Computational Physics I *

Ulli Wolff
mit
Peter Galler, Tomasz Korzec

Institut für Physik der HU
Computational Physics

e-mail: uwolff@physik.hu-berlin.de
e-mail: galler@physik.hu-berlin.de
e-mail: korzec@physik.hu-berlin.de

10. Juni 2014

Zusammenfassung

Die Skripten CP1 (und auch schon CP2) aus früheren Semestern, die sich von Semester zu Semester allerdings ändern in der Stoffauswahl und wegen software upgrades, finden sich im Web unter `www.physik.hu-berlin.de/com` → `teachingandseminars`. Ebenfalls dort (→ `Comp. Phys. I`) ist das aktuelle laufend fortgeschriebene Skriptum.
Acknowledgement: An der Entstehung dieses Skriptums hat Burkhard Bunk maßgeblichen Anteil.
Da die Texte teilweise älter sind, liegt eine bunte Mischung aus alter und neuer Rechtschreibung vor.

*Kurs im Bachelor Modul P5 *Wissenschaftliches Rechnen* SS 2014

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 nichtlinearen Bewegungsgleichungen für Kepler–Probleme 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 (zum Beispiel bei 3–Körper Kepler–Problemen): 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 Systeme zuerst 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 Physik Vorlesungen gegeben sein. In der aktuellen Forschung findet man gelegentlich durchaus Verstöße gegen die obige Warnung. Man geht gleich auf komplizierteste nur gerade eben noch 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 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, 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 Tausende von Prozessoren simultan und koordiniert einsetzen. Auch die PCs im Pool haben soviel Rechenleistung und sind so gut vernetzt, dass sie teilweise (im Hintergrund) als Parallelrechner arbeiten. Darüber hinaus gibt es solche Rechencluster in manchen Arbeitsgruppen.

Neben der Größe der Rechner spielt auch die Wahl der Verfahren und Algorithmen eine erhebliche Rolle. An Beiträgen zu diesem Gebiet wird auch hier in der Arbeitsgruppe COM (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 Arbeiten zur Verfügung (PC-Pool mit Linux). Eine solche Umgebung wird auch in vielen

[1]Das ist natürlich sehr subjektiv!

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

Arbeitsgruppen benutzt und gibt für eine spätere Master- oder Doktorarbeit auf den verschiedensten Gebieten eine zukunftssichere Basis ab, insbesondere bei Bedarf an Höchstrechenleistung.

Für unsere Probleme stellt sich nun die Frage nach der Software bzw. Programmiersprache. Im numerischen Bereich ist nach wie vor neben C und C++ die schon recht alte Sprache FORTRAN als f77 und inzwischen auch f90, f95 populär, 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 (und auch CP2 im Master Studium) 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 Vergleich mit Paketen wie MATHEMATICA und MAPLE. Letztere erlauben zwar auch die interessante Behandlung von Formeln, sind jedoch numerisch ziemlich langsam. Darüber hinaus sehen wir im Erlernen von MATLAB auch eine sinnvolle Ausbildung, da dieses Paket an vielen industriellen Arbeitsplätzen von Physikern verwendet wird. Mit ausgelöst durch diesen Kurs, wird inzwischen MATLAB in der AG COM auch bei vielen Forschungsprojekten eingesetzt, z. B. zur komfortablen Datenanalyse.

UNIX bzw. LINUX und MATLAB sind den Bachelor Studenten aus dem anfänglichen EDV Kurs (Modul P0) schon in einem gewissen Umfang bekannt. Zur Wiederholung wird im folgenden Abschnitt und in den ersten Übungen ein Minimum an Kommandos vorgestellt, so daß wir mit dem eigentlichen Rechnen beginnen können. In der Vorlesung wird dieser Teil eher knapp gehalten. In den Abschnitten zu einzelnen Problemen, die dann folgen, wird nach und nach der 'Wortschatz' erweitert. Dies wird so geschehen, daß hauptsächlich die Namen der relevanten Kommandos klar werden. Zu diesen kann man sich dann interaktiv am Terminal 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 primär für Master-Studenten. Dort sollen dann, auf dem in diesem Semester Erlernten aufbauend, etwas größere Simulationsprojekte bearbeitet werden. Regelmäßig gibt es inzwischen auch einen Kurs 'Computational Physics III' (Paralleles Rechnen) von Dr. Burkhard Bunk.

2 MATLAB zum Anfangen

Nun folgen erste Gehversuche in MATLAB [3]. Dabei handelt es sich um ein Programmsystem für u. a. 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 ca. 30 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 aus einem Pull-Down Menu, falls vorhanden, ansonsten mit dem Befehl **matlab** am Unix–Prompt. Es öffnet sich dann eine grafische Oberfläche, in die das Kommandofenster integriert ist. In dem Menu `File` kann man auch einen MATLAB -spezifischen Editor öffnen. Einige Details hängen von der früheren Einrichtung der Oberfläche ab, das meiste erschließt sich inutitiv. Wenn im Folgenden Text MATLAB Reaktionen gezeigt werden, können diese geringfügig abweichen von dem, was jetzt kommt, da sie mit Vorgängerversionen von MATLAB erstellt wurden. Sitzt man an einer für die ganze Grafik zu langsamen Verbindung, dann kann man mit **matlab -nodesktop** auch im `xterm` ASCII Fenster starten und arbeiten.

2.1 Variablen 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, man muss sie also nicht als Typ integer vordeklarieren. Aus den folgenden Beispielen wird klar, wie diese einzugeben sind:

```
>> a=4.7
```

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

```
a =

    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    ans  b    c    d    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

>>
```

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

2.2 Hilfesysteme

Ein umfangreiches Online–Hilfesystem ist schnell erreicht: im Terminalfenster sagt man **doc** und öffnet damit die Hilfe im HTML-Browser, in der grafischen Oberfläche kann man das Hilfesystem direkt anklicken (oder mit der Taste `F1`.). Im help Menü unte Examples findet sich weiteres Material und Beispiele. Zum Beispiel findet man "Traveling salesman"unter "Other Examples". Das klassische Optimierungsproblem is so definiert: gegeben sei eine Anzahl von Städten zusammen mit den Kosten für die Reise zwischen jedem Paar von Städten. Gesucht ist die billigste Verbindung durch alle Städte, welche zum Ausgangspunkt zurückkehrt und jede Stadt nur einmal enthält. Wir werden im Laufe dieses Kurses dieses komplizierte Problem nicht lösen. Wir werden aber in "Computational Physics II" Begriffen und Verfahren begegnen, welche eine Verbindung zum "traveling salesman" Problem haben.

Als Beispiel zur Online–Hilfe wollen wir herausfinden, wie wir die Exponentialfunktion unter MATLAB benutzen. In "Search Documentation" geben wir als (englischen!) Suchbegriff "exponential" ein. Wir bekommen eine Liste von Titeln, welche wir anklicken können. Es erscheinen dann Stellen in der MATLAB Dokumentation, wo das Wort "exponential" vorkommt. Wenn wir schon einen Funktionsnamen wissen, wie zum Beispiel `exp` oder `expm` und wollen die Dokumentation dazu ansehen, dann können wir ihn direkt eingeben.

Zusätzlich zu der Online–Hilfe im Hypertextformat gibt es noch einen (älteren) Zugang zu Funktionsbeschreibungen direkt vom MATLAB –Prompt aus. Im Gegensatz zu UNIX (**man**) bekommt man in MATLAB Auskunft mit **help**, und zwar in einer für die Schnittstelle Mensch deutlich geeigneteren Form. Wir werden in der Vorlesung gelegentlich Beispiele vom **help** Output zeigen. Die Online–Hilfe ist im Allgemeinen aber ausführlicher. Eine angenehme Eigenschaft ab MATLAB Version 7.0 ist, dass man die Online–

Hilfe direkt aus dem **help** Output aufrufen kann. Z.B. wenn man am MATLAB –Prompt **help expm** eingibt, führt der angezeigte link **doc expm** zur Online–Hilfe. Ausserdem kann man z.B. **logm** anklicken und es erscheint die **help** Hilfe zur Funktion `logm`. Es ist dringend zu empfehlen, selber mit der Online–Hilfe und mit **help** zu experimentieren!

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 der Online–Hilfe wenn man nach "Search for" den Begriff "arithmetic" eintippt. Dann finden wir "Arithmetic Operators + - * / \ ˆ '". Oder im MATLAB -Prompt mit

```
>> help arith

 Arithmetic operators.
  +   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) Ope-**

rationen .* .^ ./ die Matrizen *elementweise* verknüpfen. Hier zeigt MATLAB die volle Stärke einer **Vektorsprache!**. 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.

In der Online–Hilfe können wir weitere interessante Befehle (z.B. \) und Informationen finden. In den Übungsaufgaben gibt es mehr Gymnastik mit den normalen und gepunkteten Operationen.

2.5 Vordefinierte Funktionen in MATLAB

In MATLAB gibt es zahlreiche eingebaute Funktionen. Wenn wir in der Online–Hilfe nach "elementary math" suchen (oder **help elfun** am MATLAB -Prompt), 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. Diesen File kann man mit einem beliebigen Editor erstellen. Eine gute Möglichkeit ist der spezielle MATLAB Editor: Im Menu `File` wählt man `New` um einen neuen File zu kreieren oder `Open` um einen vorhandenen zu bearbeiten. `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). Wichtig ist, daß der Aufruf der Funktion mit dem *Filenamen* übereinstimmt. Zum Beispiel im file `mitteldiff.m` könnten wir auch schreiben `function [m,d] = anyname(u,v)` (das ist aber kein gutes Programmieren!) und trotzdem heißt der Aufruf `>> mitteldiff(x,y)`. `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`, wie alle anderen Variablen im Funktionsprogramm, sind *interne* (oder *lokale*)

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 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 **plot(x,y,'*')** bekommt man die einzelnen Punkte mit dem Symbol * gezeigt. Wenn man mit der Online–Hilfe nach "plot" sucht (oder **help plot**) kann

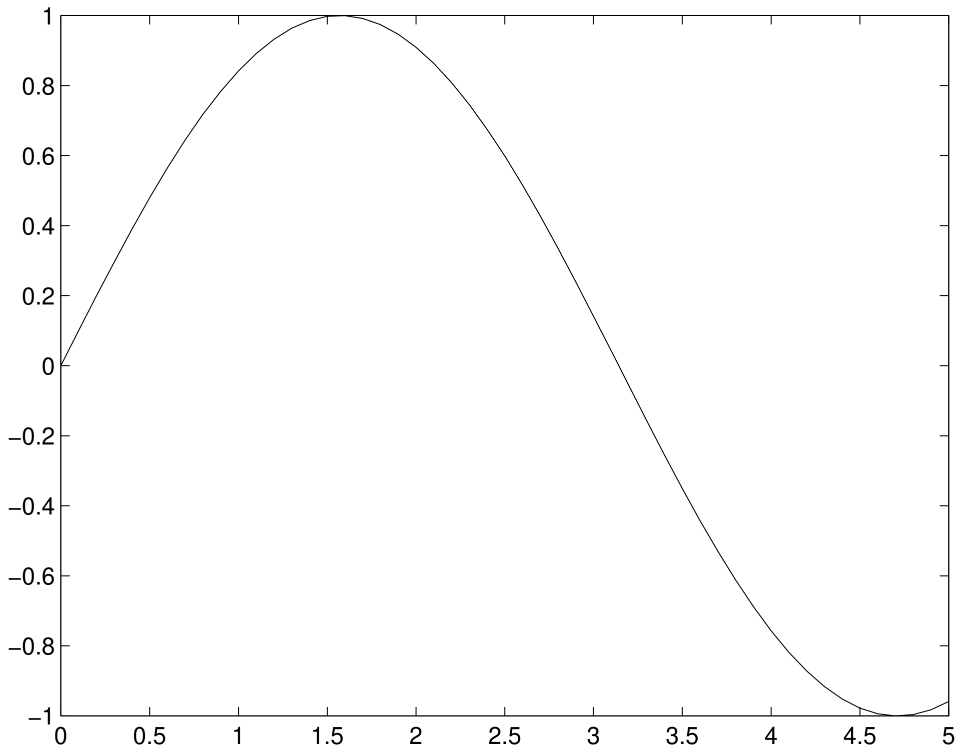

Abbildung 1: plot(x,y)

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 Conditionally execute statements.
    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.

  Reference page in Help browser
        doc if
>>
```

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

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 unter **help** die Schlüsselworte **if, elseif** usw in Großbuchstaben (veraltet?), sie *müssen* aber klein geschrieben werden! Die Online–Hilfe im Browser hat dieses Problem nicht.

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 [1] 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 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

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

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 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(x); % 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    6.9e-03
 1e-15    2.7e-02    2.7e-02
 1e-16   -1.0e+00    2.7e-02
>>
```

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, daß 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 der Physik geht das meist in Ordnung, wenn man mit dem Problem angepassten Einheiten arbeitet. In praktischen Rechnungen ist es jedenfalls oft vorteilhaft bis nötig, die Abhängigkeit von Numerik-bedingten Parametern wie h und auch verschiedene Näherungsformeln zu studieren. D. h., man muss sie variieren statt einfach irgendeinen Wert zu nehmen.

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
x=1;
n=10.^(1:12);
s=(1+x./n).^n;
delta=(s-exp(x))/exp(x)
ndelta=delta.*n;
%
y=[n;delta;ndelta];
fprintf('    n       delta        n*delta \n\n')
```

```
fprintf('%10.4e %11.4e %11.4e \n',y)
```

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 -5.0008e-07 -5.0008e-01
1.0000e+07 -4.9416e-08 -4.9416e-01
1.0000e+08 -1.1077e-08 -1.1077e+00
1.0000e+09  8.2240e-08  8.2240e+01
1.0000e+10  8.2690e-08  8.2690e+02
1.0000e+11  8.2735e-08  8.2735e+03
1.0000e+12  8.8905e-05  8.8905e+07
```

Wir kommen nur auf eine relative Genauigkeit von etwa 10^{-8}, 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,

$$\text{Gesamtfehler} \sim 1/n + \epsilon n \qquad (3.13)$$

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 $x = 0$,

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

Nach dem Aufruf 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 (x=1)
%
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 -4.5198e-12
14 -2.9995e-13
15 -1.8624e-14
16 -9.8023e-16
17  0.0000e+00
18  0.0000e+00
19  0.0000e+00
20  0.0000e+00
```

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, z. B. bei den Polynomen, die beim Lösen der Schrödinger Gleichung vorkommen. 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    0.1603
16.0000    0.1538
17.0000    0.1043
18.0000    0.8417
19.0000  -13.2742
20.0000  268.2026
```

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    0.1603    0.1604    0.1604
   16.0000    0.1538    0.1515    0.1515
   17.0000    0.1043    0.1433    0.1434
   18.0000    0.8417    0.1392    0.1362
   19.0000  -13.2742    0.0742    0.1297
   20.0000  268.2026    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.

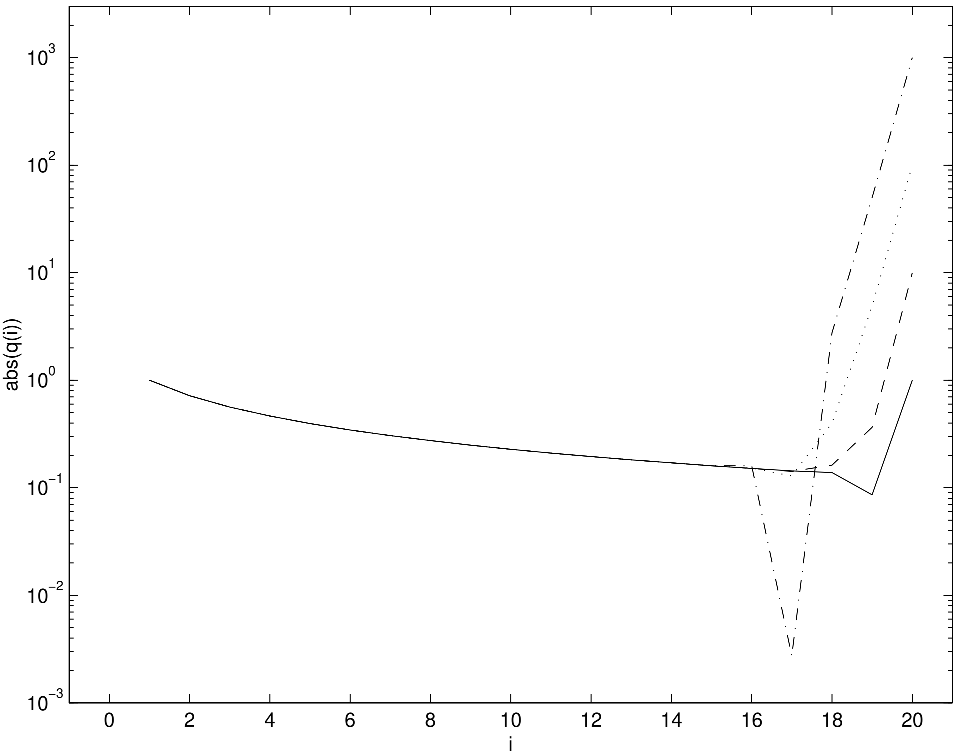

Abbildung 2: semilogy(i,abs(q(i))), verschiedene Startwerte.

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 2 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 expression
         statements
       END

    The statements are executed while the real part of the expression
    has all non-zero elements. The expression is usually the result of
    expr rop expr where rop is ==, <, >, <=, >=, or ~=.

    The BREAK statement can be used to terminate the loop prematurely.

    For example (assuming A already defined):

            E = 0*A; F = E + eye(size(E)); N = 1;
            while norm(E+F-E,1) > 0,
               E = E + F;
               F = A*F/N;
               N = N + 1;
```

```
         end

    See also for, if, switch, break, continue, end.

    Reference page in Help browser
       doc while
```

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 (± **INF**, vgl. Übungsblatt 1) 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. Eine Legende wird mit dem Befehl **legend(string1,string2,string3)** erzeugt, wobei die Strings die Beschriftung der Kurven spezifizieren.

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 [2] und [1] 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 (die Variable, wie x vorher) und β die inverse Temperatur (ein äußerer Parameter), 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 in 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, eventuell auch noch zweidimensionalen Nullstellenproblemen nützlich, erst mal einen Plot der relevanten Funktionen zu erstellen.

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

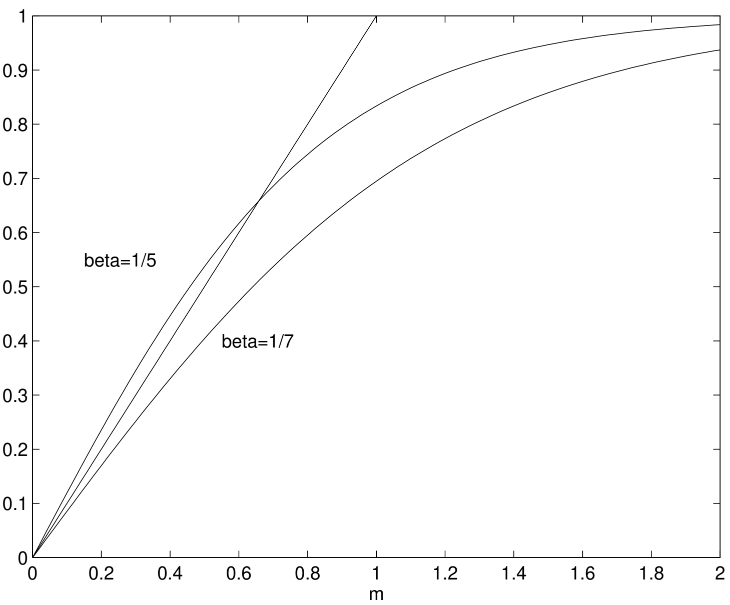

Abbildung 3: Beide Seiten von Gl.(4.2)

```
%
% file mffunc.m
%
% Funktion, deren Nullstellen die Loesung fuer die Magnetisierung
% m im Beispiel Problem geben
%
function [y] = mffunc(m,beta)
y = m - tanh(6*beta*m);
% funktioniert auch wenn m ein Vektor/Matrix ist
```

deren Nullstellen die gesuchte Magnetisierung sind. Mit der Kommandosequenz

```
>> beta=0.2
>> m=0:0.05:1;
>> plot(m,mffunc(m,beta),[0 1],[0 0]);
```

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.

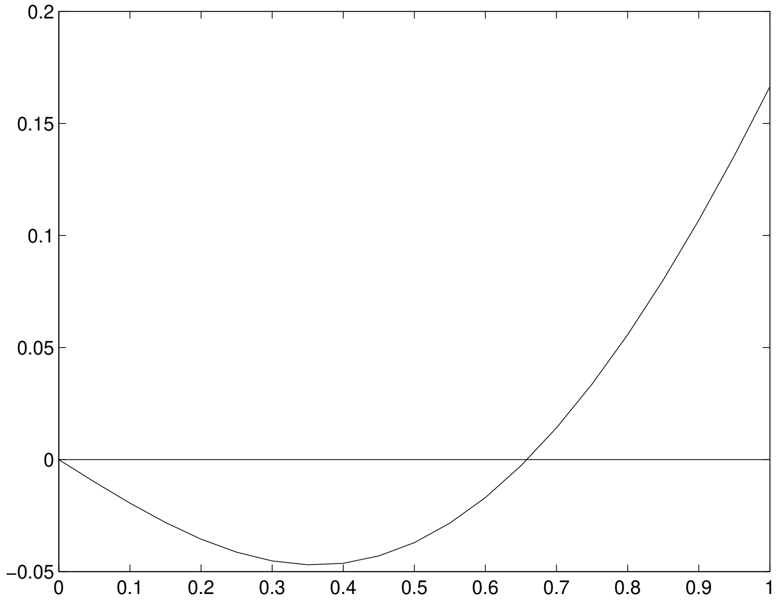

Abbildung 4: Die Funktion (4.3) mit $\beta = 0.2$

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 monoton wachsende und konvexe Funktion hat, die durch den Ursprung geht.

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. Man beachte, dass hier f^{-1} die Umkehrfunktion ist, nicht die reziproke. 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 ein 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 Fall. Dann wird das Intervall fortlaufend halbiert und die Hälfte weiterverwendet, in der der Vorzeichenwechsel liegt.

Wir demonstrieren dies an einem MATLAB Programm.

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

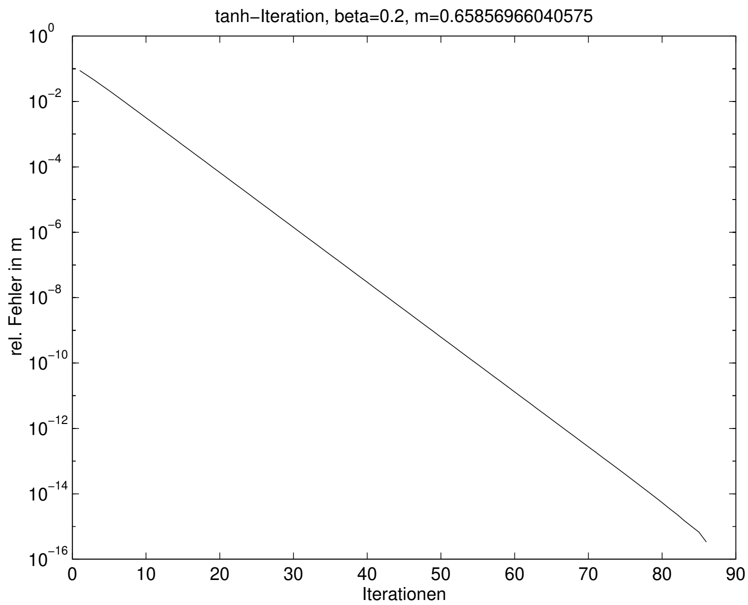

Abbildung 5: Konvergenz von Gl.(4.4)

```
%
% 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: function handle auf Funktion func(x)
% tol: Gewuenschte absolute Genauigkeit
%
  function [x0] = bisect(func,x1,x2,tol)
  f1=func(x1); if f1 == 0, x0=x1; return; end
  f2=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=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 =

   3.141592653589794

>> ans-pi

ans =

     4.440892098500626e-16

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

ans =

   0.658569660405754

>>
```

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 im Argument `func` ein sog. 'function-handle' übergeben. Bei `@sin` wird einfach $sin(x)$ verwendet. Unser `bisect` braucht immer eine Funktion von nur einer Variablen. Andererseits hatte `mffunc` zwei Argumente. In einem wird die Nullstelle gesucht, das andere (`beta`) ist dabei nur 'Zuschauer'. Die (moderne) Lösung dieses Problems besteht in der Konstruktion einer 'anonymous function' die nur von einem variablen Argument abhängt und das andere fixiert. Genau das wird von dem Gebilde `@(x) mffunc(x,beta)` bewirkt. Grundsätzlich kann hinter `@(x)` ein beliebiger Ausdruck stehen, der den Funktionswert berechnet. Hier benutzt er eine weitere Funktion und den Parameter `beta`, der natürlich definiert sein muss. Eine früher benutzte aber weniger empfehlenswerte Alternative wäre es, `beta` als globale Variable zu liefern und `mffunc` gleich mit nur einem Argument zu definieren.

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 [1] 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 [2]). 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 [1]

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

wobei τ eher etwas größer als ϵ sein sollte.

Zur Fehlerbetrachtung leiten wir zunächst aus (4.14) ab

$$\begin{aligned} \delta_{n+1} &\simeq \delta_n - (\delta_n f'(x_*) + \delta_n^2 f''(x_*)/2)\,\frac{\delta_n - \delta_{n-1}}{(\delta_n - \delta_{n-1})f'(x_*) + \frac{1}{2}(\delta_n^2 - \delta_{n-1}^2)f''(x_*)} \\ &\simeq \frac{f''(x_*)}{2f'(x_*)}\delta_n\delta_{n-1} + \mathrm{O}(\delta^3) \end{aligned} \tag{4.16}$$

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 [1]. 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 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  Scalar nonlinear zero finding.
    X = FZERO(FUN,X0) tries to find a zero of the function FUN near X0,
    if X0 is a scalar.  It first finds an interval containing X0 where the
    function values of the interval endpoints differ in sign, then searches
    that interval for a zero. FUN accepts real scalar input X and returns
    a real scalar function value F, evaluated at X.  The value X returned
    by FZERO is near a point where FUN changes sign (if FUN is continuous),
    or NaN if the search fails.

    X = FZERO(FUN,X0), where X0 is a vector of length 2, assumes X0 is an
    interval where the sign of FUN(X0(1)) differs from the sign of FUN(X0(2)).
    An error occurs if this is not true.  Calling FZERO with an interval
    guarantees FZERO will return a value near a point where FUN changes
    sign.

    X = FZERO(FUN,X0), where X0 is a scalar value, uses X0 as a starting
    guess. FZERO looks for an interval containing a sign change for FUN and
    containing X0.  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. Note: if
    the option FunValCheck is 'on', then an error will occur if an NaN or
    complex value is found.

    X = FZERO(FUN,X0,OPTIONS) minimizes with the default optimization
    parameters replaced by values in the structure OPTIONS, an argument
    created with the OPTIMSET function.  See OPTIMSET for details.  Used
    options are Display, TolX, FunValCheck, and OutputFcn. Use OPTIONS = []
    as a place holder if no options are set.

    [X,FVAL]= FZERO(FUN,...) returns the value of the objective function,
    described in FUN, at X.

    [X,FVAL,EXITFLAG] = FZERO(...) returns an EXITFLAG that describes the
    exit condition of FZERO. Possible values of EXITFLAG and the corresponding
    exit conditions are
```

```
  1  FZERO found a zero X.
 -1  Algorithm terminated by output function.
 -3  NaN or Inf function value encountered during search for an interval
      containing a sign change.
 -4  Complex function value encountered during search for an interval
      containing a sign change.
 -5  FZERO may have converged to a singular point.

[X,FVAL,EXITFLAG,OUTPUT] = FZERO(...) returns a structure OUTPUT
with the number of function evaluations in OUTPUT.funcCount, the
algorithm name in OUTPUT.algorithm, the number of iterations to
find an interval (if needed) in OUTPUT.intervaliterations, the
number of zero-finding iterations in OUTPUT.iterations, and the
exit message in OUTPUT.message.

Examples
  FUN can be specified using @:
     X = fzero(@sin,3)
  returns pi.
     X = fzero(@sin,3,optimset('disp','iter'))
  returns pi, uses the default tolerance and displays iteration information.

  FUN can also be an anonymous function:
     X = fzero(@(x) sin(3*x),2)

If FUN is parameterized, you can use anonymous functions to capture the
problem-dependent parameters. Suppose you want to solve the equation given
in the function MYFUN, which is parameterized by its second argument A. Here
MYFUN is an M-file function such as

    function f = myfun(x,a)
    f = cos(a*x);

To solve the equation for a specific value of A, first assign the value to A.
Then create a one-argument anonymous function that captures that value of A
and calls MYFUN with two arguments. Finally, pass this anonymous function to
FZERO:
```

```
        a = 2; % define parameter first
        x = fzero(@(x) myfun(x,a),0.1)

    Limitations
         X = fzero(@(x) abs(x)+1, 1)
      returns NaN since this function does not change sign anywhere on the
      real axis (and does not have a zero as well).
         X = fzero(@tan,2)
      returns X near 1.5708 because the discontinuity of this function near the
      point X gives the appearance (numerically) that the function changes sign
      at X.

See also roots, fminbnd, function_handle.

    Reference page in Help browser
       doc fzero
```

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 2% unter- und oberhalb des vorgelegten Startwerts ($[-1/50, 1/50]$, falls dieser Null ist). Ist dort kein Vorzeichenwechsel, so wird das Intervall jeweils um einen Faktor $\sqrt{2}$ vergrößert, bis ein Wechsel gefunden ist. Danach wird mit Bisektion und Interpolation (linear oder quadratisch) zwischen den Intervallgrenzen gearbeitet. Letztere stellt eine weitere Methode dar. Es ist klar, daß dieses Programm für vernünftige Funktionen gedacht ist, die auf einer nichtpathologischen 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 wie möglich zu verstehen. Es ist allerdings mit der Fortentwicklung von MATLAB immer komplizerter geworden. 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 überhaupt einsehen 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,

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

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
%
%  Argument: Startwert; Antwort: Nullstelle
%
function [z] = nr3(zn,tol)
```

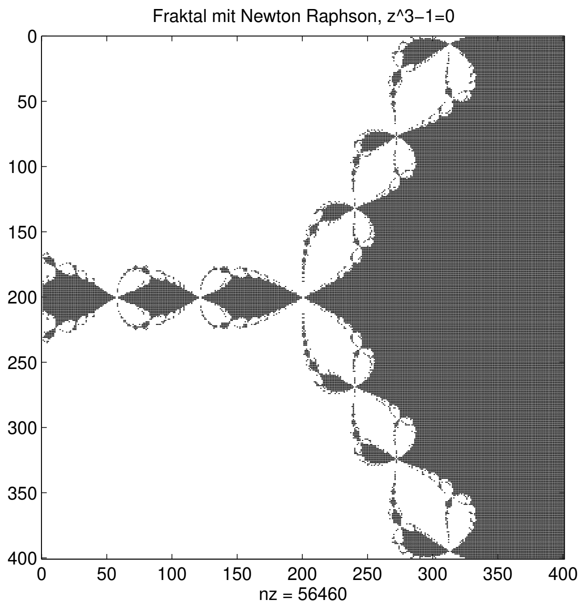

Abbildung 6: Basin of Attraction zu Newton-Raphson, $z^3 = 1$

```
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
%
tol=1.e-13; % Konvergenzkriterium fuer NR
tol10=10*tol;
```

```
%
t=cputime; % zur Bestimmung der Rechenzeit
maxy=2.0;
re=-maxy:(maxy/199.5):maxy; % 400 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),tol)-1) < tol10); % m(,)=1 g.d.w. N.-R. -> 1
    end
end
%
cputime-t %Bestimmung und Ausdruck der verbrauchten CPU-Zeit
spy(m);   % graphische Darstellung der Besiedlung einer Matrix
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 ('Workstation' der Antike (1994), ca. 50000 DM) etwa 7 min. gedauert, auf einem aktuellen Linux–PC dagegen nur noch 3.5 sek! Und zwar ohne Optimierung.

5 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. Mehrkörperkräften oder Reibung, oder vielen Teilchen (mehr als zwei) sind hier numerische Simulationen oft die einzige Möglichkeit, an näherungsunabhängige Lösungen zu kommen.

5.1 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

$$m\frac{d\vec{v}}{dt} = \vec{F}(\vec{v}) - mg\hat{y}; \quad \frac{d\vec{r}}{dt} = \vec{v}. \tag{5.1}$$

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

$$x(t) = v_0 \cos(\theta)t \tag{5.2}$$

$$y(t) = v_0 \sin(\theta)t - \frac{1}{2}gt^2. \tag{5.3}$$

Nach der Flugzeit t_{fl},

$$t_{\text{fl}} = \frac{2v_0}{g}\sin(\theta) \tag{5.4}$$

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

$$y_{\max} = \frac{v_0^2}{2g}\sin^2(\theta) \tag{5.5}$$

und die Weite

$$x_{\max} = \frac{v_0^2}{g}\sin(2\theta) \tag{5.6}$$

Für den Flug eines Baseballs mit Reibung ist das phänomenologische Reibungsgesetz

$$\vec{F}(\vec{v}) = -\frac{1}{2}C_d\rho\pi R^2|\vec{v}|\vec{v} \tag{5.7}$$

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)

5.2 Euler–Methode

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

$$\frac{d\vec{v}}{dt} = \vec{K}(\vec{r}, \vec{v}); \quad \frac{d\vec{r}}{dt} = \vec{v}. \tag{5.8}$$

Man beachte, dass hier auf der rechten Seite $\vec{K}$ (=Kraft/Masse) als gegebene Funktion von Ort und Geschwindigkeit zu betrachten ist. Deren Argumente ebenso wie die anderen vorkommenden Größen sind alle zur Zeit t gemeint: die DGL verknüpft Werte und Ableitungen von $\vec{r}$ und $\vec{v}$ zur *selben* Zeit. Nähern wir nun die Ableitungen durch die einfache asymmetrische Formel

$$\frac{d\vec{v}}{dt} = \frac{\vec{v}(t+\tau) - \vec{v}(t)}{\tau} + \mathrm{O}(\tau), \tag{5.9}$$

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

$$\vec{v}(t+\tau) = \vec{v}(t) + \tau\vec{K}(\vec{r}(t), \vec{v}(t)) + \mathrm{O}(\tau^2) \tag{5.10}$$
$$\vec{r}(t+\tau) = \vec{r}(t) + \tau\vec{v}(t) + \mathrm{O}(\tau^2). \tag{5.11}$$

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 $O(\tau^2)$. 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 der, bei festem t, nur noch proportional zu τ kleiner wird[6]. Der relative Fehler δ (z. B. Flugzeit = Wert $\times(1 \pm \delta)$) ist immer dimensionslos und damit proportional zum Quotienten von τ über einer anderen charakteristischen Zeit des Problems. Beim Ball bildet v_0/g eine Zeitskala und t_{fl} in von der gleichen Größenordnung, solange θ nicht sehr klein ist. Dann wäre also aus Dimensionsgründen zu erwarten, daß $\delta \simeq c\tau/t_{\mathrm{fl}}$ gilt mit einem dimensionslosen c der Ordnung 1, was sich numerisch bestätigen wird.

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
K = [0 -g]; % in diesem Fall konstante Kraft (/Masse)
%
maxstep = 10000; % 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*K;                 % Euler Schritt
```

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

```
%
  if( r(2) < 0 )  % loop abbrechen, Aufschlag
    break;
  end
end
%
xplot(istep+1) = r(1);  yplot(istep+1) = r(2); % letzter Punkt
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
```

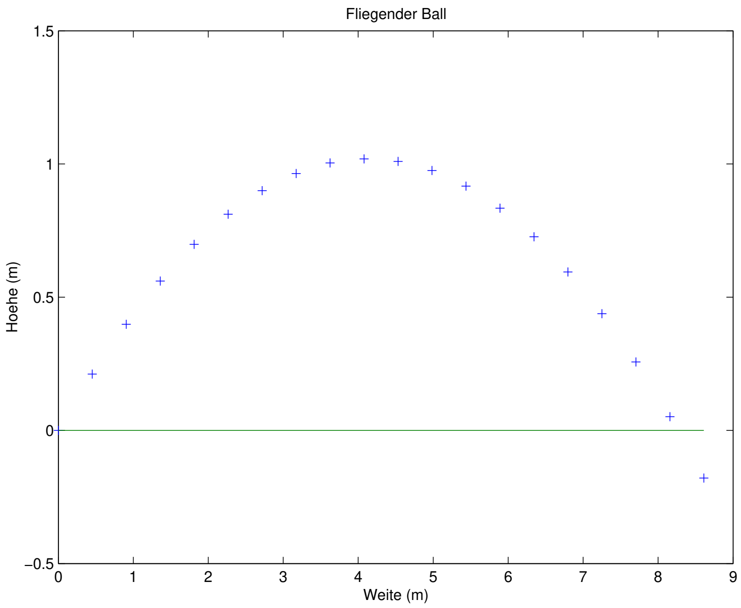

Abbildung 7: Ball Trajektorie

```
 Flugzeit 0.95 sekunden
 rel. Abweichung  Flugzeit: 0.102591
 rel. Abweichung  Weite   : 0.102591
>> print -depsc ball.eps
```

mit dem zugehörigen Bild in Abb.7. Mit $t/\tau = 19$ ergab sich also ein Fehler 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.

5.3 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 [7]:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = F(t, x) \tag{5.12}$$

ist äquivalent zu

$$\frac{\mathrm{d}\vec{y}}{\mathrm{d}t} = \vec{f}(t, \vec{y}) \tag{5.13}$$

mit

$$\vec{f} = (y_2, F(t, y_1)). \tag{5.14}$$

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 (5.13) 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.

5.4 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 [2].

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

$$\vec{y}(t + \tau) = \vec{y}(t) + \tau \vec{f}(t, \vec{y}(t)) + \mathrm{O}(\tau^2). \tag{5.15}$$

Es wird 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 (5.15) auf viele Weisen schreiben läßt, die sich nur in $\mathrm{O}(\tau^2)$ unterscheiden. Diese

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

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:

$$\vec{y}(t+\tau) = \vec{y}(t) + \tau \left[w_1 \vec{f}(t, \vec{y}(t)) + w_2 \vec{f}(t+\alpha\tau, \vec{y}_*) \right] \tag{5.16}$$

mit

$$\vec{y}_* = \vec{y}(t) + \beta\tau \vec{f}(t, \vec{y}(t)). \tag{5.17}$$

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

$$\vec{y}(t+\tau) - \vec{y}(t) \stackrel{!}{=} \tau \vec{f} + \frac{\tau^2}{2} \frac{\mathrm{d}\vec{f}}{\mathrm{d}t} + \mathrm{O}(\tau^3), \tag{5.18}$$

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

$$\frac{\mathrm{d}\vec{f}}{\mathrm{d}t} = \frac{\partial \vec{f}}{\partial t} + \sum_i \frac{\partial \vec{f}}{\partial y_i} \frac{\mathrm{d}y_i}{\mathrm{d}t} = \frac{\partial \vec{f}}{\partial t} + \sum_i \frac{\partial \vec{f}}{\partial y_i} f_i \tag{5.19}$$

gilt. Durch Entwickeln in (5.16) ergibt sich

$$\vec{y}(t+\tau) - \vec{y}(t) = \tau(w_1 + w_2)\vec{f} + w_2\alpha\tau^2 \frac{\partial \vec{f}}{\partial t} + w_2\tau \sum_i (y_* - y)_i \frac{\partial \vec{f}}{\partial y_i} + \mathrm{O}(\tau^3) \tag{5.20}$$

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

$$w_1 + w_2 = 1 \tag{5.21}$$

$$w_2\alpha = w_2\beta = \frac{1}{2}. \tag{5.22}$$

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

$$w_1 = 0,\ w_2 = 1,\ \alpha = \beta = \frac{1}{2}. \tag{5.23}$$

oder

$$w_1 = w_2 = \frac{1}{2},\ \alpha = \beta = 1. \tag{5.24}$$

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

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

$$\begin{aligned} \vec{k}_1 &= \tau \vec{f}\,(t, \vec{y}(t)) && (5.25)\\ \vec{k}_2 &= \tau \vec{f}\left(t+\tau, \vec{y} + \vec{k}_1\right) && (5.26)\\ \vec{k}_3 &= \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{y} + \frac{1}{4}(\vec{k}_1 + \vec{k}_2)\right). && (5.27) \end{aligned}$$

Offenbar entspricht die Variante (5.24) der 2. Ordnung–Formel nun der Evolution

$$\vec{y}(t+\tau) = \vec{y}(t) + \frac{1}{2}[\vec{k}_1 + \vec{k}_2] + \mathrm{O}(\tau^3) \tag{5.28}$$

Indem man eine Ordnung weiter entwickelt, läßt sich zeigen:

$$\vec{y}(t+\tau) = \vec{y}(t) + \frac{1}{6}[\vec{k}_1 + \vec{k}_2 + 4\vec{k}_3] + \mathrm{O}(\tau^4). \tag{5.29}$$

5.6 Runge-Kutta–Formel 4. Ordnung

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

$$\begin{aligned} \vec{k}_1 &= \tau \vec{f}\,(t, \vec{y}(t)) && (5.30)\\ \vec{k}_2 &= \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{y} + \frac{1}{2}\vec{k}_1\right) && (5.31)\\ \vec{k}_3 &= \tau \vec{f}\left(t + \frac{1}{2}\tau, \vec{y} + \frac{1}{2}\vec{k}_2\right) && (5.32)\\ \vec{k}_4 &= \tau \vec{f}\left(t+\tau, \vec{y} + \vec{k}_3\right) && (5.33) \end{aligned}$$

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

$$\vec{y}(t+\tau) = \vec{y}(t) + \frac{1}{6}[\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4] + \mathrm{O}(\tau^5) \tag{5.34}$$

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.

5.7 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.

    Reference page in Help browser
       doc norm
```

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*.

5.8 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 Komponenten von $\vec{y}$ stark variieren, 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:

$$\vec{y}(t) \xrightarrow{\tau} \vec{Y}$$

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

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

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

$$\delta = \max_i \frac{|Y_i - y_i(t+\tau)|}{|y_i|(2^n - 1)}. \tag{5.35}$$

Um sicher zu sein, wird hier der größte relative Fehler von allen Komponenten von $\vec{y}$ genommen [8].

[8]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.

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:

$$\tau' = \tau \left(\frac{\epsilon}{\delta}\right)^{\frac{1}{n+1}} . \tag{5.36}$$

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 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.

5.9 Beispiel eines MATLAB Runge Kutta Solvers

In diesem Abschnitt werden wir ein Runge-Kutta-Programm mit automatischer Anpassung der Schrittweite analysieren, das früher der MATLAB -Biliothek angehörte, inzwischen aber durch ein moderneres, jedoch weniger transparentes Programm ersetzt wurde. Der folgende Programmcode ist daher 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, werden wir einige neue MATLAB –Kommandos kennenlernen.

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.
```

```
% Initialization
pow = 1/3;
if nargin < 5, tol = 1.e-3; end
if nargin < 6, trace = 0; end

t = t0;
hmax = (tfinal - t)/16;
h = hmax/8;
y = y0(:);
chunk = 128;
tout = zeros(chunk,1);
yout = zeros(chunk,length(y));
k = 1;
tout(k) = t;
yout(k,:) = y.';

if trace
   clc, t, h, y
end

% The main loop

while (t < tfinal) & (t + h > t)
   if t + h > tfinal, h = tfinal - t; end

   % Compute the slopes
   s1 = feval(ypfun, t, y); s1 = s1(:);
   s2 = feval(ypfun, t+h, y+h*s1); s2 = s2(:);
   s3 = feval(ypfun, t+h/2, y+h*(s1+s2)/4); s3 = s3(:);

   % Estimate the error and the acceptable error
   delta = norm(h*(s1 - 2*s3 + s2)/3,'inf');
   tau = tol*max(norm(y,'inf'),1.0);

   % Update the solution only if the error is acceptable
   if delta <= tau
      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. 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 5.5) (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 [9] 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` 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.

5.10 Aktuelle MATLAB -Programme ode23 und ode45

MATLAB stellt inzwischen ein ganzes Arsenal von Programmen zur numerischen Lösung von Differentialgleichungen zur Verfügung. Das Programm **ode23** ist das Nachfolgeprogramm von rk23, das wir im letzten Abschnitt vorgestellt haben. ODE steht für 'ordinary differential equation' (gewöhnliche DGL) 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. Man kann sich diese Programme mit den Kommandos **type** und **dbtype** anschauen (Unterschied: **dbtype**

[9] Bitte verifizieren!

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.

```
help ode23
 ODE23  Solve non-stiff differential equations, low order method.
    [TOUT,YOUT] = ODE23(ODEFUN,TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates
    the system of differential equations y' = f(t,y) from time T0 to TFINAL
    with initial conditions Y0. ODEFUN is a function handle. For a scalar T
    and a vector Y, ODEFUN(T,Y) must return a column vector corresponding
    to f(t,y). Each row in the solution array YOUT corresponds to a time
    returned in the column vector TOUT.  To obtain solutions at specific
    times T0,T1,...,TFINAL (all increasing or all decreasing), use TSPAN =
    [T0 T1 ... TFINAL].

    [TOUT,YOUT] = ODE23(ODEFUN,TSPAN,Y0,OPTIONS) solves as above with default
    integration properties replaced by values in OPTIONS, an argument created
    with the ODESET function. See ODESET for details. Commonly used options
    are scalar relative error tolerance 'RelTol' (1e-3 by default) and vector
    of absolute error tolerances 'AbsTol' (all components 1e-6 by default).
    If certain components of the solution must be non-negative, use
    ODESET to set the 'NonNegative' property to the indices of these
    components.

(....Rest weggelassen)
```

Wir wollen nun die Differentialgleichung

$$\frac{d}{dt}\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} y_2 \\ -y_1 \end{pmatrix} \tag{5.37}$$

numerisch für die Anfgangsbedingung bei $t = 0$

$$y_1(0) = 0, \quad y_2(0) = 1 \tag{5.38}$$

bestimmen. Die Differentialgleichung läßt sich natürlich auch analytisch lösen,

$$y_1^{\text{ex}}(t) = \sin t, \quad y_2^{\text{ex}}(t) = \cos t \ . \tag{5.39}$$

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:

```
%
% odetest.m
%
% 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);
%
%opts = odeset('RelTol',1e-5);
%[t,y] = ode23(@ftest,tspan,y0,opts);
%
[t,y] = ode45(@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. 8 werden die Abweichungen $y_1(t) - y_1^{\mathrm{ex}}(t)$ und $y_2(t) - y_2^{\mathrm{ex}}(t)$ als Funktion von t dargestellt.

Man erkennt, daß die maximale Abweichung während eines Umlaufs etwa linear mit t anwächst. In der Abb. 9 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

>> size(y)

ans =

   607     2
```

Die entsprechenden Ergebnisse für das Programm **ode45** sind in den Abb. 10 und 11 dargestellt. Die Abweichungen sind deutlich kleiner sind als im Falle des Programms **ode23**.

Die hier erzielten Genauigkeiten wurden mit default Werten für die Fehlerkontrolle erzielt. Laut MATLAB **help** wird dabei für den Einzelschritt angestrebt, dass der relative Fehler $\leq$ `RelTol` $= 10^{-3}$ ist *oder* der absolute $\leq$ `AbsTol` $= 10^{-6}$, jeweils für alle Koponenten. Wie diese Werte modifiziert werden können, sieht man in den auskommentierten Zeilen von **odetest**. Es werden mit **odeset** Größen `opts` erzeugt, in die man beliebige Werte für `RelTol` und `AbsTol` schreiben kann. Dort können auch weitere Attribute gesetzt werden, die wir hier nicht näher betrachten.

5.11 Kepler–Problem

5.11.1 Einheiten im Sonnensystem

Zunächst wollen wir für die Betrachtung astronomischer Probleme im Sonnensystem praktische Einheiten einführen, die zu nicht extremen Zahlen

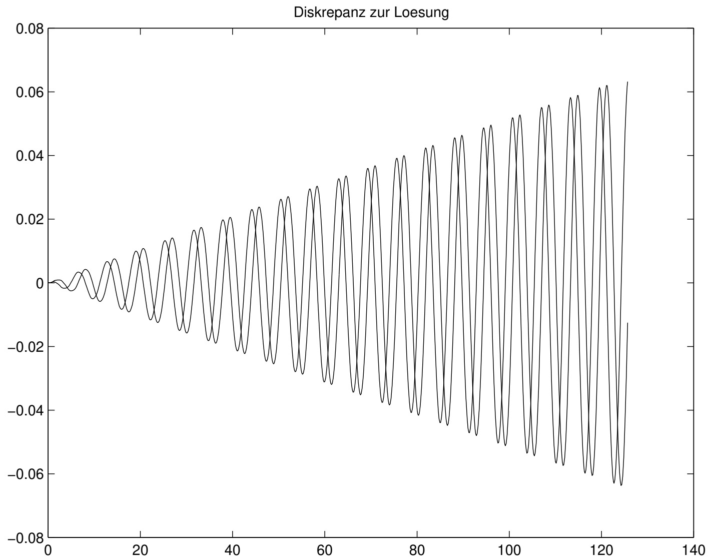

Abbildung 8: Absoluter Fehler bei `odetest/ode23`

.

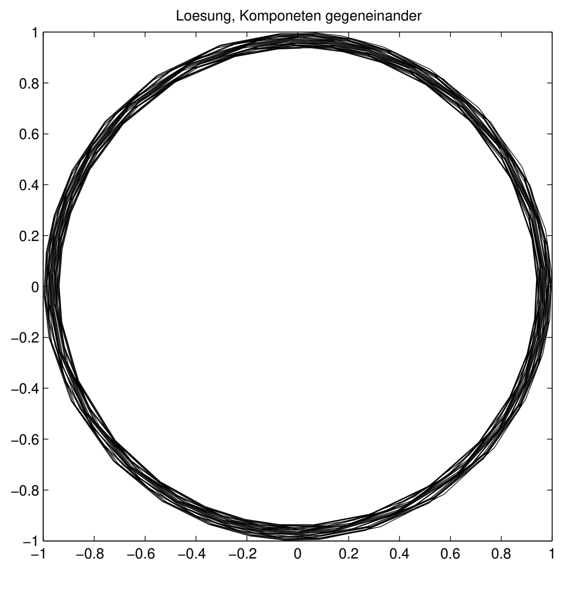

Abbildung 9: `y(2)` gegen `y(1)` bei `odetest/ode23`

.

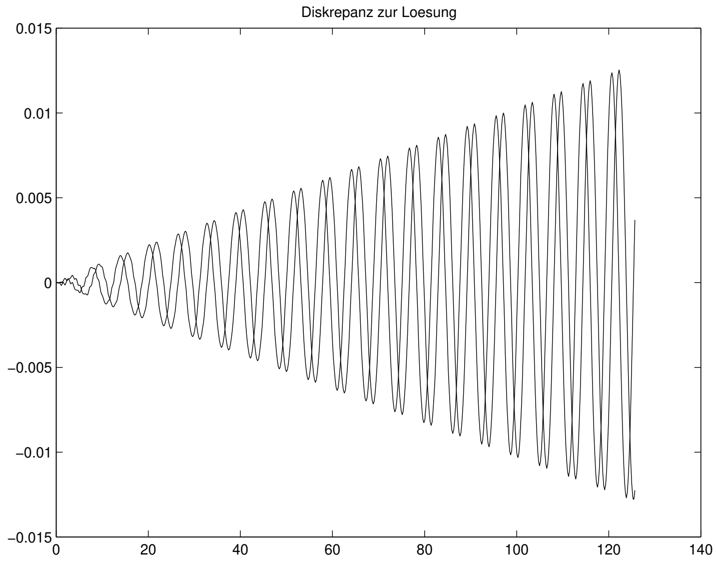

Abbildung 10: Absoluter Fehler bei `odetest/ode45`

.

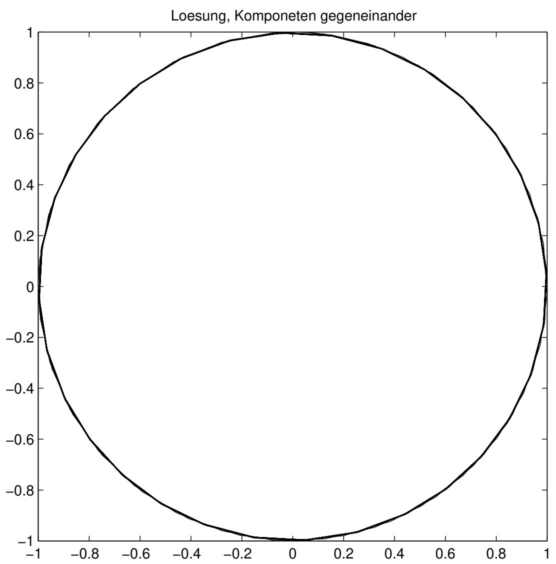

Abbildung 11: `y(2)` gegen `y(1)` bei `odetest/ode45`

.

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.

$$1\mathrm{AU} = 1.496 \times 10^{11}\mathrm{m}. \tag{5.40}$$

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

$$m\omega^2 R = \frac{GmM}{R^2}. \tag{5.41}$$

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

$$GM = \omega^2 a^3 = 4\pi^2 \, \frac{\mathrm{AU}^3}{\mathrm{yr}^2} \tag{5.42}$$

5.11.2 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

$$\ddot{\vec{r}} = -\frac{GM}{|\vec{r}|^3}\vec{r} \tag{5.43}$$

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

$$\vec{L} = m\,\vec{r} \times \vec{v} \tag{5.44}$$

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

$$E = m\left(\frac{1}{2}\vec{v}^2 - \frac{GM}{|\vec{r}|}\right). \tag{5.45}$$

Wie aus der Mechanik bekannt, ist die Bahngleichung (5.43) geschlossen lösbar und liefert Kegelschnitte mit der Sonne in einem Brennpunkt. Für $E < 0$ sind es Ellipsen.

5.12 Mehrkörperprobleme

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

$$m_i \, \ddot{\vec{r}}_i = -Gm_i \sum_{i \neq j=1}^{N} \frac{m_j}{|\vec{r}_i - \vec{r}_j|^3} \, (\vec{r}_i - \vec{r}_j), \quad i = 1 \ldots N. \tag{5.46}$$

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 [3], dies ist hier jedoch zu aufwendig.

5.13 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. 12. Die beiden ersten Argumente von **subplot** sind m und n und

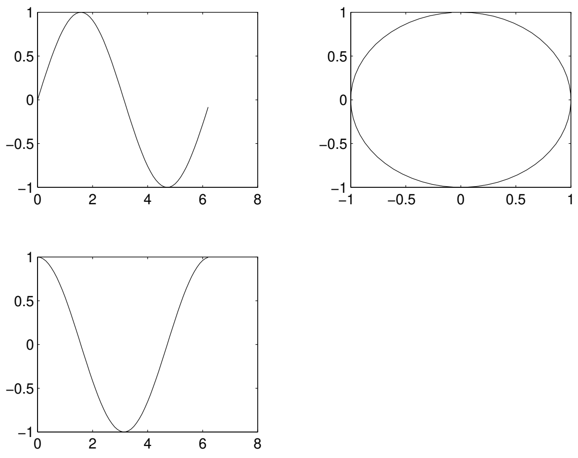

Abbildung 12: Anwendung von **subplot**

.

das dritte die Plotnummer innerhalb der Matrix, wobei zeilenweise von links oben nach rechts unten durchgezählt wird.

6 Molekulardynamik

In diesem Abschnitt wollen wir ein Vielkörper Problem untersuchen. Eine große Zahl von Molekülen mit gewissen entfernungsabhängigen Paarkräften soll sich in einem Behälter befinden. Mit mehr oder weniger zufälligen Anfangsbedingungen werden dann ihre Bahnen mit Hilfe der Newton Mechanik berechnet. Es ähnelt so einem Gas wie es in der Thermodynamik betrachtet wird, und in der Tat werden wir schon mit moderat vielen Teilchen dort beschriebene Phänomene sehen. Ohne numerische Methoden lässt sich bereits ein entsprechendes Dreikörper Problem nicht mehr exakt lösen. Die Darstellung hier ist [4] entnommen, wobei es aber einige Änderungen gibt.

6.1 Vorbetrachtung

Zunächst wollen wir kurz diskutieren, wann eine solche Beschreibung mit punktförmigen Molekülen, die klassisch miteinander wechselwirken, überhaupt eine sinnvolle Approximation ist. Klarerweise haben wir es im Prinzip mit Quantenmechanik zu tun, und spätestens, wenn die Moleküle sich so nahe kommen, dass die Elekronenschalen ineinander gequetscht werden, spielt deren diskreter Aufbau, das Pauli Prinzip etc. eine Rolle. Es gibt jedoch Situationen, wo Materie eine genügend kleine Dichte und auch sonst die nötigen Eigenschaften hat, so dass wir eine akzeptable klassische Näherung betrachten können. Das ist vielleicht ähnlich wie man Planeten und Sonne als Punktteilchen nähern kann trotz der zweifellos vorhandenen Ausdehnung. Dort ist die Kleinheit des Verhältnisses Durchmesser zu Bahnradius entscheidend. Solche Näherungen sind in der Physik immer eine Frage der vorhandenen Skalen, genauer von kleinen dimensionslosen Verhältnissen solcher. Im vorliegenden Fall kann man Energien vergleichen. Die Bindungsenergie von Elektronen in Atomen oder Molekülen liegt in der Größenordnung 10 eV, während die thermische kinetische Energie bei Raumtemperatur bei 26 meV liegt. Bei solchen Kollisionen kann also ein Molekül nicht zerreißen. Eine weitere Bestätigung liefern die involvierten quantenmechanischen Längenskalen. Die quantenmechanischen deBroglie Wellenlängen von Atomen oder Molekülen bei Raumtemperatur liegt typisch unterhalb von 1 Å. Dadurch werden alle quantenmechanischen Interferenzeffekte 'verschmiert' und der klassische Limes ist eine sehr gute Näherung (wie geometrische Optik zur Wellenoptik), solange die relevanten Abstände deutlich größer sind. Dies ist der Fall bei hinreichend kleiner Dichte, wo man weiß, dass Gase klas-

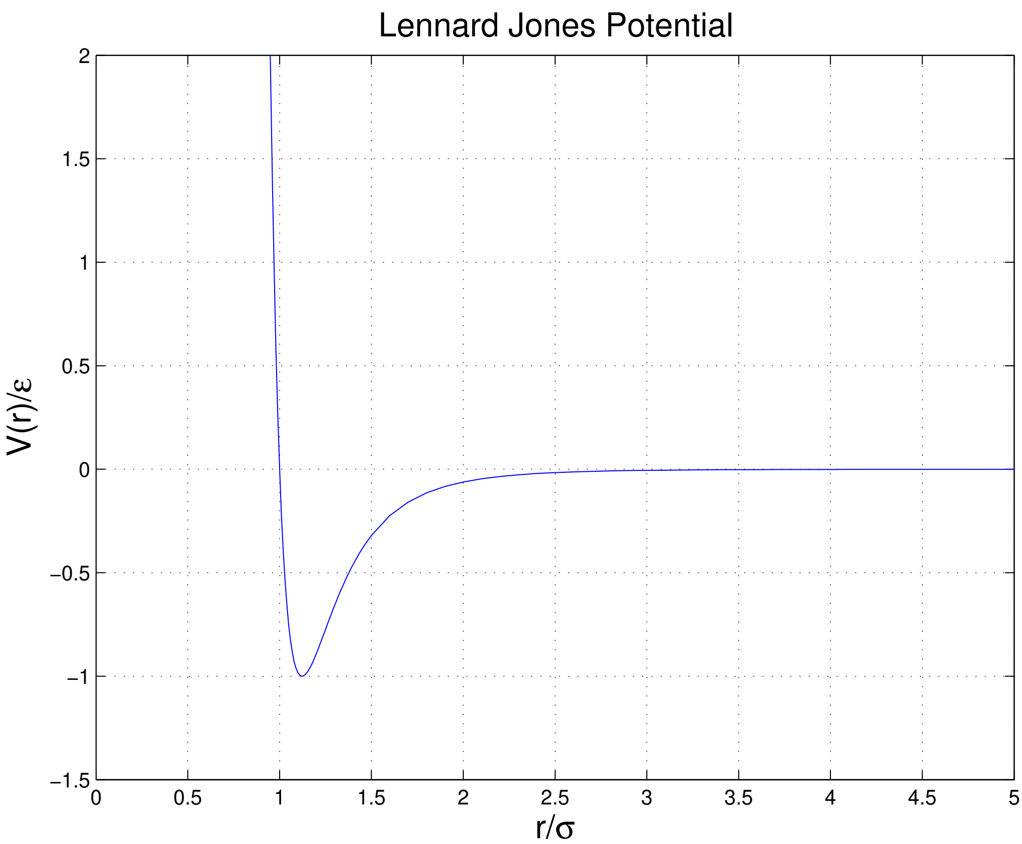

Abbildung 13: Plot des Potentials zwischen Gasteilchen

sisch genähert werden können. Im Prinzip ist dies alles auch abhängig von der Wechselwirkung, für die wir nun eine empirisch bewährte Form angeben werden. Ein System in der Natur, wo alle unsere Näherungen relativ gut erfüllt sind, ist das Edelgas Argon, von dessen Simulation wir daher reden werden.

Die elektromagnetischen Kräfte nebst Quanteneffekten zwischen den Bestandteilen der Moleküle führen bei geeigneten Abständen zu einer durch das Lennard-Jones Potential gegeben effektiven 'Restkraft' zwischen neutralen Molekülen. Zwischen zwei Molekülen im Abstand r herrscht das Potential

$$V_{\mathrm{LJ}}(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]. \tag{6.1}$$

Kommentare dazu:

- s. Fig. 13
- 2 Terme und 2 freie Konstanten: σ [Länge] und $\epsilon > 0$ [Energie]
- beide Kraftkomponenten schnell abfallend
- abstoßend bei sehr kurzen Abständen, anziehend bei mittleren, dann schnell verwindend

Diese Kraft ist eine genäherte Beschreibung mit 2 Parametern von viel komplizierteren mikroskopischen Dingen. Es handelt sich nicht um eine fundamentale Wechselwirkung wie die Coulomb Kraft, sondern eher um so etwas wie bei Reibungstermen. Die beiden Effekte heben sich gerade auf, wo $V'_{\mathrm{LJ}} = 0$ gilt, bei $r \approx 1.12\sigma$.

Wie schon im letzten Abschnitt diskutiert ist es vereinfachend und auch numerisch besser, dem Problem angepasste Einheiten zu wählen. Die folgenden Rechnungen sind durch das Lennard-Jones Potential dominiert, und hier ist es praktisch, Energien in Vielfachen von ϵ und Längen in Einheiten σ zu messen, oder, wie Physiker gerne sagen, $\epsilon = 1$ und $\sigma = 1$ 'zu setzen'. Wenn man dieses auf physikalisches Argon bezieht, so entspricht es $\sigma = 3.4$ Å und ϵ ist gleich der thermischen Energie $k_B T$ bei $T = 120$K. Man hat noch eine dritte Einheit frei, und wir setzten noch die Masse des Argon Atoms $m = 1$ im obigen Sinn. Damit ist aber alles fixiert, und die Einheit der Zeit ist z.B. gegeben durch $\sqrt{m\sigma^2/\epsilon}$, etwa 2 Pico-Sekunden.

Für die numerische Simulation hier werden wie auch an verschiedenen anderen Stellen in diesem Kurs die physikalischen drei Raumdimensionen durch eine nur zweidimensionale Welt ersetzt. Damit wird die Näherung natürlich quantitativ wenig sinnvoll, es sei denn es gibt in der Natur dünne Filme, wo die Bewegung tatsächlich so eingeschränkt ist, was vorkommt. So ist das hier aber nicht gemeint, sondern es wird sich zeigen, dass man mit viel weniger Rechenleistung die meisten Effekte qualitativ korrekt zu sehen bekommt. Auch hat die Reduktion für die Visualisierung auf dem Bildschirm offensichtliche Vorteile. Ein zweidimensionaler Container für ein Gas wäre nun im einfachsten Fall ein Quadrat mit 'Volumen' L^2, d.h. den Atomen stehen z.B. die Ortsvektoren (x, y) zur Verfügung mit $0 \le x, y \le L$.

Die Begrenzung wird durch Materie gebildet, in die unsere Gasteilchen nicht eindringen können. Stattdessen werden sie annähernd elastisch reflektiert, wenn sie auf einen der Ränder zufliegen. Wir erfassen dies, indem wir ein Potenzial einführen, das im Inneren des Containers praktisch verschwindet und an den Rändern steil ansteigt auf Energien, die Teilchen (bei einer gewissen Temperatur) nicht besitzen. Mehr oder weniger ad hoc können wir z. B. nehmen

$$V_{\text{Box}}^{(\lambda,B)}(\vec{r}) = B\{\exp(-x/\lambda) + \exp(-(L-x)/\lambda) + (x \to y)\}, \tag{6.2}$$

Der Parameter λ stellt die Wanddicke dar und B ist die Energieskala wenn Teilchen sich der Wand auf $\mathrm{O}(\lambda)$ nähern. In den Anwendungen werden wir den idealisierten Limes

$$V_{\text{Box}}^{\text{Step}}(\vec{r}) = \lim_{\lambda \to 0_+, B>0} V_{\text{Box}}^{(\lambda,B)}(\vec{r}) \tag{6.3}$$

betrachten. Dieses Potential verschwindet im Inneren und ist aussen unendlich. Die Wanddicke ist Null. Seine Behandlung wird allerdings besondere Massnahmen erfordern.

Eine interessante Möglichkeit wäre auch ein runder Container, der die Rotationsinvarinz nicht bricht. Man könnte ihn analog mit dem Potenzial

$$V_{\text{Disk}}(\vec{r}) = B\exp(-(R-|\vec{r}|)/\lambda) \tag{6.4}$$

realisieren. Eine weitere Alternative wären periodische Randbedingungen. Die festen Wände erscheinen bei diesem Problem aber physikalisch plausibler.

6.2 Bewegungsgleichungen und Leapfrog Algorithmus

Wir betrachten N Teilchen mit Koordinaten $\vec{r}_i = (x_i, y_i)$, die paarweise über das Lennard-Jones Potential aufeinander wirken und ausserdem jeweils das Box-Potenzial spüren. Die gesamte Hamilton Funktion (Energie) lautet

$$H(\vec{r}_i, \vec{v}_i) = \sum_i \left(\frac{1}{2}\vec{v}_i^2 + V_{\text{Box}}^{(\lambda,B)}(\vec{r}_i) \right) + \sum_{i<j} V_{\text{LJ}}(|\vec{r}_i - \vec{r}_j|), \tag{6.5}$$

wobei wir hier nicht zwischen kanonischen Impulsen und Geschwindigkeiten unterscheiden müssen ($m = 1$).

Die kanonischen Bewegungsgleichungen lauten damit

$$\dot{\vec{r}}_i = \vec{v}_i \tag{6.6}$$

$$\dot{\vec{v}}_i = -\vec{\nabla}_i \left[V_{\mathrm{Box}}^{(\lambda,B)}(\vec{r}_i) + \sum_{j \neq i} V_{\mathrm{LJ}}(|\vec{r}_i - \vec{r}_j|) \right] \tag{6.7}$$

mit $i, j = 1, \ldots, N$ und $\nabla_i = (\partial/\partial x_i, \partial/\partial y_i)$. Die Bewegungsgleichungen sind somit schon in userer Standardform (wenn wir alle Komponenten $\vec{r}_i, \vec{v}_i$ in ein $\vec{y}$ stecken) und man könnte das Problem gut mit `ode45` attackieren. Wir wollen hier jedoch noch einen neuen äusserst einfachen Integrator kennenlernen, den Leapfrog Algorithmus.

Leapfrog ist dem Eulerverfahren sehr ähnlich, aber symmetrischer und daher neben anderen guten Eigenschaften eine Ordnung genauer. Wir wählen eine Zeitdiskretisierung so, dass wir die Orte zu Vielfachen der Zeit τ betrachten, $\vec{r}_i(n\tau), n = 0, 1, 2,$ die Impulse jedoch dazwischen, $\vec{v}_i((n + 1/2)\tau)$. Damit lautet der Euler-artige Rekursionsschritt, wenn beide Größen für ein ganzahliges n bereits vorliegen,

$$\vec{r}_i((n+1)\tau) = \vec{r}_i(n\tau) + \tau\vec{v}_i((n+1/2)\tau) \tag{6.8}$$

$$\vec{v}_i((n+3/2)\tau) = \vec{v}_i((n+1/2)\tau) + \tau\vec{F}_i[\vec{r}_j((n+1)\tau)]. \tag{6.9}$$

Die Kraft $\vec{F}_i$ ist der negative Gradient des Potenzial wie in (6.7). Entscheidend ist, dass die gerade neu berechneten Orte nach dem Differenzieren ins Argument gesetzt werden.

Der gerade beschriebene Algorithmus ist exakt reversibel. Das heisst, man kann die Richtung umkehren, formal durch $\tau \to -\tau$ und geht dann durch *dieselben* Werte (bis auf Rundungsfehler) wieder zurück. Diese Eigenschaft der exakten DGL wird also trotz Diskretisierung erfüllt, sozusagen auch von den Diskretisierunsfehlern. Dadurch kann man auch ablesen, dass der Einzelschrittfehler $\mathrm{O}(\tau^3)$ ist, da gerade Potenzen ausgeschlossen sind. Darüber hinaus handelt es sich um einen symplektischen Integrator, was impliziert dass er das Liouville Theorem erfüllt. Evolviert man ein 'Bahnenbüschel' mit Leapfrog – generiert also einen Fluss im Phasenraum – so ist dieser inkompressibel oder Volumen erhaltend.

Hat man ein normales Anfangswertproblem mit $\vec{r}_i(0)$ und $\vec{v}_i(0)$ vorgegeben, so muss man noch einen extra Halbschritt am Anfang machen,

$$\vec{v}_i(\tau/2) = \vec{v}_i(0) + \frac{\tau}{2}\vec{F}_i[\vec{r}_j(0)], \tag{6.10}$$

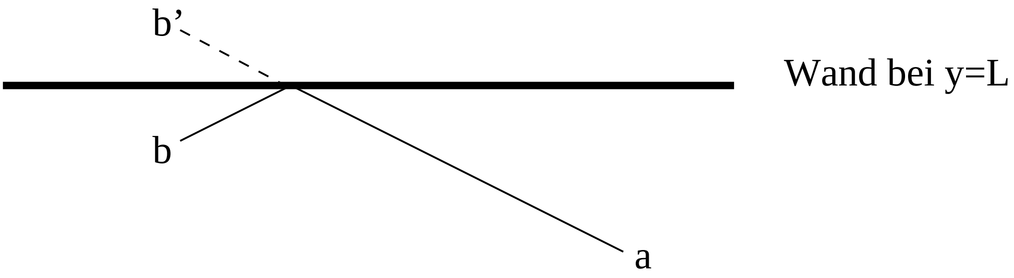

Abbildung 14: Ein Teilchen, das gemäß (6.8) von a nach b' laufen würde, wird durch Berücksichtigung von $V_{\mathrm{Box}}^{\mathrm{Step}}$ reflektiert und landet bei b. Dabei wird das Vorzeichen von $v_y((n+1/2)\tau)$ geflippt und damit dann weitergerechnet.

was hier, wo wir die Impulse sowieso zufällig wählen werden, aber nicht benötigt wird.

Wir wissen aus dem Kapitel Anfangswertprobleme, dass dort, wo Kräfte groß werden, eine besonders feine Zeit Diskretisierung erforderlich ist. Das ist hier insbesondere der Fall, wenn der Term $-\vec{\nabla}_i V_{\mathrm{Box}}^{(\lambda,B)}(\vec{r}_i)$ in (6.7) für die Reflektion eines Teilchens an der Wand sorgt. Tatsächlich wollen wir den Prozess der Umkehr nicht genau simulieren, sondern nur das Teilchen in der Box halten. Betrachten wir (6.8),(6.9), so sehen wir, dass beim Leapfrog Algorithmus die Bewegung zerlegt wird in Folgen von kleinen Stücken freier gleichförmiger Bewegung (6.8) und dazwischen Korrekturen der Geschwindigkeiten. Die Strategie wird nun sein, $\vec{F}_i$ in (6.9) *nur noch* mit dem Paarpotential V_{LJ} zu bilden und die idealisierte Box $V_{\mathrm{Box}}^{\mathrm{Step}}$ in dem gleichförmigen Teil der Bewegung zu berücksichtigen.

Bei diesen Abschnitten evolvieren die Teilchen unabhängig voneinander, jedes mit seiner (momentanen) Geschwindigkeit. Bleibt ein Teilchen beim Schritt (6.8) in der Box, so hat $V_{\mathrm{Box}}^{\mathrm{Step}}$ keinerlei Wirkung auf es. Kreuzt aber die Gerade von $\vec{r}_i(n\tau)$ (Punkt a in Abb.14) nach $\vec{r}_i(n\tau) + \tau\vec{v}_i((n+1/2)\tau)$ (Punkt b') eine der Box Wände, so wird das Teilchen elastisch reflektiert an der (unendlichen) Potentialstufe, wobei die Geschwindigkeits- bzw. Impulskomponente *senkrecht* zur Wand das Vorzeichen wechselt. Dies ist in Abb.14 gezeigt und wird einfach entsprechend im Code durchgeführt.

Es ist noch zu beachten, dass $V_{\mathrm{Box}}^{\mathrm{Step}}$ keinen Beitrag zur erhaltenen potenziellen Energie gibt, da alle Teilchen durch die Reflektionen strikt in der Box bleiben, wo $V_{\mathrm{Box}}^{\mathrm{Step}}$ ja verschwindet.

6.3 MATLAB Realisierung

Unser Leapfrog Integrator (einschließlich Reflektionschritten) besteht aus dem folgenden weitgehend selbsterklärenden MATLAB code

```
function [X,Y,VX,VY]=leapfrog(x,y,vx,vy,L,tau,nstep)

N=length(x); % Teilchenzahl
X =zeros(N,nstep+1); Y =zeros(size(X));
VX=zeros(N,nstep+1); VY=zeros(size(X));

 X(:,1)= x;  Y(:,1)= y;             % Startwerte
VX(:,1)=vx; VY(:,1)=vy;

for n=1:nstep
    X(:,n+1)=X(:,n)+tau*VX(:,n); % Orte: n*tau -> (n+1)*tau
    Y(:,n+1)=Y(:,n)+tau*VY(:,n); % mittels v bei (n+1/2)*tau
    % Reflektieren <-> V_Box^Step:
    for i=1:N
        if X(i,n+1)<0, X(i,n+1)=   -X(i,n+1); VX(i,n)=-VX(i,n); end
        if X(i,n+1)>L, X(i,n+1)=2*L-X(i,n+1); VX(i,n)=-VX(i,n); end
        if Y(i,n+1)<0, Y(i,n+1)=   -Y(i,n+1); VY(i,n)=-VY(i,n); end
        if Y(i,n+1)>L, Y(i,n+1)=2*L-Y(i,n+1); VY(i,n)=-VY(i,n); end
    end

    [fx,fy]=LJkraft(X(:,n+1),Y(:,n+1));

    VX(:,n+1)=VX(:,n)+tau*fx; % Impulse: (n+1/2)*tau -> (n+3/2)*tau
    VY(:,n+1)=VY(:,n)+tau*fy; % mittels fx,fy bei (n+1)*tau
end
```

Im Argument sind die Anfangswerte bei $t = 0$ bzw. $t = \tau/2$. Es werden `nstep` leapfrog Schritte gemacht und alles in einer Matrix gespeichert, ähnlich wie bei `ode`. Da es in MATLAB natürlich nur ganzzahlige Indizes gibt, werden die Komponenten von $\vec{v}_i((n+1/2)\tau)$ in den gleichen Matrix Spalten gespeichert wie $\vec{r}_i(n\tau)$.

Die aufgerufene Kraft wird von folgender Routine berechnet

```
function [fx fy] = LJkraft(x,y)
```

```
2     % file LJkraft.m
3     %
4     % Kraft aus Lennard-Jones Potential fuer N Teilchen (2dim.)
5     N=length(x);
6     fx=zeros(size(x)); fy=zeros(size(fx));     % reservieren
7     % loop ueber Paare von verschiedenen Teilchen (jedes Paar 1 mal, i<j !)
8     %
9     for i=1:N-1
10        for j=i+1:N
11        r2=(x(i)-x(j))^2+(y(i)-y(j))^2; % Abstand^2
12        ri2=1/r2;
13        fac=24*ri2^4*(2*ri2^3-1);
14        fx(i)=fx(i)+(x(i)-x(j))*fac;% Beitrag dieser Paarkraft AUF i
15        fy(i)=fy(i)+(y(i)-y(j))*fac;%
16        fx(j)=fx(j)-(x(i)-x(j))*fac;% Beitrag dieser Paarkraft AUF j
17        fy(j)=fy(j)-(y(i)-y(j))*fac;% Vorz: actio=reactio
18        end % loop j
19    end     % loop i; alle Kraefte beruecksichtigt
```

Bemerkungen zu diesem **function** m-file, wobei schon bekannte MATLAB Kommandos nicht wieder kommentiert werden:

- Z. 5: Anzahl Teilchen N wird aus Argument x bestimmt
- Z. 9,10: Doppelloop über Teilchenpaare $i < j$
- Z. 14,15: Kraftwirkung auf Teilchen i
- Z. 16,17: Kraftwirkung auf Teilchen j, umgekehrtes Vorzeichen, actio = reactio

Man überzeuge sich, dass der LJ-Teil der Kräfte (6.7) nun berücksichtigt ist (insbesondere keine fehlenden Faktoren 2 oder 1/2).

Die Kosten der Kraftberechnung wachsen proportional zu N^2 und limitieren für viele Teilchen die sinnvoll auf einem gegebenen Rechner simulierbare Problemgröße. Prinzipiell gäbe es Möglichkeiten zu beschleunigen. Die Genauigkeit des Integrators könnte vielleicht niedriger sein, also größere Schrittweite bei leapfrog. Da die Kraft bei großen Abständen schnell zu Null geht, könnte man dort versuchen, die Rechnung zu vereinfachen. All diese Fehler müssten aber kontrolliert werden, was die Sache verkompliziert.

Die Berechnung von Paarkräften ist ein wichtiges Problem, und die effektive Implementierung auf Parallelrechnern ist ein algorithmischer Forschungszweig. In [5] wird z.B. ein Algorithmus diskutiert, der die Skalierung zumindest effektiv auf das Verhalten $\propto N^{3/2}$ verbessern soll.

Nun betrachten wir noch eine Testanwendung. Das Skript ist:

```
% testprogramm Molekulardynamik
clf;hold off; % ggf. Plotfenster freigeben

L=6;  % Kantenlaenge der Box

% Anfangskonfiguration:
N=5*5;  % Teilchenzahl
x=zeros(N,1); y=zeros(N,1); % reservieren
vx=zeros(N,1); vy=zeros(N,1); % reservieren

i=0;
for yi=1:5
    for xi=1:5
        i=i+1;
        x(i)=xi; y(i)=yi;
    end
end

% Bild Startkonfiguration:
plot(x,y,'*r');axis([0 L 0 L]);hold on;
pause;

% zufaellige Anfangsimpulse
vmax=1; % Maximalwert
vx=vmax*(2*rand(N,1)-1);vy=vmax*(2*rand(N,1)-1); % 0 < rand < 1 zufaellig
vx=vx-mean(vx);vy=vy-mean(vy); % Gesamtimpuls Null

tf=20;              % Zielzeit der Integration
tau=0.005;          % Schrittweite
nstep=fix(tf/tau);  % Anzahl Schritte

[X,Y,VX,VY]=leapfrog(x,y,vx,vy,L,tau,nstep);
```

```
33
34    % plot aller Trajektorien mit duennen Punkten
35    for i=1:N
36        plot(X(i,:),Y(i,:),'MarkerSize',1,'Marker','.','LineStyle','none');
37    end
38    pause
39    print -depsc molecular.eps
40
41    % zeitliche Folge von Konfigurationen
42    strob=fix((tf/50)/tau); % Haeufigkeit Plot
43    for n=1:strob:nstep+1
44        clf
45        plot(X(:,n),Y(:,n),'.');axis([0 L 0 L])
46        pause(0.2)
47    end
48    %print -depsc mol_final.eps
49    % Parameter und letzte Konfiguration speichern
50    x=X(:,nstep+1);y=Y(:,nstep+1); vx=VX(:,nstep+1);vy=VY(:,nstep+1);
51    save gas.mat N L x y vx vy
```

Das meiste erklärt sich über Kommentarzeilen, aber hier noch einige Kommentare:

- Z. 6–17: hier werden Teilchen in eine willkürliche Anfangskonfiguration gesetzt mit Abstand 1. Nimmt man hier wesentlich weniger als das Minimum von V_{LJ}, so führt man eine sehr hohe Anfangsenergie ein.

- Z. 23–26: hier werden zufällige Anfangsgeschwindigkeiten mit Maximalbetrag 1 in jeder Richtung gewählt bei ruhendem Schwerpunkt.

- Z. 50,51 speichert Parameter und letzte Konfiguration; man kann diese z. B. als Start zum Weiterintegrieren später einlesen.

In Fig.15 sehen wir nun die Startkonfiguration (Sterne) und die daraus erwachsende Konfiguration (Linien). Im unteren Teil des Skript wird eine Art Film der aufeinander folgenden Konfigurationen produziert, den wir hier nicht sehen können. In Fig.16 ist die letzte solche Konfiguration gezeigt. Sie hat keinerlei Ähnlichkeit mehr mit den Startwerten.

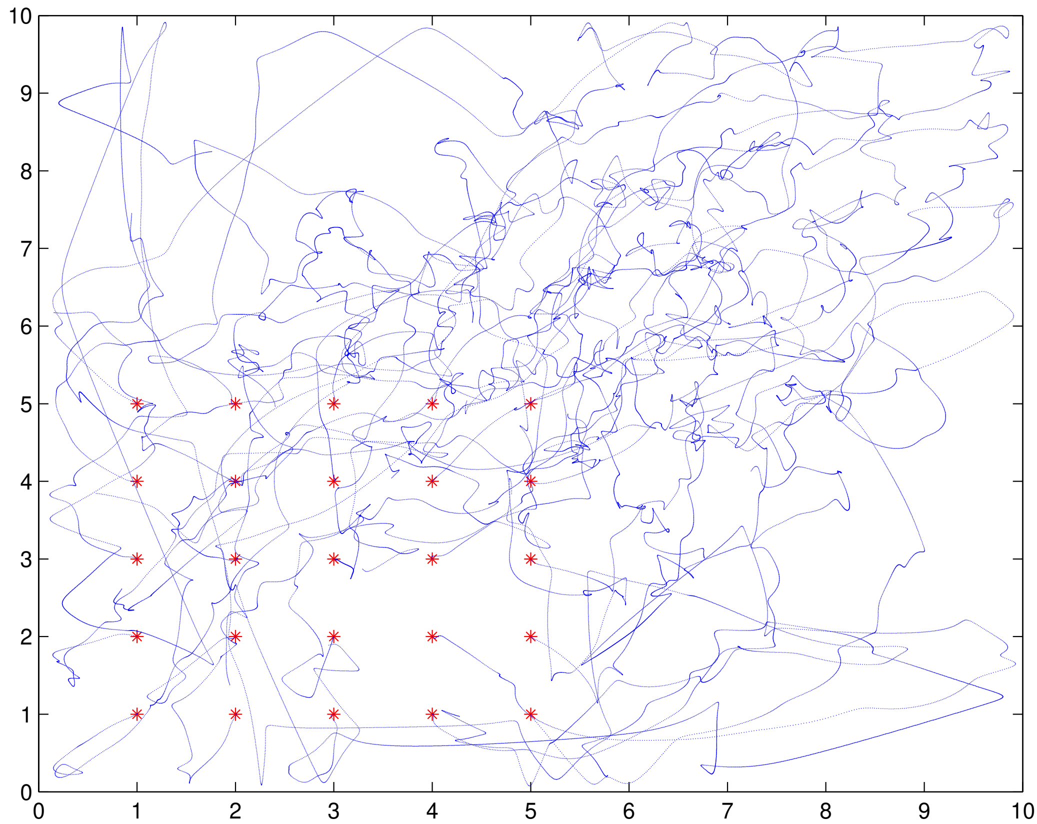

Abbildung 15: Anfangspositionen und Trajektorien, $t = 0, .., 20$.

6.4 Interpretation

Wir haben Teilchen in einem abgeschlossenen Volumen mit gegenseitiger semi-realistischer molekularer Wechselwirkung elvolvieren lassen. Obwohl die Teilchenzahl $N = 25$ nicht allzu makroskopisch ist, haben wir die Situation, die Gegenstand der Thermodynamik ist. Die Energie in unserem System

$$E = E_{\rm kin} + E_{\rm pot} = \frac{1}{2}\sum_{i=1}^{N}(\dot{\vec{r}}_i)^2 + \sum_{i<j} V_{\rm LJ}(\vec{r}_i - \vec{r}_j) \tag{6.11}$$

ist exakt erhalten. Modelliert man ein endliches Box-Potenzial, so kommt bei $E_{\rm pot}$ noch $\sum_i V_{\rm Box}^{(\lambda,B)}(\vec{r}_i)$ hinzu. Die Gesamtenergie ist also durch unseren Anfangszustand bestimmt. Die Thermodynamik lebt davon, dass eine solche

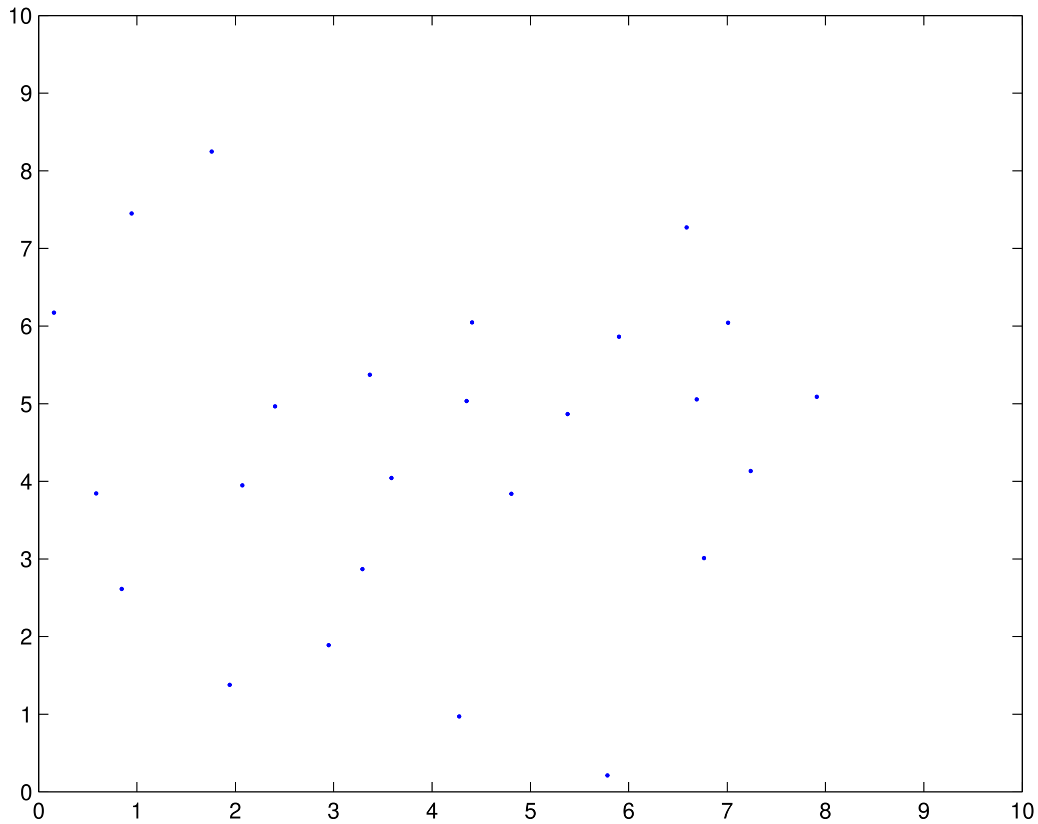

Abbildung 16: Endkonfiguration im vorherigen Bild.

Konfiguration nach einiger Zeit, wo sie noch durch unseren willkürlichen Start bestimmt ist, ins thermodynamische Gleichgewicht kommt. Dann werden 'typische' Konfigurationen durchlaufen, über deren mittlere Eigenschaften die statistische Physik bzw. Thermodynamik Aussagen machen kann mit Größen wie Temperatur, Druck etc. Das ist nicht die komplette Information, aber es geht auch in Fällen, wo das Lösen der mikroskopischen Bewegungsgleichungen hoffnungslos ist. Hier schaffen wir das noch numerisch und können die resultierende Bewegung trotz nicht sehr großem N nun thermodynamisch analysieren.

Ein abgeschlossenes System mit fester Energie nennt man mikrokanonisch. Im Gleichgewicht gehört dann zu jeder Energie eine Temperatur, die man aus der mittleren kinetischen Energie pro Teilchen ableiten kann

$$\frac{1}{2}k_B T = \langle\frac{1}{2}v_x^2\rangle = \langle\frac{1}{2}v_y^2\rangle. \tag{6.12}$$

Die Mittelung kann hier sowohl über die N Gasteilchen als auch über zeitlich aufeinanderfolgende Konfigurationen geführt werden, nachdem das System im Gleichgewicht ist. Ein notwendiges (nicht hinreichendes) Kriterium ist, dass sowohl E_{kin} als auch E_{pot} um einen mittleren Wert herum fluktuieren. Ihre Summe ist fixiert, aber die Aufteilung 'wählt' das System im thermischen Gleichgewicht selbst. Erst dann kann man von Temperatur reden. Hier schwankt diese Größe natürlich noch, bei $N = 10^{23}$ würde sich aber relativ zu den (extensiven) Mittelwerten nicht mehr viel tun (Gesetz der großen Zahlen). Weiteres zur Thermodynamik unseres 2-D Argon, z.B. was bei weiterem Abkühlen passiert, wird in den Übungen untersucht.

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

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

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 $\{x_i, i = 0 \dots n\}$, 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 in führender Ordnung 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

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

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:

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

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)*(func(a) + 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 + 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 [2]. 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 $P_n(x)$:

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

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 $P_0(x), \dots, P_{n-1}(x)$ 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 [2] beschrieben. Man findet die Parameter aber auch in Büchern wie der 'Zahlenbibel' [6], 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
.148874338981631d0      .295524224714753d0
.433395394129247d0      .269266719309996d0
.679409568299024d0      .219086362515982d0
.865063366688985d0      .149451349150581d0
.973906528517172d0      .066671344308688d0
```

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 `quadl`. 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 wir leistungsfähige Verfahren, auch solche mit automatisch sich anpassender Schrittweite ja schon im Kapitel Anfangswertprobleme kennengelernt haben.

8 Lineare Gleichungssysteme

Die natürlichste Fortsetzung des Abschnitts 'Nullstellensuche' wäre es, viele simultan zu lösende nichtlineare Gleichungen zu betrachten. Hier gibt es bei voller Nichtlinearität allerdings keine allgemein tauglichen "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

$$\vec{f}(\vec{x}) = 0 \tag{8.1}$$

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

$$f_i(\vec{x}) = \sum_{j=1}^{n} a_{ij} x_j - b_i,\ i = 1, \ldots, m. \tag{8.2}$$

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.

8.1 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[10]

$$Ax = b. \tag{8.3}$$

Nun wird das a_{i1}/a_{11} fache der ersten Gleichung (=Zeile) von der 2. bis 4. abgezogen, wobei sich die Lösung ja nicht ändert. Dann werden offenbar in der modifizierten Koeffizientenmatrix 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
```

[10]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 *unten* 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

$$x_1 = 3, x_2 = 1, x_3 = -2, x_4 = 1. \tag{8.4}$$

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
  for i=k+1:n    % loop ueber zu subtrahierende Zeilen
    aux=B(i,k)/B(k,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)*aux; % 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:

$$\sum_{j=i}^{n} a_{ij}x_j = b_i \Rightarrow x_k = \frac{b_k - \sum_{j=k+1}^{n} a_{kj}x_j}{a_{kk}}. \tag{8.5}$$

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 alle Spalten der Einheitsmatrix an, 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/kap8`.

Wir wollen nun einen Test machen zur numerischen Genauigkeit unserer Routine. Dazu betrachten wir 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.17 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.

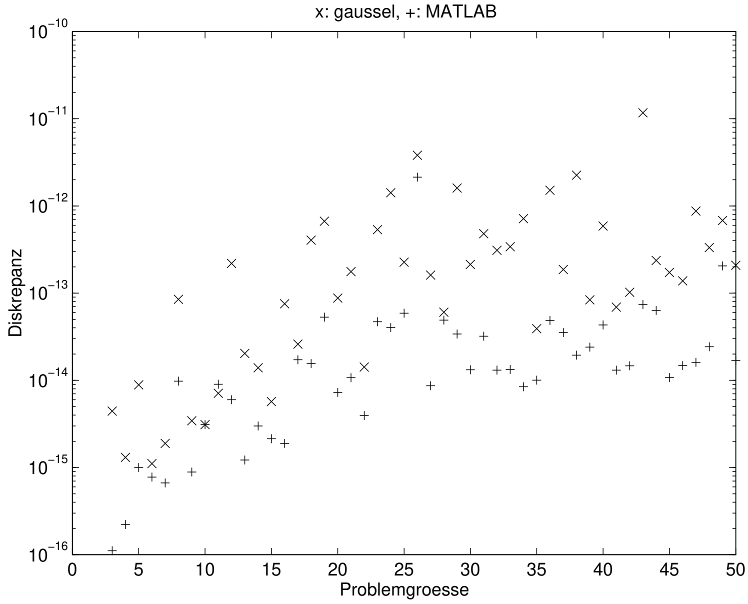

Abbildung 17: Fehler beim Lösen als Funktion der Problemgröße mit Zufallsmatrizen

Nur nebenbei soll erwähnt werden, daß unser Programm etwa 110 mal mehr CPU-Zeit für das Lösen brauchte, nämlich ca. 7.7 Sek. auf einem Linux-PC mit Intel Pentium 4 CPU/2.66 GHz für $n = 2 \ldots 100$. 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.

8.2 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 zusätzlich 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 [2] 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[11] 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 nun nur noch dann 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 [1] lohnt der Zusatzaufwand der skalierten Pivotisierung gegenüber der einfach partiellen oft nicht. Das gilt wohl, wenn die Matrixelemente "natürliche" Größen haben.

8.3 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 verbessern. Sei x^* die exakte Lösung der Gleichung und $\delta x = x^{(0)} - x^*$ der

[11] Natürlich keine Vertauschung, falls gerade $|a_{11}|$ maximal ist

Fehler unserer numerischen Lösung. Es gilt

$$Ax^* = A(x^{(0)} - \delta x) = \tilde{b} - \delta b = b \quad , \tag{8.6}$$

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

$$A\delta x = \delta b \tag{8.7}$$

als Bestimmungsgleichung für δx, die wir mit dem gleichen Verfahren lösen wie die ursprüngliche Gleichung. Die verbesserte Lösung ist $x^{(1)} = x^{(0)} - \delta x$.

Der Aufwand wird bei Verwendung von Gauß-Elimination verdoppelt. Beim nun diskutierten Verfahren der LU–Zerlegung wird eine zweite Lösung mit derselben Matrix (und anderer rechter Seite) wesentlich billiger erhältlich sein, insbesondere für große n.

8.4 LU–Zerlegung

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

$$A = LU, \quad l_{ij} = u_{ji} = 0 \text{ wenn } i < j \tag{8.8}$$

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 die Diagonalelemente von L zu Eins machen kann[12] , $l_{ii} = 1$. 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:

$$Ly = b \tag{8.9}$$

$$Ux = y. \tag{8.10}$$

Der zweite Schritt ist genau die Rückwärtssubstitution aus dem letzten Abschnitt in (8.5). 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 ja mit-manipuliert werden mußten.

Zur Faktorisierung ist es nützlich, sie in Komponenten auszuschreiben:

$$a_{ij} = \sum_{k=1}^{\min(i,j)} l_{ik} u_{kj}. \tag{8.11}$$

[12]Konvention, U ginge auch stattdessen

Für $i \leq j$ kann man daraus bekommen

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \tag{8.12}$$

und für $i > j$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right). \tag{8.13}$$

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[2].

Auch hier ist es i. a. erforderlich zu pivotisieren, was etwas trickreich ist [2]. 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 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

$$c_i = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \; i = j, j+1, \ldots, n \tag{8.14}$$

was sowohl in (8.12) vorkommt als auch in (8.13), 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, $k > j$, eine Vertauschung von c_j mit c_k impliziert. 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 durch Umspeichern 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 (8.12), (8.13) 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 [2] 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 =

   8.1069e-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.

8.5 Householder–Reduktion

Die Householder–Reduktion ist charakterisiert durch geschickt konstruierte (orthogonale) Transformationen, die besonders gut geeignet 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

$$x \to x - 2w(w^T x) = Px; \quad P = 1 - 2ww^T. \tag{8.15}$$

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

$$w = \frac{x-y}{|x-y|}. \tag{8.16}$$

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

$$a^{(1)} \to -\sigma_1 e_1 \tag{8.17}$$

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

$$\sigma_1 = \pm|a^{(1)}|, \tag{8.18}$$

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.(8.16):

$$\begin{aligned}
P_1 &= 1 - \frac{uu^T}{H} && (8.19)\\
\text{mit}\quad u^T &= (a_{11}+\sigma_1, a_{21}, \cdots, a_{n1}) && (8.20)\\
\sigma_1^2 &= \sum_{i=1}^{n}(a_{i1})^2 && (8.21)\\
H &= \frac{1}{2}|u|^2. && (8.22)
\end{aligned}$$

Nun läßt sich der Normierungsfaktor H umformen zu

$$\begin{aligned}
H &= \frac{1}{2}(|a^{(1)}|^2 + 2\sigma_1 a_{11} + \sigma_1^2)\\
&= \sigma_1(\sigma_1 + a_{11}) && (8.23)
\end{aligned}$$

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

$$\sigma_1 = \mathrm{sgn}(a_{11})\,|a^{(1)}| \tag{8.24}$$

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 $P_1 Ax = P_1 b$.

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

$$u^T = (0, a'_{22} + \sigma_2, a'_{32}, \cdots, a'_{n2}) \tag{8.25}$$

$$\text{mit} \quad \sigma_2^2 = \sum_{i=2}^{n} (a'_{i2})^2 \tag{8.26}$$

$$\mathrm{sgn}(\sigma_2) = \mathrm{sgn}(a'_{22}) \tag{8.27}$$

$$\Rightarrow P_2 = 1 - \frac{uu^T}{H} \tag{8.28}$$

$$\text{mit} \quad H = \sigma_2(\sigma_2 + a'_{22}) \tag{8.29}$$

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

$$Q^T = P_{n-1} P_{n-2} \cdots P_2 P_1 \tag{8.30}$$

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:

$$Q = P_1 P_2 \cdots P_{n-2} P_{n-1} \tag{8.31}$$

$$Q^T Q = 1. \tag{8.32}$$

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. Bei dieser Zählung ist zu beachten, dass die Multiplikation eines Vektors mit einem Projektor $P = 1 - 2ww^T$ natürlich **mit nur** $2n$ **Multiplikationen** realisiert wird und nicht, wie bei allgemeinen Matrizen, n^2 kostet[13].

Mit dem größeren Aufwand bei Householder erkauft man sich den Vorteil größerer numerischer Stabilität, und man braucht nicht zu pivotisieren. Das ist auf manchen Parallelrechner Architekturen ein wichtiger Punkt.

[13] Auch bei Übungsaufgaben beachten!

9 Das Fitten von Daten

9.1 Normalverteilte Messwerte, χ^2

Wir nehmen an, dass eine gemessene physikalische Größe Y durch ein bekanntes gegebenes Gesetz

$$Y = f(x) \tag{9.1}$$

von einem frei wählbaren Kontrollparameter abhängt. [Beispiel: x = Auslenkung, Y = rücktreibende Kraft, Hooke'sches Gesetz: $f(x) = cx$. Die Formeln werden aber allgemein sein]. Wenn wir nun ein x_0 wählen und dort messen, so bekommen wir ein Resultat y. Wegen Messfehlern wird, selbst wenn das Gesetz perfekt gilt, im allgemeinen $y \neq Y = f(x_0)$ sein.

In der Physik nimmt man oft an, dass die Fehler von Messungen normal oder Gauss-verteilt sind. Das bedeutet, dass, wenn der wahre Wert Y ist, ein Messwert ins Intervall $[y, y + dy]$ fällt mit der Wahrscheinlichkeit

$$P_{Y,\sigma}(y)dy = \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-\frac{(y-Y)^2}{2\sigma^2}} dy. \tag{9.2}$$

Das ist die berühmte Gauss-Glocke mit Maximum bei Y und Breite σ, die normiert ist,

$$\int_{-\infty}^{\infty} dy P_{Y,\sigma}(y) = 1, \tag{9.3}$$

denn irgendwo muss die Messung landen. Warum diese Verteilung? Wenn der Fehler durch Addition vieler statistisch unabhängig schwankender Einflüsse zustande kommt, dann führt bemerkenswerterweise ein mathematischer Satz (zentraler Grenzwertsatz, central limit theorem) zur Gauss-Verteilung. Ansonsten kann man die Verteilung testen, indem man die Messung oft wiederholt und die Verteilung (Histogramm) mit (9.2) vergleicht. Wir kommen darauf zurück.

Man sagt nun, dass y eine Messung für die exakte Größe Y mit einem Fehler σ darstellt. Dann kann man durch Berechnung von

$$p_n = p\{|y - Y| < n\sigma\} = \int_{Y-n\sigma}^{Y+n\sigma} dy P_{Y,\sigma}(y) \tag{9.4}$$

die bekannten Wahrscheinlichkeiten $p_1 \approx 0.68, p_2 \approx 0.95, p_3 \approx 0.99$ etc. berechnen.

Wir betrachten nun den Fall, dass wir N Messungen mit Ergebnissen y_i zu verschiedenen Parameterwerten x_i machen und nehmen zunächst an, dass wir *wissen*, dass der Fehler (Breite) jeweils σ_i ist. Dann ist die zugehörige gesamte quadratische Abweichung χ^2 definiert durch

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - Y_i)^2}{\sigma_i^2} \geq 0 \tag{9.5}$$

mit $Y_i = f(x_i)$. Abweichungen zählen also relativ zu den Skalen σ_i.

Eine später wichtige Frage wird sein: Welche Verteilung hat χ^2? Im Prinzip einfach: Wir betrachten χ^2 als Funktion eines N-Tupels von Messwerten, die *unabhängig* voneinander, d.h. mit der Produktwahrscheinlichkeit, gemäß (9.2) verteilt sind. Dann ist die Wahrscheinlichkeit, dass ein Messtupel $\chi^2 \geq C^2$ liefert, gegeben durch

$$Q(C^2) = p\{\chi^2 \geq C^2\} = \int d^N y P_{Y_1,\sigma_1}(y_1) \cdots P_{Y_N,\sigma_N}(y_N) \theta(\chi^2 - C^2) \tag{9.6}$$

mit Normierung $Q(0) = 1$. Wir substituieren $z_i = (y_i - Y_i)/\sigma_i$,

$$Q(C^2) = (2\pi)^{-N/2} \int d^N z \mathrm{e}^{-\frac{1}{2}z^2} \theta(z^2 - C^2), \quad z^2 \equiv \sum_{i=1}^{N} z_i^2. \tag{9.7}$$

Mit Polarkoordinaten in N Dimensionen erhält man

$$1 - Q(C^2) = |S_{N-1}|(2\pi)^{-N/2} \int_0^C dr r^{N-1} \mathrm{e}^{-\frac{1}{2}r^2}, \tag{9.8}$$

wobei $|S_{N-1}|$ das Resultat der Winkelintegrationen, also die Oberfläche der Einheitskugel in N Dimensionen ist. Wir kennen jedenfalls schon $|S_1| = 2\pi, |S_2| = 4\pi$. Wenn wir nun noch $t = r^2/2$ einführen, dann ergibt sich

$$1 - Q(C^2) = \frac{|S_{N-1}|}{2\pi^{N/2}} \int_0^{C^2/2} dt t^{N/2-1} \mathrm{e}^{-t} = \Gamma_{\mathrm{inc}}(C^2/2, N/2). \tag{9.9}$$

Die unvollständige Gamma Funktion Γ_{inc} ist definiert durch

$$\Gamma_{\mathrm{inc}}(A, b) = \frac{\int_0^A dt t^{b-1} \mathrm{e}^{-t}}{\int_0^\infty dt t^{b-1} \mathrm{e}^{-t}} \tag{9.10}$$

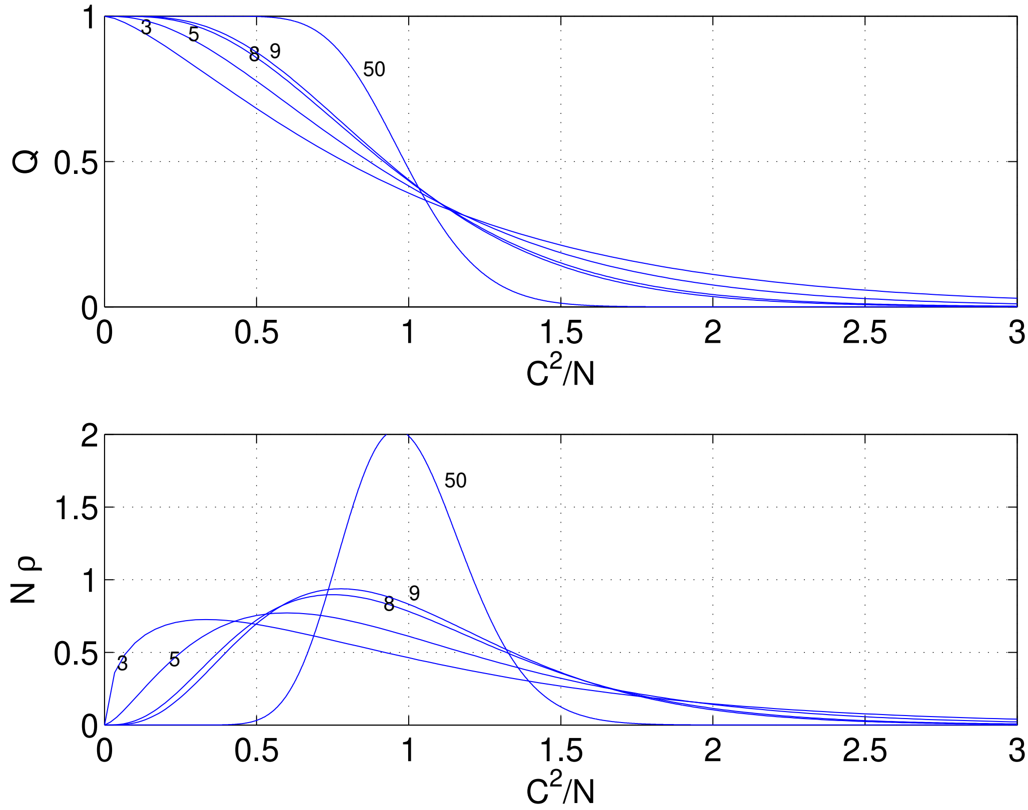

Abbildung 18: Wahrscheinlichkeit Q für $\chi^2 > C^2$ (oben) und Verteilung $N \times \rho(C^2)$ (unten) gegen C^2/N für mehrere N Werte.

mit der gewöhnlichen Gamma Funktion $\Gamma(b)$ im Nenner. Bei $C^2 = \infty$ gilt $Q = 0$ und so lernen wir nebenbei, dass gilt

$$|S_{N-1}| = \frac{2\pi^{N/2}}{\Gamma(N/2)}. \tag{9.11}$$

Mit $\Gamma(n+1) = n\Gamma(n), \Gamma(1) = 1, \Gamma(1/2) = \sqrt{\pi}$ kann man nun alle Werte bekommen. $\Gamma_{\text{inc}}(A, b)$ ist in `matlab` als `gammainc`(A,b) vorhanden.

Diese Funktion wurde benutzt um Fig. 18 oben zu produzieren. Man sieht, dass für viele Messungen die Wahrscheinlichkeit für $\chi^2 > N$ mit einer Art gerundeter Stufenfunktion verschwindet. Man bezeichnet Q auch als die Qualität der Übereinstimmung zwischen Gesetz (9.1) und den Daten. Für ein gefundenes χ^2 ist $Q(\chi^2)$ die Wahrscheinlichkeit, einen solchen oder einen noch größeren Wert zu finden, wenn das Gesetz stimmt. Falls das so ist, wird man selten auf Anhieb χ^2 mit sehr kleinen Q finden und schließt dann eher, dass das Gesetz nicht stimmt. In der Praxis akzeptiert man noch Werte bis $Q = 0.1$, bei einem sehr kleinen Wert wie z.B. 10^{-4} gilt der Ansatz eher als falsifiziert. Natürlich gibt es schwierige Grenzfälle, insbesondere bei wenigen Freiheitsgraden. Die Interpretation von Q ist etwas intuitiver als die von χ^2, obwohl es natürlich die gleiche Information ist.

Man kann auch die tatsächliche Verteilung von χ^2 betrachten. Sie ist gegeben durch

$$\rho(C^2) = -\frac{dQ}{dC^2} \tag{9.12}$$

denn die negative Ableitung verwandelt im Integranden von (9.6) $\theta(\chi^2 - C^2)$ in $\delta(\chi^2 - C^2)$. Auch dafür kann man leicht einen Plot erstellen, der im unteren Teil von Fig. 18 zu sehen ist. Für große N wird es immer wahrscheinlicher $\chi^2/N \approx 1$ zu finden und weder viel größer noch viel kleiner.

9.2 Fits

In der Praxis ist es nun oft so, dass man zwar die allgemeine Form von $f(x)$ weiß (zu wissen glaubt), diese aber einige freie Parameter enthält, $f \equiv f(x; c_1, \ldots, c_R)$, die man nicht kennt ($c$ bei Hooke). Dann geht es darum, genug Messungen y_i zu Werten x_i zu machen, um die $\{c_\alpha\}$ zu bestimmen. Idealerweise möchte man dabei auch noch wissen, ob der Ansatz für f "passt", oder ob dies eventuell für *keine Wahl* der $\{c_\alpha\}$ mit den Messdaten verträglich ist.

9.2.1 Minimales χ^2

Wir bilden

$$\chi^2 = \sum_{i=1}^{N} \frac{(y_i - f(x_i; c_\alpha))^2}{\sigma_i^2} \tag{9.13}$$

und betrachtet nun für *vorliegende Messungen mit Fehlern* $0 \leq \chi^2 \equiv \chi^2(c_1, \ldots, c_R)$ als Funktion der Fitparameter c_α. Es ist nun plausibel, diese als das (globale) Minimum dieser Funktion zu bestimmen. Damit wählt man aus der Schar von Theorien $f \equiv f(x; c_1, \ldots, c_R)$ diejenige aus, bei der der tatsächlich gefundene Datensatz $\{y_i\}$ die größte Wahrscheinlichkeit hat aufzutreten, denn es gilt ja, dass

$$\prod_{i=1}^{N} P_{f(x_i;c_\alpha),\sigma_i}(y_i) \propto \mathrm{e}^{-\chi^2/2} \tag{9.14}$$

maximal ist für minimales χ^2. Daher spricht man auch von einem Maximum Likelihood Fit.

Wie klein kann man χ^2 am Minimum erwarten? Zunächst braucht man generisch $N \geq R$ Datenpunkte um überhaupt alle c_α zu fixieren indem in χ^2 die N positiven Summanden sämtlich klein “gemacht” werden müssen. Bei $N > R$ kann $\chi^2 = 0$ normalerweise nicht erreicht werden. Weitere Theorie, die wir hier nicht im Detail darstellen, zeigt nun, dass auch bei dieser Prozedur – unter der Voraussetzung dass f für irgendeinen Parametersatz wirklich passt – die Wahrscheinlichkeiten verschiedene χ^2 zu finden den Kurven aus Fig. 18 folgt. Dabei ist aber das dortige N durch die Zahl der Freiheitsgrade

$$N_{\mathrm{dgf}} = N - R. \tag{9.15}$$

zu ersetzen. Stark vereinfacht kann man sagen, dass die ersten R Messungen in die Fixierung der Fitparameter gehen, und der Rest dann statistisch um die Werte des physikalischen Gesetzes herum schwanken. Findet man nun einen sehr unwahrscheinlichen Wert χ^2/N_{dgf} (Chi Quadrat pro Freiheitsgrad) viel größer als Eins, so hat der Ansatz nicht gepasst und muss verworfen werden (vgl. Fig. 18). Andernfalls wurden seine freien Parameter bestimmt.

In der Teilchenphysik sagt z.B. eine Theorie, dass ein neues Teilchen (z.B. das Higgs) eine Beule (bump) in einem Streuquerschnitt als Funktion der Energie macht. Die Form wird von der Theorie mit Masse und Lebensdauer als freie Parameter gegeben, die dann durch den Fit bestimmt werden. Die meisten abgeleiteten physikalischen Resultate kommen so zustande. Man erkennt sofort die wichtige Frage, mit welcher natürlich auch nur endlichen Genauigkeit die Fitparameter durch die fehlerbehafteten Daten fixiert werden, worauf wir noch zurückkommen.

9.2.2 Lineare Fits

Ein häufig auftretender vereinfachter Fall ist, wenn f linear in der Fitparametern ist

$$f(x; c_1, \ldots, c_R) = \sum_{\alpha=1}^{R} c_\alpha g_\alpha(x), \tag{9.16}$$

wobei es oft (aber nicht notwendig) um Polynome geht, $g_\alpha(x) = x^{\alpha-1}$. Dann führt die χ^2 Minimierung auf ein lineares Problem,

$$Ac = b \quad \Leftrightarrow \quad \sum_{\beta=1}^{R} a_{\alpha\beta} c_\beta = b_\alpha, \quad \alpha, \beta = 1, \ldots, R \tag{9.17}$$

mit

$$a_{\alpha\beta} = \sum_{i=1}^{N} \frac{g_\alpha(x_i) g_\beta(x_i)}{\sigma_i^2}, \quad b_\alpha = \sum_{i=1}^{N} \frac{g_\alpha(x_i) y_i}{\sigma_i^2} \tag{9.18}$$

Die Fehler der $c = A^{-1}b$ folgen nun aus denen von y_i durch lineare Fehlerfortpflanzung. Wenn die y_i 'wackeln', dann wackelt c_α gemäss

$$\delta c_\alpha = \sum_{i=1}^{N} d_{\alpha i} \delta y_i, \quad d_{\alpha i} = \sum_{\beta=1}^{r} a_{\alpha\beta}^{-1} \frac{g_\beta(x_i)}{\sigma_i^2} \tag{9.19}$$

Man betrachtet nun die c_α als Funktionen der $\{y_i\}$, integriert diese über ihre Verteilung wie in (9.6) und bestimmt die Breite der c_α. Das Resultat ist

$$\sigma_{c_\alpha} = \sqrt{M_{\alpha\alpha}}, \quad M_{\alpha\beta} = \sum_{i=1}^{N} \sigma_i^2 d_{\alpha i} d_{\beta i} \tag{9.20}$$

wobei die c_α wiederum normalverteilt sind. Es sei hier nur angemerkt, dass es neben Fehlern der c_α auch Korrelationen zwischen ihnen gibt. Die Breite σ_{c_α} sagt einem, wie weit der Parameter c_α variieren kann (mit den anderen *festgehalten*) bevor χ^2 stark hochgeht. In komplizierten Fällen kann es aber dann noch kollektive Änderungen mehrerer c_α *gemeinsam* geben, die größer sein können. Diese Information steckt in Eigenvektoren und Eigenwerten der gesamten Matrix $M_{\alpha\beta}$, was hier aber zu weit führen würde.

9.2.3 Fits mit einem nichtlinearen Parameter

Wenn alle Fitparameter nichtlinear eingehen, so wird die Minimierung von χ^2 ein u. U. schwieriges Optimierungsproblem. Dafür gibt es allgemeine Methoden, s. z. B. [2]. Die können sich aber relativ leicht in Nebenminima verirren und man muss aufpassen.

Ein häufiger Fall in der Physik ist z.B. $f(x) = c\mathrm{e}^{-\lambda x}$. Hier erscheint λ zunächst nichtlinear. Man kann aber das ganze Problem transformieren in

$\ln f = \ln c - \lambda x$ mit linearen Fitparametern $c_1 = \ln c, c_2 = -\lambda$. Allerdings müssen dann auch die Daten logarithmiert werden, also $\ln f$ wird an $\ln y_i$ gefittet. Um dieses χ^2 zu bilden, benötigen wir die Fehler $\sigma_i^{\ln}$ von $\ln y_i$, kennen aber zunächst mit σ_i nur diejenigen von y_i. Wenn die Fehler klein sind, dann kann man wegen $\delta(\ln y) = \delta y/y$ nehmen

$$\sigma_i^{\ln} = \frac{\sigma_i}{|y_i|}. \tag{9.21}$$

Etwas komplizierter ist z. B. der Fall $f(x) = a + c\mathrm{e}^{-\lambda x}$ mit noch einer Konstanten. Wir verbuchen das unter dem allgemeinen Fall, wo *ein* Fitparameter λ (von R) nichtlinear ist

$$f(x; c_1, \ldots, c_{R-1}; \lambda) = \sum_{\alpha=1}^{R-1} c_\alpha g_\alpha(x; \lambda) \tag{9.22}$$

In diesem Fall kann man für jeden festen Wert von λ bezüglich $c_1 \ldots, c_{R-1}$ das lineare Problem (9.17) lösen, bekommt also Werte $\bar{c}_\alpha(\lambda)$ für alle c_α. Nun kann man ein Programm für

$$F(\lambda) = \sum_{i=1}^{N} \frac{\left(y_i - \sum_{\alpha=1}^{R-1} c_\alpha g_\alpha(x_i; \lambda)\right)^2}{\sigma_i^2} \Bigg|_{c_\alpha = \bar{c}_\alpha(\lambda)} \tag{9.23}$$

schreiben und F plotten. Das Minimum kann nun visuell gefunden werden. Falls dort $F = \chi^2$ nicht viel grösser als N_{dgf} ist, dann ist der Fit plausibel.

Eine praktische Fehlerbestimmung (hier ohne Bergründung) für λ ist es, ein Intervall um das Minimum zu nehmen, wo an den Grenzen F jeweils um 1 gestiegen ist, $\Delta F = 1$. Bei dieser Vorgehensweise sieht man gut, wenn es mehrere (fast) entartete Minima gibt, was nicht so selten ist. Wissen zum Problem sagt einem oft, welches das 'richtige' ist, während voll nichtlineare Minimierungsprogramme u. U. in irgendeinem lokalen Minimum 'hängenbleiben' und dieses im hochdimensionalen Parameterraum nicht leicht zu diagnostizieren ist.

9.3 Praktische Erwägungen

Woher kennt man die Fehler σ_i der Messungen? Eine Möglichkeit ist, eine Messreihe durchzuführen, d.h. bei jedem $x = x_i$ wiederholt zu messen mit

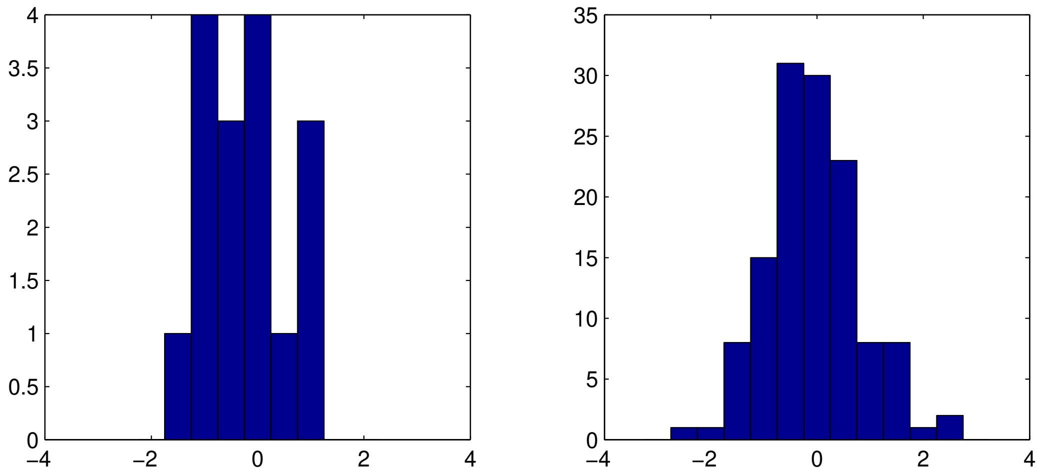

Abbildung 19: In Wahrheit Gauss verteilte Werte mit Mittelwert 0 und Breite 1 bei endlicher Statistik. Links 16 ‘Messungen’ in 7 Bins, rechts 128 Werte in 11 Bins.

Resultaten $y_{i,a}, a = 1, \ldots, n_i$. Wenn jede einzelne Messung gemäß (9.2) verteilt ist, dann zeigt man, dass aus der Streuung der Ergebnisse sich die Breite ihrer Verteilung ermitteln lässt zu

$$\sigma_i^2 = \frac{1}{n_i - 1} \sum_{a=1}^{n_i} (y_{i,a} - \overline{y}_i)^2, \quad \overline{y}_i = \frac{1}{n_i} \sum_{a=1}^{n_i} y_{i,a}. \tag{9.24}$$

Als besten Messwert nimmt man dann natürlich das Mittel $\overline{y}_i$. Dessen Breite bzw. Fehler $\overline{\sigma}_i$ ist kleiner als der der einzelnen $y_{i,a}$ um den bekannten Faktor $1/\sqrt{n_i}$,

$$\overline{\sigma}_i = \frac{\sigma_i}{\sqrt{n_i}}. \tag{9.25}$$

Wie schon erwähnt, können wir aus den $y_{i,a}$ ein Histogramm erstellen und so ein wenig testen, ob sie einigermaßen Gauss-verteilt aussehen. Allerdings sollten wir hier keine allzu perfekte Form bei kleinen Datensätzen erwarten, s. Fig. 19.

10 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 ggf. parallel gehörten Modul P3 oder einer ähnlichen Vorlesung Quantenmechanik I vorhanden sind. Dann müsste der harmonische Oszillator bekannt sein. Wir werden dennoch die für uns hier wesentlichen Fakten zunächst zusammenfassen.

10.1 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

$$H = -\frac{\hbar^2}{2m}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x). \tag{10.1}$$

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,

$$V(-x) = V(x). \tag{10.2}$$

Dies induziert einen Spiegelungs- oder Paritätsoperator P auf den Wellenfunktionen,

$$(P\psi)(x) = \psi(-x). \tag{10.3}$$

Er vertauscht nach Voraussetzung mit H, und kann daher 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

$$V(x) = \frac{k}{\alpha!}|x|^\alpha, \;\; k, \alpha > 0 \tag{10.4}$$

betrachten[14] , 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.

10.2 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

$$H = \frac{1}{2}\left(-\frac{\mathrm{d}^2}{\mathrm{d}x^2} + x^2\right). \tag{10.5}$$

Mit Hilfe der Erzeuger und Vernichter

$$\begin{aligned} a &= \frac{1}{\sqrt{2}}\left(\frac{\mathrm{d}}{\mathrm{d}x} + x\right) \\ a^\dagger &= \frac{1}{\sqrt{2}}\left(-\frac{\mathrm{d}}{\mathrm{d}x} + x\right), \end{aligned} \tag{10.6}$$

die die Vertauschungsrelation

$$aa^\dagger - a^\dagger a = \left[a, a^\dagger\right] = 1 \tag{10.7}$$

erfüllen, gilt dann

$$H = \frac{1}{2} + a^\dagger a. \tag{10.8}$$

[14]Für nicht ganzzahlige $\alpha > 0$ gilt $\alpha! = \Gamma(\alpha + 1)$, `gamma(.)` ist in MATLAB verfügbar.

Es gibt Eigenzustände

$$H\psi_n = E_n\psi_n \tag{10.9}$$

mit

$$E_n = \frac{1}{2} + n. \tag{10.10}$$

Der normierbare Grundzustand wird von a vernichtet,

$$a\psi_0 = 0 \to \psi_0 \propto \exp(-x^2/2), \tag{10.11}$$

und die höheren Zustände ergeben sich durch Anwenden von $a^\dagger$,

$$\psi_n \propto \left\{a^\dagger\right\}^n \psi_0. \tag{10.12}$$

Dies und die Eigenwerte (10.10) lassen sich induktiv mit Hilfe von (10.7) zeigen. Da der Grundzustand symmetrisch unter P ist und $Pa^\dagger = -a^\dagger P$ gilt, hat der n-te Zustand die Parität

$$P\psi_n = (-1)^n\psi_n. \tag{10.13}$$

10.3 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

$$(H - E)\psi = 0 \tag{10.14}$$

soll betrachtet werden mit

$$H = -\frac{1}{2}\frac{\mathrm{d}^2}{\mathrm{d}x^2} + \frac{1}{\alpha!}|x|^\alpha \tag{10.15}$$

in geeigneten Einheiten, um herauszufinden, für welche E sie Lösungen hat. Es muss also gelten

$$\psi''(x) = 2(V(x) - E)\psi, \tag{10.16}$$

und — erst dies legt E fest! — die Randbedingungen müssen erfüllt sein. Wir suchen also Lösungen mit

$$\psi(\pm R) \stackrel{!}{=} 0. \tag{10.17}$$

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 vorgegeben statt Anfangs- und Endort. Man kann nun aber (im Prinzip) folgendermaßen vorgehen:

- wähle einen Wert E
- 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 (10.16) *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. [7]. 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 (10.16) vernachlässigbar, und es gibt je eine exponentiell wachsende und eine fallende Lösung[15]. 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. Man kann es sich auch so vorstellen: Auf einem hypothetischen unendlich genauen Rechner gibt es natürlich ein E dass zu $\psi(R) = 0$ führt. Wegen des exponentiellen Wachsens gehören winzige Änderungen von $E + \delta E$ zu Werten $\psi(R)$, die stattdessen von der Ordnung Eins sind. Wenn dann $\delta E/E < 10^{-16}$ ist, hat man auf realen Rechnern das beschriebene Problem. Dies wird hier mit Hilfe der Paritätssymmetrie umgangen. Man erledigt sozusagen den zweiten problematischen Teil der Integration mit einem Symmetrieargument. Überlegen Sie sich, wie man auch ohne Spiegelsymmetrie Integrationen in die instabilen Richtung vermeiden kann.

10.4 Hinweise zur MATLAB -Implementierung

Aus den bisherigen Kapiteln 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 sei nochmal daran erinnert, wie **ode23 und ode45** (vgl. Kap. 5.10) den Fehler der Lösung kontrolliert. Es wird sowohl eine relative als auch eine absolute Fehlerschranke angegeben (oder Voreinstellungen verwendet!). Normalerweise wird der relative Fehler eingehalten. Wenn allerdings Komponenten der (wahren) Lösung durch Null gehen (z.B. Knoten), dann ist dies nicht sinnvoll bzw. möglich, und die absolute Schranke wird massgeblich.

[15] Wir argumentieren hier wieder für $R = \infty$. Unser R ist per Voraussetzung so groß, dass das geschilderte Verhalten einsetzt, bevor man R erreicht hat.

Diese hängt jedoch von der Skala der Lösung ab, z. B. wie gross sie maximal wird. Damit das alles korrekt ist, sollten natürliche Einheiten gewählt sein, in denen die Skala Ordnung Eins ist. In unserem Beispiel sind das Maximalwerte von $|\psi|, |\psi'|$. Das wird mit $\psi'(-R) = 1$ nicht erfüllt sein, da dies im exponentiell kleinen nichtklassischen Bereich ist.

Die Eigenwertgleichung für große $|x|$ ist

$$\psi'' \approx \frac{2}{\alpha!}|x|^\alpha \psi \tag{10.18}$$

in unserem Problem mit der näherungsweisen ($|x| \gg 1$) (abfallenden!) Lösung

$$\psi \propto \exp\left\{ -\frac{2}{\alpha+2}\sqrt{\frac{2}{\alpha!}}\,|x|^{\frac{\alpha}{2}+1} \right\}. \tag{10.19}$$

Daraus kann man eine natürlichere Größenordnung für $\psi'(-R) \neq 0$ beziehen. Sollte man dann immer noch mit extremen Werten in der Umgebung $x = 0$ ankommen, so kann man diese nehmen, um $\psi'(-R)$ besser zu wählen, und nochmals integrieren. Erst wenn man auf diese Art die Lösung überall auf O(1) skaliert hat (Ordnung: 10 ist auch okay, aber nicht z.B. $10^{\pm 8}$), dann ist die Fehlerkontrolle zuverlässig.

Die Eigenwerte werden nun als Nullstellen einer Funktion, $F(E_n) = 0$, gefunden. Der erste Versuch wird darin bestehen, verschiedene E zu probieren und einen Vorzeichenwechsel einzukreisen, z. B. indem man ein Intervall nimmt und dieses fortgesetzt halbiert, indem man noch das Vorzeichen in der Mitte bestimmt (Bisektion).

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 Kap. 4 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 als 'function handle' übergibt, dann kann man **fzero** benutzen, um E_n genau zu lokalisieren. Welche Lösung gefunden wird, hängt vom starting guess (Anfangsschätzung) ab.

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 zeitunabhängigen Schrödinger Gleichung anwendbar. Die Klassifizierung der PDEs sei anhand des Falles mit zwei unabhängigen Variablen x, y angegeben. Die allgemeine Gleichung

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

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 [8, 9]

$$\vec{\nabla} \cdot \vec{E} = \frac{\rho}{\varepsilon_0}, \tag{11.2}$$

$$\vec{\nabla} \times \vec{E} = 0 \tag{11.3}$$

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

$$\vec{E} = -\vec{\nabla}\Phi \tag{11.4}$$

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

$$-\triangle\Phi = \frac{\rho}{\varepsilon_0} \tag{11.5}$$

bzw. die Laplace–Gleichung

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

im leeren Raum. Nur Ladungen produzieren letztlich ein nichttriviales Potential. Oft betrachtet man aber nur ein begrenztes ladungsfreies Gebiet und berücksichtigt Ladungen im Außenraum in Form von Randbedingungen an das Potential. In diesem Kurs wird es hauptsächlich darum gehen, die Laplace–Gleichung mit solchen Randbedingungen zu lösen.

11.2 Elektrostatische Energie

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

$$U = \frac{\varepsilon_0}{2}\int \mathrm{d}V \vec{E}^2 = \frac{\varepsilon_0}{2}\int \mathrm{d}V (\vec{\nabla}\Phi)^2 \tag{11.7}$$

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[16], und kann nicht variiert werden. Dann gilt

$$0 \stackrel{!}{=} \delta U = \int_V \mathrm{d}V (\vec{\nabla}\Phi)\cdot\vec{\nabla}\delta\Phi = -\int_V \mathrm{d}V \delta\Phi\triangle\Phi \tag{11.8}$$

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. (11.7) ist ein positives quadratisches Funktional in Φ und man kann zeigen, dass das durch die Laplace–Gleichung gegebene Extremum ein Minimum ist. Dieser Zugang zur Laplace–Gleichung über ein Extremalprinzip ist günstig für die Diskretisierung und spätere numerische Behandlung.

11.3 Diskretisierung der Laplace–Gleichung

Kontinuierliche Felder wie Φ und die Komponenten von $\vec{E}$ entsprechen selbst über einem endlichen Volumen unendlich vielen Parametern. Dies sieht man

[16]Hier werden Dirichlet Randbedingungen betrachtet.

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 und Auflösungen haben, so daß es immer genügt, Felder für endlich viele genügend dicht gelegene Punkte zu kennen. Das ist analog zu gewöhnlichen Differentialgleichungen, wo wir auch die Zeit diskretisiert haben.

Im Folgenden diskutieren wir zwei statt der physikalischen drei Raumdimensionen. Man kann dies so interpretieren, dass Potential und Feld von der z-Koordinate nicht abhängen, in dieser Richtung also nicht variieren. Größen wie die Gesamtenergie sind dann eigentlich als Dichte pro Länge in z zu interpretieren. Alternativ kann man sagen, wir betrachten eine hypothetische 2-D Welt um Methoden zu entwickeln.

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

$$\Phi_{i,j} = \Phi(\vec{r}_{i,j}),\ \vec{r}_{i,j} = ([i-1]h_x, [j-1]h_y) \tag{11.9}$$

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 wesentlich variieren.

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

$$\begin{aligned} \partial_x\Phi &\simeq \frac{\Phi_{i+1,j} - \Phi_{i,j}}{h_x}, \\ \partial_y\Phi &\simeq \frac{\Phi_{i,j+1} - \Phi_{i,j}}{h_y}. \end{aligned} \tag{11.10}$$

Damit folgt für die Energie

$$U = \frac{\varepsilon_0}{2} h_x h_y \sum_{i,j} \left\{ \left(\frac{\Phi_{i+1,j} - \Phi_{i,j}}{h_x} \right)^2 + \left(\frac{\Phi_{i,j+1} - \Phi_{i,j}}{h_y} \right)^2 \right\}, \tag{11.11}$$

wobei wir das Integral durch eine Riemann Summe genähert haben[17]. Der Summationsbereich muß passend zur Geometrie und den Rändern gewählt

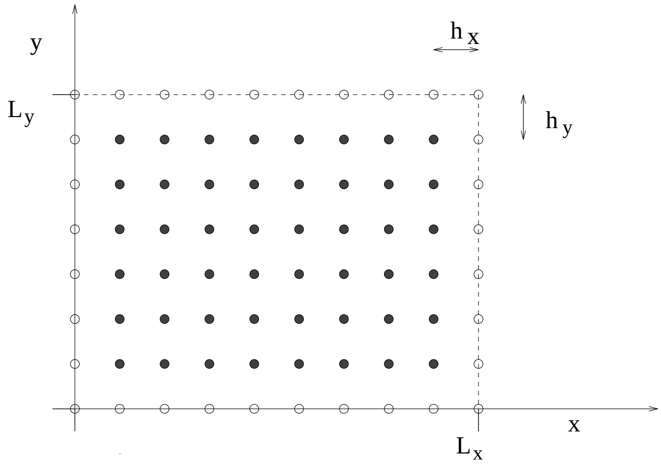

Abbildung 20: Beispiel für Gitterdiskretisierung

werden. Als Beispiel zeigt Abb.20 den schon diskutierten rechteckigen Bereich der Größe $L_x \times L_y$. Die *inneren* Gitterpunkte sind dann aufzuzählen mit

$$i = 2, \ldots, N_x - 1,\ j = 2, \ldots, N_y - 1,\ L_x = (N_x - 1)h_x,\ L_y = (N_y - 1)h_y, \tag{11.12}$$

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 (11.11) 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),

$$\begin{aligned} 0 &= \frac{1}{\varepsilon_0 h_x h_y} \frac{\partial U}{\partial \Phi_{i,j}} = \qquad (11.13) \\ &= \frac{1}{h_x^2}(2\Phi_{i,j} - \Phi_{i+1,j} - \Phi_{i-1,j}) + \frac{1}{h_y^2}(2\Phi_{i,j} - \Phi_{i,j+1} - \Phi_{i,j-1}) =: (-\triangle\Phi)_{i,j}. \end{aligned}$$

[17]Vgl. Definition Integral als Limes solcher Summen.

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

$$A\Phi = \Phi_R, \tag{11.14}$$

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. dass in jeder Zeile und Spalte nur wenige von Null verschiedenene Matrixelemente sind. Das liegt daran, daß in (11.13) jedes Φ nur mit den 4 Nachbarn verknüpft wird.

11.4 Separationslösung

Ein einfaches Problem – zunächst wieder im Kontinuum – 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

$$\Phi(x,y) = X(x)Y(y) \tag{11.15}$$

kann hier zur Lösung benutzt werden. Wir betrachten auch hier wieder der Einfachheit halber ein ebenes Problem, daher kein Faktor $Z(z)$. Damit gibt (11.6)

$$\frac{1}{X}\frac{\mathrm{d}^2 X}{\mathrm{d}x^2} = -\frac{1}{Y}\frac{\mathrm{d}^2 Y}{\mathrm{d}y^2}. \tag{11.16}$$

Das Standardargument besagt, daß eine Funktion von x (linke Seite) nur identisch mit einer Funktion von y (rechte 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

$$\begin{aligned} X(x) &= C_s \sin(kx) + C_c \cos(kx) \\ Y(y) &= C_s' \sinh(ky) + C_c' \cosh(ky). \end{aligned} \tag{11.17}$$

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

$$\Phi = \sum_{n=1}^{\infty} \sin(k_n x) \left[c_n \sinh(k_n y) + d_n \cosh(k_n y)\right] \tag{11.18}$$

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

11.5 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 einen 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 (11.11) 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 (11.13) *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 im Inneren, wobei aber eine Schätzung des Resultats, soweit erhältlich, zu schnellerer Konvergenz führt und eingesetzt werden sollte.

11.6 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 relativ leicht analytisch beweisen. 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. (11.13). 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. Man kann beweisen (siehe z. B. Festkörperphysik) dass diese Transformation umkehrbar ist (gleichviele reelle Parameter!), so daß dies immer möglich ist. Der Nutzen hier beruht darauf, daß der Gitter Laplace–Operator in dieser Darstellung diagonal ist[18],

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

[18]Eine detaillierte Diskussion der Fourier Entwicklung erfolgt in CP2.

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

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

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.

11.7 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 geeignet 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 [2], daß

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

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

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

Für kompliziertere Geometrien muß ω_{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 beim Lösen der Laplace–Gleichung nicht immer monoton fällt.

11.8 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[19]. 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 [10].

11.9 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 ∼`uwolff/CP1/kap11` und kann von dort als Basis für die Übungen heruntergeladen werden. Hier ein Listing:

```
% file jacob.m; L"osung der Laplace--Gleichung.
% quadratisches Gitter h_x=h_y=h, L_x=L_y=L
clear; help jacob;
N=20; %N = input(' N(=N_x=N_y)? - ');
L = 1;          % System Groesse
h = L/(N-1);    % Gitterabstand
phi0 = 1;       % Phi_0
tol = 1e-4;     % Abbruchkriterium mittlere Aenderung in Phi
%%%%% Anfangswerte, Flag Feld etc.
x = (0:N-1)*h;
y = (0:N-1)*h;
%
flag = zeros(N,N); % einhuellende Matrix mit Nullen
flag(2:N-1,2:N-1) = ones(N-2,N-2); % innen Einsen
```

[19] 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.

```
%
phi = zeros(N,N); % = phineu
%
phi(:,N) = phi0*min(x/L,1-x/L).'; % Randbed. y=L, manifest symm. x->L-x
%
for j = 2:N-1,
  phi(:,j) = phi(:,N)*(j-1)/(N-1);
end % linear interpoliert zum Start
%
max_iter = 2*N^2;  % Grenze zur Sicherheit
time=cputime;
for iter=1:max_iter
  phialt = phi; % fuer Jacobi-Verfahren
  temp = 0;

  for i = 1:N, for j = 1:N, if flag(i,j) % loop Gitterinneres
%    Jabobi:
    phi(i,j) = .25*(phialt(i+1,j  )+phialt(i-1,j  )+ ...
                    phialt(i  ,j-1)+phialt(i  ,j+1));
    temp = temp + abs(phi(i,j)-phialt(i,j));
  end, end, end  % Ende loop Gitterinneres
%
  phialt = phi;
  change(iter) = temp/(phi0*(N-2)^2); % mittlere Aenderung
%  fprintf('Nach %g Iterationen, Aenderung = %g\n',iter,change(iter));
  if( change(iter) < tol )
    disp(' konvergiert => Abbruch ');
    break;
  end
end % Ende loop iter
time=cputime-time;
fprintf('Iterationszeit = %g\n',time);
%
subplot(2,2,1)
mesh(x,y,phi.'); % 3d-Plot der Funktion phi(x,y),
%                  phi transponiert, sonst x<->y vertauscht!
xlabel('x/L'); ylabel('y/L'); zlabel('Phi/Phi0');
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 = max(abs(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

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

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.21 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

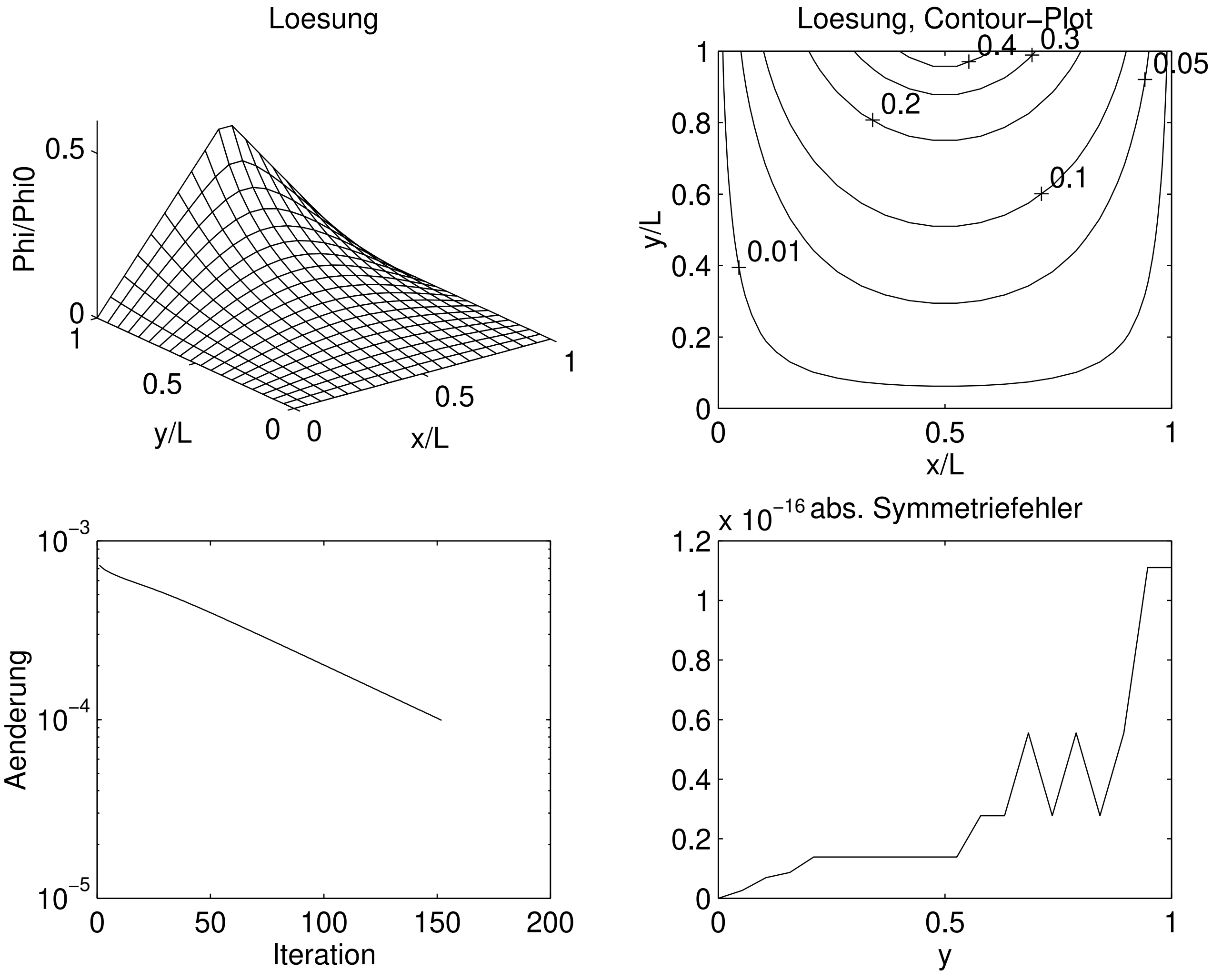

Abbildung 21: Plot Output von `jacob.m`

Index x entspricht, transponiert werden muß[20].

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 Betragsmaxima, d.h. also für jedes y. Wie in Abb.21 zu sehen ist, ist die Symmetrie offenbar in Maschinengenauigkeit erfüllt. Dies gilt für Jacobi, da sie bei jeder einzelnen Iteration erhalten bleibt (bitte nachprüfen), nicht jedoch bei GS und SOR.

In der Abbildung 22 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 Iterationszeit 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 (11.14) 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.

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

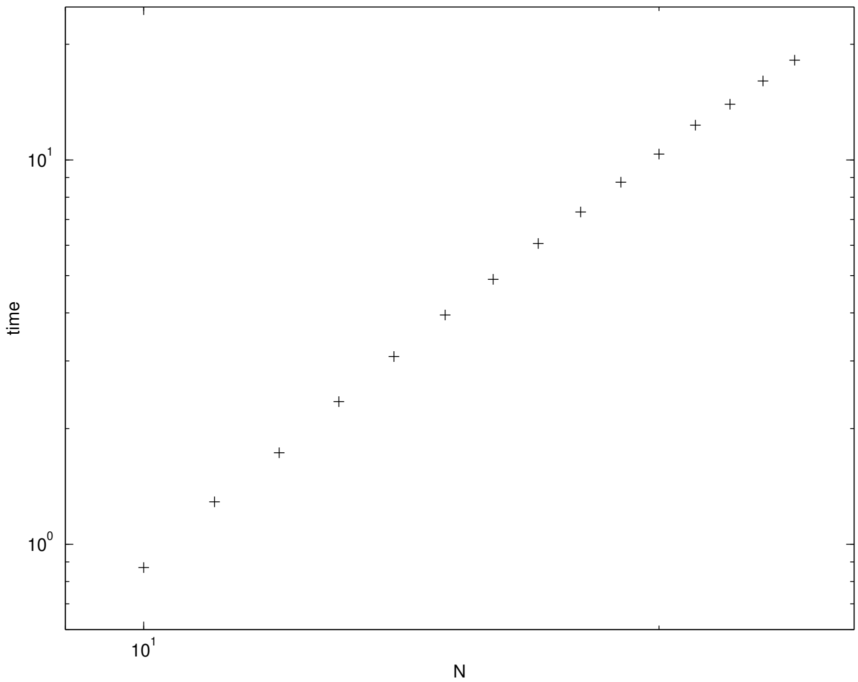

Abbildung 22: Iterationszeit beim Jacobi-Verfahren als Funktion von N in doppelt-logarithmischer Auftragung.

Literatur

[1] W. Cheney und D. Kincaid, *Numerical Mathematics and Computing*, Brooks/Cole Publishing Company, Pacific Grove, California

[2] 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

[3] W. Gander und Jiri Hrebicek, *Solving Problems in Scientific Computing Using Maple and MATLAB*, Springer Verlag

[4] N. J. Giordano, *Computational Physics*, Prentice Hall

[5] T. Lippert, A. Seyfried, A. Bode and K. Schilling, "Hypersystolic parallel computing," `http://de.arxiv.org/abs/hep-lat/9507021`

[6] M. Abramowitz und I. A. Stegun,
Handbook of Mathematical Functions, Dover Publications, New York

[7] W. Nolting, *Grundkurs Theoretische Physik, Quantenmechanik, Band 5, Teil 1*, Springer, Berlin

[8] J. D. Jackson, *Klassische Elektrodynamik*, (de Gruyter, Berlin)

[9] Feynman Lectures on Physics, Vol. 2

[10] P.L.DeVries, A first course in COMPUTATIONAL PHYSICS, Jogn Wiley and Sons, INC. (New York)