinflussung der Schrittweite h herangezogen.
Ist m7 größer als Eins, dann war der ausgeführte Schritt ungültig. Es erfolgt eine Verzweigung zur Routine in den Zeilen 28140 bis 28150, die die Ergebnisse des vorangegangenen Integrationsschritts auf die Variablen t und x() zurückschreibt.
Die eigentliche Veränderung der Schrittweite h geschieht in der Zeile 28120.
Durch geeignete Wahl der lokalen Variablen h9 sowie Einfügung eines Faktors ungleich Eins in Zeile 28120 kann die Kennlinie der Schrittweitensteuerung in weiten Grenzen variiert werden. Dies wird besonders dann nötig sein, wenn der verwendete Rechner eine von $u = 2.328 \cdot 10^{-10}$ abweichende Maschinenkonstante aufweist.
Die Anweisungen in den Zeilen 28122 bis 28134 dienen sowohl der Programmablaufsteuerung als auch dem Setzen des Fehlerflags e1%, falls dies notwendig sein sollte.
Die Anweisung **goto** 28074 in Zeile 28134 ist ein aus Gründen der Strukturierung verbotener Rücksprung über eine große Distanz zur Einleitung des nächsten Integrationsschritts. Der Strukturierungspurist unter den Lesern könnte an dieser Stelle z. B. eine **for-next**-Schleife mit ausreichend großer oberer Grenze (10000 od. ä.) einfügen.
Abschließend einige wenige Bemerkungen zur Initialisierungsroutine. In den Zeilen 28004 bis 28008 wird die Maschinenkonstante u ermittelt. Sie ist für die Festlegung der minimalen Schrittweite h8 (28020) und auch für die Berechnung der Verfahrens- und Rundungsfehler (28110) erforderlich.
Nach Test bzw. Festlegung der Größen h7 (maximal zulässige Schrittweite), h8 (minimal sinnvolle Schrittweite), h (Startschrittweite), m1 (Anzahl der zulässigen Auswertungen der rechten Seiten) sowie der Dimensionierung benötigter Felder werden noch einige lokale Variable deklariert. Der BASIC-Interpreter des vom Autor zur Programmentwicklung benutzten Rechners mit 6510-Prozessor reagiert auf die Vereinbarungen in den Zeilen 28048 bis 28056

durch Konvertierung der angegebenen dezimalen numerischen Konstanten ins interne Gleitkommaformat und Ablegen der Daten – unter den entsprechenden (lokalen) Variablennamen – in den Variablenspeicher.
Beim Aufruf der entsprechenden Variablen im Programm ist also ein bereits konvertierter Wert vorhanden, so daß die gewünschten Rechenoperationen ohne Zeitverzug ablaufen können.
Werden die benötigten numerischen Konstanten jedoch als ASCII-Quelltext ins Programm geschrieben, dann muß dieser Quelltext vor Ausführung einer arithmetischen Operation in das interne Gleitkommaformat konvertiert werden.
Mithin verliert man Zeit, spart aber – abhängig von der Länge des Variablennamens – Speicherplatz, da nicht sowohl Variablenname als auch Wert abgespeichert werden müssen.
Die Wahl der Vorgehensweise sei an dieser Stelle dem Leser überlassen.
Im Endeffekt bringt die Umwandlung von numerischen Konstanten wohl einen Geschwindigkeitsgewinn, macht aber das Programm wegen der Vielzahl der dann auftretenden lokalen Variablen auch unübersichtlich.
Beabsichtigt man, das Programm nach Austestung zu compilieren, so sind beide Wege weitgehend gleichwertig. Durch die Anweisung **return** in Zeile 28060 wird die Initialisierungsroutine verlassen, falls keine Fehler aufgetreten sind. In Zeile 28064 befindet sich der Fehlerausgang der Routine.

Programm P 4.06. RKV5.INIT. Initialisierung des Moduls P 4.08 RKV5.PROG sowie seiner Steuerung P 4.07 RKV5.CONT.

```
28299 rem !-------------------------------------------------!
28300 rem !              Initialisierung RKV5               !
28301 rem !-------------------------------------------------!
28302   if e1%<>1 then 28362
28304   u=1
28306   if 1+u/2<>1 then u=u/2
28308   if 1+u/2<>1 then 28306
28310   if (n<=0 or n>12) then 28362
28312   if f1<0 or f2<0 or (f1+f2)=0 then 28362
28314   h7=t1-t
28316   h8=abs(t)
28318   if abs(t1)>h8 then h8=abs(t1)
28320   h8=10*u*h8
28322   if abs(h7)<h8 then 28362
28324   if h=0 then h=h8
28326   if h7/h<0 then h=-h
28328   if abs(h)>abs(h7)/2 then h=h7/3
28330   if (h1=0 or abs(h1)>abs(h)) then h1=h8
28332   if m1<=0 then m1=1000
28334   dim d(n)
28336   dim h(n)
28338   dim i(n)
28340   dim j(n)
28342   dim k(n)
28344   dim l(n)
28346   h7=1/2
28348   h8=1/3
28350   h9=2/3
28352   s7=1/15
28354   s8=4096
28356   s9=6.5536e-4
28358   e1%=32
28360   goto 28364
28362   e1%=2
28364 return
28365 rem !-------------------------------------------------!
```

Alle globalen Eingangsparameter werden auf Gültigkeit geprüft. Nach der Dimensionierung benötigter Felder sowie der Vereinbarung einiger numerischer Konstanten erfolgt die normale Beendigung der Routine mit e1%=32. Treten Fehler auf, dann ist der Rückkehrkode e1%=2.

Programm P 4.07. RKV5.CONT. Schrittweitensteuerung durch Richardson-Extrapolation für die Runge-Kutta-(4,4)-Formel von P 4.08 RKV5.PROG.

```
28365 rem !-------------------------------------------------!
28366 rem ! Schrittweitensteuerung durch R.-Extr. für RKV5 !
28367 rem !-------------------------------------------------!
28368   if e1%<>32 then 28436
28370   e1%=0
28372   y7=0
28374   for i7=1 to n
28376     i(i7)=x(i7)
28378   next i7
28380   t8=t
28382   h=h/h7
28384   gosub 28452
28386   for i7=1 to n
28388     j(i7)=x(i7)
28390     x(i7)=i(i7)
28392   next i7
28394   h=h*h7
28396   t=t8
28398   gosub 28452
28400   gosub 28452
28402   m7=0
28404   for i7=1 to n
28406     d(i7)=s7*(x(i7)-j(i7))
28408     x(i7)=x(i7)+d(i7)
28410     d(i7)=h7*h7*abs(d(i7))/(f1*abs(x(i7))+f2+u)
28412     if d(i7)>m7 then m7=d(i7)
28414   next i7
28416   if m7<s9 then m7=s9
28418   if m7>1 then gosub 28440
28420   h=h9*h/(sqr(sqr(m7)))
28422   if abs(h)<=abs(h1) then h=h1
28424   if h=h1 then e1%=e1% or 8
28426   if abs(t)>=abs(t1) then 28438
28428   if abs(t1-t)<abs(h/h7) then h=(t1-t)*h7
28430   if y7>=m1 then e1%=e1% or 4
28432   if y7>=m1 then 28438
28434   goto 28374
28436   e1%=e1% or 16
28438 return
28440   if m7>s8 then m7=s8
28442   t=t8
28444   for i7=1 to n
28446     x(i7)=i(i7)
28448   next i7
28450 return
28451 rem !-------------------------------------------------!
```

Die Routine generiert die Fehlerkodes 0,4,8,16. Die Bedeutung der Fehlerkodes ist Tafel 4.4 zu entnehmen.

Programm P 4.08. RKV5.PROG. Klassisches vierstufiges Runge-Kutta-Verfahren der globalen Konvergenzordnung 4.

```
28541 rem !-----------------------------------------------!
28452 rem !  RK-(4,4)-Verfahren: Integration von t bis t+h !
28453 rem !-----------------------------------------------!
28454   for i7=1 to n
28456     h(i7)=x(i7)
28458     l(i7)=0
28460     k(i7)=0
28462   next i7
28464   t7=t
28466   for z7=1 to 4
28468     t=t7+h*h7*int(z7*h7)
28470     for i7=1 to n
28472       x(i7)=h(i7)+l(i7)*int(z7*h7)
28474     next i7
28476     for i7=1 to n
28478       if i7<=6 then on i7  gosub 30000,30100,30200,
          30300,30400,30500
28480       if i7>6  then on i7-6gosub 30600,30700,30800,
          30900,31000,31100
28482       l(i7)=r7*h*h7
28484       if (z7=1 or z7=4) then k(i7)=k(i7)+l(i7)*h8
28486       if (z7=2 or z7=3) then k(i7)=k(i7)+l(i7)*h9
28488     next i7
28490   next z7
28492   for i7=1 to n
28494     x(i7)=h(i7)+k(i7)
28496   next i7
28498   t=t7+h
28500 return
28501 rem !-----------------------------------------------!
```

Die Routine wird durch P 4.07 RKV5.CONT aufgerufen und gesteuert. Durch die in P 4.07 RKV5.CONT implementierte Richardson-Extrapolation wird die globale Konvergenzordnung auf 5 gesteigert. Die Routine integriert die vorgegebenen GDGln. von t bis t + h. In der Routine werden keine Fehlerkodes generiert.
Falls auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x > = y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y) > = z * u **then** . . .

mit z = max(abs(x),abs(y)).

4.3.2. Runge-Kutta-Verfahren

Das *Runge-Kutta-Verfahren 4. Ordnung* ist eines der meistverwendeten numerischen Integrationsverfahren für GDGln., da es mittlere Genauigkeitsansprüche mit sehr geringem numerischem und programmtechnischem Aufwand befriedigen kann.
Im **Programm P 4.06 RKV5.INIT, P 4.07 RKV5. CONT** und **P 4.08 RKV5.PROG** ist es mit einer Richardson-Extrapolation implementiert, so daß die globale Konvergenzordnung fünf erreicht wird.
Die Implementationsdetails (Initialisierungsroutine: P 4.06 RKV5.INIT, Richardson-Extrapolation mit Schrittweitensteuerung: P 4.07 RKV5.CONT, Integrationsschritt: P 4.08 RKV5.PROG) entsprechen denen des im Abschnitt 4.3.1 beschriebenen Trapezverfahrens.

Somit kann auf weitere Erläuterungen verzichtet werden.
Die Integrationsformel lautet:

$$
\begin{aligned}
k_1 &= hf(t_0, x_0) \\
k_2 &= hf\left(t_0 + \frac{h}{2}, x_0 + \frac{k_1}{2}\right) \\
k_3 &= hf\left(t_0 + \frac{h}{2}, x_0 + \frac{k_2}{2}\right) \\
k_4 &= hf(t_0 + h, x_0 + k_3) \\
x_1 &\approx x_1^* = x_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).
\end{aligned}
\tag{4.68}
$$

Man findet sie in den Zeilen 28466 bis 28490 des P 4.08 RKV5.PROG wieder.
Die Schrittweitensteuerung wird durch die Anweisungen in den Zeilen 28410 und 28420 des P 4.07 RKV5.CONT realisiert.
Die in diesen Anweisungen vorhandenen Faktoren wurden für eine 5-Byte-Gleitkommaarithmetik optimiert.
Bei einer von $u = 2.328 \cdot 10^{-10}$ abweichenden Maschinenkonstante werden an dieser Stelle Änderungen notwendig.

4.3.3. Runge-Kutta-Fehlberg-Verfahren

Runge-Kutta-Fehlberg-Verfahren beruhen auf einer Approximation der Lösungen der GDGln. (4.9) durch ihre Taylor-Entwicklung und anschließender Anwendung eines Runge-Kutta-Verfahrens zur Ermittlung von Näherungswerten für die gesuchten Funktionen.
Die Herleitung der einzelnen RKF-Verfahren findet man in der Originalliteratur [Fehl69], [Fehl70] bzw. bei [Enge85]. RKF-Verfahren sind explizite Einschrittverfahren, die eine Schrittweitensteuerung mit sehr geringem Aufwand ermöglichen, da das Verfahren der Ordnung $q_g - 1$ im Verfahren der Ordnung q_g eingebettet ist.
Es ist also im Gegensatz zu der Vorgehensweise in den Programmen TRA4 und RKV5 nicht notwendig, jeweils mit einfacher und doppelter Schrittweite in jedem Integrationsschritt zu rechnen, um dann eine Richardson-Extrapolation ausführen zu können. Dadurch arbeiten RKF-Verfahren effektiver als klassische RK-Verfahren.
Dabei ist zu beachten, daß sich diese Effektivitätssteigerung (Rechenzeitverringerung durch geringere Anzahl notwendiger Auswertungen der rechten Seiten der GDGln.) erst ab einem gewissen Umfang der rechten Seiten auswirken kann.
Dies liegt darin begründet, daß RKF-Verfahren einen höheren Implementationsaufwand (Programmspeicherplatz, Zahl der Anweisungen) erfordern als ein in der Konvergenzordnung gleichwertiges RK-Verfahren.
Im folgenden sollen zwei gebräuchliche RKF-Verfahren vorgestellt werden.
Es handelt sich hierbei um ein RKF-Verfahren der globalen Fehlerordnung fünf (**Programm P 4.09 RKF5.INIT, P 4.10 RKF5.PROG**) und eines der globalen Fehlerordnung sechs (**Programm P 4.11 RKF6.INIT, P 4.12 RKF6.CONT, P 4.13 RKF6.PROG**). Die Näherungen für die gesuchten Werte x_1 werden dabei jeweils durch die im Verfahren höherer Ordnung (q_g) eingebettete Approximationsformel der Konvergenzordnung $q_g + 1$ für das Verfahren niedrigerer Ordnung berechnet. Im Programm RKF5 wird mit einer Formel der Konvergenzordnung $q_g = 4$ gerechnet, bei RKF6 mit einer Formel der Konvergenzordnung $q_g = 5$. Die beiden Verfahren höherer Ordnung dienen jeweils zur Ermittlung des lokalen Verfahrensfehlers und zur Schrittweitensteuerung.
Bei beiden Verfahren tritt erstmals in diesem Abschnitt der Effekt auf, daß die Anzahl der Stufen, d. h. die Anzahl der Auswertungen der rechten Seiten, ungleich der globalen Konvergenzordnung ohne Richardson-Extrapolation ist.

Programm P 4.09. RKF5.INIT. Die globalen Eingangsgrößen werden auf Gültigkeit geprüft; benötigte Felder werden dimensioniert.

```
28599 rem !-------------------------------------------------!
28600 rem !               Initialisierung RKF5              !
28601 rem !-------------------------------------------------!
28602   if e1%<>1 then 28638
28604   u=1
28606   if 1+u/2<>1 then u=u/2
28608   if 1+u/2<>1 then 28606
28610   if (n<=0 or n>12) then 28638
28612   if f1<0 or f2<0 or (f1+f2)=0 then 28638
28614   h7=abs(t1-t)
28616   h1=abs(t)
28618   if abs(t1)>abs(t) then h1=abs(t1)
28620   h1=13*u*h1
28622   if h7<h1 then 28638
28624   if h8<=0 or h8<h7 then h8=h7
28626   if abs(h)<=13*u*abs(t) then h=h7
28628   s8=0
28630   dim l(n)
28632   dim n(n,6)
28634   e1%=32
28636   goto 28640
28638   e1%=2
28640 return
28641 rem !-------------------------------------------------!
```

Bei ordnungsgemäßem Abschluß ist e1%=32, sonst wird e1%=2 gesetzt.

Programm P 4.10. RKF5.PROG. Sehr genaues Runge-Kutta-Fehlberg-Verfahren der globalen Konvergenzordnung 5.

```
28641 rem !-------------------------------------------------!
28642 rem !     Schrittweitensteuerung/Fehlertest RKF5      !
28643 rem !-------------------------------------------------!
28644   rem h begrenzen-Schritt ausführen-Fehler schätzen
28646   if e1%=32 then 28652
28648   e1%=16
28650   goto 28828
28652   y7=0
28654   t8=t
28656   h=abs(h)
28658   if h7<h then h=h7
28660   if t1<t8 then h=-h
28662   if abs(t1-t8)>1.25*abs(h) then 28670
28664   s8=1
28666   h=t1-t8
28668   rem Schritt ausführen
28670   t=t8
28672   k7=1
28674   gosub 28818
28676   for i7=1 to n
28678     l(i7)=x(i7)
28680   next i7
28682   for i7=1 to n
28684     x(i7)=l(i7)+h*n(i7,1)/4
28686   next i7
28688   t=t8+h/4
28690   k7=2
28692   gosub 28818
```

```
28694   for i7=1 to n
28696     s7=n(i7,1)*3/32+n(i7,2)*9/32
28698     x(i7)=l(i7)+h*s7
28700   next i7
28702   t=t8+h*3/8
28704   k7=3
28706   gosub 28818
28708   for i7=1 to n
28710     s7=n(i7,1)*1932/2197-n(i7,2)*7200/2197
          +n(i7,3)*7296/2197
28712     x(i7)=l(i7)+h*s7
28714   next i7
28716   t=t8+h*12/13
28718   k7=4
28720   gosub 28818
28722   for i7=1 to n
28724     s7=n(i7,1)*439/216-n(i7,2)*8+n(i7,3)*3680/513
28726     s7=s7-n(i7,4)*845/4104
28728     x(i7)=l(i7)+h*s7
28730   next i7
28732   t=t8+h
28734   k7=5
28736   gosub 28818
28738   for i7=1 to n
28740     s7=-n(i7,1)*8/27+n(i7,2)*2-n(i7,3)*3544/2565
28742     s7=s7+n(i7,4)*1859/4104-n(i7,5)*11/40
28744     x(i7)=l(i7)+h*s7
28746   next i7
28748   t=t8+h/2
28750   k7=6
28752   gosub 28818
28754   for i7=1 to n
28756     s7=n(i7,1)*25/216+n(i7,3)*1408/2565
28758     s7=s7+n(i7,4)*2197/4104-n(i7,5)/5
28760     x(i7)=l(i7)+h*s7
28762   next i7
28764   rem Fehlerschätzung
28766   m7=0
28768   for i7=1 to n
28770     s7=n(i7,1)/360-n(i7,3)*128/4275-n(i7,4)*2197/75240
28772     s7=s7+n(i7,5)/50+n(i7,6)*2/55
28774     d7=abs(s7)/(f1*abs(x(i7))+f2+u)
28776     if d7>m7 then m7=d7
28778   next i7
28780   rem Schrittweitensteuerung:h verkleinern bei m7>1
28782   if m7>1 then 28794
28784   t8=t8+h
28786   if s8<>1 then 28792
28788   e1%=0
28790   goto 28828
28792   if m7<6.5536e-4 then m7=6.5536e-4
28794   if m7>4096 then m7=4096
28796   h=.8*h/sqr(sqr(m7))
28798   if abs(h)>h1 then 28804
28800   e1%=e1% or 8
28802   h=h1
28804   if y7<m1 then 28810
28806   e1%=e1% or 4
28808   goto 28828
28810   if m7<=1 then 28656
28812   s8=0
28814   goto 28682                                    ! Schritt
28816   rem Berechnung der Korrekturwerte             ! wiederholen
28818   for j7=1 to n
28820     if j7<=6 then on j7  gosub 30000,30100,30200,
          30300,30400,30500
```

```
28822      if j7>6  then on j7-6gosub 30600,30700,30800,
           30900,31000,31100
28824      n(j7,k7)=r7
28826    next j7
28828  return
28829  rem !-----------------------------------------------!
```

Die Routine beinhaltet sowohl die Steuerung und Fehlerschätzung als auch die Ausführung des eigentlichen Integrationsschritts. Die Routine generiert die Fehlerkodes 0,4,8,16. Die Bedeutung der Fehlerkodes ist Tafel 4.4 zu entnehmen.
Falls auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x >= y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y) >= z * u **then** . . .

mit z = max(abs(x),abs(y)).

Programm P 4.11. RKF6.INIT. Initialisierung von P 4.13 RKF6.PROG sowie seiner Steuerung P 4.12 RKF6.CONT.

```
28899 rem !-----------------------------------------------!
28900 rem !              Initialisierung RKF6             !
28901 rem !-----------------------------------------------!
28902   if e1%<>1 then 28972
28904   u=1
28906   if 1+u/2<>1 then u=u/2
28908   if 1+u/2<>1 then 28906
28910   if (n<=0 or n>12) then 28972
28912   if f1<0 or f2<0 or (f1+f2)=0 then 28972
28914   h7=t1-t
28916   h8=abs(t)
28918   if abs(t1)>h8 then h8=abs(t1)
28920   h8=10*u*h8
28922   if abs(h7)<h8 then 28972
28924   if h=0 then h=h8
28926   if h7/h<0 then h=-h
28928   if abs(h)>abs(h7)/2 then h=h7/3
28930   if (h1=0 or abs(h1)>abs(h)) then h1=h8
28932   if m1<=0 then m1=1000
28934   dim p(n)
28936   dim q(n)
28938   dim r(n)
28940   dim s(n)
28942   dim t(n)
28944   dim u(n)
28946   dim v(n)
28948   dim w(n)
28950   dim y(n)
28952   dim z(n)
28954   a7=0
28956   c7=1
28958   c8=1
28960   e7=5*f1
28962   if s7=0 then s7=1
28964   if abs(s7)<>1 then s7=0
28966   s9=1
28968   e1%=32
28970   goto 28974
28972   e1%=2
28974 return
28975 rem !-----------------------------------------------!
```

Die globalen Eingangsgrößen werden auf ihre Richtigkeit überprüft. Der Rückkehrkode bei ordnungsgemäßer Initialisierung ist e1%=32, sonst e1%=2.

Programm P 4.12. RKF6. CONT. Schrittweitensteuerung und Fehlertest für das Runge-Kutta-Fehlberg-Verfahren P 4.13 RKF6.PROG.

```
28975 rem !-----------------------------------------------!
28976 rem !     Schrittweitensteuerung/Fehlertest RKF6    !
28977 rem !-----------------------------------------------!
28978   if e1%<>32 then 29108
28980   e1%=0
28982   c7=1
28984   y7=0
28986   c8=1
28988   a7=0
28990   s9=1
28992   t7=t                                              ! Schritt in Ord-
28994   for i7=1 to n                                     ! nung,weiter
28996     w(i7)=x(i7)
28998   next i7
29000   h7=h                                              ! Fehler zu gross
29002   if c8=0 then e1%=e1% or 8                         ! Schritt nochmal
29004   p7=t+h-t1
29006   s8=1
29008   if not (h<=0 and p7<0 or h>=0 and p7>0) then 29014
29010   h=t1-t
29012   c7=0
29014   for k7=1 to 8
29016     for i7=1 to n
29018       if i7<=6 then on i7  gosub 30000,30100,30200,
          30300,30400,30500
29020       if i7>6  then on i7-6gosub 30600,30700,30800,
          30900,31000,31100
29022       y(i7)=r7
29024     next i7
29026     for i7=1 to n
29028       on k7gosub 29112,29122,29132,29142,29152,
          29164,29174,29186
29030       x(i7)=w(i7)+h*p7
29032       if k7<>8 then 29084
29034       if s7=0 then 29070
29036       m7=.09*abs((p(i7)+u(i7)-v(i7)-y(i7))*h)/
          (f1*abs(x(i7))+f2+u)
29038       r8=m7*(f1*abs(x(i7))+f2+u)
29040       if (m7<1 or c8=0) then 29068
29042       c7=1
29044       s9=0
29046       a7=1
29048       z(i7)=0
29050       if m7>4096 then m7=4096
29052       h=.7*h/sqr(sqr(m7))
29054       if abs(h)<h1 then c8=0
29056       if abs(h)<h1 then h=sign(h)*h1
29058       for j7=1 to n
29060         x(j7)=w(j7)
29062       next j7
29064       t=t7
29066       goto 29000
29068       if m7>=.95 then s8=0
29070       a7=a7+1
29072       if abs((q(i7)-p(i7))*h*.083333334)<r8
          then z(i7)=z(i7)+1
29074       if a7<=384 then 29084
29076       if z(i7)<=24 then 29080
29078       e1%=e1% or 64
```

```
29080        z(i7)=0
29082        if a7>392 then a7=0
29084      next i7
29086    next k7
29088    if y7>=m1 then e1%=e1% or 4
29090    if y7>=m1 then 29110
29092    if s7=0 then 29104
29094    if (s8=0 or s9=0 or c7=0) then 29102
29096    if m7<6.5536e-4 then m7=6.5536e-4
29098    h=.7*h/sqr(sqr(m7))
29100    c8=1
29102    s9=1
29104    if c7=1 then 28992
29106    goto 29110
29108    e1%=16
29110 return
29111 rem !-------------------------------------------------!
```

Die zu integrierenden GDGln. werden auf Steifheit untersucht, da das Verfahren steife Probleme nicht effektiv bearbeiten kann. Die Routine generiert die Fehlerkodes 0,4,8,16,64. Die Bedeutung der Fehlerkodes ist Tafel 4.4 zu entnehmen.
Falls auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

Programm P 4.13. RKF6.PROG. Sehr genaues achtstufiges Runge-Kutta-Fehlberg-Verfahren der globalen Konvergenzordnung 6. Die Routine integriert die vorgegebenen GDGln. von t bis t+h. Sie generiert keine Fehlerkodes.

```
29111 rem !-------------------------------------------------!
29112 rem !        Stützstellen des Verfahrens RKF6         !
29113 rem !-------------------------------------------------!
29114   p(i7)=y(i7):rem  1. Stützstelle
29116   if i7=1 then t=t7+h*.166666667
29118   p7=p(i7)*.166666667
29120 return
29122 rem  2. Stützstelle
29124   q(i7)=y(i7)
29126   if i7=1 then t=t7+h*.266666667
29128   p7=(p(i7)+4*q(i7))*.053333333
29130 return
29132 rem  3. Stützstelle
29134   r(i7)=y(i7)
29136   if i7=1 then t=t7+h*.666666667
29138   p7=p(i7)*.833333333-2.66666667*q(i7)+2.5*r(i7)
29140 return
29142 rem  4. Stützstelle
29144   s(i7)=y(i7)
29146   if i7=1 then t=t7+h*.8
29148   p7=-1.6*p(i7)+q(i7)*5.76-4*r(i7)+.64*s(i7)
29150 return
29152 rem  5. Stützstelle
29154   t(i7)=y(i7)
29156   if i7=1 then t=t7+h
29158   p7=1.128125*p(i7)-q(i7)*3.6+3.1796875*r(i7)
        -.1375*s(i7)
29160   p7=p7+.4296875*t(i7)
```

```
29162 return
29164 rem  6. Stützstelle
29166   u(i7)=y(i7)
29168   if i7=1 then t=t7
29170   p7=-.0171875*p(i7)+r(i7)*.04296875-s(i7)*.06875
        +.04296875*t(i7)
29172 return
29174 rem  7. Stützstelle
29176   v(i7)=y(i7)
29178   if i7=1 then t=t7+h
29180   p7=.1453125*p(i7)-q(i7)*3.6+3.13671875*r(i7)
        -.06875*s(i7)+v(i7)
29182   p7=p7+t(i7)*.38671875
29184 return
29186 rem  8. Stützstelle
29188   if i7=1 then t=t7+h
29190   p7=.080729167*p(i7)+r(i7)*.399502841+s(i7)
        *.28125+.16276042*t(i7)
29192   p7=p7+u(i7)*.0757575758
29194 return
29195 rem !-------------------------------------------------!
```

Vorzugsweise sollte das Verfahren für gutkonditionierte Probleme und mit einer Maschinenkonstante u kleiner als 10^{-10} eingesetzt werden.
Falls auf Rechnern mit $u \neq 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

Im Programm RKF5 ist für die eigentliche Integration (eingebettetes Verfahren) eine fünfstufige Integrationsformel der globalen Konvergenzordnung vier implementiert, während das zur Fehlerschätzung herangezogene Verfahren sechsstufig mit $q_g = 5$ ist.
Im Programm RKF6 ist die zur Integration benutzte Formel fünfstufig mit $q_g = 5$, während zur Schrittweitensteuerung eine achtstufige Formel mit $q_g = 6$ herangezogen wird.
Es ist deutlich erkennbar, daß ein gewünschter höherer Konvergenzgrad durch eine erhöhte Anzahl von Auswertungen der rechten Seiten erkauft werden muß.
Für Mikrorechnerapplikationen dürfte bei einer achtstufigen Formel die Grenze für eine sinnvolle Implementation (Speicherplatz, Rechenzeit) erreicht sein.
Nun sollen einige Bemerkungen zu den Implementationsdetails der beiden Verfahren folgen. Dabei wird so vorgegangen, daß jeweils zuerst das Kernstück des Verfahrens – die Integrationsformel – erläutert wird und dann einige Hinweise zu den Softwaredetails gegeben werden.
Der Formelsatz für das im Modul RKF5 implementierte Verfahren lautet [Albr79]:

$$
\begin{aligned}
k_1 &= hf(t_0, x_0) \\
k_2 &= hf\left(t_0 + \frac{h}{4},\ x_0 + \frac{h}{4} k_1\right) \\
k_3 &= hf\left(t_0 + \frac{3}{8}h,\ x_0 + \frac{3}{32} k_1 + \frac{9}{32} k_2\right) \\
k_4 &= hf\left(t_0 + \frac{11}{13}h,\ x_0 + \frac{1932}{2197} k_1 - \frac{7200}{2197} k_2 + \frac{7296}{2197} k_3\right)
\end{aligned}
\tag{4.69}
$$

$$k_5 = hf\left(t_0 + h,\ x_0 + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\right)$$

$$k_6 = hf\left(t_0 + \frac{h}{2},\ x_0 - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\right).$$

Die Berechnung der aktuellen Zeit t8 sowie die Verzweigung zur Subroutine in den Zeilen 28816 bis 28828 zur Berechnung der k_i erfolgt in den Zeilen:

k_1: 28670, 28674
k_2: 28688, 28692
k_3: 28702, 28706
k_4: 28716, 28720
k_5: 28732, 28736
k_6: 28748, 28752

Zum Verständnis des Programms ist zu beachten, daß vor der Verzweigung zur Routine in den Zeilen 28816 bis 28828 die entsprechenden Werte des Feldes x() aus den korrespondierenden Werten l() und den bis dahin ermittelten $k_1, k_2, \ldots, k_{i-1}$ berechnet werden müssen. Die Vorbereitungen zur Ermittlung der k_i (i = 2(1)6) werden also im Programmsegment für k_{i-1} getroffen.
Gemäß

$$x^*_{1,5} = \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28651}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6 \quad (q_9 = 5) \tag{4.70}$$

bzw.

$$x^*_{1,4} = \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5 \quad (q_9 = 4) \tag{4.71}$$

werden die k_i zur Approximation der Lösung der zu integrierenden GDGl. verknüpft.
In der vorliegenden Implementation wird explizit lediglich $x^*_{1,4}$ (28756, 28758) berechnet, während $x^*_{1,5}$ gemäß

$$e^*_{5,4} = x^*_{1,5} - x^*_{1,4} \tag{4.72}$$

zur Fehlerschätzung in den Zeilen 28770, 28772 herangezogen wird.
Obwohl die Software gut kommentiert ist, sollen die einzelnen Programmabschnitte im folgenden angegeben werden:

- Initialisierung
  (28600 bis 28640)
- Schrittweitenbegrenzung, Fehlertest
  (28644 bis 28666)
- Integrationsschritt
  (28668 bis 28762)
- Fehlerschätzung
  (28764 bis 28778)
- Schrittweitensteuerung, Fehlertest und entsprechende Verzweigungen
  (28780 bis 28814)
- Berechnung der k_i
  (28816 bis 28828).

Wichtige lokale Variable und Felder sind:

n() speichert die k_i, wobei k7 der aktuelle Index ist

l() speichert die x_0-Werte

s8 Steuergröße für den Programmablauf
s8 = 0: Integration kann weiterlaufen, Ausstieg nur durch Fehler
s8 = 1: letzter Integrationsschritt, Abschluß des Programms erfolgt ordnungsgemäß (28786 bis 28790)

t8 aktuelle Zeit im Integrationsschritt

Alle anderen lokalen Variablen sind entweder Hilfsgrößen (s7, d7, h7, h8) oder haben die gleiche Bedeutung wie in den Programmen TRA4 und RKV5 (z. B. m7).
Auch im Programm RKF6 gibt es Ansatzmöglichkeiten für die Verringerung der Laufzeit. Dazu ist z. B. die Prozedur zur Berechnung der k_i (28816 bis 28828) an den Anfang des Moduls zu legen, und das Programm ist so umzustrukturieren, daß die Rücksprünge in den Zeilen 28810 und 28814 entfallen können.
Dies überläßt der Autor allerdings dem Leser.
Soll das Programm auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ eingesetzt werden, so sind für optimale Laufeigenschaften Änderungen in den Zeilen 28774 (abs(s7 . . .)) und 28796 (h = . . .h/ . . .) vorzunehmen.
Nun einige Ausführungen zum Programm RKF6 (P 4.11 RKF6.INIT, P 4.12 RKF6. CONT, P 4.13 RKF6.PROG).
Der Formelsatz ist gegeben durch

$$
\begin{aligned}
k_1 &= hf(t_0, x_0)\\
k_2 &= hf\left(t_0 + \frac{h}{6}, x_0 + \frac{k_1}{6}\right)\\
k_3 &= hf\left(t_0 + \frac{4}{15}h, x_0 + \frac{4}{75}k_1 + \frac{16}{75}k_2\right)\\
k_4 &= hf\left(t_0 + \frac{2}{3}h, x_0 + \frac{5}{6}k_1 - \frac{8}{3}k_2 + \frac{5}{2}k_3\right)\\
k_5 &= hf\left(t_0 + \frac{4}{5}h, x_0 - \frac{8}{5}k_1 + \frac{144}{25}k_2 - 4k_3 + \frac{16}{25}k_4\right)\\
k_6 &= hf\left(t_0 + h, x_0 + \frac{361}{320}k_1 - \frac{18}{5}k_2 + \frac{407}{128}k_3 - \frac{11}{80}k_4 + \frac{55}{128}k_5\right)\\
k_7 &= hf\left(t_0, x_0 - \frac{11}{640}k_1 + \frac{11}{256}k_3 - \frac{11}{160}k_4 + \frac{11}{256}k_5\right)\\
k_8 &= hf\left(t_0 + h, x_0 + \frac{93}{640}k_1 - 3.6k_2 + \frac{803}{256}k_3 - \frac{11}{160}k_4 + \frac{99}{256}k_5 + k_7\right).
\end{aligned}
\tag{4.73}
$$

In den Zeilen 29112 bis 29194 werden die k_i berechnet. Dies ist jeweils durch "rem 1. Stützstelle" usw. kenntlich gemacht.
Jede dieser Berechnungen ist als Prozedur aufgebaut und wird durch **return** abgeschlossen. Auch hierbei ist wieder zu beachten, daß die Vorbereitungen zur Berechnung der k_i (i = 2(1)8) in den Prozeduren zur Ermittlung von k_{i-1} vorgenommen werden.
In der Subroutine zur Ermittlung von k_8 wird gleichzeitig die Approximation für x_1 berechnet gemäß

$$x_1 \approx x_1^* = \frac{31}{384}k_1 + \frac{1125}{2816}k_2 + \frac{9}{32}k_3 + \frac{9}{32}k_4 + \frac{125}{768}k_5. \tag{4.74}$$

Die Fehlerschätzungsformel lautet:

$$e^*_{6,5} = x^*_{1,6} - x^*_{1,5} = \frac{5}{66}(k_1 + k_6 - k_7 - k_8). \tag{4.75}$$

Sie ist in Zeile 29036 implementiert, wobei es sich zeigte, daß der theoretisch optimale Faktor 5/66 beim zur Programmentwicklung und Austestung verwendeten Rechner mit 6510-Prozessor (4 Mantissenbytes) besser durch 0.09 zu ersetzen war. Im Gegensatz zum Programm RKF5, wo zur Speicherung der k_i ein zweidimensionales Feld verwendet wurde, sind bei Programm RKF6 die k_i in eindimensionalen Arrays der Dimension n untergebracht:

k_1 : p ()
k_2 : q ()
k_3 : r ()
k_4 : s ()
k_5 : t ()
k_6 : u ()
k_7 : v ()
k_8 : y () (= f (t, x)).

Im Feld w () werden die x_i aufgehoben, während z () für den *Steifheitstest nach Shampine* benötigt wird.
Der Programmablauf von RKF6 wird durch die logische Verknüpfung einer Reihe von Flags gesteuert, deren Bedeutung im folgenden angegeben ist:

c7 = 1 Integrationsziel noch nicht erreicht; es ist mindestens noch ein Schritt mit der Schrittweite h notwendig
c7 = 0 letzter Integrationsschritt mit der Schrittweite h = t1 – t
c8 = 1 wird in der Initialisierungsroutine gesetzt;
die Schrittweite h kann nur verkleinert werden, falls c8 = 1 gilt
c8 = 0 |h| < h1; h = sign (h)*h1
s7 = 1 Schrittweitensteuerung ein
Steifheitstest ein
s7 = 0 Schrittweitensteuerung aus
Steifheitstest ein
s8 = 1 es sind zu große Fehler im Integrationsschritt aufgetreten, der Schritt muß wiederholt werden
s8 = 0 Fehlerschätzung entspricht etwa geforderter Genauigkeit, h kann deshalb nicht vergrößert werden
s9 = 1 gilt nach Ausführung der Initialisierung und nach jedem Integrationsschritt, signalisiert:
alles in Ordnung
s9 = 0 Fehler zu groß, h noch nicht zu klein, h darf nicht vergrößert werden (s. auch c8)

In Analogie zu Programm RKF5 sei die Aufteilung des Moduls in die einzelnen Funktionselemente angegeben:

- Initialisierung (28900 bis 28974)
- Schrittweitenbegrenzung, Fehlertest, Programmablaufsteuerung (28976 bis 29012)
- Auswertung der rechten Seiten, Ermittlung der k_i (29014 bis 29034)
- Fehlerschätzung, Schrittweitensteuerung (29036 bis 29068, 29096 bis 29102, optimiert für $u = 2.328 \cdot 10^{-10}$)
- Steifheitstest nach *Shampine*
  (29070 bis 29086)
- Prozeduren für die k_i (29112 bis 29194).

Hierbei ist zu beachten, daß wegen der relativ komplexen Programmstruktur keine klare Abgrenzung der einzelnen Programmteile möglich ist.
Zwei Laufanweisungen werden geschachtelt, um die Rechenzeit trotz des doch erheblichen numerischen Aufwands in Grenzen zu halten. Der Autor hofft aber dennoch, einen vertretbaren Kompromiß zwischen Schnelligkeit und Übersichtlichkeit des Programms gefunden zu haben.
Der Modul RKF6 enthält das Verfahren der höchsten Konvergenzordnung der vorgestellten expliziten Einschrittverfahren. Deshalb ist, wie im Abschnitt 4.1.9 bereits dargelegt, ein Steifheitstest angebracht. Dieser Test spricht an, wenn ein Differentialgleichungsproblem für das verwendete RKF6-Verfahren zu steif ist.
Die Idee dieses Tests stammt von *Shampine* [Sham77]; der Vorschlag zur Implementation ist aus [Albr79] entnommen. Im RKF6-Verfahren ist über die Koeffizienten k_1 und k_2 der lokale Verfahrensfehler des Euler-Cauchy-Verfahrens berechenbar:

$$e_{EC} \approx \frac{1}{2}(k_2 - k_1).$$

Dieser Fehler muß i. allg. wesentlich größer als der in der Zeile 29036 ermittelte lokale Verfahrensfehler des RKF6-Verfahrens sein. In jedem gültigen Schritt wird nun getestet, ob dies der Fall ist. Tritt in einer Anzahl von Schritten (hier mehr als 24 von 384) das Gegenteil auf, d. h., die Schrittweite des RKF6-Verfahrens ist wegen Stabilitätsproblemen sehr klein geworden, so wird Steifheit (e1% = 64) angezeigt.
In diesem Fall ist ein Verfahren geringerer Konvergenzordnung zur Behandlung des Problems einzusetzen.
Im Gegensatz zu den Programmen TRA4 und RKV5 sind bei den Programmen RKF5 und RKF6 die Koeffizienten der Approximationsformeln in den Quelltext als numerische Konstanten aufgenommen worden – mit allen bereits diskutierten Vor- und Nachteilen.
Es soll an dieser Stelle darauf hingewiesen werden, wie unübersichtlich die Programmlistings durch die Einführung von lokalen Variablen für diese Konstanten geworden wären.

4.3.4. Extrapolationsverfahren nach *Gragg-Stoer*

Das in den **Programmen P 4.14 EXGS.INIT, P 4.15 EXGS.CONT** und **P 4.16 EXGS.PROG** vorgestellte Extrapolationsverfahren nach *Gragg-Stoer* basiert auf einer in [Herr83] angegebenen Implementation. Diese wurde so modifiziert, daß Systeme von GDGln. (4.9) integriert werden können. Außerdem ist wie bei allen bisher angegebenen Programmen eine Schrittweitensteuerung durch Vorgabe des gewünschten relativen bzw. absoluten Fehlers möglich. Das Extrapolationsverfahren beruht auf der Ermittlung der gesuchten Werte $\bar{x}_1^*$ durch die *Graggsche Schrittfunktion* S, die eine Fehlerentwicklung in Termen von h^2 zuläßt. Somit wird eine *Romberg-Extrapolation* zur Verbesserung der Genauigkeit der Integrationsergebnisse x_1^* möglich. Weitere Ausführungen zu den theoretischen Grundlagen findet man in [Albr79] [Enge85] [Herr 83].

Programm P 4.14. EXGS.INIT. Initialisierung des Moduls P 4.16 EXGS.PROG sowie seiner Steuerung P 4.15 EXGS.CONT.

```
29199 rem !-------------------------------------------------!
29200 rem !               Initialisierung EXGS              !
29201 rem !-------------------------------------------------!
29202   if e1%<>1 then 29252
29204   u=1
29206   if 1+u/2<>1 then u=u/2
29208   if 1+u/2<>1 then 29206
```

```
29210   if (n<=0 or n>12) then 29252
29212   if f1<0 or f2<0 or (f1+f2)=0 then 29252
29214   h7=t1-t
29216   h8=abs(t)
29218   if abs(t1)>h8 then h8=abs(t1)
29220   h8=10*u*h8
29222   if abs(h7)<h8 then 29252
29224   if h=0 then h=h8
29226   if h7/h<0 then h=-h
29228   if abs(h)>abs(h7)/2 then h=h7/3
29230   if (h1=0 or abs(h1)>abs(h)) then h1=h8
29232   if m1<=0 then m1=1000
29234   dim r(n)
29236   dim s(5,n)
29238   dim t(5,n)
29240   dim u(n)
29242   dim v(n)
29244   dim w(n)
29246   dim z(n)
29248   e1%=32
29250   goto 29254
29252   e1%=2
29254 return
29255 rem !-------------------------------------------------!
```

Die globalen Eingabeparameter werden auf Gültigkeit geprüft. Bei Fehlern wird e1%=2 gesetzt; sonst erfolgt Rückkehr mit e1%=32.

Programm P 4.15. EXGS.CONT. Schrittweitensteuerung und Fehlertest für P 4.16 EXGS.-PROG.

```
29255 rem !-------------------------------------------------!
29256 rem !   Schrittweitensteuerung/Fehlertest für EXGS    !
29257 rem !-------------------------------------------------!
29258   if e1%<>32 then e1%=16
29260   if e1%<>32 then 29304
29262   e1%=0
29264   y7=0
29266   t7=t
29268     for i7=1 to n                                  ! Retten der
29270     z(i7)=x(i7)                                    ! Anfangswerte
29272   next i7
29274   gosub 29306
29276   if (e1%and251)<>e1% then 29304
29278   if m7<6.5536e-4 then m7=6.5536e-4
29280   if m7>1 then gosub 29294
29282   h=.7*h/(sqr(sqr(m7)))
29284   if abs(h)<=abs(h1) then h=h1
29286   if h=h1 then e1%=e1% or 8
29288   if abs(t)>=abs(t1) then 29304
29290   if abs(t1-t)<abs(h) then h=(t1-t)
29292   goto 29266
29294   if m7>4096 then m7=4096
29296   t=t7
29298   for i7=1 to n
29300     x(i7)=z(i7)
29302   next i7
29304 return
29305 rem !-------------------------------------------------!
```

Die Routine generiert die Fehlerkodes 0,8,16. Die Bedeutung der Fehlerkodes ist Tafel 4.4 zu entnehmen.

Programm P 4.16. EXGS.PROG. Integration eines Systems von GDGln. nach dem Extrapolationsverfahren von *Gragg-Stoer*.

```
29305 rem !-----------------------------------------------!
29306 rem !          Extrapolation von t bis t+h          !
29307 rem !-----------------------------------------------!
29308   for k7=1 to 5
29310     if y7>m1 then e1%=e1% or 4
29312     if y7>m1 then 29432
29314     z7=2^k7
29316     h7=h/z7
29318     for i7=1 to n
29320       u(i7)=z(i7)
29322       x(i7)=z(i7)
29324     next i7
29326     t=t7
29328     for i7=1 to n
29330       if i7<=6 then on i7  gosub  30000,30100,30200,
          30300,30400,30500
29332       ifi7>6   then on i7-6gosub  30600,30700,30800,
          30900,31000,31100
29334       v(i7)=z(i7)+h7*r7
29336     next i7
29338     for j7=1 to z7-1
29340       t=t7+j7*h7
29342       for i7 =1 to n
29344         x(i7)=v(i7)
29346       next i7
29348       for i7 1 to n
29350         if i7<=6 then  on i7  gosub 30000,30100,30200,
            30300,30400,30500
29352         if i7>6  then  on i7-6gosub 30600,30700,30800,
            30900,31000,31100
29354         w(i7)=u(i7)+2*h7*r7
29356         u(i7)=v(i7)
29358         v(i7)=w(i7)
29360       next i7
29362     next j7
29364     t=t7+h
29366     for i7=1 to n
29368       x(i7)=v(i7)
29370     next i7
29372     for i7=1 to n
29374       if i7<=6 then on i7 gosub 30000,30100,30200,
          30300,30400,30500
29376       if i7>6  then oni7-6gosub 30600,30700,30800,
          30900,31000,31100
29378       s(k7,i7)=(u(i7)+v(i7)+h7*r7)/2
29380     next i7
29382     if k7=1 then 29424
29384     p7=1
29386     for j7=k7 to 2 step -1
29388       p7=p7*4
29390       for i7=1 to n
29392         t(j7,i7)=(p7*s(j7,i7)-s(j7-1,i7))/(p7-1)
29394       next i7
29396     next j7
29398     if k7=2 then 29424
29400     m7=0
29402     v7=1e38
29404     for i7=1 to n
29406       p7=.09*abs(t(k7,i7)-r(i7))/(f1*abs(r(i7))+f2+u)
29408       if p7>m7 then m7=p7
29410       if p7<v7 then v7=p7
```

```
29412      next i7
29414      if k7<4 then 29416
29416      if (m7<1 or (v7<1e-3*m7 and v7<1)) then 29426
29418      for i7=1 to n
29420        r(i7)=t(k7,i7)
29422      next i7
29424    next k7
29426    for i7=1 to n
29428      x(i7)=r(i7)
29430    next i7
29432  return
29433  rem !-----------------------------------------------!
```

Die Routine generiert den Fehlerkode 4. Die Routine integriert die vorgegebenen GDGln. von t bis t+h. Oszillationen im Extrapolationsschritt werden vermieden, um eine möglichst geringe Rechenzeit zu gewährleisten.
Falls auf Rechnern mit $u \neq 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x−y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

Der Algorithmus lautet in der in diesem Abschnitt üblichen Notation (Zeilennummern in Klammern):

$k = 1\,(1)\,5$ (29308)

$m = 2^k$ (29314)

$h^k = H/2^k$ (29316)

$x^1 = x_0 + h^k f\,(t_0, x_0)$ (29334)

$t^i = t_0 + ih^k$ (29340)

$(i = 1\,(1)\,m,\ t^m = t_1 = t_0 + H)$

$x^{i+1} = x^{i-1} + 2h^k f(t^i, x^i)$ (29354)

$(i = 1(1)m - 1)$

$S^k(t_1, h^k) = 0.5(x_1 + x^{m-1} + h^k f(t_1, x_1))$ (29378)

$T^j = (2^{2(k-j+1)}\,S^k - S^{k-1})/(2^{2(k-j+1)} - 1)$

$(j = k(-1)\,2;\ j \geqq k).$ (29392)

Wichtige lokale Variable und Felder sind:

r	()	Wert von x_1^* nach Abschluß der Extrapolation mit h^k
s	()	Graggsche Schrittfunktion
t	()	Tableau der Romberg-Extrapolation
u	()	x_0, x^{i-1}, x^{m-1}
v	()	x^1, x^{i+1}, x^m
w	()	x^{i+1}
y	()	x_0
h7		h^k
k7		k
m7		geschätzter maximaler Fehler (29408)
v7		geschätzter minimaler Fehler (29410)
z7		m.

Der Modul EXGS ist straff gegliedert, so daß die einzelnen Funktionselemente gut erkennbar sind:

- Initialisierung (29200 bis 29254)
- Schrittweitensteuerung, Fehlertest und Fehlerbehandlung (29306 bis 29398)
- Berechnung der Schrittfunktion S, Romberg-Extrapolation (29306 bis 29398)
- Schätzung des minimalen und maximalen Fehlers, entsprechende Aktionen (29400 bis 29432).

Besondere Beachtung verdient das Modulsegment im Zeilenbereich von 29400 bis 29418. Es ermittelt näherungsweise den maximalen und den minimalen im Extrapolationsschritt aufgetretenen Fehler. Außerdem wird überprüft, ob im Romberg-Tableau Oszillationen auftreten. Ist das der Fall, wird die Extrapolation beendet. Diese Vorgehensweise dient sowohl der Genauigkeitssteigerung der Rechenergebnisse als auch der Einsparung von Rechenzeit.
Die Schrittweitensteuerung für den Modul EXGS weist gegenüber den bereits vorgestellten Versionen keine Besonderheiten auf.
Eine Erhöhung der Leistungsfähigkeit des Programms EXGS ist möglich, falls der in [Albr79] vorgestellte Algorithmus eingesetzt wird.
Bei einer von $u = 2.328 \cdot 10^{-10}$ abweichenden Maschinenkonstante sind wie auch bei den bereits vorgestellten Programmen Änderungen in bestimmten Anweisungen notwendig:

- Zeile 29282 (Faktor 0.7)
- Zeile 29406 (Faktor 0.09).

Eine andere Möglichkeit der Anpassung ist die Erhöhung der oberen Grenze für k7, die in der vorliegenden Implementation mit fünf festgelegt ist. Es ist jedoch zu beachten, daß in diesem Fall die Felder s () und t () neu dimensioniert werden müssen. Außerdem steigen die Rechenzeiten mit der Vergrößerung des Maximums von k7 exponentiell an.
Die Fehlerordnung des Verfahrens hängt von der Tableaugröße der Romberg-Extrapolation ab. Für die Größe q_g kann die folgende Beziehung angegeben werden:

$$q_g = 2(k - 1); \; k = 2,3, \ldots$$

In der vorliegenden Implementation gilt $k_{max} = 5$, somit ist die maximal erreichbare globale Konvergenzordnung $q_g = 8$.

4.3.5. Test und Einschätzung der Verfahren

Die im Abschnitt 4 vorgestellten Programme sollen nunmehr einem Test unterzogen werden.
Anhand der Testergebnisse können Hinweise zur Verfahrensauswahl abgeleitet werden. Als Testaufgabe dient eine Differentialgleichung mit wohlbekannter Lösung und geringem Aufwand bei der Auswertung der rechten Seiten. Gewählt wurde die bereits im P 4.01 GDGL.EQNS vorgestellte lineare GDGl. 2. Ordnung $\ddot{x} = -x$.
Der folgende Parametersatz gilt für alle Testrechnungen:

$f1 = 10^{-3}, 10^{-4}, \ldots, 10^{-10}$
$f2 = 0$

Anfangsschrittweite $h = 1$
Integrationsintervall $t_0, t_1 \sim 0{,}2\,\pi$

$x(1) = 0, \; x(2) = 1 \quad (t = 0)$.

Die Applikationssoftware für dieses Testproblem zeigt **Programm P 4.17 GDGL.PREC**.
Hinter den beiden **gosub**-Anweisungen sind die Zeilennummern gemäß Tafel 4.1 einzutragen. Gerechnet wurde stets mit einer 5-Byte-Gleitkommaarithmetik.

Programm P 4.17. GDGL.PREC. Genauigkeitstest der Integrationsverfahren am Beispiel der Sinusfunktion.

```
 1 n=2:m1=300000:dim x(2):f1=1e-3:f2=0:t=0:t1=2*3.141592654
 2 e1%=1:h=1:rem Verfahrensparameter setzen
 3 gosub Init
 4 data 1e-3,1e-4,1e-5,1e-6,1e-7,1e-8,1e-9,1e-10
 5 print "genauigkeitstest am beispiel der sinus-funktion"
 6 print :print
 7 for i=1 to 8:read f1:t=0:t1=2*pi:x(1)=0:x(2)=1:h=1
 8 c=ti:gosub Step:c=ti-c
 9 print "f1=";f1,"x(1)=";x(1),"x(2)=";x(2)
10 print "rechenzeit : ";c/50;" sekunden"
11 print "fkt.aufrufe: ";y7
12 print "fehlerflag : ";e1%:print:print:print
13 e1%=32:next i
14 end
```

Die Bedeutung der globalen Parameter entspricht der in der Legende zu P 4.1 GDGL.TEST. Die Zeilenzahlen für Init und Step sind aus Tafel 4.1 zu entnehmen.

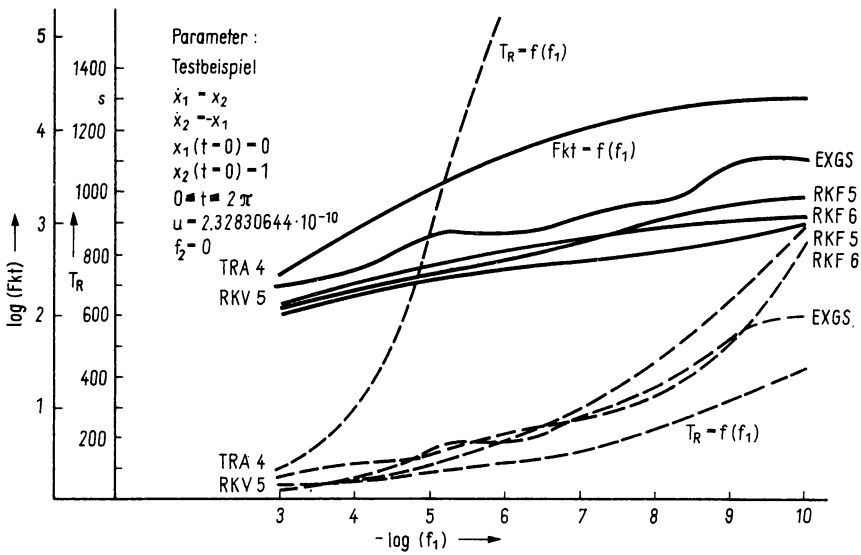


Bild 4.2. Rechenzeit und Anzahl der Funktionsaufrufe als Funktion der geforderten relativen Genauigkeit f1

Die Rechenergebnisse sind im **Bild 4.2** und in der **Tafel 4.5** dargestellt. Die Variable T_R ist die für die Lösung des Problems benötigte Rechenzeit. Die Größe Fkt gibt die Anzahl der Auswertungen der rechten Seiten an, wobei die Zählung mit y7 erfolgte.
Die in Tafel 4.5 dargestellten numerischen Werte können außer zur Einschätzung der Leistungsfähigkeit der Verfahren vom Leser ebenfalls zur Überprüfung der Richtigkeit seiner Implementation herangezogen werden. Allerdings sollte dabei beachtet werden, daß bei

von u = 2.32803644 · 10^{-10} abweichenden Maschinenkonstanten andere Ergebnisse erwartet werden müssen. Die angegebenen Werte können dann aber immer noch als Anhaltspunkt für die zu erwartenden Resultate dienen. Bild 4.2 zeigt deutlich, daß bezüglich der Rechenzeit der Modul RKV5 die günstigsten Eigenschaften aufweist. Eine geforderte relative Genauigkeit von f1 = 10^{-10} ist jedoch nicht erreichbar. Diese Genauigkeit kann lediglich durch den Modul RKF5 gewährleistet werden. Dann muß aber eine wesentlich höhere Rechenzeit in Kauf genommen werden.

Tafel 4.5. Ergebnisse des Testbeispiels Sinusfunktion (u = 2.328 · 10^{-10})

f1	x_i	TRA4	RKV5	RKF5	RKF6	EXGS
10^{-3}	x_1	$-5.4899424 \cdot 10^{-3}$	$-6.2066661 \cdot 10^{-4}$	$4.3168347 \cdot 10^{-4}$	$4.2257073 \cdot 10^{-4}$	$-8.1532794 \cdot 10^{-4}$
	x_2	1.0114202	1.00084296	1.00113965	.999740368	1.00017544
10^{-4}	x_1	$-3.4045118 \cdot 10^{-4}$	$-4.2950184 \cdot 10^{-5}$	$7.2802551 \cdot 10^{-5}$	$4.0726398 \cdot 10^{-4}$	$-7.5156603 \cdot 10^{-4}$
	x_2	1.00146632	1.0000977	1.00006481	.999953101	.999526016
10^{-5}	x_1	$-2.0008593 \cdot 10^{-5}$	$-2.4653689 \cdot 10^{-6}$	$1.0704785 \cdot 10^{-5}$	$6.6501555 \cdot 10^{-6}$	$-6.4038929 \cdot 10^{-5}$
	x_2	1.00017959	1.00000958	1.00000494	.999988719	.999958744
10^{-6}	x_1	$-1.0595134 \cdot 10^{-6}$	$-1.665748 \cdot 10^{-7}$	$1.1111947 \cdot 10^{-6}$	$7.1951808 \cdot 10^{-7}$	$-2.8574018 \cdot 10^{-5}$
	x_2	1.00002002	1.00000102	1.00000028	.999998122	.999990464
10^{-7}	x_1	$-4.8091065 \cdot 10^{-8}$	$-1.1197214 \cdot 10^{-8}$	$1.0079214 \cdot 10^{-7}$	$1.6228944 \cdot 10^{-7}$	$-3.0160291 \cdot 10^{-6}$
	x_2	1.00000213	1.00000011	1.00000002	.999999503	.999999492
10^{-8}	x_1	$-2.3518489 \cdot 10^{-8}$	$-6.0720292 \cdot 10^{-9}$	$5.7737424 \cdot 10^{-9}$	$3.8826272 \cdot 10^{-8}$	$-4.7021736 \cdot 10^{-7}$
	x_2	1.00000023	1.00000001	1.	.999999909	.999999955
10^{-9}	x_1	$-2.0441718 \cdot 10^{-8}$	$3.2887328 \cdot 10^{-9}$	$9.5745634 \cdot 10^{-10}$	$1.9606773 \cdot 10^{-8}$	$-1.1863424 \cdot 10^{-7}$
	x_2	1.00000004	1.	1.	.999999984	1.
10^{-10}	x_1	$-5.3841471 \cdot 10^{-8}$	$1.7306185 \cdot 10^{-8}$	$3.8690473 \cdot 10^{-10}$	$2.0436042 \cdot 10^{-8}$	$-6.7451415 \cdot 10^{-8}$
	x_2	1.00000002	1.	1.	.999999995	1.

Global kann eingeschätzt werden, daß bei geringen und mittleren Genauigkeitsanforderungen (f1 = 10^{-3} ... 10^{-6}) kein Verfahren eindeutig zu favorisieren ist. Lediglich der Modul TRA4 zeigt bereits bei f1 = 10^{-4} einen starken Anstieg der Rechenzeit. Er sollte deshalb nur für die Lösung von solchen Differentialgleichungsproblemen herangezogen werden, bei denen die anderen Verfahren versagen könnten. An dieser Stelle sei eine Warnung gestattet. Es ist natürlich möglich, z. B. mit dem Modul RKF5 eine steife GDGl. zu integrieren. Das numerische Ergebnis kann dann aber vom wirklichen Resultat, das ja meist unbekannt ist, erheblich abweichen. Beim Verdacht auf Steifheit sollte also mit Hilfe des Programms TRA4 eine Überprüfung der mit einem anderen Verfahren erzielten Ergebnisse vorgenommen werden. Aus Bild 4.2 ist ersichtlich, daß der Modul RKF6 die geringste Anzahl von Auswertungen der rechten Seiten zur Lösung eines bestimmten Problems benötigt. Dies konnte auch erwartet werden, da der Modul RKF6 die höchste globale Konvergenzordnung der in diesem Abschnitt vorgestellten Verfahren besitzt. Die trotzdem hohe Rechenzeit ist durch den nicht unerheblichen Implementationsaufwand bedingt. Der Modul RKF6 ist demnach besonders zur Integration von gewöhnlichen Differentialgleichungen mit umfangreichen rechten Seiten geeignet.
Zum Extrapolationsverfahren soll angemerkt werden, daß die in der Literatur [Enge85] [Albr79] hervorgehobenen positiven Eigenschaften nicht festgestellt werden konnten. Das

mag zum einen an der vorliegenden Implementation und zum anderen an der doch relativ geringen Rechengenauigkeit des genutzten BASIC-Interpreters liegen. Es ist aber nicht auszuschließen, daß der Modul EXGS in bestimmten Fällen leistungsfähiger als die anderen Verfahren sein kann.
Schlußfolgernd ist festzustellen, daß für GDGln. mit nicht zu umfangreichen rechten Seiten und geringen bis mittleren Genauigkeitsansprüchen das Verfahren RKV5 angewendet werden sollte. Bei besonders hohen Genauigkeitsansprüchen ist dem Programm RKF5 der Vorzug zu geben, während bei umfangreichen rechten Seiten das Programm RKF6 Verwendung finden sollte.
Weitere Hinweise zur Auswahl von Integrationsverfahren für bestimmte Problemklassen sind der Literatur [Enge85] [Albr79] [Phil84] zu entnehmen.

4.3.6. Anwendungsbeispiel Digital geregelter Gleichstrommotor

Das dynamische Verhalten des im **Bild 4.3** dargestellten Antriebssystems soll untersucht werden. Es besteht aus einem translatorischen Gleichstrommotor, dessen Lage x_1 durch ein inkrementales Längenmeßsystem erfaßt wird. Die vom Meßsystem gelieferte Information

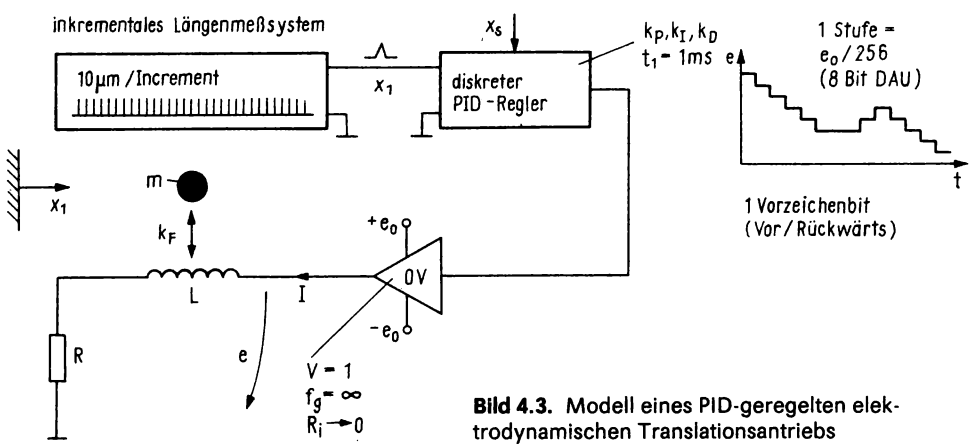


Bild 4.3. Modell eines PID-geregelten elektrodynamischen Translationsantriebs

wird von einem digitalen PID-Regler ausgewertet. Die Ansteuerung des Motors erfolgt durch einen als Stromquelle geschalteten Operationsverstärker. Zur Vereinfachung der Modellierung für dieses Demonstrationsbeispiel werden praktisch auftretende Störeinflüsse, wie

- Reibung
- Reglertotzeit
- endliche Grenzfrequenz sowie endlicher Innenwiderstand des Operationsverstärkers,

vernachlässigt. Im ersten Schritt der Modellbildung soll von einem analogen Reglermodell ausgegangen werden.
Als Modellgleichungen erhält man somit

$$m\ddot{x} = k_F I \tag{4.76}$$

$$e = RI + L\dot{I} + k_F\dot{x}_1 \tag{4.77}$$

$$x_w = x_s - x_1 \tag{4.78}$$

$$e = k_P(x_w + \bar{k}_I \int x_w dt + \bar{k}_D \cdot dx_w/dt); \tag{4.79}$$

m Ankermasse (kg)
k_F Ohmscher Widerstand des Ankers (Ω)
L Induktivität der Ankerspule (H)
k_P Reglerkonstante für den Proportionalanteil (V/m)
$\overline{k}_I$ Reglerkonstante für den Integralanteil (s^{-1})
$\overline{k}_D$ Reglerkonstante für den Differentialanteil (s)
x_s Sollwert, d. h. zu erreichende Endposition (m)
x_w Führungsgröße für den Regler (m)
e Steuergröße, in diesem Fall treibende EMK (V).

Aus Gründen der Übersichtlichkeit soll gelten:

$m = 1\,kg$
$k_F = 1\,N/A$
$R = 1\,\Omega.$

Dies entspricht durchaus praktisch vorkommenden Systemparametern. Um einer möglichen Steifheit des zu integrierenden Systems von GDGln. vorzubeugen, kann man Li vernachlässigen.
Durch diese Maßnahme erreicht man außerdem kurze Simulationszeiten.
Unter Berücksichtigung der vorgeschlagenen Vereinfachungen lauten die Gleichungen des zu simulierenden Systems in Normalform:

$$\dot{x}_1 = x_2 \tag{4.80}$$
$$\dot{x}_2 = e - x_2. \tag{4.81}$$

Die Größe x_2 ist hierbei die Geschwindigkeit des Motors. Da das zu untersuchende System einen digitalen PID-Regler enthält, müssen die analogen Reglergleichungen (4.78), (4.79) entsprechend umgeformt werden. Als Reglermodell erhält man somit folgenden Formelsatz:

$$x_w = x_s - x_1$$
$$s = s + x_w \tag{4.82}$$
$$d = x_w - x_a \tag{4.83}$$
$$e = k_P(x_w + k_I s + k_D d). \tag{4.84}$$

Die Variablen s, d, x_a sind Hilfsgrößen. Sie sind vor Beginn der Berechnung Null zu setzen.
Die Reglerkonstanten k_I und k_D werden durch die Diskretisierung dimensionslos.
Näherungsweise gilt

$$\overline{k}_I = k_I \cdot dt \approx k_I t_1$$
$$\overline{k}_D = k_D/dt \approx k_D/t_1.$$

Die Größe t_1 ist die Abtastzeit oder auch Reglertotzeit. Sie wird durch die Programmlaufzeit für den Regelalgorithmus im Mikrorechner hervorgerufen. Ein digitaler Regler kann im Gegensatz zu seinem analogen Äquivalent nur zu bestimmten diskreten Zeitpunkten $t_i = t_0 + it_1$; $i \in N$ die Werte x_1 aufnehmen und verarbeiten. Im vorliegenden Beispiel sei $t_1 = 1$ ms. Die maximal zulässige Integrationsschrittweite muß deshalb ebenfalls 1 ms betragen, um mögliche Änderungen der Steuergröße e unverzüglich im Integrationsschritt erfassen zu können.
Der Digital-Analog-Umsetzer zur Ausgabe der Steuergröße e soll eine Kanalbreite von 8 Bit aufweisen. Der aus (4.84) resultierende Wert der Steuergröße e muß deshalb mit Hilfe der Anweisung

e = int(e*256)/256

umgeformt werden. Der sich ergebende Wert wird als gültige Steuergröße deklariert, wenn $|e| < e_0$ ist. Anderenfalls wird der Steuergröße der Wert

$$e = e_0 * \text{sign}(e)$$

zugewiesen. Dies simuliert die Begrenzung der Steuergröße e und trägt somit den praktischen Gegebenheiten Rechnung.
Bei der rechentechnischen Aufbereitung der Modellgleichungen ist außerdem zu beachten, daß das inkrementale Längenmeßsystem eine Auflösung von 10 µm aufweisen soll. Durch den Integrator gelieferte Werte für x_1 sind folglich mit der Anweisung

```
x(1) = int(x(1)*1e5)/1e5
```

in das Meßraster zu bringen. **Bild 4.4** zeigt die Struktur des Simulationsprogramms. In die Subroutinenebene für die rechten Seiten ab der Zeilennummer 30000 werden nur die elektromechanischen GDGln. (4.80) und (4.81) aufgenommen. Falls die Reglergleichungen ebenfalls in diesen Bereich eingefügt werden, entstehen beträchtliche Probleme. Durch die im Integrator vorhandene Schrittweitensteuerung wird versucht, eine möglichst große Schrittweite h zu erzielen. Die dabei zwangsläufig auftretenden ungültigen Schritte liefern den Reglergleichungen sehr ungenaue Werte für x_1. Der Regelalgorithmus interpretiert das als Abweichung vom Sollwert x_s und generiert unsinnige Steuergrößen e. Dieser Effekt kann sich soweit akkumulieren, daß die sich einstellende Bewegungsrichtung der beabsichtigten Richtung entgegengesetzt ist.

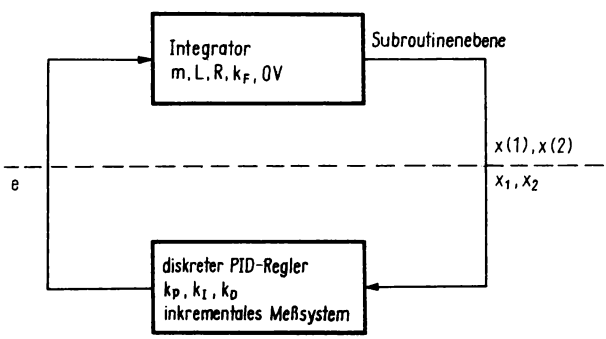


Bild 4.4. Rechentechnische Umsetzung des mathematischen Modells

In Modellen mit Regelalgorithmen ist also stets zu beachten, daß ausschließlich gültige Werte der auszuregelnden Größen verarbeitet werden dürfen. Im betrachteten Beispiel wird das durch die Anordnung des Regelalgorithmus im Hauptprogramm erreicht. Dort liegt die gesuchte Koordinate x_1 mit einer durch f1 und f2 bestimmten relativen und absoluten Genauigkeit vor, so daß Schwierigkeiten nicht zu befürchten sind.
Programm P 4.18 GDGL.EXAM ist das Hauptprogramm zur Simulation des untersuchten Antriebssystems; **Programm P 4.19 EXAM.EQNS** enthält die rechten Seiten der GDGln. (4.80), (4.81). Außer den Anweisungen zur Simulation des Antriebssystems enthält P 4.18 GDGl. EXAM noch einige für den verwendeten 6510-Rechner typische Diskettenbefehle zur Abspeicherung der durch das Simulationssystem generierten Files.
Als Integrator für die das System modellierenden GDGln. wird das sehr genaue Runge-Kutta-Fehlberg-Verfahren (Programm RKF5) herangezogen. Die Initialisierung erfolgt durch die Anweisung **gosub** 28600, während die eigentliche Integrationsroutine durch den Befehl **gosub** 28644 aufgerufen wird.

Programm P 4.18. GDGL.EXAM. Beispiel für die Anwendung der Integrationsverfahren.

```
10 rem Simulation GSM mit PID-Regelkreis
12 rem ********************************
14 dim x%(5000),v%(5000)
16 input"dateiname";dn$
18 s=0:xa=0:kp=500:ki=.01:kd=100:e0=50
20 dim x(2):n=2:xs=.1:x(1)=0:x(2)=0:t=0:e1%=1:f1=1e-3:t1=.02:m1=2000
22 gosub 28600:rem Initialisierung RKF6
24 xw=xs-x(1):e=e0*sgn(xw)
26 for t1=1e-3 to 5 step 1e-3
28 gosub 28644:if e1%<>0 then 48:rem Integrationsschritt RKF6
30 xw=xs-int(1e5*x(1))/1e5
32 s=s+xw:d=xw-xa:xa=xw:e=kp*(xw+ki*s+kd*d):e=int(256*e)/256
34 if abs(e)>e0 then e=e0*sgn(e)
36 e1%=32:t=t1
38 print int(1e3*t1),int(1e3*x(1)),int(x(2)*10)/10
40 x%(int(1e3*t1))=int(1e4*x(1))
42 v%(int(1e3*t1))=int(100*x(2))
44 get a$:if a$<>"" then gosub 52:stop
46 next t1
48 print "probleme:";e1%;"zur zeit t=";t1
50 end
52 print "diskettenlaufwerk vorbereiten"
54 get a$:if a$="" then 54
56 dn$=dn$+",s,w":open 2,8,2,dn$
58 print#2,str$(kp);",";str$(ki);",";str$(kd);","
60 for i=1 to int(1e3*t1)
62 print#2,str$(x%(i));",";str$(v%(i))
64 next i
66 close2
68 print "daten gespeichert"
70 return
```

Simuliert wird das Verhalten eines Gleichstromantriebs mit inkrementalem Positionsgeber und diskretem PID-Regler. Die Ergebnisse der Simulation werden nach entsprechender Normierung auf den Feldern x%( ) (Koordinate) und v%( ) (Geschwindigkeit) gespeichert. Die Simulationsergebnisse sind in den Bildern 4.5 bis 4.8 dargestellt.

Programm P 4.19. EXAM.EQNS. Rechte Seiten der GDGln. für das Simulationsbeispiel P 4.18 GDGL.EXAM.

```
29999 rem !-------------------------------------------------!
30000 rem !     Rechte Seiten für GSM/PID-Modellierung      !
30001 rem !-------------------------------------------------!
30097   r7=x(2)
30098   y7=y7+1
30099 return
30100   r7=e-x(2)
30199 return
30200 rem !-------------------------------------------------!
```

Die **Bilder 4.5** bis **4.8** zeigen die Ergebnisse von vier Simulationsrechnungen. Dabei wurde von den Parametern

$k_P = 500$ V/m

$k_I = 0.01$

$k_D = 100$

ausgegangen, die nacheinander variiert wurden. Aus den Integrationsergebnissen ist der Einfluß der Reglerparameter auf das dynamische Verhalten des Positioniersystems klar zu erkennen. Das günstigste dynamische Verhalten scheinen die in Bild 4.6 angegebenen Reglerparameter zu gewährleisten.

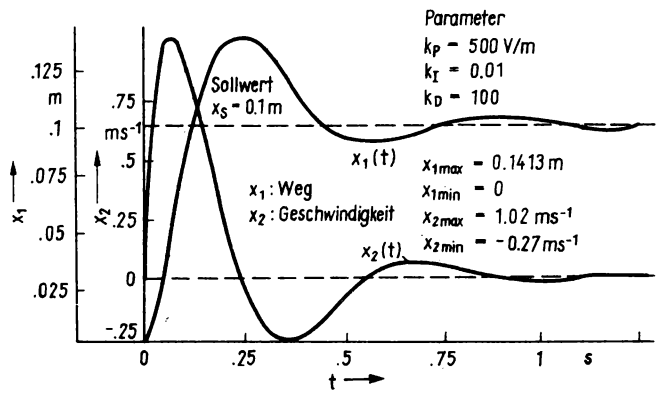


Bild 4.5. Weg und Geschwindigkeit als Funktion der Zeit und der Reglerparameter (Standardwerte)

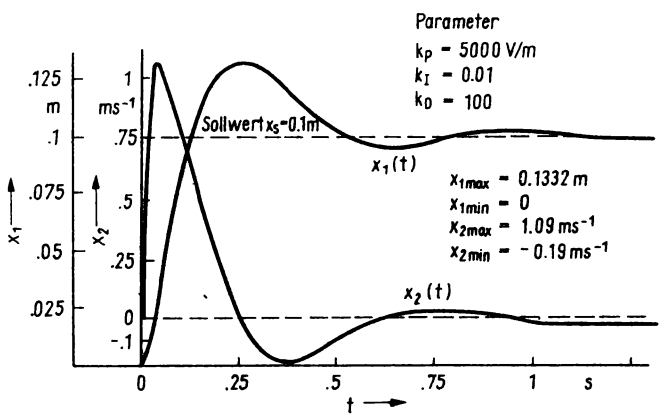


Bild 4.6. Weg und Geschwindigkeit als Funktion der Zeit und der Reglerparamter (k_p geändert)

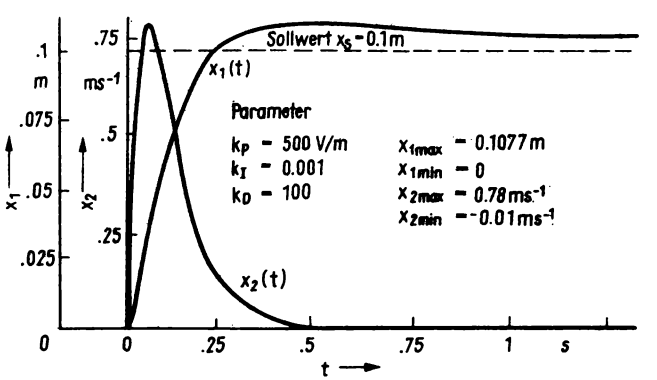


Bild 4.7. Weg und Geschwindigkeit als Funktion der Zeit und der Reglerparameter (k_I geändert)

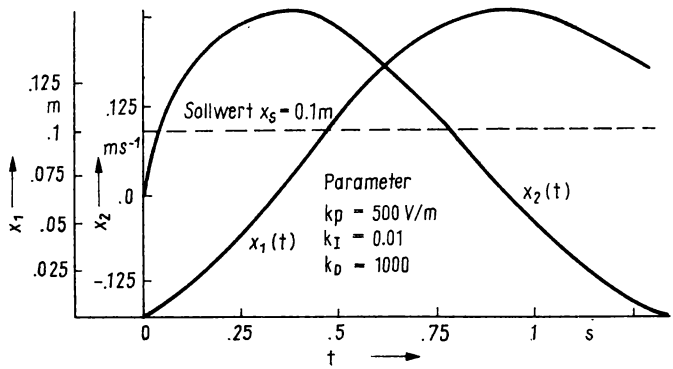


Bild 4.8. Weg und Geschwindigkeit als Funktion der Zeit und der Reglerparameter (k_D geändert)

Definitiv kann diese Frage jedoch erst nach einer Optimierungsrechnung beantwortet werden.
Zum Beispiel ist es möglich,

$$J = \int_0^T (x_s - x_1(t))^2 \, dt \Rightarrow \text{Minimum}$$

zu fordern. Aus den Stationaritätsbedingungen für das quadratische Gütefunktional erhält man dann die optimalen Werte für die Reglerparameter k_P, k_I und k_D.

4.4. Lösen von Randwertproblemen

Zur numerischen Behandlung von Randwertproblemen existieren zwei Klassen von Verfahren:

- Differenzenverfahren
- integrative Verfahren.

Bei den *Differenzenverfahren* werden die Differentialquotienten in den GDGln. durch finite Ausdrücke ersetzt. Im Fall linearer Randwertprobleme kann man mit dieser Vorgehensweise die Lösung einer GDGl. auf die Lösung eines (meist drei- oder fünfdiagonalen) linearen Gleichungssystems zurückführen (s. Abschn. 2.3.6 und 2.3.8).
Numerische Verfahren und Programme hierfür sind ausführlich im Abschnitt 2 besprochen worden.
Integrative Verfahren transformieren das zu lösende Randwertproblem in ein Anfangswertproblem.
Durch mehrfach wiederholte numerische Integration der GDGln. mit geeignet variierten Anfangswerten versucht man, die Randbedingungen zu erfüllen.
Da die Vorgehensweise dies nahelegt, spricht man in diesem Zusammenhang auch von *Schießverfahren* (shooting methods).
Einfachschießverfahren integrieren die GDGln. des Randwertproblems im Intervall $[t_0, t_1]$.
Bei *Mehrfachschießverfahren* (parallel shooting) hingegen erfolgt eine Aufspaltung des Integrationsintervalls $[t_0, t_1]$ in mehrere Teilintervalle.
Aus diesem Grund weisen Mehrfachschießverfahren auch eine höhere numerische Stabilität als Einfachschießverfahren auf. Eine schlechte Konditionierung des Randwertproblems in ei-

nem Teilintervall wirkt sich bei Mehrfachschießverfahren weit weniger ungünstig auf die Lösung des Problems aus als bei Einfachschießverfahren.
Bei der Modellierung von Prozessen in Natur und Technik treten meist nichtlineare Randwertprobleme auf.
Zur numerischen Behandlung dieser Differentialgleichungsprobleme eignen sich vor allem die integrativen Verfahren. Die im weiteren besprochenen Algorithmen sollen auf Mikrorechnern in der Programmiersprache BASIC implementiert werden.
Wegen der geringen Rechengeschwindigkeit dieser Maschinen sowie des eingeschränkten Programm- und Datenspeicherplatzes sind Einfachschießverfahren die für diese Computer am besten geeigneten Methoden zur Lösung von Randwertproblemen.

4.4.1. Herleitung des Verfahrens

Alle weiteren Betrachtungen beziehen sich auf das aus (4.3), (4.12) ableitbare Zweipunktrandwertproblem

$$x^{(n)} = f_k\,(t, x_i, \dot{x}_i, \ldots, x_i^{(n-1)})$$
$$B_j^* = (x_i\,(t_0), \dot{x}_i\,(t_0), x_i\,(t_1), \dot{x}_i\,(t_1)) = 0;$$
$$i,k = 1\,(1)\,f,\ j = (1)\,n,\ n \text{ gerade},\ n \geqq 2.$$

Weitere Voraussetzungen sind

- *Existenz* und *Unität* der Lösungen
- *Stabilität* der Lösungen im Hinblick auf Variationen der freien linken Randwerte.

Zur Lösung des Zweipunktrandwertproblems wird so vorgegangen, daß eine vektorielle Zielfunktion

$$\overline{Z}\,(x_i^{(n-1)}\,(t_0), x_i^{(n-2)}\,(t_0), \ldots, x_i^{(n-n/2)}\,(t_0); x_i\,(t_1), \dot{x}_i\,(t_1)) \tag{4.85}$$

formuliert wird, die sowohl die freien linken Randwerte als auch die zu realisierenden rechten Randwerte beinhaltet. Der Betrag dieser Zielfunktion wird genau dann Null, wenn zur Zeit $t = t_1$ die geforderten rechten Randwerte $x_i\,(t_1)$, $\dot{x}_i\,(t_1)$ erreicht werden. Die Variablen dieser Vektorfunktion sind die freien linken Randwerte $x_i^{(n-1)}(t_0), x_i^{(n-2)}(t_0), \ldots, x_i^{(n-n/2)}(t_0)$.
Auf diese Weise hat man das Zweipunktrandwertproblem auf ein Anfangswertproblem zurückgeführt.
Die Wurzeln der nichtlinearen und auch explizit nicht bekannten Vektorfunktion (4.85) werden durch ein geeignet modifiziertes Newton-Verfahren gesucht. Die für das Newton-Verfahren benötigte Matrix der partiellen Ableitungen der Zielfunktion (4.85) wird numerisch aufgebaut.
Somit kann der folgende Algorithmus zur Lösung des (i. allg. nichtlinearen) Zweipunktrandwertproblems angegeben werden:

- Vorgabe des Vektors der geforderten rechten Randwerte
$$\overline{R}^T = (x_1\,(t_1), x_2\,(t_1), \ldots, x_f\,(t_1); \dot{x}_1\,(t_1), \dot{x}_2\,(t_1), \ldots, \dot{x}_f\,(t_1)) \tag{4.86}$$

- näherungsweise Ermittlung bzw. Schätzung des Vektors der freien Anfangswerte
$$\begin{aligned}\overline{A}^T = (&x_1^{(n-n/2)}\,(t_0), x_2^{(n-n/2)}\,(t_0), \ldots, x_f^{(n-n/2)}\,(t_0);\\ &x_1^{(n-n/2+1)}\,(t_0), x_2^{(n-n/2+1)}\,(t_0),\\ &\ldots, x_f^{(n-n/2+1)}\,(t_0); \ldots;\\ &x_1^{(n-1)}\,(t_0), x_2^{(n-1)}\,(t_0), \ldots, x_f^{(n-1)}\,(t_0))\end{aligned} \tag{4.87}$$

- numerische Integration der GDGln. im Intervall $[t_0, t_1]$, man erhält den Ergebnisvektor
$$\begin{aligned}\overline{E}^T = &x_1\,(t_1), x_2\,(t_1), \ldots, x_f\,(t_1);\\ &\dot{x}_1\,(t_1), \dot{x}_2\,(t_1), \ldots, \dot{x}_f\,(t_1)\end{aligned} \tag{4.88}$$

- Berechnung der Zielfunktion

$$\overline{Z}^{(0)} = \overline{E} - \overline{R} \tag{4.89}$$

- Tastschritte

$$x^{(n-j)}(t_0) = x_k{}^{(n-j)}(t_0) + \Delta x_k{}^{(n-j)}(t_0) \tag{4.90}$$

numerische Integration von (4.83)

$$\overline{Z}_{kj} = \overline{E}_{kj} - \overline{R} \tag{4.91}$$

- Berechnung der Jacobi-Matrix

$$J = (a_{ik}) = (\delta z_i / \delta x_k{}^{(j)}(t_0)) \tag{4.92}$$

$$\approx (\Delta z_i / x_k{}^{(j)}(t_0)) \tag{4.93}$$

$i = 1(1)f\cdot(n - n/2)$
$k = 1(1)f\cdot(n - n/2)$
$j = (n - n/2)(1)(n - 1)$
n gerade, $n \geqq 2$

- Inversion der Jacobi-Matrix (Regularität vorausgesetzt)
- Berechnung des Vektors der Variationen der Anfangswerte

$$\overline{V} = J^{-1} * \overline{Z}^{(0)} \tag{4.94}$$

- Ermittlung der verbesserten Anfangswerte

$$\overline{A}_{neu} = \overline{A} - \overline{V} \tag{4.95}$$

- Abbruchtest

numerische Integration der GDGln. mit $\overline{A}_{neu}$
Ermittlung von $\overline{Z}_{neu}{}^{(0)} = \overline{E} - \overline{R}$
Falls $\| \overline{Z}_{neu}{}^{(0)} \| < \varepsilon$ gilt, wird die Iteration beendet, sonst wird der Algorithmus mit weiteren Tastschritten fortgesetzt.
Dieser Algorithmus ähnelt in einigen Details der Vorgehensweise bei der Lösung eines nichtlinearen Gleichungssystems der Form

$$f_i(x_1, x_2, \ldots, x_n) = 0;\ i = 1(1)n$$

mit dem Newton-Verfahren. Man sollte aber nicht übersehen, daß sich der vorgestellte Algorithmus in zwei grundlegenden Details vom herkömmlichen Newton-Verfahren unterscheidet:

- In den Algorithmus ist eine numerische Integration eines Systems von meist nichtlinearen GDGln. eingebettet.
- Die Matrix der partiellen Ableitungen des Systems (Jacobi-Matrix) ist nicht explizit bekannt, sondern wird sukzessive durch die mit den Tastschritten verbundenen numerischen Integrationen aufgebaut.

Falls das zu lösende Zweipunktrandwertproblem die Voraussetzungen bezüglich Existenz, Unität und Stabilität der Lösungen erfüllt und außerdem nicht zu schlecht konditioniert ist, genügen meist einige wenige Iterationen bis zur Ermittlung einer hinreichend genauen Lösung.
Es ist zu beachten, daß sämtliche Fehler bei der Lösung des Anfangswertproblems (Verfahrens- und Rundungsfehler) Eingang in die Lösung des Zweipunktrandwertproblems finden.
Als Integrator für die GDGln. sollte also ein genaues und wegen der oftmaligen Ausführung der numerischen Integration auch effektiv arbeitendes Verfahren eingesetzt werden.
Einen Beweis für die Konvergenz des Verfahrens findet der interessierte Leser in [Enge 85].

4.4.2. Programm zur Lösung von Zweipunktrandwertproblemen

Die Module des Programms BVPS *(Boundary Value Problem Solver* – Randwertproblemlöser) dienen der numerischen Realisierung des durch (4.86) bis (4.95) gegebenen Algorithmus.
Das Programm BVPS besteht aus den Modulen **(P 4.20) BVPS.INIT (P 4.21) BVPS.JMAT (P 4.22) BVPS.PROG.** Der Modul P 4.20 BVPS.INIT belegt den Zeilenbereich von 29750 bis 29804. Er dient der Initialisierung des Randwertproblemlösers.
Die globalen Eingabeparameter

- n Ordnung des Differentialgleichungssystems; $n \geqq 2$, n gerade
- n1 Norm der zulässigen Abweichung vom Ziel (vorgeschriebene rechte Randwerte)
- i1 zulässige Zahl der Iterationen
- p(n/2) absolute Variationen der freien linken Anfangswerte; $p(i) \neq 0$, $i = 1(1)n/2$

werden auf Zulässigkeit geprüft. In den Zeilen 29758 bis 29774 werden die benötigten Arrays

Programm P 4.20. BVPS.INIT. Initialisierung der Module P 4.20 BVPS.INIT und P 4.22 BVPS.PROG.

```
29749 rem !-----------------------------------------------!
29750 rem !       Boundary Value Problem Solver-Init      !
29751 rem !-----------------------------------------------!
29752   if e1%<>1 then 29802
29754   n7=n/2
29756   if n7<>int(n7) then 29802
29758   dim a(n7,n7)
29760   dim b(n7,n7)
29762   dim c(n7+1,n7)
29764   dim m(n7)
29766   dim o(n7)
29768   dim r(n7)
29770   dim t(n7,n7)
29772   dim u(n7+1,n7)
29774   dim y(n)
29776   for i7=1 to n7
29778     if p(i7)=0 then 29802
29780   next i7
29782   for i7=1 to n
29784     y(i7)=x(i7)
29786   next i7
29788   if n1<=0 then n1=1e-7
29790   if i1<=0 then i1=2
29792   t=t0
29794   h=h2
29796   gosub 28300
29798   if e1%=2 then 29804
29800   goto 29806
29802   e1%=2
29804 return
29805 rem !-----------------------------------------------!
```

Überprüft werden die globalen Parameter n,n1, und i1 sowie das Feld p(n7). Die Bedeutung dieser Variablen ist der Legende zu P 4.22 BVPS.TEST zu entnehmen.
Läuft die Initialisierung ordnungsgemäß ab, so erfolgt Rückkehr mit e1%=32, sonst mit e1%=2.
Die Routine P 4.20 BVPS.INIT ruft den Modul P 4.06 RKV5.INIT auf.Die globalen Variablen für diese Routine sind vor dem Aufruf vom P 4.20 BVPS.INIT bereitzustellen.

Programm P 4.21. BVPS.JMAT. Aufbau der Jacobi-Matrix des Randwertproblems.

```
29805 rem !-----------------------------------------------!
29806 rem !             Aufbau der Jacobi-Matrix          !
29807 rem !-----------------------------------------------!
29808   i9=0
29810   rem Eintrittspunkt für neue Iteration
29812   for k7=0 to n7
29814     rem Setzen der Anfangswerte
29816     t=t0
29818     h=h2
29820     for i7=1 to n
29822       rem Linke Randwerte für Integration setzen
29824       x(i7)=y(i7)
29826     next i7
29828     rem Integration mit alten Anfangswerten
29830     if k7=0 then 29840
29832     if n7=1 then x(2)=x(2)+p(1)
29834     if n7=1 then 29840
29836     if int(k7/2)=k7/2 then x(2*k7)=x(2*k7)+p(k7)
29838     if int(k7/2)<>k7/2 then x(2*k7+1)=x(2*k7+1)+p(k7)
29840     e1%=32
29842     gosub 28366
29844     for i7=1 to n7
29846       if int(i7/2)=i7/2 then u(k7+1,i7)=x(2*i7-2)
29848       if int(i7/2)<>i7/2 then u(k7+1,i7)=x(2*i7-1)
29850     next i7
29852     if n7=1 then u(k7+1,1)=x(1)
29854   next k7
29856   rem Tastschritte beendet
29858   rem Jacobi-Matrix aufbauen
29860   for i7=1 to n7+1
29862     for k7=1 to n7
29864       rem Abweichung vom Ziel
29866       c(i7,k7)=u(i7,k7)-w(k7)
29868     next k7
29870   next i7
29872   for k7=1 to n7
29874     for i7=1 to n7
29876       rem  Jacobi-Elemente
29878       t(k7,i7)=(c(i7+1,k7)-c(1,k7))/p(i7)
29880     next i7
29882   next k7
29883 rem !-----------------------------------------------!
```

Dies erfolgt durch Integration der GDGln. mit sukzessive veränderten Anfangswerten. Die Routine P 4.21 BVPS.JMAT ruft den Modul P 4.07 RKV5.CONT auf und generiert dazu den Initkode e1%=32.
Da P 4.07 RKV5.CONT den Modul P 4.08 RKV5.PROG aufruft, ist dieser ebenfalls einzubinden.
Die Routine P 4.21 BVPS.JMAT generiert keine Fehlerkodes und überprüft die vom P 4.07 RKV5.CONT gelieferten Fehlerkodes nicht.
Falls auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

Programm P 4.22. BVPS.PROG. Iteration neuer Anfangswerte zur Lösung des Randwertproblems.

```
29883 rem !-----------------------------------------------!
29884 rem !          Neue Anfangswerte iterieren          !
29885 rem !-----------------------------------------------!
29886   for p7=1 to n7
29888     for q7=1 to n7
29890       r(q7)=0
29892     next q7
29894     rem Inversion durch n-maliges Lösen des LGS
29896     r(p7)=1
29898     for i7=1 to n7
29900       for j7=1 to n7
29902         rem Jacobi-Matrix auf a(i7,j7)
29904         a(i7,j7)=t(i7,j7)
29906       next j7
29908     next i7
29910     rem Jordan-Algorithmus
29912     for k7=1 to n7
29914       for i7=1 to n7
29916         if i7=k7 then 29930
29918         r8=a(i7,k7)/a(k7,k7)
29920         if k7+1>n7 then 29928
29922         for j7=k7+1 to n7
29924           a(i7,j7)=a(i7,j7)-r8*a(k7,j7)
29926         next j7
29928         r(i7)=r(i7)-r8*r(k7)
29930       next i7
29932       for i7=1 to n7
29934         rem Spalte der Inversen aufgebaut
29936         o(i7)=r(i7)/a(i7,i7)
29938       next i7
29940     next k7
29942     for l7=1 to n7
29944       b(l7,p7)=o(l7)
29946     next l7
29948   next p7
29950   rem Inverse liegt vor
29952   rem Neue Anfangswerte iterieren
29954   for i7=1 to n7
29956     m(i7)=0
29958     for k7=1 to n7
29960       m(i7)=m(i7)+c(1,k7)*b(i7,k7)
29962     next k7
29964   next i7
29966   if n7=1 then y(2)=y(2)-m(1)
29968   if n7=1 then 29982
29970   n9=0
29972   for i7=1 to n7
29974     if int(i7/2)=i7/2 then y(2*i7)=y(2*i7)-m(i7)
29976     if int(i7/2)<>i7/2 then y(2*i7+1)=y(2*i7+1)-m(i7)
29978     n9=n9+c(1,i7)*c(1,i7)
29980   next i7
29982   n9=sqr(n9)
29984   rem Ende Iterationsschritt
29986   i9=i9+1
29988   if i9=1 then 29812
29990   if (i9<i1 and n9>n1) then 29812
29992 return
29993 rem !-----------------------------------------------!
```

Dazu wird die im P 4.21 BVPS.JMAT aufgebaute Jacobi-Matrix invertiert. Eine Prüfung auf Singularität der Jacobi-Matrix erfolgt nicht. Die Routine generiert keine Fehlerkodes.

Falls auf Rechnern mit $u \approx 2.328 \cdot 10^{-10}$ gerechnet wird, sind Vergleiche der Form

if x>=y **then** . . .

gegebenenfalls umzuformulieren in

if abs(x–y)>=z*u **then** . . .

mit z = max(abs(x),abs(y)).

---

dimensioniert. Vor Aufruf der Routine P 4.20 BVPS.INIT ist zu sichern, daß der für die Abarbeitung des Programms BVPS genutzte Rechner über genügend freien Hauptspeicherplatz für diese Felder verfügt. Das Programm BVPS verwendet zur numerischen Integration der GDGln. das Verfahren RKV5. Deswegen ruft die Routine P 4.20 BVPS.INIT in Zeile 29796 den Modul P 4.06 RKV5.INIT (**gosub** 28300) auf.
Vor dem Aufruf von P 4.20 BVPS.INIT müssen alle für den Modul RKV5 benötigten globalen Variablen (s. Abschnitt 4.3.2 bzw. Legende zu **Programm P 4.23 BVPS.TEST)** bereitgestellt worden sein.

Programm P 4.23. BVPS.TEST. Testbeispiel zum Randwertproblemlöser.

```
 10 rem BVPS-Applikationsprogramm
 20 n=4:dim x(4):x(1)=0:x(2)=0:x(3)=0:x(4)=1:rem Schätzung
 30 t0=0:t1=1:w(1)=1:w(2)=0:p(1)=.1:p(2)=.1
 40 i1=2
 50 n1=1e-10
 60 f1=1e-8:f2=0
 70 h1=1e-3
 80 h2=.1
 90 e1%=1:gosub 29750
100 print y(3),y(4),i9,n9,y7
110 stop



30000   r7=x(2)
30098   y7=y7+1
30099 return
30100   r7=x(3)
30199 return
30200   r7=x(4)
30299 return
30300   r7=0
30399 return
```

Gelöst wird ein Randwertproblem für die GDGl. $x^{(4)}=0$.
Globale Parameter sind

n	Ordnung des Differentialgleichungssystems (gerade!)
x(n)	Anfangswerte, Zustände
t0	Startpunkt der Integration
t1	Endpunkt der Integration
w(n/2)	geforderte rechte Randwerte
p(n/2)	absolute Variationen der Anfangswerte für die Bestimmung der Jacobi-Matrix
i1	zulässige Anzahl der Iterationen
n1	zulässige Abweichung vom Ziel
f1	relative Integrationsgenauigkeit

f2	absolute Integrationsgenauigkeit
h1	minimale zulässige Integrationsschrittweite
h2	maximale zulässige Integrationsschrittweite
y(n)	iterierte Anfangswerte für die Lösung des Randwertproblems; sie sind auf y(n/2+1), y(n/2+2), . . . , y(n) gespeichert
i9	tatsächliche Anzahl der Iterationen bis zum Abbruch
n9	Norm der tatsächlichen Abweichungen vom Ziel
z7	Anzahl der Aufrufe der rechten Seiten.

Geschieht dies nicht ordnungsgemäß, kann die Initialisierung des Programms BVPS nicht erfolgen. Die Routine P 4.20 BVPS.INIT wird in diesem Fall mit dem Fehlerkode e1 % = 2 verlassen. Besonders sensibel reagiert das in diesem Abschnitt beschriebene Verfahren zur Lösung von Zweipunktrandwertproblemen auf die Wahl der Werte für die absoluten Variationen der freien linken Randwerte p(i); i = 1(1)n/2. Sie dürfen weder zu groß noch zu klein vorgegeben werden. Anderenfalls kann das Verfahren bei schlecht konditionierten Zweipunktrandwertproblemen instabil werden.
Im Modul P 4.21 BVPS.JMAT (29806 bis 29882) wird die Jacobi-Matrix des Zweipunktrandwertproblems aufgebaut. Es soll hervorgehoben werden, daß dies für die praktisch relevante Form der Gleichungen (4.84) geschieht:

$$\begin{aligned} x_i(t_0) &= x_{i0} \\ \dot{x}_i(t_0) &= x_{i0} \\ x_i(t_1) &= x_{i1} \\ \dot{x}_i(t_1) &= x_{i1}; i = 1(1)f \end{aligned} \tag{4.96}$$

In den Zeilen 29830 bis 29854 werden die durch (4.90), (4.91) definierten Tastschritte ausgeführt.
Der numerische Aufbau der Jacobi-Matrix erfolgt in den Zeilen 29860 bis 29882 entsprechend der durch (4.93) formulierten Berechnungsvorschrift.
Nach Abarbeitung der Routine ist auf dem Feld t(k7,i) die Jacobi-Matrix des behandelten Zweipunktrandwertproblems gespeichert.
Die im Modul P 4.21 BVPS.JMAT aufgebaute Matrix der partiellen Ableitungen von (4.85) wird im Programmsegment P 4.22 BVPS.PROG zur Iteration verbesserter freier linker Randwerte herangezogen. Die Inversion der Jacobi-Matrix erfolgt in den Zeilen 29896 bis 29948 durch n-maliges Anwenden des *Gauß-Jordan-Verfahrens* (s. Abschn. 2. 3. 1).
In den Anweisungen der Zeilen 29954 bis 29980 werden die verbesserten freien linken Anfangswerte gemäß der durch (4.94), (4.95) gegebenen Vorschrift berechnet.
In den Zeilen 29982 bis 29990 erfolgt der Abbruchtest. Es wird geprüft, ob die zulässige Zahl der Iterationen (i1) erreicht ist oder ob die wirkliche Norm der Abweichungen von den vorgegebenen rechten Randwerten (n9) kleiner als eine vorgegebene zulässige Abweichungsnorm (n1) ist.
Die Bedeutung der im Programm BVPS verwendeten globalen Variablen kann der Legende zu P 4.23 BVPS.TEST entnommen werden.

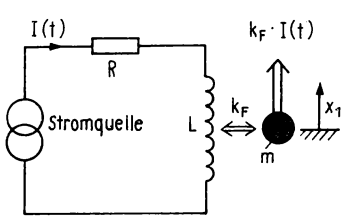


Bild 4.9. Modell eines einfachen Geleichstromantriebs

4. 4. 3. Beispiel

Zur Veranschaulichung der Wirkungsweise des Programms BVPS soll die energieoptimale Steuerung des im **Bild 4. 9** dargestellten einfachen translatorischen Gleichstromantriebs untersucht werden.
Die Steuerungsaufgabe kann wie folgt mathematisch formuliert werden:

$$J = \int_0^T P_{vel}(t)dt \Rightarrow \text{Minimum}$$
$$x_1(t=0)=0 \qquad x_1(t=T)=x_{1E} \tag{4.97}$$
$$\dot{x}_1(t=0)=0 \qquad \dot{x}_1(t=T)=\dot{x}_{1E}$$
$$m\ddot{x}_1 = k_F I(t)$$
$$E = RI + L\dot{I} + k_F\ddot{x}_1.$$

Vernachlässigt man in der Maschengleichung die Induktionsanteile $L\dot{I}$ und $k_F\dot{x}_1$, kann man unter Beachtung von $P_{vel} = RI^2$ schreiben:

$$J = k\int_0^T \ddot{x}^2{}_1 dt \Rightarrow \text{Minimum;} \tag{4.98}$$
$$k = Rm^2/Tk_F^2.$$

Beschränkungen für $\ddot{x}_1$ sollen vorerst unbeachtet bleiben. Somit kann man zur Bestimmung des Extremums des Funktionals (4.98) die klassische Variationsrechnung heranziehen. Die Euler-Gleichung

$$\delta F/\delta x - (\delta F/\delta \dot{x})^{\cdot} + (\delta F/\delta \ddot{x})^{\cdot\cdot} = 0 \tag{4.99}$$

liefert auf (4.98) angewendet

$$x^{(4)} = 0. \tag{4.100}$$

Die Lösung des Variationsproblems (4.97) ist auf die Lösung des Zweipunktrandwertproblems

$$x^{(4)} = 0$$
$$x_1(0)=0 \qquad x_1(T)=x_{1E} \tag{4.101}$$
$$\dot{x}_1(0)=0 \qquad \dot{x}_1(T)=\dot{x}_{1E}$$

zurückgeführt worden. Dieses einfache Problem kann analytisch gelöst werden. Die Extremalen des Variationsproblems sind kubische Parabeln:

$$x_1(t) = C_1t^3 + C_2t^2 + C_3t + C_4. \tag{4.102}$$

Die vier Konstanten in (4.102) können durch die vier Randbedingungen bestimmt werden. Für die numerischen Rechnungen mit dem Programm BVPS müssen die formalen Parameter geeignet spezifiziert werden. Für dieses Testbeispiel soll gelten:

$$x_1(T) = 1\ \text{m}, \qquad T = 1\ \text{s}$$
$$\dot{x}_1(T) = 0$$
$$m = 1\ \text{kg}$$
$$R = 1\ \Omega$$
$$k_F = 1\ \text{N/A}.$$

Die Lösung des Zweipunktrandwertproblems (4.101) wird durch den im Abschnitt 4.4.1 vorgestellten Algorithmus auf die Lösung eines Anfangswertproblems zurückgeführt. Im betrachteten Beispiel sind die freien linken Randwerte $\ddot{x}_1(0)$ und $\dddot{x}_1(0)$.
Die (bei diesem Beispiel mögliche) analytische Bestimmung dieser beiden Größen aus (4.102) und den vier Randbedingungen ergibt:

$\ddot{x}_1(0) = 6\ m/s^2$

$\dddot{x}_1(0) = -12\ m/s^3$.

Das aus diesen Werten resultierende Systemverhalten ist im **Bild 4. 10** dargestellt.

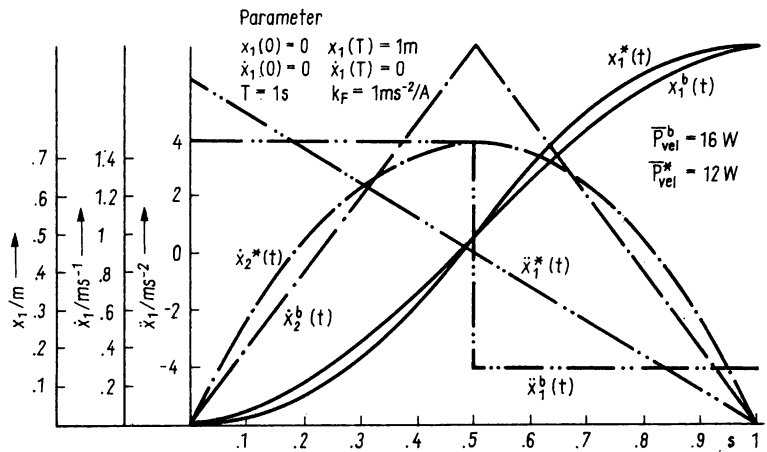


Bild 4.10. Vergleich zwischen zeit- und energieoptimaler Steuerung

Zum Vergleich wurde eine die gleichen Randwerte realisierende Steuerung mit konstanter Beschleunigung und Bremsung eingezeichnet.
Diese Steuerung setzt 16 W Verlustleistung (P_{vel}) im Motor um, während die energieoptimale Steuerung den Wandler lediglich mit 12 W belastet.
P 4.23 BVPS.TEST zeigt die zur Lösung dieses Zweipunktrandwertproblems erforderliche Applikationssoftware. In Zeile 20 werden sowohl die Anfangswerte gesetzt (x(1) = 0, x(2) = 0) als auch die Schätzungen für die freien linken Randwerte auf die entsprechenden globalen Variablen gespeichert (x(3) = 0, x(4) = 1). Man kann erkennen, daß selbst recht grobe Schätzungen für die freien linken Randwerte möglich sind.
In Zeile 30 erfolgt das Setzen der geforderten rechten Randwerte durch die Anweisungen

w(1) = 1 ($x_1(T) = 1$)
w(2) = 0 ($\dot{x}_1(T) = 0$).

Die absoluten Variationen der freien linken Randwerte werden jeweils zu 0.1 festgelegt:

p(1) = 0.1 ($\Delta\ddot{x}_1(0)$)
p(2) = 0.1 ($\Delta\dddot{x}_1(0)$).

Die Anzahl der Iterationen (i1) wird auf zwei festgesetzt; die zulässige Norm der Abweichungen von den geforderten rechten Randwerten (n1) soll kleiner als 10^{-10} sein. Alle anderen initialisierten Größen (t0, t1, f1, f2, h1, h2) sind für die ordnungsgemäße Arbeit des Programms RKV5 erforderlich.
In Zeile 90 wird das Fehlerflag e1 % = 1 gesetzt und der Modul P 4.20 BVPS.INIT aufgerufen, der seinerseits die weiteren Module der Programme BVPS und RKV5 aufruft.
In Zeile 100 werden die Ergebnisse der Berechnungen ausgedruckt:

- $\ddot{x}_1(0)$ (y(3))
- $\dddot{x}_1(0)$ (y(4))
- Anzahl der Iterationen bis zum Abbruch (i9)

- Norm der Abweichungen von den geforderten rechten Randwerten bei Abbruch der Berechnungen (n9)
- Anzahl der beim letzten Aufruf von RKV5 benötigten Auswertungen der rechten Seiten der GDGln. (y7)

Ab Zeile 30 000 muß die GDGl. (4.100) in Normalform eingegeben werden. Die Verfahrensweise hierfür ist ausführlich in den Abschnitten 4.1.1 sowie 4.3 beschrieben worden.
Nach zwei Iterationen liefert das Programm BVPS die folgenden Ergebnisse:

$\ddot{x}_1(0) = 6$ m/s^2 (y(3))

$\dddot{x}_1(0) = -12$ m/s^3 (y(4))

$n9 = 1.95854724 \cdot 10^{-8}$

$y7 = 24.$

Die für die Lösung des Problems mit einem 6510-Prozessor (Taktfrequenz 1 MHz) benötigte Rechenzeit betrug 81.36 s. Die geforderten rechten Randwerte wurden im Rahmen der Rechengenauigkeit der verwendeten Maschine exakt approximiert:

$x_1(T) = 0.999999999$ m (x(1), $(1 - x(1))/u = 4$)

$\dot{x}_1(T) = -2.77844568 \cdot 10^{-9}$ m/s. (x(2)).

Da die Beendigung des Programms BVPS stets in der Routine P 4.21 BVPS.JMAT erfolgt, kann man die angegebenen Werte $x_1(T)$, $\dot{x}_1(T)$ (x(1), x(2)) nur durch nochmalige Integration der GDGl. (4.100) mit den durch P 4.21 BVPS.JMAT vor dem Abbruch iterierten freien linken Randwerten y(3), y(4) ($\ddot{x}_1(0)$, $\dddot{x}_1(0)$) gewinnen. Falls dies gewünscht wird, sind im P 4.23 BVPS.TEST noch die folgenden Anweisungen aufzunehmen:

```
110 for i = 1 to 4
120 x(i) = y(i): rem Setzen der Anfangswerte
130 next i
140 e1 % = 32: t = 0: h = h1
150 gosub 28366: rem Integration mit RKV5
160 print x(1), x(2), e1%
170 stop
```

Es ist zu erkennen, daß das Newton-Verfahren rasch konvergiert und die freien linken Randwerte $\ddot{x}_1(0)$, $\dddot{x}_1(0)$ überraschend genau approximiert.
Die schnelle Konvergenz des Verfahrens konnte durch Testrechnungen zur Lösung verschiedener auch nichtlinearer Zweipunktrandwertprobleme bestätigt werden.

Anhang 1. Verwendeter Zeichensatz

Zeichen	ASCII-Kode	Bedeutung
NUL	Ø	leeres Zeichen; nicht zu verwechseln mit SPACE
BEL	7	akustisches Signal
TAB	9	Tabulator
RETURN	13	Wagenrücklauf; wird in Kombination mit Zeilenvorschub ausgeführt
SPACE	32	Zwischenraum; hat in BASIC außer in Strings und Kommentaren keine Bedeutung; zur besonderen Kennzeichnung in Texten durch ␣ dargestellt
	34	Anführungszeichen (quotation mark); dient als Begrenzer (delimiter) für Stringkonstanten
$	36	Dollarzeichen; wird bei manchen Geräten auch als Escape (¤) geschrieben; dient zur Kennzeichnung von Stringkonstanten, -variablen und -funktionen
%	37	Prozentzeichen; dient zur Kennzeichnung von Integerkonstanten und -variablen; hat bei dieser Verwendung nichts mit der üblichen Bedeutung von Prozent zu tun
'	39	Apostroph (single quotation mark); Funktion wie Anführungszeichen
(	40	Klammer auf; in Funktionen und arithmetischen Ausdrücken verwendet
)	41	Klammer zu; in Funktionen und arithmetischen Ausdrücken verwendet
*	42	Multiplikationsoperator
+	43	Additionsoperator/Operator zur Stringverknüpfung/Vorzeichen
	44	Komma; Trennzeichen in Printanweisungen Nicht als Dezimalzeichen verwendbar!
–	45	Subtraktionsoperator/Vorzeichen
.	46	Dezimalpunkt
/	47	Divisionsoperator
Ø	48	Ziffer
1	49	Ziffer
2	50	Ziffer
3	51	Ziffer
4	52	Ziffer
5	53	Ziffer
6	54	Ziffer
7	55	Ziffer
8	56	Ziffer
9	57	Ziffer
:	58	Doppelpunkt; Trennzeichen für Mehrfachanweisungen; wird nachfolgend nur für Signaturzwecke verwendet

Zeichen	ASCII-Kode	Bedeutung
;	59	Semikolon; Trennzeichen in Printanweisungen
<	60	Vergleichsoperator „kleiner als"
=	61	Vergleichsoperator „gleich"; auch als Zuweisungsoperator benutzt
>	62	Vergleichsoperator „größer als"
A	65	Buchstabe
B	66	Buchstabe
C	67	Buchstabe
D	68	Buchstabe
E	69	Buchstabe
F	70	Buchstabe
G	71	Buchstabe
H	72	Buchstabe
I	73	Buchstabe
J	74	Buchstabe
K	75	Buchstabe
L	76	Buchstabe
M	77	Buchstabe
N	78	Buchstabe
O	79	Buchstabe
P	80	Buchstabe
Q	81	Buchstabe
R	82	Buchstabe
S	83	Buchstabe
T	84	Buchstabe
U	85	Buchstabe
V	86	Buchstabe
W	87	Buchstabe
X	88	Buchstabe
Y	89	Buchstabe
Z	90	Buchstabe
^	94	Potenzierungsoperator

Anhang 2. BASIC-Anweisungen und -Funktionen

Im Anhang 2 ist das Verzeichnis der in diesem Buch verwendeten BASIC-Anweisungen und -Funktionen erläutert.
In den *geschweiften* Klammern *muß* vom Benutzer ein entsprechender Inhalt angegeben sein. In den *eckigen* Klammern *kann* ein entsprechender Inhalt angegeben sein. Der Inhalt der *doppelten eckigen* Klammern kann bis zur maximalen Zeilenlänge beliebig wiederholt angegeben sein. Die Angaben zur Implementierung beziehen sich auf Standards oder Quasi-Standards. "Standard" bedeutet Minimal-BASIC nach ISO [ISO-84]. "Tiny" bedeutet Tiny-BASIC ("winziges BASIC") entsprechend der Angabe in *Klein* [Klei82]. "BASICODE" ist der Quasi-Standard für Rundfunkaussendungen von BASIC-Programmen [Wieg84].

---

abs( )

Liefert den Absolutbetrag des Arguments.

Syntax:

abs({ num. konstante / num. variable / num. ausdruck })

Typ: numerische Funktion

Implementierung: Basicode, Standard, Tiny

Beispiel

```
100 y = abs(3*sin(z + 1))
```

Fehlermöglichkeiten:

- String im Argument

---

asc( )

Wandelt einen aus einem Zeichen bestehenden String in sein ASCII-Äquivalent um (s. Zeichentabelle Anhang 1). Das Ergebnis von **asc( )** ist eine Integerzahl.

Syntax:

asc({ stringkonstante / stringvariable / stringausdruck })

Typ: Stringfunktion mit numerischem Ergebnis

Implementierung: Basicode

Beispiel

```
100 x = asc("a")
110 b1$ = "("
120 y1 = asc(b1$)
```

Fehlermöglichkeiten:

- Argument enthält mehr als ein Zeichen oder ist ein Leerstring.
- Manche Rechner verwenden abweichende ASCII-Kodezahlen, z. B. auch für die Unterscheidung von Groß- und Kleinbuchstaben.

---

atn( )

Liefert den Arcustangens im Bogenmaß. Das Argument darf den gesamten zulässigen Zahlenbereich überstreichen. Bei extrem großen oder kleinen Argumenten kann sich die Rechenzeit beträchtlich verlängern. Das Ergebnis von **atn( )** liegt im Bereich von $-$**pi**/2 bis $+$**pi**/2.
Die Mehrwertigkeit der Funktion wird von atn( ) nicht berücksichtigt. Ebenso gibt es in BASIC keine dem ATAN2 von FORTRAN entsprechende Funktion, die die Lage in einem der vier Quadranten berücksichtigt; man muß eine solche Funktion ggf. programmieren.

Syntax:

atn(num. konstante | num. variable | num. ausdruck)

Typ: numerische Funktion

Implementierung: Basicode, Standard

Beispiel

100 y=**atn**(3*__*__**sqr**(z+1))

Fehlermöglichkeiten:

- Argumentfehler (String oder Überlauf)

---

chr$( )

Umkehrung von **asc( )**. Erzeugt aus einer ASCII-Kodenummer einen aus einem Zeichen bestehenden String. Die eingegebene Zahl wird von einigen Rechnern modulo 128 gedeutet, d. h., bei Werten größer als 128 wird ein geeignetes Vielfaches von 128 subtrahiert. Wir setzen indessen voraus, daß der Argumentwert zwischen 0 und 127 liegt. Mit **chr$( )** können auch nicht druckbare oder mit der Tastatur nicht erzeugbare Zeichen generiert werden. Viele Rechner können mit **chr$**(7) zur Abgabe eines akustischen Signals veranlaßt werden.

Syntax:

chr$(stringkonstante | stringvariable | stringausdruck)

Typ: Stringfunktion mit Stringergebnis

Implementierung: Basicode

Beispiel

100 a3$ = **chr$**(66)
a3$ wird der Buchstabe B zugewiesen.

Fehlermöglichkeiten:

- Argument ist negativ oder ein String.
- Die meisten Rechner bilden automatisch den Integerwert, wenn das Argument Dezimalteile enthält; andere reklamieren in solchen Fällen jedoch Fehler.
- Manche Rechner verwenden abweichende Kodezahlen.

cos( )

Liefert den Kosinus des im Bogenmaß anzugebenden Arguments. Das Argument darf den gesamten zulässigen Zahlenbereich überstreichen. Bei extrem großen oder kleinen Argumenten kann sich die Rechenzeit beträchtlich verlängern.

Syntax:

cos(num. konstante | num. variable | num. ausdruck)

Typ: numerische Funktion

Implementierung: Basicode, Standard

Beispiel

```
100 y = cos(3 * sqr(z + 1))
```

Fehlermöglichkeiten:

- String im Argument

data,[read],[restore]

Die Anweisungen erlauben die Bildung und Übernahme von Datenblöcken bei laufendem Programm. Mit **read** werden in fortlaufender Reihenfolge die in den Datenblöcken stehenden Konstanten übernommen. Zu Programmbeginn steht der Zeiger (pointer), der auf das aktuelle Element im **data**-Block zeigt, auf dem ersten Element. Nach jedem **read** wird der Zeiger um eine Einheit weitergesetzt. Es können beliebige Datentypen verarbeitet werden; die von **read** geforderten und die von **data** gelieferten Konstanten müssen jedoch bezüglich des Typs übereinstimmen. Alle im Programm beliebig verstreuten **data**-Anweisungen bilden einen einzigen Datenblock; es kann also auch von **data**-Zeilen gelesen werden, die eine höhere Zeilennummer als die **read**-Anweisung haben. Indessen bleibt ein Programm übersichtlicher, wenn alle **data**-Zeilen am Anfang oder am Ende des Programms konzentriert werden. Bei Erreichen der **restore**-Anweisung wird der Zeiger wieder auf das erste Element des Datenblocks gesetzt.

Syntax:

read { variablenname 1 } [[,variablenname n]]
data { konstante 1 } [[,konstante n]]
restore

Typ: Anweisung (**read**), nicht ausführbare Anweisung (**data**)

Implementierung: Basicode, Standard

Beispiel

```
10 data "abc","xy"
20 data 100.1,200.7
:
60 read a$,b$,c1,d
70 data 3.1415
80 restore
90 read x8$,y8$,x,y,z
:
```

Fehlermöglichkeiten:

- BASIC-Anweisungen in der **data**-Zeile, z. B. Doppelpunkt oder **rem**

dim . . .

Mit **dim** . . . wird dem Programm die Größe des für die Feldvariablen zu reservierenden Speicherplatzes bekanntgegeben. Es können Felder für alle Variablentypen dimensioniert werden. Obwohl bei den meisten Rechnern die Dimensionierung unabhängig von der Position der **dim**-Anweisung sofort bei Programmbeginn realisiert wird, sollten **dim**-Anweisungen zum Zwecke besserer Übersichtlichkeit in einem Block am Programmanfang stehen. Wir setzen voraus, daß nach einer **dim**-Anweisung mehrere Variablennamen mit zugehörigen Indizes folgen können (s. letztes Beispiel). Die Zulässigkeit dieser Vereinfachung ist ggf. zu prüfen. – Die Dimensionszählung beginnt bei Null. **dim** a(10) reserviert demnach elf Speicherplätze. – Viele Rechner dimensionieren beim ersten Auftreten einer Feldvariablen automatisch ein Feld bis zum Index 10. Damit wird aber bei mehrdimensionalen Feldern u. U. unnötig Speicherplatz verbraucht. Wir werden daher, auch in Übereinstimmung mit Basicode, alle Felder unabhängig von ihrer Größe explizit dimensionieren.

Syntax:

dim { variablenname } ({ konstante 1 } [,konstante 2])

Typ: Anweisung

Implementierung: Basicode, Standard

Beispiel

```
100 dim x(100)
100 dim s%(5)
100 dim t$(10,12)
100 dim y(100),t%(5),u$(10,12)
```

Fehlermöglichkeiten:

- Es wurde versucht, ein Feld zu redimensionieren, d. h., zwei **dim**-Anweisungen für die gleiche Variable zu schreiben.
- Die Feldgröße wurde durch Variable oder Ausdrücke angegeben.
- Beim Programmlauf hat einer der Indizes den dimensionierten Bereich überschritten oder wurde negativ.
- Manche Dialekte vertragen es nicht, daß der gleiche Variablenname für ein- und zweidimensionale Felder verwendet wird.

end

Beendet den Programmablauf und schließt alle offenen Kanäle. Steht am physikalischen Ende des Programms, bildet also die letzte Programmzeile. Die Verwendung ist bei vielen Rechnern optional. Wenn man **end** nicht verwendet, achte man aber auf den Öffnungszustand der Kanäle. Wir legen fest, daß immer die Programmzeile 63999 die **end**-Anweisung enthält. Unterprogramme, die auf das Hauptprogramm folgen, sind durch **goto** 63999 zu überbrücken, damit der Programmablauf nicht unbeabsichtigt in die Unterprogramme gerät.

Syntax:

end

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
63999 end
```

exp( )

Liefert e zur Potenz des Arguments. Bei sehr großen oder sehr kleinen Argumentwerten kann sich die Rechenzeit beträchtlich verlängern.

Syntax:

exp({num. konstante / num. variable / num. ausdruck}**)**

Typ: numerische Funktion

Implementierung: Standard

Beispiel

100 y = **exp**(3 * **sqr**(z + 1))

Fehlermöglichkeiten:

- String im Argument
- Das Argument liegt außerhalb des Bereichs $-88.7228391 < \text{arg} < 88.0296919$; damit wird der Zahlenbereich überschritten. Überschreitung wird als fataler Fehler angesehen, d. h., es erfolgt Programmabbruch.

for . . ., to . . .,[step . . .], next . . .

Schleifen (loops) erlauben die Wiederholung von Programmteilen. Die als Schleifenzähler fungierende Variable (Schleifensteuerungszahl, Schleifenindex, loop index) hat bei vielen Dialekten unbedingt eine Gleitpunktvariable zu sein, auch wenn Anfangs-, End- und Schrittweitenwerte ganzzahlig sind.

Der Ablauf geschieht in folgender Weise: Der Schleifensteuerungszahl wird der angegebene Anfangswert zugewiesen, und dann wird getestet, ob der angegebene Endwert bereits überschritten ist. Ist das nicht der Fall, so wird die Schleife bis zur **next**-Anweisung ausgeführt; danach wird zur **for**-Anweisung zurückgesprungen. Die Schleifensteuerungszahl wird um den Wert der angegebenen Schrittweite erhöht, und es erfolgt wieder der Test auf Überschreitung des Endwerts. Ggf. wird die Schleife erneut durchlaufen. Ist die Schleifensteuerungszahl indessen größer als der angegebene Endwert, so wird zu der auf die **next**-Anweisung folgenden Zeile verzweigt. Die Ausdrücke 1 bis 3 können auch gebrochene Zahlen sein. Nach regulärem Verlassen der Schleife behält die Schleifenvariable den Wert, mit dem die Schleife das letztemal ausgeführt worden ist. Der im Anschluß an die Ausführung vergrößerte und zu groß befundene Wert der Schleifenvariablen wird beim Verlassen der Schleife um den Betrag der Schrittweite wieder verringert.

for und **next** können nur in Kombination verwendet werden und müssen den gleichen Variablennamen tragen. Der Schleifenzähler darf in unserem Subset innerhalb der Schleife nicht verändert werden. Es ist aber zulässig, daß die Schleife vor Erreichen der **next**-Anweisung durch eine Sprunganweisung verlassen wird. Wenn der Wert von Ausdruck 2 von vornherein kleiner als der Wert des Ausdrucks 1 ist, dann wird i. allg. angenommen, daß die gesamte Schleife beim Programmdurchlauf übersprungen wird, obwohl manche Versionen die Schleifen in jedem Falle mindestens einmal durchlaufen.

Der Inhalt einer Schleife darf nur über die **for**-Anweisung erreicht werden; es ist nicht zulässig, mit einer Sprunganweisung in die Schleife hineinzuspringen. Jede Schleife kann in ihrem Innern weitere Schleifen enthalten (geschachtelte Schleifen, nested loops).

Um Schleifen besser überschaubar zu schreiben, wird im folgenden der Inhalt jeder Schleife um zwei Zeichen eingerückt. Die Einrückung ist nicht funktionswichtig.

Wenn die Anweisung **step** fehlt, dann wird Schrittweite 1 angenommen. Beim Durchlaufen

im Sinne von großen zu kleinen Werten ist indessen die Angabe einer negativen Schrittweite notwendig, z. B. **step** −1. Hinter **next** muß ein Variablenname folgen; die Zusammenfassung mehrerer Variablennamen oder das Weglassen der Variablennamen, was bei einigen Rechnern möglich ist, wollen wir wegen der mangelnden Übersichtlichkeit vermeiden.

Syntax:

for {gleitpunktvar.} = {num.konst. / num.var. / num.ausdr.} **to** {num.konst. / num.var. / num.ausdr.} [**step** {num.konst. / num.var. / num.ausdr.}]
next {gleitpunktvariablenname}

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
100 for j = 30 to 10 step -1
110   print  j
120 next j
100 for j = a to sqr(a + 3) step (x^2 + y^2)
110  . . .
120 next j
100 for k1 = 2.2 to 3.1 step .25
110  . . .
120 next k1
100 for a = 1 to 3   geschachtelte Schleife
110   for b = 1 to 12
120     print a * b
130   next b
140 next a
```

Fehlermöglichkeiten:

- Es wurde mit einer Sprunganweisung in eine Schleife gesprungen; der Programmablauf hat das **for** nicht passiert und findet ein nicht erwartetes **next**.
- Bei **for** und bei **next** stehen verschiedene Variablennamen.
- Der Schleifenzähler wurde innerhalb der Schleife verändert (wird nicht von allen Dialekten als Fehler angesehen).
- Geschachtelte Schleifen überschneiden sich, z. B.

```
100 for a = 1 to 3
110 for b = 11 to 12
120 print a * b
130 next a
140 next b
```

- Bei nicht ganzzahligen Schrittgrößen kann infolge von Rechenungenauigkeiten die Schleife vorzeitig verlassen werden. Der Fehler tritt rechnerabhängig auf. Folgendes Beispiel verdeutlicht die Verhältnisse:

```
100 for x = 0 to 0.34 step 0.17
110   print x
120 next x
```

Manche Rechner drucken 0 und 0.17; der Wert 0.34 wird nicht erreicht. Nach Durchlauf der Schleife mit dem Wert 0.17 wurde die Schrittweite 0.17 zum Schleifenzähler addiert, und es wurde festgestellt, daß der nun erreichte Wert um den Betrag der Rechenunge-

nauigkeit größer als 0.34 geworden wäre, und deshalb wurde die Schleife beendet. Man vermeidet den Fehler, indem man die Schrittweite ganzzahlig wählt und den gewünschten Wert erst in der Schleife berechnet oder indem man Ausdruck 2 geringfügig größer als den theoretischen Endwert setzt.

- Manche Dialekte verhalten sich nicht abweisend (z. B. Commodore BASIC 2.0). Auch wenn der Anfangswert größer als der Endwert ist, wird die Schleife erst einmal abgearbeitet. Man testet z. B. durch

```
100 for i = 2 to 1
110   print "Schleife nicht abweisend"
120 next i
```

Solches Verhalten führt in vielen Fällen dazu, daß in der Literatur angegebene Programme nicht korrekt laufen. Man muß dann vor Schleifen, in denen solche Verhältnisse auftreten können, eine bedingte Sprunganweisung schreiben, z. B.

```
100 if a > b goto 200
110 for i = a to b
    . . . .
190 next i
200 . . . .
```

Die Programme dieses Buches sind i. allg. gegen nicht abweisende Schleifensteuerung gesichert. Sofern sich dadurch zusätzliche Programmzeilen der o. a. Form ergeben haben, so sind diese mit ungeradzahligen Zeilennummern versehen. Bei abweisendem Verhalten des gegebenen Rechners können solche Zeilen entfallen.

---

gosub . . ., return

Ein Unterprogramm (subroutine) ist ein Anweisungsblock, der bestimmte Operationen ausführt und nach Abarbeitung die Programmablaufsteuerung an die Stelle im Hauptprogramm zurückverweist, die der Anweisung zum Unterprogrammsprung folgt. Es ist auf diese Weise möglich, Prozeduren mehrfach aufzurufen; man kann Standardprozeduren als vorgefertigte Einheiten in viele verschiedene Programme einbauen, und man kann durch Gliederung der Programme in abgeschlossene Prozeduren die Gesamtprogramme übersichtlicher gestalten.
Der Unterprogrammsprung wird mit **gosub** {zeilennummer} angewiesen. Die angegebene Zeilennummer ist die erste Zeile des angesprungenen Unterprogramms. Eine Kopfzeile für das Unterprogramm ist nicht erforderlich; aus Gründen der Übersichtlichkeit empfiehlt sich aber eine entsprechende Kommentarzeile. Das Unterprogramm muß durch **return** beendet werden; fehlendes **return** ist ein fataler Fehler.
Aus Unterprogrammen heraus können weitere Unterprogramme aufgerufen werden. Die Anzahl der möglichen Hierarchieebenen (levels) ist rechnerabhängig begrenzt; sie wird i. allg. > 6 sein. Ggf. ist mit der folgenden Routine zu testen. Das letzte angezeigte n vor der Fehlermeldung ist die Anzahl der zulässigen Hierarchieebenen. Allerdings sind Versionen denkbar, wenn auch bisher nicht beobachtet, bei denen die zulässige Anzahl der Hierarchieebenen vom Belegungszustand anderer Speicherbereiche abhängt.

```
100 gosub 200
200 n = n + 1
210 print n; "Ebenen durchlaufen"
220 goto 100
```

Da Unterprogramme funktionell keine Kopfzeilen haben, kann man an beliebigen Einsprungstellen (entry points) in sie hineinspringen. Mit dem Ziel strukturierter Programme (s. u.) wollen wir diese Möglichkeit nicht ausnutzen. Ebenso sollten Unterprogramme nur durch eine einzige **return**-Anweisung verlassen werden.
Im Gegensatz zu **goto** wird bei Verwendung von **gosub** die Stelle des Programms gespei-

chert, von der aus gesprungen wurde. Das Programm muß also beim Rücksprung nicht wieder mit großem Zeitaufwand vollständig nach der Fortsetzungszeile durchsucht werden.

Syntax:

gosub {zeilennummer}
. . .
return

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
100 x = 2.5
110 gosub 1000
120 print y
. . .
999 goto 63999
1000 rem UP SINUS
1010 y = sin(180 * x/pi)
1030 return
. . .
63999 end
```

Fehlermöglichkeiten:

- Es ist ein wesentlicher Nachteil einfacher BASIC-Dialekte, daß nur globale Variable verwendet werden, also Variable, die von allen Teilen des Gesamtprogramms verändert werden können. Es muß daher darauf geachtet werden, daß keinesfalls im Hauptprogramm verwendete Variable unbeabsichtigt in den Unterprogrammen verändert werden. Wir benutzen daher in den Unterprogrammen nur Variablennamen, deren zweites Zeichen 7, 8 oder 9 ist.
- Wenn die Unterprogramme auf das Hauptprogramm folgen, dann muß verhindert werden, daß das Hauptprogramm nach Beendigung seiner Arbeit unbeabsichtigt in die Unterprogramme hineinläuft. Bei Sprachversionen, die die **end**-Anweisung nur am physischen Programmende akzeptieren, muß zwischen Hauptprogramm und Unterprogrammen ein unbedingter Sprung zum Programmende stehen.
- In der **gosub**-Anweisung sind keine Variablennamen zulässig. **gosub** a ist also in unserem Subset ein Fehler.

---

go to oder **goto**

Unbedingter Sprung (unconditional transfer or jump) zu der im Anschluß angegebenen Zeilennummer. Diese Zeilennummer kann höher oder niedriger als die aktuelle sein.

Syntax:

goto {zeilennummer}

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
100 goto 475
```

Fehlermöglichkeiten:

- Die Zielzeile existiert nicht.
- Durch Rücksprünge wurde eine unendliche Schleife (infinite loop) erzeugt, die der Rech-

ner nicht mehr von selbst beenden kann. Durch regelmäßigen Ausdruck geeigneter Zwischenergebnisse oder entsprechende Kontrolle im Programm muß man bei unübersichtlichen Programmen versuchen, solche Loops rechtzeitig zu entdecken.
- Unbedingte Sprünge sind besonders geeignet, Programme unübersichtlich zu machen. Man sollte sie wo möglich vermeiden.
- In der **goto**-Anweisung dürfen keine Variablennamen verwendet werden (s. **gosub**).

---

if . . . then . . .
if . . . goto . . .

Bedingter Sprung (conditional transfer). Zwischen **if** und **then** bzw. **goto** steht ein logischer Ausdruck. Wenn der logische Ausdruck das Ergebnis "wahr" ergibt, dann erfolgt eine Verzweigung (branching) zu der nach **goto** angegebenen Zeilennummer, oder es wird die nach **then** stehende Anweisung ausgeführt. Ist das Ergebnis des logischen Ausdrucks "falsch", dann wird mit der folgenden Programmzeilennummer fortgefahren. Der logische Ausdruck kann ein numerischer oder ein Stringvergleich sein. Im ursprünglichen BASIC gab es nur die bedingte Sprunganweisung **if . . . then . . .**, und darauf konnte dann **goto** zusätzlich folgen. Diese Anweisungsfolge wird auch von allen modernen Versionen verstanden, wir verwenden sie aber mit dem Ziel kürzerer Listings nicht. Man teste sein BASIC daraufhin und ändere ggf. die Programme entsprechend ab. – Die in erweiterten Versionen zulässige ELSE-Anweisung wird explizit hier nicht verwendet.

Syntax:

if {logischer ausdruck} **goto** {zeilennummer} / **then** {anweisung}

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
100 if w$="Sprung" then x=x+1
120 if x %>3 goto 2000
130 if sqr(y+1)<3 goto 100
```

Fehlermöglichkeiten:

- Die angegebene Zeilennummer existiert nicht.
- Wenn gegen das Ergebnis eines arithmetischen Ausdrucks verglichen wird, dann ist wegen der begrenzten Rechengenauigkeit ein Toleranzbereich vorzugeben; andernfalls wird u. U. die Bedingung praktisch nie erfüllt.
  Statt

  if sin(x) = **pi then** . . .

  ist richtig zu schreiben

  if abs(sin(x) − **pi)** < 1e − 6 **then** . . .

---

input

---

input erlaubt Dateneingabe nach Beginn des Programmlaufs. Das Programm hält an und meldet sich mit Fragezeichen. Einer **input**-Anweisung kann eine im Rahmen der zulässigen Zeilenlänge beliebige lange Liste von zu übernehmenden Daten folgen. Das Programm läuft erst weiter, wenn die in der Anweisung angegebene Anzahl von Konstanten eingegeben wurde. Bei Eingabe von zu vielen Daten wird der Überschuß ignoriert; u. U. erscheint die Fehlermeldung "extra ignored". Der Fehler ist nicht fatal. Die Variablennamen in der Anwei-

sungszeile sind durch Komma zu trennen. Bei der Dateneingabe werden Komma und RETURN als Trennzeichen akzeptiert. Bei der Eingabe von Strings sind Anführungszeichen i. allg. erforderlich. Bei vielen Rechnern werden sich Probleme ergeben, wenn ein Komma als Teil des Strings eingegeben werden soll.
Es ist Ausdruck guten Programmierstils, wenn vor einer **input**-Anweisung ein kurzer Text gedruckt wird, der den Bedienenden auf Art, Format usw. der erwarteten Dateneingabe hinweist. Wenn die Daten ohne Zeilenschaltung nach dem erläuternden Text eingegeben werden sollen, dann ist der Text mit Semikolon abzuschließen. Der Text sollte am Ende kein Fragezeichen enthalten; das erzeugt der Interpreter ohnehin.

Syntax:

input {variablenname 1} [[,variablenname n]]

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
10 input x
20 input z, a %, fj
10 print "Geburtsdatum Tag, Monat, Jahr   dd,mm,yy";
20 input d1,m1,y1
RUN
Geburtsdatum Tag, Monat, Jahr   dd,mm,yy ? 24,07,75
```

Fehlermöglichkeiten:

- Eingegebene Daten stimmen bezüglich Anzahl und Typ nicht mit den Angaben in der Variablenliste überein.
- Das letzte Zeichen der Anweisung war ein Komma.
- Ein zu einem eingegebenen String gehörendes Komma wurde als Trennzeichen erkannt.

---

int( )

Liefert den ganzzahligen Anteil des Arguments. Es wird nicht gerundet, sondern der Nachkommateil wird abgeschnitten.
Verschiedene BASIC-Versionen verhalten sich bei der Integer-Bildung von negativen Zahlen unterschiedlich, indem entweder die nächstkleinere oder die nächstgrößere negative Zahl ermittelt wird. Als Normalfall wird angesehen, daß **int**(x) immer Werte liefert, die kleiner als x sind, also **int**(−2.2) = −3.
Soll auf s Stellen richtig gerundet werden, so verwendet man die Anweisung

y = (**int**(x * 10^s + .5))/10^s

Syntax:

int({ num. konstante / num. variable / num. ausdruck })

Typ: numerische Funktion

Implementierung: Basicode, Standard, Tiny

Beispiel:

```
100 y = int(3 * sin(z + 1))
```

Fehlermöglichkeiten:

- String im Argument

left$( )

Isoliert aus einem String eine Anzahl von Zeichen, die durch eine Integerkonstante im Argument der Funktion anzugeben ist. Die Zeichen werden beginnend beim Stringanfang isoliert. Im allgemeinen ist es erforderlich, daß mindestens ein Zeichen aus dem String in den neuen String übernommen wird. Ist die Anzahl der geforderten Zeichen größer als die Stringlänge, so wird der gesamte String übernommen und bis zur angegebenen Länge mit Leerzeichen (NUL) aufgefüllt; jedoch verhalten sich verschiedene Rechner diesbezüglich unterschiedlich. Mit der folgenden Routine testet man, ob in derartigen Fällen Fehler entstehen und ob der ausgebenene String bis zur vollen Länge aufgefüllt wird:

```
100 a$ = left$("string",10)
200 print len(a$)
300 for i = 1 to 10
400    print asc(mid$(a$,i,1))
500 next i
```

Syntax:

left$({ stringkonstante / stringvariable / stringausdruck } , { num. konstante / num. variable / num. ausdruck })

Typ: Stringfunktion mit Stringergebnis

Implementierung: Basicode

Beispiel

b$ = **left$**("stringfunktion",6)
b$ wird "string" zugewiesen

Fehlermöglichkeiten:

- Argumentfehler

len( )

Liefert die Länge (Zeichenzahl) eines Strings, wobei Leerzeichen (spaces) mitgezählt werden. Das Ergebnis von **len( )** ist eine Integerzahl.

Syntax:

len({ stringkonstante / stringvariable / stringausdruck })

Typ: Stringfunktion mit numerischem Ergebnis

Implementierung: Basicode, Standard, Tiny

Beispiel

100 x = **len**("ABCD")
x wird der Wert 4 zugewiesen

Fehlermöglichkeiten:

- Argumentfehler

let

let weist einer Variablen einen Wert zu, der durch den rechts vom Gleichheitszeichen ste-

henden Ausdruck gegeben ist, z. B. **let** a = 47 oder **let** a = a + 1. Der mit dem Variablennamen a gekennzeichnete Speicherplatz wird mit dem Wert des rechts vom Gleichheitszeichen stehenden Ausdrucks beschrieben. Die Verwendung von **let** ist bei den meisten Rechnern optional; wir verwenden es deshalb nicht und schreiben einfach a = 47 oder a = a + 1. Die Zulässigkeit dieser Vereinfachung ist ggf. zu prüfen. Das Gleichheitszeichen hat damit die Rolle eines Zuweisungsoperators, nicht die übliche Bedeutung des Vergleichsoperators. Die Zuweisung kann für alle Variablen- und Konstantenarten angewendet werden.

Syntax:

let {variablenname} = $\begin{Bmatrix} \text{konstante} \\ \text{variable} \\ \text{ausdruck} \end{Bmatrix}$

{variablenname} = $\begin{Bmatrix} \text{konstante} \\ \text{variable} \\ \text{ausdruck} \end{Bmatrix}$

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
let a1 = 22
a3 = a3 + 1
b% = 17
c9$ = "XYZ"
```

Fehlermöglichkeiten:

- Einer Stringvariablen wurde eine numerische Konstante zugewiesen oder umgekehrt.
- Fehler im Ausdruck, z. B. Zahlenbereichsüberschreitung.
- Links vom Gleichheitszeichen steht eine Konstante oder ein Ausdruck.
- Der Variablenname beginnt nicht mit einem Buchstaben.

---

log( )

Liefert den logarithmus naturalis des Arguments. Das Argument darf den gesamten Bereich vom Rechner verarbeitbarer positiver Zahlen überstreichen, also $2^{-128} < \text{arg} < 2^{127}$. Bei extrem großen oder kleinen Argumenten kann sich die Rechenzeit beträchtlich verlängern. Der dekadische Logarithmus wird erhalten durch

```
y = log(x)/log(10)
```

Syntax:

log($\begin{Bmatrix} \text{num. konstante} \\ \text{num. variable} \\ \text{num. ausdruck} \end{Bmatrix}$)

Typ: numerische Funktion

Implementierung: Basicode, Standard

Beispiel

```
100 y = log(3 * sqr(z + 1))
```

Fehlermöglichkeiten:

- String im Argument
- negatives oder verschwindendes Argument.

mid$( )

Isoliert eine anzugebende Anzahl von Zeichen aus einem String. Im Argument von **mid$** haben zuerst der Stringname, dann die Position des ersten Zeichens, das isoliert werden soll, und schließlich die Anzahl zu isolierender Zeichen zu stehen.

Syntax:

mid$({stringkonstante / stringvariable / stringausdruck} , {num. konstante / num. variable / num. ausdruck} , {num. konstante / num. variable / num. ausdruck})

Typ: Stringfunktion mit Stringergebnis

Implementierung: Basicode

Beispiel

```
c$ = mid$("vor" + d$,i + 1,y * 2)
a$ = "STRINGFUNKTION"
b$ = mid$(a$,3,8)
```

b$ wird "RINGFUNK" zugewiesen.

Fehlermöglichkeiten:

- Eines der numerischen Argumente ist negativ.
- Die numerischen Argumente sind größer, als es für den angegebenen String sinnvoll ist. Das Verhalten ist analog zur Prüfung bei **left$** zu testen.

on . . . gosub . . .

Die Anweisung bewirkt den Sprung zu einem von mehreren Unterprogrammen. Durch eine ganzzahlige Variable oder einen Ausdruck wird angegeben, zu welchem einer Anzahl von nach **gosub** aufgelisteter Unterprogramme gesprungen werden soll. Hat die Variable bzw. der Ausdruck den Wert 1, so wird zum ersten angegebenen Unterprogramm gesprungen, hat er den Wert 2, so erfolgt der Sprung zum zweiten Unterprogramm usw. Die Anzahl möglicher Unterprogramme ist i. allg. nur durch die mögliche Programmzeilenlänge begrenzt.

Syntax:

on { num. variable / num. ausdruck } **gosub** {zeilennr.1} [[,zeilennr.n]]

Typ: Anweisung

Implementierung: Basicode

Beispiel

```
on x + 1 gosub 1000, 1200, 1250, 1700, 320
```

Fehlermöglichkeiten:

- Nach **gosub** steht ein Variablenname (in manchen Versionen zulässig).
- Der numerische Wert ist kleiner als 1 oder größer als die Anzahl aufgelisteter Unterprogramme. Der letzte Fall wird von einigen Versionen als fatal betrachtet; andere fahren mit der auf **on . . . gosub . . .** folgenden Programmzeile fort.
- Ein in der auf **gosub** folgenden Liste angegebenes Unterprogramm ist nicht vorhanden.

on . . . goto

Wie **on . . . gosub . . .**, es werden jedoch statt Unterprogrammsprüngen Verzweigungen zu den in der Liste aufgeführten Programmzeilennummern ausgeführt.

next . . .

Siehe unter **for . . .**

pi

Liefert die Zahl pi = 3.14159 26535 89793 23846 . . . Die Funkion ist ggf. durch eine Zuweisung p1 = 3.14159 . . . oder p1 = 4* **atn** (1) zu realisieren.

Syntax:

pi

Typ: numerische Funktion

Implementierung: –

Beispiel

```
100 y = cos(x*pi/180)
```

print

Mit **print** können Konstanten auf dem Bildschirm ausgegeben werden. **print** wird i. allg. von einer oder mehreren Konstanten, Variablen oder Ausdrücken gefolgt. Ausgabe auf anderen Peripheriegeräten wird ebenfalls mit **print** und zusätzlichen Vorschriften entsprechend den jeweiligen Rechnern realisiert. Die in diesem Buch angegebenen Programme sehen nur **print** vor und müssen deshalb ggf. modifiziert werden.
Wird als Trennzeichen zwischen zwei auszugebenden Konstanten in der **print**-Anweisung ein Semikolon verwendet, so wird kein zusätzliches Space ausgegeben; die Konstanten werden unmittelbar nacheinander ausgegeben. Scheinbar überschüssige Spaces sind frei gelassene Plätze für eventuelle Vorzeichen. Wird als Trennzeichen ein Komma verwendet, so bewirkt dieses Komma das Vorrücken bis zum Beginn der folgenden Tabulatorposition oder Druckzone. Diese Druckzonen sind auf Werte modulo m eingestellt. m hat rechnerspezifisch Werte zwischen 6 und 16. Nach Ablauf des folgenden Programms hat man einen Überblick über die aktuellen Verhältnisse:

```
10 for i = 0 to 7
20   print chr$(48+i)+"  "; :rem 9 SPACES
30 next i
40 for i = 0 to 7
50   print "0123456789";
60 next i
70 print
80 print "!","!","!","!"

RUN
0         1         2         3         4
01234567890123456789012345678901234567890123 . . .
!         !         !         !         !  . . .
```

Die Schleifengrenzen in Zeilen 10 und 40 sind herabzusetzen, falls der Bildschirm nicht die Breite von 80 Zeichen verarbeiten kann.
Der Abschluß einer Zeile ohne Trennzeichen (Komma oder Semikolon) bewirkt Zeilenvorschub (line feed) bei der Ausgabe. Eine Leerzeile (blank line) wird durch Angabe einer **print**-Anweisung ohne Ausgabeliste erzeugt.
Ohne weitere Spezifikationen gibt BASIC Zahlen in einem der Größe der Zahlen von Fall zu

Fall angepaßten Format aus; ggf. wird auf E-Format umgeschaltet. Die folgende Routine führt die Verhältnisse beim gegebenen Rechner vor:

```
10 for i = 1 to 20
20    print i,2^(-i),2^i
30 next i
```

Syntax:

print { konstante 1 / variable 1 / ausdruck 1 } [{ ; / , } { konstante n / variable n / ausdruck n }] [{ ␣ / ; / , }]

Semikola und Kommata können gemischt in den Listen verwendet werden.

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
10 print 187.15; 189.33
20 print 187, 189, 195
30 print 187,,189,
40 print 195
50 print
60 print "STRING"
RUN
187.15 189.33
187            189            195
187                           189            195
STRING
```

Fehlermöglichkeiten:

- Andere Trennzeichen als ; und ,
- Überschreitung der Bildschirmbreite.
- In einer Schleife standen mit Komma oder Semikolon abgeschlossene **print**-Anweisungen, und nach Ende der Schleife wurde kein Rücklauf in Form eines einzelnen **print** gegeben.

---

read

Siehe unter **data.**

---

rem

Alle nach **rem** eingefügten Zeichen sind bis auf einige rechnerspezifische Ausnahmen Kommentar und haben auf den Programmablauf keinen Einfluß. Manche Rechner lassen als Kurzzeichen für **rem** auch das Ausrufzeichen, andere den Apostroph zu. Wegen der besseren typografischen Erscheinung werden wir weitgehend das "!" verwenden; ggf. hat man es durch **"rem"** zu ersetzen.

Syntax:

rem [kommentartext]

Typ: nicht ausführbare Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
1000 rem  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1010 rem  UNTERPROGRAMM POLYNOMBERECHNUNG
1020 rem  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1030 rem
1040 . . .
1000 ! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1010 !    UNTERPROGRAMM POLYNOMBERECHNUNG
1020 ! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1030 !
1040 . . .
```

Fehlermöglichkeiten:

- Manche Rechner, die Mehrfachanweisungszeilen akzeptieren, betrachten Kommentartext nach einem Trennzeichen für solche Mehrfachanweisungen als ausführbare Anweisung, was i. allg. unerwünscht ist. Andere Rechner vertragen nicht den gesamten Zeichensatz im Kommentar und eliminieren z. B. nach runden Klammern alle Spaces.

---

restore

Siehe unter **data**.

---

return

Siehe unter **gosub**.

---

right$( )

Isoliert aus einem angegebenen String von rechts beginnend Zeichen, deren Anzahl im zweiten Argument gegeben ist. Die übrigen Eigenschaften von **right$** entsprechen denen von **left$.**

---

sgn( )

Liefert das Vorzeichen des Arguments in der Form +1, 0 oder −1.

Syntax:

sgn({ num. konstante / num. variable / num. ausdruck })

Typ: numerische Funktion

Implementierung: Basicode

Beispiel

```
100 y=sgn(sin(x))
```

Fehlermöglichkeiten:

- Argumentfehler

---

sin( )

Liefert den Sinus des im Bogenmaß anzugebenden Arguments. Das Argument darf den ge-

samten zulässigen Zahlenbereich überstreichen. Bei extrem großen oder kleinen Argumenten kann sich die Rechenzeit beträchtlich verlängern.

Syntax:

sin(num. konstante | num. variable | num. ausdruck)

Typ: numerische Funktion

Implementierung: Basicode, Standard

Beispiel:

```
100 y=sin(3*sqr(z+1))
```

Fehlermöglichkeiten:

- String im Argument

---

sqr( )

Liefert die Quadratwurzel des Arguments. Bei extrem großen oder kleinen Argumenten kann sich die Rechenzeit beträchtlich verlängern. Negative Argumente führen bei manchen Rechnern zu nichtfatalen Fehlern (Ergebnis wird Null gesetzt); bei anderen werden sie als fatal angesehen. Wir setzen den zweiten Fall voraus.

Syntax:

sqr (num. konstante | num. variable | num. ausdruck)

Typ: numerische Funktion

Implementierung: Basicode, Standard

Beispiel

```
100 y=cos(3*sqr(z+1))
```

Fehlermöglichkeiten:

- String im Argument
- Argument negativ

---

step . . .

Siehe **for . . .**

---

stop

Beendet wie **end** den Programmablauf, kann aber an beliebiger Stelle im Programm stehen und ist für die Programmtestung vorteilhaft verwendbar. Der Programmablauf kann durch **goto** {nächste Zeilennummer} (im Direktmodus) fortgesetzt werden; manche Rechner haben dafür den Befehl CONT (continue, fortsetzen). **stop** schließt keine Kanäle und verändert keine Variableninhalte. Die Anweisung BREAK mancher Dialekte ist **stop** äquivalent.

Syntax:

stop

Typ: Anweisung

Implementierung: Basicode, Tiny

Beispiel

```
1070 stop
RUN
GOTO 1080
```

str$( )

Wandelt eine Zahl in einen String um. Die Funktion ist für die formatierte Zahlenausgabe nützlich.

Syntax:

str$(num. konstante / num. variable / num. ausdruck **)**

Typ: Stringfunktion mit Stringergebnis

Implementierung: –

Beispiel

```
100 a$ = str$(17.33)
```

Fehlermöglichkeiten:

- Argumentfehler

tab( )

tab( ) entspricht der Tabulatorfunktion bei einer Schreibmaschine. **tab( )** in Verbindung mit einer **print**-Anweisung bewirkt Vorrücken auf die durch den Ausdruck gegebene Spalte (column). Die Ausdrücke in der **tab**-Anweisung sind als absolute Werte zu verstehen; sie beziehen sich auf ein festes Zeichenraster auf dem Bildschirm, das mit Null beginnt. Es sind nur Verschiebungen nach rechts möglich. Es ist nicht gemeint, daß von der aktuellen Schreibposition um die im Ausdruck angegebene Zahl von Positionen weitergerückt werden soll. Ist die aktuelle Position größer als die mit **tab( )** geforderte, so werden alle weiteren **tab( )** ignoriert. Als Trennzeichen für **tab( )** in **print**-Anweisungen sollten Semikola bevorzugt werden. Gebrochene Argumente werden als Integer-Werte behandelt. **tab**(0);**print**"x" führt dazu, daß x auf die erste Position in der Druckzeile gedruckt wird.

Syntax:

tab(konstante / variable / ausdruck **)**

Typ: Anweisung

Implementierung: Basicode, Standard, Tiny

Beispiel

```
10 for i = 0 to 7
20    print tab(10 * i);right$(str$(i),1);
30 next i
40 print
50 for i = 0 to 7
60    print "0123456789";
70 next i
80 print
90 print 188;tab(10);"Stueck";tab(25);66.31;tab(38);"Kilogramm"
```

Fehlermöglichkeiten:

- Aktuelle Position ist größer als **tab**-Wert.
- Negativer **tab**-Wert.
- Kommata in der **print**-Anweisung bewirken u. U. Vorrücken auf Positionen, die größer als der **tab**-Wert sind.

tan( )

Liefert den Tangens des im Bogenmaß anzugebenden Arguments. Die übrigen Eigenschaften entsprechen denen des Sinus.

then . . .

Siehe **if . . .**

ti

Liefert die Zeit in Sekunden. Die Zählung beginnt mit dem Einschalten des Rechners. Manche Rechner liefern die Zeit als Stringfunktion in der Form hh:mm:ss. In diesem Fall ist z. B. für Zeitbedarfsmessungen ein kurzes Umwandlungsprogramm erforderlich. Ist der erste Zeitpunkt mit t1$ = **ti$** und der zweite Zeitpunkt mit t2$ = **ti$** zugeordnet worden, so ermittelt sich die Zeitdifferenz in Sekunden folgendermaßen:

```
1000 t3 = 3600 * val(left$(t2$,2)) + 60 * val(mid$(t2$,3,2)) + val(right$(t2$,2))
1002 t3 = t3 - 3600 * val(left$(t1$,2)) - 60 * val(mid$(t1$,3,2)) - val(right$(t1$,2))
1004 if t3 > 0 goto 1010
1006 t3 = t3 + 86400
1008 goto 1004
1010 return
```

Syntax:

ti kann wie eine Variable aufgerufen werden.

Typ: Funktion ohne Argument

Implementierung: –

Beispiel

```
100 print ti
100 t1 = ti
110 gosub 2000
120 print ti - t1
```

to . . .

siehe **for . . .**

val( )

Wandelt einen String in sein numerisches Äquivalent um. Der String darf nur die Ziffern 0 . . . 9, den Buchstaben E, die Zeichen + und – sowie den Dezimalpunkt enthalten, und es muß sich um die Stringdarstellung einer Zahl handeln. Die Funktion ist für die formatierte Zahlenausgabe nützlich. Das Verhalten beim Auftreten anderer als der angegebenen Zeichen im String ist rechnerspezifisch verschieden und ggf. zu testen.

Syntax:

val({ stringkonstante / stringvariable / stringausdruck })

Typ: Stringfunktion mit numerischem Ergebnis

Implementierung: Basicode

Beispiel

```
100 x = val("+ 0.2345E - 03")
```

Fehlermöglichkeiten:

- Argumentfehler

Anhang 3. Zeitbedarf für die hauptsächlichen Rechenoperationen

Es wird der Zeitbedarf für die einmalige Ausführung elementarer BASIC-Funktionen und -Anweisungen auf verschiedenen Rechnern angegeben. Die im Tabellenkopf dargestellten Programmzeilen wurden für die Messung in das Programm P 1.04 ZEITMS eingefügt. Die Zeitangaben bedeuten Millisekunden. Die Byteangaben beziehen sich auf die Mantissenlänge.

Rechner	1018 print "Test";	1018 print "Test"	1018 rem Test	1018 goto 1019	1018 gosub 1039, 1039 return	1018 for i = 1 to 0 : next i	1018 a = 1	1018 a = −1.23456789e−12	1018 a = b [a = 1.23 b = 3.21]	1018 c = a + b	1018 c = a * b	1018 c = a/b
E60 BASIC-11, 3 Byte	10.0	31.0	.4	.5	1.0	4.1	1.8	1.2	1.0	2.4	3.1	3.2
E60 BASIC-11, 7 Byte	10.0	31.0	.4	.5	.9	5.2	1.7	1.3	1.0	2.6	8.0	8.0
SM4 BASIC-11, 3 Byte	9.0	96.0	.4	.4	.2	2.6	1.0	.8	.4	1.6	.8	1.2
Robot.1715, BASI, 3 Byte	8.8	25.8	2.3	2.4	3.0	3.2	3.4	3.9	3.5	4.9	6.3	7.7
KC-85/3, BASIC, 3 Byte	72.0	572.0	4.5	4.0	5.5	9.5	5.0	112.0	5.0	7.0	9.0	10.5
IBM-PC, BASIC-A, 3 Byte	8.6	27.3	.4	.5	1.2	4.1	1.4	1.6	1.3	2.4	2.6	3.5
IBM-PC, BASIC-A, 7 Byte	8.6	27.3	.4	.5	1.2	4.1	1.1	1.5	1.3	3.1	4.4	20.4
IBM-PC, BASCOM-Compiler	6.6	23.9	0	0	0	.3	.1	.1	.1	.4	.5	.6
Apple II [1]	4.0	21.0	.4	1.0	2.0	1.0	2.0	64.0	1.4	2.7	4.5	4.5
Commodore-Amiga 500 [3]	11.4	92.2	.8	.7	1.0	2.3	1.1	1.1	.9	1.2	1.2	1.3
Commodore 64, 4 Byte	5.9	37.0	.4	.7	1.6	6.6	2.0	71.3	1.4	2.7	4.9	5.2
HP-85, 5 Byte [5]	14.9	81.5	.4	.4	1.2	3.8	1.1	1.4	1.4	2.9	3.8	8.7
Tandy-200 [1]	10.0	32.0	.5	1.0	2.0	3.0	2.4	5.3	1.6	3.5	7.5	23.0
ZX Spektrum +	9.9	61.7	.6	2.8	4.9	9.1	2.4	3.1	2.9	5.2	6.4	6.8

1) Angaben von *Feichtinger* [Feic85,2]
2) seg$(a$,1,3)
3) Wortlänge 3 Byte. Interlace on.
4) Bei sehr großen und u. U. auch bei sehr kleinen Argumenten ergeben sich mit manchen Interpretern (z. B. BASIC-11) extrem verlängerte Rechenzeiten
5) 12 Stellen. Im short-mode (5 Stellen) praktisch identische Ergebnisse

Rechner	1018 c = sin(a)[4]	1018 c = log(a)[4]	1018 c = atn(a)[4]	1018 c = sqr(a)[4]	1018 c = a^b	1018 if a > b goto 1019	1018 a$ = b$ [a$ = "abcde" b$ = "fghij"]	1018 c$ = a$ + b$	1018 c$ = mid$(a$,3,1)
E60 BASIC-11, 3 Byte	13.0	13.4	16.8	3.2	23.6	2.5	1.9	4.9	9.9[2]
E60 BASIC-11, 7 Byte	80.8	69.2	93.3	11.3	138.	2.6	1.9	4.9	9.9[2]
SM4 BASIC-11, 3 Byte	2.2	2.2	1.8	1.4	2.8	.6	.8	1.6	3.0
Robot.1715, BASI, 3 Byte	19.7	22.0	20.5	43.0	42.5	4.6	.6	.9	1.1
KC-85/3, BASIC, 3 Byte	28.5	26.0	37.0	53.5	73.0	7.0	5.2	7.0	11.0
IBM-PC, BASIC-A, 3 Byte	18.4	8.7	9.7	8.3	15.7	2.4	1.4	2.8	3.5
IBM-PC, BASIC-A, 7 Byte	18.8	9.1	9.9	8.7	16.2	2.6	1.4	2.8	3.5
IBM-PC, BASCOM-Compiler	3.3	4.2	4.1	1.1	9.0	0.2	.6	.7	.7
Apple II [1]	26.0	21.0	41.0		47.0	2.5	1.2	2.7	5.5
Commodore-Amiga 500 [3]	3.3	4.6	3.5	1.3	7.7	1.3	1.0	2.1	2.8
Commodore 64, 4 Byte	28.6	23.4	45.5		52.6	2.5	1.3	2.8	5.9
HP-85, 5 Byte [5]	46.3	32.1	25.7	8.7	55.6	2.7	2.1	4.0	2.9
Tandy-200 [1]	110.	180.	175.		320.	3.0	1.2	3.0	4.5
ZX Spektrum +	45.0	72.3	68.0		116.	4.4	5.3	8.4	

Anhang 4. Genauigkeits- und Zeitbedarfsvergleich für verschiedene Rechner

Es wird der Genauigkeits- und Zeitbedarfsvergleich für verschiedene Rechner und Programmiersprachen nach *Savage* [Kels87] angegeben.
In einer Zählschleife wird eine Variable von 1 bis 2500 hochgezählt. Diese Variable wird jeweils den Operationen Quadrieren, Logarithmusbildung und Arcustangensbildung unterworfen, und diese Operationen werden sofort durch Radizieren, Exponentialfunktion und Tangensbildung wieder rückgängig gemacht.
Die Fehler bei den Funktionsbildungen pflanzen sich dabei fort. Die Anwendung des Tests liefert Ergebnisse, die allerlei Zufällen unterworfen sind, gestattet aber doch einen ungefähren Gerätevergleich. Insbesondere sind Zeit- und Genauigkeitswerte fast ausschließlich durch die genannten Funktionen bestimmt, andere Eigenschaften von Rechner und Sprache haben praktisch keinen Einfluß. Werte ohne Literaturangabe sind durch eigene Messungen gewonnen (s. auch Abschnitt 1.2).

Großrechner	rel. Fehler	Zeit/s
ESER 1056, FORTRAN-OE-Compiler, s.p.	.6359 e+00	2.25
ESER 1056, FORTRAN-OE-Compiler, d.p.	.4153 e−09	3.12
ESER 1040, FORTRAN-ST-Compiler, s.p.	Argument-fehler	(0.70)
ESER 1040, FORTRAN-ST-Compiler, d.p.	.4153 e−09	3.30
ESER 1040, PASCAL 360-Compiler,	.9990 e+00	2.66
CRAY X-MP/24, FORTRAN-Compiler V1.14	.1900 e−22	0.746[1)]

16-Bit-Minirechner	rel. Fehler	Zeit/s
Robotron K 1620, LAOS, BASIC	.5268 e−01	235
Robotron K 1620, FORTRAN 1600, s.p.	.5268 e−01	119
Robotron K 1620, FORTRAN 1600, d.p.	−.8017 e−04	549
SM-4 (SU), FORTRAN-IV V1.3, s.p.	.1891 e−01	15.6
SM-4 (SU), FORTRAN-IV V1.3, d.p.	−.2631 e−10	248
SM-4 (SU), BASIC-11, s.p.	.1891 e+00	31.3
SM-3 (SU), BASIC-11, s.p.	.5268 e−01	162
E82 (SU), PASCAL, s.p.	.9395 e−02	1.17
E82 (SU), PASCAL, d.p.	.2630 e−12	2.14
E-60 (SU), BASIC, s.p.	.5268 e−01	193
E-60 (SU), BASIC, s.p.	−.2632 e−10	1041
E-60 (SU), FORTRAN-IV, s.p.	.5268 e−01	38
E-60 (SU), FORTRAN-IV, d.p.	−.2632 e−10	631
DEC PDP 11/70, FORTRAN IV-Plus, s.p.	.1079 e−00	1.32
DEC PDP 11/70, FORTRAN IV-Plus, d.p.	.6727 e−10	1.88

16-Bit-Personalcomputer	rel. Fehler	Zeit/s
CPU Intel 8086 und Intel 8088		
Robotron A 7100, SCP-BASIC, s.p.	−.1281 e−00	127
Robotron A 7100, SCP-BASIC, d.p.	−.1281 e−00	131
Robotron A 7100, DR-FORTRAN-77, V4.1, s.p.	−.9102 e−02	655
Robotron A 7100, DR-FORTRAN-77, V4.1, d.p.	.4400 e−12	665
CPU 8086, 8 MHz, MS-DOS V2.11, TURBO-PASCAL V3.01	.1854 e−05	119[1)]
Commodore PC-10, CPU 8088, 4.77 MHz, TURBO-PASCAL	.1854 e−05	193
IBM-PC, CPU 8088, 4.77 MHz, IBM-PC-BASIC VA3.10, s.p.	.8608 e−01	184
IBM-PC, CPU 8088, 4.77 MHz, IBM-PC-BASIC VA3.10, d.p.	−.1180 e−10	892
IBM-PC, CPU 8088, 4.77 MHz, mit MS-Quick-BASIC-Compiler V1.00 s.p.	−.3867 e−01	49
IBM-PC, CPU 8088, 4.77 MHz, mit MS-Quick-BASIC-Compiler V1.00 d.p.	−.2360 e−10	176
IBM-PC, CPU 8088, 4.77 MHz, BL-TURBO-PASCAL V3.01A	.1854 e−05	192
Schneider-PC 1512, DR-FORTRAN-77, V4.1, s.p.	−.9102 e−02	300
Schneider-PC 1512, DR-FORTRAN-77, V4.1, d.p.	.4710 e−12	305
Schneider-PC 1512, MS-FORTRAN-77, V3.3, s.p.	−.9102 e−02	108
Schneider-PC 1512, MS-FORTRAN-77, V3.3, d.p.	.4710 e−12	108
Schneider-PC 1512, MS-GW-BASIC, V3.11, s.p.	−.8678 e−01	72
Schneider-PC 1512, MS-GW-BASIC, V3.11, d.p.	−.8678 e−01	71
Schneider-PC 1512, Better-BASIC, s.p.	−.3802 e−01	37
Schneider-PC 1512, Better-BASIC, d.p.	.8008 e−10	108
CPU Intel 8086 oder Intel 8088 mit math. Coprozessor Intel 8087		
CPU 8086+8087, 8 MHz, MS-DOS V2.11, Turbo-PASCAL V3.01	.4720 e−12	6.0[1)]
Commodore PC-10, CPU 8088+8087, 8 MHz, Turbo-PASCAL	.4708 e−12	8.0
IBM-PC, CPU 8088+8087, 4.77 MHz, MS-FORTRAN-77 V3.31, s.p.	−.9102 e−02	7.0
IBM-PC, CPU 8088+8087, 4.77 MHz, MS-FORTRAN-77 V3.31, d.p.	.4709 e−12	7.3
IBM-PC, CPU 8088+8087, 4.77 MHz, DR-FORTRAN-77, V4.1 s.p.	−.9102 e−02	5.6
IBM-PC, CPU 8088+8087, 4.77 MHz, DR-FORTRAN-77, V4.1 d.p.	.4709 e−12	6.0
IBM-PC, CPU 8088+8087, 4.77 MHz, BL-TURBO-PASCAL V3.01A, mit 8087 math support	.4708 e−12	7.7
IBM-PC, CPU 8088+8087, 4.77 MHz, MS-PASCAL-Compiler V3.11	−.9395 e−02	9.0
IBM-PC, CPU 8088+8087, 4.77 MHz, DR-MT86-PASCAL-Compiler V3.2 mit Libraries 87 reals+87 trans	.4709 e−12	8.4
IBM-PC, CPU 8088+8087, 4.77 MHz, mit BL-TURBO-BASIC-Compiler, s.p.	−.9102 e−02	6.0
Schneider-PC 1512, 8 MHz, BL-TURBO-BASIC, s.p.	−.9102 e−02	0.97
Schneider-PC 1512, 8 MHz, BL-TURBO-BASIC, d.p.	.4709 e−12	0.97
CPU Intel 80286		
CPU 80286, 8 MHz, TURBO-PASCAL V3.01	.1854 e−05	54[1)]
CPU 80286, 8 MHz, MS-DOS V3.1, BASIC-A V3.11, s.p.	.8678 e−01	41[1)]

CPU 80286, 8 MHz, MS-DOS V3.1, BASIC-A V3.11, d.p.	.8678 e−01	42[1)]
CPU 80286, 8 MHz, MS-DOS V3.1, MS Quick-BASIC-Compiler, s.p.	.8867 e−01	11[1)]
CPU 80286, 8 MHz, MS-DOS V3.1, MS Quick-BASIC-Compiler, d.p.	.8867 e−01	34[1)]
Commodore PC-40, CPU 80286, 10 MHz, GW-BASIC V3.20	.8678 e−01	34
Commodore PC-40, CPU 80286, 10 MHz, TURBO-PASCAL	.1854 e−05	43
CPU Motorola 68000		
Atari 1040, CPU 68000, 8 MHz, GfA-BASIC	−.1480 e−07	15.6
Commodore Amiga 500, CPU 68000, 7.16 MHz, Workbench 1.2 V33.56, A.-BASIC, s.p.	.1079 e−00	34.5
Commodore Amiga 500, CPU 68000, 7.16 MHz, Workbench 1.2 V33.56, A.-BASIC, d.p.	−.1271 e−09	73.9
Sinclair QL, CPU 68008, BASIC	−.5150 e−03	100
Sinclair QL, CPU 68008, BASIC mit Compiler	−.5150 e−03	57

8-Bit-Mikrorechner	rel. Fehler	Zeit/s
CPU Z 80 und U 880		
Z80A, 4 MHz, CP/M 2.2, s.p.	−.7806 e−01	260[1)]
Z80A, 4 MHz, CP/M 2.2, d.p.	−.7806 e−01	274[1)]
Z80A, 4 MHz, CP/M 2.2, MS-Compiler V5.2, s.p.	−.7806 e−01	212[1)]
Z80A, 4 MHz, CP/M 2.2, MS-Compiler V5.2, d.p.	−.5202 e−10	2302[1)]
Z80A, 4 MHz, CP/M 2.2, MS-FORTRAN V3.34, s.p.	−.7806 e−01	212[1)]
Z80A, 4 MHz, CP/M 2.2, MS-FORTRAN V3.34, d.p.	−.7100 e−04	2227[1)]
Z80A, 4 MHz, CP/M 2.2, TURBO-PASCAL	.1854 e−05	404[1)]
U880, RIO-BASIC	.4620 e−07	1020
U880, CP/M-BASIC	−.7806 e−01	365
Robotron 1715, BASI, s.p.	−.7806 e−01	363
Robotron A5120, SCP-BASIC (BASI), s.p.	−.7800 e−01	366
Robotron A5120, SCP-BASIC (BASI), d.p.	−.7800 e−01	389
Robotron A5120, SCP-BASIC mit Compiler BASC, s.p.	−.7800 e−01	310
Robotron A5120, AMOS-BASIC V1.3	.8908 e−03	551
Robotron A5120, MBASIC (BASIC 80 Rev. 5.2, BCU-Version), s.p.	−.7806 e−01	369
Robotron A5120, MBASIC (BASIC 80 Rev. 5.2, BCU-Version), d.p.	−.7806 e−01	389
Robotron KC 87/1, 2.458 MHz, BASIC	.5269 e−01	334
Robotron KC 85/3, 1.7 MHz, BASIC	.5269 e−01	468
Robotron Z 1013.01, 1 MHz, BASIC	.5269 e−01	820
Olivetti M10	.1037 e−06	432
Sinclair Spectrum, BASIC	−.1297 e−04	971
Sinclair Spectrum, PASCAL	−.8719 e−00	59
Sinclair ZX81, BASIC, Fast-mode	−.1297 e−03	931
CPU 6502 und 6510		
Commodore C116, BASIC	.3600 e−07	442
Commodore C64, BASIC V2.0 (4-Byte-Mantisse)	.3586 e−07	523
Commodore C64, BASIC-Erweiterung ARITH13 (M. Olbrich) (5-Byte-Mantisse)	.1467 e−09	1055

Commodore C64, DB-PASCAL 64	.3586 e−07	511
Commodore C64, Markt & Technik MT-PASCAL V1.4	.3586 e−07	518
Commodore C64, DB Profi-C-Compiler	−.9175 e−00	839
Commodore C128, BASIC V7.0, slow-mode	.3600 e−07	556
Commodore C128, BASIC V7.0, fast-mode	.3600 e−07	266
Commodore C128, BASIC V7.0, Compiler BASIC 128, fast-mode	−.7246 e−03	66
Commodore C128, CP/M, TURBO-PASCAL	.1852 e−05	730
Sonstige oder unbekannte CPU		
Hewlett-Packard HP-85	−.2304 e−06	282
Texas-Instruments TI 99/4A	.4013 e−06	2580
Robotron K1001	.1379 e−04	12900
Robotron K1002	.1379 e−04	10840

Programmierbare Taschenrechner	rel. Fehler	Zeit/s
Elektronika MK 54 (SU)	−.9200 e−01	17400
Casio PB-300	−.3412 e−06	859
Casio FX-720P	.1763 e+00	870
Hewlett-Packard HP 41-C	−.1187 e−04	3144
Hewlett-Packard HP 33-E	−.9370 e−05	8400
Sharp PC-1248	.1760 e+00	684
Sharp PC-1404	.1700 e−05	1500
Texas-Instruments Programmable TI 59	.8908 e−05	3770
Texas-Instruments Programmable TI 58-C	.8908 e−05	4500

[1] Werte sind Messungen nach Kelso [Kels87]
MS = Microsoft DR = Digital-Research DB = Data-Becker MT = Markt & Technik BL = Borland
DEC = Digital-Equipment-Corporation
s.p. = single precision = einfache Genauigkeit, d. h. i. allg. 3-Byte-Mantisse
d.p. = double precision = doppelte Genauigkeit, d. h. i. allg. 7-Byte-Mantisse

Literaturverzeichnis

Abschnitt 1

[Abra68] *Abramowitz, M.; Stegun, I. A.:* Handbook of Mathematical Functions. New York: Dover Publ. Inc. 1968

[Budd87] *LaBudde, K.:* Structured Programming Concepts. New York usw.: McGraw-Hill 1987

[Chri85] *Christensen, B.R.:* Strukturierte Programmierung mit COMAL 80. München, Wien: Oldenbourg-Verlag 1985

[Cody80] *Cody, Waite:* Software Manual for the Elementary Functions. Englewood Cliffs, N. Y.: Prentice Hall 1980

[Dahl72] *Dahl, O.-J.; Dijkstra, E. W.; Hoare, C.A.R.:* Structured Programming. New York u. a.: Academic Press 1972

[Dijk68] *Dijkstra, E.W.:* GO TO Statement Considered Harmful. (Letter to the Editor). Communications of the ACM 11 (1968) S. 147f.

[Feic85] *Feichtinger, H.:* Hier irrt der Computer. mc München (1985) H. 10, S. 64–66

[Feic85,2] *Feichtinger, H.:* Wie schnell ist BASIC? mc München (1985) H. 10, S. 68–69

[Hart78] *Hart, J. F.; Cheney, E. W.; Lawson, C. L.; Maehly, H. J.; Mesztenyi, C. K.; Rice, J. R.; Thacher, H. C.; Witzgall, C.:* Computer Approximations. New York: John Wiley 1968. Reprinted and corrected: Huntington, New York: Robert E. Krieger Pub. Co. 1978

[Hast55] *Hastings, C.:* Approximations for Digital Computers. Princeton, N. J.: Princeton Univ.Press 1955

[Hopf86] *Hopfer, R.; Müller, R.:* BASIC. Einführung in das Programmieren. Leipzig: Fachbuchverlag 1986

[ISO-84] International Standard ISO 6373. Data Processing Programming Languages. Minimal-BASIC. 1984.

[Jack79] *Jackson, M. A.:* Grundsätze des Programmentwurfs. Darmstadt: S. Toeche-Mittler 1979

[Jone86] *Jones, C.* ed.: Programming Productivity. 2nd ed. Amsterdam: North-Holland/Elsevier 1986

[Kels87] *Kelso, T. S.:* Astronomical Software Benchmarks. Sky and Telescope (1987) No. 3, S. 309–310

[Klei82] *Klein, R. D.:* Basic-Interpreter. Funktionsweise und Implementierung in 8080/Z80 Computern. München: Franzis-Verlag 1982

[Leus86] *Leuschner, B.:* Programmieren Sie strukturiert!, Teil 1–3. 64'er (München: Verl. Markt u. Technik)
Teil 1: Ausgabe 1 (Jan. 1986), S. 120–125, 154
Teil 2: Ausgabe 2 (Feb. 1986), S. 156, 158, 160
Teil 3: Ausgabe 6 (Jun. 1986), S. 140, 142–144
Teil 4: Ausgabe 8 (Aug. 1986), S. 128–132

[Luke76] *Luke, Y. L.:* Mathematical Functions and Their Approximations. New York u. a.: Academic Press 1976

[Musa87] *Musa, J.:* Software Reliability. New York u. a.: McGraw-Hill 1987

[Paul81] *Paulin, G.; Schiemangk, H.:* Programmieren mit PASCAL. Berlin: Akademie-Verlag 1981

[Rice83] *Rice, J. R.:* Numerical Methods, Software, and Analysis. New York u. a.: McGraw-Hill 1983

[Roth87] *Rothardt, G.:* Praxis der Softwareentwicklung. Berlin: Verlag Technik 1987; Heidelberg: Dr. Alfred Hüthig Verlag 1987

[Schu82] *Schulz, A.:* Methoden des Softwareentwurfs und strukturierte Programmierung. 2. Aufl. Berlin, New York: de Gruyter 1982

[Scra84] *Scraton, R. E.:* Basic Numerical Methods. London: Edward Arnold 1984

[Stet76] *Stetter, H. J.:* Numerik für Informatiker. München, Wien: Oldenbourg 1976

[Wals72] *Walsh, D.:* Anleitung zur Software-Dokumentation. München: Hanser 1972

[Wern86] *Werner, D.:* BASIC für Mikrorechner. 2. Aufl. Berlin: Verlag Technik 1987

[Wieg84] *Wiegand, M.: Fillinger, M.; Fillinger, H.:* BASICODE. Ravensburg: Otto-Maier-Verlag 1984

[Wilk69] *Wilkinson, J. H.:* Rundungsfehler. Berlin, Heidelberg, New York: Springer 1969 (Heidelberger Taschenbücher Bd. 44)
[Wirt71] *Wirth, N.:* Program Development by Stepwise Refinement. Communications of the ACM 14 (1971), S. 221ff.
[Wirt73] *Wirth, N.:* Systematisches Programmieren. Stuttgart: Teubner 1972

Abschnitt 2

[Bach82] *Bachmann, W.: Haacke, R.:* Matrizenrechnung für Ingenieure. Berlin, Heidelberg, New York: Springer-Verlag 1982
[Böhm85] *Böhm, W.; Gose, G.; Kahmann, J.:* Methoden der numerischen Mathematik. Braunschweig, Wiesbaden: Vieweg 1985
[Enge85] *Engeln-Müllges, G.; Reutter, F.:* Formelsammlung zur Numerischen Mathematik mit BASIC-Programmen. Mannheim, Wien, Zürich: Bibliographisches Institut 1985
[Fors71] *Forsythe, G. E.; Moler, C. B.:* Computerverfahren für lineare algebraische Systeme. München, Wien: Oldenbourg Verlag 1971
[Gast72] *Gastinel, N.:* Lineare numerische Analysis. Berlin: VEB Deutscher Verlag der Wissenschaften 1972
[Herr85] *Herrmann, D.:* Angewandte Matrizenrechnung. Braunschweig, Wiesbaden: Vieweg 1985
[Hewl79] Hewlett-Packard-Company HP-85 Standard Paket. Druckschrift-Nr. 00085-90021. Dec. 1979
[Hewl80] Hewlett-Packard-Company HP-85 Matrix ROM Manual. Druckschrift-Nr. 00085-90144. March 1980
[Hilb77] *Hilbert, A.:* Matrizenrechnung. 2. Aufl. Berlin: Volk und Wissen 1977
[Hous64] *Householder, A. S.:* The theory of matrices in numerical analysis. New York: Blaisdell 1964
[Kahm85] *Kahmann, J.:* BASIC-Programme zur Numerischen Mathematik. Braunschweig, Wiesbaden: Vieweg 1985
[Ruck81] *Ruckdeschel, F. R.:* BASIC Scientific Subroutines. Peterborough, N. H.: Byte/McGraw-Hill 1981
[Schw68] *Schwarz, H. R.; Rutishauser, H.; Stiefel, E.:* Numerik symmetrischer Matrizen. Stuttgart: B. G. Teubner 1968
[Scra84] *Scraton, R. E.:* BASIC Numerical Methods. London: Edward Arnold Publications 1984
[Shou85] *Shoup, T.:* Numerische Verfahren für Arbeitsplatzrechner. München, Wien: Hanser 1985
[Stoe83] *Stoer, J.:* Einführung in die Numerische Mathematik. 4. Aufl. Berlin, Heidelberg, New York, Tokyo: Springer 1983
[Wilk69] *Wilkinson, J. H.:* Rundungsfehler. Berlin, Heidelberg, New York: Springer 1969
[Zurm50] *Zurmühl, R.:* Matrizen. Berlin, Göttingen, Heidelberg: Springer 1950

Abschnitt 3

[Enge85] *Engeln-Müllges, G.: Reutter, F.:* Formelsammlung zur Numerischen Mathematik mit BASIC-Programmen. Mannheim, Wien, Zürich: Bibliografisches Institut 1985
[Mull56] *Muller, D. E.:* A method for solving algebraic equations using an automatic computer. Math. Tables Aids Comp. 10 (1956), S. 208–215
[Rued65] *Rüdiger, D.; Kneschke, A.:* Technische Mechanik. Band 2: Festigkeitslehre. Leipzig: B. G. Teubner Verlagsgesellschaft 1965
[Sham73] *Shampine, L. F.; Allen, R. C.:* Numerical Computing. An Introduction. Philadelphia: W. B. Saunders Co. 1973

Abschnitt 4

[Albr79] *Albrecht, P.:* Die numerische Behandlung gewöhnlicher Differentialgleichungen. Berlin: Akademie-Verlag 1979
[Enge85] *Engeln-Müllges, G.; Reutter, F.:* Formelsammlung zur Numerischen Mathematik mit BASIC-Programmen. Mannheim, Wien, Zürich: Bibliografisches Institut 1985
[Fehl69] *Fehlberg, E.:* Klassische Runge-Kutta-Formeln fünfter und siebenter Ordnung mit Schrittweitenkontrolle und ihre Anwendung auf Wärmeleitungsprobleme. Computing 4 (1969), S. 93–106
[Fehl70] *Fehlberg, E.:* Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweitenkontrolle und ihre Anwendung auf Wärmeleitungsprobleme. Computing 6 (1970), S. 61–71
[Gear69] *Gear, C. W.:* The automatic integration off stiff ordinary differential equations. Information Processing 68, Vol. 1, S. 187–193, North-Holland 1969

[Gear71] *Gear, C. W.:* Numerical initial value problems in ordinary differential equations. Englewood Cliffs: Prentice Hall 1971

[Herr83] *Herrmann, D.:* Numerische Mathematik – 40 BASIC-Progamme. Braunschweig: Friedrich Vieweg und Sohn 1983

[Kell68] *Keller, H. B.:* Numerical Methods for two point boundary value problems. Massachusetts, Toronto, London: Blaisdell 1968

[Lamb73] *Lambert, J. D.:* Computational methods in ordinary differential equations. London: John Wiley and Sons 1973

[Phil84] *Phillipow, E.:* Taschenbuch Elektrotechnik. Band 4: Systeme der Informationstechnik. Berlin: VEB Verlag Technik 1984

[Scra84] *Scraton, R. E.:* BASIC numerical methods. London: Edward Arnold Ltd. 1984

[Sham73] *Shampine, L. F.; Allen, R. C.:* Numerical Computing. An Introduction. Philadelphia: W. B. Saunders Co. 1973

[Sham77] *Shampine, L. F.:* Stiffness and nonstiff differential equation resolvers II. Assoc. Comp. Mach. 3 (1977), S. 44–53

Sachwörterverzeichnis

Das Buch beschreibt bewährte Algorithmen in Form von BASIC-Programmodulen, die auf einfache Weise in eigene Programme eingebunden werden können. Die Verwendung der Module führt zu erhöhter Produktivität bei der Programmentwicklung und Mikrorechnernutzung. Behandelt werden lineare, nichtlineare und Differentialgleichungen. Einführend werden grundlegende Regeln für numerische Berechnungen, zum Programmentwurf, zur Dokumentation und zur Fehlerbeseitigung erläutert.

ISBN 3-341-00621-4