Numerische Methoden in der Physik (515.421)

Heinrich Sormann
Institut für Theoretische Physik - Computational Physics
TU Graz

Skriptum WS 2007/2008

Kapitel 1

Einführung

1.1 Grundbegriffe

Die Lehrveranstaltung *Numerische Methoden in der Physik* beschäftigt sich mit der Lösung physikalisch-technischer Probleme unter Verwendung von Methoden der Numerischen Mathematik, wobei die Komplexizität der Problemstellungen den Einsatz eines Computers erforderlich macht.

Eine Definition des Begriffes *Numerische Mathematik* findet sich z.B. in [1], S.8:

> *Die Numerische Mathematik befaßt sich mit der Lösung von mathematischen Problemen mit Hilfe zahlenmäßigen Rechnens. Darunter ist eine endliche Folge von Grundrechenoperationen, also Addition, Subtraktion, Multiplikation und Division zu verstehen, wobei Zahlen mit einer beschränkten Stellenanzahl verwendet werden.*

Ebenfalls in [1] ist die grundsätzliche Methodik der Numerischen Mathematik zusammen mit den unvermeidlich auftretenden Fehlern schematisch dargestellt:

```
|--------------------|       |------------------------------|
| Eingabeinformation | ===>  |        Eingabefehler         |
|       (Input)      |       | (Messfehler; Rundungsfehler) |
|--------------------|       |------------------------------|
          |                                 |
          |                                 |
|--------------------|       |------------------------------------|
|    Algorithmus     | ===>  |      Algorithmusfehler             |
|                    |       | (Verfahrensfehler; Rundungsfehler) |
|--------------------|       |------------------------------------|
          |                                 |
          |                                 |
|--------------------|       |------------------------------|
| Ausgabeinformation | ===>  |         Ausgabefehler        |
|      (Output)      |       |       (Rundungsfehler)       |
|--------------------|       |------------------------------|
```

Diagramm zur Methodik der Numerischen Mathematik.

Ein zentraler Begriff der Numerischen Mathematik ist der Begriff *Algorithmus.* Darunter versteht man nach [2], S.9

eine endliche Menge von genau beschriebenen Anweisungen, die mit vorgegebenen Anfangsdaten in bestimmter Reihenfolge nacheinander auszuführen sind, um die Lösung zu ermitteln.

1.2 Allgemeines zum Thema: Fehler

Wie aus dem obigen Diagramm hervorgeht, tragen alle Aktivitäten bei der Durchführung einer numerischen Rechnung (Dateneingabe; eigentlicher Rechenvorgang; Datenausgabe) zum Fehler des Ergebnisses bei. Ursache dafür sind die auftretenden *Rundungsfehler* und *Verfahrensfehler*, auf die in den folgenden Kapiteln ausführlich eingegangen wird.

Ergebnisse von Computerberechnungen sind also (bis auf ganz wenige Ausnahmen) fehlerbehaftet, d.h. sie entsprechen nicht exakt den „wahren“ Ergebnissen des behandelten Problems.

Es ist daher von entscheidender Wichtigkeit, bei der Arbeit mit einem Computer die auftretenden Fehler unter Kontrolle (d.h. so klein wie irgend möglich) zu halten bzw. die Genauigkeit der erhaltenen Ergebnisse so gut als möglich beurteilen zu können.

Als Literatur zum Thema Fehler bei der Anwendung numerischer Verfahrenßind die einschlägigen Kapitel aus [7] und [8] sehr zu empfehlen.

1.2.1 Absoluter und relativer Fehler. Maschinengenauigkeit.

Bevor auf die einzelnen Fehlertypen näher eingegangen wird, sollen im folgenden die Begriffe *absoluter* und *relativer* Fehler erläutert werden:

Ist x der (unbekannte) wahre Wert eines Rechenergebnisses und $\bar{x}$ der entsprechende Näherungswert, so nennt man

$$\epsilon_a = x - \bar{x} \tag{1.1}$$

den *absoluten* Fehler und

$$\epsilon_r = \frac{\epsilon_a}{\bar{x}} = \frac{x - \bar{x}}{\bar{x}} \tag{1.2}$$

den *relativen* Fehler von $\bar{x}$.

In der Praxis wird meistens der relativen Fehlerbetrachtung nach (1.2) der Vorzug gegeben. Der Grund dafür wird im folgenden einfachen Beispiel klar:

	x	$\bar{x}$	ϵ_a	ϵ_r
(a)	0.1	0.09	0.01	≈ 0.1
(b)	1000.0	999.99	0.01	≈ 0.00001

Struktogramm 1 — Ermittlung der Kenngröße τ

```
tau:=1.0
walt:=1.0
    tau:=tau/2.0
    wneu:=walt + tau
wneu=walt
print: 2*tau
```

Es leuchtet sofort ein, daß die Näherung (b) weit befriedigender ist als die Näherung (a), obwohl der absolute Fehler in beiden Fällen gleich ist.

Ein weiterer wichtiger Begriff ist die **Maschinen-Genauigkeit**: Es ist dies die kleinste (positive) Zahl τ, für welche gilt:

$$1 + \tau > 1$$

Für einen (nicht-existierenden) Super-Rechner, der reelle Zahlen mit beliebiger Genauigkeit abspeichern kann, erfüllt natürlich jedes τ die obige Ungleichung. Für einen realen Computer ist τ jedoch eine wichtige Kenngröße, die mit dem Algorithmus im Struktogramm 1 leicht ermittelt werden kann.
Die folgenden Ergebnisse wurden auf einem IBM-PC erhalten (wie alle in diesem Skriptum enthaltenen Testergebnisse, soferne es nicht ausdrücklich anders angegeben ist):

	Bytes	τ	Anz. sign. Stellen
C-float	4	$1.19 \cdot 10^{-7}$	7
C-double	8	$2.22 \cdot 10^{-16}$	16
F90-real	4	$1.19 \cdot 10^{-7}$	7
F90-double prec.	8	$2.22 \cdot 10^{-16}$	16
Pascal-single	4	$1.19 \cdot 10^{-7}$	7
Pascal-real	6	$1.82 \cdot 10^{-12}$	12
Pascal-double	8	$2.22 \cdot 10^{-16}$	16
Pascal-extended	10	$1.08 \cdot 10^{-19}$	19

Realisierung des Struktogramms 1 in C und F90:

```
// C-PROGRAMM ZUR BERECHNUNG DER MASCHINEN-GENAUIGKEIT

#include <iostream.h>

void main()
{
  float tau,walt,wneu;

  tau=1.0;
  walt=1.0;
  do {
    tau/=2.0;
    wneu=walt+tau;
  } while (wneu != walt);
  cout<<"TAU = "<<2*tau<<"\n";
}
```

```
PROGRAM maschin

! F90-Programm zur Berechnung der Maschinen-Genauigkeit

IMPLICIT NONE

REAL tau,walt,wneu

tau=1.0
walt=1.0

DO
  tau=tau/2.0
  wneu=walt+tau
  IF(wneu .EQ. walt)EXIT
END DO

PRINT '(" tau = ",E12.6)',2.0*tau

END PROGRAM maschin
```

1.2.2 Eingabe- oder Input-Fehler. Schlecht konditionierte Probleme.

Eingabe- bzw. *Input*-Fehler (*inherent errors*) sind Unsicherheiten der eingegebenen Daten, mit welchen der Computer die Berechnung durchführt. Diese Unsicherheiten können verschiedene Ursachen haben, z.B. experimentelle Meßfehler oder *Rundungsfehler*.

Wenn die Ergebnisse einer Berechnung stark von diesen Eingabefehlern beeinflußt werden, spricht man von einem 'schlecht konditionierten Problem'.

Dazu als Beispiel ein schlecht konditioniertes lineares Gleichungssystem [7], S.97:

$$x + 5.0y = 17.0$$

$$1.5x + 7.501y = a$$

Angenommen, die Größe a sei der experimentell ermittelte Wert

$$a = 25.503 \pm 0.001$$

Wie man sich leicht überzeugen kann, lauten die exakten Lösungen dieses Systems für $a = 25.503$: $x = 2.0$ und $y = 3.0$.
Die folgende Tabelle zeigt nun, wie massiv sich die Ergebnisse des Gleichungssystems unter der Annahme ändern, daß die letzte Ziffer von a um eine Einheit unsicher ist:

a	x	y
25.503	2.	3.
25.502	7.	2.
25.504	-3.	4.

Das obige Gleichungssystem ist also extrem schlecht konditioniert, und die Ergebnisse sind (bei Annahme einer experimentell bedingten Unsicherheit von a) völlig sinnlos!

1.2.3 Algorithmusfehler

sind Fehler, die bei der Auswertung der Rechenvorschrift geschehen.
Die Ursache dafür sind entweder *Rundungsfehler* oder *Verfahrensfehler*.

1.2.4 Verfahrensfehler

Verfahrensfehler entstehen dadurch, daß das gegebene mathematische Problem durch ein vereinfachtes Problem ersetzt wird.
Ein Beispiel dafür ist die numerische Integration:

Das bestimmte Integral

$$I = \int_{x=1}^{2} \frac{dx}{x^2}$$

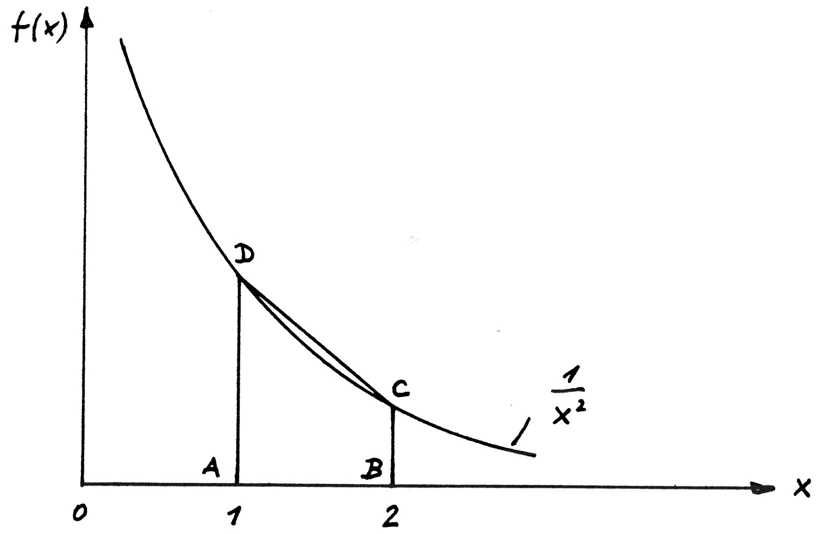


Abbildung 1.1: Prinzip einer numerischen Integration.

mit der exakten Lösung 0.5 kann z.B. numerisch dadurch gelöst werden, daß man die 'wahre' Fläche durch die Trapezfläche $ABCD$ ersetzt (Abb.1.1). Das 'numerische' Ergebnis lautet in diesem Fall 0.625, d.h. es tritt ein (absoluter) Verfahrensfehler

$$\epsilon_V = 0.5 - 0.625 = -0.125$$

auf. Weitere typische Verfahrensfehler ergeben sich z.B. aus dem vorzeitigen Abbrechen einer unendlichen Reihe (*Abbruchfehler*, engl. *truncation error*) usw.

1.2.5 Rundungsfehler

Die Tatsache, daß für die Abspeicherung einer reellen Zahl (genauer gesagt: für deren Mantisse) nur eine endliche Anzahl von Bits zur Verfügung steht, ist die Ursache für die in fast [1] jedem Computerprogramm auftretenden, unvermeidlichen [2] Rundungsfehler.

Ein simples Beispiel:

```
   Rechnung "von Hand":            Computer (7 sign. Stellen):

     123456.789                        0.1234568E+06
  +        9.876543                 +  0.9876543E+01
  ----------------                  ----------------
     123466.665543                     0.1234667E+06
```

[1]Außer man kommt z.B. mit reiner INTEGER-Arithmetik aus, was sehr selten ist.

[2]'Unvermeidlich' heißt, wie noch ausführlich demonstriert wird, keineswegs, daß man keine Möglichkeiten hat, Rundungsfehler zu reduzieren!

Struktogramm 2 — Demonstration Rundungsfehler-Effekt

wert:=0.1 sum:=0.0
i=1(1)50
sum:=sum + wert
print: sum

Ein weiteres Beispiel: Die Auswertung des Struktogramms 2 zeigt ein auf den ersten Blick unverständliches Phänomen:
Auf dem PC lautet das Ergebnis 4.999998, d.h. man erhält einen Rundungsfehlereffekt, obwohl nur 50mal die Konstante 0.1 aufaddiert wird, die ja exakt abgespeichert werden kann.

Die Ursache für diesen Effekt ist darin begründet, daß die Zahl 0.1 im Dezimalsystem zwar ohne Schwierigkeiten hingeschrieben werden kann, nicht aber im vom Computer verwendeten Dualsystem. In diesem ist 0.1 nämlich periodisch:

$$0.1_{10} = 0.000110011001100\ldots_2$$

1.3 Verfahrens- und Rundungsfehler

Im folgenden soll das Fehlerverhalten von Computerprogrammen an Hand einiger typischer Beispiele demonstriert werden. Es sollen dabei Möglichkeiten gezeigt werden, wie man Verfälschungen von Resultaten auf Grund von Verfahrens- und Rundungsfehlern erkennen bzw. möglichst klein halten kann.

1.3.1 Zusammenhang zwischen Rundungsfehler und Algorithmus

Das erste Beispiel zeigt, wie die Rundungsfehler bei der Auswertung eines mathematischen Ausdrucks durch geeignete Umformulierung des Algorithmus reduziert werden können.

Es geht um die numerische Auswertung des Ausdruckes

$$F(x) = \frac{\sqrt{1+x} - \sqrt{1-x}}{x}$$

für $x << 1$. Da für $F(x)$ eine geschlossene Formel vorliegt, gibt es hier natürlich keinen Verfahrensfehler. Auch die Ein- und Ausgabefehler sollen bei diesem Beispiel vernachlässigt werden.

Programmiert man nun die Formel ohne weitere Umformung, also etwa

$$F := (SQRT(1. + X) - SQRT(1. - X))/X \quad ,$$

so erhält man für kontinuierlich kleiner werdende x-Werte das folgende Ergebnis:

Tab.1.1: Numerische Auswertung von F(x) bzw. F1(x).

x	F	F1
10^0	$0.1414214E + 01$	$0.1414214E + 01$
10^{-1}	$0.1001256E + 01$	$0.1001256E + 01$
10^{-2}	$0.1000013E + 01$	$0.1000013E + 01$
10^{-3}	$0.1000041E + 01$	$0.1000000E + 01$
10^{-4}	$0.1000153E + 01$	$0.1000000E + 01$
10^{-5}	$0.1001357E + 01$	$0.1000000E + 01$
10^{-6}	$0.1013279E + 01$	$0.1000000E + 01$
10^{-7}	$0.1192093E + 01$	$0.1000000E + 01$
10^{-8}	$0.0000000E + 01$	$0.1000000E + 01$

Bei den F-Werten in der obigen Tabelle fallen 2 Dinge auf:

- Die Funktionswerte streben nicht - wie zu erwarten - gegen 1, sondern nehmen ab ca. $x = 10^{-2}$ wieder zu.
- Ab einem minimalen x-Wert erhält man für F immer exakt Null.

Dieses Verhalten ist leicht erklärbar:

- Für kleine x repräsentieren die beiden Wurzeln $\sqrt{1+x}$ und $\sqrt{1-x}$ zwei *fast gleich große Zahlen*. Bei der Subtraktion dieser Zahlen und der anschließenden Division durch x wirken sich die (an sich kleinen) Rundungsfehler der einzelnen Terme nach dem Fehlerfortpflanzungsgesetz sehr stark aus ('subtractive cancellation').
- Für $x < \tau$ ergibt sich $1 + x = 1$ und $1 - x = 1$; somit wird der Zähler von $F(x)$ exakt Null.

Soweit die Fehlerdiagnose. Die Therapie ist in diesem Fall sehr einfach:

Formt man nämlich $F(x)$ in

$$F1(x) \equiv F(x) = \frac{(\sqrt{1+x} - \sqrt{1-x})}{x} \cdot \frac{(\sqrt{1+x} + \sqrt{1-x})}{(\sqrt{1+x} + \sqrt{1-x})} = \frac{2}{\sqrt{1+x} + \sqrt{1-x}}$$

um, so erhält man die korrekten Ergebnisse (s. Tabelle 1.1).

Ein weiteres Beispiel ([9], S.178) soll demonstrieren, daß man auch bei (scheinbar) eindeutigen Problemen in Schwierigkeiten geraten kann, wenn man bekannte Formeln 'einfach drauflos programmiert'.

Es geht um die numerische Auswertung quadratischer Gleichungen

$$ax^2 + bx + c = 0$$

mit den reellen Koeffizienten a, b und c.

Die wohlbekannten Lösungen lauten

$$(a) \quad x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad (b) \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Man sieht jedoch sofort, daß für sehr kleine Koeffizienten a und/oder c die Gefahr einer 'subtractive cancellation' im Zähler von x_1 (für $b > 0$) bzw. für x_2 (für $b < 0$) gegeben ist.
Es ist aber leicht möglich, die obigen Formeln äquivalent umzuformen zu

$$(c) \quad x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \qquad (d) \quad x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}} \quad .$$

Aus den vorherigen Überlegungen ergibt sich, daß für $b > 0$ die Formeln (c) und (b) rundungsfehler-stabil sind, für $b < 0$ die Formeln (a) und (d).

Eine Zusammenfassung dieser Ausdrücke führt zum folgenden Algorithmus:

$$x_1 = \frac{q}{a} \qquad x_2 = \frac{c}{q}$$

mit

$$q \equiv -\frac{1}{2}\left[b + sgn(b)\sqrt{b^2 - 4ac}\right] \quad .$$

1.3.2 Rundungs- und Verfahrensfehler bei der numerischen Differentiation

Das Grundprinzip einer numerischen Differentiation besteht darin, daß man den gewünschten *Differentialquotienten* durch einen entsprechenden *Differenzenquotienten* ersetzt, also z.B.

$$\frac{d}{dx} f(x) \mid_{x=x_o} \approx \frac{f(x_o + h) - f(x_o)}{h} \tag{1.3}$$

schreibt.

Aus (1.3) geht hervor, daß der Differenzenquotient für die *Schrittweite* $h \to 0$ zur ersten Ableitung der Funktion $f(x)$ an der Stelle x_o konvergiert. Für eine *endliche* Schrittweite h ist natürlich mit einem *Verfahrensfehler* ϵ_V zu rechnen, wobei ϵ_V mit kleiner werdendem h abnehmen wird.
Theoretische Überlegungen zeigen, daß ϵ_V die Form

$$\epsilon_V = C_V(h) \cdot h \quad , \tag{1.4}$$

hat, wobei die Größe C in vielen Fällen nur schwach von h abhängt und in der Praxis durch eine Konstante ersetzt werden kann:

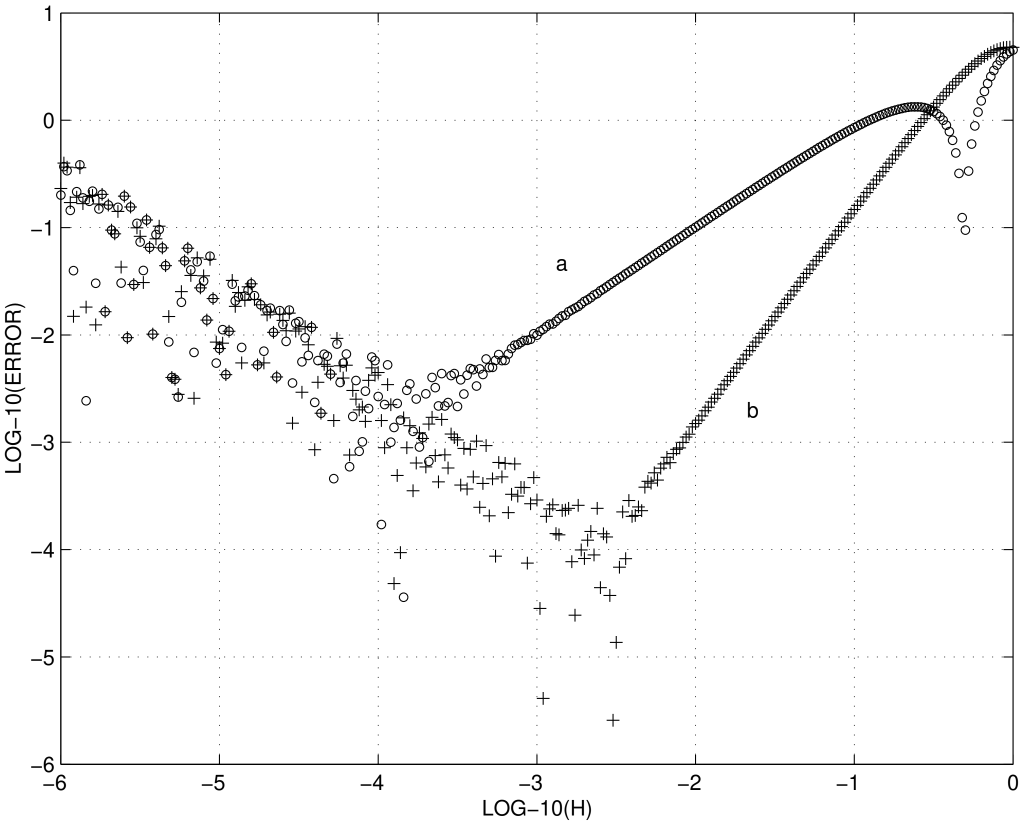


Abbildung 1.2: Fehlerdiagramm zur numerischen Differentiation. (a) Formel (1.3), (b) Formel (1.7)

$$C_V(h) \approx C_V \quad .$$

Trägt man nun ϵ_V in Abhängigkeit von h in einem *doppelt-logarithmischen* Diagramm auf, so ergibt sich als *Fehlerkurve* die lineare Beziehung

$$\lg \epsilon_V = \lg C_V + \lg h \tag{1.5}$$

mit der Steigung $+1$.
Diese Zusammenhänge sind in der Abb.1.2, Kurve (a) dargestellt, und zwar für das konkrete Beispiel

$$f(x) = \ln x \cdot \sin(5x)$$

$$\frac{d}{dx} f(x) \mid_{x_o=3} = \frac{\sin(15)}{3} + 5 \ln 3 \cos(15) \quad .$$

Offensichtlich ist der Zusammenhang (1.4) für den Bereich $10^{-3} < h < 10^{-1}$ sehr gut erfüllt. Für größere und für kleinere Schrittweiten zeigt jedoch das 'wahre' Fehlerverhalten starke Abweichungen vom theoretisch zu erwartenden Verfahrensfehler.
Die Ursachen dafür sind die folgenden:

- Die Abweichung im Bereich $h > 10^{-1}$ ist darin begründet, daß bei zu großen Schrittweiten die Konstanz von C_V nicht mehr erfüllt ist.

- Interessanter ist das Fehlerverhalten im Bereich $h < 10^{-3}$, wo die Abhängigkeit des Fehlers von h offensichtlich prinzipiell nicht mehr der Glg.(1.4) entspricht, sondern durch ein starkes Überhandnehmen des Rundungsfehlers geprägt ist. Die Ursache dafür ist wieder eine ßubtractive cancellationïm Zähler von (1.3).

Die Abhängigkeit des Rundungsfehlers von h entspricht keiner glatten Kurve, hat aber tendenziell die Form

$$\epsilon_R = \frac{C_R}{h} \quad . \tag{1.6}$$

Aus dem bisher Gesagten ergeben sich die folgenden wichtigen Konsequenzen:

- Bei der Verwendung einer *optimalen* Schrittweite h_{opt} erreicht man einen minimalen Gesamtfehler (im konkreten Beispiel beträgt dieser Fehler ca. $5 \cdot 10^{-3}$), der durch eine Variation von h *nicht weiter unterschritten werden kann*!

- Die Verwendung einer Schrittweite $h < h_{opt}$ führt zu keiner Verbesserung des numerischen Ergebnisses, sondern - im Gegenteil - zu einer Verschlechterung!

Eine Reduktion dieses minimalen Fehlers kann auf zwei Arten geschehen:

- Der Rundungsfehler kann durch eine Erhöhung der Abspeicherungs-- Genauigkeit der reellen Zahlen (single $\rightarrow$ double; float $\rightarrow$ double etc.) verkleinert werden. Dadurch ist es möglich, mit kleineren Schrittweiten zu arbeiten.

- Der Verfahrensfehler kann durch die Verwendung eines *verbesserten Algorithmus* reduziert werden, etwa durch die Verwendung der leistungsfähigeren Formel

$$\frac{d}{dx} f(x) \mid_{x=x_o} \approx \frac{f(x_o + h) - f(x_o - h)}{2h} \tag{1.7}$$

für den Differenzenquotienten. Da bei dieser Formel ϵ_V mit abnehmenden h bedeutend schneller abnimmt als dies bei der Formel (1.3) der Fall war, nämlich

$$\epsilon_V = \bar{C}_V(h) \cdot h^2 \quad , \tag{1.8}$$

der Rundungsfehler jedoch im wesentlichen dieselbe h-Abhängigkeit hat wie vorher, kann der erreichbare Mindestfehler um etwa eine Größenordnung gesenkt werden (vgl. die Kurve (b) in der Abb.1.2).

Zusammenfassend kann gesagt werden:

Die Fehlerdiagnose in diesem Beispiel (und in vielen Bereichen der numerischen Mathematik) wird dadurch entscheidend erleichtert, daß unter bestimmten Voraussetzungen (hier: in einem bestimmten h-Bereich) *der Verfahrensfehler den Rundungsfehler stark dominiert.*

1.3.3 Fehlerdiagnose bei der numerischen Auswertung der Errorfunktion.

In vielen Anwendungen ist die numerische Auswertung der Funktion

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$$

erforderlich, wobei $\mathrm{erf}(x)$ die *Gauss'sche Fehlerfunktion* darstellt.

Die Integraldarstellung bzw. die Darstellung in Form einer Taylorreihe lautet:

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_{z=0}^{x} dz e^{-z^2} = 1 - \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \quad .$$

Die numerische Auswertung von $\mathrm{erfc}(x)$ kann von dieser Taylorreihe ausgehen, deren Konvergenz im Prinzip für alle reellen Argumente x gesichert ist:

$$\mathrm{erfc}(x) \approx 1 - \frac{2}{\sqrt{\pi}} \sum_{n=0}^{nmax} a_n \quad mit \quad a_n = \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \quad .$$

EINSCHUB: Num. Auswertung einer Taylorreihe

- Vom Standpunkt der Rechenökonomie wäre es sehr ungünstig, jeden Term a_n der Reihe unabhängig von den anderen zu bestimmen. Viel besser ist es, jedes a_n aus dem vorherigen a_{n-1} zu berechnen. Im konkreten Fall findet man durch Berechnung des Quotienten a_n/a_{n-1} leicht den *rekursiven* Zusammenhang

  $$a_n = \left[-\frac{x^2(2n-1)}{n(2n+1)} \right] a_{n-1} \qquad \text{für} \quad n = 1, 2, \ldots$$

  mit

  $$a_0 = x\,.$$

- Durch das Abbrechen der Reihe bei $n = nmax$ wird ein Verfahrensfehler auftreten. Da diese Reihe aber aus monoton kleiner werdenden Gliedern [3] mit alternierenden Vorzeichen besteht, ist ϵ_V dem Betrage nach sicher nicht größer als der erste vernachlässigte Term a_{nmax+1}. Durch diese einfache Abschätzmöglichkeit für den Verfahrensfehler ist es möglich, die Reihe bis auf Maschinengenauigkeit auszuwerten, d.h. solange, bis der aufsummierte Wert durch weitere Terme nicht mehr verändert wird. D.h.: In den Grenzen der Maschinengenauigkeit tritt kein Verfahrensfehler auf!

[3] Zumindest ab einem bestimmten a_n!

Struktogramm — Auswertung einer Taylorreihe

```
x:= .....
-----------------------------------------
n:=0
sumnew:=x
a:=x
-----------------------------------------
   | n:=n+1
   | taylor:=sumnew
   | a:= -x*x*(2*n-1)/n/(2*n+1) * a
   | sumnew:=sumnew+a
-----------------------------------------
sumnew=taylor
-----------------------------------------
print: x, taylor
```

Aus dem vorher Gesagten geht hervor, daß alle Abweichungen der numerischen von den exakten Ergebnissen nur durch Rundungsfehler-Akkumulationen zustande kommen. Dies kann man sofort erkennen, wenn man die Auswertung sowohl mit einfacher als auch mit doppelter Genauigkeit durchführt. Die Ergebnisse dieses Tests für eine Reihe von x sind in der Tabelle 1.2, Spalte 2, angegeben.

Aus dem Vergleich der Ergebnisse für die Auswertung der Taylorreihe ergibt sich, daß die numerische Berechung von erfc(x) für kleine Argumente x sehr gut funktioniert, für größere x jedoch total versagt.
In diesem Argumentbereich muß ein anderer Algorithmus verwendet werden.

Dafür bietet sich ein sogenannter *Kettenbruch* [11] an, der die Form

$$\mathrm{erfc}(x) = \frac{e^{-x^2}}{\sqrt{\pi}} \cdot \left(\frac{1}{x+} \frac{1/2}{x+} \frac{1}{x+} \frac{3/2}{x+} \cdots \right) \qquad .$$

hat. Allerdings hat die übliche numerische Auswertung eines Kettenbruches den Nachteil, daß bereits vor Beginn der Auswertung feststehen muß, wieviele Terme des Bruches in die Rechnung miteinbezogen werden. Zeigt es sich, daß eine Erhöhung der Term-Anzahl nötig ist (um den Verfahrensfehler zu reduzieren), so muß die Auswertung ganz neu begonnen werden, und man kann nicht (wie etwa bei der Auswertung einer Taylorreihe) einfach zusätzliche Terme zum 'alten' Ergebnis hinzufügen [4].

Im konkreten Fall wurde der Kettenbruch jeweils für 31 und für 61 Terme ausgewertet, wieder sowohl mit einfacher als auch mit doppelter Genauigkeit. Aus dem Vergleich der Ergebnisse (Tabelle 1.2, dritte und vierte Spalte) können die folgenden Schlüsse gezogen werden:

- Unterschiede bei den Ergebnissen mit 31 bzw. 61 Termen → *Verfahrensfehler.*

[4] Eine solche Möglichkeit in Form einer besseren Auswertung eines Kettenbruches findet man z.B. in [9], S.135f).

- Unterschiede bei den Ergebnissen mit einfacher bzw. doppelter Genauigkeit → *Rundungsfehler*.

Tab.1.2: Auswertung der Funktion erfc(x) mittels der Taylorreihe bzw. des Kettenbruches. Alle Ergebnisse sind in halb-exponentieller Schreibweise mit 7 Mantissenstellen angegeben, wobei der Exponent E eine Berechnung in einfacher Genauigkeit bedeutet, der Exponent D eine Berechnung in doppelter Genauigkeit.

x	Taylorreihe $\epsilon_V = 0$	Kettenbruch 31 Terme	Kettenbruch 61 Terme
0.01	0.9887166E+00	0.1261318E+02	0.9042203E+01
	0.9887166D+00	0.1261318D+02	0.9042202D+01
0.1	0.8875371E+00	0.1401417E+01	0.1131721E+01
	0.8875371D+00	0.1401417D+01	0.1131721D+01
0.5	0.4795001E+00	0.4802107E+00	0.4795305E+00
	0.4795001D+00	0.4802107D+00	0.4795305D+00
1.	0.1572992E+00	0.1572995E+00	0.1572992E+00
	0.1572992D+00	0.1572995D+00	0.1572992D+00
2.	0.4677685E-02	0.4677735E-02	0.4677735E-02
	0.4677735D-02	0.4677735D-02	0.4677735D-02
3.	0.2991446E-04	0.2209050E-04	0.2209050E-04
	0.2209050D-04	0.2209050D-04	0.2209050D-04
4.	0.2364931E-02	0.1541726E-07	0.1541726E-07
	0.1544033D-07	0.1541726D-07	0.1541726D-07
5.	0.4645361E+02	0.1537460E-11	0.1537460E-11
	0.5458862D-07	0.1537460D-11	0.1537460D-11

Als Zusammenfassung der Ergebnisse kann folgendes festgestellt werden:

- Taylorreihe: Verfahrensfehler im gesamten x-Bereich Null, Rundungsfehler mit steigendem x stark ansteigend.

  Die Taylorreihen-Auswertung gibt bis ca. $x = 1$ Ergebnisse mit mind. 7 sicheren Ziffern.

- Kettenbruch: Verfahrensfehler nimmt mit steigendem x ab, der Rundungsfehler ist im gesamten untersuchten x-Bereich sehr klein.

  Die Kettenbruch-Auswertung mit 31 Termen gibt ab ca. $x = 2$ Ergebnisse mit mind. 7 sicheren Ziffern.

1.3.4 Beispiel für stabile und instabile Algorithmen

Viele numerische Verfahren beruhen auf *Rekursionsformeln* vom Typus

$$y_n = a \cdot y_{n-1} + b \cdot y_{n-2} \qquad n = 2, 3, \ldots \quad .$$

Bei der Anwendung solcher Formeln muß besonders auf die *Stabilität* des Algorithmus geachtet werden:

Ein Algorithmus heißt stabil oder instabil, je nachdem ein im n-ten Rechenschritt zugelassener Rechnungsfehler bei exakter Rechnung in den Folgeschritten abnimmt oder anwächst (aus [2], S.9).

Ein sehr instruktives Beispiel stellt in diesem Zusammenhang die numerische Auswertung von *sphärischen Besselfunktionen* mittels einer „Vorwärts-Rekursion“ dar:

$$j_o(x) = \frac{\sin x}{x} \qquad j_1(x) = \frac{\sin x}{x^2} - \frac{\cos x}{x}$$

$$j_l(x) = \frac{2l-1}{x} \cdot j_{l-1}(x) - j_{l-2}(x) \qquad l = 2, 3, \ldots, lmax$$

Bei der numerischen Auswertung dieser Formel gibt es natürlich überhaupt keine Verfahrensfehler; alle auftretenden Fehler können nur Rundungsfehler sein! Die Größe der Rundungsfehler kann wiederum durch den Vergleich von Ergebnissen abgeschätzt werden, die mit einfacher bzw. doppelter Genauigkeit berechnet wurden (s. Tabelle 1.3).

Tab.1.3: Numerische Berechnung der sphärischen Besselfunktionen der Ordnungen 0-9 mittels Vorwärts-Rekursion.

```
         einfache Gen.                 doppelte Gen.
x =  0.3000
L=  0    0.9850674E+00            0    0.9850674D+00
    1    0.9910274E-01            1    0.9910289D-01
    2    0.5960023E-02            2    0.5961525D-02
    3    0.2309625E-03            3    0.2558598D-03
    4    -.5708978E-03            4    0.8536426D-05
    5    -.1735790E-01            5    0.2329701D-06
    6    -.6358853E+00            6    0.5811086D-08
    7    -.2753767E+02            7    0.1884359D-07
    8    -.1376248E+04            8    0.9363682D-06
    9    -.7795982E+05            9    0.5304202D-04
x =  1.0000
L=  0    0.8414710E+00            0    0.8414710D+00
    1    0.3011687E+00            1    0.3011687D+00
    2    0.6203508E-01            2    0.6203505D-01
    3    0.9006739E-02            3    0.9006581D-02
    4    0.1012087E-02            4    0.1011016D-02
    5    0.1020432E-03            5    0.9256116D-04
    6    0.1103878E-03            6    0.7156936D-05
    7    0.1332998E-02            7    0.4790142D-06
    8    0.1988459E-01            8    0.2827691D-07
    9    0.3367050E+00            9    0.1693277D-08
x = 10.0000
L=  0    -.5440211E-01            0    -.5440211D-01
    1    0.7846694E-01            1    0.7846694D-01
    2    0.7794219E-01            2    0.7794219D-01
    3    -.3949584E-01            3    -.3949584D-01
    4    -.1055893E+00            4    -.1055893D+00
    5    -.5553451E-01            5    -.5553451D-01
    6    0.4450132E-01            6    0.4450132D-01
    7    0.1133862E+00            7    0.1133862D+00
    8    0.1255780E+00            8    0.1255780D+00
    9    0.1000964E+00            9    0.1000964D+00
x = 20.0000
L=  0    0.4564726E-01            0    0.4564726D-01
    1    -.1812174E-01            1    -.1812174D-01
    2    -.4836553E-01            2    -.4836552D-01
    3    0.6030358E-02            3    0.6030359D-02
    4    0.5047615E-01            4    0.5047615D-01
    5    0.1668391E-01            5    0.1668391D-01
    6    -.4130000E-01            6    -.4130000D-01
    7    -.4352891E-01            7    -.4352891D-01
    8    0.8653319E-02            8    0.8653319D-02
    9    0.5088423E-01            9    0.5088423D-01
```

Die Ergebnisse von Tabelle 1.3 zeigen deutlich, daß - insbesondere für kleine Argumente x - das verwendete Verfahren stark instabil ist. Diese Instabilität führt mit steigender Ordnung der Besselfunktion zu einem völligen Versagen des Verfahrens.
Für größere Argumente von x wird die Stabilität des Verfahrens zunehmend besser!

Um auch im Bereich kleiner Argumente brauchbare numerische Ergebnisse zu erhalten, muß von der bisher verwendeten „Vorwärts-Rekursion“ auf eine „Rückwärts-Rekursion“ übergegangen werden:

$$j_{L+1}(x) = 0 \qquad j_L(x) = \delta$$

$$j_l(x) = \frac{2l+3}{x} \cdot j_{l+1}(x) - j_{l+2}(x) \qquad l = L-1, L-2, \ldots, lmax, lmax-1, \ldots, 0$$

Dabei ist δ, eine beliebige (kleine) Zahl, der Startwert der Rückwärts-Rekursion. L ist eine natürliche Zahl > lmax. Es zeigt sich trotz der willkürlichen Startwerte der Rekursion, daß die Werte für kleiner werdende l sehr rasch gegen eine Zahlenfolge $\alpha \cdot j_l$ konvergieren. Die vorerst unbekannte Konstante α kann durch eine Normierung des Wertes für $l = 0$ auf die Größe $\sin x/x$ bestimmt werden.

Wegen der Willkür von δ sind die so erhaltenen Ergebnisse mit einem Verfahrensfehler behaftet, der umso kleiner ist, je größer der Startindex L gewählt wurde. Um eine Abschätzung von ϵ_V zu erhalten, wurde die Auswertung zweimal durchgeführt, und zwar mit $L = 18$ und mit $L = 27$. In beiden Fällen wurde die Rechnung sowohl mit einfacher als auch mit doppelter Genauigkeit durchgeführt.

Die Ergebnisse dieses Tests sind in der Tabelle 1.4 zusammengefaßt.
Es zeigt sich, daß die Rückwärts-Rekursion für den ganzen untersuchten x-Bereich einen sehr stabilen Algorithmus darstellt: Die Ergebnisse von einfacher und doppelter Genauigkeit unterscheiden sich erst in der siebenten Ziffer der Mantissen um maximal zwei Einheiten. Was den Verfahrensfehler betrifft, so ist mit steigendem Argument x eine steigende Tendenz dieses Fehlers zu beobachten.

Resumee aus den Tabellen 1.3 und 1.4:

- Für nicht zu große Argumente x (für $lmax = 9$ bis ca. $x = 10.$) ist wegen der größeren Stabilität unbedingt die Rückwärts-Rekursion zu verwenden.

- Erst für den Bereich großer x (für $lmax = 9$ ab ca. $x = 10.$) ist die Vorwärts-Rekursion wegen der nicht vorhandenen Verfahrensfehler vorzuziehen.

Tab.1.4: Numerische Berechnung der sphärischen Besselfunktionen der Ordnungen 0-9 mittels Rückwärts-Rekursion.

		einfache Genauigkeit		doppelte Genauigkeit	
		rueckwaerts(18)	rueckwaerts(27)	rueckwaerts(18)	rueckwaerts(27)
x = 0.3000					
L=	0	0.9850674E+00	0.9850674E+00	0.9850674D+00	0.9850674D+00
	1	0.9910290E-01	0.9910290E-01	0.9910289D-01	0.9910289D-01
	2	0.5961526E-02	0.5961525E-02	0.5961525D-02	0.5961525D-02
	3	0.2558598E-03	0.2558598E-03	0.2558598D-03	0.2558598D-03
	4	0.8536426E-05	0.8536425E-05	0.8536426D-05	0.8536426D-05
	5	0.2329583E-06	0.2329583E-06	0.2329583D-06	0.2329583D-06
	6	0.5378445E-08	0.5378444E-08	0.5378444D-08	0.5378444D-08
	7	0.1076069E-09	0.1076069E-09	0.1076069D-09	0.1076069D-09
	8	0.1899475E-11	0.1899474E-11	0.1899474D-11	0.1899474D-11
	9	0.2999847E-13	0.2999847E-13	0.2999847D-13	0.2999847D-13
x = 1.0000					
L=	0	0.8414710E+00	0.8414710E+00	0.8414710D+00	0.8414710D+00
	1	0.3011687E+00	0.3011687E+00	0.3011687D+00	0.3011687D+00
	2	0.6203505E-01	0.6203505E-01	0.6203505D-01	0.6203505D-01
	3	0.9006580E-02	0.9006579E-02	0.9006581D-02	0.9006581D-02
	4	0.1011016E-02	0.1011016E-02	0.1011016D-02	0.1011016D-02
	5	0.9256115E-04	0.9256114E-04	0.9256116D-04	0.9256116D-04
	6	0.7156936E-05	0.7156935E-05	0.7156936D-05	0.7156936D-05
	7	0.4790134E-06	0.4790133E-06	0.4790134D-06	0.4790134D-06
	8	0.2826499E-07	0.2826498E-07	0.2826499D-07	0.2826499D-07
	9	0.1491376E-08	0.1491376E-08	0.1491377D-08	0.1491377D-08
x = 10.0000					
L=	0	-.5440211E-01	-.5440211E-01	-.5440211D-01	-.5440211D-01
	1	0.7846695E-01	0.7846695E-01	0.7846695D-01	0.7846694D-01
	2	0.7794219E-01	0.7794220E-01	0.7794220D-01	0.7794219D-01
	3	-.3949586E-01	-.3949586E-01	-.3949585D-01	-.3949584D-01
	4	-.1055893E+00	-.1055893E+00	-.1055893D+00	-.1055893D+00
	5	-.5553451E-01	-.5553451E-01	-.5553451D-01	-.5553451D-01
	6	0.4450133E-01	0.4450133E-01	0.4450133D-01	0.4450132D-01
	7	0.1133862E+00	0.1133862E+00	0.1133862D+00	0.1133862D+00
	8	0.1255780E+00	0.1255780E+00	0.1255780D+00	0.1255780D+00
	9	0.1000964E+00	0.1000964E+00	0.1000964D+00	0.1000964D+00
x = 20.0000					
L=	0	0.4564727E-01	0.4564726E-01	0.4564726D-01	0.4564726D-01
	1	-.8801447E-01	-.1812270E-01	-.8801444D-01	-.1812269D-01
	2	-.5884944E-01	-.4836567E-01	-.5884943D-01	-.4836567D-01
	3	0.7330211E-01	0.6031280E-02	0.7330208D-01	0.6031273D-02
	4	0.8450518E-01	0.5047661E-01	0.8450516D-01	0.5047661D-01
	5	-.3527479E-01	0.1668320E-01	-.3527476D-01	0.1668320D-01
	6	-.1039063E+00	-.4130086E-01	-.1039063D+00	-.4130085D-01
	7	-.3226432E-01	-.4352875E-01	-.3226432D-01	-.4352875D-01
	8	0.7970807E-01	0.8654292E-02	0.7970804D-01	0.8654284D-02
	9	0.1000162E+00	0.5088490E-01	0.1000161D+00	0.5088490D-01

Kapitel 2

Numerische Methoden zur Lösung linearer, inhomogener Gleichungssysteme

2.1 Das grundsätzliche Problem

Gegeben sind die n Gleichungen

$$
\begin{array}{c}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\
\cdot \\
\cdot \\
\cdot \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n
\end{array}
\tag{2.1}
$$

wobei die a_{ij} und die b_i reellwertige Größen sind. Außerdem soll gelten:

$$| b_1 | + | b_2 | + \cdots + | b_n | \neq 0$$

Unter diesen Voraussetzungen stellt (2.1) ein *reellwertiges, lineares, inhomogenes Gleichungssystem n-ter Ordnung* dar. Die Größen $x_1, \ldots, x_n$, welche die gegebenen Gleichungen simultan erfüllen, nennt man die *Lösungen* des Systems.

(2.1) wird gewöhnlich in Form der Matrixgleichung

$$A \cdot \mathbf{x} = \mathbf{b} \tag{2.2}$$

angeschrieben.

Die Lösung derartiger Systeme stellt ein zentrales Problem der numerischen Mathematik dar, weil eine ganze Reihe von numerischen Verfahren, wie z.B. die Interpolationsverfahren, die Least-Squares-Verfahren, die Differenzenverfahren usw. auf die Lösung inhomogener linearer Gleichungssysteme zurückgeführt werden können.

Vom theoretischen Standpunkt aus bereitet die Lösung von (2.1) keinerlei Schwierigkeiten, solange die Determinante der *Koeffizientenmatrix* A nicht verschwindet, solange also das Problem nicht *singulär* ist:

$$\det(A) \neq 0$$

In diesem Fall kann das Problem mit Hilfe der *Cramer'schen Regel* angegangen werden. In der Praxis zeigt sich jedoch, daß die Anwendung dieser Regel für $n \geq 4$ recht kompliziert und aus verschiedenen Gründen für die Anwendung am Computer ungeeignet ist.

Wie auch in anderen Bereichen der numerischen Mathematik gibt es auch hier zwei Gruppen von Methoden:

- **Direkte Methoden:**

  Diese enthalten keine Verfahrensfehler und führen deshalb - abgesehen von Rundungsfehler-Einflüssen - stets zur exakten Lösung. Allerdings können Rundungsfehler gerade bei direkten Verfahren, die häufig sehr rechenintensive Algorithmen aufweisen, massiv in Erscheinung treten.

  Als Beispiel dazu wird im folgenden das *Eliminationsverfahren von Gauss* in der Formulierung von *Doolittle und Crout (LU decomposition)* behandelt.

- **Iterative Methoden**

  zeichnen sich oft durch einen besonders einfachen Rundungsfehler-stabilen und Speicherplatz-sparenden Algorithmus aus, sie haben jedoch den Nachteil, daß sie nicht in jedem Fall konvergieren, also nicht auf alle an sich lösbaren linearen Gleichungssysteme angewendet werden können.

  Außerdem erhält man auch im Falle der Konvergenz nie die exakte Problemlösung, sondern nur eine mit einem Abbruchfehler behaftete Näherungslösung.

  Als Beispiel für ein Iterationsverfahren zur Lösung linearer Gleichungssysteme steht in diesem Kapitel das *Gauss-Seidel-Verfahren.*

2.2 Ziel der direkten Methoden: Überführung der Koeffizientenmatrix in eine obere Dreiecksmatrix.

Bei allen direkten Methoden geht es letzten Endes darum, das gegebene System

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

in ein äquivalentes (d.h. denselben Lösungsvektor aufweisendes) System

$$U \cdot \mathbf{x} = \mathbf{y} \tag{2.3}$$

überzuführen, wobei U eine sogenannte *obere Dreiecksmatrix* mit der Eigenschaft

$$U = [u_{ij}] \qquad \text{mit} \qquad u_{ij} = 0 \qquad \text{für} \qquad i > j$$

darstellt. Der Grund dafür ist einfach: Systeme der Art (2.3) sind durch *Rück-Substitution* (s. Glg.en (2.12)und (2.13)) leicht aufzulösen.

2.3 Das Eliminationsverfahren von Gauss in der Formulierung von Doolittle und Crout (LU-Aufspaltung).

Die *Gauss'sche Elimination* besteht im wesentlichen aus zwei Schritten, nämlich aus der Überführung von $A\mathbf{x} = \mathbf{b}$ in das äquivalente System
$U\mathbf{x} = \mathbf{y}$ und der Lösung dieses Systems mittels einer Rück-Substitution.

Das Verfahren beruht auf einem Satz aus der linearen Algebra:
Ein lineares Gleichungssystem bleibt unverändert, wenn zu einer seiner Gleichungen eine Linearkombination der restlichen Zeilen addiert wird.

Doolittle und Crout haben nun gezeigt, daß der Gauss'sche Algorithmus wie folgt formuliert werden kann: Eine reelle Matrix A kann stets als Produkt zweier reeller Matrizen L und U dargestellt werden, also

$$A = L \cdot U \tag{2.4}$$

wobei U eine *obere* Dreieckmatrix ist:

$$U = \begin{pmatrix} u_{11} & u_{12} & .... & u_{1n} \\ 0 & u_{22} & .... & u_{1n} \\ . & & & . \\ . & & & . \\ . & & & . \\ 0 & 0 & .... & u_{nn} \end{pmatrix}$$

Die Matrix L stellt eine *untere* Dreiecksmatrix der Form

$$L = \begin{pmatrix} 1 & 0 & 0 & .... & 0 \\ m_{21} & 1 & .... & 0 & \\ m_{31} & m_{32} & 1 & .... & 0 \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ m_{n1} & m_{n2} & m_{n3} & .... & 1 \end{pmatrix}$$

dar.

Man nennt die Zerlegung (2.4) einer Matrix A die *LU- Zerlegung (LU decomposition)* von A.

<u>Ohne Ableitung</u>: ein einfacher Formelapparat für die LU-Zerlegung.

Die Umformung erfolgt spaltenweise, also $j = 1, 2, \ldots, n$:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} m_{ik} u_{kj} \qquad i = 1, \ldots, j-1 \tag{2.5}$$

$$\gamma_{ij} = a_{ij} - \sum_{k=1}^{j-1} m_{ik} u_{kj} \qquad i = j, \ldots, n \tag{2.6}$$

$$u_{jj} = \gamma_{jj} \tag{2.7}$$

$$m_{ij} = \frac{\gamma_{ij}}{\gamma_{jj}} \qquad i = j+1, \ldots, n \tag{2.8}$$

Da die Hauptdiagonalelemente von L stets Einser sind, braucht man diese Werte (die m_{jj}) nicht extra abzuspeichern, sondern man kann alle relevanten Werte in einer n x n-Matrix (der 'LU-Matrix') unterbringen:

$$\text{LU-Matrix}: \begin{pmatrix} u_{11} & u_{12} & u_{13} & \ldots\ldots & u_{1n} \\ m_{21} & u_{22} & u_{23} & \ldots\ldots & u_{2n} \\ m_{31} & m_{32} & u_{33} & \ldots\ldots & u_{3n} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ m_{n1} & m_{n2} & m_{n3} & \ldots\ldots & u_{nn} \end{pmatrix} \tag{2.9}$$

Die Aufspaltung der Matrix A in die Matrizen L und U kann nun zur Berechnung des Lösungsvektors herangezogen werden:

$$A \cdot \mathbf{x} \equiv L \cdot U \cdot \mathbf{x} = L \cdot (U \cdot \mathbf{x}) = \mathbf{b}$$

Mit $U \cdot \mathbf{x} \equiv \mathbf{y}$ erhält man das System $L \cdot \mathbf{y} = \mathbf{b}$, das wegen der besonderen Form der Matrix L (untere Dreiecksmatrix) mittels einer *Vorwärts-Substitution* gelöst werden kann:

$$y_1 = b_1 \tag{2.10}$$

$$y_i = b_i - \sum_{j=1}^{i-1} m_{ij} y_j \qquad i = 2, 3, \ldots, n \tag{2.11}$$

Da U eine obere Dreiecksmatrix ist, ist bei Kenntnis des Hilfsvektors $\mathbf{y}$ das System $U \cdot \mathbf{x} = \mathbf{y}$ durch *Rückwärts-Substitution* ebenfalls leicht zu lösen:

$$x_n = \frac{y_n}{u_{nn}} \tag{2.12}$$

$$x_i = \frac{1}{u_{ii}} \left[y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right] \qquad i = n-1, n-2, \ldots, 1 \tag{2.13}$$

Bei der konkreten Anwendung des Doolittle-Crout-Verfahrens geht man am besten so vor, daß man zwei Programme verwendet, von denen eines die LU-Zerlegung der Koeffizientenmatrix durchführt, während das zweite den Lösungsvektor unter Verwendung der Formeln (2.10-2.13) berechnet.

2.3.1 Demonstration einer speicherplatz-sparenden LU-Decomposition

an Hand einer allgemeinen 3x3-Matrix.

Gegeben:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

j = 1: erste Spalte

(2.6) i=1 $\gamma_{11} = a_{11}$
i=2 $\gamma_{21} = a_{21}$
i=3 $\gamma_{31} = a_{31}$

.

.

(2.7) $u_{11} = \gamma_{11}$

.

.

(2.8) i=2 $m_{21} = \gamma_{21}/\gamma_{11}$
i=3 $m_{31} = \gamma_{31}/\gamma_{11}$

Speichert man nun die eben berechneten Koeffizienten u_{11}, m_{21} und m_{31} *anstelle der entsprechenden und nie wieder gebrauchten Koeffizienten der Matrix A ab*, so ergibt sich die Matrix

$$\begin{pmatrix} u_{11} & a_{12} & a_{13} \\ m_{21} & a_{22} & a_{23} \\ m_{31} & a_{32} & a_{33} \end{pmatrix}$$

j=2: zweite Spalte

(2.5) i=1 $u_{12} = a_{12}$

.

.

(2.6) i=2 $\gamma_{22} = a_{22} - m_{21}u_{12}$
i=3 $\gamma_{32} = a_{32} - m_{31}u_{12}$

.

.

(2.7) $u_{22} = \gamma_{22}$

.

.

(2.8) i=3 $m_{32} = \gamma_{32}/\gamma_{22}$

$$\begin{pmatrix} u_{11} & u_{12} & a_{13} \\ m_{21} & u_{22} & a_{23} \\ m_{31} & m_{32} & a_{33} \end{pmatrix}$$

j=3: dritte Spalte

(2.5) i=1 $\quad u_{13} = a_{13}$

i=2 $\quad u_{23} = a_{23} - m_{21}u_{13}$

.

.

(2.6)i=3 $\quad \gamma_{33} = a_{33} - m_{31}u_{13} - m_{32}u_{23}$

.

.

(2.7) $\quad u_{33} = \gamma_{33}$

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ m_{21} & u_{22} & u_{23} \\ m_{31} & m_{32} & u_{33} \end{pmatrix} = \text{LU-Matrix} \quad [\text{s. Glg.}(2.9)]$$

Entscheidend ist also, daß durch die sukzessive Überspeicherung der gegebenen Koeffizientenmatrix A durch die Spalte für Spalte entstehende LU-Matrix *zu keiner Zeit ein Informationsmanko entsteht!*

Da andererseits auch die neuen u_{ij}- bzw. m_{ij}-Werte stets verschiedene Speicherplätze einnehmen, *kann im Programm auf eine unterschiedliche Bezeichnung der Elemente a_{ij}, u_{ij}, γ_{ij} und m_{ij} überhaupt verzichtet werden, und alle diese Elemente werden im Struktogramm 3 (LUDCMP) auf <u>einem</u> Feld A(,) abgespeichert!*

Man spart damit sehr viel Speicherplatz, allerdings auf Kosten der Lesbarkeit des Programms.

2.3.2 Rundungsfehler-Optimierung durch partielle Pivotisierung.

Angenommen, es existiert ein Programm, das die Auswertung eines inhomogenen Gleichungssystems nach der LU-Zerlegung durchführt. Die mittels dieses Programmes erhaltenen numerischen Ergebnisse enthalten keine Verfahrensfehler (direktes Verfahren!); jede Abweichung von den 'wahren' Ergebnissen muß daher auf Rundungsfehler zurückgehen.

Dazu ein konkretes Beispiel: Gegeben sei das folgende System von 3 Gleichungen:

$$\begin{aligned} x_1 + 5923181x_2 + 1608x_3 &= 5924790 \\ 5923181x_1 + 337116x_2 - 7x_3 &= 6260290 \\ 6114x_1 + 2x_2 + 9101372x_3 &= 9107488 \end{aligned} \tag{2.14}$$

Dieses System hat die exakte Lösung $x_1 = x_2 = x_3 = 1$.

Die numerische Auswertung (FORTAN, einfache Genauigkeit, d.h. 4 Byte pro reeller Zahl) liefert die folgende LU-Matrix und den folgenden Lösungsvektor:

	0.1000000E+01	0.5923181E+07	0.1608000E+04
LU-Matrix:	0.5923181E+07	-.3508407E+14	-.9524475E+10
	0.6114000E+04	0.1032216E-02	0.9101371E+07

	0.9398398E+00
x:	0.1000000E+01
	0.9995983E+00

Wie man sieht, ist der Rundungsfehler-Einfluß (besonders beim x_1) beträchtlich!

Wenn man nun die Reihenfolge der Gleichungen in (2.14) abändert (was theoretisch natürlich keinen Einfluß auf den Lösungsvektor hat), und zwar so, daß die ursprüngliche Reihenfolge 1/2/3 übergeht in die Reihenfolge 2/3/1, so ergibt dasselbe Programm:

	0.5923181E+07	0.3371160E+06	-.7000000E+01
LU-Matrix:	0.1032216E-02	-.3459764E+03	0.9101372E+07
	0.1688282E-06	-.1712019E+05	0.1558172E+12

mit

	0.9999961E+00
x:	0.1000068E+01
	0.1000000E+01

bereits wesentlich kleineren Rundungsfehlern. Mit der Zeilenfolge 2/1/3 erhält man den korrekten Lösungsvektor in Maschinengenauigkeit:

	0.5923181E+07	0.3371160E+06	-.7000000E+01
LU-Matrix:	0.1688282E-06	0.5923181E+07	0.1608000E+04
	0.1032216E-02	-.5841058E-04	0.9101372E+07

	0.1000000E+01
x:	0.1000000E+01
	0.1000000E+01

Das bedeutet also, daß die richtige Zeilenfolge eines linearen Gleichungssystems einen entscheidenden Einfluß auf die Größe der Rundungsfehler und somit auf die Qualität der numerischen Lösung hat!

Was bedeutet nun 'richtige' Reihenfolge? Vergleicht man die m_{ij}-Werte in den LU-Matrizen, so erkennt man sofort, daß die Rundungsfehler dann am geringsten sind, *wenn die Beträge der m_{ij}-Werte möglichst klein sind.* Um die m_{ij} so klein als möglich zu halten, müssen die in der Gleichung (2.8) auftretenden γ_{jj}-Werte dem Betrage nach möglichst groß sein. Dies kann einfach dadurch erreicht werden, daß man aus allen γ_{ij} nach Glg. (2.6) das betragsgrößte Element bestimmt und jene Zeile, in welcher dieses Element vorkommt, mit der j-ten Zeile vertauscht. Erst danach werden die Gleichungen (2.7) und (2.8) ausgewertet.

Eine derartige Strategie nennt man *eine LU-Zerlegung mit teilweiser Pivotisierung.* In der Regel hat man – vor allem bei Systemen höherer Ordnung – ohne Pivotisierung keine Chance auf korrekte Ergebnisse. Es gibt aber auch (s. [2], S. 56) Gleichungssysteme, bei welchen die LU-Decomposition auch ohne Pivotisierung numerisch stabil ist, nämlich dann, wenn die Koeffizientenmatrix *symmetrisch, tridiagonal, zyklisch tridiagonal, diagonal-dominant*

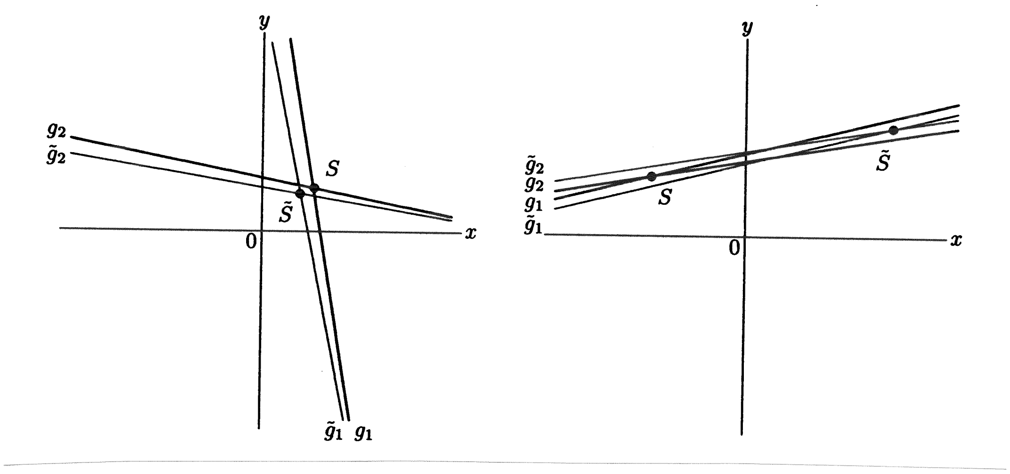


Abbildung 2.1: Gut und schlecht konditioniertes 2x2-System (aus: [22], S. 228).

bzw. allgemein positiv definit ist.

Es ist klar, daß mit der Strategie der teilweisen Pivotisierung auch das Nullwerden eines Diagonalelementes γ_{jj} und die damit auftretenden Probleme in Glg.(2.8) vermieden werden können. Stellt sich nämlich heraus, daß während der Rechnung *alle* $\gamma_{ij}, \quad i = j, \ldots, n$ *simultan* verschwinden, ist die Koeffizientenmatrix *singulär* und das Gleichungssystem mit dem gegebenen Verfahren nicht lösbar!

2.3.3 Kondition eines Gleichungssystems.

Die mit Hilfe direkter Methoden ermittelte Lösung eines linearen Gleichungssystems ist meist nicht die exakte Lösung, weil

- im Verlauf der Rechnung Rundungsfehler auftreten, deren Akkumulation zur Verfälschung der Ergebnisse führen kann.
- Ungenauigkeiten in den Ausgangsgrößen bestehen können, die Ungenauigkeiten in den Lösungen hervorrufen.

Wenn kleine Änderungen in den Ausgangsdaten große Änderungen in der Lösung hervorrufen, spricht man von einem *schlecht konditionierten* System (s. Abb. 2.1).
Die sog. *Konditionszahl* der Koeffizientenmatrix kann dabei als Maß für die Güte der erhaltenen Näherungslösung für den Lösungsvektor verwendet werden.

In der mathematischen Literatur wird eine Reihe von Konditionszahlen diskutiert. Im folgenden Programm LUDCMP ist die Berechnung der sog. *Hadamarschen Konditionszahl* der Matrix A inkludiert, welche in der Form [2]

$$K_{\mathrm{H}}(A) = \frac{|det(A)|}{\alpha_1 \alpha_2 \cdots \alpha_n} \qquad \text{mit} \qquad \alpha_i = \sqrt{a_{i1}^2 + a_{i2}^2 + \ldots + a_{in}^2}$$

gegeben ist. Für die Praxis gelten die Erfahrungswerte

- $K_{\mathrm{H}}(A) < 0.01$ schlechte Kondition,
- $K_{\mathrm{H}}(A) > 0.1$ gute Kondition,
- $0.01 \leq K_{\mathrm{H}}(A) \leq 0.1$ keine Aussage.

2.3.4 Das Programm LUDCMP

Quelle: [9], S.38f; [10], S. 46f (mit geringen Änderungen).

Das Programm LUDCMP (LU DeCoMPosition) führt eine LU-Aufspaltung einer reellen quadratischen Matrix durch.

INPUT Parameter:

A(,): Koeffizientenmatrix des linearen Systems.

N: Ordnung des Systems = Anzahl der Zeilen und Spalten von A.

OUTPUT Parameter:

A(,): 'LU Matrix' der ursprünglichen Matrix A (s. Anmerkung 1).

INDX( ): Indexvektor, der die Zeilenvertauschungen auf Grund der partiellen Pivotisierung speichert.

D: Determinante der Koeffizientenmatrix (s. Abschnitt 2.3.6).

KHAD: Hadamardsche Konditionszahl (s. Abschnitt 2.3.2).

Struktogramm 3 — LUDCMP(A,N,INDX,D,KHAD)

```
KHAD:=1.0
I=1(1)N
    SUM:=0.0
    J=1(1)N
        SUM:=SUM + A(I,J)**2
    KHAD:=KHAD*SQRT(SUM)
D:=1.0
J=1(1)N
    I=1(1)J-1
        SUM:=A(I,J)
        K=1(1)I-1
            SUM:=SUM-A(I,K)*A(K,J)
        A(I,J):=SUM
    AAMAX:=0.0
    I=J(1)N
        SUM:=A(I,J)
        K=1(1)J-1
            SUM:=SUM-A(I,K)*A(K,J)
        A(I,J):=SUM     DUM:=| SUM|
        DUM ≥ AAMAX
        Y: IMAX:=I     AAMAX:=DUM
        N: ......
    J ≠ IMAX
    Y: K=1(1)N
           DUM:=A(IMAX,K)
           A(IMAX,K):=A(J,K)
           A(J,K):=DUM
       D:=–D
    N: ......
    INDX(J):=IMAX
    A(J,J) = 0.0
    Y: A(J,J):=TINY
    N: ......
```

. — .

```
.
    DUM:=1.0/A(J,J)
    I=J+1(1)N
        A(I,J):=A(I,J)*DUM
    D:=D*A(J,J)
KHAD:=| D |/KHAD
(return)
```

Anmerkungen zum Programm LUDCMP:

1. Um Speicherplatz zu sparen, werden die Koeffizienten der LU Matrix sukzessive auf den entsprechenden Speicherplätzen der gegebenen Matrix A abgespeichert. Dies hat allerdings den Nachteil, daß die gegebene Matrix zerstört wird und daher gegebenenfalls (z.B. wenn A für spätere Rechnungen wieder gebraucht wird) **vor** Aufruf von LUDCMP gesichert werden muß.

2. Der Output-Parameter D hat Bedeutung bei der Berechnung der Determinante von A.

3. Die Konstante TINY ist in [9] mit 10^{-20} definiert. Wie aus dem Struktogramm 3 hervorgeht, verhindert sie ein crash-down des Programmes, wenn ein Divisor in Glg.(2.8) exakt Null wird. Zu diesem Thema siehe Kap.2.5.

4. In [9] werden die Matrixkoeffizienten für die Pivotsuche zeilenweise skaliert. Da optimale Skalierungen von linearen Systemen ein überraschend schwieriges Problem darstellen (s. z.B. [7], S.140ff und [12], S.41ff), wurde diese Skalierung aus dem hier präsentierten Programm herausgenommen.

2.3.5 Das Unterprogramm LUBKSB

Quelle: [9], S. 39; [10], S. 47 (mit Änderungen).

Das Programm LUBKSB (LU BacKSuBstitution) berechnet den Lösungsvektor $\mathbf{x}$ des Systems $LU\mathbf{x} = \mathbf{b}$ unter Verwendung der Gleichungen (2.10 - 2.13).

INPUT-Parameter:

A(,): LU-Matrix der Koeffizientenmatrix, wie sie z.B. vom Programm LUDCMP berechnet wird.

N: Ordnung des Systems = Zeilen und Spalten von A.

INDX( ): Indexvektor, der Informationen über die von LUDCMP vorgenommenen Zeilenvertauschungen enthält.

B( ): inhomogener Vektor des Systems.

OUTPUT Parameter:

X( ): Lösungsvektor.

Interner Vektor:

Y( ): .

Struktogramm 4 — LUBKSB(A,N,INDX,B,X)

```
I=1(1)N
    BB(I):=B(I)
I=1(1)N
    LL:=INDX(I)
    SUM:=BB(LL)
    BB(LL):=BB(I)
    J=1(1)I-1
        SUM:=SUM-A(I,J)*Y(J)
    Y(I):=SUM
I=N(-1)1
    SUM:=Y(I)
    J=I+1(1)N
        SUM:=SUM-A(I,J)*X(J)
    X(I):=SUM/A(I,I)
(return)
```

2.3.6 Die Verwendungsmöglichkeiten für die Programme LUDCMP und LUBKSB.

Lösung eines inhomogenen Gleichungssystems $A\mathbf{x} = \mathbf{b}$:

```
||---------------------------||
||  LUDCMP(A,N,INDX,D,KHAD)  ||
||---------------------------||
||   LUBKSB(A,N,INDX,B,X)    ||
||---------------------------||
```

Ein wesentlicher Vorteil des hier vorgestellten Verfahrens besteht darin, daß bei Auftreten verschiedener inhomogener Vektoren $\mathbf{b}_1$, $\mathbf{b}_2$, ..., aber gleichbleibender Koeffizientenmatrix A die LU-Zerlegung nur einmal durchgeführt werden muß:

```
||-------------------------||
|| LUDCMP(A,N,INDX,D,KHAD) ||
||-------------------------||
             |
||-------------------------||
|| LUBKSB(A,N,INDX,B1,X1)  ||
|| LUBKSB(A,N,INDX,B2,X2)  ||
||           .             ||
||           .             ||
||-------------------------||
```

Invertierung einer Matrix:

Mit Hilfe der Programme LUDCMP und LUBKSB kann eine gegebene Matrix A *Spalte für Spalte* invertiert werden:

$$A \cdot X = E \qquad E = \text{Einheitsmatrix} \qquad X \equiv A^{-1}$$

Diese Matrixgleichung kann man nun in eine Reihe von inhomogenen linearen Gleichungssystemen der Art

$$A \begin{pmatrix} x_{11} \\ . \\ . \\ . \\ x_{n1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ . \\ . \\ 0 \end{pmatrix} \quad \cdots \quad A \begin{pmatrix} x_{1n} \\ . \\ . \\ . \\ x_{nn} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ . \\ . \\ 1 \end{pmatrix}$$

zerlegen, d.h. man hat n Systeme mit gleicher Koeffizientenmatrix A und mit wechselnden inhomogenen Vektoren (= Spaltenvektoren der Einheitsmatrix) zu berechnen:

Struktogramm — Invertierung einer Matrix

```
LUDCMP (A,N,INDX,D,KHAD)
J=1(1)N
    I=1(1)N
        VEKTOR(I):=0.0
    VEKTOR(J):=1.0
    LUBKSB (A,N,INDX,VEKTOR,X)
    I=1(1)N
        AINV(I,J):=X(I)
```

Berechnung der Determinante einer Matrix:

Für die Determinante der Matrix A gilt

$$det(A) = det(L \cdot U) = det(L)\, det(U)\,.$$

Weil beide Matrizen L und U Dreiecksmatrizen sind, sind deren Determinanten gleich den jeweiligen Produkten der Diagonalelemente. Weil aber alle Diagonalelemente der L-Matrix den Wert 1 haben, gilt

$$det(L) = 1\,,$$

und es ergibt sich

$$det(A) = det(U) = \sum_{i=1}^{n} u_{ii}\,.$$

Die Berechnung von $det(U)$ erfolgt während der LU-Dekomposition im Programm LUDCMP; dabei ist zu berücksichtigen, daß bei jeder Zeilenvertauschung sich das <u>Vorzeichen</u> der Determinante ändert.

2.3.7 Testbeispiele für die Programme LUDCMP und LUBKSB.

Gegeben ist die Matrix mit vier Zeilen und Spalten

$$A = \begin{pmatrix} 1.1161 & 0.1254 & 0.1397 & 0.1490 \\ 0.1582 & 1.1675 & 0.1768 & 0.1871 \\ 0.1968 & 0.2071 & 1.2168 & 0.2271 \\ 0.2368 & 0.2471 & 0.2568 & 1.2671 \end{pmatrix}$$

Die Konditionszahl lautet $K_{\mathrm{H}} = 0.752$, das System ist also *gut konditioniert!*

- Wie lautet der Lösungsvektor $\mathbf{x}$ des linearen inhomogenen Gleichungssystems

$$A \cdot \mathbf{x} = (-1.8367, 1.1944, 3.2368, -0.7232)^T \quad ?$$

Das numerische Ergebnis ist bis auf Maschinengenauigkeit exakt:

$$\mathbf{x} = (-.2000000E{+}01, 0.1000000E{+}01, 0.3000000E{+}01, -.1000000E{+}01)^T$$

- Wie lautet die inverse Matrix von A?

Das numerische Ergebnis ist fast bis auf Maschinengenauigkeit exakt (bis auf maximal zwei Einheiten in der siebenten Stelle hinterm Komma) und lautet:

```
Invertierte Matrix:

  .9379443E+00   -.6843720E-01   -.7960770E-01   -.8592076E-01
 -.8852433E-01    .9059826E+00   -.9919081E-01   -.1055899E+00
 -.1113511E+00   -.1169667E+00    .8784252E+00   -.1270733E+00
 -.1354557E+00   -.1401826E+00   -.1438075E+00    .8516058E+00

Test-Matrix:

  .9999999E+00    .1126516E-07    .2518815E-08   -.1120211E-08
 -.4888310E-08    .1000000E+01   -.6621389E-08   -.3829147E-08
 -.2647183E-08    .5092004E-08    .9999999E+00    .1665558E-07
  .7356128E-08   -.1727934E-07   -.1442864E-07    .1000000E+01
```

Die Test-Matrix stellt einfach das Multiplikations-Ergebnis: gegebene Matrix mal (numerisch) invertierte Matrix dar. Zu erwarten wäre im Idealfall die Einheitsmatrix.

- Welchen Wert hat die Determinante der Matrix A?

Das numerische Ergebnis (auf Maschinengenauigkeit exakt) lautet

$$\text{Determinante} = 0.1758306E + 01$$

2.4 Verbesserung eines Lösungsvektors durch Nachiteration.

Es ist klar, daß die Genauigkeit der numerischen Lösung eines linearen Gleichungssystems durch die Maschinengenauigkeit begrenzt ist. Leider ist es in vielen Fällen so, daß diese Genauigkeit wegen auftretender Rundungsfehler-Kumulationen nicht erreicht wird.

In solchen Fällen gibt es die Möglichkeit einer *iterativen Verbesserung* der numerischen Lösung, deren (sehr einfache) Theorie nun kurz erläutert wird:

Die numerische Lösung des Systems

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

sei der durch Rundungsfehler verfälschte Vektor $\bar{\mathbf{x}} = \mathbf{x} + \delta\mathbf{x}$. Setzt man nun $\bar{\mathbf{x}}$ in das Gleichungssystem ein, so erhält man den verfälschten inhomogenen Vektor $\mathbf{b} + \delta\mathbf{b}$:

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

bzw.

$$\underbrace{(A\mathbf{x} - \mathbf{b})}_{=0} + A \cdot \delta\mathbf{x} = \delta\mathbf{b}\,.$$

Somit ergibt sich der Fehlervektor $\delta\mathbf{x}$ als Lösung des Systems

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

wobei A die *unveränderte* Koeffizientenmatrix und $\delta\mathbf{b}$ den sogenannten *Residuenvektor* $A\bar{\mathbf{x}} - \mathbf{b}$ darstellt.

Das Struktogramm 5 zeigt die Lösung eines linearen Gleichungssystems mit anschließender iterativer Verbesserung des Lösungsvektors:

$$\mathbf{x} = \bar{\mathbf{x}} - \delta\mathbf{x} \quad .$$

Dazu noch einige Anmerkungen:

1. Da beim ersten Aufruf von LUDCMP die Matrix A zerstört wird, muß diese Größe rechtzeitig gesichert werden.

2. Wegen der großen Gefahr einer *subtractive cancellation* sollte die Berechnung des Residuenvektors - wenn irgend möglich - mit doppelter Genauigkeit erfolgen.

3. Manche Programm-Bibliotheken (z.B. NAG) enthalten Programme zur Lösung inhomogener Gleichungssysteme, die eine iterative Verbesserung bereits inkludiert haben.

Struktogramm 5 — Iterative Verbesserung des Lösungsvektors

```
I=1(1)N
    J=1(1)N
        ASP(I,J):=A(I,J)
LUDCMP (A,N,INDX,D,KHAD)
LUBKSB (A,N,INDX,B,X)
    I=1(1)N
        SUMDP:=-B(I)
        J=1(1)N
            SUMDP:=SUMDP + DBLE(ASP(I,J))*DBLE(X(J))
        RES(I):=SUMDP
    LUBKSB (A,N,INDX,RES,DELX)
    I=1(1)N
        X(I):=X(I)-DELX(I)
(Bedingung, unter welcher die Iteration
verlassen wird)
(Ausgabe und return)
```

2.5 Rundungsfehler-Probleme mit schlecht konditionierten bzw. singulären Systemen.

Die Methode der partiellen Pivotisierung bewirkt eine entscheidende Reduktion der beim Gauss'schen Verfahren auftretenden Rundungsfehler. Dennoch kann bei *schlecht konditionierten* Gleichungssystemen der numerisch ermittelte Lösungsvektor deutlich fehlerbehaftet sein, wie das folgende Beispiel zeigt:

```
0.10000E+01 0.50000E+00 0.33333E+00 0.25000E+00 0.20000E+00   0.228333E+01

0.50000E+00 0.33333E+00 0.25000E+00 0.20000E+00 0.16667E+00   0.145000E+01

0.33333E+00 0.25000E+00 0.20000E+00 0.16667E+00 0.14286E+00   0.109286E+01

0.25000E+00 0.20000E+00 0.16667E+00 0.14286E+00 0.12500E+00   0.884530E+00

0.20000E+00 0.16667E+00 0.14286E+00 0.12500E+00 0.11111E+00   0.745640E+00
```

Die Koeffizientenmatrix stellt eine *Hilbert-Matrix* 5. Ordnung mit den auf 5 signifikante Stellen gerundeten Komponenten

$$a_{ij} = \frac{1}{i+j-1}$$

dar, und der inhomogene Vektor ist so gewählt, daß die exakte Lösung

$$x_1 = x_2 = x_3 = x_4 = x_5 = 1$$

lautet. Die Konditionszahl lautet $K_{\mathrm{H}} = 0.55{\cdot}10^{-10}$; es liegt also ein sehr schlecht konditioniertes System vor, und Schwierigkeiten sind zu erwarten!

Das exakte Ergebnis wird im folgenden dem numerischen Ergebnis $\bar{\mathbf{x}}$ gegenübergestellt (FORTRAN, single precision). Weiters ist zur Kontrolle der Vektor $A \cdot \bar{\mathbf{x}}$ ausgegeben:

$\mathbf{x}$ (exakt)	$\bar{\mathbf{x}}$ (numerisch)	$A \cdot \bar{\mathbf{x}}$
1.	0.9999459E+00	0.2283330E+01
1.	0.1000955E+01	0.1450000E+01
1.	0.9960009E+00	0.1092860E+01
1.	0.1005933E+01	0.8845300E+00
1.	0.9971328E+00	0.7456400E+00

Das Dilemma ist deutlich zu sehen: Obwohl einige Komponenten des numerisch erhaltenen Lösungsvektors beträchtlich vom exakten Wert abweichen, ist das lineare Gleichungssystem mit Maschinengenauigkeit erfüllt! Auch eine Nachiteration bringt keine Verbesserung der Ergebnisse.

Dieses Beispiel enthüllt noch ein weiteres ernstes Problem bei der Verwendung numerischer Methoden zur Lösung linearer Systeme: Es gibt trotz zahlreicher theoretischer Untersuchungen (s. z.B. [12], S.109ff) keine einfache, also in der Praxis einsetzbare Abschätzung des im Lösungsvektor enthaltenen Rundungsfehlers.

Eine weitere unangenehme Rundungsfehler-Konsequenz besteht darin, daß es bei der numerischen Auswertung praktisch unmöglich ist, klar zwischen *singulären* und *fast-singulären* Koeffizientenmatrizen zu unterscheiden. Im Programm wird eine Matrix dann als singulär erkannt, wenn alle Elemente einer Pivot-Spalte exakt Null sind. Da aber diese Elemente ihrerseits Ergebnisse arithmetischer Operationen sind, ist wegen der dabei auftretenden

Rundungsfehler *auch im Falle der Singularität der Koeffizientenmatrix* nicht mit exakten Nullen zu rechnen (das folgende aus [12], S.63):

Die Lage wird dadurch besonders kompliziert, daß die scharfe mathematische Unterscheidung von singulären und nicht-singulären Matrizen nur in der Idealwelt des Mathematikers existiert. Sobald wir mit Matrizen in gerundeter Arithmetik arbeiten, wird die Unterscheidung notwendigerweise ungerechtfertigt. So können gewisse nicht-singuläre Matrizen als Ergebnis von durch die Rundungsfehler eingeführten Störungen singulär werden. Noch wahrscheinlicher wird eine wirklich singuläre vorgegebene Matrix durch den Rundungsfehler in eine benachbarte nicht-singuläre Matrix abgeändert.

Aus diesem Grund verzichten viele Programme auf eine Singularitäts-Diagnose und überlassen diese dem Benutzer.
Im Programm LUDCMP wird zwar abgefragt, ob im Laufe der Rechnung irgendein Diagonalelement γ_{jj} exakt(!) Null ist, aber nur, um in diesem (unwahrscheinlichen) Fall eine Division durch Null zu verhindern!

2.6 Direkte Lösungsverfahren für Systeme mit speziellen Koeffizientenmatrizen.

Nachdem direkte Verfahren zur Lösung linearer Gleichungssysteme relativ rechen- und speicherplatz-intensiv sind, ist es in der Praxis von sehr großer Bedeutung, spezielle Koeffizientenmatrizen auszunutzen, um vereinfachte Algorithmen zu formulieren.

2.6.1 Lösung von Gleichungssystemen mit tridiagonaler Koeffizientenmatrix.

Eine Reihe wichtiger numerischer Verfahren (z.B. die Spline-Interpolation) führen zu linearen Gleichungssystemen der Form

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \cdots & 0 \\ a_2 & b_2 & c_2 & 0 & \cdots & 0 \\ 0 & a_3 & b_3 & c_3 & \cdots & 0 \\ . & & & & & . \\ . & & & & & . \\ . & & & & & c_{n-1} \\ 0 & 0 & 0 & 0 & \cdots & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ . \\ . \\ . \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ . \\ . \\ . \\ r_n \end{pmatrix} \tag{2.15}$$

wobei die *tridiagonale* Koeffizientenmatrix eindeutig durch die drei Vektoren **b** (Hauptdiagonale) sowie **a** und **c** (untere bzw. obere Nebendiagonale) gegeben ist. Wendet man in diesem Fall die LU-Zerlegung (<u>ohne</u> Pivotisierung, s.u.) an, so erhält man die Struktur

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_2 & 1 & 0 & \cdots & 0 \\ 0 & m_3 & 1 & \cdots & 0 \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \underbrace{\begin{pmatrix} u_1 & c_1 & 0 & \cdots & 0 \\ 0 & u_2 & c_2 & \cdots & 0 \\ . & & & & . \\ . & & & & . \\ . & & & & c_{n-1} \\ 0 & 0 & 0 & \cdots & u_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{pmatrix}}_{\equiv \mathbf{y}} = \begin{pmatrix} r_1 \\ r_2 \\ . \\ . \\ . \\ r_n \end{pmatrix}$$

wobei sich bei Anwendung der Glg.en (2.5-2.13) die folgenden einfachen Formeln ergeben:

$$\begin{aligned} u_1 &= b_1 \\ y_1 &= r_1 \\ \\ m_j &= a_j/u_{j-1} \\ u_j &= b_j - m_j \cdot c_{j-1} \qquad j = 2, \ldots, n \\ y_j &= r_j - m_j \cdot y_{j-1} \end{aligned} \tag{2.16}$$

Die Komponenten des Lösungsvektors ergeben sich wieder mittels Rück-Substitution:

$$\begin{aligned} x_n &= y_n/u_n \\ x_j &= (y_j - c_j \cdot x_{j+1})/u_j \qquad j = n-1, \ldots, 1 \end{aligned} \tag{2.17}$$

Diese Gleichungen sind die Basis für das folgende Programm.

2.6.2 Das Programm TRID.

Quelle: [9], S. 43 (mit Änderungen).

Das Programm TRID berechnet den Lösungsvektor eines linearen, inhomogenen Gleichungssystems mit tridiagonaler Koeffizientenmatrix.

INPUT-Parameter:

A( ),B( ),C( ): Vektoren der tridiagonalen Matrix.

R( ): inhomogener Vektor.

N: Ordnung des Systems.

OUTPUT-Parameter:

X( ): Lösungsvektor.

Interne Vektoren:

U( ), Y( ): .

Struktogramm 6 — TRID(A,B,C,R,N,X)

```
Y(1):=R(1)
U(1):=B(1)
U(1) = 0.0
  Y: (Abbruch mit Fehlermitteilung!)
  N: ......
J=2(1)N
    M:=A(J)/U(J-1)
    U(J):=B(J)-M*C(J-1)
    U(J) = 0.0
      Y: (Abbruch mit Fehlermitteilung!)
      N: ......
    Y(J):=R(J)-M*Y(J-1)
X(N):=Y(N)/U(N)
J=N-1(-1)1
    X(J):=(Y(J)-C(J)*X(J+1))/U(J)
(return)
```

Anmerkungen zu TRID:

1. Der Hauptvorteil von TRID liegt natürlich in einer beträchtlichen Speicherplatz-Ersparnis, da ja als Input-Informationen nur die vier eindimensionalen Felder A, B, C und R eingegeben werden müssen. Im Programm kommen dazu noch die ebenfalls eindimensionalen internen Felder U und Y sowie der Lösungsvektor X. Bei einem Problem n-ter Ordnung bedeutet das einen Speicherplatzbedarf von $7n$ gegenüber $n^2 + n$ beim normalen LU-Verfahren.

2. Ein weiterer Vorteil gegenüber dem normalen LU-Verfahren ist die beträchtlich verminderte Rechenzeit.

3. Allerdings bedeutet der Verzicht auf die Pivotisierung, daß es auch bei *nicht-singulären* Problemen zu einem Abbruch des Verfahrens kommen kann! Prinzipiell würde der Einbau einer Pivotisierung ins Programm TRID möglich sein, es würde sich jedoch dabei die Bandbreite der Matrix erhöhen (und zwar im ungünstigsten Fall verdoppeln).

   Es kann jedoch gezeigt werden, daß bei allen sogenannten *diagonal- dominanten* tridiagonalen Matrizen, d.h. bei solchen, die die Eigenschaft $\mid b_j \mid > \mid a_j \mid + \mid c_j \mid$ für alle $j = 1, \ldots, n$ haben, keine derartigen Probleme auftreten. Diese Feststellung ist von Bedeutung, weil viele in der Praxis auftretenden tridiagonalen Matrizen diese Eigenschaft der *Diagonal-Dominanz* haben!

4. Viele Programmbibliotheken enthalten Programme für die Lösung linearer Gleichungssysteme mit tridiagonalen Koeffizientenmatrizen: siehe z.B. [2], S.304 und [9], S.42.

2.6.3 Weitere Sonderformen der Koeffizientenmatrix.

In der Literatur findet man noch eine große Anzahl von Algorithmen bzw. Programmen für die numerische Behandlung von linearen Gleichungssystemen, deren Koeffizientenmatrizen besondere Strukturen haben. Insbesondere soll auf [2] und auf [9] hingewiesen werden, wo man Informationen zu den Themen: symmetrische Matrizen (Cholesky-Verfahren), zyklisch-tridiagonale Matrizen, fünfdiagonale Matrizen, allgemeine Bandmatrizen, Blockmatrizen usw. erhält.

2.7 Das Gauss-Seidel-Verfahren.

2.7.1 Allgemeines.

Das *Gauss-Seidel-Verfahren* (sowie seine Variationen, die in der Literatur unter den Namen *Einzelschrittverfahren, Gesamtschrittverfahren, Relaxationsverfahren, SOR-Verfahren usw.* zu finden sind) ist ein *iteratives Verfahren zur Lösung von linearen inhomogenen Gleichungssystemen.*

Wie schon zu Beginn dieses Kapitels erwähnt wurde, weisen *iterative* Verfahren gegenüber den *direkten* Verfahren einige Vorteile (Einfachheit des Algorithmus usw.), aber auch Nachteile (eventuell Konvergenzprobleme) auf.

Der wichtigste Vorteil der iterativen Verfahren liegt aber wohl in der Tatsache, daß *die gegebene Koeffizientenmatrix während des Iterationsprozesses nicht verändert wird.* Dieser Vorteil wirkt sich - wie im folgenden gezeigt wird - ganz besonders positiv aus, wenn man lineare Gleichungssysteme von hoher Ordnung auszuwerten hat, die *schwach besetzte Koeffizientenmatrizen* haben.

Eine Matrix heißt schwach besetzt, wenn nur wenige ihrer Koeffizienten von Null verschieden sind.

Da sich einige sehr wichtige Probleme der Numerischen Mathematik (z.B. die Lösung von Randwertproblemen mittels der Differenzenmethode) auf lineare Gleichungssysteme mit schwach besetzten Matrizen zurückführen lassen, soll im folgenden vor allem auf derartige Probleme eingegangen werden.

2.7.2 Die Grundprinzipien des Gauss-Seidel-Verfahrens.

Ausgangspunkt ist wieder das lineare System (2.1):

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= f_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= f_2 \\
&\cdot \\
&\cdot \\
&\cdot \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= f_n
\end{aligned}$$

Unter der Voraussetzung, daß *alle* Diagonalelemente der Koeffizientenmatrix ungleich Null sind, kann nun jede Gleichung des Systems nach der zum jeweiligen Diagonalelement gehörenden x-Komponente aufgelöst werden:

$$x_1 = -\frac{1}{a_{11}}\left[a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n - f_1\right]$$

$$x_2 = -\frac{1}{a_{22}}\left[a_{21}x_1 + a_{23}x_3 + \cdots + a_{2n}x_n - f_2\right]$$

bzw. allgemein

$$x_i = -\frac{1}{a_{ii}}\left[\sum_{j=1(j\neq i)}^{n} a_{ij}x_j - f_i\right] \quad . \tag{2.18}$$

Das so umgeformte Gleichungssystem kann als Matrixgleichung

$$\mathbf{x} = C \cdot \mathbf{x} + \mathbf{b}$$

mit

$$C = \{c_{ij}\} \quad \text{mit} \quad c_{ij} = \begin{cases} -a_{ij}/a_{ii} & i \neq j \\ 0 & i = j \end{cases} \qquad \text{sowie} \qquad b_i = \frac{f_i}{a_{ii}} \tag{2.19}$$

geschrieben werden. Mit der trivialen Umformung von Glg. (2.18)

$$x_i = x_i - \left(x_i + \frac{1}{a_{ii}} \left[\sum_{j=1 (j \neq i)}^{n} a_{ij} x_j - f_i \right] \right)$$

erhält man sofort die *Iterationsvorschrift für das Gauss-Seidel-Verfahren.* Setzt man nämlich in die rechte Seite der obigen Gleichung irgendwelche Näherungswerte für die x_i ein, so ergeben sich - Konvergenz vorausgesetzt - *verbesserte* Näherungswerte für die Komponenten des Lösungsvektors:

$$x_i^{(t+1)} = x_i^{(t)} - \Delta x_i^{(t)} \qquad (i = 1, \ldots, n) \tag{2.20}$$

mit

$$\Delta x_i^t = x_i^{(t)} + \frac{1}{a_{ii}} \left[\sum_{j=1 (j \neq i)}^{n} a_{ij} x_j^{(t)} - f_i \right] \quad . \tag{2.21}$$

Auf diese Weise kann im Falle einer Konvergenz eine *Folge von Näherungsvektoren*

$$\mathbf{x}^{(0)} \quad ; \quad \mathbf{x}^{(1)} \quad ; \quad \mathbf{x}^{(2)} \quad ; \quad \ldots \quad ; \quad \mathbf{x}^{(t)}$$

für den exakten Lösungsvektor $\mathbf{x}$ berechnet werden, wobei man diesen zwar niemals erhält, sich aber - abgesehen von Rundungsfehlern - beliebig nahe an ihn 'heranarbeiten' kann:

$$\lim_{t \to \infty} \mathbf{x}^{(t)} \to \mathbf{x}$$

Der Vektor $\mathbf{x}^{(0)}$ heißt der *Startvektor* der Iteration.

Typisch für alle Iterationsverfahren sind die sogenannten *Abbruchbedingungen*, die festlegen, wann eine Iteration zu beenden ist.

In <u>jedem</u> Iterationsprogramm sollten zumindest die beiden folgenden Abbruchbedingungen enthalten sein:

- Die Iteration ist zu beenden, wenn die Ergebnisse eine vorgegebene Genauigkeit erreicht haben.

- Die Iteration ist <u>in jedem Fall</u> zu beenden, wenn eine vorgegebene maximale Zahl von Iterationsschritten erreicht ist ('Notausgang' im Falle einer Divergenz bzw. einer sehr langsamen Konvergenz).

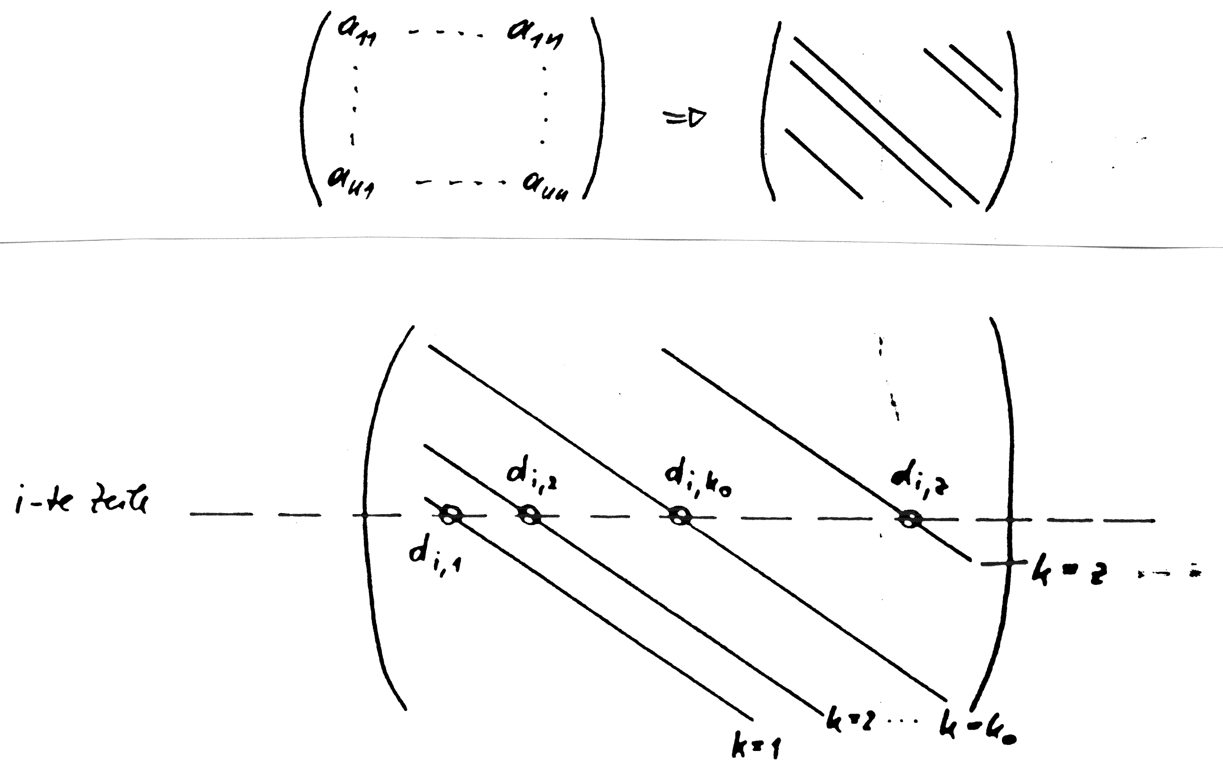


Abbildung 2.2: Zur Abspeicherung einer Bandmatrix.

2.7.3 Das Gauss-Seidel-Verfahren für Bandmatrizen.

Eine Bandmatrix ist eine Matrix, die nur entlang einiger Diagonalen Komponenten enthält, die von Null verschieden sind (s. Abbildung 2.2 (oben)).

Es ist sofort einzusehen, daß es vom Standpunkt der Speicherplatz-Ökonomie ein Nonsens ist, eine solche Matrix als $(n$ x $n)$-Matrix abzuspeichern. Die gesamte Information kann nämlich viel platzsparender abgespeichert werden, wenn man *die Diagonalen als Vektoren speichert und über die Lage der einzelnen Diagonalen innerhalb der Matrix buchführt.*
Wie dies z.B. geschehen kann, ist in der Abbildung 2.2 (unten) schematisch dargestellt:
z sei die Anzahl der in der Matrix vorkommenden Diagonalen. Die Lage jeder Diagonale wird nun durch eine ganze Zahl angegeben, die angibt, wo sich diese Diagonale *relativ zur Hauptdiagonale* befindet. Damit kann die Abspeicherung der gesamten Information in einem zweidimensionalen Feld d_{ik} erfolgen, wobei der Index i die jeweilige Matrixzeile bedeutet. Der zweite Index (k) gibt an, um welche der z Diagonalen es sich handelt. Zusätzlich wird ein INTEGER-Feld $s(k)$ mit den relativen Positionen definiert:

$$s(k) = \begin{cases} 0 & \text{Hauptdiagonale} \\ +(-)1 & \text{erste obere(untere) Nebendiagonale} \\ +(-)t & t\text{–te obere(untere) Nebendiagonale} \end{cases}$$

Da beim Gauss-Seidel-Verfahren die Elemente der Hauptdiagonale eine besonders wichtige Rolle spielen, soll der Index k, der die Hauptdiagonale bezeichnet, als k_o hervorgehoben werden. Es gilt also

$$s(k = k_o) = 0$$

Die Zuordnung der Matrixelemente a_{ij} (i-te Zeile, j-te Spalte) zu den Elementen d_{ik} ist somit gegeben durch

$$a_{ij} = d_{ik} \quad \text{mit} \quad j = s(k) + i \quad , \tag{2.22}$$

wobei $i = 1, \ldots, n$ und $k = 1, \ldots, z$.

Auf Grund all dieser Überlegungen ist es kein Problem, die Iterationsvorschrift (2.21) im Hinblick auf (2.22) zu variieren:

$$x_i^{(t+1)} = x_i^{(t)} - \Delta x_i^{(t)} \tag{2.23}$$

mit

$$\Delta x_i^{(t)} = x_i^{(t)} + \frac{1}{d_{i,k_o}} \left[\sum_{k=1 (k \neq k_o)}^{z} d_{ik} x_{s(k)+i}^{(t)} - f_i \right] \tag{2.24}$$

Anmerkung: In der Summe in Glg.(2.24)werden nur jene Summanden zugelassen, für welche gilt:

$$0 < s(k) + i \leq n$$

2.7.4 Konvergenzkriterien und Fehlerverhalten.

Es erhebt sich nun die wichtige Frage, ob man es einem gegebenen linearen Gleichungssystem schon a priori 'ansieht', daß es Schwierigkeiten bei der Konvergenz geben wird.

Nun gibt es in der Literatur in der Tat eine ganze Reihe von *Konvergenz-Kriterien*, d.h. von Bedingungen, die im Falle ihrer Erfüllung die Konvergenz der Gauss-Seidel-Iteration sicherstellen. Solche Kriterien sind zwar *hinreichend*, aber i.a. nicht *notwendig*, d.h. auch manche Systeme, die diese Kriterien nicht erfüllen, können mittels des Gauss-Seidel- Verfahrens gelöst werden.
Aus diesem Grund möchte ich hier auf die Angabe von mathematisch formulierten Konvergenz-Kriterien verzichten und mich auf eine in der Praxis bewährte Faustregel beschränken:

Alle linearen Gleichungssysteme, in deren Koeffizientenmatrizen die Hauptdiagonalelemente die anderen Matrixelemente dominieren, haben gute Chancen auf eine Gauss-Seidel-Konvergenz.

Ein konvergentes Verhalten der Iteration bedeutet natürlich, daß der auftretende Verfahrensfehler ϵ_V mit fortschreitender Iteration immer kleiner wird, daß also gilt:

$$\lim_{t\to\infty} \epsilon_V^{(t)} \to 0$$

In der Praxis wird solange iteriert, bis der Korrekturvektor $\Delta\mathbf{x}$ [Glg. (2.24)] genügend klein geworden ist. Man verwendet in diesem Fall als Abbruchkriterien die *absolute* Fehlerabfrage

$$\max \mid \Delta x_i \mid < \mathrm{GEN}$$

bzw. die *relative* Fehlerabfrage

$$\max \mid \frac{\Delta x_i}{x_i} \mid < \mathrm{GEN}$$

wobei GEN eine vom Programmbenutzer festgesetzte Fehler-Toleranz ist.

Achten Sie darauf, die Größe GEN nicht zu klein anzusetzen, da im Falle von schlecht konditionierten Problemen die entsprechende Rechengenauigkeit durch auftretende Rundungsfehler nicht erreicht werden kann!

Nun gibt es jedoch die Aussage, daß iterative Methoden i.a. wesentlich geringere Rundungsfehler aufweisen als die direkten Methoden. Es ist insbesondere eine Eigenschaft von Iterationsmethoden, daß jeweils nur die Rundungsfehler *des letzten Iterationsschrittes* einen Effekt haben: es gibt also keine ϵ_R-Kumulation über alle Iterationen hinweg! Auf Grund dieser Tatsache ist die Verwendung des Gauss-Seidel-Verfahrens in manchen Fällen auch dann einem direkten Verfahren vorzuziehen, wenn es höhere Rechenzeiten erfordert.

2.7.5 Das Unterprogramm GAUSEI.

Das Programm GAUSEI (GAUss-SEIdel) berechnet den Lösungsvektor eines inhomogenen linearen Gleichungssystems mit einer allgemeinen Bandmatrix als Koeffizientenmatrix.

INPUT-Parameter:

N: Ordnung des Systems.

NDIAG: Anzahl der in der Bandmatrix vorkommenden Diagonalen.

S( ): INTEGER-Feld mit den relativen Positionen der Diagonalen.

DIAG(,): Feld mit den Matrix-Koeffizienten: der erste Index spezifiziert die Matrixzeile, der zweite die Diagonale.

F( ): inhomogener Vektor des Systems.

TMAX: maximale Anzahl der Iterationsschritte.

W: Relaxationsparameter (s. Abschnitt 2.7.6).

IREL: IREL $\neq$ 1: *absolute* Fehlerabfrage,
IREL = 1: *relative* Fehlerabfrage.

GEN: absoluter oder relativer Fehler, der im Laufe der Iteration zu unterschreiten ist.

OUTPUT-Parameter:

LOES( ): Lösungsvektor.

T: Anzahl der von GAUSEI durchgeführten Iterationsschritte.

FEHLER: logische Variable zur Fehlerdiagnostik:

FEHLER hat nach der Exekution von GAUSEI den Wert 'falsch', wenn die geforderte Genauigkeit erreicht wurde, und den Wert 'wahr', wenn

- nicht alle Werte der Hauptdiagonalen von Null verschieden sind.
- innerhalb von TMAX Iterationsschritten die geforderte Genauigkeit nicht erreicht wurde.

wichtige interne Variable:

K0: Index der Hauptdiagonalen.

DX: Iterations-Korrekturwert gemäß Glg.(2.24).

ISCH: Steuervariable bei der Genauigkeitsüberprüfung.

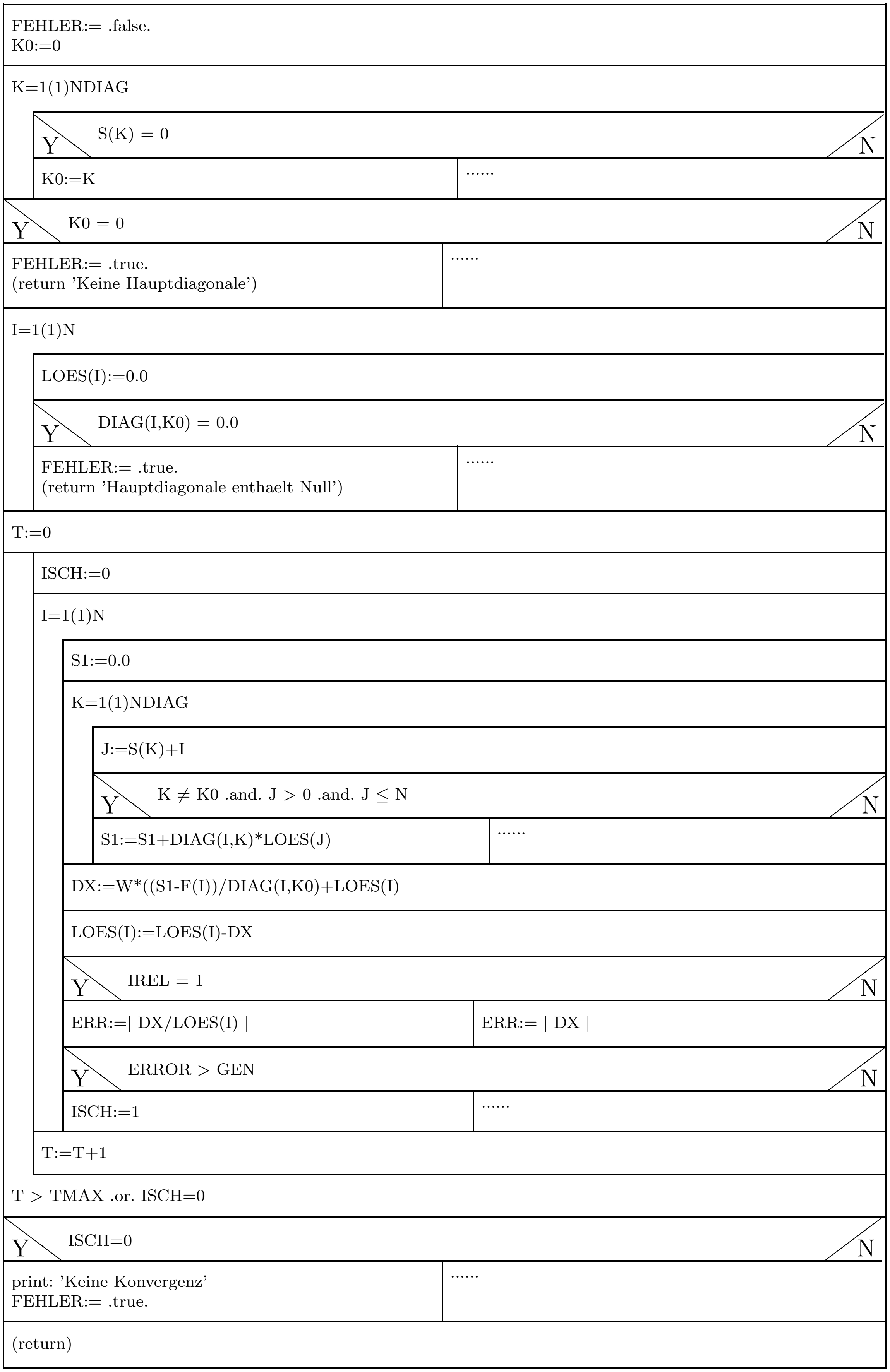
Programmstruktur von GAUSEI:

1. Check, welche der übergebenen Diagonalen die Hauptdiagonale ist. Der entsprechende Index k wird als K0 abgespeichert. Ist keine der übergebenen Diagonalen eine Hauptdiagonale, wird das Programm abgebrochen.

2. Check, ob die Hauptdiagonale Nullen enthält. Wenn ja, wird das Programm abgebrochen. Gleichzeitig wird der Startvektor der Iteration (= Nullvektor) definiert.

3. Durchführung der Gauss-Seidel-Iteration mit Fehlerabfrage.

Struktogramm 8 — GAUSEI(N,NDIAG,S,DIAG,F,TMAX,W,IREL,GEN, LOES,T,FEHLER)



2.7.6 Eine Variation des Gauss-Seidel-Verfahrens.

Die Iterationsvorschrift (2.23)

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \Delta\mathbf{x}^{(t)}$$

kann durch die formale Einführung eines Faktors ω im Sinne von

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \omega \cdot \Delta\mathbf{x}^{(t)} \tag{2.25}$$

erweitert werden. Im Falle von $\omega = 1$ liegt also die normale Gauss-Seidel-Iteration vor, bei $\omega > 1$ spricht man von einer *Überrelaxationsmethode*, bei $\omega < 1$ von einer *Unterrelaxationsmethode*.

Durch die richtige Wahl des Relaxationsparameters ω kann in manchen Fällen die Konvergenzgeschwindigkeit außerordentlich erhöht werden. Ein gutes Beispiel für diesen Effekt findet man in [7], S.192ff. Iteriert man das einfache lineare System

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

bis zum Unterschreiten des absoluten Fehlers GEN=10^{-8} (C, double precision), so erhält man die folgende Abhängigkeit der erforderlichen Zahl von Iterationsschritten t von ω:

ω	t
0.65	20
0.70	18
0.75	15
0.8	14
0.85	12
0.9	12
0.95	21
1.0	31
1.05	48

Es wäre eine enorme Effizienzsteigerung der Gauss-Seidel-Methode, wenn man den *idealen Relaxationsparameter* ω_{ideal} *a priori* bestimmen könnte. Zu diesem Thema folgen nun einige Anmerkungen:

- Die Relaxationsmethode ist <u>nur</u> konvergent für $0 \leq \omega \leq 2$.
- 'Unter gewissen mathematischen Restriktionen', die i.a. von Matrizen erfüllt werden, welche bei der Anwendung von Differenzenverfahren (s. Kap. 9) zur Lösung von (insbesondere elliptischen) Randwertproblemen auftreten, liegt ω_{ideal} stets im Bereich zwischen 1 und 2, d.h. es liegt immer eine Überrelaxation vor.
- In solchen Fällen gilt:

  $$\omega_{\text{ideal}} = \frac{2}{1 + \sqrt{1 - \lambda_1^2}}\,, \tag{2.26}$$

  wobei λ_1 den größten Eigenwert der Matrix C [Glg. (2.19)] darstellt.

- Da jedoch die genaue Berechnung von λ_1 häufig zu aufwendig ist, verweise ich auf einen Vorschlag in Ref. [14], S. 63, der sich in der Praxis gut bewährt hat und sehr einfach ins Programm GAUSEI eingebaut werden kann:

  Man kann zeigen, daß der Ausdruck

$$\frac{|\Delta \mathbf{x}^{(t)}|}{|\Delta \mathbf{x}^{(t-1)}|} \tag{2.27}$$

  für $t \to \infty$ den Grenzwert λ_1^2 hat. Wenn man z.B. im Programm GAUSEI die Iteration mit $\omega = 1$ beginnt und die ersten t_0 Iterationen durchführt[1], so kann man mittels Glg. (2.27) in vielen Fällen schon eine ausreichend gute Approximation für λ_1^2 und damit mittels Glg (2.26) auch für ω_{ideal} erhalten. Mit diesem ω-Wert wird dann die Iteration fortgesetzt.

Wie gut diese Strategie funktioniert, wird in den Übungen zu dieser Vorlesung demonstriert.

2.7.7 Effizienz des Gauss-Seidel-Verfahrens

In den vorigen Abschnitten wurde als Vorteil der Gauss-Seidel-Methode angegeben, daß die Koeffizientenmatrix sich während der Iteration nicht ändert, was vor allem im Falle schwach besetzter Bandmatrizen günstig ist, weil man nur die besetzten Diagonalen abspeichern muß.

Dies soll an Hand eines konkreten Beispiels dokumentiert werden, und zwar für die numerische Auswertung einer sehr bedeutenden partiellen Differentialgleichung, nämlich der Laplace-Gleichung. Unter Verwendung der Differenzenmethode wird diese Differentialgleichung samt ihren Nebenbedingungen in ein System von linearen Gleichungen umgewandelt. Die Ordnung dieses Systems ergibt sich dabei aus der Zahl der Stützpunkte im Grundgebiet der zu lösenden Laplace-Gleichung. Die folgende Abbildung 2.3 (a) zeigt, wie viele Prozent der Elemente der Koeffizientenmatrix als Funktion der Stützpunktzahl tatsächlich besetzt sind. Man sieht, daß dieser Wert ab etwa 200 Stützpunkten deutlich unter 10 Prozent liegt.
Wie sich das auf die im Arbeitsspeicher des Computers nötigen Speicherplätze auswirkt, ist in Abb. 2.3 (b) dargestellt.

Überhuber ([22], S. 392) gibt ein Beispiel aus der industriellen Anwendung linearer Methoden (Karosserie-Festigkeitsberechnung im Automobilbau): Eine Matrix BCSSTK32 hat die Ordnung 44609, also $2 \cdot 10^9$ Elemente; von diesen sind aber 'nur' 1029655 Nicht-Nullelemente, d. h. der Besetzungsgrad beträgt 0.05 %.

[1] In der Praxis wird für t_0 ein Wert zwischen 20 und 100 genommen.

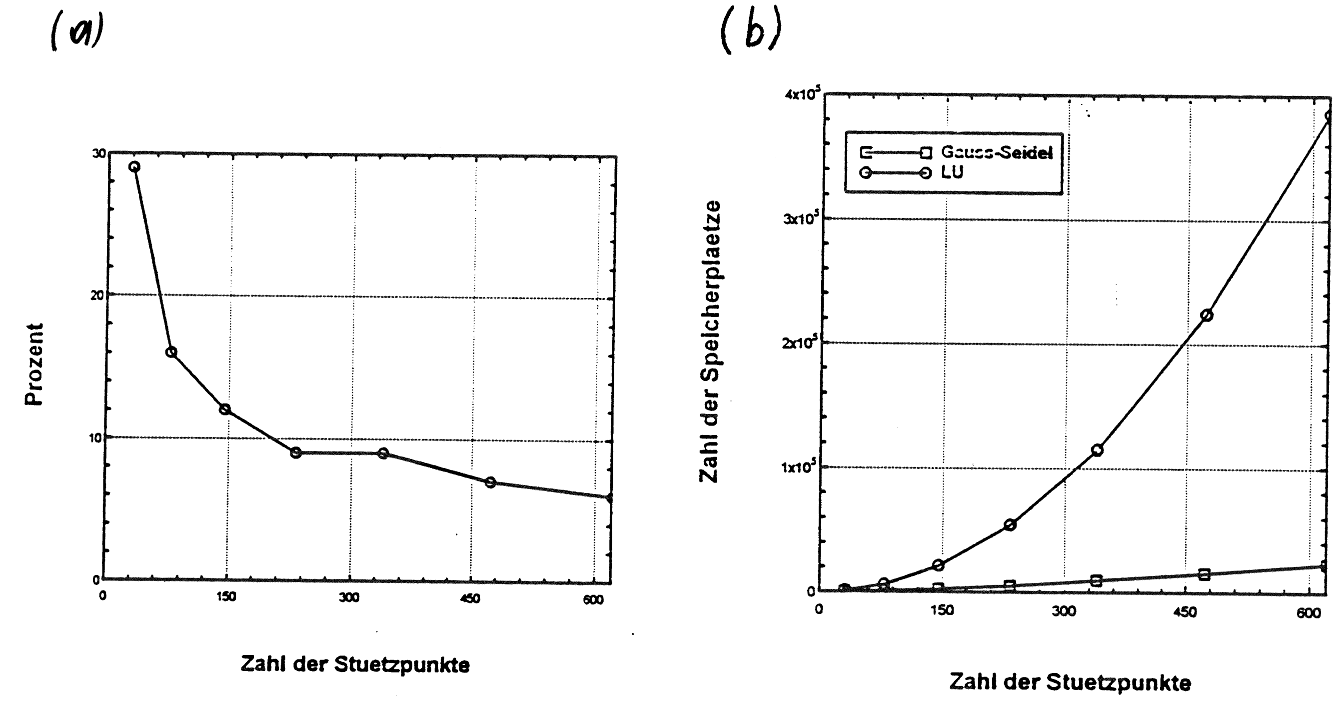


Abbildung 2.3: (a) Prozentueller Anteil der besetzten Matrixplätze als Funktion der Zahl der Stützpunkte. (b) Erforderliche Speicherplätze als Funktion der Zahl der Stützpunkte.

2.8 Software-Angebot

Am Ende jedes Hauptkapitels dieses Skriptums finden Sie unter dem Titel 'Software-Angebot' Hinweise auf Programme, die Sie sich entweder kostenlos übers Internet beschaffen können (*public domain* (p.d.) Programme), oder die kommerziell vertrieben werden.

Quellen: (1) Bücher von Ch. Überhuber [21], [22].
(2) Erfahrungen des Autors dieses Skriptums.

Diese Anmerkungen können auf Grund des riesigen Software-Angebotes nur erste Anregungen für Sie sein, dieses Angebot zu nutzen.

Zitat aus [21], S. 322:
Von herausragender Bedeutung für die rasche, einfache und effiziente Beschaffung von frei erhältlicher (public domain)-Software aus dem Numerik-Bereich ist der Internet-Dienst NETLIB (Dongara, Grosse, 'Distribution of Mathematical Software via Electronic Mail', Comm. ACM 30 (1987), 403).

Auf Programme aus dem NETLIB kann über verschiedene Internet-Dienste zugegriffen werden (email, FTP, Internet); am einfachsten ist ein 'Downladen' aus dem www-System. Ich werde mich im folgenden auf diese Möglichkeit beschränken.
Zuvor noch ein weiteres Zitat ([22], S. 183) mit speziellem Bezug auf das Kap. 2 dieses Skriptums:
Für die numerische Lösung linearer Gleichungssysteme gibt es, mehr als in jedem anderen Gebiet der Numerischen Datenverarbeitung, eine Fülle qualitativ hochwertiger Softwareprodukte: LAPACK, TEMPLATES, ITPACK, NSPCG, SLAP, UMFPACK, Teile der IMSL- und NAG-Bibliotheken. Die eigene Entwicklung von Software zur Lösung linearer Gleichungssysteme wäre aus diesem Grund ein sehr unökonomisches Unterfangen.

- Zugriff auf NETLIB: *www.netlib.org*
- Einen ersten Überblick über das Angebot:
  Klick auf *Browse the Netlib repository*

Die wichtigsten Libraries für den Problemkreis Lineare Gleichungssystemeßind die folgenden:

LAPACK:

Umfangreiche Sammlung von F77-Programmen zu den Themen 'Lösung linearer Gleichungssysteme', 'Eigenwertprobleme' u.a.
Die komplette aktuelle Bibliothek (31.5.2000) inklusive aller sog. BLAS-Hilfsprogramme[2] umfaßt ca. 5 MByte und kann durch Anklicken von *lapack.tgz* downgeladen werden. Danach einfach auf Ihrem LINUX-Rechner dekomprimieren und in die einzelnen Programme aufteilen:

[2]BLAS: Basic Linear Algebra Subprograms.

```
gunzip -c lapack.tgz | tar xvf -
```

Achtung: Alle Programme (ohne OBJ- und EXE-Files) in den 4 Datentypen REAL, DOUBLE PRECISION, COMPLEX und COMPLEX*16 benötigen ca. 34 MByte.

- Wenn Sie LAPACK-Programme intensiv verwenden wollen, ist es vernünftig, sich den *LAPACK Users' Guide* (E. Anderson et al, Society for Industrial and Applied Mathematics, 1999) zuzulegen. Dieses Manual ist auch im Internet unter *www.netlib.org/lapack/lug/lapack_lug.html* erhältlich.

- Wenn Sie das ganze LAPACK auf Ihren Rechner installieren wollen, ist es auch nützlich, den *LAPACK Installation Guide* (ps-File, 232 kByte, 25 Seiten) downzuladen.

- Wenn Sie eine bestimmte LAPACK-Routine laden wollen, gehen Sie wie folgt vor: Klicken Sie im LAPACK-Verzeichnis *netlib.org/lapack/index.html* die entsprechende Eintragung in den *Individual Routines* an.
  Angenommen, Sie wollen ein DOUBLE-PRECISION-Lösungsprogramm für lineare Gleichungssysteme in Ihren Rechner laden. In diesem Fall klicken Sie auf *lib double*; als erste Eintragung in der erscheinenden Programmliste finden Sie:

  ```
  file dgesv.f   dgesv.f plus dependencies
  prec double
  for  Solves a general system of linear equations AX=B.
  ```

  Sie könnten nun durch Anklicken von *dgesv.f* das Programm speichern. Allerdings hat das i.a. wenig Sinn, da die meisten LAPACK-Programme eine ganze Kaskade von weiteren Programmen aufrufen. Es ist daher besser, diese ganze 'Programmfamilie' durch Anklicken von *dgesv.f plus dependencies* zu laden. Im erscheinenden Dialogfenster wählen Sie als Packaging formatTar-gzip, und die nötigen BLAS-Files laden Sie am besten gleich dazu.
  Wenn das geklappt hat, haben Sie in Ihrem Verzeichnis den File *netlibfiles.tgz*. Die weitere Verarbeitung lautet (s.o.):

  ```
  gunzip -c netlibfiles.tgz | tar xvf -
  ```

  Damit haben Sie sofort, auf drei neu erstellte Verzeichnisse aufgeteilt, die gesamte Programmfamilie von *dgesv.f* zur Verfügung:

  ```
  blas/blas2test.f
  blas/dgemm.f
  blas/dger.f
  blas/dscal.f
  blas/dswap.f
  ```

```
blas/dtrsm.f
blas/idamax.f
blas/xerbla.f
lapack/double/dgesv.f
lapack/double/dgetf2.f
lapack/double/dgetrf.f
lapack/double/dgetrs.f
lapack/double/dlaswp.f
lapack/util/ieeeck.f
lapack/util/ilaenv.f
```

Die folgende knappe Zusammenstellung gibt Ihnen eine Übersicht über das LAPACK-Angebot ([22], S. 266). Die Begriffe *simple drivers* und *expert drivers* bedeuten:
simple drivers: *black box*–Programme, die primär die numerische Lösung der jeweiligen Problemstellung liefern.
expert drivers liefern neben den Lösungen auch noch zusätzliche Informationen (Konditionsabschätzungen, Fehlerschranken usw.).

Matrixtyp (Speicherung)	*driver*	REAL	COMPLEX
allgemeine Matrix	*simple* *expert*	**sgesv** **sgesvx**	**cgesv** **cgesvx**
allgemeine Bandmatrix	*simple* *expert*	**sgbsv** **sgbsvx**	**cgbsv** **cgbsvx**
allgemeine Tridiagonalmatrix	*simple* *expert*	**sgtsv** **sgtsvx**	**cgtsv** **cgtsvx**
symmetrische/Hermitesche positiv definite Matrix	*simple* *expert*	**sposv** **sposvx**	**cposv** **cposvx**
symmetrische/Hermitesche positiv definite Matrix (gepackte Speicherung)	*simple* *expert*	**sppsv** **sppsvx**	**cppsv** **cppsvx**
symmetrische/Hermitesche positiv definite Bandmatrix	*simple* *expert*	**spbsv** **spbsvx**	**cpbsv** **cpbsvx**
symmetrische/Hermitesche positiv definite Tridiagonalmatrix	*simple* *expert*	**sptsv** **sptsvx**	**cptsv** **cptsvx**
symmetrische/Hermitesche indefinite Matrix	*simple* *expert*	**ssysv** **ssysvx**	**chesv** **chesvx**
symmetrische/Hermitesche indefinite Matrix (gepackte Speicherung)	*simple* *expert*	**sspsv** **sspsvx**	**chpsv** **chpsvx**
komplexe symmetrische Matrix	*simple* *expert*		**csysv** **csysvx**
komplexe symmetrische Matrix (gepackte Speicherung)	*simple* *expert*		**cspsv** **cspsvx**

LAPACK bietet nur direkte Verfahren in FORTRAN-77. Diese Programmiersprache ist zwar immer noch weitverbreitet im technisch-naturwissenschaftlichen Bereich; da aber neue Fortranversionen wie F90, F95 sowie andere Sprachen wie C und C++ immer wichtiger werden, gibt es inzwischen auch schon die entsprechenden LAPACK-Routinen:

LAPACK95

stellt ein Interface zu F77 dar, d. h. Sie benötigen auch die LAPACK-F77 Programme. LAPACK95 erhalten Sie über
www.netlib.org/lapack95/index.html.
Downladen von *lapack95.tgz* und nachfolgende Dekomprimierung erzeugt in Ihrer Directory das Unterverzeichnis LAPACK95. In diesem gibt es das Verzeichnis SRC mit allen zur Verfügung stehenden Routinen.

CLAPACK

erhalten Sie über *www.netlib.org/clapack*.
Es handelt sich dabei um eine *vollständige* C-Implementierung von LAPACK, wobei die Konvertierung von F77 in C automatisch erfolgte. Der Nachteil dieses Vorgehens ist allerdings, daß CLAPACK wertvolle Eigenschaften von C, wie z. B. dynamische Speicherallokation, nicht nutzt.
Darüber hinaus können Sie nicht einfach irgendein einzelnes CLAPACK-Programm herunterladen, sondern es ist eine komplette Installation von
clapack.tgz (Version 3.0, ca. 6 MByte) erforderlich. Dieser File enthält auch die Installationshilfen
readme.install und *readme.maintain* sowie den Header-File *clapack.h*

LAPACK++

stellt eine auf C++ basierende objektorientierte Schnittstelle zu LAPACK dar. Der Sourcecode heißt *lapack++.tgz* und enthält in komprimierter Form 66 kByte. Genauere Information über dieses System sind in den folgenden Postcript-Files zu finden:
overview.ps *classref.ps* *install.ps* *user.ps*

Eine weitere interessante Internet-Adresse zum Thema LAPACK++:
http://math.nist.gov/lapack++

SCALAPACK

Anwendung von verteilten und parallelen Rechnersystemen zur Lösung von Gleichungssystemen, Ausgleichsproblemen und Eigenwertproblemen mit sehr großen Koeffizientenmatrizen ($n > 5000$).
Eine kurze und prägnante Einführung in dieses Thema (Stand: 1995) s. [22], S. 283-286.

Mathematica und MATLAB:

Mathematica erlaubt sowohl symbolische als auch numerische Behandlung wichtiger Probleme der Linearen Algebra, wie z.B.:

Inverse[m] Matrix-Inversion
Det[m] Determinante
LinearSolve[m,b] Lösung des inhomogenen linearen Systems.

MATLAB:[3] im Prinzip bietet diese Benutzersoftware die einfachste Realisierung der Lösung des linearen Gleichungssystems A.x=f :

```
>>% Eingabe der Koeffizientenmatrix A:
>> A=[ ... ; ... ];
>>% Eingabe des inhomogenen Vektors f:
>> f=[ ... ],
>>% Loesung des Gleichungssystems:
>> x=A\f
```

Eine alternative Lösungsmöglichkeit bietet Matlab durch folgende Befehle:

```
>>% Der inhomogene Vektor wird zur Koeffizientenmatrix als letzte Spalte
>>% hinzugefuegt:
>> Aplus=[A f];
>>% Diese erweiterte Matrix wird als Argument der Funktion  rref
>>% verwendet. Nun wird die urspruengliche Matrix mittels des Verfahrens
>>% von Gauss-Jacobi in eine Einheitsmatrix umgewandelt, und die letzte
>>% Spalte enthaelt den Loesungsvektor:
>> Aneu=rref(Aplus);
>> x=Aneu(:,end)
>>% Dieses Verfahren hat den Vorteil, dass es etwas stabiler gegenueber
>>% Rundungsfehlern ist als die obige "einfache" Loesung.
```

2.8.1 Iterative Lösung schwach-besetzter Systeme

TEMPLATES

Diese NETLIB-Library enthält eine Reihe von Programmpaketen mit iterativen Lösungsmethoden für lineare Gleichungssysteme:

Dateiname	Inhalt der Datei
sctemplates.tgz	C-Routinen, einfache Genauigkeit
dctemplates.tgz	C-Routinen, doppelte Genauigkeit
cpptemplates.tgz	C++-Routinen
sftemplates.tgz	F77-Routinen, einfache Genauigkeit
dftemplates.tgz	F77-Routinen, doppelte Genauigkeit
mltemplates.tgz	Matlab-Routinen

Eine interessante Matlab-Routine ist *sor.m.* Dieses Programm löst ein lineares Gleichungssystem mittels der Sukzessiven Überrelaxationsmethode (allerdings ohne Berücksichtigung einer eventuellen Bandstruktur der Koeffizientenmatrix).

[3] MATrix LABoratory.

Kapitel 3

Interpolation von Punktmengen.

3.1 Definition des Problems.

Es seien die x- und die y-Koordinaten von n Punkten gegeben. Die Zuordnung der Koordinaten sei *eindeutig* d.h. zu jedem x-Wert gehöre ein und nur ein y-Wert. Weiters soll angenommen werden, daß die x-Werte nach ihrer Größe geordnet vorliegen; sie müssen aber keineswegs äquidistant sein.

Eine wichtige Forderung an die Punktmenge ist mathematisch schwer zu definieren. Sie lautet: Die *Stützpunkte* $x_i \mid y_i$, $i = 1, \ldots, n$ sollen *einen funktionellen Zusammenhang zwischen x und y phänomenologisch befriedigend repräsentieren.* Die Abb.3.1 zeigt, wie das gemeint ist.
Geht man nun davon aus, daß die gegebene Punktmenge im Intervall $[x_1, x_n]$ repräsentativ ist, so kann eine *Interpolationsfunktion* $I(x)$ wie folgt definiert werden:

- $I(x)$ soll exakt durch die Stützpunkte hindurchgehen. Es soll also gelten:
$$I(x_i) = y_i \quad \text{für} \quad i = 1, \ldots, n \tag{3.1}$$

- $$I(x) \text{ soll die Stützpunkte } \textit{möglichst glatt} \text{ verbinden.} \tag{3.2}$$

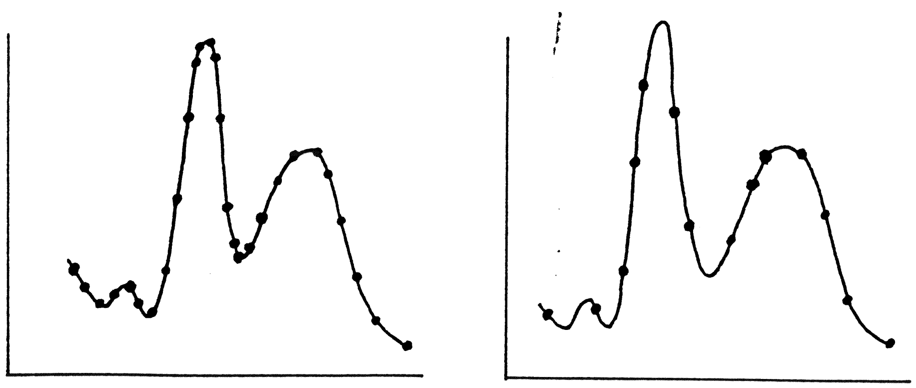


Abbildung 3.1: Punktmenge repräsentativ nicht- repräsentativ

Das folgende aus [13], S.127: Der Konstruktionsweg für die Interpolationsfunktion besteht darin, eine endliche Menge Funktionen $\varphi_k(x)$ zugrunde zu legen, die über dem Intervall $[x_1, x_n]$ linear unabhängig sind, und die Interpolationsfunktion als *Linearkombination* dieser Funktionen anzusetzen, also

$$I(x) = \sum_{k=1}^{n} c_k \varphi_k(x) \quad .$$

Typische Entwicklungsfunktionen für $I(x)$ sind

- Potenzfunktionen: 1 ; x ; x^2 ; x^3 ; ...
- Exponentialfunktionen: 1 ; $e^{\pm iax}$; $e^{\pm 2iax}$; $e^{\pm 3iax}$; ...

3.2 Interpolation durch Potenzfunktionen.

Sehr häufig angewendet wird die *polynomiale Interpolation*, d.h. als Ansatz dient ein Polynom $(n-1)$-ten Grades mit den (vorerst) allgemeinen Koeffizienten c_k:

$$I(x) = \sum_{k=1}^{n} c_k x^{k-1} \tag{3.3}$$

Die n Koeffizienten c_k können nun dadurch ermittelt werden, daß man den Ansatz 3.3 in die Glg. 3.1 einsetzt:

$$I(x_i) = \sum_{k=1}^{n} c_k x_i^{k-1} = y_i \quad \text{für} \quad i = 1, \ldots, n$$

Dies stellt ein lineares inhomogenes Gleichungssystem für die Koeffizienten $c_1 \cdots c_n$ dar. Die Lösung eines solchen Problems ist *eindeutig* d.h. es gibt nur *ein* Polynom vom Grade $n-1$, das durch alle Stützpunkte exakt hindurchgeht. Damit ist die Bedingung (3.1) erfüllt.

Lagrange hat eine Formel angegeben, die es erlaubt, ein derartiges *Interpolationspolynom* in geschlossener Form anzuschreiben:

$$I(x) = \sum_{k=1}^{n} y_k \prod_{i=1 (i \neq k)}^{n} \frac{(x - x_i)}{(x_k - x_i)} \tag{3.4}$$

Die numerische Berechnung von Werten des Interpolationspolynoms kann direkt aus der Gleichung (3.4) erfolgen. In der einschlägigen Literatur findet man jedoch zahlreiche Algorithmen für die Auswertung von (3.4), die für den Computer besser geeignet sind, wie z.B. den *Neville'schen Algorithmus* ([9], S.80ff) oder den *Formalismus von Newton* ([2], S.135ff, Programme S.345ff). In diesem Skriptum wird auf keines dieser Bibliotheksprogramme näher eingegangen, weil sie in der Praxis von nur beschränkter Bedeutung sind. Die Begründung dafür folgt im folgenden Beispiel (aus [14], S.2ff):

Gegeben seien 11 Punkte der Funktion

$$f(x) = \frac{1}{1 + x^2} \tag{3.5}$$

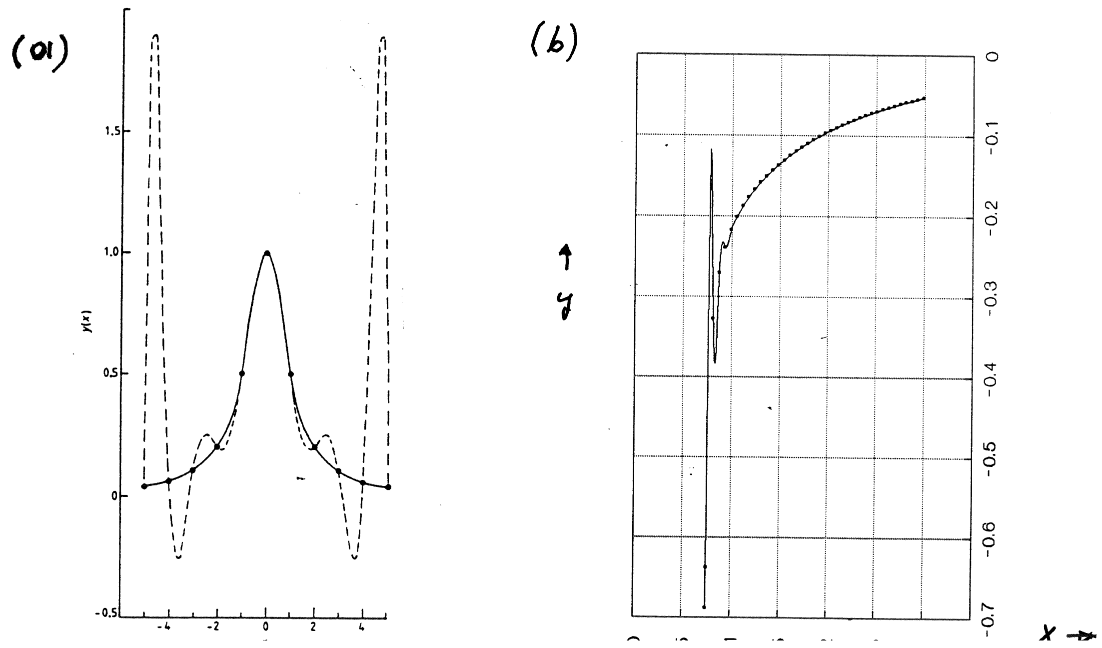


Abbildung 3.2: (a) Polynomiale Interpolation und stückweise Spline-Interpolation der Funktion (3.5); (b) Oszillationen einer Spline-Interpolation.

für die Argumente $x = -5, -4, \ldots, 4, 5$. Legt man durch diese 11 Stützpunkte das entsprechende Interpolationspolynom 10.ten Grades, so erhält man als Ergebnis die strichlierte Kurve in Abb. 3.2 (a).
Damit ist das Problem evident: das Polynom beschreibt die zu interpolierende Funktion nur im Bereich $x \approx 0$ gut. An den Rändern des Interpolationsintervalles kommt es hingegen zu massiven Verfahrensfehlern. Die Ursache dafür ist die Neigung von Interpolationspolynomen, zwischen den Stützpunkten zu oszillieren. Da diese Oszillationen mit steigendem Grad des Polynoms i.a. zunehmen, sind die gemäß (3.4) bestimmten Interpolationskurven in vielen Fällen in eindeutigem Widerspruch zur 'Glätte-Bedingung' (3.2) und somit unbrauchbar!

Im folgenden wird nun gezeigt, daß es meist zielführender ist, eine gegebene Punktmenge *stückweise* unter Verwendung von Polynomen niedriger Grade zu interpolieren.

3.2.1 Stückweise Interpolation mittels kubischer Splines.

Ausgangspunkt für die Spline-Interpolation sind Polynome vom Grad 3:

$$P^i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \quad , \qquad (3.6)$$

wobei der Index i bedeutet, daß das entsprechende Polynom nur im Intervall $[x_i, x_{i+1}]$ betrachtet wird. Die Interpolationskurve $I(x)$ besteht also (im Falle von n Stützpunkten) aus insgesamt $n-1$ Polynom-Stücken:

$$I(x) = \begin{cases} P^1(x) & \text{für} \quad x \in [x_1, x_2] \\ P^2(x) & \text{für} \quad x \in [x_2, x_3] \\ & \quad . \\ & \quad . \\ P^{n-1}(x) & \text{für} \quad x \in [x_{n-1}, x_n] \end{cases}$$

Die insgesamt $4(n-1)$ Polynomkoeffizienten a_i, b_i, c_i und d_i werden nun so ermittelt,daß

- $P^i(x)$ durch die Stützpunkte y_i bzw. y_{i+1} hindurchgeht.
- $P^i(x)$ am Punkt x_{i+1} *dieselbe erste und zweite Ableitung* hat wie das rechts anschließende Polynomstück $P^{i+1}(x)$.

Die erste dieser beiden Bedingungen definiert eine Interpolationskurve, die zweite macht aus $I(x)$ *eine kubische Spline-Interpolation*.

Im Prinzip können zur Spline-Interpolation auch Polynome höherer Grade herangezogen werden. Da aber die kubischen Splines am weitaus häufigsten verwendet werden, soll hier nur von diesen die Rede sein (Informationen, Formeln und Programme zu Splines 5. Grades s. z.B. [2],S. 154ff bzw. S.358ff).

Für die kubischen Splines ergeben sich also die folgenden Bedingungen:

$$\begin{aligned} P^i(x_i) &= y_i \\ & & i = 1, \ldots, n-1 \\ P^i(x_{i+1}) &= y_{i+1} \\ \tfrac{d}{dx} P^i(x_{i+1}) &= \tfrac{d}{dx} P^{i+1}(x_{i+1}) \\ & & i = 1, \ldots, n-2 \\ \tfrac{d^2}{dx^2} P^i(x_{i+1}) &= \tfrac{d^2}{dx^2} P^{i+1}(x_{i+1}) \end{aligned} \tag{3.7}$$

Das sind $4n-6$ Gleichungen für die insgesamt $4n-4$ Polynomkoeffizienten. Es bedarf also zweier weiterer Gleichungen, um das System eindeutig lösbar zu machen. Diese beiden zusätzlichen Bedingungen sind im Prinzip beliebig. *Häufig fordert man, daß die 2. Ableitungen der Interpolationsfunktion am Beginn und am Ende des Interpolationsintervalls verschwinden*:

$$\frac{d^2}{dx^2} P^1(x_1) = 0 \qquad \frac{d^2}{dx^2} P^{n-1}(x_n) = 0 \tag{3.8}$$

Die so erhaltene Interpolationskurve ist stetig bzgl. des Funktionswertes sowie der ersten und der zweiten Ableitung und gewährleistet eine möglichst glatte Verbindung der gegebenen Stützpunkte. In der Literatur werden Splines mit den Nebenbedingungen (3.8) *natürliche* kubische Splines (*natural cubic splines*)genannt. In vielen Fällen ist es jedoch wünschenswert, diese Nebenbedingungen den Gegebenheiten des konkreten Problems anzupassen. Solche Variationen zum Thema kubische Splines findet man in [2], S.146ff, ausführlich beschrieben. Hier soll nur kurz auf eine Form der Nebenbedingungen eingegangen werden, die in der Literatur (z. B. bei [21], S. 450f) oft den natürlichen Splines vorgezogen wird: die kubische Splinefunktion mit *not-a-knot*-Bedingung. Diese Bedingung mit

$$P^1(x) = P^2(x) \quad \text{für} \quad x_1 \leq x \leq x_3$$

und

$$P^{n-2}(x) = P^{n-1}(x) \quad \text{für} \quad x_{n-2} \leq x \leq x_n$$

besagt ([2], S.147), daß das erste und zweite Subintervall durch dasselbe Polynom und auch das vorletzte und letzte Subintervall durch dasselbe Polynom approximiert werden. Damit sind x_2 und x_{n-1} keine 'echten' Knoten der Splinefunktion mehr ('not-a-knot').

Die folgenden Ausführungen sind nun wieder für *natürliche* kubische Splines: Im Prinzip könnte man aus dem Gleichungssystem (3.7) und (3.8) alle Spline-Koeffizienten berechnen. In der Praxis geht man etwas anders vor, nämlich über die folgenden Gleichungen, die eindeutig aus (3.7) und (3.8) resultieren. Die entsprechende Ableitung ist nicht schwierig, aber sehr langwierig, und soll daher hier wegbleiben:

$$\begin{aligned} a_i &= y_i \quad i = 1, \ldots, n \\ c_1 &= 0 \\ c_n &= 0 \end{aligned} \tag{3.9}$$

$$h_{i-1}c_{i-1} + 2c_i(h_{i-1} + h_i) + h_i c_{i+1} = \frac{3r_i}{h_i} - \frac{3r_{i-1}}{h_{i-1}} \quad i = 2, \ldots, n-1 \tag{3.10}$$

mit $h_i = x_{i+1} - x_i$ und $r_i = y_{i+1} - y_i$.

(3.10) bedeutet ein lineares inhomogenes Gleichungssystem $(n-2)$-ter Ordnung für die Unbekannten $c_2 \cdots c_{n-1}$, wobei die Koeffizientenmatrix *tridiagonal und symmetrisch* ist.

Aus der Kenntnis der a_i und c_i ergeben sich für die übrigen Spline-Koeffizienten die Bestimmungsgleichungen

$$\left.\begin{aligned} b_i &= \tfrac{1}{h_i} r_i - \tfrac{h_i}{3}(c_{i+1} + 2c_i) \\ d_i &= \tfrac{1}{3h_i}(c_{i+1} - c_i) \end{aligned}\right. \quad i = 1, \ldots, n-1 \tag{3.11}$$

3.2.2 Das Programm SPLINE.

Das Programm SPLINE berechnet die Koeffizienten $a_i \ldots d_i$ natürlicher kubischer Splines für eine gegebene Menge von Stützpunkten $(x_i \mid y_i)$, $i = 1, \ldots, n$, wobei die x_i monoton ansteigend geordnet sein müssen.

INPUT-Parameter:

N: Anzahl der Stützpunkte.

X( ),Y( ): Koordinaten der Stützpunkte.

OUTPUT-Parameter:

A( ),B( ),C( ),D( ): eindimensionale Felder mit den Spline-Koeffizienten a_i, b_i, c_i und d_i.

Interne Felder:

H( ),R( ),HD( ),UD( ),OD( ),INHOM( ),LOES( ): .

Programmstruktur:

1. Check, ob die x_i monoton ansteigen. Wenn nicht, Fehlermitteilung und Return.
2. Berechnung der Koeffizienten des Gleichungssystems (3.10) und Lösung dieses Systems mittels TRID (s. Abschnitt 2.6.1).
3. Berechnung der Spline-Koeffizienten gemäß den Formeln (3.9) und (3.11).

Struktogramm 9 — SPLINE(N,X,Y,A,B,C,D)

```
I=1(1)N-1
    X(I) < X(I+1)
    Y:
        H(I):=X(I+1)-X(I)
        R(I):=Y(I+1)-Y(I)
    N:
        (return 'Reihenfolge der x inkorrekt')
J=1(1)N-2
    HD(J):=2.0*(H(J)+H(J+1))
    UD(J):=H(J)
    OD(J):=H(J+1)
    INHOM(J):=3.0*(R(J+1)/H(J+1)-R(J)/H(J))
TRID(UH,HD,OD,INHOM,N-2,LOES)
J=1(1)N-2
    C(J+1):=LOES(J)
C(1):=0.0
C(N):=0.0
I=1(1)N-1
    A(I):=Y(I)
    B(I):=R(I)/H(I)-H(I)*(C(I+1)+2.0*C(I))/3.0
    D(I):=(C(I+1)-C(I))/3.0/H(I)
(return)
```

3.2.3 Das Programm SPLVAL.

Das Programm SPLVAL (SPLine VALue) berechnet den Interpolationswert natürlicher kubischer Splines für ein beliebiges Argument $x \in [x_1, x_n]$.

N: Anzahl der Stützpunkte.

X( ): x-Koordinaten der Stützpunkte.

A( ),B( ),C( ),D( ): Koeffizienten kubischer Splines, wie sie z.B. vom Programm SPLINE berechnet werden.

XX: Argument, für welches die Interpolationsfunktion ausgewertet werden soll.

OUTPUT-Parameter:

YY: Interpolationswert.

YYD: erste Ableitung der Interpolationsfunktion an der Stelle XX.

YYDD: zweite Ableitung der Interpolationsfunktion an der Stelle XX.

FEHLER: diese Integer-Variable zur Fehlerdiagnostik hat nach der Exekution von SPLVAL den Wert '1', wenn XX außerhalb des Interpolationsbereiches liegt, sonst den Wert '0'.

Im Fall FEHLER=1 kehrt das Programm SPLVAL **ohne brauchbare Werte für YY, YYD und YYDD** ins aufrufende Programm zurück.

Anmerkung:

Der erste Teil von SPLVAL, in dem der Index J jenes Intervalls bestimmt wird, zu welchem XX gehört, ist ein Zitat aus dem Programm LOCATE ([9], S. 111).

Struktogramm 10 — SPLVAL(N,X,A,B,C,D,XX,YY,YYD,YYDD,FEHLER)

```
EPS:=0.00001
XX < (X(1)-EPS) or XX > (X(N)+EPS)
  Y: (error 'Argument nicht im
     Interpolationsbereich')
     FEHLER:=1
     (return)
  N: ......
FEHLER:=0
JL:=0
JU:=N+1
JU-JL > 1
    JM:=(JU+JL)/2
    XX > X(JM)
      Y: JL:=JM
      N: JU:=JM
J:=JL
J=0
  Y: J:=1
  N: ......
J=N
  Y: J:=N-1
  N: ......
Z:=XX-X(J)
YY:=((D(J)*Z+C(J))*Z+B(J))*Z+A(J)
YYD:=(3.0*D(J)*Z+2.0*C(J))*Z+B(J)
YYDD:=6.0*D(J)*Z+2.0*C(J)
(return)
```

3.2.4 Beispiele für die kubische Spline-Interpolation.

Beispiel 1: Demonstration einer Interpolation der fünf Stützpunkte

x	y
1.0	0.2
1.6	−0.1
1.9	−0.6
2.3	0.0
2.7	0.5

mittels natürlicher kubischer Splines.

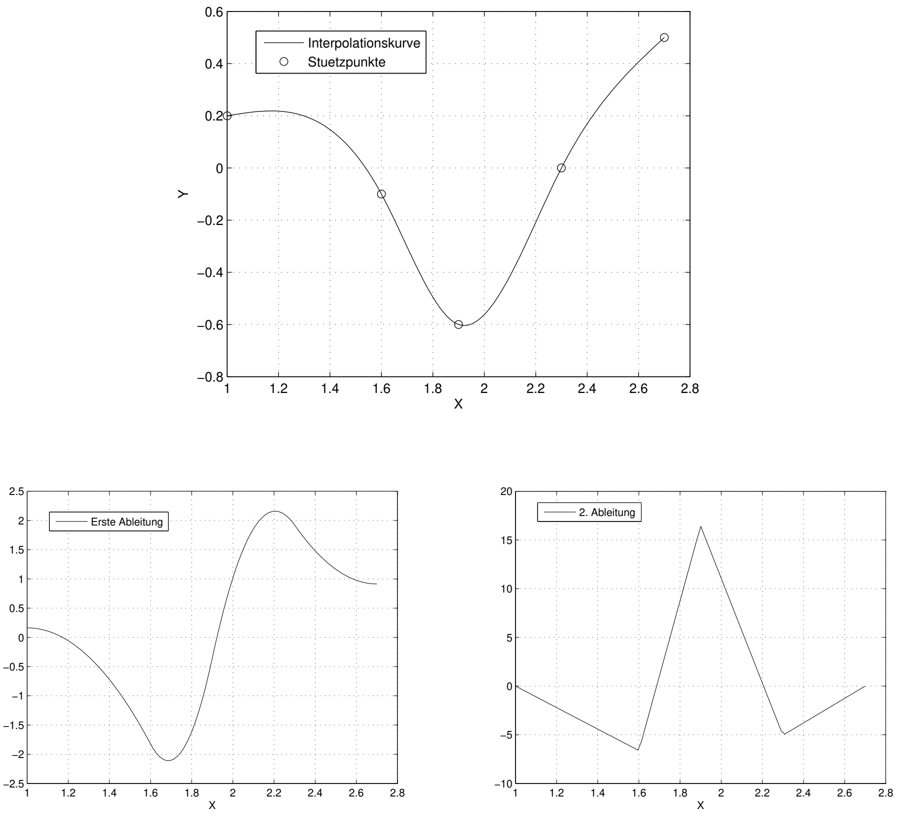


Abbildung 3.3: Demonstration einer Spline-Interpolation (Beispiel 1).

SPLINE liefert die folgenden Koeffizienten:

	A	B	C	D
1	0.2000	0.1628	0.0000	-1.8410
2	-0.1000	-1.8256	-3.3139	12.8117
3	-0.6000	-0.3547	8.2167	-8.9497
4	0.0000	1.9228	-2.5229	2.1024

Die entsprechende Interpolationsfunktion $I(x)$ sowie die erste und zweite Ableitung von $I(x)$ sind in der Abbildung 3.3 dargestellt.

Beispiel 2:

Die Leistungsfähigkeit einer Interpolation mittels kubischer Splines, insbesondere was die Glattheit der Interpolationskurve betrifft, kann auch der Abb.3.1 entnommen werden (ausgezogene Kurve).

Beispiel 3: Effektives Kernpotential eines Kalium-Atoms.

Das elektrische Potential in einem Atom kann in der sogenannten 'Hartree-Näherung' in der Form

$$V(\mathbf{r}) = \frac{Ze}{4\pi\epsilon_o r} - \frac{e}{4\pi\epsilon_o}\int d\mathbf{z}\frac{\rho_e(\mathbf{z})}{\mid \mathbf{r}-\mathbf{z}\mid} \tag{3.12}$$

geschrieben werden, wobei der erste Term das Kernpotential (Z = Kernladungszahl) und der zweite Term das Potential der den Kern umhüllenden Elektronen beschreibt. $\rho_e(|\mathbf{z}|)$ bedeutet die radialsymmetrische Näherung der Wahrscheinlichkeitsdichte der Elektronen an einem bestimmten Aufpunkt $\mathbf{z}$. Gleichung (3.12) läßt sich in der Form

$$V(r) = \frac{e}{4\pi\epsilon_o r}\underbrace{\left(Z - 4\pi\int_{z=0}^{r} dz z^2\rho_e(z) - 4\pi r\int_{z=r}^{\infty} dz z\rho_e(z)\right)}_{Z_{eff}(r)} \tag{3.13}$$

darstellen, wobei der Klammerausdruck die vom Radius r abhängige *effektive Kernladungszahl* darstellt.

Im folgenden soll nun $Z_{eff}(r)$ für das Kalium-Atom berechnet werden (Z=19), wobei $z^2\rho_e(z)$ in Form einer Tabelle[1] gegeben ist:

```
Radiale Elektronendichte fuer das Kalium-Atom.
C. Froese-Fischer, Atomic Data 4, 301-399 (1972).

Erste Spalte der Tabelle: Radius z in Bohr
Zweite Spalte der Tabelle: z*z*rho(z)
mit rho(z)=Wahrscheinlichkeitsdichte der Elektronen.

Zahl der Tabellenwerte:
98

 0.0000     0.0000000E+00
 0.0001     0.4498521E-04
 0.0002     0.1799408E-03
 0.0004     0.7158627E-03
 0.0006     0.1597434E-02
 0.0008     0.2819985E-02
 0.0010     0.4370591E-02
   .
   .
10.0000     0.1369750E-02
11.0000     0.6779086E-03
12.0000     0.3228840E-03
```

[1] Die zur Erstellung dieser Tabelle erforderlichen Elektronen-Wellenfunktionen stammen aus: C. Froese Fischer, Atomic Data **4**,301-399 (1972). Wenn Sie dieses Beispiel rechnen wollen, können Sie diese Daten von der Web-Seite dieser Vorlesung unter *testdaten/kalium.dat* herunterladen.

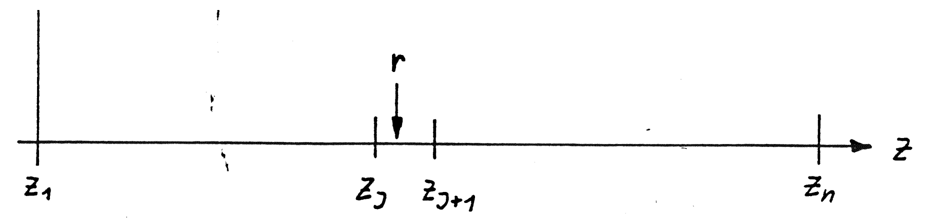


Abbildung 3.4: Integrationsbereiche der Integrale in Glg.3.12.

```
13.0000      0.1491913E-03
14.0000      0.6738352E-04
15.0000      0.2994823E-04
16.0000      0.1283439E-04
17.0000      0.5481809E-05
18.0000      0.2320359E-05
```

Die numerische Auswertung der beiden in (3.13) enthaltenen Integrale kann nun so erfolgen, daß man die Spline-Koeffizienten der Stützpunkte $[r_i \mid r_i^2 \rho_e(r_i)]$ (erstes Integral) bzw. der Stützpunkte $[r_i \mid r_i \rho_e(r_i)]$ (zweites Integral berechnet. Bezeichnet man die zur ersten bzw. zweiten Funktion gehörenden Spline-Koeffizienten mit $A \ldots D$ bzw. $a \ldots d$, so ergibt sich:

$$z^2 \rho_e(z) \approx A_i + B_i(z - z_i) + C_i(z - z_i)^2 + D_i(z - z_i)^3$$

beziehungsweise

$$z \rho_e(z) \approx a_i + b_i(z - z_i) + c_i(z - z_i)^2 + d_i(z - z_i)^3$$

für $z \in [z_i, z_{i+1}]$ und $i = 1, \ldots, n-1$. Mit diesen Approximationen geht man in die beiden Integrale, wobei zu berücksichtigen ist, daß der Integrationsbereich der ersten Integrals von 0 bis r und der des zweiten Integrals von r bis ∞ geht (∞ wird für die numerische Rechnung mit dem Argument des letzten Tabellenwertes gleichgesetzt). Liegt r im J-ten Intervall der Wertetabelle, so umfaßt der erste Integrationsbereich die ersten $J-1$ Intervalle sowie einen Teil des J-ten Intervalls, der zweite Integrationsbereich den Rest des J-ten Intervalls sowie alle restlichen Intervalle (siehe Abb.3.4).
Man erhält somit für das erste Integral näherungsweise

$$\sum_{i=1}^{J-1} \int_{z=z_i}^{z_{i+1}} dz \left[A_i + B_i(z - z_i) + C_i(z - z_i)^2 + D_i(z - z_i)^3\right] + \\ + \int_{z=z_J}^{r} dz \left[A_J + B_J(z - z_J) + C_J(z - z_J)^2 + D_J(z - z_J)^3\right]$$

und für das zweite Integral

$$\int_{z=r}^{z_{J+1}} dz \left[a_J + b_J(z - z_J) + c_J(z - z_J)^2 + d_J(z - z_J)^3\right] + \\ + \sum_{i=J+1}^{n-1} \int_{z=z_i}^{z_{i+1}} dz \left[a_i + b_i(z - z_i) + c_i(z - z_i)^2 + d_i(z - z_i)^3\right]$$

Diese Integrale können leicht ausgewertet werden, und man erhält auf diese Weise einen einfachen Algorithmus für die Berechnung der effektiven Kernladungszahl für einen beliebigen Kernabstand r im Intervall $r \in [0, z_n]$:

r[Bohr]	Zeff
0.00	18.9999
0.25	9.1943
0.50	5.8019
0.75	4.0352
1.00	2.7497
1.50	1.3052
2.00	0.7475
2.50	0.5122
3.00	0.3805
4.00	0.2152
5.00	0.1152
7.00	0.0279
10.00	0.0025
14.00	0.0001
18.00	0.0001

Dieses Ergebnis ist physikalisch leicht zu interpretieren: in großer Kernnähe wirkt die Ladung des Kalium-Kerns mit Ihren 19 Elementarladungen ungestört. Entfernt man sich jedoch vom Kern, so kompensieren die Hüllenelektronen sukzessive die Kernladung. In einem größeren Abstand ist diese 'Abschirmung' der Kernladung total, und das Atom bietet sich als neutrales Gebilde dar.

Beispiel 4:

Auch die Interpolation mit kubischen Splines ist kein fehlerfreies Verfahren. Wie die Abb. 3.2 (b) zeigt, kann es bei ungünstigen Stützpunkt-Verteilungen auch beim *splining* zu Oszillationen kommen.

3.2.5 Weitere Stichworte zum Thema: Interpolation.

Selbstverständlich gäbe es zum Thema *Interpolation* im allgemeinen und *Spline-Interpolation* im besonderen noch eine Menge Interessantes zu sagen. Ich muß mich hier auf einen knappen Themenkatalog beschränken. Einige dieser Themen werde ich in meiner ab dem SS 2002 angebotenen Lehrveranstaltung 'Ausgewählte Kapitel aus Numerische Methoden in der Physik' behandeln.

- Einen Überblick über die Verwendung von Spline-Funktionen samt Algorithmen finden Sie in dem Buch von H. Späth, *Spline-Algorithmen zur Konstruktion glatter Kurven und Flächen*, Oldenburg-Verlag München, 1978.

- Eine kurze Einführung in die Theorie mit vielen Formeln (ohne mathematische Beweise) bietet [2], S. 146ff. Dieses Buch (und die entsprechenden Werke für C [3] und PASCAL [4]) enthält auch eine Reihe von Programmlistings wie z.B.:
  Interpolation mittels kubischer Splines mit wählbaren Randbedingungen; Interpolation mittels periodischer kubischer Splines; mittels parametrischer kubischer Splines; Interpolation mittels Splines fünften Grades; Polynomiale Ausgleichssplines dritten Grades; ...; zweidimensionale Polynom-Splines dritten Grades usw.
  Es soll hier auch auf zwei neuere Publikationen von Frau Prof. Engeln-Müllges hingewiesen werden, nämlich:

  - [5] G. Engeln-Müllges und F. Reutter, *Numerik-Algorithmen mit Programmen in FORTRAN, C und Turbo Pascal*, VDI-Verlag, 1996.
  - [6] G. Engeln-Müllges and F. Uhlig, *Numerical Algorithms with C or FORTRAN*, Springer-Verlag, 1996.

  Die Programmcodes aus dem letzteren dieser Bücher [6] stehen Ihnen auf den PC's im Computerraum Physik unter

  ```
  /usr/local/numeric/num_alg_c/...     bzw.
  /usr/local/numeric/num_alg_f/...
  ```

  zur Verfügung.

- Zahlreiche Infomationen zum Thema Interpolation mit vielen Hinweisen auf entsprechende Software finden Sie auch im Buch von Überhuber [21], Kap. 9.

- Interpolation mittels rationaler Funktionen:

  $$I(x) = \frac{p_0 + p_1 x + \ldots + p_\mu x^\mu}{q_0 + q_1 x + \ldots + q_\nu x^\nu}$$

  mit $\nu + \mu + 1$ gleich Anzahl der Stützpunkte (q_0 beliebig!).

  <u>Literatur</u>: [10], S. 111ff, [2], S. 176/S. 399ff.

- Mehrdimensionale Interpolation:

  Literatur: [10], S. 123ff, [2], S. 164ff/S. 392ff.

3.2.6 Software-Angebot

Zusätzlich zu den im vorigen Abschnitt erwähnten Programmen weise ich auf die folgenden Software-Produkte hin:

PPPACK: (p.d.) Paket, erhältlich über NETLIB, für die Berechnung und Manipulation stückweiser Polynome. Allerdings nur F77-Programme, z. B. die Programme *cubspl.f* und *ppvalu.f*, die im wesentlichen die Funktionen der Programme SPLINE und SPLVAL in diesem Skriptum haben, jedoch für wählbare Randbedingungen (inklusive der not-a-knotBedingung).

IMSL: Diese kommerzielle Bibliothek[2] gibt es für Großrechner, Workstations und PSs in Fortran und C.
Sie enthält eine Reihe von Unterprogrammen, mit deren Hilfe kubische Splinefunktionen berechnet werden können bzw. Programme, mit welchen kubische Splinefunktionen an vorgegebener Stelle ausgewertet werden können:

IMSL/MATH-LIBRARY

/csint	Splines mit *not-a-knot* Randbedg.
/csdec	Splines mit vom Benutzer gewählten Randbedg.
/cscon	*optisch glatte* kubische Splinefunktion.
/csval	Auswertung einer Splinefunktion
/csval	Auswertung und Ableitung einer Splinefunktion.

Mathematica: Interpolation[table,InterpolationOrder → n]
InterpolatingFunction[range,table]

MATLAB: Einige einfach zu verwendende Funktionen gehören zur Grundausstattung von MATLAB, nämlich die ein- zwei- und dreidimensionalen polynomialen Interpolationsroutinen *interp1, interp2* und *interp3*.

Ein sehr umfangreiches Angebot bietet die 'Spline Toolbox' von MATLAB, über die Sie sich mittels der MATLAB-HELP-Funktion informieren können. Im folgenden finden Sie eine (unvollständige) Liste der angebotenen Funktionen sowie eine Kurzbeschreibung der Funktion *csape*:

[2]International Mathematical and Statistical Libraries, Visual Numerics Inc., Houston, Texas, USA).

csape - Cubic spline interpolation with various end-conditions
csapi - Cubic spline interpolant with not-a-knot end condition
csaps - Cubic smoothing spline
cscvn - `Natural' or periodic cubic spline curve
getcurve - Interactive creation of a cubic spline curve
ppmak - Put together a spline in ppform
spap2 - Least squares spline approximation
spapi - Spline interpolation
spaps - Smoothing spline
spcrv - Spline curve from control points by uniform subdivision
spmak - Put together a spline in B-form
rsmak - Put together a rational spline in rBform
rpmak - Put together a rational spline in rpform

csape

Purpose

Cubic spline interpolation with end conditions

Syntax

```
pp = csape(x,y)
pp = csape(x,y,conds,valconds)
```

Description

A cubic spline *s* (in ppform) with knot sequence `x` is constructed that satisfies *s*(`x(j)`) = `y(:,j)` for all `j`, as well as an additional *end condition* at the first and at the last data site, as specified by *conds* and *valconds*.

conds may be a *string* whose first character matches one of the following: `'complete'` or `'clamped'`, `'not-a-knot'`, `'periodic'`, `'second'`, `'variational'`, with the following meanings.

`'complete'`	Match endslopes (as given in *valconds*, with default as under "default")
`'not-a-knot'`	Make second and second-last sites inactive knots (ignoring *valconds* if given)
`'periodic'`	Match first and second derivatives at first site with those at last site
`'second'`	Match end second derivatives (as given in *valconds*, with default [0 0], i.e., as in `'variational'`)
`'variational'`	Set end second derivatives equal to zero (ignoring *valconds* if given)
default	Match endslopes to the slope of the cubic that matches the first four data at the respective end (i.e., Lagrange)

By giving *conds* as a 1-by-2 matrix instead, it is possible to specify *different* conditions at the two endpoints. Explicitly, $D^i s$ is given the value *valconds*(*j*) at the left (*j* = 1) respectively right (*j* = 2) endpoint in case conds(*j*) = *i*, *i* = 1,2. There are default values for *conds* and/or *valconds*.

3.3 Fourier-Analyse diskreter Daten

Gegeben sei eine Menge von n äquidistant verteilten Stützpunkten

$$(x_j = x_0 + j\Delta | y_i) \qquad j = 0, 1, \ldots, n-1 \quad ,$$

wobei die x reellwertig und die y (i. a.) komplexwertig sind. Um die Formeln zu vereinfachen, wird im folgenden $x_0 = 0$ angenommen. Diese Punktmenge soll nun unter Verwendung der im Abschnitt 3.1 erwähnten exponentiellen Basisfunktionen interpoliert werden:

$$y(x) = \frac{1}{n} \sum_{k=0}^{n-1} Y_k \, \mathrm{e}^{-i\alpha kx} \,. \tag{3.14}$$

Gleichzeitig soll angenommen werden, daß die n gegebenen Punkte Teil einer periodischen Funktion mit der Periode

$$P = \Delta \, n$$

sind.[3] Demzufolge muß die Interpolationsfunktion $I(x)$ ebenfalls bzgl. P periodisch sein, d.h. es muß gelten:

$$I(x + P) = I(x) \,.$$

Wie man leicht zeigen kann, ist diese Bedingung äquivalent mit einer speziellen Wahl der bisher freien Konstanten α:

$$\frac{1}{n} \sum_{k=0}^{n-1} Y_k \, \mathrm{e}^{-i\alpha k(x+P)} = \frac{1}{n} \sum_{k=0}^{n-1} Y_k \, \mathrm{e}^{-i\alpha kx} \,.$$

Dies ist offenbar der Fall, wenn gilt

$$\mathrm{e}^{-i\alpha kP} = \mathrm{e}^{-i\alpha kn\Delta} = 1 \,,$$

was der Bedingung

$$\alpha = \frac{2\pi}{\Delta n}$$

entspricht. Es ergibt sich somit für die Interpolationsfunktion der Ausdruck

$$I(x) = \frac{1}{n} \sum_{k=0}^{n-1} Y_k \, \exp\left[-i\frac{2\pi kx}{\Delta n}\right] \,. \tag{3.15}$$

Nach der Interpolationsbedingung (3.1) gilt nun

$$I(x_j) = \frac{1}{n} \sum_{k=0}^{n-1} Y_k \, \exp\left(-i\frac{2\pi kj}{n}\right) = y_j \quad (j = 0, 1, \ldots, n-1) \tag{3.16}$$

mit den i. a. komplexwertigen Fourierkoeffizienten Y_k.

[3] Es spielt dabei keine Rolle, ob die zu interpolierenden Funktionswerte *tatsächlich* eine periodische Funktion beschreiben!

Selbstverständlich könnten die Y_k direkt aus dem linearen Gleichungssystem (3.16) ermittelt werden. Im Falle der Fourier-Entwicklung geht das aber sehr viel einfacher. Multipliziert man nämlich die Glg. (3.16) mit $\exp(i2\pi k'j/n)$ und summiert über den Index j, so ergibt sich

$$\sum_{j=0}^{n-1} y_j \exp\left(i\frac{2\pi k'j}{n}\right) = \frac{1}{n}\sum_{k=0}^{n-1} Y_k \underbrace{\sum_{j=0}^{n-1} \exp\left(i\frac{2\pi j}{n}(k'-k)\right)}_{(A)} . \qquad (3.17)$$

o.B.: der Ausdruck (A) hat im Falle äquidistanter Stützpunkte das einfache Ergebnis

$$(A) = n \cdot \delta_{k,k'} \qquad \text{'Kronecker-Delta': } 1 \text{ für } k = k' \text{ sonst Null.}$$

Die Ursache dafür ist, daß die Fourier-Entwicklungsfunktionen bzgl. einer äquidistanten Punktmenge *orthogonal* sind.
Somit erhält man aus Glg. (3.17) unmittelbar

$$\sum_{j=0}^{n-1} y_j \exp\left(i\frac{2\pi k'j}{n}\right) = \frac{1}{n}\sum_{k=0}^{n-1} Y_k \, n \, \delta_{k,k'} = Y_{k'} .$$

Es ergibt sich somit für die *Fourierkoeffizienten* Y_k die einfache Bestimmungsgleichung

$$Y_k = \sum_{j=0}^{n-1} y_j \exp\left(i\frac{2\pi kj}{n}\right) \qquad (k = 0, 1, \ldots, n-1) . \qquad (3.18)$$

Man nennt die Auswertung (3.18), also die Berechnung der Y_k aus den y_j, die Fourier-Transformation (FT) der y_j; die Auswertung von (3.16) heißt dementsprechend die inverse FT.

Wegen der komplexwertigen Funktionenbasis der Fourier-Summe sind die Fourierkoeffizienten Y_k im allgemeinen ebenfalls komplexwertig. Ein in der Praxis wichtiger Sonderfall ist gegeben, wenn die Datenwerte y_j reell sind. In diesem Fall haben die Fourierkoeffizienten die Eigenschaft[4]

$$Y_k = Y^{\star}_{n-k} \quad . \qquad (3.19)$$

3.3.1 Numerische Berechnung der Fourierkoeffizienten

Die numerische Auswertung von (3.18) ist ein Prozess der Ordnung n^2, d.h. es sind n Summenterme für jeden der n Koeffizienten zu berechnen. Dies bedeutet für große n einen sehr zeitintensiven Rechenprozess.

In bezug auf die Rechenzeit bietet die sogenannte *Fast Fourier Transform FFT*, die auf einer rekursiven Berechnung der in (3.18) vorkommenden Summe beruht, einen wesentlichen Fortschritt. Dieser Algorithmus, der auf den

[4] Bitte selbst verifizieren.

Ergebnissen einer Arbeit von Danielson und Lanczos (1942) beruht, ist zwar im Prinzip einfach, seine geschickte Programmierung ist jedoch im Detail etwas trickreich. Er soll deshalb hier nur prinzipiell erläutert werden, und zwar für das Beispiel von $n = 8$ Stützpunkten. Spaltet man die Summe (3.18) in zwei Teilsummen auf, und zwar in der Form

$$Y_k = \sum_{j=0}^{7} y_j e^{i \cdot 2\pi kj/8} = \sum_{j=0(2)}^{6} y_j e^{i \cdot 2\pi kj/8} + \sum_{j=1(2)}^{7} y_j e^{i \cdot 2\pi kj/8} \quad ,$$

so erhält man durch geeignete Transformation der Summenindizes den Ausdruck

$$Y_k = \sum_{j=0}^{3} y_{2j} e^{i \cdot 2\pi kj/4} + e^{i \cdot 2\pi k/8} \sum_{j=0}^{3} y_{2j+1} e^{i \cdot 2\pi kj/4}$$

bzw. unter Verwendung der Abkürzung $W_m = \exp(i \cdot 2\pi/m)$

$$Y_k = \sum_{j=0}^{7} W_8^{j \cdot k} y_j = \underbrace{\sum_{j=0}^{3} W_4^{j \cdot k} y_{2j}}_{Y_k^0} + W_8^k \underbrace{\sum_{j=0}^{3} W_4^{j \cdot k} y_{2j+1}}_{Y_k^1}$$

Die beiden Teilsummen Y_k^0 und Y_k^1 können nun ihrerseits wieder zerlegt werden:

$$Y_k^0 = \underbrace{\sum_{j=0}^{1} W_2^{j \cdot k} y_{4j}}_{Y_k^{00}} + W_4^k \underbrace{\sum_{j=0}^{1} W_2^{j \cdot k} y_{4j+2}}_{Y_k^{01}}$$

$$Y_k^1 = \underbrace{\sum_{j=0}^{1} W_2^{j \cdot k} y_{4j+1}}_{Y_k^{10}} + W_4^k \underbrace{\sum_{j=0}^{1} W_2^{j \cdot k} y_{4j+3}}_{Y_k^{11}}$$

Die letzte Aufspaltung der vier Größen Y_k^{00}, Y_k^{01}, Y_k^{10} und Y_k^{11} lautet:

$$Y_k^{00} = y_0 + W_2^k y_4 \qquad Y_k^{01} = y_2 + W_2^k y_6$$

$$Y_k^{10} = y_1 + W_2^k y_5 \qquad Y_k^{11} = y_3 + W_2^k y_7$$

Es sind somit die Fourierkoeffizienten 'auf die Ebene der y-Werte reduziert'. Davon ausgehend, kann der gewünschte Koeffizient Y_k schrittweise aufgebaut werden, wie dies schematisch in der Abbildung 3.5 dargestellt ist.

Die eben präsentierte Vorgangsweise funktioniert jedoch nur dann, wenn die bei dem Aufbau der Koeffizienten gemäß Abb. 3.5 berücksichtigten Stützpunkte von Schritt zu Schritt verdoppelt werden können, d.h. *wenn die Stützpunktanzahl n eine Potenz von 2 ist* [5].

Wie ebenfalls aus Abb. 3.5 hervorgeht, müssen die gegebenen y_j-Werte $(j = 0, \ldots, n-1)$ am Beginn der Kaskade (links in der Abb.) 'geeignet'

[5] Diese strenge Bedingung ist in einigen modernen FFT-Programmen (s. Abschnitt 3.4.5) *nicht mehr* gegeben.

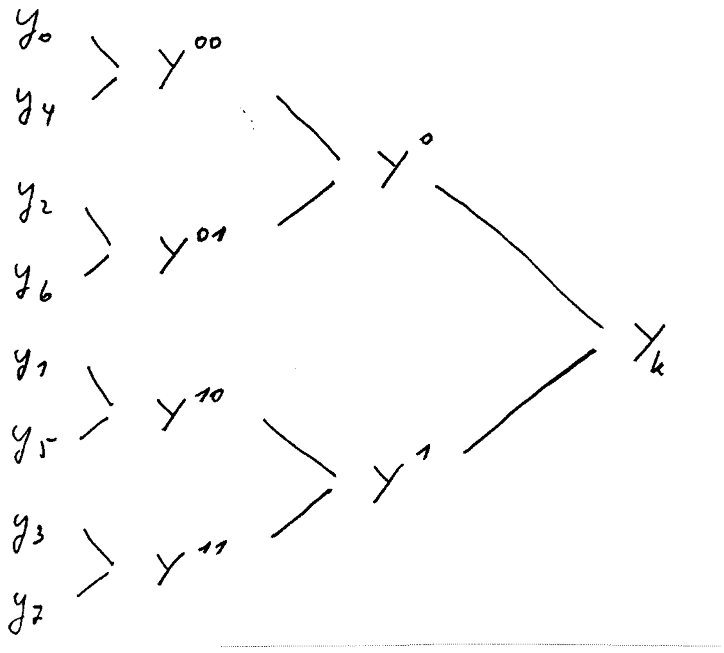


Abbildung 3.5: Grundprinzip der 'Fast Fourier Transform'.

geordnet werden. Es zeigt sich, daß diese Indexordnung sich einfach aus einer *Bitumkehr* der ursprünglichen Indizes in dualer Schreibweise ergibt:

ursprüngliche Folge (Index)	Dual	BITUMKEHR	neue Folge (Index)
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Der große Vorteil der FFT gegenüber der direkten Auswertung von (3.18) liegt in der beträchtlichen Einsparung an Rechenzeit. Während nämlich bei der Berechnung der Fourierkoeffizienten gemäß (3.18) die Auswertung von n^2 Summentermen nötig ist, reduziert sich diese Zahl bei der FFT auf $n \ln_2(n)$:

n	n^2	$n \ln_2(n)$
128	16384	896
1024	1048576	10240
.	.	.
.	.	.
1048576	$\approx 10^{12}$	$\approx 2.1 \cdot 10^7$

Bei sehr großen Datenmengen ist eine Fourier-Transformation mittels kleiner Computer (PC) überhaupt nur mittels der FFT möglich!

3.3.2 FFT-Programme in C, F90 und MATLAB

FFT-Programme sind häufig recht raffiniert programmiert. Es erscheint im Rahmen dieser LV nicht sinnvoll, auf alle Details der Algorithmen einzugehen. Aus diesem Grund werden im folgenden (ausnahmsweise) keine Struktogramme, sondern direkt FFT-Programmlistings in den Sprachen C, F90 und MATLAB präsentiert. Es handelt sich dabei um Programme zur schnellen Fourier-Transformation einer diskreten komplexwertigen Punktmenge mit 2^m Werten (m = positive Integerzahl) und mit äquidistanten x-Komponenten.

Das C-Programm FOUR1

Quelle: [10], S. 507f.

INPUT-Parameter:

DATA( ): eindimensionales Real-Feld, das die Real- und Imaginärteile der Größen enthält, die Fourier-transformiert werden sollen.

Die DATA-Werte müssen wie folgt abgespeichert sein:

```
RT{y0}         DATA(1)
IT{y0}         DATA(2)

RT{y1}         DATA(3)
IT{y1}         DATA(4)
   .
   .
RT{y(n-1)}     DATA(2n-1)
IT{y(n-1)}     DATA(2n)
```

DATA muß also mindestens $2n$ Werte aufnehmen können!

NN: Anzahl der n (i. a. komplexen) Werte.

ISIGN=+1: Fourier-Transformation.

ISIGN=−1: inverse Fourier-Transformation (aber ohne den Normierungsfaktor $1/n$).

OUTPUT-Parameter:

DATA( ): Real-Feld mit den Real- und Imaginärteilen der Fourier-Transformierten.

```c
#include <math.h>
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
```

```
void four1(double data[], unsigned long nn, int isign)
{
unsigned long n,mmax,m,j,istep,i;
double wtemp,wr,wpr,wpi,wi,theta;
double tempr,tempi;

n=nn << 1;
j=1;
for (i=1;i<n;i+=2) {
if (j > i) {
SWAP(data[j],data[i]);
SWAP(data[j+1],data[i+1]);
}
m=n >> 1;
while (m >= 2 && j > m) {
j -= m;
m >>= 1;
}
j += m;
}
mmax=2;
while (n > mmax) {
istep=mmax << 1;
theta=isign*(6.28318530717959/mmax);
wtemp=sin(0.5*theta);
wpr = -2.0*wtemp*wtemp;
wpi=sin(theta);
wr=1.0;
wi=0.0;
for (m=1;m<mmax;m+=2) {
for (i=m;i<=n;i+=istep) {
j=i+mmax;
tempr=wr*data[j]-wi*data[j+1];
tempi=wr*data[j+1]+wi*data[j];
data[j]=data[i]-tempr;
data[j+1]=data[i+1]-tempi;
data[i] += tempr;
data[i+1] += tempi;
}
wr=(wtemp=wr)*wpr-wi*wpi+wr;
wi=wi*wpr+wtemp*wpi+wi;
}
mmax=istep;
}
}
#undef SWAP
/* (C) Copr. 1986-92 Numerical Recipes Software +. */
```

Das F90-Programm FFT

Quelle: [23], S. 322f.

INPUT-Parameter:

A(): eindimensionables Datenfeld vom Typ

`COMPLEX(KIND=KIND(0.0d0))` ,

das die komplexwertigen Größen enthält, die Fourier-transformiert werden sollen.

M: $2**M$ ist die Zahl n der (äquidistanten) Datenwerte.

INV=+1: Fourier-Transformation.

INV$\neq -1$: inverse Fourier-Transformation.

OUTPUT-Parameter:

A( ): COMPLEX-Feld mit den Real- und Imaginärteilen der Fourier-Transformierten.

```
      SUBROUTINE fft(a,m,inv)
! Paul L. DeVries, Department of Physics, Miami University

! This subroutine performs the Fast Fourier Transform by
! the method of Cooley and Tukey

!        start:                1965
!  last modified:              1993
!  Modifications H. Sormann:   2000
!

!  Parameter description by H. Sormann 2000
!  PARAMETERS:   a     vector of complex data which are to be
!                      Fourier-transformed
!                      this vector is VARIABLY dimensioned (see below)

!                      On return, a  contains the FT coefficients.
!
!                m     2**m  is the number of data points
!
!                inv equal 1        Fourier transform
!                inv not equal 1    INVERSE Fourier transform
!
      IMPLICIT NONE
```

```
      INTEGER                          :: m,inv,n,nd2,i,j,k,l,le,le1,ip
      COMPLEX(KIND=KIND(0.0d0))                  :: u,w,t
      COMPLEX(KIND=KIND(0.0d0)),DIMENSION(*) :: a   ! * = var. dimension
      DOUBLE PRECISION                 :: ang,pi
!      PARAMETER (pi=3.141592653589793d0)
      pi=4.d0*atan(1.d0)

      n=2**m
      nd2=n/2
      j=1
      DO i=1,n-1
        IF(i .lt. j)THEN
          t=a(j)
          a(j)=a(i)
          a(i)=t
        ENDIF
        k=nd2
100     IF(k .lt. j)THEN
          j=j-k
          k=k/2
          goto 100
        ENDIF
        j=j+k
      END DO
      le=1
      DO l=1,m
        le1=le
        le=le+le
        u=(1.d0,0.d0)
        ang=pi/dble(le1)
        w=cmplx(cos(ang), -sin(ang),KIND(0.d0))
        IF(inv .eq. 1)w=conjg(w)
        DO j=1,le1
          DO i=j,n,le
            ip=i+le1
            t=a(ip)*u
            a(ip)=a(i)-t
            a(i)=a(i)+t
          END DO
          u=u*w
        END DO
      END DO
      IF(inv .ne. 1)THEN
        DO i=1,n
          a(i)=a(i)/dble(n)
        END DO
```

```
      ENDIF
      END SUBROUTINE fft
```

Das MATLAB-Programm fft

Es gibt eine interne MATLAB-Routine mit dem Funktionsnamen *fft.m*. Wenn Sie sich mittels *help fft* über diese Routine informieren, erhalten Sie den folgenden Text:

```
%FFT Discrete Fourier transform.
%   FFT(X) is the discrete Fourier transform (DFT) of vector X.  If the
%   length of X is a power of two, a fast radix-2 fast-Fourier
%   transform algorithm is used.  If the length of X is not a
%   power of two, a slower non-power-of-two algorithm is employed.
%   For matrices, the FFT operation is applied to each column.
%   For N-D arrays, the FFT operation operates on the first
%   non-singleton dimension.
%
%   FFT(X,N) is the N-point FFT, padded with zeros if X has less
%   than N points and truncated if it has more.
%
%   FFT(X,[],DIM) or FFT(X,N,DIM) applies the FFT operation across the
%   dimension DIM.
%
%   For length N input vector x, the DFT is a length N vector X,
%   with elements
%                    N
%      X(k) =       sum  x(n)*exp(-j*2*pi*(k-1)*(n-1)/N), 1 <= k <= N.
%                   n=1
%   The inverse DFT (computed by IFFT) is given by
%                    N
%      x(n) = (1/N) sum  X(k)*exp( j*2*pi*(k-1)*(n-1)/N), 1 <= n <= N.
%                   k=1
%
%   The relationship between the DFT and the Fourier coeffs a and b in
%            N/2
%   x(n)=a0+ sum a(k)*cos(2*pi*k*t(n)/(N*dt))+b(k)*sin(2*pi*k*t(n)/(N*dt))
%            k=1
%   is
%      a0 = X(1)/N, a(k) = 2*real(X(k+1))/N, b(k) = -2*imag(X(k+1))/N,
%   x is a length N discrete signal sampled at times t with spacing dt.
%
%   See also IFFT, FFT2, IFFT2, FFTSHIFT.

%   Copyright (c) 1984-98 by The MathWorks, Inc.
%   $Revision: 5.9 $  $Date: 1998/06/30 18:43:41 $

%   Built-in function.
```

Ein MATLAB-Programm, welches das Testproblem im Abschnitt 3.3.3 lösen soll, könnte etwa so aussehen:

```
%Eingabe der Testdaten von Skriptum:
n=8;
delta=1.0;
period=n*delta;

index=0:1:n-1;

werte=[0.7013+0.0437i -0.0724+0.5133i 0.0988-0.2688i 0.0715-0.1162i ...
       0.4013+0.1188i -0.0901-0.1408i -0.1263-0.0688i 0.2660-0.3813i];


yy=fft(werte);
```

Nun folgt eine zwar im Prinzip harmlose, aber lästige Komplikation: wenn man die Ergebnisse der Fourier-Transformation mittels dieses Matlab-Programms mit den entsprechenden Resultaten des C-Programms *four1.c* bzw. des Fortran-Programms *fft.f90* vergleicht, so erkennt man, daß *die Reihenfolge der Fourierkoeffizienten bei Matlab sich von den anderen Ergebnissen unterscheidet* (vgl. das Testbeispiel auf S. 83).

Das heißt nicht etwa, daß das Matlab-Programm falsch rechnet; die Ursache ist vielmehr, daß die Matlab-Version von einer zu Glg. (3.15) etwas verschiedenen - wenn auch mathematisch äquivalenten - Grundformel für die Interpolationsfunktion ausgeht.

Um nun alle daraus folgenden Unsicherheiten zu eliminieren, habe ich neue, sehr einfache Variationen der Matlab-Routinen *fft.m* bzw. *ifft.m* geschrieben, bei denen durch einfache Umordnungen der Fourierkoeffizienten dieses Problem nicht mehr auftritt:

```
function fourier = fft_sor(x)

% H. Sormann   20-9-2007
% last update  20-9-2007

% Um auch bei den Matlab-Programmen fft.m und ifft.m dieselbe
% Reihenfolge der Fourierkoeffizienten zu erhalten wie bei
% den im Skriptum praesentierten C-Programm (four1.c) bzw.
% F90-Programm (fft.f90), habe ich dieses einfache Programm
% geschrieben, bei welchem NACH der Anwendung von fft.m
% "automatisch" eine Umordnung der Matlab-Ergebnisse erfolgt.

% Wichtig ist, dass Sie die Programme fft_sor.m und ifft_sor.m
% paarweise verwenden, also nicht fft_sor.m mit ifft.m
% kombinieren und umgekehrt.

  n=length(x);
  d=fft(x);

  fourier(1)=d(1);
  for ind=2:n
    fourier(ind)=d(n+2-ind);
  end


function ifourier = ifft_sor(d)

% H. Sormann   20-9-2007
% last update  20-9-2007

% Um auch bei den Matlab-Programmen fft.m und ifft.m dieselbe
% Reihenfolge der Fourierkoeffizienten zu erhalten wie bei
% den im Skriptum praesentierten C-Programm (four1.c) bzw.
% F90-Programm (fft.f90), habe ich dieses einfache Programm
% geschrieben, bei welchem VOR der Anwendung von ifft.m
% "automatisch" eine Umordnung der Matlab-Ergebnisse erfolgt.

% Wichtig ist, dass Sie die Programme ifft_sor.m und fft_sor.m
% paarweise verwenden, also nicht ifft_sor.m mit fft.m
% kombinieren und umgekehrt.

  n=length(d);
  x(1)=d(1);
  for ind=2:n
    x(ind)=d(n+2-ind);
  end

  ifourier=ifft(x);
```

3.3.3 Ein Test für die FFT-Programme.

Dieses Beispiel beschreibt die Fourier-Transformation der folgenden Testdaten ($n = 8$, $\Delta = 1.$):

j	RT{y(j)}	IT{y(j)}
0	0.7013	0.0437
1	-0.0724	0.5133
2	0.0988	-0.2688
3	0.0715	-0.1162
4	0.4013	0.1188
5	-0.0901	-0.1408
6	-0.1263	-0.0688
7	0.2660	-0.3813

Die gegebenen Stützstellen repräsentieren somit eine periodische Funktion mit der Periode $P = 8$, und das Programm FOUR1 liefert die folgenden Fourierkoeffienten:

k	RT{Y(k)}	IT{Y(k)}
0	1.2501	-0.3001
1	0.0001	0.3000
2	0.2601	0.0001
3	-0.7000	-0.7003
4	0.9001	-0.0501
5	0.9999	0.0000
6	2.0001	1.0001
7	0.9000	0.0999

Beachten Sie, daß das *originale* Matlab-Programm *fft.m* eine veränderte Reihenfolge der Fourierkoeffizienten ausgibt (s. die Diskussion auf Seite 82f):

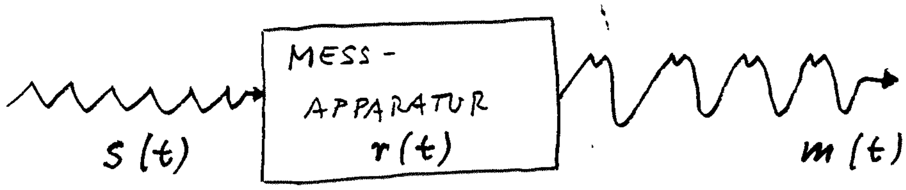
k	RT{Y(k)}	IT{Y(k)}
0	1.2501	-0.3001
1	0.9000	0.0999
2	2.0001	1.0001
3	0.9999	-0.0000
4	0.9001	-0.0501
5	-0.7000	-0.7003
6	0.2601	0.0001
7	0.0001	0.3000

3.4 Wichtige Anwendungen der FT

In diesem Abschnitt sollen als Beispiele für die vielfältigen Verwendungsmöglichkeiten der FT in der Physik, Meßtechnik etc. drei Anwendungsbereiche erläutert werden.

3.4.1 Faltungsintegrale

Wenn irgendein Signal mittels einer Meßapparatur registriert wird, wird in den meisten Fällen das ursprüngliche Signal durch die Eigenschaften der Meßapparatur verändert:



Für die konkrete Auswertung der Meßdaten ist es i. a. sehr wichtig, diesen Einfluß der Apparatur im nachhinein so gut als möglich zu eliminieren.

o. B.: Nennt man die Signalfunktion $s(t)$ und die gemessene Funktion $m(t)$, so ist der Zusammenhang zwischen diesen Funktionen durch das sogenannte Faltungsintegral

$$m(t) = \int_{-\infty}^{+\infty} dt'\, r(t')\, s(t-t') \tag{3.20}$$

gegeben. Der Einfluß der Meßapparatur wird dabei durch die *Apparatefunktion* oder *Auflösungsfunktion (resolution function)* $r(t)$ beschrieben.

Es soll nun angenommen werden, daß die Funktionen $m(t)$ und $s(t)$ periodisch bzgl. der Periode P sind. Außerdem soll angenommen werden, daß die Apparatefunktion auf diese Periode *beschränkt* sei, d.h. daß gilt:

$$r(t) = 0 \qquad \text{außerhalb} \quad [0, P]\,.$$

Unter diesen Umständen kann der Integrationsbereich in (3.20) auf $0 \le t' \le P$ beschränkt werden:

$$m(t) = \int_0^P dt'\, r(t')\, s(t-t')\,.$$

Dieses Integral kann nun mit Hilfe der Trapezformel (s. Kap. 6) näherungsweise ausgewertet werden. Mittels der im Integrationsintervall $[0, P]$ äquidistant verteilten Punkte

$$t_j = j\Delta \qquad \Delta = \frac{P}{n} \qquad (j = 0, 1, \ldots, n-1)$$

erhält man

$$m(t) \approx \Delta \sum_{j=0}^{n-1} r(t_j)\, s(t-t_j)\,.$$

Interessiert man sich nur für die Argumente

$$t_\ell = \ell\Delta\,,$$

so ergibt sich weiters

$$m(t_\ell) \approx \Delta \sum_{j=0}^{n-1} r(t_j)\, s(t_\ell - t_j)$$

bzw.

$$m_\ell \approx \Delta \sum_{j=0}^{n-1} r_j\, s_{\ell-j}\,. \tag{3.21}$$

Beachten Sie, daß bei der Auswertung der obigen Summe für r negative Indizes vorkommen. Dies ist jedoch kein Problem, da wegen der (angenommenen) Periodizität von $r(t)$ gelten muß:

$$s_{-i} = s_{n-i}\,.$$

Nun kommt die FT ins Spiel! Setzt man nämlich für die drei in Glg. (3.21) vorkommenden Größen m, r und s die entsprechenden Fourierreihen gemäß (3.16) ein, so ergibt sich

$$\begin{aligned}
\frac{1}{n}\sum_{k=0}^{n-1} M_k\,\mathrm{e}^{-i2\pi k\ell/n} &\approx \Delta \sum_{j=0}^{n-1} \frac{1}{n}\sum_{k=0}^{n-1} S_k\,\mathrm{e}^{-i2\pi k(\ell-j)/n}\,\frac{1}{n}\sum_{k'=0}^{n-1} R_{k'}\,\mathrm{e}^{-i2\pi k' j/n} \\
&= \frac{\Delta}{n^2}\sum_{k=0}^{n-1}\sum_{k'=0}^{n-1} S_k\,R_{k'}\,\mathrm{e}^{-i2\pi k\ell/n} \underbrace{\sum_{j=0}^{n-1} \mathrm{e}^{i2\pi(k-k')j/n}}_{n\,\delta_{k,k'}} \\
&= \frac{\Delta}{n}\sum_{k=0}^{n-1} R_k\,S_k\,\mathrm{e}^{-i2\pi k\ell/n}\,.
\end{aligned}$$

Daraus folgt sofort die sehr wichtige Beziehung

$$M_k = \Delta\, R_k\, S_k\,. \tag{3.22}$$

D. h.: Abgesehen von der Konstanten Δ ist die FT des Faltungsergebnisses $m(t)$ einfach durch das Produkt der FT von r und s gegeben. Die Bedeutung von (3.22) für die Praxis ergibt sich daraus, daß man diese Gleichung ohne Probleme nach der *ungestörten* Signalfunktion auflösen kann:

$$S_k = \frac{1}{\Delta}\,\frac{M_k}{R_k}\,. \tag{3.23}$$

Kennt man also die gemessene Funktion $m(t)$ und die Apparatefunktion $r(t)$ im realen Raum, so kann man wie folgt vorgehen:

1. FT von $m(t) \quad\to\quad M_k$
2. FT von $r(t) \quad\to\quad R_k$
3. Berechnung von S_k aus Glg. (3.23)
4. Inverse FT von $S_k \quad\to\quad s(t)$

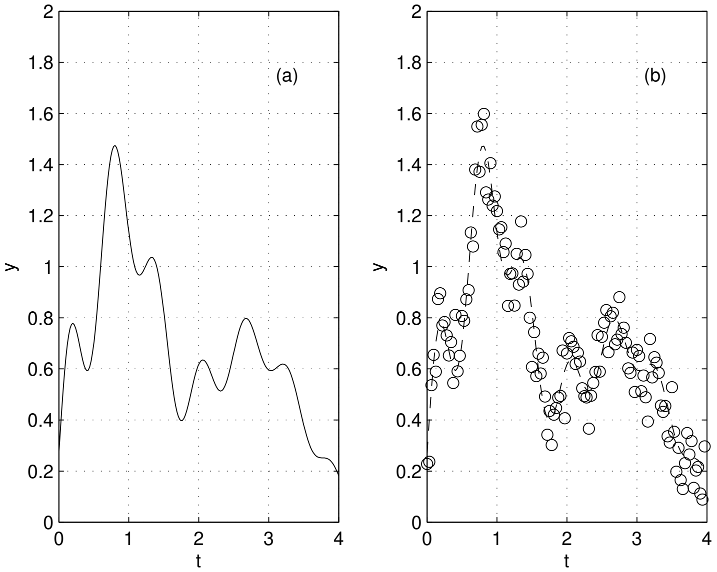


Abbildung 3.6: Demonstration einer Datenglättung mittels FT:
(a) ungestörtes Signal, (b) verrauschtes Signal.

3.4.2 Datenglättung

Zur Demonstration dieser Anwendung der FT zeigt die Abbildung 3.6 die Testfunktion

$$y(x) = \mathrm{e}^{-x/2}\left[2\mathrm{e}^{-2(x-1)^2} + 3\mathrm{e}^{-(x-3)^2} + \frac{1}{3}\sin(10x)\right]$$

im Intervall $0 \leq x \leq 4$. Abb. 3.6 (a) enthält die ungestörte Funktion, und in Abb. 3.6 (b) sind dem $y(x)$ statistische Fehler mit einer Standardabweichung von $\sigma = 0.1$ überlagert, wie sie bei Messungen unvermeidlich auftreten.
Die entsprechenden Beträge der Fourierkoeffizienten für n=128 sind in der Abb. 3.7 dargestellt. Wie man deutlich sieht, wirken sich die statistischen Fehler so aus, daß sich im 'mittleren Bereich' von k (zum Unterschied von der glatten Kurve) von Null verschiedene Fourierkoeffizienten zeigen. Es liegt daher nahe, diese Koeffizienten dem *noise* zuzuschreiben und in geeigneter Weise zu unterdrücken.
Dies kann am einfachsten durch die *Regel von Dirichlet* geschehen, welche durch

$$\hat{Y}_k = Y_k \qquad \text{für} \quad 0 \leq k < k_0 \quad \text{und} \quad n - k_0 < k < n$$
$$\hat{Y}_k = 0 \qquad \text{für} \quad k_0 \leq k \leq n - k_0$$

definiert ist, wobei der Index k_0 frei wählbar ist.

Es ist natürlich klar, daß die Qualität der *noise*-Unterdrückung entscheidend vom gewählten k_0 abhängt. In unseren Beispiel hat sich $k_0 = 9$ als sehr gute Wahl herausgestellt; allerdings muß zugegeben werden, daß die Kenntnis der *Idealkurve* - die natürlich im Praxisfall nicht gegeben ist - diese Wahl sehr erleichtert! Die hohe Qualität der Dirichlet-Glättung wird in der Abb. 3.8 demonstriert.

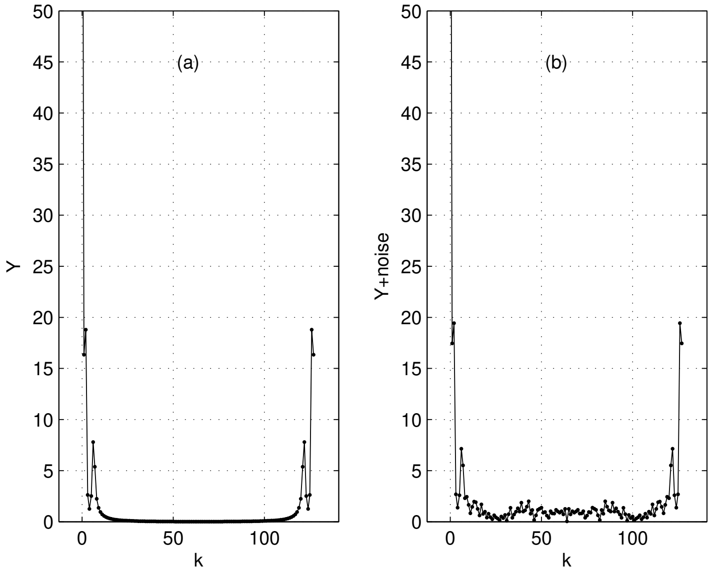


Abbildung 3.7: Demonstration einer Datenglättung mittels FT:
(a) FT des ungestörten Signals, (b) FT des verrauschten Signals.

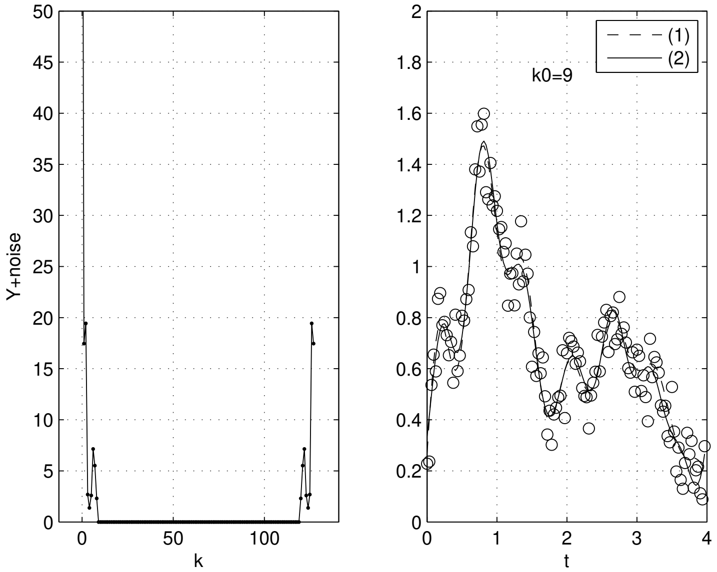


Abbildung 3.8: Demonstration einer Datenglättung mittels FT:
(a) Durchführung der Dirichlet-Dämpfung im Fourierraum für $k_0 = 9$,
(b) Wirkung dieser Dirichlet-Dämpfung im realen Zeitraum:
strichlierte Kurve: ungestörtes Signal;
volle Kurve: Dirichlet-behandeltes Signal.

3.4.3 Frequenz-Analyse

Ausgangspunkt für die folgenden Überlegungen ist die Interpolationskurve (3.15)

$$I(t) = \frac{1}{n}\sum_{k=0}^{n-1} Y_k \exp\left[-i\frac{2\pi kt}{P}\right] = \exp\left(-i2\pi\nu_k t\right) \ .$$

Die Frequenzen, die in die obige Entwicklung einbezogen werden, lauten

$$\nu_k = \frac{k}{P} \qquad \text{mit} \quad k = 0, \ldots, n-1$$

mit n als Zahl der Meßpunkte und P als Periode des Meß-Signals.

Aus dieser Darstellung geht auch hervor, warum man diese Fourier-Transformation als Transformation *aus dem Zeitraum in den Frequenzraum* bezeichnet. Mathematisch völlig äquivalent ist auch die Transformation *vom Ortsraum in den Wellenzahlraum.*

Nun soll angenommen werden, daß nicht nur die gegebenen (gemessenen) Werte y_j, sondern die gesamte Funktion $f(t)$, welche 'hinter diesen Werten steht', reell ist. In diesen Fall interessiert vor allem der Realteil der Interpolationsfunktion, nämlich

$$\Re\{I(t)\} = \sum_{k=0}^{n-1}\left[\Re\left\{\frac{Y_k}{n}\right\}\cos\left(\frac{2\pi kt}{P}\right) + \Im\left\{\frac{Y_k}{n}\right\}\sin\left(\frac{2\pi kt}{P}\right)\right] \ . \tag{3.24}$$

- Man kann nun sagen: die Funktion $f(t)$ besteht aus *cosinus*-Anteilen mit den Amplituden 'Realteil von Y_k/n' und aus *sinus*-Anteilen mit den Amplituden 'Imaginärteil von Y_k/n'.

Es stellt sich nun die Frage, ob die Interpolation Glg. (3.24) die bestmögliche ist, d.h., ob diese Funktion die Meßwerte wirklich so glatt wie möglich verbindet (vgl. Abschnitt 3.1).
Die Antwort ist nein, und die Begründung dafür finden Sie in der Abb. 3.9. Wegen der Periodizität der Interpolationsfunktion sind auch die Fourierkoeffizienten Y_k periodisch, und zwar mit der Periode n, d.h. es gilt:

$$Y_k = Y_{k+\mu n} \tag{3.25}$$

für jede beliebige ganze Zahl μ.

Man kann nun zeigen, daß *jede zusammenhängende Gruppe von n Fourierkoeffizienten mit den zugehörigen Frequenzen eine Interpolationsfunktion durch die gegebenen Punkte ergibt*, egal in welchem Bereich der Frequenzachse diese Gruppe liegt. Natürlich erhält man dabei jedesmal eine verschiedene Interpolationsfunktion!

Wenn man nun fragt, welche der möglichen Interpolationen diejenige sein wird, welche die geringste Neigung zur Oszillation zwischen den Stützpunkten hat, so ist die Antwort einfach: natürlich jene, welche die (dem Betrag nach) kleinsten Frequenzen enthält. Wenn man also n Koeffizienten benötigt,

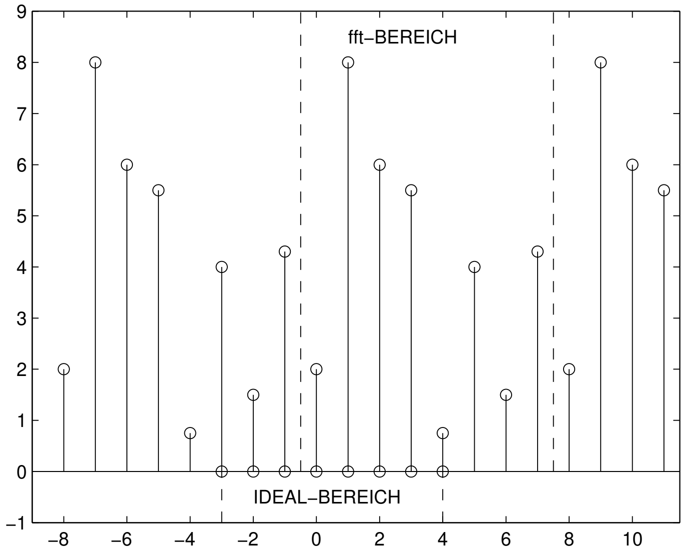


Abbildung 3.9: Periodizität der Fourierkoeffizienten im k=Raum für n=8. Es sind zwei Bereiche markiert: 'fft-BEREICH' bedeutet den Bereich der Fourierkoeffizienten, den die Fourier-Programme liefern (k=0...7), und 'IDEAL-BEREICH' bedeutet den Bereich k=–3,–2,–1,0,1,2,3,4, der die 'glattestmögliche' Interpolationsfunktion ergibt.

wird der ideale Frequenzbereich *um die Frequenz Null* liegen, also im Bereich $-n/2 < k \leq +n/2$.
Um diese ideale Interpolation formelmäßig darzustellen, bedarf es nur einer einfachen Umformung der Glg. (3.24):

$$\Re\{I(t)\} = \sum_{k=-(n/2)+1}^{-1} \left[\Re\left\{\frac{Y_k}{n}\right\} \cos\left(\frac{2\pi kt}{P}\right) + \Im\left\{\frac{Y_k}{n}\right\} \sin\left(\frac{2\pi kt}{P}\right)\right] + \\ + \sum_{k=0}^{n/2} \left[\Re\left\{\frac{Y_k}{n}\right\} \cos\left(\frac{2\pi kt}{P}\right) + \Im\left\{\frac{Y_k}{n}\right\} \sin\left(\frac{2\pi kt}{P}\right)\right] .$$

Führt man nun im ersten Term der obigen Gleichung die Index-Transformation

$$k' = n + k$$

durch, so ergibt sich weiter

$$\Re\{I(t)\} = \sum_{k'=(n/2)+1}^{n-1} \left[\Re\left\{\frac{Y_{k'-n}}{n}\right\} \cos\left(\frac{2\pi(k'-n)t}{P}\right) + \Im\left\{\frac{Y_{k'-n}}{n}\right\} \sin\left(\frac{2\pi(k'-n)t}{P}\right)\right] + \\ + \sum_{k=0}^{n/2} \left[\Re\left\{\frac{Y_k}{n}\right\} \cos\left(\frac{2\pi kt}{P}\right) + \Im\left\{\frac{Y_k}{n}\right\} \sin\left(\frac{2\pi kt}{P}\right)\right] .$$

Unter Berücksichtigung der Periodizität der Y_k (3.25) gilt

$$Y_{k'-n} = Y_{k'} ,$$

und man erhält (mit $k \to k'$) das Ergebnis

$$\Re\{I(t)\}_{\text{ideal}} = \sum_{k=0}^{n-1} \left[\Re\left\{\frac{Y_k}{n}\right\} \cos(2\pi\nu_k t) + \Im\left\{\frac{Y_k}{n}\right\} \sin(2\pi\nu_k t) \right] \tag{3.26}$$

mit

$$\nu_k = \frac{k}{P} \qquad \text{für} \quad k = 0, \ldots, \frac{n}{2} \tag{3.27}$$

und

$$\nu_k = \frac{k-n}{P} \qquad \text{für} \quad k = \frac{n}{2} + 1, \ldots, n-1 \tag{3.28}$$

Diese Darstellung hat den Vorteil, daß in ihr nur die Y_k-Werte vorkommen, die von *four1.c, ftt.f90 und fft_sor.m* geliefert werden, und zwar genau in der entsprechenden Reihenfolge!

Das heißt natürlich nicht, daß (3.26)–(3.28) die (unbekannte) Funktion $f(t)$ 'hinter den Stützpunkten' ideal repräsentieren. Das geht schon daraus hervor, daß nur diskrete Frequenzen mit einer Auflösung von

$$\Delta\nu = \frac{1}{P} \tag{3.29}$$

sowie innerhalb des Frequenz-Intervalls

$$-\left(\frac{n}{2P}\right) < \nu \leq \left(\frac{n}{2P}\right) \tag{3.30}$$

erfaßt werden. Die 'Grenzfrequenz' $\nu_{\text{Ny}} = n/(2P)$ wird in der Literatur 'Nyquist-Frequenz' genannt.

Man kann also, was die Qualität einer Frequenzanalyse betrifft, wie folgt zusammenfassen:

- große Periode P $\longrightarrow$ hohe Frequenz-Auflösung,
- große Punktanzahl n $\longrightarrow$ großer Frequenzbereich.

Aliasing effect

Wenn das zu analysierende Signal eine Frequenz-Komponente $\nu_0 = k_0/P$ enthält, welche außerhalb des Bereichs (3.30) liegt, kommt es zu einem störenden Effekt, der darin besteht, daß die entsprechende Amplitude mit dem Index k_0 *in den Nyquist-Bereich hineingespiegelt wird*, d.h. es wird eine Frequenz-Komponente

$$\nu_0' = \frac{n - k_0}{P} \tag{3.31}$$

angezeigt, die tatsächlich nicht existiert. Dieses *aliasing effect* wird in Abb. 3.10 schematisch dargestellt.

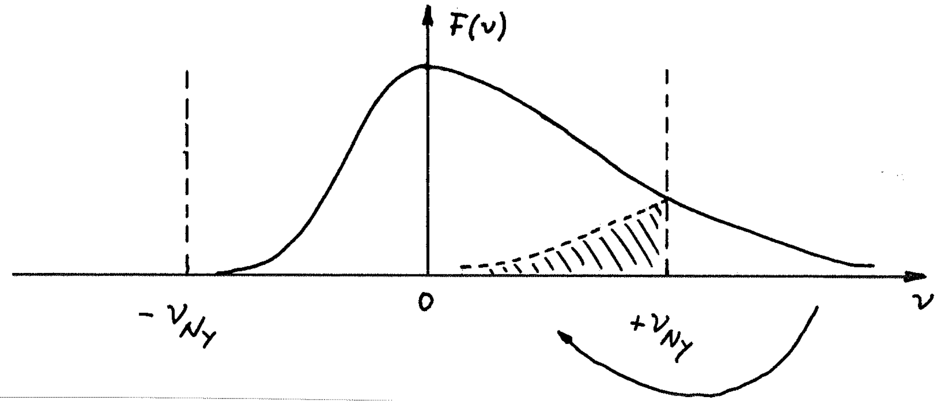


Abbildung 3.10: Der *aliasing effect* bei der Frequenzanalyse (schematisch).

Ein einfaches Beispiel zur Frequenz-Analyse

Im folgenden sollen die Zusammenhänge dieses Abschnittes an Hand eines Testbeispiels erläutert werden. Zu diesem Zweck definiert man eine analytische Funktion, die nur aus Cosinus- und Sinus-Termen besteht, und berechnet aus dieser Funktion eine Reihe von Werten, die man als Daten für eine Frequenz-Analyse mittels FT nimmt. Man erwartet, daß die FT-Analyse genau die Amplituden und Frequenzen der Testfunktion liefert.

Testfunktion:

$$f(t) = 2\cos(2\pi \cdot 2 \cdot t) - 3\sin(2\pi \cdot 4 \cdot t) - cos(2\pi \cdot 4 \cdot t) + 2\sin(2\pi \cdot 7 \cdot t)$$

Die Frequenzen sowie die Amplituden lauten also:

Frequenz (Hz)	cos-Ampl.	sin-Ampl.
2	+2	--
4	-1	-3
7	--	+2

Die Periode der Testfunktion ist P=1 s, und es wurden n=64 Stützpunkte berechnet. Für die Frequenz-Analyse bedeutet das eine Frequenzauflösung von $\Delta\nu$=1 Hz sowie einen Frequenzbereich von –31 bis +32 Hz [s. Glgen. (3.29) und (3.30)].

Wenn man dieses Problem mit Hilfe des Programms FOUR1 angeht (ohne die in diesem Abschnitt beschriebene Frequenz-Optimierung), erhält man das Ergebnis

Frequenz (Hz)	RT(Y)/n (cos)	IT(Y)/n (sin)
2.000	1.0000	0.0000
4.000	-0.5000	-1.5000
7.000	-0.0000	1.0000
57.000	0.0000	-1.0000
60.000	-0.5000	1.5000
62.000	1.0000	0.0000

Die ersten drei Frequenzen sind korrekt, die drei letzten Ergebnisse liegen weit über den Testvorgaben. Dennoch ist die entsprechende Sin-Cos-Kurve eine *Interpolationskurve* [s. Abb. 3.11 (a)], allerdings kommt es zwischen den Stützpunkten zu starken Oszillationen.

Führt man jedoch die Frequenz-Optimierung im Sinne der Glgen. (3.26)–(3.28) durch, ergibt sich das Ergebnis

Frequenz (Hz)	RT(Y)/n (cos)	IT(Y)/n (sin)
2.000	1.0000	0.0000
4.000	-0.5000	-1.5000
7.000	-0.0000	1.0000
-7.000	0.0000	-1.0000
-4.000	-0.5000	1.5000
-2.000	1.0000	0.0000

Dieses Resultat entspricht exakt der Testvorgabe, und zwar sowohl in bezug auf die Frequenzen als auch in bezug auf die Amplituden. So werden z.B. zwei Cosinusse angezeigt, jeweils mit der Amplitude 1.000, und mit den Frequenzen +2 und –2. Das bedeutet:

$$\cos(2\pi \cdot 2 \cdot t) + \cos(-2\pi \cdot 2 \cdot t) = 2\cos(2\pi \cdot 2 \cdot t)\,,$$

d.h. der Kosinus-Term mit der Frequenz 2 Hz besitzt die Gesamtamplitude 2, was exakt mit dem ersten Term der Testfunktion übereinstimmt.
Ebenso werden z.B. zwei Sinusse angezeigt, einer mit Frequenz +4 Hz und Amplitude –1.5, und ein zweiter mit Frequenz –4 Hz und der Amplitude +1.5. Dies ergibt

$$-1.5\sin(2\pi \cdot 4 \cdot t) + 1.5\sin(-2\pi \cdot 4 \cdot t) = -3\sin(2\pi \cdot 4 \cdot t)\,,$$

also exakt den zweiten Term der Testfunktion, usw.
Die entsprechende Interpolationsfunktion [s. Abb. 3.11 (b)] entspricht genau der Testkurve.

Zum Abschluß soll noch der *aliasing effect* demonstriert werden. Angenommen, die Testfunktion laute

$$f(t) = 2\cos(2\pi \cdot 2 \cdot t) - 3\sin(2\pi \cdot 4 \cdot t) - cos(2\pi \cdot 4 \cdot t) + 2\sin(2\pi \cdot 55 \cdot t)\,.$$

D. h., sie ist bis auf den vierten Term identisch, der eine Frequenz von 55 Hz hat. Wegen der Periode von 1 s und den n=64 Stützpunkten hat die Nyquist-Frequenz den Wert 32 Hz. Die Frequenz des vierten Terms liegt also *außerhalb* des Nyquist-Bereiches.
Das FT-Ergebnis inklusive *Frequenz-Optimierung* lautet in diesem Fall

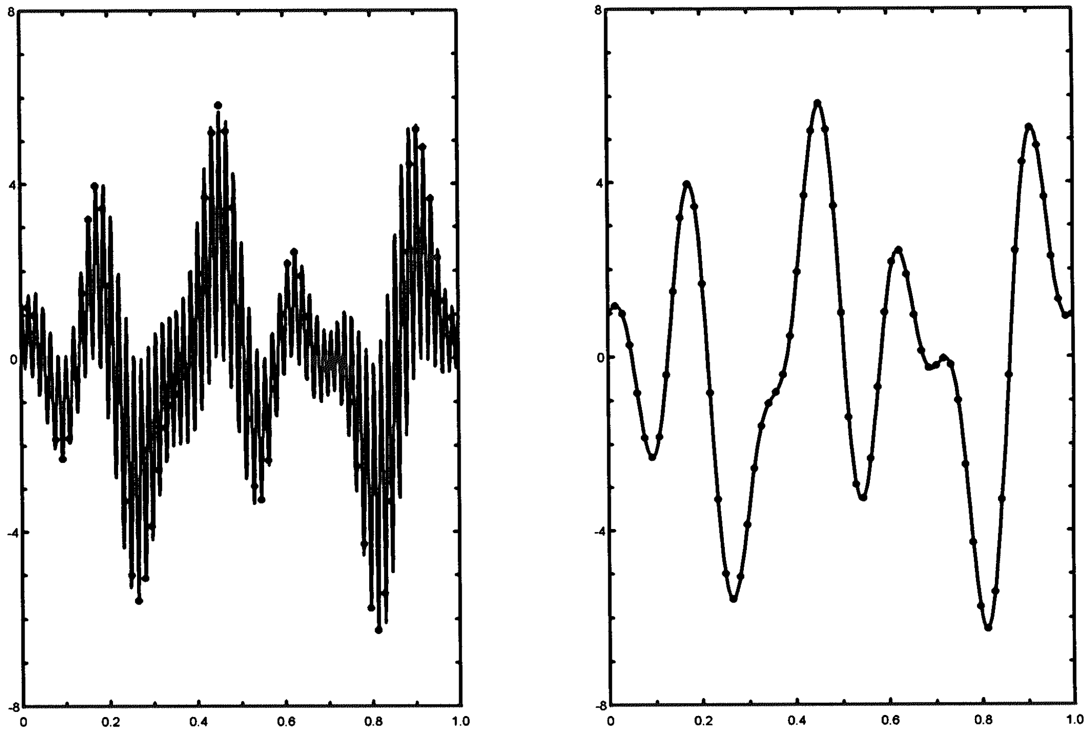


Abbildung 3.11: (a) FT-Frequenz-Analyse ohne Frequenz-Optimierung, (b) FT-Frequenz-Analyse mit Frequenz-Optimierung. Die Kreise sind die Stützpunkte.

Frequenz (Hz)	RT(Y)/n (cos)	IT(Y)/n (sin)
2.000	1.0000	0.0000
4.000	-0.5000	-1.5000
9.000	0.0000	-1.0000
-9.000	-0.0000	1.0000
-4.000	-0.5000	1.5000
-2.000	1.0000	0.0000

Wie Sie sehen, werden die ersten drei Terme der Testfunktion exakt erhalten, der letzte Term mit seinen 55 Hz scheint hingegen nicht auf. Statt dessen wird eine Frequenz von $\pm$ 9 Hz angezeigt, die in der Testfunktion nicht vorkommt. Ein typischer *aliasing effect* s. Glg. (3.31)] wegen

$$64 - 55 = 9\,.$$

Die Gesamtamplitude dieses *aliasing*-Terms ist, wie aus der obigen Tabelle hervorgeht, $-1 - (1) = -2$. In der entsprechenden Testfunktion hat dieser Term (bei 55 Hz) die Amplitude $+2$. Die Ursache dafür ist, daß beim *aliasing* eines Cosinus-Terms dieser unter Beibehaltung seines Vorzeichens in den Nyquist-Raum gespiegelt wird, während ein Sinus-Term dabei sein Vorzeichen ändert.

3.4.4 Weitere Stichworte zum Thema: Diskrete Fourier-Transformation

Wie auch in anderen Kapiteln dieses Skriptums muß ich mich auch bei der FT auf die wichtigsten Grundlagen beschränken. Auch hier möchte ich versuchen, Ihnen weitere – insbesondere für die Anwendung – wichtige Themen in meiner LV im SS *Ausgewählte Kapitel aus 'Numerische Methoden in der Physik'* zu vermitteln:

- Mehr zum Stichwort 'Faltung und Korrelation' (inkl. Anwendungsbeispiele).
- Fortgeschrittene Filtermethoden zur FT-Behandlung von statistisch fehlerhaften Daten.
- Grundlagen der Computer-Tomografie.

3.4.5 Software-Angebot

Da die Fourierentwicklung eine außerordentlich wichtige Methode darstellt, mathematische und physikalische Probleme zu behandeln, ist das Angebot an einschlägigen Programmen besonders groß. Ich kann daher im folgenden nur sehr allgemeine Informationen geben:

FORTRAN-User :

- F-Programme für FFT in den 'Numerical Recipes' [9] bzw. im Buch von Engeln-Müllges [6]. Zugang auf dem PCs des Computerraums Physik:

  ```
  /usr/local/numeric/numrec/f        bzw.
  /usr/local/numeric/num_alg/f
  ```

- Schauen sich sich im NETLIB das Programmpacket *FFTPACK* an.

C-User :

- Eine Reihe von C-Programmen zum Thema *Fast-Fourier-Transform* finden Sie in den 'Numerical Recipes C' [10] sowie in [6] bzw. auf den 'Physik-PCs' unter

  ```
  /usr/local/numeric/numrec/c       bzw.
  /usr/local/numeric/num_alg/c
  ```

- Ein interessantes C-Programm fuer FFT können Sie aus dem Internet herunterladen, nämlich das C-Programm *fftw.c*. Die Autoren M. Frigo und S. G. Johnson behaupten, daß es schneller sei als alle anderen Programme am Markt. Außerdem ist es nicht an die Bedingung $n = 2^m$ gebunden.
  Adresse: www.fftw.org

Mathematica: Fourier[$a_0, a_1, \ldots$] InverseFourier[$b_0, b_1, \ldots$]
für alle n, wobei aber *the efficiency significantly increases when n is a power of two.*

Ein Beispiel: *Smoothing of noised data by convolution with a kernel:*[6]

```
data=Table[N[Bessel][1,10 n/256] + 0.2 (Random[]-1/2)],{n,256}];
kern=Table[N[Exp[-200 (n/256)^2]],{n,256}];
conv=InverseFourier[Fourier[data]/Fourier[kern]];
ListPlot[Chop[conv]]
```

MATLAB: Das Programm *fft* habe ich bereits vorgestellt. Dazu gibt es noch ein entsprechendes Programm zur inversen FFT mit Namen *ifft*.

Vergessen Sie bitte nicht meinen Rat, anstelle dieser Routinen die von mir variierten Programme *fft_sor* bzw. *ifft_sor* zu verwenden.

Weitere Fourier-Transform-Programme von MATLAB:

fft2	zweidim. Fast-Fourier-Transform
ifft2	inverse zweidimensionale FFT
fftshift	Programm zur Frequenzverschiebung (vgl. Abschnitt 3.4.3).

[6]S. Wolfram, *Mathematica*, Addison-Wesley Publishing Comp., 1991, S. 681ff.

Kapitel 4

Least-Squares Approximation

4.1 Das Grundproblem.

So leistungsfähig die Methode der Interpolation in vielen Fällen auch sein mag, gibt es doch sehr häufig Approximationsprobleme, bei welchen die gegebene Punktmenge keine glatte Funktion repräsentiert. Dies trifft vor allem dann zu, wenn die Stützpunktkoordinaten *statistisch fehlerbehaftet* sind, weil sie z.B. Ergebnisse experimenteller Messungen darstellen.

Betrachtet man beispielsweise die Bestimmung des Widerstandswertes R eines metallischen Leiters mittels der Versuchsanordnung Abb.4.1, so erhält man ein *Spannungs-Strom-Diagramm* der Art Abb.4.2.

Der Zusammenhang zwischen Spannung und Strom ist offenbar ein linearer: es gilt das *Ohm'sche Gesetz*

$$U = R * I$$

mit R als dem *elektrischen Widerstand*. An der grundsätzlichen Linearität zwischen U und I ändern auch die statistischen Abweichungen bzw. offensichtliche Fehlmessungen nichts!

Die in einem solchen Fall anzuwendende Approximationsfunktion muß also eine Gerade (ein Polynom ersten Grades) sein. Es ist klarerweise völlig sinnlos, hier eine Interpolationsfunktion dazu zu zwingen, genau durch alle Meßpunkte hindurchzugehen. Es ist im Gegenteil eine *Glättung* ohne übermäßige Berücksichtigung von Fehlmessungen erwünscht.

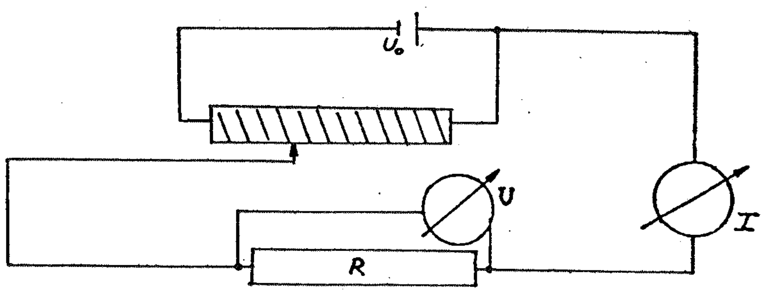


Abbildung 4.1: Versuchsanordnung: Bestimmung des elektrischen Widerstandes.

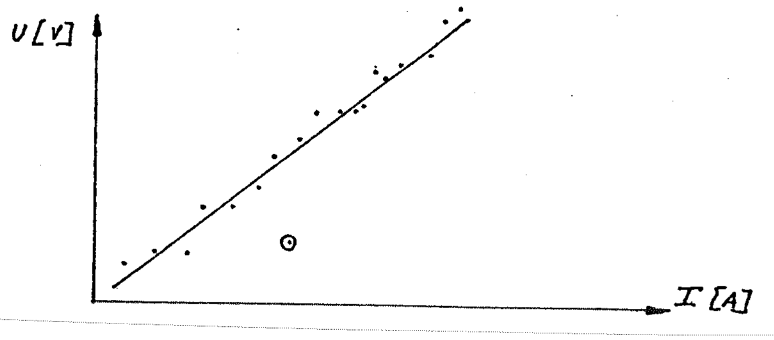


Abbildung 4.2: Ein Spannungs-Strom-Diagramm.

Abbildung 4.2: Ein Spannungs-Strom-Diagramm.

4.2 Mathematische Formulierung des Problems.

Es soll durch n gegebene Punkte $(x_i \mid y_i)$ *eine Kurve* $f(x)$ *gelegt werden, die den Punkten möglichst nahekommt, wobei jedoch die auftretenden Meßunsicherheiten ausgeglichen werden sollen. Zusätzlich soll es noch möglich sein, den Einfluß der einzelnen Meßpunkte auf* $f(x)$ *durch Gewichtsfaktoren zu verändern.*

Diese Forderungen lassen sich mathematisch folgendermaßen formulieren (Grundgleichung der Least-Squares (LSQ) Approximation):

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; \mathbf{a})\right]^2 \quad \rightarrow \quad \text{Minimum} \tag{4.1}$$

χ^2 ist die *gewichtete Fehlersumme*, $g_k > 0$ ist der *Gewichtsfaktor* des k-ten Punktes, und $f(x; \mathbf{a})$ ist die *Modellfunktion* mit den q *Modell- oder Fit-Parametern* $\mathbf{a} = a_1, a_2, \ldots, a_q$.

Die Summe der gewichteten Fehlerquadrate zwischen den gegebenen und den approximierten Funktionswerten soll ein Minimum werden. Die Bedeutung der Gewichtsfaktoren wird vor allem dann klar, wenn man sich für die statistische Auswertung von Punktmengen interessiert. Davon wird im nächsten Abschnitt dieses Kapitels die Rede sein.

Die Modellfunktion f dient in vielen Fällen nicht bloß dazu, 'eine schöne Kurve zu formen', sondern oft haben die Modellparameter auch eine konkrete *physikalische* Bedeutung.

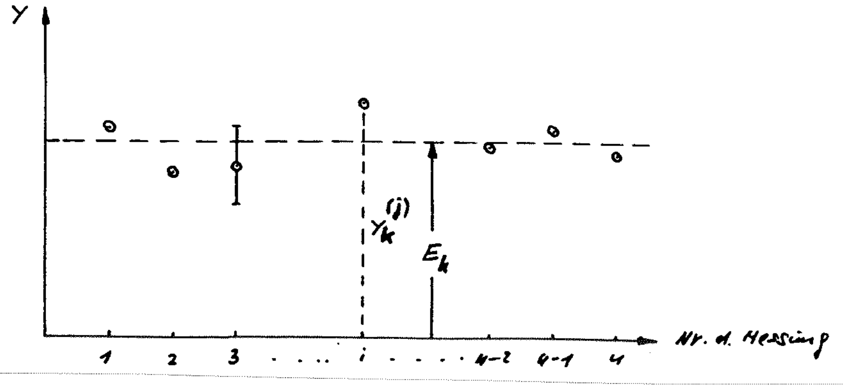


Abbildung 4.3: Erwartungswert von Einzelmessungen.

4.3 Die statistische Auswertung des Least-Squares Problems.

4.3.1 Grundbegriffe: Erwartungswert und Standardabweichung eines Meßwertes.

Die LSQ-Methode beruht auf dem Vergleich von (meist) experimentell ermittelten Meßwerten y_k mit den entsprechenden Modellwerten $f(x_k; \mathbf{a})$. Es ist evident,daß die Qualität der Ergebnisse (Fit-Parameter) sehr von der statistischen Qualität der y_k-Werte abhängen wird.

Angenommen, eine Messung wurde unter genau denselben Bedingungen n mal wiederholt. Als Meßwerte für $x = x_k$ wurden dabei die (i.a.) *verschiedenen* Werte $y_k^{(1)}, y_k^{(2)}, \ldots, y_k^{(n)}$ erhalten. Trägt man diese Werte in ein Diagramm ein (vgl. Abb.4.3), so erhält man eine Punktmenge, die um den *Erwartungswert* E_k oder *Mittelwert* der Messung verteilt ist:

$$E_k = \frac{1}{n} \sum_{j=1}^{n} y_k^{(j)} \tag{4.2}$$

Ein Maß für die *Streuung* der einzelnen Meßwerte um den Erwartungswert ist die *Standardabweichung*

$$\sigma_k = \left[\frac{\sum_{j=1}^{n} (y_k^{(j)} - E_k)^2}{n-1} \right]^{1/2} \tag{4.3}$$

In sehr vielen praktischen Fällen ist die *Wahrscheinlichkeit* P, mit der eine bestimmte Abweichung des Meßwertes vom Erwartungswert E auftritt, durch die *Wahrscheinlichkeitskurve*

$$P(y - E) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp^{-(y-E)^2/(2\sigma^2)} \tag{4.4}$$

gegeben. Dies ist die sehr häufig vorkommende *Gauss'sche Normalverteilung* (Abb.4.4). Erwähnenswert ist in diesem Zusammenhang auch, daß $P(y - E)$ genau an den Stellen $y - E = \pm\sigma$ Wendepunkte hat.

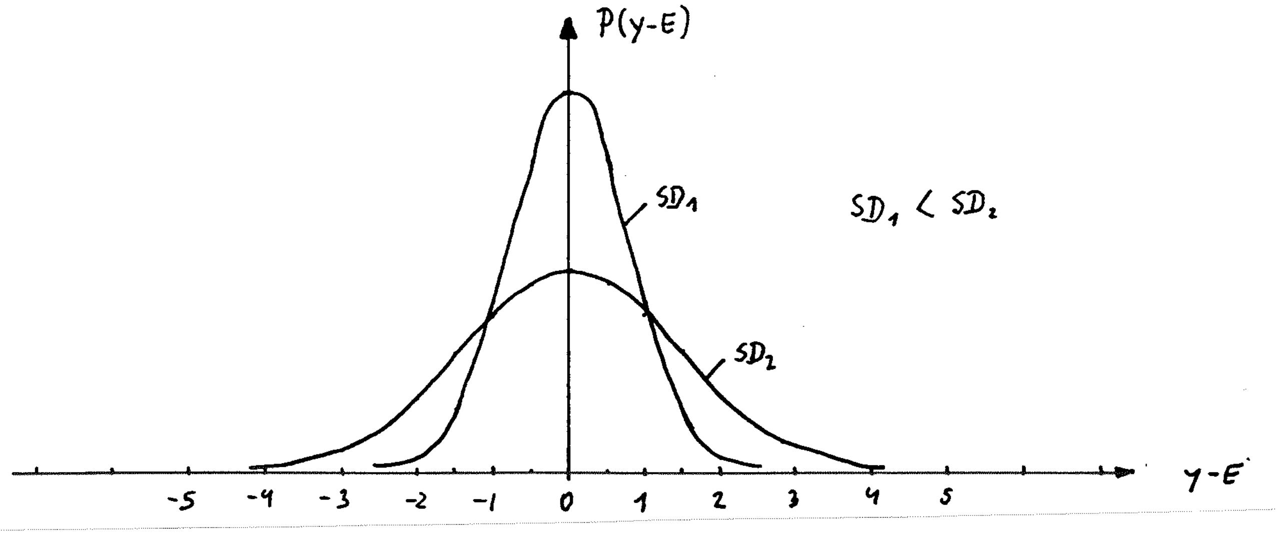


Abbildung 4.4: Die Gauss'sche Normalverteilung.

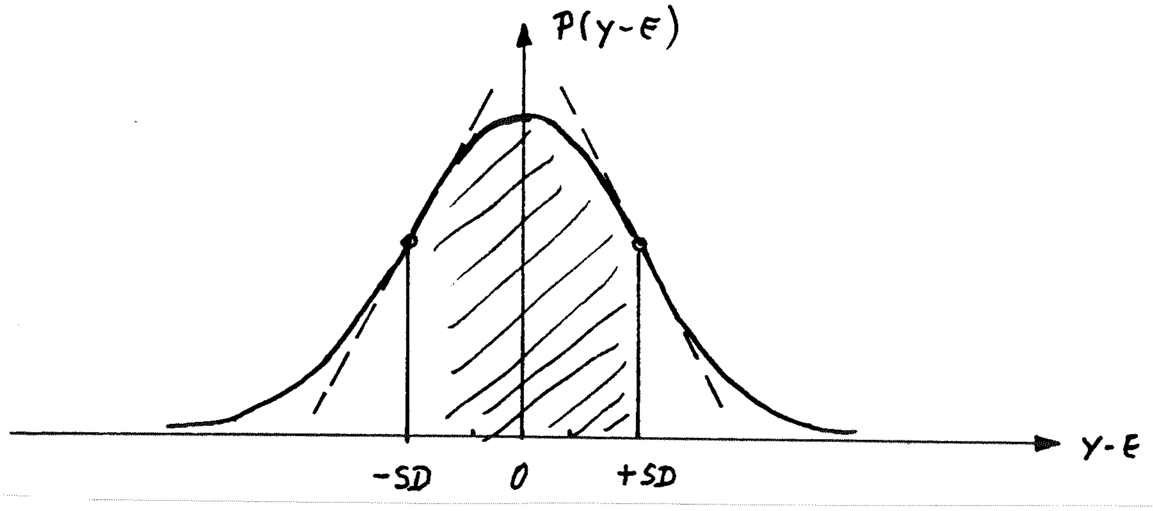


Abbildung 4.5: Zur geometrischen Bedeutung der Standardabweichung σ.

Der schraffierte Bereich in Abb.4.5 bedeutet offenbar jene Wahrscheinlichkeit, mit der ein Meßwert im Intervall $[E-\sigma, E+\sigma]$ liegt. Diese Wahrscheinlichkeit beträgt

$$\int_{-\sigma}^{+\sigma} d(y-E)\frac{1}{\sqrt{2\pi}\sigma}\exp^{(y-E)^2/(2\sigma^2)} = 0.685 \quad .$$

Bei Vorliegen einer Normalverteilung liegen also ca. 69 Prozent ($\approx 2/3$) aller Einzelwerte im Intervall $[E-\sigma, E+\sigma]$.

Normalverteilungen sind typisch für Meßvorgänge, bei denen die Meßgröße - zumindest innerhalb eines gewissen Intervalls - jeden beliebigen Wert annehmen kann (*analoge* Meßvorgänge). Die für *digitale* Meßvorgänge typische *Poisson-Verteilung* wird im nächsten Abschnitt diskutiert.

4.3.2 Berücksichtigung statistischer Größen im LSQ-Prozess.

Da diese LV. keine Statistik-Vorlesung ist, müssen die folgenden Überlegungen zwangsläufig ohne die an und für sich notwendigen Beweisführungen präsentiert werden.

- Die statistischen Unsicherheiten der y_k-Werte werden im LSQ-Prozess dadurch berücksichtigt, daß als Gewichtsfaktoren in (4.1) *die reziproken Quadrate der entsprechenden Standardabweichungen* genommen werden:

$$g_k \equiv \frac{1}{\sigma_k^2} \quad . \tag{4.5}$$

- Der LSQ-Prozess liefert die statistischen Erwartungswerte der Fit-Parameter. Die entsprechenden Standardabweichungen erhält man auf die folgende Weise:

  Man berechnet zuerst die sogenannte *Normalmatrix* N mit den Koeffizienten

$$[N]_{ij} = \frac{1}{2}\frac{\partial^2\chi^2}{\partial a_i \partial a_j} \quad . \tag{4.6}$$

  Durch Invertierung der Normalmatrix erhält man die *Kovarianzmatrix* C:

$$C = N^{-1} \tag{4.7}$$

  Die Kovarianzmatrix enthält zwei wichtige Informationen: einerseits stellen ihre Diagonalelemente die Quadrate der Standardabweichungen der Fit-Parameter dar, also

$$\sigma_{a_i} = \sqrt{c_{ii}} \quad , \tag{4.8}$$

  andererseits ergeben sich aus C die *Korrelationskoeffizienten* r_{ij} zwischen den Fit-Parametern:

$$r_{ij} = \frac{c_{ij}}{\sqrt{c_{ii}c_{jj}}} \quad . \tag{4.9}$$

Die r_{ij}-Werte, die stets im Intervall $[-1, +1]$ liegen, geben eine Information darüber, wie stark der i-te und der j-te Fit-Parameter einander beeinflussen.

- Berechnung der *Varianz* V bzw. deren Standardabweichung:

$$V = \frac{\chi^2}{n-q} \quad \text{und} \quad \sigma_V = \sqrt{\frac{2}{n-q}} \quad . \tag{4.10}$$

  V ist im Falle eines *idealen Modells* und für $n >> 1$ approximativ normalverteilt mit dem Mittelwert 1 und der Standardabweichung σ_V. Die Größe $n-q$ (Anzahl der Stützpunkte minus Anzahl der Fit-Parameter) wird als die *Anzahl der Freiheitsgrade* bezeichnet.

  Die Varianz eines Fits kann als Indikator für die Güte der verwendeten Modellfunktion dienen: wenn V signifikant außerhalb des Intervalles $[1-\sigma_V, 1+\sigma_V]$ liegt, paßt das Modell nicht gut zur gegebenen Punktmenge.

4.3.3 Die Bestimmung der Standardabweichungen der Einzelwerte.

Die statistische Auswertung eines LSQ-Problems setzt voraus, daß man die Standardabweichungen σ_k der Einzelwerte kennt. Woher bekommt man diese σ_k-Werte?

Für die Praxis sind vor allem die folgenden beiden Fälle von Bedeutung:

- Die Meßwerte sind um ihre Erwartungswerte normalverteilt:

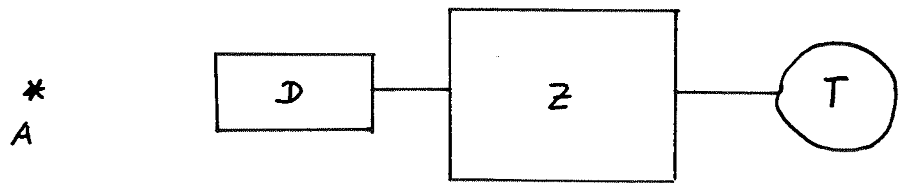
  Wie bereits erläutert, kommt diese Situation in der experimentellen Praxis sehr häufig vor. In diesem Fall gilt meistens wenigstens approximativ, daß die σ's *aller Meßwerte gleich sind*, daß also

$$\sigma_1 \approx \sigma_2 \approx \cdots \approx \sigma_n$$

  gilt. Diese repräsentative Standardabweichung kann dadurch bestimmt werden, daß man für einen ausgewählten x-Wert die Messung mehrmals wiederholt und das entsprechende σ mittels (4.3) berechnet.

- Wie bereits kurz erwähnt, gibt es aber noch eine andere wichtige Meßwert-Statistik, die bei allen Experimenten auftritt, die auf einer *Zählung von Ereignissen* beruhen (Digital-Messungen). In diesen Fällen sind die Einzelwerte um die jeweiligen Erwartungswerte *Poissonverteilt.*

Zählexperimente spielen z.B. in der Kernphysik eine dominierende Rolle. Die Versuchsanordnungen entsprechen dabei dem Schema



Dabei könnte A ein radioaktives Präparat sein, D ein Strahlendetektor, Z ein Digitalzähler und T eine Uhr.

Ohne auf die Theorie der Poisson-Statistik näher einzugehen, soll hier nur die wichtigste Eigenschaft dieser Statistik erwähnt werden:

Bei einer Poisson-Verteilung ist für jeden Meßwert sein Erwartungswert E mit der entsprechenden Standardabweichung σ über die einfache Beziehung

$$\sigma = \sqrt{E}$$

verknüpft. Da man aber den Erwartungswert i.a. nicht kennt, nimmt man näherungsweise stattdessen den Meßwert selbst und erhält

$$\sigma \approx \sqrt{y} \quad .$$

In diesem Fall haben also die Gewichtsfaktoren beim LSQ-Verfahren die Form

$$g_k \approx \frac{1}{y_k} \quad . \tag{4.11}$$

4.4 Modellfunktionen mit linearen Parametern.

Unter einer *Modellfunktion mit linearen Parametern* versteht man eine Form

$$f(x; \mathbf{a}) = \sum_{j=1}^{m} a_j \cdot \varphi_j(x) \tag{4.12}$$

mit den beliebigen (linear unabhängigen) *Basisfunktionen* $\varphi_j(x)$. Die oft verwendete saloppe Bezeichung *lineare Modellfunktionen* ist mißverständlich: $f(x; \mathbf{a})$ muß nämlich keineswegs linear in x sein, sondern linear in den Fit-Parametern. Die vielzitierte *lineare Regression* hat als Modellfunktion

$$f(x; a_1, a_2) = a_1 + a_2\, x$$

eine Gerade. Es handelt sich dabei um eine sehr einfache Sonderform eines Modells mit linearen Parametern.

Setzt man ein Modell (4.12) in die LSQ-Grundgleichung (4.1) ein, so erhält man

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - \sum_{j=1}^{m} a_j \varphi_j(x_k) \right]^2 \quad \rightarrow \quad \text{Minimum!}$$

Die gewünschte Minimalisierung der Fehlersumme wird durch *Nullsetzen der partiellen Ableitungen von χ^2 nach den Modellparametern* **a** erreicht:

$$\frac{\partial \chi^2}{\partial a_i} = \sum_{k=1}^{n} g_k \varphi_i(x_k) \left[y_k - \sum_{j=1}^{m} a_j \varphi_j(x_k) \right] = 0 \quad i = 1, \ldots, m \quad .$$

Dies führt zu einem linearen, inhomogenen Gleichungssystem m-ten Grades für die a_i:

$$\sum_{j=1}^{m} a_j \sum_{k=1}^{n} g_k \varphi_i(x_k) \varphi_j(x_k) = \sum_{k=1}^{n} g_k y_k \varphi_i(x_k)$$

für alle $i = 1, \ldots, m$, also zu einem Problem

$$A \cdot \mathbf{a} = \beta$$

mit der symmetrischen, positiv-definiten Koeffizientenmatrix

$$A \equiv [\alpha_{ij}] \quad \text{mit} \quad \alpha_{ij} = \sum_{k=1}^{n} g_k \varphi_i(x_k) \varphi_j(x_k) \tag{4.13}$$

und dem inhomogenen Vektor

$$\beta_i = \sum_{k=1}^{n} g_k y_k \varphi_i(x_k) \quad . \tag{4.14}$$

Für lineare Modellfunktionen (4.12) ist die Aufstellung der Normalmatrix gemäß (4.6) sehr einfach; es gilt nämlich:

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} = \sum_{k=1}^{n} g_k \varphi_i(x_k) \varphi_j(x_k) = \alpha_{ij} \quad . \tag{4.15}$$

4.4.1 Standardabweichungen der Fitparameter

Im Abschnitt 4.3.2 wurde (ohne Beweisführung) angegeben, daß sich die Standardabweichungen der Fitparameter aus den Diagonalelementen der Kovarianzmatrix ergeben [Glg. (4.8)]

Dieser Sachverhalt soll im folgenden an Hand eines sehr einfachen Modells demonstriert werden, nämlich mittels einer *linearen Regression* mit der Modellfunktion

$$f(x; a, b) = a + b\,x$$

Least-Squares-Formel:

$$\chi^2 = \sum_{k=1}^{n} g_k \, [y_k - a - b\,x_k]^2 \quad \rightarrow \quad \text{Minimum für} \quad a = a_{opt}, \quad b = b_{opt} \, .$$

Die Normalmatrix des Problems lautet nach Skriptum (4.6)

$$N = \begin{pmatrix} \alpha_{11} & \alpha_{12} \\ \alpha_{12} & \alpha_{22} \end{pmatrix}$$

mit

$$\alpha_{11} = \sum_{k=1}^{n} g_k \qquad \alpha_{12} = \sum_{k=1}^{n} g_k x_k \qquad \alpha_{22} = \sum_{k=1}^{n} g_k x_k^2$$

Die optimierten Parameter ergeben sich als Lösung des inhomogenen, linearen Gleichungssystems

$$N \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}$$

mit

$$\beta_1 = \sum_{k=1}^{n} g_k y_k \qquad \beta_2 = \sum_{k=1}^{n} g_k x_k y_k \quad .$$

Die Lösung dieses Gleichungssystems lautet

$$a = a_{opt} = \frac{\beta_1 \alpha_{22} - \beta_2 \alpha_{12}}{D} \qquad b = b_{opt} = \frac{\beta_2 \alpha_{11} - \beta_1 \alpha_{12}}{D} \tag{4.16}$$

mit D als der Determinanten der Normalmatrix

$$D = \alpha_{11}\alpha_{22} - \alpha_{12}^2 \quad .$$

Die *Kovarianzmatrix* C ist die Inverse der Normalmatrix, also

$$C = N^{-1} = \frac{1}{D} \begin{pmatrix} \alpha_{22} & -\alpha_{12} \\ -\alpha_{12} & \alpha_{11} \end{pmatrix}$$

Daraus erhält man unmittelbar die *Standardabweichungen der Fitparameter* als

$$\sigma_a^2 = \frac{\alpha_{22}}{D} \qquad \sigma_b^2 = \frac{\alpha_{11}}{D} \,. \tag{4.17}$$

Die Berechnung dieser Standardabweichungen kann aber auch auf eine andere Art geschehen, nämlich auf der Basis des Fehlerfortpflanzungsgesetzes (FFG).

Es ist klar, daß die optimierten Parameter a und b Funktionen aller $x-$ und $y-$Komponenten der gegebenen Punkte sind, also

$$a = a(x_1, \ldots, x_n; y_1, \ldots, y_n) \qquad b = b(x_1, \ldots, x_n; y_1, \ldots, y_n)$$

Nach dem FFG gilt nun für die Standardabweichungen von a und b

$$\sigma_a^2 = \sum_{l=1}^{n} \left[\left(\frac{\partial a}{\partial x_l} \right)^2 \sigma(x_l)^2 + \left(\frac{\partial a}{\partial y_l} \right)^2 \sigma(y_l)^2 \right]$$

und

$$\sigma_b^2 = \sum_{l=1}^{n} \left[\left(\frac{\partial b}{\partial x_l} \right)^2 \sigma(x_l)^2 + \left(\frac{\partial b}{\partial y_l} \right)^2 \sigma(y_l)^2 \right]$$

Dabei sind die $\sigma(x_l)$ und $\sigma(y_l)$ die Standardabweichungen (Fehler) der entsprechenden $x-$ bzw $y-$Komponenten. Nimmt man die $x-$Komponenten als exakt an, gilt $\sigma(x_l) = 0$, und für die Fehler der $y-$Komponenten gilt nach Skriptum (4.5)

$$g_l = \frac{1}{\sigma(y_l)^2}$$

Es ergibt sich somit

$$\sigma_a^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left(\frac{\partial a}{\partial y_l} \right)^2 \qquad \text{und} \qquad \sigma_b^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left(\frac{\partial b}{\partial y_l} \right)^2$$

Die partiellen Ableitungen in diesen Gleichungen können nun aus (4.16) berechnet werden, und es ergibt sich

$$\sigma_a^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left[\frac{1}{D} \left(\alpha_{22} g_l - \alpha_{12} g_l x_l \right) \right]^2$$

und

$$\sigma_b^2 = \sum_{l=1}^{n} \frac{1}{g_l} \left[\frac{1}{D} \left(-\alpha_{12} g_l + \alpha_{11} g_l x_l \right) \right]^2$$

Die nun folgenden Umformungen sind elementar:

$$\sigma_a^2 = \frac{1}{D^2} \sum_{l=1}^{n} g_l \left[\alpha_{22} - \alpha_{12} x_l \right]^2$$

$$= \frac{1}{D^2} \sum_{l=1}^{n} \sum_{k=1}^{n} \sum_{k'=1}^{n} g_l g_k g_{k'} \left[x_k^2 x_{k'}^2 - 2 x_l x_k^2 x_{k'} + x_l^2 x_k x_{k'} \right]$$

Durch geeignete Umbenennung der Summenindizes $l; k; k'$ ergibt sich daraus

$$\sigma_a^2 = \frac{1}{D^2} \sum_{l,k,k'} g_l g_k g_{k'} \, x_k^2 \left[x_{k'}^2 - x_{k'} x_l \right] = \frac{1}{D^2} \sum_{k=1}^{n} g_k x_k^2 \left[\sum_{l=1}^{n} g_l \sum_{k'=1}^{n} g_{k'} x_{k'}^2 - \left(\sum_{l=1}^{n} g_l x_l \right)^2 \right]$$

Der Ausdruck in der eckigen Klammer in der letzten Zeile ist aber offenbar die Determinante D der Normalmatrix, und es ergibt sich schlußendlich

$$\sigma_a^2 = \frac{1}{D^2} \, D \sum_{k=1}^{n} g_k x_k^2 = \frac{\alpha_{22}}{D} \, ,$$

also dasselbe Ergebnis wie aus der Kovarianzmatrix (4.17), was zu beweisen war.

Eine ganz ähnliche Rechnung kann auch für σ_b durchgeführt werden.

4.4.2 Das Programm LFIT.

Quelle: [9], S.513ff, [10], S. 674ff mit Änderungen.

INPUT-Parameter:

X( ), Y( ): Koordinaten der Stützpunkte.

SIG( ): Standardabweichungen der y-Werte.

NDATA: Anzahl der Stützpunkte.

MA: Anzahl der Terme der linearen Modellfunktion.

OUTPUT-Parameter:

A( ): Feld mit den optimierten Fit-Parametern.

YF( ): Feld mit den Funktionswerten der Ausgleichskurve an den gegebenen x-Werten.

COVAR(,): Kovarianzmatrix.

CHISQ: die gewichtete Fehlersumme χ^2.

von LFIT benötigte Prozeduren:

FUNCS: In dieser Prozedur werden die in der Modellfunktion verwendeten Basisfunktionen definiert (s. Glg. (4.12)):

```
void funcs(double x, double afunc[], int ma)
{  ..... Definition lokaler Variabler .....
   afunc[1]=   .....  phi_{1}(x);
     .
     .
   afunc[ma] = .....  phi_{m}(x);
}
```

LUDCMP bzw. LUBKSB: Prozeduren zur Lösung des linearen Gleichungssystems [Glg.en (4.13),(4.14)] bzw. zur Invertierung der Normalmatrix, s. Kap. 2.

Programmstruktur:

1. Berechnung der Matrixkoeffizienten α_{ij} bzw. der Komponenten β_i des inhomogenen Vektors gemäß (4.13) und (4.14). Abspeicherung dieser Größen auf NORMAL(,) und BETA( ). Gemäß Glg.(4.15) stellt NORMAL gleichzeitig die Normalmatrix des Problems dar.

2. Berechnung der optimierten Fit-Parameter sowie der Kovarianzmatrix mittels der Prozeduren LUDCMP und LUBKSB.

3. Berechnung der Funktionswerte der Ausgleichskurve für die gegebenen x-Koordinaten und Berechnung der minimalen Fehlerquadratsumme.

Struktogramm 11 — LFIT(X,Y,SIG,NDATA,MA,A,YF,COVAR,CHISQ)

```
I=1(1)MA
    J=1(1)MA
        NORMAL(I,J):=0.0
    BETA(I):=0.0
K=1(1)NDATA
    FUNCS(X(K),AFUNC,MA)
    G:=1.0/SIG(K)/SIG(K)
    I=1(1)MA
        J=1(1)I
            NORMAL(I,J):=NORMAL(I,J) + G*AFUNC(I)*AFUNC(J)
        BETA(I):=BETA(I) + G*Y(K)*AFUNC(I)
I=2(1)MA
    J=1(1)I-1
        NORMAL(J,I):=NORMAL(I,J)
LUDCMP(NORMAL,MA,INDX,D,KHAD)
LUBKSB(NORMAL,MA,INDX,BETA,LOES)
J=1(1)MA
    A(J):=LOES(J)
CHISQ:=0.0
K=1(1)NDATA
    FUNCS(X(K),AFUNC,MA)
    G:=1.0/SIG(K)/SIG(K)
    SUM:=0.0
    J=1(1)MA
        SUM:=SUM + A(J)*AFUNC(J)
    YF(K):=SUM
    CHISQ:=CHISQ + G*(Y(K)-SUM)*(Y(K)-SUM)
J=1(1)MA
    I=1(1)MA
        BETA(I):=0.0
    BETA(J):=1.0
    LUBKSB(NORMAL,MA,INDX,BETA,LOES)
    I=1(1)MA
        COVAR(I,J):=LOES(I)
(return)
```

Anmerkungen zu LFIT:

1. Das Programm verlangt die Eingabe der Standardabweichungen σ_k der einzelnen y-Werte (das Feld SIG). Dazu gibt es, wie bereits im Abschnitt 4.3.3 dieses Kapitels erläutert, mehrere Möglichkeiten:

   - Stastistik der Stützpunkte unbekannt $\rightarrow$ alle $\sigma_k = 1$.

     In diesem Fall ist natürlich keine Aussage über die Varianz des Fits bzw. über die Statistik der optimierten Modellparameter möglich.

   - Statistik der normalverteilten Stützpunkte wird beschrieben durch eine typische Standardabweichung σ mit $\sigma_k \approx$ alle σ_k.

   - Die Stützpunkte sind Poisson-verteilt $\rightarrow \sigma_k \approx \sqrt{y}_k$.

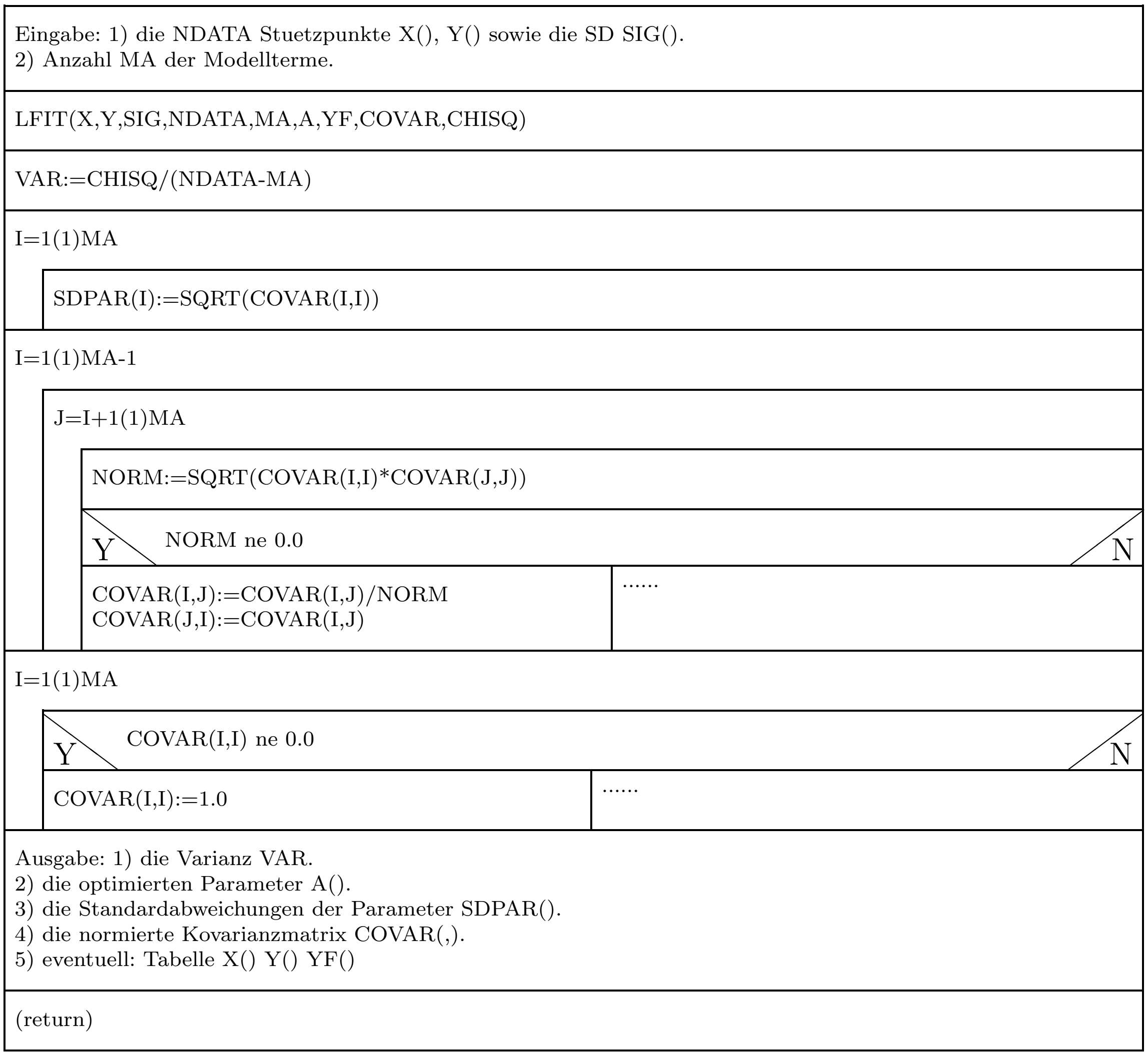
2. Das Programm liefert neben den optimierten Fit-Parametern auch noch die gewichtete Fehlersumme χ^2 und die Kovarianzmatrix. Daraus können bei Bedarf die Varianz des Fits sowie die Standardabweichungen bzw. die Korrelationskoeffizienten der gefitteten Parameter berechnet werden.

3. Es soll hier auch erwähnt werden, daß viele angebotene LSQ-Programme (z. B. die in den *Numerical Recipes* [9] und [10]) eine in der Praxis sehr brauchbare Option enthalten: es kann beim Aufruf des LSQ-Programms entschieden werden, welche der insgesamt MA Parameter der Modellfunktion tatsächlich optimiert (gefitted) werden sollen (Fit-Parameter), und welche während des gesamten LSQ-Prozesses unverändert bleiben sollen (fixe Parameter).

4.4.3 Anwendung von LFIT.

Die Anwendung der Prozedur LFIT kann z.B. im Rahmen des folgenden Hauptprogrammes erfolgen:

Programmschema — :



Ein Testbeispiel.

Für diesen Test wurden 101 Stützpunkte berechnet, die um ihre jeweiligen Erwartungswerte normalverteilt sind, wobei ein konstantes σ für alle Punkte angenommen wurde.

Die Erwartungswerte aller Punkte liegen exakt auf der Polynomkurve dritten Grades

$$y(x) = 0.5 - x - 0.2x^2 + 0.1x^3 \quad .$$

Die ideale Modellfunktion für dieses Problem ist also ein Polynom dritten Grades mit den Parametern $a_1, \ldots, a_4$:

$$f(x; \mathbf{a}) = \sum_{j=1}^{4} a_j x^{j-1}$$

Dementsprechend lautet die von LFIT aufgerufene Prozedur FUNCS z.B. in C wie folgt:

```
void funcs(double x, double afunc[], int ma)
{ int i;
  afunc[1]=1.0;
  for(i=2;i<=ma;i++) afunc[i]=afunc[i-1]*x;
}
```

Für den <u>ersten Datensatz</u> wurde eine Normalverteilung mit $\sigma = 0.1$ simuliert.

<u>Interpretation der Tabelle 4.1 (a):</u>

Je nach der Anzahl der Modellterme beträgt die Anzahl der Freiheitsgrade zwischen 96 und 100. Dementsprechend sollte sich für ein gutes Modell gemäß (4.10) eine Varianz zwischen ca. 0.86 und ca. 1.14 ergeben.

Wegen der relativ großen Streuung der Stützpunkte [vgl. Abb.4.6 (a)] ist eine Unterscheidung darüber, ob das ideale Modell ein Polynom 2. oder 3. Grades ist, <u>nicht</u> möglich! Diese Unsicherheit wirkt sich auch darin aus, daß die σ's der gefitteten Parameter ab MA=4 oft größer sind als die Beträge der entsprechenden Parameter-Mittelwerte, die auch für das an und für sich richtige Polynom 3. Grades sehr schlecht zu den simulierten Parametern a_j^s passen:

$$\begin{array}{ll} a_1^s = 0.5 & a_1^{Fit} = 0.5097 \\ a_2^s = -1.0 & a_2^{Fit} = -1.1152 \\ a_3^s = -0.2 & a_3^{Fit} = -0.0517 \\ a_4^s = 0.1 & a_4^{Fit} = 0.0543 \end{array}$$

Die Situation wird natürlich viel besser, wenn die Stützpunkte eine bessere Statistik haben. Für den <u>zweiten</u> Datensatz wurde angenommen: $\sigma = 0.025$.

<u>Interpretation der Tabelle 4.1 (b):</u>

Hier ist die Bestimmung des idealen Modells sehr viel leichter. Die Polynome nullten, ersten und zweiten Grades haben eine zu hohe Varianz, und erst ab MA$\leq$4 sind die Modellfunktionen als gut zu bezeichnen. Da aber für MA>4 einige Fit-Parameter 'in ihren σ's untergehen', ist das Polynom dritten Grades die beste Modellfunktion [s. Abb.4.7 (b)].

Tabelle 4.1 (a):

MA	Varianz	optimierte Parameter	Anmerkungen
1	37.034	$a_1 = -0.5652 \pm 0.0100$	schlechtes Modell
2	1.138	$a_1 = 0.4574 \pm 0.0198$	an der Grenze
		$a_2 = -1.0226 \pm 0.0171$	
3	1.032	$a_1 = 0.5307 \pm 0.0293$	
		$a_2 = -1.2449 \pm 0.0676$	gutes Modell
		$a_3 = 0.1111 \pm 0.0327$	
4	1.035	$a_1 = 0.5097 \pm 0.0384$	
		$a_2 = -1.1152 \pm 0.1670$	gutes Modell,
		$a_3 = -0.0517 \pm 0.1945$	schlechte Statistik
		$a_4 = 0.0543 \pm 0.0639$	s. Abb.4.6
5	1.046	$a_1 = 0.5134 \pm 0.0469$	
		$a_2 = -1.1543 \pm 0.3284$	gutes Modell
		$a_3 = 0.0372 \pm 0.6726$	schlechte Statistik
		$a_4 = -0.0151 \pm 0.5065$	
		$a_5 = 0.0174 \pm 0.1256$	

Tabelle 4.1 (b):

MA	Varianz	optimierte Parameter	Anmerkungen
1	598.559	$a_1 = -0.5639 \pm 0.0025$	schlechtes Modell
2	2.883	$a_1 = 0.4773 \pm 0.0049$	schlechtes Modell
		$a_2 = -1.0413 \pm 0.0043$	
3	1.204	$a_1 = 0.5472 \pm 0.0073$	
		$a_2 = -1.2530 \pm 0.0169$	schlechtes Modell
		$a_3 = 0.1059 \pm 0.0082$	
4	0.899	$a_1 = 0.5128 \pm 0.0096$	gutes Modell
		$a_2 = -1.0413 \pm 0.0417$	Fit-Parameter haben
		$a_3 = -0.1600 \pm 0.0486$	günstige σ's.
		$a_4 = 0.0886 \pm 0.0160$	siehe Abb.4.6
5	0.908	$a_1 = 0.5141 \pm 0.0117$	gutes Modell,
		$a_2 = -1.0548 \pm 0.0821$	aber manche Fit-Parameter
		$a_3 = -0.1294 \pm 0.1681$	haben sehr schlechte Statistik.
		$a_4 = 0.0647 \pm 0.1266$	'mixing of parameters'
		$a_5 = 0.0060 \pm 0.0314$	

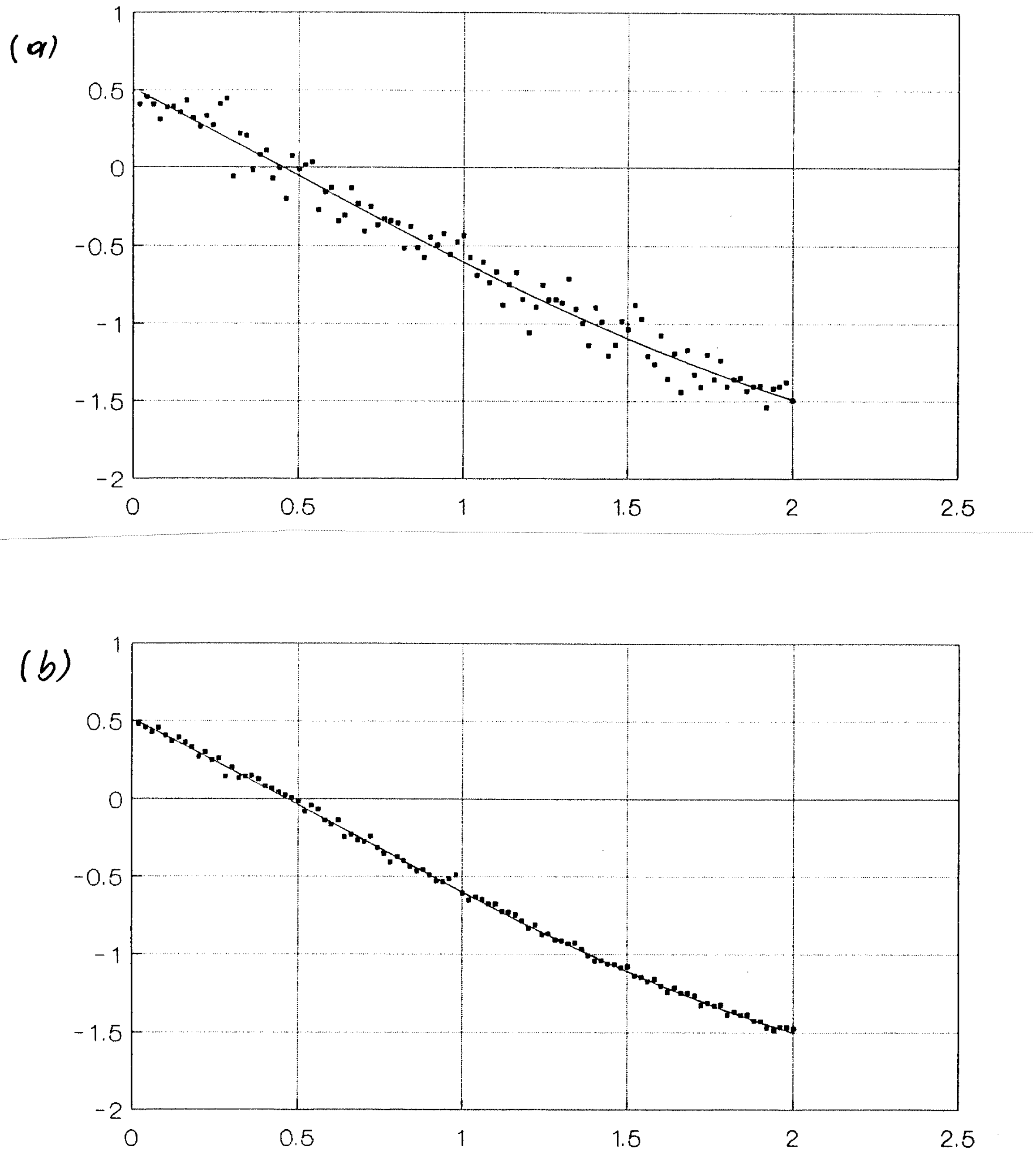


Abbildung 4.6: Lineare LSQ-Auswertung simulierter Datenwerte: (a) $\sigma = 0.1$, (b) $\sigma = 0.025$.

4.5 Modellfunktionen mit nicht-linearen Parametern.

4.5.1 Was sind nicht-lineare Parameter?

Setzt man die Modellfunktion

$$f(x; a, b) = a \cdot e^{-bx}$$

in die LSQ-Grundgleichung (4.1) ein, so erhält man

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - a \cdot e^{-bx_k} \right]^2 \quad \rightarrow \quad \text{Minimum!}$$

Geht man nun wie gewohnt vor, und differenziert nach dem Parameter a, so erhält man die bzgl. a <u>lineare</u> Gleichung

$$a \cdot \sum_{k=1}^{n} g_k e^{-2bx_k} = \sum_{k=1}^{n} g_k y_k e^{-bx_k} \quad .$$

Eine Ableitung nach b führt hingegen zu der <u>nicht-linearen</u> Gleichung

$$a \cdot \sum_{k=1}^{n} g_k x_k e^{-2bx_k} = \sum_{k=1}^{n} g_k x_k y_k e^{-bx_k} \quad .$$

Dementsprechend nennt man a einen *linearen* und b einen *nicht-linearen* Parameter der Modellfunktion $f(x)$.

Aus diesem einfachen Beispiel ist bereits zu ersehen, daß man bei nicht-linearen Modellfunktionen mit dem im Abschnitt 4.4 beschriebenen Verfahren nicht weiterkommt. Eine Möglichkeit der Behandlung nicht-linearer Modelle im Rahmen der LSQ Approximation stellt der im Abschnitt 4.5.3 behandelte *Gauss-Newton Formalismus* dar.

Vorher soll aber noch gezeigt werden, wie man <u>bestimmte</u> nicht-lineare Modellfunktionen auf einfache Weise linearisieren kann.

4.5.2 Linearisierung nicht-linearer Probleme.

Eine in der Praxis häufig vorkommende Meßpunkt-Verteilung ist in der Abb.4.7 dargestellt. Die $(x_k \mid y_k)$-Werte gehorchen offenbar einem *Exponentialgesetz*. Ihre Verteilung in einem halblogarithmischen $(x \mid \ln y)$-Koordinatensystem ist daher annähernd *linear*.

Für die so verteilten Punkte kommt als Modellfunktion offenbar

$$f(x; a, \lambda) = a \cdot e^{-\lambda x}$$

in Betracht, die im halblogarithmischen System die lineare Form

$$\ln y = \ln a - \lambda x$$

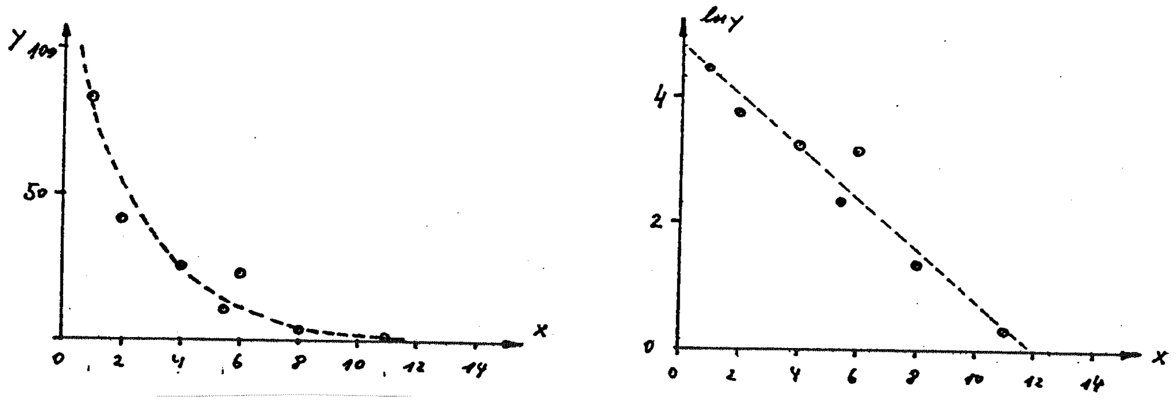


Abbildung 4.7: Nicht-lineare Modellfunktion (Exponentialfunktion) im normalen und halblogarithmischen Koordinatensystem.

hat. Das lineare Gleichungssystem für die beiden Fit-Parameter lautet gemäß den Gleichungen (4.13) und (4.14):

$$\begin{pmatrix} n & \sum_k x_k \\ \sum_k x_k & \sum_k x_k^2 \end{pmatrix} \cdot \begin{pmatrix} \ln a \\ -\lambda \end{pmatrix} = \begin{pmatrix} \sum_k \ln y_k \\ \sum_k x_k \ln y_k \end{pmatrix} \quad ,$$

wobei die Gewichtsfaktoren $g_k = 1$ gesetzt wurden!
Dieses System ist einfach zu lösen und man erhält

$$\begin{aligned} a &= \exp\left[\left(\sum \ln y_k \cdot \sum x_k^2 - \sum x_k \cdot \sum x_k \ln y_k\right)/D\right] \\ \lambda &= -\left(n \cdot \sum x_k \ln y_k - \sum x_k \cdot \sum \ln y_k\right)/D \qquad (4.18) \\ D &= n \cdot \sum x_k^2 - \left(\sum x_k\right)^2 \end{aligned}$$

Dazu ein Beispiel: Die in Abb.4.7 dargestellten 7 Punkte haben die Koordinaten

x:	1.	2.	4.	5.5	6.	8.	11.
y:	83.2	41.7	25.1	10.5	22.9	3.8	1.4

Daraus erhält man unter Anwendung von (4.18)

$$a = 118.90 \quad \text{und} \quad \lambda = 0.398 \quad .$$

Die Ausgleichskurve lautet demnach

$$f(x) = 118.90\, e^{-0.398x}$$

(siehe die gestrichelten Kurven in Abb.4.7), und als Summe der Fehlerquadrate erhält man

$$\chi^2_{min} = 307.3 \quad .$$

Anmerkungen:

1. Diese 'minimale Fehlersumme' ist natürlich nur das Minimum der Summe der quadratischen Abweichungen zwischen den $\ln y_k$ und den $\ln f(x_k)$!

2. Ein großer Nachteil dieser Linearisierungs-Methode besteht auch darin, daß die Angabe von statistischen Gewichtsfaktoren meist nicht möglich ist.

3. Die exponentielle Modellfunktion ist ein gutes Beispiel für *physikalisch relevante* Modellparameter: so könnte $a \cdot \exp(-\lambda x)$ eine radioaktive Zerfallskurve mit a als Anfangsaktivität und λ als Zerfallskonstante darstellen.

In [7], S.330ff findet man eine Reihe weiterer Beispiele zur Linearisierung einfacher nicht-linearer Modellfunktionen.

Abschließend noch ein Hinweis: vermeiden Sie in Ihren Modellfunktionen 'versteckte Korrelationen' zwischen Fit-Parametern. So ist z. B. das Modell

$$f(x; a, b, c) = a\,\mathrm{e}^{-bx+c}$$

fehlerhaft, weil die Parameter a und c numerisch nicht getrennt werden können:

$$f(x; a, b, c) = \underbrace{a\,\mathrm{e}^{c}}_{=\text{ nur 1 Parameter}} \cdot \mathrm{e}^{-bx}\,.$$

4.5.3 Das Gauss-Newton- (GN-)Verfahren.

In all jenen Fällen, in denen eine Linearisierung der Modellfunktion im Sinne des Abschnitts 4.5.2 nicht möglich ist oder nicht gewünscht wird, empfiehlt sich die folgende Vorgangsweise:

Ausgangspunkt ist die völlig allgemeine Modellfunktion

$$f(x; a_1, a_2, \ldots, a_q) \tag{4.19}$$

mit den Parametern $\mathbf{a} \equiv a_1, a_2, \ldots, a_q$.
Die Minimalisierung der gewichteten Fehlersumme

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; \mathbf{a})\right]^2$$

kann nun durch den folgenden *iterativen Prozess* geschehen:

1. Man wählt einen Satz von Anfangswerten (*guessed values*) für die Parameter:
$$\mathbf{a} = \mathbf{a}^0 \quad .$$

2. Man macht eine *Taylor-Entwicklung der Modellfunktion* bzgl. der Parameter an der Stelle $\mathbf{a}^0$ und bricht nach dem linearen Term ab:
$$f(x; \mathbf{a}) \approx f(x; \mathbf{a}^0) + \sum_{l=1}^{q} \left(\frac{\partial f(x; \mathbf{a})}{\partial a_l}\right)_{\mathbf{a}=\mathbf{a}^0} \cdot (a_l - a_l^0) \quad .$$

3. Diese (näherungsweise) *linearisierte Modellfunktion* setzt man nun in die LSQ-Grundgleichung ein:
$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; \mathbf{a}^0) - \sum_{l=1}^{q} \left(\frac{\partial f(x_k; \mathbf{a})}{\partial a_l}\right)_{\mathbf{a}=\mathbf{a}^0} \cdot (a_l - a_l^0)\right]^2 \quad ,$$
wobei für die folgenden Gleichungen die Abkürzungen
$$df_{k,l} \equiv \left(\frac{\partial f(x_k; \mathbf{a})}{\partial a_l}\right)_{\mathbf{a}=\mathbf{a}^0} \qquad \text{sowie} \quad f_k \equiv f(x_k; \mathbf{a}^0)$$
verwendet werden.

4. Erst jetzt wird χ^2 nach den Modellparametern abgeleitet und die Ableitungen werden Null gesetzt:
$$\frac{\partial \chi^2}{\partial a_j} = -2 \sum_{k=1}^{n} g_k \left[y_k - f_k - \sum_{l=1}^{q} df_{k,l}(a_l - a_l^0)\right] \cdot df_{k,j} = 0$$
für $j = 1, \ldots, q$. Als Ergebnis erhält man ein lineares, inhomogenes Gleichungssystem für die q Ausdrücke $(a_l - a_l^0)$:
$$A \cdot (\mathbf{a} - \mathbf{a}^0) = \beta$$
mit
$$A = [\alpha_{ij}] \quad \alpha_{ij} = \sum_{k=1}^{n} g_k df_{k,i} df_{k,j} \quad \text{und} \quad \beta_i = \sum_{k=1}^{n} g_k (y_k - f_k) df_{k,i} \quad . \tag{4.20}$$

5. Die Lösungen dieses Systems, $(a_l - a_l^0)$, kann man nun als Differenzen zwischen den *guessed values* und *verbesserten Werten* der gesuchten Modellparameter ansehen:

$$a_l - a_l^0 \equiv \Delta a_l \rightarrow a_l^1 = a_l^0 + \Delta a_l \quad (l = 1, \ldots, q) \quad .$$

6. Mit diesem *verbesserten Satz von Parametern* wird nun der Ablauf $(2 \rightarrow 3 \rightarrow 4 \rightarrow 5)$ wiederholt. Auf diese Weise (Konvergenz des Verfahrens vorausgesetzt) ist es möglich, die Modellparameter *iterativ zu verbessern*:
$$a_l^t = a_l^{t-1} + \Delta a_l^t \quad (t = 1, 2, \ldots) \tag{4.21}$$

7. Wie bei jedem iterativen Verfahren braucht man auch hier ein *Abbruchkriterium*. In der einschlägigen Literatur werden eine Reihe von möglichen Kriterien diskutiert. Eines von ihnen (nicht immer das beste!) lautet:

   Die Iteration wird abgebrochen, wenn <u>alle</u> Parameter die relative Genauigkeitsabfrage
$$\mid a_l^t - a_l^{t-1} \mid < \mid a_l^t \mid \cdot \epsilon \tag{4.22}$$
   erfüllen (ϵ = Genauigkeit) oder wenn eine maximale Zahl von Iterationen überschritten wird.

8. Nach Erreichen der gewünschten Genauigkeit wird die Normalmatrix des Problems berechnet, deren i, j-ter Koeffizient gemäß (4.6) lautet:

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} \quad .$$

   Mit einer nicht-linearen Modellfunktion erhält man

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} = \sum_{k=1}^{n} g_k \left[\frac{\partial f(x_k; \mathbf{a})}{\partial a_i} \frac{\partial f(x_k; \mathbf{a})}{\partial a_j} - (y_k - f(x_k; \mathbf{a})) \frac{\partial^2 f(x_k; \mathbf{a})}{\partial a_i \partial a_j} \right] \quad .$$

   In den meisten Programmen der nicht-linearen LSQ-Approximation wird nun der zweite Term im obigen Klammerausdruck vernachlässigt. Dies ist dadurch gerechtfertigt, daß im Falle einer erfolgreichen Iteration die Differenz $[y_k - f(x_k; \mathbf{a})]$ i.a. klein sein wird. Man erhält also mit der auf Seite 115 definierten Abkürzung

$$\frac{1}{2} \frac{\partial^2 \chi^2}{\partial a_i \partial a_j} \approx \sum_{k=1}^{n} g_k df_{k,i} df_{k,j} \quad . \tag{4.23}$$

   In dieser Näherung ist also wieder (wie beim linearen Modell) die Normalmatrix identisch mit der Koeffizientenmatrix des LSQ Systems!

9. Invertierung der Normalmatrix = Kovarianzmatrix. Diese enthält wieder die statistische Information über den Fit und die Fit-Parameter.

4.5.4 Konvergenzprobleme beim GN-Verfahren. Die Variation von Marquardt.

Eine primäre Forderung für die praktische Verwendbarkeit numerischer Verfahren, insbesondere iterativer Verfahren, ist die nach deren *Stabilität*. Es muß gewährleistet sein, daß das Verfahren *konvergiert*, und zwar auch dann, *wenn die Anfangswerte der Iteration ungünstig gewählt wurden*.

Im folgenden Testbeispiel wird nun gezeigt, daß das GN-Verfahren diese Forderung keineswegs erfüllt.

Ausgangspunkt für diesen Test ist die Funktion

$$f(x) = 10e^{-3x} + 5e^{-x/2} \quad .$$

Dies entspricht der parametrisierten Funktion

$$f(x; a_1, a_2, a_3, a_4) = a_1 e^{-a_3 x} + a_2 e^{-a_4 x} \tag{4.24}$$

mit den (exakten) Parametern

$$a_1 = 10. \quad a_2 = 5. \quad a_3 = 3. \quad a_4 = 0.5 \quad .$$

Diese Funktion kann dazu verwendet werden, einen *Testdatensatz* für den GN-Algorithmus zu liefern:

```
10 Datenwerte:
            X               Y=F(X)
 1    .1000000E+01     .3530524E+01
 2    .2000000E+01     .1864185E+01
 3    .3000000E+01     .1116885E+01
 4    .4000000E+01     .6767378E+00
 5    .5000000E+01     .4104280E+00
 6    .6000000E+01     .2489355E+00
 7    .7000000E+01     .1509869E+00
 8    .8000000E+01     .9157819E-01
 9    .9000000E+01     .5554498E-01
10    .1000000E+02     .3368973E-01
```

Faßt man diese Daten als Stützpunkt-Koordinaten für eine LSQ-Problem mit (4.24) als Modellfunktion auf, so sollte das Iterationsergebnis des GN-Prozesses natürlich lauten:

$$a_1 \to 10. \quad a_2 \to 5. \quad a_3 \to 3. \quad a_4 \to 0.5$$

Im folgenden wird nun das Konvergenzverhalten eines Programms, das auf dem im Abschnitt 4.5.3 erläuterten Algorithmus beruht, untersucht. Zu diesem Zweck wurde von verschiedenen *guessed values* für die a_3 und a_4 ausgegangen, während für a_1 und a_2 stets die Startwerte 9. und 4. gewählt wurden.

Die Ergebnisse dieses Tests sind in der Abb.4.8 zusammengefaßt.

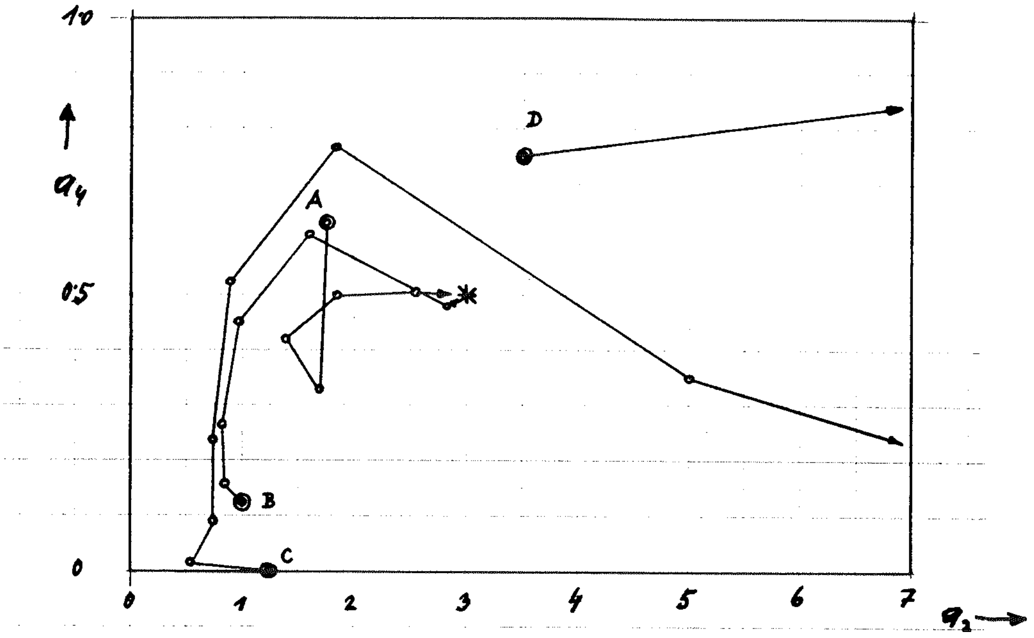


Abbildung 4.8: Konvergenzverhalten des Gauss-Newton Algorithmus.

Diese Testergebnisse sind natürlich sehr unbefriedigend! Es ist nämlich in der Praxis äußerst schwierig, den Konvergenzbereich abzuschätzen. Die Methode, die *guessed values* für die Parameter-Startwerte solange zu variieren, bis die Iteration endlich einmal klappt, ist inakzeptabel!

Einen Ausweg aus diesem Dilemma wurde 1963 von D.W. Marquardt gefunden [1]. Im folgenden werden die Ergebnisse seiner Untersuchungen präsentiert, ohne auf die mathematische Beweisführung einzugehen.

Ausgangspunkt für die *Variation des GN-Verfahrens nach Marquardt* ist das lineare Gleichungssystem (4.20):

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1q} \\ \alpha_{12} & \alpha_{22} & \cdots & \alpha_{2q} \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ \alpha_{1q} & \alpha_{2q} & \cdots & \alpha_{qq} \end{pmatrix} \cdot \begin{pmatrix} \Delta a_1 \\ \Delta a_2 \\ . \\ . \\ . \\ \Delta a_q \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ . \\ . \\ . \\ \beta_q \end{pmatrix}$$

In der Matrizen-Schreibweise sieht das so aus:

$$A \cdot \Delta \mathbf{a} = \beta$$

Gleichungssysteme dieser Art sind also im Rahmen des GM-Algorithmus zu lösen. *Nach der Variation von Marquardt löst man statt dessen das System*

$$(A + \lambda D) \cdot \Delta \mathbf{a} = \beta \quad , \tag{4.25}$$

wobei D eine Diagonalmatrix der Form

$$d_{ii} = \alpha_{ii} \quad (i = 1, \ldots, q) \tag{4.26}$$

[1] D.W. Marquardt, J. Soc. Ind. Appl. Math. **11**,431 (1963)

darstellt. Die Größe λ ist vorläufig noch unbestimmt.

Angenommen, der t-te Iterationsschritt während des GN-Prozesses habe die Fehlerquadratsumme χ_t^2 ergeben. Der darauffolgende Iterationsschritt liefere den Wert χ_{t+1}^2.
Es kann nun ohne weiteres geschehen, daß gilt:

$$\chi_{t+1}^2 > \chi_t^2 \quad .$$

Dies kann jedoch (vgl. den obigen Test, Startpunkt C) unter Umständen zu einer Divergenz des Iterationsprozesses führen.

Die allgemeingültige Aussage, die Marquardt streng mathematisch bewiesen hat, lautet nun:
Es ist immer möglich, die in (4.25) vorkommende Größe λ so zu wählen, daß gilt:

$$\chi_{t+1}^2 \leq \chi_t^2$$

Damit ist aber die Konvergenz des Verfahrens gesichert.

Um die Wirkung der Vergrößerung der (positiven) Größe λ zu demonstrieren, kann man von einem Gleichungssystem (4.25) ausgehen, wobei angenommen wird, daß das verwendete Modell nur 2 Parameter enthält:

$$\begin{pmatrix} \alpha_{11}(1+\lambda) & \alpha_{12} \\ \alpha_{12} & \alpha_{22}(1+\lambda) \end{pmatrix} \cdot \begin{pmatrix} \Delta a_1 \\ \Delta a_2 \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix}$$

Die analytischen Lösungen dieses Systems lauten:

$$\Delta a_1(\lambda) = \frac{\beta_1 \alpha_{22}(1+\lambda) - \beta_2 \alpha_{12}}{\alpha_{11}\alpha_{22}(1+\lambda)^2 - \alpha_{12}^2} \quad \text{und} \quad \Delta a_2(\lambda) = \frac{\beta_2 \alpha_{11}(1+\lambda) - \beta_1 \alpha_{12}}{\alpha_{11}\alpha_{22}(1+\lambda)^2 - \alpha_{12}^2} \quad .$$

Das bedeutet:

- Für $\lambda \to 0$ erhält man die GN Methode.
- Für $\lambda \to \infty$ gehen Δa_1 und Δa_2 gegen Null!

Zusätzlich sieht man sofort, daß auch das Verhältnis der Korrektur-Komponenten, $\Delta a_1/\Delta a_2$, eine Funktion von λ ist. Das bedeutet, daß bei einer λ-Variation nicht nur die Korrekturen kleiner ('vorsichtiger') werden, sondern daß sich auch die Korrektur-Richtung ändert: die Umgebung des Ausgangspunktes wird abgesucht.

<u>Man kann also zusammenfassen:</u>

- *Eine Vergrößerung des Marquardt'schen Parameters λ führt zu einer Verkleinerung der Korrekturwerte für die Modellparameter und damit zu einer Verkleinerung der Konvergenzgeschwindigkeit.*
- *Andererseits garantiert jedoch eine (ausreichende) Vergrößerung von λ die monotone Abnahme der Fehlersummen:*

$$\chi_1^2 \geq \chi_2^2 \geq \chi_3^2 \cdots \geq \chi_t^2 \geq \chi_{t+1}^2 \cdots$$

Eine praktikable Methode, λ möglichst klein (um eine möglichst große Konvergenzgeschwindigkeit zu haben) und im Bedarfsfall doch groß genug zu halten (um die Konvergenz zu sichern), ist die sogenannte *Marquardt'sche Strategie*:

1. Start der Iteration mit einem kleinen (positiven, sonst beliebigen) Wert für λ (im folgenden Programm ist λ z.B. 0.0003).
2. Vor jedem neuen Iterationsschritt wird λ um einen fixen – ebenfalls beliebigen – Faktor verkleinert, um die Konvergenzgeschwindigkeit möglichst groß zu halten (im folgenden Programm um den Faktor 5).
3. Tritt der Fall $\chi^2_{t+1} > \chi^2_t$ auf, wird λ schrittweise um einen fixen Faktor (im folgenden Programm ebenfalls 5) erhöht, und die Auswertung des Gleichungssystems (4.25) wird solange wiederholt, bis $\chi^2_{t+1} \leq \chi^2_t$ erreicht wird. Erst dann wird
4. zum nächsten Iterationsschritt übergegangen.

Testet man den *GN-Algorithmus mit Marquardt-Variation*[2] unter Verwendung der Angaben des gegebenen Testproblems, so erhält man die in Abb.4.9 zusammengefaßten Ergebnisse. Wie man sieht, konvergiert das Verfahren *auch von jenen Startpunkten C und D aus*, bei welchen ohne die Marquardt-Variation eine Divergenz zu beobachten war (vgl. Abb.4.8). Für den Startpunkt D ist im folgenden auch der Rechenverlauf dargestellt, wobei jedes * eine Vergrößerung des Marquardt'schen Parameters λ bedeutet:

```
 Startpunkt D    OHNE MARQUARDT-VARIATION:
 =========================================
 t       chisq             a1            a2            a3            a4

   0   .368339E+01   .900000E+01   .400000E+01   .350000E+01   .750000E+00
                     Exekutionsabbruch wegen Exponenten-Overflow!

 Startpunkt D    MIT MARQUARDT-VARIATION:
 ========================================
 t       chisq             a1            a2            a3            a4

 0    .368339E+01      .900000E+01   .400000E+01   .350000E+01   .750000E+00
*
*
*
 1    .327945E+01      .116540E+02   .478102E+01   .326392E+01   .298864E+00
 2    .221123E+00      .139565E+02   .407836E+01   .262388E+01   .386921E+00
 3    .660544E-02      .776914E+01   .456550E+01   .253007E+01   .468859E+00
 4    .383322E-04      .750890E+01   .492223E+01   .264024E+01   .496633E+00
```

[2]Dieser Algorithmus wird in der Literatur auch als 'Levenberg-Marquardt Methode' bezeichnet

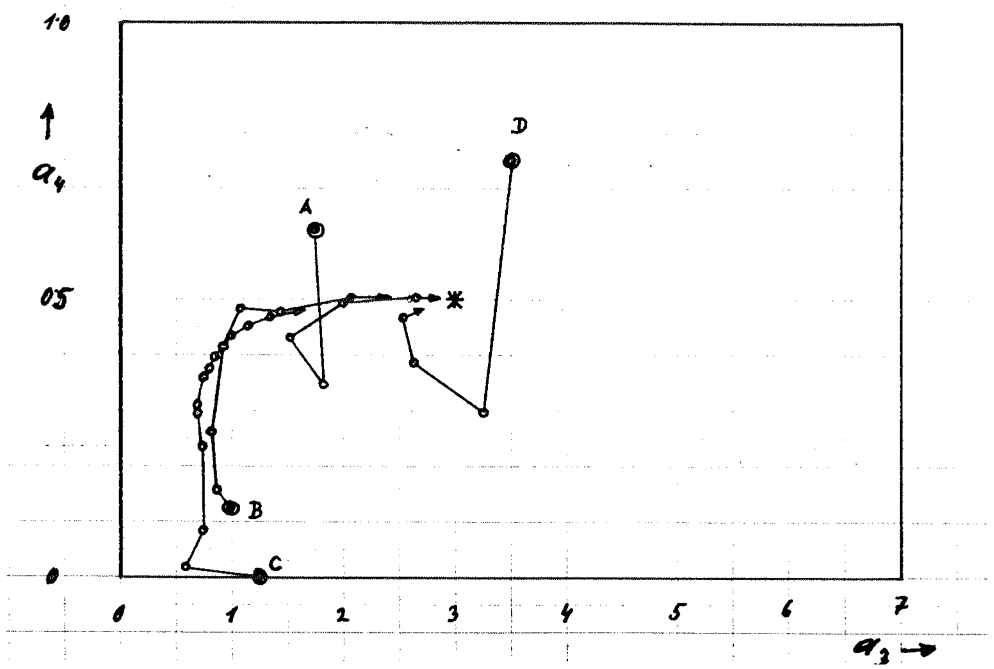


Abbildung 4.9: Konvergenzverhalten des Gauss-Newton-Marquardt Algorithmus.

*					
5	.708586E-05	.823696E+01	.497283E+01	.278430E+01	.498760E+00
*					
6	.320911E-05	.885081E+01	.498274E+01	.286453E+01	.499211E+00
7	.315072E-05	.961475E+01	.499580E+01	.296028E+01	.499807E+00
8	.101244E-06	.996144E+01	.499965E+01	.299644E+01	.499984E+00
9	.147690E-10	.999937E+01	.499999E+01	.299994E+01	.500000E+00
10	.903999E-13	.100000E+02	.500000E+01	.300000E+01	.500000E+00
11	.340838E-13	.999999E+01	.500000E+01	.300000E+01	.500000E+00

4.5.5 Das Programm MRQMIN.

Quelle: [9], S.526f; [10], S. 683ff mit einigen Änderungen.

Das Programm MRQMIN (MaRQuardt MINimalisation) führt einen Iterationsschritt im Rahmen des Gauss-Newton-Marquardt Verfahrens durch.

INPUT-Parameter:

X( ),Y( ): Koordinaten der Stützpunkte.

SIG( ): Standardabweichungen der y-Werte.

NDATA: Anzahl der Stützpunkte.

MA: Anzahl der Parameter in der Modellfunktion.

A( ): Feld der Modellparameter (*guessed values*).

ALAMDA: Steuer-Parameter, Marquardt'scher Parameter λ.

OUTPUT-Parameter:

A( ): Feld der *optimierten* Fit-Parameter.

ALPHA(,): Koeffizientenmatrix des linearen Systems (4.20).

COVAR(,): Kovarianz-Matrix.

BETA(): inhomog. Vektor des linearen Systems (4.20).

CHISQ: gewichtete Fehlerquadratsumme χ^2.

OCHISQ: gewichtete Fehlersumme des vorherigen Iterationsschrittes.

ALAMDA: aktueller Marquardt'scher Parameter.

VMAR: logische Variable, die anzeigt, ob der Iterationsschritt im Sinne einer χ^2-Abnahme erfolgreich war (VMAR=false) oder nicht (VMAR=true).

Interne Felder:

NORMAL(,): Marquardt'sche Koeffizientenmatrix (4.25).

DA(): Korrekturvektor für die Parameter.

VEKTOR(), LOES() .

Benötigte Prozeduren:

MRQCOF: Berechnung der Matrix ALPHA und des Vektors BETA gemäß (4.19) sowie der gewichteten Fehlersumme CHISQ.

LUDCMP und LUBKSB: Lösung des linearen Gleichungssystems (4.25) bzw. Invertierung der Normalmatrix.

Programmstruktur:

1. Beginn der Iteration mit negativem ALAMDA. ALAMDA wird auf den Wert 0.0003[3] gesetzt, dann erfolgt der erste Aufruf von MRQCOF. Aus den dort berechneten Feldern ALPHA(,) und BETA( ) werden die Koeffizienten des Gleichungssytems (4.25) berechnet. Dieses System wird mittels LU-Decomposition gelöst.

2. Wenn ALAMDA Null ist (letzter Aufruf von MRQMIN!), wird durch Invertierung der Normalmatrix NORMAL(,) die Kovarianzmatrix COVAR(,) berechnet. Ende der Rechnung.

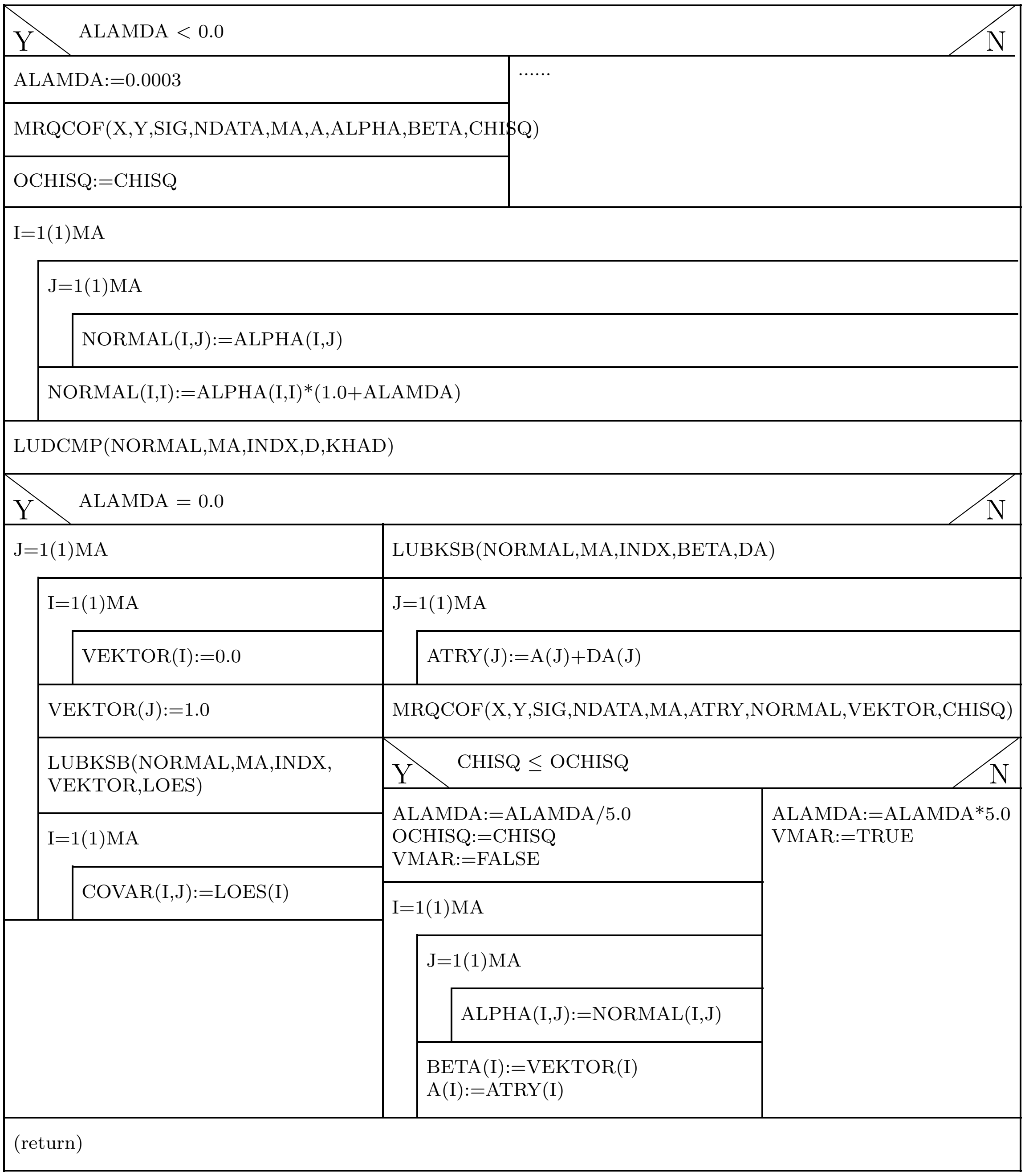
3. Wenn ALAMDA > Null, werden *versuchsweise* (ATRY) die verbesserten(?) Fit-Parameter berechnet. Neuerlicher Aufruf von MRQCOF.

[3]Dieser Startwert für λ ist ebenso wie der weiter unten angegebene Faktor (=5.) willkürlich und hat sich nur empirisch als praktikabel erwiesen.

4. Nun wird die neue, von MRQCOF gelieferte Fehlersumme CHISQ berechnet und mit der alten Fehlersumme OCHISQ verglichen:

    - CHISQ $\leq$ OCHISQ: Der Iterationsschritt war erfolgreich. Die logische Variable MVAR wird 'false' gesetzt, und ATRY( ) wird endgültig auf A( ) abgespeichert. ALAMDA wird durch den Faktor 5. dividiert.
    - CHISQ $>$ OCHISQ: Der Iterationsschritt war *nicht* erfolgreich. Die logische Variable wird 'true' gesetzt, und ALAMDA wird mit dem Faktor 5. multipliziert.

Struktogramm 12 — MRQMIN(X,Y,SIG,NDATA,MA,A,ALPHA,COVAR, BETA,CHISQ,OCHISQ,ALAMDA,VMAR)



4.5.6 Das Programm MRQCOF.

Quelle: [9], S.527f; [10], S. 687.

Das Programm MRQCOF (MaRQuardt COeFficients) berechnet die Matrix ALPHA und den Vektor BETA gemäß (4.20) sowie die gewichtete Fehlerquadratsumme χ^2.

INPUT-Parameter:

X( ),Y( ),SIG( ),NDATA,MA,A( ): siehe Beschreibung von MRQMIN, Abschnitt 4.5.5.

OUTPUT-Parameter:

ALPHA(,),BETA( ): Matrix ALPHA und Vektor BETA gemäß (4.20).

CHISQ: gewichtete Fehlerquadratesumme χ^2.

Benötigte Prozeduren:

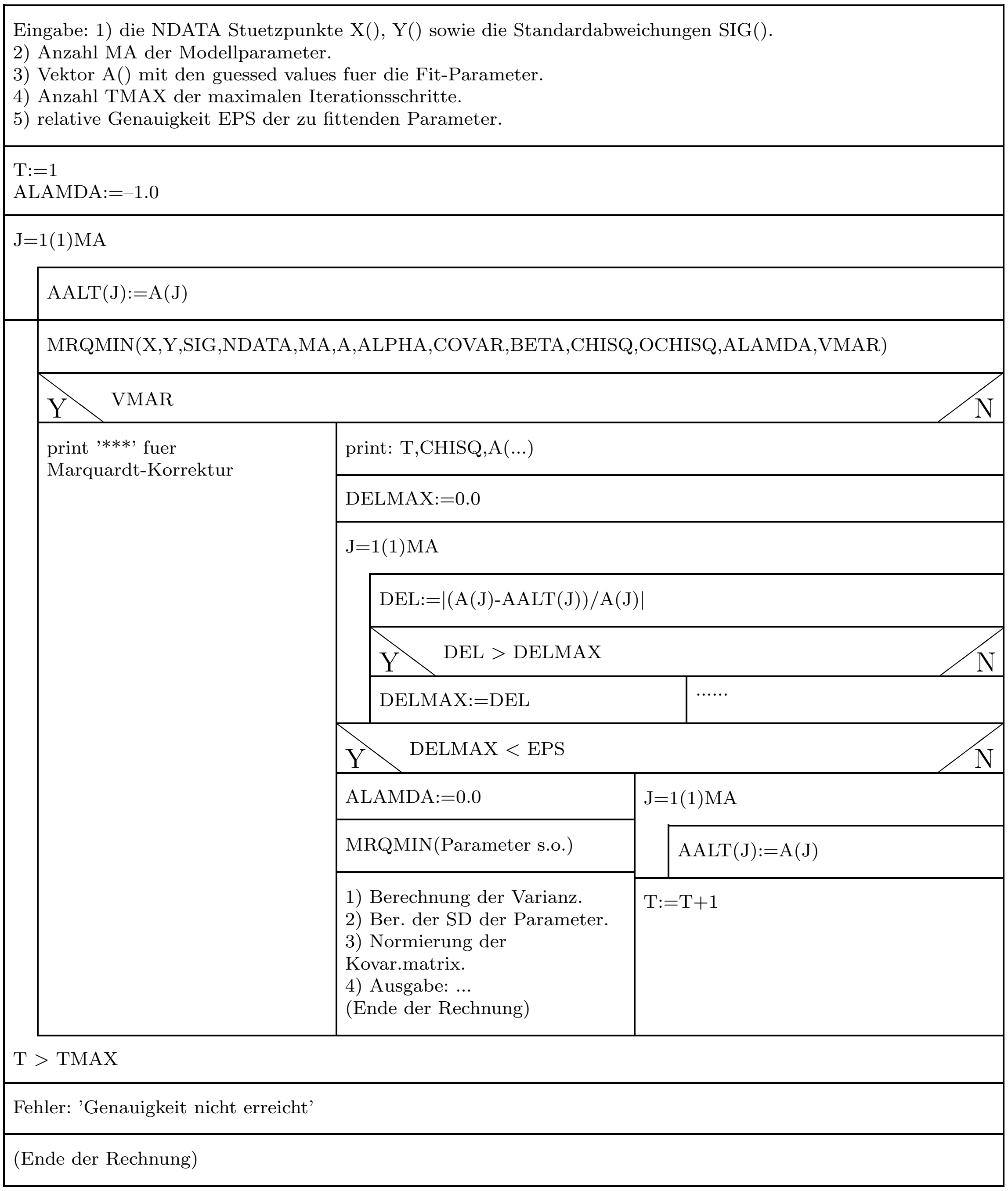
FUNCS: Prozedur, die für das gegebene Argument X und die Modellparameter A( ) den Funktionswert Y der Modellfunktion sowie alle partiellen Ableitungen der Modellfunktion nach den Modellparametern berechnet. Diese MA Ableitungen werden im Feld DYDA( ) gespeichert.

Struktogramm 13 — MRQCOF(X,Y,SIG,NDATA,MA,A,ALPHA,BETA,CHISQ)

```
I=1(1)MA
    J=1(1)I
        ALPHA(I,J):=0.0
    BETA(I):=0.0
CHISQ:=0.0
K=1(1)NDATA
    FUNCS(X(K),A,YMOD,DYDA,MA)
    G:=1.0/SIG(K)/SIG(K)
    DY:=Y(K)-YMOD
    I=1(1)MA
        J=1(1)I
            ALPHA(I,J):=ALPHA(I,J) + G*DYDA(I)*DYDA(J)
        BETA(I):=BETA(I) + G*DY*DYDA(I)
    CHISQ:=CHISQ + G*DY*DY
I=2(1)MA
    J=1(1)I-1
        ALPHA(J,I):=ALPHA(I,J)
(return)
```

4.5.7 Anwendung von MRQMIN und MRQCOF

. — .



Ein Testbeispiel.

Gegeben sei ein radioaktives Präparat, das aus einem Gemisch von J radioaktiven Isotopen besteht. Beide Kernarten zerfallen, ausgehend von einer Anfangsaktivität A_j mit einer Halbwertzeit T_j, gemäß dem *radioaktiven Zerfallsgesetz*:

$$A_j = A_j e^{-\ln 2 \cdot t/T_j} \quad .$$

Die Gesamtaktivität der Quelle beträgt demnach

$$A(t) = \sum_{j=1}^{J} A_j e^{-\ln 2 \cdot t/T_j} \quad .$$

Die Messung geht nun wie folgt vor sich: Während einer fixen Zeitspanne Δ werden die von der Quelle herrührenden Zerfälle gezählt und das Ergebnis wird gespeichert. Danach wird die Zählung für eine zweite Zeitspanne Δ durchgeführt usw. Man erhält auf diese Weise eine Reihe von Zählwerten Z_k, wobei

$$Z_k = \int_{t=(k-1)\Delta}^{k\Delta} dt A(t) \quad (k = 1, 2, \ldots) \quad .$$

Ein solches Experiment ist natürlich ein typisches Zählexperiment d.h. die Meßwerte sind um ihre jeweiligen Erwartungswerte *poisson-verteilt* (s. Abschnitt 4.3.3).

Durch die Auswertung des obigen Integrals erhält man die Modellfunktion mit $2J$ Parametern

$$Z(k; A_1, \ldots, A_J, T_1, \ldots, T_J) = \sum_{j=1}^{J} \frac{A_j}{\ln 2} T_j \left(e^{+\Delta \ln 2/T_j} - 1 \right) e^{-\Delta \ln 2 \cdot k/T_j} \quad .$$

Die Ableitungen $\partial Z/\partial A_j$ bzw. $\partial Z/\partial T_j$ können daraus ohne Probleme berechnet werden, und die Prozedur FUNCS lautet in C-Version etwa folgendermaßen:

```
#define DELTA 15.0    // Konstante des Experiments
void funcs(double x, double a[], double *z, double dzda[], int ma)
// C-VERSION
{
  int   mterm,j,ind;
  double con,fac1,fac2,fac3,fac4;
  con=DELTA*log(2.0);
  mterm=ma/2;
  *z=0.0;
  for(j=1;j<=mterm;j++) {
    ind=mterm+j;
    fac1=con/a[ind];
    fac2=exp(fac1);
    fac3=fac2-1.0;
    fac4=exp(-fac1*x);

    dzda[j]=a[ind]*fac3*fac4/log(2.0);
    *z=*z + a[j]*dzda[j];
    dzda[ind]=a[j]/log(2.0)*fac4*(fac3*(1.0+fac1*x)-fac1*fac2);
  }
}
```

Wie Sie sehen, wird die Routine *funcs* für jeden x-Wert gesondert aufgerufen. Eine solche Vorgangsweise ist im Falle einer MATLAB-Realisierung nicht optimal, weil die Fähigkeit dieser Sprache, sehr effektiv mit Vektoren umzugehen, nicht ausgenutzt wird. Besser ist es, wie im folgenden Beispiel dargestellt, die Eingabegröße x von vornherein als Vektor aller Werte x_i mit $i = 1, \ldots, NDATA$ vorzusehen:

```
  function [z,dzda] = funcs(x,a,ma,ndata);
% MATLAB-VERSION  x ist ein Vektor mit ndata Komponenten

  dzda=zeros(ma,ndata);
  delta=15.0;             % Messzeit
  con=delta*log(2);
  mterm=fix(ma/2);

    z=zeros(1,ndata);
    for j=1:mterm
      fac1=con/a(mterm+j);
      fac2=exp(fac1);
      fac3=fac2-1;
      fac4=exp(-fac1*x);

      dzda(j,:)=a(mterm+j)*fac3.*fac4/log(2);
      z=z+a(j)*dzda(j,:);
      dzda(mterm+j,:)=a(j)/log(2).*fac4.*(fac3*(1+fac1*x)-fac1*fac2);
    end
```

Nun die Angaben zu einem konkreten Beispiel:

Gemessen wurden die Zählraten eines aus zwei Komponenten bestehenden radioaktiven Präparates d.h. $J = 2$. Es wurden 40 Messungen durchgeführt, jede über einen Zeitraum von $\Delta = 15$ Sekunden.

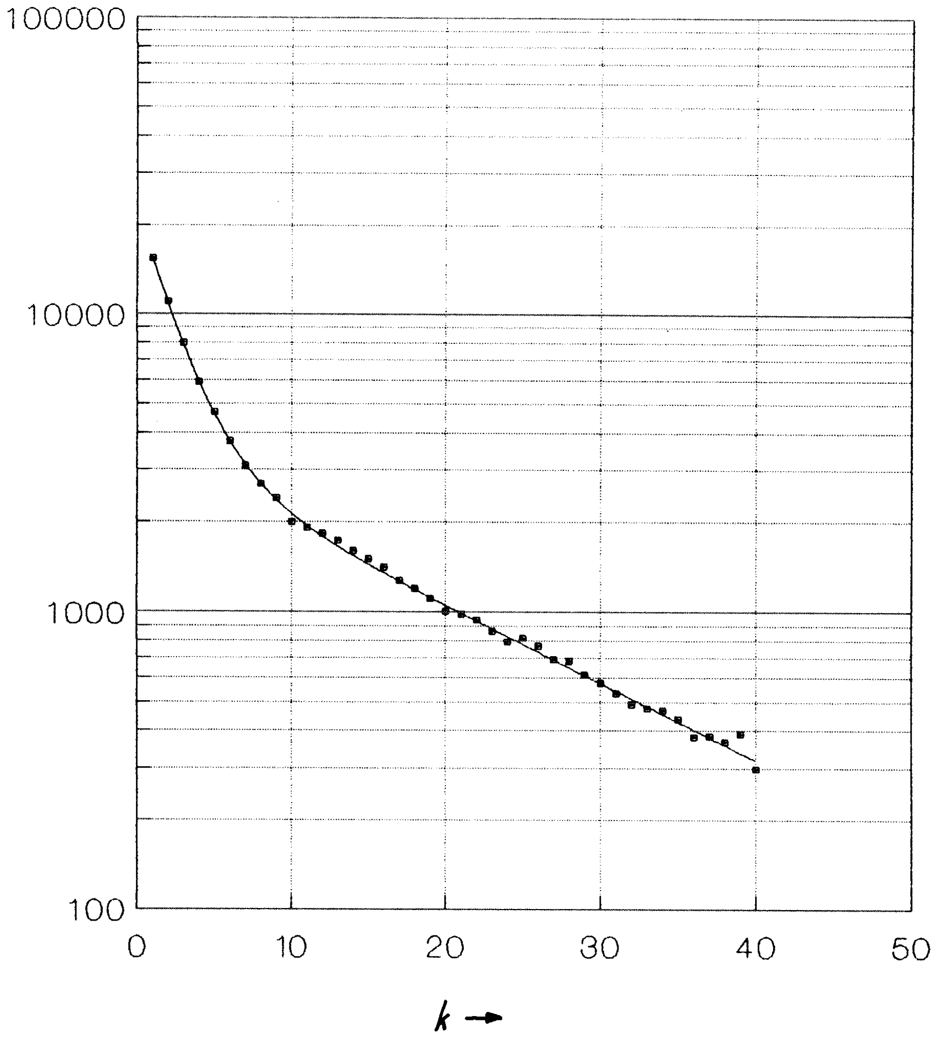
```
  40 Datenwerte:

   1  15376.0          21      981.0
   2  10903.0          22      939.0
   3   7950.0          23      857.0
   4   5865.0          24      790.0
   5   4653.0          25      814.0
   6   3721.0          26      766.0
   7   3089.0          27      691.0
   8   2683.0          28      681.0
   9   2396.0          29      614.0
  10   1992.0          30      576.0
  11   1910.0          31      529.0
  12   1820.0          32      488.0
  13   1726.0          33      472.0
  14   1600.0          34      464.0
  15   1495.0          35      434.0
```

16	1410.0	36	380.0
17	1271.0	37	382.0
18	1197.0	38	365.0
19	1106.0	39	387.0
20	1004.0	40	296.0

Der Verlauf der Iteration mittels des Gauss-Newton-Marquardt-Verfahrens ist:

```
  t     chisq         A1         A2         T1         T2

  0   196876.304   2000.000   500.000   30.000   200.000
  1     1233.069    976.760   237.577   26.378   184.784
  2       45.645    998.414   229.228   22.949   172.341
  3       43.535   1005.360   226.315   23.159   173.250
  4       43.535   1005.458   226.349   23.153   173.245
  5       43.535   1005.457   226.348   23.153   173.246
Konvergenz wurde erreicht!

VARIANZ =  1.209  (sollte zwischen 0.764 und 1.236 liegen)

          Modellparameter        SD

  A1          1005.457          10.182
  A2           226.348           4.129
  T1(s)         23.153           0.353
  T2(s)        173.246           2.320

 Die Korrelationsmatrix:
            A1         A2         T1         T2

  A1      1.0000    -0.0494    -0.4642     0.0811
  A2     -0.0494     1.0000    -0.7345    -0.9370
  T1     -0.4642    -0.7345     1.0000     0.6405
  T2      0.0811    -0.9370     0.6405     1.0000
```

Zum Abschluß noch eine grafische Darstellung der gegebenen Meßpunkte und der gefitteten Modellfunktion:


100000
10000
1000
100
0
10
20
30
40
50
k →

4.6 Ergänzungen

Zwangsbedingungen

Im folgenden wird kurz die Vorgangsweise erläutert, wie man beim Least-Squares-Prozess verfährt, wenn die Modellparameter bestimmten linearen Zwangsbedingungen (*constraints*) genügen müssen. Solche Nebenbedingungen kommen in der Praxis relativ häufig vor.

Angenommen, man hat eine Modellfunktion mit q Parametern vorliegen. In diesem Fall kann es maximal $q-1$ Zwangsbedingungen geben. Unter der Voraussetzung, daß diese Bedingungen <u>linear</u> in den Parametern sind, können sie so geschrieben werden:

$$\sum_{l=1}^{q} \gamma_{t,l}\, a_l \overset{!}{=} \delta_t \qquad (t = 1, 2, \ldots, L < q)\,. \tag{4.27}$$

Es gibt also L Bedingungen, wobei die $\gamma_{t,l}$ und δ_t fixe Größen sind.

In diesem Fall hat die Least-Squares-Grundgleichung die Form

$$\chi^2 = \sum_{k=1}^{n} g_k \left[y_k - f(x_k; a_1, \ldots, a_q)\right]^2 + \sum_{t=1}^{L} \mu_t \left[\sum_{l=1}^{q} \gamma_{t,l}\, a_l - \delta_t\right] \quad \longrightarrow \quad \text{Min.}\,, \tag{4.28}$$

wobei die neuen Fit-Parameter μ_t (die sog. *Lagrange*-Parameter), die wie die übrigen Parameter behandelt werden, in vielen Fällen jedoch keine physikalische Bedeutung haben.

Least-squares when both variables have uncertainties

Ein wichtiges Problem bei der Auswertung experimenteller Daten mittels LSQ-Methoden besteht darin, daß in vielen Fällen nicht nur die y-Daten mit Meßfehlern behaftet sind, sondern auch die x-Daten: denken Sie etwa an die Messung irgendeiner physikalischen Größe in Abhängigkeit von der Zeit: in solchen Fällen wird in der Regel nicht nur die eigentliche Meßgröße, sondern auch die (auf der Abszisse aufgetragene) Zeit statistische Fehler aufweisen.

In derartigen Situationen verbietet sich die Anwendung von *Standard-LSQ-Methoden*, wie sie in den bisherigen Abschnitten dieses Kapitels verwendet wurden, und man sollte die sog. *effective variance method* einsetzen. Diese Methode werde ich im Rahmen dieser LV auf Basis der folgenden Publikation erläutern:

J. Orear, Am. J. Phys. **50**, 912 (1982)

4.7 Software-Angebot

Wie bei vielen Themen, ist das Problem bei der Internet-Suche nicht ein zu wenig, sondern ein zu viel: eine Eingabe des Stichwortes 'Levenberg-Marquardt' in die GOOGLE-Suchmaschine gibt über 6000 Eintragungen ...

- **FORTRAN**-User sollten in die bewährte NETLIB hineinschauen, insbesondere in das Programmpaket **ODRPACK**, das auf dem Levenberg-Marquardt-Verfahren beruht. Für Leute, die professionell mit diesem Programm arbeiten wollen, gibt es dort eine Menge Information, z.B. auf dem PS-File *guide.ps*.

- Für die **C**-User bin ich noch auf der Suche. Ich bin sicher, daß es auch sehr gute Levenberg-Marquardt-Realisationen in dieser Sprache gibt!

- *MATLAB* bietet eine Menge, sowohl für *linear* als auch für *non-linear fitting*. Ausführlich können Sie sich unter

  `www.mathworks.com/access/helpdesk/help/toolbox/optim/optim.shtml`

  über die *Optimization Toolbox* informieren. Durch Anklicken der Stichworte 'Gauss-Newton Method' bzw. 'Levenberg-Marquardt Method' erfahren Sie Interessantes zum diesen Themen.

- **Mathematica:** Fit[data,functions,variables]
  Ein Beispiel finden Sie im *Mathematica-Handbook* von S. Wolfram, S. 674.
  Im Paket *Statistics'NonlinearFit'* gibt es die einschlägigen Routinen *NonlinearFit* und *NonlinearRegress*.

Kapitel 5

Numerische Lösung von transzendenten Gleichungen

5.1 Das grundsätzliche Problem.

Gegeben sei eine reellwertige Funktion $F(x)$. Gesucht seien jene reellen Werte von x, für welche die Gleichung

$$F(x) = 0 \tag{5.1}$$

erfüllt ist. Die x-Werte, die diese Bedingung erfüllen, werden *Lösungen, Nullstellen, Wurzeln, ...* von (5.1) genannt.

Aus der Fülle der in der Literatur beschriebenen Methoden sollen in diesem Kapitel die folgenden Verfahren behandelt werden:

- Iterationsverfahren (Newton-Raphson-Verfahren; Regula Falsi).
- Das Intervallschachtelungsverfahren.
- Die numerische Behandlung nicht-linearer Gleichungssysteme.

Anmerkung: Die Betonung auf *transzendente* Gleichungen im Titel dieses Kapitels schließt natürlich die Behandlung *algebraischer* Gleichungen vom Typus

$$F(x) \equiv P_m(x) = \sum_{j=1}^{m} \alpha_j x^{j-1} = 0$$

nicht aus! In der hier verwendeten Terminologie sind algebraische Gleichungen lediglich Spezialfälle transzendenter Gleichungen.
Für die numerische Bestimmung von Nullstellen von algebraischen Gleichungen (Polynomen) werden in der Literatur eine Reihe von speziellen Verfahren beschrieben (s. z.B. das Verfahren von Lobatschewski und Graeffe [20], S. 60ff etc.), die im Rahmen dieser Lehrveranstaltung nicht behandelt werden.

5.2 Iterationsverfahren.

5.2.1 Allgemeines.

Die Gleichung $F(x) = 0$ kann immer (und fast immer auf mehrere Arten) auf die Form

$$x = f(x) \tag{5.2}$$

gebracht werden. Auf diese Weise erhält man einen typischen *Iterationsansatz* für die Ermittlung einer Nullstelle:

$$\begin{array}{rcl} x_0 & & \text{Startwert} \\ x_1 & = & f(x_0) \\ x_2 & = & f(x_1) \\ & . & \\ & . & \\ x_{t+1} & = & f(x_t) \qquad (t = 0, 1, 2, \ldots) \end{array} \tag{5.3}$$

Im Falle einer Konvergenz führt eine solche Iteration beliebig genau (bis auf Rundungsfehler) an die exakte Lösung des Problems heran. Es gilt

$$\lim_{t\to\infty} x_t \to x_{exakt}$$

Ein solches Konvergenzverhalten ist jedoch keineswegs selbstverständlich, sondern *hängt sehr von der Art der Umformung von (5.1) in (5.2) ab*, wie im folgenden Beispiel demonstriert werden soll.

Gesucht ist die Nullstelle der Funktion $F(x) = x^3 - x - 5$ *in der Umgebung von 2*. Es soll nun gezeigt werden, wie verschieden das Konvergenzverhalten ist, wenn die Umformung auf (5.2) auf verschiedene Weise erfolgt:

$$a) \quad x = x^3 - 5 \qquad b) \quad x = \frac{5}{x^2 - 1} \qquad c) \quad x = \sqrt[3]{x + 5}$$

```
                x0 = 2.0

t          (a)          (b)           (c)
---------------------------------------------
0            2         2             2
1            3          1.6667       1.9129
2           22          2.8125       1.9050
3        10643          0.7236       1.9042
4            .        -10.4944       1.9042
.            .         .              .
.            .         .              .
           DIV        DIV           KONV
```

(b) zeigt ein interessantes Verhalten: s. Demo in der Vorlesung!

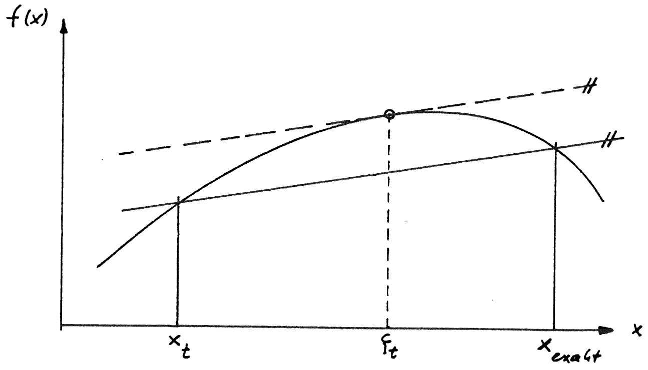


Abbildung 5.1: Zur Fehlerabschätzung bei Iterationen.

5.2.2 Konvergenzkriterien und Fehlerabschätzungen.

Um die Bedingung zu erhalten, unter der eine Konvergenz der Iteration gewährleistet ist, geht man von der Gleichung (5.2) für die exakte Lösung

$$x_{exakt} = f(x_{exakt}) \tag{5.4}$$

aus. Beim $(t+1)$-ten Iterationsschritt erhält man hingegen

$$x_{t+1} = f(x_t) - \delta_t \quad , \tag{5.5}$$

wobei δ_t den Rundungsfehler darstellt, der bei der numerischen Auswertung der Funktion f für $x = x_t$ auftritt. Subtraktion von (5.5) von (5.6) ergibt

$$x_{exakt} - x_{t+1} = f(x_{exakt}) - f(x_t) + \delta_t \,.$$

Unter Anwendung des Mittelwertsatzes (s. Abb.5.1) kann man schreiben:

$$\frac{f(x_{exakt}) - f(x_t)}{x_{exakt} - x_t} = f'(\xi_t) \quad \text{mit} \quad \xi_t \in [x_t, x_{exakt}] \quad .$$

Somit ergibt sich weiters

$$x_{exakt} - x_{t+1} = f'(\xi_t) \cdot (x_{exakt} - x_t) + \delta_t \tag{5.6}$$

Diese Gleichung kann Schritt für Schritt reduziert werden zu

$$\begin{aligned} x_{exakt} - x_{t+1} \quad = \quad & f'(\xi_t) f'(\xi_{t-1}) \cdots f'(\xi_0) \cdot (x_{exakt} - x_0) + \\ & + \delta_t + f'(\xi_t)\delta_{t-1} + f'(\xi_t) f'(\xi_{t-1})\delta_{t-2} + \\ & + \cdots + f'(\xi_t) f'(\xi_{t-1}) \cdots f'(\xi_1) \cdot \delta_0 \end{aligned}$$

Für eine *Fehlerabschätzung* soll nun angenommen werden, daß

1. die Funktion im Intervall I als Maximalbetrag ihrer ersten Ableitung den Wert m habe, wobei gilt, daß $x_0, x_1, \ldots, x_{t+1}, \ldots, x_{exakt} \in I$:

$$m = \max_I \mid f'(x) \mid \tag{5.7}$$

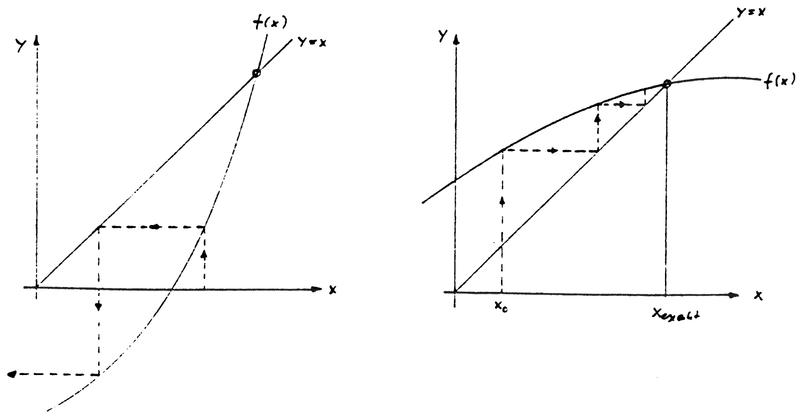


Abbildung 5.2: Konvergenzverhalten der Iteration (5.3).

2. *δ nicht vom Iterationsindex abhängt*, d.h. daß für alle $t = 0, 1, \ldots$ gilt:

$$\delta_t \approx \delta \quad .$$

Dann ergibt sich

$$\mid x_{exakt} - x_{t+1} \mid \leq m^{t+1} \mid x_{exakt} - x_0 \mid + \mid \delta \mid (1 + m + m^2 + \cdots m^t)$$

Der Grenzwert dieses Ausdruckes lautet:

$$\lim_{t\to\infty} \mid x_{exakt} - x_{t+1} \mid \leq \underbrace{\mid x_{exakt} - x_0 \mid \cdot \lim_{t\to\infty} m^{t+1}}_{\text{Verfahrensfehler}} + \underbrace{\frac{\mid \delta \mid}{1-m}}_{\text{Rundungsfehler}} \quad .$$

Beide Terme divergieren für $m \geq 1$, d.h. das *Konvergenzkriterium* für die Iteration lautet:

$$0 \leq m < 1 \quad . \tag{5.8}$$

Ist diese Bedingung erfüllt, ergibt sich

$$\lim_{t\to\infty} \mid x_{exakt} - x_{t+1} \mid \leq \frac{\mid \delta \mid}{1-m} \quad . \tag{5.9}$$

Dieses Ergebnis ist von entscheidender Bedeutung! Es zeigt die *numerische Stabilität* des Iterationsverfahrens: Die Größe des auftretenden Rundungsfehlers konvergiert ebenfalls zu einem Grenzwert. All diese Zusammenhänge sind in den Abbildungen 5.2 und 5.3 dargestellt.

Was nun die Möglichkeiten betrifft, die auftretenden Fehler *abzuschätzen*, kann man wieder von der Gleichung (5.6) ausgehen:

$$\begin{aligned}
x_{exakt} - x_{t+1} &= f'(\xi_t) \cdot (x_{exakt} - x_t) + \delta_t \\
&= f'(\xi_t) \cdot (x_{exakt} - x_t + x_{t+1} - x_{t+1}) + \delta_t \\
(x_{exakt} - x_{t+1}) \cdot (1 - f'(\xi_t)) &= f'(\xi_t) \cdot (x_{t+1} - x_t) + \delta_t \\
(x_{exakt} - x_{t+1}) &= \frac{f'(\xi_t)}{1 - f'(\xi_t)} (x_{t+1} - x_t) + \frac{\delta_t}{1 - f'(\xi_t)} \quad .
\end{aligned}$$

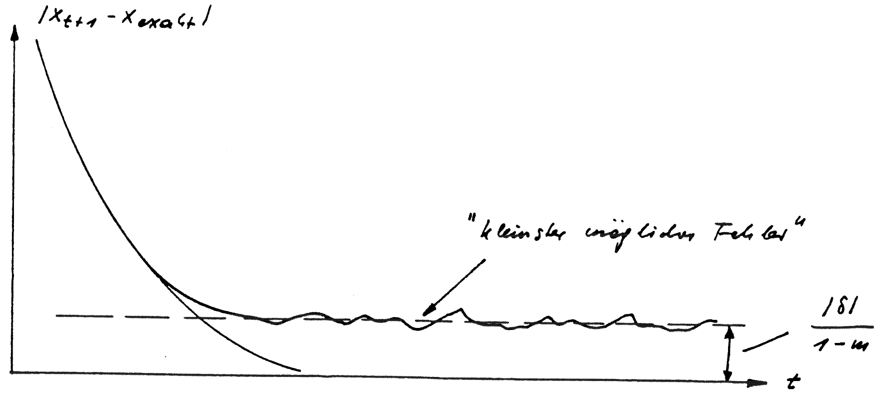


Abbildung 5.3: Fehlerentwicklung der Iteration (5.3).

Im Falle einer Konvergenz gilt nun die Fehlerabschätzung

$$\mid x_{exakt} - x_{t+1} \mid \leq \frac{m}{1-m} \mid x_{t+1} - x_t \mid + \frac{\mid \delta \mid}{1-m} \quad . \tag{5.10}$$

Wenn es möglich ist, die maximale erste Ableitung der Funktion $f(x)$ im Intervall I zu bestimmen bzw. die Größe δ abzuschätzen, kann (5.10) als Basis für eine *Fehlerkontrolle* der Iteration genommen werden: Man iteriert solange, bis der Ausdruck rechts in (5.10) kleiner geworden ist als eine gewünschte Genauigkeitsschranke ϵ, wobei jedoch zu berücksichtigen ist, daß ϵ nicht kleiner sein darf als die durch $\mid \delta \mid /(1-m)$ gegebene *Grenzgenauigkeit*!

In der Praxis ist jedoch die Bestimmung von m und δ meistens nicht möglich oder zu kompliziert. Dann behilft man sich mit der stark vereinfachten Fehlerabschätzung

$$\mid x_{exakt} - x_{t+1} \mid \leq \mid x_{t+1} - x_t \mid \quad , \tag{5.11}$$

woraus sich diese

Abbruchbedingung — .

Y | |x(t+1)-x(t)| < EPS	N
Ende der Iteration	naechster Iterationsschritt

für die Iteration ergibt.
Es muß aber ausdrücklich darauf hingewiesen werden, daß (5.11) nur für $m \leq 1/2$ gilt, hingegen für $1/2 < m < 1$ grob falsch sein kann! Außerdem ist (5.11) nur solange nützlich, *als der Verfahrensfehler größer ist als der Rundungsfehler!*

Das folgende Beispiel soll einige der eben besprochenen Zusammenhänge illustrieren:

Die Funktion, deren Nullstelle numerisch ermittelt werden soll, lautet

$$F(x) = a + (1-a)x^2 - x \quad .$$

Wie man sich leicht überzeugen kann, hat diese quadratische Funktion die beiden Nullstellen

$$x_1 = 1 \qquad \text{und} \qquad x_2 = \frac{a}{1-a} \quad .$$

Die Nullstelle x_1 soll nun durch Ausführung der Iteration

$$x_{t+1} = a + (1-a)x_t^2$$

approximiert werden.

Die erste Ableitung von $f(x)$ hat an der Stelle $x = 1$ den Wert $2(1-a)$. Dies ist gleichzeitig der Maximalwert der Ableitung im Iterationsbereich, d. h. es gilt

$$m = |2(1-a)| \, .$$

Es ist auf Grund der Ergebnisse (5.8) und (5.10) zu erwarten, daß

- die Iteration für $m \geq 1$ d. h. für $a \leq 0.5$ bzw. $a \geq 1.5$ divergiert;
- die vereinfachte Fehlerabfrage (5.11) für $1/2 < m < 1$, d.h. für $a < 0.75$ versagt.

Ergebnisse dieser Testrechnung:

```
.
Parameter  a = 2.500  > 1.5  d.h. Divergenz ist zu erwarten:

   t        x_{t}      x_{ex}-x_{t+1}    x_{t+1}-x_{t}

   0    .1200000E+01
   1    .3400000E+00    .6600000E+00     -.8600000E+00
   2    .2326600E+01   -.1326600E+01      .1986600E+01
   3   -.5619601E+01    .6619601E+01     -.7946201E+01
   4   -.4486988E+02    .4586988E+02     -.3925028E+02
   5   -.3017459E+04    .3018459E+04     -.2972589E+04
   6   -.1365759E+08    .1365759E+08     -.1365457E+08
   7   -.2797945E+15    .2797945E+15     -.2797945E+15
   8   -.1174274E+30    .1174274E+30     -.1174274E+30
   9   -.2068380E+59    .2068380E+59     -.2068380E+59
  10   -.6417295+117    .6417295+117     -.6417295+117
```

.

```
Parameter  a = 0.600  ===> m=0.8  d.h.: Konvergenz, aber
                                  Versagen von (5.11):

   t        x_{t}      x_{ex}-x_{t+1}    x_{t+1}-x_{t}

   0    .6000000E+00
   1    .7440000E+00    .2560000E+00      .1440000E+00
   2    .8214144E+00    .1785856E+00      .7741440E-01
   3    .8698886E+00    .1301114E+00      .4847425E-01
   4    .9026825E+00    .9731750E-01      .3279386E-01
   5    .9259343E+00    .7406572E-01      .2325178E-01
   6    .9429417E+00    .5705828E-01      .1700744E-01
   7    .9556556E+00    .4434437E-01      .1271392E-01
   8    .9653111E+00    .3468892E-01      .9655443E-02
   9    .9727302E+00    .2726981E-01      .7419114E-02
  10    .9784816E+00    .2151839E-01      .5751419E-02


Parameter  a = 1.200  ===> m=0.4  d.h.: Konvergenz und
                                  Gueltigkeit von (5.11):


   0    .6000000E+00
   1    .1128000E+01   -.1280000E+00      .5280000E+00
   2    .9455232E+00    .5447680E-01     -.1824768E+00
   3    .1021197E+01   -.2119718E-01      .7567398E-01
   4    .9914313E+00    .8568734E-02     -.2976591E-01
   5    .1003413E+01   -.3412809E-02      .1198154E-01
   6    .9986325E+00    .1367453E-02     -.4780262E-02
   7    .1000547E+01   -.5466072E-03      .1914060E-02
   8    .9997813E+00    .2187027E-03     -.7653099E-03
   9    .1000087E+01   -.8747150E-04      .3061742E-03
  10    .9999650E+00    .3499013E-04     -.1224616E-03
```

5.3 Das Newton-Raphson-Verfahren.

Im Abschnitt 5.2.1 wurde darauf hingewiesen, daß es gewöhnlich mehrere Arten gibt, die Gleichung $F(x) = 0$ auf die äquivalente Form $f(x) = x$ zu bringen. Eine sehr wichtige Umformung ist gegeben durch

$$f(x) = x - \frac{F(x)}{F'(x)} \quad , \tag{5.12}$$

woraus sich aus (5.4) die bekannte *Newton-Raphson-Iteration* mit der Vorschrift

$$x_{t+1} = x_t - \frac{F(x_t)}{F'(x_t)} \tag{5.13}$$

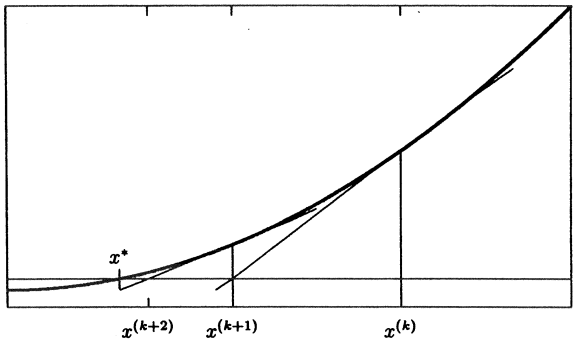


Abbildung 5.4: Grafische Interpretation der Newton-Raphson-Iteration.

ergibt. Die grafische Interpretation dieser Formel ('Tangentenmethode') ist hinlänglich bekannt (s. Abb.5.4).

Für die Fehlerabschätzung gilt wieder (5.10), wobei aber m gemäß (5.9, 5.12) die konkrete Form

$$m = \max_I \frac{d}{dx}\left(x - \frac{F(x)}{F'(x)}\right) = \max_{[x_0, x_{exakt}]}\left(\frac{F(x)F''(x)}{[F'(x)]^2}\right)$$

hat. Wegen der Konvergenzbedingung (5.8), nach der m echt kleiner als Eins sein muß, kann also gesagt werden:
Bei der Newton-Raphson-Iteration ist mit Schwierigkeiten zu rechnen, wenn in der Umgebung der gesuchten Nullstelle die Steigung von $F(x)$ klein wird, wie dies z.B. bei nahe beieinanderliegenden Nullstellen vorkommen kann.

Wenn aber die Newton-Raphson-Methode funktioniert, dann konvergiert sie sehr schnell, wie die folgende Rechnung demonstrieren soll:

Entwickelt man die Funktion $F(x)$ an der Stelle x_t in eine Taylorreihe bis zum (einschließlich) dritten Term, so ergibt sich

$$F(x) = F(x_t) + F'(x_t)(x - x_t) + \frac{1}{2}F''(x_t)(x - x_t)^2 \,.$$

Setzt man für x die exakte Nullstelle, so erhält man nach Division durch $F'(x_t)$

$$0 = -\frac{F(x_t)}{F'(x_t)} - (x_{exakt} - x_t) - \frac{1}{2}\frac{F''(x_t)}{F'(x_t)}(x_{exakt} - x_t)^2 \,.$$

Der erste Term rechts entspricht dem Korrekturterm in der Newton-Raphson-Formel, d.h. man erhält weiters

$$0 = x_{t+1} - x_t - x_{exakt} + x_t - \frac{1}{2}\frac{F''(x_t)}{F'(x_t)}(x_{exakt} - x_t)^2$$

bzw.

$$(x_{exakt} - x_{t+1}) = -\frac{1}{2}\frac{F''(x_t)}{F'(x_t)}(x_{exakt} - x_t)^2 \,. \tag{5.14}$$

Aus diesem Ergebnis geht hervor, daß der absolute Fehler der Nullstelle zum $(t+1)$-ten Iterationsschritt proportional zum *Quadrat* des absoluten Fehlers

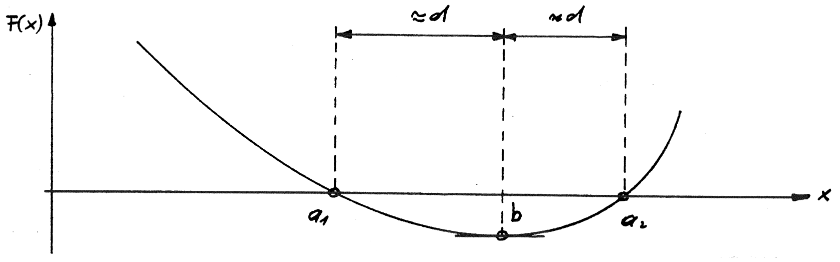


Abbildung 5.5: Zur Methode von Macon.

zum $t-$ten Iterationsschritt ist.
Man sagt: *die Newton-Raphson-Methode ist quadratisch konvergent.*

Eine Möglichkeit, mit eng nebeneinander liegenden Nullstellen-Paaren fertigzuwerden, ist die

5.3.1 Methode von Macon.

Im allgemeinen bereitet es keine Schwierigkeiten, das Minimum oder Maximum zwischen den beiden Nullstellen a_1 und a_2 zu lokalisieren. Nimmt man an, der Abszissenwert des Extremums, b, sei hinlänglich genau bekannt, so kann man $F(x)$ an der Stelle b in eine Taylorreihe entwickeln:

$$F(x) = F(b) + (x-b)F'(b) + \frac{1}{2}(x-b)^2 F''(b) + \cdots \quad . \tag{5.15}$$

Man kann nun in erster Näherung annehmen, daß die beiden Nullstellen a_1 und a_2 *symmetrisch* um b verteilt sind und jeweils den Abstand d vom Extremum haben. Dann gilt

$$F(b+d) \approx 0 \qquad \text{und} \qquad F(b-d) \approx 0 \quad .$$

Eingesetzt in (5.15) ergibt das

$$0 \approx F(b) + \frac{1}{2}(+d)^2 F''(b) \qquad 0 \approx F(b) + \frac{1}{2}(-d)^2 F''(b) \quad .$$

Diese beiden Gleichungen sind äquivalent und erlauben die näherungsweise Berechnung von

$$d = \pm\sqrt{-\frac{2F(b)}{F''(b)}} \quad . \tag{5.16}$$

Startet man nun die Newton-Raphson-Iteration mit $b-d$ bzw. $b+d$ als Anfangswerte, gibt es meist keine Konvergenzprobleme.

5.3.2 Das Programm RTNEWT.

Quelle: [9],S. 257f mit Änderungen.

Das Funktions-Unterprogramm RTNEWT (RooT NEWTon) berechnet innerhalb eines gegebenen Intervalls eine reelle Nullstelle mittels der Newton-Raphson-Iteration.

INPUT-Parameter:

X1,X2: Anfang und Ende eines Intervalls, in welchem die Nullstelle liegt.

JMAX: maximale Anzahl von Iterationsschritten.

XACC: relative Genauigkeitsschranke gemäß Glg.(5.11).

OUTPUT-Parameter:

RTNEWT: Näherungswert für die Nullstelle.

FEHLER: für die Fehlerdiagnostik:

FEHLER = 0:	Newton-Iteration ok.
FEHLER = 1:	Kein Vorzeichenwechsel in [X1,X2]
FEHLER = 2:	Keine Konvergenz innerhalb JMAX Iterationen
FEHLER = 3:	'jumped out of brackets' während Newton.

Interne Variable:

FUNC,DF: $F(x)$ bzw. $F'(x)$. Die Prozedur, in der diese Größen berechnet werden, muß den Namen **FUNC** haben.

DX: Iterations-Korrektur.

Anmerkungen:

- An Beginn des Programms wird abgefragt, ob (mindestens) eine Nullstelle im Suchintervall [X1,X2] liegt. Dies wird daran erkannt, ob die Funktion $F(x)$ bzgl. der Intervallgrenzen einen Vorzeichenwechsel hat.

- Die im Programm verwendete *relative* Fehlerabfrage führt zu Konvergenzproblemen, wenn die gesuchte Nullstelle den Wert 0.0 hat.

- Der erste 'Notausgang' des Programms RTNEWT ('jumped out of brackets') wird verwendet, wenn während der Iteration das vom aufrufenden Programm vorgegebene Iterations-Intervall verlassen wird (s. Abb.5.6, links).

- Der zweite 'Notausgang' ('exceeding maximum iterations') ist notwendig im Falle einer Divergenz oder auch einer zu langsamen Konvergenz, aber auch in speziellen Fällen, wo sich die Iteration in einem Loop verfängt (s. Abb.5.6, rechts).

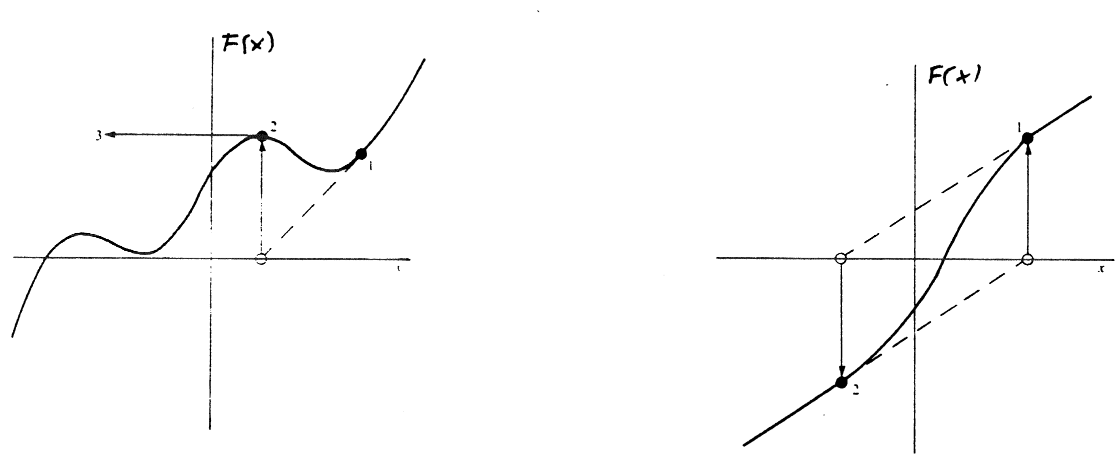


Abbildung 5.6: Probleme bei der Newton-Raphson-Iteration.

Struktogramm 14 — FUNCTION RTNEWT(X1,X2,JMAX, XACC,FEHLER)

X:=0.5*(X1+X2)

Y (FUNC(X1,DF)*FUNC(X2,DF)) > 0.0 N

Y	N
FEHLER:=1 RTNEWT:=X print:ERR(1) 'no zero in interval' (return)	......

FEHLER:=2
J:=0

> DX:=FUNC(X,DF)/DF
> X:=X-DX
>
> Y (X1-X)*(X-X2) < 0.0 N
>
> | Y | N |
> |---|---|
> | FEHLER:=3 | Y $\vert$DX/X$\vert$ < XACC N
Y: FEHLER:=0
N: ...... |
>
> J:=J+1

J>JMAX .or. FEHLER≠2

Y FEHLER=2 N

Y	N
print: 'ERR(2): no convergence'	......

Y FEHLER=3 N

Y	N
print: 'ERR(3): jumped out of brackets'	......

RTNEWT:=X

(return)

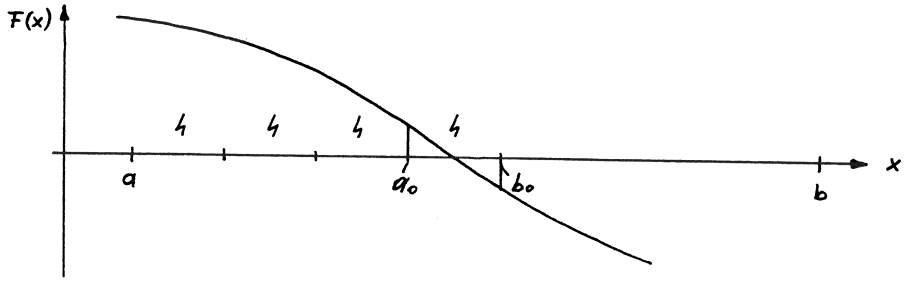


Abbildung 5.7: Prinzip einer Nullstellen-Grobsuche.

5.3.3 Ein Testprogramm für RTNEWT.

Im allgemeinen hat die gegebene Funktion $F(x)$ mehr als eine reelle Nullstelle. In solchen Fällen ist es vernünftig, die Newton-Raphson-Iteration mit einer sogenannten 'Grob-Suche' zu kombinieren.
Das Prinzip dieser einfachen Strategie ist in der Abb. 5.7 dargestellt: die x-Achse wird in Schritten der Breite h auf einen Vorzeichenwechsel der Funktion $F(x)$ untersucht. Wenn ein Intervall mit der Eigenschaft $F(a_0) \cdot F(b_0)$ auftritt, so befindet sich in $[a_0, b_0]$ (mindestens) <u>eine</u> Nullstelle, und die Newton-Iteration kann beginnen.
Solch eine 'Grob-Suche' ist auch Teil des in Abschnitt 5.5 behandelten *Intervallschachtelungsverfahrens*.

Es folgt nun ein Anwendungsbeispiel in C:

```
#include <stdio.h>
#include <math.h>

double func(double x,double *df)
// Diese Funktion berechnet die Funktionswerte sowie
// die Werte der ersten Ableitung fuer das Testbeispiel 5.3.3
{
  *df=4*pow(x,3)-27*pow(x,2)-4*x+120;
  return pow(x,4)-9*pow(x,3)-2*pow(x,2)+120*x-130;
}

double rtnewt(double x1, double x2, int jmax, double xacc, int *fehler)
// Diese Funktion bestimmt die reelle Nullstelle der Funktion 'func'
// im Intervall [x1,x2] mit der relativen Genauigkeit 'xacc'.
// Nach Beendigung der Rechnung kann die Variable 'fehler' einen der
// folgenden Werte haben:

//          fehler=0       Die Newton-Raphson Iteration ist ok.
//          fehler=1       Kein Vorzeichenwechsel der Funktion 'func'
//                         im Intervall [x1,x2].
//          fehler=2       Keine Konvergenz waehrend 'jmax' Iter.schritten.
//          fehler=3       Waehrend der Iteration springt der x Wert
//                         aus dem Intervall [x1,x2].
```

```
{
  int j,pause;
  double x,df,dx;

  x=0.5*(x1+x2);

  if((func(x1,&df)*func(x2,&df)) > 0.0) {
    *fehler=1;
    printf("ERROR(1): kein Vorzeichenwechsel in [x1,x2]\n");
    return x;
  }

  *fehler=2;
  j=0;

  do {
    dx=func(x,&df)/df;
    x=x-dx;

    if(((x1-x)*(x-x2))<0.0) *fehler=3;
    else {
      if(fabs(dx/x) < xacc) *fehler=0;
    }
    j++;
  } while ((j<=jmax) && (*fehler==2));

  if(*fehler==2)printf("ERROR(2):  keine Konvergenz\n");
  if(*fehler==3)printf("ERROR(3):  x aus [x1,x2] gesprungen\n");

  return x;
}


//            ****** main program ******
void main()
{
  int jmax,fehler,pause;
  double a,b,hgrob,eps,xl,xr,yl,yr,df,zero;

// Parameter fuer die Grobsuche:
  a=-10.0;  b=10.0;  hgrob=0.5;

// Parameter fuer die Newton-Raphson-Iteration:
  jmax=20;
  eps=0.0000001;

  printf("TEST:  NEWTON-RAPHSON-ITERATION MIT GROBSUCHE\n");
  printf("\n   Grobsuch-Bereich:       %7.3f  bis  %7.3f\n",a,b);
```

```
  printf("   Schrittweite Grobsuche: %7.3f\n",hgrob);
  printf("       Par. fuer Newton:  rel. Genauigkeit   = %12.10f\n", eps);
  printf("                          max. Iter.schritte = %4i\n\n", jmax);

// Grobsuche:
  xl=a;
  yl=func(xl,&df);
  xr=a+hgrob;
  yr=func(xr,&df);

  do {
    if(yl*yr < 0.0) {
// Grobsuche hat Vorzeichenwechsel gefunden; Newton-Iteration wird gestartet:
      zero=rtnewt(xl,xr,jmax,eps,&fehler);

      if(fehler==0)
        printf("  Nullstelle = %9.6f\n",zero);
      else printf("   fehler(%1i) in RTNEWT\n",fehler);
    }
    xl=xr;
    yl=yr;
    xr=xl+hgrob;
    yr=func(xr,&df);
  } while(xl <= b);
  printf("\nRechnung beendet.\n");
}
```

```
*********************************************
TEST:  NEWTON-RAPHSON-ITERATION MIT GROBSUCHE
*********************************************

   Grobsuch-Bereich:       -10.000  bis   10.000
   Schrittweite Grobsuche:   0.500
       Par. fuer Newton:  rel. Genauigkeit   = 0.0000001000
                          max. Iter.schritte =   20

  Nullstelle = -3.600135
  Nullstelle =  1.228589
  Nullstelle =  3.972068
  Nullstelle =  7.399477

Rechnung beendet.
```

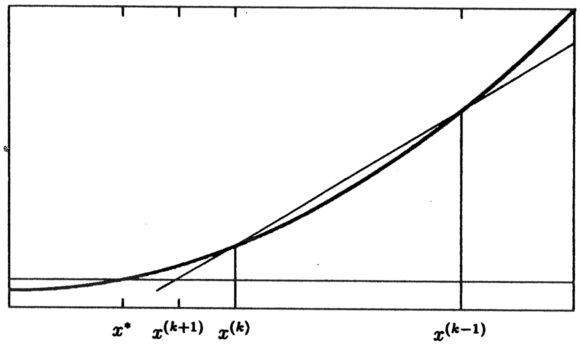


Abbildung 5.8: Grafische Interpretation der Regula Falsi.

5.4 Die Regula Falsi.

Diese mit dem Newton-Raphson-Verfahren nahe verwandte Methode wird häufig dann verwendet, wenn die Berechnung der Ableitung der Funktion $F(x)$ aus irgendwelchen Gründen Schwierigkeiten bereitet. In solchen Fällen kann der *Differentialquotient* in (5.13) durch den *Differenzenquotienten* ersetzt werden:

$$x_{t+1} = x_t - F(x_t) \cdot \frac{x_t - x_{t-1}}{F(x_t) - F(x_{t-1})} \tag{5.17}$$

Grafisch bedeutet das, daß die Newton'sche *Tangentenmethode* durch eine *Sekantenmethode* ersetzt wird (s. Abb.5.8).

5.5 Das Intervallschachtelungsverfahren.

Um die rellen Nullstellen der Funktion $F(x)$ in einem vorgegebenen Intervall $[a, b]$ zu ermitteln, wird - ausgehend von a - mit einer vorgegebenen Schrittweite h eine *grobe Lokalisierung* der Nullstellen vorgenommen.

Angenommen, die erste Nullstelle befinde sich im Intervall $[a_0, b_0]$ (s. Abb.5.7). Dies wird daran erkannt, daß innerhalb dieses Intervalls die Funktion $F(x)$ einen *Vorzeichenwechsel* hat, daß also gilt:

$$F(a_0) \cdot F(b_0) < 0\,.$$

Ist ein Intervall mit der obigen Eigenschaft gefunden worden, folgt die eigentliche Intervall<u>schachtelung</u>. Diese beginnt mit der Mittelung des Intervalls $[a_0, b_0]$:

$$x_0 = \frac{a_0 + b_0}{2}\,.$$

<u>Es gibt nun drei Möglichkeiten:</u>

1. $F(x_0) = 0$: x_0 (genau) die gesuchte Nullstelle.
   (Dieser Fall wird wegen der Rundungsfehler bei der Funktionsauswertung selten auftreten).

2. $F(a_0) \cdot F(x_0) < 0$ In diesem Fall liegt die gesuchte Nullstelle im <u>linken</u> Halbintervall $[a_0, x_0]$.

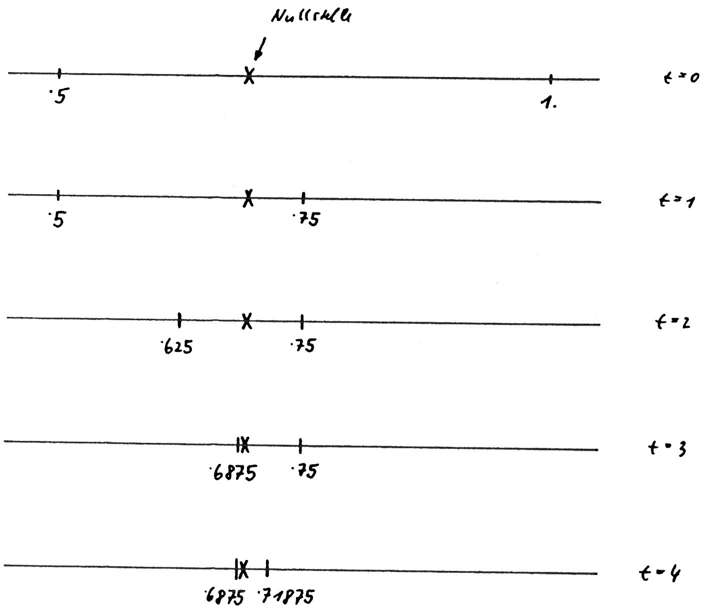
3. $F(a_0)\cdot F(x_0) > 0$ In diesem Fall liegt die gesuchte Nullstelle im rechten Halbintervall $[a_0, x_0]$.

Im Fall (1) ist die Nullstelle gefunden, in den Fällen (2) und (3) wird das linke bzw. rechte Halbintervall weiter geteilt. Dieser Prozess wird solange fortgesetzt, bis die Nullstelle mit der gewünschten Genauigkeit ermittelt ist, d. h. bis die aktuelle Intervallbreite nach t Teilungen ($[a_t, b_t]$) kleiner ist als die geforderte Genauigkeit der Nullstelle.

Dieses einfache Algorithmus wird durch das folgende Beispiel verdeutlicht: gesucht sei die Nullstelle von

$$F(x) = \mathrm{e}^{-x} - \frac{1}{2},$$

welche den Wert $x_{exakt} = \ln 2 = 0.6931472\ldots$ hat. Das mittels einer Grobsuche gefundene Anfangs-Intervall $[a_0, b_0]$ sei [0.5,1.0]. Die Intervallschachtelung läuft nun wie folgt ab:



Es soll noch erwähnt werden, daß die Intervallschachtelungsmethode zu jenen numerischen Verfahren gehört, bei denen – zumindest für den absoluten Fehler – eine *a priori* Fehlerabschätzung in der Form

$$|x_t - x_{exakt}| \leq \frac{1}{2}(b_t - a_t) = \frac{b_0 - a_0}{2^{t+1}}, \tag{5.18}$$

möglich ist, wobei t die Anzahl der erfolgten Intervallhalbierungen ist. Diese Formel zeigt einen Nachteil dieses Verfahrens auf, nämlich seine relativ langsame Konvergenz: man braucht 3.3 Rechenschritte, um die absolute Genauigkeit des Ergebnisses um eine Zehnerpotenz zu erhöhen (s. Abschnitt 5.5.2).

5.5.1 Probleme beim Intervallschachtelungsverfahren.

Im Prinzip ist diese Methode sehr sicher. Ist eine Nullstelle erst einmal im Intervall $[a_0, b_0]$ lokalisiert, so ist ihre Berechnung (abgesehen von Rundungs-

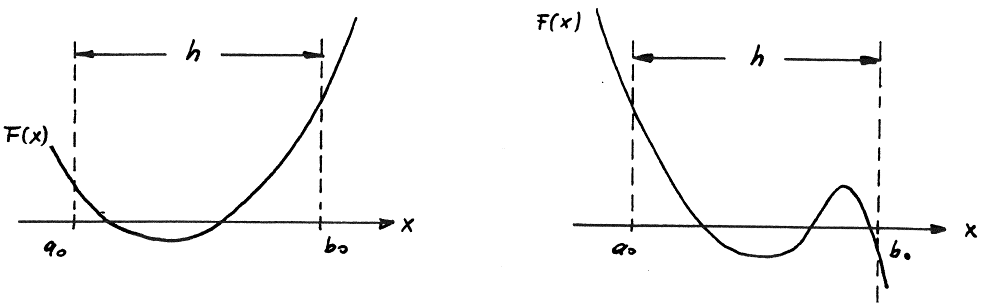


Abbildung 5.9: Probleme beim Intervallschachtelungsverfahren.

fehlern) mit beliebiger Genauigkeit möglich. Es gibt also keinerlei Konvergenzprobleme.
Die einzige Unsicherheitsquelle liegt in der *Wahl der Schrittweite h für die Grobsuche*. Das Kriterium: Vorzeichenwechsel = Nullstelle ist nämlich keinesfalls unproblematisch (s. Abb. 5.9):

- Im Fall Abb. 5.9, links findet *kein Vorzeichenwechsel statt, obwohl sich innerhalb des Intervalls Nullstellen befinden.* Dies trifft offenbar immer dann zu, wenn sich im Intervall $[x_1, x_2]$ eine *gerade* Anzahl von Nullstellen befindet.

- Im Fall Abb. 5.9, rechts *resultiert der Vorzeichenwechsel aus einer ungeraden Anzahl von Nullstellen.*

In beiden Fällen ist also die Anfangs-Schrittweite h zu groß gewählt worden! h stellt also die *Auflösungsgrenze* des Verfahrens dar:

Es können (im allgemeinen) nur solche Nullstellen gefunden werden, deren Abstand nicht kleiner ist als die Anfangs-Schrittweite h.

5.5.2 Das Programm INTSCH.

Quelle: [2],S.281f mit einigen Änderungen und Vereinfachungen.

INPUT-Parameter:

ANF,AEND: Beginn und Ende des Grobsuchbereiches.

H: Schrittweite der Grobsuche.

GEN: Fehlergrenze (i. a. relativer Fehler; s. Anmerkung).

ANZMAX: maximale Zahl der Nullstellen gemäß dem Speicherplatz im Ausgabefeld NULLST.

OUTPUT-Parameter:

NULLST( ): Feld mit den berechneten reellen Nullstellen.

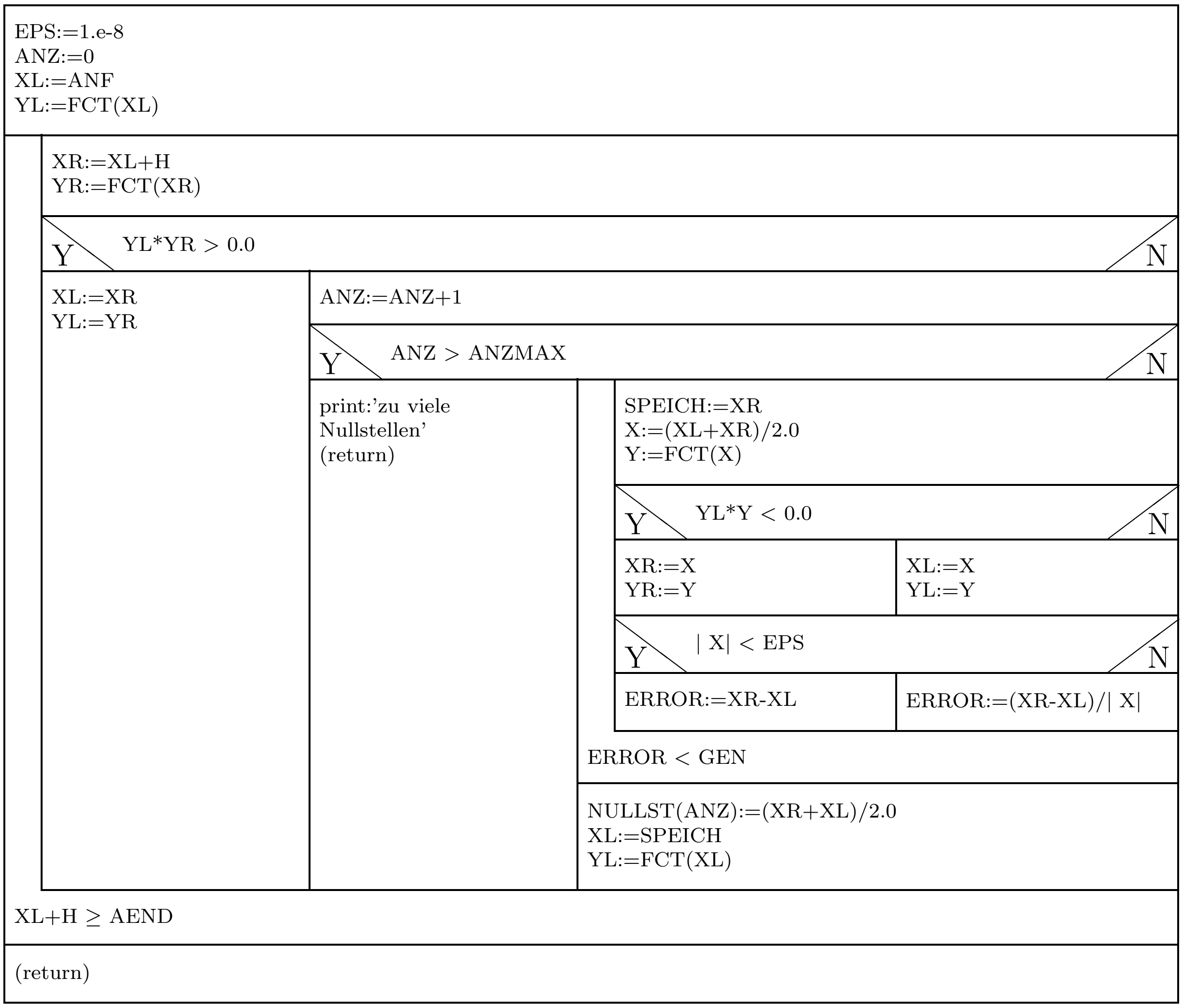
ANZ: Anzahl der von INTSCH berechneten Nullstellen.

Fehlerdiagnostik:
ANZ=0: es wurden keine Nullstellen gefunden.
ANZ=ANZMAX+1: es wurden mehr als ANZMAX Nullstellen gefunden, aber nur die ANZMAX ersten Nullstellen wurden im Feld NULLST abgespeichert.

allgemeine Anmerkung:
Das im folgenden präsentierte Programm ist gegenüber der Quelle deutlich vereinfacht; insbesondere ist der Fall nicht berücksichtigt, daß eine Nullstelle *exakt* auf einer Intervallgrenze liegt.

Anmerkung zur Fehlerabfrage:
Im Programm INTSCH wird standardmäßig eine relative Fehlerabfrage durchgeführt. Nur wenn die Nullstelle sehr klein ist (dem Betrage nach kleiner als 10^{-8}), wird auf eine absolute Fehlerabfrage umgeschaltet.

Struktogramm 15 — INTSCH(ANF,AEND,H,GEN,ANZMAX,NULLST,ANZ)



Eine Testbeispiel für INTSCH

Die reellen Nullstellen der algebraischen Gleichung

$$F(x) = x^4 - 9x^3 - 2x^2 + 120x - 130 = 0$$

sind im Bereich von -10.0 bis $+10.0$ mit einer relativen Genauigkeit von GEN=10^{-7} zu bestimmen; die Intervallbreite bei der Grobsuche sei 0.5.

INTSCH liefert die folgenden vier Nullstellen:

```
.
     1      -3.600135
     2       1.228589
     3       3.972068
     4       7.399477
```

Dieses Ergebnis ist natürlich exakt dasselbe wie bei der Newton-Iteration (vgl. Abschnitt 5.3.3). Es ist hier interessant, den Aufwand zu untersuchen, den beide Methoden benötigen, um ein Ergebnis dieser Genauigkeit zu erzielen.
Wenn man die Anzahl der Funktionsaufrufe von 'func' (im Newton-Raphson-Programm) abzählt, kommt man auf 66 Aufrufe; die gleiche Analyse für das Intervallschachtelungsprogramm ergibt 131 Aufrufe von 'fct'. Da aber beim Newton-Verfahren bei jedem Aufruf von 'func' 2 Funktionen berechnet werden, nämlich der Funktionswert sowie der Wert der ersten Ableitung, ist der numerische Aufwand in beiden Programmen etwa gleich.

Diese Analyse gibt allerdings ein falsches Bild von der Leistungsfähigkeit der Newton-Raphson-Iteration. Ein großer Teil der Funktionsaufrufe von 'func' geschieht bei der Grobsuche, bei der die Berechnung der Ableitungen unnötig ist! Es wäre daher ökonomischer, beim Newton-Raphson-Verfahren zwei unabhängige Funktionen (z.B. 'func' und 'dfunc') zu definieren, wobei 'func' <u>nur</u> die Funktion und 'dfunc' nur die Ableitung berechnet.
Geht man so vor, ergeben sich bei der Anwendung des Programms RTNEWT auf das Testproblem nur mehr 66 Aufrufe von 'func' und 15 Aufrufe von 'dfunc', also insgesamt 81 Funktionsaufrufe.

D.h.: Je ökonomischer man die Grobsuche gestaltet, desto stärker wird die Überlegenheit der Newton-Iteration gegenüber der Intervallschachtelung deutlich. Betrachten Sie etwa die erste Nullstelle: sie liegt im 'Grobsuch-Intervall' [-4.0, -3.5]. Im folgenden sehen Sie die weitere Annäherung an die Nullstelle, links mittels Newton-Raphson und rechts mittels Intervallschachtelung:

```
Newton-Raphson    Int.schachtelung
-3.750000            -3.750000
-3.609011            -3.625000
-3.600169            -3.562500
-3.600135            -3.593750
-3.600135            -3.609375
                     -3.601562
                     -3.597656
                     -3.599609
                     -3.600586
                     -3.600098
                     -3.600342
                     -3.600220
                     -3.600159
                     -3.600128
                     -3.600143
                     -3.600136
                     -3.600132
                     -3.600134
                     -3.600135
                     -3.600135
                     -3.600135
                     -3.600135
```

Ein Vorteil der Intervallschachtelungsmethode gegenüber Newton-Raphson besteht natürlich darin, daß erstere keine Ableitungen der Funktion $F(x)$ benötigt! Diesen Vorteil teilt sie mit der im Abschnitt 5.4 kurz beschriebenen Regula Falsi.

5.5.3 Ein Anwendungsbeispiel aus der Quantenmechanik.

Im folgenden Beispiel geht es um die Energie-Eigenwerte eines eindimensionalen Potentialtopfes:

Gesucht sind jene Energiewerte E, die ein Teilchen der Masse m unter dem Einfluß des *Kastenpotentials* (s.Abb.5.10)

$$V(x) = \begin{array}{lll} 0 & \text{für} \quad -\infty < x \leq 0 & \text{Bereich I} \\ -V_0 & \text{für} \quad 0 < x \leq a & \text{Bereich II} \\ 0 & \text{für} \quad a < x < \infty & \text{Bereich III} \end{array}$$

annehmen kann, wobei nur *gebundene Zustände* berücksichtigt werden sollen:

$$-V_0 < E < 0$$

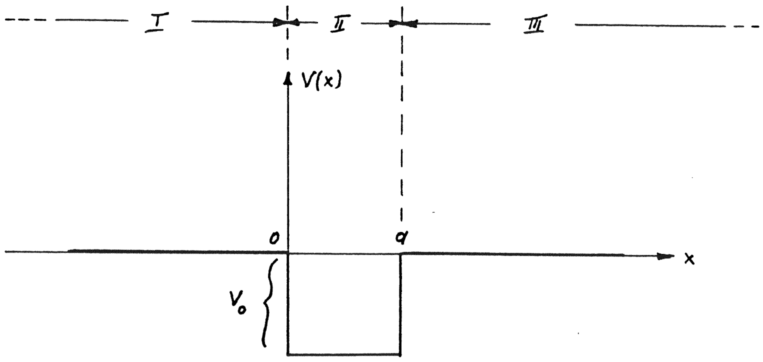


Abbildung 5.10: Der eindimensionale Potentialtopf.

Zu lösen ist also die eindimensionale Schrödingergleichung

$$-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}\psi(x) + V(x)\psi(x) = E\psi(x) \quad .$$

In atomaren Einheiten, d.h. Längen in Bohr, Energien in Rydberg ergibt sich $\hbar^2/2m \equiv 1$, und man erhält die Differentialgleichung

$$\psi''(x) + [E - V(x)]\psi(x) = 0$$

mit den Randbedingungen

$$\psi(+\infty) \to 0 \qquad \text{und} \qquad \psi(-\infty) \to 0 \quad .$$

Lösungsansätze:

- Für die Bereiche I und III:

$$\psi''(x) + E \cdot \psi(x) = 0 \quad \to \quad \begin{array}{c} \psi_I(x) = A_I e^{\sqrt{-E}x} + B_I e^{-\sqrt{-E}x} \\ \psi_{III}(x) = A_{III} e^{\sqrt{-E}x} + B_{III} e^{-\sqrt{-E}x} \end{array}$$

  Die Randbedingungen sind nur für $B_I = A_{III} = 0$ erfüllt, und man erhält:

$$\psi_I(x) = A_I e^{\sqrt{-E}x} \qquad \text{und} \qquad \psi_{III}(x) = B_{III} e^{-\sqrt{-E}x} \tag{5.19}$$

- Bereich II:

$$\psi''(x) + [E + V_0]\psi(x) = 0 \quad \to$$

$$\psi_{II}(x) = A_{II} \sin\left(x\sqrt{E + V_0}\right) + B_{II} \cos\left(x\sqrt{E + V_0}\right) \tag{5.20}$$

Zusätzlich sind noch die 4 *Anschlußbedingungen*

$$\begin{aligned} \psi_I(0) &= \psi_{II}(0) \\ \psi_I'(0) &= \psi_{II}'(0) \\ \psi_{II}(a) &= \psi_{III}(a) \\ \psi_{II}'(a) &= \psi_{III}'(a) \end{aligned} \tag{5.21}$$

zu erfüllen. Setzt man nun die Ansätze (5.19) und (5.20) in die Gleichungen (5.21) ein, so erhält man 4 lineare, homogene Gleichungen für die noch unbekannten Koeffizienten A_I, A_{II}, B_{II} und B_{III}:

(Abkürzung: $\sqrt{E+V_0} \equiv \kappa$).

$$\begin{aligned} A_I - B_{II} &= 0 \\ A_I\sqrt{-E} - A_{II}\kappa &= 0 \\ A_{II}\sin\kappa a + B_{II}\cos\kappa a - B_{III}e^{-\sqrt{-E}a} &= 0 \\ A_{II}\kappa\cos\kappa a - B_{II}\kappa\sin\kappa a + B_{III}\sqrt{-E}e^{-\sqrt{-E}a} &= 0 \end{aligned}$$

In Matrix-Schreibweise hat dieses System die Form

$$\underbrace{\begin{pmatrix} 1 & 0 & -1 & 0 \\ \sqrt{-E} & -\kappa & 0 & 0 \\ 0 & \sin\kappa a & \cos\kappa a & -e^{-\sqrt{-E}a} \\ 0 & \kappa\cos\kappa a & -\kappa\sin\kappa a & \sqrt{-E}e^{-\sqrt{-E}a} \end{pmatrix}}_{M(E)} \begin{pmatrix} A_I \\ A_{II} \\ B_{II} \\ B_{III} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad (5.22)$$

Der Lösungsvektor des Systems (5.22) ist nur dann vom Nullvektor verschieden, wenn die Determinante der Koeffizientenmatrix $M(E)$ verschwindet, d.h. das Teilchen kann nur jene Energiewerte haben, für welche gilt:

$$\det(M(E)) = 0$$

Die analytische Auswertung der Determinante ist elementar, und man erhält den Ausdruck

$$\det(M(E)) = -e^{-a\sqrt{-E}}\left[(V_0+2E)\sin\left(a\sqrt{E+V_0}\right) - 2\sqrt{-E(E+V_0)}\cos\left(a\sqrt{E+V_0}\right)\right]$$

Unter Berücksichtigung der Tatsache, daß $\exp(-a\sqrt{-E})$ stets > 0 ist, ergibt sich: *Die Energie-Eigenwerte des Problems sind die reellen Lösungen der Gleichung*

$$F(E) = (V_0+2E)\sin\left(a\sqrt{E+V_0}\right) - 2\sqrt{-E(E+V_0)}\cos\left(a\sqrt{E+V_0}\right)$$

Die Nullstellensuche für beliebige a und V_0 soll mittels der Intervallschachtelungs-Methode durchgeführt werden. Diese ist im konkreten Fall der Newton-Raphson-Iteration vorzuziehen, weil man sich dadurch die Berechnung der Ableitung der Funktion $F(E)$ erspart.

Ergebnis für einen Potentialtopf mit $a = 2$ Bohr und $V_0 = 225$ Rydberg:

```
.
Es wurden 10 Energiewerte mit der rel. Genauigkeit von 0.000001
ermittelt:

     1   Energie [Ry] = -222.83185
     2   Energie [Ry] = -216.33258
     3   Energie [Ry] = -205.51910
     4   Energie [Ry] = -190.42145
     5   Energie [Ry] = -171.08820
     6   Energie [Ry] = -147.59515
     7   Energie [Ry] = -120.06418
     8   Energie [Ry] =  -88.70779
     9   Energie [Ry] =  -53.96208
    10   Energie [Ry] =  -17.15278
```

5.6 Nichtlineare Gleichungssysteme.

Probleme dieser Art sollen hier nur prinzipiell erläutert werden, weil sie sich - wie sogleich gezeigt wird - mittels des bereits ausführlich behandelten Gauss-Newton-Marquardt-Verfahrens numerisch lösen lassen.

Hat man nämlich anstatt einer einzelnen Gleichung $F(x) = 0$ ein *System von n nicht-linearen Gleichungen mit den n Unbekannten* x_1, x_2,..., x_n vorliegen, also

$$\begin{aligned} F_1(x_1, x_2, \ldots, x_n) &= 0 \\ &. \\ &. \\ F_k(x_1, x_2, \ldots, x_n) &= 0 \\ &. \\ &. \\ F_n(x_1, x_2, \ldots, x_n) &= 0 \quad , \end{aligned} \tag{5.23}$$

so werden diese Gleichungen nur dann simultan erfüllt, wenn gilt:

$$S = \sum_{k=1}^{n} [F_k(x_1, x_2, \ldots, x_n)]^2 = 0$$

In diesem Fall ist also Null das anzustrebende Minimum der Summe S, und gesucht wird jener Vektor (werden jene Vektoren) $\mathbf{x}$, der (die) dies bewerkstelligt (bewerkstelligen)!

$$S = \sum_{k=1}^{n} [F_k(\mathbf{x})]^2 \rightarrow \text{Minimum} \tag{5.24}$$

Damit ist das Problem (5.23) auf ein *spezielles nicht-lineares Least-Squares-Problem zurückgeführt*, und man kann das Gauss-Newton-Marquardt-Verfahren (s. Kap. 4) anwenden:

- Linearisierung des Modells durch Taylorreihen-Entwicklungen der Funktionen $F_k(\mathbf{x})$ an der Stelle $\mathbf{x} = \mathbf{x}^o$ ('Startvektor'):

$$F_k(\mathbf{x}) = F_k(\mathbf{x}^o) + \sum_{j=1}^{n} \left(\frac{\partial F_k(\mathbf{x})}{\partial x_j} \right)_{\mathbf{x}^o} \cdot (x_j - x_j^o) + \cdots \quad . \tag{5.25}$$

  Im weiteren werden wieder die Abkürzungen

$$dF_{k,j} \equiv \left(\frac{\partial F_k(\mathbf{x})}{\partial x_j} \right)_{\mathbf{x}^o} \qquad \text{und} \qquad F_k \equiv F_k(\mathbf{x}^o)$$

  verwendet.

- Einsetzen der Entwicklungen in (5.24) und partielle Ableitung der Summe nach den gesuchten Parametern x_1 bis x_n:

$$S \approx \sum_{k=1}^{n} \left(F_k + \sum_{j=1}^{n} dF_{k,j} \cdot (x_j - x_j^o) \right)^2$$

$$\frac{\partial S}{\partial x_l} = 2 \sum_{k=1}^{n} \left(F_k + \sum_{j=1}^{n} dF_{k,j} \cdot (x_j - x_j^o) \right) \cdot F_{k,l} = 0$$

  für alle $l = 1, \ldots, n$.

- Daraus resultiert das folgende lineare Gleichungssystem für eine *iterative Verbesserung* der Lösungen:

$$A \cdot (\mathbf{x} - \mathbf{x}^o) = \beta \qquad \text{mit}$$

$$A = [\alpha_{ij}] \qquad \alpha_{ij} = \sum_{k=1}^{n} dF_{k,i} \cdot dF_{k,j} \qquad \text{und} \qquad \beta_i = -\sum_{k=1}^{n} F_k \cdot dF_{k,i} \tag{5.26}$$

  Vergleicht man nun das System (5.26) mit dem System (4.20), so ist die nahe Verwandtschaft sofort evident. Auch zur Lösung von (5.26) ist es sehr zu empfehlen, die 'Marquardt-Variation' zu verwenden, um eine hohe Konvergenz-Sicherheit der Iteration zu erhalten.

5.6.1 Ein Testbeispiel.

Man löse das Gleichungssystem[1]

$$x^3 - 3xy^2 - 1 = 0 \qquad y^3 - 3x^2y = 0$$

nach der im Kap. 5.6. angegebenen Methode.

Exakte Lösungen:
$(x = 1,\, y = 0)$ $\quad$ $(x = -1/2,\, y = \sqrt{3}/2)$ $\quad$ $(x = -1/2,\, y = -\sqrt{3}/2)$

Numerische Lösung mittels der Gauss-Newton-Marquardt-Methode:

```
.
abs. Gen. der Fitwerte = .100000E-06

tmax =    50

guessed values:   x(1) = -.2000000E+00
                  x(2) = -.5000000E+00

  0    .740389E+00     -.200000E+00     -.500000E+00
*
*
*
*
*
  1    .100543E+00     -.595187E+00     -.823118E+00
  2    .374490E-02     -.513359E+00     -.850427E+00
  3    .478015E-05     -.500104E+00     -.865304E+00
  4    .227994E-09     -.500000E+00     -.866020E+00
  5    .543610E-15     -.500000E+00     -.866025E+00
  6    .543610E-15     -.500000E+00     -.866025E+00

Ergebnisse mittels MRQMIN und MRQCOF:
=====================================
  x(1) = -.50000E+00      (exakt: -0.5000000)
  x(2) = -.86603E+00      (exakt: -0.8660254)
```

Anmerkung: Wie bei iterativen Methoden in der Anwendung auf nicht-lineare Probleme üblich, hängt es von den Startwerten des GNM-Prozesses ab, welche der drei Lösungen man erhält. Mit den oben gewählten Startwerten bekommt man offenbar die dritte Lösung.

Die erste Lösung würde man z.B. für die Startwerte $x(1)$=–1.0 und $x(2)$=0.0 erhalten, die zweite Lösung für die Startwerte $x(1)$=–1.0 und $x(2)$=1.0.

[1] aus: F. Stummel, K. Hainer, *Praktische Mathematik*, Teubner Studienbücher, Stuttgart 1971.

5.7 Software-Angebot

TOMS (www.netlib.org) bietet eine Implementierung des Newton-Verfahrens in Verbindung mit Intervallschachtelung unter TOMS-681 sowie
unter TOMS-378 eine Programmbeschreibung in ALGOL: *Discretized Newton-like method for solving a system of simultaneous nonlinear equations.*

NAG: *c05ajf* und *c05axf*
verwenden die *Regula falsi* zur Lösung nicht-linearer Gleichungen.

Numerical Recipes: Abgesehen vom Newton-Raphson-Programm 'rtnewt', das im Abschnitt 5.3.2 dieses Skriptums vorgestellt wird, bieten die NumRec-Bücher eine Reihe anderer Routinen zur Auffindung reeller Nullstellen reellwertiger Funktionen: hervorzuheben ist hier die Routine 'zbrent', welche auf Wijngaarden, Dekker und Brent zurückgeht (1960). Für die iterative Annäherung an komplexwertige Nullstellen von Polynomen mit komplexwertigen Koeffizienten sollten Sie das Programm 'laguer' probieren (Methode von Laguerre).

Numerical Algorithms with C or FORTRAN: Im Kapitel 2 dieses Buches (bzw. im entsprechenden Unterverzeichnis des Linux-PC) finden Sie die folgenden Programme:

```
fnewton.c    newpsz.f90    Newtonverfahren fuer reelle Funktionen
fpegasus.c   pegasu.f90    Pegasusverfahren fuer reelle Funktionen
froots.c                   Pegasus-, Pegasus-King-, Anderson-Bjoerck-
                           und Anderson-Bjoerck-King-Verfahren
fzeroin.c    zeroin.f90    Zeroin-Verfahren fuer reelle Funktionen

fmueller.c   muller.f90    Verfahren von Mueller zur Berechnung aller
                           Nullstellen eines reellen Polynoms
fbauhube.c   baupol.f90    Verfahren von Bauhuber zur Berechnung aller
                           Nullstellen eines komplexen Polynoms
flaguer.c    laguer.f90    Verfahren von Laguerre zur Berechnung aller
                           reellen Nullstellen eines reellen Polynoms
```

MATLAB bietet neben anderen Routinen die Programme

```
roots.m        Berechnung der Nullstellen eines Polynoms,
               wobei das Funktionsargument die Koeffizienten
               des Polynmos sind.


fzero.m        Berechnung der Nullstelle einer Funktion F(x)
               in der Naehe eines Anfangswertes x0.
```

Kapitel 6

Numerische Integration

In diesem Kapitel geht es um die *numerische Auswertung von bestimmten Integralen*

$$I(f) = \int_c^d dx f(x) \quad .$$

Numerische Methoden kommen dann zur Anwendung, wenn

- der Integrand nur punktweise (z.B. in Tabellenform) gegeben ist. In einem solchen Fall empfiehlt es sich, durch die gegebenen Punkte eine *Interpolationskurve* zu legen und diese dann stückweise zu integrieren (s. Abschnitt 6.1).
- der Integrand zwar als Funktion gegeben ist, das Integral jedoch nicht analytisch integrierbar ist, wie z.B.

$$\int_c^d dx \frac{\sin x}{x} \quad .$$

- das Integral zwar im Prinzip analytisch integrierbar ist, die analytische Auswertung jedoch unverhältnismäßig viel Rechenaufwand erfordern würde [1].

In den beiden letzten Fällen werden fast ausschließlich sogenannte *Quadraturformeln* eingesetzt, von denen ab dem Abschnitt 6.2 ausführlich die Rede sein wird.

6.1 Numerische Integration punktweise gegebener Integranden.

Angenommen, eine zu integrierende Funktion $f(x)$ sei nur bzgl. der n Stützpunkte

$$[x_i \mid f(x_i)] \qquad i = 1, \ldots, n$$

[1] Hier soll jedoch ausdrücklich auf die immer populärer werdende Computer-Software hingewiesen werden, die auch analytische Rechenoperationen erlaubt, wie z.B. die Systeme REDUCE, MAPLE, Mathematica usw.

bekannt, wobei der Einfachheit halber angenommen werden soll, daß der erste und letzte Stützpunkt, x_1 und x_n, gleichzeitig die Integrationsgrenzen sind. In einem solchen Fall kann $f(x)$ durch die entsprechende Interpolationskurve (z.B. eine kubische Spline-Kurve) $I(x)$ ersetzt werden:

$$\int_{x_1}^{x_n} dx f(x) \approx \int_{x_1}^{x_n} dx I(x) = \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} dx P^i(x)$$

$$= \sum_{i=1}^{n-1} \int_{x_i}^{x_{i+1}} dx \left[a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \right]$$

Diese Integrale können leicht analytisch ausgewertet werden, und man erhält:

$$\int_{x_1}^{x_n} dx f(x) \approx \sum_{i=1}^{n-1} \left[a_i(x_{i+1} - x_i) + \frac{b_i}{2}(x_{i+1} - x_i)^2 + \frac{c_i}{3}(x_{i+1} - x_i)^3 + \frac{d_i}{4}(x_{i+1} - x_i)^4 \right] \quad . \tag{6.1}$$

6.2 Verwendung von Quadraturformeln.

Für alle weiteren Ausführungen soll gelten: die (reellwertige) Integrandenfunktion $f(x)$ ist analytisch gegeben und im gesamten Integrationsintervall $[c; d]$ stetig.

6.2.1 Das grundsätzliche Problem

besteht in der näherungsweisen Berechnung von bestimmten Integralen mit Hilfe von sogenannten Quadraturformeln:

$$I(f) = \int_c^d dx f(x) \quad \rightarrow \quad Q_m(f) = \frac{(d-c)}{2} \sum_{j=1}^{m} g_j f(x_j) \quad . \tag{6.2}$$

Die rechte Seite von (6.2), die Quadraturformel $Q_m(f)$, ist in der Lage, den Integralwert $I(f)$ *mit jeder gewünschten Genauigkeit* [2] zu approximieren, wenn man

m**:** den Grad der Quadraturformel

$x_1, \ldots, x_m$**:** die Stützpunkte im Intervall $[c, d]$ sowie

$g_1, \ldots, g_m$**:** die Gewichts-Koeffizienten

geeignet wählt.

Die Differenz

$$E_V(f) = I(f) - Q_m(f)$$

nennt man den Verfahrensfehler der Quadratur.

Während der Grad der Quadraturformel, d.h. die Anzahl m der Summenterme, im Prinzip frei wählbar ist, müssen die Stützpunkte und die Gewichtsfaktoren möglichst optimal gewählt werden. Im weiteren soll nun eine Optimierungsvorschrift für diese Größen angegeben werden.

[2] unter Vernachlässigung von Rundungsfehlern

6.2.2 Eine Optimierungsvorschrift für die x_i und g_i.

Für die Quadraturformel (6.2) m-ten Grades gibt es offenbar $2m$ zu optimierende Parameter. Bei zahlreichen Quadraturformeln wird jedoch diese Anzahl noch durch zusätzliche *einschränkende Bedingungen* vermindert. Im allgemeinen sind daher nur

$$m' \leq 2m$$

Parameter zu optimieren.

Man geht nun von einem allgemeinen Polynom vom Grade $m' - 1$ aus:

$$P_{m'-1}(x) = a_1 + a_2 x + a_3 x^2 + \cdots + a_{m'} x^{m'-1} \tag{6.3}$$

Es sollen nun die Parameter der Quadraturformel so gewählt werden, daß für alle Polynome vom Grade $m' - 1$ oder darunter *kein Verfahrensfehler auftritt*, d.h. daß gilt:

$$\int_c^d dx P_{m'-1}(x) \stackrel{!}{=} Q_m(P_{m'-1}) \tag{6.4}$$

Die Gleichung (6.4) stellt somit eine *Optimierungsvorschrift* für die zu optimierenden Parameter dar.

6.2.3 Die Trapezformel

ist eine Quadraturformel zweiten Grades, d.h.

$$Q^T(f) = \frac{(d-c)}{2}\left[g_1 f(x_1) + g_2 f(x_2)\right] \quad ,$$

wobei als Nebenbedingung $x_1 = c$ und $x_2 = d$ gilt. Es bleiben somit noch $m' = 4 - 2 = 2$ Parameter zu optimieren, nämlich die Gewichtfaktoren g_1 und g_2. Durch Anwendung der Vorschrift (6.4) erhält man

$$\int_c^d dx(a_1 + a_2 x) = \frac{(d-c)}{2}\left[g_1(a_1 + a_2 c) + g_2(a_1 + a_2 d)\right] \quad .$$

Das exakte Integral ist leicht auszuwerten, und durch Koeffizientenvergleich bzgl. a_1 und a_2 erhält man:

$$g_1 = 1 \qquad \text{und} \qquad g_2 = 1 \quad .$$

Die Trapezformel lautet somit

$$Q^T(f) = \frac{(d-c)}{2}\left[f(c) + f(d)\right] \quad . \tag{6.5}$$

Der Name 'Trapezformel' hat geometrische Gründe (s. Abb. 6.1).

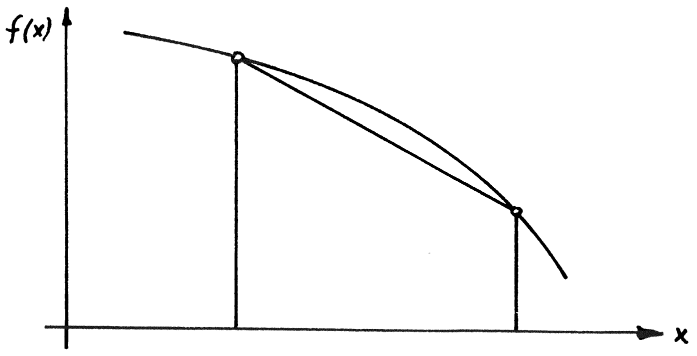


Abbildung 6.1: Anwendung der Trapezformel.

6.2.4 Die Simpson-Formel

ist eine Quadraturformel dritten Grades, d.h.

$$Q^S(f) = \frac{(d-c)}{2}\left[g_1 f(x_1) + g_2 f(x_2) + g_3 f(x_3)\right] \quad .$$

Als Nebenbedingung wird eine *äquidistante* Stützpunktverteilung gefordert, also

$$x_1 = c \qquad x_2 = \frac{c+d}{2} \qquad x_3 = d \quad .$$

Die Optimierung der 3 Gewichtsfaktoren ergibt $g_1 = g_3 = 1/3$; $g_2 = 4/3$, was zur Simpson-Formel

$$Q^S(f) = \frac{(d-c)}{6}\left[f(c) + 4f\left(\frac{c+d}{2}\right) + f(d)\right] \tag{6.6}$$

führt.

6.2.5 Verfahrensfehler von Trapez- und Simpsonformel.

Die Verfahrensfehler der Quadraturformeln (6.5) und (6.6) können mittels Anwendung des Mittelwertsatzes der Integralrechnung bestimmt werden. Sie lauten

$$E_V^T = -\frac{(d-c)^3}{12} f''(\xi) \qquad \xi \in [c,d] \tag{6.7}$$

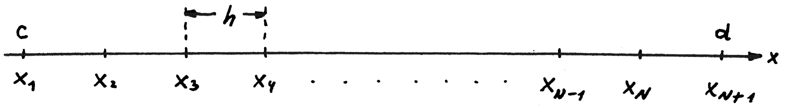
und

$$E_V^S = -\frac{(d-c)^5}{2880} f^{(4)}(\xi) \quad \xi \in [c,d] \quad . \tag{6.8}$$

Die Verfahrensfehler steigen also mit der 3. bzw. der 5. Potenz der Breite des Integrationsintervalls an. Das bedeutet natürlich, daß eine Anwendung von (6.5) bzw. (6.6) für größere Intervallbreiten zu vollkommen obsoleten Ergebnissen führt!

6.3 Die summierte Trapez- und Simpsonformel.

Die einzige Möglichkeit, mittels Trapez- und Simpsonformel zu brauchbaren Ergebnissen zu kommen, ist eine Aufteilung des gegebenen Intervalls $[c, d]$ in eine Reihe von N Subintervallen mit der Intervallbreite h :



$$h = \frac{(d-c)}{N} \qquad x_i = c + (i-1)h \qquad (i = 1, \ldots, N+1)$$

Nun wendet man die Trapezformel (6.5) auf jedes einzelne Subintervall an und summiert über die N Teilergebnisse: Dies ergibt[3]

$$Q^{ST}(f) = \sum_{i=1}^{N} \frac{h}{2} [f(x_i) + f(x_{i+1})]$$

bzw.

$$Q^{ST}(f) = h \left[\frac{1}{2} f(c) + f(x_2) + f(x_3) + \cdots + f(x_N) + \frac{1}{2} f(d) \right] \quad . \tag{6.9}$$

Ebenso kann man die Simpsonformel (6.6) auf *jeweils 2 nebeneinander liegende* Subintervalle anwenden, was voraussetzt, daß man für N eine gerade Zahl wählt.[4] Eine Summierung über die $N/2$ Teilergebnisse ergibt

$$Q^{SS}(f) = \frac{h}{3} \left[f(c) + 2 \sum_{i=1}^{N/2-1} f(x_{2i+1}) + 4 \sum_{i=1}^{N/2} f(x_{2i}) + f(d) \right]$$

bzw. ausgeschrieben:

$$Q^{SS}(f) = \frac{2h}{3} \left[\frac{1}{2} f(c) + 2f(x_2) + f(x_3) + 2f(x_4) + \ldots + f(x_{N-1}) + 2f(x_N) + \frac{1}{2} f(d) \right] \quad . \tag{6.10}$$

6.3.1 Verfahrensfehler der summierten Quadraturformeln.

Auch für die summierten Formeln (6.9) und (6.10) werden die Verfahrensfehler ohne Ableitung angegeben:

$$E_V^{ST}(f) = -\frac{(d-c)}{12} \cdot h^2 \cdot f''(\xi) \qquad \xi \in [c, d] \tag{6.11}$$

bzw.

$$E_V^{SS}(f) = -\frac{(d-c)}{180} \cdot h^4 \cdot f^{(4)}(\xi) \qquad \xi \in [c, d] \quad . \tag{6.12}$$

[3]'ST': Summierte Trapezformel, 'SS' : Summierte Simpsonformel.

[4]In der Literatur gibt es ähnliche Formeln auch für *ungerade* N, wie die sog. '3/8-Formel' (s. z. B. [2], S. 191).

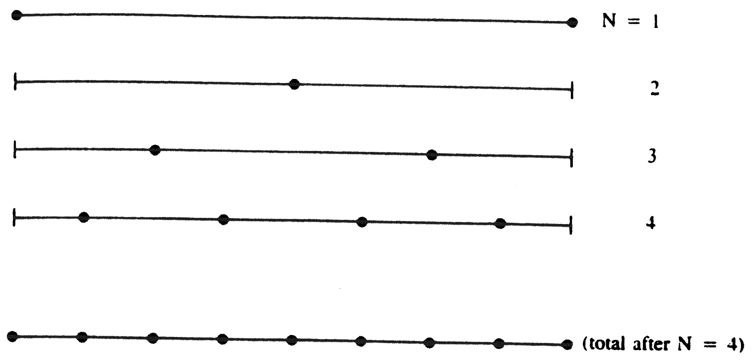


Abbildung 6.2: Erhöhung der Subintervalle bei der Anwendung der Trapezformel.

Diese Fehler sind also nur mehr linear von der Breite des Integrationsintervalls abhängig, dafür aber proportional zu h^2 bzw. h^4: Das heißt, daß durch eine immer feinere Aufteilung von $[c, d]$ die Verfahrensfehler beliebig klein gemacht werden können.

6.4 Die ökonomische Auswertung der Trapezformel.

Im allgemeinen wird man keine Information darüber haben, wieviele Subintervalle in die Rechnung einbezogen werden müssen, um ein numerisches Ergebnis der gewünschten Genauigkeit zu erhalten (eine a-priori Fehlerabschätzung auf Basis von (6.11) ist wegen der Unkenntnis von $f''(\xi)$ nicht möglich). Man muß sich also an das gewünschte Ergebnis 'herantasten', wobei man die Stützpunktanzahl sukzessive erhöht. Am günstigsten ist es, die Stützpunktanzahl immer wieder zu verdoppeln (s. Abb. 6.2).
Man geht dabei von der simpelsten Näherung aus, nämlich von zwei Stützpunkten ($M = 1$ Intervall), und man erhält

$$Q^{ST}_{M=1} = (d - c)\left[\frac{1}{2}f(c) + \frac{1}{2}f(d)\right] \qquad \text{M = Subintervall-Anzahl.}$$

Eine Verdoppelung der Intervalle ($M = 2$) erfordert die Berechnung eines weiteren Funktionswertes mit dem Argument $(c + d)/2$, und es ergibt sich

$$Q^{ST}_{M=2} = \frac{(d-c)}{2}\left[\frac{1}{2}f(c) + f\left(\frac{c+d}{2}\right) + \frac{1}{2}f(d)\right] = \frac{1}{2}\left[Q^{ST}_{M=1} + (d-c)f\left(\frac{c+d}{2}\right)\right] \quad .$$

Eine weitere Verdoppelung auf 4 Subintervalle erfordert lt. Abb. 6.2 die Berechnung von $f(c+(d-c)/4)$ und $f(c+3(d-c)/4)$:

$$Q_{M=4}^{ST} = \frac{1}{2}\left\{Q_{M=2}^{ST} + \frac{(d-c)}{2}\left[f\left(c+\frac{(d-c)}{4}\right) + f\left(c+\frac{3(d-c)}{4}\right)\right]\right\} \quad .$$

Offensichtlich gilt bei jeder Verdoppelung der Subintervalle die <u>Rekursionsformel</u>

$$Q_{2M}^{ST} = \frac{1}{2}\left[Q_M^{ST} + \frac{(d-c)}{M}\sum_{j=1(2)}^{2M-1} f\left(c+j\frac{(d-c)}{2M}\right)\right] \quad . \tag{6.13}$$

(6.13) bildet die Basis für die Programme QTRAP und TRAPZD im Abschnitt 6.5.

6.4.1 Genauigkeitsabfragen

Durch die fortgesetzte Anwendung der Formel (6.13) erhält man also eine Folge von Näherungswerten für ein bestimmtes Integral nach Maßgabe der summierten Trapezformel:

$$Q_1^{ST} \quad ; \quad Q_2^{ST} \quad ; \quad Q_4^{ST} \quad ; \quad Q_8^{ST} \quad \ldots ,$$

wobei der untere Index die Zahl der Subintervalle in [c,d] darstellt.
Nun stellen sich die Fragen: *'Wie genau sind die erhaltenen numerischen Werte?'* bzw. *'Wieviele Subintervall-Verdoppelungen sind nötig, um eine bestimmte vorgegebene (absolute oder relative) Genauigkeit zu erreichen?'*.

Das Problem einer solchen *internen Fehlerdiagnose* liegt darin, daß während des Programmablaufes eine möglichst zuverlässige Fehlerabschätzung, basierend auf dem vorliegenden Datenmaterial, erfolgen muß. Dabei kann man z. B. wie folgt vorgehen:

Wenn man die Rundungsfehler im Moment außer Acht läßt, kann man unter Verwendung von Glg. (6.11) und unter Berücksichtigung von $h \propto 1/M$ schreiben:

$$\begin{aligned} I_{exakt} &= Q_M^{ST} + C(M)/M^2 \\ &\qquad . \\ I_{exakt} &= Q_{2M}^{ST} + C(2M)/(2M)^2 \, , \end{aligned} \tag{6.14}$$

wobei $C(M)$ und $C(2M)$ die in (6.11) enthaltene Konstante für die Zahl der Subintervalle M und $2M$ ist.
Nimmt man nun an, daß näherungsweise

$$C(M) \approx C(2M) = C \tag{6.15}$$

gilt (was durchaus nicht immer der Fall ist!), so stellen die Gleichungen (6.14) ein lineares Gleichungssystem für die Unbekannten I_{exakt} und C dar. Für C erhält man nach kurzer Rechnung

$$\frac{(2M)^2\,(Q_{2M}^{ST} - Q_M^{ST})}{3}$$

bzw. für den Verfahrensfehler des numerischen Bestwertes Q_{2M}^{ST}

$$E_V[Q_{2M}^{ST}] \approx \frac{Q_{2M}^{ST} - Q_M^{ST}}{3} \,. \tag{6.16}$$

Daraus kann man Abbruchbedingungen für ein auf (6.13) beruhendes Programm gewinnen:

Die Rekursion (6.13) wird solange fortgesetzt, bis der vorgegebene absolute oder relative Fehler GEN unterschritten wird, also bis gilt:

$$\frac{|Q_{2M}^{ST} - Q_M^{ST}|}{3} < GEN \qquad \text{oder} \qquad \frac{|Q_{2M}^{ST} - Q_M^{ST}|}{3|Q_{2M}^{ST}|} < GEN \,. \tag{6.17}$$

Beachten Sie bitte, daß die relative Fehlerabfrage versagt, wenn das numerische Ergebnis gegen Null geht. Es ist auch nochmals darauf hinzuweisen, daß die einfachen Abfragen (6.17) nur dann funktionieren, wenn (1) die Näherung (6.15) erfüllt ist und wenn (2) der Verfahrensfehler den Rundungsfehler dominiert.

6.5 Die Programme QTRAP und TRAPZD.

Quelle: [9], S. 111ff mit Änderungen.

INPUT-Parameter:

C,D: Grenzen des Integrationsintervalls.

JMAX: maximale Anzahl von Aufrufen von TRAPZD.

IREL: IREL=1 relative Fehlerabfrage, sonst absolute Fehlerabfrage.

GEN: relative oder absolute Genauigkeit.

OUTPUT-Parameter:

S: Näherungswert für das bestimmte Integral.

FEHLER: Fehlerdiagnostik: FEHLER = FALSE Berechnung ok
FEHLER = TRUE Genauigkeit wurde nicht erreicht.

Struktogramm 16 — QTRAP(C,D,JMAX,IREL,GEN,S,FEHLER)

```
FEHLER:= TRUE
JMAX < 2
  Y: JMAX:=2
  N: ......
TRAPZD(C,D,S,1)
OLDS:=S
J:=2
    TRAPZD(C,D,S,J)
    ERROR:=| S-OLDS |/3
    IREL=1
      Y: ERROR:=ERROR/| S |
      N: ......
    ERROR < GEN
      Y: FEHLER:=FALSE
      N: OLDS:=S
    J:=J+1
J > JMAX .or. notFEHLER
J > JMAX
  Y: print: 'QTRAP Gen. nicht erreicht.'
  N: ......
(return)
```

Struktogramm 17 — TRAPZD(C,D,S,J)

```
J=1
  Y: S:=0.5*(D-C)*(FUNC(C)+FUNC(D))
  N: TNM:=2**(J-2)
     DEL:=(D-C)/TNM
     X:=C + 0.5*DEL
     SUM:=0.0
     I=1(1)TNM
         SUM:=SUM + FUNC(X)
         X:=X+DEL
     S:=0.5*(S+(D-C)*SUM/TNM)
(return)
```

Anmerkung: Die Routine TRAPZD muß hintereinander für J=1,2,3,... aufgerufen werden.

Der folgende einfache Test soll Ihnen das korrekte Funktionieren der Programme QTRAP und TRAPZD dokumentieren:

$$I_{exakt} = \int_0^1 dx\, \frac{\ln(1+x)}{x(1+x)} = \frac{\pi^2}{12} - \frac{\ln(2)^2}{2} = 0.582240\ldots$$

Fordert man vom Programm QTRAP (F90-DOUBLE PRECISION) eine absolute Genauigkeit GEN=0.000001, so erhält man

```
TEST FUER DIE ROUTINEN   QTRAP  UND  TRAPZD
NUM. METHODEN IN DER PHYSIK  WS 2000/2001       28-8-2000
H. Sormann  F90  DOUBLE PRECISION

   J       Funkt.   num. Erg.      wahrer       geschaetzter       GEN
          aufrufe                abs. Fehler    abs. Fehler

   2          3     0.60695347   0.247129E-01   0.221111E-01   0.100000E-05
   3          5     0.58858506   0.634453E-02   0.612280E-02   0.100000E-05
   4          9     0.58383827   0.159774E-02   0.158226E-02   0.100000E-05
   5         17     0.58264071   0.400184E-03   0.399185E-03   0.100000E-05
   6         33     0.58234062   0.100093E-03   0.100030E-03   0.100000E-05
   7         65     0.58226555   0.250263E-04   0.250223E-04   0.100000E-05
   8        129     0.58224678   0.625675E-05   0.625651E-05   0.100000E-05
   9        257     0.58224209   0.156420E-05   0.156418E-05   0.100000E-05
  10        513     0.58224092   0.391051E-06   0.391050E-06   0.100000E-05

EXAKT =      0.5822405265
Ergebnis =  0.5822409175
geforderte Genauigkeit (abs) =  0.000001000
Dazu waren      513  Funktionsaufrufe noetig.
```

6.6 Das Romberg-Verfahren.

Wie aus den Gleichungen (6.11) und (6.12) für die Verfahrensfehler der summierten Trapezformel (Ordnung m=2) und der summierten Simpsonformel (Ordnung m=3) hervorgeht, sind diese gegeben durch

$$E_V^{(m)} \quad \text{proportional} \quad h^{2m-2} \quad ,$$

wobei m die Ordnung der Quadraturformel bedeutet. Wegen $h \propto 1/M$ (M = Anzahl der Subintervalle) gilt also:

$$E_V^{(m)}(M) = \frac{C(M)}{M^{2m-2}} \quad . \tag{6.18}$$

Man geht nun genauso vor wie im Abschnitt 6.4.1, d.h. man stellt den (unbekannten) exakten Integralwert durch die Gleichungen

$$I_{exakt} = Q_M^{(m)} + \frac{C(M)}{M^{2m-2}}$$

bzw.

$$I_{exakt} = Q_{2M}^{(m)} + \frac{C(2M)}{(2M)^{2m-2}}$$

dar, wobei die Konstante C als näherungsweise unabhängig von M angenommen wird, also

$$C(M) \approx C(2M) = C \quad .$$

Unter diesen Umständen stellen die obigen Gleichungen ein lineares System für C und I_{exakt} dar, wobei die Lösung

$$I_{exakt} = \frac{1}{4^{m-1} - 1} \left(4^{m-1} Q_{2M}^{(m)} - Q_M^{(m)} \right)$$

lautet. Dabei ist 'I_{exakt}' wegen der Approximation von C natürlich nicht wirklich die exakte Integrallösung, sondern nur eine Verbesserung.

Man kann zeigen:
'I_{exakt}' entspricht dem Ergebnis einer Quadraturformel mit $2M$ Subintervallen und *einer um Eins höheren Ordnung:*

$$Q_{2M}^{(m+1)} = \frac{1}{4^{m-1} - 1} \left(4^{m-1} Q_{2M}^{(m)} - Q_M^{(m)} \right) \quad . \tag{6.19}$$

Verifikation der Glg. (6.19) durch ein Beispiel:
Geht man von der Trapezformel ($m = 2$) sowie von zwei Subintervallen ($M = 2$) aus, so erhält man

$$Q_2^{(2)} = \frac{(d-c)}{2} \left[\frac{1}{2} f(c) + f\left(\frac{c+d}{2} \right) + \frac{1}{2} f(d) \right] \quad .$$

Eine Verdoppelung auf vier Subintervalle ergibt

$$Q_4^{(2)} = \frac{(d-c)}{4} \left[\frac{1}{2} f(c) + f\left(c + \frac{(d-c)}{4} \right) + f\left(\frac{c+d}{2} \right) + f\left(c + 3\frac{(d-c)}{4} \right) + \frac{1}{2} f(d) \right] \quad .$$

Setzt man die beiden letzten Gleichungen in die Formel (6.19) ein, so ergibt sich

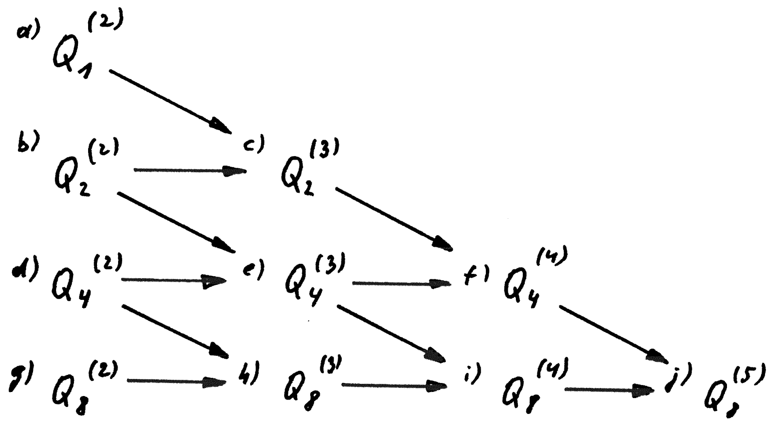
$$Q_4^{(3)} = \frac{(d-c)}{6} \left[\frac{1}{2} f(c) + 2f\left(c + \frac{(d-c)}{4} \right) + f\left(\frac{c+d}{2} \right) + 2f\left(c + 3\frac{(d-c)}{4} \right) + \frac{1}{2} f(d) \right] \quad ,$$

was exakt dem Ergebnis der summierten Simpsonformel (Quadraturformel der Ordnung 3) für 4 Subintervalle entspricht:

$$Q_4^{(3)} \equiv Q_4^{SS}$$

Man erhält also durch Anwendung der Formel (6.19) auf 2 Trapezformeln genau die Simpsonformel.

Die Strategie von Romberg, die auf der Gleichung (6.19) beruht, ist mit Hilfe des folgenden Schemas leicht zu verstehen:



Die Kleinbuchstaben links neben den Q-Symbolen bezeichnen die Reihenfolge der Berechnung, und die zusammenlaufenden Pfeile bedeuten die Anwendung von (6.19). D.h., von Zeile zu Zeile verdoppelt sich die Anzahl der Subintervalle, von Spalte zu Spalte erhöht sich die Ordnung der Quadraturformel.
Es ist nun plausibel anzunehmen, daß die Diagonalelemente des obigen Schemas schneller zum gewünschten Integralwert konvergieren als die Zeilen- und Spaltenelemente, weil ja entlang der Diagonale sowohl die Subintervall-Zahl als auch die Ordnung der Quadratur erhöht werden.
Man kann in der Tat zeigen, daß für viele praktische Anwendungen der Romberg-Strategie

$$|Q_{2M}^{(m)} - Q_M^{(m-1)}| \approx |Q_M^{(m-1)} - I_{exakt}| \tag{6.20}$$

gilt, was bedeutet, daß

die Differenz zwischen zwei aufeinanderfolgenden Diagonalelementen des Romberg-Schemas eine gute Näherung für den absoluten Fehler des Elementes $Q_M^{(m-1)}$ darstellt.

Diese Fehlerabschätzung hat aber eine unschöne Seite: man muß offensichtlich, um die Genauigkeit des Diagonalelementes des Zeile M bestimmen zu können, die gesamte folgende Zeile $2M$ berechnen!

6.6.1 Das Programm ROMB.

Quellen: [9], S. 114f, [2], S. 205ff und 407f, mit Änderungen.

Das Programm ROMB berechnet einen Näherungswert für ein bestimmtes Integral.

INPUT-Parameter:

C,D: Grenzen des Integrationsintervalls.

KMIN: Ordnung des Romberg-Prozesses, *ab welchem die Fehlerdiagnose beginnt.* Diskussion siehe S. 178f.

KMAX: maximale Ordnung des Romberg-Prozesses (≥ 3).

IREL: IREL=1 relative Fehlerabfrage, ansonsten absolute Fehlerabfrage.

GEN: relativer oder absoluter zu unterschreitender Fehler.

OUTPUT-Parameter:

S: Näherungswert für das bestimmte Integral.

ERROR: Abschätzung für den absoluten bzw. relativen Fehler dieser Näherung.

FEHLER: Fehlerdiagnostik: FEHLER = FALSE Berechnung ok
FEHLER = TRUE Genauigkeit
wurde nicht erreicht.

Das Programm ROMB benötigt die bereits besprochene Routine TRAPZD.

Anmerkung: Was die Leistungsfähigkeit des Romberg-Verfahrens betrifft, so ist (frei zitiert nach [9], S. 115) ... die Romberg-Routine sehr leistungsfähig für genügend glatte Integranden ... Unter solchen Umständen braucht das Programm ROMB *viel, viel weniger* Funktionsaufrufe als jedes andere Programm, das auf der Trapez- oder der Simpsonformel beruht.

Struktogramm 18: — ROMB(C,D,KMIN,KMAX,IREL,GEN,S,ERROR,FEHLER)

```
FEHLER:=TRUE
KMAX < 3
  Y: KMAX:=3
  N: ......
TRAPZD(C,D,Q(2),1)
K:=3
    M=2(1)K-1
        QALT(M):=Q(M)
    TRAPZD(C,D,Q(2),K-1)
    FAC:=1
    M=2(1)K-1
        FAC:=FAC*K
        Q(M+1):=(FAC*Q(M)-QALT(M))/(FAC-1)
    ERROR:=| QALT(K-1)-Q(K) |
    IREL = 1
      Y: ERROR:=ERROR/| QALT(K-1)|
      N: ......
    K > KMIN
      Y: ERROR < GEN
           Y: S:=QALT(K-1)
              FEHLER:=FALSE
           N: ......
      N: ......
    K:=K+1
K > KMAX .or. notFEHLER
K > KMAX
  Y: print: 'ROMB Gen. nicht erreicht.'
  N: ......
(return)
```

Das folgende Output-Protokoll zeigt Ihnen an Hand des Beispiels von S. 169 die Überlegenheit des Romberg-Algorithmus gegenüber der Trapezmethode:

```
NUM. METH. IN DER PHYSIK  WS 2000/2001  ROMBERG-FORMEL  28-8-2000
INTEGRAL  Skriptum S. 169
H. Sormann  F90  DOUBLE PRECISION

m      Funkt.    num. Erg.      wahrer     geschaetzter     absGEN
      aufrufe               abs. Fehler   abs. Fehler

2        3      0.67328680  0.910463E-01  0.884444E-01  0.100000E-05
3        5      0.58484236  0.260183E-02  0.253878E-02  0.100000E-05
4        9      0.58230358  0.630557E-04  0.623029E-04  0.100000E-05
5       17      0.58224128  0.752829E-06  0.749125E-06  0.100000E-05

EXAKT =     0.5822405265
Ergebnis =  0.5822412793
geforderte Gen. (abs) = 0.100000E-05
Dazu waren       17 Funktionsaufrufe noetig.
```

6.6.2 Romberg-Verfahren: Tests und Anmerkungen

Das eben erläuterte Verfahren wird im folgenden auf Integrale vom Typus

$$I_n = \int_1^9 dx \, \cos\left[\frac{n\pi}{2}x\right] \mathrm{e}^{-0.2x} \qquad n = 1, 2, \ldots \tag{6.21}$$

angewendet[5]. Es ist unmittelbar klar, daß die numerische Auswertung solcher Integrale mit größerwerdendem n schwieriger wird, da die Oszillation der Integrandenfunktion immer intensiver wird. Zur Demonstration zeigt die Abb. 6.3 die Integrandenfunktion für $n = 20$. Es ist zu erwarten, daß in solchen Fällen für die numerische Integration eine erhebliche Zahl von Stützpunkten erforderlich ist.

In den folgenden Tests geht es hauptsächlich um die Zuverlässigkeit der auf Glg. (6.20) basierenden Fehlerdiagnostik im Romberg-Programm. Das Verhalten des *absoluten* Fehlers ist in den *oberen* Diagrammen der Abbildungen 6.4 (für $n = 2$) und 6.5 (für $n = 100$) dargestellt.

Sowohl für den schwach ($n = 2$) als auch für den stark oszillierenden Integranden ($n = 100$) ist im Bereich $5 \leq m \leq 16$ eine sehr gute Übereinstimmung zwischen dem 'wahren' Fehler und dem vom Programm geschätzten Fehler zu beobachten. Die Kurven liegen praktisch aufeinander, und die Fehler zwischen numerischer Näherung und exakter Lösung nehmen mit fortschreitendem Romberg-Prozess (=mehr Stützpunkte, höhere Ordnung der Quadraturformel) monoton ab. Auf diese Weise kann der absolute Verfahrensfehler bis auf $\approx 10^{-16}$ reduziert werden. Natürlich wird dieser Wert für den schwach oszillierenden Integranden bedeutend früher erreicht ($m = 11$)

[5] Alle Ergebnisse dieses Abschnittes wurden mit einem F90-Programm, DOUBLE-PRECISION, erhalten.

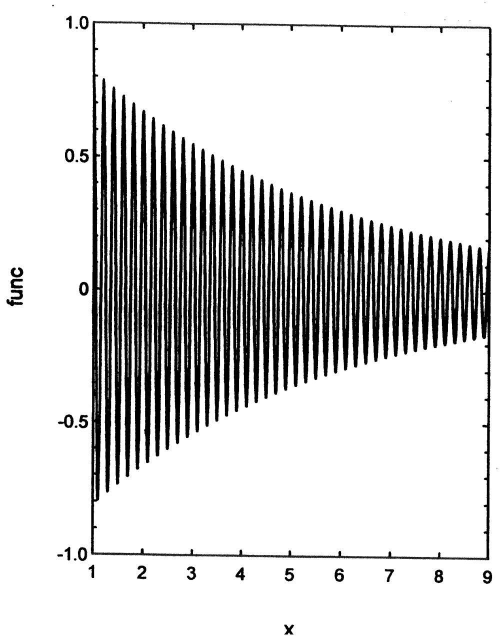


Abbildung 6.3: Die Integrandenfunktion von (6.21) für $n = 20$.

als für den stark oszillierenden Integranden ($m = 16$).
Ab $m = 11$ (für $n = 2$) bzw. $m = 16$ (für $n = 100$) kann der Fehler des numerischen Ergebnisses nicht mehr weiter reduziert werden, er nimmt im Gegenteil sogar leicht zu! Die Ursache dafür sind natürlich *Rundungsfehler*, die mit Zunahme der Stützpunktzahl größer werden und bei einem gewissen $m = 10$ den Verfahrensfehler übersteigen.

Man muß also beim Numerischen Integrieren (und nicht nur dort) zur Kenntnis nehmen:
Auf Grund der Rundungsfehler kann die Genauigkeit einer numerischen Integration nur bis zu einer gewissen Grenze gesteigert werden. Ab dieser Grenze bringt eine weitere Erhöhung des Rechenaufwandes keine Verbesserung der Ergebnisse, sondern oft sogar eine Verschlechterung.

Man sieht auch, daß der numerisch ermittelte Fehler ab $m = 16$ plötzlich auf Null zurückgeht. Überlegen Sie sich die Ursache für dieses Verhalten.

Interessant ist das Verhalten des absoluten Fehlers für kleine Werte von m: in diesem Bereich weicht die Fehlerabschätzung (6.20) stark vom 'wahren' Fehler ab. Der Grund dafür liegt darin, daß in diesem Bereich die Stützpunktdichte viel zu klein ist, um die (oszillierende) Integrandenfunktion numerisch korrekt beschreiben zu können.

Zusammenfassend kann jedoch gesagt werden, daß die Abschätzung des *absoluten* Fehlers im Romberg-Programm sehr gut funktioniert.

Viel problematischer ist die Fehlerdiagnostik für den *relativen* Fehler, der an und für sich (s. Kapitel 1) viel mehr interessiert als der absolute Fehler. Das Dilemma wird in den *unteren* Diagrammen der Abbildungen 6.4 und 6.5 demonstriert:

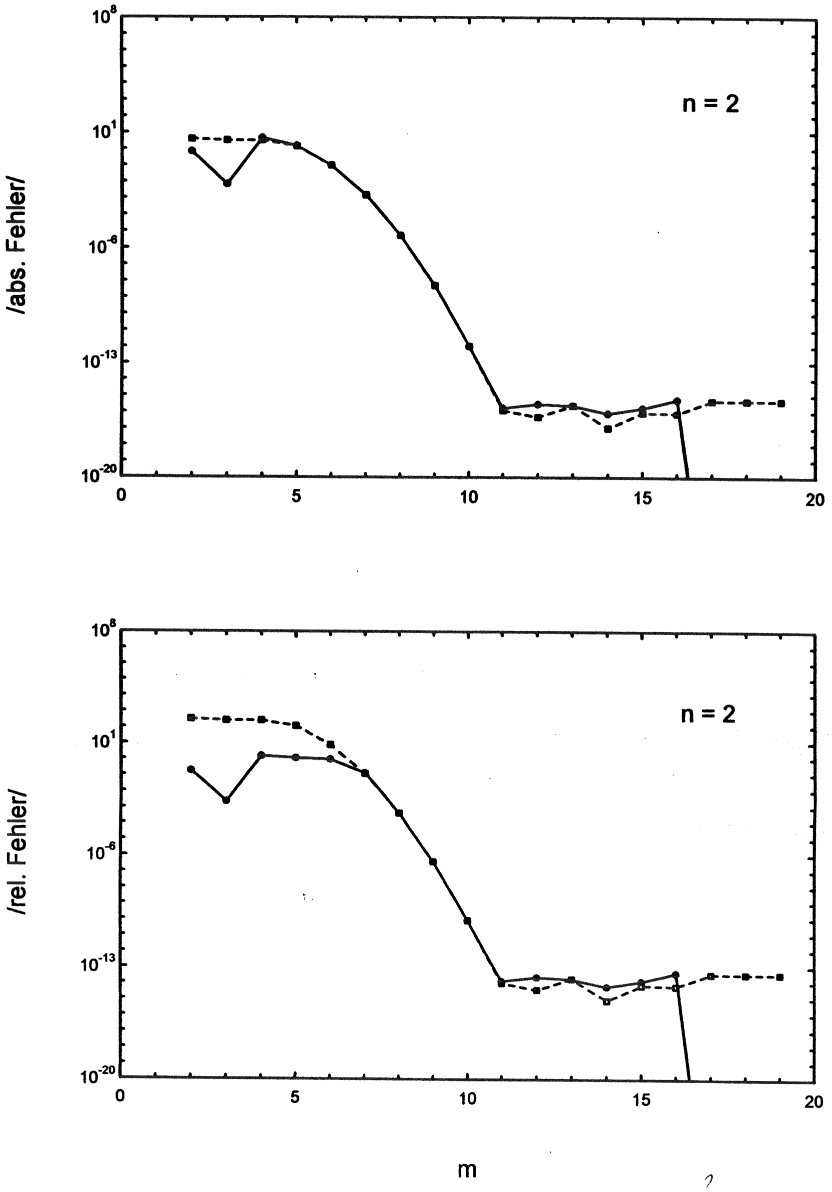


Abbildung 6.4: Romberg-Tests für das Integral (6.21), $n = 2$. Dargestellt ist der *absolute* (oben) und der *relative* Fehler (unten) der 'Romberg-Näherung' in bezug auf das exakte Ergebnis in Abhängigkeit von der Ordnung des Romberg-Prozesses m.
Volle Kurve: Fehlerabschätzung laut Glg. (6.20),
gestrichelte Kurve: 'wahrer' Fehler.

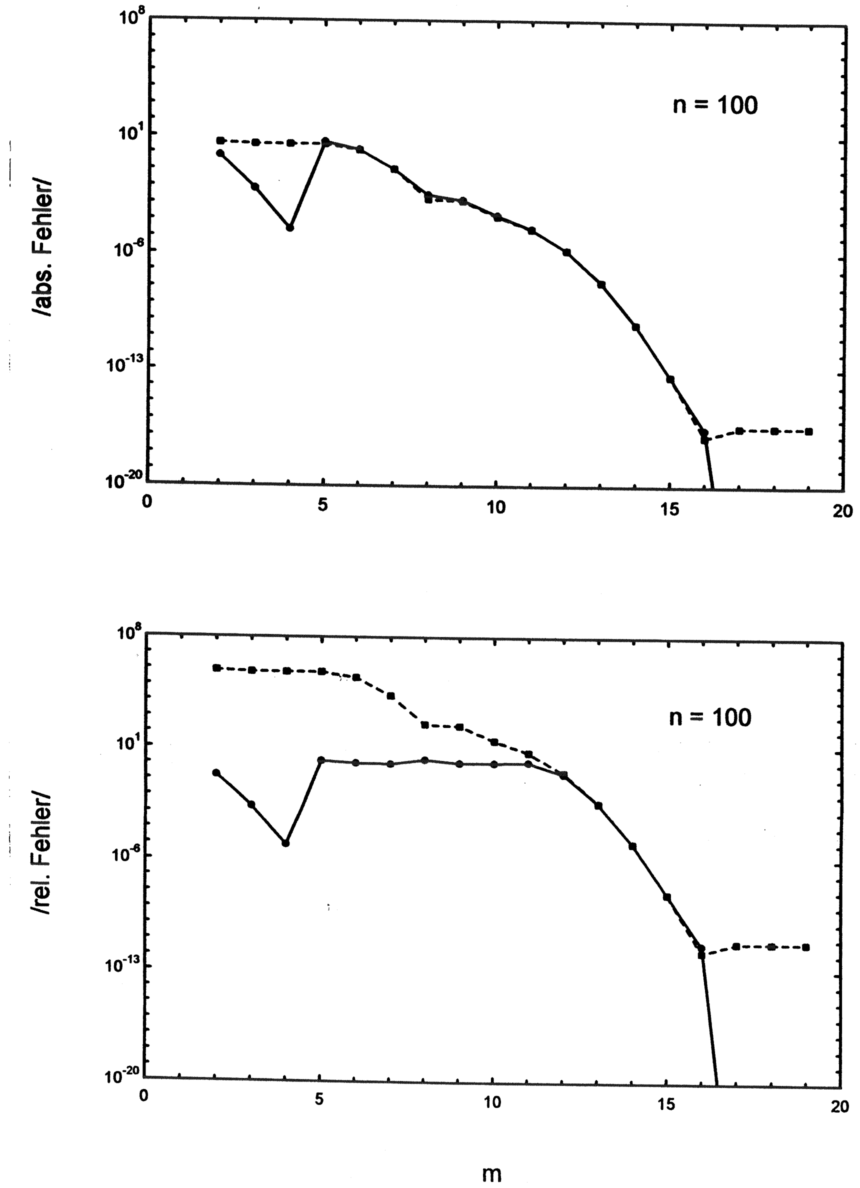


Abbildung 6.5: Romberg-Tests für das Integral (6.21), $n = 100$. Dargestellt ist der *absolute* (oben) und der *relative* Fehler (unten) der 'Romberg-Näherung' in bezug auf das exakte Ergebnis in Abhängigkeit von der Ordnung des Romberg-Prozesses m.
Volle Kurve: Fehlerabschätzung laut Glg. (6.20),
gestrichelte Kurve: 'wahrer' Fehler.

Da für kleine und mittlere Werte m des Romberg-Prozesses die Integrationswerte noch sehr stark vom 'wahren' Integralwert abweichen, kommt es für den *relativen* Fehler in diesem Bereich von m zu sehr großen Unterschieden zwischen dem 'wahren' und dem geschätzten relativen Fehler (s. speziell Abb. 6.5 (unten), wo Unterschiede von bis zu 11(!) Zehnerpotenzen vorkommen). Das daraus resultierende Fehlverhalten des Romberg-Programms wird im folgenden gezeigt:

Angenommen, das Integral (6.21) soll auf eine *relative* Genauigkeit von GEN=10^{-5} berechnet werden. In diesem Fall liefert das Romberg-Programm das folgende Ergebnis:

```
NUM. METH. IN DER PHYSIK  WS 2000/2001  ROMBERG-FORMEL  28-8-2000
INTEGRAL  Skriptum (6.21)  fuer  n = 100
H. Sormann  F90  DOUBLE PRECISION

m      Funkt.    num. Erg.      wahrer     geschaetzter     relGEN
      aufrufe                rel. Fehler   rel. Fehler

2        3      3.93611857  0.743152E+06  0.168200E+00  0.100000E-04
3        5      3.27406321  0.618154E+06  0.210069E-02  0.100000E-04
4        9      3.26718542  0.616855E+06  0.797950E-05  0.100000E-04

EXAKT =      0.0000052965
Ergebnis =  3.2671854208
geforderte Gen. (rel) = 0.100000E-04
Wahre Genauigkeit =     0.616855E+06
Dazu waren        9 Funktionsaufrufe noetig.
```

Auf Grund des Versagens der Fehlerdiagnose stoppt das Programm bereits bei $m = 4$ ab und gibt ein Ergebnis, dessen relativer Fehler um mehr als 10 Zehnerpotenzen größer ist als GEN.

Zumindest eine gewisse Möglichkeit, einem solchen Verhalten gegenzusteuern, bietet der im Abschnitt 6.6.1 angegebene Parameter KMIN. Dieser Wert gibt an, *ab welcher Stufe des Romberg-Prozesses die Fehlerdiagnostik beginnen soll.* Wenn man nun weiß, daß die Integrandenfunktion 'schwierig' ist (starke Oszillationen, Spitzen etc.), kann man durch die Eingabe eines geeignet großen Wertes von KMIN solche Probleme entschärfen.

Zum Beispiel gibt das Romberg-Programm für KMIN=6 das richtige Ergebnis:

```
NUM. METH. IN DER PHYSIK  WS 2000/2001  ROMBERG-FORMEL  28-8-2000
INTEGRAL  Skriptum (6.21)  fuer  n = 100
H. Sormann  F90  DOUBLE PRECISION

(*) am Ende der Zeile bedeutet: keine Fehlerdiagnose.

 m      Funkt.    num. Erg.      wahrer     geschaetzter     relGEN
       aufrufe              rel. Fehler   rel. Fehler

 2        3      3.93611857  0.743152E+06  0.168200E+00  0.100000E-04 *
 3        5      3.27406321  0.618154E+06  0.210069E-02  0.100000E-04 *
 4        9      3.26718542  0.616855E+06  0.797950E-05  0.100000E-04 *
 5       17      3.26715935  0.616850E+06  0.144805E+01  0.100000E-04 *
 6       33     -1.46384560  0.276380E+06  0.107212E+01  0.100000E-04
 7       65      0.10557102  0.199312E+05  0.984072E+00  0.100000E-04
 8      129      0.00168158  0.316489E+03  0.180823E+01  0.100000E-04
 9      257     -0.00135911  0.257604E+03  0.111862E+01  0.100000E-04
10      513      0.00016122  0.294385E+02  0.114146E+01  0.100000E-04
11     1025     -0.00002281  0.530572E+01  0.130154E+01  0.100000E-04
12     2049      0.00000688  0.298361E+00  0.232677E+00  0.100000E-04
13     4097      0.00000528  0.373724E-02  0.376226E-02  0.100000E-04
14     8193      0.00000530  0.109633E-04  0.109709E-04  0.100000E-04
15    16385      0.00000530  0.768462E-08  0.768636E-08  0.100000E-04

 EXAKT =      0.0000052965
 Ergebnis =  0.0000052965
 geforderte Gen. (rel) = 0.100000E-04
 berechnete Gen. (rel) = 0.768636E-08
 Wahre Genauigkeit =     0.768462E-08
 Dazu waren    16385 Funktionsaufrufe noetig.
```

6.7 Die Gauss'sche Quadraturformel.

Ausgangspunkt ist wieder die Gleichung (6.2), wobei vorläufig nur das Integrationsintervall $[-1,+1]$ betrachtet wird:

$$\int_{-1}^{+1} dx f(x) \approx \sum_{j=1}^{m} g_j f(x_j) \quad . \tag{6.22}$$

Zum Unterschied von der Trapez- und der Simpsonformel werden nun *keinerlei einschränkende Bedingungen* bzgl. der Optimierung der Parameter $g_1 \dots g_m$ und $x_1 \dots x_m$ gemacht. Es sind also für eine Gauss-Quadraturformel m-ten Grades insgesamt $2m$ Parameter zu optimieren. Die Optimierungsvorschrift (6.4) lautet also in diesem Fall

$$\int_{-1}^{+1} dx P_{2m-1}(x) = g_1 P_{2m-1}(x_1) + \cdots + g_m P_{2m-1}(x_m) \quad .$$

Man kann nun zeigen, daß die obige Gleichung erfüllt ist, wenn man für $x_1 \dots x_m$ *die Nullstellen des Legendre-Polynoms* m*-ter Ordnung* (L_m) *verwen-*

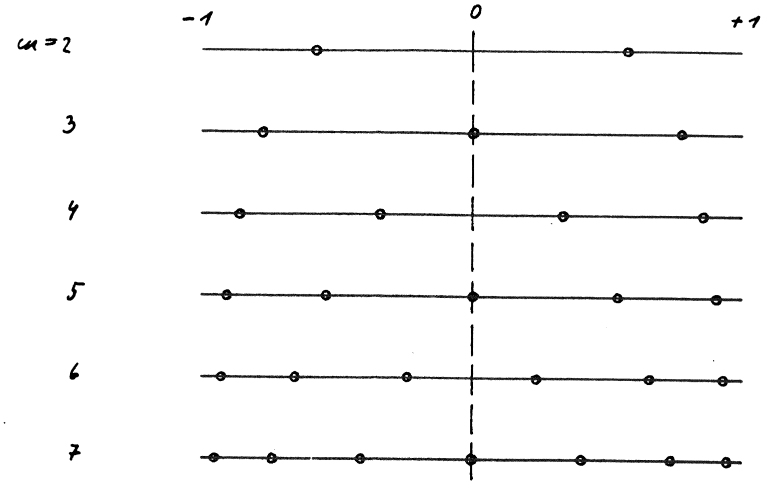


Abbildung 6.6: Stützpunktverteilung bei der Gauss-Quadratur.

det. Aus diesem Grund wird diese Quadraturformel auch als *Gauss-Legendre-Formel* bezeichnet.
(o. B:) Die zugehörigen Gewichtsfaktoren ergeben sich aus

$$g_j = \frac{2}{(1-x_j^2)\left[\frac{d}{dx}L_m(x)\right]^2_{x=x_j}} .$$

Was die Verteilung der Stützpunkte im Intervall $[-1,+1]$ betrifft, so ist diese natürlich nicht mehr äquidistant (s. Abb. 6.6), sondern hat die folgenden Eigenschaften:

- Die Stützstellen sind bzgl. des Intervallmittelpunktes symmetrisch verteilt. Jedem Stützpunkt x_j im rechten Halbintervall entspricht ein Punkt $-x_j$ im linken Halbintervall, wobei beide Punkte denselben Gewichtsfaktor haben.

- Die Stützpunkt-Dichte nimmt zu den Intervallgrenzen hin zu.

- Die Intervallgrenzen selbst sind keine Stützpunkte.[6]

Bisher wurden alle Überlegungen zur Gauss-Quadratur für das spezielle Intervall $[-1,+1]$ durchgeführt. Die entsprechende Quadraturformel

$$\int_{-1}^{+1} dx f(x) \approx \frac{(+1)-(-1)}{2}\sum_{j=1}^{m} g_j f(x_j)$$

kann nun mittels der *linearen Transformation*

$$z_j = \frac{d+c}{2} + \frac{d-c}{2} x_j$$

[6]Die Gauss-Formel ist eine *offene* Quadraturformel, während die Trapez- und die Simpson-Formel *geschlossene* Quadraturformeln sind.

leicht auf die allgemeine Form

$$\int_c^d dx f(x) \approx \frac{d-c}{2} \sum_{j=1}^{m} g_j f(z_j) \quad , \tag{6.23}$$

gebracht werden, wobei der Vorfaktor $(d-c)/2$ meistens in die Gewichtsfaktoren mit einbezogen wird:

$$w_j \equiv \frac{(d-c)}{2} g_j \quad .$$

6.7.1 Der Verfahrensfehler bei der Gauss-Quadratur.

Wegen der Kompromisslosigkeit bei der Optimierung der Parameter ist zu erwarten, daß die Gauss-Quadraturformel allen anderen Quadraturformeln überlegen ist. Dies ist in vielen Fällen auch tatsächlich so: Betrachtet man nämlich den Verfahrensfehler der Gauss-Quadratur (s. z.B. [16], S.199)

$$E_V^G = \frac{(d-c)^{2m+1}(m!)^4}{(2m+1)[(2m)!]^3} f^{(2m)}(\xi) \qquad \xi \in [c,d] \quad , \tag{6.24}$$

so erkennt man, daß der dominante Nenner-Term $[(2m)!]^3$ bei steigendem m für eine rasante Verkleinerung des Verfahrensfehlers sorgt!

6.7.2 Die Berechnung der Gauss-Parameter.

Ein wichtiges Kriterium für die Anwendung der Gauss-Quadratur ist die bequeme Verfügbarkeit der Gauss-Parameter z_j und w_j in (6.23). Es gibt wohl keine ernstzunehmende Programm-Library, die nicht über ein einschlägiges Generierungsprogramm verfügt. 'Numerical Recipes' enthält die Routine GAULEG in FORTRAN ([9], S. 125f), PASCAL ([9], S. 700f) und C ([10], S. 152).

Die Routinen GAULEG.F90 bzw. GAULEG.C berechnen die Parameter der Gauss-Quadratur ('Gauss-Legendre-Parameter') für ein beliebiges Intervall und eine beliebige Ordnung:

<u>INPUT-Parameter:</u>

X1,X2: Intervallgrenzen für die Integration.

N: Grad der Gauss-Quadratur.

<u>OUTPUT-Parameter:</u>

X( ): Feld mit den Abszissenwerten x_j der Stützpunkte.

W( ): Feld mit den Gewichtsfaktoren w_j.

6.7.3 Das Programm GAUINT.

Das Programm GAUINT berechnet näherungsweise ein bestimmtes Integral mittels der Gauss'schen Quadraturformel. Die erforderlichen Parameter werden von der Routine GAULEG (s. o.) berechnet.

INPUT-Parameter:

C,D: Integrationsgrenzen.

IREL: IREL=1 relative Fehlerabfrage, ansonsten absolute Fehlerabfrage.

GEN: relative oder absolute Genauigkeit.

MANF: Anfangs-Ordnung der Quadratur.

MINKR: Erhöhung der Ordnung bei Nicht-Erreichen der Genauigkeit.

MMAX: maximale Ordnung der Quadratur.

OUTPUT-Parameter:

S: Näherung für das bestimmte Integral.

ERROR: Abschätzung für den absoluten bzw. relativen Fehler dieser Näherung.

FEHLER: logische Fehlervariable: FALSE, wenn alles o.k., sonst TRUE.

Interne Parameter:

Z( ),W( ): eindimensionale Felder für die Gauss-Parameter.

Struktogramm 19: Das Programm GAUINT.

Struktogramm 19 — GAUINT(C,D,IREL,GEN,MANF,MINKR,MMAX,S,ERROR, FEHLER)

```
FEHLER:=TRUE
Y  MANF, MINKR < 1  N
   Y: print: 'GAUINT 'MANF oder MINKR < 1'
      (return)
   N: ......
M:=MANF
OLDS:=1.E30
   GAULEG(C,D,Z,W,M)
   S:=0.0
   J=1(1)M
      S:=S+W(J)*FUNC(Z(J))
   ERROR:= | S–OLDS |
   Y  IREL = 1  N
      Y: ERROR:=ERROR/ | OLDS |
      N: ......
   Y  ERROR < GEN  N
      Y: FEHLER:=FALSE
         S:=OLDS
      N: ......
   OLDS:=S
   M:=M+MINKR
M > MMAX .or. notFEHLER
(return)
```

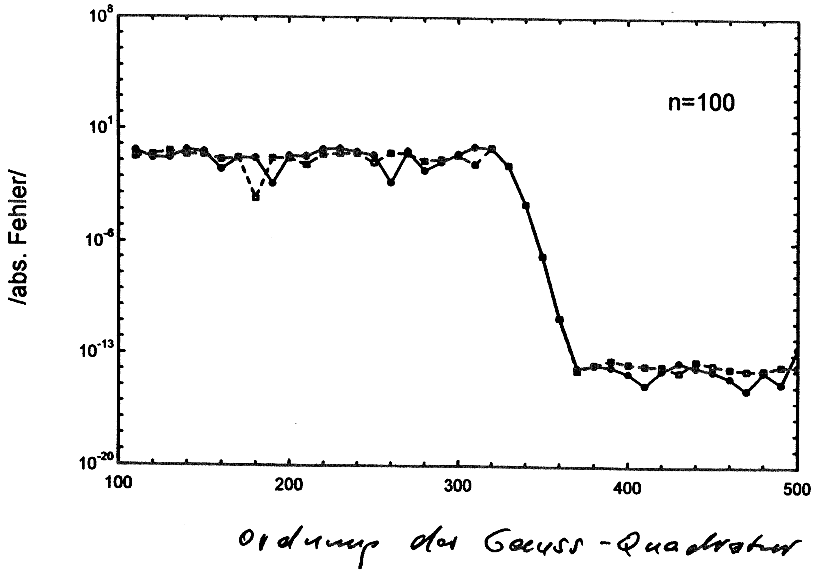


Abbildung 6.7: Anwendung des Programms GAUINT auf das Integral (6.21) für $n = 100$. Dargestellt sind der 'wahre' absolute Fehler (strichlierte Kurve) und der numerisch abgeschätzte Fehler (volle Kurve) als Funktion der Ordnung der Gauss-Legendre-Formel.
Parameter: MANF=100 MINKR=10.

6.7.4 Genauigkeitsabfragen

Beim Romberg-Verfahren gibt es mit (6.20) eine Formel, die auf einfache Weise eine in vielen Fällen zuverlässige Fehlerabschätzung ermöglicht. Diese Methode ist auch im Programm GAUINT realisiert, d.h., man geht davon aus, daß die Differenz zweier aufeinanderfolgender Näherungen der Gauss-Quadratur (mit den Ordnungen M und M+MINKR) eine vernünftige Abschätzung des absoluten Fehlers des Ergebnisses mit der Ordnung M darstellt. Meine Tests haben gezeigt, daß diese Aussage (für nicht zu große MINKR) $\approx$ unabhängig vom MINKR ist.

6.7.5 Ein Qualitätsvergleich von GAUINT mit QTRAP und ROMB

Im folgenden soll das Gauss-Legendre-Verfahren auf das 'schwierige' Integral (6.21) für $n = 100$ angewendet werden. Die Abb. 6.7 zeigt das Verhalten des absoluten Fehlers in Abhängigkeit von der Ordnung der Quadraturformel, d.h. von der Zahl der verwendeten Funktionswerte, wobei wieder der 'wahre' Fehler dem numerisch geschätzten Fehler gegenübergestellt ist.
Die Ergebnisse sind im Prinzip ähnlich wie beim Romberg-Verfahren: bis zu einer 'kritischen Ordnung' von etwa m=320 ist der Verfahrensfehler sehr groß; danach fällt er innerhalb eines relativ kleinen Bereichs (m zwischen 320 und 370) auf den erreichbaren Mindestfehler von ca. 10^{-14} ab. Zumindest in diesem Bereich ist die Fehlerabschätzung gemäß Abschnitt 6.7.4 recht zuverlässig.

Abb. 6.7 zeigt beeindruckend die große Leistungsfähigkeit der Gauss-

Quadratur gegenüber den anderen in diesem Kapitel behandelten Methoden: selbst bei diesem sehr schwierig auszuwertenden Integral genügen etwa 370 'Gauss-Stützpunkte', um eine Genauigkeit des Integrals von $< 10^{-14}$ zu erreichen! Beim Romberg-Verfahren ist dafür etwa eine Ordnung 15 nötig (s. Abb. 6.5 oben), was weit mehr 10000 Stützpunkte bedeutet!

Diese Überlegenheit der Gauss-Quadratur wird zwar auch im folgenden sehr deutlich, muß aber dennoch - wie sogleich demonstriert wird - kritisch betrachtet werden:

Für einen Vergleich der in diesem Kapitel präsentierten Verfahren soll als *Qualitätskriterium* die Rechenzeit herangezogen werden, die zur numerischen Berechnung eines bestimmten Integrals mit einer bestimmten Genauigkeit erforderlich ist. Da i.a. die zeitintensivste Komponente aller präsentierten Programme die Auswertung der Integrandenfunktion ist, kann man das Qualitätskriterium wie folgt definieren:

Jenes numerische Verfahren zur Berechnung bestimmter Integrale ist das beste, das zur Erlangung einer bestimmten Genauigkeit die kleinste Zahl von Stützpunkten benötigt.

Es soll nun als Test für die Trapezformel (Programm QTRAP), das Romberg-Verfahren (Programm ROMB) und die Gaussformel (Programm GAUINT) das bereits früher (s. S. 169) verwendete bestimmte Integral

$$\int_0^1 dx \frac{\ln(1+x)}{x(1+x)} = \frac{\pi^2}{12} - \frac{1}{2}(\ln 2)^2 = 0.5822405\ldots$$

numerisch ausgewertet werden. Die Ergebnisse dieses Tests sind in der Abb. 6.8 zusammengefaßt:

- Der Abbildung ist sofort zu entnehmen, daß die Trapezformel sowohl dem Romberg-Verfahren wie auch der Gauss-Formel stark unterlegen ist. Sie benötigt weit mehr als 1 Million (!) Stützpunkte, um den durch den Rundungsfehler gegebenen *minimalen Fehler* von ca. 10^{-15} zu erreichen.

- Das Romberg-Verfahren schafft das schon bei ≈ 100 Stützpunkten.

- Auf den ersten Blick konkurrenzlos: die Gauss-Legendre-Quadratur, die lediglich 10 (!) Stützpunkte benötigt, um das Test-Integral bis auf einen absoluten Fehler von 10^{-15} auszuwerten!

- Dazu ist jedoch zu bemerken, daß beim Gauss-Verfahren – zum Unterschied vom Romberg-Verfahren – eine Reihe von Stützpunkten *im Laufe des 'Herantastens' an die gewünschte Genauigkeit* berechnet werden müssen, die in die eigentliche Berechnung des Näherungswertes nicht eingehen. Diese unökonomische Eigenschaft des Programms GAUINT reduziert etwas die Überlegenheit dieser Methode gegenüber dem Verfahren von Romberg.

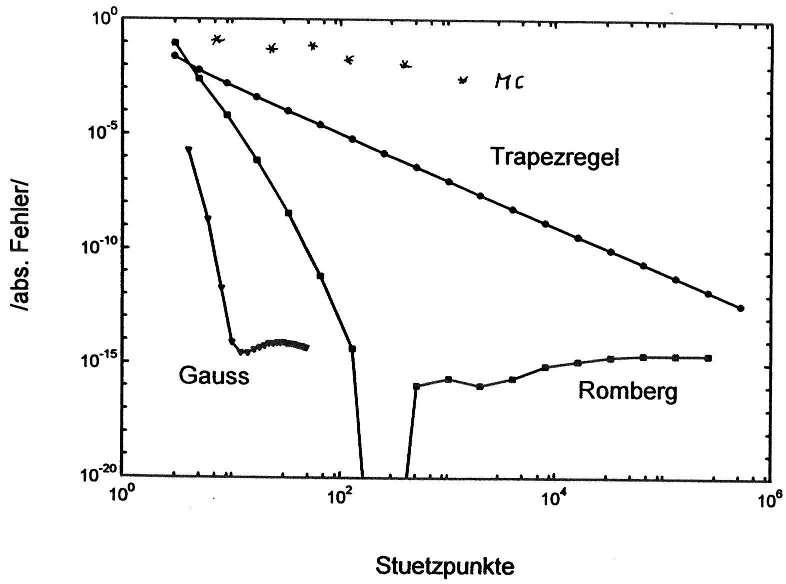


Abbildung 6.8: Ein Genauigkeitstest für QTRAP, ROMB und GAUINT.
(*) 'Monte-Carlo'-Ergebnisse.

- Nochmals soll hier auf das *Rundungsfehlerverhalten* der Quadraturformeln hingewiesen werden:
  Es ist wegen dieses Fehlers nicht möglich, die Genauigkeit des numerisch ermittelten Integrationsergebnisses durch Erhöhung der Stützpunktanzahl beliebig zu steigern. Wie das Diagramm 6.8 deutlich zeigt, kippen die Fehlerkurven für alle getesteten Methoden bei ungefähr derselben Genauigkeit (Fehler ca. 10^{-14}–10^{-15})!

- Es muß aber an dieser Stelle auch betont werden, daß ein einzelner Test natürlich nicht ausreicht, klare Aussagen über das allgemeine Verhalten der verschiedenen Methoden zu machen. Trotzdem bietet er einen Einblick in das typische Verhalten dieser Programme in vielen praktischen Fällen!

6.8 Numerische Auswertung uneigentlicher Integrale.

Definition: Ein uneigentliches Integral liegt vor, wenn

- die obere und/oder die untere Integrationsgrenze unbeschränkt sind/ist (Typ 1),

- die zu integrierende Funktion an einer der beiden oder an beiden Intervallgrenzen oder im Inneren des Intervalls unbeschränkt ist
  (Typ 2),

wobei trotz dieser Unbeschränktheiten die Integralfläche *endlich* ist.

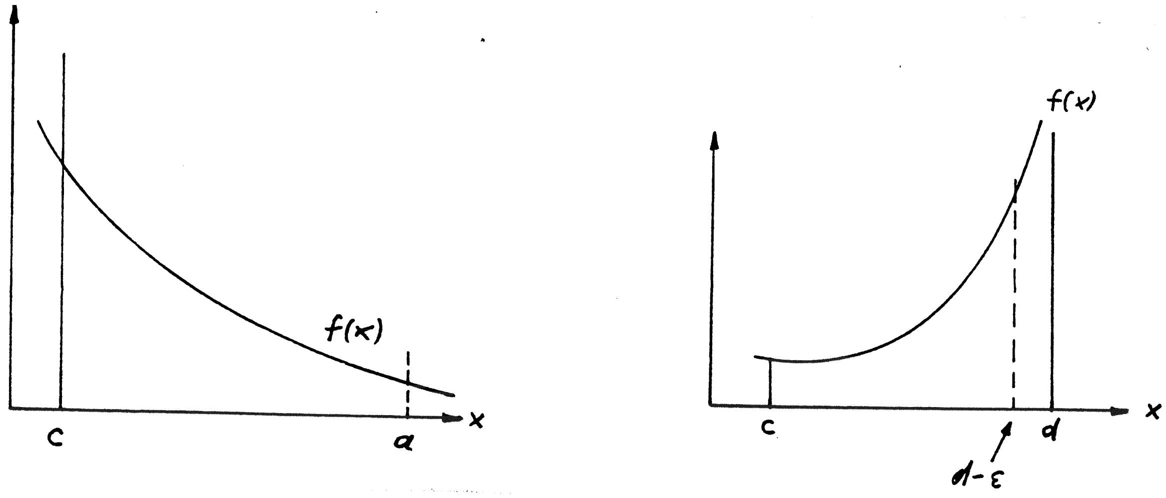


Abbildung 6.9: Uneigentliche Integrale vom Typ 1 und vom Typ 2.

- Hier sollen - mathematisch nicht ganz korrekt - auch jene Integrale als uneigentliche Integrale angesehen werden, bei welchen der Integrand an einer der beiden (oder an beiden) Grenzen des Integrationsintervalls eine *unbestimmte Form* annimmt (Typ 3).

Selbstverständlich können die Probleme (1)-(3) auch in Kombination vorkommen!

Mathematische Formulierung:

Typ 1 (s. Abb. 6.9, links):

$$I(a) = \int_c^a dx f(x) \qquad \lim_{a\to\infty} I(a) \to I$$

Typ 2 (s. Abb. 6.9, rechts):

$$I(\epsilon) = \int_c^{d-\epsilon} dx f(x) \qquad \lim_{\epsilon\to 0} I(\epsilon) \to I$$

6.8.1 Auswertung von Integralen vom Typ 1.

Eine durchaus praktikable, wenn auch nicht sehr elegante Methode, mit Integralen der Art

$$\int_c^\infty dx f(x) \qquad \int_{-\infty}^d dx f(x) \qquad \int_{-\infty}^\infty dx f(x)$$

fertigzuwerden, ist die folgende: man setzt anstatt 'unendlich' eine hinlänglich große Zahl a_1 und berechnet das Integral

$$I_1 = \int_c^{a_1} dx f(x) \quad .$$

Dann erhöht man a_1 auf a_2 und erhält I_2.
Ist nun $\mid I_2 - I_1 \mid$ absolut oder relativ genügend klein, kann I_2 als Näherungwert für das uneigentliche Integral angesehen werden.

Diese einfache Methode ist jedoch nicht immer brauchbar. So müßte z.B., wenn man das uneigentliche Integral

$$I = \int_0^\infty \frac{dx}{(1+x^2)^{4/3}}$$

auf 5 Dezimalen genau berechnen will, für '∞' eine Zahl > 1000 gesetzt werden (s. [8], S.219)!
In solchen Fällen ist eine *nicht-lineare Transformation der Integrationsvariablen* zu empfehlen. So kann mit

$$t = \frac{1}{1+x}$$

das obige Integral in die Form

$$I = \int_0^1 dt \frac{t^{2/3}}{[t^2 + (1-t)^2]^{4/3}}$$

übergeführt werden. Dabei wird also das nach oben offene Intervall $[0, \infty]$ auf das geschlossene Intervall $[0, 1]$ abgebildet. In dieser Form kann das Integral numerisch sehr genau ausgewertet werden (im folgenden eine Auswertung mittels GAUINT):

.

```
  2 Stuetzpunkte       I = .103746E+01
  6 Stuetzpunkte       I = .111758E+01
 12 Stuetzpunkte       I = .112029E+01
 20 Stuetzpunkte       I = .112031E+01
 30 Stuetzpunkte       I = .112028E+01
 42 Stuetzpunkte       I = .112027E+01
 56 Stuetzpunkte       I = .112026E+01
 72 Stuetzpunkte       I = .112026E+01
 90 Stuetzpunkte       I = .112026E+01
110 Stuetzpunkte       I = .112025E+01
132 Stuetzpunkte       I = .112025E+01
```

6.8.2 Auswertung von Integralen vom Typ 2.

Als Beispiel diene das uneigentliche Integral

$$I = \int_0^1 dx \frac{e^x}{\sqrt{x}} = 2.9253034918... \tag{6.25}$$

mit einer Unendlichkeitsstelle am Nullpunkt. Für die Auswertung gibt es nun die folgenden Möglichkeiten:

1. Man ignoriert einfach die Singularität. Dies ist jedoch nur möglich, wenn man die Gauss-Quadratur oder eine andere *offene* Quadraturformel verwendet, weil ja in diesem Fall der Integrand nicht an der Stelle $x = 0$ ausgewertet werden kann!

   Wie man allerdings der Abb. 6.10 entnehmen kann, muß man diese Ignoranz der Polstelle mit einer sehr schlechten Konvergenz des Verfahrens bezahlen!

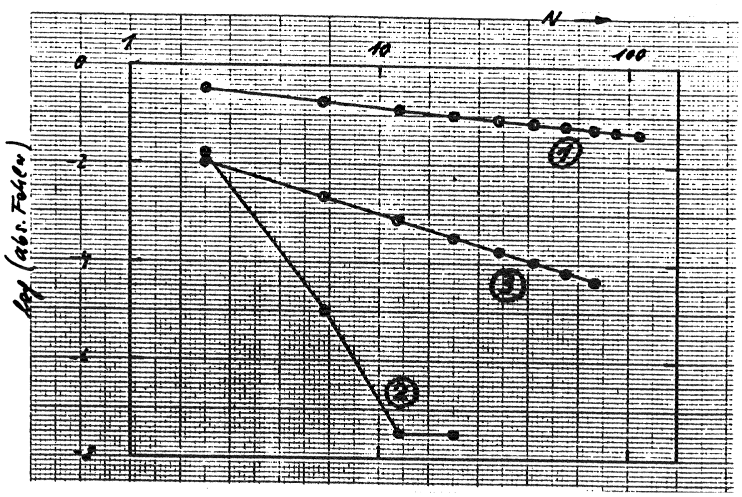


Abbildung 6.10: Numerische Auswertung des Integrals (6.25). Bedeutung der Nummern siehe Text.

2. Man eliminiert die Singularität des Integranden durch *geeignete analytische Umformungen*, wie z.B. mittels der Variablen-Transformation

$$x = t^2 \quad ,$$

die zum Integral

$$I = 2 \int_0^1 dt e^{t^2}$$

führt. Dieses Integral hat ausgezeichnete Konvergenz-Eigenschaften.

3. Auch eine *partielle Integration* des Integrals führt zum Ziel:

$$I = 2e - 2 \int_0^1 dx \sqrt{x} e^x \quad .$$

Im Diagramm 6.10 ist dargestellt, wie das Gauss-Verfahren für die drei besprochenen Varianten konvergiert.

6.8.3 Auswertung von Integralen vom Typ 3.

Integranden mit Unbestimmtheitsstellen an den Intervallgrenzen bereiten offenen Quadraturformeln überhaupt keine Probleme. Verwendet man geschlossene Quadraturformeln, muß man allerdings die Unbestimmtheitsstellen explizite berücksichtigen. So würde z.B. das Funktionsunterprogramm für den Integranden des Integrals

$$I = \int_0^1 dx \frac{\sin x}{x}$$

wie folgt aussehen:

```
FUNCTION FUNC(x: real): real;
BEGIN
  IF(x = 0.0)THEN func:=1.0 ELSE func:=sin(x)/x
END;
```

6.9 Variationen der Gauss-Quadratur.

Die soeben besprochene Gauss-Quadraturformel (6.22) ist nur ein Spezialfall einer Gruppe von Quadraturformeln für die bestimmten Integrale

$$I = \int_c^d dx W(x) f(x) \approx \sum_{i=1}^{m} w_i f(x_i) \quad , \tag{6.26}$$

wobei der als *Gewichtsfunktion* $W(x)$ bezeichnete Teil des Integranden fix gegeben ist. Die Formel (6.22) ergibt sich also aus (6.26) durch $W(x) \equiv 1$. Da die Stützpunkte in diesem Fall (für $c = -1$ und $d = +1$) die Nullstellen der Legendre-Polynome sind, wird (6.22) auch die *Gauss-Legendre-Formel* genannt.
Andere, für die Praxis wichtige Spezialfälle von $W(x)$ sind

$$W(x) = \mathrm{e}^{-x} \qquad \text{für das Intervall} \quad [0, \infty)$$

sowie

$$W(x) = \mathrm{e}^{-x^2} \qquad \text{für das Intervall} \quad (-\infty, \infty)\,.$$

Da die entsprechenden Stützpunkte die Nullstellen der *Laguerre'schen* bzw. der *Hermite'schen* Polynome darstellen, heißen die davon hergeleiteten Quadraturformeln
die <u>Gauss-Laguerre-Formel</u>

$$\int_0^{+\infty} dx\, \mathrm{e}^{-x} f(x) \approx \sum_{i=1}^{m} w_i^{Lag} f(x_i^{Lag}). \tag{6.27}$$

beziehungsweise die <u>Gauss-Hermite-Formel</u>:

$$\int_{-\infty}^{+\infty} dx\, \mathrm{e}^{-x^2} f(x) \approx \sum_{i=1}^{m} w_i^{H} f(x_i^{H})\,. \tag{6.28}$$

Formeln dieser Art sind speziell für die numerische Behandlung von uneigentlichen Integralen vom Typ (1) (s. Abschnitt 6.8) interessant.

Viele Libraries bieten Programme an, die die Stützpunkte und Gewichtsfaktoren einer ganzen Reihe solcher Sonderfälle berechnen (die *Numerical Recipes C* enthalten z. B. die Routinen GAULAG (Gauss-Laguerre), GAUHER (Gauss-Hermite) und andere); siehe dazu auch Abschnitt 6.11).

6.10 Mehrfach-Integrale

sind im Prinzip kein Problem. Wenn man sich auf Mehrfach-Integrale mit konstanten Grenzen beschränkt, so gilt einfach

$$\begin{aligned} &\int_{c_[1}^{d_1} \int_{c_2}^{d_2} \cdots \int_{c_k}^{d_k} dx_1 dx_2 \cdots dx_k \quad f(x_1, x_2, \ldots, x_k) \\ &\approx \frac{(d_1 - c_1)(d_2 - c_2) \cdots (d_k - c_k)}{2^k} \cdot \sum_{j_1=1}^{m_1} g_{j_1} \sum_{j_2=1}^{m_2} g_{j_2} \cdots \sum_{j_k=1}^{m_k} g_{j_k} \cdot f(x_{j_1}^{(1)}, x_{j_2}^{(2)}, \ldots, x_{j_k}^{(k)}) \quad . \end{aligned} \tag{6.29}$$

Die Anwendbarkeit solcher *mehrdimensionalen Quadraturformeln* ist allerdings in der Praxis wegen des immens hohen Rechenaufwandes auf Integrale nicht zu hoher Ordnung ($k = 1, 2, 3(?), ...$) beschränkt.

Für Integrale höherer Ordnung bzw. für Mehrfach-Integrale mit komplizierten Integranden sind *statistische Methoden (Monte-Carlo-Methoden)* von großer Bedeutung, insbesondere dann, *wenn keine allzu hohe Genauigkeit gefordert wird!*

6.11 Software-Angebot

NAG: In dieser Library finden Sie z. B. mit der Routine *d01bcf* ein gutes Beispiel für ein Programm, das die numerische Auswertung sechs verschiedener Gauss-Integralformeln ermöglicht. Das Programm liefert nicht das Ergebnis der Quadraturformel, sondern lediglich die Stützpunkte (Feld ABSCIS) und die Gewichtsfaktoren (Feld WEIGHT) für die Quadraturformel.
Die Headline des Programmes (in FORTRAN) lautet wie folgt:

```
SUBROUTINE D01BCF(ITYPE,A,B,C,D,N,WEIGHT,ABSCIS,IFAIL)
INTEGER ITYPE,N,IFAIL
REAL*8  A,B,C,D,WEIGHT(N),ABSCIS(N)
     .
     .
```

ITYPE=0 Gauss-Legendre $I = \int_a^b dx\, f(x)$

ITYPE=1 Gauss-Jacobi $I = \int_a^b dx\, (b-x)^c (x-a)^d\, f(x)$

ITYPE=2 Gauss-Exponential $I = \int_a^b dx\, |x - \frac{a+b}{2}|^c\, f(x)$

ITYPE=3 Gauss-Laguerre $I = \int_a^\infty dx\, |x-a|^c\, \mathrm{e}^{-bx}\, f(x)$
when $b > 0$

ITYPE=4 Gauss-Hermite $I = \int_{-\infty}^{+\infty} dx\, |x-a|^c\, \mathrm{e}^{-b(x-a)^2}\, f(x)$

ITYPE=5 Gauss-Rational $I = \int_a^\infty dx\, \frac{|x-a|^c}{|x+b|^d}\, f(x)$
when $a + b > 0$.

Die Ermittlung des Näherungswertes für das Integral lautet in allen Fällen

$$I \approx \sum_{1=1}^{n} w_i\, f(x_i)\,.$$

Ebenfalls in **NAG**: *d01daf* 2-D quadrature, finite region.

d01fbf Multi-dimensional Gauss.

d01gbf Multi-dimensional Monte Carlo.

QUADPACK: (p.d.) Paket, erhältlich über NETLIB *für die Berechnung bestimmter Integrale und Integraltransformationen von Funktionen einer Veränderlichen.*

TOMS: (p.d.) $NETLIB/TOMS/655$ enthält ebenfalls Programme für Variationen des Gauss-Quadraturformel.

Kapitel 7

Eigenwerte und Eigenvektoren reeller Matrizen

7.1 Einleitung: allgemeine und reguläre Eigenwertprobleme.

Im Kapitel 2 standen lineare *inhomogene* Probleme der Art

$$A \cdot \mathbf{x} = \mathbf{f} \qquad A\text{: reelle Matrix}$$

im Mittelpunkt. Im folgenden geht es um *homogene* Probleme, d.h. um solche, bei denen der Vektor $\mathbf{f}$ den *Nullvektor* darstellt:

$$A \cdot \mathbf{x} = 0 \tag{7.1}$$

Bei der Lösung eines solchen Problems ist laut Theorie zwischen zwei Fällen zu unterscheiden:

- Die Determinante der Koeffizientenmatrix A ist von Null verschieden: Dann ist der Lösungsvektor $\mathbf{x}$ einfach der Nullvektor.

- Die Determinante von A ist Null: Unter diesen Umständen gibt es neben der *trivialen* Lösung $\mathbf{x} = 0$ auch noch eine *nicht-triviale* Lösung $\mathbf{x} \neq 0$. So hat z.B. das homogene System

  $$\begin{pmatrix} 1 & 2 & 3 \\ -1 & 13 & 2 \\ 0 & 3 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = 0$$

  die nicht-triviale Lösung $\mathbf{x} = (-7, -1, +3)$. Es ist evident, daß dieser Lösungsvektor nur bis auf eine willkürliche Konstante bestimmt ist: mit $\mathbf{x}$ ist natürlich auch jeder Vektor $c \cdot \mathbf{x}$ Lösung des obigen Systems.

Es sind aber nicht Systeme vom Typus (7.1), um die es im folgenden geht. Viel wichtiger für die praktische Anwendung sind nämlich homogene Systeme der Art

$$A(\lambda) \cdot \mathbf{x} = 0 \qquad \text{mit} \qquad A(\lambda) = [a_{ij}(\lambda)] \quad , \tag{7.2}$$

bei denen die Matrixelemente von einer Variablen λ abhängen.
Die Lösungen von (7.2) sind:

- die triviale Lösung $\mathbf{x} = 0$.
- nicht-triviale Lösungen $\mathbf{x}_1, \mathbf{x}_2, \ldots$ immer dann, wenn λ einen der Werte $\lambda_1, \lambda_2, \ldots$ annimmt, die zu einer *Singularität* der Matrix A führen.

Die *Bedingungsgleichung für nicht-triviale Lösungen* lautet demnach:

$$det[A(\lambda)] = 0 \tag{7.3}$$

Die (i.a. komplexwertigen) Lösungen der Gleichung (7.3), $\lambda_1, \lambda_2, \ldots$ nennt man die *Eigenwerte*, und die entsprechenden Lösungsvektoren $\mathbf{x}_1, \mathbf{x}_2, \ldots$ nennt man die *Eigenvektoren* des Systems (7.2).
Die Ermittlung der Eigenwerte und -vektoren von (7.2) heißt Lösung des *allgemeinen Eigenwertproblems*. Für solche Probleme bleibt zumeist nur die aufwendige Methode, die Nullstellen der Gleichung (7.3) numerisch zu ermitteln, was die Berechnung von sehr vielen Determinanten erfordert und dementsprechend lange Rechenzeiten erfordert.

Ein sehr wichtiger Sonderfall des allgemeinen Eigenwertproblems (7.2) ist das *reguläre Eigenwertproblem* mit

$$A(\lambda) = A_0 - \lambda \cdot E \qquad \text{E ... Einheitsmatrix} \quad , \tag{7.4}$$

wobei die Matrix A_0 nicht von λ abhängt! Für derartige Probleme stehen effizientere numerische Methoden zur Verfügung.

Für reguläre Eigenwertprobleme

$$(A_0 - \lambda E)\mathbf{x} = 0 \qquad \text{bzw.} \qquad A_0\,\mathbf{x} = \lambda\,\mathbf{x} \tag{7.5}$$

lautet die Bedingungsgleichung für nicht-triviale Lösungen offenbar

$$det(A_0 - \lambda E) = 0 \quad . \tag{7.6}$$

Da im folgenden ausschließlich von regulären Eigenwertproblemen die Rede sein wird, wird ab nun auf die Indizierung von A_0 verzichtet.

Die Auswertung von (7.6) führt zu einem Polynom n-ten Grades von λ, wenn n die Ordnung des linearen Systems (7.5) ist. Man nennt dieses Polynom *das charakteristische Polynom der Matrix A*:

$$det(A - \lambda E) \equiv P_n(\lambda) = \lambda^n + \sum_{i=1}^{n} p_i \lambda^{n-i} \tag{7.7}$$

Die $p_1, \ldots, p_n$ sind die (reellen) Koeffizienten des charakteristischen Polynoms. Dieses Polynom hat genau n (nicht notwendig verschiedene) Nullstellen, die entweder reell sind oder konjugiert-komplexe Wertepaare darstellen. Diese Nullstellen sind wegen (7.6) die n Eigenwerte der Matrix A. Dabei kann es ohne weiteres vorkommen, daß das charakteristische Polynom *Mehrfach-Nullstellen* hat, daß also gilt:

$$P_n(\lambda) = 0 \qquad \text{für} \qquad \lambda : \lambda_1, \lambda_2, \ldots, \lambda_k = \lambda_{k+1}, \ldots = \lambda_{k+q-1}, \lambda_{k+q}, \ldots, \lambda_n \quad .$$

Man nennt den Eigenwert λ_k *q-fach entartet*.

Zu jedem Eigenwert λ_i gehört nun gemäß (7.5) ein Eigenvektor $\mathbf{x}_i$.

7.1.1 Eigenwerte und Eigenvektoren wichtiger Matrixformen

Wie bei den *inhomogenen* linearen Gleichungssystemen spielt auch bei den Eigenwertproblemen die konkrete Form der Matrix A eine wichtige Rolle. Im folgenden einige Fakten zu diesem Thema:

- Eine *reelle* Matrix A heißt *symmetrisch*, wenn gilt:

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

  wobei A^T die zu A transponierte Matrix mit den Komponenten

  $$(a_{i,j})^T = a_{j,i}$$

  darstellt.

- Eine *komplexwertige* Matrix A heißt *hermitesch*, wenn gilt:

  $$A = A^\dagger \,,$$

  wobei $A^\dagger$ die zu A hermitesch konjugierte Matrix mit den Komponenten

  $$(a_{i,j})^\dagger = a^*_{j,i}$$

  darstellt.

Es gilt:
Alle Eigenwerte einer symmetrischen oder hermiteschen Matrix sind reell.

- Eine reelle Matrix A heißt *orthogonal*, wenn gilt:

  $$A\,A^T = E \qquad \text{d. h.} \qquad A^T = A^{-1}\,,$$

  wobei E die Einheitsmatrix darstellt.

- Eine komplexwertige Matrix A heißt *unitär*, wenn gilt:

  $$A\,A^\dagger = E \qquad \text{d. h.} \qquad A^\dagger = A^{-1}\,.$$

- Eine reelle bzw. komplexwertige Matrix wird *normal* genannt, wenn sie mit der zu ihr transponierten bzw. hermitesch konjugierten Matrix vertauscht:

  $$A\,A^T = A^T\,A \qquad \text{bzw.} \qquad A\,A^\dagger = A^\dagger\,A\,.$$

  Es ist evident, daß sowohl symmetrische als auch hermitesche Matrizen zur Klasse der normalen Matrizen gehören.

Eine sehr wichtige Frage ist es auch, ob eine Matrix *diagonalisierbar* ist oder nicht.
Diagonalisierbar heißt, daß eine nicht-singuläre Matrix U existiert, welche die Matrix A in eine Diagonalmatrix D überführt:

$$U^{-1}\, A\, U = D\,. \tag{7.8}$$

Eine derartige Transformation ist eine *Ähnlichkeitsoperation*, bei der sich das Eigenwertspektrum der Matrix nicht ändert; es gilt also:

$$\text{Eigenwerte von } D = \text{Eigenwerte von } A\,.$$

- Da das Eigenwertproblem von D einfach zu lösen ist (die Eigenwerte sind die Diagonalelemente von D), ist eine Diagonalisierung gemäß (7.8) äquivalent mit der Bestimmung der Eigenwerte von A:

$$\lambda_i = d_{ii}\,. \tag{7.9}$$

- Man kann leicht zeigen, daß die Spaltenvektoren von U gleichzeitig die Eigenvektoren von A sind:

  Multipliziert man nämlich die Tranformationsvorschrift (7.8) von links mit U, so ergibt sich

$$U \cdot U^{-1} A U = U \cdot D \qquad \text{und}$$

$$A \cdot U = U \cdot D \quad . \tag{7.10}$$

  Nun kann aber die Matrix U auch *als ein System von n Spaltenvektoren betrachtet werden*:

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ u_{21} & u_{22} & \cdots & u_{2n} \\ . & . & & . \\ . & . & & . \\ u_{n1} & u_{n2} & \cdots & u_{nn} \end{pmatrix} \equiv (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n) \quad .$$

  Die komponentenweise Berechnung der rechten Seite von (7.10) ergibt:

$$(UD)_{ij} = \sum_{l=1}^{n} u_{il} d_{lj} = u_{ij} d_{jj} = \lambda_j u_{ij} \quad .$$

  Auch die Matrix UD kann unter Berücksichtigung von (7.9) als Vektoren-System geschrieben werden:

$$UD \equiv (\lambda_1 \mathbf{u}_1, \lambda_2 \mathbf{u}_2, \ldots, \lambda_n \mathbf{u}_n) \quad .$$

  Somit hat (7.10) die Form

$$A \cdot (\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_n) = (\lambda_1 \mathbf{u}_1, \lambda_2 \mathbf{u}_2, \ldots, \lambda_n \mathbf{u}_n) \quad .$$

  Damit ist aber für jeden Spaltenvektor $\mathbf{u}_j$ die Eigenwertbedingung erfüllt, und man kann sagen:

  Die j-te Spalte der Transformationsmatrix U stellt den j-ten Eigenvektor der gegebenen Matrix A dar.

Zusammenfassung: Wenn es gelingt, die Transformationsmatrix U zu finden, ist das vollständige Eigenwertproblem für die Matrix A gelöst!

Welche Matrizen sind diagonalisierbar?

- Alle Matrizen, welche ein *vollständiges, linear unabhängiges* Eigenvektor-System haben.

- Insbesondere kann gezeigt werden, daß alle *normalen* Matrizen ein solches Eigenvektor-System haben; darüber hinaus sind die Eigenvektoren solcher Matrizen auch noch *orthonormal*, d.h. sie gehorchen der Beziehung
$$\mathbf{x}_i \cdot \mathbf{x}_j^* = \delta_{i,j} \,.$$
Da diese Eigenvektoren die Spaltenvektoren der Transformationsmatrix U sind, haben diese Matrizen im Falle (reeller bzw. komplexwertiger) Matrizen A die Eigenschaft
$$U\,U^T = E \qquad \text{bzw.} \qquad U\,U^\dagger = E\,,$$
d.h. sie sind *orthogonal* bzw. *unitär.*

Für normale Matrizen gilt also die wichtige Aussage:
Normale Matrizen können mittels orthogonaler bzw. unitärer Transformationen auf Diagonalform gebracht werden:
$$U^T\,A\,U = D \qquad \text{bzw.} \qquad U^\dagger\,A\,U = D\,. \tag{7.11}$$

7.2 Numerische Behandlung regulärer Eigenwertprobleme.

7.2.1 Allgemeines

Die numerische Behandlung von regulären Eigenwertproblemen stellt ein außerordentlich wichtiges und deshalb sehr gut entwickeltes Gebiet der Numerischen Mathematik dar. Typisch für dieses Gebiet ist die starke Spezialisierung der einzelnen Methoden. Der Benutzer muß also genau wissen, was er will, um das für sein Problem richtige Verfahren auszuwählen.

Einen kleinen Überblick über das vielfältige Software-Angebot bzgl. der Lösung von Eigenwertproblemen s. in Abschnitt 7.6.

Im folgenden soll versucht werden, Ihnen an Hand einiger einfacher und dennoch sehr effektiver numerischer Methoden die Grundideen der wichtigsten Verfahren klarzumachen. Eine auch nur annähernd vollständige Beschreibung der in der Literatur enthaltenen Methoden ist im Rahmen dieser LV. unmöglich und wird daher nicht angestrebt.

Im folgenden beschränken wir uns auf *reelle* Matrizen und beschreiben

- das Verfahren von v. Mises (Verfahren der Vektoriteration);

- das Jacobi-Verfahren;
- die Anwendung des Hyman-Verfahrens auf Hessenberg-Matrizen.

7.3 Das Verfahren von v. Mises.

Dieses iterative Verfahren liefert den betragsgrößten (bzw., wie später gezeigt wird, auch den betragskleinsten) Eigenwert einer reellen Matrix A.
Voraussetzung für die Anwendbarkeit dieses Verfahrens ist, daß die gegebene Matrix diagonalisierbar ist (d.h., ein linear unabhängiges Eigenvektorsystem hat), und daß ein betragsmäßig dominanter Eigenwert existiert:

$$| \lambda_1 | > | \lambda_2 | \geq | \lambda_3 | \geq \ldots \geq | \lambda_{n-1} | > | \lambda_n | \quad . \tag{7.12}$$

In Fall linear unabhängiger Eigenvektoren kann jeder Vektor $\mathbf{v}^{(0)}$ (mit Ausnahme des Nullvektors) als Linearkombination der Eigenvektoren dargestellt werden:

$$\mathbf{v}^{(0)} = \sum_{i=1}^{n} \alpha_i \mathbf{x}_i \qquad \text{mit} \qquad | \alpha_1 | + | \alpha_2 | + \ldots + | \alpha_n | \neq 0 \quad .$$

Der beliebige [1] Vektor $\mathbf{v}^{(0)}$ ist der Anfangsvektor für die folgende Iteration: $\mathbf{v}^{(0)}$ wird von links mit der Matrix A multipliziert:

$$A \cdot \mathbf{v}^{(0)} \equiv \mathbf{v}^{(1)} = \sum_{i=1}^{n} \alpha_i A \mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i \mathbf{x}_i \quad ,$$

wobei die letzte Identität aus der Eigenwertgleichung (7.5) folgt. Nun wird der Vektor $\mathbf{v}^{(1)}$ seinerseits von links mit der Matrix A multipliziert (*fortgesetzte Vektormultiplikation*):

$$A \cdot \mathbf{v}^{(1)} \equiv \mathbf{v}^{(2)} = \sum_{i=1}^{n} \alpha_i \lambda_i A \mathbf{x}_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 \mathbf{x}_i \quad .$$

Durch die ständige Wiederholung dieser Rechenschrittes erhält man eine Vektorfolge $\mathbf{v}^{(0)}$, $\mathbf{v}^{(1)}$, …, $\mathbf{v}^{(t)}$, $\mathbf{v}^{(t+1)}$, … mit

$$\mathbf{v}^{(t)} = \sum_{i=1}^{n} \alpha_i \lambda_i^t \mathbf{x}_i \qquad (t = 0, 1, \ldots) \quad . \tag{7.13}$$

Nun bedeutet jedoch die Gültigkeit von (7.12), daß mit Fortschreiten der Iteration der erste Summenterm in (7.13) wegen der immer höheren Potenzen von λ immer stärker dominiert. Für nicht zu kleine t-Werte gilt also in guter Näherung

$$\mathbf{v}^{(t)} \approx \alpha_1 \lambda_1^t \mathbf{x}_1 \qquad \text{und} \qquad \mathbf{v}^{(t+1)} \approx \alpha_1 \lambda_1^{t+1} \mathbf{x}_1 \quad . \tag{7.14}$$

Die obigen Gleichungen sind Vektorgleichungen, d.h. sie müssen für alle Komponenten gelten, so z.B. auch für die l-te Komponente. Es gilt demnach

$$\frac{v_l^{(t+1)}}{v_l^{(t)}} \approx \lambda_1 \qquad \text{für alle} \qquad l = 1, 2, \ldots, n \quad . \tag{7.15}$$

[1] vgl. aber Abschnitt 7.3.5.

D.h.: Im Falle der Konvergenz strebt das Verhältnis der gleichen Komponenten aufeinanderfolgender Vektoren $\mathbf{v}^{(t)}$ dem betragsgrößten Eigenwert zu:

$$\lim_{t\to\infty} \frac{v_l^{(t+1)}}{v_l^{(t)}} \to \lambda_1 \quad . \tag{7.16}$$

Ebenso ergibt sich unmittelbar aus (7.14), daß gilt:

$$\lim_{t\to\infty} \mathbf{v}^{(t)} \to \quad \text{prop.} \quad \mathbf{x}_1 \quad . \tag{7.17}$$

Im Prinzip könnte man ein beliebiges l auswählen und die Gleichung (7.15) auswerten. Nun ist jedoch nicht auszuschließen, daß während der Iteration ein $v_l^{(t)}$ Null bzw. sehr klein wird. Um die dadurch auftretenden Probleme zu vermeiden, berechnet man anstelle von (7.15) den Ausdruck

$$\frac{1}{n'} \sum_{\mu} \frac{v_\mu^{(t+1)}}{v_\mu^{(t)}} \approx \lambda_1 \quad , \tag{7.18}$$

wobei über alle Indizes μ summiert wird, für die gilt:

$$|v_\mu^{(t)}| > \epsilon \, . \tag{7.19}$$

n' ist die Anzahl der Summanden, die diese Bedingung erfüllen.

7.3.1 Die Berechnung des betragskleinsten Eigenwertes

einer reellen Matrix ist für viele praktische Anwendungen von größerem Interesse als die Kenntnis des betragsgrößten Eigenwertes. Das Verfahren von v. Mises ist in der Lage, auch dieses Problem zu lösen.

Ausgehend von der Eigenwertgleichung (7.5)

$$A \cdot \mathbf{x}_i = \lambda_i \cdot \mathbf{x}_i$$

erhält man durch Multiplikation mit der zu A inversen Matrix A^{-1}:

$$\mathbf{x}_i = \lambda_i \cdot A^{-1}\mathbf{x}_i \quad .$$

Die *Eigenwertgleichung für die inverse Matrix* lautet demnach einfach

$$A^{-1} \cdot \mathbf{x}_i = \frac{1}{\lambda_i} \cdot \mathbf{x}_i \quad . \tag{7.20}$$

Das heißt:

- Die Eigenwerte einer Matrix A^{-1} sind einfach die Kehrwerte der Eigenwerte von A.

- Die Matrizen A^{-1} und A haben dasselbe Eigenvektor-System.

Nachdem aber die Kehrwerte der betragskleinsten Eigenwerte von A natürlich die betragsgrößten Eigenwerte von A^{-1} sind, braucht man nur das v. Mises-Verfahren auf die Inverse der gegebenen Matrix anzuwenden, um den betragskleinsten Eigenwert von A zu erhalten.

Es gibt aber noch eine etwas andere Methode, die Mises-Iteration für den betragskleinsten Eigenwert durchzuführen. Zu diesem Zweck braucht man nur die Iterationsvorschrift für die *Inverse* von A umzuformen:

$$\mathbf{v}^{(t+1)} = A^{-1}\,\mathbf{v}^{(t)}\,.$$

Multipliziert man diese Gleichung von links mit A, so erhält man

$$A\,\mathbf{v}^{(t+1)} = \mathbf{v}^{(t)}\,. \tag{7.21}$$

Dies kann aber als lineares Gleichungssystem mit der Koeffizientenmatrix A, dem inhomogenen Vektor $\mathbf{v}^{(t)}$ und dem Lösungsvektor $\mathbf{v}^{(t+1)}$ interpretiert werden. Wenn man zur Lösung dieses Systems die in Kapitel 2 beschriebene LU-Decomposition verwendet, hat man den Vorteil, daß man nur einmal (vor Beginn der Mises-Iteration) die zu A gehörende LU-Matrix berechnen muß, und danach während jedes Iterationsschrittes nur das Programm LUBKSB anwenden muß.
Auf diese Weise erhält man eine Serie von Vektoren mit der Eigenschaft

$$\lim_{t\to\infty} \frac{v_l^{(t)}}{v_l^{(t+1)}} = \lambda_n\,, \tag{7.22}$$

wobei λ_n der betragskleinste Eigenvektor von A ist.

7.3.2 Konvergenzsteigerung durch Spektralverschiebung

Auch ohne mathematische Beweisführung leuchtet ohne weiteres ein, daß die Konvergenz der Iteration (7.22) umso besser sein wird, je größer das Verhältnis

$$\frac{1}{\lambda_n} \Big/ \frac{1}{\lambda_{n-1}} = \frac{\lambda_{n-1}}{\lambda_n}$$

ist. Auf diesem Faktum beruht die folgende Idee:

Angenommen, man kennt mit λ_0 einen brauchbaren Schätzwert für den betragskleinsten Eigenwert λ_n. Nimmt man nun anstelle der gegebenen Matrix A die Matrix

$$A' = A - \lambda_0\,E\,,$$

so verschieben sich alle Eigenwerte um den Wert λ_0, also auch die Eigenwerte λ_{n-1} und λ_n, wobei gilt:

$$\frac{\lambda_{n-1} - \lambda_0}{\lambda_n - \lambda_0} > \frac{\lambda_{n-1}}{\lambda_n}\,.$$

Dementsprechend ist zu erwarten, daß der Grenzwert

$$\lim_{t\to\infty} \frac{v_l^{(t)}}{v_l^{(t+1)}} + \lambda_0 = \lambda_n \tag{7.23}$$

bedeutend rascher approximiert wird als der Grenzwert (7.22).

Die Überlegungen der Abschnitte 7.3.1 und 7.3.2 werden im folgenden Programm MISES realisiert.

7.3.3 Das Programm MISES

Quelle: [2], S.94ff, Programm S.331f mit Änderungen.

Das Programm MISES berechnet den betragskleinsten Eigenwert und den zugehörigen Eigenvektor einer reellen Matrix (inklusive Spektralverschiebung).

INPUT-Parameter:

A(,): reelle Matrix, deren betragskleinster Eigenwert zu bestimmen ist.

N: Ordnung der Matrix.

TMAX: maximale Anzahl von Iterationsschritten.

GEN: relative Genauigkeit für den Eigenwert.

V( ): Startvektor $\neq$ Nullvektor.

EIGW0: Schätzwert für den betragskleinsten Eigenwert.

OUTPUT-Parameter:

EIGW: Näherung für den betragskleinsten Eigenwert.

V( ): auf den Betrag 1 normierter, zu λ_1 gehörender Eigenvektor.

FEHLER: logische Fehlervariable: TRUE wenn keine Konvergenz innerhalb von TMAX ereicht wurde, sonst FALSE.

INTERNE Parameter:

W: Hilfsvektor.

EPS: Konstante s. Glg. (7.19).

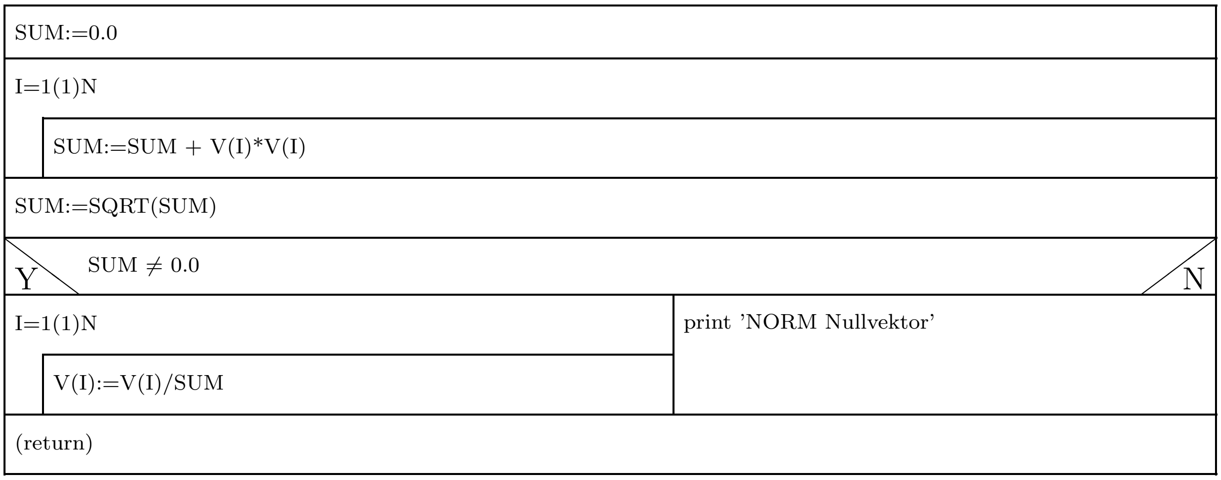
VERWENDETE PROZEDUREN:

Das Programm MISES braucht die Prozeduren NORM sowie die Prozeduren LUDCMP und LUBKSB (s. Kapitel 2).

Struktogramm 21 — MISES(A,N,TMAX,GEN,V,EIGW0,EIGW,FEHLER)

```
EPS:=1.0E-6
FEHLER:=TRUE
EIGALT:=0.0
I=1(1)N
    A(I,I):=A(I,I) - EIGW0
LUDCMP(A,N,INDX,D,KHAD)
T:=0
    NORM(V,N)
    LUBKSB(A,N,INDX,V,W)
    SUM:=0.0
    N1:=0
    I=1(1)N
        | W(I)| > EPS
        Y:
            SUM:=SUM + V(I)/W(I)
            N1:=N1 + 1
        N:
            ......
    EIGW:=SUM/N1
    EIGW:=EIGW + EIGW0
    |(EIGW-EIGALT)/EIGW| < GEN
    Y:
        FEHLER:=FALSE
    N:
        I=1(1)N
            V(I):=W(I)
        EIGALT:=EIGW
    T:=T+1
T ≥ TMAX .or. notFEHLER
FEHLER
Y:
    print 'MISES1 keine Konvergenz'
N:
    NORM(V,N)
(return)
```

Struktogramm 21a — NORM(V,N)



7.3.4 v. Mises-Verfahren: Tests und Probleme.

Die folgenden Test-Beispiele sollen dazu dienen, die Zuverlässigkeit des Programms MISES zu überprüfen.

Die Tests beginnen mit der 4x4-Matrix

```
3.8000    1.8000   -2.0000   -0.6000
5.4000    6.2000   -7.2000   -1.0000
2.0000    2.4000   -2.0000    0.0000
1.8000    1.0000    0.0000    1.0000
```

Es soll der betragskleinste Eigenwert bzw. der zugehörige Eigenvektor berechnet werden. Das exakte Ergebnis lautet

$$\lambda_4 = 0.6 \qquad \text{und} \qquad \mathbf{x}_4 = \frac{1}{\sqrt{23}} \begin{pmatrix} 1 \\ -3 \\ -2 \\ 3 \end{pmatrix} = \begin{pmatrix} 0.2085144 \\ -0.6255432 \\ -0.4170288 \\ 0.6255432 \end{pmatrix} .$$

Es soll nun die Wirkung der Spektralverschiebung (s. Abschnitt 7.3.3) getestet werden. Der erste Test erfolgte *ohne* diese Technik, d. h. der Schätzwert für λ_4 wurde Null gesetzt:

```
.
MISES-Test:   n =    4   rel. Gen. =  1.0000000000E-06

Schaetzwert fuer EW =   0.000000

Matrix und Anfangsvektor:
```

```
    3.8000    1.8000   -2.0000   -0.6000       1.0000
    5.4000    6.2000   -7.2000   -1.0000       1.0000
    2.0000    2.4000   -2.0000    0.0000       1.0000
    1.8000    1.0000    0.0000    1.0000       1.0000

betragskleinster Eigenwert nach 20 Iterationen =   0.600000

                 Eigenvektor:
       1     -0.208514
       2      0.625543
       3      0.417029
       4     -0.625543
```

Nimmt man als Schätzwert $\lambda_0 = 0.5$, so liefert das Programm MISES1:

```
.
MISES-Test:   n =    4   rel. Gen. =  1.0000000000E-06

Schaetzwert fuer EW =   0.500000

Matrix und Anfangsvektor:

    3.8000    1.8000   -2.0000   -0.6000       1.0000
    5.4000    6.2000   -7.2000   -1.0000       1.0000
    2.0000    2.4000   -2.0000    0.0000       1.0000
    1.8000    1.0000    0.0000    1.0000       1.0000

betragskleinster Eigenwert nach 8 Iterationen =   0.600000

                 Eigenvektor:
       1     -0.208514
       2      0.625543
       3      0.417029
       4     -0.625543
```

Anmerkung:
Wie Sie sehen, gibt MISES1 nicht den theoretisch vorgegebenen Eigenvektor, sondern diesen Vektor mal dem Faktor (–1). Liegt hier ein Fehlverhalten des Programmes vor?

Zuletzt noch ein MISES1-Test für die 3x3-Matrix

```
    1.0000    0.0000   -1.0000
    1.0000    2.0000    1.0000
   -2.0000   -2.0000    2.0000
```

mit den entarteten Eigenwerten λ_1 und λ_2. Der dritte (und betragskleinste) Eigenwert bzw. sein Eigenvektor lauten:

$$\lambda_3 = 1.0 \qquad \text{und} \qquad \mathbf{x}_3 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.7071068 \\ -0.7071068 \\ 0.0000000 \end{pmatrix} .$$

```
.
MISES-Test:   n =    3   rel. Gen. =  1.0000000000E-06

Schaetzwert fuer EW =   0.000000

Matrix und Anfangsvektor:

    1.0000    0.0000   -1.0000       1.0000
    1.0000    2.0000    1.0000       0.0000
   -2.0000   -2.0000    2.0000       0.0000

betragskleinster Eigenwert nach 24 Iterationen =   1.000000

                 Eigenvektor:
       1      0.707107
       2     -0.707107
       3      0.000000
```

7.4 Das Verfahren von Jacobi.

Dieses Verfahren ist geeignet, *das vollständige Eigenwertproblem symmetrischer Matrizen zu lösen.*
Gegeben ist also eine reelle Matrix

$$A = [a_{ij}] \qquad \text{mit} \qquad a_{ij} = a_{ji} \quad .$$

Es ist eine wesentliche allgemeine Eigenschaft reeller symmetrischer Matrizen, daß sie *ausschließlich reelle Eigenwerte* haben.

Das Prinzip des Jacobi-Verfahrens besteht aus einer orthogonalen Transformation der Matrix A in eine Diagonalmatrix D (vgl. Kap. 7.1.1):

$$U^T \cdot A \cdot U = D \tag{7.24}$$

Wie bereits ausführlich diskutiert, ist bei Kenntnis der Matrix U das Eigenwert- und Eigenvektor-Problem von A vollständig gelöst:

- Die Diagonalelemente von D sind die Eigenwerte von A:

$$\lambda_i = d_{ii} \qquad (i = 1, \ldots, n)\,.$$

- Die Spalten der Transformationsmatrix U sind die (auf Betrag Eins normierten) Eigenvektoren von A.

7.4.1 Eine iterative Annäherung an die Transformationsmatrix U.

Das Problem besteht nun darin, die Transformationsmatrix U zu ermitteln. Beim Jacobi-Verfahren wird die Transformation (7.24) durch eine sukzessive Folge von Ähnlichkeitsoperationen ersetzt. Selbstverständlich müssen auch die dabei verwendeten Transformationsmatrizen $U_t \quad (t = 0, 1, 2, \ldots)$ orthogonal sein.

Anstelle von (7.24) gilt also:

$$U_0^T A U_0 \equiv A^{(1)}$$

$$U_1^T A^{(1)} U_1 \equiv A^{(2)} = U_1^T U_0^T A U_0 U_1$$

$$\cdot$$

$$\cdot$$

$$U_t^T A^{(t)} U_t \equiv A^{(t+1)} = U_t^T U_{t-1}^T \cdots U_0^T A U_0 \cdots U_{t-1} U_t\,,$$

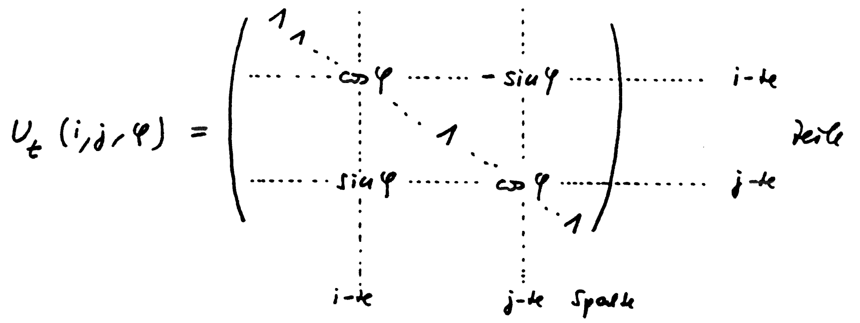
wobei die Einzel-Transformationen so beschaffen sein sollen, daß die Grenzwerte dieses Prozesses wie folgt lauten:

$$\lim_{t\to\infty} A^{(t)} \quad \to \quad D$$

$$\lim_{t\to\infty} U_0 U_1 \cdots U_{t-1} U_t \quad \to \quad U$$

Die beim Jacobi-Verfahren verwendeten Matrizen U_t haben die allgemeine Form einer *orthogonalen Drehmatrix*

$$. \qquad (7.25)$$



Wie man sofort sieht, ist U_t durch drei Angaben eindeutig bestimmt:
i und j geben jene Zeilen bzw. Spalten der Matrix an, an deren Schnittpunkten sich die Drehmatrix von einer Einheitsmatrix unterscheidet. φ ist der Drehwinkel. Es kann leicht gezeigt werden, daß $U(i,j,\varphi)$ – unabhängig von den Argumenten – stets orthogonal ist, d.h. daß gilt:

$$U_t^T(i,j,\varphi) \cdot U_t(i,j,\varphi) = E$$

Bevor nun über die optimale Wahl der Parameter i, j, φ für jeden Iterationsschritt gesprochen wird, soll im Detail untersucht werden, was bei einer Transformation

$$U^T(i_{t-1}, j_{t-1}, \varphi_{t-1}) \cdot A^{(t-1)} \cdot U(i_{t-1}, j_{t-1}, \varphi_{t-1}) \quad \rightarrow \quad A^{(t)} \qquad (7.26)$$

geschieht. Komponentenweise ausgeschrieben, entspricht die Transformation dem Ausdruck

$$a_{kl}^{(t)} = \sum_{m=1}^{n} \sum_{m'=1}^{n} u_{mk} u_{m'l} a_{mm'}^{(t-1)} \quad . \qquad (7.27)$$

Wie man der obigen Gleichung sofort entnehmen kann, bleibt die Symmetrie der Matrix $A^{(t-1)}$ bei dieser Transformation erhalten; es gilt also

$$a_{kl}^{(t)} = a_{lk}^{(t)} \quad .$$

Als nächstes soll (7.27) unter der Bedingung ausgewertet werden, daß *k und l weder i noch j sind*. Unter diesen Umständen sind die Komponenten der Drehmatrix (7.25) einfach 'Kronecker-Deltas', und man erhält

$$a_{kl}^{(t)} = \sum_{m=1}^{n} \sum_{m'=1}^{n} \delta_{mk} \delta_{m'l} a_{mm'}^{(t-1)} = a_{kl}^{(t-1)} \quad .$$

Das bedeutet: *Bei der Transformation bleiben alle Komponenten von A, die weder den Index i noch den Index j enthalten, unverändert.*

Die noch fehlenden Komponenten aus der i-ten und j-ten Zeile und Spalte können ebenfalls unter Verwendung von (7.27) bestimmt werden:

$$l = i \qquad k = 1, \ldots, n \quad \text{mit} \quad k \neq i, j \quad :$$

$$a_{ki}^{(t)} = a_{ik}^{(t)} = a_{ki}^{(t-1)} \cos\varphi + a_{kj}^{(t-1)} \sin\varphi \tag{7.28}$$

$$l = j \qquad k = 1, \ldots, n \quad \text{mit} \quad k \neq i, j \quad :$$

$$a_{kj}^{(t)} = a_{jk}^{(t)} = a_{kj}^{(t-1)} \cos\varphi - a_{ki}^{(t-1)} \sin\varphi \tag{7.29}$$

Nun fehlen nur noch die Komponenten

$$a_{ii}^{(t)} \qquad a_{jj}^{(t)} \qquad a_{ij}^{(t)} = a_{ji}^{(t)} \quad .$$

Für diese lauten die Transformationsgleichungen:

$$a_{ii}^{(t)} = a_{ii}^{(t-1)} \cos^2\varphi + 2a_{ij}^{(t-1)} \cos\varphi \sin\varphi + a_{jj}^{(t-1)} \sin^2\varphi \tag{7.30}$$

$$a_{jj}^{(t)} = a_{jj}^{(t-1)} \cos^2\varphi - 2a_{ij}^{(t-1)} \cos\varphi \sin\varphi + a_{ii}^{(t-1)} \sin^2\varphi \tag{7.31}$$

$$a_{ij}^{(t)} = a_{ij}^{(t-1)} (\cos^2\varphi - \sin^2\varphi) + (a_{jj}^{(t-1)} - a_{ii}^{(t-1)}) \cos\varphi \sin\varphi \tag{7.32}$$

7.4.2 Die richtige Wahl der Parameter i, j und φ.

Der Zweck jeder der orthogonalen Transformationen, der die Matrix A unterzogen wird, ist der, die entstehenden Matrizen

$$A^{(1)}, A^{(2)}, \ldots, A^{(t-1)}, A^{(t)}, A^{(t+1)}, \ldots$$

iterativ der gewünschten Diagonalform immer näher zu bringen.

Ein Maß dafür, inwieweit dies gelungen ist, ist offenbar *die Summe S der Quadrate der Nicht-Diagonalelemente der Matrix*:

$$S^{(t)} = 2 \sum_{m=1}^{n-1} \sum_{m'=m+1}^{n} \left(a_{mm'}^{(t)} \right)^2 \quad .$$

Wenn nun im Laufe der Jacobi-Iteration die Transformation

$$U_{t-1}^T \cdot A^{(t-1)} \cdot U_{t-1} \quad \rightarrow \quad A^{(t)}$$

durchgeführt wird, so ist diese dann als erfolgreich zu bezeichnen, wenn

$$S^{(t-1)} > S^{(t)} \quad ,$$

gilt, wobei $S^{(t-1)}$ und $S^{(t)}$ die Summen der Quadrate der Nicht-Diagonalelemente der Matrizen $A^{(t-1)}$ und $A^{(t)}$ sind.

Ohne Beweis: Es läßt sich zeigen, daß die Differenz von $S^{(t-1)}$ und $S^{(t)}$ einfach dem Ausdruck

$$S^{(t-1)} - S^{(t)} = 2 \left[\left(a_{ij}^{(t-1)} \right)^2 - \left(a_{ij}^{(t)} \right)^2 \right] \tag{7.33}$$

entspricht, also nur durch die Elemente a_{ij} *vor und nach* der Transformation bestimmt wird.

Die Gleichung (7.33) führt zur sinnvollsten Strategie bei der Auswahl der noch offenen Parameter i, j und φ für einen bestimmten Iterationsschritt: *Die Verkleinerung von S ist dann ein Maximum,*

- *wenn $a_{ij}^{(t-1)}$ dem Betrage nach möglichst groß ist und*
- *wenn $a_{ij}^{(t)}$ durch die Transformation zu Null wird.*

i und j sind also einfach die Indizes der betragsgrößten Nicht-Diagonalelemente der Matrix A vor dem jeweiligen Transformationsschritt[2], und die zweite Bedingung kann durch eine geeignete Wahl des Drehwinkels φ erfüllt werden. Durch das Nullsetzen von (7.32) erhält man[3]

$$\tan 2\varphi = \frac{2a_{ij}^{(t-1)}}{a_{ii}^{(t-1)} - a_{jj}^{(t-1)}} \quad . \tag{7.34}$$

Als Sonderfall ergibt sich für $a_{ii}^{(t-1)} = a_{jj}^{(t-1)}$ der Drehwinkel

$$\varphi = \frac{\pi}{4} \quad .$$

Wählt man nun vor jedem Iterationsschritt die Parameter der Drehmatrix (7.25) auf die soeben beschriebene Weise, ist eine *monotone* Abnahme von S

$$S^{(0)} > S^{(1)} > S^{(2)} > \cdots > S^{(t)} > \cdots$$

und dadurch eine immer stärkere Annäherung an die gewünschte Diagonalform garantiert.

Zusätzlich soll auch noch jene Matrix berechnet werden, die sich aus dem Produkt der einzelnen Transformationsmatrizen ergibt:

$$B^{(t-1)} = U_0 U_1 U_2 \cdots U_{t-2} U_{t-1} \quad , \tag{7.35}$$

weil die Matrix näherungsweise die gesuchten Eigenvektoren der Matrix A enthält. Wieder kann gezeigt werden, daß bei der Multiplikation von B mit einer Drehmatrix $U_t(i,j,\varphi)$ nur die Elemente der i-ten und j-ten Spalte von B verändert werden:

$$\Big(B^{(t-1)} U_t(i,j,\varphi)\Big)_{k,i} = b_{ki}\cos\varphi + b_{kj}\sin\varphi \tag{7.36}$$

$$\Big(B^{(t-1)} U_t(i,j,\varphi)\Big)_{k,j} = b_{kj}\cos\varphi - b_{ki}\sin\varphi \tag{7.37}$$

für $k = 1, \ldots, n$.

[2] Allerdings ist das Aufsuchen des betragsgrößten Nicht-Diagonalelementes vor jeder Matrix-Transformation sehr zeitintensiv. Aus diesem Grund wird in den meisten Programmen ein etwas einfacheres Auswahlverfahren angewendet, s. Abschnitt 7.4.3.

[3] Eine Variation dieser Vorgangsweise für C-Programme s. Abschnitt 7.4.4.

7.4.3 Das Programm JACOBI.

Das Programm JACOBI löst das vollständige Eigenwertproblem einer reellen, symmetrischen Matrix.

INPUT-Parameter:

A(,): die Komponenten einer reellen, symmetrischen Matrix. Da das Programm JACOBI so formuliert ist, daß immer nur auf die Matrixelemente auf und oberhalb der Hauptdiagonale zugegriffen wird, brauchen *nur diese Elemente vom aufrufenden Programm bereitgestellt werden.*

N: Zeilen und Spalten der Matrix.

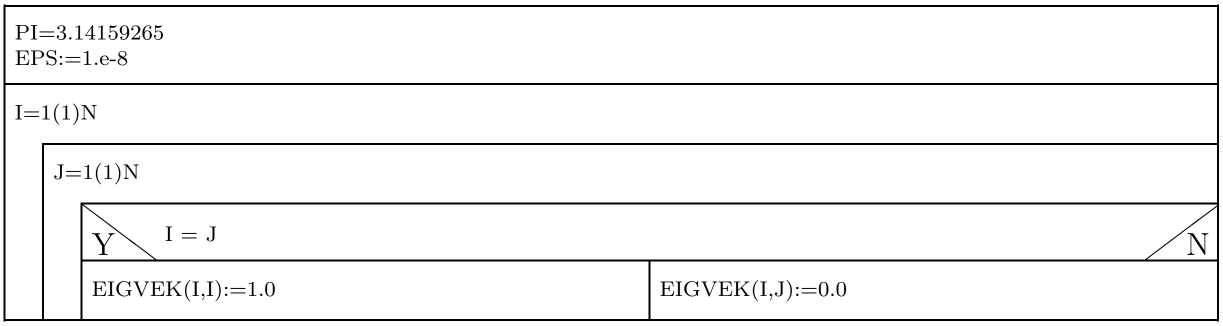
TMAX: maximale Anzahl der Rechendurchläufe.

OUTPUT-Parameter:

A(,): Näherung für eine Diagonalmatrix mit demselben Eigenwertspektrum wie die ursprüngliche Matrix A. Die Diagonalelemente dieser Matrix sind somit Näherungswerte für die Eigenwerte.

EIGVEK(,): Diese Matrix enthält die Eigenvektoren der Matrix A, wobei die j-te Spalte von EIGVEK dem j-ten Eigenvektor von A entspricht.

Struktogramm 22 — JACOBI(A,N,TMAX,EIGVEK)



Struktogramm 23 — (Fortsetzung)

```
T:=0
  T:=T+1
  SUMME:=0.0
  I=1(1)N-1
    J=I+1(1)N
      SUMME:=SUMME+2.0*A(I,J)*A(I,J)
  IF(SUMME=0.0) ===> (return)
  GRENZE:=SQRT(SUMME)/N/N
  I=1(1)N-1
    J=I+1(1)N
      | A(I,J) | > GRENZE
      Y:
        NENNER:=A(I,I)–A(J,J)
        | NENNER/A(I,J) | < EPS
        Y: PHI:=PI/4.0
        N: PHI:=0.5*ATAN(2.0*A(I,J)
           /NENNER)
        SINUS:=SIN(PHI)
        COSIN:=COS(PHI)
        K=I+1(1)J-1
          SPEICH:=A(I,K)
          A(I,K):=COSIN*A(I,K)+SINUS*A(K,J)
          A(K,J):=COSIN*A(K,J)–SINUS*SPEICH
        K=J+1(1)N
          SPEICH:=A(I,K)
          A(I,K):=COSIN*A(I,K)+SINUS*A(J,K)
          A(J,K):=COSIN*A(J,K)–SINUS*SPEICH
        K=1(1)I-1
          SPEICH:=A(K,I)
          A(K,I):=COSIN*A(K,I)+SINUS*A(K,J)
          A(K,J):=COSIN*A(K,J)–SINUS*SPEICH
        SPEICH:=A(I,I)
        A(I,I):=COSIN**2*A(I,I)+2.0*COSIN*SINUS*A(I,J)+SINUS**2*A(J,J)
        A(J,J):=COSIN**2*A(J,J)–
        2.0*COSIN*SINUS*A(I,J)+SINUS**2*SPEICH
        A(I,J):=0.0
        K=1(1)N
          SPEICH:=EIGVEK(K,J)
          EIGVEK(K,J):=COSIN*EIGVEK(K,J)–SINUS*EIGVEK(K,I)
          EIGVEK(K,I):=COSIN*EIGVEK(K,I)+SINUS*SPEICH
      N:
        ......
T > TMAX
print 'JACOBI Konvergenz nicht erreicht'
(return)
```

Programm-Struktur:

1. Nach der Definition der Konstanten PI und der 'Pseudonull' EPS=10^{-8} wird auf das Feld EIGVEK eine Einheitsmatrix abgespeichert.

2. UNTIL-Schleife für die (maximal) TMAX Rechendurchläufe:

   - Bestimmung der Summe der Quadrate der Nicht-Diagonalelemente sowie des 'Schwellenwertes' GRENZE.
   - Abfrage, ob diese Summe bereits Null ist.

     Die Konvergenz des Jacobi-Verfahrens ist zumeist so gut, daß man die Rechnung bis zum 'Underflow' von SUMME treiben kann. Die obige Abfrage auf die exakte Null funktioniert jedoch nur dann, wenn vom System her ein Underflow als Null interpretiert wird!

     Wenn SUMME=0.0, Return ins aufrufende Programm.

     Wenn SUMME>0.0, werden Matrix-Transformationen gemäß (7.28 - 7.32) bzgl. *all jener Nicht-Diagonalelemente durchgeführt, deren Beträge > GRENZE sind.*
   - Nach jeder Transformation wird die Matrix EIGVEK mit der bei diesem Iterationsschritt verwendeten orthogonalen Matrix U_t multipliziert (7.36 und 7.37).

3. Ist nach TMAX Rechendurchgängen SUMME immer noch größer als Null, erfolgt eine Fehlermitteilung und ein Return ins aufrufende Programm.

7.4.4 Variation des JACOBI-Algorithmus in C.

In FORTRAN-, PASCAL-, ... -Programmen kann die Berechnung des idealen Drehwinkels gemäß Glg. (7.34) erfolgen.
In Programmiersprachen ohne Arcustangens-Funktion wie z. B. in C kann die Berechnung von $\cos\varphi$ und $\sin\varphi$ auf die folgende Weise geschehen (mathematische Darstellung s. [10], S. 464f):

```
.
  g=100*fabs(a[i][j]);
  if(fabs(a[i][j]) > grenze) {
    h=a[i][i]-a[j][j];
// This if statement prevents an overflow of theta*theta in the following
// statement.
    if(fabs(h)+g == fabs(h)) tfac=a[i][j]/h;
    else {
      theta=h/2.0/a[i][j];
      tfac=1.0/(fabs(theta)+sqrt(1.0+theta*theta));
      if(theta<0.0)tfac=-tfac;
    }
    cosin=1.0/sqrt(tfac*tfac+1.0);
    sinus=tfac*cosin;
```

7.4.5 Zwei Testbeispiele für JACOBI.

Erstes Beispiel:

```
.
Matrix:   5.000   4.000   1.000   1.000
          4.000   5.000   1.000   1.000
          1.000   1.000   4.000   2.000
          1.000   1.000   2.000   4.000

exakte Loesung:
Eigenwert:                    Eigenvektor:

10  (2,2,1,1)/sqrt(10)   = (0.6324555, 0.6324555, 0.3162278, 0.3162278)
 1  (-1,1,0,0)/sqrt(2)   = (-0.7071068, 0.7071068, 0.0, 0.0)
 5  (-1,-1,2,2)/sqrt(10) = (-0.3162278, -0.3162278, 0.6324555, 0.6324555)
 2  (0,0,-1,1)/sqrt(2)   = (0.0, 0.0, -0.7071068, 0.7071068)

Jacobi-Loesung:
Eigenwert:                    Eigenvektor:

10.000000       0.632456    0.632456    0.316228    0.316228
 1.000000      -0.707107    0.707107   -0.000000   -0.000000
 5.000000      -0.316228   -0.316228    0.632456    0.632456
 2.000000      -0.000000   -0.000000   -0.707107    0.707107
```

Zweites Beispiel:

```
.
Matrix:   6.000   4.000   4.000   1.000
          4.000   6.000   1.000   4.000
          4.000   1.000   6.000   4.000
          1.000   4.000   4.000   6.000

exakte Loesung:
Eigenwert:                    Eigenvektor:
15              0.5         0.5         0.5         0.5
-1              0.5        -0.5        -0.5         0.5
 5             -0.5         0.5        -0.5         0.5
 5             -0.5        -0.5         0.5         0.5

Jacobi-Loesung:
Eigenwert:                    Eigenvektor:

15.000000       0.500000    0.500000    0.500000    0.500000
-1.000000      -0.500000    0.500000    0.500000   -0.500000
 5.000000       0.226248   -0.669934    0.669934   -0.226248
 5.000000      -0.669934   -0.226248    0.226248    0.669934
```

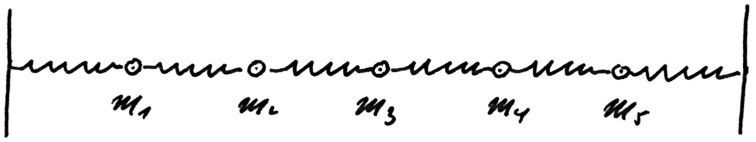
Zu diesen Ergebnissen noch zwei Anmerkungen:

1. Die normierten Eigenvektoren sind natürlich nur bis auf eine etwaige (multiplikative) Konstante (–1) bestimmt. Es ist daher *kein Fehler*, wenn das JACOBI-Programm den Eigenvektor zum Eigenwert -1 mit verändertem Vorzeichen ausgibt!

2. Noch stärker wirkt sich die Nicht-Eindeutigkeit von Eigenvektoren im Falle entarteter Eigenwerte aus. Im Fall des hier vorkommenden zweifach entarteten Eigenwertes 5 bedeutet das, daß jede Linearkombination der von JACOBI gelieferten Eigenvektoren wieder einen Eigenvektor zum Eigenwert 5 ergibt. Sie können sich ohne weiteres überzeugen, daß es möglich ist, die beiden Vektoren $(-0.5, 0.5, -0.5, 0.5)$ und $(-0.5, -0.5, 0.5, 0.5)$ so linear zu kombinieren, daß man die mittels JACOBI berechneten Eigenvektoren erhält.

7.4.6 Ein Anwendungsbeispiel für JACOBI.

Gegeben ist ein System von n gekoppelten Federpendeln, wobei die schwingenden Körper die Massen m_i, $i = 1, \ldots, n$ haben:

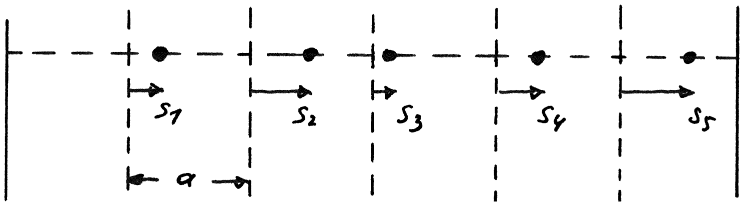


Für die weitere Diskussion werden die folgenden Annahmen gemacht:

- Die Federkräfte seien so groß bzw. die Massen so klein, daß Gravitationskräfte vernachlässigt werden können.

- Die Schwingungsamplituden seien klein gegenüber den Abständen der Teilchen-Ruhepositionen.

Die momentanen Auslenkungen der Massenpunkte werden mit s_i bezeichnet, und die Kräfte zwischen den Massen werden im Sinne einer *harmonischen Näherung* als proportional zu ihren Abständen angenommen. Der Proportionalitätsfaktor zwischen Abstand und Kraft, die *Federkonstante D*, wird in cgs-Einheiten (dyn/cm) angegeben. a ist der Abstand zwischen 2 Massepunkten in der Ruhelage:



Wird ein solches System in Schwingungen versetzt und dann sich selbst überlassen, so läßt sich für jeden Massenpunkt eine Bewegungsgleichung der Art

$$m_i \ddot{s}_i + D(a - s_{i-1} + s_i) - D(a - s_i + s_{i+1}) = 0$$

aufstellen (m_i ist die Masse des i-ten Massenpunktes, und $s_i(t)$, $s_{i-1}(t)$, $s_{i+1}(t)$ sind die momentanen Auslenkungen des i-ten Punktes sowie seines linken und rechten Nachbarn). Man erhält schließlich ein *gekoppeltes System von n gewöhnlichen Differentialgleichungen zweiter Ordnung*:

$$\begin{aligned} m_1\ddot{s}_1 + 2Ds_1 - Ds_2 &= 0 \\ m_i\ddot{s}_i - Ds_{i-1} + 2Ds_i - Ds_{i+1} &= 0 \qquad i = 2, 3, \ldots, n-1 \\ m_n\ddot{s}_n - Ds_{n-1} + 2Ds_n &= 0 \end{aligned} \tag{7.38}$$

Beschränkt man sich bei der Lösung dieses Systems auf kleine Auslenkungen, so kann das System (7.38) unter Verwendung der Ansatzfunktion

$$s_i(t) = \frac{b_i}{\sqrt{m_i}} \cdot e^{i\omega t}$$

in das *homogene, lineare Gleichungssystem*

$$\begin{aligned} \left(\frac{2D}{m_1} - \omega^2\right) b_1 - \frac{D}{\sqrt{m_1 m_2}} b_2 &= 0 \\ -\frac{D}{\sqrt{m_i m_{i-1}}} b_{i-1} + \left(\frac{2D}{m_i} - \omega^2\right) b_i - \frac{D}{\sqrt{m_i m_{i+1}}} b_{i+1} &= 0 \quad (i = 2, 3, \ldots, n) \\ -\frac{D}{\sqrt{m_n m_{n-1}}} b_{n-1} + \left(\frac{2D}{m_n} - \omega^2\right) b_n &= 0 \end{aligned}$$

übergeführt werden. Auf diese Weise erhält man das *reguläre Eigenwertproblem*

$$\begin{pmatrix} \frac{2D}{m_1} & -\frac{D}{\sqrt{m_1 m_2}} & & \\ \ddots & \ddots & \ddots & \\ -\frac{D}{\sqrt{m_i \cdot m_{i-1}}} & \frac{2D}{m_i} & -\frac{D}{\sqrt{m_i \cdot m_{i+1}}} & \\ & \ddots & \ddots & \\ & -\frac{D}{\sqrt{m_n \cdot m_{n-1}}} & \frac{2D}{m_n} & \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} = \lambda \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_i \\ \vdots \\ b_n \end{pmatrix} \tag{7.39}$$

wobei $\omega^2 = \lambda$ gesetzt wurde. Die (reellen) Eigenwerte der obigen *symmetrisch-tridiagonalen Matrix* sind also die Quadrate der Eigenfrequenzen des schwingenden Systems (kleinste Frequenz=Grundschwingung und $n-1$ Oberschwingungen).

```
Testbeispiel "Federpendel" fuer Jacobi:
=======================================

Federkonstante [dyn/cm] = 25.000
5 Massenpunkte:
  m1 = 3 g   m2 = 6 g   m3 = 9 g   m4 = 2 g   m5 = 6 g

JACOBI liefert das folgende Resultat:

   nr    lambda(1/s/s)     Kreisfrequenz(1/s)

   1      19.858498             4.456287
   2       1.135214             1.065464 (Grundschwingung)
   3       5.525477             2.350633
   4      29.036367             5.388540
   5       8.333333             2.886751
```

7.4.7 Erweiterte symmetrische Eigenwertprobleme

Viele interessante physikalisch-technische Problemstellungen führen nicht unmittelbar zu einem Eigenwertproblem vom Typus (7.5), sondern auf ein homogenes lineares Gleichungssystem in der Art

$$(A - \lambda S)\mathbf{x} = 0 \tag{7.40}$$

mit der symmetrischen Matrix A und der ebenfalls symmetrischen und zusätzlich noch *positiv-definiten* Matrix S (in manchen physikalischen Anwendungen wird S als *Strukturmatrix* bezeichnet).

Wie kann man nun das *erweiterte Eigenwertproblem* (7.40) auf die Form

$$(A' - \lambda E)\mathbf{x} = 0$$

bringen? Diese Frage scheint einfach zu lösen zu sein: man multipliziert die Glg. (7.40) von links mit der Inversen von S und erhält sofort

$$\left(S^{-1}A - \lambda E\right)\mathbf{x} = 0$$

mit der neuen Koeffizientenmatrix $A' = S^{-1}A$.

Leider ist diese simple Umformung nicht sehr brauchbar, weil nämlich die neue Koeffizentenmatrix A' die wichtige Eigenschaft der Symmetrie verloren hat:

$$A'_{i,j} = \sum_{k=1}^{n} s^{-1}_{i,k}\, a_{k,j}$$

ist im allgemeinen nicht identisch mit

$$A'_{j,i} = \sum_{k=1}^{n} s^{-1}_{j,k}\, a_{k,i}\,.$$

Eine bessere, allerdings etwas aufwändigere Methode der Umformung soll nun präsentiert werden:

Ausgangspunkt ist die sogenannte *Cholesky decomposition* der symmetrischen, positiv-definiten Matrix S:

$$S = L\,L^T \tag{7.41}$$

Beachten Sie die Ähnlichkeit dieser Aufspaltung mit der im Kapitel 2 behandelten *LU decomposition* (2.4): der Unterschied besteht nur darin, daß eine symmetrische, positiv definite Matrix als Produkt einer unteren Dreiecksmatrix L und deren um die Hauptachse transponierten oberen Dreieckmatrix L^T dargestellt werden kann.

Hat man die Cholesky-Prozedur erledigt, ist die weitere Vorgangsweise relativ einfach: Einsetzen von Glg. (7.41) in die erweiterte Eigenwertgleichung (7.40) gibt

$$\left(A - \lambda L L^T\right)\mathbf{x} = 0\,.$$

Hebt man (von rechts!) die Matrix L^T aus der Klammer heraus, folgt weiter

$$\left(A(L^T)^{-1} - \lambda L\right)\left(L^T\,\mathbf{x}\right) = 0$$

und

$$\left(\underbrace{L^{-1}A(L^T)^{-1}}_{C} - \lambda E\right)\left(L^T\mathbf{x}\right) = 0$$

bzw.[4]

$$(C - \lambda E)\mathbf{y} = 0 \qquad \text{mit} \quad C = L^{-1}A\left(L^{-1}\right)^T \qquad \text{und} \quad \mathbf{y} = L^T\mathbf{x}\,. \tag{7.42}$$

Die Eigenwerte der Matrix C (daß sie symmetrischi sind, ist leicht zu beweisen - probieren Sie es selbst) sind identisch mit den Eigenwerten von A in Glg. (7.40).

Aus den Eigenvektoren $\mathbf{y}$ des Systems (7.42) können mittels

$$\mathbf{x} = \left(L^{-1}\right)^T\,\mathbf{y} \tag{7.43}$$

die Eigenvektoren $\mathbf{x}$ von Glg. (7.40) bestimmt werden.

Die Formeln, die zur numerischen Ausführung der *Cholesky decomposition* erforderlich sind, können ohne Schwierigkeiten hergeleitet werden. Man geht dabei von der Komponentendarstellung von Glg. (7.41) aus:

$$s_{i,j} = \sum_{k=1}^{n} \ell_{ik}\ell_{jk} = \sum_{k=1}^{j} \ell_{ik}\ell_{jk}\,.$$

[4] Hier wird die Identität $\left(L^T\right)^{-1} = \left(L^{-1}\right)^T$ verwendet.

Die Berechnung der s_{ij} erfolgt nun spaltenweise (Index j), wobei wegen der Dreiecksform von L die Bedingung $i \leq j$ gilt.

Für die erste Spalte ($j = 1$) erhält man sofort

$$s_{11} = \ell_{11}^2 \rightarrow \ell_{11} = \sqrt{s_{11}}$$

sowie

$$s_{i1} = \ell_{i1}\ell_{11} \rightarrow \ell_{i1} = \frac{s_{i1}}{\ell_{11}} \quad \text{für} \quad i = 2, \ldots, n\,.$$

Analog erhält man für die zweite Spalte ($j = 2$)

$$s_{22} = \ell_{21}^2 + \ell_{22}^2 \rightarrow \ell_{22} = \sqrt{s_{22} - \ell_{21}^2}$$

und

$$s_{i2} = \ell i1\ell 21 + \ell_{i2}\ell_{22} \rightarrow \ell_{i2} = \frac{s_{i2} - \ell_{i1}\ell_{21}}{\ell_{22}} \quad \text{für} \quad i = 3, \ldots, n\,.$$

Aus diesen Gleichungen kann man ohne weiteres auf die allgemeinen Cholesky-Formeln schließen:

$$\ell_{jj} = \sqrt{s_{jj} - \sum_{t=1}^{j-1} \ell_{jt}^2}\,,$$

$$\ell_{i,j} = \frac{s_{ij} - \sum_{t=1}^{j-1} \ell_{it}\ell_{jt}}{\ell_{jj}} \quad \text{für} \quad (i = j+1, \ldots, n) \tag{7.44}$$

Zu diesem Gleichungen noch 2 Anmerkungen:

- Offensichtlich gibt es Probleme, wenn das Argument einer Quadratwurzel im Laufe der Cholesky-Rechnung Null oder negativ wird. Genau das kann man aber bei einer *positiv-definiten* Matrix S ausschließen. Im Programm muß also vor jeder Wurzelziehung abgefragt werden, ob

  $$\left(s_{jj} - \sum_{t=1}^{j-1} \ell_{jt}^2 \right) > 0\,.$$

  Ist dies nicht der Fall, ist die Matrix S nicht *positiv-definit* und die Rechnung muß abgebrochen werden!

- Eine genaue Analyse der Gleichungen (7.44) zeigt, daß die zur Berechnung der ℓ_{ij} erforderlichen s_{ij}-Werte jeweils nur einmal gebraucht werden und später nie wierder: das gibt die Möglichkeit, die Werte der L-Matrix (spaltenweise uns suzessive) auf den Speicherplätzen der S-Matrix abzuspeichern. Eine ähnlich platzsparende Vorgangsweise wurde auch im Programm LUDCMP realisiert.
  Nachteil: die ursprüngliche S-Matrix geht dabei verloren.

7.4.8 Das Programm CHOLESKY.

Die numerische Auswertung der Gleichungen (7.44) wird im folgenden Struktogramm Nr. 23 CHOLESKY beschrieben. Um genau zu sein, im ersten (kleineren) Teil dieses Programms. Der zweite Teil von CHOLESKY beschreibt die numerische Auswertung der *neuen* Matrix C gemäß Glg. (7.42). Den entsprechenden Algorithmus-Teil im Struktogramm 23 habe ich dem Algol-Programm *reduc1* von Martin und Wilkinson[5] entnommen. Ich muß gestehen, daß ich bisher noch keine Zeit fand, diesen Algorithmus schlüssig aus der Gleichung (7.42) nachzuvollziehen.

<u>INPUT-Parameter</u>:

N: Zeilen und Spalten der Matrizen A uns S.

A(,): die Komponenten einer reellen, symmetrischen Matrix.

S(,): die Komponenten einer reellen, symmetrischen, *positiv-definiten* Matrix.

<u>OUTPUT-Parameter</u>:

S(,): die Komponenten der tridiagonalen Matrix L.

A(,): die Komponenten der symmetrischen Matrix C.

FEHLER: Fehlerdiagnostik: FEHLER=0 kein Fehler aufgetreten
FEHLER=1 Achtung:
Input-Matrix S ist <u>nicht</u> positiv-definit.

[5] Martin and Wilkinson, Num. Math. **11**, 99 (1968); *Handbook for Autom. Computing*, vol. II, p. 303 (1971).

Struktogramm 23 — CHOLESKY(N,A,S,FEHLER)

```
FEHLER:=0
I=1(1)N
    J=I(1)N
        X:=S(I,J)
        K=1(1)I-1
            X:=X-S(I,K)*S(J,K)
        J ≠ I
        Y:  S(J,I):=X/Y
        N:  X ≤ 0.0
            Y:  FEHLER:=1
                print 'Matrix S nicht pos-def.'
                (return)
            N:  ......
            Y:=SQRT(X)
            S(I,I):=Y
Anmerkung: Ende der Cholesky Decomposition.
Anmerkung: Nun beginnt die Berechnung der Matrix C (Achtung: Abspeicherung auf A):
I=1(1)N
    Y:=B(I,I)
    J=I(1)N
        X:=A(I,J)
        K=1(1)I-1
            X:=X-S(I,K)*A(J,K)
        A(J,I):=X/Y
J=1(1)N
    I=J(1)N
        X:=A(I,J)
        K=J(1)I-1
            X:=X-A(K,J)*S(I,K)
        K=1(1)J-1
            X:=X-A(J,K)*S(I,K)
        A(I,J):=X/S(I,I)
I=1(1)N-1
    J=I+1(1)N
        A(I,J):=A(J,I)
        S(I,J):=0.0
```

```
TESTBEISPIEL:
=============

Numerische Loesung des erweiterten Eigenwertproblems

       [A - lambda . B] x  =  0

Matrix A (symmetrisch):
 5.0000   4.0000   1.0000   1.0000
 4.0000   5.0000   1.0000   1.0000
 1.0000   1.0000   4.0000   2.0000
 1.0000   1.0000   2.0000   4.0000

Matrix B (symmetrisch und positiv-definit):
 5.0000   7.0000   6.0000   5.0000
 7.0000  10.0000   8.0000   7.0000
 6.0000   8.0000  10.0000   9.0000
 5.0000   7.0000   9.0000  10.0000

Das Programm CHOLESKY gewinnt aus den gegebenen Matrizen
A und B mittels Cholesky-Aufspaltung von B:

               B = L . L^T

die Matrix

C = L^(-1) . A . (L^(-1))^T

Die untere Dreiecksmatrix  L  lautet:

   2.2361     0.0000     0.0000     0.0000
   3.1305     0.4472     0.0000     0.0000
   2.6833    -0.8944     1.4142     0.0000
   2.2361    -0.0000     2.1213     0.7071

Diese Matrix  C  ist symmetrisch und hat dasselbe
Eigenwertspektrum wie das gegebene erweiterte Eigenwertproblem:

   1.0000    -3.0000    -3.4785     7.9057
  -3.0000    18.0000    16.4438   -41.1096
  -3.4785    16.4438    18.0000   -43.0000
   7.9057   -41.1096   -43.0000   110.0000

Mit dem Jacobi-Programm koennen die 4 Eigenwerte der
Matrix  C  ohne weiteres bestimmt werden.

Die Eigenwerte lauten: 0.2623  2.3078  1.1530  143.2769
```

7.4.9 'More advanced programs'

Das Jacobi-Verfahren ist im Prinzip für alle reellen symmetrischen Matrizen ausgezeichnet geeignet. Dennoch bieten die meisten 'Profi-Bibliotheken' noch weitere leistungsfähige Programme an, die für Matrizen höherer Ordnung (> 20) deutlich schneller sind als das Jacobi-Verfahren[6].
Die Grundidee dieser Programme ist ein *Zwei-Stufen-Prozess*:

1. Die gegebene symmetrische Matrix A wird mittels eines *endlichen* (d.h. nicht-iterativen) Rechenprozesses auf eine einfachere Form gebracht, und zwar im konkreten Fall auf *tridiagonal-symmetrische* Form. Dazu wird häufig der *Householder*-Algorithmus verwendet.

2. Aus dieser Tridiagonalmatrix werden dann mittels des sog. *QR*- oder *QL*-Algorithmus die Eigenwerte und Eigenvektoren von A berechnet.

Sowohl die Householder- als auch die QR- (QL-)Methode kann im Rahmen dieser Lehrveranstaltung nicht im Detail erörtert werden. Eine sehr gute und kompakte Darstellung der Theorie sowie die entsprechenden Programme TRED2 (Householder) und TQLI (QL-Algorithmus) finden Sie in [9] und in [10].

Genauere Informationen über diese modernen Methoden erhalten Sie auch in meiner weiterführenden Lehrveranstaltung

Ausgewählte Kapitel aus 'Numerische Methoden in der Physik'
SS 2003 2 VO 2 UE (515.433 / 515.434)

7.5 Eigenwerte allgemeiner reeller Matrizen

Dieses Problem wird in der Numerik i. a. ähnlich angegangen wie bei den symmetrischen Matrizen, nämlich mittels eines zweistufigen Prozesses. Dabei wird (1) die Matrix A durch einen nicht-iterativen Algorithmus in eine einfachere Form gebracht (s. u.), und (2) werden die Eigenwerte der so erhaltenen Matrix bestimmt.

Die Grundidee basiert auf den folgenden Aussagen aus der Matrizentheorie:
Jede Matrix kann durch einen nicht-iterativen Ähnlichkeits-Prozess auf eine Upper Hessenberg Form (UHF) gebracht werden.

und

Bei einer Upper-Hessenberg-Matrix sind alle Matrix-Komponenten unterhalb der ersten unteren Nebendiagonale Null.

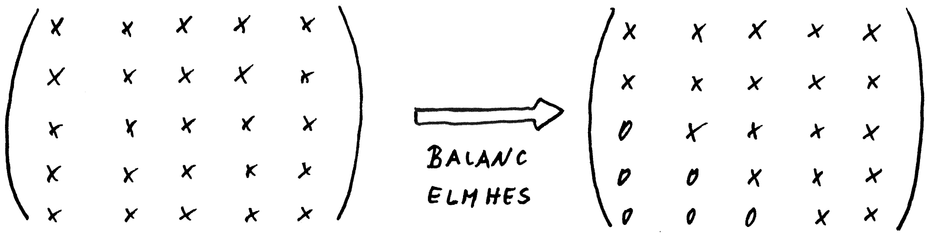


Abbildung 7.1: Reduktion einer allgemeinen Matrix auf die Upper-Hessenberg-Form.

7.5.1 Reduktion einer Matrix auf die Upper-Hessenberg-Form.

Die Umformung einer allgemeinen reellen Matrix auf eine UHF geschieht mit einem Algorithmus, der dem Gauss'schen Eliminationsverfahren (s. Kap. 2, LU-Decomposition) ähnlich ist, nur mit dem wichtigen Unterschied, daß im gegebenen Fall jede Umwandlung der Matrix eine Ähnlichkeitsoperation sein muß. Im folgenden soll dieser Algorithmus kurz erläutert werden:

Angenommen, eine 7x7-Matrix soll in die UHF gebracht werden, und der Algorithmus habe bereits einen Teil der Arbeit erledigt d.h. er habe (z. B.) bereits die ersten drei Spalten der Matrix auf Upper-Hessenberg gebracht. Dann sieht die Matrix so aus:

```
x  x  x  x  x  x  x

x  x  x  x  x  x  x

0  x  x  x  x  x  x

0  0  x  x  x  x  x

0  0  0  a  x  x  x

0  0  0  b  x  x  x

0  0  0  c  x  x  x
```

Im nun folgenden Schritt soll die vierte Spalte auf UHF gebracht werden, d.h. die letzten beiden Werte dieser Spalte sollen Null werden. Dies erreicht man dadurch, daß man die drittletzte Zeile der Matrix mit den Faktoren (b/a) bzw. (c/a) multipliziert und diese zur vorletzten bzw. letzten Zeile addiert.

[6]In diesem Zusammenhang ein interessantes Detail: Die hochangesehene Programmbibliothek LAPACK hatte eine Zeitlang überhaupt keine Jacobi-Routine im Repertoire, bis man im *User's Guide* von 1995, S. 18, den folgenden Satz findet: *In the future LAPACK will include routines based on the Jacobi algorithm ..., which are slower than the above routines (QR etc) but can be significantly more accurate.*

Wie Sie sich erinnern (Kap. 2, LU-Decomposition), ist es bei solchen Rechnungen wichtig, darauf zu achten, daß die Multiplikatoren dem Betrage nach so klein als möglich gehalten werden: dies erreicht man durch *Pivotisierung*, d.h. dadurch, daß man die in Diskussion stehenden Zeilen (im konkreten Fall diejenigen, welche a, b und c enthalten) so vertauscht, daß an der Stelle 'a' die betragsgrößte der Zahlen a, b, c zu stehen kommt.
In einem wichtigen Punkt unterscheidet sich das hier zu diskutierende Verfahren von der LU-Decomposition: *um zu gewährleisten, daß die Umformung der gegebenen Matrix A in eine UHF eine Ähnlichkeitsoperation ist, müssen alle Zeilenvertauschungen bei der Pivotisierung und alle Zeilenadditionen zur Erreichung der 'Hessenberg-Nullen' durch analoge Operationen bzgl. der Spalten ergänzt werden.*

7.5.2 Das Programm ELMHES.

Das Programm ELMHES reduziert eine allgemeine reelle Matrix in die entsprechende 'Upper-Hessenberg-Matrix'.

Quelle: [9], S. 752; [10], S. 485f.

INPUT-Parameter:

A(,): die Komponenten einer reellen Matrix.

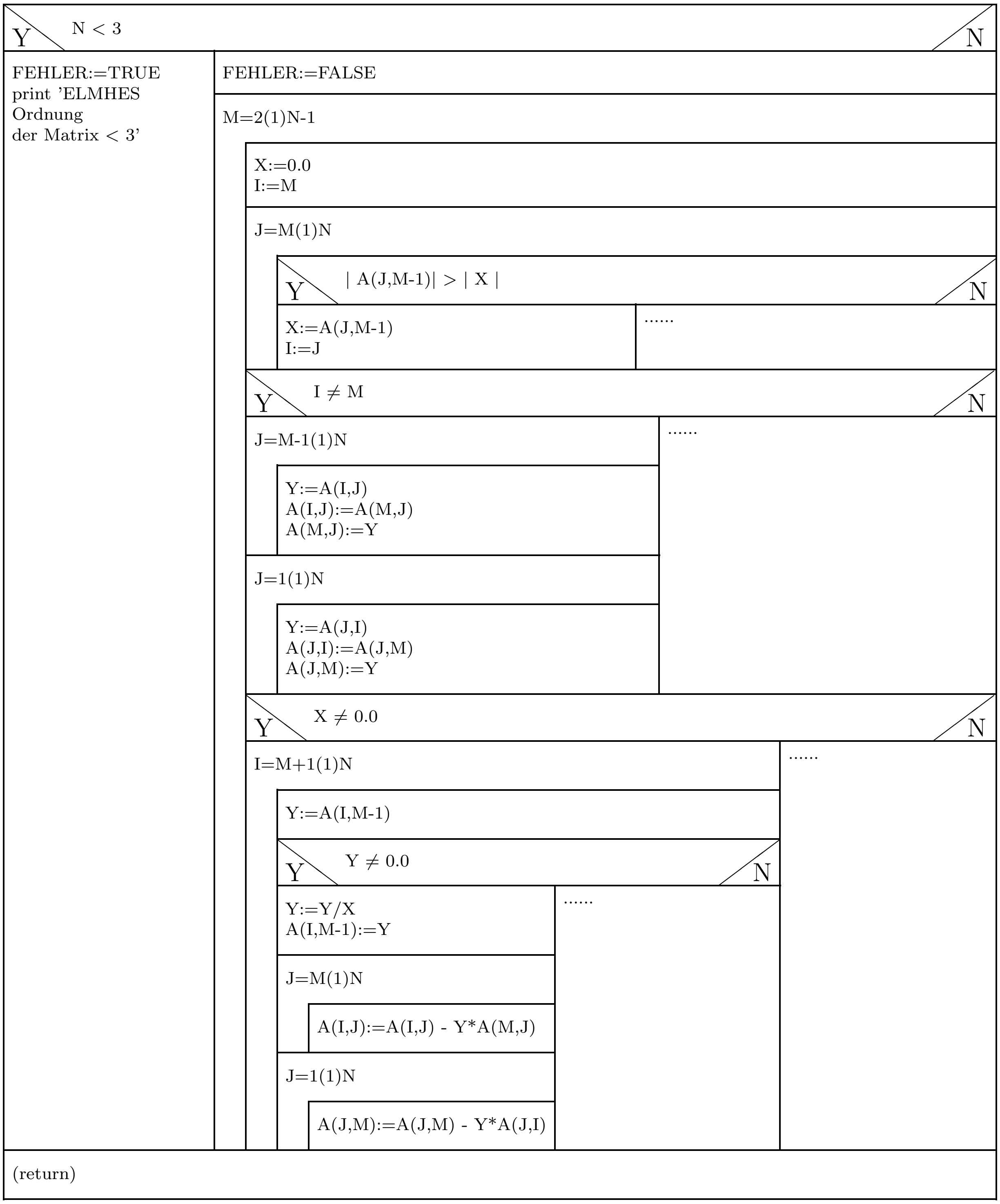
N: Zeilen und Spalten der Matrix.

OUTPUT-Parameter:

A(,): 'Upper-Hessenberg-Matrix' mit denselben Eigenwerten wie die ursprüngliche Matrix.

FEHLER: TRUE wenn Ordnung der Matrix kleiner als 3, sonst FALSE.

Struktogramm 24 — ELMHES(A,N,FEHLER)



7.5.3 Bestimmung der Eigenwerte einer Matrix in UHF.

Um die reellen und konjugiert-komplexen Eigenwerte einer reellen Upper-Hessenberg-Matrix zu berechnen, kann wieder der bereits im Abschnitt 7.4.9 erwähnte QR-Algorithmus zum Einsatz kommen. Ein Beispiel für ein derarti-

ges Programm finden Sie in den Numerical Recipes (Theorie und FORTRAN-Programm in [9], S. 374f; PASCAL-Programm in [9], S. 753f; Theorie und C-Programm in [10], S. 486ff.
Weitere einschlägige Programme siehe auch im Abschnitt 7.6.

Auch über den QR-Algorithmus können Sie in meiner weiterführenden Lehrveranstaltung

Ausgewählte Kapitel aus 'Numerische Methoden in der Physik'
SS 2003 2 VO 2 UE (515.433 / 515.434)

mehr erfahren.

Wenn man sich mit den *reellen* Eigenwerten und Eigenvektoren einer Matrix in UHF begnügt, gibt es auch eine viel einfachere Methode als QR; diese soll im folgenden beschrieben werden:

Wie Sie dem Abschnitt 7.1 entnehmen können, sind die Eigenwerte einer Matrix zugleich die Nullstellen ihres *charakteristischen Polynoms*

$$P_n(\lambda) = \lambda^n + \sum_{i=1}^{n} p_i \, \lambda^{n-i} \, .$$

Es gibt nun Algorithmen, mit denen man die Koeffizienten p_i eines solchen Polynoms bestimmen kann. Kennt man die p_i, kann man die Nullstellen des Polynoms mit einem geeigneten Nullstellen-Suchprogramm ermitteln. Dieses Verfahren ist jedoch sehr umständlich und vor allem sehr rundungsfehleranfällig und wird daher in der Praxis kaum angewendet.
Man kann jedoch im Falle einer Upper-Hessenberg-Matrix das charakteristische Polynom auswerten, *ohne explizite dessen Koeffizienten zu kennen*; dies wird im folgenden Abschnitt erläutert.

7.5.4 Die Methode von Hyman.

Hyman konnte zeigen, daß für Hessenberg-Matrizen das Polynom $P_n(\lambda)$ die Form

$$P_n(\lambda) = (-1)^{n+1} \cdot a_{21} a_{32} \cdots a_{n,n-1} \cdot H(\lambda)$$

annimmt. Natürlich sind die Polynom-Nullstellen mit den Nullstellen der Funktion $H(\lambda)$ identisch. Die Funktionswerte von $H(\lambda)$ können nun aus dem folgenden einfach strukturierten (und numerisch gut konditionierten) linearen Gleichungssystem ermittelt werden:

$$(A - \lambda \cdot E) \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_{n-1} \\ 1 \end{pmatrix} = H(\lambda) \begin{pmatrix} 1 \\ 0 \\ . \\ . \\ . \\ . \\ 0 \end{pmatrix} \quad . \tag{7.45}$$

Dabei ist A die gegebene Hessenberg-Matrix, und $(x_1, x_2, \ldots, x_{n-1}, 1)$ ist ein Hilfsvektor.

Eine einfache Rechnung zeigt nun, daß die $n-1$ unbekannten Koeffizienten des Vektors $\mathbf{x}$ mittels

$$x_n = 1$$

$$x_i = \frac{1}{a_{i+1,i}} \left[\lambda \, x_{i+1} - \sum_{l=0}^{n-i-1} a_{i+1,n-l} x_{n-l} \right] \qquad i = n-1, n-2, \ldots, 2, 1 \quad (7.46)$$

ausgewertet werden können. Aus der ersten Gleichung von (7.45) folgt nun

$$H(\lambda) = \sum_{i=1}^{n} a_{1,i} x_i - \lambda x_1 \quad . \qquad (7.47)$$

Diese Art und Weise der Berechnung der Funktionswerte von $H(\lambda)$ kann sehr effizient mit einem leistungsfähigen numerischen Nullstellen-Suchprogramm (z.B. mit INTSCH, s. Kapitel 5) kombiniert werden.

Die Funktion HYMAN

<u>INPUT-PARAMETER:</u>

UHM: reelle Upper-Hessenberg-Matrix

N: Zeilen und Spalten der Matrix

LAM: Funktionsargument λ

Struktogramm 25 — FUNCTION HYMAN(UHM,N,LAM)

```
X(N):=1.0
I=N-1(-1)1
    SUM:= LAM*X(I+1)
    L=0(1)N-I-1
        SUM:=SUM - UHM(I+1,N-L)*X(N-L)
    X(I):= SUM/UHM(I+1,I)
SUM:=-LAM*X(1)
I=1(1)N
    SUM:=SUM + UHM(1,I)*X(I)
HYMAN:=SUM
(return)
```

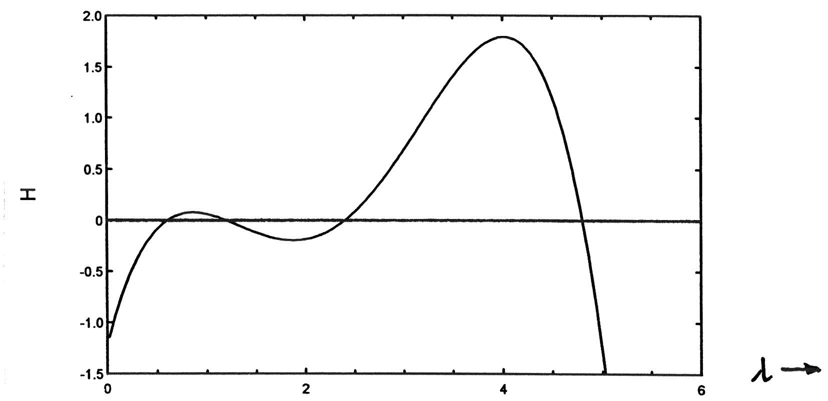


Abbildung 7.2: Die Hyman-Funktion $H(\lambda)$ der 4x4-Matrix von Seite 207.

Ein Testbeispiel

Wenn man die gegebene Matrix mittels ELMHES auf die UHF bringt, und diese Matrix als Input von HYMAN nimmt, so erhält man schnell und genau die Funktionswerte von $H(\lambda)$. Die Nullstellen dieser Kurve sind bekanntlich die (reellen) Eigenwerte der gegebenen Matrix. Um diese Nullstellen numerisch zu ermitteln, müssen also die Programme ELMHES, HYMAN und (z. B.) INTSCH geeignet kombiniert werden.
Die Abb. 7.2 zeigt die Funktion $H(\lambda)$ für die 4x4-Matrix von Seite 207. Die vier Eigenwerte lauten: 0.6, 1.2, 2.4 und 4.8.

7.5.5 Eigenwerte tridiagonaler Matrizen.

Die Hyman'sche Methode kann besonders einfach auf tridiagonale Matrizen der Form

$$\begin{pmatrix} b_1 & c_1 & & & \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & c_{n-1} \\ & & & a_n & b_n \end{pmatrix}$$

angewendet werden, also auf Matrizen, die durch die drei Diagonalvektoren **a**, **b** und **c** bestimmt werden. Man erhält durch Spezialisierung von (7.46) und (7.47) die Formeln

$$x_n = 1$$

$$x_{n-1} = -\frac{1}{a_n}(b_n - \lambda)$$

$$x_{n-2} = -\frac{1}{a_{n-1}}\left(c_{n-1} - \lambda x_{n-1} + b_{n-1}x_{n-1}\right)$$

$$x_{i-1} = -\frac{1}{a_i}\left((b_i - \lambda)x_i + c_i x_{i+1}\right) \qquad i = n-2, n-1, \ldots, 3, 2$$

$$H(\lambda) = (b_1 - \lambda)\, x_1 + c_1 x_2\,. \tag{7.48}$$

Es folgt nun ein Programmiervorschlag (in C) für die numerische Bestimmung der reellen Eigenwerte allgemeiner reeller tridiagonaler Matrizen:

Programm-Struktur C

```
.
#include <stdio.h>
#include <math.h>
#include "nrutil.c"

int n;
double *a,*b,*c;


double hymtri(double lam)
// Diese Routine dient zur Auswertung der 'Hyman-Funktion' fuer
// tridiagonale Matrizen gemaess Glg. (7.43).
// Die Groesse der Matrix (n) und die Vektoren a, b und c sind
// global definiert.
{
  int i;
  double x1,x2,x;

  if(n <= 2) {
    printf("Grad zu klein\n");
    return 0.0;
  }

  else {
    x2=-(b[n]-lam)/a[n];
    x1=-((b[n-1]-lam)*x2 + c[n-1])/a[n-1];
    for(i=n-2;i>1;i--){
      x= -((b[i]-lam)*x1 + c[i]*x2)/a[i];
      x2=x1;
      x1=x;
    }
    return (b[1]-lam)*x1 + c[1]*x2;
  }
}


#include "intsch.c"
```

```
/********************** main program ******************/
void main()
{
  int anzmax=100;
  double *nullst;
   .
   .
  n=....;      //Groesse der Matrix

  a=dvector(1,n);
  b=dvector(1,n);
  c=dvector(1,n);
  nullst=dvector(1,n);

// Einlesen der Vektoren  a  b  c   der tridiagonalen Matrix:
   .
   .

// Eingabe der INTSCH-Parameter:
  anf=....;     // Beginn des Grobsuch-Bereichs
  aend=....;    // Ende des Grobsuch-Bereichs
  h=....;       // Schrittweite im Grobsuch-Bereich
  gen=....;     // rel. Genauigkeit der Eigenwerte


  intsch(&hymtri,anf,aend,h,gen,anzmax,nullst,&anz);

// Ausgabe der Nullstellen = Eigenwerte der tridiag. Matrix:
   .
   .

  free_dvector(a,1,n);
  free_dvector(b,1,n);
  free_dvector(c,1,n);
  free_dvector(nullst,1,n);
}
```

Anmerkung: Das Nullstellen-Suchprogramm INTSCH wird im Abschnitt 5.5.2 genau erklärt. Der im obigen Beispiel angegebene Aufruf von INTSCH enthält jedoch noch einen zusätzlichen Parameter, nämlich den aktuellen Namen der Funktion, deren Nullstellen von INTSCH eruiert werden sollen. Im konkreten Fall ist das die 'Hyman-Funktion' für tridiagonale Matrizen mit dem Namen 'hymtri'.
Eine solche Möglichkeit, auch Namen von Routinen über Parameterlisten

zu übergeben, bieten viele Programmiersprachen (C, F90, Matlab, ...). Im Falle eines C-Programms müßten Sie die Headline von INTSCH wie folgt schreiben:

```
void intsch(double (*fct)(double),double anf, double aend, double h,
            double gen, int anzmax, double nullst[], int *anz)
```

Zusätzlich müßten Sie im Programm INTSCH jeden Aufruf der Funktion anpassen:

```
anstatt  ... fct(x) ...        ... (*fct)(x) ...
```

7.6 Software-Angebot.

EISPACK (in www.netlib.org) stellte viele Jahre den 'De-facto-Standard' in bezug auf Eigenwert-Programme dar[7]. Es bietet Fortran-Unterprogramme zur Berechnung von Eigenwerten und Eigenvektoren für die folgenden Klassen von Matrizen: komplex, Hermitesch komplex, reell, symmetrisch reell, symmetrisch reell mit Bandstruktur, symmetrisch tridiagonal reell, 'special' tridiagonal reell, allgemein reell, allgemein symmetrisch reell. Außerdem sind manche Programme spezialisiert auf Teillösungen (nur Eigenwerte, alle Eigenwerte und ausgewählte Eigenvektoren, einige Eigenwerte, einige Eigenwerte und zugehörige Eigenvektoren).
1992 ist EISPACK zusammen mit LINPACK zu dem Programmsystem LAPACK zusammengefaßt worden. Einige Anmerkungen zu LAPACK s. Abschnitt 2.8 dieses Skriptums.

Numerical Recipes: Die folgenden einschlägigen Programme in F77, F90 und C in den Büchern sowie in

```
/usr/local/numeric/numrec/c  (oder fortran-77 oder fortran-90);

                         reelle Matrix    C/F77/F90
-------------------------------------------------------
Jacobi-Verfahren             symm.         jacobi

Householder                  symm.         tred2

QL                       symm.,tridiag.    tqli

Balancing                   general        balanc

Reduction to Hessenberg     general        elmhes

Eigenvalues                 Hessenb.       hqr

(Anmerkung: die Programme 'tred2', 'tql1', 'balanc',
 'elmhes' und 'hqr' entstammen der EISPACK-Bibliothek)
-------------------------------------------------------
```

Numerical Algorithms with C or FORTRAN: In diesem Werk [6] sind zwei Programme zur Lösung von Eigenwertproblemen allgemeiner reeller Matrizen enthalten:

```
Mises-Verfahren (betragsgroesster Eigenwert und -vektor:
       fmises.c   eigval.f90   eigval.for
```

[7] Smith et al, *EISPACK Guide*, in: *Lecture Notes in Computational Science* (G. Goos, Ed.), Springer Berlin 1974.

```
Verfahren von Martin, Parlett, Peters, Reinsch, Wilkinson
               (alle Eigenwerte und -vektoren, QR-Methode):
       feigen.c   eigen.f90    eigen.for

/usr/local/numeric/num_alg/c/ansic/ansic/07  ('07' bedeutet: Kap. 7)
/usr/local/numeric/num_alg/f77/f77/kap07
/usr/local/numeric/num_alg/f77/f90/kap07
```

MATLAB.6: Alles Wissenswerte über die Eigenwert-Routinen von Matlab können Sie online erfahren, wenn Sie den folgenden Weg gehen:

```
MATLAB Help -- MATLAB Function Listed by Category -- Mathematics --
Linear Algebra -- Eigenvalues and Singular Values
```

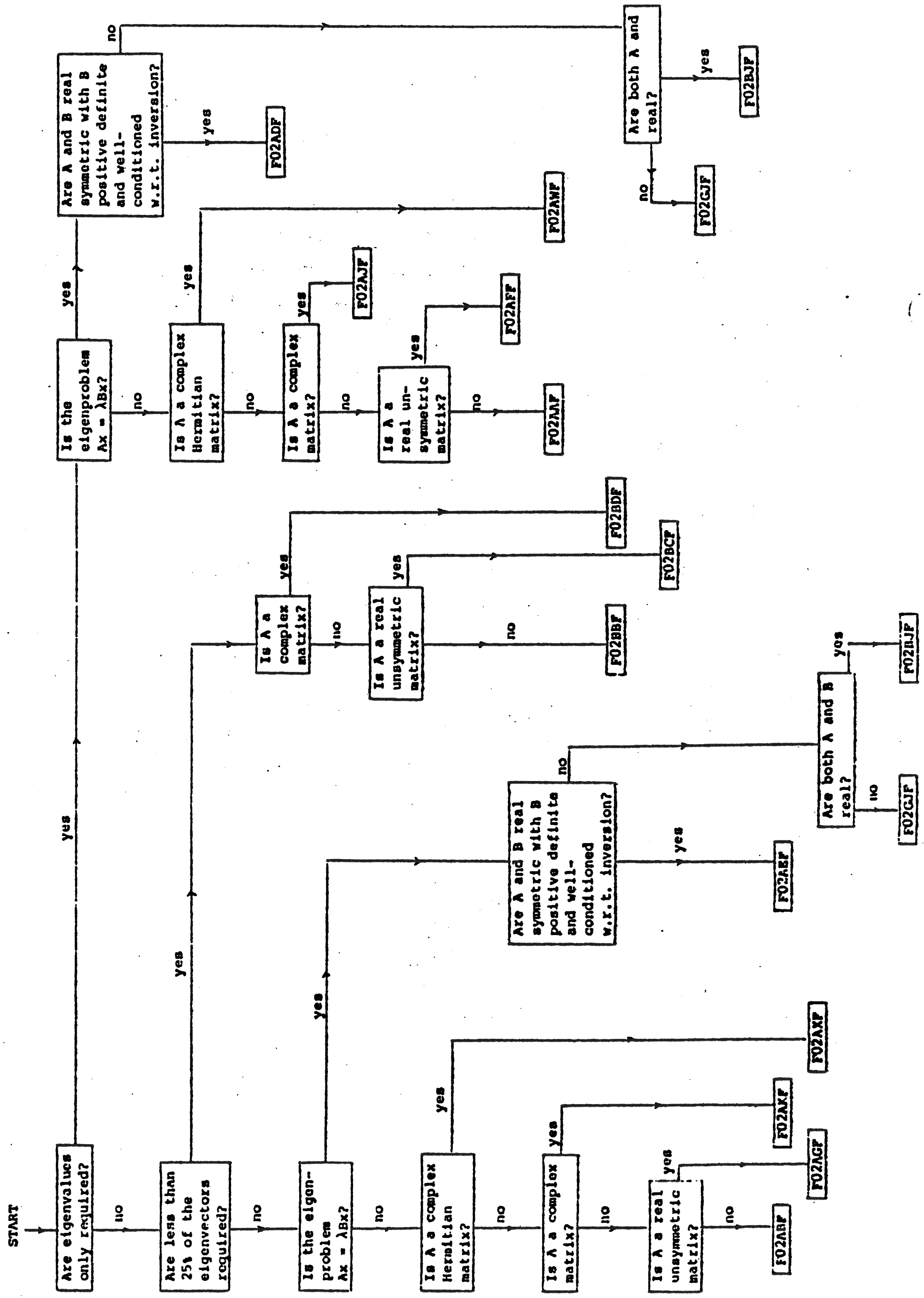
Sie erhalten dann die folgende Liste von Funktionen:

Eigenvalues and Singular Values

balance	Improve accuracy of computed eigenvalues
cdf2rdf	Convert complex diagonal form to real block diagonal form
condeig	Condition number with respect to eigenvalues
eig	Eigenvalues and eigenvectors
eigs	Eigenvalues and eigenvectors of sparse matrix
gsvd	Generalized singular value decomposition
hess	Hessenberg form of matrix
poly	Polynomial with specified roots
polyeig	Polynomial eigenvalue problem
qz	QZ factorization for generalized eigenvalues
rsf2csf	Convert real Schur form to complex Schur form
schur	Schur decomposition
svd	Singular value decomposition
svds	Singular values and vectors of sparse matrix

Alle in dieser Liste enthaltenen Funktionen verwenden LAPACK-Programme! Durch Anklicken der entsprechenden Namen (besonders wichtig sind die Funktionen *eig* und *eigs*) erhalten Sie genaue Informationen, insbesondere über die Parameterliste.

NAG: Auch in dieser Library gibt es zahlreiche einschlägige Programme, wie der folgende *decision tree* zeigt:



Kapitel 8

Numerische Methoden zur Lösung von gewöhnlichen Differentialgleichungen: Anfangswertprobleme.

8.1 Allgemeines.

Das Auffinden von Lösungen von Differentialgleichungen (Dgl.en) gehört zu den wichtigsten Aufgaben der numerischen Mathematik, und zwar aus zwei Gründen:

- Obwohl es eine ganze Reihe von analytischen Methoden zur Lösung von Dgl.en gibt, sind analytische Lösungen häufig sehr kompliziert und so aufwendig zu ermitteln, daß es oft nicht dafür steht. Dies soll aber die Bedeutung von analytischen Methoden keineswegs herabsetzen. Das Motto aus dem Numerik-Buch von Dorn/McCracken [7] *'Numerical methods are no excuse for poor analysis'* gilt auch hier.

- Für eine ganze Reihe von Dgl.en gibt es überhaupt keine geschlossen darstellbaren analytischen Lösungen. Dies ist zum Beispiel bei der relativ 'harmlos' aussehenden Dgl.

  $$y'(x) = x^2 + y(x)^2$$

  der Fall! In solchen Fällen ist nur eine numerische Lösung möglich.

Im folgenden wird ausschließlich von *expliziten* Dgl.en die Rede sein, d.h. von solchen, die nach der jeweils höchsten vorkommenden Ableitung aufgelöst werden können. Dgl.en von der Art

$$y'(x) + \log y'(x) = 1$$

werden also in diesem Abschnitt nicht behandelt. Außerdem funktionieren die im weiteren zu behandelnden Verfahren nur für Dgl.en bzw. für Systeme von Dgl.en *erster Ordnung.* Diese Einschränkung ist jedoch bedeutungslos,

da ja jede explizite Dgl. höherer Ordnung in ein äquivalentes System von Dgl.en erster Ordnung umgewandelt werden kann.
So ist die Dgl. n-ter Ordnung

$$y^{(n)} = F(x; y, y', y'', \dots, y^{(n-1)})$$

äquivalent mit dem System

$$\begin{array}{ll}
y_1' = y_2 & \equiv f_1(x) \\
y_2' = y_3 & \equiv f_2(x) \\
. & \\
. & \\
. & \\
y_{n-1}' = y_n & \equiv f_{n-1}(x) \\
y_n' = F(x; y_1, y_2, \dots, y_n) & \equiv f_n(x)
\end{array}$$

Faßt man die Größen y_i', y_i und f_i als Vektor-Komponenten auf, so kann ein solches System in der Form

$$\mathbf{y}'(x) = \mathbf{f}(x; \mathbf{y}) \tag{8.1}$$

geschrieben werden.

Alle numerischen Verfahren zur näherungsweisen Lösung von Dgl.en liefern natürlich keine Funktionen mit allgemeinen Integrationskonstanten, sondern berechnen punktweise jene Lösungen, die durch die entsprechenden *Nebenbedingungen* spezifiziert werden. In diesem Kapitel werden nur *Anfangswert-Probleme* behandelt, d.h. solche, bei denen die Nebenbedingungen das Verhalten der Lösungsfunktionen $\mathbf{y}(x)$ in einem Punkt $x = x_o$ festlegen. Ein *vollständig definiertes Anfangswert-Problem* hat daher die Form

$$\mathbf{y}'(x) = \mathbf{f}(x; \mathbf{y}) \qquad \text{mit} \quad \mathbf{y}(x_o) = \mathbf{y}_o \quad . \tag{8.2}$$

8.2 Eine Taylorreihenentwicklung der Lösungsfunktionen.

Ausgangspunkt aller weiteren Überlegungen ist die Potenzreihenentwicklung der i-ten Lösungsfunktion des Systems (8.2)

$$y_i(x) = \sum_{\nu=0}^{p} \frac{(x - x_o)^\nu}{\nu!} \left[\frac{d^\nu}{dx^\nu} y_i(x) \right]_{x_o, \mathbf{y}_o} + R_i \tag{8.3}$$

mit dem *Lagrange'schen Restglied*

$$R_i = \frac{(x - x_o)^{p+1}}{(p+1)!} \left[\frac{d^{p+1}}{dx^{p+1}} y_i(x) \right]_{x=\xi} \qquad x_o \le \xi \le x \quad . \tag{8.4}$$

Der Zweck aller folgenden Verfahren besteht nun darin, ausgehend von dem Anfangspunkt des Dgl.ssystems x_o einen Näherungswert für die y_i an der Stelle $x_o + h$ zu berechnen. Dabei nennt man *p die Ordnung des Verfahrens*:

$$\hat{y}_i(x_o + h) = \sum_{\nu=0}^{p} \frac{h^\nu}{\nu!} \left[\frac{d^\nu}{dx^\nu} y_i(x) \right]_{x_o, \mathbf{y}_o} \quad . \tag{8.5}$$

Anmerkung: Hier und im folgenden werden Näherungswerte von y stets mit $\hat{y}$ bezeichnet.

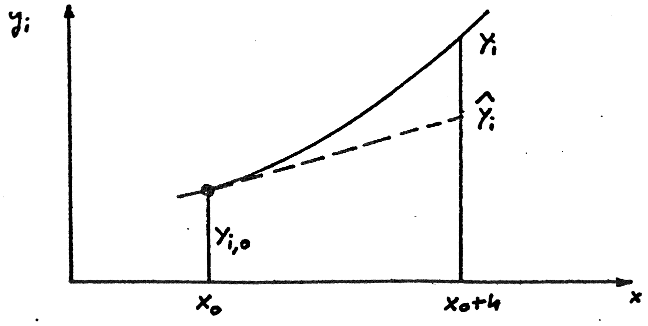


Abbildung 8.1: Geometrische Interpretation der Methode von Euler.

8.3 Die Methode von Euler.

Die älteste Näherungsmethode zur numerischen Behandlung von Dgl.ssystemen stammt von Leonhard Euler. In diesem Verfahren *erster Ordnung* (p=1) wird die Entwicklung (8.5) bereits nach dem zweiten Term abgebrochen, und man erhält

$$\hat{y}_i(x_o + h) = y_i(x_o) + h \cdot y'(x) \mid_{x=x_o} = y_i(x_o) + h \cdot f_i(x_o, \mathbf{y}_o) \tag{8.6}$$

Die geometrische Interpretation dieser Formel ist denkbar einfach: Es wird die exakte Lösung $y_i(x)$ durch deren Tangente im Anfangspunkt x_o angenähert (s. Abb. 8.1).
Es leuchtet ein, daß die Euler'sche Methode nur für sehr kleine Schrittweiten brauchbare Resultate liefert.

8.4 Die Runge-Kutta-Methoden.

Runge-Kutta-Methoden sind die weitaus populärsten zur numerischen Behandlung von Anfangswertproblemen, vor allem deshalb, weil sie sich außerordentlich gut für die Rechenmaschine eignen und in vielen Fällen Resultate hoher Genauigkeit mittels akzeptablen Rechenaufwandes liefern.
Der - wie noch zu erläutern sein wird - größte Nachteil der Runge-Kutta-Methoden liegt in der ziemlich schwierigen Fehlerabschätzung (s. Abschnitt 8.4.5).

Runge-Kutta-Methoden sind durch die folgenden drei Prinzipien charakterisiert:

- Sie leiten sich aus Taylorreihen der gesuchten Lösungsfunktionen ab, die nach dem Term mit h^p abgebrochen werden. p ist die *Ordnung* der jeweiligen Runge-Kutta-Methode.
- Sie sind sogenannte *Einschritt-Verfahren*, d.h. zur Berechnung von Näherungswerten an der Stelle $x + h$ genügt die Information über den vorhergehenden Punkt x.
- Zur Berechnung der Näherungswerte für die $y_i(x)$ werden nur die gegebenen Funktionen $f_i(x, \mathbf{y})$ benötigt, *nicht aber deren Ableitungen!*

Dies ist in der Praxis ein wichtiger Vorteil gegenüber der Taylorreihen-Entwicklung.

Aus diesen drei Charakteristiken für Runge-Kutta-Methoden ergibt sich sofort:

Die Methode von Euler ist eine Runge-Kutta-Methode erster Ordnung.

8.4.1 Runge-Kutta-Methoden zweiter Ordnung.

Es soll nun gezeigt werden, wie Runge-Kutta-Formeln zweiter Ordnung gewonnen werden können. Der Einfachheit halber wird hier ein 'Dgl.ssystem' behandelt, das nur aus der einzigen Gleichung

$$y'(x) = f(x, y)$$

mit der Anfangsbedingung

$$y(x_o) = y_o$$

besteht. Ausgangspunkt für eine Runge-Kutta-Formel 2. Ordnung ist der *Ansatz*

$$\hat{y}(x_o + h) = y_o + h \cdot (c_1 g_1 + c_2 g_2) \quad . \tag{8.7}$$

In (8.7) stellen die Größen g_1 und g_2 Funktionswerte von $f(x, y)$ dar, und zwar

$$\begin{aligned} g_1 &= f(x_o, y_o) \\ g_2 &= f(x_o + a_2 h, y_o + b_{2,1} h g_1) \end{aligned} \tag{8.8}$$

c_1, c_2, a_2 und $b_{2,1}$ sind die *Runge-Kutta-Koeffizienten*. Diese sind so zu wählen, daß der obige Ansatz mit dem Taylorreihen-Ansatz (8.5) für $p = 2$

$$\hat{y}(x_o + h) = y_o + h \cdot f(x_o, y_o) + \frac{h^2}{2} \left[f_x + f \cdot f_y \right]_{x_o, y_o} \tag{8.9}$$

(näherungsweise) identisch wird.
Dazu entwickelt man g_2 (8.8) an der Stelle $h = 0$ nach Potenzen von h und bricht diese Entwicklung nach dem linearen Term ab:

$$\begin{aligned} g_2(h) &= g_2(h = 0) + h \cdot \left[\frac{d}{dh} g_2(h) \right]_{h=0} & (8.10) \\ &= f(x_o, y_o) + h \cdot \left[f_x(x_o, y_o) \cdot a_2 + f(x_o, y_o) \cdot f_y(x_o, y_o) \cdot b_{2,1} \right] \end{aligned}$$

Setzt man nun (8.10) zusammen mit $g_1 = f(x_o, y_o)$ in den Runge-Kutta-Ansatz (8.7) ein, so ergibt sich weiter

$$\begin{aligned} \hat{y}(x_o + h) &= y_o + h \cdot c_1 \cdot f(x_o, y_o) + h \cdot c_2 \cdot f(x_o, y_o) + & (8.11) \\ &\quad + h^2 \cdot c_2 \cdot a_2 \cdot f_x(x_o, y_o) + h^2 \cdot c_2 \cdot b_{2,1} \cdot f(x_o, y_o) \cdot f_y(x_o, y_0) \quad . \end{aligned}$$

Durch Gleichsetzen von (8.9) und (8.11) und einen Koeffizientenvergleich bzgl. der Größen f, f_x und $f \cdot f_y$ erhält man das folgende *nicht-lineare Gleichungssystem für die gesuchten Runge-Kutta-Koeffizienten zweiter Ordnung*:

$$
\begin{aligned}
c_1 + c_2 &= 1 \\
c_2 \cdot a_2 &= \frac{1}{2} \\
c_2 \cdot b_{2,1} &= \frac{1}{2}
\end{aligned}
\tag{8.12}
$$

Dieses System ist jedoch *unterbestimmt*: vier zu ermittelnden Koeffizienten stehen nur drei Gleichungen gegenüber! *Es kann also einer der vier Koeffizienten frei gewählt werden.* Die Konsequenz daraus ist, daß es nicht nur <u>eine</u> Runge-Kutta-Methode zweiter Ordnung gibt, sondern <u>unendlich</u> <u>viele</u>, wobei es von der Geschicklichkeit des Rechnenden abhängt, einen Koeffizienten so zu wählen, daß die übrigen drei Werte möglichst einfach werden.

<u>Dazu zwei Beispiele:</u>

- Mit der Wahl $c_1 = 0$ erhält man für

  $$c_2 = 1 \qquad a_2 = \frac{1}{2} \qquad \text{und} \qquad b_{2,1} = \frac{1}{2} \quad .$$

  Die aus (8.7) resultierende Runge-Kutta-Formel

  $$\hat{y}(x_o + h) = y_o + h \cdot f(x_o + \frac{h}{2}, y_o + \frac{h}{2} f(x_o, y_o)) \tag{8.13}$$

  wird *modifizierte Euler'sche Methode* genannt.

- Mit der Wahl $c_1 = 1/2$ erhält man hingegen

  $$c_2 = \frac{1}{2} \qquad a_2 = 1 \qquad b_{2,1} = 1 \quad .$$

  Die entsprechende Lösung

  $$\hat{y}(x_o + h) = y_o + h \left\{ \frac{1}{2} f(x_o, y_o) + \frac{1}{2} f\left[x_o + h, y_o + h f(x_o, y_o)\right] \right\} \tag{8.14}$$

  heißt *verbesserte Euler'sche Methode.*

Die grafische Interpretation dieser beiden Näherungsformeln geht aus der Abb.8.2 hervor.

- modifizierte Euler-Formel: Die Lösungsfunktion wird angenähert durch eine Gerade durch (x_o, y_o) mit der Steigung von $y(x)$ im Mittelpunkt des Intervalls $[x_o, x_o + h]$.
- verbesserte Euler-Formel: Die Lösungsfunktion wird angenähert durch eine Gerade durch (x_o, y_o), deren Steigung das arithmetische Mittel aus $y'(x_o)$ und $\hat{y}'(x_o + h)$ ist.

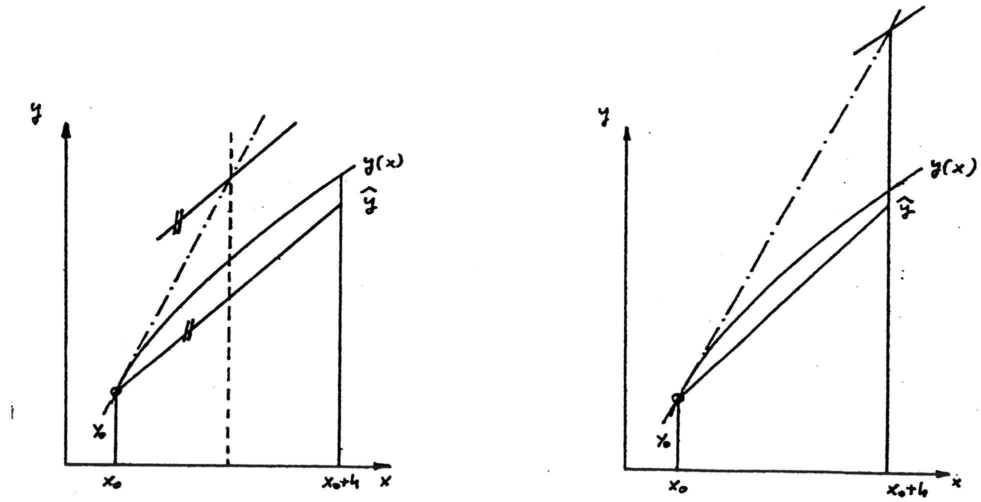


Abbildung 8.2: Grafische Interpretation der 'modifizierten Euler-Formel' (links) und der 'verbesserten Euler-Formel' (rechts).

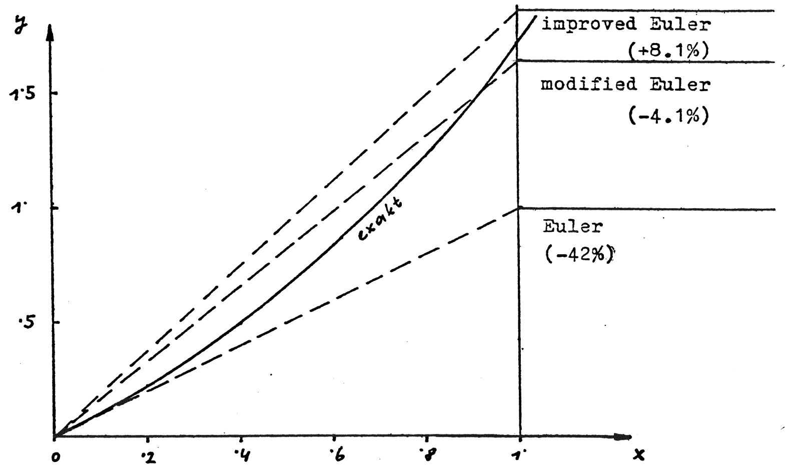


Abbildung 8.3: Ein Testbeispiel für Runge-Kutta-Methoden erster und zweiter Ordnung.

Im Diagramm 8.3 werden die bisher vorgestellten Runge-Kutta-Formeln (8.6), (8.13) und (8.14) auf das einfache Beispiel

$$y'(x) = e^x \qquad y(0) = 0$$

angewendet. Die exakte Lösung ist

$$y(x) = e^x - 1 \quad ,$$

und gesucht ist $\hat{y}(1)$.
Aus den in Abb. 8.3 dargestellten Ergebnissen geht klar hervor, daß die beiden Runge-Kutta-Methoden zweiter Ordnung eine Klasse besser sind als das einfache Euler-Verfahren erster Ordnung.
Es wäre hingegen falsch, aus der Tatsache, daß beim gegebenen Beispiel die 'modifizierte Euler-Formel' den exakten Lösungswert besser approximiert als die 'verbesserte Euler-Formel', zu schließen, daß erstere Methode besser ist als letztere. Dies ist reiner Zufall und kann bei einem anderen Beispiel umgekehrt sein!

Jede Runge-Kutta-Formel derselben Ordnung ist prinzipiell gleichwertig!

8.4.2 Runge-Kutta-Methoden höherer Ordnung.

Für Dgl.ssysteme von n Gleichungen lautet der *allgemeine Runge-Kutta-Ansatz p-ter Ordnung*:

$$\hat{y}_i(x_o + h) = y_{i,o} + h \cdot \sum_{j=1}^{p} c_j g_{i,j} \tag{8.15}$$

mit

$$g_{i,1} = f_i(x_o; y_{1,o}, y_{2,o}, \ldots, y_{n,o}) \tag{8.16}$$

und

$$\begin{aligned} g_{i,j} \quad = \quad & f_i(x_o + a_j h; y_{1,o} + h \sum_{\ell=1}^{j-1} b_{j,\ell} g_{1,\ell}, \ldots, \\ & y_{n,o} + h \sum_{\ell=1}^{j-1} b_{j,\ell} g_{n,\ell}) \end{aligned} \tag{8.17}$$

mit $i = 1, \ldots, n$ und $j = 1, \ldots, p$.
[(8.17) ist offenbar eine *Rekursionsformel*: Zur Berechnung der $g_{i,j}$ ist die Kenntnis aller vorherigen g-Werte $g_{k,\ell}$, $\ell = 1, \ldots, j-1$ und $k = 1, \ldots, n$ erforderlich.]

Der Runge-Kutta-Ansatz (8.15) enthält genau $(p^2+3p-2)/2$ Koeffizienten, und zwar

- die p Werte $\quad c_1, \dots, c_p \quad$,
- die $p-1$ Werte $\quad a_2, \dots, a_p \quad$ und
- die $p(p-1)/2$ Werte $\quad b_{2,1}, \dots, b_{p,p-1} \quad$.

Zur besseren Übersichtlichkeit können diese Koeffizienten auch als Schema der Art

$$
\begin{array}{c|ccccc}
a_2 & b_{2,1} & & & & \\
a_3 & b_{3,1} & b_{3,2} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
a_p & b_{p,1} & b_{p,2} & \cdots & b_{p,p-1} & \\
\hline
 & c_1 & c_2 & \cdots & c_{p-1} & c_p
\end{array}
$$

geschrieben werden. Die Berechnung der Koeffizienten erfolgt im Prinzip gleich wie für den Fall $p=2$. Für größere Werte von p wird die Rechnung jedoch ziemlich umfangreich (s. z.B. [19]). In jedem Fall ergeben sich aber für die Koeffizienten unterbestimmte Gleichungssysteme, was auch für $p>2$ zu einer unendlichen Mannigfaltigkeit von Runge-Kutta-Formeln führt.

Natürlich braucht der Benutzer diese Formeln nicht selbst abzuleiten, sondern er kann sie der Literatur entnehmen. Eine Sammlung von Runge-Kutta-Formeln der Ordnungen 1-8 ist z.B. in [2], S. 215-217, zu finden.

In der Praxis werden heute hauptsächlich Runge-Kutta-Methoden <u>vierter</u> Ordnung angewendet, darunter vor allem

`die 3/8-Formel:`

$$
\begin{array}{c|cccc}
1/3 & 1/3 & & & \\
2/3 & -1/3 & 1 & & \\
1 & 1 & -1 & 1 & \\
\hline
 & 1/8 & 3/8 & 3/8 & 1/8
\end{array}
$$

`die 'klassische' Runge-Kutta-Formel:`

$$
\begin{array}{c|cccc}
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
$$

Das im Abschnitt 8.5 präsentierte Programm beruht auf der *klassischen Runge-Kutta-Formel*. Aus diesem Grund seien die entsprechenden Formeln nun explizite hingeschrieben:

$$\hat{y}_i(x_o + h) = y_{i,o} + h\left[\frac{1}{6}g_{i,1} + \frac{1}{3}g_{i,2} + \frac{1}{3}g_{i,3} + \frac{1}{6}g_{i,4}\right] \tag{8.18}$$

mit

$$\begin{aligned}
g_{i,1} &= f_i(x_o; y_{1,o}, \ldots, y_{n,o}) \\
g_{i,2} &= f_i(x_o + \frac{h}{2}; y_{1,o} + \frac{h}{2}g_{1,1}, \ldots, y_{n,o} + \frac{h}{2}g_{n,1}) \\
g_{i,3} &= f_i(x_o + \frac{h}{2}; y_{1,o} + \frac{h}{2}g_{1,2}, \ldots, y_{n,o} + \frac{h}{2}g_{n,2}) \\
g_{i,4} &= f_i(x_o + h; y_{1,o} + hg_{1,3}, \ldots, y_{n,o} + hg_{n,3})
\end{aligned} \tag{8.19}$$

und $i = 1, \ldots, n$. Wie man sieht, hat die Tatsache, daß die Koeffizienten $b_{3,1}$, $b_{4,1}$ und $b_{4,2}$ Null sind, die angenehme Konsequenz, daß die in (8.17) auftretenden Summen zu *einem* Term reduziert werden.

Anmerkung: Das klassische Runge-Kutta-Verfahren würde, auf das Beispiel von Seite 240 angewendet, einen Fehler von lediglich $= 0.034\%$ ergeben.

8.4.3 Die Anwendung von Runge-Kutta-Formeln.

Die Formeln (8.18) und (8.19) dienen also dazu, ausgehend von bekannten Anfangswerten $y_{i,o}$ an der Stelle $x = x_o$ Näherungswerte für die Lösungsfunktionen an der Stelle $x = x_o + h$ zu berechnen.

Nun ist man aber i.a. nicht nur an den Näherungslösungen für den einen Abszissenwert $x_1 = x_o + h$ interessiert, sondern auch an den $\hat{y}_i$ für die weiteren Abszissenwerte $\cdots x_4 > x_3 > x_2 > x_1$! Das heißt aber, daß das Runge-Kutta-Verfahren nicht nur einmal angewendet wird, sondern oftmals hintereinander, wobei man sich Schritt für Schritt vom Anfangspunkt x_o wegbewegt:

$$\hat{y}_i(x_o + h) \equiv \hat{y}_{i,1} = y_{i,o} + h\sum_{j=1}^{p} c_j g_{i,j}(x_o; y_{1,o}, \ldots, y_{n,o})$$

$$\hat{y}_i(x_o + 2h) \equiv \hat{y}_{i,2} = \hat{y}_{i,1} + h\sum_{j=1}^{p} c_j g_{i,j}(x_o + h; \hat{y}_{1,1}, \ldots, \hat{y}_{n,1})$$

Der wesentliche Unterschied zwischen dem ersten und allen weiteren Runge-Kutta-Schritten besteht darin, daß die x/y-Werte auf der rechten Seite der ersten Gleichung die exakt bekannten Anfangswerte des gegebenen Problems sind, während die x/y-Werte der folgenden Gleichungen nur Näherungswerte darstellen!

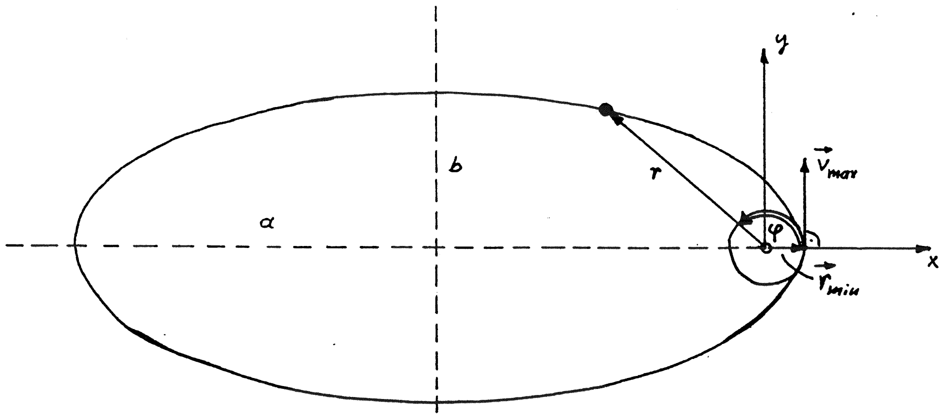


Abbildung 8.4: Bahnkurve eines Erdsatelliten.

8.4.4 Ein Testbeispiel: Qualitätsprobleme.

Es soll nun angenommen werden, daß ein Programm RUNGETEST vorliegt, mit Hilfe dessen ein Anfangswertproblem (8.2) mittels der Formeln (8.18), (8.19) gelöst werden kann, *wobei die (vom Benutzer festgelegte) Schrittweite h während des ganzen Runge-Kutta-Prozesses konstant ist.*
Die Leistungsfähigkeit von RUNGETEST soll im folgenden an Hand eines durchaus nicht-trivialen Problems getestet werden: Es geht dabei um die Bahnkurve eines Erdsatelliten (s. Abb. 8.4).

Definition des Problems:

Ein Satellit wird von der Oberfläche der Erde tangential mit der Geschwindigkeit v_{max} in Bewegung gesetzt. Zu berechnen ist seine Bahnkurve unter den folgenden Bedingungen bzw. Idealisierungen:

- v_{max} < Fluchtgeschwindigkeit.
- Die Erde sei eine homogene, ideale Kugel mit dem Radius r_{min}.
- Vernachlässigung des Einflusses der Lufthülle der Erde.
- Vernachlässigung des Einflusses aller anderen Himmelskörper.

Unter diesen Voraussetzungen lautet die *Bewegungsgleichung* des Satelliten

$$\ddot{\mathbf{r}} = -\frac{\gamma M}{r^3}\mathbf{r} \tag{8.20}$$

mit der Gravitationskonstanten $\gamma = 6.67 \cdot 10^{-11}\ m^3/kgs^2$ und
der Masse der Erde $M = 5.977 \cdot 10^{24}\ kg$. $\mathbf{r}$ ist der Radiusvektor vom Erdmittelpunkt zur momentanen Satelliten-Position.
Die Gleichung (8.20) führt letztlich zum folgenden System von 2 Dgl.en zweiter Ordnung für (1) den Abstand r und (2) den Drehwinkel φ:

$$\begin{aligned}\ddot{r} &= r\dot{\varphi}^2 - \frac{\gamma M}{r^2} \\ \ddot{\varphi} &= -\frac{2\dot{r}\dot{\varphi}}{r}\end{aligned} \tag{8.21}$$

Setzt man

$$r \rightarrow y_1 \quad \varphi \rightarrow y_2 \quad \dot{r} \rightarrow y_3 \quad \dot{\varphi} \rightarrow y_4 \quad ,$$

erhält man daraus das folgende System von 4 Dgl.en erster Ordnung:

$$\begin{aligned}\dot{y}_1 &= y_3 & y_1(t=0) &= r_{min} \\ \dot{y}_2 &= y_4 & y_2(t=0) &= 0 \\ \dot{y}_3 &= y_1 y_4^2 - \frac{\gamma M}{y_1^2} & y_3(t=0) &= 0 \\ \dot{y}_4 &= -\frac{2y_3y_4}{y_1} & y_4 &= \frac{v_{max}}{r_{min}} = 58.29527\end{aligned} \tag{8.22}$$

Für die konkreten Testläufe war $r_{min} = 6.37 \cdot 10^6$ m und $v_{max} = 10.4 \cdot 10^3$ m/s.

Wenn man nun alle Längen in Einheiten r_{min} mißt, alle Geschwindigkeiten in Einheiten v_{max}, und alle Zeiten in Einheiten der (theoretisch berechenbaren) exakten Umlaufzeit, ergibt sich für den in Glg. (8.21) vorkommenden Faktor

$$\alpha \equiv \gamma M = 1966.39\,.$$

Ein entscheidendes Kriterium für die Zuverlässigkeit der numerischen Methode ist natürlich die Bahnstabilität, d.h. die Bahnellipse sollte sich Umlauf für Umlauf exakt wiederholen. Jede Abweichung von dieser Stabilität deutet auf Verfahrens- und/oder Rundungsfehler[1] im Runge-Kutta-Programm hin! Es leuchtet unmittelbar ein, daß die Verfahrensfehler des Programmes umso kleiner sein werden, je kleiner man die Schrittweite des Runge-Kutta-Prozesses wählt. Diese Tatsache ist in der Abbildung 8.5 eindrucksvoll zu sehen, wo jeweils 5 volle Umläufe dargestellt sind. Die verwendeten Schrittweiten waren (a) 1/50, (b) 1/60, (c) 1/70 und (d) 1/80 der Umlaufzeit des Satelliten.
Diese Testergebnisse zeigen deutlich, welch großen Einfluß die Schrittweite auf die performance des Runge-Kutta-Prozesses hat: Bei einer Schrittweite von 1/50 der Umlaufzeit (a) ist der Prozess vollkommen instabil, während bei einer Schrittweite, die nur um einen Faktor 1.6 kleiner ist (d), eine fast perfekte Bahnstabilität vorliegt.

Man kann also sagen:

- Das Programm RUNGETEST ist für die Praxis ungeeignet!
- Jedes praktisch verwendbare Runge-Kutta-Programm muß unbedingt über eine *Fehlerdiagnostik* bzw. eine damit verbundene *automatische Schrittweitensteuerung* verfügen!

[1]Wie eine genaue Analyse der Testergebnisse zeigt, spielen Rundungsfehler bei diesem Test eine nur untergeordnete Rolle.

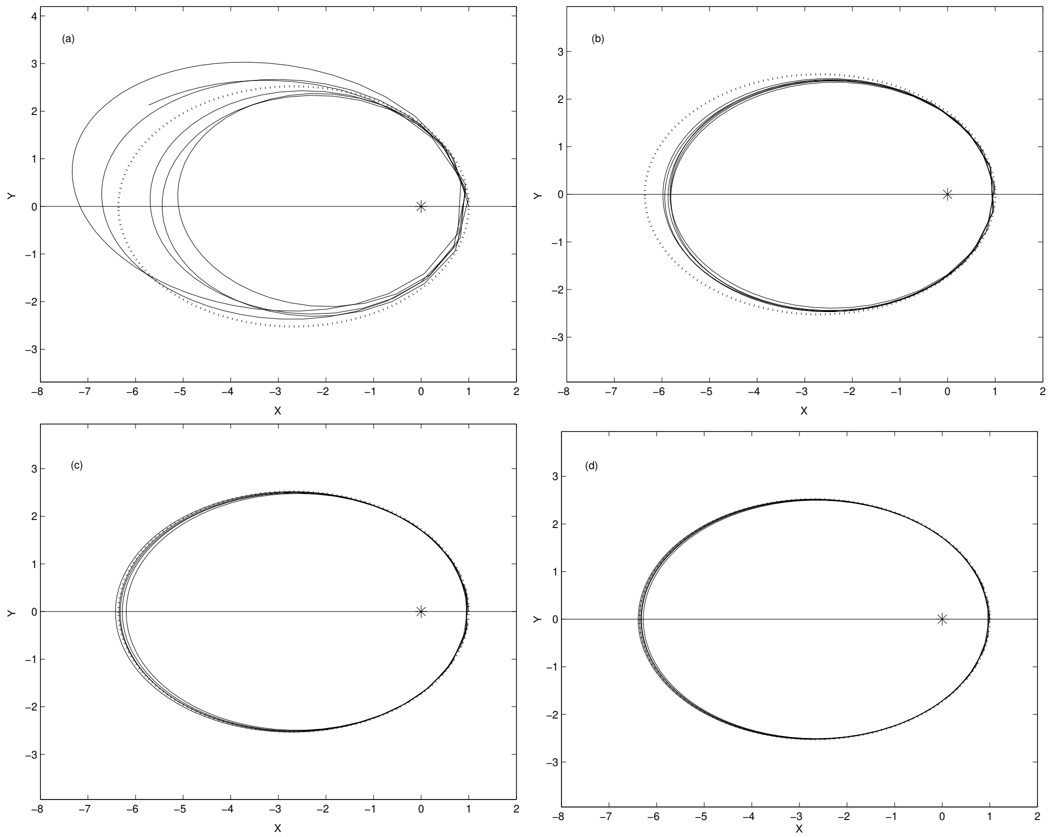


Abbildung 8.5: Ein Stabilitätstest für RUNGETEST. Die konstanten Schrittweiten wurden wie folgt gewählt: (a) $h = 1/50$, (b) $h = 1/60$, (c) $h = 1/70$, (d) $h = 1/80$ der Umlaufzeit. Der Stern markiert den Erdmittelpunkt, und die punktierte Linie stellt die analytisch berechnete, exakte Ellipsenbahn des Satelliten dar.

8.4.5 Fehlerabschätzung und Schrittweiten-Steuerung beim Runge-Kutta-Verfahren.

Wie schon oft erwähnt, wird ein numerisches Ergebnis erst durch die Angabe seines (absoluten oder relativen) Fehlers brauchbar. Man hätte ja ansonsten keinerlei Anhaltspunkte für die Qualität des Näherungsergebnisses.

Da die Runge-Kutta-Methoden eng mit dem Taylorreihen-Ansatz (8.3) verwandt sind, liegt es nahe, das entsprechende Restglied von Lagrange (8.4) zur Abschätzung des Verfahrensfehlers heranzuziehen. Für das klassische Runge-Kutta-Verfahren (Ordnung $p = 4$) ergibt sich daraus

$$E_V = \frac{h^5}{120}\left[y_i^{(5)}(x)\right]_{x=\xi} \qquad x_o \leq \xi \leq x_o + h$$

bzw.

$$E_V = C(h) \cdot h^5 \quad . \tag{8.23}$$

Da aber E_V den Fehler zwischen dem exakten Wert der Lösungsfunktion $y_i(x)$ an der Stelle $x_o + h$ und der entsprechenden Näherungslösung $\hat{y}_i(x_o + h)$ darstellt, gilt weiter

$$y_i(x_o + h) = \hat{y}_i(x_o + h) + C(h) \cdot h^5 \quad . \tag{8.24}$$

Ebenso kann man aber, um von x_o nach x_o+h zu gelangen, *nacheinander zwei Runge-Kutta-Schritte mit der Schrittweite* $h/2$ *machen*. Dies ergibt

$$y_i(x_o+h) = \hat{y}_i(x_o + 2 \cdot \frac{h}{2}) + C(h/2) \cdot 2 \cdot \left(\frac{h}{2}\right)^5 \quad . \tag{8.25}$$

Im weiteren soll nun der Index i bei den y-Werten weggelassen werden, und außerdem werden die Abkürzungen

$$\hat{y}(x_o+h) \equiv \hat{y}(h) \qquad \text{und} \qquad \hat{y}(x_o+2\frac{h}{2}) \equiv \hat{y}(h/2)$$

eingeführt. Wenn man nun annimmt, daß die h-Abhängigkeit der Größe C nicht allzu groß ist, daß also in guter Näherung

$$C(h) \approx C(h/2) = C$$

ist, kann man diese Konstante (und damit den Verfahrensfehler $E_V(h) = Ch^5$) durch Gleichsetzen von (8.24) und (8.25) berechnen:

$$E_V(h) = \frac{16}{15}\left[\hat{y}(h/2) - \hat{y}(h)\right] \approx \hat{y}(h/2) - \hat{y}(h) \quad . \tag{8.26}$$

D.h.: Die Differenz zwischen den beiden Runge-Kutta-Ergebnissen $\hat{y}(h/2)$ (= Näherungsergebnis nach 2 Halbschritten) und $\hat{y}(h)$ (= Näherungsergebnis nach 1 Ganzschritt) kann *approximativ* mit dem auftretenden (absoluten) Verfahrensfehler gleichgesetzt werden.

Aus der Kombination der Gleichungen (8.25) und (8.26) folgt auch sofort eine Beziehung, die es erlaubt, *die erhaltenen Näherungswerte* $\hat{y}$ *auf einfache Weise zu verbessern*:

$$\hat{y}_{verbessert} = \hat{y}(h/2) + \frac{\hat{y}(h/2) - \hat{y}(h)}{15} \, . \tag{8.27}$$

Diese Korrektur hat jedoch ein Problem: man erhält zwar (i. a.) einen deutlich verbesserten Näherungswert für die Lösungsfunktion y, hat aber *keinerlei Information über die Qualität dieses korrigierten* y*!* Aus diesem Grund wird in vielen Programmen auf die Anwendung der obigen Formel verzichtet.

Nun kommt der entscheidende Punkt: Man kann sich die Frage stellen, *wie groß die ideale Schrittweite* h *hätte sein müssen (oder sein dürfen), um zu gewährleisten, daß* E_V *eine vorgegebene Fehlerschranke* ϵ *gerade nicht überschreitet.*
Da aber der Verfahrensfehler mit der fünften Potenz von h variiert, ist diese Frage leicht zu beantworten. Es gilt einfach:

$$\frac{\epsilon}{E_V(h)} = \left(\frac{h_{ideal}}{h}\right)^5$$

Daraus erhält man sofort *eine Bestimmungsgleichung für* h_{ideal}*:*

$$h_{ideal} = h \cdot \left(\frac{\hat{y}(h/2) - \hat{y}(h)}{\epsilon}\right)^{-\frac{1}{5}} \tag{8.28}$$

Diese Gleichung ist die Basis für eine *Schrittweitensteuerung*, wie sie im Programm RKQC (Abschnitt 8.5.2) verwendet wird. Dabei wird der aktuelle Verfahrensfehler gemäß (8.26) berechnet und mit der Fehlerschranke ϵ verglichen. Die weitere Vorgangsweise ist die folgende:

- $\epsilon \geq \hat{y}(h/2) - \hat{y}(h)$: Der Fehler des letzten Runge-Kutta-Schrittes ist kleiner als ϵ. *Für den nächsten Runge-Kutta-Schritt kann daher die Schrittweite gemäß (8.28) vergrößert werden.*

- $\epsilon < \hat{y}(h/2) - \hat{y}(h)$: Der Fehler des letzten Runge-Kutta-Schrittes ist größer als ϵ. *Die Schrittweite wird gemäß (8.28) verkleinert und der letzte Runge-Kutta-Schritt wird wiederholt.*

Mit dieser Strategie wird zweierlei erreicht: Zum ersten ist gewährleistet, daß die Schrittweite stets klein genug ist, um den jeweiligen (lokalen) Verfahrensfehler unter der geforderten Grenze ϵ zu halten, zum zweiten wird aber stets mit einer möglichst großen Schrittweite gearbeitet, was sich auf die Rechenzeit positiv auswirkt.

Abschließend soll noch einmal darauf hingewiesen werden, daß die hier vorgestellte Methode der Schrittweitensteuerung auf einer Reihe von Annahmen basiert (etwa der Annahme der Unabhängigkeit der Größe C von h), die durchaus nicht immer erfüllt sein müssen! Zusätzlich wird hier immer nur der *lokale* Fehler berücksichtigt, und dabei außer acht gelassen, daß sich in vielen Fällen die von Schritt zu Schritt auftretenden lokalen Fehler zu einem viel drastischeren *globalen* Fehler aufsummieren können.

Die Schrittweitensteuerung, die im nun folgenden Programm verwendet wird, versucht auf einige dieser Probleme Rücksicht zu nehmen.

8.5 Die Programme ODEINT, RKQC und RK4.

Quelle: [9], S. 550-560, vereinfacht.

Die Programme ODEINT, RKQC und RK4 dienen zur numerischen Auswertung eines Anfangswertproblems (8.1).

8.5.1 Das Programm ODEINT.

ODEINT (Ordinary Differential Equations INTegrator) ist das 'driver program' des Programmsystems:

INPUT-Parameter:

YSTART( ): Vektor $\mathbf{y}_o$ der Anfangswerte des Dgl.ssystems.

N: Anzahl n der Gleichungen des Systems.

X1, X2: Startpunkt und Endpunkt des Integrationsintervalls.

EPS: Geforderte *relative* Genauigkeit (s. die folgenden Anmerkungen).

HANF: Guessed value für die Schrittweite des Runga-Kutta-Prozesses.

HMIN: Minimalwert, unter den die Arbeitsschrittweite nicht sinken darf.

NSTMAX: Maximale Anzahl von Stützpunkten, die in den Feldern XX bzw. YY abgespeichert werden kann.

OUTPUT-Parameter:

NWERTE: Anzahl der abgespeicherten Stützpunkte.

XX( ): Abszissenwerte der Stützpunkte.

YY(,): Ordinatenwerte der Lösungsfunktionen:
erster Index = Index der Funktion,
zweiter Index = Kennzeichnung des Abszissenwertes.

Anmerkungen zu ODEINT:

Dieses Programm kontrolliert die schrittweise Abarbeitung des Runge-Kutta-Prozesses im Bereich X1≤X≤X2. Die (von der Routine RKQC gelieferten) qualitäts-kontrollierten Ergebnisse an den Stützpunkten werden auf den Feldern XX und YY abgespeichert. Auf die Zahl und die Verteilung dieser Werte im Intervall [X1,X2] hat man keinen Einfluß.
Die Näherungswerte $\hat{y}_i$ für das Ende des Integrationsintervalls (X2) werden zusätzlich über das Feld YSTART an das aufrufende Programm zurückgegeben. Dies ist praktisch für den Fall, daß diese $\hat{y}_i$-Werte als Anfangswerte für ein weiteres, unmittelbar anschließendes Integrationsintervall dienen sollen.
Eine weitere wichtige Aufgabe von ODEINT besteht in der Berechnung der

Skalierungswerte für die Fehlerdiagnostik. Da mit EPS eine *relative* Fehlerschranke gemeint ist, wäre es sinnvoll, bzgl. der Beträge der jeweiligen $\hat{y}_i$-Werte zu skalieren, also

```
YSCAL(I):= |Y(I)|
```

zu schreiben. Eine solche Skalierung würde aber im Falle einer Nullstelle von y_i zu einem Programmabbruch führen (Division durch Null im Programm RKQC); aus diesem Grund wird im Programm ODEINT wie folgt skaliert:

```
YSCAL(I):= |Y(I)| + |H*F(I)| + 'TINY'        ,
```

wodurch jede derartige Komplikation ausgeschlossen ist.

8.5.2 Das Programm RKQC.

RKQC (Runge-Kutta Quality Control): Dieses Programm führt die Qualitätskontrolle der einzelnen Runge-Kutta-Schritte und die damit zusammenhängende Schrittweiten-Steuerung durch.

INPUT-Parameter:

Y( ): $\hat{y}_i$-Werte des letzten Runge-Kutta-Schrittes =
Startwerte für den nächsten Schritt.

F( ): Feld der entsprechenden f_i-Werte.

N: Anzahl der Gleichungen des Dgl.ssystems.

X: Abszissenwert des aktuellen Startwertes.

HTRY: Schrittweite für den nächsten Schritt.

EPS: Geforderte *relative* Genauigkeit.

YSCAL( ): Scaling factors für die nächste Genauigkeitsabfrage.

OUTPUT-Parameter:

Y( ): Neu berechnete $\hat{y}_i$-Werte.

F( ): Feld der entsprechenden f_i-Werte.

X: Neu berechneter Abzissenwert.

HNEXT: Für den nächsten Schritt vorgeschlagene Schrittweite.

Anmerkungen zu RKQC:

In diesem Programm werden jeweils die drei für die Fehlerdiagnostik erforderlichen Runge-Kutta-Schritte durchgeführt, davon zwei Schritte mit $h/2$ und ein Schritt mit h. Danach erfolgt die approximative Berechnung des Verfahrensfehlers gemäß (8.26) für jede Lösungsfunktion $\hat{y}_i$. Der Maximalwert der *skalierten* Verfahrensfehler wird berechnet und unter ERRMAX abgespeichert. ERRMAX entspricht also dem Ausdruck

$$\mathrm{Max}\,[|\ \hat{y}(h/2) - \hat{y}(h)\ |_{rel}]$$

im Sinne von (8.26).

Dann erfolgt eine Programmverzweigung: Im Falle von

$$\mathrm{ERRMAX} \leq \mathrm{EPS}$$

war der Runge-Kutta-Schritt erfolgreich. Nun wird, abgesehen von einer letzten Korrektur der $\hat{y}$ gemäß (8.27), die ideale Schrittweite für den nächsten Schritt berechnet. Der entsprechende Befehl

$$\mathrm{HNEXT} := \mathrm{SAFETY{*}H{*}(ERRMAX/EPS){*}{*}}(-1.0/5.0)$$

entspricht bis auf die 'Sicherheitskonstante' SAFETY:=0.9 genau der Gleichung (8.28). Allerdings gibt es im Falle sehr kleiner Werte von ERRMAX noch eine Sicherheitsabfrage[2]. Um zu verhindern, daß HNEXT zu stark anwächst, wird im Falle

$$\mathrm{ERRMAX/EPS} < \mathrm{ERRCON}$$

die nächste Schrittweite einfach wie folgt berechnet:

$$\mathrm{HNEXT} := 4. * \mathrm{H}$$

Dann erfolgt der Rücksprung ins Programm ODEINT.

Im Falle von

$$\mathrm{ERRMAX} > \mathrm{EPS}$$

war der Runge-Kutta-Schritt nicht erfolgreich und muß daher mit einer kleineren Schrittweite wiederholt werden. Der entsprechende Befehl

$$\mathrm{H{:=}SAFETY{*}H{*}(ERRMAX/EPS){*}{*}}(-1.0/4.0)$$

unterscheidet sich von (8.28) außer durch SAFETY noch *durch einen veränderten Exponenten*: dies soll der bereits erwähnten Tatsache Rechnung tragen, daß die *globale* Fehlerzunahme in manchen Fällen dramatischer verläuft als die *lokale* Fehlerzunahme. Der gegenüber (8.28) verkleinerte Exponent sorgt in diesem Sinne für noch kleinere Schrittweiten und dementsprechend für eine noch günstigeres Fehlerverhalten des Runge-Kutta-Prozesses.

[2] ERRCON, die 'error control' Konstante, hat im Programm RKQC recht willkürlich den Wert $6 \cdot 10^{-4}$.

8.5.3 Die Programme RK4 und DERIVS.

RK4 (Runge-Kutta 4) führt die Auswertung der einzelnen Runge-Kutta-Schritte nach Maßgabe der Gleichungen (8.18) und (8.19) durch:

INPUT-Parameter:

Y( ): Startwerte für den aktuellen Runge-Kutta-Schritt.

G1( ): Feld der f_i-Werte am aktuellen Startpunkt.

N: Anzahl n der Gleichungen des Dgl.ssystems.

X: Abszissenwert des Startpunktes.

H: Schrittweite des aktuellen Runge-Kutta-Schrittes.

OUTPUT-Parameter:

YOUT( ): Runge-Kutta-Näherungswerte $\hat{y}_i$ an der Stelle X+H.

DERIVS:

Dieses Programm, das von allen drei Programmen dieses Systems aufgerufen wird, wird vom Benutzer erstellt und enthält die Definition der f_i-Funktionen (z. B. in C):

```
void derivs(double x, double y[], double f[])
{
  f[1]= .....;  // entspricht f_{1}(x,y1,y2,...)
  f[2]= .....;  // entspricht f_{2}(x,y1,y2,...)
    .
    .
  f[n]= .....;  // entspricht f_{n}(x,y1,y2,...)
}
```

Struktogramm 26 — ODEINT(YSTART,N,X1,X2,EPS,HANF,HMIN,NSTMAX, NWERTE,XX,YY)

```
TINY:=1.0E-30
X:=X1
XX(1):=X
H:=HANF

I=1(1)N
    Y(I):=YSTART(I)
    YY(I,1):=Y(I)

NSTP=1(1)NSTMAX-1
    DERIVS(X,Y,F)
    I=1(1)N
        YSCAL(I):=|Y(I)| + |F(I)*H| + TINY
    X+H > X2
        Y: H:=X2-X
        N: ......
    RKQC(Y,F,N,X,H,EPS,YSCAL,HNEXT)
    XX(NSTP+1):=X
    I=1(1)N
        YY(I,NSTP+1):=Y(I)
    X ≥ X2
        Y: I=1(1)N
               YSTART(I):=Y(I)
           NWERTE:=NSTP+1
           (return)
        N: ......
    |HNEXT| < HMIN
        Y: print: 'Schrittweite kleiner HMIN'
           (pause)
        N: H:=HNEXT

print:'ODEINT: mehr als NSTMAX Stuetzpunkte'
NWERTE:=NSTMAX
(return)
```

Struktogramm 27 — RKQC(Y,F,N,X,HTRY,EPS,YSCAL,HNEXT)

```
SAFETY:=0.9
ERRCON:=0.0006
XSAV:=X
I=1(1)N
    YSAV(I):=Y(I)
    FSAV(I):=F(I)
H:=HTRY
STEPOK:=FALSE
    HH:=H/2.0
    RK4(YSAV,FSAV,N,XSAV,HH,YTEMP)
    X:=XSAV+HH
    DERIVS(X,YTEMP,F)
    RK4(YTEMP,F,N,X,HH,Y)
    X:=XSAV+H
    X = XSAV
    Y: print: 'RKQC: Schrittweite zu klein.'
       (pause)
    N: ......
    RK4(YSAV,FSAV,N,XSAV,YTEMP)
    ERRMAX:=0.0
    I=1(1)N
        ERRV(I):=Y(I)-YTEMP(I)
        TEMP:=| ERRV(I)/YSCAL(I)|
        ERRMAX < TEMP
        Y: ERRMAX:=TEMP
        N: ......
    ERRMAX > EPS
    Y: H:=SAFETY*H*
       EXP(-0.25*LOG(ERRMAX/EPS))
    N: STEPOK:=TRUE
       ERRMAX/EPS < ERRCON
       Y: HNEXT:=4*H
       N: HNEXT:=SAFETY*H*
          EXP(-0.2*LOG(ERRMAX/EPS))
       I=1(1)N
           Y(I):=Y(I)+ERRV(I)/15.0 (s. Kommentar S. 247)
STEPOK
(return)
```

Struktogramm 28 — RK4(Y,G1,N,X,H,YOUT)

I=1(1)N
YT(I):=Y(I)+H/2.0*G1(I)
XTEMP:=X+H/2.0
DERIVS(XTEMP,YT,G2)
I=1(1)N
YT(I):=Y(I)+H/2.0*G2(I)
DERIVS(XTEMP,YT,G3)
I=1(1)N
YT(I):=Y(I)+H*G3(I)
XTEMP:=X+H
DERIVS(XTEMP,YT,G4)
I=1(1)N
YOUT(I):=Y(I)+H/6.0*(G1(I)+2*(G2(I)+G3(I))+G4(I))

8.5.4 Anwendung von ODEINT+RKQC+RK4 auf das 'Satelliten-Problem'.

Das eben erläuterte Programm-System soll nun auf das 'Satelliten-Testproblem' angewendet werden. Das dem Dgl.ssystem (8.22) entsprechende Unterprogramm DERIVS hat z.B. in C die Form

```
void derivs(float x, float y[], float f[])
{
  f[1]=y[3];
  f[2]=y[4];
  f[3]=y[1]*y[4]*y[4] -alpha/y[1]/y[1];   // alpha GLOBAL = gamma*M
                                          //              = 1966.390
  f[4]=-2.0*y[3]*y[4]/y[1];
}
```

Die Ergebnisse dieses Tests sind in der Abb.8.6 zusammengefaßt, umd zwar wird in dieser Abbildung die (analytisch berechnete) exakte Bahnellipse mit Runge-Kutta-Ergebnissen verglichen, die mittels eines Programmes ohne (RUKUTEST) bzw. mit Schrittweitensteuerung (ODEINT+RKQC+RK4) berechnet wurden.

Die Ergebnisse:

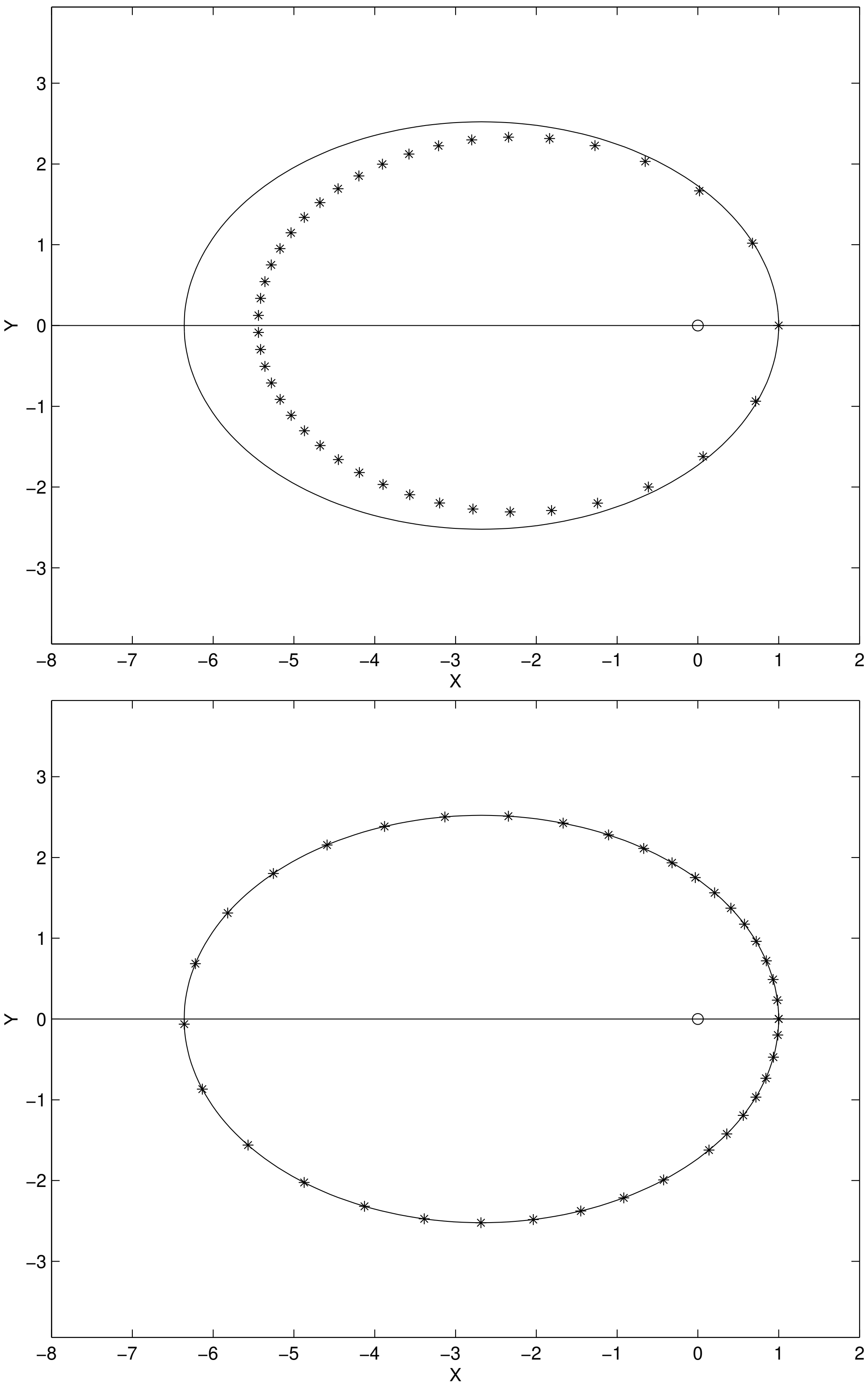


Abbildung 8.6: Effizienz der Schrittweitensteuerung beim 'Satelliten-Problem'. Vergleich der exakten elliptischen Bahnkurve (volle Linie) mit numerischen Werten (Sterne). Oben: Runge-Kutta *ohne* Schrittweiten-Steuerung, unten: Runge-Kutta *mit* Schrittweiten-Steuerung.

- Programm ohne Schrittweitensteuerung (Abb.8.6 oben):

  Bei diesem Testlauf wurde als konstante Schrittweite 1/50 der Umlaufzeit genommen. Diese Konstanz von h im *Zeitraum* hat folgende Konsequenz: Da bekanntlich ein Satellit sich umso langsamer bewegt, je entfernter er von der Erde ist (2. Kepler'sches Gesetz), haben die Stützpunkte im *Ortsraum* gerade dort die größte Dichte, wo der Satellit die kleinste Bahngeschwindigkeit hat. Dort aber, wo diese Geschwindigkeit am größten ist (und sich am stärksten ändert), nämlich im Bereich, wo der Satellit der Erde am nächsten kommt, ist die Punktdichte am kleinsten. Gerade in diesem Bereich sollte sie aber im Interesse eines kleinen Verfahrensfehlers möglichst groß sein!

  Die Folge dieser völlig unangepaßten Stützpunktverteilung ist im Diagramm links deutlich zu sehen: Man erkennt sehr große Differenzen zwischen der exakten und der numerisch ermittelten Bahnkurve.

- Programm mit Schrittweitensteuerung, EPS=0.0001 (Abb.8.6 unten):

  In diesem Fall hat die Punktdichte genau dort ein Maximum, wo es numerisch am sinnvollsten ist (also im Bereich des kleinsten Satellit-Erde-Abstandes). Der Effekt ist evident: Bei einer ungefähr gleichen Gesamtzahl von Stützpunkten ist (zumindest grafisch) kein Unterschied zwischen der analytischen und der numerischen Bahnkurve zu erkennen!

8.6 Das Runge-Kutta-Fehlberg-Verfahren.

Die im Abschnitt (8.4.5) beschriebene Fehleranschätzung und Schrittweiten-Steuerung im vorgestellten Runge-Kutta-Programm ist zwar sehr effektiv, verbraucht aber viel Rechenzeit, denn es laufen ja zwei unabhängige Runge-Kutta-Prozesse nebeneinander.

Ebenso effektiv, aber deutlich sparsamer bzgl. der Rechenzeit ist die folgende Variante der Schrittweiten-Steuerung, die auf *Fehlberg* zurückgeht. Die dazu nötigen Formeln werden im folgenden wieder für den einfachen Fall eines Differentialgleichungs'systems' der Ordnung 1 erläutert[3]:

Ein Rechenschritt von x_0 nach $x_0 + h$ mittels eines Runge-Kutta-Verfahrens der Ordnung p ergibt

$$\hat{y}^{(p)}(x_0 + h) = y_{exakt} + Ch^{p+1} \,. \tag{8.29}$$

Dieselbe Rechnung mittels einer Runge-Kutta-Formel der nächsthöheren Ordnung $p + 1$ führt zu

$$\hat{y}^{(p+1)}(x_0 + h) = y_{exakt} + \tilde{C}h^{p+2} \,. \tag{8.30}$$

Die Differenz von (8.29) und (8.30)

$$\hat{y}^{(p)}(x_0 + h) - \hat{y}^{(p+1)}(x_0 + h) = Ch^{p+1} - \tilde{C}h^{p+2} \approx Ch^{p+1} \quad \text{für kleine } h$$

[3]Literatur dazu s. z. B. [10], S. 714ff, [23], S. 229ff

kann nach C aufgelöst werden:

$$C = \frac{\hat{y}^{(p)}(x_0+h) - \hat{y}^{(p+1)}(x_0+h)}{h^{p+1}}\,. \tag{8.31}$$

Mit diesem Näherungswert für C kann der Fehler für den Runge-Kutta Schritt der Ordnung p (8.29) abgeschätzt werden. Daraus kann man nun jene (ideale) Schrittweite h_{ideal} berechnen, die eine vorgegebene Fehlerschranke ϵ *gerade nicht überschreitet*:

$$\epsilon = C h_{ideal}^{p+1} = \left(\frac{h_{ideal}}{h}\right)^{p+1} |\hat{y}^{(p)} - \hat{y}^{(p+1)}|$$

bzw.

$$h_{ideal} = h\left(\frac{|\hat{y}^{(p)}(x_0+h) - \hat{y}^{(p+1)}(x_0+h)|}{\epsilon}\right)^{-1/(p+1)}\,. \tag{8.32}$$

Vergleichen Sie dieses Ergebnis mit Glg. (8.28).

Damit ist die Vorgangsweise beim *Runge-Kutta-Fehlberg-Verfahren* klar: man berechnet für den Schritt $x_0 \to x_0+h$ die Näherungswerte $\hat{y}^{(p)}(x_0+h)$ und $\hat{y}^{(p+1)}(x_0+h)$. Daraus bestimmt man mittels (8.32) die 'ideale' Schrittweite. Nun gibt es zwei Möglichkeiten:

1. wenn $h_{ideal} < h$ $\quad\to\quad$ Schritt von x_0 nach x_0+h wird mit h_{ideal} wiederholt,

2. wenn $h_{ideal} \geq h$ $\quad\to\quad$ der nächste Schritt wird mit h_{ideal} durchgeführt.

Wo liegt nun gegenüber der Vorgangsweise in Abschnitt 8.4.5 der Vorteil? Wie Sie aus der Runge-Kutta-Theorie wissen, gibt es zu jeder Ordnung $p > 1$ unendlich viele im Prinzip gleichwertige Runge-Kutta-Formeln. Fehlberg hat es nun geschafft, das folgende Formelsystem zu finden:

$$\begin{aligned}
f_0 &= f(x_0, y_0)\,, \\
f_1 &= f(x_0 + \frac{h}{4}, y_0 + \frac{h}{4}f_0)\,, \\
f_2 &= f(x_0 + \frac{3h}{8}, y_0 + \frac{3h}{32}f_0 + \frac{9h}{32}f_1)\,, \\
f_3 &= f(x_0 + \frac{12h}{13}, y_0 + \frac{1932h}{2197}f_0 - \frac{7200h}{2197}f_1 + \frac{7296h}{2197}f_2)\,, \\
f_4 &= f(x_0 + h, y_0 + \frac{439h}{216}f_0 - 8hf_1 + \frac{3680h}{513}f_2 - \frac{845h}{4104}f_3)\,, \\
f_5 &= f(x_0 + \frac{h}{2}, y_0 - \frac{8h}{27}f_0 + 2hf_1 - \frac{3544h}{2565}f_2 + \frac{1859h}{4104}f_3 - \frac{11h}{40}f_4)\,.
\end{aligned}$$

Mit diesen Definitionen kann sowohl eine Runge-Kutta-Formel vierter Ordnung gebildet werden, nämlich

$$\hat{y}^{(4)} = y_0 + h\left(\frac{25}{216}f_0 + \frac{1408}{2565}f_2 + \frac{2197}{4104}f_3 - \frac{1}{5}f_4\right)$$

als auch eine Runge-Kutta-Formel fünfter Ordnung, nämlich

$$\hat{y}^{(5)} = y_0 + h\left(\frac{16}{135}f_0 + \frac{6656}{12825}f_2 + \frac{28561}{56430}f_3 - \frac{9}{50}f_4 + \frac{2}{55}f_5\right) .$$

Damit ergibt sich sofort für den Fehler der Ausdruck

$$\hat{y}^{(4)}(x_0+h) - \hat{y}^{(5)}(x_0+h) = -h\left(\frac{1}{360}f_0 - \frac{128}{4275}f_2 - \frac{2197}{75240}f_3 + \frac{1}{50}f_4 + \frac{2}{55}f_5\right) ,$$

und die Berechnung der idealen neuen Schrittweite erfolgt für p=4 gemäß (8.32) durch

$$h_{ideal} = h\left(\frac{|\hat{y}^{(4)}(x_0+h) - \hat{y}^{(5)}(x_0+h)|}{\epsilon}\right)^{-1/5} . \qquad (8.33)$$

8.7 Weitere Verfahren zur numerischen Behandlung von Anfangswertproblemen.

Dem letzten Abschnitt dieses Kapitels sei ein ganz im saloppen Stil amerikanischer Publikationen gehaltener Absatz aus den 'Numerical Recipes' [9] vorangestellt:

For many scientific users, forth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact, you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm ... Keep in mind, however, that the old workhorse's last trip may well be to take you to the poorhouse: Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. Those methods are the high-strung racehorses. Runge-Kutta is for ploughing the fields.

In der Tat sind die hier im Detail vorgestellten Verfahren (Runge-Kutta und Runge-Kutta-Fehlberg) nicht der Weisheit allerletzter Schluß, wenn es um die numerische Behandlung von Anfangswertproblemen geht. Dennoch erschien es dem Autor dieses Skriptums besser, ein klassisches Verfahren (auch wenn es ein alter Ackergaul ist) in einem sehr robusten und brauchbaren Programm vorzustellen, als die manchmal (nicht immer!) überlegenen 'Rennpferde' (Bulirsch-Stoer-Methode und predictor-corrector-Methode). Für ein an sich wünschenswertes 'Sowohl-Als auch' ist jedoch der hier gesteckte Rahmen einfach zu knapp.

Ich muß daher (nicht zum ersten Mal in diesem Skriptum) an die Eigeninitiative der an numerischen Verfahren interessierten Hörer appellieren und gebe im folgenden nur eine kleine Literaturliste:

- Näherungsweise Lösung gewöhnlicher Differentialgleichungen: [19], S.127-228 (ausführliche Theorie).

- Ordinary Differential Equations: [7], S.360-414 (wie immer sehr zu empfehlen: Theorie, numerische Probleme, Vergleich von Methoden usw.).

- Prädiktor-Korrektor-Verfahren nach Adams-Bashford-Moulton: [2], S.220ff (Theorie), S.428ff (Programm).
- Extrapolationsverfahren nach Bulirsch-Stoer: [2], S.233ff (Theorie), S.434ff (Programm).
- Richardson Extrapolation and the Bulirsch-Stoer Method: [9], S.563ff (kurze Beschreibung und Programme).
- Predictor-Corrector Methods: [9], S.569ff (kurze Beschreibung).
- Prädiktor-Korrektor-Verfahren nach Adams: [13], S.288f (Theorie), S.290f (Algol-Programm).
- Natürlich hat jede professionelle numerisch-mathematische Library eine Reihe von einschlägigen Programmen (s. Abschnitt 8.7).

8.7.1 Steife Systeme von Differentialgleichungen

Noch einige Anmerkungen zu sog. *stiff sets of differential equations*, die in der Praxis immer wieder vorkommen. Es handelt sich dabei um Systeme, deren Lösungsfunktionen aus Termen bestehen, die extrem verschiedene Abhängigkeits-Empfindlichkeiten von der unabhängigen Variablen haben.
Am einfachsten ist dies durch ein Beispiel zu erläutern (aus [10], S. 734):

Angenommen, es soll das System

$$y_1'(x) = 998\, y_1 + 1998\, y_2$$
$$y_2'(x) = -999\, y_1 + 1999\, y_2$$

mit den Anfangsbedingungen $y_1(0) = 1$ und $y_2(0) = 0$ gelöst werden.

Die exakte Lösung dieses Problems lautet

$$y_1(x) = 2\,\mathrm{e}^{-x} - \mathrm{e}^{-1000x} \qquad \text{and} \qquad y_2(x) = -\mathrm{e}^{-x} + \mathrm{e}^{-1000x}\,. \tag{8.34}$$

Es zeigt sich, daß die numerische Berechnung solcher Probleme mit konventionellen Methoden (z. B. mit Runge-Kutta) zu einer *Instabilität* des Verfahrens führt, wenn man nicht mit extrem kleinen Schrittweiten [im konkreten Fall mit $h << 1/1000$ (!)] arbeitet. Dies, obwohl die zweiten Terme in (8.34) bereits für sehr kleine x praktisch auf Null abgeklungen sind und die danach die Lösungen simple $\exp(-x)$-Funktionen darstellen.

Für solche Probleme gibt es spezielle Lösungsverfahren, wie zum Beispiel die Methode von Kaps und Rentrop[4]. Entsprechende C-Programme mit theoretischen Erläuterungen finden Sie in [10], S. 734ff.

Mehr zum Thema 'Numerische Behandlung von Anfangswertproblemen' können Sie in meiner Lehrveranstaltung im SS *Ausgewählte Kapitel aus 'Numerische Methoden in der Physik'* erfahren.
Stichworte: Methode von Bulirsch und Stör; *predictor-corrector*-Methode; steife Systeme usw.

[4] P. Kaps und P. Rentrop, *Numerische Mathematik* **33**, S. 55ff.

8.8 Software-Angebot

NAG: Die folgende Tabelle gibt einen Überblick über das Angebot dieser Library in bezug auf Cauchy-Probleme. Verwendet wird primär die Runge-Kutta-Methode (Programme *d02puf* und *d02pcf*). Bei Problemen mit großem Integrationsbereich und hohen Genauigkeitsansprüchen kommt die Methode nach <u>Adams</u> zum Einsatz (Programme *d02cjf* und andere). Als drittes Verfahren wird die BDF-Methode (*Backward Differentiation Formula*) angeboten; diese Methode hat sich vor allem bei *stiff equations* (s. Abschnitt 8.7.1) bewährt.

Problem	Routine		
	RK method	Adams method	BDF method
Initial-value Problems **Driver Routines**			
Integration over a range with optional intermediate output and optional determination of position where a function of the solution becomes zero	–	D02CJF	D02EJF
Integration over a range **–with intermediate output**	D02PCF	D02CJF	D02EJF
–until function of solution becomes zero	D02BHF*	D02CJF	D02EJF
Comprehensive Integration routines	D02PCF, D02PDF D02PVF, D02PWF D02PXF, D02PYF	D02QFF, D02QGF D02QWF, D02QXF D02QYF, D02QZF	D02M routines D02N routines D02XKF, D02XJF and D02ZAF
Package for Solving Stiff Equations	D02M-D02N Subchapter		
Package for Solving Second-order Systems of Special Form	D02L routines		

ODEPACK (in www.netlib.org): (p.d.) Paket mit F77-Programmen für Anfangswertprobleme gewöhnlicher Differentialgleichungen, sowohl für nicht-steife (*predictor-corrector*-Methode von Adams) als auch für steife Systeme (*backward differentiation formula* von Gear).

toms-504.f (in www.netlib.org): *Public domain* Programm 'GERK' von H. A. Watts und L. F. Shampine: *Fehlberg fourth (fifth) order Runge-Kutta method with global error assessment to solve a system of differential equations.* Beschreibung der Parameter im Programmlisting.

Numerical Recipes: In diesen Büchern finden Sie eine Reihe von ODE-Programmen[5] in F77, F90 und C:

```
odeint+rkqs+rkck             Runge-Kutta-Fehlberg-Methode
odeint+bsstep+mmid+pzextr    Bulirsch-Stoer-Methode
stiff+jacobn+ludcmp+lubksb   stiff differential systems
```

Eine Beschreibung dieser Programme bzw. Testprogramme finden Sie in [9], [10] sowie auf Ihrem PC im Computerraum in den Verzeichnissen

[5]ODE = *Ordinary Differential Equations.*

```
/usr/local/numeric/numrec/c  (oder fortran-77 oder fortran-90)
```

Numerical Algorithms with C or FORTRAN In diesem Werk werden ebenfalls etliche ODE-Programme(F77/F90/C) angeboten. Die Parameter bzw. abhängigen Unterprogramme werden in den Listings beschrieben. Hier eine Auswahl:

```
rk_fehl.c           Runge-Kutta-Fehlberg-Methode
bulirsch.c          Bulirsch-Stoer-Gregg-Methode
ab_mou.c            Praediktor-Korrektor-Verfahren von
                    Adams-Bashforth-Moulton
gear.c              Verfahren von Gear fuer steife Dgl.en

(inkl. verschiedene Header-Dateien)

rktrb.f90  plus dependencies    Runge-Kutta-Methoden
frkfsy.f90 plus dependencies    Runge-Kutta-Fehlberg-Methode
hull.f90   plus dependencies    Runge-Kutta-Hull-Methode
irkdrv.f90 plus dependencies    Implicite Runge-Kutta-Methoden
gear4.f90  plus dependencies    stiff differential equations

Alle diese Programme finden Sie in den Verzeichnissen
/usr/local/numeric/num_alg/c/ansic/ansic/17   ('17' bedeutet: Kap. 17)
/usr/local/numeric/num_alg/f77/f90/kap17
```

MATLAB.6: Dieses Programmsystem bietet ebenfalls eine grosse Auswahl von ODE-Programmen an; Infos darüber s. das *MATLAB function reference*. Das Standardprogramm für *non-stiff systems* heißt *ode45*, für *stiff systems* gibt es *ode15s*.
Eine ausführliche Beschreibung des jeweiligen Matlab-Programms inkl. Testbeispiel und Programm-Listing erhalten Sie durch Eingabe der folgenden Befehle (z. B.) im Matlab-Fenster:

```
more on
type ode45
more off
```

Wer mehr über dieses Thema wissen möchte, beschaffe sich das folgende Buch:
D. Arnold and J. C. Polking, *Ordinary Differential Equations Using Matlab*, Prentice Hall 1999.

Kapitel 9

Numerische Methoden zur Lösung von gewöhnlichen Differentialgleichungen: Randwertprobleme.

9.1 Das lineare Randwertproblem zweiter Ordnung.

Es ist im Rahmen dieser Lehrveranstaltung völlig unmöglich, das sehr umfangreiche Gebiet der numerischen Behandlung von *Randwertproblemen* (RWP) umfassend zu behandeln. Ich habe daher aus der großen Mannigfaltigkeit der Probleme jene Gruppe herausgegriffen, die in der Praxis von Physik und Technik die wichtigste Rolle spielt, nämlich

lineare Randwertprobleme zweiter Ordnung mit entkoppelten Randbedingungen.

Dieses Problem besteht in mathematischer Hinsicht aus der gewöhnlichen linearen Differentialgleichung (Dgl.) zweiter Ordnung

$$r(x)\, y''(x) + s(x)\, y'(x) + q(x)\, y(x) = f(x) \qquad (x \in [a, b]) \tag{9.1}$$

mit den in bezug auf die Intervallgrenzen a und b *entkoppelten* Randbedingungen

$$\alpha_0\, y(a) + \alpha_1\, y'(a) = \gamma \qquad (|\alpha_0| + |\alpha_1| \neq 0) \tag{9.2}$$

und

$$\beta_0\, y(b) + \beta_1\, y'(b) = \delta \qquad (|\beta_0| + |\beta_1| \neq 0) \quad . \tag{9.3}$$

Man nennt ein Problem (9.1–9.3) homogen, wenn gilt:

$$f(x) \equiv 0 \qquad \gamma = 0 \qquad \delta = 0\,.$$

In allen anderen Fällen liegt ein inhomogenes RWP vor.

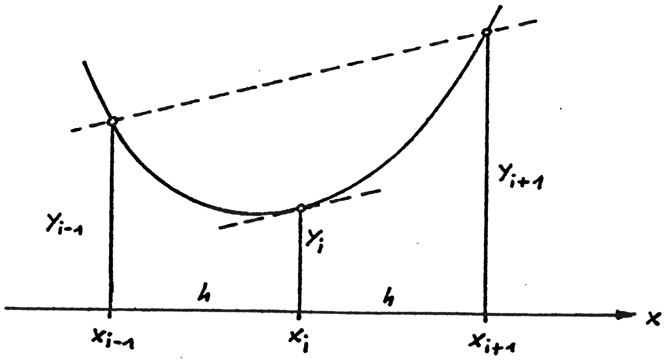


Abbildung 9.1: Grafische Interpretation des finiten Ausdrucks (9.5).

9.2 Numerische Behandlung des inhomogenen RWP mittels des Differenzenverfahrens.

Der wichtigste Schritt beim Differenzenverfahren besteht darin, daß das kontinuierliche Intervall, innerhalb dessen die Lösungsfunktion $y(x)$ gesucht wird, durch *Aufteilung in N (gleich-breite) Subintervalle* diskretisiert wird. Man erhält auf diese Weise die $N+1$ äquidistanten Stützpunkte

$$x_i = a + (i-1)h \qquad \text{mit } h = \frac{b-a}{N} \quad \text{und } (i = 1, \ldots, N+1)\,. \tag{9.4}$$

An diesen Stützpunkten werden nun alle in der Dgl. und in den Randbedingungen auftretenden *Differentialquotienten* durch *geeignete Differenzenquotienten* ersetzt.
Es gibt nun in der Literatur zahlreiche Formeln für solche Differenzenquotienten (s. z. B. [20], S. 267 und 267). In vielen Fällen sind jedoch die einfachsten derartigen Formeln die numerisch praktikabelsten. Aus diesem Grund werden in diesem Kapitel aussschließlich die sogenannten 'symmetrischen Dreipunktsformeln'[1]

$$y_i' \approx \frac{y_{i+1} - y_{i-1}}{2h} \tag{9.5}$$

bzw.

$$y_i'' \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \tag{9.6}$$

verwendet.
Die grafische Interpretation von (9.5) und (9.6) ist besonders einfach: Die Steigung der Kurve im Punkt x_i wird approximiert durch die Steigung der Verbindungsgeraden, die durch den rechten und den linken Nachbarpunkt hindurchgeht (s. Abb. 9.1).
Entsprechend wird die zweite Ableitung der Funktion $y(x)$ im Punkt x_i durch die zweite Ableitung jener Parabel angenähert, die durch den Punkt (x_i, y_i) selbst sowie durch seinen rechten und linken Nachbarn hindurchgeht.

[1] Im folgenden wird stets die Nomenklatur $y(x_i) \equiv y_i$ etc. verwendet.

Ersetzt man nun die erste und zweite Ableitung in (9.1) durch die Ausdrücke (9.5) und (9.6), so erhält man für $i = 1, \ldots, N+1$ die *Differenzengleichungen*

$$r_i \frac{(y_{i+1} - 2y_i + y_{i-1})}{h^2} + s_i \frac{(y_{i+1} - y_{i-1})}{2h} + q_i y_i = f_i$$

bzw.

$$\left(\frac{r_i}{h^2} - \frac{s_i}{2h} \right) y_{i-1} + \left(q_i - \frac{2r_i}{h^2} \right) y_i + \left(\frac{r_i}{h^2} + \frac{s_i}{2h} \right) y_{i+1} = f_i \,. \tag{9.7}$$

Dieses System von Differenzengleichungen bildet ein lineares, inhomogenes Gleichungssystem für die unbekannten Funktionswerte der Lösungsfunktion des RWP. an den $N+1$ Stützpunkten.
Allerdings ist das System (9.7) noch unterbestimmt, denn es enthält (in der ersten Gleichung) die Unbekannte y_0 sowie (in der $(N+1)$ten Gleichung) die Unbekannte y_{N+2}. Diese beiden Größen können, wie gleich gezeigt wird, mittels der beiden Randbedingungen (9.2) und (9.3) aus dem System eliminiert werden.

Einbeziehung der ersten Randbedingung:

Diese Randbedingung beschreibt das Verhalten der Lösungsfunktion an der Stelle $x = a$ d.h. für $i = 1$:

$$y(a) \equiv y_1 \qquad \text{und} \qquad y'(a) \equiv y'_1 \approx \frac{y_2 - y_0}{2h}$$

bzw.

$$\alpha_0 y_1 + \alpha_1 \frac{y_2 - y_0}{2h} = \gamma \,.$$

Diese Gleichung wird nach y_0 aufgelöst, und man erhält *unter der Voraussetzung* $\alpha_1 \neq 0$

$$y_0 = \frac{1}{\alpha_1} \left(2h\alpha_0 y_1 + \alpha_1 y_2 - 2h\gamma \right) \,. \tag{9.8}$$

y_0 setzt man in die erste Differenzengleichung von (9.7) ein, also in die Gleichung

$$\left(\frac{r_1}{h^2} - \frac{s_1}{2h} \right) y_0 + \left(q_1 - \frac{2r_1}{h^2} \right) y_1 + \left(\frac{r_1}{h^2} + \frac{s_1}{2h} \right) y_2 = f_1 \,.$$

Dies ergibt die neue erste Differenzengleichung

$$\left[q_1 - \frac{2r_1}{h^2} - \frac{\alpha_0}{\alpha_1} \left(s_1 - \frac{2r_1}{h} \right) \right] y_1 + \frac{2r_1}{h^2} y_2 = f_1 - \frac{\gamma}{\alpha_1} \left(s_1 - \frac{2r_1}{h} \right) \,. \tag{9.9}$$

Dies gilt, wie bereits gesagt, für $\alpha_1 \neq 0$. Im Falle $\alpha_1 = 0$ ergibt sich direkt aus der ersten Randbedingung die Gleichung

$$y_1 = \frac{\gamma}{\alpha_0} \,. \tag{9.10}$$

Einbeziehung der zweiten Randbedingung:

Die Behandlung der Randbedingung (9.3) geschieht äquivalent zu den obigen Überlegungen und führt unter der Bedingung $\beta_1 \neq 0$ zur folgenden neuen *letzten* (d. h. $(N+1)$-ten) Differenzengleichung

$$\frac{2r_{N+1}}{h^2} y_N + \left[q_{N+1} - \frac{2r_{N+1}}{h^2} - \frac{\beta_0}{\beta_1} \left(s_{N+1} + \frac{2r_{N+1}}{h} \right) \right] y_{N+1} = f_{N+1} - \frac{\delta}{\beta_1} \left(s_{N+1} + \frac{2r_{N+1}}{h} \right) . \tag{9.11}$$

Im Falle $\beta_1 = 0$ ergibt sich aus (9.3) sofort

$$y_{N+1} = \frac{\delta}{\beta_0} . \tag{9.12}$$

Auf diese Weise hat man das gegebene RWP in ein inhomogenes, lineares Gleichungssystem von $N+1$ Gleichungen für die $N+1$ Unbekannten $y_1, y_2, \ldots, y_{N+1}$.
Die erste Gleichung dieses Systems ist gegeben durch (9.9) bzw. (9.10), die zweite bis N-te Gleichung hat die Form (9.7), und die $(N+1)$-te Gleichung lautet (9.11) bzw. (9.12).

Wie bereits erwähnt, stellen die Lösungen dieses Systems die approximativen Werte der Lösungsfunktion $y(x)$ des RWP an den Stützpunkten $x_1 \ldots x_{N+1}$ dar. Dieses System muß also (i. a. mit numerischen Mitteln) gelöst werden. Eine Analyse der Differenzengleichungen zeigt Ihnen sofort, daß die Koeffizientenmatrix des linearen Gleichungssystems eine *tridiagonale* Matrix ist. Diese einfache Form ist der speziellen Wahl der finiten Elemente (9.5) und (9.6) zu danken!

Wie Sie sich erinnern, gibt es für die Lösung von linearen Gleichungssystemen mit tridiagonaler Matrix sehr effizente numerische Methoden. Eine solche finden Sie im Abschnitt 2.6.1 dieses Skriptums.

9.2.1 Fehlerdiagnostik und Fehlerkorrektur.

Wiederholt man die Berechnung der y_i mit der *doppelten Zahl von Subintervallen* (d.h.: $N \to 2N$), so entspricht wegen der äquidistanten Stützpunkte jeder zweite Punkt der '$2N$-Rechnung' einem Punkt der 'N-Rechnung'. Für diese Punkte kann man nun eine sehr einfache und äquidistante Reduktion des beim Differenzenverfahren auftretenden Verfahrensfehlers vornehmen; diese Methode beruht auf dem in der Literatur beschriebenen Faktum, daß der Verfahrensfehler für y_i die Form

$$E_V = \frac{C(N)}{N^2} \tag{9.13}$$

hat. Der (unbekannte) exakte Wert von y_i kann also in der Form

$$y_{i,exakt} = \hat{y}_i(N) + \frac{C_i(N)}{N^2}$$

geschrieben werden, wobei $\hat{y}(N)$ den mittels des Differenzenverfahrens (N Subintervalle) erhaltenen Näherungswert bedeutet.
Verdoppelt man die Anzahl der Subintervalle, so ergibt sich

$$y_{i,exakt} = \hat{y}_i(2N) + \frac{C_i(2N)}{(2N)^2} \,.$$

Wie im ganz ähnlichen Fall der Fehlerabschätzung beim Runga-Kutta-Verfahren (s. Abschnitt 8.4.5), macht man auch hier die Approximation

$$C_i(N) \approx C_i(2N) = C_i$$

und man erhält sofort

$$C_i = \frac{4N^2}{3} \left[\hat{y}_i(2N) - \hat{y}_i(N)\right]$$

bzw. die Korrekturformel

$$y_{i,korr} = \frac{4\hat{y}_i(2N) - \hat{y}_i(N)}{3} \,. \tag{9.14}$$

Die Leistungsfähigkeit dieser Formel wird in den folgenden Testbeispielen demonstriert.

9.2.2 Das Programm DIFF1.

Dieses Programm löst das RWP (9.1)–(9.3) mittels des Differenzenverfahrens. Die Hauptpunkte des Programmes sind:

1. Berechnung der Funktionswerte von $r(x)$, $s(x)$, $q(x)$ und $f(x)$ für die Stützpunkte $x_i = a + (i-1)h$ mit $h = (b-a)/N$ und $i = 1, \ldots, N+1$ mittels der Prozedur **FCT1**.

2. Berechnung der 3 Vektoren $\mathbf{a}, \mathbf{b}, \mathbf{c}$ der tridiagonalen Matrix sowie des inhomogenen Vektors gemäß den Gleichungen (9.9; 9.10), (9.7) und (9.11;9.12).
   Achtung auf die Sonderfälle $\alpha_1 = 0$ und/oder $\beta_1 = 0$.

3. Aufruf des Programmes **TRID** (s. Abschnitt 2.6.1). Der Lösungsvektor $\mathbf{y}$ stellt die Näherungen der Lösungsfunktion $y(x)$ des RWP an den Stützpunkten x_i dar.

4. Die Schritte (1)–(3) werden zweimal durchgeführt, einmal mit der gewählten Intervallanzahl N, und ein zweitesmal mit der Intervallanzahl $2N$.

5. Die entsprechenden $\hat{y}$-Werte werden mittels der Fehlerkorrekturformel (9.14) verbessert.

INPUT-Parameter:

ANF, AEND: a und b in (9.1).

ALP0,ALP1,BET0,BET1: Parameter der entkoppelten Randbedingungen (9.2) und (9.3).

GAMMA, DELTA: inhomogene Parameter in (9.2) und (9.3).

N0: Anzahl der Subintervalle für *den ersten Durchlauf* in DIFF1.

OUTPUT-Parameter:

Y(): eindim. Feld mit den Näherungswerten der Lösungsfunktion $y(x)$ an den Stützpunkten im Intervall $[a,b]$.

INTERNE FELDER:

YY(,): zweidim. Feld mit den Näherungswerten der Lösungsfunktion $y(x)$ für N und $2N$ Subintervalle.

Das Programm DIFF1 benötigt die Prozeduren FCT1 und TRID.

Struktogramm 29 — DIFF1(ANF,AEND,ALP0,ALP1,BET0,BET1,GAMMA, DELTA,N0,Y)

```
T=1(1)2
    N:=T*N0
    H:=(AEND-ANF)/N
    HQ:=H*H
    I=2(1)N
        X:=ANF+(I-1)*H
        FCT1(X,R,S,Q,F)
        A(I):=R/HQ-S/2.0/H
        B(I):=Q-2.0*R/HQ
        C(I):=R/HQ+S/2.0/H
        INHOM(I):=F
    ALP1=0.0
    Y:
        B(1):=1.0
        C(1):=0.0
        INHOM(1):=GAMMA/ALP0
    N:
        FCT1(ANF,R,S,Q,F)
        B(1):=Q-2.0*R/HQ - ALP0/ALP1*(S-2.0*R/H)
        C(1):=2.0*R/HQ
        INHOM(1):=F - GAMMA/ALP1*(S-2.0*R/H)
    BET1=0.0
    Y:
        A(N+1):=0.0
        B(N+1):=1.0
        INHOM(N+1):=DELTA/BET0
    N:
        FCT1(AEND,R,S,Q,F)
        A(N+1):=2.0*R/HQ
        B(N+1):=Q-2.0*R/HQ - BET0/BET1*(S+2.0*R/H)
        INHOM(N+1):=F - DELTA/BET1*(S+2.0*R/H)
    NP1:=N+1
    TRID(A,B,C,INHOM,NP1,Y)
    I=1(1)N0+1
        T=1
        Y:
            INDEX:=I
        N:
            INDEX:=2*I-1
        YY(I,T):=Y(INDEX)
H:=(AEND-ANF)/N0
I=1(1)N0+1
    X:=ANF+(I-1)*H
    Y(I):=(4.0*YY(I,2)-YY(I,1))/3.0
(return)
```

9.2.3 Ein Testbeispiel für DIFF1.

Das nun folgende Beispiel soll Ihnen die Leistungsfähigkeit von DIFF1 demonstrieren. Es ist der Referenz [19], S. 252f entnommen und lautet:

$$y''(x) - 2xy'(x) - 2y = -4x \qquad (x \in [0,1])$$

und

$$y(0) - y'(0) = 0 \qquad \text{bzw.} \qquad y(1) = 1 + \mathrm{e}\,.$$

Die exakte Lösung dieses RWP lautet:

$$y(x) = x + \mathrm{e}^{x^2}\,.$$

Der Benutzer hat also eine Prozedur FCT1 zu schreiben, welche z. B. die Form hat:

```
.
PROCEDURE FCT1(x: real; VAR r,s,q,f: real);

BEGIN
  r:=1.0;
  s:=-2.0*x;
  q:=-2.0;
  f:=-4.0*x
END;
```

Die weiteren Parameter lauten:

```
.
  anf=0.0    aend=1.0

  alp0=1.0   alp1=-1.0     bet0=1.0   bet1=0.0
        gamma=0.0              delta=1.0 + e
```

Die Auswertung mittels DIFF1 wurde für N0=10 Subintervalle durchgeführt, d.h. das Programm DIFF1 führte die Rechnung einmal mit 10 Subintervallen und ein zweitesmal mit 20 Subintervallen durch; dann wurde die Korrekturformel (9.14) angewendet.
Die Ergebnisse sehen Sie in den folgenden Tabellen sowie in der Abb. 9.2.

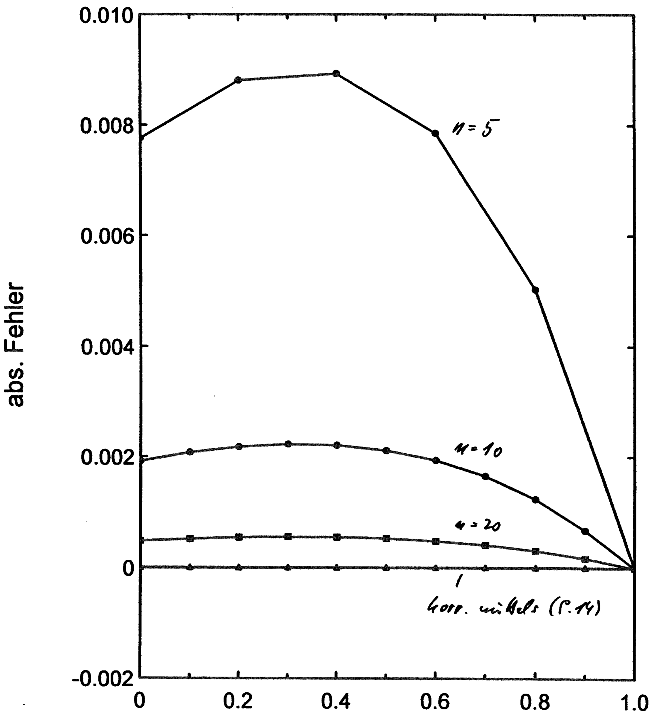


Abbildung 9.2: Absolute Fehler der Näherungslösungen des Testbeispiels für DIFF1.

.

Testergebnisse fuer DIFF1: absolute Fehler zwischen Naeherungswerten und exakten Werten:

x	abs. Fehler n=10	n=20	korr. mittels (9.14)
0.0	1.9162406497E-03	4.7755794913E-04	-2.0029510779E-06
0.1	2.0768600371E-03	5.1777355657E-04	-1.9219369278E-06
0.2	2.1803285636E-03	5.4365274082E-04	-1.9058661564E-06
0.3	2.2261855847E-03	5.5508881815E-04	-1.9434355636E-06
0.4	2.2088322276E-03	5.5069313748E-04	-2.0198913262E-06
0.5	2.1176171067E-03	5.2782016428E-04	-2.1121486498E-06
0.6	1.9371537346E-03	4.8265275109E-04	-2.1809100872E-06
0.7	1.6485163578E-03	4.1051038716E-04	-2.1582709451E-06
0.8	1.2326037795E-03	3.0670399065E-04	-1.9292747311E-06
0.9	6.7823138306E-04	1.6857800802E-04	-1.3064491213E-06
1.0	1.0913936421E-11	1.0913936421E-11	1.0913936421E-11

9.3 Numerische Behandlung des homogenen RWP mittels des Differenzenverfahrens.

Im folgenden soll nun die numerische Auswertung des *homogenen* RWP (9.1)–(9.3) diskutiert werden, und zwar in der Form eines Eigenwertproblems:

$$r(x)\,y''(x) + s(x)\,y'(x) + q(x)\,y(x) = \lambda\,y(x) \qquad (x \in [a,b]) \tag{9.15}$$

mit den Randbedingungen

$$\alpha_0\,y(a) + \alpha_1\,y'(a) = 0 \qquad (|\alpha_0| + |\alpha_1| \neq 0) \tag{9.16}$$

und

$$\beta_0\,y(b) + \beta_1\,y'(b) = 0 \qquad (|\beta_0| + |\beta_1| \neq 0) \quad . \tag{9.17}$$

Für ein solches Problem gibt es immer die triviale Lösung $y(x) \equiv 0$ sowie *für bestimmte Werte von λ nicht-triviale Lösungen*. Man spricht von den Eigenwerten und Eigenfunktionen (Eigenlösungen) des homogenen RWP.

Die mathematische Behandlung von (9.15)–(9.17) ist identisch mit der Vorgangsweise in Abschnitt 9.2. Wieder wird das RWP durch Diskretisierung des Intervalles $[a,b]$ in ein lineares, diesmal natürlich *homogenes* Gleichungssystem umgewandelt. Unter der Voraussetzung $\alpha_1 \neq 0$ und $\beta_1 \neq 0$ erhält man

$$\left[\underbrace{q_1 - \frac{2r_1}{h^2} - \frac{\alpha_0}{\alpha_1}\left(s_1 - \frac{2r_1}{h}\right)}_{b_1} - \lambda\right] y_1 + \underbrace{\frac{2r_1}{h^2}}_{c_1}\, y_2 = 0\,, \tag{9.18}$$

für $(i = 2, \ldots, N)$

$$\underbrace{\left(\frac{r_i}{h^2} - \frac{s_i}{2h}\right)}_{a_i} y_{i-1} + \left(\underbrace{q_i - \frac{2r_i}{h^2}}_{b_i} - \lambda\right) y_i + \underbrace{\left(\frac{r_i}{h^2} + \frac{s_i}{2h}\right)}_{c_i} y_{i+1} = 0 \tag{9.19}$$

und

$$\underbrace{\frac{2r_{N+1}}{h^2}}_{a_{N+1}}\, y_N + \left[\underbrace{q_{N+1} - \frac{2r_{N+1}}{h^2} - \frac{\beta_0}{\beta_1}\left(s_{N+1} + \frac{2r_{N+1}}{h}\right)}_{b_{N+1}} - \lambda\right] y_{N+1} = 0\,. \tag{9.20}$$

Auf diese Weise hat man das Eigenwertproblem des RWP in das Eigenwertproblem einer tridiagonalen Matrix umgewandelt:

$$\begin{pmatrix} b_1 - \lambda & c_1 & & & & \\ a_2 & b_2 - \lambda & c_2 & & & \\ & & \cdot & & & \\ & & \cdot & & & \\ & & \cdot & & & \\ & & a_N & b_N - \lambda & c_N \\ & & & a_{N+1} & b_{N+1} - \lambda \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_N \\ y_{N+1} \end{pmatrix} = 0\,. \tag{9.21}$$

Das Auftreten von $\alpha_1 = 0$ und/oder $\beta_1 = 0$ bedeutet eine harmlose Komplikation: in diesem Fall gehen die Randbedingungen (9.16) und/oder (9.17) in die simple Form

$$y(a) \equiv y_1 = 0 \qquad \text{und/oder} \qquad y(b) \equiv y_{N+1} = 0$$

über, d. h., daß die erste und/oder letzte Komponente des Lösungsvektors bereits feststeht *und daher in (9.21) nicht mehr als Unbekannte aufscheinen muß.*
In einem solchen Fall muß die Koeffizientenmatrix in (9.21) *um die erste und/oder letzte Zeile und Spalte verjüngt werden!* Wie dies im Detail geschieht, können Sie dem Struktogramm Nr. 30 entnehmen.

In jedem Fall hat man nun die Eigenwerte einer tridiagonalen Matrix zu ermitteln. Dies kann z. B. mittels des *Hyman-Verfahrens* in Zusammenarbeit mit einem Nullstellen-Suchprogramm wie INTSCH geschehen. Genaueres zu diesem Thema finden Sie im Abschnitt 7.5.5 dieses Skriptums.

9.3.1 Fehlerkorrektur der Eigenwerte.

Ohne weitere theoretische Ableitungen soll hier eine Formel für die Fehlerkorrektur der so erhaltenen Näherungen der Eigenwerte des RWP angegeben werden. Bezeichnet man mit $\lambda(N)$ bzw. mit $\lambda(2N)$ einen Eigenwert der tridiagonalen Matrix in (9.21), wobei das Intervall $[a, b]$ in N bzw. $2N$ Subintervalle geteilt wurde, so kann der Verfahrensfehler bzgl. dieses Eigenwertes mittels der Formel

$$\lambda_{korr} = \frac{4\lambda(2N) - \lambda(N)}{3} \tag{9.22}$$

signifikant reduziert werden (vgl. Abschnitt 9.2.1).

Es soll hier aber ein ganz wesentlicher Nachteil dieser Korrekturformel nicht verschwiegen werden: man hat keinerlei Information über die Genauigkeit der korrigierten Eigenwerte!

9.3.2 Das Programm DIFF2.

Dieses Programm berechnet einige reelle Eigenwerte des homogenen RWP (9.15)–(9.17) mittels des Differenzenverfahrens sowie des Verfahrens von Hyman (s. Abschnitt 7.5.5).
Die Hauptpunkte des Programmes sind:

1. Zählschleife mit T=1(1)2: Um die Korrekturformel (9.22) anwenden zu können, muß die Eigenwertbestimmung *zweimal* durchgeführt werden, und zwar sowohl mit der gegebenen Subintervallanzahl N0 als auch mit der doppelten Subintervallanzahl.

2. Bestimmung der Funktionswerte der Koeffizientenfunktionen $r(x)$, $s(x)$ und $q(x)$ durch Aufruf der Prozedur **FCT1**.

3. Berechnung der 3 Vektoren der tridiagonalen Matrix in (9.21) *unter Berücksichtigung eines eventuellen Nullwerdens von α_1 oder/und β_1.*

4. Aufruf des Nullstellen-Suchprogrammes **INTSCH** (Abschnitt 5.5). Dieses Programm ruft die HYMAN-Funktion auf, wobei die drei Vektoren der tridiagonalen Matrix [die Felder A(), B() und C()] zusammen mit der aktuellen Ordnung der Matrix an die Funktion HYMAN übergeben werden müssen, gegebenenfalls durch die Definition *globaler Parameter* (Details zum Zusammenwirken von HYMAN und INTSCH siehe Abschnitt 7.5.5).
   INTSCH liefert die 'unkorrigierten' Eigenwerte. Die Anzahl der beim ersten bzw. zweiten Durchlauf gefundenen Eigenwerte ist ANZ(1) bzw. ANZ(2). Abspeicherung der unkorrigierten Eigenwerte auf das Feld EWERTE(...,T).

5. Nach dem zweiten Durchlauf (mit doppelter Subintervallanzahl) werden die entsprechenden Eigenwerte mittels (9.22) verbessert. *Dabei muß sichergestellt sein, daß nur zusammengehörende Eigenwerte in die Formel (9.22) eingesetzt werden. Um dies mit ausreichender Sicherheit zu gewährleisten, wird vor der Fehlerkorrektur abgefragt, ob in beiden Rechendurchläufen dieselbe Zahl von Eigenwerten eruiert wurde.* Ist dies nicht der Fall, Fehleranzeige und Return.

INPUT-Parameter:

ANF,AEND: a und b in (9.15).

ALP0,ALP1,BET0,BET1: Parameter α_0, α_1, β_0 und β_1 in den Randbedingungen (9.16) und (9.17).

Die folgenden 4 Input-Parameter beziehen sich auf die von DIFF2 aufgerufene Prozedur INTSCH:

ANFEIG,ENDEIG: Anfang und Ende des Grobsuchbereiches für die Eigenwerte.

HEIG: Schrittweite für die Eigenwertsuche.

GENEIG: Fehlerschranke für die von INTSCH ermittelten Eigenwerte.

N0: Anzahl der Subintervalle für den ersten Durchlauf von DIFF2.

OUTPUT-Parameter:

ANZEIG: Zahl der ermittelten (korrigierten) Eigenwerte.

EIGW( ): Feld der ermittelten (korrigierten) Eigenwerte.

FEHL1: (logische) Fehlerdiagnostik-Variable:
FEHL1=TRUE: Eigenwert-Korrektur war nicht möglich;
FEHL1=FALSE: Eigenwert-Korrektur ok.

INTERNE FELDER:

EWERTE(,): zweidimensionales Feld mit den unkorrigierten Eigenwerten für N0 bzw. 2 N0 Subintervalle.

Das Programm DIFF2 benötigt die Prozeduren FCT1, INTSCH und HYMAN.

Anmerkung:
Um die Verfahrensfehler von INTSCH möglichst klein zu halten, sollte die Fehlerschranke GENEIG sehr klein gewählt werden,
z.B.: GEN = 0.0000001.
Um dieser Anforderung gerecht zu werden, sollte DIFF2 – wenn möglich – mit *doppelter Genauigkeit* gerechnet werden.

Struktogramm 30 — DIFF2(ANF,AEND,ALP0,ALP1,BET0,BET1,ANFEIG, ENDEIG,HEIG,GENEIG,N0,ANZEIG,EIGW,FEHL1)

```
T=1(1)2
    N:=T*N0
    H:=(AEND-ANF)/N
    HQ:=H*H
    I=2(1)N
        X:=ANF+(I-1)*H
        FCT1(X,R,S,Q)
        A(I):=R/HQ-S/2.0/H
        B(I):=Q-2.0*R/HQ
        C(I):=R/HQ+S/2.0/H
    ALP1=0.0
    Y:  IANF:=2
    N:  IANF:=1
        FCT1(ANF,R,S,Q)
        B(1):=Q-2.0*R/HQ - ALP0/ALP1*(S-2.0*R/H)
        C(1):=2.0*R/HQ
    BET1=0.0
    Y:  IEND:=N
    N:  IEND:=N+1
        FCT1(AEND,R,S,Q)
        A(N+1):=2.0*R/HQ
        B(N+1):=Q-2.0*R/HQ - BET0/BET1*(S+2.0*R/H)
    N:=IEND-IANF+1
    ALP1=0.0
    Y:  I=1(1)N
            A(I):=A(I+1)
            B(I):=B(I+1)
            C(I):=C(I+1)
    N:  ......
    INTSCH(ANFEIG,ENDEIG,HEIG,GENEIG,ANZMAX,EIGW,ANZ(T))
    ANZ(T)=0
    Y:  print: DIFF2 'keine Nullst. in INTSCH'
    N:  I=1(1)ANZ(T)
            EWERTE(I,T):=EIGW(I)
ANZEIG:=ANZ(2)
ANZ(1)=ANZ(2)
Y:  I=1(1)ANZEIG
        EIGW(I):=(4.0*EWERTE(I,2) - EWERTE(I,1))/3.0
    FEHL1:=FALSE
N:  print: DIFF2 'Keine Korrektur'
    I=1(1)ANZEIG
        EIGW(I):=EWERTE(I,2)
    FEHL1:=TRUE
(return)
```

9.3.3 Ein Testbeispiel für DIFF2.

Als Test für die Leistungsfähigkeit des eben besprochenen Algorithmus soll nun das lineare, homogene EWP zweiter Ordnung

$$
\begin{aligned}
-y''(\vartheta) - \frac{\cos\vartheta}{\sin\vartheta}\, y'(\vartheta) &= \lambda\, y(\vartheta) \\
y(\pi/2) = 0 \qquad\qquad & \qquad y'(\pi) = 0
\end{aligned}
\tag{9.23}
$$

mittels DIFF2 ausgewertet werden.

Die Eigenwerte λ dieses Problems gehorchen der Formel

$$\lambda_m = (2m-1)\cdot 2m \qquad m = 1, 2, \ldots \tag{9.24}$$

und die entsprechenden Eigenfunktionen sind die Legendre-Polynome 1. Art mit ungeraden Ordnungen.

Im Bereich $0 \le \lambda \le 100$ gibt es also die 5 Eigenwerte

$$2 \qquad 12 \qquad 30 \qquad 56 \qquad 90\,.$$

Die INPUT-Parameter und die Prozedur FCT1 lauten in diesem Fall:

```
.
anf:=1.57079633    aend:=3.14159265

alp0:=1.0   alp1:=0.0   bet0:=0.0   bet1:=1.0

n0:=50

anfeig:=0.0    endeig:=100.0    heig:=1.0    geneig:=0.0000001



PROCEDURE FCT1(x: double; VAR r,s,q: double);

BEGIN
  r:=-1.0;
  IF(abs(x-PI) < 1.e-15)THEN s:=0.0 ELSE s:=-cos(x)/sin(x);
  q:=0.0;
END;
```

Anmerkung zu FCT1:

Bei der Auswertung der Koeffizientenfunktion $s(x)$ ist darauf zu achten, daß sie für $x = \pi$ eine Singularität hat. In der obigen Routine wird s für diesen Fall einfach Null gesetzt. Dies ist aber nur deshalb möglich, weil der in der Dgl. (9.23) vorkommende Term

$$\frac{\cos\vartheta}{\sin\vartheta}\, y'(\vartheta)$$

für $\vartheta \to \pi$ **nicht** divergiert! Die Folge davon ist nämlich (was hier nicht im Detail bewiesen werden soll), daß der dritte Term des in (9.20) definierten Elementes b_{N+1},

$$-\frac{\beta_0}{\beta_1}\left(s_{N+1} + \frac{2r_{N+1}}{h}\right) ,$$

für $\beta_0 = 0$ verschwindet.

Löst man das vorliegende Problem mit Hilfe der im vorigen Abschnitt präsentierten Prozedur DIFF2, so erhält man das folgende Ergebnis:

```
.
 Zahl der Subintervalle <=    50
 GENEIG = 0.0000001

          N=50        N=100     korr. nach    exakt
                                 (9.22)

    1     1.99901    1.99975    2.00000         2
    2    11.97850   11.99463   12.00001        12
    3    29.88705   29.97181   30.00006        30
    4    55.64148   55.91053   56.00021        56
    5    89.12707   89.78212   90.00047        90
```

9.4 Die 'shooting method'.

Anschließend soll hier noch die in der Praxis häufig verwendete Methode besprochen werden, das homogene RWP (9.15–9.17) durch Überführung in ein *Anfangswertproblem* zu lösen.
Zu diesem Zweck bringt man (9.15) in die Form

$$y''(x) = -\frac{s(x)}{r(x)}\, y'(x) - \frac{[q(x) - \lambda]}{r(x)}\, y(x) \tag{9.25}$$

für $x \in [a, b]$.

Die Anfangsbedingungen:

Man benötigt für die Dgl. *zweiter Ordnung* natürlich <u>zwei</u> Anfangsbedingungen. Diese folgen sofort aus der ersten Randbedingung (für $x = a$), welche die Form

$$\alpha_0\, y(a) + \alpha_1\, y'(a) = 0$$

hat. Im Fall $\alpha_1 \neq 0$ kann man nun für $y(a)$ *jede beliebige reelle Zahl (außer Null)* setzen. Dies ist möglich, weil die Lösungsfunktion eines homogenen Problems *stets nur bis auf eine multiplikative Konstante* fixiert ist. Man kann also z. B. einfach festsetzen:

$$y(a) = 1.0\,. \tag{9.26}$$

Der Anfangswert von $y'(x)$ ergibt sich dann sofort als

$$y'(a) = -\frac{\alpha_0}{\alpha_1} y(a) = -\frac{\alpha_0}{\alpha_1}\,. \tag{9.27}$$

Ist hingegen $\alpha_1 = 0$, so bleibt für $y(x)$ nur mehr die Möglichkeit

$$y(a) = 0.0\,, \tag{9.28}$$

und die Funktion $y'(x)$ kann am Anfangswert jeden beliebigen Wert außer Null annehmen, also z. B.

$$y'(a) = 1.0\,. \tag{9.29}$$

Die Gleichungen (9.25)–(9.29) definieren ein Anfangswertproblem, das z. B. mittels der im Kapitel 8 besprochenen Verfahren gelöst werden kann.

Bestimmung der Eigenwerte:

Die Eigenwerte λ sind noch unbekannt. Man kann nun wie folgt vorgehen: Man löst das Cauchy-Problem (9.25)–(9.29) im Intervall $[a, b]$ für ein beliebiges 'Probe'–λ.

Nun überprüft man, ob die erhaltene Lösungsfunktion $y(x)$ an der Stelle des rechten Randes, also für $x = b$, die zweite Randbedingung (9.17) erfüllt, d.h. ob gilt:

$$\beta_0\, y(b) + \beta_1\, y'(b) = 0\,. \tag{9.30}$$

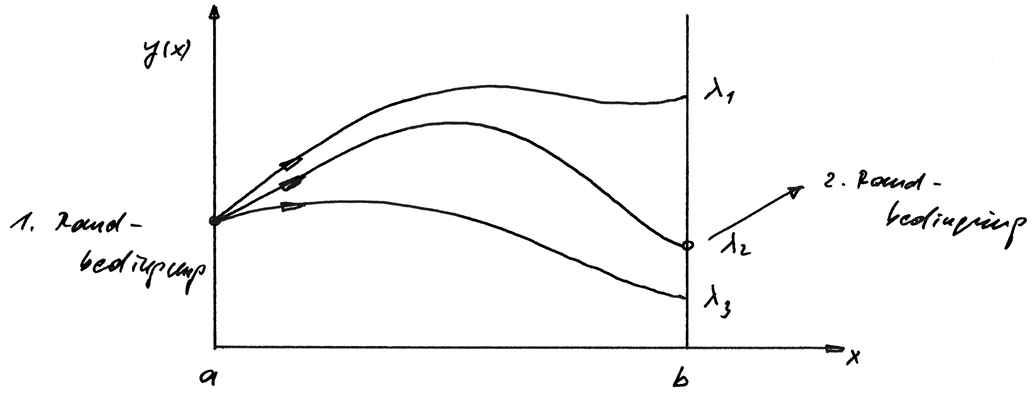


Abbildung 9.3: Das Grundprinzip der *shooting method.*

Wenn dies der Fall ist, ist das gewählte λ einer der Eigenwerte des gegebenen RWP, und $y(x)$ ist (eine Näherung) für die zugehörige Eigenfunktion. Wenn die Bedingung (9.30) nicht erfüllt wird (was natürlich i. a. der Fall sein wird), muß die Auswertung des Anfangswertproblems mit einem neuen Probewert für λ wiederholt werden.
Abb. 9.3 zeigt das Prinzip dieser Methode: λ_1 und λ_3 'treffen daneben', λ_2 'trifft' die Randbedingung bei b und stellt daher einen Eigenwert dar. Die Formulierung 'treffen' bzw. 'daneben treffen' ist bewußt gewählt, um Ihnen den Namen dieser Methode (*shooting method*) klar zu machen.

Die *shooting method* hat jedoch einen gravierenden Nachteil: um das Eigenwertproblem mit genügender Genauigkeit zu lösen, müssen sehr viele 'Probe'–λ verwendet werden d. h. es müssen sehr viele Cauchy-Probleme gelöst werden, was natürlich häufig große Rechenzeiten bedeutet.

Aus diesem Grund wird die *shooting method* in der Praxis vor allem in Kombination mit einer sehr schnellen und dennoch genauen Lösungsmethode für das Cauchy-Problem kombiniert, die auf Numerov zurückgeht.

9.4.1 Die Numerov-Methode.

Diese Methode ist nur anwendbar, wenn es gelingt, die Dgl. des homogenen RWP auf die Form

$$y''(x) + k(x)\,y(x) = 0 \qquad x \in [a,b] \tag{9.31}$$

zu bringen. Dies ist für viele wichtige Anwendungsfälle in Physik und Technik möglich. Ein gutes Beispiel ist etwa die eindimensionale, zeitunabhängige (stationäre) *Schrödingergleichung*, welche bekanntlich die Form

$$\psi''(x) + \frac{2m}{\hbar^2}\,[E - V(x)]\;\psi(x) = 0$$

hat. In diesem Falle wäre die in (9.31) definierte Funktion $k(x)$ also durch

$$k(x) = \frac{2m}{\hbar^2}\,[E - V(x)]$$

bzw., bei Verwendung geeigneter Längen- und Energie-Einheiten, durch

$$k(x) = E - V(x) \tag{9.32}$$

gegeben.

Die Numerov-Methode ist wieder eine Differenzenmethode, d.h. man teilt das gegebene Intervall in N Subintervalle mit den Stützpunkten

$$x_i = a + (i-1)h \qquad (i = 1, \ldots, N+1) \qquad \text{mit} \quad h = \frac{b-a}{N}\,.$$

Der nächste Schritt ist eine Taylorreihenentwicklung der Funktion $y(x)$ an der Stelle x_i für die Werte $x_i \pm h$:

$$y(x_i \pm h) \equiv y_{i\pm 1} = y_i \pm h\, y_i' + \frac{h^2}{2}\, y_i'' \pm \frac{h^3}{6}\, y_i^{(3)} + \frac{h^4}{24}\, y_i^{(4)} \pm \cdots$$

Daraus folgt sofort

$$y_{i+1} + y_{i-1} = 2y_i + h^2\, y_i'' + \frac{h^4}{12}\, y_i^{(4)} + \cdots \tag{9.33}$$

Für y_i'' kann man nun gemäß (9.31) setzen:

$$y_i'' = -k_i\, y_i\,. \tag{9.34}$$

Nun kommt der entscheidende Trick: man diskretisiert die vierte Ableitung von $y(x)$ an der Stelle x_i mittels des finiten Elementes (9.6) und erhält mit (9.34)

$$y_i^{(4)} \approx \frac{y_{i+1}'' - 2y_i'' + y_{i-1}''}{h^2} = \frac{2k_i y_i - k_{i+1}y_{i+1} - k_{i-1}y_{i-1}}{h^2}\,. \tag{9.35}$$

Setzt man nun (9.34) und (9.35) in die Glg. (9.33) ein und löst diese Gleichung nach y_{i+1} auf, ergibt sich die folgende *Rekursionsformel von Numerov*:

$$y_i \approx \frac{2\left(1 - \frac{5h^2}{12}k_{i-1}\right) y_{i-1} - y_{i-2} - \frac{h^2}{12}k_{i-2}y_{i-2}}{1 + \frac{h^2}{12}k_i} \tag{9.36}$$

für $i = 3, \ldots, N$, wobei die Werte k_i die Funktionswerte von $k(x)$ [Glg. (9.32)] an den Stützpunkten x_i bedeuten:

$$k_i \equiv k(x_i) = E - V(x_i) \qquad \text{mit} \quad i = 1, \ldots, N+1\,. \tag{9.37}$$

Nun noch einige wichtige Anmerkungen zum shooting-Numerov-Verfahren:

Als ersten Schritt des Numerov-Prozesses muß die Glg. (9.36) für $i = 3$ ausgewertet werden. Dies führt zur Formel

$$y_3 \approx \frac{2\left(1 - \frac{5h^2}{12}k_2\right) y_2 - y_1 - \frac{h^2}{12}k_1y_1}{1 + \frac{h^2}{12}k_3}\,. \tag{9.38}$$

- Um den Numerov-Prozess starten zu können, muß man also die Werte $y_1 \equiv y(a)$ und $y_2 \equiv y(a+h)$ kennen. Diese Größen lassen sich gewöhnlich aus den Nebenbedingungen des Problems eruieren.

- In der obigen Formel für y_3 kommt die Größe $k_1 = E - V(a)$ vor. Man braucht also den Wert der Potentialfunktion am linken Rand a des Integrationsintervalls. In vielen konkreten Fällen - z.B. wenn $V(x)$ ein *Coulomb*-Potential ist - hat diese Funktion für $x = a$ eine Singularität und kann daher dort nicht ausgewertet werden.
  Dann kann man oft wie folgt vorgehen: man schreibt die Glg. (9.38) in der Form

$$y_3 \approx \frac{2\left(1 - \frac{5h^2}{12}k_2\right)y_2 - y_1 - \frac{h^2}{12}\left[Ey_1 - V(x_1)y_1\right]}{1 + \frac{h^2}{12}k_3}$$

  und berechnet analytisch den Grenzwert

$$\mathrm{V1Y1} = \lim_{x \to a} V(x)y(x)\,.$$

  Beispiele für ein solches Vorgehen finden Sie im Abschnitt 9.4.3 sowie in der Übungsbeschreibung.

- Wie aus den obigen Erläuterungen hervorgeht, muß für eine numerische Berechnung der Eigenwerte der Numerov-Prozess in der Regel *sehr oft* durchlaufen werden, wobei sich die verwendeten Werte von k_i nur durch einen neuen *trial value* für E unterscheiden, während die Potentialwerte $V(x_i)$ unverändert bleiben.

  Es wäre nun ausgesprochen ungeschickt, bei jedem Aufruf des *Numerov*-Programms die Potentialwerte $V(x_i)$ aufs Neue zu berechnen, denn wenn die Potentialfunktion kompliziert ist, würde das eine Menge Rechenzeit kosten. Viel besser ist es, das Feld der Werte $V(x_2)$, $V(x_3)$, ..., $V(x_{N+1}$ vor dem ersten *Numerov*-Aufruf ein für alle Mal zu berechnen und dieses Feld über die Parameterliste an *Numerov* zu übergeben. Eine solche Vorgangsweise ist in der folgenden Beschreibung des *Numerov*-Programms berücksichtigt.

9.4.2 Das Programm NUMEROV.

Die Prozedur NUMEROV führt eine numerische Integration der Dgl. (9.31) durch, wobei diese Gleichung den Eigenwert-Parameter λ enthält.

INPUT-Parameter:

ANF,AEND: a und b in (9.31).

Y1,Y2: Werte für $y(a)$ und $y(a+h)$.

V1Y1: Wert für $V(a)y(a)$ (s. dazu die Erläuterungen im vorigen bzw. im Abschnitt 9.4.3).

EIGW: Probewert für den Eigenwert (*shooting method!*).

POT: Vektorfeld mit den Potentialwerten $POT(2) = V(x_2)$, $POT(3) = V(x_3)$, ..., $POT(N+1) = V(x_{N+1})$.

N: Anzahl der Subintervalle in $[a, b]$.

<u>OUTPUT-Parameter:</u>

YF(): Näherungswerte der Lösungsfunktion $y(x)$ an den Stützpunkten $x_1, x_2, \ldots, x_{N+1}$.

Struktogramm 31 — NUMEROV(ANF,AEND,Y1,Y2,V1Y1,EIGW,POTF,N,YF)

```
H:=(AEND-ANF)/N
HH12:=H*H/12.0

YF(1):=Y1
YF(2):=Y2
YLL:=Y1
YL:=Y2
KL:=EIGW-POTF(2)
K:=EIGW-POTF(3)

Y:=(2.0*(1.0-5.0*HH12*KL)*YL – YLL – HH12*(EIGW*YLL-V1Y1)) / (1.0+HH12*K)

YF(3):=Y

I=4(1)N+1
    YLL:=YL
    YL:=Y
    KLL:=KL
    KL:=K
    K:=EIGW-POTF(I)

    Y:=(2.0*(1.0-5.0*HH12*KL)*YL – YLL – HH12*KLL*YLL) / (1.0+HH12*K)

    YF(I):=Y

(return)
```

9.4.3 Testbeispiel für die shooting-Numerov-Methode.

Im folgenden soll nun diese Methode auf ein Problem der Metallphysik angewendet werden. Gesucht ist die Grundzustandsenergie und die entsprechende Wellenfunktion eines Valenzelektrons in metallischem Natrium in der *Wigner-Seitz-Näherung.*

Die Wigner-Seitz-Näherung besteht darin, daß man die i. a. geometrisch recht komplizierte Form der Einheitszelle des Kristalls durch eine volumsgleiche Kugel mit dem Radius r_{WS} ersetzt. Innerhalb der Kugel wird das Kristallpotential durch ein radialsymmetrisches Potential angenähert:

$$V(\mathbf{r}) \approx V(r) \qquad \text{für } r \in [0, r_{WS}] \, .$$

Unter diesen Umständen sind die Energieeigenwerte E und die Eigenfunktionen eines quantenmechanischen Teilchens (Elektron) durch die *stationäre* Schrödingergleichung

$$y''(r) + \left\{ \frac{2m}{\hbar^2} [E - V(r)] - \frac{\ell(\ell+1)}{r^2} \right\} y(r) = 0$$

gegeben, wobei $y(r)$ der mit r multiplizierte *Radialteil der Wellenfunktion des Elektrons* ist. Der Parameter ℓ stellt die Bahndrehimpuls-Quantenzahl dar; im Falle der Natrium-Valenzelektronen gilt $\ell = 0$.

In atomaren Einheiten (Längen in Bohr, Energien in Rydberg) gilt außerdem $2m/\hbar^2 = 1$, und man erhält

$$y''(r) + k(r)\, y(r) = 0 \qquad \text{mit} \qquad k(r) = E - V(r) \tag{9.39}$$

für $r \in [0, r_{WS}]$.

Der Wigner-Seitz-Radius für das Natrium-Metall beträgt 3.9405 Bohr, und als Potential innerhalb der WS-Zelle verwendeten Wigner und Seitz in ihrer Originalarbeit das im folgenden angegebene Prokofjew-Potential:

$$V(r) = -\frac{2Q(r)}{r^2} \tag{9.40}$$

mit

$$Q(r) = \begin{cases} 11r & 0 \leq r \leq 0.01 \\ -26.4r^2 + 11.53r - 0.00264 & 0.01 \leq r \leq 0.15 \\ -2.84r^2 + 4.46r + 0.5275 & 0.15 \leq r \leq 1.00 \\ +1.508r^2 - 4.236r + 4.876 & 1.00 \leq r \leq 1.55 \\ 0.1196r^2 + 0.2072r + 1.319 & 1.55 \leq r \leq 3.30 \\ 0.0005r^2 + 0.9933r + 0.0222 & 3.30 \leq r \leq 6.74 \\ r & 6.74 \leq r < \infty \end{cases} \tag{9.41}$$

.

```
FUNCTION POT(r: double): double;
VAR
  q : double;
BEGIN
  IF(r <= 0.01)THEN q:=11.0*r
  ELSE IF(r <= 0.15)THEN q:=-26.4*r*r+11.53*r-0.00264
  ELSE IF(r <= 1.00)THEN q:=-2.84*r*r+4.46*r+0.5275
  ELSE IF(r <= 1.55)THEN q:=1.508*r*r-4.236*r+4.876
  ELSE IF(r <= 3.30)THEN q:=0.1196*r*r+0.2072*r+1.319
  ELSE IF(r <= 6.74)THEN q:=0.0005*r*r+0.9933*r+0.0222
  ELSE q:=r;
  pot:=-2.0*q/r/r
END;
```

Die Randbedingungen:

Die Bedingungen, die eine Eigenfunktion $y(r)$ von (9.39) erfüllen muß, sind die folgenden:

- In der Nähe des linken Randpunktes (d. h. für $r << 1$) ist das Kristallpotential $V(r)$ praktisch identisch mit den Coulomb-Potential des Atomkerns mit der Ladungszahl Z (für unser Beispiel Natrium ist $Z = 11$). Das heißt, für den Bereich $r << 1$ kann die Dgl. (9.39) als
$$y''(r) + \left(E + \frac{2Z}{r}\right) y(r) = 0$$
geschrieben werden. Die Lösung dieses Problems hat für kleine r die einfache Form
$$y(r) \propto r\,.$$
Aus diesem Verhalten ergeben sich sofort die Startwerte für die Numerov-Rekursion, nämlich
$$y_1 = 0 \qquad \text{und} \qquad y_2 = h\,.$$

  Nun noch ein wichtiges Detail:
  Die Funktion $V(r)$ in (9.39) divergiert offenbar für $r = 0$ wegen
$$V(r) = -\frac{2Z}{r} \qquad \text{für} \quad r \to 0\,.$$
Das Produkt aus den Funktionen $V(r)$ und $y(r)$ bleibt jedoch auch für $r = 0$ beschränkt und ergibt
$$\lim_{r\to 0} V(r)y(r) = -2Z\,.$$
Dieser Wert ist unter dem Namen V1Y1 an die Prozedur NUMEROV zu übergeben (s. Abschnitt 9.4.2).

- Nun wird die *shooting method* angewendet, d. h. es wird die Variable λ solange variiert, bis die mittels NUMEROV erhaltene Funktion $y(x)$ auch die zweite Randbedingung an der rechten Grenze b (im konkreten Fall also für $r = r_{WS}$) erfüllt.
Die Randbedingung, welcher die Eigenfunktion des Valenzelektrons an der Oberfläche der Wigner-Seitz-Kugel genügen muß, ist die in der Theoretischen Festkörperphysik wohlbekannte *Bloch-Bedingung*
$$y(r_{WS}) - r_{WS}\, y'(r_{WS}) \stackrel{!}{=} 0\,.$$
Überträgt man diese Bedingung in die entsprechende Differenzengleichung, so ergibt sich die Bedingung
$$y_{N+1} - b\frac{(y_{N+2} - y_N)}{2h} \stackrel{!}{=} 0.0\,. \tag{9.42}$$
Die Inputgröße EIGW ist also solange zu variieren, bis (9.42) erfüllt wird.

Anmerkung: wie Sie sehen, braucht man zur Auswertung der Glg. (9.42) noch den Wert y_{N+2} rechts neben dem Ende des eigentlichen Integrationsintervalls. Um diesen Wert zu erhalten, müssen Sie im Programm *Numerov* die Zählschleife von I=4(1)N+1 auf I=4(1)N+2 erweitern, und auch die Felder POTF und YF müssen dementsprechend um 1 Speicherplatz erweitert werden.

Die folgende Tabelle zeigt die 'Grobsuche' für das konkrete Problem:

```
.
TESTRECHNUNG   SHOOTING-NUMEROV  1999

Wigner-Seitz-Radius [Bohr] = 3.9405
Zahl der Subintervalle =  400

  eigw        y(b)        b*y'(b)      diff (soll Null sein!)

-0.7500      0.3844      0.6288     -0.2444
-0.6500      0.3259      0.3886     -0.0627
-0.6400      0.3203      0.3664     -0.0461
-0.6300      0.3147      0.3445     -0.0298
-0.6200      0.3092      0.3229     -0.0138
-0.6100      0.3037      0.3017      0.0020
-0.6000      0.2982      0.2808      0.0175
-0.5900      0.2928      0.2601      0.0327
-0.5800      0.2875      0.2398      0.0476
-0.5700      0.2821      0.2198      0.0623
-0.5600      0.2769      0.2002      0.0767
-0.5500      0.2716      0.1808      0.0908
-0.4500      0.2215      0.0036      0.2179
```

Der gesuchte Energie-Eigenwert E für das Natrium-Valenzelektron liegt also im Bereich zwischen –0.62 und –0.61 Rydberg. Eine genauere Rechnung führt zum Ergebnis

$$E = -0.611277 \quad \text{Rydberg}\,.$$

Die Abb. 9.4 zeigt die entsprechende (unnormierte) Eigenfunktion y(x).

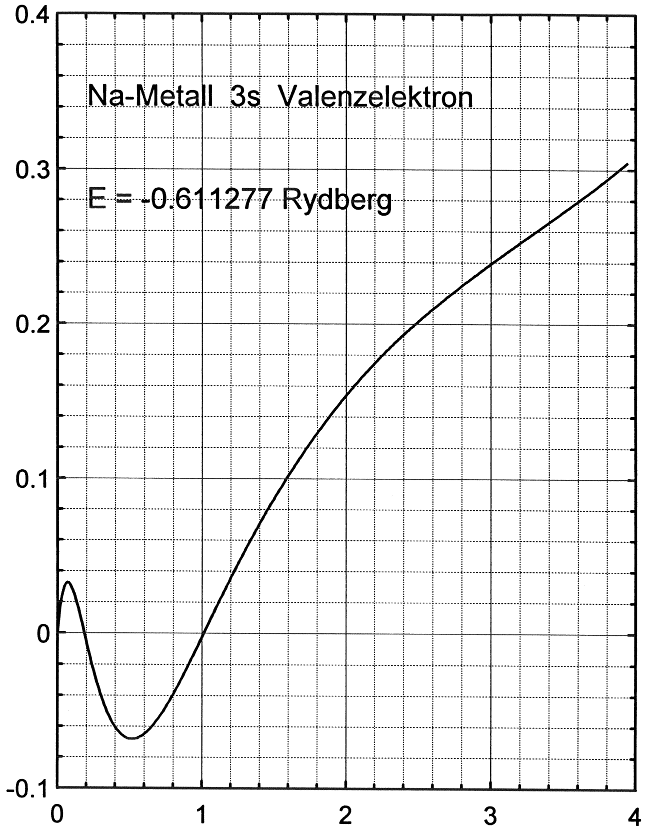


Abbildung 9.4: Ein Ergebnis der *shooting*-Numerov-Methode: die Eigenfunktion eines Valenzelektrons im Natrium-Kristall (400 Subintervalle im Intervall $[0, r_{WS}]$ mit r_{ws}=3.9405 Bohr).
Die Eigenenergie beträgt -0.611277 Rydberg.

9.5 Software-Angebot

In bezug auf Randwertprobleme gewöhnlicher Differentialgleichungen hat die NAG-Bibliothek einiges zu bieten, wie die folgende Tabelle zeigt:

Boundary-value Problems Shooting Method	
simple parameters	**D02HAF**
generalised parameters	**D02HBF, D02AGF**
additional facilities	**D02SAF**
Boundary-value Problems Finite-difference Method	
simple parameters	**D02GAF**
linear problem	**D02GBF**
full nonlinear problem	**D02RAF**
Chebyshev Collocation	
single equation	**D02JAF**
first-order system	**D02JBF**
general system	**D02TGF**
Sturm-Liouville Eigenvalue Problems	
regular problems	**D02KAF**
general problems	**D02KDF**
eigenfunction calculation	**D02KEF**

Sie finden in dieser Tabelle bekannte Begriffe wie *shooting method* und *finite-difference method*, wobei die Differenzenmethode nicht nur auf lineare, sondern auch auch nicht-lineare RWP angewendet werden kann. Die *Kollokationsmethode* nach Chebyshev ist für lineare Probleme ebenfalls sehr leistungsfähig; es handelt sich dabei um eine Entwicklung der Lösungsfunktion nach Chebyshev-Polynomen und um eine numerische Berechnung der Polynom-Koeffizienten.

Obwohl in dieser LV. die Besprechung der numerischen Behandlung von *partiellen Differentialgleichungen* (leider) keinen Platz hat, möchte ich abschließend zumindest einige Hinweise auf das sehr umfangreiche Software-Angebot in bezug auf diese Probleme geben:
Einen sehr guten Überblick über p.d. und kommerizielle Software finden Sie in den schon häufig zitierten Büchern von Ch. Überhuber ([21],[22]). So ist z. B. in [21], S. 331ff eine Fallstudie 'Software für elliptische partielle Differentialgleichungen' enthalten, die einschlägige Programme aus den (p.d.) Bibliotheken ELLPACK und TOMS sowie aus NAG und IMSL vorstellt.

Literaturverzeichnis

[1] H. Ernst, *Numerische Methoden für Kleincomputer*, Luther-Verlag 1984.

[2] G. Engeln-Müllges und F. Reutter, *Formelsammlung zur Numerischen Mathematik mit Standard-FORTRAN-Programmen*, BI Mannheim 1984.

[3] G. Engeln-Müllges und F. Reutter, *Formelsammlung zur Numerischen Mathematik mit C-Programmen*, BI Mannheim 1990.

[4] G. Engeln-Müllges und F. Reutter, *Formelsammlung zur Numerischen Mathematik mit PASCAL-Programmen*, BI Mannheim 1985.

[5] G. Engeln-Müllges und F. Reutter, *Numerik-Algorithmen mit Programmen in FORTRAN, C und Turbo Pascal*, VDI-Verlag, 1996.

[6] G. Engeln-Müllges and F. Uhlig, *Numerical Algorithms with C or FORTRAN*, Springer-Verlag, 1996.

[7] W.S. Dorn and D.D. McCracken, *Numerical Methods with Fortran IV Case Studies*, Wiley 1972.

[8] A. Björck and G. Dahlquist, *Numerische Methoden*, Oldenburg Verlag 1972.

[9] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes*, 2nd ed., Cambridge Uni Press 1992.

[10] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*, 2nd ed., Cambridge Uni Press 1992.

[11] M. Abramowitz and I.A. Segun, *Handbook of Mathematical Functions*, Dover Publications 1968.

[12] G.E. Forsythe und C.B. Moler, *Computer-Verfahren für lineare algebraische Systeme*, Oldenbourg Verlag 1971.

[13] G. Paulin und E. Griepentrog, *Numerische Verfahren der Programmiertechnik*, VEB Verlag Berlin 1975.

[14] P.K. MacKeown and D.J. Newman, *Computational Techniques in Physics*, Hilger 1987.

[15] I.S. Beresin und N.P. Shidkow, *Numerische Methoden 2*, VEB Verlag Berlin 1971.

[16] I.S. Beresin und N.P. Shidkow, *Numerische Methoden 1*, VEB Verlag Berlin 1970.

[17] Y.A. Shreider, *The Monte Carlo Method*, Pergamon Press Oxford 1967.

[18] R. Frühwirt, M. Regler, *Monte-Carlo-Methoden*, BI Mannheim 1983.

[19] B. P. Demidowitsch, L. A. Maron und E. S. Schuwalowa, *Numerische Methoden der Analysis*, VEB Verlag Berlin 1968.

[20] G. N. Poloshi, *Mathematisches Praktikum*, Teubner Verlag Leipzig 1963.

[21] Ch. Überhuber, *Computer-Numerik 1*, Springer-Verlag Berlin 1995.

[22] Ch. Überhuber, *Computer-Numerik 2*, Springer-Verlag Berlin 1995.

[23] P. L. DeVries, *Computerphysik*, Spektrum Akademischer Verlag Heidelberg, 1995.