Computational Physics I *

Ulli Wolff, Burkhard Bunk, Francesco Knechtli

Institut für Physik der HU
Computational Physics

e-mail: uwolff@physik.hu-berlin.de
e-mail: knechtli@physik.hu-berlin.de

14. Februar 2003

Zusammenfassung

Die Skripten CP1 (und auch schon CP2) aus früheren Semestern, die sich von Semester zu Semester allerdings geringfügig ändern (u. a. wegen software upgrades), finden sich im Web:
`http://www-com.physik.hu-berlin.de/comphys/comphys.html`
Bitte nicht unnötig drucken, das jeweils benötigte wird in aktualisierter Fassung zu Beginn der Vorlesung ausgeteilt!
Kap. 14 in dieser Fassung ist neuer Stoff !!

*Kurs im Wahlpflichtfach *Wissenschaftliches Rechnen* WS 2002/2003

Inhaltsverzeichnis

1 Einleitung

In diesem Kurs befassen wir uns zunächst mit elementaren numerischen Methoden und deren Erprobung. Als Beispiel sei hier etwa das numerische Lösen von linearen Gleichungssystemen genannt. Mit einer praktischen Grundausrüstung an solchen mathematischen Kenntnissen soll dann bald Physik getrieben werden, die ein wenig an den Stellen anschließt, wo in einer Vorlesung normalerweise aufgehört werden muß: wenn Gleichungen nicht mehr geschlossen, d. h. durch Ausdrücke in elementaren Funktionen, gelöst werden können. Wir wollen dabei auch noch Fälle einschließen, wo ein geschlossener Ausdruck zwar vielleicht existiert — das ist nicht immer leicht zu entscheiden —, aber so kompliziert und unübersichtlich ist, daß er auch nicht erhellt. Dann sind numerische Methoden angesagt, von denen wir einige in Beispielen anwenden wollen.

Eine Warnung: Numerische Rechnungen liefern letztendlich nur Zahlenkolonnen, die man drucken oder plotten kann. Um damit etwas Vernünftiges anfangen zu können, ist es natürlich essentiell, erst die Prinzipien mit exakt lösbaren Fällen zu verstehen. Weiter sollte man möglichst einfache qualitativ ähnliche Probleme erst studieren, um damit das Problem so aufzubereiten, daß einen die Zahlen wirklich etwas lehren. Beispiel: Was sagt die Newton'sche Gleichung über fliegende Steine und andere Objekte? Klarerweise sollte man erst mit Papier und Bleistift Wurfparabeln herleiten und die funktionalen Abhängigkeiten von Anfangsgeschwindigkeit etc. physikalisch-intuitiv verstehen. Dann könnte man schwache Reibung durch einen phänomenologischen Term einführen und die Abweichung betrachten. Erst dann wird man sich an von komplizierten Reibungstermen dominierte Fälle wagen. In diesem Kurs sollte diese Voraussetzung nach gelegentlicher Auffrischung von Stoff aus den Vorlesungen gegeben sein. In der aktuellen Forschung findet man gelegentlich durchaus Verstöße gegen die obige Warnung. Man geht gleich auf komlizierteste behandelbare Probleme los. Die Interpretation der Resultate ist dann oft wenig aussagekräftig und überzeugend.

Die beschriebenen bescheidenen Übungen, die wir hier planen, sind unter dem Gebiet "Computational Physics" einzuordnen. Wir bleiben hier bei dem englischen Begriff an Stelle mehr oder weniger verkrampfter Übersetzungen. Wir werden auch bei anderen Begriffen ungehemmt so verfahren, allerdings nur, wo es sinnvoll erscheint[1]. Manche Leute sehen heute Computational

[1] Das ist natürlich sehr subjektiv!

Physics als dritte Unterdisziplin, angesiedelt zwischen Experimentalphysik und Theoretischer Physik.

Häufig wird dabei die zeitliche Evolution von komplexen Systemen gemäß vorgegebener "Naturgesetze" Schritt für Schritt konstruiert und verfolgt. Die Simulation stellt ein Abbild des Experiments dar[2]. Dies ist im Gegensatz zur analytischen Lösung zu sehen, wo eine Formel mit einem Schlag das Verhalten für alle Zeiten gibt und sozusagen eine Abkürzung liefert. Das ist aber nur in den wenigen Ausnahmefällen möglich [1], die mit gutem Grund (s. oben) im Vordergrund von Vorlesungen stehen. Im Gegensatz zur Natur können simulierte Gesetze auch modifiziert werden, z. B. durch andere Werte von Naturkonstanten, die die relative Stärke konkurrierender Kräfte regeln. Indem man sieht, was dabei herauskommt, gewinnt man zusätzliche Informationen und Intuition. In einem weiteren Typ von Simulationen ("Monte Carlo") werden endliche, aber (hoffentlich) hinreichend große Ensembles gebildet, die statistisch-thermodynamischen Gesetzen asymptotisch folgen. In der Elementarteilchenphysik werden ein Teil der Konsequenzen von Quantenfeldtheorien wie der Quantenchromodynamik (= Theorie der starken Wechselwirkung) auf diese Weise ausgelotet. Solche Simulationen, die versuchen zu modellieren, was in einem 4-dimensionalen Stück Raum-Zeit-Kontinuum an elementaren Prozessen passiert, erreichen trotz noch immer erheblicher Vereinfachung unweigerlich die Grenzen des auf heutigen Rechnern Machbaren. Man findet solche Rechnungen daher auf den größten Parallelrechnern, die etwa Hunderte von Prozessoren simultan und koordiniert einsetzen.

Neben der Größe der Rechner spielt auch die Wahl der Verfahren und Algorithmen eine erhebliche Rolle. An Beiträgen zu diesem Gebiet wird auch hier am Lehrstuhl für Computational Physics geforscht.

Zurück zum hiesigen Kurs "Computational Physics I", der sich mit kleinen Problemen befaßt. Auch hier stellt sich die Frage nach der Hardware- und Software-Rechenplattform. Wir haben uns für das Arbeiten an UNIX–Rechnern entschieden, sie stehen den Studenten zum freien Arbeiben zur Verfügung (HP-Pool mit HP-UX, PC-Pool mit Linux). Diese Umgebung wird auch in vielen Arbeitsgruppen benutzt und gibt für eine spätere Diplom- oder Doktorarbeit auf den verschiedensten Gebieten eine zukunftssichere Basis ab, insbesondere bei Bedarf an Höchstrechenleistung.

Für Workstations stellt sich die Frage nach der Software bzw. Programmiersprache. Die im numerischen Bereich nach wie vor dominierende Spra-

[2]oder auch nicht, wenn sich das vorgeschlagene Gesetz als falsch erweist!

che ist FORTRAN als f77 und inzwischen auch f90, wobei hier allerdings das wichtige Instrument der grafischen Darstellung fehlt und von außen weniger elegant hinzugefügt werden müßte. Unsere Wahl für den überwiegenden Teil des Kurses fiel schließlich auf MATLAB als ein Paket, das viele höhere Funktionen einschließlich Grafik automatisch zur Verfügung stellt, leicht erlernbar und extrem kompakt ist, und trotzdem relativ schnell ist im Gegensatz zu Paketen wie MATHEMATICA und MAPLE. Letztere erlauben zwar auch die interessante Behandlung von Formeln, sind jedoch numerisch sehr langsam. Darüber hinaus sehen wir im Erlernen von MATLAB auch eine sinnvolle Ausbildung, da dieses Paket an vielen industriellen Arbeitsplätzen von Physikern verwendet wird. Einige Übungsaufgaben in Computational Physics II werden auch in der Sprache C behandelt werden, die dort dan (in rudimentärer Form) eingeführt werden wird.

UNIX ist vielen Studenten der höheren Semester schon vertraut, wer noch Probleme hat, wird in den Übungen eine Einführung erhalten. Kenntnisse in MATLAB werden nicht vorausgesetzt, sondern während des Kurses erworben. Im folgenden Abschnitt wird ein Minimum an Kommandos vorgestellt, so daß wir mit dem eigentlichen Rechnen beginnen können. In den Abschnitten zu einzelnen Problemen, die dann folgen, wird nach und nach der Wortschatz (eigentlich Kommandoschatz) erweitert. Dies wird so geschehen, daß hauptsächlich die Namen der relevanten Kommandos klar werden. Zu diesen kann man sich dann interaktiv am Terminal (und wenn nötig in Büchern) informieren. Mehr technische Einzelheiten zu den zur Verfügung gestellten Arbeitsplätzen werden in den Übungen vermittelt.

Im folgenden Semester wird es eine Fortsetzung "Computational Physics II" geben. Dort sollen dann, auf dem in diesem Semester Erlernten aufbauend, etwas größere Simulationsprojekte bearbeitet werden.

2 MATLAB zum Anfangen

Nun folgen erste Gehversuche in MATLAB [3]. Dabei handelt es sich um ein Programmsystem für Matrixrechnungen und Visualisierung. Im Gegensatz zu Programmiersprachen wie FORTRAN oder C kann MATLAB nicht nur vorbereitete Programme ausführen, sondern auch interaktiv genutzt werden. Es ist äußerst kompakt und effektiv. Variable können zwar, müssen aber i. a. nicht deklariert werden. Z. B. löst $C = A * B$ automatisch die richtige Matrixmultiplikation aus, wenn nur B soviele Zeilen wie A Spalten hat (sonst Fehlermeldung), und C wird passend neu erzeugt. Ein sqrt(-1) führt automatisch in die komplexen Zahlen, etc. Plots können mit einem einfachen Kommando erzeugt werden. MATLAB ist über mehr als 10 Jahre entstanden unter Mitwirkung der Leute, die auch an den bekannten Bibliotheken LINPACK und EISPACK für lineare Algebra beteiligt waren.

Am einfachsten startet man MATLAB mit dem Befehl **matlab** am Unix–Prompt. Je nach Rechner verwandelt sich das Terminalfenster in ein Matlab–Kommandofenster mit dem Prompt >> oder es kommt eine eigene grafische Oberfläche, in die wiederum das Kommandofenster integriert ist. Auf jeden Fall erreicht man schnell ein umfangreiches Online–Hilfesystem: im Terminalfenster sagt man `>> helpdesk` und öffnet damit die Hilfe im HTML-Browser, in der grafischen Oberfläche kann man das Hilfesystem direkt anklicken. Unter "Getting started" findet sich dann eine Einführung in MATLAB und unter "Using MATLAB" eine vollständige Beschreibung aller Funktionen. Gedruckte Handbücher sind unter diesen Umständen entbehrlich.

2.1 Variable in MATLAB

Alle numerischen Größen faßt MATLAB als reelle oder auch komplexe Matrizen auf. Zahlen entsprechen dem Spezialfall der 1×1 –Matrix, Zeilen- und Spaltenvektoren der Länge n den $1 \times n$ und $n \times 1$ –Matrizen. Selbst ganze Zahlen werden so behandelt, es gibt also keinen Typ integer. Aus den folgenden Beispielen wird klar, wie diese einzugeben sind:

```
>> a=4.7

a =
```

[3] Im Text schreiben wir MATLAB groß, der UNIX–Befehl zum Start ist aber **matlab**.

```
    4.7000

>> b=[1 2 3]

b =

     1     2     3

>> c=[4; 5; 6]

c =

     4
     5
     6

>> b*c

ans =

    32

>> c*b

ans =

     4     8    12
     5    10    15
     6    12    18

>> d=[1 2 3; 4 5 6]

d =

     1     2     3
     4     5     6

>> e=sqrt(-1)
```

```
e =

        0 + 1.0000i

>> e*e

ans =

    -1

>>
```

Hinter dem prompt ist jeweils das Eingegebene und darunter eine Antwort, die erscheint. Jede Eingabe wird bestätigt. Dies kann man auch unterdrücken, wenn man die Eingabe mit einem Semikolon ; abschließt. Wenn man einen schon definierten Namen oder algebraischen Ausdruck tippt, kommt der Wert als Antwort. Bei einem Ausdruck ist das jeweils letzte Ergebnis auf der Variablen `ans` (answer) gespeichert und kann unter diesem Namen weiterverwendet werden. Man kann natürlich auch gleich auf eine neue Größe zuweisen, z. B. `xx=e*e`. Um nachzusehen, welche Variablen definiert sind, dienen **who** und (detaillierter) **whos**:

```
>> who

Your variables are:

a         b         d
ans       c         e

>>> whos
Name      Size          Bytes  Class
a         1x1               8  double array
ans       1x1               8  double array
b         1x3              24  double array
c         3x1              24  double array
d         2x3              48  double array
e         1x1              16  double array (complex)
```

```
Grand total is 15 elements using 128 bytes

>>
```

Bei dieser Gelegenheit kann man sich davon überzeugen, daß MATLAB Groß– und Kleinschreibung *unterscheidet*: die Variablen `ab, AB, Ab` sind voneinander verschieden.

2.2 Help-Funktionen

Zusätzlich zu der oben genannten Online–Hilfe im Hypertextformat gibt es noch einen (älteren) Zugang zu Funktionsbeschreibungen direkt vom MATLAB –Prompt aus. Das soll, weil es bei einfachen Fragen der schnellste Weg ist, hier näher dargestellt werden.

Im Gegensatz zu UNIX (**man**) bekommt man in MATLAB Auskunft mit **help**, und zwar in einer für die Schnittstelle Mensch deutlich geeigneteren Form. Zum Kommando selbst kann man mit **help help** etwas erfahren (gelegentlich bitte am Terminal ausprobieren). **help** alleine gibt einem eine übersicht über Help-Themen:

```
>> help

HELP topics:

matlab/general       -  General purpose commands.
matlab/ops           -  Operators and special characters.
matlab/lang          -  Programming language constructs.
matlab/elmat         -  Elementary matrices and matrix manipulation.
matlab/elfun         -  Elementary math functions.
matlab/specfun       -  Specialized math functions.
matlab/matfun        -  Matrix functions - numerical linear algebra.
matlab/datafun       -  Data analysis and Fourier transforms.
matlab/polyfun       -  Interpolation and polynomials.
matlab/funfun        -  Function functions and ODE solvers.
matlab/sparfun       -  Sparse matrices.
matlab/graph2d       -  Two dimensional graphs.
matlab/graph3d       -  Three dimensional graphs.
matlab/specgraph     -  Specialized graphs.
```

```
matlab/graphics      -  Handle Graphics.
matlab/uitools       -  Graphical user interface tools.
matlab/strfun        -  Character strings.
matlab/iofun         -  File input/output.
matlab/timefun       -  Time and dates.
matlab/datatypes     -  Data types and structures.
matlab/demos         -  Examples and demonstrations.
simulink/simulink    -  Simulink
simulink/blocks      -  Simulink block library.
simulink/simdemos    -  Simulink demonstrations and samples.
simulink/dee         -  Differential Equation Editor
toolbox/local        -  Preferences.
nag/nag              -  NAG Foundation Toolbox - Numerical & Statistical Library
nag/examples         -  NAG Foundation Toolbox - Numerical & Statistical Library
toolbox/optim        -  Optimization Toolbox.
toolbox/splines      -  Splines Toolbox.
stateflow/stateflow  -  Stateflow
stateflow/sfdemos    -  (No table of contents file)
toolbox/tour         -  MATLAB Tour

For more help on directory/topic, type "help topic".


>>
```

Mit diesen Themen (bis auf nicht installierte toolboxes) kann man weiter einsteigen, z. B.

```
>> help general


General purpose commands.
MATLAB Toolbox   Version 5.3.1 (R11.1)  08-Sep-1999

General information
   help        - On-line help, display text at command line.
   helpwin     - On-line help, separate window for navigation.
   helpdesk    - Comprehensive hypertext documentation and troubleshooting.
   demo        - Run demonstrations.
```

```
   ver         - MATLAB, SIMULINK, and toolbox version information.
   whatsnew    - Display Readme files.
   Readme      - What's new in MATLAB

Managing the workspace.
   who         - List current variables.
   whos        - List current variables, long form.
   clear       - Clear variables and functions from memory.
   pack        - Consolidate workspace memory.
   load        - Load workspace variables from disk.
   save        - Save workspace variables to disk.
   quit        - Quit MATLAB session.

Managing commands and functions.
   what        - List MATLAB-specific files in directory.
   type        - List M-file.
   edit        - Edit M-file.
   lookfor     - Search all M-files for keyword.
   which       - Locate functions and files.
   pcode       - Create pre-parsed pseudo-code file (P-file).
   inmem       - List functions in memory.
   mex         - Compile MEX-function.

Managing the search path
   path        - Get/set search path.
   addpath     - Add directory to search path.
   rmpath      - Remove directory from search path.
   editpath    - Modify search path.

Controlling the command window.
   echo        - Echo commands in M-files.
   more        - Control paged output in command window.
   diary       - Save text of MATLAB session.
   format      - Set output format.

Operating system commands
   cd          - Change current working directory.
   pwd         - Show (print) current working directory.
```

```
   dir          - List directory.
   delete       - Delete file.
   getenv       - Get environment variable.
   !            - Execute operating system command (see PUNCT).
   dos          - Execute DOS command and return result.
   unix         - Execute UNIX command and return result.
   vms          - Execute VMS DCL command and return result.
   web          - Open Web browser on site or files.
   computer     - Computer type.

Debugging M-files.
   debug        - List debugging commands.
   dbstop       - Set breakpoint.
   dbclear      - Remove breakpoint.
   dbcont       - Continue execution.
   dbdown       - Change local workspace context.
   dbstack      - Display function call stack.
   dbstatus     - List all breakpoints.
   dbstep       - Execute one or more lines.
   dbtype       - List M-file with line numbers.
   dbup         - Change local workspace context.
   dbquit       - Quit debug mode.
   dbmex        - Debug MEX-files (UNIX only).

Profiling M-files.
   profile      - Profile M-file execution time.

See also PUNCT.
>>
```

help funktioniert nur mit einer relativ engen Kategorie von Begriffen, i. a. Namen von Kommandos, Funktionen und Programmen. Darüber hinaus ist **lookfor** nützlich, wie das folgende Beispiel zeigt:

```
 help root

root.m not found.

>> lookfor root
```

```
DOCROOT A utility to determine the root directory of MATLAB Help
SQRT   Square root.
SQRTM  Matrix square root.
POLY Convert roots to polynomial.
ROOTS  Find polynomial roots.
MATLABROOT Root directory of MATLAB installation.
CPLXROOT Riemann surface for the n-th root.
BDROOT Return the name of the top-level Simulink system.
FILTFUN Returns frequency response and roots for DFILDEMO.
FILTFUN2 Return frequency response norm and roots for DFILDEMO.
walk.m: %RESULT = WALK( ROOT, STATE, FROM, METHOD )
>>
```

Mit **lookfor** kann man also nach mehr Stichworten suchen und bekommt Begriffe genannt, zu denen dann **help** etwas weiß. Bei **help** oder **lookfor** (sowie bei anderem Output) gibt es manchmal das Problem, daß zuviel Text durchrauscht, bevor man ihn gesehen hat. Da hilft es, **more on** zu tippen. Nun bleibt der Text nach einem Schirm voll jeweils stehen, und erst nachdem man die breite Leerzeichentaste (spacebar) gedrückt hat, geht es weiter. Falls man den Rest nicht mehr sehen will, bricht ein **q** die Sache ab. **more off** schaltet diese Funktion wieder ab (es rauscht dann wieder).

2.3 Sitzung unterbrechen und/oder aufzeichnen

Es kann verschieden(e) dringende Gründe geben, eine interaktive Rechnung mit MATLAB zu unterbrechen. Falls man viele Variablen definiert hat, entsteht u. U. der Wunsch, irgendwann da weiterzumachen, wo man jetzt ist, aber zwischenzeitlich auszuloggen und das Terminal freizugeben. Dies geschieht, indem man **save filename** eingibt. Nun entsteht ein binärer (für uns unlesbarer) file mit dem Namen `filename.mat` in dem directory, wo man MATLAB aufgerufen hat. Mit **quit** kann man nun MATLAB schließen. Nach dem erneuten Start von MATLAB sind nach **load filename** alle Variablen vom letzten Mal wieder da.

Eine andere Funktion hat **diary**. Das Kommando **diary filename** bewirkt, daß von allem, was im folgenden auf dem Schirm erscheint, also Eingaben und Antworten, eine Kopie (Mitschnitt) im file `filename` festgehalten wird. Damit kann man selbst nachsehen und anderen zeigen (und als Übungen abgeben), was man gemacht hat. Dieses Protokoll kann dann mit **diary**

on und **diary off** ein– und ausgeschaltet werden, z.B. bei längerem Output oder wenn man erstmal ins Unreine probieren will. Nach Wiedereinschalten (oder wann immer `filename` schon existiert) wird hinten angehängt.

2.4 Operationen in MATLAB

Wir wollen uns nun mit Verknüpfungen zwischen den in Abschnitt 2.1 eingeführten Variablen befassen. Informationen gibt es mit

```
>> help arith

  +   Plus.
      X + Y adds matrices X and Y.  X and Y must have the same
      dimensions unless one is a scalar (a 1-by-1 matrix).
      A scalar can be added to anything.

  -   Minus.
      X - Y subtracts matrix X from Y.  X and Y must have the same
      dimensions unless one is a scalar.  A scalar can be subtracted
      from anything.

  *   Matrix multiplication.
      X*Y is the matrix product of X and Y.  Any scalar (a 1-by-1 matrix)
      may multiply anything.  Otherwise, the number of columns of X must
      equal the number of rows of Y.

  .*  Array multiplication
      X.*Y denotes element-by-element multiplication.  X and Y
      must have the same dimensions unless one is a scalar.
      A scalar can be multiplied into anything.

  ^   Matrix power.
      Z = X^y is X to the y power if y is a scalar and X is square. If y is an
      integer greater than one, the power is computed by repeated
      multiplication. For other values of y the calculation
      involves eigenvalues and eigenvectors.
      Z = x^Y is x to the Y power, if Y is a square matrix and x is a scalar,
      computed using eigenvalues and eigenvectors.
```

```
      Z = X^Y, where both X and Y are matrices, is an error.

  .^  Array power.
      Z = X.^Y denotes element-by-element powers.  X and Y
      must have the same dimensions unless one is a scalar.
      A scalar can operate into anything.

>>
```

Neben den normalen Operationen (für Matrizen!) + - * (auch /) finden wir die Potenzierung mit erweiterten Funktionen. Man sieht hier die Bestrebung in MATLAB alles, was mathematisch Sinn hat und eindeutig ist, syntaktisch auch zu erlauben. Weiter gibt es die gepunkteten (dotted) Operationen .* .^ ./ die Matrizen *elementweise* verknüpfen. Wieder einige Beispiele:

```
>> x=[2 4 6]

x =

     2     4     6

>> y=x/2

y =

     1     2     3

>> x+y

ans =

     3     6     9

>> x*y
??? Error using ==> *
Inner matrix dimensions must agree.

>> x.*y
```

```
ans =

     2     8    18

>> x.^y

ans =

     2    16   216

>>
```

Die Operation ' nimmt das hermitesch konjugierte einer Matrix (speziell auch komplexe Konjugation von Zahlen), während .' nur transponiert.

Es ist dringend zu empfehlen, mit Hilfe von **help** ausgiebig in dieser Art am Terminal zu experimentieren.

2.5 Vordefinierte Funktionen in MATLAB

In MATLAB gibt es zahlreiche eingebaute Funktionen. Mit **help elfun** (elementary functions) werden die wichtigsten aufgelistet. I. a. haben sie die üblichen Namen sin, cos, exp, log (zur Basis e, sonst log10) usw (klein geschrieben!). Soweit sinnvoll, können sie auf Matrizen angewandt werden und wirken elementweise:

```
>> d

d =

     1     2     3
     4     5     6

>> exp(d)

ans =

    2.7183    7.3891   20.0855
   54.5982  148.4132  403.4288
```

```
>> log(ans)

ans =

     1     2     3
     4     5     6

>>
```

2.6 Eigene Programme und Funktionen

Bisher haben wir MATLAB wie einen komfortablen Taschenrechner benutzt. Wenn man aber lange und raffiniertere Sequenzen von Befehlen tippt und diese, oder Teile davon, öfter benutzt bzw. modifiziert, entwickelt und testet, dann ist es besser, ein Programm (script) zu verfassen. Man schreibt die Befehle einfach in einen file, z.B. `prog.m` . Die extension `.m` ist hier bindend. `prog.m` im aktuellen directory, wo auch MATLAB läuft, enthalte z.B.

```
% file prog.m
% Dies soll ein Programm zeigen
clear
x = [0 pi/2 pi 3*pi/2 2*pi]
y = sin(x)
M = [ x ; y ]
% Ende prog.m
```

In MATLAB rufen wir nun **prog** (ohne .m) auf

```
>> prog

x =

         0    1.5708    3.1416    4.7124    6.2832


y =

         0    1.0000    0.0000   -1.0000    0.0000
```

```
M =

         0    1.5708    3.1416    4.7124    6.2832
         0    1.0000    0.0000   -1.0000    0.0000

>> who

Your variables are:

M         x         y

>> help prog

  file prog.m
  Dies soll ein Programm zeigen

>>
```

Das Resultat ist wie beim Eintippen der Zeilen. Der Befehl **clear** im Programm löscht alle vorhandenen Variablen (Vorsicht!), die Verhältnisse sind dann wie nach dem Start von MATLAB . Weiter sehen wir, daß die Variable `pi` ohne eigene Zuweisung schon vorbesetzt ist. Mit **who** sehen wir die in `prog.m` definierten Größen nach dem Ablauf im Speicher vorhanden. Selbst **help** kennt automatisch unser `prog`. Die mit % (im Gegensatz zu # in UNIX) beginnenden Kommentarzeilen in MATLAB *bis zum ersten Befehl* in `prog.m` erscheinen unter **help**. Die späteren nicht mehr, sie dienen nur der Klarheit. Oft schreibt man im file selbst **help prog** als ersten Befehl, dann erscheinen diese Informationen bei jedem Aufruf von **prog**.

Ähnlich wie Programme definiert man Funktionen, mit dem Unterschied, daß Argumente und Resultatwerte übergeben werden können. Es existiere der folgende file mit Namen `mitteldiff.m`

```
function [m,d] = mitteldiff(u,v)
% Fuer Argumentmatrizen u und v gleicher Groesse
% werden Mittelwert und Differenz gebildet durch
% Aufrufen von [m,d] = mitteldiff(u,v)
m = (u + v)/2;
```

```
d =  u - v;
```

Auf die von **prog** erzeugten Vektoren `x, y` wenden wir **mitteldiff** an:

```
>> help mitteldiff

  Fuer Argumentmatrizen u und v gleicher Groesse
  werden Mittelwert und Differenz gebildet durch
  Aufrufen von [m,d] = mitteldiff(u,v)

>> mitteldiff(x,y)

ans =

         0    1.2854    1.5708    1.8562    3.1416

>> [a,b] = mitteldiff(x,y)

a =

         0    1.2854    1.5708    1.8562    3.1416


b =

         0    0.5708    3.1416    5.7124    6.2832

>> m
??? Undefined function or variable m.

>> d
??? Undefined function or variable d.
```

Es gibt für jede Funktion einen eigenen file mit dem Namen der Funktion als Filename wie oben (mit .m). `u, v` sind die Argumente, an deren Stelle beim Aufruf andere definierte Größen treten, hier `x, y`. Zurückgegeben werden hier zwei Matrizen der gleichen Größe wie die Argumente. Wir sehen, daß, wenn man nicht mit einer Resultatliste (hier `[a,b]`) aufruft, nur der erste

Wert zurückkommt. `m, d` sind *interne* Variable der Funktion und nach deren Berechnung nicht mehr vorhanden.

Ein Unterprogramm (Funktion) kommuniziert mit dem rufenden Programm also über die Argumente und Resultatwerte. Man kann aber auch globale Größen definieren. Dazu ist eine Zeile **global x y z ....** in allen Programmen und Funktionen nötig, die auf diese Variablen gemeinsam zugreifen wollen. Dabei sollte die Anweisung **global** jeweils vor Anweisungen stehen, die die betreffenden Variablen verwenden. Dies ist also insbesondere eine Möglichkeit, Parameter an Funktionen anders als durch Argumente zu übergeben. Damit sollte man jedoch sparsam umgehen, da die Analyse, was genau passiert, erschwert wird.

Funktionen werden beim ersten Aufruf kompiliert (in Maschinensprache übersetzt) und bleiben im Speicher, so daß sie schnell ausgeführt werden. Mit **clear mitteldiff** (für unser Beispiel) kann man die kompilierte Fassung aber explizit aus dem Speicher löschen (ebenso wie einzelne Variable).

2.7 Einfache Plots

Eine der Stärken von MATLAB besteht in der Möglichkeit, schnell Plots zu erstellen. Hier ein Beispiel:

```
>> clear
>> x=[0 : 0.1 : 5];
>> y=sin(x);
>> plot(x,y)
>> whos
  Name      Size          Bytes  Class

  x         1x51            408  double array
  y         1x51            408  double array

Grand total is 102 elements using 816 bytes
```

Nach der Zeile mit **plot** öffnet sich (wenn alles funktioniert) ein neues Fenster mit dem Bild eines Stückes der Sinus-Kurve, s. Abb. 1. Ein Novum ist hier noch die Erzeugung von `x`, einem Vektor der Länge 51 ist mit Werten von 0 bis 5 mit Abständen 0.1. `y` sind natürlich die zugehörigen Sinus–Werte. Im Beispiel werden die Punkte im Bild mit Linien verbunden (Standard). Mit

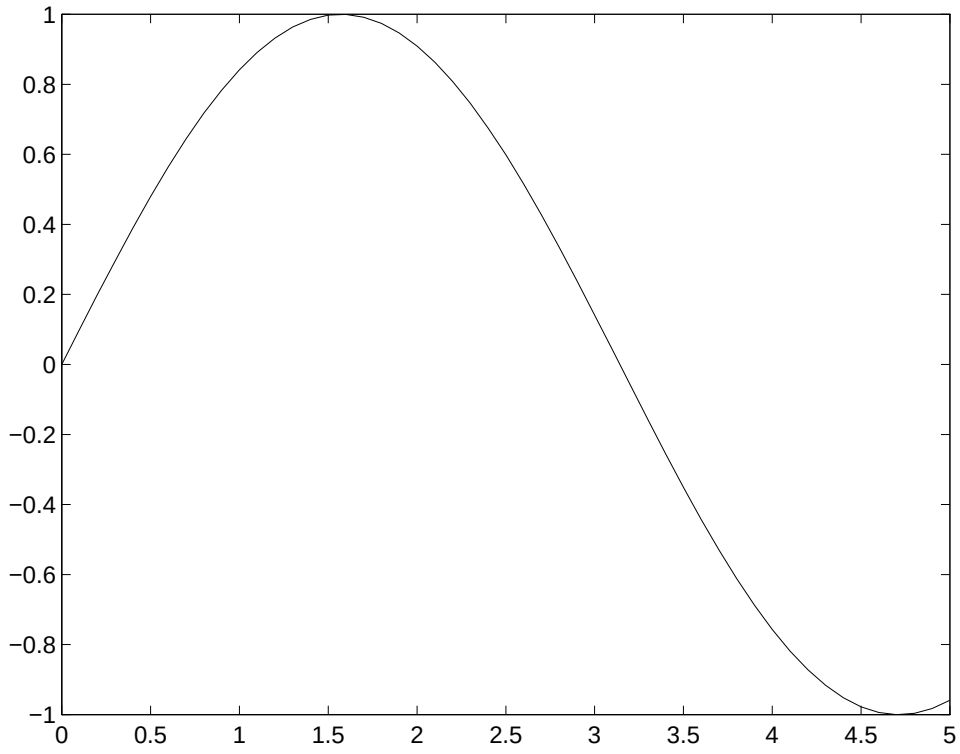

Abbildung 1: plot(x,y)

plot(x,y,'*') bekommt man die einzelnen Punkte mit dem Symbol * gezeigt. Mit **help plot** kann man viele weitere Möglichkeiten erkunden. Häufig verwendet man u. a. die Kommandos **title, xlabel, ylabel**. Bitte ausprobieren. Mit dem Kommando **axis** kann man die Skalen auf der Abszisse und Ordinate festlegen.

Es ist auch möglich, mehrere Graphikfenster auf dem Bildschirm zu erzeugen. Jedes Graphikfenster besitzt eine Nummer, die im oberen Rand des Fensters angezeigt wird. Mit dem Kommando **figure(2)** kann man ein zweites Graphikfenster öffnen, das nun das aktuelle Fenster ist. Wenn man erneut einen **plot**-Befehl eingibt, wird das entstehende Bild in dieses neue Fenster mit der Nummer 2 gezeichnet. Das Fenster mit der Nummer 1 kann mit dem Befehl **figure(1)** wieder reaktualisiert werden. Nach der Eingabe eines **plot**-Befehls erscheinen die Graphiken dann wieder im Fenster 1. Wenn man mit einer Sequenz von **plot**-Befehlen mehrere Kurven hintereinander in dasselbe

Bild zeichnen möchte, gibt man den Befehl **hold on** ein. Dieser Befehl verhindert, daß eine bereits existierende Graphik durch einen neuen **plot**-Befehl gelöscht wird. Der **hold on**-Befehl kann durch **hold off** wieder aufgehoben werden.

Das Bild, das man gerade im Plotfenster hat, kann man in einen file abspeichern. Dazu dient das Kommando **print filename**, das im aktuellen directory `filename.ps` entstehen läßt, wobei die Endung `.ps` für postscript steht, eine Grafik–Beschreibungssprache. Einen solchen file kann man auf einem postscript-fähigen Drucker — ein solcher sollte im Kurs zur Verfügung stehen — als Bild ausdrucken. Der Befehl **print** allein schickt das aktuelle Bild direkt auf den Standard–Drucker. Für den Fall, daß dieser Drucker nicht im Übungsraum steht, sollte der Plot den Namen des Besitzers in irgendeiner Form tragen. Mit dem Kommando **title('Bild von N.N.')** erhält der Plot eine entsprechende Überschrift.

2.8 Ein- und Ausgabe

Die einfachste Ausgabe kennen wir schon: einfach den Variablennamen tippen (Vorsicht bei sehr großen Matrizen!). Das Kommando **format** erlaubt es, die Darstellung zu beeinflussen:

```
>> a

a =

    4.7000

>> format long
>> a

a =

   4.70000000000000

>> format long e
>> a

a =
```

```
     4.700000000000000e+00

>> disp(' Irgendein Text')
 Irgendein Text
>> disp(a)
     4.700000000000000e+00

>>
```

Mit **disp** (display) kann man Variablen anzeigen ohne das `a =` , z. B. in einem Programm. ' Irgendein Text' ist eine Zeichenkette (string), die dann ebenfalls ausgegeben wird. Mehr Gestaltungsmöglichkeiten hat man beim formatierten Schreiben mit **fprintf**

```
>> fprintf(' variable a ist %4.2f, oder? \n',a)
 variable a ist 4.70, oder?
>> fprintf(' variable a ist %8.6f, oder? \n',a)
 variable a ist 4.700000, oder?
>> fprintf(' variable a ist %12.6e, oder? \n',a)
 variable a ist 4.700000e+00, oder?
>>
```

Das erste Argument ist ein String, der das Format spezifiziert. Er kann Text enthalten und zu jeder Variablen (hier `a`) eine Spezifizierung. Hier bedeutet `%8.6f` eine mindestens 8 Zeichen breite Dezimalzahl mit 6 Stellen hinter dem Komma, das Format `e` gibt eine normierte Darstellung mit separatem Exponenten. Das \n erzeugt den Sprung in die nächste Zeile. Mit **fprintf** kann man auch in files schreiben (später).

Eine elegante Art, ein Programm nach Eingabeparametern fragen zu lassen, geht wie folgt:

```
>> z=input(' eingabe fuer z, bitte ');

 eingabe fuer z, bitte 3
>> disp(z)
     3

>>
```

Bei **input** erscheint also der Argumenttext, so daß man weiß, was gefragt ist; dann pausiert das Programm, bis die Eingabe erfolgt ist. Die erste 3 ist also die eingetippte Antwort.

2.9 Noch einige Kommandos

Bei der Erzeugung von `x` im Plot–Beispiel haben wir implizit schon eine Schleife (for-loop, do-loop) kennengelernt. Eine explizite Möglichkeit, genau das Gleiche zu bekommen, ist

```
x(1)=0;
for i=2:51,
  x(i)=x(i-1)+0.1;
end
```

Programme und Funktionen werden meist erst schlau, wenn man den Programmfluß von Bedingungen abhängig machen kann. Dazu dient u. a. das **if** statement:

```
>> help if

 IF IF statement condition.
    The general form of the IF statement is

       IF expression
         statements
       ELSEIF expression
         statements
       ELSE
         statements
       END

    The statements are executed if the real part of the expression
    has all non-zero elements. The ELSE and ELSEIF parts are optional.
    Zero or more ELSEIF parts can be used as well as nested IF's.
    The expression is usually of the form expr rop expr where
    rop is ==, <, >, <=, >=, or ~=.

    Example
```

```
      if I == J
  A(I,J) = 2;
      elseif abs(I-J) == 1
        A(I,J) = -1;
      else
        A(I,J) = 0;
      end

  See also RELOP, ELSE, ELSEIF, END, FOR, WHILE, SWITCH.
>>
```

Im Beispiel des kurzen Help-Textes[4] wird auf Gleichheit zweier Ausdrücke (expression, expr) getestet (==), und die anderen Vergleichsoperatoren (relational operator, rop) werden angegegeben.

Schließlich erwähnen wir hier das Kommando **pause**. Ein Programm stoppt an dieser Stelle, bis irgendeine Taste gedrückt wird. Dies ist z.B. nützlich, wenn sich ein Bild während der Ausführung ändert, damit man es erst betrachten kann, bevor z. B. weitere Kurven hinzukommen.

[4] Dummerweise erscheinen hier die Schlüsselworte **if, elseif** usw in Großbuchstaben (veraltet?), sie *müssen* aber klein geschrieben werden!

3 Numerische Fehler und Grenzen

Zusammen mit dem weiteren Einüben in MATLAB wollen wir uns nun mit den aus der Endlichkeit der Computer–Wortlänge folgenden Begrenzungen des Zahlenbereichs und der Genauigkeit befassen. Das Thema kommt in fast allen Büchern über Numerik vor. Besonders elementar und ausführlich wird es in Kapitel 2 in [2] diskutiert, von wo auch einige der folgenden Beispiele stammen.

Im letzten Unterabschnitt dieses und der folgenden Kapitel werden — soweit erforderlich — jeweils einige neue MATLAB –Befehle vorgestellt, die im Zusammenhang mit dem Thema und den Übungsaufgaben zu dem jeweiligen Abschnitt nötig sind.

3.1 Zahlenbereich

MATLAB greift, wie Fortran oder C, auf die floating-point Arithmetik des Rechners zurück. Es werden *double precision* Zahlen verwendet. Diese werden in 64 Bits, d.h. 8 Bytes abgelegt. (Auf einer 32-bit Maschine in zwei Wörtern). Diese Bits sind in i.a. rechnerabhängiger Weise aufgeteilt um das Vorzeichen, die Mantisse und den Exponenten der Gleikommazahlen zu speichern. Bei den hier benutzten HP–workstations ist der *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985* implementiert.

Bei diesem Standard werden für eine *double precision* Zahl 1 Bit für das Vorzeichen, 11 Bits für den Exponenten und 52 Bits für die Mantisse verwendet.

Eine *single precision* Zahl besteht aus 1 Bit für das Vorzeichen, 8 Bits für den Exponenten und 23 Bits für die Mantisse.

Rechnen wir dies in das Dezimalsystem um, erhalten wir für *single precision*:

$$\text{Gleitkommabereich(32Bit)}: \quad \sim 10^{-38} \ldots \sim 10^{+38} \tag{3.1}$$

und für *double precision*:

$$\text{Gleitkommabereich(64Bit)}: \quad \sim 10^{-308} \ldots \sim 10^{+308} \tag{3.2}$$

Wo das nicht reicht, wird meist mit merkwürdigen Algorithmen und unangepaßten physikalischen Einheiten gerechnet bzw. man ist mit der Fehlersuche (debugging = Entwanzung) noch nicht fertig. Typischer ist es, daß die beschränkte Länge der Mantisse ein (numerisches) Problem bildet.

3.2 Genauigkeit

Rechnen wir die 52 bits der Mantisse in Stellen ins Dezimalsystem um, erhalten wir knapp 16 Stellen. Dies hat unter anderem zur Folge, daß z.B. "$1 + 10^{-16} = 1$". D.h. es gibt eine Zahl $\epsilon > 0$, so daß für die vom Rechner durchgeführte Addition gilt

$$\begin{aligned} 1 + z &= 1 \quad \text{für} \quad 0 \le z \le \epsilon = \mathrm{O}(10^{-16}) \\ 1 + z &> 1 \quad \text{für} \quad z > \epsilon. \end{aligned} \tag{3.3}$$

ϵ ist die "Maschinengenauigkeit" (machine accuracy). ϵ kann als der typische *relative* Fehler angesehen werden, mit dem unsere Rechengrößen immer behaftet sind. Damit ist gemeint, daß in Verallgemeinerung des obigen Beispiels Zahlen A und $A + \epsilon A$ im Rechner nicht unterscheidbar sind. Der Grund liegt natürlich darin, daß mit gegebener Stellenzahl nur eine endliche Untermenge von *rationalen* Zahlen in die reellen Zahlen hineingelegt werden kann. Eine beliebige reelle Zahl wird also durch eine naheliegende solche *darstellbare* Zahl genähert. Für Zahlen der Größenordnung 1 kann man diese als regelmäßiges eindimensionales Gitter mit einer Gitterkonstanten $\mathrm{O}(\epsilon)$ als "Auflösung" ansehen, und dies ist der typische Fehler, auch Rundungsfehler genannt.

Genauer gesagt ist dies i. a. eine untere Schranke an den relativen Fehler auftretender Zahlen. Wenn eine Größe x z. B. durch

$$x = A - B, \quad |x|/A = \mathrm{O}(10^{-n}), \quad A, B > 0 \tag{3.4}$$

gebildet und weiterverwendet wird (kleine Differenz großer Zahlen), so ist diese nur noch auf $16-n$ Stellen genau. Bezeichnen wir die Fehler mit $\delta x, \delta A, \delta B$, so gilt

$$|\delta x| = |\delta A - \delta B| \sim \epsilon A + \epsilon B = \mathrm{O}(\epsilon A). \tag{3.5}$$

Dies ist eine Größenordnungsbetrachtung. Die Fehler $\delta A, \delta B$ können beide Vorzeichen haben. Man muß daher vom ungünstigen Fall ausgehen und Beträge addieren. Weiter sind für diese Betrachtung A und B (etwa) gleichgroß, und der dem entsprechende Faktor 2 vor $\mathrm{O}(\epsilon A)$ wurde ignoriert. Somit ist der relative Fehler von x wie behauptet

$$\frac{|\delta x|}{|x|} \sim \frac{\epsilon A}{|x|} \sim 10^n \, \epsilon, \tag{3.6}$$

und es sind n Stellen "verlorengegangen". In welchem Maße dieser Signifikanzverlust passiert, hängt zwar oft von der Organisation der Rechnung ab, aber ganz vermeiden läßt er sich meist nicht. Das gilt etwa für das folgende Beispiel.

3.3 Numerische Ableitung

Angenommen, man kann zwar auf Werte einer Funktion zugreifen, nicht aber auf die Ableitung in geschlossener Form. Dann muß diese numerisch genähert werden. Man beginnt mit der Taylorentwicklung

$$f(x \pm h) = f(x) \pm f'(x)h + \frac{1}{2}f''(x)h^2 + \mathrm{O}(h^3) \tag{3.7}$$

und erhält Näherungsformeln mit Differenzenquotienten:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathrm{O}(h) \tag{3.8}$$

und die verbesserte symmetrische Form

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathrm{O}(h^2). \tag{3.9}$$

Sowohl (3.8) als auch (3.9) beinhalten für kleine h die Subtraktion fast gleicher Zahlen. Ist h aber groß, dann bekommen wir Fehler durch das Restglied in (3.7). Irgendwo muß ein optimaler Wert für h liegen. Dazu das folgende MATLAB –Programm:

```
% programm numdiff
% tabelliert numerische Ableitungen von sin(x) bei x=1
% als Funktion der Schrittweite
%
help numdiff
%
h = 10.^(-[1:16]); % zu testende Schrittweiten
x = 1;
dex = cos(1); % exakt
d1 = (sin(x+h)-sin(x))./h;       % asym. Formel
d2 = (sin(x+h)-sin(x-h))./(h+h); %  sym. Formel
y = [ h ; (d1-dex)/dex ; (d2-dex)/dex ];
%
fprintf('   h       asymm.     symm. \n\n')
fprintf(' %5.0e %10.1e %10.1e \n',y)
```

Zum besseren Verständnis sollte man hier genau analysieren, welche Größen welche Matrixstruktur haben. Ggf. bitte einzelne Komponenten `h, x+h, sin(x+h), sin(x),...` separat betrachten (interaktiv). Das Resultat bei Ablauf des Programms sieht so aus:

```
>> numdiff

  programm numdiff
  tabelliert numerische Ableitungen von sin(x) bei x=1
  als Funktion der Schrittweite


   h       asymm.     symm.

 1e-01   -7.9e-02   -1.7e-03
 1e-02   -7.8e-03   -1.7e-05
 1e-03   -7.8e-04   -1.7e-07
 1e-04   -7.8e-05   -1.7e-09
 1e-05   -7.8e-06   -2.1e-11
 1e-06   -7.8e-07   -5.1e-11
 1e-07   -7.7e-08   -3.6e-10
 1e-08   -5.5e-09    4.8e-09
 1e-09    9.7e-08   -5.5e-09
 1e-10   -1.1e-07   -1.1e-07
 1e-11   -2.2e-06   -2.2e-06
 1e-12    8.0e-05   -2.3e-05
 1e-13   -1.4e-03   -3.3e-04
 1e-14    6.9e-03   -3.4e-03
 1e-15    2.3e-01    1.3e-01
 1e-16   -1.0e+00   -1.0e+00
>>
```

Wir sehen hier, daß (3.8) eine optimale Genauigkeit (relative Abweichung) von etwa 10^{-8} erreicht bei $h = 10^{-8}$, während (3.9) auf 10^{-11} bei $h = 10^{-5}$ kommt. Das ist also 1000 mal genauer bei etwa gleichem Aufwand.

Diese Zahlen sind durch Größenordnungsbetrachtungen leicht zu verstehen. In beiden Fällen erwarten wir für die gebildete Differenz eine relative Genauigkeit von $\sim \epsilon f/(hf')$ wegen Rundungsfehler und Signifikanzverlust. Der relative Fehler in f' durch den nächsten Term der Taylorentwicklung ist hf''/f' in (3.8) und $h^2 f'''/f'$ in (3.9) (ohne Faktoren 2 etc!). Optimales h_o ist bei gleichgroßen Fehlern beiden Typs gegeben, und die Abschätzung liefert

$$h_o \sim \epsilon^{1/2}; \quad \delta f'/f' \sim \epsilon^{1/2} \tag{3.10}$$

für Formel (3.8) und

$$h_o \sim \epsilon^{1/3}; \quad \delta f'/f' \sim \epsilon^{2/3} \tag{3.11}$$

für Formel (3.9). Hier wurden Werte und Ableitungen von f bei x als O(1) angenommen, was für $f = \sin$ und $x = 1$ gilt. Man sieht, das die grobe Abschätzung gut zu den Zahlen paßt. Gleichzeitig wird auch klar, daß das nicht für beliebige pathologische Funktionen gelten kann. Auch bei speziellen Argumenten muß man die Analyse abändern, z. B. an Nullstellen der Ableitungen. Der typische, "generische" Fall ist aber erklärt. In praktischen Rechnungen ist es jedenfalls oft vorteilhaft bis nötig, die Abhängigkeit von Parametern wie h und auch verschiedene Näherungsformeln zu studieren.

3.4 Numerische Grenzwertbildung

Die Exponentialfunktion besitzt für $n \to \infty$ die folgende Grenzwertdarstellung:

$$\exp(x) = \left(1 + \frac{x}{n}\right)^n \; (1 + \mathrm{O}(x^2/n)). \tag{3.12}$$

Es stellt sich die Frage, ob diese Formel numerisch brauchbar ist. Nach dem bisher gesagten ist das Optimum erreicht, wenn der relative Fehler für $\exp(x)$ auf etwa 10^{-16} gedrückt werden kann. Um festzustellen, ob dies möglich ist, benutzen wir das folgende MATLAB –Programm:

```
% das programm test_exp_1.m bestimmt exp(1) mittels
% der Formel (1+1/n)^n und vergleicht das Resultat
% mit dem exakten Ergebnis.
%
clear
n=10.^[1:1:12];
s=(1+1./n).^n;
delta=(s-exp(1))/exp(1)
ndelta=delta.*n;
%
x=[n;delta;ndelta];
fprintf('    n       delta        n*delta \n\n')
fprintf('%10.4e %11.4e %11.4e \n',x)
```

Nach dem Aufruf des Programms bekommen wir die folgende Tabelle geliefert:

```
1.0000e+01 -4.5815e-02 -4.5815e-01
1.0000e+02 -4.9546e-03 -4.9546e-01
1.0000e+03 -4.9954e-04 -4.9954e-01
1.0000e+04 -4.9995e-05 -4.9995e-01
1.0000e+05 -4.9999e-06 -4.9999e-01
1.0000e+06 -5.0006e-07 -5.0006e-01
1.0000e+07 -4.9472e-08 -4.9472e-01
1.0000e+08 -1.5474e-08 -1.5474e+00
1.0000e+09  7.4442e-08  7.4442e+01
1.0000e+10  7.9100e-08  7.9100e+02
1.0000e+11  7.4185e-08  7.4185e+03
1.0000e+12  8.8895e-05  8.8895e+07
```

Wir kommen nur auf eine relative Genauigkeit von etwa 10^{-7}, dann wird es wieder schlechter! In der letzten Spalte wird der Fehler mit n multipliziert. Da der *systematische* Fehler der Formel $\mathrm{O}(1/n)$ ist, müßte dies konstant werden. Das ist eine Weile wahr, offenbar mit Koeffizient $-1/2$ (das können Sie bestimmt beweisen!), dann geht es auf und davon. Es ist ziemlich klar, was los ist. In der Klammer hat $a = 1+1/n$ einen intrinsischen Darstellungsfehler von etwa $\epsilon = \mathrm{O}(10^{-16})$. Nun gilt $(a+\epsilon)^n \approx a^n(1+n\epsilon/a)$ solange $n\epsilon \ll 1$. Der typische Rundungsfehler erscheint also um einen Faktor n vergrößert. Wir haben also für den Gesamtfehler, die Summe aus systematischem und Rundungsfehler,

$$\text{Gesamtfehler} \sim 1/n + \epsilon n \tag{3.13}$$

Diese Kombination ist minmal für $n \simeq \epsilon^{-1/2} \simeq 10^8$ und hat dann die Größenordnung 10^{-8}. Für kleinere n dominiert der erste Term (systematischer Fehler), für größere der zweite (Rundungsfehler). Das paßt gut zum Experiment und zeigt, daß die Formel (3.12) unbrauchbar ist, um e^x in voller Genauigkeit zu berechnen.

Besser ist da schon die gewöhnliche Taylorreihe um $x = 0$,

$$\exp(x) = \sum_{i=0}^{n} \frac{x^i}{i!} + \mathrm{O}\left(\frac{x^{n+1}}{(n+1)!}\right) \ . \tag{3.14}$$

Nach dem Auruf des Programms

```
% Das Program exp_test_2.m bestimmt exp(1) mittels
% der gew"ohnlichen Taylorreihe und vergleicht das Resultat
% mit dem exakten Ergebnis.
```

```
%
clear
% Bestimmung von n!
%
nfac(1)=1;
for i=2:20, nfac(i)=nfac(i-1)*i; end
%
% Ausf"uhren der Summation
%
s(1)=1+1/nfac(1);
for i=2:20, s(i)=s(i-1)+1/nfac(i); end
delta=(s-exp(1))/exp(1);
%
% Ausdruck
%
fprintf('  i      delta \n\n')
for i=1:20,
fprintf('%2i %11.4e \n',i,delta(i))
end
```

erhalten wir die Tabelle:

```
i      delta

 1 -2.6424e-01
 2 -8.0301e-02
 3 -1.8988e-02
 4 -3.6598e-03
 5 -5.9418e-04
 6 -8.3241e-05
 7 -1.0249e-05
 8 -1.1252e-06
 9 -1.1143e-07
10 -1.0048e-08
11 -8.3161e-10
12 -6.3598e-11
13 -4.5197e-12
14 -2.9979e-13
15 -1.8461e-14
```

```
16 -8.1686e-16
17  1.6337e-16
18  1.6337e-16
19  1.6337e-16
20  1.6337e-16
```

Die von MATLAB oder von Compilern (Bibliotheken) zur Verfügung gestellte Exponentialfunktion ist sicher raffinierter. Insbesondere muß sie mit großen Argumenten $|x|$ anders verfahren, da dann die obige Reihe zwar noch konvergiert, aber kein effektives Verfahren mehr darstellt.

3.5 Rekursionsformeln

Oft begegnet man Rekursionen. Ein einfacher Fall ist durch die folgenden Integrale gegeben:

$$p_n = \int_0^1 \mathrm{d}x x^n \exp(x). \tag{3.15}$$

Durch partielle Integration erhält man leicht die Rekursionsbeziehung

$$p_{n+1} = e - (n+1)p_n. \tag{3.16}$$

und den Startwert $p_1 = 1$. Im Prinzip, d.h. mit beliebig genauer Arithmetik, kann man nun von p_1 ausgehend durch Anwenden von (3.16) leicht zu beliebigen p_n kommen. Wenn wir dies für die ersten 20 Terme tun, passiert folgendes:

```
>> e=exp(1);
>> p(1)=1;
>> for i=1:19, p(i+1)=e-(i+1)*p(i); end
>> [[1:20]' p']
ans =

    1.0000    1.0000
    2.0000    0.7183
    3.0000    0.5634
    4.0000    0.4645
    5.0000    0.3956
    6.0000    0.3447
    7.0000    0.3055
```

```
 8.0000    0.2744
 9.0000    0.2490
10.0000    0.2280
11.0000    0.2103
12.0000    0.1951
13.0000    0.1820
14.0000    0.1705
15.0000    0.1605
16.0000    0.1503
17.0000    0.1624
18.0000   -0.2043
19.0000    6.5991
20.0000 -129.2637
```

Ein kurzer Blick auf das Integral zeigt, daß p_n in Wahrheit monoton mit n fallen muß, da der Integrand für jedes $0 \leq x < 1$ mit wachsendem n fällt. Das numerische Resultat kann also zumindest bei größeren n nicht stimmen!

Angenommen, wir kennen die exakte Lösung p_n^* und hätten beliebige Genauigkeit. Starten wir die Rekursion von einem leicht abweichenden Wert $p_n = p_n^* + \delta_n$, so pflanzt sich gemäß (3.16) die Abweichung fort mit

$$\delta_{n+1} = -(n+1)\delta_n. \tag{3.17}$$

Das wächst wie $n!$, und es braucht nur wenige Schritte, bis eine Abweichung der Größe ϵ sich so aufbläst, daß die eigentliche Lösung völlig untergeht. Das oszillierende Vorzeichen ist im Zahlenbeispiel klar zu sehen.

Hier ergibt sich gleich eine hilfreiche Idee. Die gleichen Argumente zeigen, daß man in umgekehrter Richtung *stabil* iterieren kann,

$$p_n = \frac{e - p_{n+1}}{(n+1)}, \tag{3.18}$$

wobei die Abweichung bei jedem Schritt mit $1/(n+1)$ gedämpft wird. Das ist so stark, daß man mit einem beliebigen p_{20} der Ordnung 1 beginnen kann, nach zwei bis drei Schritten sicher prozentgenau ist und schließlich mit Genauigkeit ϵ bei den unteren p_n landet:

```
>> q(20)=1.23456789;
>> for i=19:-1:1, q(i)=(e-q(i+1))/(i+1); end
>> [[1:20]' p' q' qe']
```

```
ans =

    1.0000    1.0000    1.0000    1.0000
    2.0000    0.7183    0.7183    0.7183
    3.0000    0.5634    0.5634    0.5634
    4.0000    0.4645    0.4645    0.4645
    5.0000    0.3956    0.3956    0.3956
    6.0000    0.3447    0.3447    0.3447
    7.0000    0.3055    0.3055    0.3055
    8.0000    0.2744    0.2744    0.2744
    9.0000    0.2490    0.2490    0.2490
   10.0000    0.2280    0.2280    0.2280
   11.0000    0.2103    0.2103    0.2103
   12.0000    0.1951    0.1951    0.1951
   13.0000    0.1820    0.1820    0.1820
   14.0000    0.1705    0.1705    0.1705
   15.0000    0.1605    0.1604    0.1604
   16.0000    0.1503    0.1515    0.1515
   17.0000    0.1624    0.1433    0.1434
   18.0000   -0.2043    0.1392    0.1362
   19.0000    6.5991    0.0742    0.1297
   20.0000 -129.2637    1.2346    0.1238
```

Der Vektor `qe` in der letzten Spalte ist die bis auf ϵ exakte Lösung, die durch Abwärtsrekursion von $n = 30$ gewonnen wurde. In der Abbildung 2 wurde $|q(i)|$ für eine Reihe verschiedener Anfangswerte $q(20)$ halblogarithmisch als Funktion der Iterationszahl i aufgetragen.

Die Situation, daß eine Rekursion nur in einer Richtung stabil ist, ist recht typisch. Nachdem man dies versteht, hat man eine elegante Methode, die gesuchten Integrale zu berechnen. Gleichzeitig haben wir aber gesehen, wie man mit 16 Stellen Genauigkeit leicht Unsinn produzieren kann.

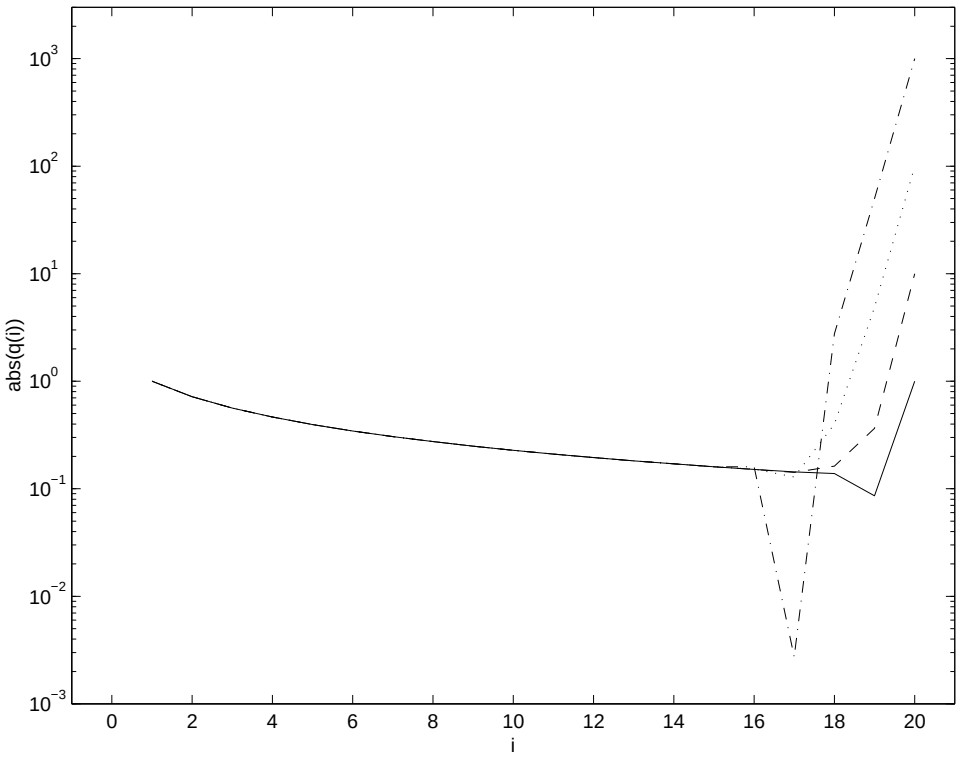

Abbildung 2: semilogy(i,abs(q(i)))

3.6 Mehr MATLAB

Bei den Beispielen dieses Abschnitts war es notwendig, eine Matrix zu transponieren. Für eine Matrix `a` ist in MATLAB `a'` die adjungierte Matrix, d. h. transponiert und komplex konjugiert. Da wir hier reelle Größen hatten, war dieser Unterschied egal. Reine Transposition ist aber mit `a.'` auch möglich.

```
>> a

a =

   1.0000            2.0000            3.0000 + 1.0000i
   4.0000            5.0000            6.0000

>> a'
```

```
ans =

   1.0000             4.0000
   2.0000             5.0000
   3.0000 - 1.0000i   6.0000

>> a.'

ans =

   1.0000             4.0000
   2.0000             5.0000
   3.0000 + 1.0000i   6.0000
```

Bei den Aufgaben in Übungsblatt 3 ist es bequem, Schleifen mit **while** laufen zu lassen.

```
>> help while

 WHILE  Repeat statements an indefinite number of times.
        The general form of a WHILE statement is:

           WHILE variable, statement, ..., statement, END

        The statements are executed while the variable has all
        non-zero elements. The variable is usually the result of
        expr rop expr where rop is ==, <, >, <=, >=, or ~=.
        For example (assuming A already defined):

                E = 0*A; F = E + EYE(E); N = 1;
                WHILE NORM(E+F-E,1) > 0,
                   E = E + F;
                   F = A*F/N;
                   N = N + 1;
                END
```

Als logisches Argument für **while** kann man u. a. die Funktion **isinf(x)** einsetzen. Sie ist logisch wahr (1 in MATLAB), wenn x "unendlich" ist ($\pm$

INF, vgl. Übungsblatt 2) und falsch (0 in MATLAB) sonst. Mit ~ kann auch negiert werden.

Neben **plot** gibt es noch weitere Plotbefehle **semilogx, semilogy** und **loglog**, bei denen jeweils die Teilung der x-, y- oder beider Achsen logarithmisch ist. Die möglichen Argumente sind immer die gleichen. Mit jedem dieser Befehle kann man auch simultan mehrere Kurven in ein Bild zeichnen. Ein häufiger Fall ist, daß man einen Zeilenvektor x mit x-Werten hat und Zeilen y1, y2, y3 (oder mehr) für die Kurven. Dann leistet der Befehl **plot(x,[y1 ; y2 ; y3])** das Gewünschte. Man hat also aus den 3 Zeilen eine Matrix gemacht und als y-Werte vorgegeben.

Will man anders plotten als mit durch Linien verbundenen Punkten wie bisher, so ist ein String als drittes Argument erforderlich wie z. B. in **plot(x,y,'*')**. Hier werden die Punkte mit dem Symbol '*' gezeichnet und nicht verbunden. Analoges passiert bei '+' , '.' , 'o' , 'x'. Verschiedene Verbindungslinientypen gibt es mit '−' , ':' , '−− , '−.' Der String hat auch noch weitere Funktionen wie Farbsteuerung. Will man mehrere Kurven mit verschiedenen Symbolen, so kann man **plot(x,y1,s1,x,y2,s2,x,y3,s3)** nehmen, wobei die Strings s1, s2, s3 Zeichenanweisungen für die einzelnen Kurven sind.

4 Nullstellensuche

In diesem Kapitel diskutieren wir Methoden zum Auffinden von Nullstellen beliebiger nichtlinearer Funktionen. Da man allgemein die Bestimmung von reellen Größen $x_i, i = 1, \ldots, n$ durch irgendwelche Beziehungen in die Form

$$\vec{f}(\vec{x}) = 0 \tag{4.1}$$

bringen kann, handelt es sich um ein sehr allgemeines Problem trotz des harmlosen Aussehens. I. a. kann es passieren, daß, auch wenn $\vec{f}$ genau n Komponenten hat und damit n Gleichungen für n Unbekannte vorliegen, das System keine oder aber mehrere Lösungen hat.

Wir betrachten hier nur den Fall einer Unbekannten, $n = 1$. Selbst dann können die erwähnten Entartungen auftreten, aber im generischen Fall, für den wir Algorithmen studieren wollen, wird $f(x)$ in einem gewissen Intervall eine Nullstelle mit *Vorzeichenwechsel* haben, die wir als Lösung genau bestimmen wollen. Die meisten Bücher über Numerik nehmen sich dieses Themas an; hier sollen [3] und [2] empfohlen sein.

4.1 Zum Beispiel

In der sogenannten Mean–Field–Näherung für die klassische Ising–Theorie des Ferromagnetismus fixiert die folgende Gleichung die spontane Magnetisierung für ein kubisches Gitter:

$$m = \tanh(6\beta m). \tag{4.2}$$

Hier ist m die Magnetisierung pro Spin und β die inverse Temperatur, jeweils in geeigneten Einheiten. Gesucht wird also eine Nullstelle von

$$f(m) = \tanh(6\beta m) - m \tag{4.3}$$

Es gibt immer die Lösung $m = 0$. Wie an Abb.3 graphisch illustriert wird, kommt aber für $\beta \geq 1/6$, also für genügend tiefe Temperatur (unterhalb der Curie–Temperatur), eine weitere Lösung $m \neq 0$ hinzu, die hier interessiert, da sie im Modell dem Ferromagnetismus entspricht. Überhaupt ist es bei eindimensionalen Nullstellenproblemen nützlich, erst mal einen Plot der relevanten Funktionen zu erstellen.

In MATLAB definieren wir eine Funktion `mffunc(m)`

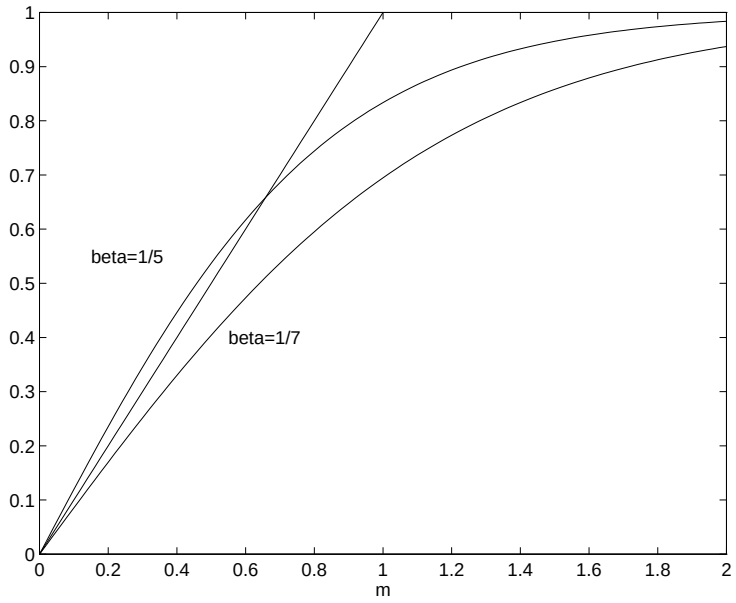

Abbildung 3: Beide Seiten von Gl.(4.2)

```
%
% file mffunc.m
%
% Funktion, deren Nullstellen die Loesung fuer die Magnetisierung
% m im Beispiel Problem geben
%
% beta wird als globale Variable benoetigt
%
function [y] = mffunc(m)
global beta
y = m - tanh(6*beta*m)
```

deren Nullstellen die gesuchte Magnetisierung sind. β wird als globale Variable übergeben. Mit der Kommandosequenz

```
>> global beta
>> beta=0.2;
>> m=0:.05:1;
>> plot(m,mffunc(m),[0 1],[0 0])
```

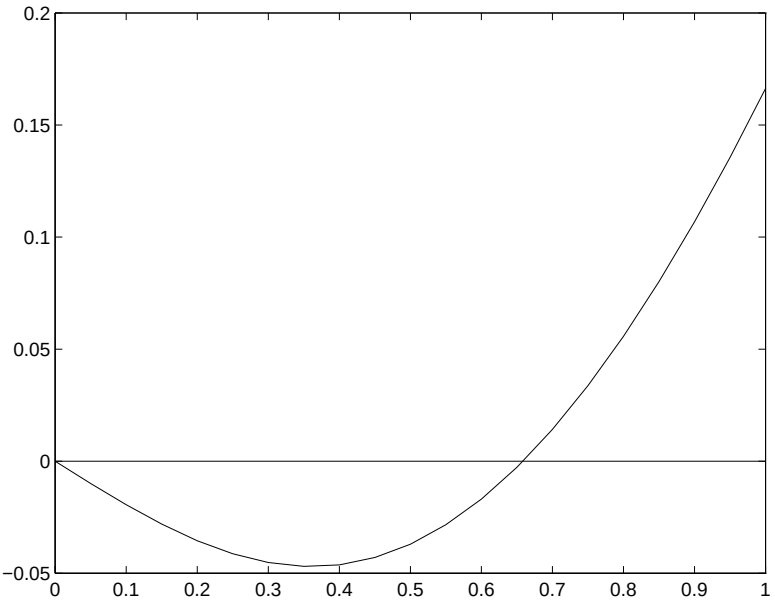

Abbildung 4: Die Funktion (4.3) mit $\beta = 0.2$

erhält man den Plot der Funktion in Abb.4, der klar eine Nullstelle zwischen $m = 0.6$ und $m = 0.7$ zeigt. Die beiden hinteren Argumente in `plot` sorgen für die waagerechte Nulllinie.

4.2 Eine spezielle Methode

Im vorliegenden Problem sieht man leicht, daß eine Iteration der Form

$$x_{n+1} = \tanh(6\beta x_n), \tag{4.4}$$

ausgehend von irgendeinem $x_1 > 0$, eine Folge liefert, die gegen eine Lösung von (4.2) konvergiert. Und zwar wird die von Null verschiedene Nullstelle angesteuert, sobald sie existiert (je nach β). Hierbei erfolgt die Konvergenz monoton von der Seite her, wo gestartet wird. Das hier gesagte sieht man ein, indem man sich ein qualitatives Bild wie Abb.3 malt und die Werte der Folge konstruiert. Man beachte, daß dies für unseren Spezialfall eine globale Analyse darstellt und nicht nur eine Betrachtung in einer kleinen Umgebung der Lösung. Dies funktioniert offenbar genauso, wenn man statt tanh eine andere monton konvexe Funktion hat.

Das Konvergenzverhalten für Probleme von der Form

$$g(x) = h(x) \tag{4.5}$$

bei Iteration

$$x_{n+1} = g^{-1}(h(x_n)) \tag{4.6}$$

kann man lokal quantitativ analysieren. Es sei x_* die gesuchte Lösung. Dann iteriert der Abstand $\delta_n = x_n - x_*$ in der Nähe der Lösung mit

$$\delta_{n+1} = \frac{h'(x_*)}{g'(x_*)}\delta_n. \tag{4.7}$$

In unserem Beispiel ist $g' = 1$ und $h'(x_*) < 1$ und damit Konvergenz grundsätzlich gesichert. Man nennt dieses Verhalten, wo δ_{n+1} proportional zu δ_n ist und abnimmt, *lineare* Konvergenz[5]. Obwohl dieses Verhalten erst in der Nähe der Lösung gilt, kann man damit versuchen, die Anzahl der benötigten Schritte zu schätzen. Wenn $(h')^N \sim 1/10$ gilt, also $N \sim -1/\log_{10}(h')$, dann gewinnt man mit je N Iterationsschritten eine Dezimalstelle Genauigkeit; d. h. etwa $16N$ Schritte für Maschinengenauigkeit wenn x_* und der Anfangsfehler δ_1 die gleiche Größenordnung haben. Offenbar wird die Sache problematisch, wenn h' nahe bei 1 liegt, also wenn in unserem Beispiel x_* klein ist. Ein Beispiel für die Konvergenz zeigt Abb.5. Man braucht also ca. 90 Schritte zur bestmöglichen Genauigkeit in MATLAB , schon bei unkritischen Parameterwerten. Wir werden sehen, daß es hier wesentlich effektivere Methoden gibt. Dabei muß man sich vor Augen halten, daß bei vielen Anwendungen das Berechnen der eingehenden Funktion teuer sein kann, so daß es wirklich darauf ankommt.

4.3 Bisektion

Bisektion ist eine einfaches Verfahren, einen Vorzeichenwechsel der Funktion $f(x)$, deren Nullstelle gesucht ist, immer weiter einzukreisen. Zum Start muß ein Intervall $[x_1, x_2]$ angegeben werden, in dem die Nullstelle liegt. Dies geht durch Plotten und ansehen oder durch einfache Suchprogramme, die den Wertebereich absuchen oder durch zusätzliche, z. B. physikalische Informationen. In unserem Beispiel ginge $x_1 =$ einige ϵ, $x_2 = 1$ für den nichtrivialen

[5] obwohl zu beachten ist, daß der Fehler *exponentiell* fällt

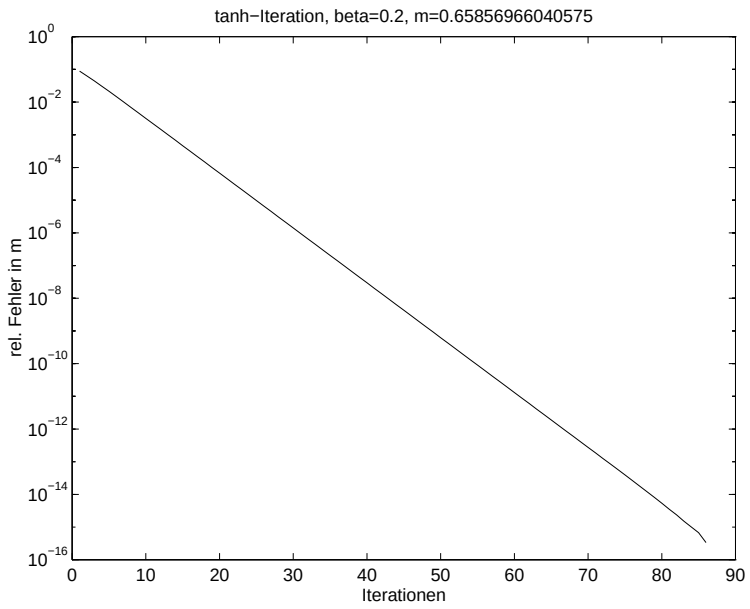

Abbildung 5: Konvergenz von Gl.(4.4)

Fall. Dann wird das Intervall fortlaufend halbiert und die Hälfte weiterverwendet, in der der Vorzeichenwechsel liegt.

Wir demonstrieren dies an einem MATLAB Programm.

```
%
% file bisect.m
%
% Programm zur Nullstellensuche durch Bisektion
%
% Aufruf: x0 = bisect(func,x1,x2,tol)
%
% x1,x2: Anfangsintervall; x0: Nullstelle
% ES MUSS SEIN: x1 < x2, func(x1)*func(x2) <= 0
% func: Funktionsname als Stringkonstante, z.B. 'sin'
% tol: Gewuenschte absolute Genauigkeit
%
  function [x0] = bisect(func,x1,x2,tol)
  f1=feval(func,x1); if f1 == 0, x0=x1; return; end
```

```
  f2=feval(func,x2); if f2 == 0, x0=x2; return; end
  if f1*f2 >= 0 | x1 >= x2
    error(' keine Nullstelle wg. x1, x2');
  end
% Bisektionsloop:
  for i=1:100,
    x0 = 0.5*(x2+x1);
    if x2 - x1 < tol, return; end
    f0=feval(func,x0);
    if f0*f1 <= 0
      x2=x0; f2=f0;
    else
      x1=x0; f1=f0;
    end
  end
  error(' keine Nullstelle gefunden')
```

Hier gleich zwei Anwendungsbeispiele:

```
>> bisect('sin',3,3.5,1.e-15)

ans =

   3.14159265358979

>> ans-pi

ans =

     4.440892098500626e-16

>> global beta; beta=0.2;
>> bisect('mffunc',0.1,1,1.e-15)

ans =

   0.65856966040575

>>
```

Wir wollen gleich hier auf ein paar neue MATLAB –Elemente hinweisen. Da ist zum einen der **return** Befehl, der aus einer **function** ins aufrufende Programm oder (interaktiv) ans Terminal zurückspringt. **error('text')** ist ähnlich im Fehlerfall, steigt aus, gibt eine Fehlermeldung mit Programmnamen und `text` auf dem Terminal aus. Am File-Ende einer **function** gibt es automatisch den Rücksprung. Zum anderen wird mit `func` ein Funktionsname übergeben. Um mit dieser beliebigen eingebauten oder selbstdefinierten Funktion zu rechnen, dient **feval**. `feval(name,1)` gibt also das gleiche wie `sin(1)`, wenn vorher `name='sin'` definiert wurde.

Das Konvergenzverhalten von Bisektion ist einfach zu überblicken. Der absolute Fehler nach n Schritten ist

$$\delta_n = |x_2 - x_1| 2^{-n-1}. \tag{4.8}$$

Da in MATLAB $\epsilon = 2^{-52}$ gilt, ist 52 eine typische maximal sinnvolle Schrittzahl. Es handelt sich also wieder um lineare Konvergenz, und pro Schritt ist eine Funktionsberechnung nötig.

4.4 Newton-Raphson–Verfahren

Die Newton-Raphson–Methode approximiert die Funktion $f(x)$ am jeweils untersuchten Argument durch ihre Taylor–Entwicklung. Dies geht im einfachsten Fall bis zur ersten (linearen) Ordnung. D. h. neben einer Routine, die $f(x)$ liefert, muß es eine weitere geben für $f'(x)$. Diese Ableitung soll analytisch vorliegen; falls das nicht der Fall ist, dann ist es besser, z. B. die Sekantenmethode des folgenden Abschnitts zu nehmen, die ohne Ableitung auskommt, als Newton-Raphson zusammen mit einer numerischen Ableitungsberechnung.

Angenommen, wir sind in der Iterationsfolge bei x_n. Nun wird approximiert

$$f(x) \approx f(x_n) + (x - x_n) f'(x_n) \stackrel{!}{=} 0 \tag{4.9}$$

und man bekommt

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{4.10}$$

Geometrisch legt man also bei x_n eine Tangente an die Funktion und approximiert sie mit dieser Geraden. Bitte aufmalen.

Die Fehlerbetrachtung — vorausgesetzt man ist in der Nähe der Nullstelle und die Taylor–Näherung ist gut — geht wie folgt. Durch Abziehen der Lösung auf beiden Seiten gilt

$$\delta_{n+1} = \delta_n - \frac{f(x_n)}{f'(x_n)}. \tag{4.11}$$

Sodann gilt

$$\frac{f(x_n)}{f'(x_n)} \simeq \frac{\delta_n f'(x_*) + \frac{1}{2}\delta_n^2 f''(x_*)}{f'(x_*) + \delta_n f''(x_*)} \simeq \delta_n \left(1 - \delta_n \frac{f''(x_*)}{2f'(x_*)}\right) + \mathrm{O}(\delta_n^3) \tag{4.12}$$

und damit kann man näherungsweise die Fortpflanzung der Abweichung δ_n angeben zu

$$\delta_{n+1} \approx \delta_n^2 \frac{f''(x_*)}{2f'(x_*)}. \tag{4.13}$$

Das ist quadratische Konvergenz. Einmal im Gültigkeitsbereich dieser Formel, wird in jedem Schritt die Zahl der korrekten Stellen *verdoppelt*. In [2] ist gezeigt, daß es für hinreichend glatte Funktionen immer eine Umgebung der Nullstelle gibt, wo eine Fehlerformel des obigen Typs als echte Ungleichung gilt. Die Größe dieser Umgebung hängt aber von $f(x)$ ab und wird i. a. nicht bekannt sein. Da bei beliebigen Funktionen und endlicher Entfernung von der Lösung die abgebrochene Taylor Entwicklung aber beliebig falsch sein *kann*, selbst wenn die mathematischen Voraussetzungen erfüllt sind, ist Newton-Raphson ein Verfahren ohne Garantie. Obwohl oft schnell konvergent, kann die Iteration auch nach unendlich abwandern. Oft wird die Methode nur für den letzten Schliff benutzt ("polishing of roots" in [3]). D. h. wenn man z. B. mit Bisektion ganz in der Nähe ist, dann wird in wenigen Schritten (quadratische Konvergenz) Maschinengenauigkeit erreicht.

4.5 Sekantenverfahren

Newton-Raphson benötigt analytische Kontrolle über die Ableitung. Ist dies nicht gegeben, z. B. wenn $f(x)$ einer komplizierten Simulation entstammt, so arbeitet man am besten gleich mit der Sekante statt der Tangente. In der Iteration liegt es nahe, dazu die letzten beiden x-Werte zu nehmen und aus (4.10) wird

$$x_{n+1} = x_n - f(x_n)\,\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}. \tag{4.14}$$

Offenbar braucht man hier zwei Werte, um die Iteration zu starten. Die Iterationsformel ist offensichtlich anfällig gegen Signifikanzverlust und overflow–Probleme, da in Zähler und Nenner des Bruchs bei Konvergenz zur Lösung kleine Differenzen entstehen. Ein empfohlenes Abbruchkriterium (von eventuell mehreren) ist [2]

$$|f(x_n) - f(x_{n-1})| \leq \tau |f(x_n)| \tag{4.15}$$

wobei τ eher etwas größer als ϵ sein sollte.

Zur Fehlerbetrachtung leiten wir zunächst aus (4.14) ab

$$\begin{aligned} \delta_{n+1} &\simeq \delta_n - \delta_n f'(x_*) \frac{\delta_n - \delta_{n-1}}{(\delta_n - \delta_{n-1}) f'(x_*) + \frac{1}{2}(\delta_n^2 - \delta_{n-1}^2) f''(x_*)} \\ &\simeq \frac{f''(x_*)}{2f'(x_*)} \delta_n \delta_{n-1} + \mathrm{O}(\delta_n^2) \end{aligned} \tag{4.16}$$

Die Rekursion $\delta_{n+1} = C\delta_n\delta_{n-1}$ ist mathematisch recht interessant. Ihre Lösung hängt mit der Folge der Fibonacci–Zahlen zusammen [2]. Wir wollen hier nur erwähnen, daß das Resultat ein Verhalten

$$|\delta_{n+1}| \approx c|\delta_n|^\alpha \tag{4.17}$$

ist. Wenn man diesen Ansatz in (4.16) einsetzt ergibt sich eine quadratische Gleichung $\alpha(\alpha - 1) = 1$. Deren positive Lösung ist

$$\alpha = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62. \tag{4.18}$$

Die strengere Analyse der Fehler–Rekursion zeigt, daß tatsächlich diese Asymptotik in der Nähe der Nullstelle angenommen wird. Damit ist a posteriori auch die Vernachlässigung von $\mathrm{O}(\delta_n^2)$ in (4.16) gerechtfertigt. Die Konvergenzgeschwindigkeit des Sekantenverfahrens liegt also zwischen der sicheren Bisektion und Newton-Raphson. Mit letzterem teilt es das Fehlen allgemeiner Konvergenz und wird daher auch oft mit Bisektion kombiniert.

4.6 Nullstellenprogramm in MATLAB

In MATLAB gibt es auch ein fertiges Nullstellenprogramm. Es heißt **fzero**, und **help** weiß folgendes darüber:

```
>> help fzero
FZERO  Find zero of function of one variable.
       FZERO(F,X) tries to find a zero of F.
       F is a string containing
       the name of a real-valued function of a single
       real variable. The value returned is near a point
       where F changes sign, or NaN if the search fails.

       FZERO(F,X), where X is a vector of length 2,
       assumes X is an interval where the sign of F(X(1))
       differs from the sign of F(X(2)).  An error occurs
       if this is not true.  Calling FZERO with an interval
       guarantees FZERO will return a value near a point
       where F changes sign.

       FZERO(F,X), where X is a scalar value, uses X as
       a starting guess. FZERO looks for an interval containing
       a sign change for F and containing X.  If no such
       interval is found, NaN is returned. In this case,
       the search terminates when the search interval is
       expanded until an Inf, NaN, or complex value is found.

       FZERO(F,X,TOL) sets the relative tolerance for
       the convergence test. FZERO(F,X,TOL,TRACE) displays
       information at each iteration when TRACE is nonzero.
       FZERO(F,X,TOL,TRACE,P1,P2,...) allows for
       additional arguments which are passed to the
       function, F(X,P1,P2,...).  Pass an empty matrix for
       TOL or TRACE to use the default value.
```

Wie man sieht, braucht es nur einen Startwert in der Nähe der Nullstelle, nicht ein Intervall. Das Programm sucht zunächst dann ein Intervall, in dem ein Vorzeichenwechsel liegt. Dazu betrachtet es (willkürlich!) ein Intervall von 5% unter- und oberhalb des vorgelegten Startwerts ($[-1/20, 1/20]$, falls dieser Null ist). Ist dort kein Vorzeichenwechsel, so wird das Intervall jeweils um einen Faktor 2 vergrößert, bis ein Wechsel gefunden ist. Danach wird mit Bisektion und Interpolation (linear oder quadratisch) zwischen den Intervall-

grenzen gearbeitet. Letzters stellt eine weitere Methode dar. Es ist klar, daß dieses Programm für vernünftige Funktionen gedacht ist, die auf einer nicht-pathologischen Skala variieren wie $\sin(x)$, etc. Mit $f(x) = \sin(1000x)$ wird es schwierig. Bei physikalischen Problemen ist dies meist erreichbar durch Wahl geeigneter Einheiten, wo die Maßzahlen nicht zu extrem sind.

Es ist klar, daß auch dieses Programm nicht narrensicher ist. Gibt es mehrere Nullstellen, so entscheidet der Startwert, welche gefunden wird; jede hat also einen gewissen Anziehungsbereich ("basin of attraction") im Raum der Startwerte, was für alle Algorithmen gilt. Mit dem Kommando **type fzero** wird das in MATLAB geschriebene Programm `fzero.m`, das als Teil der MATLAB –Installation automatisch aus einer Programm–Bibliothek kommt, auf dem Schirm (bzw. im diary-file) ausgeben. **dbtype** macht das gleiche, versieht das Programm aber mit Zeilennummern. Versuchen Sie, das Programm soweit wir möglich zu verstehen. Am besten ausdrucken. Wir haben nicht vollständig die Fehlerabsicherung dieses 'professionellen' Programms analysiert. Sie scheint aber eher unvollständig. Testen Sie die Routine mit legalen und illegalen Fällen. (Falls MATLAB mal nicht zurückkommt oder zu lange dauert: <ctrl>c). Wir bekommen hier einen realistischen Eindruck vom Nutzen und eventuellen Problemen mit "black box"–Routinen, wobei es hier noch güstig ist, daß man den Quellcode analysieren kann.

4.7 Newton-Raphson und Fraktale

Hinter der Frage, welche Startwerte zu welcher Nullstelle konvergieren, verbirgt sich bei geeigneten Systemen interessante Mathematik. Betrachten wir die Funktion

$$f(z) = z^3 - 1 \tag{4.19}$$

für *komplexe* z. Es ist klar, daß die Nullstellen durch die Einheitswurzeln $1, \exp(\pm 2\pi i/3)$ gegeben sind. Die interessante Frage ist, wie sich die komplexe Ebene der möglichen Startwerte aufteilt in verschiedene Bereiche, von denen aus man zu den verschiedenen Nullstellen konvergiert. Dabei gehen diese und auch ihre Basins of Attraction ineinander über unter Drehungen um $2\pi/3$. Es genügt, die Konvergenz zu einer von ihnen zu studieren.

Für komplex differenzierbare Funktionen gilt nun das Analogon zu (4.10) mit den komplexen Größen,

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)} = z_n - \frac{z_n^3 - 1}{3z_n^2}. \tag{4.20}$$

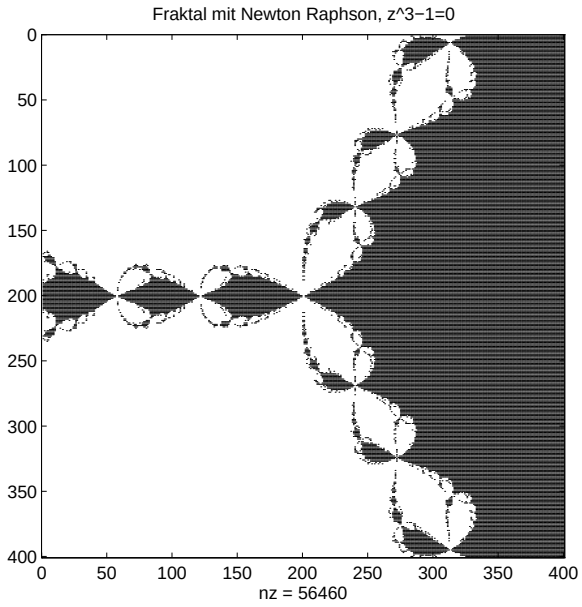

Abbildung 6: Basin of Attraction zu Newton-Raphson, $z^3 = 1$

Das folgende MATLAB –Programm führt von einem als Argument gegebenen Startwert eine Newton-Raphson–Iteration durch bis zur Genauigkeit `tol`:

```
%
%  file nr3.m
%
%  Newton Raphson Iteration fuer die komplexe Funktion z^3-1
%
%  Aurgument: Startwert; Antwort: Nullstelle
%  tol wird als globale Variable benoetigt
%
function [z] = nr3(zn);
global tol;
zo = inf;
del = tol*abs(zn);
while abs(zn-zo) > del
 zo=zn;
 zn = zo - (zo^3-1.0)/(3.0*zo^2);
```

```
end
z=zn;
```

Hier kommt nur bekannte Syntax vor. Das Hauptprogramm ist nun

```
%
%  file nr3frac.m
%
%  Fuer ein Gitter in der komplexen Ebene wird
%  entschieden welche Punkte mit Newton-Raphson
%  zur Loesung z=1 der Gleichung z^3-1=0 iterieren
global tol; tol=1.e-13; % Konvergenzkriterium fuer NR
tol10=10*tol;
%
t=cputime; % zur Bestimmung der Rechenzeit
size=199.5;
maxy=2.0;
re=[-size:size]*(maxy/size); % Gitter-Werte fuer den Realteil
im=re*sqrt(-1);              % Gitter-Werte fuer den Imaginaerteil
%
n=length(re);
m=zeros(n,n); % leere Matrix vordefiniert
%
for i=1:n, for j=1:n
  m(i,j) = (abs(nr3(re(j)+im(i))-1) < tol10); % m(,)=1 g.d.w. N.-R. -> 1
end,end
%
cputime-t %Bestimmung und Ausdruck der verbrauchten CPU-Zeit
spy(m);
title('Fraktal mit Newton-Raphson, z^3-1=0');
```

Hier ist neu die Funktion **cputime** (ohne Armumente), die die seit Programmstart verstrichene Rechnerzeit liefert. Damit kann man die in bestimmten Abschnitten verbrauchte Zeit ermitteln, z. B. um Programme zu optimieren. Die Zeile `m=zeros(n,n);` erzeugt eine $n \times n$ Matrix mit Nullen. Diese ist damit praktisch vordeklariert und reserviert, was Vorteile bezüglich der Geschwindigkeit bietet. Für ein Gitter in der komplexen Ebene der Größe 400×400 im Beispiel wird für jeden Punkt ermittelt, ob er zur 1 konvergiert mit Funktion `nr3`. Die Matrix `m` erhält logisch ja (=1 in MATLAB),

wo das der Fall ist und falsch (=0 in MATLAB) sonst. Der Befehl **spy(m)** stellt die Matrix graphisch dar. Das Resultat ist nun in Abb.6 zu sehen. Die Berechnung dieses Plots hat auf einer HP 735 etwa 7 min. gedauert, auf einem Linux–PC mit AMD Athlon/900MHz dagegen nur noch 41 sek! Ein mit Optimierung kompiliertes FORTRAN–Programm erzeugt die gleiche Matrix etwa 10 mal schneller.

5 Lineare Gleichungssysteme

Die natürlichste Fortsetzung des letzten Abschnitts wäre es nun, viele simultan zu lösende nichtlineare Gleichungen zu betrachten. Hier gibt es bei voller Nichtlinearität allerdings wenig allgemein taugliche "rezeptartige" Verfahren. Dies ist jedoch der Fall bei Systemen von u. U. auch sehr vielen *linearen* Gleichungen. Es geht also wieder um die Lösung von

$$\vec{f}(\vec{x}) = 0 \tag{5.1}$$

für Vektoren $\vec{x}$ mit Komponenten $x_j, j = 1, \ldots, n$. Dabei soll speziell gelten

$$f_i(\vec{x}) = \sum_{j=1}^{n} a_{ij} x_j - b_i, \;\; i = 1, \ldots, m. \tag{5.2}$$

Alle hier beteiligten Größen können komplex sein, es wird aber vorläufig der reelle Fall betrachtet. Häufig wird das System quadratisch sein, $m = n$, wobei dann im generischen Fall, wo die Matrix A mit Matrixelementen a_{ij} nicht singulär ist, genau eine Lösung $\vec{x}$ existiert, die es zu finden gilt.

5.1 Naive Gauß–Elimination

Bei der Gauß–Elimination werden Gleichungen kombiniert, bis sie aufgelöst werden können. Statt allgemeiner Formeln wollen wir zunächst ein Beispiel betrachten. Die folgenden Koeffizienten sollen bei $m = n = 4$ verwendet werden und wurden in MATLAB eingegeben:

```
A =

     6    -2     2     4
    12    -8     6    10
     3   -13     9     3
    -6     4     1   -18
b =

    16
    26
   -19
   -34
```

Gesucht ist die Lösung x von[6]

$$Ax = b. \tag{5.3}$$

Nun wird das a_{i1}/a_{11} fache der ersten Gleichung (=Zeile) von der 2. bis 4. abgezogen. Dann werden offenbar in der ersten Spalte unter der Diagonalen Nullen erzeugt. Praktisch ist es dabei eine 4×5 Matrix zu bilden, indem man die Spalte b noch an A dranschreibt:

```
>> B=[A b]

B =

     6    -2     2     4    16
    12    -8     6    10    26
     3   -13     9     3   -19
    -6     4     1   -18   -34

>> for i=2:4, B(i,:)=B(i,:)-B(1,:)*B(i,1)/B(1,1); end
>> B

B =

     6    -2     2     4    16
     0    -4     2     2    -6
     0   -12     8     1   -27
     0     2     3   -14   -18
```

Hier war ein neues MATLAB Element zu sehen: die Adressierung von Teilmatrizen. Mit `B(i,:)` bekommen wir einen Zeilenvektor, bestehend aus der i'ten Zeile der Matrix B. Mit : in einer Indexposition spricht man also den gesamten Wertebereich an. Allgemein kann man Teilmatrizen bekommen wie z. B. die rechte untere Ecke von A,

```
>> disp( A(3:4,3:4) )
     9     3
     1   -18
```

[6] Wir wollen die Vektorpfeile von nun an weglassen: x statt bisher $\vec{x}$

Weiteres ist mit **help colon** zu erfahren.

Nun fahren wir fort, Nullen in der zweiten Spalte zu erzeugen,

```
>> for i=3:4, B(i,:)=B(i,:)-B(2,:)*B(i,2)/B(2,2); end
>> B

B =

     6    -2     2     4    16
     0    -4     2     2    -6
     0     0     2    -5    -9
     0     0     4   -13   -21
```

Nach einem weiteren Schritt haben wir

```
B =

     6    -2     2     4    16
     0    -4     2     2    -6
     0     0     2    -5    -9
     0     0     0    -3    -3
```

und können nun von *hinten* auflösen durch Rückwärtssubstitution

$$\begin{aligned} -3x_4 &= -3 \\ 2x_3 &= -9 + 5x_4 \\ -4x_2 &= -6 - 2x_3 - 2x_4 \\ 6x_1 &= 16 + 2x_2 - 2x_3 - 4x_4 \end{aligned}$$

und bekommen

$$x_1 = 3, x_2 = 1, x_3 = -2, x_4 = 1. \tag{5.4}$$

Dieses Resultat überprüfen wir durch Bildung von Ax oder dadurch, daß in MATLAB die Lösung von linearen Systemen schon vorgesehen ist. Dabei wird die Lösung von $Ax = b$ geschrieben als $x = A\backslash b$ (Merkregel: "beide Seiten von links durch A teilen mit \ ").

```
>> x=A\b

x =
```

```
    3.0000
    1.0000
   -2.0000
    1.0000
```

Das gibt es auch für die transponierte Gleichung $yA^T = b^T$, wo y eine Zeile ist: `y = b'/A'` in MATLAB . Siehe **help slash**.

Nach dieser Vorbereitung ist es recht einfach, ein allgemeines Programm zu schreiben. Hier ist es:

```
%
%  file gaussel.m
%
%  x=gaussel(A,b) liefert die Loesung des linearen
%  Systems A*x=b durch Gauss Elimination ohne Pivotisierung.
%  b und x koennen mehrere Spalten haben
%
%
function [x] = gaussel(A,b)
[m,n]=size(A);
if m~=n | n~=size(b,1), error('kein quadratisches problem'); end;

B=[A b];
N=size(B,2);

% Triangularisierung (Vorwaertselimination):

for k=1:n-1,    % loop ueber Spalten wo Nullen entstehen
  fac=1/B(k,k);
  for i=k+1:n   % loop ueber zu subtrahierende Zeilen
    fac1=fac*B(i,k); % Multiplikator
    B(i,k)=0; % neue Null per Konstruktion
    B(i,k+1:N)=B(i,k+1:N)-B(k,k+1:N)*fac1; % Subtraktion
  end
end

% Aufloesung (Rueckwaertssubstitution):
x=zeros(size(b)); % x vordefinieren
```

```
for k=n:-1:1
  x(k,:)=B(k,n+1:N);
  for j=k+1:n
    x(k,:)=x(k,:)-B(k,j)*x(j,:);
  end
  x(k,:)=x(k,:)/B(k,k);
end
```

Die MATLAB –Funktion **size** liefert die Größe einer Matrix; bei `size(A)` werden zwei Zahlen zurückgegeben, `size(A,1)` gibt nur die Zahl der Zeilen, `size(A,2)` die der Spalten. In `gaussel` sieht man eine typische Verwendung. Man muß also nicht wie z. B. in Fortran die Dimension als Parameter übergeben.

Die Rückwärtssubstitution, nachdem A Dreiecksform hat, sieht in Formeln wie folgt aus:

$$\sum_{j=i}^{n} a_{ij}x_j = b_i \Rightarrow x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj}x_j}{a_{kk}}. \tag{5.5}$$

Benutzt man nun diese Gleichungen in der Reihenfolge $k = n, n-1, \ldots, 1$, so ist die rechte Seite jeweils komplett vorhanden und man hat die explizite Auflösung.

Das obige Programm funktioniert auch für mehrere rechte Seiten, die dann als Spalten in b stehen. Setzt man insbesondere für b die Einheitsmatrix ein, so liefert das Programm die zu A inverse Matrix.

```
>> b=eye(4)

b =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

>> x=gaussel(A,b)

x =
```

```
   -3.4861    2.1528   -0.6944    0.3056
    8.2917   -4.7917    1.4167   -0.5833
   11.9167   -6.9167    2.1667   -0.8333
    3.6667   -2.1667    0.6667   -0.3333

>> x*A

ans =

    1.0000    0.0000    0.0000    0.0000
    0.0000    1.0000    0.0000    0.0000
         0    0.0000    1.0000    0.0000
         0    0.0000    0.0000    1.0000

>> A*x

ans =

    1.0000    0.0000    0.0000    0.0000
    0.0000    1.0000    0.0000    0.0000
    0.0000    0.0000    1.0000    0.0000
         0    0.0000    0.0000    1.0000
```

Das Programm `gaussel` kann kopiert werden von ∼`uwolff/CP1/kap5`.

Wir wollen nun einen Test machen zur numerischen Genauigkeit unserer Routine. Dazu betrachten wir für eine Folge von Problemen mit zunehmender Größe n und erzeugen jeweils eine Matrix A mit zufälligen Elementen. Das geht mit **rand(m,n)** für eine $m \times n$ Matrix. Weiter bilden wir ebenso je einen zufälligen Vektor x. Nun *berechnen* wir $b = Ax$ und *lösen* $Ay = b$. Dann kann man für jedes n die maximale Diskrepanz zwischen den Komponenten x und y prüfen. Das Resultat ist in Abb.7 wiedergegeben. Neben dem Resultat von `gaussel` sehen wir auch die mit dem eingebauten $A\backslash b$ gewonnenen. Zunächst beobachten wir den erwarteten Trend des Fehlers, mit n zuzunehmen. Die Irregularität stammt daher, daß Zufallsmatrizen mal mehr und mal weniger "problematisch", d.h. fast singulär, sind. Weiter fällt auf, daß der Fehler von MATLAB fast stets kleiner ist, typischerweise um Faktoren 10–100. Tatsächlich benutzt MATLAB intern Gauß–Elimination mit Pivotisierung, und wir wollen uns diesem Punkt als nächstes zuwenden.

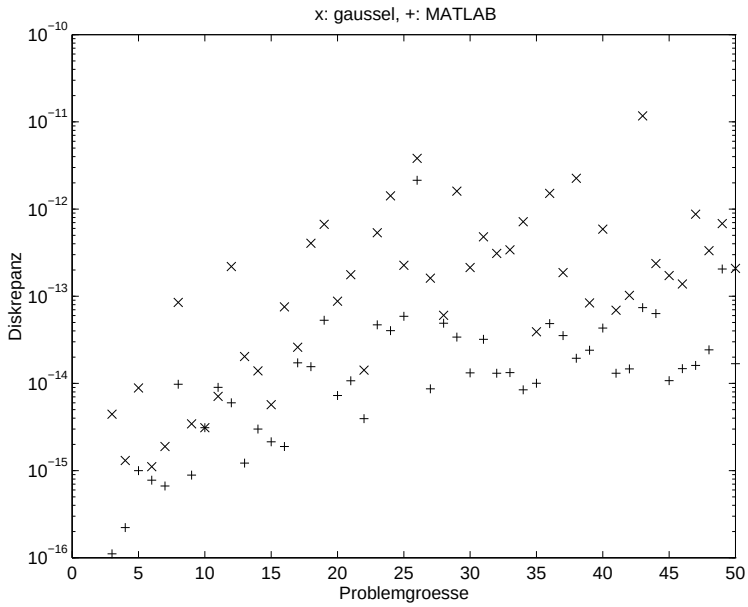

Abbildung 7: Fehler beim Lösen als Funktion der Problemgröße mit Zufallsmatrizen

Nur nebenbei soll erwähnt werden, daß unser Programm etwa 50 mal mehr CPU-Zeit brauchte, nämlich ca. 13 Sek. auf einer HP 735 für $n = 2 \ldots 50$. Die eingebaute Routine ist sicherlich optimal in C geschrieben und binär in den MATLAB –kernel integriert. Schließlich handelt es sich um ein Paket für lineare Algebra, und das Lösen linearer Systeme ist absolut zentral.

5.2 Pivotisierung

Es ist vielleicht schon aufgefallen, daß obige Gauß–Elimination in einer Hinsicht unnatürlich ist. Die Reihenfolge, in der man die n Gleichungen (Zeilen) schreibt, ist willkürlich. Dennoch benutzen wir die erste in spezieller Weise, um sie von allen anderen abzuziehen und in der ersten Spalte Nullen zu erzeugen, dann die zweite etc. Noch ein bedenkliches Zeichen ist, daß im ersten Schritt durch a_{11} zu dividieren ist. Dieses Element kann verschwinden bei einer völlig regulären Matrix.

Bei der Pivotisierung nutzt man die Freiheit systematisch, über die bisher in zufälliger Weise verfügt wurde. Man kann im Prozeß der Lösung zusätz-

lich Zeilen (inkl. b) vertauschen. Wird nur dies durchgeführt, so spricht man von Teilpivotisierung. Bei der vollen Pivotisierung werden auch noch Spalten vertauscht, wobei sich allerdings dann auch die Lösungskomponenten x_i vertauschen und eine Buchhaltung nötig ist, um dies am Ende rückgängig zu machen. Nach [3] bietet Vollpivotisierung nur geringe Vorteile bei erheblicher Komplikation. Wir wollen uns daher hier auf Teilpivotisierung beschränken.

Eine offensichtliche Strategie der Teilpivotisierung ist nun, zuerst die erste Zeile zu vertauschen[7] mit der j'ten, wenn j das maximale $|a_{j1}|$ hat und dann zu verfahren wie bisher. Sodann wird die zweite Zeile mit der j'ten, $j \geq 2$, $|a_{j2}|$ maximal, vertauscht, und dann werden die Nullen in Spalte zwei unter der Diagonalen erzeugt usw. Man kann sich überlegen, daß hier nur eine Division durch Null droht, wenn die Zeilen oder Spalten linear abhängig sind, die Matrix also singulär ist.

Bei der Implementierung als Programm muß man nicht unbedingt ganze Zeilen umspeichern — man denke an große Matrizen — eine Buchführung über die Vertauschungen genügt. Dazu nimmt man einen Vektor p, der am Anfang mit $p = [1\ 2 \ldots n]$ belegt ist. Eine Vertauschung nimmt man nun an den Komponenten von p vor. Wenn man nun p zum Indizieren benutzt, z. B. `a(p(1),:)`, so erreicht man offenbar das Gleiche wie mit Umspeichern.

Gegen das hier beschriebene Verfahren kann man immer noch einwenden, daß es von der Skalierung abhängt. D. h. wenn man die erste Gleichung z. B. mit 10^6 multipliziert, so wird ein $a_{11} \neq 0$ typischerweise erster Pivot werden, d. h. ein Vielfaches (mit a_{11} im Nenner) der ersten Zeile wird zu den anderen addiert. Die skalierte Teilpivotisierung geht gegen diese Willkür vor. Man bestimmt für jede Zeile einen Skalenfaktor $s_i = \max_j |a_{ij}|$. Dann richtet man sich beim Vertauschen nach der Größe von $|a_{j1}|/s_j$ usw. Gemäß [2] lohnt der Zusatzaufwand der skalierten Pivotisierung gegenüber der einfach partiellen oft nicht. Das gilt wohl, wenn die Matrixelemente "natürliche" Größen haben.

5.3 LU-Zerlegung

Beim Verfahren der LU-Zerlegung faktorisiert man die $n \times n$–Matrix A,

$$A = LU, \quad l_{ij} = u_{ji} = 0 \text{ wenn } i < j \tag{5.6}$$

wobei L eine untere (lower) Dreiecksmatrix ist und U eine obere (upper). Es wird unten konstruktiv gezeigt, daß dies möglich ist und daß man sogar

[7] Grenzfall der "Vertauschung mit sich selbst" möglich

die Diagonalelemente von L zu Eins machen kann[8] , $l_{ii} = 1$. Damit hat L $n(n-1)/2$ Parameter und U hat $n(n+1)/2$, zusammen n^2.

Das ursprüngliche Problem ist nun in zwei Schritten zu lösen:

$$Ly = b \tag{5.7}$$
$$Ux = y. \tag{5.8}$$

Der zweite Schritt ist genau die Rückwärtssubstitution aus dem letzten Abschnitt in (5.5). Der erste Schritt ist in trivialer Weise analog auszuführen, wobei man nun mit der oberen Gleichung beginnt (Vorwärtssubstitution). Hat man also die LU–Faktorisierung, so kann man viele Probleme mit der Matrix A leicht lösen, ohne alle rechten Seiten vorher kennen zu müssen wie bei der Gauß–Eimination, wo diese manipuliert werden mußten.

Zur Faktorisierung ist es nützlich, sie in Komponenten auszuschreiben:

$$a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj}. \tag{5.9}$$

Für $i \leq j$ kann man daraus bekommen

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \tag{5.10}$$

und für $i > j$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right). \tag{5.11}$$

Das Verblüffende ist, daß man mit diesen beiden Gleichungen alle nichttrivialen Elemente von L und U explizit bekommt, wenn man sie in folgender Reihenfolge bestimmt: $u_{11}, l_{21}, l_{31} \ldots l_{n1}, u_{12}, u_{22}, l_{32}, l_{42} \ldots l_{n2}, u_{13} \ldots\ldots u_{nn}$. D. h. auf den rechten Seiten treten ausschließlich jeweils schon bekannte Elemente auf. Das ganze ist der Crout'sche Algorithmus[3].

Auch hier ist es i. a. erforderlich zu pivotisieren, was etwas trickreich ist [3]. Es handelt sich wieder um Teilpivotisierung in dem Sinne, daß man nicht A selbst, sondern eine zeilenpermutierte Form von A zerlegt. Wenn man die Permutation speichert, so ist dies offenbar äquivalent, da man mit dieser Information die Permutation am Ende kompensieren kann. Angenommen, wir

[8] Konvention, U ginge auch stattdessen

sind an der Stelle, wo u_{jj},l_{j+1j},l_{j+2j},...,l_{nj} gemäß obigem Schema berechnet werden. Dazu bilden wir zunächst hilfsweise

$$c_i = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \; i = j, j+1, \ldots, n \tag{5.12}$$

was sowohl in (5.10) vorkommt als auch in (5.11), allerdings fehlt noch die Division. Ohne Pivotisierung wäre nun $u_{jj} = c_j$, $l_{ij} = c_i/c_j, i = j+1, \ldots, n$. Nun kann man sich klarmachen, daß an diesem Punkt eine ursprüngliche Vertauschung der j'ten mit der k'ten Zeile von A, $k > j$, äquivalent ist zur Vertauschung von c_j mit c_k. Davon macht man nun Gebrauch, um durch das vom Betrag her größtmögliche c_j zu teilen und Nulldivisionen zu vermeiden. Die Vertauschung wird in einem Permutationsvektor gespeichert und in dem bereits erstellten Teil von L und in A wirklich durchgeführt.

Auch hier bringt wieder wie bei der Gauß–Elimination die skalierte Pivotisierung einen weiteren Fortschritt, bei der man am Anfang für jede Zeile von A eine Skala bestimmt und beim Größenvergleich berücksichtigt.

Zu erwähnen ist noch, daß man bei großen Problemen Speicherplatz sparen kann, indem man die nichttrivialen Elemente von L, U im Speicherbereich von A ablegt. Man geht also spaltenweise durch A und transformiert es in U bzw. L. Bemerkenswerterweise sind (5.10), (5.11) so gebaut, daß das betreffende a_{ij} nicht mehr weiter benötigt wird und jeweils überschrieben werden kann. Auch bei den Zeilenvertauschungen zur Pivotisierung gibt es hier keine Probleme. Die ursprüngliche Matrix ist natürlich am Ende nicht mehr explizit vorhanden.

Die LU–Zerlegung kostet im wesentlichen $n^3/3$ Multiplikationen. Danach kann man ein lineares Gleichungssystem mit jeweils n^2 Multiplikationen pro rechter Seite auflösen. Beim Invertieren mit rechten Seiten e_1, $e_2 \ldots e_n$ kommt man bei Berücksichtigung der Nullen mit insgesamt n^3 Operationen aus. Bei dieser Analyse werden nur Multiplikationen und Divisionen gezählt und Beiträge der relativen Ordnung $1/n$ vernachlässigt.

In [3] gibt es Programme zur LU-Zerlegung und dem anschließenden Auflösen von linearen Systemen mit Hilfe der beiden Faktoren. Diese kurzen Routinen haben sich als Standard–"Arbeitspferd" für lineare Gleichungen und Matrixinversion in vielen Anwendungen bewährt. Auch in MATLAB gibt es ein Programm **lu**:

```
>> A=rand(4,4);
```

```
>> [L,U,P]=lu(A)

L =

    1.0000         0         0         0
    0.2433    1.0000         0         0
    0.5115   -0.8021    1.0000         0
    0.6387   -0.2069    0.2490    1.0000


U =

    0.9501    0.8913    0.8214    0.9218
         0    0.5453    0.2449    0.5140
         0         0    0.5682    0.3465
         0         0         0   -0.3924


P =

     1     0     0     0
     0     1     0     0
     0     0     0     1
     0     0     1     0

>> norm(P*A-L*U)

ans =

   9.2218e-17
```

Die Permutation vom Pivotisieren ist hier als Matrix zurückgegeben worden. Dieses Programm wird von anderen MATLAB –Routinen aufgerufen. Es ist binär und kann nicht mit **type** inspiziert werden, entspricht aber laut Handbuch dem Programm ZGEFA der LINPACK–Bibliothek.

5.4 Iterative Verbesserung der Lösung

Angenommen wir haben die Lösung $x^{(0)}$ der Gleichung $Ax = b$ numerisch gefunden. Diese Lösung ist mit Rundungsfehlern behaftet und im allgemeinen werden weniger als 16 Stellen genau sein. Nun kann man die Lösung iterativ verbesseren. Sei x^* die exakte Lösung der Gleichung und $\delta x = x^{(0)} - x^*$ der Fehler unserer numerischen Lösung. Es gilt

$$Ax^* = A(x^{(0)} - \delta x) = \tilde{b} - \delta b = b \quad , \tag{5.13}$$

wobei $\tilde{b} = Ax^{(0)}$ und $\delta b = \tilde{b} - b$. Damit erhalten wir

$$A\delta x = \delta b \tag{5.14}$$

als Bestimmungsgleichung für δx. Die verbesserte Lösung ist $x^{(1)} = x^{(0)} - \delta x$.

Hier sehen wir einen klaren Vorteil der LU-Zerlegung gegenüber der Gausselimination. δb ist erst bekannt, nachdem wir x berechnet haben.

5.5 Householder–Reduktion

Die Householder–Reduktion ist charakterisiert durch geschickt konstruierte (orthogonale) Transformationen, die besonders gut geignet sind, eine Matrix in Dreiecksform zu bringen, ohne numerische Instabilitäten zu riskieren. Sie ersetzt die Vorwärts–Elimination in der Gauß–Methode. Die Rückwärts–Substitution schließt sich dann wie gewohnt an.

Den entscheidenden Baustein bilden Reflexionen der Form

$$x \to x - 2w(w^T x) = Px; \quad P = 1 - 2ww^T. \tag{5.15}$$

Hier ist w ein Einheitsvektor, $w^T w = 1$. Es wird also das Vorzeichen der Komponente von x, die parallel zu w ist, geflippt und der Rest bleibt unverändert, was man auch als Reflexion an der ($n-1$-dimensionalen) Hyperebene senkrecht zu w bezeichnen kann. Intuitiv klar und leicht nachzurechnen ist, daß $P^2 = 1, P = P^T$ gilt, P also symmetrisch und orthogonal ist. Interessant ist, daß man für zwei beliebige Vektoren $x \neq y$ *gleicher Länge* erreichen kann, daß $Px = y$ gilt mit

$$w = \frac{x - y}{|x - y|}. \tag{5.16}$$

Zum Beweis kann man z. B. verifizieren, daß gilt

$$ww^T x = (x - y)\,\frac{x^T x - y^T x}{|x - y|^2} = \frac{1}{2}(x - y).$$

Sei nun $a^{(1)}$ die erste Spalte der Matrix A, dann besteht der erste Schritt der Elimination darin, eine Householder–Reflexion anzuwenden, die

$$a^{(1)} \to -\sigma_1 e_1 \tag{5.17}$$

transformiert, also die ganze erste Spalte in die obere linke Ecke der Matrix spiegelt. Dabei ist natürlich

$$\sigma_1 = \pm|a^{(1)}|, \tag{5.18}$$

denn eine Reflexion erhält die Länge des Vektors. Über das Vorzeichen verfügen wir später, zunächst konstruieren wir die Transformation P_1 mit dem Reflexionsvektor aus Glg.(5.16):

$$\begin{aligned} P_1 &= 1 - \frac{uu^T}{H} && (5.19)\\ \text{mit} \quad u^T &= (a_{11} + \sigma_1, a_{21}, \cdots, a_{n1}) && (5.20)\\ \sigma_1^2 &= \sum_{i=1}^{n} (a_{i1})^2 && (5.21)\\ H &= \frac{1}{2}|u|^2. && (5.22) \end{aligned}$$

Nun läßt sich der Normierungsfaktor H umformen zu

$$\begin{aligned} H &= \frac{1}{2}(|a^{(1)}|^2 + 2\sigma_1 a_{11} + \sigma_1^2) \\ &= \sigma_1(\sigma_1 + a_{11}) && (5.23) \end{aligned}$$

und man minimiert den Rundungsfehler, wenn σ_1 dasselbe Vorzeichen hat wie a_{11}:

$$\sigma_1 = \mathrm{sgn}(a_{11})\, |a^{(1)}| \tag{5.24}$$

Eine wichtige Beobachtung ist, daß man (im Unterschied zur Gauß–Methode) nie zu pivotisieren braucht: es wird immer die Norm von $a^{(1)}$ auf die Diagonale gebracht, unabhängig davon, wie die Komponenten in der Spalte verteilt sind. Wenn aber diese Norm verschwindet, dann ist die Matrix A singulär, und das Gleichungssystem ist (mathematisch!) nicht für allgemeine b lösbar. Man muß allenfalls überwachen, daß $|a^{(1)}| > \epsilon a$ bleibt, wobei ϵ die Maschinengenauigkeit ist und a eine typische Größe der a_{ij} bedeutet.

Nun wendet man die Reflexionsmatrix P_1 von links auf das Gleichungssystem $Ax = b$ an. Das bedeutet, daß auch die anderen Spalten A und

die rechte Seite b zu transformieren sind, nicht aber die Komponenten von x (genauso wie bei der Gauß–Elimination!). So entsteht das transformierte Gleichungssystem $P_1 A x = P_1 b$.

Entsprechend verfährt man mit den Komponenten der zweiten Spalte von der Diagonale an abwärts, die man mit der Transformation P_2 auf die Diagonale spiegelt. Dabei wird die erste Zeile nicht mehr berührt, d.h. man hat einen Reflexionsvektor der Form

$$\begin{aligned} u^T &= (0, a'_{22} + \sigma_2, a'_{32}, \cdots, a'_{n2}) && (5.25)\\ \text{mit} \quad \sigma_2^2 &= \sum_{i=2}^{n} (a'_{i2})^2 && (5.26)\\ \mathrm{sgn}(\sigma_2) &= \mathrm{sgn}(a'_{22}) && (5.27)\\ \Rightarrow P_2 &= 1 - \frac{u u^T}{H} && (5.28)\\ \text{mit} \quad H &= \sigma_2(\sigma_2 + a'_{22}) && (5.29) \end{aligned}$$

Die Matrixelemente sind hier mit a'_{ij} bezeichnet, um daran zu erinnern, daß sie im vorangehenden Eliminationsschritt schon verändert wurden.

So arbeitet man sich vor bis zur Spalte $n-1$. Schließlich hat man eine Dreiecksmatrix $R = Q^T A$ mit

$$Q^T = P_{n-1} P_{n-2} \cdots P_2 P_1 \tag{5.30}$$

und ein ebenso transformiertes $b' = Q^T b$. Die Produktmatrix Q^T wird beim Eliminationsverfahren nicht berechnet (aber das könnte man ohne weiteres tun). Sie ist aus Reflexionen aufgebaut und deshalb natürlich orthogonal:

$$\begin{aligned} Q &= P_1 P_2 \cdots P_{n-2} P_{n-1} && (5.31)\\ Q^T Q &= 1. && (5.32) \end{aligned}$$

Demnach ist $A = QR$ eine Produktzerlegung von A in eine orthogonale und eine dreieckige Matrix. Da sich beide Faktoren leicht invertieren lassen, spielt diese **QR–Zerlegung** eine ähnliche Rolle wie die zuvor besprochene LU–Zerlegung: man bekommt z.B. leicht das Inverse $A^{-1} = R^{-1} Q^T$.

Die Householder–Methode braucht (bei großen Matrizen) doppelt so viele Operationen wie die Gauß–Elimination, auch ist für jede Spalte eine Wurzel zu ziehen. Dafür hat sie den Vorteil größerer numerischer Stabilität, und man braucht nicht zu pivotisieren.

6 Eigenwerte

Ein Eigenwertproblem besteht, in der Notation des letzten Abschnitts, aus der Frage nach Vektoren $x \neq 0$ und Zahlen λ, so daß gilt

$$Ax = \lambda x. \tag{6.1}$$

Im allgemeinen Fall sind alle Größen hier komplex. Selbst reelle A können auf komplexe λ führen (paarweise zu einander konjugiert).

6.1 Tatsachen aus der linearen Algebra

Wir wollen einige Resultate referieren, die z. B. in [3] zusammengefaßt sind und in [4] ausführlich diskutiert werden. Aus (6.1) folgt, daß λ genau dann ein Eigenwert ist, wenn gilt

$$P(\lambda) = \det(A - \lambda 1) = 0, \tag{6.2}$$

denn dann hat das lineare System $(A - \lambda 1)x = 0$ eine nicht verschwindende Lösung. Bedenkt man die Definition von det, so ist klar, daß P ein Polynom n'ten Grades ist für eine $n \times n$–Matrix A, das charakteristische Polynom von A. Dieses hat im Komplexen n Nullstellen, die auch für spezielle Wahlen von a_{ij}, also im nicht generischen, aber doch häufigen Fall zusammenfallen können. Das Auffinden dieser Werte ist i. a. ein transzendentes Problem, zu dem es keine Algorithmen mit fester Zahl von Operationen, wie z.B. bei der Lösung linearer Systeme, geben wird. Stattdessen wird man bestenfalls iterative Methoden wie bei der allgemeinen Nullstellensuche haben, die dann irgendwie konvergieren.

Im generischen Fall gibt es n verschiedene linear unabhängige Eigenvektoren, die also den ganzen Raum aufspannen. Für lauter verschiedene Werte λ sind die Eigenvektoren sogar eindeutig bis auf einen Faktor, der in (6.1) irrelevant ist und somit durch eine Konvention fixiert werden kann. Für beliebige Matrizen und entartete, d.h. zusammenfallende Eigenwerte *kann* es "zu wenige" Eigenvektoren geben. Der Fall ist nicht so exotisch, wie das folgende 2×2 Beispiel zeigt:

$$A = \begin{pmatrix} a & 1 \\ 0 & b \end{pmatrix}, \tag{6.3}$$

$$P(\lambda) = (a - \lambda)(b - \lambda) \;\Rightarrow \lambda_1 = a, \lambda_2 = b. \tag{6.4}$$

Die zugehörigen Eigenvektoren sind trivial zu bestimmen,

$$x^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, x^{(2)} = \begin{pmatrix} 1 \\ b-a \end{pmatrix}, \tag{6.5}$$

wo hier die Konvention $x_1 = 1$ verwendet ist. Im Spezialfall $a = b$ gibt es offenbar nur einen Eigenvektor zu einem doppelten Eigenwert.

Wir setzen nun n Eigenvektoren zu $\lambda_1 \dots \lambda_n$ voraus und schreiebn diese als Spalten in die Matrix X_R der Rechtseigenvektoren. Dann gilt die Matrixgleichung

$$AX_R = X_R \,\mathrm{diag}(\lambda_1 \dots \lambda_n), \tag{6.6}$$

wobei $\mathrm{diag}(\lambda_1 \dots \lambda_n)$ die diagonale Matrix aus den Eigenwerten ist. Durch Betrachten des zu (6.1) transponierten Problems

$$yA = \lambda y \tag{6.7}$$

für Zeilen y (gleiche λ !) erhält man auch

$$X_L A = \mathrm{diag}(\lambda_1 \dots \lambda_n) X_L \tag{6.8}$$

Aus beiden Gleichungen zusammen ergibt sich

$$X_L X_R \mathrm{diag}(\lambda_1 \dots \lambda_n) = \mathrm{diag}(\lambda_1 \dots \lambda_n) X_L X_R. \tag{6.9}$$

Die Matrix $X_L X_R$ besteht aus den Skalarprodukten der Links- und Rechtseigenvektoren. Wenn $T = X_L X_R$ mit der Diagonalmatrix der Eigenwerte kommutiert, so heißt das in Komponenten $T_{ij}(\lambda_i - \lambda_j) = 0$. Daraus folgt für den nichtentarteten Fall $\lambda_i \neq \lambda_j$, daß $X_L X_R$ selbst diagonal ist. Die Links- und Rechtseigenvektoren sind orthogonal und können so normiert werden, daß gilt

$$y^{(i)} x^{(j)} = \delta_{ij}. \tag{6.10}$$

Bei entarteten Eigenwerten *kann* der Fall der unvollständigen Eigenvektoren auftreten. Ist dies nicht der Fall, so kann man (6.10) immer noch erreichen. Damit sind dann X_L und X_R invers zueinander, und es gilt

$$\mathrm{diag}(\lambda_1 \dots \lambda_n) = X_R^{-1} A X_R = X_L A X_R \tag{6.11}$$

und A ist ähnlich zur Diagonalmatrix seiner Eigenwerte.

Bisher wurden allgemeine komplexe Matrizen betrachtet. Wir wollen nun zunehmend spezialisieren. Wenn gilt

$$A^\dagger A = AA^\dagger, \tag{6.12}$$

so heißt eine Matrix normal, wobei die adjungierte Matrix $A^\dagger$ tansponiert und komplex konjugiert ist. Solche Matrizen haben immer ein vollständiges System von Eigenvektoren, die man orthonormal wählen kann

$$x^{(i)\dagger} x^{(j)} = \delta_{ij}, \tag{6.13}$$

so daß X_R unitär ist. Dann gilt $X_L = X_R^{-1} = X_R^\dagger$, und die Linkseigenvektoren sind die adjungierten Rechtseigenvektoren.

Noch spezieller, aber physikalisch (Quantenmechanik) von größter Bedeutung ist der hermitesche Fall, $A = A^\dagger$, woraus die Eigenschaft "normal" trivial folgt. Hier sind dann alle Eigenwerte reell, und aus Obigem folgt die unitäre Diagonalisierbarkeit. Ein Spezialfall der hermiteschen Matrizen wiederum sind die reellen symmetrischen, $A = A^T$, auf die wir uns im folgenden beschränken wollen.

6.2 Jacobi–Methode

Wir wollen nun die iterative Jacobi Methode für eine reell symmetrische Matrix A einführen. Wir wissen, daß diese orthogonal diagonalisierbar ist,

$$\mathrm{diag}(\lambda_1 \dots \lambda_n) = O^T A O, \tag{6.14}$$

wobei die Spalten von O die Rechtseigenvektoren sind. Die Idee ist, O aus besonders einfachen orthogonalen Transformationen sukzessive aufzubauen,

$$O = P_1 P_2 P_3 \dots, \tag{6.15}$$

d. h. man transformiert

$$A \to P_1^T A P_1 \to P_2^T P_1^T A P_1 P_2 \to \dots \tag{6.16}$$

solange, bis die modifizierte Matrix (in der gewünschten Näherung) diagonal ist. Will man auch die Eigenvektoren, so muß man gleichzeitig die verwendeten P_k aufmultiplizieren. Bei der Wahl der P_k wird eine Strategie verfolgt, die durch die Transformationen ständig die "Fehlerfunktion"

$$S = \sum_{i \neq j} |a_{ij}|^2 \tag{6.17}$$

verkleinert. Es bleibt nun, die Form der P_k anzugeben. Hier wählt man Drehungen in einer beliebigen pq-Ebene ($p < q$) des n-dimensionalen Raumes,

$$p_{ij} = \begin{cases} 1 & \text{für } i = j \notin \{p, q\} \\ c & \text{für } i = j \in \{p, q\} \\ s & \text{für } i = p, j = q \\ -s & \text{für } i = q, j = p \\ 0 & \text{sonst} \end{cases} \tag{6.18}$$

mit $c = \cos(\phi), s = \sin(\phi)$ für den Drehwinkel ϕ. Für einen solchen Schritt $A' = P^T A P$ schreiben wir nun die Matrixelemente von A', die gegen A verändert sind ($i \notin \{p, q\}$):

$$a'_{ip} = ca_{ip} - sa_{iq} \tag{6.19}$$
$$a'_{iq} = ca_{iq} + sa_{ip} \tag{6.20}$$
$$a'_{pp} = c^2 a_{pp} + s^2 a_{qq} - 2sca_{pq} \tag{6.21}$$
$$a'_{qq} = s^2 a_{pp} + c^2 a_{qq} + 2sca_{pq} \tag{6.22}$$
$$a'_{pq} = (c^2 - s^2)a_{pq} + sc(a_{pp} - a_{qq}) \tag{6.23}$$

Verlangt man nun $a'_{pq} = 0$, so wird der Drehwinkel bestimmt durch

$$\theta = \cot(2\phi) = \frac{c^2 - s^2}{2sc} = \frac{a_{qq} - a_{pp}}{2a_{pq}}. \tag{6.24}$$

Mit $t = s/c$ führt dies auf

$$t^2 + 2t\theta - 1 = 0. \tag{6.25}$$

Es hat sich numerisch bewährt, die Lösung dieser quadratischen Gleichung zu nehmen, die zu $\phi \to 0$ führt für $a_{pq} \to 0$,

$$t = \frac{\mathrm{sgn}(\theta)}{|\theta| + \sqrt{\theta^2 + 1}}. \tag{6.26}$$

Diese Form ist gegen Rundungsfehler stabil. Sollte θ^2 den Zahlenbereich überschreiten, so kann $t = 0$ gesetzt werden. Für den Ersetzungsschritt $A \to A'$ kann man noch umschreiben

$$a'_{ip} = a_{ip} - s(a_{iq} + \tau a_{ip}) \tag{6.27}$$
$$a'_{iq} = a_{iq} + s(a_{ip} - \tau a_{iq}) \tag{6.28}$$
$$a'_{pp} = a_{pp} - ta_{pq} \tag{6.29}$$
$$a'_{qq} = a_{qq} + ta_{pq} \tag{6.30}$$
$$a'_{pq} = 0 \tag{6.31}$$

mit

$$c = \frac{1}{\sqrt{t^2+1}}, \tag{6.32}$$

$$s = tc, \tag{6.33}$$

$$\tau = \frac{s}{1+c}. \tag{6.34}$$

Durch einen solchen Schritt wird S reduziert gemäß

$$S' = S - 2|a_{pq}|^2. \tag{6.35}$$

Solange also noch Nebendiagonalelemente da sind, wird S echt kleiner, was die Konvergenz des Verfahrens beweist. Die aufeinander folgenden Schritte sind nicht unabhängig in dem Sinne, daß $a_{pq} = 0$ beim Wegtransformieren anderer Nebendiagonalelemente wieder zerstört wird. Somit handelt es sich um ein echtes Iterationsverfahren, für das in [3] quadratische Konvergenz zitiert wird. Um die Iteration durchzuführen, muß noch festgelegt werden, wie wir die Ebenen pq gewählt werden. Es ist klar, daß alle vorkommen müssen, damit alle Nebendiagonalelemente eliminiert werden können. Standard ist die zyklische Jacobi–Methode mit $pq = 12, 13, \ldots, 1n, 23{,}24, \ldots$.

6.3 Transformation auf Tridiagonalform

Im Verfahren des letzten Abschnitts wurden Nebendiagonalelemente von A wegtransformiert. Da diese bei den nachfolgenden Schritten nicht Null bleiben, braucht man viele Schritte um iterativ die Matrix zu diagonalisieren. Wir hatten schon gesagt, daß es keine Verfahren mit endlicher Schrittzahl geben wird. In diesem Abschnitt werden wir sehen, daß man aber A mit endlicher Schrittzahl in eine (symmetrische) Tridiagonalmatrix T transformieren kann. So entsteht

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & 0 & \cdots & 0 \\ t_{21} & t_{22} & t_{23} & 0 & 0 & \cdots & 0 \\ 0 & t_{32} & t_{33} & t_{34} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & & \cdots & 0 & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & & \cdots & & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}, \tag{6.36}$$

wo also gilt: $t_{ij} = 0$ wenn $|i-j| > 1$ und $t_{ij} = t_{ji}$, da die Symmetrie erhalten bleibt. Diese Matrix muß dann noch immer iterativ diagonalisiert werden,

wofür es aber Verfahren gibt, die wesentlich schneller konvergieren als Jacobi bei der ursprünglichen Matrix.

Zu diesem Zweck bieten sich die im vorigen Kapitel eingeführten Householder Reflexionen an. Sie liefern orthogonale[9] Matrizen P_i, die, als Ähnlichkeitstransformationen angewandt, besonders effektiv zur Tridiagonalform führen. Der erste Schritt ist

$$A' = P_1 A P_1 \tag{6.37}$$

und dabei wird P_1 so konstruiert, daß $a'_{31} = \ldots = a'_{n1} = 0$ entsteht und damit wegen Symmetrie auch $a'_{13} = \ldots = a'_{1n} = 0$. Dies läßt sich erreichen mit

$$\begin{aligned} P_1 &= 1 - \frac{uu^T}{H} && (6.38)\\ u^T &= (0, a_{21} + \sigma_1, a_{31}, \cdots, a_{n1}) && (6.39)\\ \sigma_1^2 &= \sum_{i=2}^{n} (a_{i1})^2 && (6.40)\\ \mathrm{sgn}(\sigma_1) &= \mathrm{sgn}(a_{21}) && (6.41)\\ H &= \frac{1}{2}|u|^2 && (6.42)\\ &= \sigma_1(\sigma_1 + a_{21}). && (6.43) \end{aligned}$$

Das Element a_{11} bleibt hier ungeändert, $P_{ij} = \delta_{ij}$ wenn $i = 1$ oder $j = 1$, da sonst die Faktoren P von links und rechts miteinander auf der Diagonalen interferieren würden. Nach diesem Schritt haben wir

$$A' = \begin{pmatrix} a_{11} & -\sigma_1 & 0 & \ldots & 0 \\ -\sigma_1 & a'_{22} & a'_{23} & \cdots & \cdots \\ 0 & a'_{32} & a'_{33} & \cdots & \cdots \\ 0 & \vdots & \vdots & \vdots & \vdots \end{pmatrix}. \tag{6.44}$$

So kann man fortfahren, bis die modifizierte Matrix A Tridiagonalform hat. Im k-ten Schritt gilt

$$\begin{aligned} u^T &= (0, \cdots, 0, a_{k+1,k} + \sigma_k, a_{k+2,k}, \cdots, a_{nk}) && (6.45)\\ \sigma_k^2 &= \sum_{i=k+1}^{n} (a_{ik})^2 && (6.46)\\ \mathrm{sgn}(\sigma_k) &= \mathrm{sgn}(a_{k+1,k}), && (6.47) \end{aligned}$$

[9] Wir bleiben hier bei der reellen Form, obwohl alles leicht komplex geschrieben werden kann.

und hierbei werden die ersten $k-1$ Zeilen und Spalten sowie das aktuelle Element a_{kk} nicht verändert. Die jeweilige Modifikation von A kann man wie folgt effektiv durchführen. Nach Berechnung von σ, u, H bilden wir

$$p = \frac{Au}{H} \tag{6.48}$$

$$q = p - u\frac{u^T p}{2H} \tag{6.49}$$

$$A' = A - uq^T - qu^T. \tag{6.50}$$

Will man am Ende neben Eigenwerten auch Eigenvektoren ausgeben, so muß man das Produkt $P_1 P_2 \cdots P_{n-2}$ aufmultiplizieren und als orthogonale Matrix speichern.

Das folgende Demoprogramm in MATLAB führt die Formeln vor:

```
%
% file house.m
%
% demoprogramm fuer Householder Reduktion
%
n=5;
disp('Ausgangsmatrix:');
a=rand(n,n);                % Zufallsmatrix
a = a + a.'                 % symmetrisch
%
O=eye(n);                   % Einheitsmatrix
% Reduktionsloop:
for k=1:n-2,
  fprintf('Schritt: %i',k)
  sigma=norm(a(k+1:n,k));
  if a(k+1,k) < 0, sigma = -sigma; end;       %Vorzeichen fixiert
  u = [zeros(k,1) ; a(k+1:n,k)]; u(k+1)=u(k+1)+sigma;
  H = 0.5*norm(u)^2;
  p = a*u/H;
  q = p - u*((u.'*p)/(2*H));
  O = O -O*u*u.'/H;      % Transformation
  a = a - u*q.' - q*u.'  % modifiziertes A
end
disp('Transformation:');
```

```
0
```

Sein Ouput ist

```
Ausgangsmatrix:

a =

    1.9003    0.9932    1.2223    0.8917    0.9492
    0.9932    0.9129    0.8104    1.7569    0.7976
    1.2223    0.8104    1.8436    1.6551    0.9894
    0.8917    1.7569    1.6551    0.8205    0.9035
    0.9492    0.7976    0.9894    0.9035    0.2778

Schritt: 1
a =

    1.9003   -2.0437    0.0000    0.0000    0.0000
   -2.0437    4.4473   -0.0570   -0.7092    0.5329
    0.0000   -0.0570    0.5728   -0.0098    0.1672
    0.0000   -0.7092   -0.0098   -0.9323   -0.2693
    0.0000    0.5329    0.1672   -0.2693   -0.2329

Schritt: 2
a =

    1.9003   -2.0437    0.0000    0.0000    0.0000
   -2.0437    4.4473    0.8889         0    0.0000
    0.0000    0.8889   -0.4310    0.2064    0.3475
    0.0000    0.0000    0.2064   -0.0440   -0.6799
    0.0000    0.0000    0.3475   -0.6799   -0.1174

Schritt: 3
a =

    1.9003   -2.0437    0.0000    0.0000    0.0000
   -2.0437    4.4473    0.8889    0.0000    0.0000
    0.0000    0.8889   -0.4310   -0.4041    0.0000
    0.0000    0.0000   -0.4041   -0.6954   -0.2929
```

```
    0.0000    0.0000    0.0000   -0.2929    0.5340

Transformation:

O =

    1.0000         0         0         0         0
         0   -0.4860    0.1080    0.6938   -0.5204
         0   -0.5981   -0.0207    0.1713    0.7826
         0   -0.4363   -0.7661   -0.3880   -0.2688
         0   -0.4645    0.6333   -0.5821   -0.2108
```

6.4 Eigenwerte von tridiagonalen Matrizen, QL–Methode

Die betrachtete reelle symmetrische Matrix A können wir zerlegen

$$A = QL, \tag{6.51}$$

wobei Q orthogonal ist und L eine untere Dreiecksmatrix. Dies ist wieder möglich durch Householder–Reflexionen[10]. Nun bildet man das modifizierte A' als

$$A' = LQ = Q^T AQ, \tag{6.52}$$

das also zu A orthogonal ähnlich ist und dieselben Eigenwerte hat. Es gibt einen Satz [3], der besagt, daß die Iteration dieses Schrittes A in Dreiecksform bringt mit den Eigenwerten auf der Diagonalen[11]. Direkt auf das voll besetzte A angewandt ist diese Methode mit $\mathrm{O}(n^3)$ Operationen pro Iteration aber nicht effektiv, wohl aber für tridiagonale, wo man mit $\mathrm{O}(n)$ auskommen kann. Dabei ist wichtig, daß sich auch noch zeigen läßt, daß A' wieder tridiagonal ist, wenn dies für A gilt. Dann werden die garantierten Nullen natürlich nicht berechnet. Im tridiagonalen Fall ist es Standard, Q aus Rotationen wie beim Jacobi–Verfahren aufzubauen. Man wählt in jeder Iteration die Folge von Ebenen $12, 23, \ldots, n{-}1n$, um die Elemente $a_{12}, a_{23}, \ldots, a_{n-1n}$ zu annihilieren. Damit wird die tridiagonale Matrix symmetrisch dreieckig, also diagonal. Aus der hierbei entstandenen Ähnlichkeitstransformation, zusammen mit der vom Householder–Schritt, kann man auch die Eigenvektoren gewinnen.

[10] Hier muß man zuerst führende Nullen in der letzten Spalte von A erzeugen etc.

[11] Im Fall entarteter $|\lambda_i|$ gibt es für volle Matrizen noch Komplikationen die für tridiagonale aber keine Rolle spielen.

Über die Konvergenz wird in [3] gesagt, daß im s-ten Schritt in der Matrix $A^{(s)}$ die Elemente oberhalb der Diagonalen gegen Null gehen wie

$$a_{ij}^{(s)} \propto \left(\frac{\lambda_i}{\lambda_j}\right)^s, \tag{6.53}$$

wobei die Eigenwerte der Größe nach geordnet sind, so daß $|\lambda_i| < |\lambda_j|$ gilt[12]. Man kann diese Konvergenz u. U. verbessern durch eine Verschiebung der Eigenwerte $A \to A + k1$.

6.5 Eigenwertbestimmung in MATLAB

Es gibt in MATLAB wieder ein fertiges Programm mit Namen **eig** zur Bestimmung von Eigenwerten und Eigenvektoren:

```
>> help eig

 EIG    Eigenvalues and eigenvectors.
        EIG(X) is a vector containing the eigenvalues of a square
        matrix X.

        [V,D] = EIG(X) produces a diagonal matrix D of
        eigenvalues and a full matrix V whose columns are the
        corresponding eigenvectors so that X*V = V*D.

        [V,D] = EIG(X,'nobalance') performs the computation with
        balancing disabled, which sometimes gives more accurate results
        for certain problems with unusual scaling.

        Generalized eigenvalues and eigenvectors.

        EIG(A,B) is a vector containing the generalized eigenvalues
        of square matrices A and B.

        [V,D] = EIG(A,B) produces a diagonal matrix D of general-
        ized eigenvalues and a full matrix V whose columns are the
        corresponding eigenvectors so that A*V = B*V*D.
```

[12] Für $\lambda_i = \lambda_j$ läßt sich zeigen, daß $a_{ij} = 0$ schon nach dem Tridiagonalisieren gilt.

Mit der im vorletzten Abschnitt erzeugten Matrix `a` erhält man:

```
>> [v,d]=eig(a)

v =

   -0.0910   -0.8505    0.0930   -0.1934    0.4716
   -0.6303    0.2677   -0.5804   -0.1450    0.4162
    0.7505    0.2141   -0.3001   -0.1574    0.5255
   -0.1761    0.3973    0.7513   -0.1443    0.4753
   -0.0172   -0.0366   -0.0052    0.9466    0.3199


d =

    0.6038         0         0         0         0
         0    0.9042         0         0         0
         0         0   -1.0939         0         0
         0         0         0   -0.3406         0
         0         0         0         0    5.6816

>> for k=1:n, delta(k)=norm(a*v(:,k) - d(k,k)*v(:,k)); end
>> delta

delta =

   1.0e-14 *

    0.1027    0.0749    0.0835    0.1911    0.3447

>> diary off
```

6.6 Singular Value Decomposition

Die Singular Value Decomposition (SVD) ist ein Verfahren zur Lösung des linearen Gleichungssystems $Ax = b$. Es ist von Interesse, wenn es lineare Abhängigkeiten zwischen Zeilen oder Spalten gibt, also für $m \times n$–Matrizen A mit $m \neq n$ oder für quadratische singuläre Probleme. Für letzteren Fall

wollen wir uns hier interessieren. Eigentlich gehört dieser Abschnitt ins vorige Kapitel, aber weil das Verfahren wesentlich auf der Bestimmung eines Eigenwertspektrums beruht, wird es erst hier besprochen.

In der SVD hat man

$$A = UWV^T \tag{6.54}$$

wobei U und V orthogonal sind und W diagonal ist mit Diagonalelementen $w_i \geq 0$. Mit Hilfe der Spaltenvektoren $u^{(i)}$, $v^{(i)}$ von U bzw. V gilt auch

$$A = \sum_{i=1}^{n} u^{(i)} w_i \, v^{(i)T}. \tag{6.55}$$

Mathematische Sätze [3] garantieren, daß es diese Zerlegung immer gibt, auch im singulären Fall. Sie ist nahezu eindeutig. Sind alle w_i verschieden, so bleibt nur ein gemeinsamer Vorzeichenwechsel bei $u^{(i)}$, $v^{(i)}$ und gemeinsame Permutationen von $u^{(i)}$, $v^{(i)}$ und w_i, über die man durch Ordnen der w_i verfügen kann. Sind mehrere w_i gleich, so kann man in den zugehörigen Unterräumen die Basen der $u^{(i)}$, $v^{(i)}$ rotieren.

Für den Fall, daß A regulär ist, sind alle $w_i > 0$ und das Inverse ist gegeben durch

$$A^{-1} = VW^{-1}U^T \tag{6.56}$$

mit trivialem W^{-1}. Im singulären Fall sei angenommen, es verschwinden[13] die letzten k Diagonalelemente $w_n, w_{n-1}, \ldots w_{n-k+1}$. Dann sind die zugehörigen Vektoren $v^{(i)}$ eine Basis des Nullraumes der durch A gegebenen linearen Abbildung. Die ersten $n-k$ Vektoren $u^{(i)}$ bilden eine Basis des vom Bild von A aufgespannten Raumes. Das lineare System $Ax = b$ hat offenbar nur eine Lösung, wenn b im Bild von A liegt, und diese ist dann um beliebige Vektoren aus dem Nullraum unbestimmt.

Die Durchführung der SVD ist kompliziert und soll hier nicht dargestellt werden. Programme in Fortran und C gibt es in [3], und auch MATLAB stellt eine (binär implementierte) Routine **svd** bereit:

```
>> A=rand(4,4);
>> A(1,:)=A(2,:)*pi

A =
```

[13] In der Praxis wird man hier wegen Rundungsfehlern eine numerische Schranke setzen.

```
    1.8503    0.2889    2.8599    2.3125
    0.5890    0.0920    0.9103    0.7361
    0.9304    0.6539    0.7622    0.3282
    0.8462    0.4160    0.2625    0.6326

>> [U,W,V] = svd(A)

U =

    0.8955    0.3255    0.0048    0.3033
    0.2851    0.1036    0.0015   -0.9529
    0.2662   -0.7227   -0.6379    0.0000
    0.2142   -0.6008    0.7701    0.0000


W =

    4.5971         0         0         0
         0    0.8922         0         0
         0         0    0.4125         0
         0         0         0    0.0000


V =

    0.4903   -0.5800    0.1650   -0.6293
    0.1192   -0.6937   -0.2308    0.6718
    0.6699    0.3550   -0.6516    0.0239
    0.5446    0.2373    0.7035    0.3901



>> A=rand(4,4);
>> A(1,:)=A(2,:)*pi

A =

    1.8503    0.2889    2.8599    2.3125
```

```
    0.5890    0.0920    0.9103    0.7361
    0.9304    0.6539    0.7622    0.3282
    0.8462    0.4160    0.2625    0.6326

>> [U,W,V] = svd(A)

U =

    0.8955    0.3255    0.0048    0.3033
    0.2851    0.1036    0.0015   -0.9529
    0.2662   -0.7227   -0.6379    0.0000
    0.2142   -0.6008    0.7701    0.0000


W =

    4.5971         0         0         0
         0    0.8922         0         0
         0         0    0.4125         0
         0         0         0    0.0000


V =

    0.4903   -0.5800    0.1650   -0.6293
    0.1192   -0.6937   -0.2308    0.6718
    0.6699    0.3550   -0.6516    0.0239
    0.5446    0.2373    0.7035    0.3901
```

7 Numerische Integration

Ziel dieses Kapitels ist es, einfache Verfahren zur numerischen Berechnung von bestimmten Integralen

$$I = \int_a^b dx f(x) \tag{7.1}$$

vorzustellen. Wir beschränken uns dabei auf den eindimensionalen Fall: mehrdimensionale Integrale zerlegt man meistens in eindimensionale nach dem Muster

$$\int dxdy f(x,y) = \int_{x_1}^{x_2} dx \left(\int_{y_1(x)}^{y_2(x)} dy f(x,y) \right) \tag{7.2}$$

und wendet ein eindimensionales Verfahren für die innere wie auch für die äußere Integration an.

7.1 Interpolationspolynome

Grundlage der numerischen Integration einer Funktion $f(x)$ ist in den meisten Fällen die Idee, die Funktion durch ein Polynom zu nähern und dieses dann exakt zu integrieren. Man wählt also einen Satz von $n+1$ Stützstellen $\{x_i, i = 0 \ldots n\}$, wertet die Funktion dort aus und legt ein Polynom n–ten Grades durch die Funktionswerte $f_i \equiv f(x_i)$. Das läßt sich (für nicht zu große n) von Hand durchführen, im Hinblick auf eine spätere Anwendung ist es aber eleganter, sich auf die Lagrange–Interpolationsformel zu berufen. Dazu bildet man hilfsweise die Polynome

$$l_i(x) = \prod_{j(\neq i)} \frac{x - x_j}{x_i - x_j} \quad \Rightarrow \quad l_i(x_j) = \delta_{ij} \tag{7.3}$$

und kann dann sofort hinschreiben:

$$f(x) \approx p_n(x) = \sum_i l_i(x) f_i. \tag{7.4}$$

Das Integral soll nun approximiert werden durch

$$\int_a^b dx f(x) \approx \sum_i w_i f_i \tag{7.5}$$

mit

$$w_i = \int_a^b dx\, l_i(x). \tag{7.6}$$

Wenn $f(x)$ selbst ein Polynom vom Grad $\leq n$ ist, dann wird es nach Konstruktion durch (7.4) exakt integriert. Zum Beweis überlegt man sich, daß in diesem Falle $f(x) - p_n(x)$ ein Polynom vom Grad $\leq n$ ist, das $n+1$ verschiedene Nullstellen x_i hat und deshalb identisch verschwinden muß. Wie sieht es mit der Genauigkeit der Approximation im allgemeinen Fall aus? Denkt man sich $f(x)$ in $x \in [a,b]$ nach Potenzen von $(x-a)$ entwickelt, dann ergibt die Restglied–Formel der Taylorreihe

$$f(x) = \sum_{k=0}^{n} \frac{(x-a)^k}{k!} f^{(k)}(a) + R_n(x) \tag{7.7}$$

$$R_n(x) = \frac{(x-a)^{n+1}}{(n+1)!} f^{(n+1)}(\xi) \quad , \quad \xi \in [a,b] \tag{7.8}$$

zusammen mit der Schranke $|f^{(n+1)}(\xi)| \leq M^{(n+1)}$ eine Fehlerabschätzung

$$|\int_a^b dx\, R_n(x)| \leq \int_a^b dx \frac{(x-a)^{n+1}}{(n+1)!} M^{(n+1)} = \frac{(b-a)^{n+2}}{(n+2)!} M^{(n+1)}. \tag{7.9}$$

7.2 Trapez– und Simpsonregel

In diesem Abschnitt nehmen wir äquidistante Stützstellen der Schrittweite h an:

$$x_i = x_0 + ih \quad i = 0 \ldots n \tag{7.10}$$

und bilden das Integral von $a = x_0$ bis $b = x_n$.

Im einfachsten Fall einer linearen Interpolation ($n = 1$) zwischen den Endpunkten findet man $w_0 = w_1 = h/2$ und damit die bekannte *Trapezregel*

$$\int_{x_0}^{x_1} dx f(x) = h \left[\frac{1}{2} f_0 + \frac{1}{2} f_1\right] + O(h^3). \tag{7.11}$$

Die Fehlerabschätzung folgt aus Glg.(7.9) mit $(b-a) = h$.

Die nächste Möglichkeit ist $n = 2$, hier erhält man nach kurzer Rechnung:

$$w_0 = w_2 = \frac{h}{3} \tag{7.12}$$

$$w_1 = \frac{4}{3} h \tag{7.13}$$

und damit die *Simpsonregel*

$$\int_{x_0}^{x_2} dx f(x) = h \left[\frac{1}{3} f_0 + \frac{4}{3} f_1 + \frac{1}{3} f_2\right] + O(h^5). \tag{7.14}$$

Der Fehlerterm ist hier eine Ordnung höher als eigentlich zu erwarten war, weil die Symmetrie der Stützstellen und Gewichte dazu führt, daß auch x^3 noch exakt integriert wird (s. Übungen). Deshalb kann man hier Glg.(7.9) mit $n = 3$ und $(b - a) = 2h$ anwenden.

Dieses Spiel kann man offenbar beliebig fortsetzen, es entstehen die sogenannten *Newton–Cotes*–Formeln. Sie sind aber mehr von historischem Interesse, weil man tatsächlich mit der Trapezregel als 'Baustein' einfacher Algorithmen gut auskommt. Zur Beschleunigung der Konvergenz sind die unten behandelten Gauß–Formeln auch viel wirksamer.

7.3 Wiederholte Trapez– und Simpsonregel

Die einfache Trapezregel (7.11) eignet sich gut als Baustein für ein Integrationsverfahren, bei dem die gewünschte Genauigkeit durch Verfeinerung der Intervallteilung angestrebt wird.

Wenn man das Integrationsintervall in N gleiche Teile zerlegt, in jedem die Trapezregel (7.11) anwendet und alles addiert, bekommt man sofort die *wiederholte* Trapezregel (extended trapezoidal rule)

$$T_N = h\left[\frac{1}{2}f_0 + f_1 + f_2 + \ldots + f_{N-1} + \frac{1}{2}f_N\right]. \tag{7.15}$$

mit $h = (b-a)/N$. Da jeder der N Schritte einen Fehler $O(h^3)$ oder $O(N^{-3})$ mit sich bringt, folgt die Abschätzung des Gesamtfehlers

$$I = T_N + O(N^{-2}). \tag{7.16}$$

Man wird also durch fortgesetztes Halbieren der Teilintervalle der Reihe nach T_1, T_2, T_4, T_8 usw. berechnen und abbrechen, wenn die gewünschte Genauigkeit erreicht ist. Dabei kann man bei jedem Halbierungsschritt die zuvor berechneten Funktionswerte weiterverwenden und braucht jeweils nur die Werte an den neuen Teilpunkten zu berechnen.

Man kann aber noch weiter gehen und die Folge der T_N zu folgendem Trick benutzen: da der Fehler von T_{2N} gerade 1/4 des Fehlers von T_N ist, kann sich in der Kombination $4T_{2N} - T_N$ der führende Fehlerterm wegheben, und

$$S_{2N} = \frac{4}{3}T_{2N} - \frac{1}{3}T_N \tag{7.17}$$

sollte eine genaueres Ergebnis geben. Man erhofft Konvergenz mit N^{-3}, aber es kommt noch besser: wenn man (7.17) ausschreibt, ergibt sich

$$S_{2N} = h\left[\frac{1}{3}f_0 + \frac{4}{3}f_1 + \frac{2}{3}f_2 + \frac{4}{3}f_3 + \ldots + \frac{2}{3}f_{2N-2} + \frac{4}{3}f_{2N-1} + \frac{1}{3}f_{2N}\right] \quad (7.18)$$

mit $h = (b-a)/(2N)$. Das ist gerade die *wiederholte* Simpson–Regel (7.14), die je Teilintervall eine Genauigkeit $O(h^5)$ aufweist, also nach Summation immer noch eine Konvergenz der Ordnung N^{-4} erreicht:

$$I = S_{2N} + O(N^{-4}). \quad (7.19)$$

So entsteht ein einfaches und zugleich effektives Integrationsverfahren: man berechnet die Folge der Trapez–Näherungen $T_N, N = 1, 2, 4, 8, \ldots$, bildet die Simpson–Ausdrücke (7.17) und beobachtet deren Konvergenz.

Das folgende MATLAB –Programm zeigt, wie einfach die schrittweise Berechnung der Trapez–Näherungen T_N implementiert werden kann:

```
%
% trapez.m
%
% Numerische Integration von func(x) ueber [a,b]
% Berechnung von Trapez-Naeherungen
%     fuer n=0: Initialisierung von T_1
%          n>0: T_N -> T_2N , wobei 2N = 2^n
%
function [T2N] = trapez(func,a,b,TN,n);

if n < 0,
      error('negatives n in trapez');
elseif n == 0,                % nur ein Intervall
      T2N = .5*(b-a)*(feval(func,a) + feval(func,b));
else
      h = (b-a)/2^n;          % neue Schrittweite
      T2N = 0;
      for x=a+h:2*h:b,        % Summe ueber neue (innere) Punkte
            T2N = T2N + feval(func,x);
      end
      T2N = h*T2N + .5*TN;    % neue Trapez-Naeherung
end
```

Wenn diese Funktion mit $n = 0$ gerufen wird, gibt sie T_1 zurück, bei $n \geq 1$ dagegen wird für $2N = 2^n$ Intervalle der Wert von T_{2N} berechnet, wobei T_N verwendet und die Funktion nur an den neuen Teilpunkten aufgerufen wird.

Bislang haben wir nur Integrationsformeln diskutiert, die den Integranden auch an den Endpunkten des Intervalls auswerten (sog. *geschlossene* Verfahren). Das ist manchmal unbequem: man hat es überraschend oft mit Fällen wie

$$\int_0^b dx \frac{\sin x}{x} \tag{7.20}$$

zu tun, wo der Integrand bei $x = 0$ zwar mathematisch wohldefiniert ist, aber beim Programmieren besondere Vorsicht verlangt. Da wäre es einfacher, wenn alle Stützstellen *innerhalb* des Intervalls lägen. Die Bausteine für geeignete sog. *offene* oder *halboffene* Formeln findet man in [3]. Wir wollen sie hier nicht näher diskutieren, denn die nun folgenden Gauß–Formeln sind ohnehin vom offenen Typ.

7.4 Gauß'sche Integralformeln

Nach einer Idee von Gauß kann man Integrationsformeln konstruieren, die mit nicht–äquidistanten Stützstellen arbeiten, aber dafür mit der Zahl der Funktionsaufrufe wesentlich schneller konvergieren können. Um das zu erklären, muß zunächst an einige Eigenschaften orthogonaler Polynome, insbesondere der Legendre–Polynome erinnert werden.

Wir betrachten hier der Einfachheit halber das Integrationsintervall $[-1, 1]$, auf das man $[a, b]$ durch eine lineare Variablentransformation immer abbilden kann. Die Legendre–Polynome entstehen durch Orthogonalisierung der Potenzen $x^0, x^1, x^2, x^3 \ldots$ über $[-1, 1]$ und erfüllen (in konventioneller Normierung) die Orthogonalitätsrelationen

$$\int_{-1}^{1} dx P_m(x) P_n(x) = \frac{2}{2n+1} \delta_{mn}. \tag{7.21}$$

$P_n(x)$ ist (un-)gerade für (un-)gerades n. Explizit hat man $P_0 = 1$, $P_1 = x$, $P_2 = (3x^2 - 1)/2$ usw.

Wir wählen nun ein n und als Stützstellen der Integration die Nullstellen von $P_n(x)$:

$$P_n(x_i) = 0, \quad i = 1 \ldots n. \tag{7.22}$$

Wenn man damit eine Integrationsformel konstruiert wie in (7.3) bis (7.6), dann findet man das bemerkenswerte Resultat, daß damit

$$\int_{-1}^{1} dx f(x) = \sum_{i=1}^{n} w_i f(x_i) \tag{7.23}$$

exakt gilt, wenn nur $f(x)$ ein Polynom vom Grad $\leq 2n-1$ ist. Zum Beweis bemerkt man, daß man $f(x)$ darstellen kann als

$$f(x) = q(x)P_n(x) + r(x), \tag{7.24}$$

wobei $q(x)$ und $r(x)$ Polynome vom Grad $\leq n-1$ sind — das geht durch 'Polynom–Division' $f(x)/P_n(x)$ 'mit Rest'. Nun ist aber

$$\int_{-1}^{1} dx\, q(x)P_n(x) = 0,$$

weil man $q(x)$ als Linearkombination von $P_0(x), \ldots, P_{n-1}(x)$ ausdrücken und die Orthogonalität (7.21) benutzen kann. Somit hat man

$$\begin{aligned} \int_{-1}^{1} dx f(x) &= \int_{-1}^{1} dx\, r(x) \\ &= \sum_{i=1}^{n} w_i r(x_i) \\ &= \sum_{i=1}^{n} w_i f(x_i), \end{aligned} \tag{7.25}$$

weil die polynomische Integrationsvorschrift für $r(x)$ exakt ist und weil $P_n(x_i) = 0$.

Eine allgemeine (differenzierbare ...) Funktion $f(x)$ wird also durch die Gauß–Vorschrift integriert bis auf einen Fehlerterm, der vom Restglied $\sim x^{2n}$ herstammt. In einem Intervall der Länge h ergibt das einen Fehler $\sim h^{2n+1}$ und bei N Abschnitten mit $h = (b-a)/N$ summieren sich die Abweichungen zu einem Term $O(N^{-2n})$. Da Werte wie $n = 10$ kein Problem sind (s.u.), erwartet man unglaublich schnelle Konvergenz. Wie sieht die Praxis aus? Das hängt (wie könnte es anders sein) von der Funktion $f(x)$ ab: wenn sie oft differenzierbar ist und die höheren Ableitungen beschränkt sind, sieht man tatsächlich, daß die Ergebnisse der Gauß–Integration sehr schnell konvergieren. Wenn die Funktion aber eine (integrable) Singularität hat oder

ihre Ableitungen divergieren (das passiert oft am Rand des Integrationsgebiets), dann versagen die Fehlerschranken und eine Konvergenzbeschleunigung durch hohe (formale) Ordnung tritt nicht ein. Kurz gesagt: Integrationsformeln hoher Ordnung haben nur dann Sinn, wenn die Funktion hinreichend glatt ist, andernfalls kann man ebenso gut auf die Trapez– oder Simpsonregel zurückgreifen.

Woher bekommt man nun die Stützpunkte und Gewichte der Gauß–Formeln? Man kann sie sich ausrechnen, entweder durch numerische Bestimmung der Nullstellen von $P_n(x)$ oder eleganter durch Rekursionsverfahren, wie z.B. in [3] beschrieben. Man findet die Parameter aber auch in Büchern wie der 'Zahlenbibel' [5], z.B. für die 10–Punkt–Formel (nur die 5 positiven x_i und ihre Gewichte w_i sind aufgeführt, denn $P_{10}(x)$ ist gerade):

```
      x_i                    w_i
.1488743389816 31d0     .2955242247147 53d0
.4333953941292 47d0     .2692667193099 96d0
.6794095682990 24d0     .2190863625159 82d0
.8650633666889 85d0     .1494513491505 81d0
.9739065285171 72d0     .0666713443086 88d0
```

und so fort bis $n = 96$(!).

7.5 Adaptive Schrittweite

Man mag sich fragen, ob es immer geschickt ist, die Integration mit einem festen Satz von Stützstellen durchzuführen. Es könnte doch besser sein, die Teilung nur dort zu verfeinern, wo die Funktion stark veränderlich ist, und 'langweilige' Abschnitte mit großen Schritten zu durchqueren. Mit einer solchen *adaptiven Schrittweite* funktionieren tatsächlich die in MATLAB eingebauten Routinen `quad` und `quad8`. Wir wollen darauf hier aber nicht weiter eingehen und stattdessen auf ein verwandtes Problem verweisen: wenn man

$$y(x) = \int_a^x d\xi f(\xi) \tag{7.26}$$

definiert, dann findet man das gesuchte Integral als $I = y(b)$. Nun kann man y(x) als Lösung der Differentialgleichung $y'(x) = f(x)$ mit $y(a) = 0$ suchen, und das ist ein Spezialfall des Anfangswertproblems

$$y'(x) = f(x, y), \quad y(a) = 0, \tag{7.27}$$

für das es leistungsfähige Verfahren gibt, auch solche mit automatisch sich anpassender Schrittweite. Davon wird in einem späteren Kapitel noch die Rede sein.

8 Anfangswertprobleme

In diesem und in folgenden Abschnitten wollen wir mechanische Systeme betrachten. Dabei ist die Dynamik in Form gewöhnlicher (nicht partieller) Differentialgleichungen (DGL) wie der Newton–Gleichung gegeben. Solche Naturgesetze erlauben es, bei gegebenen Orten und Geschwindigkeiten zu einer Startzeit die künftige (und auch vorherige) Bahn zu berechnen. Bei komplizierten Kräften, z. B. Mehrkörperproblemen oder Reibung, sind hier numerische Simulationen die einzige Möglichkeit, an näherungsunabhängige Lösungen zu kommen.

8.1 Einfaches Beispiel

Die Trajektorie eines Balls $\vec{r}(t)$ ist bestimmt durch Anfangsort $\vec{r}_0$ und Anfangsgeschwindigkeit $\vec{v}_0$ und Lösen der Newtongleichungen

$$m\frac{d\vec{v}}{dt} = \vec{F}(\vec{v}) - mg\hat{y}; \quad \frac{d\vec{r}}{dt} = \vec{v}. \tag{8.1}$$

Hier sind $\vec{r} = (x, y)$, $\vec{v} = (v_x, v_y)$ Ort und Geschwindigkeit (Funktionen der Zeit t), und es werden nur Bewegungen in der Ebene betrachtet. Die Gravitation wirkt in negativer y-Richtung, $\hat{y}$ ist der y-Einheitsvektor. $\vec{F}$ ist die Reibungskraft.

Die Lösung mit $\vec{r}_0 = 0, \vec{v}_0 = v_0(\cos(\theta), \sin(\theta))$ zur Anfangszeit $t = 0$ ohne Reibungskraft ist bekanntlich eine Parabel und lautet

$$x(t) = v_0 \cos(\theta) t \tag{8.2}$$

$$y(t) = v_0 \sin(\theta) t - \frac{1}{2} g t^2. \tag{8.3}$$

Nach der Flugzeit $t_{\rm fl}$,

$$t_{\rm fl} = \frac{2v_0}{g} \sin(\theta) \tag{8.4}$$

trifft der Ball wieder auf dem Boden ($y = 0$) auf. Die maximale Höhe beträgt

$$y_{\max} = \frac{v_0^2}{2g} \sin^2(\theta) \tag{8.5}$$

und die Weite

$$x_{\max} = \frac{v_0^2}{g} \sin(2\theta) \tag{8.6}$$

Für den Flug eines Baseballs mit Reibung ist das phänomenologische Reibungsgesetz

$$\vec{F}(\vec{v}) = -\frac{1}{2} C_d \rho \pi R^2 |\vec{v}| \vec{v} \tag{8.7}$$

bekannt. Brauchbare Parameter sind

$m = 0.145$ kg (Masse des Balls)

$R = 3.7$ cm (Radius)

$\rho = 1.2$ kg/m^3 (Dichte der Luft)

$C_d = 0.35$ (typischer Reibungskoeffizient für einen Baseball)

8.2 Euler–Methode

Die Euler Methode ist der naivste denkbare Versuch, die obigen Gleichungen zu lösen. Wir betrachten ihre allgemeine Struktur

$$\frac{d\vec{v}}{dt} = \vec{a}(\vec{r}, \vec{v}); \quad \frac{d\vec{r}}{dt} = \vec{v}. \tag{8.8}$$

Nähern wir nun die Ableitung durch die einfache asymmetrische Formel

$$\frac{d\vec{v}}{dt} = \frac{\vec{v}(t+\tau) - \vec{v}(t)}{\tau}, \tag{8.9}$$

und analog für $\vec{r}$, so erhalten wir eine Vorwärtsrekursion

$$\vec{v}(t+\tau) = \vec{v}(t) + \tau \vec{a}(\vec{r}(t), \vec{v}(t)) + \mathrm{O}(\tau^2) \tag{8.10}$$

$$\vec{r}(t+\tau) = \vec{r}(t) + \tau \vec{v}(t) + \mathrm{O}(\tau^2). \tag{8.11}$$

Wenn man sich auf die in Vielfachen von τ diskretisierten Zeiten beschränkt und vom Fehler absieht, haben wir ein explizites Schema, um alle Größen von t nach $t+\tau$ zu evolvieren. Durch Iteration kann man also ausgehend von den Anfangswerten die Lösung bekommen.

Bevor wir ein Programm untersuchen, wollen wir versuchen, die Größenordnung des Fehlers vorherzusehen. Bei jedem einzelnen Schritt ensteht ein (lokaler) Fehler $\mathrm{O}(\tau^2)$. Um zu einer Zeit t zu evolvieren, brauchen wir t/τ Schritte. Im ungünstigen Fall, mit dem man rechnen muß, addieren sich die

Fehler zu einem globalen Fehler $O(t\tau)$, und dieser wird also bei festem t linear mit τ kleiner[14]. Der relative Fehler δ (Flugzeit = Wert $\times(1\pm\delta)$) ist immer dimensionslos. Zwei charakteristische Zeiten gibt es in unserem Problem, $t_{\rm fl}$ und v_0/g, die in der gleichen Größenordnung liegen, solange θ nicht sehr klein ist. Dann wäre also aus Dimensionsgründen zu erwarten, daß $\delta \propto \tau/t_{\rm fl}$.

Wir untersuchen nun das folgende Programm

```
% ball - Programm fuer fliegende Baelle
% Euler Methode, ohne Reibung
clear;  help ball;
%
r = [0 0]; % ev. auch eingeben
v0    = input(' Anfangsgeschwindigkeit v0 (m/sek) ? ');
theta = input(' Winkel theta(Grad) ? ')*pi/180; % gleich ins Bogenmass
tau   = input(' Zeitschritt tau(sek) ? ');
%
v = v0*[cos(theta) sin(theta)]; %v_0
g = 9.81; % g in m/sek^2
a = [0 -g]; % in diesem Fall konstante Beschleunigung
%
maxstep = 1000; % Maximale Schrittzahl
%
% Hauptschleife:
%
for istep=1:maxstep
     xplot(istep) = r(1);   % Werte behalten zum Plotten am Ende
     yplot(istep) = r(2);
     r = r + tau*v;                 % Euler Schritt
     v = v + tau*a;                 % Euler Schritt
%
  if( r(2) < 0 )  % loop abbrechen, Aufschlag
    break;
  end
end
%
xplot(istep+1) = r(1);  yplot(istep+1) = r(2); % letzter Punkt
```

[14] Dies gilt, solange nicht akkumulierte Fehler zu einem qualitativ anderen Verhalten führen, z. B. bei chaotischen Systemen.

```
fprintf(' Reichweite %g meter\n',r(1))
fprintf(' Flugzeit %g sekunden\n',istep*tau)
%
%  Plot:
%
% Bodenlinie:
xground = [0 xplot(istep+1)]; yground = [0 0];
% Graph der Trajektorie:
plot(xplot,yplot,'+',xground,yground,'-');
xlabel('Weite (m)')
ylabel('Hoehe (m)')
title('Fliegender Ball')
%
% Vgl. exakte Loesung:
%
xmaxx = v0^2*sin(2*theta)/g;
tfl   = 2*v0*sin(theta)/g;
fprintf(' rel. Abweichung  Flugzeit: %g \n',(istep*tau-tfl)/tfl)
fprintf(' rel. Abweichung  Weite   : %g \n',(r(1)-xmaxx)/xmaxx)
```

Dieses Programm sollte mittlerweile leicht zu analysieren sein. Ein einfacher run sieht so aus:

```
>> ball

  ball - Programm fuer fliegende Baelle
  Euler Methode, ohne Reibung

 Anfangsgeschwindigkeit v0 (m/sek) ? 10
 Winkel theta(Grad) ? 25
 Zeitschritt tau(sek) ? 0.05
 Reichweite 8.60992 meter
 Flugzeit 0.95 sekunden
 rel. Abweichung  Flugzeit: 0.102591
 rel. Abweichung  Weite   : 0.102591
>> print -depsc ball.eps
>> quit
```

mit dem zugehörigen Bild in Abb.8. Mit $t/\tau = 19$ ergab sich also ein Fehler

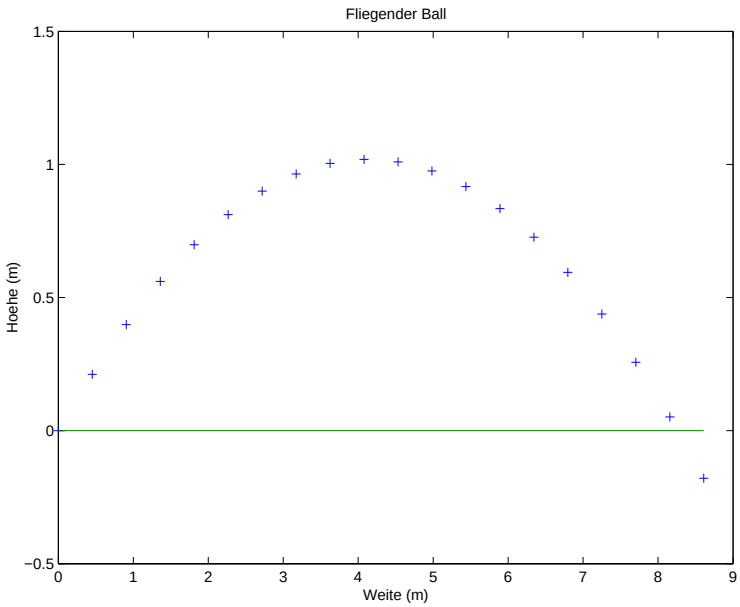

Abbildung 8: Ball Trajektorie

von 10%. Mit der halber Schrittweite ergaben sich 4.5% in guter Übereinstimmung mit unserer Schätzung. Weitere Experimente mit dem fliegenden Ball, insbesondere auch mit Reibung, sollen den Übungen vorbehalten bleiben.

8.3 Standard–Notation

Bevor wir effektivere Integrationsverfahren studieren, wollen wir eine Standardnotation für mechanische Systeme mit beliebig vielen Freiheitsgraden einführen. DGL mit höheren Ableitungen sollen stets auf solche 1. Ordnung zurückgeführt werden durch Einführen der Ableitungen als zusätzliche Komponenten. Beispiel [15]:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = F(t, x) \tag{8.12}$$

ist äquivalent zu

$$\frac{\mathrm{d}\vec{y}}{\mathrm{d}t} = \vec{f}(t, \vec{y}) \tag{8.13}$$

[15] x und y haben hier nichts mit den Komponenten von $\vec{r}$ im letzten Abschnitt zu tun

mit

$$\vec{f} = (y_2, F(t, y_1)). \tag{8.14}$$

Insbesondere erfüllt y_1 dieselbe DGL wie x, und y_2 hat zu jeder Zeit denselben Wert wie $\mathrm{d}x/\mathrm{d}t$. Wir wollen daher (8.13) in diesem Abschnitt als die Standardform des zu lösenden Problems nehmen, wobei die Anzahl der Komponenten in $\vec{y}$ beliebig ist. Die Hamilton'sche Form der Bewegungsgleichungen in der Physik führt direkt zu dieser Form, wobei dann die $\{q_i\}, \{p_i\}, i = 1, \ldots, n$ zu einem $2n$-komponentigen $\vec{y}$ zu verbinden sind.

8.4 Runge-Kutta–Formeln zweiter Ordnung

Im Rest dieses Kapitels wollen wir Integrationsverfahren nach Runge-Kutta (RK) kennenlernen. Sie stellen ein Standardverfahren zur Integration gewöhnlicher DGL dar und sind wesentlich effektiver und genauer als die Euler–Methode, die nur als Einführung diente. Es gibt RK verschiedener Ordnung, und auch diese sind noch nicht eindeutig. Weiter gibt es nicht *das* optimale Verfahren für alle Probleme, sondern es ist gut, mit verschiedenen zu experimentieren. Eine empfehlenswerte Referenz ist wiederum [3].

Das Euler–Verfahren zur Evolution einer Lösung um einen Zeitschritt $\vec{y}(t) \to \vec{y}(t+\tau)$ für kleines τ lautete

$$\vec{y}(t+\tau) = \vec{y}(t) + \tau \vec{f}(t, \vec{y}(t)) + \mathrm{O}(\tau^2). \tag{8.15}$$

Es wird Verfahren als Verfahren 1. Ordnung bezeichnet, da, wie wir gesehen haben, die lokalen Fehler 2. Ordnung in jedem Schritt nach T/τ Schritten, die man von $t = 0$ nach $t = T$ braucht, zu einem (globalen) Gesamtfehler $\mathrm{O}(\tau/T)$ führen.

Diese Ordnung wird bei RK erhöht, d. h. der Fehler für gleiche τ verkleinert. Dabei macht man sich zunutze, daß sich die rechte Seite in (8.15) auf viele Weisen schreiben läßt, die sich nur in $\mathrm{O}(\tau^2)$ unterscheiden. Diese kombiniert man so, daß sich $\mathrm{O}(\tau^2)$–Terme kompensieren! Dabei ist zu beachten, daß numerisch nur $\vec{f}$ benutzt werden soll, nicht aber seine Ableitungen, die i. a. nicht zur Verfügung stehen. Sie werden zwar in der mathematischen Diskussion vorkommen, nicht aber im zu programmierenden Algorithmus.

Im einfachsten Fall setzen wir an:

$$\vec{y}(t+\tau) = \vec{y}(t) + \tau \left[w_1 \vec{f}(t, \vec{y}(t)) + w_2 \vec{f}(t + \alpha\tau, \vec{y}_*) \right] \tag{8.16}$$

mit

$$\vec{y}_* = \vec{y}(t) + \beta\tau\vec{f}(t, \vec{y}(t)). \tag{8.17}$$

Damit wollen wir die Taylorentwicklung bis zur 2. Ordnung reproduzieren,

$$\vec{y}(t+\tau) - \vec{y}(t) \stackrel{!}{=} \tau\vec{f} + \frac{\tau^2}{2}\frac{\mathrm{d}\vec{f}}{\mathrm{d}t} + \mathrm{O}(\tau^3), \tag{8.18}$$

wobei hier die weggelassenen Argumente immer $\vec{f}(t, \vec{y}(t))$ usw. sind, und

$$\frac{\mathrm{d}\vec{f}}{\mathrm{d}t} = \frac{\partial\vec{f}}{\partial t} + \sum_i \frac{\partial\vec{f}}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}t} = \frac{\partial\vec{f}}{\partial t} + \sum_i \frac{\partial\vec{f}}{\partial y_i} f_i \tag{8.19}$$

gilt. Durch Entwickeln in (8.16) ergibt sich

$$\vec{y}(t+\tau) - \vec{y}(t) = \tau(w_1+w_2)\vec{f} + w_2\alpha\tau^2\frac{\partial\vec{f}}{\partial t} + w_2\tau\sum_i (y_*-y)_i\frac{\partial\vec{f}}{\partial y_i} + \mathrm{O}(\tau^3) \tag{8.20}$$

Durch Koeffizientenvergleich finden wir Gleichheit bis auf $\mathrm{O}(\tau^3)$, wenn

$$w_1 + w_2 = 1 \tag{8.21}$$

$$w_2\alpha = w_2\beta = \frac{1}{2}. \tag{8.22}$$

Es gibt also eine einparametrige Schar von solchen Algorithmen 2. Ordnung. Einfache Wahlen sind

$$w_1 = 0,\ w_2 = 1,\ \alpha = \beta = \frac{1}{2}. \tag{8.23}$$

oder

$$w_1 = w_2 = \frac{1}{2},\ \alpha = \beta = 1. \tag{8.24}$$

Keine dieser oder der anderen Wahlen hat einen beweisbaren Vorteil unabhängig vom zu lösenden Problem.

8.5 Runge-Kutta–Formel 3. Ordnung

Die folgenden Größen $\vec{k}_i, i = 1, 2, 3$, die man rekursiv, also jede aus den vorhergehenden, berechnen kann, sind offensichtlich alle gleich $\vec{y}(t+\tau) -$

$\vec{y}(t) + \mathrm{O}(\tau^2)$:

$$\begin{aligned}
\vec{k}_1 &= \tau \vec{f}\,(t, \vec{y}(t)) && (8.25)\\
\vec{k}_2 &= \tau \vec{f}\left(t + \tau, \vec{y} + \vec{k}_1\right) && (8.26)\\
\vec{k}_3 &= \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{y} + \frac{1}{4}(\vec{k}_1 + \vec{k}_2)\right). && (8.27)
\end{aligned}$$

Offenbar entspricht die Variante (8.24) der 2. Ordnung–Formel nun der Evolution

$$\vec{y}(t+\tau) = \vec{y}(t) + \frac{1}{2}[\vec{k}_1 + \vec{k}_2] + \mathrm{O}(\tau^3) \qquad (8.28)$$

Indem man eine Ordnung weiter entwickelt, läßt sich zeigen:

$$\vec{y}(t+\tau) = \vec{y}(t) + \frac{1}{6}[\vec{k}_1 + \vec{k}_2 + 4\vec{k}_3] + \mathrm{O}(\tau^4). \qquad (8.29)$$

8.6 Runge-Kutta–Formel 4. Ordnung

Nun bilden wir $\vec{k}_i$, $i = 1, 2, 3, 4$,

$$\begin{aligned}
\vec{k}_1 &= \tau \vec{f}\,(t, \vec{y}(t)) && (8.30)\\
\vec{k}_2 &= \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{y} + \frac{1}{2}\vec{k}_1\right) && (8.31)\\
\vec{k}_3 &= \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{y} + \frac{1}{2}\vec{k}_2\right) && (8.32)\\
\vec{k}_4 &= \tau \vec{f}\left(t + \tau, \vec{y} + \vec{k}_3\right) && (8.33)
\end{aligned}$$

Durch Entwickeln, was allerdings schon etwas länglich ist, läßt sich zeigen:

$$\vec{y}(t+\tau) = \vec{y}(t) + \frac{1}{6}[\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4] + \mathrm{O}(\tau^5) \qquad (8.34)$$

Dies ist, obwohl natürlich wiederum nicht eindeutig, eine häufige Wahl. Man kann Formeln von noch höheren Ordnungen konstruieren. Ob dies profitabel ist, hängt wieder vom Problem und von der erstrebten Genauigkeit ab. Bei glatten Lösungen $\vec{y}(t)$ kann man dann größere Schritte τ wählen. Dafür benötigt man pro Schritt mehr Funktionsaufrufe, die mehr oder weniger teuer sein können.

8.7 Fehlerkontrolle und Schrittweitensteuerung

Neben der Ordnung des Fehlers in τ, den man per Konstruktion weiß, hätte man gerne eine Information über den konkreten Fehler in einer speziellen Anwendung. Man kann dann die gewählte Schrittweite automatisch anpassen, um eine vorgegebene Genauigkeit der Einzelschritte zu erreichen. Sie muß dann sicher klein werden in Gebieten, wo $\vec{y}$ stark variiert, und kann wachsen, wenn langweilige, flache Gebiete der Funktion durchquert werden.

Ein einfaches Verfahren zur Fehlerabschätzung besteht darin, einen Schritt der Weite τ mit einer gewählten RK-Formel einmal direkt zu machen:

$$\vec{y}(t) \stackrel{\tau}{\to} \vec{Y}$$

und dann das gleiche mit zwei Schritten der Weite $\tau/2$:

$$\vec{y}(t) \quad \stackrel{\tau/2}{\to} \quad \stackrel{\tau/2}{\to} \quad \vec{y}(t+\tau).$$

Dieser als genauer erwartete Wert wird als Lösung genommen. Die Schrittweite ist also eigentlich $\tau/2$. Beim Verfahren n-ter Ordnung ist der lokale Fehler $\mathrm{O}(\tau^{n+1})$. Dann ist der Fehler einmal $c\tau^{n+1}$ und zum anderen $2c(\tau/2)^{n+1}$ mit einer Konstanten c. Damit kann man c eliminieren und erhält als Fehlerabschätzung beim aktuellen Schritt

$$\delta = \max_i \frac{|Y_i - y_i(t+\tau)|}{|y_i|(2^n - 1)}. \tag{8.35}$$

Um sicher zu sein, wird hier der größte relative Fehler von allen Komponenten von $\vec{y}$ genommen [16].

Das bei dieser Schätzung gefundene δ kann zu groß sein (τ muß kleiner werden) oder auch unnötig klein (τ darf größer werden). Unter der hier schon gemachten Annahme, daß $\delta \propto \tau^{n+1}$, können wir auch eine neue Schrittweite τ' angeben, die zu einer angestrebten relativen *lokalen* Genauigkeit ϵ gehört:

$$\tau' = \tau \left(\frac{\epsilon}{\delta}\right)^{\frac{1}{n+1}}. \tag{8.36}$$

War der Fehler kleiner als ϵ, dann könnte ein sicherheitshalber um einen Faktor von z. B. 0.9 verkleinerter Wert für den nächsten Schritt verwendet

[16] Hier gibt es ein Problem, wenn ein $|y_i| \simeq 0$ wird. Dann muß man sich anders eine charakteristische Skala für diese Komponente aus dem Problem überlegen.

werden. War δ zu groß, so muß der Schritt mit kleinerem τ wiederholt werden. Auf diese Art bekommt man natürlich keine regelmäßige Zeitdiskretisierung, sondern $\vec{y}(t)$ in unregelmäßigen Abständen, die bei größerer verlangter Genauigkeit und bei starker Variation dichter werden.

Nachdem man c weiß, könnte man auch noch um den Fehlerterm der Ordnung $n+1$ jeweils korrigieren, was man lokale Extrapolation nennt. Man kann dies tun oder lassen. Für diese weitere Korrektur hat man jedenfalls keine Fehlerschätzung, man kann also diesen eventuellen Genauigkeitszuwachs nicht kontrollieren und nicht zitieren. Auch darf man nicht vergessen, daß das Verhalten der Fehler mit τ^{n+1} nur der führende Term ist und nicht exakt gilt. Man sollte die Formel nicht überstrapazieren, sondern eher wieder als Abschätzung der Größenordnung behandeln. Bei sehr kleinen τ muß man auch die Rundungsfehler im Auge behalten, die dann das Verhalten überdecken wie bei der numerischen Differentiation.

Eine weitere Möglichkeit zur Fehlerkontrolle besteht darin, RK–Schritte n-ter *und* $n+1$-ter Ordnung zu machen und die Differenz zur Fehlerschätzung heranzuziehen. Nach dem oben zur lokalen Extrapolation gesagten ist dies dann als Verfahren n-ter Ordnung mit Fehlerkontrolle zu verkaufen. Besonders günstig ist es, wenn man für beide Schritte ganz oder weitgehend dieselben $\vec{k}_i$ verwenden kann.

8.8 Mehr MATLAB

In diesem Abschnitt ist beim Implementieren der Reibungskraft die euklidische Norm des Vektors $\vec{v}$ gefragt. Dazu gibt es in MATLAB die Funktion **norm**:

```
>> help norm

 NORM   Matrix or vector norm.
    For matrices...
      NORM(X) is the largest singular value of X, max(svd(X)).
      NORM(X,2) is the same as NORM(X).
      NORM(X,1) is the 1-norm of X, the largest column sum,
                      = max(sum(abs(X))).
      NORM(X,inf) is the infinity norm of X, the largest row sum,
                      = max(sum(abs(X'))).
      NORM(X,'fro') is the Frobenius norm, sqrt(sum(diag(X'*X))).
```

```
      NORM(X,P) is available for matrix X only if P is 1, 2, inf or 'fro'.

    For vectors...
      NORM(V,P) = sum(abs(V).^P)^(1/P).
      NORM(V) = norm(V,2).
      NORM(V,inf) = max(abs(V)).
      NORM(V,-inf) = min(abs(V)).

    See also COND, RCOND, CONDEST, NORMEST.

>> quit
```

Für Vektoren leistet also `norm(v)` ohne weitere Argumente das Gewünschte.

Bei den Plots zu den Übungsaufgaben ist es praktisch, wenn man im **title** numerische Werte wiedergeben kann, so daß man noch weiß, welches Bild zu welchen Werten gehört. **title** nimmt aber nur Text (Strings wie `'blablabla'`) als Argument. Man kann aber Zahlen zu Strings konvertieren und Strings zu Vektoren zusammensetzen:

```
>> x=1.5;
>> a=[ ' x ist gleich ' num2str(x) ' oder?'];
>> a

a =

 x ist gleich 1.5 oder?
```

Eine solche Größe kann dann in **title(a)** vorkommen. **num2str** konvertiert *numbers to strings*.

9 Programme für die Runge–Kutta–Integration

Im ersten Teil dieses Kapitels werden wir ein Runge-Kutta-Programm mit automatischer Anpassung der Schrittweite analysieren, das früher der MATLAB -Biliothek angehörte, aber in der Version 5 des MATLAB -Paketes durch ein moderneres, aber leider weniger transparentes Programm ersetzt wurde. Inzwischen gibt es unter `rk23` nur noch ein "frontend" für das neuere Programme, das heisst der folgende output ist nicht mehr unter dem aktuellen MATLAB zu bekommen.

Zum Zwecke der Fehlerkontrolle und Anpassung der Schrittweite benutzt dieses Programm die Runge-Kutta–Formeln 2. und 3. Ordnung. Bei der Analyse dieses Programms, das von MATLAB –Profis geschrieben wurde, werden wir einige neue MATLAB –Kommandos kennenlernen.

Im zweiten Teil dieses Kapitels wird anhand von einem einfachen Beispiel erklärt, wie man die beiden MATLAB -Programme **ode23** und **ode45** zur Integration von Differentialgleichungen verwenden kann. Ähnlich wie das Programm, das wir im ersten Teil besprechen, benutzen diese Programme die Runge-Kutta–Formeln 2. und 3. bzw. 4. und 5. Ordnung.

9.1 Das Programm rk23

Wir haben das Programm mit Zeilennummern versehen, um auf die verschiedenen Programmteile später im Text Bezug nehmen zu können.

```
1    function [tout, yout] = rk23(ypfun, t0, tfinal, y0, tol, trace)
2    %RK23     Solve differential equations, low order method.
3    %  RK23 integrates a system of ordinary differential equations using
4    %  2nd and 3rd order Runge-Kutta formulas.
5    %  [T,Y] = RK23('yprime', T0, Tfinal, Y0) integrates the system of
6    %  ordinary differential equations described by the M-file YPRIME.M,
7    %  over the interval T0 to Tfinal, with initial conditions Y0.
8    %  [T,Y] = RK23('yprime', T0, Tfinal, Y0, TOL, 1) uses tolerance TOL
9    %  and displays status while the integration proceeds.
10   %
11   %  INPUT:
12   %  ypfun - String containing name of user-supplied problem description.
13   %          Call: yp = fun(t,y) where ypfun = 'fun'.
14   %          t   - Time (scalar).
```

```
15  %          y   - Solution column-vector.
16  %          yp  - Returned derivative column-vector; yp(i) = dy(i)/dt.
17  %  t0    - Initial value of t.
18  %  tfinal- Final value of t.
19  %  y0    - Initial value column-vector.
20  %  tol   - The desired accuracy. (Default: tol = 1.e-3).
21  %  trace - If nonzero, each step is printed. (Default: trace = 0).
22  %
23  %  OUTPUT:
24  %  T  - Returned integration time points (column-vector).
25  %  Y  - Returned solution, one solution row-vector per tout-value.
26  %
27  %  The result can be displayed by: plot(tout, yout).
28  %
30
31  %  C.B. Moler, 3-25-87, 8-26-91, 9-08-92.
32  %  Copyright (c) 1984-94 by The MathWorks, Inc.
33
34  % Initialization
35  pow = 1/3;
36  if nargin < 5, tol = 1.e-3; end
37  if nargin < 6, trace = 0; end
38
39  t = t0;
40  hmax = (tfinal - t)/16;
41  h = hmax/8;
42  y = y0(:);
43  chunk = 128;
44  tout = zeros(chunk,1);
45  yout = zeros(chunk,length(y));
46  k = 1;
47  tout(k) = t;
48  yout(k,:) = y.';
49
50  if trace
51     clc, t, h, y
52  end
53
```

```
54   % The main loop
55
56   while (t < tfinal) & (t + h > t)
57      if t + h > tfinal, h = tfinal - t; end
58
59      % Compute the slopes
60      s1 = feval(ypfun, t, y); s1 = s1(:);
61      s2 = feval(ypfun, t+h, y+h*s1); s2 = s2(:);
62      s3 = feval(ypfun, t+h/2, y+h*(s1+s2)/4); s3 = s3(:);
63
64      % Estimate the error and the acceptable error
65      delta = norm(h*(s1 - 2*s3 + s2)/3,'inf');
66      tau = tol*max(norm(y,'inf'),1.0);
67
68      % Update the solution only if the error is acceptable
69      if delta <= tau
70         t = t + h;
71         y = y + h*(s1 + 4*s3 + s2)/6;
72         k = k+1;
73         if k > length(tout)
74            tout = [tout; zeros(chunk,1)];
75            yout = [yout; zeros(chunk,length(y))];
76         end
77         tout(k) = t;
78         yout(k,:) = y.';
79      end
80      if trace
81         home, t, h, y
82      end
83
84      % Update the step size
85      if delta ~= 0.0
86         h = min(hmax, 0.9*h*(tau/delta)^pow);
87      end
88   end
89
90   if (t < tfinal)
91      disp('Singularity likely.')
```

```
92      t
93   end
94
95   tout = tout(1:k);
96   yout = yout(1:k,:);
```

In den help-Zeilen bis 33 werden die Funktion und die Ein- und Ausgabe-Parameter beschrieben. Zeile 1 definiert wie immer die Funktion. Die Felder `tout` und `yout` werden ins rufende Programm zurückgegeben und müssen also am Ende dieser Routine Zuweisungen erhalten. Zeile 5 zeigt einen typischen Aufruf des Programms, wobei `yprime.m` eine Funktion definieren muß, die die rechte Seite der DGL liefert. Das ist genau die Vektorfunktion $\vec{f}(t, \vec{y})$, die im letzten Kapitel eingeführt wurde. Die Argumente müssen wie angegeben sein, weiteres muß ggf. als globale Variable zur Verfügung gestellt werden. Der Name (z. B. `'yprime'`) im Aufruf kann auch in einer String-Variablen wie z.B. `F='yprime'` gespeichert sein, s. Zeile 8. In Zeile 13 heißt die Beispielfunktion nun offenbar `fun` und deren Rückgabevariable nun `yprime`. Solche help-Zeilen entstehen, wenn verschiedene Leute zu verschiedenen Zeiten an einem Programm herumeditieren. Die Parameter `t0, tfinal, y0` bedürfen keiner weiteren Erklärung. Daneben gibt es noch zwei *optionale* Parameter, die man übergeben *kann*. `tol` setzt die gewüschte Genauigkeit zur Schrittweitenwahl, und ein von Null verschiedener Wert für `trace` druckt Zeit, Schrittweite und die Komponenten des Vektors $\vec{y}(t)$ bei jedem Schritt aus. (Vorsicht, wird leicht zuviel!).

Als Ergebnis bekommt man den Spaltenvektor `t` mit den Zeiten und die Matrix `y`, in deren Zeilen die Komponenten von $\vec{y}$ zu den sukzessiven Zeiten abgespeichert sind.

In Zeile 36 sieht man, wie optionale Parameter erkannt werden. In jeder Funktion steht automatisch die Variable `nargin` zur Verfügung, die die Anzahl der übergebenen Argumente enthält (number of argument inputs). Es gibt auch `nargout`. Wenn die Anzahl der Parameter kleiner als 5 ist, wird die Genauigkeit **tol** gleich 0.001 gesetzt und der Ausdruck der Zeiten und Komponenten wird unterdrückt (**trace=0**).

Die Schrittweite heißt im Programm `h` (bei uns bisher τ), und in Zeile 40 wird `hmax` gleich **(tfinal-t)/16** gesetzt, und in Zeile 41 wird 1/8 dieses Werts für den ersten Versuch genommen. In Zeile 42 wird jener Trick verwendet, der es einem erlaubt, davon unabhängig zu werden, ob `y0` eine Zeile oder Spalte ist.

Die Zeilen 44, 45 erfordern eine genauere Erklärung. In MATLAB müssen Felder nicht vorher deklariert werden. Während der Integration mit rk23, erhält die Lösung `y` bei jedem Schritt eine neue Zeile, wird also dynamisch vergrößert. Ein Programm wird schneller, wenn größere Stücke eines Feldes *auf einmal* im Speicher abgelegt werden. Dies wird hier für jeweils `chunk=128` Schritte gemacht. Die Funktion **zeros(m,n)** liefert eine $m \times n$ Matrix mit Nullen, `length(y)` gibt für Vektoren deren Länge. Es wird hier also für die ersten 128 $\vec{y}(t)$ "Platz geschaffen", und als erstes in der Zeile 48 der Anfangswert `y` eingesetzt, der natürlich transponiert werden muß. Wenn die logische Variable `trace` gleich eins ist, wird der Anfangswert auf dem Bildschirm ausgegeben. Das MATLAB –Kommando **clc** säubert den Schirm.

Der Hauptloop beginnt in Zeile 56. Die **while**-Schleife wird durchlaufen, solange `t` noch kleiner als `tfinal` und solange Addition von `h` die Zeit `t` noch vergrößert (Rundungsfehler!). Zeile 57: sollte der geplante Schritt übers Ziel `tfinal` hinausgehen, dann wir er so weit verkleinert, daß `tfinal` genau erreicht wird.

In den Zeilen 60 - 62 werden die Hilfgrößen $\vec{k}_1, \vec{k}_2, \vec{k}_3$ ausgerechnet, die für die Runge-Kutta–Methode 3. Ordnung benötigt werden (s. Kapitel 9.5) (diese Hilfsgrößen werden hier als `s1, s2, s3` bezeichnet; der Faktor τ fehlt allerdings hier noch!). Das MATLAB –Kommando **feval** dient dazu, eine Funktion, deren Name als Stringkonstante gegeben ist, auszuwerten. Der absolute Fehler `delta` wird nun als Differenz der Runge-Kutta Formeln 3. und 2. Ordnung [17] geschätzt und das Maximum über alle Komponenten genommen. Der "Sollfehler" (`tau`) wird als Produkt aus `tol` und dem Maximum der größten Komponente oder 1 gebildet. Wenn `y` sehr klein ist, wirkt `tol` also als *absolute* Genauigkeit, sonst als relative Genauigkeit. Das ist nur sinnvoll, wenn natürliche Einheiten verwendet werden, wo alle Komponenten von $\vec{y}$ typisch von der Ordnung 1 sind.

Ist der Fehler akzeptabel, so wird der Schritt 3. Ordnung durchgeführt (Zeilen 70-72). In den Zeilen 73-76 wird abgefragt, ob noch genügend Speicherplatz für die Felder `tout` und `you` vorhanden ist und ggf. neu reserviert. In den Zeilen 77 und 78 wird schließlich der neue Wert in den Feldern `tout` und `you` abgespeichert. War der Fehler zu groß, so werden die letzten Schritte ausgelassen. In den Zeilen 85-87 wird `delta` für den nächsten Schritt angepaßt. Dabei wird angenommen, daß `delta` $\propto$ `h`^{3} ist. Der Vorfaktor dient dazu, häufige Schrittwiederholungen zu vermeiden. Wenn `tfinal`

[17] Bitte verifizieren!

nicht erreicht wird, weil `h` zu klein wird, so schließt das Programm auf eine Singularität und gibt die erreichte Zeit dazu aus.

Die Zeilen 95 und 96 bringen schließlich `tout, yout` auf die genaue Länge und es wird überflüssiger, bereits reservierter Speicherplatz eliminiert.

Dieses Programm sollte in allen Details genau verstanden sein, bevor man es benutzt.

9.2 Die MATLAB -Programme ode23 und ode45

MATLAB stellt uns mehrere Programme zur numerischen Lösung von Differentialgleichungen zur Verfügung. Das Programm **ode23** ist das Nachfolgeprogramm von rk23, das wir im letzten Kapitel vorgestellt haben. Auch in **ode23** werden die Runge-Kutta–Formeln 2. und 3. Ordnung zur Fehlerabschätzung und Anpassung der Schrittweite verwendet. In analoger Weise werden in **ode45** die Runge-Kutta–Formeln 4. und 5. Ordnung eingesetzt. Beide Programme gehören nicht zum binären Kern von MATLAB , sondern sind selbst in MATLAB geschrieben ist. Man kann sich diese Programme mit den Kommandos **type** und **dbtype** anschauen (Unterschied: **dbtype** versieht das Programm mit Zeilennummern). Im folgenden wollen wir anhand eines einfachen Beispiels erklären wie man die Programme benutzt. Ferner soll ein Vergleich der Programme durchgeführt werden. Informationen über die beiden Programme, kann man sich wieder mit dem **help**–Kommando beschaffen, z.B.

Wir wollen nun die Differentialgleichung

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix} \tag{9.1}$$

numerisch für die Anfgangsbedingung

$$y_1(t=0) = 0\,, \qquad y_2(t=0) = 1 \tag{9.2}$$

bestimmen. Die Differentialgleichung läßt sich natürlich auch analytisch lösen,

$$y_1^{\text{theo.}}(t) = \sin t;\,, \qquad y_2^{\text{theo.}}(t) = \cos t\,. \tag{9.3}$$

Wir sollten also einen Einheitskreis mit Mittelpunkt im Ursprung erhalten. Die obige Differentialgleichung soll von $t = 0$ bis $t = 40\pi$ mit dem MATLAB –Programm **ode23** integriert werden. Der Kreis soll also insgesamt 20 mal durchlaufen werden. Zur Integration verwenden wir das folgende einfache MATLAB – Programm:

```
% Test fuer ode23;
% verwendet Funktion ftest
t0 = 0;     % Startwert f"ur die Zeit
y0 = [0 1]; % Startwert f"ur y0
tfinal = 40.*pi;
tspan=[t0 tfinal];
[t,y] = ode23('ftest',tspan,y0);
dis = [sin(t) cos(t)]-y; % Diskrepanz zur exakten Loesung
plot(t,dis);
title('Diskrepanz zur Loesung')
pause
plot(y(:,1),y(:,2));
axis square
title('Loesung, Komponeten gegeneinander');
```

Die rechte Seite der Differentialgleichung wird hier wieder als Funktion (hier: **ftest**) an das Programm **ode23** übergeben,

```
function yprime = ftest(t,y)
% Primitiver Test fuer ode23 und ode45
% y: 2-komponentige Spalte
yprime = [y(2); -y(1)];
```

Das Programm **odetest** produziert zwei Bilder. In der Abb. 9 werden die Abweichungen $y_1(t) - y_1^{\text{theo.}}(t)$ und $y_2(t) - y_2^{\text{theo.}}(t)$ als Funktion von t dargestellt. Man erkennt, daß die maximale Abweichung während eines Umlaufs etwa linear mit t anwächst. In der Abb. 10 wurde die Lösung selbst dargestellt. Die Trajektorie scheint sich, verursacht durch die Diskretisierungsfehler, langsam auf den Ursprung zuzubewegen. Mittels des **size**–Kommandos kann man sich anschauen, wieviele Stützstellen vom Programm **ode23** gewählt wurden,

```
>> odetest


>> size(t)

ans =

   607     1
```

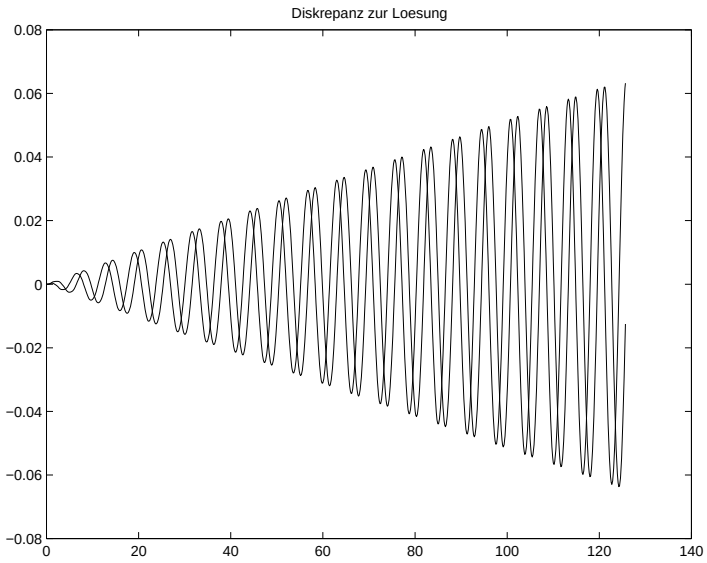

Abbildung 9: Absoluter Fehler bei `odetest/ode23`

.

```
>> size(y)

ans =

   607     2
```

Die entsprechenden Ergebnisse für das Programm **ode45** sind in den Abb. 11 und 12 dargestellt. Wie erwartet, sieht man, daß die Abweichungen deutlich kleiner sind als im Falle des Programms **ode23**.

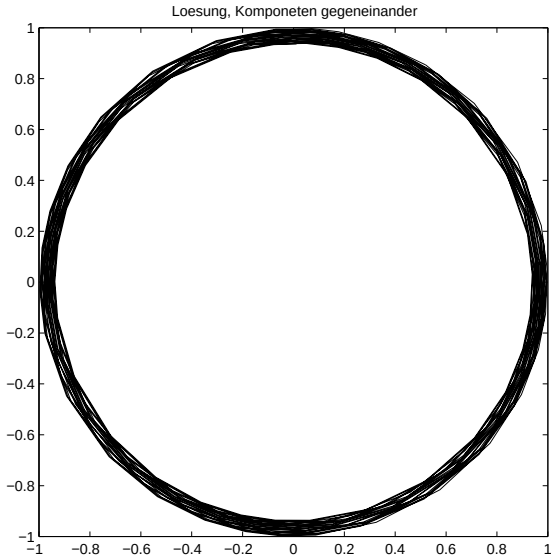

Abbildung 10: `y(2)` gegen `y(1)` bei `odetest/ode23`

.

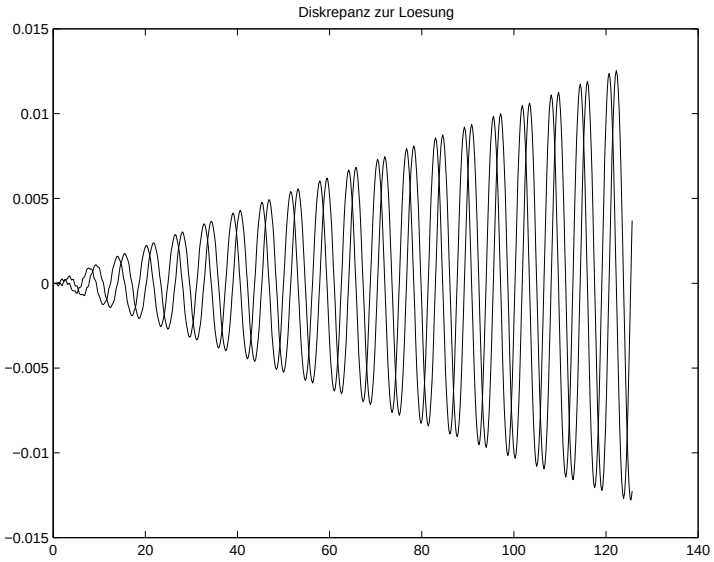

Abbildung 11: Absoluter Fehler bei `odetest/ode45`

.

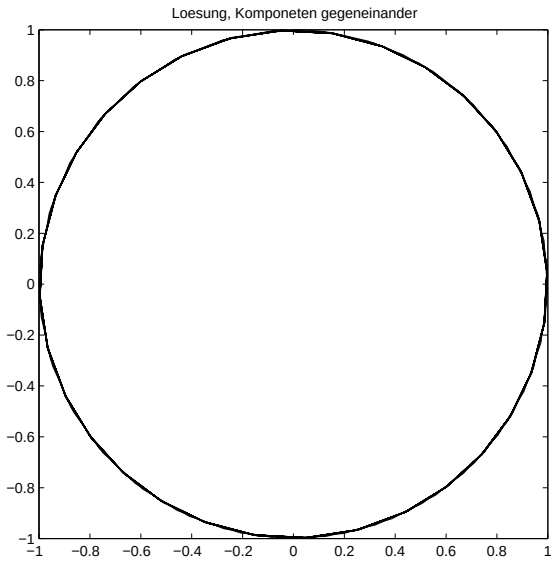

Abbildung 12: `y(2)` gegen `y(1)` bei `odetest/ode45`

.

10 Kepler–Probleme

10.1 Einheiten im Sonnensystem

Für die Betrachtung astronomischer Probleme im Sonnensystem wollen wir praktische Einheiten wählen, die zu nicht extremen Zahlen führen.

Als Längeneinheit nehmen wir die Astronomical Unit (AU), die gleich der großen Halbachse a der elliptischen Erdbahn um die Sonne ist. Da die Erdbahn fast ein Kreis ist (bis auf O(1%)), ist das in etwa der Radius.

$$1\mathrm{AU} = 1.496 \times 10^{11}\mathrm{m}. \tag{10.1}$$

Als Zeiteinheit nehmen wir die Umlaufzeit von einem Jahr, 1 yr. In diesen Einheiten wollen wir die im Gravitationsgesetz allgegenwärtige Konstante GM ausdrücken, wo G die Newtonsche Gravitationskonstante ($6.67 \times 10^{-11} \frac{\mathrm{m}^3}{\mathrm{kg\ s}^2}$) und M die Masse der Sonne ist ($M = 1.99 \times 10^{30}$ kg). Auf einer Kreisbahn mit Radius R gilt bekanntlich für einen Planeten der Masse m mit Umlauffrequenz ω das 3. Kepler'sche Gesetz in der Form

$$m\omega^2 R = \frac{GmM}{R^2} \tag{10.2}$$

Für die elliptische Bahn hat es die gleiche Form, wobei a an die Stelle von R treten muß. Damit gilt

$$GM = \omega^2 a^3 = 4\pi^2 \, \frac{\mathrm{AU}^3}{\mathrm{yr}^2} \tag{10.3}$$

10.2 Planetenbahn

In der für z. B. die Erde guten Näherung eines Planeten, der sehr viel leichter ist als die Sonne ($m_E/M \simeq 3.3 \times 10^{-6}$), nehmen wir die Sonne statisch im Koordinatenursprung. Dann ist die Bahngleichung der Erde

$$\ddot{\vec{r}} = -\frac{GM}{|\vec{r}|^3}\vec{r} \tag{10.4}$$

mit $\ddot{\vec{r}} = \mathrm{d}^2\vec{r}/\mathrm{d}t^2$. Wegen des erhaltenen Drehimpulses

$$\vec{L} = m \, \vec{r} \times \vec{v} \tag{10.5}$$

können wir die Bahn in der xy–Ebene voraussetzen. Die Energie des Planeten ist

$$E = m \left(\frac{1}{2}\vec{v}^2 - \frac{GM}{|\vec{r}|} \right). \tag{10.6}$$

Wie aus der Mechanik bekannt, ist die Bahngleichung (10.4) geschlossen lösbar und liefert Kegelschnitte mit der Sonne in einem Brennpunkt. Für $E < 0$ sind es Ellipsen.

10.3 Mehrkörperprobleme

Nun betrachten wir noch die Bewegungsgleichung für die Bewegung von N Körpern mit Massen m_i und gegenseitiger Gravitationswechselwirkung,

$$m_i \, \ddot{\vec{r}}_i = -G m_i \sum_{i \neq j = 1}^{N} \frac{m_j}{|\vec{r}_i - \vec{r}_j|^3} \, (\vec{r}_i - \vec{r}_j), \quad i = 1 \ldots N. \tag{10.7}$$

Die Bewegung von 3 und mehr Körpern wird i. a. chaotisch, d. h. sie hängt extrem empfindlich von den Anfangsbedingungen ab. Die Integration ist aufwendig und langsam, wenn sich 2 Körper nahe kommen und erfordert spezielle Vorkehrungen wie regularisierende Koordinaten. Eine solche Aufgabe gibt es in [6], dies ist hier jedoch zu aufwendig.

10.4 Mehr MATLAB

Hier wollen wir unsere MATLAB Kenntnisse in dem Umfang erweitern, der zur Simulation einer Planetenbahn nützlich ist.

Zunächst müssen wir **plot** verfeinern. Bisher wurde der gezeigte Bereich immer automatisch so gewählt, daß alle Daten sichtbar waren. Man kann dies auch selbst in die Hand nehmen, z. B. wenn man auf einen Punkt, der weit weg liegt, lieber verzichtet, was aber MATLAB natürlich nicht weiß. Dazu dient `axis`. Z. B. machen die Kommandos `v=[-1 2 0.5 1]; axis(v)` unmittelbar hinter `plot(x,y)` einen Plot mit x-Bereich von -1 bis 2 und y-Bereich von 0.5 bis 1. `v=axis` ergibt den entsprechenden Bereichsvektor des aktuellen Plots. **axis** kennt auch string Argumente, `axis('square')` macht einen Plot quadratisch statt rechteckig. Zurück mit `axis('normal')`. Es gibt noch mehr Möglichkeiten, s. **help axis**.

Man kann Bilder zu einer $m \times n$ Matrix von Plots vereinigen, z. B. wenn man $x(t)$ und $y(t)$ zusammen darstellen will. Beispiel:

```
>> t=0:.1:2*pi;
>> x=sin(t);
>> y=cos(t);
>> subplot(2,2,1), plot(t,x)
>> subplot(2,2,3), plot(t,y)
>> subplot(2,2,2), plot(x,y)
```

liefert Abb. 13. Die beiden ersten Argumente von **subplot** sind m und n und das dritte die Plotnummer innerhalb der Matrix, wobei zeilenweise von links oben nach rechts unten durchgezählt wird.

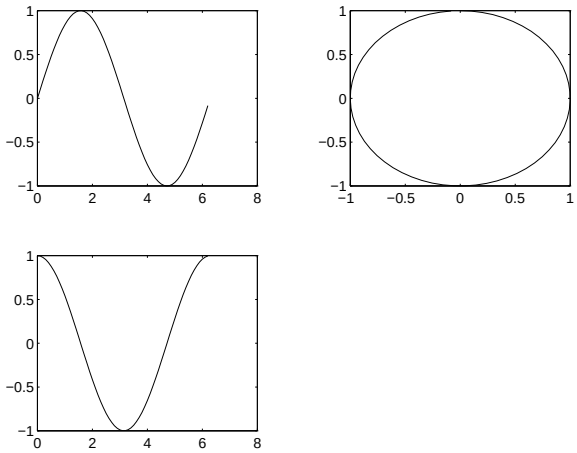

Abbildung 13: Anwendung von **subplot**

.

11 Elektrostatik

Viele Probleme der Elektrostatik, z. B. mit komplizierten Randbedingungen, müssen numerisch behandelt werden. Mathematisch geht es dabei um die Lösung der Laplace–und Poisson–Gleichungen, die Spezialfälle elliptischer partieller Differentialgleichungen (PDE) sind. Die Methoden, die in diesem Kapitel vorgestellt werden, sind aber auch in anderen Bereichen, wie z. B. auf die Lösung der zeitunabhängige Schrödinger Gleichung anwendbar. Die Klassifizierung der PDEs sei anhand des Falles mit zwei unabhängigen Variablen x, y angegeben. Die allgemeine Gleichung

$$a\frac{\partial^2 A}{\partial x^2} + b\frac{\partial^2 A}{\partial x \partial y} + c\frac{\partial^2 A}{\partial y^2} + d\frac{\partial A}{\partial x} + e\frac{\partial A}{\partial y} + f(A(x,y)) + g = 0 \qquad (11.1)$$

heißt hyperbolisch, falls $b^2 - 4ac > 0$, parabolisch, falls $b^2 - 4ac = 0$, und elliptisch, wenn $b^2 - 4ac < 0$. Letzteres ist also der Fall, wenn nur zweite Ableitungen mit gleichen Vorzeichen vorkommen. Die Benennung hat mit den Flächen zu tun, entlang derer sich Information ausbreiten kann. Damit sind dann auch die sinnvollen Anfangs bzw. Randbedingungen und damit die beschreibbare Physik verschieden. Die Natur scheint diese Gleichungen zu mögen, die Physik ist voll davon. Typische Beispiele: Wellengleichungen (hyperbolisch), Diffusion (parabolisch) und eben Poisson und Laplace.

11.1 Poisson–und Laplace–Gleichungen

Die Elektrostatik ist durch die Maxwellgleichungen [7, 8]

$$\vec{\nabla} \cdot \vec{E} = \frac{\rho}{\varepsilon_0}, \qquad (11.2)$$

$$\vec{\nabla} \times \vec{E} = 0 \qquad (11.3)$$

und Randbedingungen festgelegt mit den üblichen Bezeichnungen (in MKSA Einheiten). Mit dem Ansatz eines Potentials

$$\vec{E} = -\vec{\nabla}\Phi \qquad (11.4)$$

wird (11.3) gelöst, und wir haben die Poisson Gleichung

$$-\triangle\Phi = \frac{\rho}{\varepsilon_0} \qquad (11.5)$$

bzw. die Laplace–Gleichung

$$\triangle\Phi = 0 \tag{11.6}$$

im leeren Raum. In diesem Kurs wird es hauptsächlich darum gehen, die Laplace–Gleichung mit Randbedingungen zu lösen.

Ein einfaches Problem ist etwa auf einem Rechteck in der Ebene gegeben mit den Eckpunkten $\vec{r} = (0,0), (L_x,0), (0,L_y), (L_x,L_y)$. Wenn man dort Φ auf den Kanten vorgibt, dann legt die Lösung von (11.6) das Potential überall auf dem Rechteck fest. Der Separationsansatz

$$\Phi(x,y) = X(x)Y(y) \tag{11.7}$$

kann hier zur Lösung benutzt werden. Wir betrachten hier und im folgenden der Einfachheit halber ein ebenes Problem, daher kein Faktor $Z(z)$. Das bedeutet, daß alle Felder von z unabhängig angenommen werden. Damit gibt (11.6)

$$\frac{1}{X}\frac{\mathrm{d}^2X}{\mathrm{d}x^2} = -\frac{1}{Y}\frac{\mathrm{d}^2Y}{\mathrm{d}y^2}. \tag{11.8}$$

Das Standardargument besagt, daß eine Funktion von x (linke Seite) nur identisch mit einer Funktion von y (reche Seite) sein kann, wenn beide konstant sind. Wir nennen die Konstante $-k^2$, wo k auch imaginär sein kann. Dann können die Lösungen angegeben werden mit

$$\begin{aligned} X(x) &= C_s\sin(kx) + C_c\cos(kx) \\ Y(y) &= C'_s\sinh(ky) + C'_c\cosh(ky). \end{aligned} \tag{11.9}$$

Die Lösung für gegebene Randbedingungen ist dann eine Überlagerung solcher $X(x)Y(y)$ mit festgelegten Koeffizienten. Gilt z. B. $\Phi(0,y) = \Phi(L_x,y) = 0$, so folgt gleich, daß nur Beiträge mit $C_c = 0$ passen, und daß $k = k_n = n\pi/L_x$ gelten muß mit ganzzahligem n. Man hat dann also

$$\Phi = \sum_{n=1}^{\infty}\sin(k_nx)\left[c_n\sinh(k_ny) + d_n\cosh(k_ny)\right] \tag{11.10}$$

mit Konstanten c_n, d_n, die dann durch Randbedingungen für $\Phi(x,0)$ und $\Phi(x,L_y)$ zu bestimmen sind.

11.2 Elektrostatische Energie

In der Elektrodynamik wird gezeigt, daß die Energieerhaltung gilt, wenn man dem elektrischen Feld die Energie U,

$$U = \frac{\varepsilon_0}{2} \int \mathrm{d}V \vec{E}^2 = \frac{\varepsilon_0}{2} \int \mathrm{d}V (\vec{\nabla}\Phi)^2 \tag{11.11}$$

zuschreibt, wo das Integral über das betrachtete Volumen V geht. So wie ein Teilchen im *statischen*, d. h. zeitunabhängigen, Fall am Potentialminimum ruht, so kann man auch das $\vec{E}$-Feld in diesem Fall finden, indem man das Minimum der Energie sucht. Dann muß U stationär sein unter Variationen $\Phi \to \Phi + \delta\Phi$ im Inneren von V. Am Rand ist Φ durch die Randbedingungen fixiert[18], und kann nicht variiert werden. Dann gilt

$$0 \stackrel{!}{=} \delta U = \int_V \mathrm{d}V (\vec{\nabla}\Phi) \cdot \vec{\nabla}\delta\Phi = -\int_V \mathrm{d}V \delta\Phi \triangle\Phi \tag{11.12}$$

Hier wurde partiell integriert, wobei wegen $\delta\Phi|_{\partial V} = 0$ kein Oberflächenterm auftritt. Da $\delta\Phi$ beliebig ist, folgt hieraus die Laplace–Gleichung im Inneren von V. Weil (11.11) ein positives quadratisches Funktional in Φ ist, ist das durch die Laplace–Gleichung gegebene Extremum ein Minimum. Dieser Zugang zur Laplace–Gleichung ist günstig für die Diskretisierung und spätere numerische Behandlung.

11.3 Diskretisierung der Laplace–Gleichung

Ein kontinuierliches Feld wie Φ und $\vec{E}$ entspricht selbst über einem endlichen Volumen unendlich vielen Parametern. Dies sieht man z. B. bei Entwicklung in einem (unendlichen) Satz von Basisfunktionen. Solche Probleme sind mit endlichen Computern im strikten Sinne nicht zu behandeln. Wir wissen aber, daß physikalische Felder stetig sind und Experimente endliche Fehler haben, so daß es genügt, Felder für endlich viele genügend dicht gelegene Punkte zu kennen. Das ist analog zu gewöhnlichen Differentialgleichungen, wo wir auch die Zeit diskretisiert haben.

Wir führen ein Rechteckgitter ein und beschränken uns darauf $\Phi_{i,j}$ zu berechnen, wobei die Verbindung durch

$$\Phi_{i,j} = \Phi(\vec{r}_{i,j}),\ \vec{r}_{i,j} = ([i-1]h_x, [j-1]h_y) \tag{11.13}$$

[18] hier werden Dirichlet Randbedingungen betrachtet

gegeben ist. h_x, h_y sind Schrittweiten und kontrollieren die Feinheit der Auflösung in $x-$ und $y-$Richtung. Um eine gute Näherung ans Kontinuum zu erhalten, müssen sie natürlich klein sein gegen physikalische Längen im Problem, z. B. Ausdehnung des Volumens und Längen, über die die gestellten Randbedingungen variieren.

Die einfachste Näherung für den in der Energie auftretenden Gradienten von Φ ist nun

$$\begin{aligned} \partial_x \Phi &\simeq \frac{\Phi_{i+1,j} - \Phi_{i,j}}{h_x}, \\ \partial_y \Phi &\simeq \frac{\Phi_{i,j+1} - \Phi_{i,j}}{h_y}. \end{aligned} \tag{11.14}$$

Damit folgt für die Energie[19]

$$U = \frac{\varepsilon_0}{2} h_x h_y \sum_{i,j} \left\{ \left(\frac{\Phi_{i+1,j} - \Phi_{i,j}}{h_x} \right)^2 + \left(\frac{\Phi_{i,j+1} - \Phi_{i,j}}{h_y} \right)^2 \right\}, \tag{11.15}$$

wobei wir das Integral durch eine Riemann Summe genähert haben[20]. Der Summationsbereich muß passend zur Geometrie und den Rändern gewählt werden. Als Beispiel zeigt Abb.14 den schon diskutierten rechteckigen Bereich der Größe $L_x \times L_y$. Die *inneren* Gitterpunkte sind dann aufzuzählen mit

$$i = 2, \ldots, N_x - 1,\ j = 2, \ldots, N_y - 1,\ L_x = (N_x - 1)h_x,\ L_y = (N_y - 1)h_y, \tag{11.16}$$

und die Ränder haben $i \in \{1, N_x\}$ oder $j \in \{1, N_y\}$, das sind $2(N_x + N_y - 2)$ Punkte. Die Summe in (11.15) geht natürlicherweise über nächste Nachbar Paare, jedes einmal, einschließlich solcher, die einen Punkt auf dem Rand haben. In unserem Beispiel ist also zu summieren über $i = 1, \ldots, N_x - 1$, $j = 1, \ldots, N_y - 1$.

Nun können wir das wohldefinierte Problem betrachten, $\Phi_{i,j}$ für die inneren Punkte so zu wählen, daß U minimal wird bei festgehaltenen Rand-$\Phi_{i,j}$. Aus der Variationsableitung wird in der diskretisierten Form eine gewöhnliche partielle Ableitung, und wir erhalten eine Gleichung für jedes *innere* Paar (i, j),

$$0 = \frac{1}{\varepsilon_0 h_x h_y} \frac{\partial U}{\partial \Phi_{i,j}} = \tag{11.17}$$

[19] eigentlich Energie pro Längeneinheit in z-Richtung

[20] vgl. Definition Integral als Limes solcher Summen

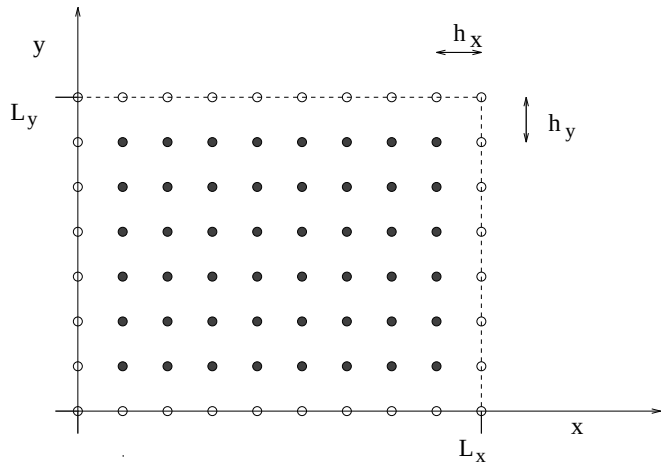

Abbildung 14: Beispiel für Gitterdiskretisierung

$$= \frac{1}{h_x^2}(2\Phi_{i,j} - \Phi_{i+1,j} - \Phi_{i-1,j}) + \frac{1}{h_y^2}(2\Phi_{i,j} - \Phi_{i,j+1} - \Phi_{i,j-1}) =: (-\triangle\Phi)_{i,j}.$$

Hier wurde die diskrete Version des Laplace–Operators definiert. Es ist leicht zu sehen, daß dies für glatte Funktionen und kleine h_x, h_y die Kontinuumsform nähert. Weiter zeigt Taylor Entwicklung, daß die Abweichung vom Kontinuum $\propto h_x^2, h_y^2$ geht (vgl. numerische Ableitung). In dieses System von $(N_x-2)(N_y-2)$ Gleichungen gehen auch die Randwerte (bis auf Eckpunkte) mit ein. Sie müssen schließlich die Lösung festlegen. Grundsätzlich handelt es sich um eine Gleichung vom linearen Typ

$$A\Phi = \Phi_R, \tag{11.18}$$

wobei man sich alle inneren $\Phi_{i,j}$ links als eine Spalte denkt und Φ_R durch Randwerte gegeben ist. Alle Verfahren zur Lösung linearer Gleichungen könnte man also verwenden. Die im folgenden diskutierten Verfahren sind jedoch für dieses Problem effektiver und nutzen aus, daß die Matrix A dünn besiedelt ist, d. h. nur nahe der Diagonalen von Null verschiedenene Matrixelemente hat. Das liegt daran, daß in (11.17) jedes Φ nur mit den 4 Nachbarn verknüpft wird.

11.4 Gauß-Seidel-Iteration

Die Idee des Gauß-Seidel-Verfahrens besteht darin, immer wieder U als Funktion *eines* $\Phi_{i,j}$ zu minimieren. Man hat eine Feldkonfiguration im Speicher und einen zugehörigen U-Wert, besucht alle inneren Gitterplätze und setzt das jeweilige $\Phi_{i,j}$ auf den Wert, der U minimiert als Funktion dieser Variablen bei festgehaltenen übrigen.

Wir setzen nun, um einfachere Formeln zu haben, $h_x = h_y = h$. Dann kann man für einen beliebigen, aber fest gewählten inneren Gitterpunkt (i, j) die Terme in (11.15) folgendermaßen organisieren,

$$U = 2\varepsilon_0(\Phi_{i,j} - \bar{\Phi}_{i,j})^2 + \tilde{U}_{i,j}, \tag{11.19}$$

wobei $\tilde{U}_{i,j}$ nur Terme enthält, die von dem betrachteten $\Phi_{i,j}$ nicht abhängen, und

$$\bar{\Phi}_{i,j} = \frac{1}{4}(\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1}) \tag{11.20}$$

über die Nachbarn mittelt. Das Gauß-Seidel (GS)-Verfahren setzt einfach jeweils

$$\Phi_{i,j} \to \bar{\Phi}_{i,j} \tag{11.21}$$

Das Verfahren ist sehr opportunistisch, jeder Feldwert setzt sich auf den Mittelwert seiner Nachbarn! Das konvergiert nicht einfach exakt nach einem Durchgang durchs Gitter, da ja dann die Nachbarn auch nicht mehr sind, was sie mal waren und (11.17) *nicht* überall erfüllt ist. Dennoch konvergiert das Verfahren asymptotisch, was dadurch plausibel ist, daß U während der Iteration monoton fällt. Gestartet wird die Iteration von einer im Prinzip beliebigen Anfangs-Feldkonfiguration, die die Randbedingungen erfüllt, wobei aber eine Schätzung des Resultats, soweit erhältlich, zu schnellerer Konvergenz führt und eingesetzt werden sollte.

11.5 Jacobi–Verfahren

Das Jacobi-Verfahren unterscheidet sich von Gauß-Seidel dadurch, daß man zwei komplette Konfigurationen im Speicher hält, Φ^{alt} und Φ^{neu}. Dann wird wieder die Gleichung (11.21) benutzt, wobei auf dem ganzen Gitter links Φ_{neu} konstruiert wird und rechts Φ_{alt} verwendet wird, also

$$\Phi^{neu}_{i,j} = \frac{1}{4}(\Phi^{alt}_{i+1,j} + \Phi^{alt}_{i-1,j} + \Phi^{alt}_{i,j+1} + \Phi^{alt}_{i,j-1}), \tag{11.22}$$

für alle inneren (i,j). Danach nimmt man Φ^{neu} als neues Φ^{alt} und iteriert weiter.

Für das Jacobi–Verfahren können wir die Konvergenz analytisch ausrechnen. Dazu bezeichnen wir für gegebene Randbedingungen die gesuchte Lösung mit Φ^*, die die Laplace–Gleichung auf dem Gitter erfüllt, $(\triangle\Phi^*)_{i,j} = 0$, vgl. (11.17). Die Folge der Jacobi–Konfigurationen sei Φ^n, und wir definieren die Differenz

$$\varphi^n = \Phi^n - \Phi^*. \tag{11.23}$$

Diese erfüllen wie Φ^n die Rekursion

$$\varphi^{n+1} = \left(1 + \frac{h^2}{4}\triangle\right)\varphi^n, \tag{11.24}$$

verschwinden aber auf dem Rand. Mit diesen Randbedingungen kann man die Fourier Entwicklung in Sinus Wellen mit rellen Koeffizienten $a^n_{i,j}$ ansetzen,

$$\varphi^n_{i,j} = \sum_{I,J} a^n_{I,J}\ \psi(I,J;i,j)\ , \tag{11.25}$$

$$\psi(I,J;i,j) = \sin\left(\frac{(i-1)I\pi}{(N_x-1)}\right)\sin\left(\frac{(j-1)J\pi}{(N_y-1)}\right), \tag{11.26}$$

wobei die Summe über $1 \le I \le N_x - 2,\ 1 \le J \le N_y - 2$ läuft. Die so geschriebenen φ^n verschwinden auf dem Rand und sind ansonsten durch die $a^n_{I,J}$ parametrisiert. Diese Transformation ist umkehrbar (gleichviele reelle Parameter!), so daß dies immer möglich ist. Der Nutzen hier beruht darauf, daß der Gitter Laplace–Operator in dieser Darstellung diagonal ist,

$$-h^2\triangle\psi(I,J;i,j) = 4(1-\lambda_{I,J})\ \psi(I,J;i,j) \tag{11.27}$$

mit

$$\lambda_{I,J} = \frac{1}{2}\left\{\cos\left(\frac{I\pi}{N_x-1}\right) + \cos\left(\frac{J\pi}{N_y-1}\right)\right\}. \tag{11.28}$$

Die Koeffizienten $a^n_{I,J}$ iterieren dann gemäß (11.24)

$$a^n_{I,J} = a^0_{I,J}(\lambda_{I,J})^n. \tag{11.29}$$

Die $a^n_{I,J}$ geben die Abweichung von der gesuchten Lösung im n-ten Schritt, wobei $a^0_{I,J}$ unserer Wahl für den Start entspricht. Wir sehen, daß die Abweichung monoton ausstirbt, da $|\lambda_{I,J}| < 1$ gilt. Am schlechtesten konvergieren offenbar die Moden $(I,J) = (1,1)$ und $(I,J) = (N_x-2, N_y-2)$ mit $(N_x, N_y \gg 1)$

$$\lambda_{1,1} = -\lambda_{N_x-2,N_y-2} \simeq 1 - \tau^{-1} \tag{11.30}$$

und

$$\tau^{-1} = \frac{\pi^2}{4}(N_x^{-2} + N_y^{-2}), \tag{11.31}$$

so daß näherungsweise gilt

$$a_{1,1}^n \propto \exp(-n/\tau). \tag{11.32}$$

Da $N_x, N_y \propto 1/h$, benötigt man i. a. $\mathrm{O}(\tau \sim h^{-2})$ Iterationen zur Konvergenz.

Die Analyse von Gauß-Seidel ist wegen der sofortigen Benutzung der jeweils wieder neuen Werte auf der rechten Seite wesentlich schwieriger. Man findet jedoch in der Literatur die Aussage, daß τ hier zwar nach wie vor $\propto h^{-2}$ geht, jedoch jeweils nur halb so groß ist. Jacobi ist andererseits weniger rekursiv, die Ersetzung (11.22) kann auf allen Gitterplätzen simultan und unabhängig gemacht werden. Das wäre für Parallel- und Vektorrechner ein Vorteil. Tatsächlich ist das nun folgende Verfahren aber beiden deutlich überlegen. Die Rekursion ist von GS-Typ. Man kann auch diese Verfahren parallelisieren, z. B. durch eine Schachbrett Einteilung des Gitters, was wir hier aber nicht weiter betrachten.

11.6 Sukzessive Überrelaxation

Im GS Schritt (11.21) kann man einen Parameter einschmuggeln,

$$\Phi_{i,j} \to \Phi_{i,j} + \omega(\bar{\Phi}_{i,j} - \Phi_{i,j}). \tag{11.33}$$

Der zweite Term ist als Korrekturterm aufzufassen, und GS entspricht $\omega = 1$. Für die Lösung verschwindet die Korrektur und sie ist ein Fixpunkt für alle Werte ω. Die Energie ändert sich nun in

$$U \to U + 2\varepsilon_0 \left[(\omega - 1)^2 - 1\right] (\bar{\Phi}_{i,j} - \Phi_{i,j})^2. \tag{11.34}$$

Damit wird sie abgesenkt für alle Werte $0 < \omega < 2$. Interessant ist der Bereich der Überrelaxation (Successive Over-Relaxation = SOR), $1 < \omega < 2$. Hier wird "überkorrigiert" über das Minimum der Parabel (als Funktion eines $\Phi_{i,j}$) hinaus. Die lokale Energieabsenkung ist nicht die maximal mögliche. Dennoch ist SOR äußerst profitabel, wenn ω geignet gewählt wird. Genauer gesagt kann man dann $\tau \propto h^{-1}$ (statt h^{-2}) erreichen, ein Gewinnfaktor von $\mathrm{O}(N)$ auf einem $N \times N$ Gitter! Leider ist diese Beschleunigung schwer plausibel zu machen. Die Überkorrektur berücksichtigt in gewisser Weise schon

Verbesserungen der Lösung, die sonst im nächsten Schritt kämen und zieht auch die Nachbarn, wo ja der neue Wert dann eingeht, mit.

Für die einfache Geometrie und Randbedingungen hier kann man zeigen [3], daß

$$\omega_{\rm opt} = \frac{2}{1+\sqrt{1-\lambda_{1,1}^2}} \tag{11.35}$$

optimal ist, mit $\lambda_{I,J}$ von der Jacobi–Konvergenzanalyse (11.28). Für $N_x, N_y \propto N \to \infty$, N_x/N_y fest, gilt allgemein

$$\omega_{\rm opt} \sim 2 - \frac{c}{N}. \tag{11.36}$$

Für kompliziertere Geometrien muß $\omega_{\rm opt}$ experimentell ermittelt werden. Nach obiger Formel reicht dafür ein groberes Gitter um c zu schätzen, und dann kann $2-\omega$ skaliert werden. Auch bezieht sich "optimal" nur auf die asymptotische Konvergenz nach vielen Iterationen. Ausgefuchste Programme können auch versuchen, diesen Parameter während der Iteration selbst zu verbessern, was aber delikat ist, da z. B. das Residuum der Laplace–Gleichung nicht immer monoton fällt.

11.7 Gittergeometrie und Randbedingungen

Die Konfiguration $\Phi_{i,j}$ wird in MATLAB als Matrix gespeichert. Dies ist bei den von uns betrachteten zweidimensionalen Fällen für MATLAB naheliegend und erlaubt einfaches Plotten am Ende[21]. Sind der geometrische Bereich und die Randbedingungen rechteckig, so kann man den Iterationsloop recht leicht auf den inneren Bereich einschränken, was ja nötig ist. Wir wollen hier aber einen etwas allgemeineren Ansatz wählen, der für die Übungen benötigt wird und mehr Freiheit gibt.

Die Matrix für $\Phi_{i,j}$ wird so groß gewählt, daß sie das Problem, das z. B. näherungsweise rund sein kann, umhüllt. In der Matrix gibt es dann, innere und Randpunkte, sowie Punkte die gar nicht zum Problem gehören, d. h. nicht in die Laplace–Gleichung eingehen. Zusätzlich dazu führen wir nun eine Matrix $f_{i,j}$ der gleichen Größe ein, deren Werte diese Fälle (und eventuelle weitere) unterscheiden. Im Beispiel, das hier folgt, genügt es $f_{i,j}=0$ zu haben genau für alle nicht inneren Punkte. Wir nennen f ein Flag-Feld [9].

[21] Eine Alternative wäre ein Vektor für alle irgendwie durchnummerierten Gitterpunkte. Dann braucht man Zeigerfelder um die Nachbarn zu finden. Dies ginge dann auch in drei Dimensionen.

11.8 Beispiel eines MATLAB Programms für Jacobi–Iteration

Wir wollen nun einige bei der Relaxation nützliche MATLAB Kommandos und Programm-Konstrukte an Hand eines Beipiel Programms für Jacobi–Iteration lernen. Es findet sich in /users/com/bunk/pub und kann von dort als Basis für die Übungen heruntergeladen werden. Hier ein Listing:

```
1    % file jacob.m; L"osung der Laplace--Gleichung.
2    % quadratisches Gitter h_x=h_y=h, L_x=L_y=L
3    clear; help jacob;
4    N=20; %N = input(' N(=N_x=N_y)? - ');
5    L = 1;         % System Groesse
6    h = L/(N-1);  % Gitterabstand
7    phi0 = 1;      % Phi_0
8    tol = 1e-4;   % Abbruchkriterium mittlere Aenderung in Phi
9    %%%%% Anfangswerte, Flag Feld etc.
10   x = (0:N-1)*h;
11   y = (0:N-1)*h;
12   %
13   flag = zeros(N,N); % einhuellende Matrix mit Nullen
14   flag(2:N-1,2:N-1) = ones(N-2,N-2); % innen Einsen
15   %
16   phi = zeros(N,N); % = phineu
17   %
18   phi(:,N) = phi0*min(x/L,1-x/L).'; % Randbed. y=L, manifest symm. x->L-x
19   %
20   for j = 2:N-1,
21     phi(:,j) = phi(:,N)*(j-1)/(N-1);
22   end % linear interpoliert zum Start
23   %
24   max_iter = 2*N^2;  % Grenze zur Sicherheit
25   time=cputime;
26   for iter=1:max_iter
27     phialt = phi; % fuer Jacobi-Verfahren
28     temp = 0;
29
30     for i = 1:N, for j = 1:N, if flag(i,j) % loop Gitterinneres
```

```
31  %    Jabobi:
32      phi(i,j) = .25*(phialt(i+1,j  )+phialt(i-1,j  )+ ...
33                      phialt(i  ,j-1)+phialt(i  ,j+1));
34      temp = temp + abs(phi(i,j)-phialt(i,j));
35    end, end, end  % Ende loop Gitterinneres
36  %
37    phialt = phi;
38    change(iter) = temp/(phi0*(N-2)^2); % mittlere Aenderung
39  %  fprintf('Nach %g Iterationen, Aenderung = %g\n',iter,change(iter));
40    if( change(iter) < tol )
41      disp(' konvergiert => Abbruch ');
42      break;
43    end
44  end % Ende loop iter
45  time=cputime-time;
46  fprintf('Iterationszeit = %g\n',time);
47  %
48  subplot(2,2,1)
49  mesh(x,y,phi.'); % 3d-Plot der Funktion phi(x,y),
50  %                  phi transponiert, sonst x<->y vertauscht!
51  xlabel('x/L'); ylabel('y/L'); zlabel('Phi/Phi0');
52  title('Loesung');
53  %
54  subplot(2,2,2)
55  V = [.01 .05 .1 .2 .3 .4]; % contour Hoehen
56  cpl = contour(x,y,phi.',V);
57  xlabel('x/L'); ylabel('y/L');
58  title('Loesung, Contour-Plot');
59  clabel(cpl,V); % label der contour Linien
60  %
61  subplot(2,2,3)
62  semilogy(change);
63  xlabel('Iteration'); ylabel('Aenderung');
64  %
65  % Symmetrie Test:
66  sym = abs(max(phi-flipud(phi)));
67  %
68  subplot(2,2,4)
```

```
69   plot(y,sym); title('abs. Symmetriefehler'); xlabel('y');
```

In den Zeilen 5 und 7 sind L und Φ_0 gleich 1 gesetzt, was der Wahl von dem Problem angemessenen Einheiten entspricht: alle Längen als Vielfache von L und alle Potentiale als Vielfache von Φ_0. In Z. 13–14 wird das Flag-Feld gesetzt. Für Φ werden in Z. 16–18 die Randbedingungen gesetzt, nämlich die Gitterversion von

$$\begin{aligned} \Phi(0,y) &= \Phi(L,y) = \Phi(x,0) = 0 \\ \Phi(x,L) &= \Phi_0 \times \begin{cases} x/L & \text{für} \quad 0 \le x \le L/2 \\ 1-x/L & \text{für} \quad L/2 \le x \le L \end{cases} , \end{aligned} \tag{11.37}$$

was der Übungsaufgabe 11.1 entspricht. Dabei wird die Symmetrie $x \to L-x$ in diskreter Form $i \to N+1-i$ exakt gewahrt, so daß wir diese in der Lösung bis auf numerische Fehler finden müssen, da die Laplace–Gleichung selbst diese Symmetrie besitzt (keine der 4 Gitterachsen Richtungen ausgezeichnet). Eine glatte Startkonfiguration wird in den Zeilen 20–22 hergestellt. Der eigentliche Jacobi loop findet sich in den Zeilen 30–35. In den Zeilen 40-43 wird das Konvergenzkriterium abgefragt.

Ab Z. 48 wird geplottet und das Ergebnis ist in Abb.15 zu sehen. Das MATLAB Kommando **mesh** erstellt einen 3-d Drahtgitter Plot des Potentials über der $x-y$ Ebene. Die Argumente sind die den Matrixindizes entsprechenden x- und y-Werte und die Feldwerte als Matrix, die, wenn ihr erster Index x entspricht, transponiert werden muß[22].

Das nächste Teilbild zeigt Höhenlinien von Φ, deren Höhenwerte in Z. 55 vorgegeben werden (geht aber auch automatisch). **contour** gibt eine Matrix (`cpl`) zurück, die in Z. 59 zum Beschriften der Linien gebraucht wird. Weiteres, s. **help**.

Nach der halblogarithmischen Darstellung des Konvergenzverhaltens wird in den Zeilen 65-69 schließlich die Symmetrie als Kontrolle und weiterer Hinweis auf numerische Fehler geprüft. Die Funktion **flipud** (flip up-down) spiegelt eine Matrix an einer horizontalen Geraden durch ihre Mitte. Das Gleiche erhielte man durch `phi(N:-1:1,:)`! Wenn die Symmetrie gilt, muß die Matrix rechts im Argument in Z. 66 verschwinden. In Z. 69 plotten wir ihre spaltenweisen Maxima, d.h. also für jedes y. Wie in Abb.15 zu sehen ist, ist die Symmetrie offenbar in Maschinengenauigkeit erfüllt. Dies gilt für Jacobi,

[22] Hier sind 2 Konventionen in Konflikt: die Reihenfolge x, y und daß der erste Matrixindex vertikal läuft, was konventionell die y-Achse ist.

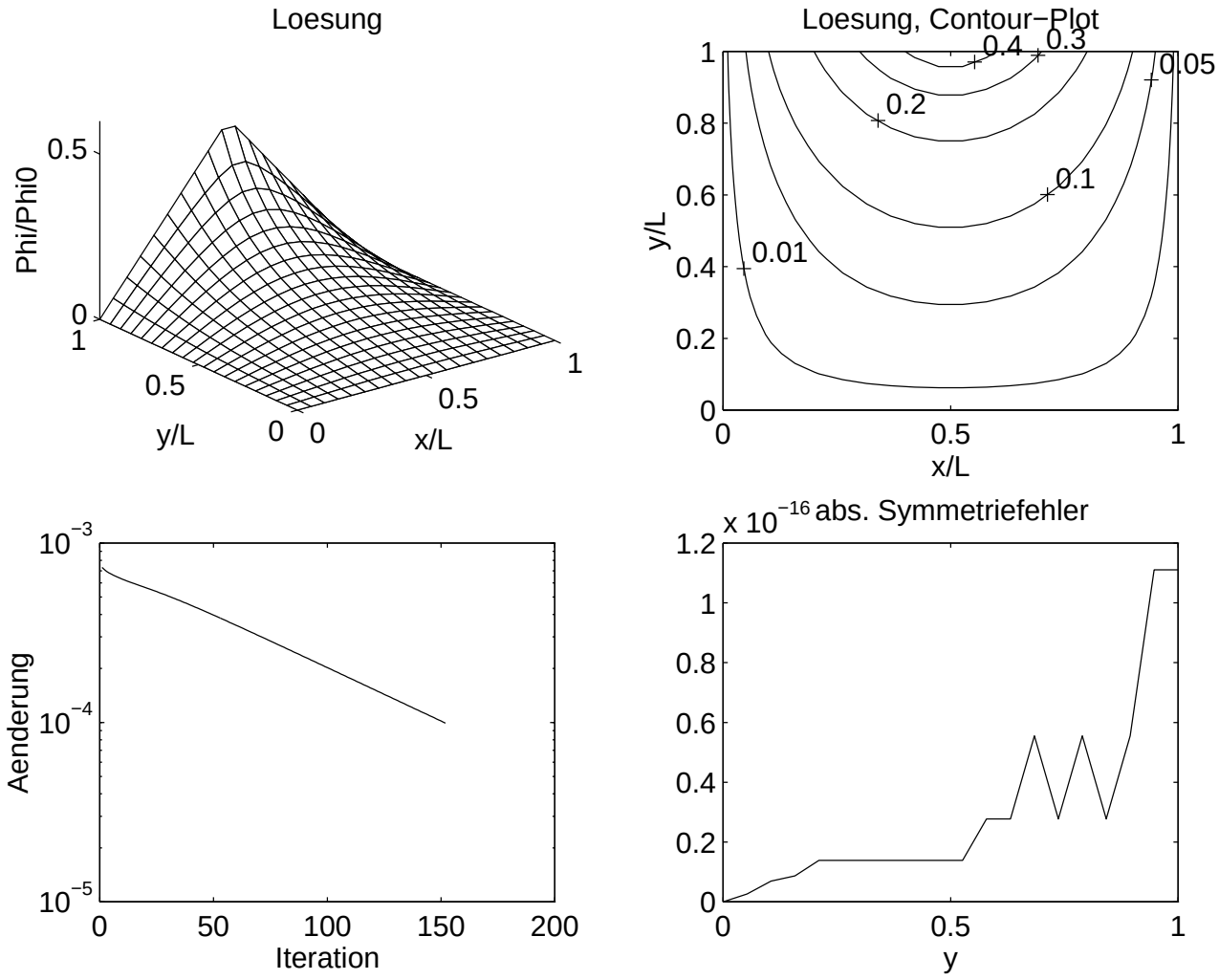

Abbildung 15: Plot Output von `jacob.m`

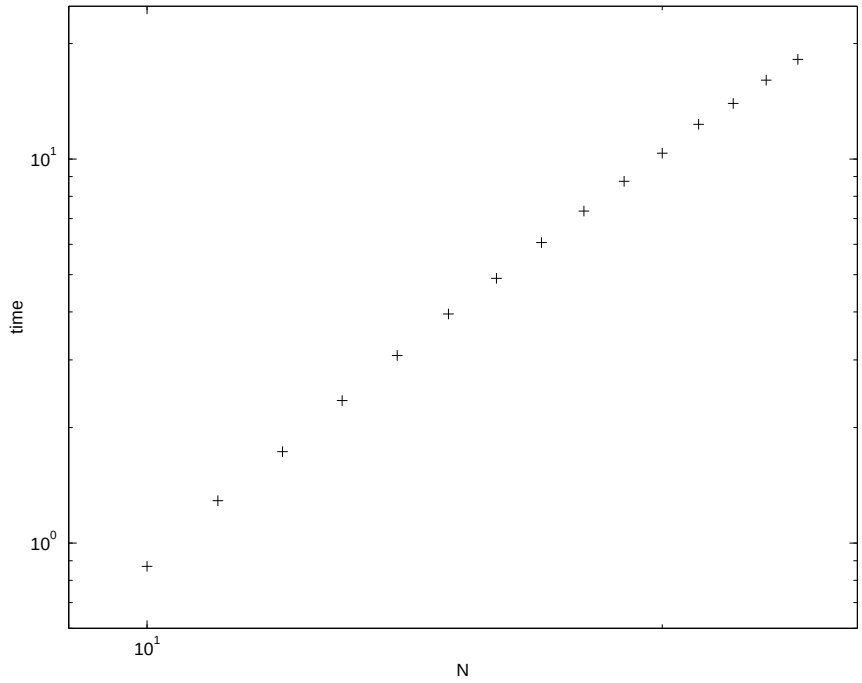

Abbildung 16: Iterationszeit beim Jacobi-Verfahren als Funktion von N in doppelt-logarithmischer Auftragung.

da sie bei jeder einzelnen Iteration erhalten bleibt (bitte nachprüfen), nicht jedoch bei GS und SOR.

In der Abbildung 16 haben wir die Iterationszeiten beim Jacobi-Verfahren doppelt-logarithmisch als Funktion von N ($10 \leq N \leq 24$) aufgetragen. Alle übrigen Parameter sind hierbei festgeblieben und haben die Werte wie im Programmlisting. Man erkennt, daß die Punkte in etwa auf einer Geraden liegen und die Interationszeit daher proportional zu N^α anwächst. Aus der Steigung ergibt sich für den Exponenten α ein Wert, der nahe bei vier liegt. Dies läßt sich leicht verstehen: Einmal hat man $\tau \propto N^2$ für die Konvergenzgeschwindigkeit, also die benötigte Anzahl von Interationen zur verlangten Genauigkeit. Die Kosten pro Iteration durchs Gitter skalieren nochmals mit N^2. Mit optimalem SOR wären wir sogar insgesamt bei N^3. Würden wir die Laplace–Gleichung auf dem Gitter mit einem Standardverfahren für beliebige Matrizen lösen, z. B. Gauß-Elimination, so hätten wir es mit einer $N^2 \times N^2$ Matrix A in (11.18) zu tun und bräuchten $\mathrm{O}(N^6)$ Operationen.

Dieses Verfahren würde gegen die iterativen Verfahren für dünn besiedelte Matrizen bei größeren N sehr bald ins Hintertreffen geraten.

12 Quantenmechanischer anharmonischer Oszillator

In diesem Abschnitt werden wir die Schrödinger Gleichung für den eindimensionalen harmonischen und anharmonischen Oszillator betrachten. Es wird alles vorbereitet, um Energien der niedrigsten Zustände im anharmonischen Fall numerisch in den Übungen zu berechnen. Obwohl wir hier die Quantenmechanik nur auf einfachem Niveau benötigen, wird vorausgesetzt, daß Grundkenntnisse aus dem integrierten Kurs Physik IV oder einer Vorlesung Quantenmechanik I vorhanden sind. Dann müßte der harmonische Oszillator bekannt sein.

12.1 Hamilton–Operator und Parität

In der Quantenmechanik sind mögliche scharfe Energien in stationären Zuständen eines betrachteten Systems die Eigenwerte des Hamilton–Operators. In der Schrödingerdarstellung hat dieser für ein eindimensionales nichtrelativistisches System die Gestalt

$$H = -\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x). \tag{12.1}$$

H wirkt auf Wellenfunktionen $\psi(x)$ über dem x-Interval $(-\infty, \infty)$. Für gebundene, auf einen endlichen Raumbereich konzentrierte Zustände, auf die wir uns hier beschränken, ist die Wellenfunktion quadratintegrabel. $|\psi(x)|^2$ ist bekanntlich als Aufenthalts-Wahrscheinlichkeitsdichte zu interpretieren.

Wir wollen noch voraussetzten, daß V invariant unter Spiegelungen ist,

$$V(-x) = V(x). \tag{12.2}$$

Dies induziert einen Spiegelungs- oder Paritätsoperator P auf den Wellenfunktionen,

$$(P\psi)(x) = \psi(-x). \tag{12.3}$$

Er vertauscht nach Voraussetzung mit H, und kann gleichzeitig mit H diagonalisiert werden. Wegen $P^2 = 1$ können nur Eigenwerte ± 1 auftreten entsprechend symmetrischen und antisymmetrischen Eigenfunktionen. Nicht entartete Energien müssen also automatisch definierte Parität haben, was uns nützlich sein wird.

Wir wollen hier verallgemeinerte Oszillatoren mit

$$V(x) = \frac{1}{2}k|x|^\alpha,\ \alpha > 0 \tag{12.4}$$

betrachten, obwohl natürlich allgemeinere Funktionen denkbar sind. Da das Potential für $|x| \to \infty$ divergiert, gibt es ausschließlich gebundene Zustände. Zunächst wenden wir uns dem Fall $\alpha = 2$ zu. Dies ist der bekannte und allseits beliebte harmonische Oszillator. Er (bzw. Systeme mit vielen solchen) macht (mindestens) die halbe Physik aus.

12.2 Harmonischer Oszillator, $\alpha = 2$

Im harmonischen Fall schreibt man meist für die Federkonstante $k = m\omega^2$, was die in Lösungen (schon klassisch) auftretende charakteristische Winkelfrequenz einführt. Es ist nun leicht zu sehen, daß, wenn man

- Energien in Vielfachen von $\hbar\omega$,
- die Länge x in Vielfachen von $\sqrt{\hbar/m\omega}$

mißt, der Hamilton–Operator übergeht in

$$H = \frac{1}{2}\left(-\frac{\mathrm{d}^2}{\mathrm{d}x^2} + x^2\right). \tag{12.5}$$

Mit Hilfe der Erzeuger und Vernichter

$$\begin{aligned} a &= \frac{1}{\sqrt{2}}\left(\frac{\mathrm{d}}{\mathrm{d}x} + x\right) \\ a^\dagger &= \frac{1}{\sqrt{2}}\left(-\frac{\mathrm{d}}{\mathrm{d}x} + x\right), \end{aligned} \tag{12.6}$$

die die Vertauschungsrelation

$$aa^\dagger - a^\dagger a = \left[a, a^\dagger\right] = 1 \tag{12.7}$$

erfüllen, gilt dann

$$H = \frac{1}{2} + a^\dagger a. \tag{12.8}$$

Es gibt Eigenzustände

$$H\psi_n = E_n\psi_n \tag{12.9}$$

mit

$$E_n = \frac{1}{2} + n. \tag{12.10}$$

Der normierbare Grundzustand wird von a vernichtet,

$$a\psi_0 = 0 \to \psi_0 \propto \exp(-x^2/2), \tag{12.11}$$

und die höheren Zustände ergeben sich durch Anwenden von $a^\dagger$,

$$\psi_n \propto \left\{a^\dagger\right\}^n \psi_0. \tag{12.12}$$

Dies und die Eigenwerte (12.10) lassen sich induktiv mit Hilfe von (12.7) zeigen. Da der Grundzustand symmetrisch unter P ist und $Pa^\dagger = -a^\dagger P$ gilt, so hat der n-te Zustand die Parität

$$P\psi_n = (-1)^n \psi_n. \tag{12.13}$$

12.3 Eigenwertgleichung als gewöhnliche Differentialgleichung

Wir wollen nun das allgemeine Oszillatorproblem dahingehend abändern, daß der x-Bereich endlich wird, $x \in (-R, R)$, und die Randbedingung $\psi(-R) = \psi(R) = 0$ gefordert wird. Man kann dies auch als modifiziertes Potential ansehen, das dann außerhalb unendlich groß wird. Physikalisch ist klar, daß dies für $R \gg 1$ die unteren Zustände kaum beinflußt, da die Aufenthalts Wahrscheinlichkeit bei $\pm R$ sowieso exponentiell klein ist, nachdem dies tief im "verbotenen" Bereich liegt wegen $V(R) \gg E$. Die Kompaktifizierung des Raums ist von Vorteil für die folgende Diskussion, die zu einer numerischen Methode für die Eigenwerte führt. Der harmonische Oszillator wird allerdings (im exakten Sinne) erheblich verkompliziert und ist nicht mehr einfach geschlossen lösbar.

Die Gleichung

$$(H - E)\psi = 0 \tag{12.14}$$

soll betrachtet werden mit

$$H = \frac{1}{2}\left(-\frac{\mathrm{d}^2}{\mathrm{d}x^2} + |x|^\alpha\right) \tag{12.15}$$

in geeigneten Einheiten, um herauszufinden, für welche E sie Lösungen hat. Es muß also gelten

$$\psi''(x) = 2(V(x) - E)\psi, \tag{12.16}$$

und — erst dies legt E fest! — die Randbedingungen müssen erfüllt sein. Wir suchen also Lösungen mit

$$\psi(\pm R) \stackrel{!}{=} 0. \tag{12.17}$$

Wenn wir uns an Anfangswertprobleme aus der Mechanik erinnern — man muß sich hier dann $\psi(x)$ als analog zu $y(t)$ denken —, so haben wir Erfahrungen mit der Lösung des Problems für gegebene $\psi(-R) = 0$ und $\psi'(-R)$ dann $\psi(x), x \geq -R$ zu berechnen als Lösung der DGL zweiter Ordnung. Leider das falsche Problem: Anfangswert und -geschwindigkeit gegeben statt Anfangs- und Endort. Man kann nun aber (im Prinzip) folgendermaßen vorgehen:

- löse das Anfangswertproblem mit $\psi(-R) = 0$, $\psi'(-R) = 1$
- betrachte das sich ergebende $\psi(+R)$ als wohldefinierte Funktion $F(E)$
- suche deren Nullstellen, $F(E_n) = 0$, welche Energieeigenwerte sind
- die zu $E = E_n$ gehörende Lösung ist dann ein (unnormierter) Eigenzustand

Wichtig ist hier, daß (12.16) *linear* ist. Mit dem obigen ψ ist ein Vielfaches auch Lösung. Daher gilt es nur irgendeinen Wert $\psi'(-R) \neq 0$ anzunehmen. Hat man dann eine Lösung zum Eigenwert E_n, so kann diese am Ende normiert werden.

Der Überblick über gefundene Lösungen wird in diesem einfachen Fall noch wesentlich dadurch erleichtert, daß aus der Quantenmechanik bekannt ist, daß Entartung hier nicht auftritt und daß die n-te reell wählbare Lösung n Knoten hat bei Zählung ab $n = 0$ (Grundzustand). Dies ist der in vielen Lehrbüchern diskutierte Knotensatz, s. z. B. [10]. Man kann also durch Plotten der Lösungen darauf achten, daß man zwischen zwei Eigenwerten keinen weiteren vergißt.

Bei der praktischen Implementierung des Verfahrens wollen wir noch die Parität berücksichtigen. Wir wissen, daß ψ_n gerade oder ungerade ist. Dann muß $\psi'(0) = 0$ (gerade) oder $\psi(0) = 0$ (ungerade) gelten. Man kann also eine dieser Bedingungen als Ziel stellen (statt $\psi(R) = 0$). Das hat mehrere Vorteile. Zum einen braucht man nur halb soweit zu integrieren, nämlich von $-R$ bis 0. Zum anderen kann man die Parität der Lösung von vornherein vorgeben. Dann sind die Nullstellen $F(E)$ weiter separiert und damit besser auffindbar.

Das wichtigste ist allerdings, daß das Fortschreiten *in* den klassisch verboteten Bereich $V(x) > E$ *hinein* numerisch instabil ist. Für große $|x|$ wird E neben $V(x)$ in (12.16) vernachlässigbar, und es gibt je eine exponentiell wachsende und eine fallende Lösung[23]. Beim Erfüllen der Randbedingungen gilt es nun E so zu finden, daß man gerade in die fallende Lösung "einmündet" und einen normierbaren Zustand hat. Nun haben aber Rundungs- und Integrationsfehler die Tendenz, immer wieder die andere (schnell wachsende aber unphysikalische) Komponente mit zunächst kleiner Amplitude beizumischen, die dann aber viel schneller wächst. Dies ist die Instabilität. Sie wird hier mit Hilfe der Paritätssymmetrie umgangen. Man erledigt sozusagen den zweiten problematischen Teil der Integration mit einem Symmetrieargument.

12.4 Hinweise zur MATLAB -Implementierung

Mittlerweile sind viele Standard–MATLAB –Kommandos bekannt, und die Eigenwertbestimmung nach dem obigen Verfahren sollte ohne weiteres durchführbar sein. Es gibt daher diesmal kein Musterprogramm. Allerdings wird nochmal erinnert, daß **ode23 und ode45** den absoluten Fehler der Lösung kontrolliert, wenn diese sehr klein wird, und sonst den relativen. Dies soll helfen durch Nulldurchgänge zu kommen. Es ist aber, wie schon erwähnt, nur sinnvoll in natürlichen Einheiten. Der Start der Integration mit $\psi'(-R) = 1$ ist aber willkürlich.

Die Frage ist, welcher Wert zu $\psi(0) = \mathrm{O}(1)$ bzw. $\psi'(0) = \mathrm{O}(1)$, je nachdem was von beiden nicht verschwindet, führt. Die Eigenwertgleichung für große $|x|$ ist

$$\psi'' \approx |x|^\alpha \psi \tag{12.18}$$

in unserem Problem mit der näherungsweisen (abfallenden!) Lösung

$$\psi \propto \exp\left\{-\frac{2}{\alpha+2}\,|x|^{\frac{\alpha}{2}+1}\right\}. \tag{12.19}$$

Daraus kann man eine Größenordnung für $\psi'(-R)$ beziehen. Sollte man immer noch mit extremen Werten bei $x = 0$ ankommen, so kann man diese nehmen, um $\psi'(-R)$ besser zu wählen, und nochmals integrieren.

Die Eigenwerte werden als Nullstellen einer Funktion, $F(E_n) = 0$, gefunden. Der erste Versuch wird darin bestehen, verschiedene E zu probieren und

[23] Wir argumentieren hier wieder für $R = \infty$. Unser R ist per Voraussetzung so groß, daß das geschilderte Verhalten einsetzt, bevor man R erreicht hat.

einen Vorzeichenwechsel einzukreisen, z. B. indem man ein Intervall nimmt und dieses fortgesetzt halbiert, indem man noch das Vorzeichen in der Mitte bestimmt.

Die Suche von Nullstellen (oder die Suche nach Lösungen von i. a. nichtlinearen Gleichungen, die man ja so formulieren kann) ist ein Standardproblem für numerische Verfahren und wurde in einem früheren Kapitel behandelt. Hier ist es am einfachsten, die MATLAB Routine **fzero** (function zero) als "black box" zu benutzen.

Wenn man also eine Funktion $F(E)$ als `.m` file definiert und diesen Namen übergibt, dann kann man **fzero** benutzen, um E_n genau zu lokalisieren. Welche Lösung gefunden wird, hängt vom starting guess (Anfangsschätzung) ab. Mit **dbtype fzero** kann sich, wer will, auch ansehen, was **fzero** tut.

13 Eindimensionale Quantenmechanik mit Matrixmethoden

In diesem Teil wollen wir einige eindimensionale quantenmechanische Systeme untersuchen. Dabei wird es nicht darum gehen, hocheffektive Methoden mit genauer Fehlerabschätzung zu studieren. Wir werden vielmehr die Operatoren der betrachteten Quantensysteme durch Diskretisieren auf Matrizen abbilden und dann die lineare Algebra Funktionalität von MATLAB verwenden. Es handelt sich dabei um einen naiven direkten Ansatz, der amüsante Veranschaulichungen von Lehrbuchproblemen ermöglicht. Eine Verallgemeinerung auf mehrere Dimensionen dürfte problematisch sein. Wirklich konkurrenzfähige Algorithmen werden wir im Kurs *Computational Physics II* behandeln.

Die Idee zu diesem Teil wurde durch Lesen von [11] ausgelöst. Dort steht allerdings von Anfang an das Feynman'sche Pfadintegral im Mittelpunkt. Wir finden dies nicht so natürlich und kommen ausgehend von der Standard Formulierung sogar zu numerisch wesentlich effektiveren Rechnungen. Daher ziehen wir es vor, den Zusammenhang mit Pfadintegralen, die konzeptionell enorm wichtig sind, im Anschluß in einem eigenen Kapitel zu diskutieren.

Wenn jemand mit den Beispielen dieses Kapitels spielen will: alle Programme finden sich in `/users/com/uwolff/CP1/kap13`.

13.1 Diskretisierte Operatoren

In der Schrödinger Darstellung der Quantenmechanik sind Zustände durch Wellenfunktionen $\psi(x)$ gegeben, wobei normalerweise $x \in (-\infty, \infty)$. Operatoren können durch Integralkerne dargestellt werden. Also eine Abbildung $\phi = \hat{A}\psi$ z. B. durch

$$\phi(x) = \int_{-\infty}^{\infty} \mathrm{d}y A(x,y)\psi(y). \tag{13.1}$$

Für den einfachen Ortsoperator $\hat{x}$, der $\psi(x)$ einfach mit x multipliziert, hat man etwa

$$A(x,y) = x\delta(x-y). \tag{13.2}$$

A kann als Matrix angesehen werden, deren Indizes kontinuierlich sind und einen unendlichen Bereich haben. Daraus muß zur numerischen Behandlung wieder eine endliche Matrix werden, die aber bis auf unvermeidliche kleine Abweichungen noch etwas mit dem Problem zu tun hat.

Klarerweise muß dann der x-Bereich endlich und dann noch diskretisiert werden. Wir schränken ein auf $x \in (-L, L)$. Für welche L ist dies physikalisch sinnvoll? Wir denken überwiegend an Systeme vom Typ harmonischer Oszillator. Dort haben alle Eigenzustände eine Breite, jenseits derer sie exponentiell abfallen. Die Breite ist durch Parameter im Hamilton Operator bestimmt. Wenn L deutlich größer als diese ist, können wir erwarten, daß die künstlich eingeführten Ränder nicht viel ausmachen. Wählt man angepaßte Einheiten, so bedeutet dies $L \gg 1$, z. B. $L = 10$, und der Einfluß ist durch Variieren zu untersuchen. Ganz ähnliche Überlegungen gelten für das Diskretisieren mit einer Schrittweite Δx, die fein genug sein muß, um Strukturen in den vorkommenden $\psi(x)$ auflösen zu können. Bei natürlichen Einheiten[24] heißt das $\Delta x \ll 1$.

Wir führen nun also ein

$$x_k = -L + k\Delta x, \quad k = 1, \dots N_x, \quad \Delta x = \frac{2L}{N_x}, \tag{13.3}$$

und ersetzen Wellenfunktionen durch Vektoren der Länge N_x mit (i. a. komplexen) Komponenten $\psi_k = \psi(x_k)$, $k = 1, \dots N_x$. Es gilt so $x_1 = -L + \Delta x$ und $x_{N_x} = L$. Obwohl der Rand im Sinne der Approximation keine Rolle spielen soll, werden wir eine Vorschrift benötigen (für diskretisierte Differentialoperatoren), wo man hinkommt, wenn man von x_1 in negativer Richtung geht. Wir wählen periodische Randbedingungen, d. h. x_{N_x} ist der linke Nachbarpunkt von x_1. Man kann auch sagen, Indizes k und $k \pm N_x$ sind identifiziert (Modulobildung). Wir haben sozusagen das Intervall zu einem Ring zusammengebogen.

Der Ortsoperator ist natürlicherweise nun durch die Matrix

$$\hat{x}_{kl} = x_k \delta_{kl} \tag{13.4}$$

gegeben. Eine einfache Idee, die unseren früheren Näherungen für Ableitungen entspricht, würde die kinetische Energie darstellen mit Hilfe von

$$(\hat{p}^2\psi)_k = \frac{\hbar^2}{\Delta x^2}(2\psi_k - \psi_{k+1} - \psi_{k-1}). \tag{13.5}$$

Man käme bald auf die Idee bessere Näherungen für $\psi''(x_k)$ durch Verwendung von 5 Punkten usw. zu verwenden. Da wir von einer dünnen Besiedelung von $\hat{p}^2$ keine großen Vorteile haben werden, wollen wir hier einen etwas anderen Weg beschreiten.

[24] Wir nehmen an, daß es nur *eine* physikalische Länge im Problem gibt wie $\sqrt{\hbar/m\omega_0}$ beim Oszillator.

Für "Funktionen" ψ_k über diskretisiertem Definitionsbereich gibt es die Fourierentwicklung. Die entscheidende Identität ist

$$\delta_{kl} = \frac{1}{N_x} \sum_p \exp[ip(x_k - x_l)], \tag{13.6}$$

wobei die p-Summe geht über (N_x sei hier gerade)

$$p = \frac{\pi}{L} j; \quad j = -N_x/2, -N_x/2+1 \ldots N_x/2-1; \quad p \in \left[-\frac{\pi}{\Delta x}, \frac{\pi}{\Delta x}\right). \tag{13.7}$$

Das so dargestellte Kronecker δ ist N_x-periodisch in k, l und somit nur mit periodischen Randbedingungen geeignet. Schreiben wir $\psi_k = \sum_l \delta_{kl}\psi_l$ mit der Fourier Darstellung, so können wir momentan die linke Seite als kontinuierliche Funktion von x_k auffassen und auch differenzieren. Damit kommt man zu folgender Matrixdarstellung ($\hbar = 1$),

$$(\hat{p}^2)_{kl} = \frac{1}{N_x} \sum_p p^2 \exp[ip(x_k - x_l)], \tag{13.8}$$

und dies ist die Matrix, die wir verwenden wollen.

13.2 Oszillator Niveaus numerisch

Nachdem wir Matrixdarstellungen von $\hat{x}$ und $\hat{p}^2$ haben können wir Systeme von Typ

$$\hat{H} = \frac{1}{2}\hat{p}^2 + \hat{V}(\hat{x}) \tag{13.9}$$

untersuchen (m=1 durch Wahl der Einheiten). Wir beginnen mit dem harmonischen Oszillator,

$$\hat{V}(\hat{x}) = \frac{1}{2}\hat{x}^2 \tag{13.10}$$

in geeigneten Einheiten. In diesem Fall ist alles exakt bekannt. Hier interessieren uns die Energieeigenwerte

$$E_n = n + 1/2, \quad n = 0, 1, \ldots \tag{13.11}$$

Zum Grundzustand gehört die Eigenfunktion

$$\psi^{(0)}(x) \propto \exp(-x^2/2), \tag{13.12}$$

die, wie schon diskutiert, eine Breite von etwa 1 hat. Bei den höheren Zuständen wird dieser Gauß Faktor noch mit Polynomen multipliziert.

Das nun folgende MATLAB Programm setzt die Geometrie und berechnet die Hamilton Matrix H:

```
%
%  file ho_fou.m
%
%
%  Erzeugung Diskretisierung und
%  Hamilton Operator fuer QM Harmonischen Oszillator
%  Aufruf [dx,x,H] = ho_fou(L,Nx)
%
function [dx,x,H] = ho_fou(L,Nx)

%  Raum:
dx = (L+L)/Nx;              % Diskretisierungslaenge
x   = [-L+dx : dx : L];     % x-Werte
p=(pi/L)*([0:Nx-1] - Nx/2); % Impulse

%  kinetischer Teil des Hamilton:
for i=1:Nx, M(i,:) = x(i) - x; end              % Matrix (x_i - x_j)
H = zeros(Nx,Nx);
for i=1:Nx, H = H + p(i)^2*cos(p(i)*M); end
H = H*(0.5/Nx);                                 % (-1/2) Laplace
%  Oszillator Teil des Hamilton:
for i=1:Nx, H(i,i)=H(i,i)+0.5*x(i)^2; end
%
disp(' H.O. Hamilton mit Fourier-diskretisiertem Laplace Operator: ')
fprintf(' Raumintervall [%g , %g] mit %i Punkten \n',[-L L Nx]);
```

Obwohl man an Matrizen der Größe O(100) denkt, wollen wir uns zum Zweck des Anschauens die Ausgabe für $L = 3, N_x = 6$ und damit $\Delta x = 1$ mal ansehen:

```
>> [dx,x,H]=ho_fou(3,6)
 H.O. Hamilton mit Fourier-diskretisiertem Laplace Operator:
 Raumintervall [-3 , 3] mit 6 Punkten
```

```
dx =

     1


x =

    -2    -1     0     1     2     3


H =

    3.7363   -1.0966    0.3655   -0.2742    0.3655   -1.0966
   -1.0966    2.2363   -1.0966    0.3655   -0.2742    0.3655
    0.3655   -1.0966    1.7363   -1.0966    0.3655   -0.2742
   -0.2742    0.3655   -1.0966    2.2363   -1.0966    0.3655
    0.3655   -0.2742    0.3655   -1.0966    3.7363   -1.0966
   -1.0966    0.3655   -0.2742    0.3655   -1.0966    6.2363

>> diary off
```

Wie es sich gehört ist H hermitesch, hier reell symmetrisch. Wir verwenden nun die MATLAB Routine **eig** um Eigenwerte und -vektoren dieser 6×6 matrix anzusehen:

```
>> [v,d]=eig(H)

v =

   -0.1023    0.2856   -0.5004    0.6469    0.3319   -0.3592
   -0.4554    0.6469   -0.3113   -0.2856   -0.3958    0.1976
   -0.7510    0.0000    0.5182    0.0000    0.3768   -0.1599
   -0.4554   -0.6469   -0.3113    0.2856   -0.3958    0.1976
   -0.1023   -0.2856   -0.5004   -0.6469    0.3319   -0.3592
   -0.0169   -0.0000   -0.1922    0.0000    0.5696    0.7989


d =
```

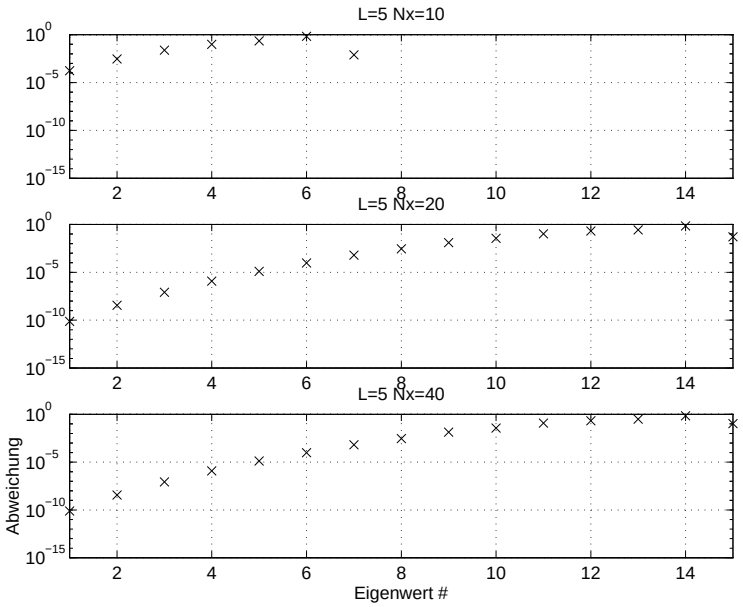

Abbildung 17: Abweichung im Spektrum des harmonischen Oszillators.

```
    0.4995         0         0         0         0         0
         0    1.5077         0         0         0         0
         0         0    2.4495         0         0         0
         0         0         0    3.7339         0         0
         0         0         0         0    4.2693         0
         0         0         0         0         0    7.4581

>> diary off
```

Die Matrix `d` enthält die Eigenwerte. Wir erkennen die unteren drei Werte $1/2, 3/2, 5/2$ mit erstaunlicher Genauigkeit, und sehen, daß sie nicht geordnet sind. Die jeweils darüberstehenden Spalten von `v` sind die Eigenvektoren. Wir sehen die korrekten (Anti)symmetrieeigenschaften unter Reflektion ($x_{k=3} = 0$).

Nun wollen wir größere Realisierungen wählen. Für verschiedene Wahlen von L und N_x plotten wir die Abweichung der numerischen Eigenwerte von den exakten in Abb.17 und Abb.18. Dazu wurden die Eigenwerte aus **eig** mit **sort** geordnet. Wir interpretieren die Bilder wie folgt: Für gegebene System-

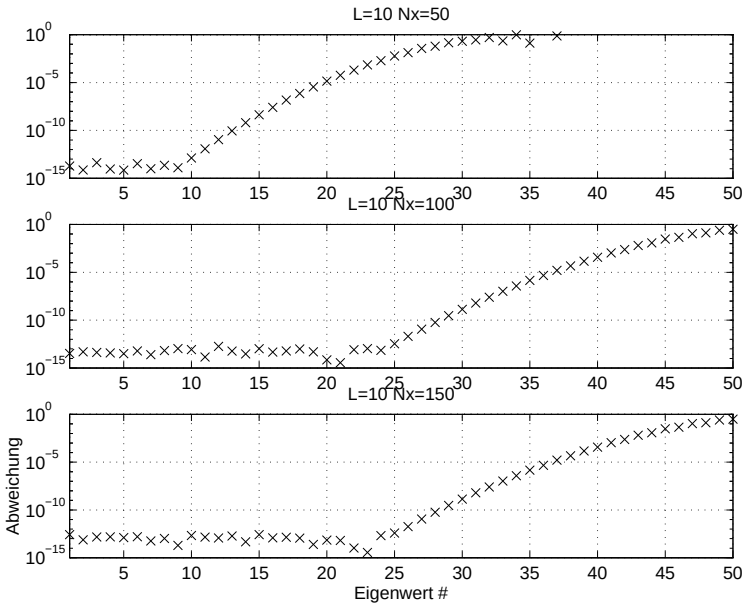

Abbildung 18: Abweichung im Spektrum des harmonischen Oszillators.

größe bekommt man eine gewisse Anzahl von unteren Anregungen genau. Die höheren spüren irgendwann den "Käfig" und werden verzerrt, egal wie fein man diskretisiert. Bei $L = 5$ bringen mehr als 20 Punkte ($\Delta x = 0.5$) nichts mehr. $L = 10$ und $N_x = 100$ ist offenbar keine schlechte Wahl, mit der man ca. 30 Eigenwerte auf 10^{-10} genau hat, die unteren 25 sogar maschinengenau. Dann nehmen die Abweichungen rasch zu. Die Diagonalisierung bei $N_x = 100$ braucht etwa 0.2 Sekunden auf einer HP 735.

Mit dem folgenden Stück code kann man auch die Eigenfunktionen bekommen, ordnen und dann plotten:

```
%
%  file evek.m
%
%  Untersuchung Eigenvektoren des Hamilton
%
%clear all;
clf;
```

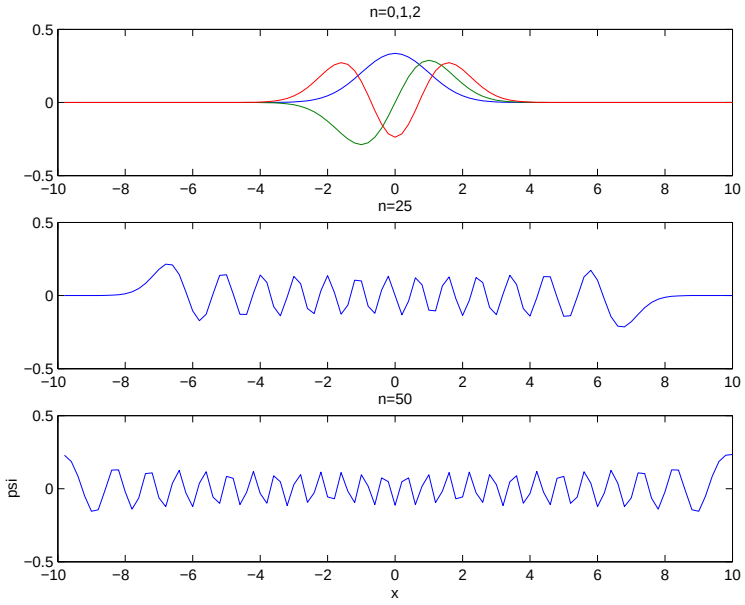

Abbildung 19: Einige numerische Wellenfunktionen ($L = 10, N_x = 100$)

```
L=10; Nx=100;
[dx,x,H] = ho_fou(L,Nx);

[v,d]=eig(H);          % Eigenwerte und -vektoren (spalten von v)
sp=diag(d);            % Spalte von Eigenwerten

% ordnen:
[sp,t]=sort(sp);
v=v(:,t(:));

subplot(3,1,1), plot(x,v(:,1:3)); title('n=0,1,2')
subplot(3,1,2), plot(x,v(:,26) ); title('n=25')
subplot(3,1,3), plot(x,v(:,51) ); title('n=50')
xlabel('x'); ylabel('psi')
```

Das Resultat ist in Abb.19 zu sehen. Es wird deutlich, daß bei $n = 50$ starke Effekte des endlichen Volumens sind und bei $n = 25$ gerade noch nicht. Erstaunlich ist, daß der zugehörige Eigenwert nur einen relativen Fehler

von 10^{-13} aufweist, obwohl Diskretisierungseffekte der Wellenfunktion schon deutlich sichtbar sind.

13.3 Zeitabhängige Probleme

Nachdem wir nun einige Kontrolle über den Oszillator Hamilton Operator dargestellt als endliche Matrix haben, wollen wir ihn benutzen um Wellenfunktionen in der Zeit evolvieren zu lassen. Schließlich hat H ja die Doppelbedeutung als Energie Observable und als Generator der Dynamik in der Schrödinger Gleichung

$$i\hbar\frac{\partial}{\partial t}\psi(t) = H\psi(t). \tag{13.13}$$

Wir arbeiten also im Schrödinger Bild (zeitabhängige Zustände, zeitunabhängige Operatoren). Unter $\psi(t)$ kann man sich entweder einen abstrakten Hilbert Raum Vektor oder eine Wellenfunktion $\psi(t,x)$ oder (am besten hier) gleich unsere diskretisierte genäherte Form $\psi_k(t)$ als N_x komponentigen Vektor vorstellen. Formal integriert ist (13.13) äquivalent zu (nun wieder $\hbar = 1$)

$$\psi(t) = \exp(-itH)\psi(0), \tag{13.14}$$

wobei man sich die Matrix Exponentiation in der Eigenbasis, wo H diagonal ist, denkt. Man kann die Matrix `v` von Eigenvektoren des letzten Abschnitts dazu verwenden zwischen Orts- und Eigenbasis hin- und herzutransformieren oder aber alles von der MATLAB Routine **expm** erledigen lassen. Diese exponenziert Matrizen im gerade diskutierten Sinn.

Wir betrachten nun das folgende Programm:

```
%
%  file zeit.m
%
%  Zeitentwicklung einer Gauss Funktion beim Harmonischen Oszillator
%
clear all; hold off; clf;
L=10; Nx=100;
[dx,x,H] = ho_fou(L,Nx);     % Hamilton

I = sqrt(-1);
T0 = 2*pi;                   % Periode
T  = T0/10;                  % Strosboskop Zeit
```

```
U = expm(I*T*H);              % Evolution ueber T

x_z=3; alfa=3;
psi = exp(-0.5*alfa*(x-x_z).^2).';  %Gauss Anfangszustand
psi = psi/norm(psi);                %Normieren
gauss=psi;
p = psi.*conj(psi);

plot(x,p,'-'); hold;
%gtext('0');
% gtext erlaubt die Kurven interaktiv per Maus zu bezeichnen;
% es stopp nach jeder Kurve und verlangt input! s. help
t=0;
for i=1:5
  psi = U*psi; t=t+T;
  p = psi.*conj(psi);
  plot(x,p,'-');
  % gtext(num2str(i));
end
xlabel('x'); ylabel('p');
disp('PAUSE'); pause;
hold off;clg;
% Zeitentwicklung des x-Mittelwertes
for i=1:100
  xmean(i) = real((psi'.*x)*psi); % psi-kreuz x psi; sowieso reell
  tplot(i)=t;
  psi=U*psi; t=t+T;
end
plot(tplot/T0,xmean); hold; plot(tplot/T0,xmean,'x')
xlabel('t/T0'); ylabel('<x>');
```

Als Anfangs Zustand wird hier ein Gauß Paket $\exp[-\alpha(x-x_z)^2/2]$ verwendet dessen Maximum bei $x_z = 3$ und dessen Breite $\alpha = 3$ ist. Dies ist also ein Zustand, der enger als der Grundzustand ist. Dies ist sicher kein Energie

Eigenzustand. Er wird mit der Zeit komplex, und wir plotten daher

$$p_k(t) = p(t, x_k) \propto |\psi_k(t)|^2, \quad \sum_k p_k = 1. \tag{13.15}$$

Die Oszillator Periode ist $T_0 = 2\pi$ in den gewählten Einkeiten. Die n'te Eigenkomponente oszilliert also $\propto \exp[-i(n+1/2)t]$, so daß $p_k(t)$ diese Periodizität hat unabhängig vom Anfangszustand. Wir bilden $U(T) = \exp(-iTH)$ mit $T = T_0/10$ und plotten die p_k für $t = 0, T, 2T \ldots T_0/2$, danach läuft man durch die gleichen Zustände wieder zurück. Man erhält hier ein stroboskopisches Bild des Quantensystems. Im Programm `zeit.m` wird nur eine Wellenfunktion ψ gespeichert. Man läßt sie dann jeweils mit $\psi \to U(T)\psi = \exp(-iTH)\psi$ um einen Schritt evolvieren (dabei haben wir H zuvor für $L = 10$ und $N_x = 100$ berechnet). Das Ergebnis wird dann gleich in ein Bild geplottet (siehe Abb.20). Es läßt sich beim Oszillator zeigen, daß eine Gauß'sche Wellenfunktion unter Zeitentwicklung stets Gaußisch bleibt[25]. Allerdings verändert sich, wie man sieht, i. a. die Breite (außer $\alpha = 1$). Nach $T_0/2$ ist das Teilchen klassisch am Umkehrpunkt, und hier ist die anfängliche Form der Wahrscheinlichkeitsverteilung wieder erreicht (nicht aber die Phasen in ψ), und die gleichen Verteilungen werden nun rückwärts angenommen bis die volle Periode durchlaufen ist.

Die Zeitentwicklung des x Mittelwertes über einige Perioden entspricht der klassischen Bewegung. Die Berechnung des Mittelwertes wird im zweiten Teil des Programms **zeit.m** durchgeführt. Die Zeitwentwicklung des Mittelwertes ist in der Abb.21 als Funktion von T/T_0 dargestellt.

13.4 Anharmonischer Oszillator

Es ist nun ein Leichtes, den harmonischen Term durch anharmonische zu ersetzen. Hier nehmen wir zum Beispiel mal willkürlich

$$\hat{H} = \frac{1}{2}(\hat{p}^2 + \hat{x}^4). \tag{13.16}$$

Dieser Hamilton Operator ist leicht auf viele Art und Weise numerisch zu behandeln, nicht aber analytisch. Wieder mit $L = 10, N_x = 100$ bekommen wir für die untersten 8 Eigenwerte

[25] Der exakte Integralkern zu $U(T)$ ist Gaußisch, und die Konvolution von Gauß Integralen führt wieder auf solche.

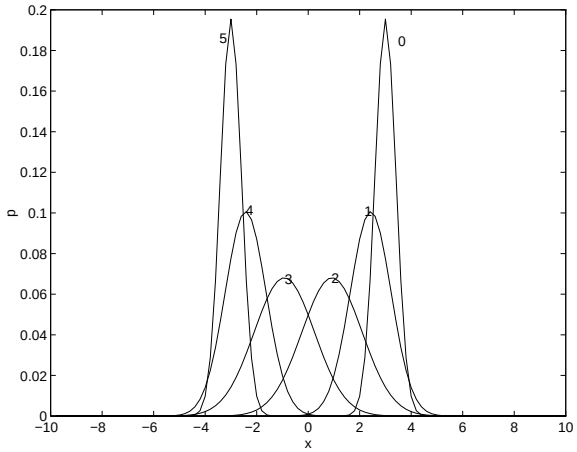

Abbildung 20: Zeitentwicklung von $|\psi|^2$ beim Gauß Paket ($L = 10$, $N_x = 100$).

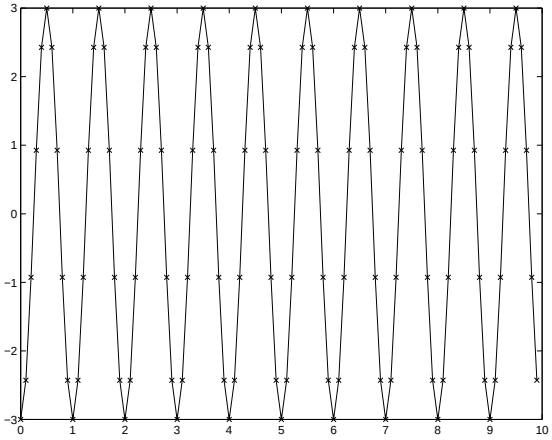

Abbildung 21: Zeitentwicklung des x-Mittelwertes ($L = 10$, $N_x = 100$).

```
>> ev0

ev0 =

   0.53018104524208
   1.89983651490030
   3.72784896899351
   5.82237275568909
   8.13091300942560
  10.61918645911819
  13.26423559184152
  16.04929885548410

>> quit
```

In dem man auch mal $L = 15$ nimmt, kann man abschätzen, daß der zu erwartende relative Fehler von 10^{-12} (Grundzustand) steigt auf etwa 10^{-7} beim größten gezeigten Wert. Durch Steigern von N_x kommt man zum Schluß, daß der Diskretisierungsfehler hier vernachlässigbar ist. Das Spektrum für diesen Fall ist deutlich verschieden vom harmonischen Oszillator. Die Niveaus sind nicht mehr äquidistant und alle höher.

13.5 Tunneln: stationäre Zustände

Ein wichtiges quantenmechanisches Problem, bei dem man weitgehend auf Näherungsverfahren oder Numerik angewiesen ist, stellt der Tunneleffekt dar. Als gängiges Beispiel mit einem Freiheitsgrad sei hier das System

$$\hat{H} = \frac{1}{2}\hat{p}^2 + \hat{V}(\hat{x}) \tag{13.17}$$

mit dem Doppelmulden–Potential

$$V(x) = (x - x_{min})^2(x + x_{min})^2/(8x_{min}^2) \tag{13.18}$$

behandelt. $V(x)$ hat zwei Minima $V(x) = 0$ bei $x = \pm x_{min}$, sie sind durch eine Barriere der Höhe

$$V(0) = \frac{1}{8}x_{min}^2 \tag{13.19}$$

getrennt. Der Normierungsfaktor von V ist so gewählt, daß $V''(\pm x_{min}) = 1$, d.h. jede Mulde für sich betrachtet ähnelt einem harmonischen Oszillator der Frequenz $\omega = 1$ (in natürlichen Einheiten).

Die Programme `dm_fou.m` und `dmevek.m` erledigen die Konstruktion des Hamilton–Operators und seine Diagonalisierung, es sind offensichtliche Abwandlungen der Routinen, die für den (an-)harmonischen Oszillator geschrieben wurden. Für drei verschiedene Werte von x_{min} erhält man so die 10 niedrigsten Eigenwerte:

```
D.M. Hamilton mit Fourier-diskretisiertem Laplace Operator:
Raumintervall [-10 , 10] mit 150 Punkten

     xmin = 2          xmin = 3          xmin = 5

     0.350239          0.460383          0.489498
     0.523186          0.473929          0.489498
     1.113986          1.132844          1.421839
     1.729921          1.369160          1.421932
     2.469131          1.863507          2.265520
     3.290646          2.386312          2.270646
     4.184444          2.981769          2.934615
     5.141611          3.629893          3.030664
     6.155675          4.325681          3.455764
     7.221563          5.064278          3.790578
```

Im ersten Fall sind die beiden Minima kaum getrennt ($V(0) = .5$), dagegen liegt im dritten Beispiel eine klare Tunnelsituation vor, denn es gibt Eigenwerte weit unterhalb der Schwelle von $V(0) = 3.125$, sie bilden enge Dubletts nahe bei den Eigenwerten des entsprechenden harmonischen Oszillators. Das mittlere Beispiel zeigt ebenfalls diese Struktur, aber weniger extrem, und wird im nächsten Abschnitt zur Illustration der Zeitentwicklung benutzt.

Einige Eigenfunktionen sind in Abb.22 und 23 dargestellt. Im Tunnelfall ($x_{min} = 5$) ist zu sehen, daß die niedrigen Eigenfunktionen abwechselnd durch gerade und ungerade Überlagerungen von Oszillator–Moden in den beiden Minima angenähert werden, bei $x_{min} = 2$ ist das natürlich nicht der Fall.

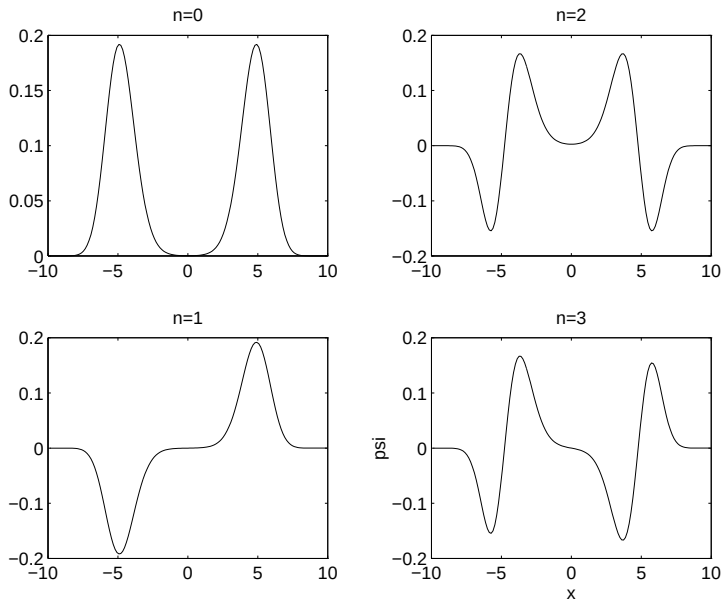

Abbildung 22: Die vier niedrigsten Eigenfunktionen für $x_{min} = 5$

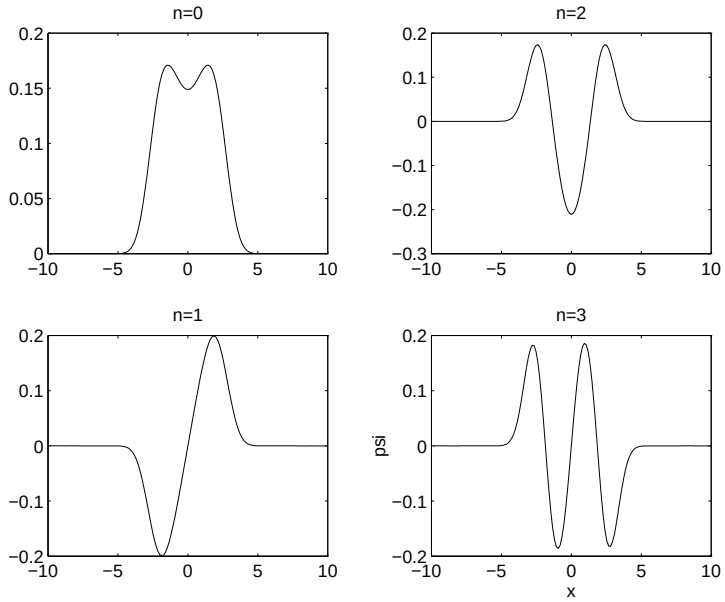

Abbildung 23: Die vier niedrigsten Eigenfunktionen für $x_{min} = 2$

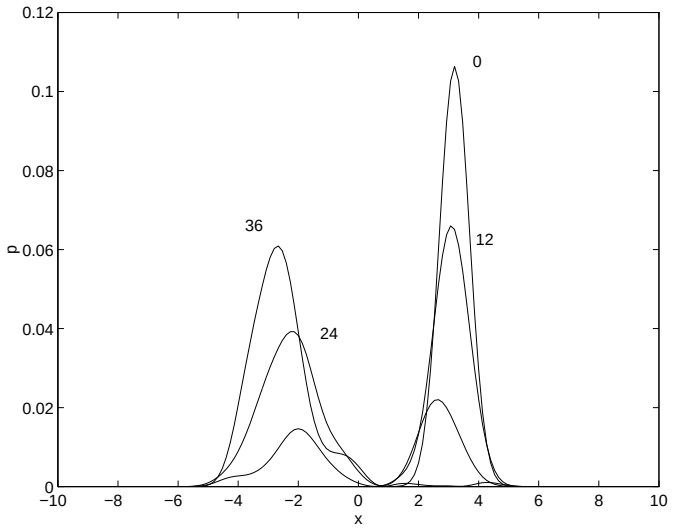

Abbildung 24: $p(x,t) = |\psi(x,t)|^2$ zu vier verschiedenen Zeiten t/T_0

13.6 Tunneln: Zeitentwicklung

Wir wollen nun, wie schon zuvor beim harmonischen Oszillator, den diskretisierten Hamiltonoperator benutzen, um den Zeitentwicklungsoperator

$$U(t) = \exp(-itH) \tag{13.20}$$

zu bilden und damit Lösungen der zeitabhängigen Schrödingergleichung darzustellen. Dies ist im Programm `dmzeit.m` implementiert. Als Anfangszustand dient wieder ein (normiertes) Gauß–Paket $\sim \exp[-\alpha(x-x_z)^2/2]$. In Abb.24 ist die Entwicklung der Aufenthaltswahrscheinlichkeit dargestellt für ein Tunnelpotential mit $x_{min} = 3$. Für den Anfangszustand ist $\alpha = 2$ und $x_z = 3.2$ angesetzt, d.h. man erwartet schon ein oszillierendes Verhalten in der einfachen Mulde – der Grundzustand hätte $\alpha = 1$ und $x_z = 3$. Dem ist nun ein Übergang in die andere Mulde überlagert. Die zeitliche Entwicklung des Mittelwerts $< x >$ in Abb.25 zeigt das noch deutlicher.

Die Tunnelzeit kann man quantitativ durch folgende einfache Überlegung verstehen: Wenn der Anfangszustand näherungsweise als Überlagerung der beiden niedrigsten Eigenfunktionen dargestellt wird, d.h. $\psi(x, t=0) \sim \psi_0(x) + \psi_1(x)$ nach Abb.22, dann folgt eine Zeitentwicklung

$$\psi(x,t) \quad \sim \quad e^{-iE_0 t}\psi_0(x) + e^{-iE_1 t}\psi_1(x)$$

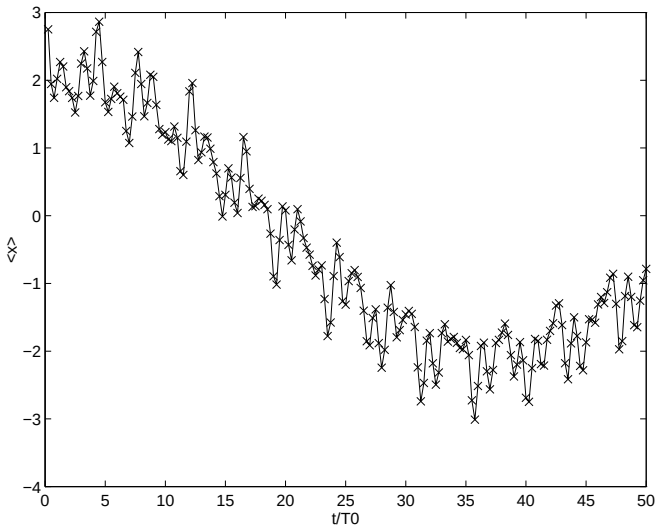

Abbildung 25: Zeitentwicklung von $< x >$ beim Tunneln

$$= \quad e^{-iE_0 t}\left(\psi_0(x) + e^{-i(E_1-E_0)t}\psi_1(x)\right). \tag{13.21}$$

Ein Übergang in die andere Mulde liegt vor, wenn $\psi(x,\tau) \sim \psi_0(x) - \psi_1(x)$, also zuerst bei

$$\tau = \frac{\pi}{E_1 - E_0}. \tag{13.22}$$

Mit der im vorigen Abschnitt angegebenen Aufspaltung ergibt sich $\tau = 74.1\pi$ oder $\tau/T_0 = 37.0$ ($T_0 = 2\pi$ war die Oszillatorperiode), in guter Übereinstimmung mit Abb.25.

Wenn bei der Zerlegung des Anfangszustands höhere Eigenfunktionen wesentlich beitragen, dann werden diese Komponenten schneller tunneln, weil $E_3 - E_2 > E_1 - E_0$ usw. Die Wellenfunktion wird sich bald auf beide Mulden verteilen, das Tunnelbild ist dann weniger deutlich.

14 Quantenmechanische Streuung in einer Dimension

In diesem Abschnitt wollen wir Streulösungen der Schrödingergleichung in einer Dimension diskutieren. Wie üblich wiederholen wir die Begriffsbildungen aus Physik 4 bzw. Quantenmechanik I, reproduzieren numerisch Resultate für die dort typischerweise behandelten einfachen Potezialstufen, um dann zu Allgemeinerem fortzuschreiten.

14.1 Wellenpakete mit Matrix Quantenmechanik

Wir wollen zunächst mit Hilfe der Matrix Quantenmechanik des letzten Abschnitts die Ausbreitung eines freien Teilchen nachbilden. Wir wählen Einheiten mit $\hbar = 1$, müssen also nicht zwischen Impulsen und Wellenzahlen unterscheiden, und setzen auch $m = 1$. Wir präparieren eine Gauss Anfangswellenfunktion

$$\psi(x, t = 0) \propto \int dk \exp(-b^2(k - k_0)^2 + ikx) \propto \exp\Big(-\frac{x^2}{4b^2} + ik_0 x\Big), \quad (14.1)$$

zentriert um Impuls k_0 mit Breite b im Ort. Im endlichen diskretisierten System des letzten Abschnitts gibt es zwar nur diskrete Impulse $p \in [-\pi/\Delta x, \pi/\Delta x)$ im Abstand π/L, die Gauss Integrale werden aber gut genähert solange $\Delta x \ll b \ll L$ gilt. Es gibt nun zwei Zeitskalen, wenn diese Anfangswellenfunktion mit dem freien Hamiltonoperator evolviert

$$\psi(t) = \exp\Big(-it\,\frac{1}{2}\hat{p}^2\Big)\psi(0). \quad (14.2)$$

Das ist zum Einen die Zeit $T = L/k_0$ die auch klassisch zur Bewegung durch das gegebene (halbe) Intervall benötigt wird. Zum anderen wissen wir, dass das Paket quantenmechanisch breitfliesst. Die Skala, auf der das passiert ist durch $t_f \sim b^2$ gegeben. Um den Weg des Wellenpakets durch unser Volumen zu verfolgen sollte zumindest nicht $T \gg t_f$ gelten. Wir betrachten folgenden Kompromiss,

$$b = \sqrt{\Delta x L}, \quad k_0 = \frac{1}{2}\pi/\Delta x, \quad (14.3)$$

der $\Delta x/b = b/L = \sqrt{\Delta x/L}$ und $T \sim t_f$ ergibt. Die Evolution in Schritten von $T/5$ ist in Fig.26 im oberen Bild zu sehen mit $L = 10$ und $N_x = 100$.

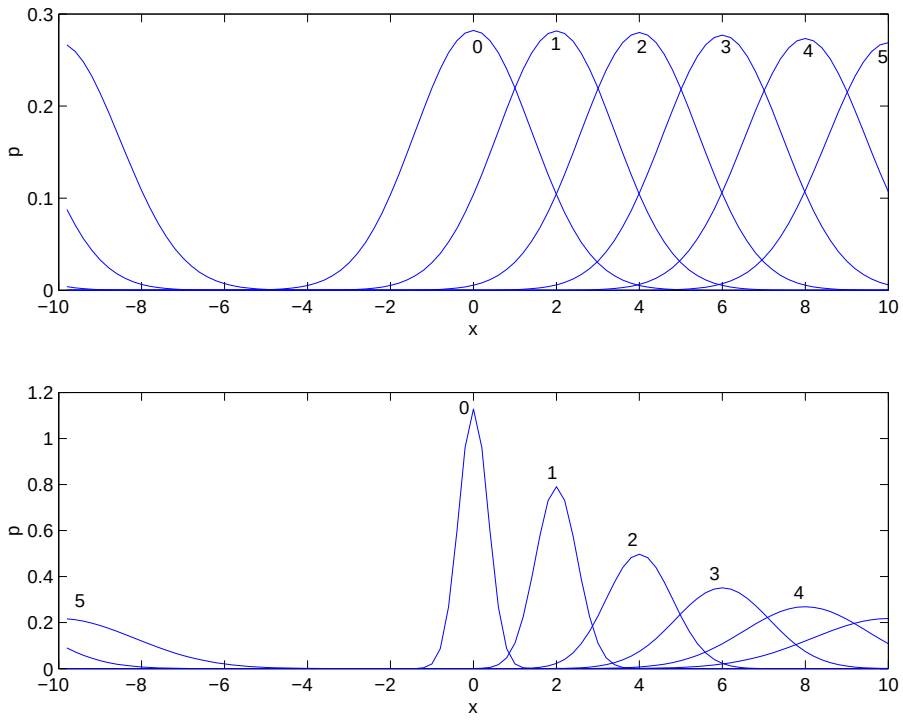

Abbildung 26: Evolution eines freien Wellenpakets mit zwei verschiedenen Breiten.

Gezeigt ist jeweils die Wahrscheinlichkeitsdichte $p(x,t) \propto |\psi(x,t)|^2$ in Kontinuumnormierung, $\sum_x \Delta x\, p(x) = 1$. Im unteren Bild ist $b \to b/4$ ersetzt. Deutlich werden auch die periodischen Randbedingungen.

Als Nächstes bauen wir eine Potentialstufe der Höhe V_0 und der Breite $2w = 2$ symmetrisch über $x = \in (-w, +w)$ auf. Mit $L = 20$ und $N_x = 200$ wird ein Wellenpaket der Breite $\sqrt{\Delta x L}/2$ startend bei $x = -13$ von links mit Impuls $k_0 = \pi/(4\Delta x)$ einlaufen gelassen. Fig.27 zeigt die Situation mit $V_0 = k_0^2$ (kinetische Energie $V_0/2$), Fig.28 bei $V_0 = 0.7 \times k_0^2/2$.

14.2 Stationäre Streulösungen

Im den eben betrachteten Lösungen der zeitabhängigen Schrödingergleichung (frei und mit Potential) bestand die Anfangskonfiguration aus überlagerten ebenen Wellen $\exp(ikx)$ mit k nahe einem $k_0 > 0$ (nur dort gab es nennens-

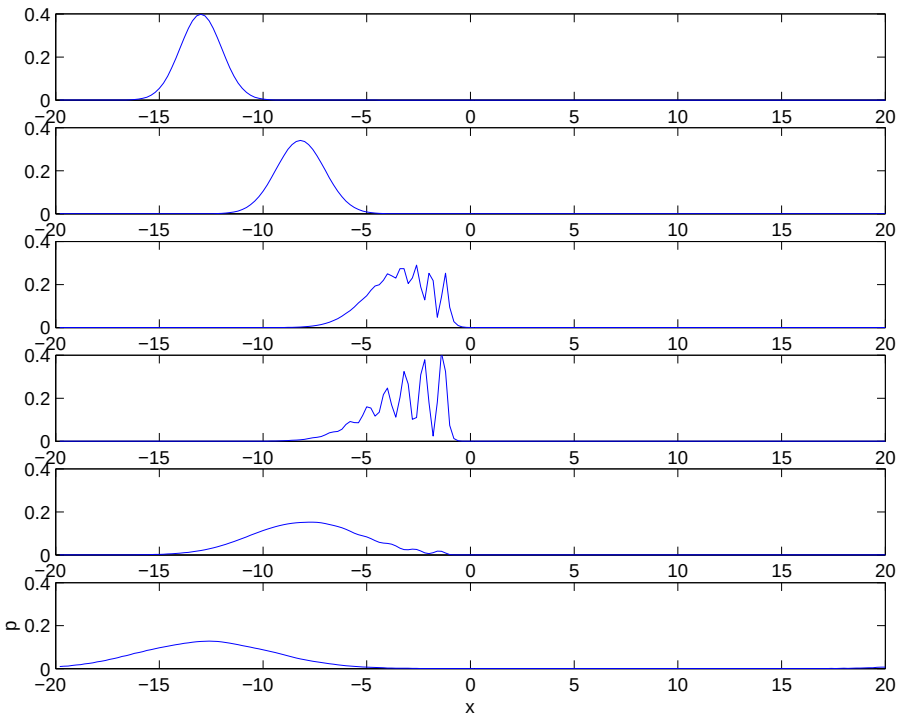

Abbildung 27: Streuung an einer Potentialstufe $V_0 = 2E$.

werte Amplituden), die ein nach rechts laufendes Wellenpaket bildeten. Es handelt sich hier nicht um Energieeigenzustände. Solche sind im freien Fall allerdings die einzelnen Komponenten,

$$H\exp(ikx) = \frac{1}{2}\hat{p}^2\exp(ikx) = E\exp(ikx), \quad E = \frac{k^2}{2}. \tag{14.4}$$

Es handelt sich jedoch um uneigentliche Elemente des Hilbertraums, da sie (für $L = \infty$) nicht normierbar sind.

Nach dem Auftreffen auf ein Potential, das auf das Intervall $(-w, +w)$ beschränkt ist, $V(x) = 0$ für $|x| > w$, beobachtet man links- und rechtslaufende, reflektierte Pulse bei $x < -w$ und nur rechtslaufende, transmittierte Pulse bei $x > +w$. Auch dieser Gesamtvorgang lässt sich durch Überlagerung von (uneigentlichen) Energieeigenzuständen, den Streuzuständen $\phi_k(x)$, darstellen

$$\psi(x,t) = \int dk\tilde{\psi}(k)\phi_k(x)\exp(-iE(k)t), \quad H\phi_k = E(k)\phi_k, \tag{14.5}$$

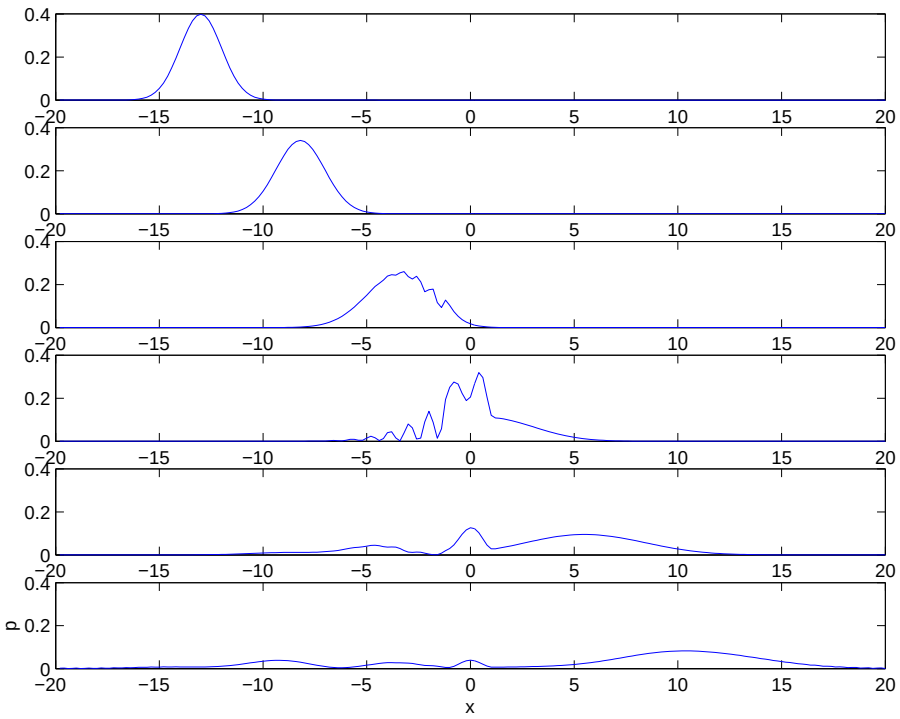

Abbildung 28: Streuung an einer Potentialstufe $V_0 = 0.7E$.

wobei hier vorläufig im unendlichen Volumen und im Kontinuum argumentiert wird und somit über $k \in (-\infty, +\infty)$ integriert wird.

Für $|x| > w$ löst $\phi_k(x)$ die freie Schrödingergleichung mit Energie $E(k)$, muss dort also aus $\exp(\pm ikx)$ mit $E(k) = k^2/2$ bestehen. Die Randbedingungen des Streuproblems erfordern es, unter den zwei unabhängigen Lösungen zum Eigenwert E diejenige zu nehmen, die für große x nur die Komponente $\exp(+ikx)$ enthält. Man normiert nun so, dass gilt

$$\phi_k(x) = \begin{cases} \exp(ikx) + r(k)\exp(-ikx) & \text{für } x < -w \\ t(k)\exp(ikx) & \text{für } x > +w \end{cases} \tag{14.6}$$

Es ist ein wenig schwierig in diesen Lösungen noch den Streuprozess zu erkennen, da die Amplitude ja im Aussenraum des Potentials höchstens an isolierten x verschwindet und der Zeitaspekt nicht mehr da ist. In der Über-

lagerung (14.5) gilt z. B. links ($x < w$)

$$\psi(x,t) = \int dk \tilde{\psi}(k)(\exp(i(kx - Et) + r(k)\exp(-i(kx + Et)). \qquad (14.7)$$

Unter der Annahme, dass $\tilde{\psi}(k)$ wieder bei k_0 konzentriert ist, ergeben sich in diesem Bereich nur grosse Beiträge, wenn bei $k = k_0$ auch noch die Phase stationär ist. Dies ist für einen der beiden Beiträge der Fall, wenn

$$x = \pm k_0 t \qquad (14.8)$$

gilt. Natürlich muss gleichzeitig $x < -w$ gelten, da wir die dafür gültige Form genommen haben. Damit bekommen wir links konstruktive Interferenz nur wenn entweder $t < -w/k_0$ und wir haben zu dieser Zeit eine nach rechts einlaufende Gruppe, oder wenn $t > w/k_0$ für eine linkslaufende reflektierte Gruppe. Analog ergeben die Randbedingungen nach Überlagerung den nach rechts transmittierten Puls zu später Zeit.

14.3 Transfermatrix Formalismus

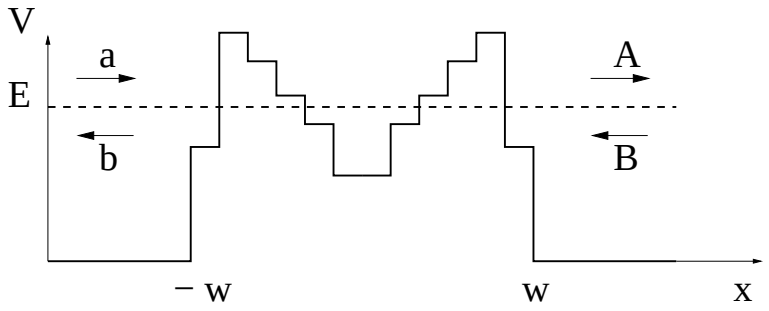

Abbildung 29: Streuung an einem symmetrischen Potential.

Wir betrachten nun ein beliebiges reelles Potential $V(x)$, das ausserhalb des Intervalles $(-w, w)$ verschwindet. Wir verallgemeinern die gerade diskutierten Randbedingungen beim Streuproblem ein wenig und suchen Eigenzustände ϕ zum Hamiltonoperator, die die asymptotische Form

$$\phi(x) = \begin{cases} a\,\mathrm{e}^{ik(x+w)} + b\,\mathrm{e}^{-ik(x+w)} & \text{für } x < -w \\ A\,\mathrm{e}^{ik(x-w)} + B\,\mathrm{e}^{-ik(x-w)} & \text{für } x > +w \end{cases} \qquad (14.9)$$

haben. Die Amplituden a und b beziehen sich auf die einlaufende und reflektierte ebene Welle mit Energie $E = k^2/2$ und analog die Amplituden A und B auf die transmittierte und eine zusätzliche ins Potential einlaufende Welle. Nur zwei dieser vier Amplituden sind frei wählbar (z. B. A, B). Wegen der Linearität der stationären Schrödingergleichung gibt es dann eine Matrixbeziehung

$$\begin{pmatrix} a \\ b \end{pmatrix} = M_V \begin{pmatrix} A \\ B \end{pmatrix} , \tag{14.10}$$

welche die Transfermatrix M_V definiert. Die Situation ist schematisch in der Abbildung 29 dargestellt. Wir werden sehen, dass M_V die gesamte Streuinformation des Potentials V in allgemeiner Form enthält.

Wir wollen zuerst einige allgemeine Eigenschaften von M_V untersuchen. Da das Potential V reell ist, ist mit ϕ auch ϕ^* eine Lösung der zeitunabhängigen Schrödinger Gleichung. Diese Eigenschaft ist äquivalent zur Zeitumkehrinvarianz und impliziert, dass zusammen mit (14.10) stets gelten muss

$$\begin{pmatrix} b^* \\ a^* \end{pmatrix} = M_V \begin{pmatrix} B^* \\ A^* \end{pmatrix} .$$

Daraus folgt $m_{11} = m_{22}^*, m_{12} = m_{21}^*$ und M_V hat die Form

$$M_V = \begin{pmatrix} m_{11} & m_{21}^* \\ m_{21} & m_{11}^* \end{pmatrix} , \tag{14.11}$$

mit komplexen Parametern m_{11} und m_{21}.

Für stationäre Lösungen ist die Wahrscheinlichkeitsdichte zeitunabhängig und der Teilchenstrom erhalten

$$\frac{\partial j}{\partial x} = 0, \quad j = \frac{1}{2i}\left(\phi^* \frac{\partial \phi}{\partial x} - \phi \frac{\partial \phi^*}{\partial x}\right) \tag{14.12}$$

Wenn wir ihn für große und kleine x auswerten, finden wir $|a|^2 - |b|^2 = |A|^2 - |B|^2$, gleiche Differenz von rechts- und linkslaufender Intensität. Für die Matrix M_V impliziert dies $|m_{11}|^2 - |m_{12}|^2 = 1$ oder $\det M_V = 1$.

Durch Anwendung von M_V auf den normalen Streuprozess und Vergleich von (14.6) mit (14.9) finden wir

$$m_{11} = \exp(-i2kw)/t, \quad m_{21} = r/t \tag{14.13}$$

mit $|t|^2 + |r|^2 = 1$ als Folge von Stromerhaltung und Zeitumkehrinvarianz.

Wenn wir auch noch voraussetzen, dass das Potential $V(x)$ symmetrisch ist, $V(-x) = V(x)$, dann muss zusammen mit (14.10) stets auch

$$\begin{pmatrix} B \\ A \end{pmatrix} = M_V \begin{pmatrix} b \\ a \end{pmatrix}$$

gelten, was $M_V^{-1} = M_V^*$ impliziert und $m_{21}^* = -m_{21}$ bzw.

$$\frac{r^*}{t^*} = -\frac{r}{t}\,. \tag{14.14}$$

An dieser Stelle wollen wir noch eine wichtige Grösse, die zur Beschreibung von Streuprozessen benutzt wird, einführen. Es handelt sich um die Streumatrix. Sie verknüpft die Amplituden A und b der nach beiden Seiten auslaufenden Wellen mit den einlaufenden Amplituden a und B

$$\begin{pmatrix} A \\ b \end{pmatrix} = \mathrm{e}^{i2kw} S \begin{pmatrix} a \\ B \end{pmatrix}\,. \tag{14.15}$$

Die zusätzliche Phase ist eine Konvention, die $S = 1$ bei $V = 0$ bewirkt.

Zeitumkehrinvarianz und Teilchenzahlerhaltung allein liefern, dass man S als

$$S = \begin{pmatrix} t & r' \\ r & t \end{pmatrix}\,, \quad r' = -r^* \frac{t}{t^*} \tag{14.16}$$

schreiben kann. Die Streumatrix ist unitär,

$$S^{-1} = S^\dagger\,. \tag{14.17}$$

Für ein symmetrisches Potential gilt noch $r' = r$.

14.4 Stufenpotentiale

Wir betrachten ein stückweise konstantes Potential $V(x)$[26], d.h. es setzt sich aus Stufen und konstanten Abschnitten zusammen. Für die einzelnen Elemente erhält man Transfermatrizen, die sich zur Transfermatrix des Gesamtpotentials zusammenmultiplizieren. Hier erkennen wir den Vorteil des betrachteten allgemeineren Streuproblems.

[26] Man könnte allgemeinere Potentiale durch stückweise konstante Abschnitte approximieren.

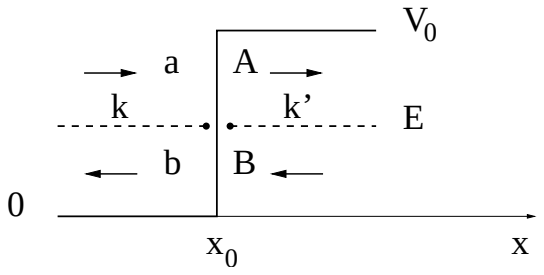

Abbildung 30: Beispiel für die Transfermatrix einer Potentialbarriere.

Wir nehmen als Beispiel die Stufe der Höhe V_0 (positiv bei Anstieg mit wachsendem x) an der Stelle x_0 wie in Abbildung 30. Die Lösung können wir ansetzen

$$\phi(x) = \begin{cases} a\,\mathrm{e}^{ik(x-x_0)} + b\,\mathrm{e}^{-ik(x-x_0)} & \text{für } x \le x_0 \\ A\,\mathrm{e}^{ik'(x-x_0)} + B\,\mathrm{e}^{-ik'(x-x_0)} & \text{für } x \ge x_0 \end{cases} \tag{14.18}$$

mit $k' = \sqrt{k^2 - 2V_0}$ bzw. $k = \sqrt{k'^2 + 2V_0}$, wobei wir hier und im Folgenden die Wurzel positiv reell oder positiv imaginär nehmen, je nach Vorzeichen des Argumentes. Stetigkeit von Wellenfunktion und ihrer Ableitung ergibt die Transfermatrix $M_\Delta(k,k')$

$$\begin{pmatrix} a \\ b \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1+\frac{k'}{k} & 1-\frac{k'}{k} \\ 1-\frac{k'}{k} & 1+\frac{k'}{k} \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = M_\Delta(k,k')\begin{pmatrix} A \\ B \end{pmatrix}. \tag{14.19}$$

Für einen konstanten Abschnitt der Breite d ergibt sich sich für Wellenzahl k' die diagonale Transfermatrix M_d

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \mathrm{e}^{-ik'd} & 0 \\ 0 & \mathrm{e}^{ik'd} \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} = M_d(k')\begin{pmatrix} A \\ B \end{pmatrix}. \tag{14.20}$$

Ein Kastenpotential wie im numerischen Beispiel besitzt nun die Gesamttransfermatix

$$M_V = M_\Delta(k,k')M_{2w}(k')M_\Delta(k',k) \tag{14.21}$$

Wenn wir ausmultiplizieren finden wir die Form (14.11) und durch Vergleich ergibt sich das bekannte Resultat

$$m_{11} = \frac{\mathrm{e}^{-i2kw}}{t} = \cosh(2ik'w) - \frac{1}{2}\left(\frac{k'}{k} + \frac{k}{k'}\right)\sinh(2ik'w) \tag{14.22}$$

$$m_{21} = \frac{r}{t} = \frac{1}{2}\left(\frac{k'}{k} - \frac{k}{k'}\right)\sinh(2ik'w). \tag{14.23}$$

Die Transmissionswahrscheinlichkeit $T = |t|^2$ ist gegeben durch

$$T^{-1} = 1 + \frac{1}{4}\sinh^2(2ik'w)\left(2 - (k/k')^2 - (k'/k)^2\right) \geq 1. \qquad (14.24)$$

Diese (Un)gleichungen gelten sowohl für den Fall mit rellem als auch mit imaginärem k'.

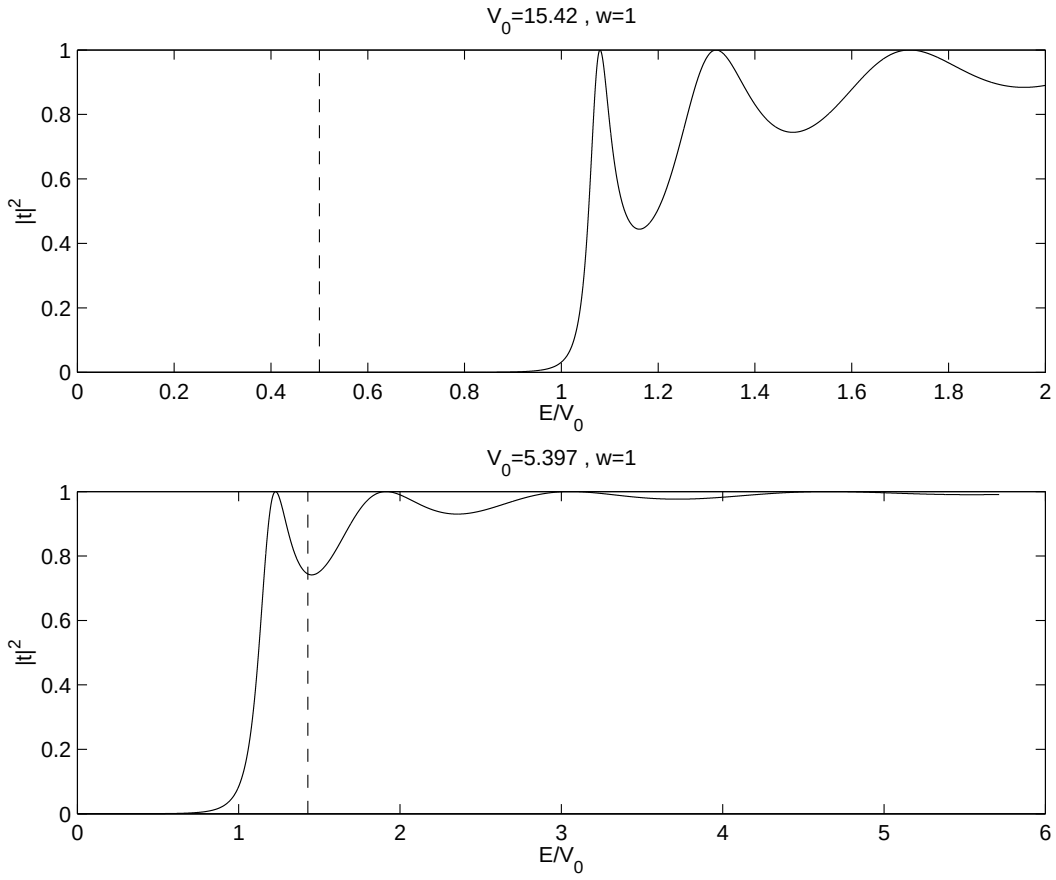

Abbildung 31: Transmissions Wahrscheinlichkeit für die Potentialstufe, numerische Experimente bei den vertikalen Linien.

Bei unserem numerischen Beispiel galt für das Zentrum des Wellenpaketes $k = \pi/(4\Delta x) \approx 3.93$ und im ersten Fall $k' = ik$, $T^{-1} = 1 + \sinh^2(2kw) \approx 1.7 \times 10^6$, also T exponentiell klein. Im zweiten Fall war $k' \approx 0.55k$, $T^{-1} =$

$1 + 0.41 \sin^2(2k'w) \approx 1.35$, also 74% Transmissionswahrscheinlichkeit, beides in qualitativer Übereinstimmung mit den 'Experimenten'. Für allgemeinere Energien ist T in Fig.31 gezeigt.

14.5 Streuung und endliche Volumen Effekte

Wir wollen nun Zusammenhänge diskutieren zwischen Streudaten $(t(k), r(k))$ und Effekten, die durch den Einschluss in ein endliches periodisches Volumen der Länge $2L$ bewirkt werden. Einen solchen Zusammenhang gibt es allgemein in der Quantenfeldtheorie, und er ist von Lüscher im Detail ausgearbeitet worden [13]. Bei Simulationen zur Elementarteilchen Physik ist dies äusserst interessant, da das endliche Volumen numerisch unvermeidlich ist, und eine direkte Simulation von Streuung durch Studium der Zeitentwicklung dort unmöglich ist. Man zieht also aus einer vermeintlich lästigen Beschränkung nützliche Information.

Betrachten wir zunächst den Fall ohne Potential. Wenn wir dennoch etwas künstlich eine "Streuung" über das Intervall $(-L, L)$ zur Energie $k^2/2$ betrachten, so gehört dazu die Transfermatrix

$$M_{2L} = \begin{pmatrix} \mathrm{e}^{-i2kL} & 0 \\ 0 & \mathrm{e}^{i2kL} \end{pmatrix}. \tag{14.25}$$

Im unendlichen Volumen sind beliebige $E \geq 0$ möglich. Eine Welle, die die Periode $2L$ hat, existiert nur, wenn die Transfermatrix einen Eigenwert 1 besitzt, $\exp(i2kL) = 1$, woraus bekanntlich die Quantisierung der Wellenzahl[27] $k = n\pi/L, n = 0, 1, 2, \ldots$ folgt. Zu jedem Wert $n > 0$ gibt es zwei entartete Eigenzustände, z. B. die rechts- und linkslaufenden Wellen, mit Energien $E_n = n^2\pi^2/2L^2$. Die sin und cos Kombinationen wären die Eigenfunktionen mit definierter Parität. Zu $E = 0$ gibt es mit der konstanten Funktion nur einen Zustand.

In Anwesenheit eines Potentials mit Transfermatrix M_V über den einhüllenden Bereich $(-w, w)$ wie bisher, das wir ab hier als symmetrisch voraussetzen, lautet die Eigenwertgleichung

$$\begin{pmatrix} \mathrm{e}^{-i2kL}/t & -r/t \\ r/t & \mathrm{e}^{i2kL}/t^* \end{pmatrix} \begin{pmatrix} 1 \\ \pm 1 \end{pmatrix} = \begin{pmatrix} 1 \\ \pm 1 \end{pmatrix} \tag{14.26}$$

[27] Bitte nicht endliches Volumen und Diskretisierung verwechseln. Hier geht es nur um ersteres, Δx verschwindet, und beliebig grosse Impulse sind möglich. Zur Numerik kann man später diskretisieren und muss dann Δx klein gegen alle anderen Skalen nehmen.

wobei die beiden Vorzeichen zu den beiden Wellenfunktionen definierter Parität $\pm$ gehören. Die Bedingung liefert auch hier eine Energiequantisierung, die durch die transzendente Gleichung

$$\mathrm{e}^{-i2kL} = t(k) \pm r(k) \tag{14.27}$$

gegeben ist. Nur für diese k bzw. E-Werte gibt es periodische Lösungen, wobei die Entartung im Allgemeinen aufgehoben wird. Kennt man umgekehrt einige Energien von Eigenzuständen im periodischen endlichen Volumen, so folgt daraus für die zugehörigen $k = \sqrt{2E}$ Werte die Phase auf der rechten Seite von (14.27). Für welche k man diese Information erhält, kann man a priori nicht wählen. Für verschiedene Wahlen von L kann man diese jedoch variieren. In Fig.32 sehen wir ein Beispiel für ein Kastenpotential der Höhe $V_0 = 3$, Breite $2w = 2$ im periodischen Volumen $2L = 30$. Die ausgezogenen Kurven für $t + r$ (ausgezogen) und $t - r$ (gestrichelt) stammen von den Formeln (14.22) und (14.23), während die Datenpunkte von den numerischen Eigenwerten von H mit Diskretisierung $N_x = 200$ kommen. Besonders stark variieren die Phasen nahe $k \approx 2.5$, wo $k^2/2 = V_0$ gilt. Die Qualität der Näherung variiert nicht sehr uniform mit N_x, was vermutlich daran liegt, dass unsere Diskretisierung der Ableitung nichtlokal ist oder dass das die Potentialsprünge hohen Fourier Komponenten entsprechen und schwierig aufzulösen sind.

Die Phasenfaktoren $t \pm r$ haben noch folgende Interpretation: die Streuphase bei Anwendung der S Matrix auf einlaufende symmetrische und antisymmetrische Wellen $(a, B = \pm a)$. Stellen wir uns ein Zweiteilchenproblem vor mit Potential $V(|x_1 - x_2|)$ zwischen Teilchen indentischer Masse $m_1 = m_2 = 2$. Wenn man nun wie üblich die freie Schwerpunktbewegung absepariert, so können wir unser Teilchen mit der effektiven Masse $1/m = 1/m_1 + 1/m_2 = 1$ als Bewegung der Relativkoordinate $x = x_1 - x_2$ interpretieren. Unterliegen die beiden Primärteilchen der Bose Statistik, so sind nur die ('Paritäts') geraden Zustände zu nehmen und $t+r$ ist die Zweiteilchen Streuphase. Entsprechendes gilt für den fermionischen Fall. In diesem Bild ist der Zusammenhang zwischen endlichen Volumeneffekten und der Streuphase besonders plausibel. Da die Teilchen gemeinsam auf einem Ring eingesperrt sind, müssen sie sich mit einer Frequenz $\propto 1/L$ begegnen und aneinander streuen. Durch diese Wechselwirkung wird die Energie gegenüber dem freien Fall verschoben, was durch die Sreuphasen in der so extrem einfachen Beziehung (14.27) quantitativ beschrieben wird.

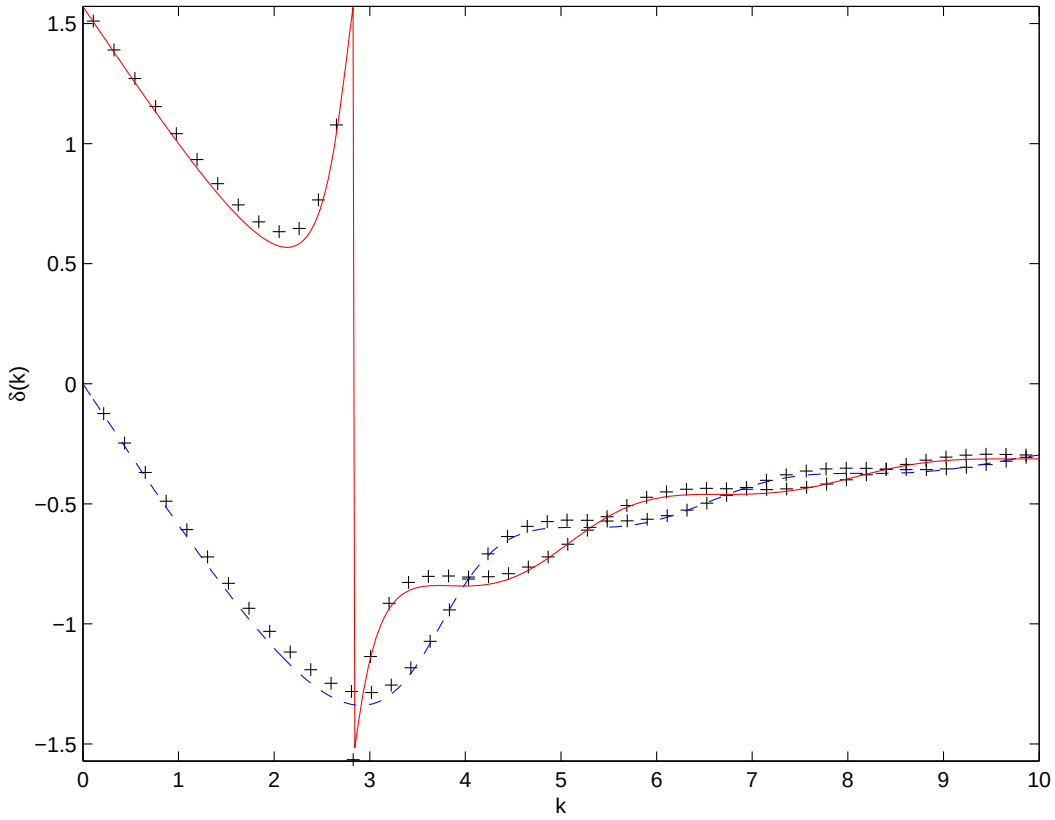

Abbildung 32: Streuphasen $\exp(2i\delta) = t \pm r$ (oben $+$, unten $-$) für das Kastenpotential, exakt und aus Energien im endlichen Volumen.

Offensichtlich können wir nun recht effektiv die Streuphasen für beliebige Potentiale numerisch über das endliche Volumen studieren, solange eine hinreichend gute Diskretisierung möglich ist. Ein Beispiel einer allgemeineren solchen Studie in einer 2-dimensionalen Quantenfeldtheorie ist in [14] zu finden, wo eine theoretisch unter gewissen Annahmen vorgeschlagene Form der S-Matrix numerisch überprüft wurde.

15 Pfadintegrale

Die Behandlung der Zeitentwicklung in den beiden vorigen Kapiteln gibt einen Zugang zum Konzept des Pfadintegrals, wie es in den 40'er Jahren von Feynman entwickelt worden ist. Wir wollen die Gelegenheit am Semesterende benutzen, diese Idee ohne numerische Implementierung etwas näher zu diskutieren.

Die Formulierung der Quantenmechanik durch Pfadintegrale bietet eine interessante Alternative zum Operatorformalismus im Hilbertraum. Symmetrien, z.B. relativistische Invarianz, sind einfacher zu überblicken. In der Quantenfeldtheorie und Quantenstatistik sind einige Näherungsverfahren (Störungstheorie, semiklassische Entwicklung) durch Pfadintegraldarstellungen inspiriert worden. Sie sind fast immer der Ausgangspunkt zur numerischen Simulation von Systemen mit vielen Freiheitsgraden.

15.1 Definition des Pfadintegrals

Die Entwicklung einer (diskretisierten) Wellenfunktion ψ über ein Zeitintervall T wurde im vorletzten Kapitel mit Hilfe der unitären Matrix

$$U(T) = \exp(-\frac{i}{\hbar}HT) \tag{15.1}$$

geschrieben als

$$\psi_i = \sum_j U(T)_{ij}\psi_j, \tag{15.2}$$

wobei sich die Indizes $\{i\}$ auf die Koordinaten $\{x_i\}$ bezogen. Die MATLAB –Programme machten auch von der Tatsache Gebrauch, daß sich $U(T)$ aus kleineren Zeitschritten $\epsilon = T/N$ zusammensetzen läßt gemäß

$$U(T) = U(\epsilon)^N. \tag{15.3}$$

Das bedeutet in Matrix–Notation:

$$U(T)_{i_N i_0} = \sum_{i_1 \ldots i_{N-1}} U(\epsilon)_{i_N i_{N-1}} \ldots U(\epsilon)_{i_2 i_1} U(\epsilon)_{i_1 i_0} \tag{15.4}$$

oder in quantenmechanischer Schreibweise ($\psi_i =< x_i|\psi >$ etc.):

$$< x_N|U(T)|x_0 >= \sum_{x_1 \ldots x_{N-1}} < x_N|U(\epsilon)|x_{N-1} > \ldots < x_2|U(\epsilon)|x_1 >< x_1|U(\epsilon)|x_0 > . \tag{15.5}$$

Diese Form macht deutlich, daß die Summationen, die von den Matrixmultiplikationen herrühren, auch als Einschübe von

$$1 = \sum_{x_i} |x_i >< x_i| \tag{15.6}$$

verstanden werden können. Die Vielfach–Summation in Glg.(15.5) wird nun interpretiert als "Summe über alle Pfade von x_0 nach x_N" in Zeitschritten ϵ.

Um die Notation zu vereinfachen, arbeiten wir von nun an wieder mit kontinuierlichen Ortskoordinaten, schreiben also

$$1 = \int dx \, |x >< x| \tag{15.7}$$

und die Pfad*summe* wird zum Pfad*integral*.

Jeder Schritt entlang des Weges trägt einen "Entwicklungskern"

$$< x_2|U(\epsilon)|x_1 > \;\; = \;\; < x_2| \exp(-\frac{i}{\hbar}\epsilon H)|x_1 > \tag{15.8}$$

zum (komplexen!) Gewichtsfaktor bei. Ihn wollen wir nun für Hamilton–Operatoren der gängigen Form

$$H = \frac{1}{2m}\hat{p}^2 + V(\hat{x}) \tag{15.9}$$

ausrechnen[12]. Das scheitert auf den ersten Blick an den nicht–kommutierenden Operatoren im Exponenten, aber für kleine Zeiten ϵ hilft die *Campbell–Baker–Hausdorff–Formel* weiter:

$$e^A e^B \;\; = \;\; \exp\left(A + B + \frac{1}{2}[A,B] + \frac{1}{12}[A-B,[A,B]] + ...\right)$$

Besonders attraktiv ist hier die symmetrische Version

$$e^{A/2} e^B e^{A/2} \;\; = \;\; \exp\left(A + B - \frac{1}{24}[A+B,[A,B]] + ...\right)$$

denn sie zeigt:

$$\exp(-\frac{i\epsilon}{2\hbar}V(\hat{x}) \exp(-\frac{i\epsilon}{2m\hbar}\hat{p}^2) \exp(-\frac{i\epsilon}{2\hbar}V(\hat{x})) \;\; = \;\; \exp\left(-\frac{i}{\hbar}\epsilon H + \mathcal{O}(\epsilon^3)\right)$$

Mit der linken Seite als Näherung für $U(\epsilon)$ kommt man nun weiter:

$$< x_2|U(\epsilon)|x_1 > \;\; \approx \;\; \exp(-\frac{i\epsilon}{2\hbar}V(x_2)) < x_2| \exp(-\frac{i\epsilon}{2m\hbar}\hat{p}^2)|x_1 > \exp(-\frac{i\epsilon}{2\hbar}V(x_1))$$

denn das verbleibende Matrixelement lässt sich mit Hilfe der Impuls–Eigenzustände $|p>$ berechnen:

$$\begin{aligned} < x|p > &= \exp(\frac{i}{\hbar}px). \\ < p'|p > &= 2\pi\hbar\delta(p'-p) \\ 1 &= \int \frac{dp}{2\pi\hbar}|p >< p| \\ \Rightarrow < x_2|\exp(-\frac{i\epsilon}{2m\hbar}\hat{p}^2)|x_1 > &= \int \frac{dp}{2\pi\hbar} < x_2|p > \exp(-\frac{i\epsilon}{2m\hbar}p^2) < p|x_1 > \\ &= \int \frac{dp}{2\pi\hbar}\exp\left(-\frac{i\epsilon}{2m\hbar}p^2 + \frac{i}{\hbar}p(x_2-x_1)\right) \\ &= C\exp\left(\frac{i}{\hbar}\frac{m}{2\epsilon}(x_2-x_1)^2\right) \qquad (15.10) \\ \text{mit} \quad C &= \left(\frac{m}{2\pi i\hbar\epsilon}\right)^{1/2} \qquad (15.11) \end{aligned}$$

So entsteht

$$< x_2|U(\epsilon)|x_1 > \approx C\exp\frac{i\epsilon}{\hbar}\left\{\frac{m}{2}\left(\frac{x_2-x_1}{\epsilon}\right)^2 - \frac{1}{2}V(x_1) - \frac{1}{2}V(x_2)\right\} \qquad (15.12)$$

Im Exponenten erkennt man eine diskretisierte Form der Lagrangefunktion

$$L(\dot{x},x) = \frac{m}{2}\dot{x}^2 - V(x) \qquad (15.13)$$

wieder. Über den gesamten Weg addiert sich das zu

$$\epsilon\left\{\sum_{k=0}^{N}\frac{m}{2}\left(\frac{x_{k+1}-x_k}{\epsilon}\right)^2 - \frac{1}{2}V(x_0) - \sum_{k=1}^{N-1}V(x_k) - \frac{1}{2}V(x_N)\right\} \qquad (15.14)$$

Darin erkennt man eine diskretisierte Form der *Wirkung*

$$S[x(t)] = \int dtL(\dot{x}(t),x(t)), \qquad (15.15)$$

wobei der Potentialterm an den Stützstellen $x_0,..,x_N$ aufsummiert wird (bei x_0 und x_N nur der halbe Wert) und der kinetische Term einer gleichförmigen Bewegung von Punkt zu Punkt entspricht.

So gelangen wir endgültig zur Pfadintegral–Darstellung des Zeitentwicklungs–Operators:

$$< x_N|U(T)|x_0 >= \int_{x_0}^{x_N} Dx(t) \exp\{\frac{i}{\hbar}S[x(t)]\}. \qquad (15.16)$$

Anfang und Ende des Pfades sind wie angegeben festzuhalten, das Integrationsmaß ist am besten in der diskretisierten Form zu denken, in der wir es auch hergeleitet haben:

$$Dx(t) = \lim_{N\to\infty} C^N dx_1 dx_2 \dots dx_{N-1}. \qquad (15.17)$$

15.2 Klassischer Grenzfall

Ein Vorzug der Pfadintegraldarstellung liegt darin, daß man ein intuitives Bild bekommt, unter welchen Bedingungen sich Teilchen klassisch verhalten, d.h. sich entlang klassischer Bahnen bewegen.

Da die Pfade in Glg.(15.16) mit einem Phasenfaktor $\exp\{\frac{i}{\hbar}S[x(t)]\}$ gewichtet sind und deshalb Kompensationen durch Interferenz vorkommen können, ist nicht klar, welche Wege womöglich den Hauptbeitrag zum Integral geben. Kandidaten für dominante Pfade sind aber solche, bei denen die Wirkung stationär ist:

$$\delta S[x] = 0 \quad \text{wobei} \quad \delta x(0) = \delta x(T) = 0. \qquad (15.18)$$

Das ist aber, wie man in der Mechanik lernt, gerade das Hamilton'sche Wirkungsprinzip, das zu den klassischen Bewegungsgleichungen führt. In der Umgebung einer solchen Bahn $x_c(t)$ ändert sich S erst in der Ordnung $(x - x_c)^2$, und diese Fluktuationen werden (grob gesagt) unterdrückt, wenn $\hbar$ "klein" ist (wogegen?). In diesem Falle tragen hauptsächlich Pfade zum Integral bei, die in einem engen "Schlauch" entlang der klassischen Bahn liegen. Andere Wege werden sich durch die schnell oszillierende Phase im Integral kompensieren.

15.3 Wick-Rotation

Um das Problem der schnell oszillierenden Phase im Pfad-Integral zu umgehen betrachtet man die sogenannte Wick-Rotation. Dabei wird eine imaginare Zeit eingeführt, indem wir T durch $-i\tau$ ersetzen.

Dies führt zu einem Problem der (klassischen) statistischen Mechanik: Aus dem Pfadintegral 14.16 wird auf diese Weise eine Zustandssumme. Damit

wird das Problem mit den Methoden der statistischen Mechanik behandelbar; z.B. der Monte-Carlo-Simulation. Insbesondere in der Quantenfeldtheorie hat sich dieser Zugang bewährt. In der CPII werden wir dieses Problem am Ende der Vorlesung aufgreifen.

Literatur

[1] Stephen Wolfram, *Undecidability and intractability in theoretical physics*, Phys.Rev.Lett.54:735,1985.

[2] W. Cheney und D. Kincaid, *Numerical Mathematics and Computing*, Brooks/Cole Publishing Company, Pacific Grove, California

[3] W. H. Press, S. A. Teukolsky, W. T. Vetterling und B. P. Flannery, *Numerical Recipes*, Cambridge University Press
Von diesem nützlichen Buch ("Bibel") gibt es verschiedene Ausgaben mit Programmen in Fortran oder C

[4] G. H. Golub und C. F. van Loan, Matrix Computations, Johns Hopkins University Press, 1989

[5] M. Abramowitz und I. A. Stegun,
Handbook of Mathematical Functions, Dover Publications, New York

[6] W. Gander und Jiri Hrebicek, *Solving Problems in Scientific Computing Using Maple and MATLAB*, Springer Verlag

[7] J. D. Jackson, *Klassische Elektrodynamik*, (de Gruyter, Berlin)

[8] Feynman Lectures on Physics, Vol. 2

[9] P.L.DeVries, A first course in COMPUTATIONAL PHYSICS, Jogn Wiley and Sons, INC. (New York)

[10] W. Nolting, *Grundkurs Theoretische Physik, Quantenmechanik, Band 5, Teil 1*, Verlag Zimmermann Neufang

[11] A. Dullweber, E.R.Hilf und E. Mendel, *Simple Quantum Mechanical Phenomena and the Feynman Real Time Path Integral*, preprint server quant-ph/9511042

[12] R.P.Feynman und A.R.Hibbs, *Quantum Mechanics and Path Integrals*, McGraw-Hill 1965

[13] M. Lüscher, "Volume Dependence Of The Energy Spectrum In Massive Quantum Field Theories. 2. Scattering States," Commun. Math. Phys. **105**, 153 (1986).

[14] M. Lüscher and U. Wolff, "How To Calculate The Elastic Scattering Matrix In Two-Dimensional Quantum Field Theories By Numerical Simulation," Nucl. Phys. B **339**, 222 (1990).