EINFÜHRUNG IN MATLAB[1]

Peter Arbenz
Computer Science Department
ETH Zürich
mailto:arbenz@inf.ethz.ch

Januar 2007 / September 2008

[1] http://people.inf.ethz.ch/arbenz/MatlabKurs/

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Was ist MATLAB [14] . . . 1
- 1.2 Geschichtliches [8] . . . 1

2 Grundlegendes **4**
- 2.1 Das MATLAB-Fenster . . . 4
- 2.2 Hilfe aus dem Befehlsfenster . . . 5
- 2.3 Matrizen als grundlegender Datentyp . . . 6
- 2.4 Zuweisungen . . . 7
- 2.5 Namen . . . 7
- 2.6 Eingabe von Matrizen . . . 8
  - 2.6.1 Matrixaufbau durch explizite Eingabe der Matrixelemente . . . 8
  - 2.6.2 Matrixaufbau durch Aufruf einer matrixbildenden Funktion . . . 10
  - 2.6.3 Aufbau einer Matrix durch ein eigenes M-File . . . 11
  - 2.6.4 Matrixaufbau durch Laden aus einem externen File . . . 12
  - 2.6.5 Aufgaben . . . 12
- 2.7 Operationen mit Matrizen . . . 13
  - 2.7.1 Addition und Subtraktion von Matrizen . . . 13
  - 2.7.2 Vektorprodukte and Transponierte . . . 14
  - 2.7.3 Zugriff auf Matrixelemente . . . 16
  - 2.7.4 Der Doppelpunkt-Operator . . . 16
  - 2.7.5 Arithmetische Operationen . . . 17
  - 2.7.6 Vergleichsoperationen . . . 18
  - 2.7.7 Logische Operationen . . . 19
  - 2.7.8 Aufgaben . . . 19
- 2.8 Ausgabeformate . . . 19
- 2.9 Komplexe Zahlen . . . 20
- 2.10 Speichern von Daten zur späteren Weiterverwendung . . . 20
- 2.11 Strukturen (Structs) . . . 21
- 2.12 Zeichenketten (Strings) . . . 22
- 2.13 Cell arrays . . . 23

3 Funktionen **24**
- 3.1 Skalare Funktionen . . . 24
- 3.2 Spezielle Konstanten . . . 24
- 3.3 Vektorfunktionen . . . 26
  - 3.3.1 Aufgabe . . . 28
- 3.4 Matrixfunktionen . . . 28
  - 3.4.1 Lineare Gleichungssysteme . . . 28
  - 3.4.2 Der Backslash-Operator in MATLAB . . . 31
  - 3.4.3 Aufgaben . . . 32
  - 3.4.4 Die Methode der kleinsten Quadrate (least squares) . . . 34
  - 3.4.5 Aufgabe (Regressionsgerade) . . . 34
  - 3.4.6 Eigenwertzerlegung . . . 35

3.4.7 Anwendung: Matrixfunktionen . . . 36
3.4.8 Anwendung: Systeme von lin. gew. Differentialgleichungen 1. Ordnung . . . 37
3.4.9 Singulärwertzerlegung (SVD) . . . 38
3.4.10 Anwendung der SVD: Bild, Kern und Konditionszahl einer Matrix . . . 39

4 Konstruktionen zur Programmsteuerung 41
4.1 Das `if`-Statement . . . 41
4.2 Die `switch-case`-Konstruktion . . . 42
4.3 Die `for`-Schleife . . . 43
4.4 Die `while`-Schleife . . . 44
4.5 Eine Bemerkung zur Effizienz . . . 45
4.6 Aufgaben . . . 45

5 M-Files 47
5.1 Scriptfiles . . . 47
5.2 Funktionen-Files . . . 48
5.2.1 Aufgabe . . . 50
5.3 Arten von Funktionen . . . 50
5.3.1 Anonyme Funktionen . . . 50
5.3.2 Primäre und Subfunktionen . . . 51
5.3.3 Globale Variable . . . 51
5.3.4 Funktionenfunktionen . . . 52
5.3.5 Funktionen mit variabler Zahl von Argumenten . . . 53
5.3.6 Aufgaben . . . 54

6 Der Matlab-Editor/Debugger 57

7 Graphik in Matlab 60
7.1 Darstellung von Linien . . . 60
7.1.1 Aufgaben . . . 61
7.2 Das Matlab-Graphiksystem . . . 62
7.2.1 Aufgaben . . . 64
7.3 Mehrere Plots in einer Figur . . . 64
7.4 Plots mit zwei Skalen . . . 65
7.5 Darstellung von Flächen . . . 66
7.5.1 Aufgabe . . . 69
7.6 Darstellung von Daten . . . 69
7.6.1 Aufgaben . . . 70

8 Anwendungen aus der Numerik 71
8.1 Kreisausgleichsproblem . . . 71
8.2 Singulärwertzerlegung . . . 74
8.3 Gewöhnliche Differentialgleichungen . . . 78
8.3.1 The child and the toy . . . 78
8.3.2 The jogger and the dog . . . 80
8.3.3 Showing motion with Matlab . . . 82
8.4 Fitting Lines, Rectangles and Squares in the Plane . . . 83
8.4.1 Fitting two Parallel Lines . . . 85
8.4.2 Fitting Orthogonal Lines . . . 86
8.4.3 Fitting a Rectangle . . . 87
8.5 Beispiel Prototyping:
Update der QR-Zerlegung . . . 89

9 Einführungen in Matlab, Resourcen auf dem Internet **93**
9.1 Tutorials . . . 93
9.2 Software . . . 94
9.3 Alternativen zu MATLAB . . . 94

Kapitel 1

Einleitung

1.1 Was ist Matlab [14]

MATLAB ist eine Hoch-Leistungs-Sprache für technisches Rechnen (Eigenwerbung). MATLAB integriert Berechnung, Visualisierung und Programmierung in einer leicht zu benützenden Umgebung (graphisches Benützer-Oberflächen, GUI). Probleme und Lösungen werden in bekannter mathematischer Notation ein- und ausgegeben. Typische Verwendungen von MATLAB sind:

- Technisch-wissenschaftliches Rechnen.
- Entwicklung von Algorithmen.
- Datenaquisition.
- Modellierung, Simulation und Prototyping.
- Datenanalyse und Visualisierung.
- Graphische Darstellung von Daten aus Wissenschaft und Ingenieurwesen.
- Entwicklung von Anwendungen, einschliesslich graphischer Benützer-Oberflächen.

MATLAB ist ein interaktives System dessen grundlegender Datentyp das Array (oder Matrix) ist, das nicht dimensioniert werden muss. Es erlaubt viele technische Probleme (vor allem jene, die in Matrix- / Vektornotation beschrieben sind) in einem Bruchteil der Zeit zu lösen, die es brauchen würde, um dafür ein Program in C oder FORTRAN zu schreiben.

MATLAB steht für MATrizen-LABoratorium. MATLAB wurde ursprünglich als interaktives Programm (in FORTRAN) geschrieben, um bequemen Zugriff auf die bekannte Software für Matrixberechnungen aus den LINPACK- and EISPACK-Projekten zu haben. Heutzutage umfasste die MATLAB-Maschine die LAPACK und BLAS-Bibliotheken, welche den *state of the art* der Matrixberechungen sind.

MATLAB hat sich aber sehr stark entwickelt. Es ist nicht mehr nur auf die Basis-Algorithmen der numerischen linearen Algebra beschränkt. Mit sogenannte *Toolboxen* kann MATLAB durch anwendungsspezifischen Lösungsverfahren erweitert werden. Toolboxen sind Sammlungen von MATLAB-Funktionen (M-Files). Gebiete für die es Toolboxen gibt sind z.B. Signalverarbeitung, Regelungstechnik, Neuronale Netzwerke, 'fuzzy logic', Wavelets, Simulation und viele andere.

1.2 Geschichtliches [8]

Die lineare Algebra, insbesondere Matrix-Algebra, ist beim wissenschaftlichen Rechnen von grosser Bedeutung, weil die Lösung vieler Probleme sich aus Grundaufgaben aus diesem Gebiet zusammensetzt. Diese sind im wesentlichen Matrixoperationen, Lösen von linearen Gleichungssystemen und Eigenwertprobleme.

Diese Tatsache wurde früh erkannt und es wurde deshalb schon in den 60-er Jahren an einer Programmbibliothek für lineare Algebra gearbeitet. Damals existierten für wissenschaftliches Rechnen nur die beiden Programmiersprachen ALGOL 60 und FORTRAN. Eine Reihe "Handbook for Automatic Computation" wurde im Springer-Verlag begonnen mit dem Ziel, eines Tages eine vollständige Bibliothek von Computerprogrammen zu enthalten. Man einigte sich als Dokumentationssprache auf ALGOL, denn

> indeed, a correct ALGOL program is the *abstractum* of a computing process for which the necessary analyses have already been performed. [1]

Band 1 des Handbuches besteht aus zwei Teilen: in Teil A beschreibt H. Rutishauser die Referenzsprache unter dem Titel "Description of ALGOL 60" [17], in Teil B "Translation of ALGOL 60" geben die drei Autoren Grau, Hill und Langmaack [9] eine Anleitung zum Bau eines Compilers.

Der zweite Band des Handbuches, redigiert von Wilkinson und Reinsch, erschien 1971. Er enthält unter dem Titel "Linear Algebra" [21] verschiedene Prozeduren zur Lösung von linearen Gleichungssystemen und Eigenwertproblemen.

Leider wurde die Handbuchreihe nicht mehr fortgesetzt, weil die stürmische Entwicklung und Ausbreitung der Informatik eine weitere Koordination verunmöglichte.

Wegen der Sprachentrennung Europa – USA:

> The code itself has to be in FORTRAN, which is the language for scientific programming in the United States.[2]

wurde Ende der siebziger Jahren am Argonne National Laboratory das LINPACK Projekt durchgeführt. LINPACK enthält Programme zur Lösung von vollbesetzten linearen Gleichungssystemen. Sie stützen sich auf die Prozeduren des Handbuchs ab, sind jedoch neu und systematisch in FORTRAN programmiert. Dies äussert sich in einheitlichen Konventionen für Namengebung, Portabilität und Maschinenunabhängigkeit (z.B. Abbruchkriterien), Verwendung von elementaren Operationen mittels Aufruf der BLAS (Basic linear Algebra Subprograms). Der LINPACK Users' Guide erschien 1979 [2]. LINPACK steht auch für den Namen Name eines Benchmarks zur Leistungsmessung eines Rechners im Bereich der Fliesskommaoperationen. Früher bestand dieser Benchmark aus zwei Teilen: Einerseits musste ein vorgegebenes FORTRAN Programm zur Lösung eines voll besetzten 100×100 linearen Gleichungssystem kompiliert und ausgeführt werden, andererseits musste ein 1000×1000 Gleichungssystem möglichst schnell (mit beliebig angepasstem Programm) gelöst werden. Dieser Benchmark wird heute in veränderter Form zur Bestimmung der 500 leistungsfähigsten Computer auf der Welt benützt, die in die halbjährlich nachgeführte `top500`-Liste aufgenommen werden, siehe `http://www.top500.org`.

Auch die Eigenwertprozeduren aus [21] wurden in FORTRAN übersetzt und sind unter dem Namen EISPACK erhältlich [20, 6]. EISPACK und LINPACK sind vor einigen Jahren von LAPACK [1] abgelöst worden. Elektronisch kann man LINPACK-, EISPACK- und LAPACK-Prozeduren (und viele mehr) von der on-line Software-Bibliothek NETLIB [22] erhalten, siehe `http://www.netlib.org`.

Ende der siebziger Jahre entwickelte Cleve Moler das interaktive Programm MATLAB (MATrix LABoratory), zunächst nur mit der Absicht, es als bequehmes Rechenhilfsmittel in Vorlesungen und Übungen einzusetzen. Grundlage dafür waren Programme aus LINPACK und EISPACK. Weil Effizienzüberlegungen nicht im Vordergrund standen, wurden nur acht Prozeduren aus LINPACK und fünf aus EISPACK für Berechnungen mit vollen Matrizen verwendet. MATLAB hat sich nicht nur im Unterricht als sehr gutes Hilfsmittel etabliert, sondern wird entgegen der ursprünglichen Absicht heute auch in Industrie und Forschung eingesetzt. Das ursprüngliche in Fortran geschriebene public domain MATLAB [13] wurde von der Firma MathWorks vollständig neu überarbeitet, erweitert und zu einem effizienten Ingenieurwerkzeug gestaltet [14]. Es ist jetzt in C geschrieben.

Diese Philosophie beim Entwickeln von MATLAB hat dazu geführt, dass laufend neue Funktionen-Pakete (sog. toolboxes) für verschiedene Anwendungsgebiete geschrieben werden. Diese Pakete sind zu einem kleinen Teil öffentlich (erhältlich via netlib) zum grössten Teil werden sie aber von The MathWorks selber bietet verschiedene sog. toolboxes an. Die neueste Information findet man immer

[1] Rutishauser in [17]

[2] aus dem Vorwort des LINPACK users guide [9]

auf der WWW-Homepage von The MathWorks [23]. Natürlich kann auch jeder Benützer MATLAB durch eigene Funktionen nach seinen Anforderungen erweitern.

Vergleiche von MATLAB mit anderen ähnlichen Systemen findet man z.B. bei Higham [10] oder bei Simon und Wilson [19].

Alternativen zu MATLAB im *public domain* sind Scilab (`www.scilab.org`), welches sich recht nahe an MATLAB anlehnt. Ebenfalls gibt es Octave (`www.octave.org`), welches aber anscheinend nicht weiterentwickelt wird. Im Statistik-Bereich gibt es die Umgebung R, siehe `http://www.r-project.org/`.

Kapitel 2

Grundlegendes

2.1 Das Matlab-Fenster

Nach dem Start von MATLAB ”durch Anklicken eines Icons oder durch Eintippen des Befehls `matlab` in einem Konsolenfenster (shell) wird ein Fenster wie in Abbildung 2.1 geöffnet[1].

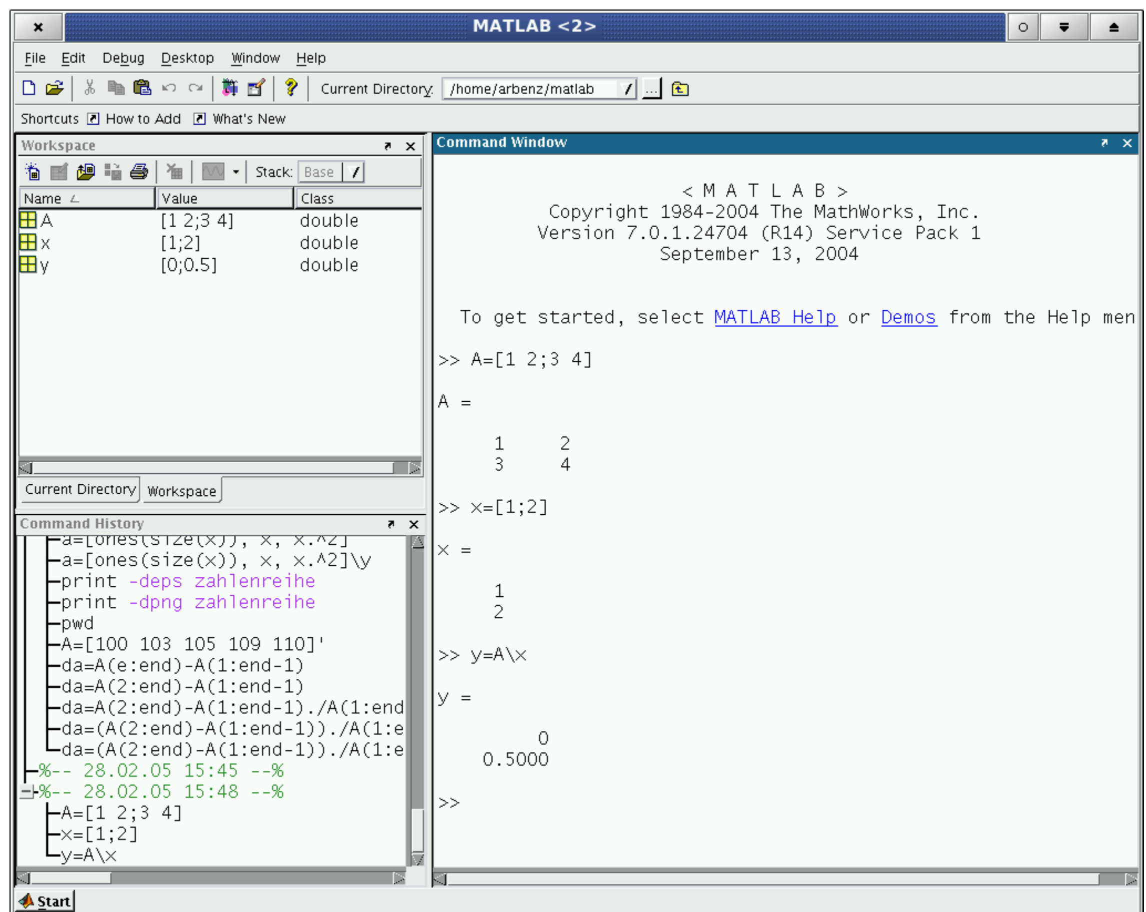

Abbildung 2.1: MATLAB-Fenster in der Default-Anordnung

Das Befehls-Fenster (*Command Window*) ist der Ort, an dem Sie MATLAB-Befehle eingeben werden. Ein Befehl wird rechts vom Doppelpfeil eingetippt und mit der `<Enter>`-Taste abgeschlos-

[1]Durch den Befehl `matlab -nodesktop` in einem Konsolenfenster wird das Kommandofenster von MATLAB im Konsolenfenster selber gestartet.

sen. Er wird dann von MATLAB ausgeführt. Eine ganze Reihe von Befehlen können zusammen in einer Datei mit der Endung `.m` abgespeichert werden. Solche Dateien werden M-Files genannt. Die Datei wird im Befehls-Fenster mit ihrem Namen (ohne Endung) aufgerufen. Der Benutzer kann die Datei `bucky.m` also mit dem Befehl `bucky` starten. Dazu muss sich die Datei im aktuellen Verzeichnis (*current directory*) befinden, welches in der Befehlsleiste oben ausgewählt wird, oder in einem Verzeichnis des `matlabpath` befinden[2].

Im Teilfenster links unten (*command history*) werden Befehle gespeichert, die Sie bereits ausgeführt haben. Durch (Doppel-)Klick auf einen Befehl in diesem Fenster wird der Befehl ins Befehlsfenster kopiert und (noch einmal) ausgeführt. Im Teilfenster links oben werden (default-mässig) die Files des gegenwärtige Verzeichnisses (*current directoy*) angezeigt und MATLAB's Arbeitsspeicher (*workspace*). Im Arbeitsspeicher befinden sich die Variablen, die sie angelegt haben. Durch einen Doppelklick auf den Variablennamen wird ein Arrayeditor geöffnet, mit welchem man die Matrixelemente editieren kann.

Die Organisation des MATLAB-Fensters kann mit der Maus via `Desktop -> Desktop Layout` verändert werden. Man kann zum Beispiel alle Fenster ausser dem Befehlsfenster löschen. Die Teilfenster können auch vom MATLAB-Fenster getrennt (*undock*) werden. (Das Fenster hat zu diesem Zweck einen kleinen gebogenen Pfeil.)

MATLAB bietet vielfältige Hilfen an. `Help` an der oberen Befehlsleiste eröffnet ein neues Teilfenster, welches sämtliche Hilfsangebote von MATLAB auflistet. Neben der Dokumentation einzelner Befehle finden sich hier auch ein MATLAB-Einführungskurs (*Getting Started*), ein Benützer-Handbuch (*User Guide*), Demos, pdf-Files der Dokumentation, und vieles mehr.

MATLAB wird durch eintippen von

```
>> quit
```

oder

```
>> exit
```

wieder verlassen. Natürlich kann auch das einfach das MATLAB-Fenster geschlossen werden.

2.2 Hilfe aus dem Befehlsfenster

Der erste Teil des Kurses beschäftigt sich mit der Bedienung des Befehlsfensters. Nach dem Start von MATLAB sieht das Befehlsfenster so aus:

```
                  < M A T L A B >
        Copyright 1984-2004 The MathWorks, Inc.
       Version 7.0.1.24704 (R14) Service Pack 1
                September 13, 2004


 To get started, select MATLAB Help or Demos from the Help menu.

>>
```

Wir wollen zunächst noch eine weitere Art erwähnen, wie Sie Hilfe zu Befehlen erhalten können: mit dem MATLAB-Befehl `help` der als Argument einen Befehlsnamen (den Sie kennen müssen) hat. Der Befehl

```
>> help help
```

zeigt eine Beschreibung der vielfältigen Möglichkeiten des Befehls `help` im Kommandofenster an. Der Befehl

```
>> doc help
```

zeigt die gleiche Information wie `help help` in einem separaten MATLAB-Fenster in html-Format an. Statt dieses Beispiel auszuführen sehen wir uns ein kürzeres Beispiel etwas genauer an.

[2]Durch den Befehl `path`, `addpath`, o.ä. kann der Pfad, den MATLAB bei der Befehlssuche durchläuft, modifiziert werden.

```
>> help tic
 TIC Start a stopwatch timer.
    TIC and TOC functions work together to measure elapsed time.
    TIC saves the current time that TOC uses later to measure
    the elapsed time. The sequence of commands:

              TIC
              operations
              TOC

    measures the amount of time MATLAB takes to complete the one
    or more operations specifed here by "operations" and displays
    the time in seconds.

    See also toc, cputime.

    Reference page in Help browser
       doc tic
```

Auf die letzte Zeile (`doc tic`) kann geklickt werden[3]. Dann öffnet ein ausführliches Hilfe-Fenster im html-Format. Ebenso kann `doc tic` eigetippt werden. Ebenso kann auf die in der Zeile 'See also' angegebenen Befehle geklickt werden, um Infomationen zu diesen zu erhalten.

`lookfor` sucht in allen Files, die via `matlabpath` zureifbar sind nach einer vorgegebenen Zeichenfolge in der *ersten* Kommentarzeile. Dies ist nützlich, wenn man den Namen einer Funktion nicht mehr genau kennt.

Der Cursor kann nicht mit der Maus auf eine vorangegengene Zeile bewegt werden, um z.B. einen falschen Befehl zu korrigieren. Ein Befehl muss neu eingegeben werden. Allerdings kann ein früher eingegebener Befehl mit den Pfeiltasten (↑, ↓) auf die aktuelle Kommandozeile kopiert werden, wo er editiert werden kann[4]. Der Cursor kann unter Benutzung der Maus oder der Pfeiltasten (←, →) auf der Zeile vor- und zurückbewegt werden. Wie schon erwähnt kann ein früherer Befehl (mit Doppelklick) auch aus der *command history* kopiert werden.

2.3 Matrizen als grundlegender Datentyp

MATLAB ist ein auf Matrizen basierendes Werkzeug. Alle Daten, die in MATLAB eingegeben werden werden von MATLAB als Matrix oder als mehrdimensionales Array abgespeichert. Sogar eine einzige Zahl wird als Matrix (in diesem Fall eine 1×1-Matrix) abgespeichert.

```
>> A = 100

A =

   100

>> whos
  Name      Size                         Bytes  Class

  A         1x1                              8  double array

Grand total is 1 element using 8 bytes
```

Der Befehl `whos` zeigt den Arbeitsspeicher an. Man beachte auch das Teilfenster *workspace*. Unabhängig davon, welcher Datentyp gebraucht wird, ob numerische Daten, Zeichen oder logische

[3] Das gilt nicht, wenn Sie MATLAB in einem Konsolenfenster mit dem Parameter `nodesktop` gestartet haben. Dort müssen Sie `doc tic` eingeben.

[4] Wenn man eine Buchstabenfolge eingibt werden beim Drücken der ↑-Taste nur die Befehle durchlaufen, die mit den angegebenen Buchstaben anfangen.

Daten, MATLAB speichert die Daten in Matrixform ab. Zum Beispiel ist in MATLAB die Zeichenfolge (*string*) “Hello World” ein 1×11 Matrix von einzelnen Zeichen.

```
>> b='hello world'

b =

hello world

>> whos
  Name      Size                    Bytes  Class

  A         1x1                         8  double array
  b         1x11                       22  char array

Grand total is 12 elements using 30 bytes
```

Bemerkung:

Man kann auch Matrizen aufbauen, deren Elemente komplizierterer Datentypen sind, wie Strukturen oder *cell arrays*. In diesem Kurs werden wir darauf nicht eingehen.

Noch eine Bemerkung:

Indizes von Vektor- und Matrixelementen haben wie in FORTRAN *immer* die Werte $1, 2, \ldots$ Das erste Element eines Vektors `x` ist somit `x(1)`. Das Element einer Matrix `y` ‘links oben’ ist `y(1,1)`.

2.4 Zuweisungen

Das Grundkonstrukt der MATLAB-Sprache ist die *Zuweisung*:

[*Variable* =] *Ausdruck*

Ein *Ausdruck* ist zusammengesetzt aus Variablennamen, Operatoren und Funktionen. Das Resultat des Ausdrucks ist eine Matrix, welche der angeführten Variable auf der linken Seite des Gleichheitszeichens zugeordnet wird. Wenn keine Variable angegeben wird, wird das Resultat in die Variable `ans` gespeichert. Wenn die Zuweisung mit einem Semicolon endet, wird das Resultat nicht auf den Bildschirm geschrieben, die Operation wird aber ausgeführt! Wenn eine Zuweisungen länger als eine Zeile wird, müssen die fortzusetztenden Zeilen mit *drei* Punkten beendet werden.

```
>> s = 1 - 1/2 + 1/3 - 1/4 + 1/5 - 1/6 + 1/7 ...
   - 1/8 + 1/9 - 1/10;
>> sin(pi/4)

ans =

    0.7071
```

Man beachte, dass der Variable auf der linken Seite des Gleichheitszeichens ein beliebiges Resultat (eine beliebige Matrix) zugewiesen werden kann. MATLAB kümmert sich um die Zuordnung des nötigen Speicherplatzes.

2.5 Namen

Namen von Variablen und Funktionen beginnen mit einem Buchstaben gefolgt von einer beliebigen Zahl von Buchstaben, Zahlen oder Unterstrichen (_). MATLAB unterscheidet zwischen Gross- und Kleinschreibung!

```
>> I
??? Undefined function or variable 'I'.

>> minus1 = i^2

minus1 =

    -1

>> a_b_c = 1*2*3*pi

a_b_c =

   18.8496

>> 1minus
??? 1minus
|
Error: Missing MATLAB operator.

>> t=i/0
Warning: Divide by zero.

t =

      NaN +    Infi

>>
```

MATLAB rechnet nach dem IEEE Standard für Fliesskommazahlen [16].

2.6 Eingabe von Matrizen

In MATLAB können Matrizen auf verschiedene Arten eingegeben werden:

1. durch explizite Eingabe der Matrixelemente,
2. durch Aufruf einer Matrix-generierenden Funktion,
3. durch Aufbau einer Matrix durch ein eigenes M-File, oder
4. durch Laden aus einem externen File.

2.6.1 Matrixaufbau durch explizite Eingabe der Matrixelemente

Die Grundregeln bei der Eingabe von Matrixelementen sind:

- Elemente einer (Matrix-)Zeile werden durch Leerzeichen oder Komma getrennt.
- Das Ende einer Zeile wird durch einen Strichpunkt (;) angegeben.
- Die ganze Liste von Zahlen wird in eckige Klammern ([ ]) geschlossen.

Da Vektoren $n \times 1$- oder $1 \times n$-Matrizen sind gilt nachfolgendes in analoger Weise für Vektoren.
Das magische Quadrat von Dürer, siehe Abb. 2.2 erhält man durch Eingabe von

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]

A =
```

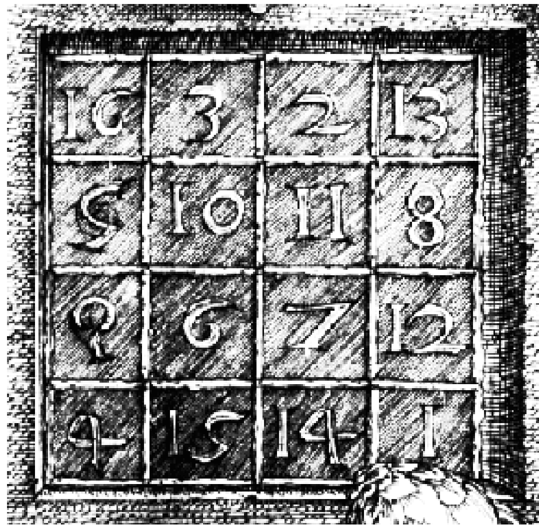

Abbildung 2.2: Das magische Quadrat aus Dürers Zeichnung "Melancholie".

```
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1

>>
```

Ein Spaltenvektor ist eine $n \times 1$-Matrix, ein Zeilenvektor ist eine $1 \times n$-Matrix und ein Skalar ist eine 1×1-Matrix. Somit ist nach obigem

```
>> u = [3; 1; 4], v = [2 0 -1], s = 7

u =

     3
     1
     4


v =

     2     0    -1


s =

     7

>>
```

Wenn klar ist, dass ein Befehl noch nicht abgeschlossen ist, so gibt MATLAB keinen Fehlermeldung. Man kann eine Matrix so eingeben:

```
>> B = [1 2;
   3 4;
```

```
    5 6]

B =

     1     2
     3     4
     5     6

>>
```

Auch Matrizen können als Elemente in neuen Matrizen verwendet werden. Hier ist ein Beispiel in welchem eine Matrix "rekursiv" definiert wird.

```
>> A=[1 2 3; 4 5 6; 7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> A = [A 2*A -A]
A =
     1     2     3     2     4     6    -1    -2    -3
     4     5     6     8    10    12    -4    -5    -6
     7     8     9    14    16    18    -7    -8    -9
>>
```

Dies ist ein schönes Beispiel fuer die dynamische Speicherplatzallokation von MATLAB. Zunächst wurden neun Fliesskommazahlen (72 Byte) für A reserviert. Nach der Ausführung des Ausdrucks belegt A dreimal mehr Speicherplatz, d.h., 216 Byte.

2.6.2 Matrixaufbau durch Aufruf einer matrixbildenden Funktion

In MATLAB gibt es verschiedene Matrix generierenden Funktionen. Die m.E. wichtigsten sind in Tabelle 2.1 aufgelistet.

`eye`	Einheitsmatrix
`zeros`	Nullmatrix
`ones`	Einser-Matrix
`diag`	siehe `help diag`
`triu`	oberes Dreieck einer Matrix
`tril`	unteres Dreieck einer Matrix
`rand`	Zufallszahlenmatrix
`magic`	magisches Quadrat

Tabelle 2.1: Matrixbildende Funktionen

`zeros(m,n)` produziert eine $m \times n$-Matrix bestehend aus lauter Nullen; `zeros(n)` ergibt eine $n \times n$-Nuller-Matrix. Wenn A eine Matrix ist, so definiert `zeros(A)` eine Nuller-Matrix mit den gleichen Dimensionen wie A.

Hier einige Beispiele

```
>> A = [ 1 2 3; 4 5 6; 7 8 9 ];
>> diag(A)   % speichert die Diagonale einer Matrix als Vektor
ans =
     1
     5
     9

>> diag([1 2 3])   % macht aus einem Vektor eine Diagonalmatrix
```

```
ans =
     1     0     0
     0     2     0
     0     0     3

>> eye(size(A))   % erzeugt die Einheitsmatrix von
                  % gleicher Groesse wie A
ans =
     1     0     0
     0     1     0
     0     0     1

>> ones(size(A))  % erzeugt eine Matrix mit Einselementen
ans =
     1     1     1
     1     1     1
     1     1     1

>> zeros(size(A))   %  erzeugt eine Nullmatrix
ans =
     0     0     0
     0     0     0
     0     0     0
>> triu(A)
ans =

     1     2     3
     0     5     6
     0     0     9

>> M = magic(4)  % Eulers magisches Quadrat, siehe Abb. 2.2
M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

2.6.3 Aufbau einer Matrix durch ein eigenes M-File

Wir werden die Struktur von M-Files später eingehend behandeln. Hier erwähnen wir nur die sogenannten *Script-Files*: Diese ASCII-Files enthalten einfach eine Reihe von MATLAB-Befehlen, die beim Aufruf des Files einer nach dem anderen ausgeführt wird als wären sie auf der Kommandozeile eingegeben worden. Sei `matrix.m` ein File bestehend aus einer einzigen Zeile mit folgendem Text:

```
A = [ 1 2 3; 4 5 6; 7 8 9 ]
```

Dann bewirkt der Befehl `matrix`, dass der Variable `A` ebendiese 3×3-Matrix zugeordnet wird.

```
>> matrix

A     =


     1     2     3
     4     5     6
     7     8     9
```

2.6.4 Matrixaufbau durch Laden aus einem externen File

Die Funktion `load` liest binäre Files, die in einer früheren MATLAB-Sitzung erzeugt wurden oder liest Textfiles, die numerische Daten enthalten. Das Textfile sollte so organisiert sein, dass es eine rechteckige Tabelle von Zahlen enthält, die durch Leerzeichen getrennt sind, eine Textzeile pro Matrixzeile enthalten. Jede Matrixzeile muss gleichviele Elemente haben. Wenn z.B. ausserhalb von MATLAB eine Datei `mm` mit den vier Zeilen

```
16.0     3.0     2.0    13.0
 5.0    10.0    11.0     8.0
 9.0     6.0     7.0    12.0
 4.0    15.0    14.0     1.0
```

generiert wurde, so liest der Befehl `load mm` das File und kreiert eine Variable `mm`, die die 4×4-Matrix enthält.

Eine weitere Möglichkeit, Daten aud Files zu lesen, bietet der Befehl `dlmread`. Der Import Wizard (`File -> Import Data...`) erlaubt es, Daten in verschiedenen Text und binären Formaten in MATLAB zu laden.

2.6.5 Aufgaben

Diese Aufgaben können am leichtesten mit hilfe eines Editors, z.B., des MATLAB-Editors gelöst werden.

1. Überlegen Sie sich Varianten, wie man in MATLAB die Matrix

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

   definieren kann.

2. Wie würden Sie vorgehen, wenn Sie zu vorgegebenem n eine $n \times n$ Matrix mit analoger Struktur wie vorher (lauter Einsen ausser auf der Diagonale, wo die Elemente null sind) aufbauen wollen.

3. Geben Sie die Matrix

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

   ein. Überlegen Sie sich, was passiert, wenn Sie den Befehl

```
A(4,6) = 17
```

   eintippen. Danach geben Sie den Befehl ein.

4. Finden Sie heraus, was der Befehl `fliplr` (oder `flipud`) macht. Konstruieren Sie danach die Matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

5. Wie wäre es mit

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}?$$

6. Definieren Sie die Matrizen O und E

```
n = 5; O=zeros(n), E=ones(n)
```

Dann erzeugen Sie das Schweizerkreuz von Figur 2.3, d.h. eine 15×15 Matrix A. Ein Punkt bedeutet dabei eine Eins, kein Punkt eine Null.

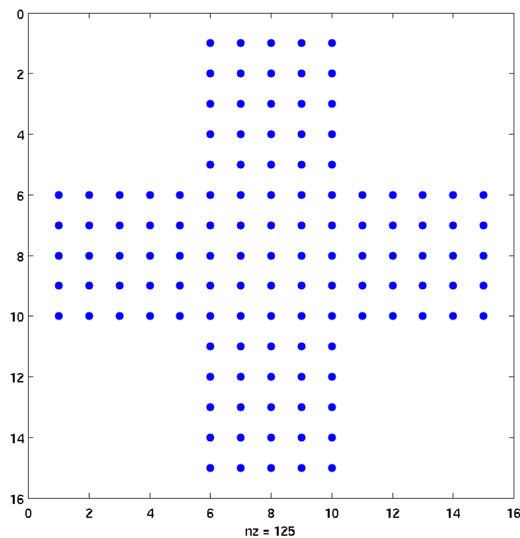

Abbildung 2.3: Einfaches Schweizerkreuz

Abbildung 2.3 können Sie mit

```
spy(A)
```

erzeugen.

2.7 Operationen mit Matrizen

2.7.1 Addition und Subtraktion von Matrizen

Addition und Subtraktion von Matrizen geschieht elementweise. Matrizen, die addiert oder voneinander subtrahiert müssen dieselben Dimensionen haben.

```
>> A=ones(3)

A =

     1     1     1
     1     1     1
     1     1     1

>> B = round(10*rand(3))  % rand: Zufallszahlenmatrix
```

```
B =

    10     5     5
     2     9     0
     6     8     8

>> B-A

ans =

     9     4     4
     1     8    -1
     5     7     7

>>
```

Falls die Dimensionen nicht zusammenpassen bring MATLAB eine Fehlermeldung.

```
>> A + eye(2)
??? Error using ==> plus
Matrix dimensions must agree.

>>
```

2.7.2 Vektorprodukte and Transponierte

Ein Zeilenvektor und ein Spaltenvektor können miteinander multipliziert werden. Das Resultat ist entweder ein Skalar (das innere oder Skalarprodukt) oder eine Matrix, das äussere Produkt.

```
>> u = [3; 1; 4];
>> v = [2 0 -1]; x = v*u

x =

     2

>> X = u*v

X =

     6     0    -3
     2     0    -1
     8     0    -4

>>
```

Bei reellen Matrizen spiegelt die Transposition die Elemente an der Diagonale. Seinen die Elemente der Matrix $m \times n$ Matrix A bezeichnet als $a_{i,j}$, $1 \leq i \leq m$ und $1 \leq j \leq n$. Dann ist die Transponierte von A die $n \times m$ Matrix B mit den Elementen $b_{j,i} = ai, j$. MATLAB benützt den Apostroph für die Transposition.

```
>> A = [1 2 3;4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9
```

```
>> B=A'

B =

     1     4     7
     2     5     8
     3     6     9

>>
```

Transposition macht aus einem Zeilenvektor einen Spaltenvektor.

```
>> x=v'

x =

     2
     0
    -1

>>
```

Das Produkt zweier reeller Spaltenvektoren gleicher Länge ist nicht definiert, jedoch sind die Produkte `x'*y` und `y'*x` gleich. Sie heissen Skalar- oder inneres Produkt.

Für einen komplexen Vektor `z` bedeutet `z'` *komplex-konjugierte* Transposition:

```
>> z = [1+2i 3+4i]

z =

   1.0000 + 2.0000i   3.0000 + 4.0000i

>> z'

ans =

   1.0000 - 2.0000i
   3.0000 - 4.0000i

>>
```

Für komplexe Vektoren sind die beiden Skalarprodukte `x'*y` und `y'*x` komplex konjugiert. Das Skalarprodukte `x'*x` ist reell. Es gibt aber auch den *punktierten* Operator `.'`

```
>> z=z.', w = [4+6i;7+8i]

z =

   1.0000 + 2.0000i
   3.0000 + 4.0000i

w =

   4.0000 + 6.0000i
   7.0000 + 8.0000i

>> z'*w, w'*z
```

```
ans =

  69.0000 - 6.0000i

ans =

  69.0000 + 6.0000i
```

2.7.3 Zugriff auf Matrixelemente

Das Element der Matrix A in Zeile i und Spalte j wird durch `A(i,j)` bezeichnet.

```
>> % Hier kommt Duerers magisches Quadrat
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
>> A(2,1) + A(2,2) + A(2,3) + A(2,4)
ans =
    34
>> A(5,1)
??? Index exceeds matrix dimensions.

>> v = [1;2;3;4;5];
>> v(4)
ans =
     4
>>
```

Bemerkung: Matrixelemente können auch mit einem einzelnen Index zugegriffen werden. MATLAB interpretiert die Matrix so, als wären alle Spalten übereinandergestapelt.

```
>> A(8)
ans =
    15
>>
```

In der 4×4-Matrix A ist des achte Element also das letzte der zweiten Spalte.

2.7.4 Der Doppelpunkt-Operator

Der Doppelpunkt ist ein äusserst wichtiger und nützlicher MATLAB-Operator. Der Ausdruck

```
1:10
```

ist ein Zeilenvektor, der die Elemente 1 bis 10 enthält. Der Ausdruck

```
1:3:10
```

liefert den Vektor `[1 4 7 10]`. Allgemein kann man schreiben

```
[i : j : k]
```

was einen Vektor mit erstem Element i erzeugt, gefolgt von $i + j$, $i + 2j$, bis zu einem Element welches $\geq k$ ist, falls $j > 0$ und $\leq k$ falls $j < 0$ ist..

```
>> 1:10
ans =
     1     2     3     4     5     6     7     8     9    10
>> 1:3:10
ans =
     1     4     7    10
>> 100:-7:50
ans =
   100    93    86    79    72    65    58    51
>> 1:3:12
ans =
     1     4     7    10
>> 100:-7:50
ans =
   100    93    86    79    72    65    58    51
>> x=[0:.25:1]
x =
         0    0.2500    0.5000    0.7500    1.0000
```

Der Doppelpunkt-Operator ist sehr bequem, wenn man Teile von Matrizen zugreiffen will.

```
>> A = [16 3 2 13; 5 10 11 8; 9 6 7 12; 4 15 14 1]
A =
    16     3     2    13
     5    10    11     8
     9     6     7    12
     4    15    14     1
>> A(2,1:4)        % 2-te Zeile von A
ans =
     5    10    11     8
>> A(2,1:3)        % Teil der 2-te Zeile von A
ans =
     5    10    11
>> A(2:3,2:3)
ans =
    10    11
     6     7
>> A(3:end,3)
ans =
     7
    14
>> A(4,:)
ans =
     4    15    14     1
>>
```

2.7.5 Arithmetische Operationen

Mit Matrizen können die in Tabelle 2.2 aufgeführten *arithmetischen Operationen* durchgeführt werden. Unter Matrixoperationen werden dabei die aus der Matrixalgebra bekannten Operationen verstanden. Durch Voranstellen eines Punkts vor die Operatoren erhält man Operationen, welche *elementweise* ausgeführt werden.

```
>> a=[1 2;3 4]

a =
```

Matrixoperationen		Elementweise Operationen	
+	Addition	+	Addition
-	Subtraktion	-	Subtraktion
*	Multiplikation	.*	Multiplikation
/	Division von rechts	./	Division von rechts
\	Division von links	.\	Division von links
^	Potenz	.^	Potenz
'	Hermitesch Transponierte	.'	Transponierte

Tabelle 2.2: Arithmetische Operationen

```
     1     2
     3     4

>> a^2

ans =

     7    10
    15    22

>> a.^2

ans =

     1     4
     9    16
```

Vergleichsoperatoren		logische Operatoren	
<	kleiner als	&	and
<=	kleiner oder gleich	|	or
>	grösser als	~	not
>=	grösser oder gleich		
==	gleich		
~=	ungleich		

Tabelle 2.3: Vergleichende und logische Operatoren

2.7.6 Vergleichsoperationen

Weiter können die in Tabelle 2.3 aufgelisteten *Vergleichsoperationen* mit Matrizen durchgeführt werden. Als Resultat einer Vergleichsoperation erhält man eine 0-1-Matrix. Erfüllen die (i, j)-Elemente der zwei verglichenen Matrizen die Bedingung, so wird das (i, j)-Element der Resultatmatrix 1, andernfalls 0 gesetzt.

```
>> b=ones(2)   %  a  wie im vorigen Beispiel

b =

     1     1
     1     1


>> a > b
```

```
ans =

     0     1
     1     1
```

2.7.7 Logische Operationen

Die *logischen Operationen* in Tabelle 2.3 werden elementweise auf 0-1-Matrizen angewandt

```
>> ~(a>b)

ans =

     1     0
     0     0
```

2.7.8 Aufgaben

1. Konstruieren Sie die 15×15-Matrix mit den Einträgen von 1 bis 225 ($= 15^2$). Die Einträge sollen lexikographisch (zeilenweise) angeordnet sein. Verwenden Sie neben dem Doppelpunkt-Operator die Funktion `reshape`.

2. Multiplizieren Sie die entstehende Matrix mit der Schweizerkreuzmatrix. Verwenden Sie das 'normale' Matrixprodukt und das elementweise. Nennen Sie die letztere Matrix C.

3. Filtern Sie jetzt aus C die Elemente zwischen 100 und 200 heraus. Dazu definieren Sie eine logische Matrix, die Einsen an den richtigen Stellen hat. Danach multiplizieren Sie diese logische Matrix elementweise mit C.

2.8 Ausgabeformate

Zahlen können in verschiendener Weise am Bildschirm angezeigt werden.

```
>> s = sin(pi/4)

s =

    0.7071

>> format long
>> s

s =

    0.70710678118655

>> format short e
>> s

s =

   7.0711e-01

>> format short
```

In Tabelle 2.4 auf Seite 20 ist eine Übersicht über die möglichen Parameter von `format` gegeben. Man beachte, dass man mit dem aus C bekannten Befehl `fprintf` die Ausgabe viel feiner steuern kann als mit `format`.

Befehl	Beschreibung
`format short`	Scaled fixed point format with 5 digits.
`format long`	Scaled fixed point format with 15 digits for double and 7 digits for single.
`format short e`	Floating point format with 5 digits.
`format long e`	Floating point format with 15 digits for double and 7 digits for single.
`format short g`	Best of fixed or floating point format with 5 digits.
`format long g`	Best of fixed or floating point format with 15 digits for double and 7 digits for single.
`format hex`	Hexadecimal format.
`format +`	The symbols +, - and blank are printed for positive, negative and zero elements. Imaginary parts are ignored.
`format bank`	Fixed format for dollars and cents.
`format rat`	Approximation by ratio of small integers.
`format compact`	Suppresses extra line-feeds.
`format loose`	Puts the extra line-feeds back in.

Tabelle 2.4: MATLAB-Formate, siehe `help format`

2.9 Komplexe Zahlen

Komplexe Zahlen werden in MATLAB in Summenform dargestellt, wobei die imaginäre Einheit $\sqrt{-1}$ mit dem Buchstaben `i` oder `j` bezeichnet wird.

```
>> i

ans =

        0 + 1.0000i

>> 3+5i

ans =

   3.0000 + 5.0000i

>> 3+5*i

ans =

   3.0000 + 5.0000i

>>
```

Man beachte, dass `i` eine vordefinierte *Funktion* ist (und ebenso `j`). Der Buchstabe i ist nicht geschützt, sondern kann (z.B.) auch als Variablennamen verwendet werden, vgl. Abschnitt 3.2.

2.10 Speichern von Daten zur späteren Weiterverwendung

Mit dem Befehl `who` oder `whos` erhält man eine Liste aller gebrauchten Variable.

```
>> who

Your variables are:
```

```
a_b_c     i         s
ans       minus1    t

>> whos
  Name         Size         Bytes  Class

  a_b_c        1x1              8  double array
  ans          1x1              8  double array (logical)
  i            1x1             16  double array (complex)
  minus1       1x1              8  double array
  s            1x1              8  double array
  t            1x1             16  double array (complex)

Grand total is 6 elements using 64 bytes
```

Oft will man seine Daten speichern, um in einer späteren MATLAB-Sitzung mit ihnen weiter arbeiten zu können.

```
>> save mydata
>> clear
>> who
>> what

MAT-files in the current directory /home/arbenz/tex/matlab

mydata

>> !ls mydata*
mydata.mat
>> load mydata
>> who

Your variables are:

a_b_c     i         s
ans       minus1    t

>>
```

In Windows 9x gibt es auch einen Save-Knopf auf dem Fensterrahmen des Befehlsfensters.

2.11 Strukturen (Structs)

Ähnlich wie die Programmiersprache C hat auch MATLAB Strukturen. Strukturen dürfen sämtliche Datenobjekte in beliebiger Mischung enthalten. Die in einer Strukturdefinition angegebenen Objekte nennt man Mitglieder.

```
>> mitglied.name = 'Arbenz';
>> mitglied.vorname = 'Peter';
>> mitglied.wohnort = 'Uetikon a.S.';
>> mitglied

mitglied =

       name: 'Arbenz'
    vorname: 'Peter'
```

```
        wohnort: 'Uetikon a.S.'

>> mitglied(2) = struct('name','Meier','vorname','Hans','wohnort','Zurich')

mitglied =

1x2 struct array with fields:
    name
    vorname
    wohnort

>> mitglied(2)

ans =

       name: 'Meier'
    vorname: 'Hans'
    wohnort: 'Zurich'

>>
```

Strukturen werden häufig bei der Übergabe von Parametern an Funktionen verwendet. Wenn z.B. ein Gleichungssystem iterative gelöst werden soll, so könnte man eine bestimmte Fehlerschranke setzen, oder die Zahl der Iterationsschritte beschränken wollen. Eine Möglichkeit

2.12 Zeichenketten (Strings)

Eine Zeichenkette ist ein Array von Zeichen (*characters*). Jedes Zeichen wird intern durch eine Zahl dargestellt.

```
>> txt='Ich lerne jetzt MATLAB'

txt =

Ich lerne jetzt MATLAB

>> num=double(txt)

num =

  Columns 1 through 12

    73    99   104    32   108   101   114   110   101    32   106   101

  Columns 13 through 22

   116   122   116    32    77    65    84    76    65    66

>> char(num)

ans =

Ich lerne jetzt MATLAB

>> whos txt num
  Name      Size         Bytes  Class

  num       1x22           176  double array
  txt       1x22            44  char array
```

```
Grand total is 44 elements using 220 bytes

>> txt(2:5)

ans =

ch l

>> strcat(txt(12:22),txt(1:11))

ans =

etzt MATLABIch lerne j
```

2.13 Cell arrays

In der Programmiersprache von MATLAB gibt es die Struktur `cell`, ein Array von Elementen mit beliebige Typ[5].

```
>> a=['Peter';'Arbenz';'Uetikon a.S.']
 ??? Error using ==> vertcat
 All rows in the bracketed expression must have the same
 number of columns.

>> c={'Peter';'Arbenz';'Uetikon a.S.'}

c =

'Peter'
'Arbenz'
'Uetikon a.S.'

>> c(1:4,2)={'Vorname';'Name';'Wohnort';[ 1 2 3 4 5]}

c =

    'Peter'           'Vorname'
    'Arbenz'          'Name'
    'Uetikon a.S.'    'Wohnort'
                []    [1x5 double]

>> c(4,2)

ans =

    [1x5 double]

>> c{4,2}

ans =

     1     2     3     4     5
```

[5]Siehe Abschnitt 5.3.5.

Kapitel 3

Funktionen

3.1 Skalare Funktionen

Viele MATLAB-Funktionen sind skalare Funktionen und werden *elementweise* ausgeführt, wenn sie auf Matrizen angewandt werden. Die wichtigsten Funktionen dieser Art sind in Tab. 3.1 gegeben. Alle Funktionen dieser Art können mit dem Befehl **help elfun** aufgelistet werden. Der Befehl **help specfun** gibt eine Liste von speziellen mathematischen Funktionen wie Gamma-, Bessel-, Fehlerfunktion, aber auch ggT oder kgV aus. Diese Funktionen werden oft auf Vektoren angewandt, wie folgendes Beispiel zeigt.

```
>> x=[0:pi/50:2*pi];
>> y = cos(x.^2);
>> plot(x,y)
>> xlabel('x-axis')
>> ylabel('y-axis')
>> title('Plot of y = cos(x^2), 0 < x < 2\pi')
```

Diese Befehlssequenz generiert Abbildung 3.1

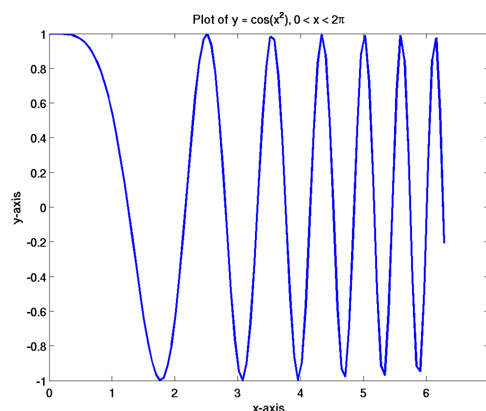

Abbildung 3.1: Plot der Funktion $\cos(x^2)$

3.2 Spezielle Konstanten

MATLAB hat einige spezielle Funktionen, die nützliche Konstanten zur Verfügung stellen, siehe Tab. 3.2.

Man beachte, dass Funktionennamen hinter Variablennamen versteckt werden können! Z.B. kann **i** mit einer Zahl (Matrix, etc.) überschrieben werden.

Kategorie	Funktion	Beschreibung (gemäss MATLAB)
Trigonometrische	`sin`	Sine
	`sinh`	Hyperbolic sine
	`asin`	Inverse sine
	`cos`	Cosine
	`cosh`	Hyperbolic cosine
	`acos`	Inverse cosine
	`acosh`	Inverse hyperbolic cosine
	`tan`	Tangent
	`tanh`	Hyperbolic tangent
	`atan`	Inverse tangent
	`atanh`	Inverse hyperbolic tangent
	`cot`	Cotangent
	`coth`	Hyperbolic cotangent
	`acot`	Inverse cotangent
	`acoth`	Inverse hyperbolic cotangent
Exponentiell	`exp`	Exponential
	`expm1`	Compute exp(x)-1 accurately
	`log`	Natural logarithm
	`log1p`	Compute log(1+x) accurately
	`log10`	Common (base 10) logarithm
	`log2`	Base 2 logarithm and dissect floating point number
	`sqrt`	Square root
	`nthroot`	Real n-th root of real numbers
Komplex	`abs`	Absolute value
	`angle`	Phase angle
	`complex`	Construct complex data from real and imaginary parts
	`conj`	Complex conjugate
	`imag`	Complex imaginary part
	`real`	Complex real part
Runden und Rest	`fix`	Round towards zero
	`floor`	Round towards minus infinity
	`ceil`	Round towards plus infinity
	`round`	Round towards nearest integer
	`mod`	Modulus (signed remainder after division)
	`rem`	Remainder after division
	`sign`	Signum

Tabelle 3.1: Elementare Funktionen

Funktion	Beschreibung
`pi`	3.14159265 . . .
`i`	Imaginäre Einheit ($\sqrt{-1}$)
`j`	Wie `i`
`eps`	Relative Genauigkeit der Fliesskomma-Zahlen ($\varepsilon = 2^{-52}$)
`realmin`	Kleinste Fliesskomma-Zahl (2^{-1022})
`realmax`	Grösste Fliesskomma-Zahl ($(2-\varepsilon)^{1023}$)
`Inf`	Unendlich (∞)
`NaN`	Not-a-number

Tabelle 3.2: Spezielle Funktionen

```
    >> i=3

    i =

         3

>> 3+5*i

ans =

    18
```

Wenn hier `i` für die imaginäre Einheit $\sqrt{-1}$ stehen soll, muss man die Variable `i` löschen.

```
>> clear i
>> i

ans =

        0 + 1.0000i

>> 3+5*i

ans =

   3.0000 + 5.0000i

>>
```

3.3 Vektorfunktionen

Eine zweite Klasse von MATLAB-Funktionen sind Vektorfunktionen. Sie können mit derselben Syntax sowohl auf Zeilen- wie auf Spaltenvektoren angewandt werden. Solche Funktionen operieren *spaltenweise*, wenn sie auf Matrizen angewandt werden. Einige dieser Funktionen sind

Funktion	Beschreibung
`max`	Largest component
`mean`	Average or mean value
`median`	Median value
`min`	Smallest component
`prod`	Product of elements
`sort`	Sort array elements in ascending or descending order
`sortrows`	Sort rows in ascending order
`std`	Standard deviation
`sum`	Sum of elements
`trapz`	Trapezoidal numerical integration
`cumprod`	Cumulative product of elements
`cumsum`	Cumulative sum of elements
`cumtrapz`	Cumulative trapezoidal numerical integration
`diff`	Difference function and approximate derivative

Tabelle 3.3: Übersicht Vektorfunktionen

```
>> z=[-3 -1 4 7 7 9 12]
```

```
z =

    -3    -1     4     7     7     9    12

>> [min(z), max(z)]

ans =

    -3    12

>> median(z)

ans =

     7

>> mean(z)

ans =

     5

>> mean(z), std(z)

ans =

     5

ans =

   5.38516480713450

>> sum(z)

ans =

    35

>> trapz(z)

ans =

  30.5000

>> (z(1) + z(end) + 2*sum(z(2:end-1)))/2

ans =

  30.5000

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

u =

     1     2     3
```

```
     4     5     6

>> max(u)

ans =

     4     5     6

>> max(max(u))

ans =

     6
```

Um das grösste Element einer Matrix zu erhalten, muss man also die Maximumfunktion zweimal anwenden.

3.3.1 Aufgabe

1. Verifizieren Sie, dass der MATLAB-Befehl `magic(n)` für $n > 2$ wirklich ein magisches n-mal-n Quadrat liefert, d.h., dass die Summen der Elemente aller Zeilen, aller Spalten und der beiden Diagonalen gleich sind. (Benötigte Befehle: `diag`, `fliplr`)

3.4 Matrixfunktionen

Die Stärke von MATLAB sind die *Matrixfunktionen*. In Tabelle 3.4 finden Sie die wichtigsten.

3.4.1 Lineare Gleichungssysteme

MATLAB bietet mehrere Möglichkeiten lineare Gleichungssysteme zu lösen. Das wichtigste Lösungsverfahren ist die *Gauss-Elimination* oder *LU-Faktorisierung*. Sei

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}$$

eine reguläre (nicht-singuläre) $n \times n$ Matrix. Dann gibt es eine Zerlegung

$$LU = PA \tag{3.1}$$

wobei

- L eine *Linksdreiecksmatrix* mit Einheitsdiagonale,
- U eine *Rechtsdreiecksmatrix* und
- P eine *Permutationsmatrix* ist.

MATLAB berechnet diese Zerlegung mit sog. Spaltenpivotsuche[1]. Ist eine solche Faktorisierung berechnet, dann kann das Gleichungssystem

$$A\mathbf{x} = \mathbf{b} \tag{3.2}$$

durch *Vorwärts-* und *Rückwärtseinsetzen* gelöst werden.

[1] Siehe `lu_demo.m`

Kategorie	Funktion	Beschreibung (gemäss MATLAB)
Matrixanalysise	`norm`	Matrix or vector norm.
	`normest`	Estimate the matrix 2-norm.
	`rank`	Matrix rank.
	`det`	Determinant.
	`trace`	Sum of diagonal elements.
	`null`	Null space.
	`orth`	Orthogonalization.
	`rref`	Reduced row echelon form.
	`subspace`	Angle between two subspaces.
Lineare Gleichungen	`\ and /`	Linear equation solution.
	`inv`	Matrix inverse.
	`cond`	Condition number for inversion.
	`condest`	1-norm condition number estimate.
	`chol`	Cholesky factorization.
	`cholinc`	Incomplete Cholesky factorization.
	`linsolve`	Solve a system of linear equations.
	`lu`	LU factorization.
	`luinc`	Incomplete LU factorization.
	`qr`	Orthogonal-triangular decomposition.
	`lsqnonneg`	Nonnegative least-squares.
	`pinv`	Pseudoinverse.
	`lscov`	Least squares with known covariance.
Eigen- und singuläre Werte	`eig`	Eigenvalues and eigenvectors.
	`svd`	Singular value decomposition.
	`eigs`	A few eigenvalues.
	`svds`	A few singular values.
	`poly`	Characteristic polynomial.
	`polyeig`	Polynomial eigenvalue problem.
	`condeig`	Condition number for eigenvalues.
	`hess`	Hessenberg form.
	`qz`	QZ factorization.
	`schur`	Schur decomposition.
Matrixfunktionen	`expm`	Matrix exponential.
	`logm`	Matrix logarithm.
	`sqrtm`	Matrix square root.
	`funm`	Evaluate general matrix function.

Tabelle 3.4: Übersicht Matrixfunktionen

Abbildung 3.2: Ein Portrait von Gauss 1777-1855

$$A\mathbf{x} = \mathbf{b} \Longleftrightarrow P^T L \underbrace{U\mathbf{x}}_{\mathbf{z}} = \mathbf{b}$$

$$L\mathbf{z} = P\mathbf{b} \qquad \text{Vorwärtseinsetzen}$$

$$U\mathbf{x} = \mathbf{z} \qquad \text{Rückwärtseinsetzen}$$

Auch bei der Berechnung der *Determinante* sollte man die Gauss-Zerlegung benützen:

$$\det A = \det P^T \cdot \det L \cdot \det U = \pm 1 \cdot \det U = \prod_{i=1}^{n} u_{ii} \tag{3.3}$$

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

L =

    1.0000         0         0
    0.7500    1.0000         0
    0.2500    1.0000    1.0000

U =

    4.0000    6.0000    9.0000
         0    0.5000   -2.7500
         0         0    4.5000

P =

     0     0     1
     0     1     0
     1     0     0

>> [L1,U1]=lu(A)
```

```
L1 =

    0.2500    1.0000    1.0000
    0.7500    1.0000         0
    1.0000         0         0

U1 =

    4.0000    6.0000    9.0000
         0    0.5000   -2.7500
         0         0    4.5000

>> P'*L - L1

ans =

     0     0     0
     0     0     0
     0     0     0
```

3.4.2 Der Backslash-Operator in Matlab

Der Befehl

```
x = A\b
```

berechnet die LU-Faktorisierung mit Spaltenpivotierung (partielle Pivotierung) und löst das System durch Vorwärts- und Rückwärtseinsetzen.

```
>> A

A =

     1     2     4
     3     5     4
     4     6     9

>> b

b =

     1
     1
     1

>> x=A\b

x =

   -1.6667
    1.1111
    0.1111

>> norm(b-A*x)

ans =
```

```
   2.2204e-16

>> y=U\L\P'*b

y =

   19.0000
  -16.5000
   16.2500

>> y=U\(L\(P'*b))

y =

   -1.6667
    1.1111
    0.1111
```

Man kann auch Gleichungssysteme mit mehreren rechten Seiten lösen.

```
>> b=[1 1;1 2;1 3]

b =

     1     1
     1     2
     1     3

>> x=A\b

x =

   -1.6667    0.3333
    1.1111    0.1111
    0.1111    0.1111

>> norm(b-A*x)

ans =

   3.1402e-16
```

Wenn man in MATLAB den Backslash-Operator braucht, wird zwar die LU-Zerlegung der involvierten Matrix berechnet. Diese Zerlegung geht aber nach der Operation verloren. Deshalb kann es sinvoll sein die Faktorisierung explizit abzuspeichern, insbesondere wenn die Gleichungssysteme hintereinander gelöst werden müssen.

3.4.3 Aufgaben

1. *Polynominterpolation.* Vorgegeben sei eine Funktion $f(x)$. Gesucht ist das Polynom

$$p(x) = a_1 x^{n-1} + a_2 x^{n-2} + \ldots + a_{n-1} x + a_n$$

vom Grad $< n$ so, dass

$$p(x_i) = f(x_i), \quad i = 1, \ldots, n.$$

Diese Interpolation ist in Matrixschreibweise

$$\begin{bmatrix} x_1^{n-1} & x_1^{n-2} & \dots & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \dots & x_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \dots & x_n & 1 \end{bmatrix} \begin{pmatrix} a_1 \\ \vdots \\ a_{n-1} \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix}$$

oder

$$V\mathbf{a} = \mathbf{f}$$

Man verwende

- Funktionen `vander` und `polyval`
- Stützstellen `x=[0:.5:3]'`.
- Funktionswerte `f=[1 1 0 0 3 1 2]'`

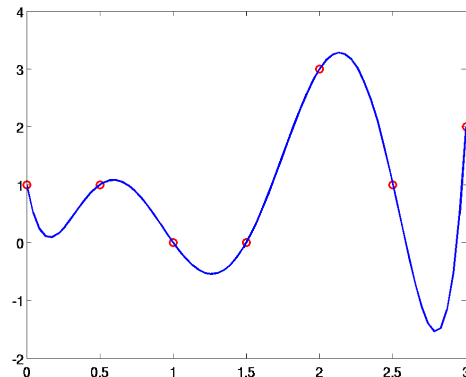

Abbildung 3.3: Polynominterpolation

Die Lösung sollte etwa so wie in Abbildung 3.3 aussehen.

2. *Was leistet Ihr Computer?*

   In dieser Aufgabe wollen wir sehen, wie schnell Ihr Computer rechnen kann.

   Die Idee ist, Gleichungssysteme mit immer höherer Ordnung mit dem Backslash-Operator zu lösen und die Ausführungszeit zu messen.

   Ihr Programm soll eine `for`-Schleife ausführen, welche für jeden der Werte in `N`,

```
N = [10:10:100,150:50:500,600:100:1000];
```

   ein Gleichungssystem der entsprechenden Grösse löst. Ein Grundgerüst für Ihr Programm könnte so aussehen

```
T = [];
for n = ...

  % Hier wird eine Matrix der Ordung n und eine rechte Seite
  % erzeugt.  Verwenden Sie dazu die Funktion rand.
  % Reservieren Sie Platz f"ur den L"osungsvektor (zeros).

  tic
```

```
  % Hier wird das Gleichungssystem geloest.

  t = toc; T = [T,t];
end
```

Sie können nun `N` gegen die zugehörige Ausführungszeiten `T` plotten. Instruktiver ist es aber, wenn Sie zu allen Problemgrössen die Mflop/s-Rate berechnen und plotten. (Aus der linearen Algebra wissen Sie sicher, dass das Auflösen eines Gleichungssystems der Ordnung n etwa $2/3n^3$ Fliesskomma-Operationen $(+, -, \times, /)$ kostet.) Wie gross ist die höchste Mflop/s-Rate, die Sie erhalten haben und wie gross ist MHz-Rate des Prozessors Ihres Computers? (Die erhaltene Kurve kann geglättet werden, indem man mehrere Messungen ausführt und jeweils die besten Resultate nimmt.)

3.4.4 Die Methode der kleinsten Quadrate (least squares)

Die sogenannte "Methode der kleinsten Quadrate" (Least Squares) ist eine Methode, um überbestimmte lineare Gleichungssysteme

$$A\mathbf{x} = \mathbf{b}, \qquad A \in \mathbb{R}^{m\times n}, \mathbf{x} \in \mathbb{R}^n, \mathbf{b} \in \mathbb{R}^m \tag{3.4}$$

zu lösen. Die $m \times n$-Matrix A hat *mehr* Zeilen als Spalten ($m > n$). Wir haben also mehr Gleichungen als Unbekannte. Deshalb gibt es im allgemeinen kein $\mathbf{x}$, das die Gleichung (3.4) erfüllt. Die *Methode der kleinsten Quadrate* bestimmt nun ein $\mathbf{x}$ so, dass die Gleichungen "möglicht gut" erfüllt werden. Dabei wird $\mathbf{x}$ so berechnet, dass der *Residuenvektor* $\mathbf{r} = \mathbf{b} - A\mathbf{x}$ minimale Länge hat. Dieser Vektor $\mathbf{x}$ ist Lösung der *Gauss'schen Normalgleichungen*

$$A^T A\mathbf{x} = A^T \mathbf{b}.$$

(Die Lösung ist eindeutig, wenn A linear unabhängige Spalten hat.) Die Gaussschen Normalgleichungen haben unter Numerikern einen schlechten Ruf, da für die Konditionszahl $\text{cond}(A^T A) = \text{cond}(A)^2$ gilt und somit die Lösung $\mathbf{x}$ durch die verwendete Methode ungenauer berechnet wird, als dies durch die Konditionszahl der Matrix A zu erwarten wäre.

Deshalb wird statt der Normalgleichungen die *QR-Zerlegung* für die Lösung der Gleichung (3.4) nach der Methode der kleinsten Quadrate vorgezogen. Dabei wird die Matrix A zerlegt als Produkt von zwei Matrizen

$$A = Q \begin{bmatrix} R \\ O \end{bmatrix} \quad \Longleftrightarrow \quad Q^T A = \begin{bmatrix} R \\ O \end{bmatrix}.$$

wobei Q $m \times m$ orthogonal und R eine $n \times n$ Rechtsdreiecksmatrix ist. Da orthogonale Matrizen die Länge eines Vektors invariant lassen, gilt

$$\begin{aligned} \|\mathbf{r}\|^2 &= \|Q^T \mathbf{r}\|^2 = \|Q^T(\mathbf{b} - A\mathbf{x})\|^2 \\ &= \left\| \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} - \begin{bmatrix} R \\ O \end{bmatrix} x \right\|^2 = \|\mathbf{y}_1 - R\mathbf{x}\|^2 + \|\mathbf{y}_2\|^2. \end{aligned}$$

Daraus ist ersichtlich, dass $||r||^2$ minimiert wird durch jenes x, welches $Rx = y_1$ löst.

In MATLAB werden überbestimmte Gleichungssysteme der Form (3.4) automatisch mit der QR-Zerlegung gelöst, wenn man den Backslash-Operator

```
x = A\b
```

benützt.

3.4.5 Aufgabe (Regressionsgerade)

1. Vorgegeben sei wieder eine Funktion $f(x)$. Gesucht ist nun die Gerade, d.h. das lineare Polynom

$$p(x) = a_1 + a_2 x$$

so, dass

$$p(x_i) \approx f(x_i), \quad i = 1, \ldots, n.$$

im Sinne der kleinsten Quadrate gilt.

$$\left\| \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} - \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{pmatrix} \right\|_2 = \text{minimal}$$

Verwenden Sie die Daten:

```
>> n=20;
>> x = [1:n]';
>> y = x + (2*rand(size(x))-1);
```

Die Systemmatrix hat als erste Spalte die Stützstellen x_i und als zweite Spalte 1. Natürlich könnte man wieder wie bei der Polynominterpolation die Funktion `vander` benützen und dann die richtigen Spalten aus der Matrix extrahieren. Hier ist es aber einfacher, die Systemmatrix direkt aufzustellen.

Die Lösung des überbestimmten Gleichungssystems ist ein Vektor mit zwei Komponenten. Werten Sie das Polynom an den Stützstellen x_i aus, direkt oder mit `polyval`. Wenn Sie diese Werte in einem Vektor `Y` speichern und den Befehl

```
plot(x,y,'ro',x,Y,'b')
```

eingeben, dann erhalten Sie (im wesentlichen) die Abbildung 3.4.

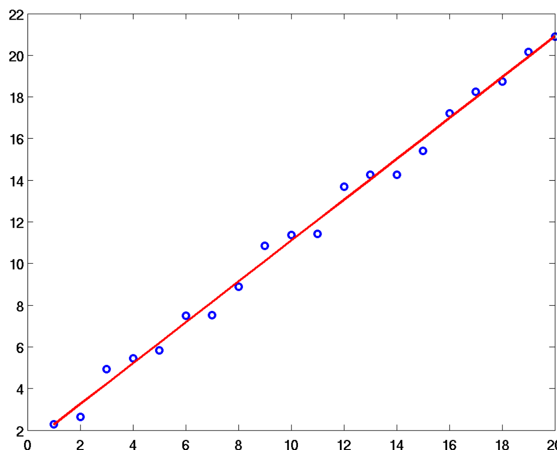

Abbildung 3.4: Regressionsgerade

3.4.6 Eigenwertzerlegung

Eine Zerlegung der Form

$$A = X \Lambda X^{-1} \tag{3.5}$$

heisst *Spektralzerlegung* (Eigenwertzerlegung) von A. (Eine solche existiert nicht immer!) In (3.5) ist $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$ eine Diagonalmatrix. Die Diagonalelemente λ_i heissen Eingenwerte, die Spalten $\mathbf{x}_i$ von $X = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ Eigenvektoren. Es gilt $A\mathbf{x}_i = \lambda \mathbf{x}_i$ for alle i.

```
>> A=[1 3 5 7;2 4 4 8; 3 1 2 3; 4 3 2 1]

A =

     1     3     5     7
     2     4     4     8
     3     1     2     3
     4     3     2     1

>> Lam=eig(A)

Lam =

   12.7448
   -4.6849
   -1.3752
    1.3153

>> [Q,L]=eig(A)

Q =

   -0.5497   -0.5826    0.3345   -0.2257
   -0.6502   -0.4848   -0.4410    0.7334
   -0.3282    0.0418   -0.6402   -0.6290
   -0.4092    0.6510    0.5328    0.1248

L =

   12.7448         0         0         0
         0   -4.6849         0         0
         0         0   -1.3752         0
         0         0         0    1.3153

>> norm(A*Q-Q*L)

ans =

   7.1768e-15
```

3.4.7 Anwendung: Matrixfunktionen

Sei

$$A = X\Lambda X^{-1}.$$

Dann ist

$$A^2 = X\Lambda X^{-1}X\Lambda X^{-1} = X\Lambda^2 X^{-1}$$

und allgemeiner

$$A^k = X\Lambda^k X^{-1} \quad \text{für natürliche } k.$$

Sei p ein *Polynom*. Dann ist

$$p(A) = Xp(\Lambda)X^{-1} = X \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} X^{-1}.$$

Allgemeiner kann man für eine Funktion f schreiben:

$$f(A) = Xf(\Lambda)X^{-1} = X \begin{bmatrix} f(\lambda_1) & & \\ & \ddots & \\ & & f(\lambda_n) \end{bmatrix} X^{-1}.$$

Die einzige Einschränkung an die Funktion f ist, dass sie an allen Stellen λ_i definiert ist.

Im Prinzip könnte man zu allen in Tabelle 3.3 aufgelisteten skalaren Funktionen eine entsprechende Matrixfunktion definieren (Algorithmus von Parlett basierend auf der Schurzerlegung [7, p.384]). Dies wurde in MATLAB ausser für den Logarithmus, die Exponential- und Wurzelfunktion nicht gemacht. Funktionen wie exp, log, cos, sin, cosh, sinh können mit durch `funm(a,@sin)`, etc. aufgerufen werden. `funm` kann auch für eigene Funktionen gebraucht werden: `funm(a,@my_fun)`.

```
>> a=[1 2 ;3 4]
a =
     1     2
     3     4
>> s=funm(a,@sin)
s =
   -0.4656   -0.1484
   -0.2226   -0.6882
>> c=funm(a,@cos)
c =
    0.8554   -0.1109
   -0.1663    0.6891
>> norm(s^2 + c^2 - eye(size(a)))
ans =
   6.9184e-16
>>
```

Die Eigenwertzerlegung in der angegebenen Form, d.h. mit diagonalem Λ existiert nicht immer. Für weitere Informationen siehe die Schurzerlegung: `help schur`.

3.4.8 Anwendung: Systeme von linearen gewöhnlichen Differentialgleichungen 1. Ordnung

Die lineare Differentialgleichung 1. Ordnung

$$y'(t) = a\, y(t), \qquad y(0) = y_0, \tag{3.6}$$

mit konstantem a hat die Lösung

$$y(t) = e^{at} = y_0 \exp(at).$$

Mit der Spektralzerlegung kann dieses Resultat einfach auf Systeme von linearen gewöhnlichen Differentialgleichungen 1. Ordnung mit konstanten Koeffizienten

$$\mathbf{y}'(t) = A\, \mathbf{y}(t), \qquad \mathbf{y}(0) = \mathbf{y}_0$$

übertragen werden, welches die Lösung

$$\mathbf{y}(t) = \mathbf{y}_0 e^{At} = \mathbf{y}_0 \exp(At)$$

hat.

Numerisch geht man so vor.

1. Berechne die Spektralzerlegung von A: $A = Q\Lambda Q^{-1}$. Die Differentialgleichung erhält dann die Form

$$\mathbf{z}'(t) = \Lambda\, \mathbf{z}(t), \quad \mathbf{z}(0) = \mathbf{z}_0; \qquad \mathbf{z}(t) = Q\mathbf{y}(t), \mathbf{z}_0 = Q\mathbf{y}_0.$$

2. Das n-dimensionale System zerfällt damit in n skalare Differentialgleichungen der Form (3.6), deren Lösung bekannt ist.

3. Die Lösung des ursprünglichen Problems erhält man durch Rücktransformation: $\mathtt{y}(t) = Q^{-1}\mathtt{z}(t)$.

3.4.9 Singulärwertzerlegung (SVD)

Für $A \in \mathbb{R}^{m\times n}$ existieren orthogonale Matrizen $U \in \mathbb{R}^{m\times m}$ und $V \in \mathbb{R}^{n\times n}$ so, dass

$$U^* A V = \Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_p), \qquad p = \min(m, n), \tag{3.7}$$

mit $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$. (Wenn A komplex ist, so sind U und V unitär.)

Die Zerlegung (3.7) heisst *Singulärwertzerlegung*.

```
A=[1 3 5 7;2 4 6 8; 1 1 1 1; 4 3 2 1;0 1 0 1]

A =

     1     3     5     7
     2     4     6     8
     1     1     1     1
     4     3     2     1
     0     1     0     1

>> rank(A)

ans =

     3

>> S=svd(A)

S =

   14.8861
    4.1961
    0.8919
    0.0000

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

U =

   -0.6102   -0.2899    0.0486    0.6913   -0.2517
   -0.7352   -0.1145    0.0558   -0.5855    0.3170
   -0.1249    0.1753    0.0071   -0.3668   -0.9050
   -0.2571    0.9338    0.0078    0.2116    0.1307
   -0.0738   -0.0120   -0.9972    0.0000   -0.0000

S =

   14.8861         0         0         0
         0    4.1961         0         0
         0         0    0.8919         0
         0         0         0    0.0000
         0         0         0         0
```

```
V =

   -0.2172    0.8083    0.2225    0.5000
   -0.3857    0.3901   -0.6701   -0.5000
   -0.5442   -0.0223    0.6733   -0.5000
   -0.7127   -0.4405   -0.2193    0.5000
```

3.4.10 Anwendung der SVD: Bild, Kern und Konditionszahl einer Matrix

Sei in der Singulärwertzerlegung (3.7) $\sigma_1 \geq \cdots \geq \sigma_r > \sigma_{r+1} = \cdots = \sigma_p = 0$. r ist der *Rang* der Matrix.

Wir zerlegen $U = [U_1, U_2]$ und $V = [V_1, V_2]$ so, dass U_1 und U_2 jeweils aus den ersten r Spalten von U resp. V besteht. Dann bilden die Spalten von V_2 eine Orthonormalbasis des Kerns (Nullraums) von A und U_1 eine Orthonormalbasis des Bilds von A. Die Spalten von U_2 bilden eine Orthonormalbasis des Kerns (Nullraums) von A^T und V_1 eine Orthonormalbasis des Bilds von A^T.

Die Matrix $A = U_1\Sigma_1 V_1^T$ ist eine ein-eindeutige Abbildung von $\mathcal{N}(A)^\perp$ in $\mathcal{R}(A)$. Diese Abbildung hat eine Inverse, die sog. Pseudoinverse: $A^+ = V_1\Sigma_1^{-1}U_1^T$.

Weitere wichtige Eigenschaften der SVD sind

- $\sigma_1 = \|A\|_2$ die *Spektralnorm* von A.
- Wenn A invertierbar ist, so ist $1/\sigma_n = \|A^{-1}\|_2$.
- Die *Konditionszahl* von A (bzgl. der Spektralnorm) ist

$$\kappa(A) = \|A\|_2 \cdot \|A^{-1}\|_2 = \sigma_1/\sigma_n.$$

- Die Konditionszahl zeigt an, wieviele Stellen man beim Lösen eines Gleichungssystems verliert.
- Beispiel:
  - Exakt: $\mathbf{y} = A^{-1}\mathbf{b}$.
  - Gerechnet $\mathbf{x} \approx \mathbf{y}$.
  - $\|\mathbf{x} - \mathbf{y}\| \approx \kappa(A)\varepsilon\|\mathbf{x}\|, \qquad \varepsilon \approx 10^{-16}$.
  - Stellenverlust ca. $\log_{10} \kappa(A)$.

Das folgende Zahlbeispiel stammt von R. Rannacher: Einführung in die Numerische Mathematik, Heidelberg 1999/2000.

```
>> format long
>> A=[1.2969,0.8648;0.2161,0.1441];
>> y=[2;-2]

y =

     2
    -2

>> b=A*y

b =

   0.86420000000000
   0.14400000000000
```

```
>> x=A\b

x =

   2.00000000480060
  -2.00000000719924

>> norm(y-x)

ans =

     8.653026991073135e-09

>> cond(A)

ans =

     2.497292668562500e+08

>> cond(A)*eps

ans =

     5.545103639731375e-08

>> norm(b-A*x)

ans =

     5.551115123125783e-17

>> norm(b-A*y)

ans =

     0
```

Kapitel 4

Konstruktionen zur Programmsteuerung

MATLAB hat verschiedenen Konstrukte, um den Programmablauf steuern zu können. In diesem Abschnitt behandeln wir `for`-, `while`-, `if`- und `switch-case`-Konstrukte. Diese Elemente der MATLAB-Programmiersprache kann man direkt im Befehlsfenster eingeben. Ihre Verwendung finden sie in M-Files, MATLAB-Funktionen, die im nächsten Abschnitt behandelt werden.

4.1 Das `if`-Statement

Die einfachste Form des `if`-Statements ist

```
>> if logischer Ausdruck
      Zuweisungen
   end
```

Die Zuweisungen werden ausgeführt, falls der logische Ausdruck wahr ist. Im folgenden Beispiel werden x und y vertauscht, falls x grösser als y ist

```
>> x=3;y=1;
>> if x > y
     tmp = y;
     y = x;
     x = tmp;
   end
>> [x y]

ans =

     1     3
```

Ein `if`-Statement kann auch auf einer einzigen Zeile geschrieben werden (falls es nicht zu lange ist). Kommata müssen aber an Stellen eingesetzt werden, an denen neue Zeilen anfangen und vorher keine Strichpunkte waren.

```
>> x=3;y=5;
>> if x > y, tmp = y; y = x; x = tmp; end
>> [x y]

ans =

     3     5
```

Zuweisungen, die nur ausgeführt werden sollen, wenn der logische Ausdruck falsch ist, können nach einem `else` plaziert werden.

```
e = exp(1);
if 2^e > e^2
  disp('2^e ist gr"osser')
else
  disp('e^2 ist gr"osser')
end
```

Mit `elsif` kann die Struktur noch verkompliziert werden.

```
if abs(i-j) > 1,
   t(i,j) = 0;
elseif i == j,
   t(i,j) = 2;
else
   t(i,j) = -1;
end
```

4.2 Die `switch-case`-Konstruktion

Wenn eine Variable eine feste Anzahl von Werten annehmen kann und für jeden von ihnen eine bestimmte Befehlsfolge ausgeführt werden soll, so bietet sich die `switch-case`-Konstruktion zur Implementation an. Diese hat die Form

```
>>  switch switch-Ausdruck (Skalar oder String)
       case case-Ausdruck
           Befehle
       case case-Ausdruck,
           Befehle
       ...
       otherwise,
           Befehle
    end
```

Beispiele:

```
>> switch grade   % grade soll eine integer Variable
                  % mit Werten zwischen 1 und 6 sein
   case {1,2}
      disp 'bad'
   case {3,4}
      disp 'good'
   case {5,6}
      disp 'very good'
   end
```

```
>> day_string = 'FrIday'

day_string =

FrIday

>> switch lower(day_string)
      case 'monday'
       num_day = 1;
      case 'tuesday'
```

```
      num_day = 2;
     case 'wednesday'
      num_day = 3;
       case 'thursday'
      num_day = 4;
       case 'friday'
      num_day = 5;
       case 'saturday'
      num_day = 6;
       case 'sunday'
      num_day = 7;
     otherwise
      num_day = NaN;
    end
>> num_day

num_day =

     5
```

4.3 Die `for`-Schleife

Die `for`-Schleife ist eine der nützlichsten Konstrukte in MATLAB. Man beachte aber die Bemerkung zur Effizienz in Abschnitt 4.5.

Die Form des `for`-Statements ist

```
>>  for Variable = Ausdruck
       Zuweisungen
    end
```

Sehr oft ist der Ausdruck ein Vektor der Form `i:s:j`, siehe Abschnitt 2.7.4. Die Zuweisungen werden ausgeführt, wobei die Variable einmal gleich jedem Element des Ausdruckes gesetzt wird. Die Summe der ersten 25 Terme der Harmonischen Reihe $1/i$ können folgendermassen berechnet werden.

```
>> s=0;
>> for i=1:25, s = s + 1/i; end
>> s

s =

    3.8160
```

Hier wird die Schleife also 25 Mal durchlaufen, wobei `i` in jedem Schleifendurchlauf den Wert ändert, angefangen bei $i = 1$ bis $i = 25$.

Der Ausdruck kann eine Matrix sein,

```
>> for i=A, i, end

i =

     1
     3


i =

     2
     4
```

Das nächste Beispiel definiert eine tridiagonale Matrix mit 2 auf der Diagonale und -1 auf den Nebendiagonalen.

```
>> n = 4;
>> for i=1:n,
      for j=1:n,
         if abs(i-j) > 1,
            t(i,j) = 0;
         elseif i == j,
            t(i,j) = 2;
         else
            t(i,j) = -1;
         end
      end
   end
>> t

t =

     2    -1     0     0
    -1     2    -1     0
     0    -1     2    -1
     0     0    -1     2
```

4.4 Die `while`-Schleife

Die allgemeine Form des `while`-Statements ist

>> `while` *Relation*
   Zuweisungen
`end`

Die Zuweisungen werden solange ausgeführt wie die Relation wahr ist.

```
>> e = 1; j = 0;
>> while 1+e>1, j = j + 1; e = e/2; end
>> format long
>> j = j - 1, e = 2*e

j =

    52

e =

     2.220446049250313e-16

>> eps     %  Zum Vergleich: die permanente Variable  eps

eps =

     2.220446049250313e-16
```

In Matlab ist eine Relation natürlich eine Matrix. Wenn diese Matrix mehr als ein Element hat, so werden die Statements im `while`-Körper genau dann ausgeführt, wenn jede einzelne Komponente der Matrix den Wert 'wahr' (d.h. 1) hat.

4.5 Eine Bemerkung zur Effizienz

In MATLAB werden die Befehle *interpretiert.* D.h., eine Zeile nach der anderen wird gelesen und ausgeführt. Dies geschieht z.B. auch in Schleifen. Es ist deshalb von Vorteil, zumindest bei zeitaufwendigeren Programmen vektorisiert zu progammieren.

```
>> x=[0:pi/100:10*pi];
>> y=zeros(size(x));
>> tic, for i=1:length(x), y(i)=sin(x(i)); end, toc

elapsed_time =

    0.0161

>> tic, y=sin(x); toc

elapsed_time =

   9.7100e-04

>> 0.0161 / 9.7100e-04

ans =

   16.5808

>>
```

4.6 Aufgaben

1. Konstruieren Sie unter Verwendung von `for`-Schleife und `if`-Verzweigung die Matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 6 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 12 \\ 0 & 0 & 0 & 13 & 14 \\ 0 & 0 & 0 & 0 & 15 \end{bmatrix}.$$

2. Berechnen Sie den Wert des Kettenbruchs

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+1}}}}}}}}}}$$

   für variable Länge des Bruchs. Um den Kettenbruch auszuwerten, müssen Sie ihn von unten nach oben auswerten. Das Resultat muss im Limes, d.h., wenn der Kettenbruch immer länger wird, den goldenen Schnitt $(1+\sqrt{5})/2$ geben.

3. *Collatz-Iteration.* Ausgehend von einer *natürlichen* Zahl n_1 berechnet man eine Folge von natürlichen Zahlen nach dem Gesetz $n_{k+1} = f(n_k)$ wobei

$$f(n) = \begin{cases} 3n+1, & \text{falls } n \text{ ungerade} \\ n/2, & \text{falls } n \text{ gerade} \end{cases}$$

   Collatz vermutete, dass diese Folge immer zum Wert 1 führt. Bewiesen ist diese Vermutung aber nicht. Schreiben Sie eine `while`-Schleife, die ausgehend von einem vorgegebenen n_0 eine Collatz-Folge berechnet. Wenn Sie wollen, können alle Zahlen bis zur Eins speichern und danach die Folge plotten [11]. Zur Bestimmung, ob eine Zahl gerade oder ungerade ist, können Sie (wenn Sie wollen) eine der Funktionen `rem` oder `mod` verwenden.

   Bemerkung: Sie können sich die Aufgabe erleichtern, wenn Sie zuerst Abschnitt 5.1 über Scriptfiles lesen.

4. *Tschebyscheff-Polynome.* Tschebyscheff-Polynome sind rekursiv definiert:

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x), \qquad T_0(x) = 1, \quad T_1(x) = x.$$

   Tschebischeff-Polynome oszillieren im Interval $[-1, 1]$ zwischen den Werten -1 und 1. Berechnen Sie einige der Polynome in diesem Interval uns plotten Sie sie, siehe Fig. 4.1. Benützen Sie dabei eine `for`-Schleife. Setzen Sie

```
x=linspace(-1,1,101)';
```

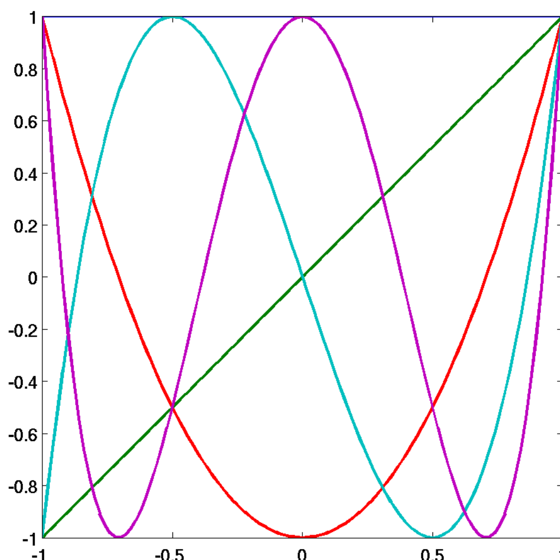

Abbildung 4.1: Tschebyscheff-Polynome $T_0(x)$ bis $T_5(x)$

Kapitel 5

M-Files

Obwohl man viele nützliche Arbeit im MATLAB-Befehlsfenster ausführen kann, wird man früher oder später M-Files schreiben wollen oder müssen. M-Files entsprechen Programmen, Funktionen, Subroutinen oder Prozeduren in anderen Programmiersprachen. M-Files bestehen zu einem grossen Teil aus einer Folge von MATLAB-Befehlen. (Sie heissen M-Files, da ihr Filetyp d.h. der letzte Teil ihres Filenamens '.m' ist.)

M-Files eröffnen eine Reihe von Möglichkeiten, die das Befehlsfenster nicht bietet:

- Experimentieren mit einem Algorithmus indem ein File editiert wird, anstatt eine lange Befehlssequenz wieder und wieder eintippen.
- Eine permanente Aufzeichnung eines Experiments machen.
- Der Aufbau von Dienstprogrammen zur späteren Benützung.
- Austausch von Programmen mit Kollegen. Viele M-Files sind von MATLAB-Enthusiasten geschrieben worden und ins Internet gestellt worden.

Es gibt zwei Arten von M-Files, *Scriptfiles* und *Funktionenfiles.*

5.1 Scriptfiles

Scriptfiles werden einfach durchlaufen und die darin enthaltenen Befehle so ausgeführt (interpretiert) wie wenn sie am Bildschirm eingetippt worden wären. Sie haben keine Eingabe oder Ausgabeparameter. Scriptfiles werden benützt, wenn man lange Befehlsfolgen z.B. zum Definieren von grossen Matrizen hat oder wenn man oft gemachte Befehlsfolgen nicht immer wieder eintippen will. Der Gebrauch von Scriptfiles ermöglicht es, Eingaben im wesentlichen mit dem Editor[1] anstatt im MATLAB-Befehlsfenster zu machen.

Der folgende MATLAB-Auszug zeigt zwei Beispiele von M-Files. Mit dem Befehl `what` kann man die M-Files im gegenwärtigen Directory auflisten lassen. Der Befehl `type` listet den Inhalt eines Files aus.

```
>> type init

A = [  7     3     4   -11    -9    -2 ;
      -6     4    -5     7     1    12 ;
      -1    -9     2     2     9     1 ;
      -8     0    -1     5     0     8 ;
      -4     3    -5     7     2    10 ;
       6     1     4   -11    -7    -1 ]
```

[1] MATLAB hat einen eingebauten Editor, den man via Fensterbalken oder dem Befehl `edit filename` oder nur `edit` aufrufen kann.

```
>> type init_demo

   %  Script-File fuer Matlab-Kurs

   disp(' Erzeugt 2 Zufallszahlenvektoren der Laenge n')

   n = input(' n = ? ');

   rand('state',0);
   x = rand(n,1);
   y = rand(n,1);

>> init_demo
 Erzeugt 2 Zufallszahlenvektoren der Laenge n

  n = ? 7
>> who

Your variables are:

n         x         y

>> [x,y]'

ans =

  0.9501  0.2311  0.6068  0.4860  0.8913  0.7621  0.4565
  0.0185  0.8214  0.4447  0.6154  0.7919  0.9218  0.7382
```

Bemerkung: Im Scriptfile `startup.m` im MATLAB-Heimverzeichnis kann man Befehle eingeben, die MATLAB am Anfang ausführen soll. Hier kann man z.B. angeben, in welchem Directory MATLAB starten soll, oder den Verzeichnis-Pfad erweitern (`addpath`).

5.2 Funktionen-Files

Funktionen-Files erkennt man daran, dass die erste Zeile des M-Files das Wort 'function' enthält. Funktionen sind M-Files mit Eingabe- und Ausgabeparameter. Der Name des M-Files und der Funktion sollten gleich sein. Funktionen arbeiten mit eigenem Speicherbereich, unabhängig vom Arbeitsspeicher, der vom MATLAB-Befehlsfenster sichtbar ist. Im File definierte Variablen sind *lokal*. Die Parameter werden *by address* übergeben. Lokale Variable wie Parameter gehen beim Rücksprung aus der Funktion verloren.

Als erstes Beispiel eines Funktionen-Files habe ich das obige Script-File `init_demo.m` in eine Funktionen-File `init1` umgeschrieben. Dadurch wird es viel einfacher zu gebrauchen.

```
function  [x,y] = init1(n)
%INIT1          [x,y] = INIT(n) definiert zwei
%               Zufallszahlenvektoren der Laenge n

%  Demo-File fuer Matlab-Kurs

rand('state',0);
x = rand(n,1);
y = rand(n,1);
```

Neben dem Stichwort `function` erscheinen auf der ersten Zeile des Files die Ausgabeparameter (`x`, `y`) und die Eingabeparameter. MATLAB-Funktionen können mit einer variablen Anzahl von Parametern aufgerufen werden.

```
>> norm(A)
ans =
   16.8481
>> norm(A,'inf')
ans =
    24
```

Wenn das zweite Argument in `norm` fehlt, berechnet die Funktion einen Defaultwert. Innerhalb einer Funktion stehen zwei Grössen, `nargin` und `nargout` zur Verfügung, die die Zahl der beim aktuellen Aufruf verwendeten Parameter angibt. Die Funktion `norm` braucht `nargin` aber nicht `nargout`, da sie immer nur einen Wert liefert.

Bemerkung: MATLAB sucht beim Eintippen nicht das `init1` der ersten Zeile des Funktionen-Files, sondern das File mit Name `init1.m`! Der auf der ersten Zeile angegebene Funktionenname ist unwesentlich! Wenn in MATLAB ein Name z.B. `xyz` eingegeben wird, so sucht der MATLAB-Interpreter

1. ob eine Variable `xyz` im Arbeitsspeicher existiert,
2. ob `xyz` eine eingebaute MATLAB-Funktion ist,
3. ob ein File namens `xyz.m` im gegenwärtigen Verzeichnis (Directory) existiert,
4. ob es ein File namens `xyz.m` im einem der in der Unix-Umgebungsvariable MATLABPATH eingetragenen Verzeichnisse gibt.

Ein eigenes Funktionen-File wird wie eine eingebaute Funktion behandelt.

Wird in MATLAB der Befehl `help init1` eingetippt, werden die im File abgespeicherte Kommentarzeilen bis zur ersten Nichtkommentarzeile auf den Bildschirm geschrieben.

```
>> help init1

 INIT1              [x,y] = INIT(n) definiert zwei
                    Zufallszahlenvektoren der Laenge n

>> init1(4)

ans =

    0.9501
    0.2311
    0.6068
    0.4860

>> [a,b]=init1(4)

a =

    0.9501
    0.2311
    0.6068
    0.4860

b =

    0.8913
    0.7621
    0.4565
    0.0185
```

Das zweite Beispiel zeigt ein M-File, welches $n!$ rekursive berechnet. Wenn die Eingabe nicht korrekt ist, wird die MATLAB-Funktion 'error' aufgerufen, die eine Fehlermeldung auf den Bildschirm schreibt und dann die Ausführung abbricht.

```
>> type fak

function y=fak(n)
%
%FAK     fak(n) berechnet die Fakultaet von  n.
%
%            fak(n) = n * fak(n-1),  fak(0) = 1

%      P. Arbenz   27.10.89

if n < 0 | fix(n) ~= n,
   error(['FAK ist nur fuer nicht-negative',...
          ' ganze Zahlen definiert!'])
end

if n <= 1,
   y = 1;
else
   y = n*fak(n-1);
end

>> fak(4)

ans =

    24

>> fak(4.5)
??? Error using ==> fak
FAK ist nur fuer nicht-negative ganze Zahlen definiert!
```

In diesem Beispiel sind die Variablen n und y lokal. Sie werden durch `who` nicht aufgelistet, wenn sie nicht schon vor dem Aufruf von `fak` definiert wurden. In letzterem Fall stimmen ihre Werte im Allg. nicht mit den in der Funktion zugewiesenen Werten überein.o

5.2.1 Aufgabe

1. Schreiben Sie eine Funktion, die die Fakultät von n mit einer `for`-Schleife berechnet, in der die Zahlen von 1 bis n einfach aufmultipliziert werden.

   Vergleichen Sie Ihre Funktion mit obiger rekursiven und messen Sie die Ausführungszeiten. Da Funktionsaufufe relativ teuer sind, sollte Ihre Funktion für grosse n klar schneller sein.

5.3 Arten von Funktionen

In MATLAB gibt es verschiedene Arten Funktionen, auch solche, die kein eigenes File haben.

5.3.1 Anonyme Funktionen

Eine anonyme Funktion besteht aus einem einzigen MATLAB-Ausdruck hat aber beliebig viele Ein- und Ausgabeparameter. Anonyme Funktionen können auf der MATLAB-Kommandozeile definiert werden. Sie erlauben, schnell einfache Funktioen zu definieren, ohne ein File zu editieren.

Die Syntax ist

```
f = @(arglist)expression
```

Der Befehl weiter unten kreiert eine anonyme Funktion, die das Quadrat einer Zahl berechnet. Wenn die Funktion aufgerufen wird, weist MATLAB der Variable `x` zu. Diese Variable wird dann in der Gleichung `x.^2` verwendet.

```
>> sqr = @(x) x.^2

sqr =

    @(x) x.^2

>> a = sqr(7)

a =

    49

>> clear i
>> sqr(i)

ans =

    -1

>> % rest: ganzzahliger Teiler und Rest
>> rest=@(x,y) [floor(x/y), x-y*floor(x/y)]

rest =

    @(x,y) [floor(x/y), x-y*floor(x/y)]

>> h=rest(8,3)

h =

     2     2

>> 8 - (h(1)*3 + h(2))

ans =

     0
```

5.3.2 Primäre und Subfunktionen

Alle Funktionen, die nicht anonym sind, müssen in M-Files definiert werden. Jedes M-File muss eine primäre Funktion enthalten, die auf der ersten Zeile des Files definiert ist. Weitere Subfunktionen können dieser primären Funktion folgen. Primäre Funktionen haben einen witeren Definitionsbereich (*scope*) als Subfunktionen. Primäre Funktionen können von ausserhalb des M-Files aufgerufen werden, z.B. von der MATLAB-Kommandozeile oder aus einer anderen Funktion. Subfunktionen sind nur in dem File sichtbar, in welchem sie definiert sind.

5.3.3 Globale Variable

Wenn mehr als eine Funktion auf eine einzige Variable Zugriff haben soll, so muss die Variable in allen fraglichen Funktionen als `global` deklariert werden. Wenn die Variable auch vom Basis-

Arbeitsspeicher zugegriffen werden soll, führt man das `global`-Statement auf der Kommandozeile aus. Eine Variable muss als `global` deklariert werden *bevor* sie das erste Mal gebraucht wird.

```
>> type myfun

function f = myfun(x)
%MYFUN  myfun(x) = 1/(A + (x-B)^2)
%       A, B are global Variables

  global A B
  f  = 1/(A + (x-B)^2);

>> global A B
>> A = 0.01; B=0.5;
>> fplot(@myfun,[0 1])
```

Da MATLAB Variablen, insbesondere Matrizen, lokal abspeichert, kann der Speicherbedarf bei rekursiven Funktionsaufrufen sehr gross werden. In solchen Fällen müssen Variablen als globale erklärt werden, um Speicherplatzüberlauf zu verhindern.

5.3.4 Funktionenfunktionen

Eine Klasse von Funktionen, sog. Funktionenfunktionen, arbeitet mit nicht-linearen Funktionen einer skalaren Variable. Eine Funktion arbeitet mit einer anderen. Dazu gehören

- Nullstellensuche
- Optimierung (Minimierung/Maximierung)
- Integration (Quadratur)
- Gewöhnliche Differentialgleichungen (ODE)

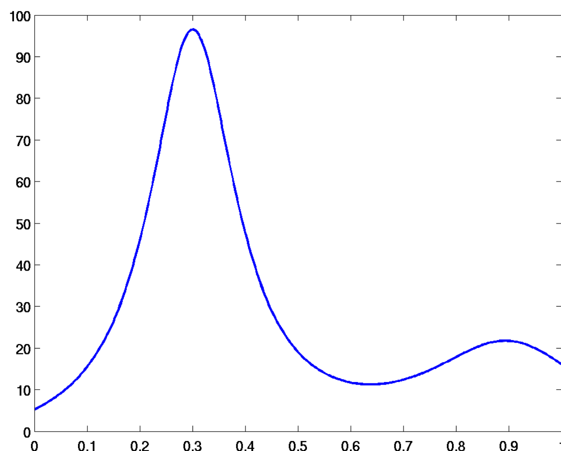

Abbildung 5.1: Graph der Funktion in `humps.m`

In MATLAB wird die nicht-lineare Funktion in einem M-File gespeichert. In der Funktion `humps` ist die Funktion programmiert

$$y(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6,$$

die ausgeprägte Maxima bei $x = 0.3$ und $x = 0.9$ hat, siehe Abb. 5.1. Aus dieser Abbildung sieht man, dass die Funktion bei $x = 0.6$ ein Minimum hat.

```
>> p = fminsearch(@humps,.5)

p =

    0.6370

>> humps(p)

ans =

   11.2528
```

Das bestimmte Integral

$$\int_0^1 \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6\,dx$$

nähert man numerisch an durch die eingebaute MATLAB-Funktion `quadl` an

```
>> Q = quadl(@humps,0,1)

Q =

   29.8583
```

Schliesslich, kann man eine Nullstelle bestimmen durch

```
>> z = fzero(@humps,.5)

z =

   -0.1316
```

Die Funktion hat keine Nullstelle im abgebildeten Interval. MATLAB findet aber eine links davon.

5.3.5 Funktionen mit variabler Zahl von Argumenten

Es gibt Situationen, in denen man die Zahl der Ein- oder Ausgabe-Parameter variabel haben will. Nehmen wir an, wir wollen die direkte Summe einer unbestimmten Anzahl von Matrizen bilden:

$$A_1 \oplus A_2 \oplus \cdots \oplus A_m = \begin{pmatrix} A_1 & O & \cdots & O \\ O & A_2 & \cdots & O \\ \vdots & \vdots & \ddots & \vdots \\ O & O & \cdots & A_m \end{pmatrix}$$

Wenn nur zwei Matrizen da wären, wäre die Lösung naheliegend.

```
function C = dirsum2(A,B)
%DIRSUM2 Direct sum of two matrices.
%
%     C = dirsum (A,B):   C = [A 0]
%                             [0 B]

% Peter Arbenz, Sep 8, 1989.

[n,m] = size(A);
[o,p] = size(B);
C = [A zeros(n,p); zeros(o,m) B];
```

Das Benützen einer variable Anzahl von Argumenten unterstützt MATLAB durch die Funktionen `varargin` und `varargout` an. Diese kann man sich als eindimensionale Arrays vorstellen, deren Elemente die Eingabe-/Ausgabeargumente sind. In MATLAB werden diese Arrays, deren Elemente verschiedene Typen haben können cell arrays genannt, siehe Abschnitt 2.13. Hier müssen wir nur wissen, dass die Elemente eines cell arrays mit geschweiften statt mit runden Klammern ausgewählt werden. Die Lösung unseres kleinen Problems könnte somit so aussehen.

```
function A = dirsum(A,varargin)
%DIRSUM Direct sum of matrices
%       C = dirsum(A1,A2,...,An)

% Peter Arbenz, May 30, 1997.

for k=1:length(varargin)
   [n,m] = size(A);
   [o,p] = size(varargin{k});
   A = [A zeros(n,p); zeros(o,m) varargin{k}];
end
```

Hier wird das erste Argument als `A` bezeichnet, alle weiteren werden mit der Funktion `varargin` behandelt.

```
>> a=[1 2 3]; b=ones(3,2);
>> dirsum(a,b,17)

ans =

     1     2     3     0     0     0
     0     0     0     1     1     0
     0     0     0     1     1     0
     0     0     0     1     1     0
     0     0     0     0     0    17
```

5.3.6 Aufgaben

1. Schreiben Sie eine Funktion, die die ersten n Tschebyscheff-Polynome an den Stützstellen gegeben durch den Vektor `x` auswertet, siehe Seite 46. Die erste Zeile des Funktionenfiles soll die Form

   ```
   function T = tscheby(x,n)
   ```

   haben.

2. *Wurzelberechnung.* Das Newton-Verfahren zur Berechnung der Wurzel einer positiven Zahl a ist gegeben durch

   $$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right).$$

   Schreiben Sie eine eigene Funktion

   ```
   [x, iter] = wurzel(a, tau),
   ```

   die $\sqrt{a}$ auf eine gewisse Genauigkeit τ berechnet:

   $$|x_{k+1} - x_k| \leq \tau.$$

   Wenn nur ein Eingabeparameter vorgegeben wird, so soll $\tau = \varepsilon =$ `eps` gesetzt werden.

   Sehen Sie auch einen Notausgang vor, wenn Folge $\{x_k\}$ nicht (genügend schnell) konvergiert.

3. Lösen Sie die gewöhnliche Differentialgleichung

$$\frac{dy}{dt} = -2ty(t) + 4t, \qquad y(0) = 0. \tag{5.1}$$

Die analytische Lösung ist

$$y(t) = ce^{-t^2} + 2, \qquad c = y(0) - 2.$$

Wir wollen die Funktion `ode23` zur Lösung dieser Differentialgleichung verwenden. `ode23` geht von einer Differentialgleichung der Form

$$y'(t) = f(t, y(t)) \qquad \text{plus Anfangsbedingungen}$$

aus. Deshalb schreiben Sie zunächst eine Funktion, z.B. `fun_ex1.m`, die *zwei* Eingabeargumente, t und $y(t)$, und ein Ausgabeargument, $f(t, y)$, hat. Danach lösen Sie die Gleichung (5.1) im Interval $[0, 3]$. `ode23` hat zwei Ausgabeparameter, Vektoren (`t`, `y`) der selben Länge, die die (approximativen) Werte der Lösung $y(t)$ and gewissen Stellen t enthält. Die Lösung kann so leicht geplottet werden.

4. *Räuber-Beute-Modell.* Im Räuber-Beute-Modell von Lotka–Volterra geht man von zwei Populationen aus, einer Beutepopulation y_1 und einer Räuberpopulation y_2. Wenn ein Räuber ein Beutetier trifft frisst er es mit einer bestimmten Wahrscheinlichkeit c. Wenn die Beutepopulation sich alleine überlassen wird, so wächst sie mit einer Rate a. (Das Nahrungsangebot ist unendlich gross.) Die Räuber haben als einzige Nahrung die Beutetiere. Wenn diese nicht mehr vorhanden sind, so sterben auch die Räuber (mit einer Rate b) aus.

   Hier betrachten wir das Beispiel von Hasen und Füchsen, deren Bestände durch $y_1(t)$ und $y_2(t)$ bezeichnet seien. Das Räuber-Beute-Modell von Lotka-Voltarra hat die Form

$$\begin{aligned} \frac{dy_1}{dt}(t) &= ay_1 - cy_1y_2, \qquad & y_1(0) &= y_1^{(0)}, \\ \frac{dy_2}{dt}(t) &= -by_2 + dy_1y_2, \qquad & y_2(0) &= y_2^{(0)}. \end{aligned} \tag{5.2}$$

   Alle Koeffizienten a, b, c und d sind positiv. Man rechen das Modell mit folgenden Parametern durch:

   - Zuwachsrate Hasen: $a = 0.08$,
   - Sterberate Füchse: $b = 0.2$,
   - Wahrscheinlichkeit, bei Treffen gefressen zu werden: $c = 0.002$,
   - Beutewahrscheinlichkeit der Füchse: $d = 0.0004$,
   - Anfangsbedingungen: Anfangsbestand Hasen 500, Anfangsbestand Füchse 20.

   Die Simulation ergibt eine periodische Schwingung, wobei die Maxima der Räubertiere jeweils eine kurze Weile nach den Maxima der Beutetiere auftreten.

   Verwenden Sie `ode34` und lösen Sie die Differentialgleichung im Interval $[0, 200]$. Das M-File, das Sie schreiben müssen, sieht wie in der vorigen Aufgabe aus, hat aber einen 2-komponentigen Vektor als Ausgabeargument. Zur Darstellung der beiden Lösungskomponenten können Sie den Befehl `plotyy` verwenden, vgl. Abschnitt 7.4.

5. Die nicht-lineare gewöhnliche Differentialgleichung dritter Ordnung

$$f'''(t) + f''(t)f(t) = 0, \qquad f(0) = f'(0) = 0, \lim_{t\to\infty} f'(t) = 1 \tag{5.3}$$

   hat keine analytische Lösung. Da MATLABs ODE-Löser Anfangswertprobleme erster Ordnung lösen, müssen wir zunächst (5.3) in ein solches umschreiben. Wir setzen

$$\begin{aligned} y_1(t) &= f(t) \qquad & y_3(t) &= \frac{dy_2(t)}{dt} = \frac{d^2y_1(t)}{dt^2} = f''(t), \\ y_2(t) &= \frac{dy_1(t)}{dt} = f'(t), \qquad & \frac{dy_3(t)}{dt} &= f'''(t). \end{aligned}$$

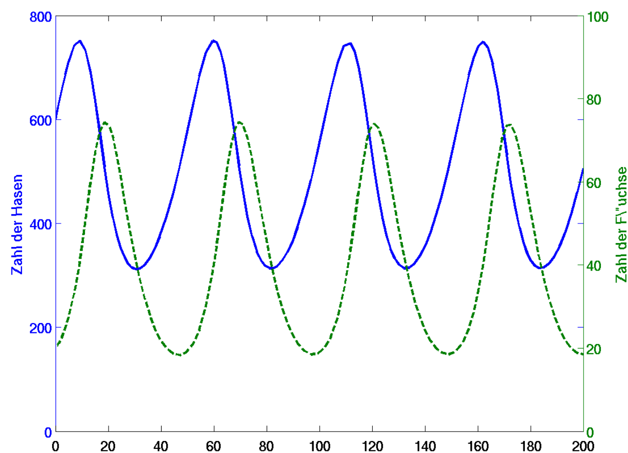

Abbildung 5.2: Simulation eines Räuber-Beute-Modell

Somit erhält die ursprüngliche gewöhnliche Differentialgleichung dritter Ordnung die Form

$$\frac{dy_1(t)}{dt} = y_2(t), \qquad \frac{dy_2(t)}{dt} = y_3(t), \qquad \frac{dy_3(t)}{dt} = -y_1(t)y_3(t).$$

Die Anfangsbedingungen für y_1 und y_2 sind bekannt: $y_1(0) = 0$ und $y_2(0) = 0$. Wir haben keine Anfangsbedingung für y_3. Wir wissen aber, dass $y_2(t)$ mit $t \to \infty$ gegen 1 konvergiert. Deshalb kann die Anfangsbedingung für y_3 verwendet werden, um $y_2(\infty) = 1$ zu erhalten. Durch versuchen erhält man $y_3(0) \approx 0.469599$. Die Lösung sieht wie in Abb. 5.3 aus.

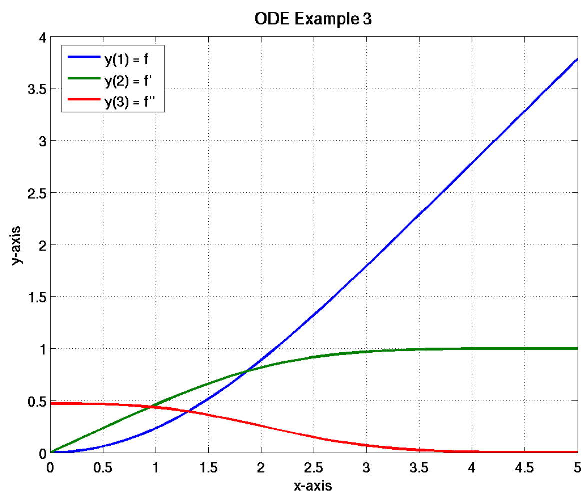

Abbildung 5.3: ODE dritter Ordnung

6. Versuchen Sie die Konstante in der Anfangsbedingung für y_3 zu bestimmen. Man muss dabei nicht bis $t = \infty$ gehen; $t = 6$ reicht. Versuchen Sie also mit `fzero` eine Nullstelle der Funktion

$$g(a) = y_2(6, a) - 1 = 0$$

zu berechnen. Hier bedeutet $y_3(6, a)$ der Funktionswert von y_3 an der Stelle $t = 6$ wenn die Anfangsbedingung $y_3(0) = a$ gewählt wurde.

Kapitel 6

Der MATLAB-Editor/Debugger

M-Files können mit irgend einem Editor bearbeitet werden. Einige können aber Programmstrukturen (Schleifen, Text, Kommentar) anzeigen und damit schon beim Eintippen helfen, Syntax-Fehler zu vermeiden. Der MATLAB-Editor kann aber bequemer im Zusammenspiel mit dem MATLAB-Debugger gebraucht werden. Wir wollen dies an dem einfachen Beispiel der Berechnung der Wurzel einer positiven Zahl zeigen, vgl. Übungsaufgabe in Abschnitt 5.3.6. Die Newton-Iteration zur Berechnung von $\sqrt{a}$, $a > 0$, d.h. zur Berechnung der positiven Nullstelle von $f(x) = x^2 - a$,

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{a}{x_k}\right)$$

soll mit dem Wert $x_0 = 1$ gestartet werden. Eine erste Implementation könnte folgendermassen aussehen.

```
function [x] = wurzel(a);
%WURZEL  x = wurzel(a) berechnet die Quadratwurzel von a
%        mit dem Newtonverfahren

x = 1;
while x^2 - a ~= 0,
    x = (x + a/x)/2;
end
```

Diese Funktion kommt aber im allgemeinen nicht zurück, d.h., die Abbruchbedingung der while-Schleife wird nie erfüllt.

Um herauszufinden, wo das Problem liegt untersuchen wir die Werte, die die Variable `x` in der while-Schleife annimmt. Wir benützen die Möglichkeit, in MATLAB sog. *Break points* zu setzen. Im MATLAB-Editor kann man auf das Minus-Zeichen vor Zeile 7 klicken, um einen Break point zu setzen, vgl. Abb. 6.1 Breakpoints können auch vom Kommandofenster gesetzt werden durch

```
>> dbstop in wurzel at 7
```

Ein Breakpoint kann wieder aufgelöst werden durch

```
>> dbclear in wurzel at 7
```

Im Editor-Fenster kann ein Breakpoint durch Klicken auf den entsprechenden roten Punkt aufgelöst werden.

Wenn MATLAB den Break point erreicht, hält das Program an, und MATLAB schaltet in den "debug mode" um. Der Prompt hat dann die Form `K>>`. `K` steht für `keyboard`[1]. Irgend ein MATLAB-Befehl ist dann zulässig. Insbesondere kann der Wert der Variable `x` angezeigt werden. Um in der Rechnung weiter zu fahren, gibt man `dbcont` oder `dbstep` oder `return` ein. Mit `dbquit` verlässt

[1] Der Befehl `keyboard` kann direkt in ein M-File geschrieben werden, um MATLAB zu veranlassen an der entsprechenden Stelle zu halten.

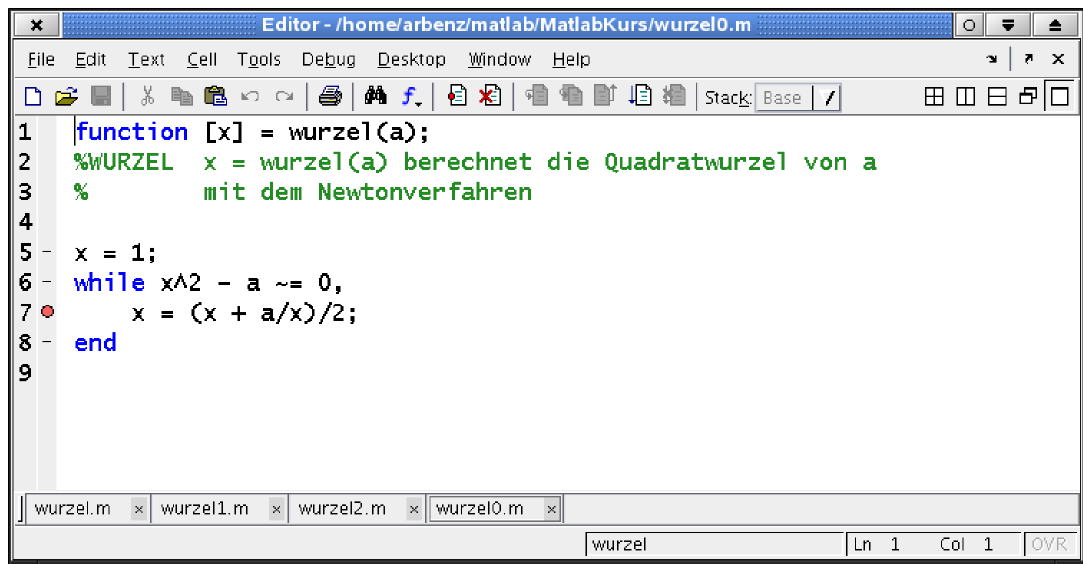

Abbildung 6.1: Anzeige eines Break points vor Zeile 7 im MATLAB-Editor

man den Debugger. Mit `dbup` kann man den Arbeitsspeicher der aufrufenden Funktion einsehen, mit `dbdown` kommt man in den Speicherbereich der nächsttieferen Funktion zurück.

In unserem Beispiel (mit $a = 3$) zeigt die Analyse, dass die x_k konvergieren, dass $x_k^2 - a$ aber nicht Null wird. Deshalb ändern wir die Abbruchbedingung der while-Schleife so, dass aufeinanderfolgende Werte von `x` verglichen werden:

```
function [x] = wurzel(a);
%WURZEL  x = wurzel(a) berechnet die Quadratwurzel von a
%        mit dem Newtonverfahren

x = 1; xalt = 0;
while x - xalt > 0,
    xalt = x;
    x = (x + a/x)/2;
end
```

Diese Iteration kann verfrüht abbrechen, wenn `xalt` grösser als `x` ist. Dies sieht man leicht ein, wenn man einen Breakpoint auf Zeile 8 (`x = (x + a/x)/2`) setzt und die Werte von `xalt` und `x` anzeigen lässt. Deshalb ändern wir das Programm so ab, dass der Abstand zwischen `x` und `xalt` berechnet wird:

```
function [x] = wurzel(a);
%WURZEL  x = wurzel(a) berechnet die Quadratwurzel von a
%        mit dem Newtonverfahren

x = 1; xalt = inf; tau = 10*eps;
while abs(x - xalt) > tau,
    xalt = x;
    x = (x + a/x)/2;
end
```

Eine Lösung von Aufgabe 2 in Abschnitt 5.3.6 schliesslich ist gegeben durch

```
function [x,n] = wurzel(a, tau);
%WURZEL  x = wurzel(a) berechnet die Quadratwurzel von a
%        mit dem Newtonverfahren
```

```
x = 1; xalt = inf; n = 0;
if nargin == 1, tau = eps; end
while abs(x - xalt) > tau,
    xalt = x;
    x = (x + a/x)/2;
    n = n+1;
end
```

Hier wird auch noch die Zahl der Iterationsschritte bis zur Konvergenz gezählt. Zudem besteht die Möglichkeit die Genauigkeit der Abbruchkriteriums zu variieren. Man beachte die Verwendung der MATLAB-internen Funktion `nargin`, die die Zahl der Eingabeparameter angibt, die an die Funktion übergeben wurden. Fehlt der zweite Parameter `tau`, so wird er gleich der Maschinengenauigkeit `eps` gesetzt.

Kapitel 7

Graphik in Matlab

Matlab hat Möglichkeiten Linien und Flächen graphisch darzustellen. Es ist auch möglich, graphische Benützeroberflächen oder 'Filme' herzustellen. Für eine eingehende Behandlung sei auf [12] verwiesen.

In dieser Einführung wollen wir uns auf das Wesentliche beschränken. Es soll nicht alle möglichen graphischen Darstellungen behandelt werden. Es sollte aber verständlich werden, wie das Graphiksystem aufgebaut ist.

7.1 Darstellung von Linien

Matlab erlaubt das Plotten von Linien in 2 und 3 DimensionenEine erste Übersicht erhält man, wenn man mit `demo` im Befehlsfenster das Matlab-Demos-fenster eröffnet und dort unter 'MATLAB Visualization' die '2-D Plots' laufen lässt.

Linien in zwei Dimensionen kann man auf verschiedene Arten darstellen. Hier einige Beispiele.

```
>> x=[1:.5:4]';
>> y1=5*x; y2=8*x.^2;
>> plot(x,y1)
>> plot(x,[y1 y2])
>> plot(x,y1,x,y2)
>> bar(x,[y1 y2])
>> stairs(x,[y1 y2])
>> errorbar(x,y2,y1/5)
```

Der erste Plot-Befehl, hat zwei Vektoren der gleichen Länge, x und y, als Eingabeparameter. Der erste Vektor wird für die Abszisse (x-Achse) der zweite für die Ordinate (y-Achse). Dabei werden die vorgegebenen Koordinatenpaare interpoliert.

Eine Legende kann beigefügt werden:

```
>> plot(x,y1,x,y2)
>> legend('erste Linie','zweite Linie')
```

Die Legende kann mit der Maus verschoben werden (linke Maustaste drücken) und mit `legend off` wieder entfernt werden. Der Befehl `legend` hat auch einen Parameter `'Position'`, mit dem man steuern kann, wohin die Legende platziert wird, siehe `help legend`. Ähnlich in drei Dimensionen

```
>> z=[0:.1:20]';
>> x=sin(z);
>> y=cos(z);
>> plot3(x,y,z)
```

Sowohl Farbe wie Art der Darstellung der einzelnen Linien kann bestimmt werden, indem nach den x-, y- und (eventuell) z-Werten eine entsprechende Zeichenkette angefügt wird.

```
>> plot3(x,y,z,'g')
>> plot3(x,y,z,'g:')
>> plot3(x,y,z,'rv')
>> hold on                   % Plot wird nun ueberschrieben
>> plot3(x,y,z,'g')
>> hold off
>> plot3(x,y,z,'rv',x,y,z,'g')
```

Einige mögliche Werte können folgender Zusammenstellung entnommen werden.

`y`	yellow	`.`	point	`>`	triangle (right)
`m`	magenta	`o`	circle	`-`	solid
`c`	cyan	`x`	x-mark	`:`	dotted
`r`	red	`+`	plus	`-.`	dashdot
`g`	green	`*`	star	`--`	dashed
`b`	blue	`v`	triangle (down)		
`w`	white	`^`	triangle (up)		
`k`	black	`<`	triangle (left)		

Eine Übersicht über alle Möglichkeiten der Kurvendarstellung findet man unter `doc plot`. Es

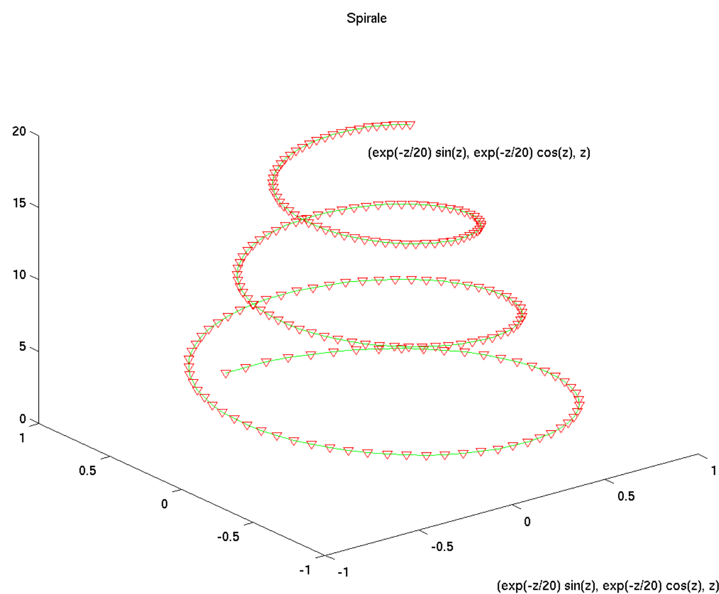

Abbildung 7.1: Beispiel eines Linien-Plot

können leicht Beschriftungen angebracht werden.

```
>> title('Spirale')
>> xlabel(' (sin(z), cos(z), z) ' )
>> text(0,0,22,' (sin(z), cos(z), z) ' )
```

Am Ende dieser Befehlssequenz hat das Graphikfenster den in Figur 7.1 gezeigten Inhalt.

7.1.1 Aufgaben

1. Führen Sie den Befehl `plot(fft(eye(17)))` aus. Verschönern Sie die Darstellung wieder mit dem Befehl `axis`: Beide Achsen sollen gleich lang sein, da die gezeichneten und durch Linien verbundenen Punkte auf dem Einheitskreis liegen. Stellen Sie die Achsen überhaupt ab.

2. Plotten Sie die Funktion
$$\frac{1}{(x-1)^2} + \frac{3}{(x-2)^2}$$
auf dem Intervall $[0, 3]$. Verwenden Sie die MATLAB-Funktion `linspace`, um die x-Koordinaten zu definieren.

   Da die Funktion Pole im vorgegebenen Interval hat, wählt MATLAB einen ungünstigen Bereich für die y-Achse. Verwenden Sie den Befehl `axis` oder `ylim`, um die obere Grenzen für die y-Achse auf 50 zu setzen.

   Verwenden Sie den Graphik-Editor, um die Achsen-Beschriftung grösser und die Linie dicker zu machen. Beschriften Sie die Achsen.

7.2 Das Matlab-Graphiksystem

`plot`, `bar`, etc., sind 'high-level' MATLAB-Befehle. MATLAB hat aber auch low-level Funktionen, welche es erlauben, Linien, Flächen und andere Graphik-Objekte zu kreieren oder zu ändern. Dieses System heisst *Handle Graphics*. Seine Bausteine sind hierarchisch strukturierte Graphik-Objekte.

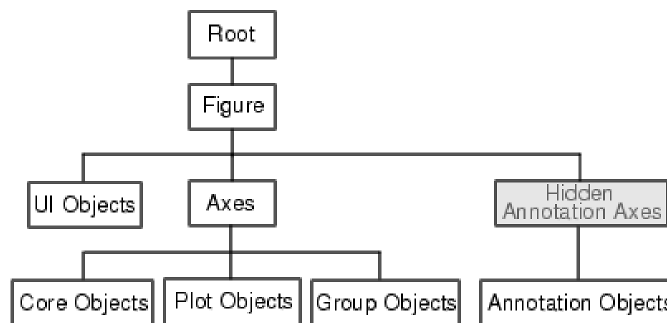

Abbildung 7.2: Objekt Hierarchie [15]

- An der Wurzel des Baums ist der Bildschirm (*root-Objekt*).
- *Figure-Objekte* sind die individuellen Fenster. Sie sind Kinder von `root`. Alle anderen Graphik-Objekte stammen von Figure-Objekten ab. Alle Objekte kreierenden Funktionen wie z.B. die höheren Graphikbefehle eröffnen eine `figure`, falls noch keine existiert. Ein Figure-Objekt kann auch mit `figure` gestartet werden.
- *Axes-Objekte* definieren Bereiche in einem `figure`-Fenster und orientieren ihre Kinder im Gebiet. Axes-Objekte sind Kinder von Figure-Objekten und Eltern von Lines-, Surfaces-, Text- und anderen Objekten.

Wenn Sie den Graphik-Editor aufrufen (`Edit` im Menu-Balken des Graphik-Fensters), dann können Sie die wichtigsten Eigenschaften der verschiedenen Objekte leicht ändern. Wenn Sie im Property Editor auf `Inspector` klicken, erhalten Sie die gesamte Liste der Eigenschaften eines Objekts. Zu beachten ist, dass diese Änderungen verloren gehen, sobald Sie einen neuen `plot`-Befehl von der Kommandozeile absetzten.

Jedes Graphik-Objekt kann mit einem sog. Handle auch von der Kommandozeile oder von einem M-File zugegriffen werden. Der Handle wird bei der Erzeugung des Objekts definiert. Befehle wie `plot`, `title`, `xlabel`, und ähnliche habe als Ausgabewerte einen oder mehrere (`plot`) handles. Wir wollen zunächst mit dem Befehl `gcf` (get current figure) das gegenwärtige (einzige) Fenster ergreifen:

```
>> gcf
>> figure
>> gcf
```

```
>> close(2)
>> gcf
```

Mit dem Befehl `gca` (get handle to current axis) können wir die gegenwärtigen Koordinatenaxen mit Zubehör (z.B. Linien) ergreifen.

```
>> gca
```

Wir sehen uns nun die Eigenschaften dieses Objekts an:

```
>> get(gca)
```

Achtung: Der Output ist lang! Diese Information kann man aber auch via 'property editor' im File-Menu auf dem Rahmen der Figur erhalten.

Wir wollen uns einige der (alphabetisch angeordneten) Attribute genauer ansehen. Einige Attribute enthalten Werte, andere sind wiederum handles.

```
>> th = get(gca,'title')
>> get(th)
```

ordnet `th` den handle zum Title zu. Mit dem Befehl `set` können Werte von Attributen geändert werden. Z.B.

```
>> set(th,'String','S P I R A L E')
```

Die wesentlichen Attribute von axis, z.B. `XTick`, `XTickLabel`, sind selbsterklärend:

```
>> get(gca,'XTick')
>> get(gca,'XTickLabel')
```

Wir können die Figur mit etwas unsinnigem beschriften:

```
>> set(gca,'XTickLabel',['a';'b';'c';'d';'e'])
```

Ein realistischeres Beispiel betrifft die Beschriftung der Achsen, Liniendicken bei Plots u.ä. Die Befehle

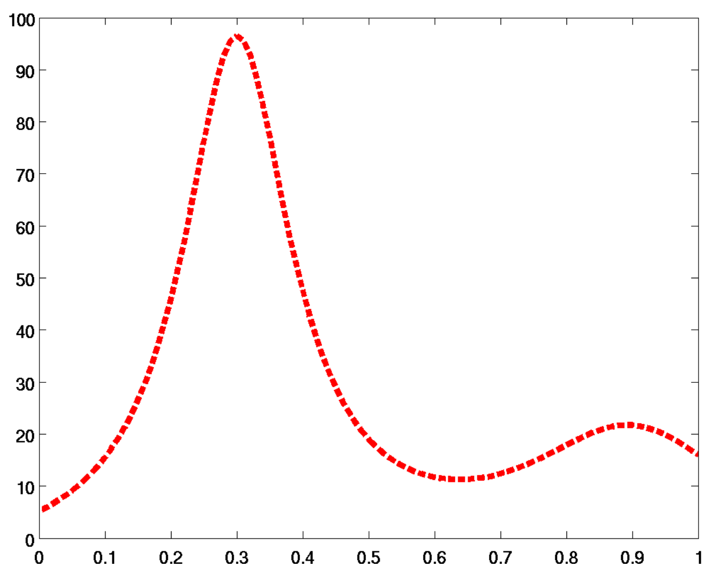

Abbildung 7.3: Plot der MATLAB-Funktion `humps` mit speziell gewählten Parametern für die Achsenbeschriftung und die Darstellung der Funktion

```
>> hp = plot(x,humps(x))

hp =

  155.0164

>> set(hp,'Linewidth',4,'Color','red','LineStyle','--')
>> set(gca,'FontSize',12,'FontWeight','bold')
```

ergeben den Plot in Abb. 7.3. Die ersten beiden Befehle können zusammengefasst werden in

```
>> plot(x,humps(x),'Linewidth',4,'Color','red','LineStyle','--')
```

7.2.1 Aufgaben

1. Führen Sie zunächst folgende Befehle aus

   ```
   z=[0:.1:20]';
   x=sin(z);
   y=cos(z);
   h=plot3(x,y,z,'rv',x,y,z,'g')
   ```

   Dann machen sie aus den Dreiecken der ersten Kurve Kreise und wechseln die Farbe der zweiten Kurve zu blau.

2. Führen Sie das folgende script aus.

   ```
   x = [1:.5:16]';
   y = x.^2+2*(rand(size(x)).*x-.5);
   plot(x,y,'o')
   ```

   (a) Berechnen Sie via Methode der kleinsten Quadrate das beste quadratische Polynom durch die Punkte. Zeichnen Sie beide Kurven uebereinander und beschriften Sie sie (`legend`).

   (b) Das gleiche in doppelt-logarithmischer Skala. Dabei sollen die beiden Achsen mit 1,2,4,8,16 beschriftet werden. Hier kann man den Befehl `loglog` verwenden.

7.3 Mehrere Plots in einer Figur

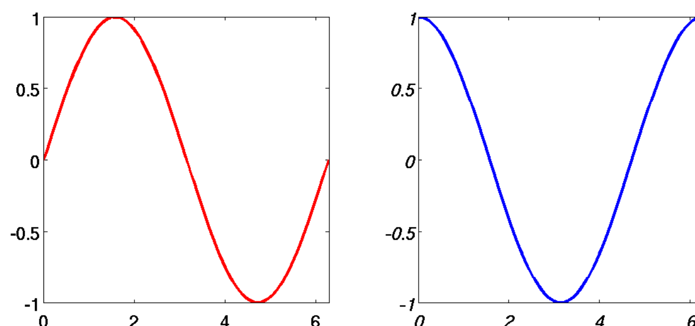

Abbildung 7.4: Beispiel zu `subplots`

Der MATLAB-Befehl `subplot` ermöglicht es, eine gewisse Zahl von Plots in einer einzigen Figur darzustellen. Wenn `subplot(mnp)` resp. `subplot(m,n,p)` eigetippt wird, so wird der p-te Plot in einem m-mal-n – Array von Plots angewählt. Nachfolgend können die graphischen Befehle normal eingegeben werden. Die Befehlsfolge

```
>> subplot(1,2,1), plot(x,sin(x),'Linewidth',2,'Color','red')
>> axis([0,2*pi,-1,1]), axis square
>> set(gca,'FontSize',12,'FontWeight','bold')
>> subplot(1,2,2), plot(x,cos(x),'Linewidth',2,'Color','blue')
>> axis([0,2*pi,-1,1]), axis square
>> set(gca,'FontSize',12,'FontWeight','bold','Fontangle','Oblique')
```

führt zum Bild in Abb. 7.4. Ein weiteres Beispiel ist

```
>> subplot(2,2,1), plot(x,sin(x),'Color','red')
>> axis([0,2*pi,-1,1]), axis square
>> subplot(2,2,2), plot(x,cos(x),'Linewidth',2,'Color','blue')
>> axis([0,2*pi,-1,1]), axis square
>> subplot(2,2,3:4) fplot(@(x)[sin(x),cos(x)],[0,4*pi])
```

was zum Bild in Abb. 7.5 führt.

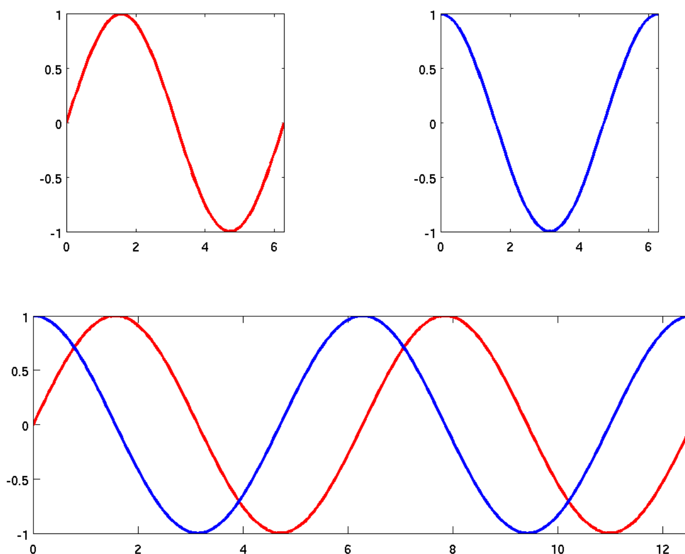

Abbildung 7.5: Zweites Beispiel zu `subplots`

7.4 Plots mit zwei Skalen

Die Funktion `plotyy` erlaubt es, zwei Funktionen mit individuellen Beschriftungen im selben Fenster darzustellen. Dies wird einfach erreicht mit der Befehlsfolge

```
x = 0:.1:10;
y1 = exp(-x).*sin(x);
y2 = exp(x);
plotyy(x,y1,x,y2)
```

Sollen Eigenschaften der Achsen und Linien geändert werden, so muss man sich Griffe (*handles*) zurückgeben lassen:

```
[ax, h1, h2] = plotyy(x,y1,x,y2);
```

Hier ist `ax` ein Vektor mit zwei Elementen, die auf die beiden Achsen zeigen. `h1` und `h2` sind Handles für die beiden Funktionen. Wir wollen Achsen sowie die Linien fett darstellen, sowie die (beiden) y-Achsen (fett) beschriften. Dies erreicht man mit

```
set(h1,'LineWidth',2)
set(h2,'LineWidth',2)
set(ax(1),'FontWeight','Bold','LineWidth',2)
set(ax(2),'FontWeight','Bold','LineWidth',2)
set(get(ax(1),'Ylabel'),'String','exp(-x)*sin(x)','FontWeight','Bold')
set(get(ax(2),'Ylabel'),'String','exp(x)','FontWeight','Bold')
```

Das Resultat ist in Fig. 7.6 abgebildet.

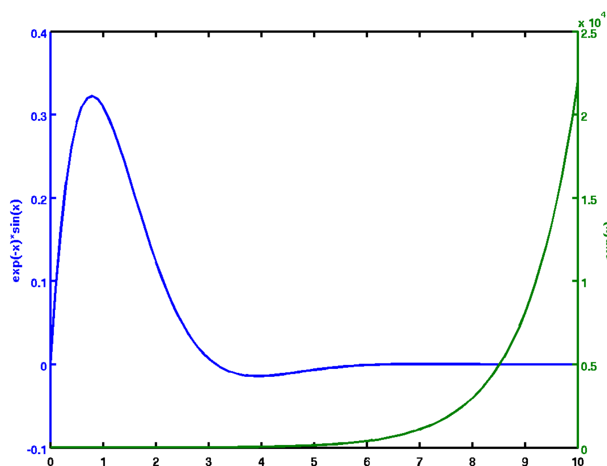

Abbildung 7.6: Gleichzeitige Darstellung der Funktionen $y_1 = \exp(-x)\sin(x)$ und $y_2 = \exp x$ mit `plotyy`

Die beiden letzten Befehle sind so zu verstehen: Beide Achsen haben eine y-Achse. Die Beschriftung ist in einem Unter-Objekt `Ylabel` der Achsen-Objekte abgespeichert. Das Unter-Objekt `Ylabel` hat viele Eigenschaften, unter anderem seinen textuellen Inhalt (Eigenschaft `String`) und zum Font, der zu wählen ist (`FontSize`, `FontWeight`, etc. Alle Eigenschaften können mit dem Befehl `get(get(ax(1),'Ylabel'))` abgefragt werden.

7.5 Darstellung von Flächen

Matlab kennt verschiedene Möglichkeiten, um 3D Objekte graphisch darzustellen. Den Befehl `plot3` haben wir schon kennen gelernt.

Mir der Funktion `mesh` können 3D-Maschenflächen geplottet werden. Zu diesem Zweck ist es von Vorteil, zunächst eine weitere Funktion `meshgrid` einzuführen. Diese Funktion generiert aus zwei Vektoren x (mit n Elementen) und y (mit m Elementen) ein Rechtecksgitter mit $n \times m$ Gitterpunkten.

```
>> x = [0 1 2];
>> y = [10 12 14];
>> [xi yi] = meshgrid(x,y)

xi =

     0     1     2
```

```
    0     1     2
    0     1     2


yi =

   10    10    10
   12    12    12
   14    14    14
```

Der Befehl `meshgrid` angewandt auf die beiden Arrays x und y erzeugen zwei Matrizen, in welchen die x- und y-Werte repliziert sind. So erhält man die Koordinaten aller Gitterpunkte gebildet mit den x_i und y_j.

Wir können jetzt z.B. die Sattelfläche $z = x^2 - y^2$ über dem Quadrat $[-1, 1] \times [-1, 1]$ zeichnen.

```
x = -1:0.05:1;
y = x;
[xi, yi] = meshgrid(x,y);
zi = yi.^2 - xi.^2;
mesh(xi, yi, zi)
axis off
```

Mit

```
meshc(xi, yi, zi)
axis off
```

erhält man dasselbe noch mit einem Kontour-Plot.

Die Maschenlinien werden als Flächen dargestellt, wenn man `mesh(c)` durch `surf(c)` ersetzt. Die Figur 7.7 ist erhalten worden durch die Befehle

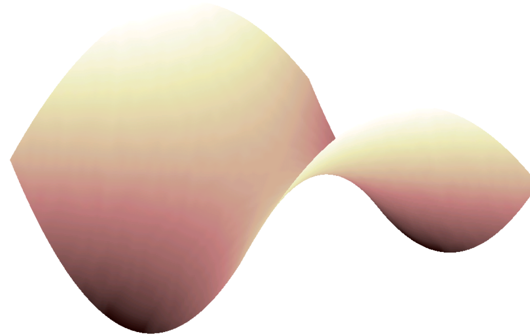

Abbildung 7.7: Die Sattelfläche dargestellt mit `surf`

```
x = -1:.05:1;  y = x;
[xi,yi] = meshgrid(x,y);
zi = yi.^2 - xi.^2;
surf(xi, yi, zi)
axis off
colormap pink
shading interp     % Interpolated shading
```

Der Befehl `colormap` wird verwendet um der Fläche eine bestimmte Farbtönung zu geben. MATLAB stellt folgende Farbpaletten zur Verfügung:

hsv	Hue-saturation-value color map (default)
hot	Black-red-yellow-white color map
gray	Linear gray-scale color map
bone	Gray-scale with tinge of blue color map
copper	Linear copper-tone color map
pink	Pastel shades of pink color map
white	All white color map
flag	Alternating red, white, blue, and black color map
lines	Color map with the line colors
colorcube	Enhanced color-cube color map
vga	Windows colormap for 16 colors
jet	Variant of HSV
prism	Prism color map
cool	Shades of cyan and magenta color map
autumn	Shades of red and yellow color map
spring	Shades of magenta and yellow color map
winter	Shades of blue and green color map
summer	Shades of green and yellow color map

Eine Farbpalette ist eine 3-spaltige Matrix mit Elementen zwischen 0 und 1. Die drei Werte einer Zeile geben die Intensität der Farben Rot, Grün und Blau an.

Sei `m` die Länge der Palette und seien `cmin` und `cmax` zwei vorgegebene Zahlen. Wenn z.B. eine Matrix mit `pcolor` (pseudocolor (checkerboard) plot) graphisch dargestellt werden soll, so wird ein Matrixelement mit Wert `c` die Farbe der Zeile `ind` der Palette zugeordnet, wobei

$$\texttt{ind} = \begin{cases} \texttt{fix((c-cmin)/(cmax-cmin)*m)+1} & \text{falls } \texttt{cmin} \leq \texttt{c < cmax} \\ \texttt{m} & \texttt{c == cmax} \\ \text{unsichtbar} & \texttt{c<cmin | c>cmax | c==NaN} \end{cases}$$

Mit dem Befehl `view` könnte man den Blickwinkel ändern.

Isolinien (Konturen) erhält man mit `contour` oder 'gefüllt' mit `contourf`.

```
x = -1:.05:1;   y = x;
[xi,yi] = meshgrid(x,y);
zi = yi.^2 - xi.^2;
contourf(zi), hold on,
shading flat                % flat = piecewise constant
[c,h] = contour(zi,'k-');
clabel(c,h)                 % adds height labels to
                            % the current contour plot
title('The level curves of z = y^2 - x^2.')
ht = get(gca,'Title');
set(ht,'FontSize',12)
```

MATLAB erlaubt es auch Vektorfelder darzustellen. Wir nehmen den Gradienten der Funktion, die in `Z` gespeichert ist, vgl. Figur 7.8.

```
>> x = [0:24]/24;
>> y = x;
>> for i=1:25
>>    for j=1:25
>>       hc = cos((x(i) + y(j) -1)*pi);
>>       hs = sin((x(i) + y(j) -1)*pi);
>>       gx(i,j) = -2*hs*hc*pi + 2*(x(i) - y(j));
>>       gy(i,j) = -2*hs*hc*pi - 2*(x(i) - y(j));
```

```
>>     end
>> end
>> quiver(x,y,gx,gy,1)
```

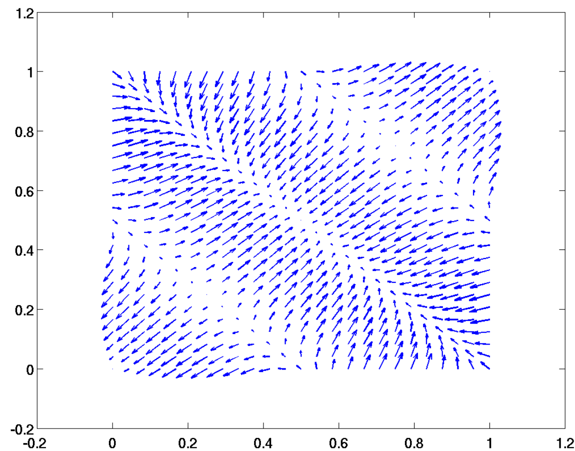

Abbildung 7.8: Das Vektorfeld $\mathbf{grad}(\sin(x+y)\cos(x+y))$

7.5.1 Aufgabe

Probieren Sie, einen Film zu machen, indem Sie in einer Schleife den Ansichtswinkel eines 3-D Plots 'kontinuierlich' ändern. (Arbeiten Sie mit `get(gca,'View')`.)

Um das ganze als Film abzuspeichern kann man die Befehle `getframe` und `movie`. Siehe `help getframe`.

7.6 Darstellung von Daten

Matlab hat vier Funktionen, um 2- und 3-dimensionale *Balkengraphiken* zu produzieren: `bar`, `barh`, `bar3`, `bar3h`. Die 3 steht für 3-dimensionale Balken, das h für horizontal. Wir gehen hier nur auf die einfachste Art der Benützung ein. Sei eine Matrix mit m Zeilen und n Spalten vorgegeben. Mit dem Befehl `bar` werden die Zahlen in Gruppen von n Balken dargestellt. Die n Zahlen der i-ten Matrixzeilen gehören in die i-te von m Gruppen. Die folgende Befehlssequenz produziert die Graphicken in Abb. 7.9.

```
>> B=round(10*rand(6,4)+0.5)

B =

     7     9     3     2
     9     5     6     2
     7     8     6     5
     6     9     2     1
     1     5     8     9
     3     3     2     1

>> bar(2:3:17,B,'grouped')
>> bar(B,'stacked')
```

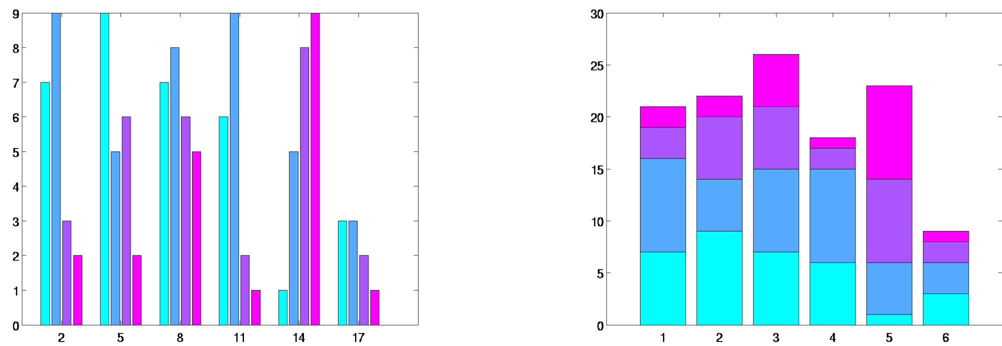

Abbildung 7.9: Darstellung einer Zahlenreihe `grouped` (links) und `stacked` (rechts).

7.6.1 Aufgaben

1. Berechnen Sie mit `rand` n im Interval $[0, 1]$ gleichverteilte Zufallszahlen, und schauen Sie, ob sie auch wirklich gleichverteilt sind. Wenden Sie den Befehl `hist` an.

2. Im Jahr 2000 war die Sprachverteilung in der Schweiz folgendermassen:

Sprache	Deutsch	Französisch	Italienisch	Rätoromanisch
Bewohner	4'640'000	1'480'000	470'000	35'000

   Stellen Sie die prozentuale Sprachverteilung durch ein Tortendiagramm dar. Dazu verwenden Sie den MATLAB-Befehl `pie`. Beschriftung mit `legend`.

Kapitel 8

Anwendungen aus der Numerik

8.1 Kreisausgleichsproblem

Gegeben sind die Punkte

$$(\xi_i, \eta_i), \qquad i = 1, \ldots, m.$$

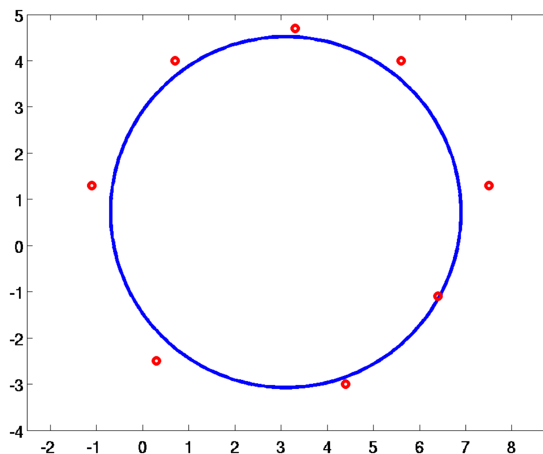

Abbildung 8.1: Kreisausgleichsproblem

Gesucht ist ein Kreis, der "möglichst gut" durch die Punkte verläuft. Die Gleichung für Kreis mit Mittelpunkt (m_1, m_2) und Radius r lautet

$$(x - m_1)^2 + (y - m_2)^2 = r^2$$

Durch Ausmultiplizieren der Klammern erhält man

$$2m_1 x + 2m_2 y + r^2 - m_1^2 - m_2^2 = x^2 + y^2. \tag{8.1}$$

Definiert man $c := r^2 - m_1^2 - m_2^2$, so ergibt (8.1) für jeden Messpunkt eine lineare Gleichung für die Unbekannten m_1, m_2 und c,

$$\begin{pmatrix} \xi_1 & \eta_1 & 1 \\ \vdots & \vdots & \vdots \\ \xi_m & \eta_m & 1 \end{pmatrix} \begin{pmatrix} 2m_1 \\ 2m_2 \\ c \end{pmatrix} = \begin{pmatrix} \xi_1^2 + \eta_1^2 \\ \vdots \\ \xi_m^2 + \eta_m^2 \end{pmatrix}. \tag{8.2}$$

Da im allgemeinen mehr als drei Messungen durchgeführt werden, sind mehr Gleichungen als Unbekannte vorhanden; die lineare Gleichung ist überbestimmt. Das Gleichungssystem muss nach der

Methode der kleinsten Quadrate gelöst werden. Der so erhaltene Ausgleichskreis hat den Nachteil, dass geometrisch nicht offensichtlich ist, was minimiert wird.

Bei Rundheitsmessungen in der Qualitätskontrolle ist es erwünscht, dass die *Abstände* d_i der gemessenen Punkte zum ausgeglichenen Kreis minimiert werden. Sei

$$d_i = \sqrt{(m_1 - \xi_i)^2 + (m_2 - \eta_i)^2} - r.$$

Dann soll nach dem *Gauss-Prinzip* $\mathbf{x} = (x_1, x_2, x_3)^T \equiv (m_1, m_2, r)^T$ so bestimmt werden, dass

$$\sum_{i=1}^{m} f_i(\mathbf{x})^2 = \text{minimal}$$

wobei

$$f_i(\mathbf{x}) = d_i = \sqrt{(x_1 - \xi_i)^2 + (x_2 - \eta_i)^2} - x_3.$$

Dieses *nichtlineare Ausgleichsproblem* kann nach *dem Gauss-Newton Verfahren* gelöst werden: Sei $\mathbf{x}$ eine Näherungslösung. Man entwickelt

$$\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))^T$$

um $\mathbf{x}$ und erhält

$$\mathbf{f}(\mathbf{x} + \mathbf{h}) \approx \mathbf{f}(\mathbf{x}) + J(\mathbf{x})\mathbf{h} \approx \mathbf{0}. \tag{8.3}$$

Gleichung (8.3) ist ein lineares Ausgleichsproblem für die Korrekturen $\mathbf{h}$:

$$J(\mathbf{x})\mathbf{h} \approx -\mathbf{f}(\mathbf{x})$$

wobei die sog. Jacobimatrix wie folgt aussieht:

$$J = \begin{pmatrix} \frac{\partial f_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial x_3} \\ \vdots & & \vdots \\ \frac{\partial f_m(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial f_m(\mathbf{x})}{\partial x_3} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{x_1 - \xi_1}{\sqrt{(x_1 - \xi_1)^2 + (x_2 - \eta_1)^2}} & \frac{x_2 - \eta_1}{\sqrt{(x_1 - \xi_1)^2 + (x_2 - \eta_1)^2}} & -1 \\ \vdots & \vdots & \vdots \\ \frac{x_1 - \xi_m}{\sqrt{(x_1 - \xi_m)^2 + (x_2 - \eta_m)^2}} & \frac{x_2 - \eta_m}{\sqrt{(x_1 - \xi_m)^2 + (x_2 - \eta_m)^2}} & -1 \end{pmatrix}$$

Als gute Startwerte für die Iteration können die Werte des linearen Problems (8.2) verwendet werden.

```
% Kreisanpassung nach der Methode der kleinsten Quadrate
% unter Verwendung des Gauss-Newton Verfahrens
% (zeigt auch einige Moeglichkeiten von MATLAB)
%
  xi  = [ 0.7 3.3 5.6 7.5  6.4  4.4  0.3 -1.1]';  %  Gegebene
                                                  %  Messpunkte
  eta = [ 4.0 4.7 4.0 1.3 -1.1 -3.0 -2.5  1.3]';
%
  [xi, eta]
  xmin = min(xi); xmax = max(xi);  ymin = min(eta); ymax = max(eta);
  dx = (xmax - xmin)/10; dy = (ymax -ymin)/10;
  axis([xmin-dx  xmax+dx  ymin-dy  ymax+dy]);
  axis('equal');          %  damit Einheiten auf beiden Achsen etwa
                          %  gleich sind
  hold;                    %  nachfolgende Plots im gleichen
                           %  Koordinatensystem
```

```
  plot(xi,eta,'o'); pause; %  Es geht weiter mit RETURN

  phi = [0:0.02:2*pi];     %  um den Naeherungskreis zu zeichnen
  h = size(xi); n = h(1);  %  Bestimmung der Anzahl Messwerte
  x = [0.1 1 1]';          %  Startwerte : m1 = x(1), m2 = x(2), r =
                           %  x(3)
%
 text(0.6,0.9, 'm1 =','sc');   % Ueberschrift
 text(0.71,0.9,'m2 =','sc');
 text(0.82,0.9,'r =','sc'); i = 0.9;
 format long; minimum = [];      % soll norm(f) fuer jede Iteration
                                 % speichern
 while norm(h)>norm(x)*1e-4,     % solange Korrektur h gross ist ...
    u = x(1) + x(3)*cos(phi);    % zeichne Naeherungskreis
    v = x(2) + x(3)*sin(phi);
    i = i-0.05;
    text(0.6,i, num2str(x(1)),'sc');
    text(0.71,i, num2str(x(2)),'sc');
    text(0.82,i, num2str(x(3)),'sc');
    plot(u,v); pause;
    a = x(1)-xi; b = x(2)-eta;
    fak = sqrt(a.*a + b.*b);
    J = [a./fak b./fak -ones(size(a))];   % Jacobimatrix
    f = fak -x(3);                  % rechte Seite
    h = -J\f;                       % Aufloesen nach Korrektur (QR)
    x = x + h;
    [x h ], pause                   % Ausdrucken x und Korrektur
    minimum = [minimum norm(f)];
 end;
 minimum'

ans =
   0.70000   4.00000                   Gegebene Punkte x_i, y_i
   3.30000   4.70000
   5.60000   4.00000
   7.50000   1.30000
   6.40000  -1.10000
   4.40000  -3.00000
   0.30000  -2.50000
  -1.10000   1.30000

                                           Bedeutung des Ausdrucks:

ans =
   3.09557664665968   2.99557664665968        m_1     dm_1
   0.62492278502735  -0.37507721497265        m_2     dm_2
   3.57466996351570   2.57466996351570        r       dr

ans =
   3.04180805000003  -0.05376859665966
   0.74825578400634   0.12333299897899
   4.10472289255764   0.53005292904193

ans =
   3.04326031235803   0.00145226235801
   0.74561749573922  -0.00263828826712
   4.10585853340459   0.00113564084696

ans =
   3.04323750977451  -0.00002280258352
   0.74567950486445   0.00006200912523
```

```
    4.10585580071342  -0.00000273269117

ans =
   12.24920034893416     ||r||
    1.63800114348265
    0.54405321190190
    0.54401144797876
```

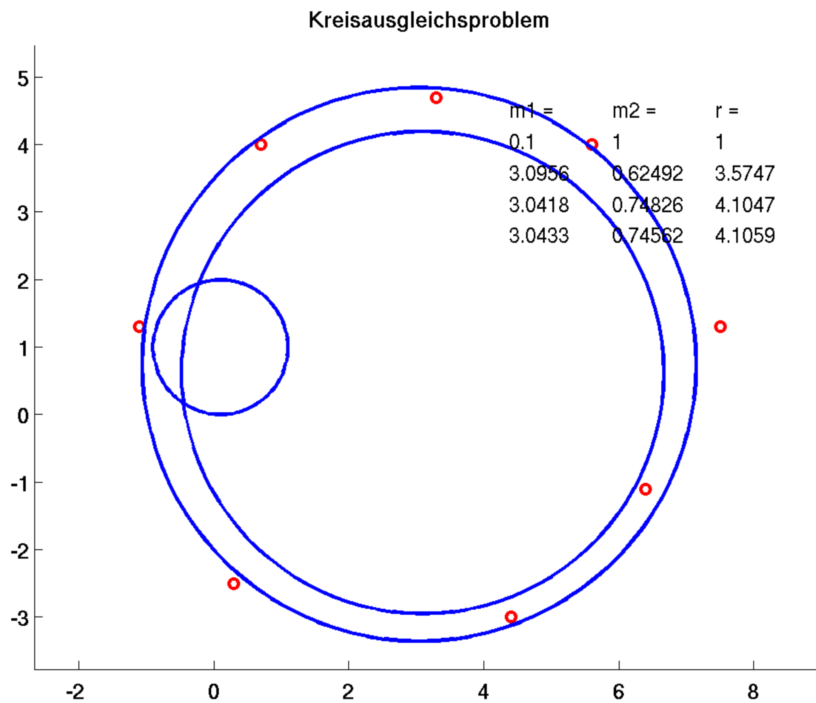

Abbildung 8.2: Kreisausgleich

8.2 Singulärwertzerlegung

Es ist einfach, neue Funktionen in Matlab einzuführen. Man muss dazu ein Matlab Programm schreiben und es als M-File abspeichern.

Funktionen können Inputparameter haben und ein oder mehrere Resultate liefern.

Matlab Funktionen werden auch als M-Files gespeichert. Im Unterschied zu einem Script-file muss die erste Zeile das Wort `function` enthalten. Ein Funktionen-File unterscheidet sich von einem Script-File indem Argumente übergeben werden können und indem Variablen, die im File definiert werden, *lokal* sind. Diese werden also aus dem Arbeitsspeicher entfernt, sobald die Funktion durchlaufen ist. Mit Funktionen-Files kann der Benutzer Matlab nach seinen Bedürfnissen erweitert.

Wir betrachten zunächst die Wurzelfunktion `sqrtm` von Matlab.

Wenn wir als Beispiel die Matrixwurzel aus der Matrix

$$A = \begin{bmatrix} 4 & 9 \\ 25 & 36 \end{bmatrix}$$

berechnen, erhalten wir:

```
 >> A = [4 9; 25 36]
 >> W = sqrtm(A)
W =
   0.8757 + 1.2019i   1.3287 - 0.2852i
   3.6907 - 0.7922i   5.5998 + 0.1880i
```

Dass W wirklich eine Wurzel ist, sieht man beim Quadrieren:

```
W*W
ans =
   4.0000 - 0.0000i   9.0000 - 0.0000i
  25.0000 + 0.0000i  36.0000 + 0.0000i
```

Die Quadratwurzel einer *diagonalisierbaren* Matrix kann mittels

```
function W1 = wurzel(A);
% Berechnet die Quadratwurzel von A
% Aufruf  W1 = wurzel(A)
[Q,D] = eig(A);
D1 = sqrt(D);
W1 = Q*D1/Q;
```

berechnet werden. Werden diese Anweisungen als File mit dem Namen `wurzel.m` abgespeichert, so kann die Funktion wie eine eingebaute MATLAB Funktion verwendet werden:

```
>> help wurzel

  Berechnet die Quadratwurzel von A
  Aufruf  W1 = wurzel(A)
```

Wir sehen, dass die Kommentarzeile nach der ersten Kopfzeilen automatisch von der Help-Funktion ausgedruckt werden. Mittels `type` wird das File ganz ausgedruckt:

```
>> type wurzel

function W1 = h(A);
% Berechnet die Quadratwurzel von A
% Aufruf  W1 = wurzel(A)
[Q,D] = eig(A);
D1 = sqrt(D);
W1 = Q*D1/Q;
```

und `wurzel` unterscheidet sich im Gebrauch nicht (wohl aber bezüglich Rundungsfehler) von `sqrtm`

```
>> wurzel(A) - sqrtm(A)
ans =
   1.0e-15 *
        0 - 0.4441i   0.2220 + 0.1110i
        0 + 0.2220i        0 - 0.1388i
```

Funktionen können *mehrere* Resultate liefern. Als Beispiel betrachten wir die Singulärwertzerlegung einer Matrix. Sei A eine $m \times n$ Matrix. Dann existieren eine orthogonale $m \times m$ Matrix U, eine orthogonale $n \times n$ Matrix V und eine $m \times n$ Diagonalmatrix Σ, so dass

$$A = U\Sigma V^T$$

gilt. Die MATLAB Funktion `svd` berechnet diese Zerlegung.

```
>> A =
    22    52     1    93    70    33
     5    83    38    85    91    63
    68     3     7    53    76    76
    68     5    42     9    26    99
    93    53    69    65     5    37
    38    67    59    42    74    25

>> svd(A)         % svd, allein aufgerufen, liefert den Vektor der
                  % singulaeren Werte der Matrix.
ans =
  305.3327
  121.8348
   89.4757
   57.6875
   37.0986
    3.0329


>> [U S V ] = svd(A) % Mit den Parametern U,S,V auf der linken Seite
                     % wird die vollstaendige Zerlegung berechnet.
                     % Dabei sind U und V  orthogonal und S diagonal.
U =
    0.3925    0.4007   -0.1347   -0.5222   -0.1297   -0.6146
    0.5068    0.4598   -0.0681    0.3545   -0.4612    0.4345
    0.4017   -0.2308   -0.5706   -0.3185    0.4366    0.4096
    0.3231   -0.6371   -0.2695    0.3813   -0.3286   -0.4046
    0.4050   -0.3802    0.6980   -0.3788   -0.1325    0.2077
    0.3992    0.1559    0.3030    0.4597    0.6741   -0.2427
S =
  305.3327         0         0         0         0         0
         0  121.8348         0         0         0         0
         0         0   89.4757         0         0         0
         0         0         0   57.6875         0         0
         0         0         0         0   37.0986         0
         0         0         0         0         0    3.0329
V =
    0.3710   -0.6348    0.1789   -0.4023    0.4171   -0.3026
    0.3717    0.3728    0.4647    0.2416   -0.1945   -0.6408
    0.2867   -0.2260    0.5365    0.4805    0.0601    0.5869
    0.4810    0.3301    0.0796   -0.6448   -0.3068    0.3787
    0.4719    0.3727   -0.4480    0.2346    0.6149    0.0676
    0.4335   -0.3989   -0.5071    0.2794   -0.5589   -0.0733

>> norm(U*S*V'-A) % Kontrolle der Matrixzerlegung
ans =
   3.0075e-13
```

Die Konditionszahl einer Matrix $cond(A)$ ist definiert als das Verhältnis des grössten zum kleinsten singulären Wert. *Die Konditionszahl der Matrix A bestimmt die Genauigkeit der Lösung x beim numerischen Auflösen eines linearen Gleichungssystems $Ax = b$.* Sei $\tilde{x}$ die numerisch berechnete (durch Rundungsfehler verfälschte) Lösung. Dann gilt die Abschätzung:

$$\frac{||x - \tilde{x}||}{||x||} \leq cond(A)\varepsilon \quad \text{mit} \quad \varepsilon = \text{Maschinengenauigkeit}$$

```
>> cond(A)
      ans =   100.6742

>> S(1,1)/S(6,6)  % Verhaeltnis der singulaeren Werte = cond(A) !
      ans =   100.6742
```

```
>> S(6,6) = 0.0003    % Wir verkleinern den kleinsten singulaeren
                      % Wert, damit machen wir die Kondition schlechter.
S =
  305.3327         0         0         0         0         0
         0  121.8348         0         0         0         0
         0         0   89.4757         0         0         0
         0         0         0   57.6875         0         0
         0         0         0         0   37.0986         0
         0         0         0         0         0    0.0003

>  AS = U*S*V'  %  Neue Matrix
```

```
AS =
   21.4361   50.8057    2.0938   93.7058   70.1259   32.8633
    5.3987   83.8443   37.2268   84.5010   90.9110   63.0966
   68.3759    3.7960    6.2710   52.5296   75.9161   76.0911
   67.6287    4.2136   42.7202    9.4647   26.0829   98.9100
   93.1906   53.4036   68.6304   64.7615    4.9574   37.0462
   37.7773   66.5284   59.4319   42.2787   74.0497   24.9460

>> cond(AS)
       ans =   1.0178e+06   % Die Kondition der Matrix AS ist viel
                            % groesser als jene der Matrix A.

>> x = [1:6]/17  % Waehlen einen Loesungsvektor

x  =
    0.0588    0.1176    0.1765    0.2353    0.2941    0.3529

>> x = x'; b = A*x; bs = AS*x;  % Wir berechnen zugehoerige rechte Seiten

>> b'  =
   61.7059   85.7647   67.2353   56.7059   53.7059   61.0000

>> bs' =
   61.8801   85.6416   67.1192   56.8206   53.6470   61.0688

>> xa = A\b       % Loesen des Gleichungssystems  A*x = b

xa =
    0.0588
    0.1176
    0.1765
    0.2353
    0.2941
    0.3529

>> norm(xa-x)/norm(x)
ans =
   7.5815e-15

>> eps                % Maschinengenauigkeit. Die Loesung xa ist etwas
eps =                 % genauer als die Schranke angibt.
   2.2204e-16

>> xas= AS\bs     % Loesen des Gleichungssystems  AS*x = bs
```

```
xas =
    0.0588
    0.1176
    0.1765
    0.2353
    0.2941
    0.3529

>> norm(xas-x)/norm(x)          % Die Loesung xas ist wie erwartet um
                                % ca.  cond(AS) Stellen falsch.
ans =
   2.1368e-11
```

8.3 Gewöhnliche Differentialgleichungen

Die Beispiele sind aus den Kapiteln "Tractrix and Similar Curves" und "Some Least Squares Problems" aus W. Gander and J. Hřebíček, ed., *Solving Problems in Scientific Computing*, Springer Berlin Heidelberg New York, 1993, second edition 1995, third edition June 1997. Das Buch entstand als Zusammenarbeit von unserem Institut für Wissenschaftliches Rechnen mit zwei Physikinstituten von Brno (Tschechische Republik).

8.3.1 The child and the toy

Let us now solve a more general problem and suppose that a child is walking on the plane along a curve given by the two functions of time $X(t)$ and $Y(t)$.

Suppose now that the child is pulling or pushing some toy, by means of a rigid bar of length a. We are interested in computing the orbit of the toy when the child is walking around. Let

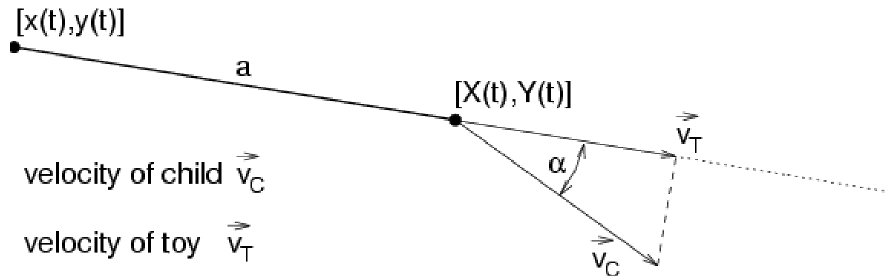

Abbildung 8.3: Velocities $\mathbf{v}_C$ and $\mathbf{v}_T$.

$(x(t), y(t))$ be the position of the toy. From Figure 8.3 the following equations are obtained:

1. The distance between the points $(X(t), Y(t))$ and $(x(t), y(t))$ is always the length of the bar. Therefore
$$(X-x)^2 + (Y-y)^2 = a^2. \tag{8.4}$$

2. The toy is always moving in the direction of the bar. Therefore the difference vector of the two positions is a multiple of the velocity vector of the toy, $\mathbf{v}_T = (\dot{x}, \dot{y})^T$:
$$\begin{pmatrix} X-x \\ Y-y \end{pmatrix} = \lambda \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \quad \text{with} \quad \lambda > 0. \tag{8.5}$$

3. The speed of the toy depends on the direction of the velocity vector $\mathbf{v_C}$ of the child. Assume, e.g., that the child is walking on a circle of radius a (length of the bar). In this special case the toy will stay at the center of the circle and will not move at all (this is the final state of the first numerical example.

   From Figure 8.3 we see that *the modulus of the velocity* $\mathbf{v_T}$ *of the toy is given by the modulus of the projection of the velocity* $\mathbf{v_C}$ *of the child onto the bar.*

Inserting Equation (8.5) into Equation (8.4), we obtain

$$a^2 = \lambda^2(\dot{x}^2 + \dot{y}^2) \quad \longrightarrow \quad \lambda = \frac{a}{\sqrt{\dot{x}^2 + \dot{y}^2}}.$$

Therefore

$$\frac{a}{\sqrt{\dot{x}^2 + \dot{y}^2}} \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \begin{pmatrix} X - x \\ Y - y \end{pmatrix}. \tag{8.6}$$

We would like to solve Equation (8.6) for $\dot{x}$ and $\dot{y}$. Since we know the modulus of the velocity vector of the toy $|\mathbf{v}_T| = |\mathbf{v}_C| \cos\alpha$, see Figure 8.3, this can be done by the following steps:

- Normalize the difference vector $(X - x, Y - y)^T$ and obtain a vector $\mathbf{w}$ of unit length.
- Determine the projection of $\mathbf{v}_C = (\dot{X}, \dot{Y})^T$ onto the subspace generated by $\mathbf{w}$. This is simply the scalar product $\mathbf{v}_C^T\mathbf{w}$, since $\mathbf{v}_C^T\mathbf{w} = |\mathbf{v}_C||\mathbf{w}| \cos\alpha$ and $|\mathbf{w}| = 1$.
- $\mathbf{v}_T = (\dot{x}, \dot{y})^T = (\mathbf{v}_C^T\mathbf{w})\mathbf{w}$.

Now we can write the function to evaluate the system of differential equations in MATLAB.

```
function zs = f(t,z)
%
[X Xs Y Ys] = child(t);
v = [Xs; Ys];
w = [X-z(1); Y-z(2)];
w = w/norm(w);
zs = (v'*w)*w;
```

The function `f` calls the function `child` which returns the position $(X(t), Y(t))$ and velocity of the child $(Xs(t), Ys(t))$ for a given time `t`. As an example consider a child walking on the circle $X(t) = 5\cos t; Y(t) = 5\sin t$. The corresponding function `child` for this case is:

```
function [X, Xs, Y, Ys] = child(t);
%
X  =  5*cos(t);  Y  =  5*sin(t);
Xs = -5*sin(t);  Ys =  5*cos(t);
```

MATLAB offers two M-files `ode23` and `ode45` to integrate differential equations. In the following main program we will call one of these functions and also define the initial conditions (Note that for $t = 0$ the child is at the point $(5, 0)$ and the toy at $(10, 0)$):

```
% main1.m
y0 = [10 0]';
[t y] = ode45('f',[0 100],y0);
clf; hold on;
axis([-6 10 -6 10]);
axis('square');
plot(y(:,1),y(:,2));
```

If we plot the two columns of y we obtain the orbit of the toy. Furthermore we add the curve of the child in the same plot with the statements:

```
t = 0:0.05:6.3
[X, Xs, Y, Ys] = child(t);
plot(X,Y,':')
hold off;
```

Note that the length of the bar a does not appear explicitly in the programs; *it is defined implicitly by the position of the toy, (initial condition), and the position of the child (function* `child`*) for* $t = 0$.

8.3.2 The jogger and the dog

We consider the following problem: a jogger is running along his favorite trail on the plane in order to get his daily exercise. Suddenly, he is being attacked by a dog. The dog is running with constant speed w towards the jogger. Compute the orbit of the dog.

The orbit of the dog has the property that the velocity vector of the dog points at every time to its goal, the jogger. We assume that the jogger is running on some trail and that his motion is described by the two functions $X(t)$ and $Y(t)$.
Let us assume that for $t = 0$ the dog is at the point (x_0, y_0), and that at time t his position will be $(x(t), y(t))$. The following equations hold:

1. $\dot{x}^2 + \dot{y}^2 = w^2$: The dog is running with constant speed.

2. The velocity vector of the dog is parallel to the difference vector between the position of the jogger and the dog:
$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \lambda \begin{pmatrix} X - x \\ Y - y \end{pmatrix} \quad \text{with } \lambda > 0.$$

If we substitute this in the first equation we obtain

$$w^2 = \dot{x}^2 + \dot{y}^2 = \lambda^2 \left|\left| \begin{pmatrix} X - x \\ Y - y \end{pmatrix} \right|\right|^2.$$

This equation can be solved for λ:
$$\lambda = \frac{w}{\left|\left| \binom{X-x}{Y-y} \right|\right|} > 0.$$

Finally, substitution of this expression for λ in the second equation yields the differential equation of the orbit of the dog:
$$\begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} = \frac{w}{\left|\left| \binom{X-x}{Y-y} \right|\right|} \begin{pmatrix} X - x \\ Y - y \end{pmatrix}. \tag{8.7}$$

Again we will make use of one of the M-files `ode23.m` or `ode45.m` to integrate the system of differential equations. We notice that the system (8.7) has a singularity when the dog reaches the jogger. In this case the norm of the difference vector becomes zero and we have to stop the integration. The above mentioned Matlab functions for integrating differential equations require as input an interval of the independent variable. Matlab 5.0 provides now also the possibility to define another termination criterion for the integration, different from a given upper bound for the independent variable. It is possible to terminate the integration by checking zero crossings of a function. In our example one would like to terminate integration when the dog reaches the jogger, i.e. when $||(X - x, Y - y)||$ becomes small. In order to do so we have to add a third input and two more output parameters to the M-function `dog.m`. The integrator `ode23` or `ode45` calls the function in two ways: The first one consists of dropping the third parameter. The function then returns only the parameter `zs`: the speed of the dog. In the second way the keyword `'events'` is assigned to the parameter `flag`. This keyword tells the function to return the zero-crossing function in the first output `zs`. The second output `isterminal` is a logical vector that tells the integrator, which components of the first output force the procedure to stop when they become zero. Every component with this property is marked with a nonzero entry in `isterminal`. The third output parameter `direction` is also a vector that indicates for each component of `zs` if zero crossings shall only be regarded for increasing values (`direction = 1`), decreasing values (`direction = -1`) or in both cases (`direction = 0`). The condition for zero crossings is checked in the integrator. The speed w of the dog must be declared global in `dog` and in the main program. The orbit of the jogger is given by the M-function `jogger.m`.

```
function [zs,isterminal,direction] = dog(t,z,flag);
%
global w  % w = speed of the dog
X= jogger(t);
```

```
h= X-z;
nh= norm(h);
if nargin < 3 | isempty(flag) % normal output
   zs= (w/nh)*h;
else
   switch(flag)
   case 'events'  % at norm(h)=0 there is a singularity
      zs= nh-1e-3;   % zero crossing at pos_dog=pos_jogger
      isterminal= 1; % this is a stopping event
      direction= 0;  % don't care if decrease or increase
   otherwise
      error(['Unknown flag: ' flag]);
   end
end
```

The main program `main2.m` defines the initial conditions and calls `ode23` for the integration. We have to provide an upper bound of the time t for the integration.

```
% main2.m
global w
y0 = [60;70];  % initial conditions, starting point of the dog
w = 10;        % w  speed of the dog
options= odeset('RelTol',1e-5,'Events','on');
[t,Y] = ode23('dog',[0,20],y0,options);
clf; hold on;
axis([-10,100,-10,70]);
plot(Y(:,1),Y(:,2));
J=[];

for h= 1: length(t),
   w  = jogger(t(h));
   J = [J; w'];
end;
plot(J(:,1), J(:,2),':');
```

The integration will stop either if the upper bound for the time t is reached or if the dog catches up with the jogger. For the latter case we set the flag `Events` of the ODE options to `'on'`. This tells the integrator to check for zero crossings of the function `dog` called with `flag = 'events'`. After the call to `ode23` the variable `Y` contains a table with the values of the two functions $x(t)$ and $y(t)$. We plot the orbit of the dog simply by the statement `plot(Y(:,1),Y(:,2))`. In order to show also the orbit of the jogger we have to compute it again using the vector t and the function `jogger`.

Let us now compute a few examples. First we let the jogger run along the x-axis:

```
function s = jogger(t);
s   = [8*t; 0];
```

In the above main program we chose the speed of the dog as $w = 10$, and since here we have $X(t) = 8t$ the jogger is slower. As we can see the dog is catching the poor jogger.

If we wish to indicate the position of the jogger's troubles, *(perhaps to build a small memorial)*, we can make use of the following file `cross.m`

```
function cross(Cx,Cy,v)
% draws at position  Cx,Cy  a cross of height 2.5v
% and width 2*v
Kx = [Cx Cx Cx Cx-v Cx+v];
Ky = [Cy Cy+2.5*v Cy+1.5*v Cy+1.5*v Cy+1.5*v];
plot(Kx,Ky);
plot(Cx,Cy,'o');
```

The cross in the plot was generated by appending the statements

```
p = max(size(Y));
cross(Y(p,1),Y(p,2),2)
```

to the main program.

The next example shows the situation where the jogger turns around and tries to run back home:

```
function s = jogger1(t);
%
if t<6, s = [8*t; 0];
else    s = [8*(12-t) ;0];
end
```

However, using the same main program as before the dog catches up with the jogger at time $t = 9.3$.

Let us now consider a faster jogger running on an ellipse

```
function s = jogger2(t);
s   = [  10+20*cos(t)
      20 + 15*sin(t)];
```

If the dog also runs fast ($w = 19$), he manages to reach the jogger at time $t = 8.97$.

We finally consider an old, slow dog ($w = 10$). He tries to catch a jogger running on a elliptic track. However, instead of waiting for the jogger somewhere on the ellipse, he runs (too slow) after his target, and we can see a steady state developing where the dog is running on a closed orbit inside the ellipse.

8.3.3 Showing motion with Matlab

It would be nice to simultaneously show the motions of both the child and the toy or the dog and the jogger instead of just plotting statically their orbits. This is possible using *handle graphics* commands in MATLAB. The main program for the child and her toy now looks as follows:

```
% main3.m
y0 = [0 20]';
options= odeset('RelTol',1e-10);
[t y] = ode45 ('f', [0 40], y0, options);
[X, Xs, Y, Ys] = child (t);

xmin = min (min (X), min (y (:, 1)));
xmax = max (max (X), max (y (:, 1)));
ymin = min (min (Y), min (y (:, 2)));
ymax = max (max (Y), max (y (:, 2)));

clf; hold on;
axis ([xmin xmax ymin ymax]);
% axis('equal');
title ('The Child and the Toy.');
stickhandle = line ('Color', 'blue', 'EraseMode', 'xor', ...
               'LineStyle', '-', 'XData', [], 'YData', []);

for k = 1:length(t)-1,
  plot ([X(k), X(k+1)], [Y(k), Y(k+1)], '-', ...
       'Color', 'red', 'EraseMode', 'none');
  plot ([y(k,1), y(k+1,1)], [y(k,2), y(k+1,2)], '-', ...
       'Color', 'green', 'EraseMode', 'none');
  set (stickhandle, 'XData', [X(k+1), y(k+1,1)], ...
      'YData', [Y(k+1), y(k+1,2)]);
  drawnow;
end;
hold off;
```

We define the variable `stickhandle` as a handle to a graphical object of type `line` associated with the stick. In the loop, we draw new segments of the child and toy orbits and move the position of the stick. The `drawnow` command forces these objects to be plotted instantaneously. Therefore, we can watch the two orbits and the stick being plotted simultaneously.

In the case of the jogger and the dog we do not even have to define a handle. All we have to do is to draw the segments of the two orbits in the proper sequence:

```
% main4.m
global w;
y0 = [60; 70];  % initial conditions, starting point of the dog
w = 10;         % w  speed of the dog
options= odeset('RelTol',1e-5,'Events','on');
[t,Y] = ode45 ('dog', [0 20], y0, options);

J=[];
for h= 1:length(t),
  w = jogger(t(h));
  J = [J; w'];
end

xmin = min (min (Y (:, 1)), min (J (:, 1)));
xmax = max (max (Y (:, 1)), max (J (:, 1)));
ymin = min (min (Y (:, 2)), min (J (:, 2)));
ymax = max (max (Y (:, 2)), max (J (:, 2)));
clf; hold on;
axis ([xmin xmax ymin ymax]);
% axis ('equal');
title ('The Jogger and the Dog.');


for h=1:length(t)-1,
  plot ([Y(h,1), Y(h+1,1)] , [Y(h,2), Y(h+1,2)], '-', ...
        'Color', 'red', 'EraseMode','none');
  plot ([J(h,1), J(h+1,1)] , [J(h,2), J(h+1,2)], '*', ...
       'Color', 'blue', 'EraseMode','none');
  drawnow;
  pause(1);
end
hold off;
```

8.4 Fitting Lines, Rectangles and Squares in the Plane

Fitting a line to a set of points in such a way that the sum of squares of the distances of the given points to the line is minimized, is known to be related to the computation of the main axes of an inertia tensor. Often this fact is used to fit a line and a plane to given points in the 3d space by solving an eigenvalue problem for a 3×3 matrix.

In this section we will develop a different algorithm, based on the singular value decomposition, that will allow us to fit lines, rectangles and squares to measured points in the plane and that will also be useful for some problems in 3d space.

Let us first consider the problem of fitting a straight line to a set of given points P_1, P_2, $\ldots$, P_m in the plane. We denote their coordinates with (x_{P_1}, y_{P_1}), (x_{P_2}, y_{P_2}), $\ldots$, (x_{P_m}, y_{P_m}). Sometimes it is useful to define the vectors of all x- and y-coordinates. We will use $\mathbf{x}_P$ for the vector $(x_{P_1}, x_{P_2}, \ldots, x_{P_m})$ and similarly $\mathbf{y}_P$ for the y coordinates.

The problem we want to solve is not *linear regression*. Linear regression means to fit the linear model $y = ax + b$ to the given points, i.e. to determine the two parameters a and b such that the

sum of squares of the residual is minimized:

$$\sum_{i=1}^{m} r_i^2 = \min, \quad \text{where} \quad r_i = y_{P_i} - a x_{P_i} - b.$$

This simple linear least squares problem is solved in MATLAB by the following statements (assuming that $\texttt{x} = \mathbf{x}_P$ and $\texttt{y} = \mathbf{y}_P$):

```
p = [ x ones(size(x))]\y;
a = p(1); b = p(2);
```

In the case of the linear regression the sum of squares of the differences of the y coordinates of the points P_i to the fitted linear function is minimized. What we would like to minimize now, however, is the *sum of squares of the distances of the points from the fitted straight line.*

In the plane we can represent a straight line uniquely by the equations

$$c + n_1 x + n_2 y = 0, \quad n_1^2 + n_2^2 = 1. \tag{8.8}$$

The unit vector (n_1, n_2) is orthogonal to the line. A point is on the line if its coordinates (x, y) satisfy the first equation. On the other hand if $P = (x_P, y_P)$ is some point not on the line and we compute

$$r = c + n_1 x_P + n_2 y_P$$

then $|r|$ is its distance from the line. Therefore if we want to determine the line for which the sum of squares of the distances to given points is minimal, we have to solve the constrained least squares problem

$$||\mathbf{r}|| = \sum_{i=1}^{m} r_i^2 = \min$$

subject to

$$\begin{pmatrix} 1 & x_{P_1} & y_{P_1} \\ 1 & x_{P_2} & y_{P_2} \\ \vdots & \vdots & \vdots \\ 1 & x_{P_m} & y_{P_m} \end{pmatrix} \begin{pmatrix} c \\ n_1 \\ n_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{pmatrix} \quad \text{and } n_1^2 + n_2^2 = 1. \tag{8.9}$$

Let $\mathbf{A}$ be the matrix of the linear system (8.9), $\mathbf{x}$ denote the vector of unknowns $(c, n_1, n_2)^T$ and $\mathbf{r}$ the right hand side. Since orthogonal transformations $\mathbf{y} = \mathbf{Q}^T\ \mathbf{r}$ leave the norm invariant ($||\mathbf{y}||_2 = ||\mathbf{r}||_2$ for an orthogonal matrix $\mathbf{Q}$), we can proceed as follows to solve problem (8.9).

First we compute the QR decomposition of $\mathbf{A}$ and reduce our problem to solving a small system:

$$\mathbf{A} = \mathbf{QR} \quad \Longrightarrow \quad \mathbf{Q}^T\mathbf{A}\mathbf{x} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} c \\ n_1 \\ n_2 \end{pmatrix} = \mathbf{Q}^T\mathbf{r} \tag{8.10}$$

Since the nonlinear constraint only involves two unknowns we now have to solve

$$\begin{pmatrix} r_{22} & r_{23} \\ 0 & r_{33} \end{pmatrix} \begin{pmatrix} n_1 \\ n_2 \end{pmatrix} \approx \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \text{subject to} \quad n_1^2 + n_2^2 = 1. \tag{8.11}$$

Problem (8.11) is of the form $||\mathbf{Bx}||_2 = \min$, subject to $||\mathbf{x}||_2 = 1$. The value of the minimum is the *smallest singular value of* $\mathbf{B}$, and the solution is given by the corresponding singular vector. Thus we can determine n_1 and n_2 by a singular value decomposition of a 2-by-2 matrix. Inserting the values into the first component of (8.10) and setting it to zero, we then can compute c. As a slight generalization we denote the dimension of the normal vector $\mathbf{n}$ by `dim`. Then, the MATLAB function to solve problem (8.9) is given by the folowing Algorithm `cslq`.

```
function [c,n] = clsq(A,dim);
% solves the constrained least squares Problem
% A (c n)' ~ 0 subject to norm(n,2)=1
% length(n) = dim
% [c,n] = clsq(A,dim)
[m,p] = size(A);
if p < dim+1, error ('not enough unknowns'); end;
if m < dim, error ('not enough equations'); end;
m = min (m, p);
R = triu (qr (A));
[U,S,V] = svd(R(p-dim+1:m,p-dim+1:p));
n = V(:,dim);
c = -R(1:p-dim,1:p-dim)\R(1:p-dim,p-dim+1:p)*n;
```

Let us test the function `clsq` with the following main program:

```
% mainline.m
Px = [1:10]'
Py = [ 0.2 1.0 2.6 3.6 4.9 5.3 6.5 7.8 8.0 9.0]'
A = [ones(size(Px)) Px Py]
[c, n] = clsq(A,2)
```

The line computed by the program `mainline` has the equation $0.4162 - 0.7057x + 0.7086y = 0$. We would now like to plot the points and the fitted line. For this we need the function `plotline`,

```
function plotline(x,y,s,c,n,t)
% plots the set of points (x,y) using the symbol s
% and plots the straight line c+n1*x+n2*y=0 using
% the line type defined by t
plot(x,y,s)
xrange = [min(x) max(x)];
yrange = [min(y) max(y)];
if n(1)==0, % c+n2*y=0  => y = -c/n(2)
   x1=xrange(1); y1 = -c/n(2);
   x2=xrange(2); y2 = y1
elseif n(2) == 0, % c+n1*x=0  => x = -c/n(1)
   y1=yrange(1); x1 = -c/n(1);
   y2=yrange(2); x2 = x1;
elseif xrange(2)-xrange(1)> yrange(2)-yrange(1),
   x1=xrange(1); y1 = -(c+n(1)*x1)/n(2);
   x2=xrange(2); y2 = -(c+n(1)*x2)/n(2);
else
   y1=yrange(1); x1 = -(c+n(2)*y1)/n(1);
   y2=yrange(2); x2 = -(c+n(2)*y2)/n(1);
end
plot([x1, x2], [y1,y2],t)
```

The picture is generated by adding the commands

```
clf; hold on;
axis([-1, 11 -1, 11])
plotline(Px,Py,'o',c,n,'-')
hold off;
```

8.4.1 Fitting two Parallel Lines

To fit two parallel lines, we must have two sets of points. We denote those two sets by $\{P_i\}, i = 1, \ldots, p$, and $\{Q_j\}, j = 1, \ldots, q$. Since the lines are parallel, their normal vector must be the same. Thus the equations for the lines are

$$\begin{aligned} c_1 + n_1 x + n_2 y &= 0, \\ c_2 + n_1 x + n_2 y &= 0, \\ n_1^2 + n_2^2 &= 1. \end{aligned}$$

If we insert the coordinates of the two sets of points into these equations we get the following constrained least squares problem:

$$||\mathbf{r}|| = \sum_{i=1}^{m} r_i^2 = \min$$

subject to

$$\begin{pmatrix} 1 & 0 & x_{P_1} & y_{P_1} \\ 1 & 0 & x_{P_2} & y_{P_2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & x_{P_p} & y_{P_p} \\ 0 & 1 & x_{Q_1} & y_{Q_1} \\ 0 & 1 & x_{Q_2} & y_{Q_2} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & x_{Q_q} & y_{Q_q} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ n_1 \\ n_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{p+q} \end{pmatrix} \quad \text{and } n_1^2 + n_2^2 = 1. \tag{8.12}$$

Again, we can use our function `clsq` to solve this problem:

```
% mainparallel.m
Px = [1:10]'
Py = [ 0.2 1.0 2.6 3.6 4.9 5.3 6.5 7.8 8.0 9.0]'
Qx = [ 1.5 2.6 3.0 4.3 5.0 6.4 7.6 8.5 9.9 ]'
Qy = [ 5.8 7.2 9.1 10.5 10.6 10.7 13.4 14.2 14.5]'
A = [ones(size(Px))  zeros(size(Px)) Px Py
    zeros(size(Qx))  ones(size(Qx)) Qx Qy ]
[c, n] = clsq(A,2)
clf; hold on;
axis([-1 11 -1 17])
plotline(Px,Py,'o',c(1),n,'-')
plotline(Qx,Qy,'+',c(2),n,'-')
hold off;
```

The results obtained by the program `mainparallel` are the two lines

$$\begin{aligned} 0.5091 - 0.7146x + 0.6996y &= 0, \\ -3.5877 - 0.7146x + 0.6996y &= 0, \end{aligned}$$

8.4.2 Fitting Orthogonal Lines

To fit two orthogonal lines we can proceed very similar as in the the case of the parallel lines. If (n_1, n_2) is the normal vector of the first line, then the second line must have the normal vector $(-n_2, n_1)$ in order to be orthogonal. Therefore again we will have four unknowns: c_1, c_2, n_1 and n_2. If the P_i's are the points associated with the first line and the Q_j's are the points associated with the second line, we obtain the following constrained least squares problem:

$$||\mathbf{r}|| = \sum_{i=1}^{m} r_i^2 = \min$$

subject to

$$\begin{pmatrix} 1 & 0 & x_{P_1} & y_{P_1} \\ 1 & 0 & x_{P_2} & y_{P_2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & x_{P_p} & y_{P_p} \\ 0 & 1 & y_{Q_1} & -x_{Q_1} \\ 0 & 1 & y_{Q_2} & -x_{Q_2} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & y_{Q_q} & -x_{Q_q} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ n_1 \\ n_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{p+q} \end{pmatrix} \quad \text{and } n_1^2 + n_2^2 = 1. \tag{8.13}$$

We only have to change the definition of the matrix A in `mainparallel` in order to compute the equations of the two orthogonal lines. To obtain a nicer plot we also chose different values for the second set of points.

```
% mainorthogonal.m
Px = [1:10]'
Py = [ 0.2 1.0 2.6 3.6 4.9 5.3 6.5 7.8 8.0 9.0]'
Qx = [ 0 1 3 5 6 7]'
Qy = [12 8 6 3 3 0]'
A = [ones(size(Px))   zeros(size(Px)) Px  Py
     zeros(size(Qx))  ones(size(Qx))  Qy -Qx  ]
[c, n] = clsq(A,2)
clf; hold on;
axis([-1 11 -1 13])
axis('equal')
plotline(Px,Py,'o',c(1),n,'-')
n2(1) =-n(2); n2(2) = n(1)
plotline(Qx,Qy,'+',c(2),n2,'-')
```

The Program `mainorthogonal` computes the two orthogonal lines

$$\begin{aligned} -0.2527 - 0.6384x + 0.7697y &= 0, \\ 6.2271 - 0.7697x - 0.6384y &= 0. \end{aligned}$$

8.4.3 Fitting a Rectangle

Fitting a rectangle requires four sets of points:

$$P_i, i = 1, \ldots, p, \quad Q_j, j = 1, \ldots, q, \quad R_k, k = 1, \ldots, r, \quad S_l, l = 1, \ldots, s.$$

Since the sides of the rectangle are parallel and orthogonal we can proceed very similarly as before. The four sides will have the equations

$$\begin{aligned} a: \quad c_1 + n_1 x + n_2 y &= 0 \\ b: \quad c_2 - n_2 x + n_1 y &= 0 \\ c: \quad c_3 + n_1 x + n_2 y &= 0 \\ d: \quad c_4 - n_2 x + n_1 y &= 0 \\ n_1^2 + n_2^2 &= 1. \end{aligned}$$

Inserting the sets of points we get the following constrained least squares problem:

$$||\mathbf{r}|| = \sum_{i=1}^{m} r_i^2 = \min$$

subject to

$$\begin{pmatrix} 1 & 0 & 0 & 0 & x_{P_1} & y_{P_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 0 & 0 & x_{P_p} & y_{P_p} \\ 0 & 1 & 0 & 0 & y_{Q_1} & -x_{Q_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & 0 & y_{Q_q} & -x_{Q_q} \\ 0 & 0 & 1 & 0 & x_{R_1} & y_{R_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 0 & x_{R_r} & y_{R_r} \\ 0 & 0 & 0 & 1 & y_{S_1} & -x_{S_1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1 & y_{S_s} & -x_{S_s} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ n_1 \\ n_2 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_{p+q+r+s} \end{pmatrix} \quad \text{and } n_1^2 + n_2^2 = 1. \tag{8.14}$$

Instead of explicitly giving the coordinates of the four sets of points, we will now enter the points with the mouse using the `ginput` function in MATLAB. `[X,Y] = ginput(N)` gets N points from the current axes and returns the x- and y-coordinates in the vectors X and Y of length N. The points have to be entered clock- or counter clock wise in the same order as the sides of the rectangle: the next side is always orthogonal to the previous.

```
% rectangle.m
clf; hold on;
axis([0 10 0 10])
axis('equal')
p=100; q=100; r=100; s=100;

disp('enter points P_i belonging to side A')
disp('by clicking the mouse in the graphical window.')
disp('Finish the input by pressing the Return key')
[Px,Py] = ginput(p); plot(Px,Py,'o')
disp('enter points Q_i for side B ')
[Qx,Qy] = ginput(q); plot(Qx,Qy,'x')
disp('enter points R_i for side C ')
[Rx,Ry] = ginput(r); plot(Rx,Ry,'*')
disp('enter points S_i for side D ')
[Sx,Sy] = ginput(s); plot(Sx,Sy,'+')

zp = zeros(size(Px)); op =  ones(size(Px));
zq = zeros(size(Qx)); oq =  ones(size(Qx));
zr = zeros(size(Rx)); or =  ones(size(Rx));
zs = zeros(size(Sx)); os =  ones(size(Sx));

A = [ op zp zp zp Px  Py
      zq oq zq zq Qy -Qx
      zr zr or zr Rx  Ry
      zs zs zs os Sy -Sx]

[c, n] = clsq(A,2)

% compute the 4 corners of the rectangle
B  = [n  [-n(2) n(1)]']
X = -B* [c([1 3 3 1])'; c([2 2 4 4])']
X = [X X(:,1)]
plot(X(1,:), X(2,:))

% compute the individual lines, if possible
if all([sum(op)>1 sum(oq)>1 sum(or)>1 sum(os)>1]),
  [c1, n1] = clsq([op Px Py],2)
  [c2, n2] = clsq([oq Qx Qy],2)
  [c3, n3] = clsq([or Rx Ry],2)
  [c4, n4] = clsq([os Sx Sy],2)

  % and their intersection points
  aaa = -[n1(1) n1(2); n2(1) n2(2)]\[c1; c2];
  bbb = -[n2(1) n2(2); n3(1) n3(2)]\[c2; c3];
  ccc = -[n3(1) n3(2); n4(1) n4(2)]\[c3; c4];
  ddd = -[n4(1) n4(2); n1(1) n1(2)]\[c4; c1];

  plot([aaa(1) bbb(1) ccc(1) ddd(1) aaa(1)], ...
       [aaa(2) bbb(2) ccc(2) ddd(2) aaa(2)],':')
end
hold off;
```

The Program `rectangle` not only computes the rectangle but also fits the individual lines to the set of points. The result is shown as Figure 8.4. Some comments may be needed to understand

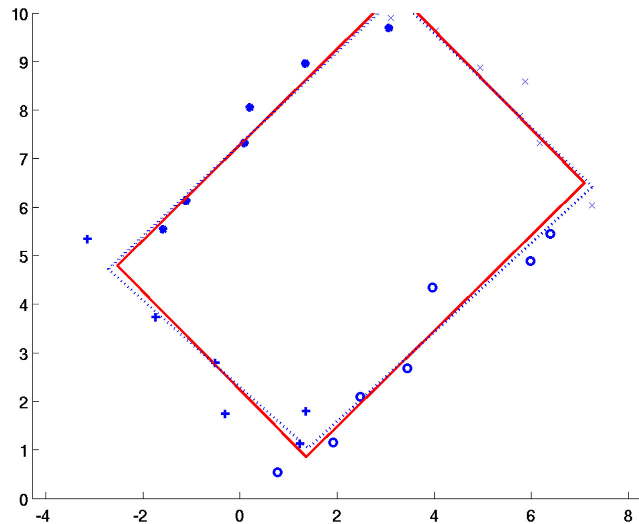

Abbildung 8.4: Fitting a Rectangle.

some statements. To find the coordinates of a corner of the rectangle, we compute the point of intersection of the two lines of the corresponding sides. We have to solve the linear system

$$\begin{array}{rcl} n_1 x + n_2 y & = & -c_1 \\ -n_2 x + n_1 y & = & -c_2 \end{array} \iff C\mathbf{x} = -\binom{c_1}{c_2}, \quad C = \begin{pmatrix} n_1 & n_2 \\ -n_2 & n_1 \end{pmatrix}$$

Since C is orthogonal, we can simply multiply the right hand side by $B = C^T$ to obtain the solution. By arranging the equations for the 4 corners so that the matrix C is always the system matrix, we can compute the coordinates of all 4 corners simultaneously with the compact statement

```
X = -B* [c([1 3 3 1])'; c([2 2 4 4])'].
```

8.5 Beispiel Prototyping: Update der QR-Zerlegung

In der Statistik kommt es oft vor, dass eine Serie von Ausgleichsproblemen gelöst werden muss, wobei die Matrix $A \in \mathbb{R}^{m \times m}$ jeweils nur wenig ändert. Wir betrachten dazu die folgende Problemstellung:

Gegeben sei die QR-Zerlegung $A = Q\binom{R}{0}$ mit $Q \in \mathbb{R}^{m \times m}$. Gegeben seien ferner die Vektoren $u \in \mathbb{R}^m$ und $v \in \mathbb{R}^n$.

Kann man die QR-Zerlegung der Matrix

$$A' = A + uv^T = Q'R'$$

einfacher aus der QR-Zerlegung von A berechen?

Es ist

$$Q^T A' = Q^T A + Q^T uv^T = \binom{R}{0} + wv^T \quad \text{mit} \quad w = Q^T u.$$

Sei $J_k = G(k, k+1, \phi_k)$ eine (orthogonale) Givensrotationsmatrix, welche bei der Multiplikation $J_k w$ die Elemente w_k und w_{k+1} verändert.

Der Algorithmus besteht aus drei Schritten:

1. Wir wählen Rotationen J_k und Winkel ϕ_k so, dass die Elemente $n+2:m$ von w zu Null rotiert werden:

$$\underbrace{J_{n+1}^T \dots J_{m-2}^T J_{m-1}^T Q^T}_{Q_1^T} A' = \begin{pmatrix} R \\ 0 \end{pmatrix} + J_{n+1}^T \dots J_{m-2}^T J_{m-1}^T w v^T$$
$$= \begin{pmatrix} R \\ 0 \end{pmatrix} + \begin{pmatrix} w' \\ 0 \end{pmatrix} v^T$$

   Dabei wird nur die Matrix Q zu Q_1 und der Vektor w geändert.

2. Nun werden weitere Givensrotationen durchgeführt, welche die Komponenten von w' bis auf die erste α annullieren. Dabei ändert Q_1 zu Q_2 und auch die Matrix R zu einer Hessenbergmatrix H_1.

$$\underbrace{J_1^T \dots J_n^T Q_1^T}_{Q_2^T} A' = H_1 + \begin{pmatrix} \alpha \\ 0 \end{pmatrix} v^T = H$$

   Der Beitrag der Rang-1 Matrix ändert nun in der letzten Gleichung nur die erste Zeile von H_1; man erhält eine neue Hessenbergmatrix H.

3. Nun wird im letzten Schritt durch weitere Givensrotationen die untere Nebendiagonale von H annulliert und man erhält die gesuchte neue QR-Zerlegung.

$$\underbrace{J_n^T \dots J_1^T Q_2^T}_{Q'^T} A' = J_n^T \dots J_1^T H = R'$$

Um diesen Algorithmus zu implementieren, konstruieren wir zunächst eine *Demonstrationsversion.* Dazu benötigt man die Givensrotationsmatrizen J_k:

```
function [G ] = rot(m,i,k,x,y);
%  [G ] = rot(m,i,k,x,y);
% konstruiert eine orthogonale Matrix G mxm,  welche y zu null rotiert
G = eye(m,m);
if y~=0,
   cot = x/y; si = 1/sqrt(1+cot^2); co = si*cot;
   G(i,i) = co; G(k ,k) = co; G(i,k) = si; G(k,i) = -si;
end;
```

Damit kann das folgende Programm geschrieben werden. Nach jeder Rotation wurde ein Pausenbefehl eingefügt, man kann damit den Ablauf des Algorithmus leicht verfolgen.

```
A= rand(7,4);
u=rand(7,1);
v=[1 2 3 4]';
[m n] = size(A);
[Q R] = qr(A);
QS =Q;
RS = R;
%
w = Q'*u
disp(' 1. Phase:  Anullieren  w(n+2:m) und aendern nur Q')
for i=m:-1:n+2,
  G= rot(m,i-1,i,w(i-1), w(i))
  w = G*w
  QS = QS*G';
  pause
end;
disp(' 2. Phase  Annullieren w(2:n+1) und aendern R und Q')
for i= n+1:-1:2,
```

```
  G= rot(m,i-1,i,w(i-1), w(i))
  w = G*w
  RS = G*RS
  QS = QS*G';
  pause
end
disp(' Addieren jetzt die Rang 1 Mod. zur ersten Zeile von R')
RS = RS + w*v'
pause
disp(' 3.Phase:  RS ist nun Hessenberg und wird reduziert')
for i = 1:n,
  G= rot(m,i,i+1,RS(i,i), RS(i+1,i))
 RS = G*RS
  QS = QS*G';
  pause
end
disp(' Kontrolle')
AS = A + u*v'; [qs rs] = qr(AS);
qs
QS
rs
RS
```

Nun ist es einfach die Demonstrationsversion zu einem effektiven Programm umzuschreiben. Wir benötigen dazu eine Funktion `giv`, um die Rotationswinkel, bzw. den sin und cos davon zu berechnen:

```
function [s,c] = giv(x,y);
% bestimmt eine Givensrotation,
% welche y zu null rotiert
if y~=0,
   cot = x/y; s = 1/sqrt(1+cot^2); c = s*cot;
else c=1; s=0;
end
```

Im nachfolgenden Programm werden die Transformationen nicht mehr durch Multiplikation mit vollen Matrizen ausgeführt. Es werden nur noch jene Elemente neu berechnet, die von den Givensrotationen betroffen sind.

Man beachte, dass in MATLAB Vektoroperationen sehr gut dargestellt werden können, z. B. wird durch

```
rs(i,i-1:n) = -si*rs(i-1,i-1:n) + co*rs(i,i-1:n);
```

die i-te Zeile der Matrix R neu berechnet. Das Programm kann dadurch mit wenig Aufwand auf einen Vektorrechner übertragen werden.

```
A= round(10*rand(7,4))
 u=rand(7,1)
 v=[1 2 3 4]'

[m n] = size(A);
[Q R] = qr(A); QS =Q; RS = R;
%
w = Q'*u
%  anullieren w(n+2:m) und aendern nur Q
for i=m:-1:n+2,
  [si , co]= giv(w(i-1), w(i));
  w(i-1) = co*w(i-1) + si*w(i);
%  QS = QS*G';
  h = QS(:,i-1);
  QS(:,i-1) = co*h + si*QS(:,i);
  QS(:,i) = -si*h + co*QS(:,i)
```

```
end
%  annullieren w(2:n+1) und aendern R und Q
for i= n+1:-1:2,
  [si , co]= giv(w(i-1), w(i));
%  w = G*w
  w(i-1) = co*w(i-1) + si*w(i);
%  RS = G*RS
  h = co*RS(i-1,i-1:n) + si*RS(i,i-1:n);
  RS(i,i-1:n) = -si*RS(i-1,i-1:n) + co*RS(i,i-1:n);
  RS(i-1,i-1:n) = h
%  QS = QS*G';
  h = QS(:,i-1);
  QS(:,i-1) = co*h + si*QS(:,i);
  QS(:,i) = -si*h + co*QS(:,i)
end
% Addieren jetzt die Rang 1 Mod. zur eRSten Zeile von R
RS(1,:) = RS(1,:) + w(1)*v'
% RS ist nun Hessenberg und wird reduziert
for i = 1:n,
  [si , co]= giv(RS(i,i), RS(i+1,i))
  % RS = G*RS
  h = co*RS(i,i:n) + si*RS(i+1,i:n);
  RS(i+1,i:n) = -si*RS(i,i:n) + co*RS(i+1,i:n);
  RS(i,i:n) = h
  %  QS = QS*G';
  h = QS(:,i);
  QS(:,i) = co*h + si*QS(:,i+1);
  QS(:,i+1) = -si*h + co*QS(:,i+1)
end
% Kontrolle
AS = A + u*v'; [qs rs] = qr(AS);
qs
QS
rs
RS
```

Kapitel 9

Einführungen in Matlab, Resourcen auf dem Internet

Die Firma The Mathworks, die MATLAB produziert und vertreibt, hat eine gute Web-Seite an der URL

http://www.mathworks.com/.

Via den MATLAB Help Navigator kann man bequem auf Informationen von MathWorks zugreiffen, vorausgesetzt der Rechner ist am Internet angeschlossen. Nach dem Start von MATLAB Help, folge man den Links

`MATLAB ▷ Printable Documentation (PDF)`

Auf der Seite

http://www.mathworks.com/support/books/

findet man eine Liste von Hunderten von Büchern zu MATLAB und SimuLink, insbesondere zu Büchern über Numerische Methoden mit MATLAB in verschiedenen Fachgebieten.

Empfehlenswerte Einführungen in MATLAB sind die Bücher

- Kermit Sigmon & Timothy A. Davis: MATLAB Primer, 6th edition. Chapman & Hall/CRC, 2002.
- Desmond J. Higham & Nicholas J. Higham: MATLAB Guide, 2nd edition, SIAM, Philadelphia, 2005.
- Walter Gander & Jirí Hřebíček: Solving Problems in Scientific Computing Using Maple and MATLAB, 4th edition. Springer, 2004.

Das erste Buch ist (bei entsprechenden Zugriffsberechtigung) elektronisch verfügbar, siehe http://www.nebis.ch/. Das letzte ist eine Sammlung von interessanten und recht realistischen Problemen, deren Lösung mit Maple and MATLAB vorgeführt wird.

9.1 Tutorials

- http://www.math.mtu.edu/~msgocken/intro/intro.html
  'A Practical Introduction to Matlab' von Mark S. Gockenbach (Michigan Technological University). Sehr einfach.
- http://www.math.siu.edu/matlab/tutorials.html
  'MATLAB Tutorials' von Edward Neuman (Southern Illinois University). Fünf PDF Files. Ausführliche Beschreibung zum Programmieren, zur linearen Algebra und zu numerischer Analysis mit MATLAB.

- http://math.ucsd.edu/~driver/21d-s99/matlab-primer.html
  'MATLAB Primer' (Second Edition) von Kermit Sigmon (Department of Mathematics, University of Florida). Gute Übersicht über die ältere MATLAB-Version 5.

9.2 Software

- Die Home page von The MathWorks ist

  http://www.mathworks.com/

  Auf dieser Seite gibt es auch public domain MATLABSoftware.

- Stimuliert durch das NA-NET, ein "Netzwerk" für und von numerischen Mathematikern, entstand 1986 NETLIB [3],

  http://www.netlib.org/

  von wo eine Unmenge frei verfügbarer (Public Domain) Software vor allem aus dem Gebiet der Numerischen Analysis gespeichert wird.

9.3 Alternativen zu Matlab

Als Alternativen zum (teuren) MATLAB können aufgefasst werden

- SCILAB: http://www.scilab.org/
- Octave: http://www.octave.org/

Beide Programme arbeiten ähnlich und mit ähnlicher Syntax wie MATLAB. m-Files sind teilweise auch in SCILAB oder Octave benützbar.

Literaturverzeichnis

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide - Release 2.0*, SIAM, Philadelphia, 1994.

[2] J. J. Dongarra, C. Moler, J. Bunch, G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[3] J. Dongarra and E. Grosse, *Distribution of Mathematical Software via Electronic Mail*, Comm. ACM, **30**, 5, pp. 403–407, 1987.

[4] T. F. Coleman und Ch. van Loan, *Handbook For Matrix Computations*, SIAM, Philadelphia, 1988.

[5] W. Gander and J. Hřebíček, ed., *Solving Problems in Scientific Computing*, 3rd edition, Springer, Berlin, 1997.

[6] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide Extension*, Springer Lecture notes in computer science 51, Springer, Berlin, 1977

[7] G. H. Golub and C. F. van Loan, *Matrix Computations*, 2nd edition, Johns Hopkins University Press, Baltimore, 1989.

[8] W. Gander, Matlab-*Einführung*. Neue Methoden des Wissenschaftliches Rechnen, Departement Informatik, 1991.

[9] A. A. Grau, U. Hill, H. Langmaack, *Translation of* ALGOL 60, Springer, Berlin, 1967. (Handbook of automatic computation ; vol. 1, part b)

[10] N. Higham, *Matrix Computations on a PC*, SIAM News, January 1989.

[11] D. J. Higham and N. J. Higham, *MATLAB Guide*, 2nd edition. SIAM, Philadelphia, 2005.

[12] P. Marchand, *Graphics and GUIs with MATLAB*, 2nd edition. CRC Press, Boca Raton, FL, 1999.

[13] C. Moler, *MATLAB Users' Guide*, University of New Mexico Report, Nov. 1980

[14] Matlab*, the Language of Technical Computing*, The MathWorks, South Natick MA, 2004. Available from URL `http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/graphg.pdf`.

[15] *Using* Matlab *Graphics*, Version 7, The MathWorks, South Natick MA, 2004. Available from URL `http://www.mathworks.com/access/helpdesk/help/pdf_doc/matlab/getstart.pdf`.

[16] Michael L. Overton *Numerical computing with IEEE floating point arithmetic*, SIAM, Philadelphia PA, 2001.

[17] Heinz Rutishauser, *Description of* ALGOL 60, Springer, Berlin, 1967 (Handbook of automatic computation ; vol. 1, part b)

[18] K. Sigmon and T. A. Davis, *MATLAB Primer*, 6th edition. Chapman & Hall/CRC, 2002.

[19] B. Simon und R. M. Wilson, *Supercalculators on the PC*, Notices of the AMS, 35 (1988),e pp. 978–1001.

[20] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, Springer Lecture notes in computer science 6, Springer, Berlin, 1976

[21] J. Wilkinson and C. Reinsch, *Linear Algebra*, Springer, Berlin, 1971

[22] URL von Netlib: `http://www.netlib.org/`.

[23] URL von The Mathworks: `http://www.mathworks.com/`.