Skript zum Kompaktkurs

Einführung in die Programmierung mit Matlab

Stephan Rave,
Kathrin Smetana

10.09.2012-14.09.2012,
17.09.2012-21.09.2012

Stand:

5. September 2012

Inhaltsverzeichnis

1 Allgemeines

1.1 Über dieses Skript

Dieses Skript ist als Begleittext für den einwöchigen Kompaktkurs „Einführung in die Programmierung mit MATLAB“ gedacht und gibt einen Überblick über die im Kurs behandelten Themen. Es sollen die grundlegenden MATLAB-Befehle vorgestellt werden und ein Einblick in die vielfältigen Anwendungsmöglichkeiten gegeben werden. Natürlich erlernt man den Umgang mit MATLAB nicht einfach durch das Studium dieses Dokuments. Erfahrungsgemäß ist es für Anfänger äußerst empfehlenswert, möglichst viel selbst herumzuexperimentieren, Beispielcodes zu verstehen und abzuändern. Die Übungsaufgaben sind als Hausaufgaben gedacht und werden zu Beginn der Kurstreffen besprochen.

1.2 Was ist MATLAB?

Das Softwarepaket MATLAB bietet eine breite Palette unterschiedlicher Funktionalitäten und hat sich in den vergangenen Jahren zu einem der Standardwerkzeuge für numerische Berechnungen in den Bereichen Industrie, Forschung und Lehre entwickelt.
Auf der einen Seite kann man MATLAB einfach als einen mächtigen Taschenrechner auffassen. Andererseits verbirgt sich dahinter eine höhere Programmiersprache mit integrierter Entwicklungsumgebung. Man kann auf eingängliche Weise Berechnungen und anschließende Visualisierungen der Ergebnisse miteinander verbinden. Mathematische Ausdrücke werden in einem benutzerfreundlichen Format dargestellt. Der Standarddatentyp ist die **Matrix** (MATLAB steht für **Mat**rix **lab**oratoy), auch Skalare werden intern als (1×1)-Matrizen gespeichert.
Die Programmiersprache MATLAB ist sehr übersichtlich, einfach strukturiert und besitzt eine eingängige Syntax. So erfordern zum Beispiel Datenstrukturen eine minimale Beachtung, da keine Variablendeklarationen nötig ist. Typische Konstrukte wie Schleifen, Fallunterscheidungen oder Methodenaufrufe werden unterstützt. MATLAB unterscheidet bei der Bezeichnung von Variablen und Funktionen zwischen Groß- und Kleinbuchstaben. Es sind viele (mathematische) Funktionen und Standardalgorithmen bereits vorgefertigt, so dass nicht alles von Hand programmiert werden muss. Durch die modulare Strukturierung in sogenannte *toolboxes* lässt sich die für eine bestimmte Aufgabenstellung benötigte Funktionalität auf einfache Weise in ein Programm einbinden. Beispiele sind die *Optimization toolbox*, die *Statistics toolbox*, die *Symbolic Math toolbox* oder die *Partial Differential Equations toolbox*.

1.3 Literatur

Zur Vertiefung des Stoffes und für die Bearbeitung der Programmieraufgaben ist folgende Literatur empfehlenswert, die Sie in der Bibliothek des Numerischen Instiuts finden:

- Desmond J. Higham: MATLAB *Guide*, Cambridge University Press, 2005 (2. Aufl.)
- Cleve B. Moler: *Numerical Computing with* MATLAB, Cambridge University Press, 2004

Weitere MATLAB-Literatur gibt es in der ULB, wie z.B.

- Walter Gander, Jiri Hrebicek: *Solving problems in scientific computing using Maple and* MATLAB, Springer Verlag, 2004 (4. Aufl.)
- Frieder Grupp, Florian Grupp: MATLAB *7 für Ingenieure. Grundlagen und Programmierbeispiele*, Oldenbourg Verlag, 2006 (4. Aufl.)
- Wolfgang Schweizer: MATLAB *kompakt*, Oldenbourg Verlag, 2008 (3. Aufl.)
- Christoph Überhuber, Stefan Katzenbeisser, Dirk Praetorius: MATLAB *7: Eine Einführung*, Springer, 2004 (1. Aufl.)

Zahlreiche kurze Übersichten und Übungen existieren im Internet, z.B.

- Prof. Gramlich: Einführung in MATLAB
  http://www.hs-ulm.de//users/gramlich/EinfMATLAB.pdf
- MATLAB Online-Kurs der Universität Stuttgart
  http://mo.mathematik.uni-stuttgart.de/kurse/kurs4/
- Getting started with MATLAB
  http://www-math.bgsu.edu/ gwade/matlabprimer/tutorial.html
- Numerical Computing with MATLAB

Der letzte Link führt auf eines der Standardwerke über MATLAB vom MATLAB-Entwickler Cleve Moler. Es enthält viele Anwendungsbeispiele und MATLAB-Programme. Sehr empfehlenswert!

1.4 Matlab starten

1.4.1 In der Uni

Die Kursteilnehmer legen bitte in Ihrem Home-Verzeichnis einen Ordner `MatlabKurs` an. Dazu öffnet man ein Terminal-/Konsolenfenster (Rechtsklick auf den Desktop → Terminal öffnen) und führt dort den Befehl `mkdir /u/<BENUTZERKENNUNG>/MatlabKurs` aus. Mit dem Befehl `cd /u/<BENUTZERKENNUNG>/MatlabKurs` kann man anschließend in dieses Verzeichnis wechseln. Für `<BENUTZERKENNUNG>` setzt man seine entsprechende Uni-Kennung ein. An den Unix- und Linux-Rechnern im Institut kann man Matlab starten, indem man im Terminal folgende Befehle nacheinander eingibt:

- `environ numeric`
- `matlab &`

Nach einigen Sekunden sollte Matlab starten.
Achtung: Die Uni verfügt über eine begrenzte Anzahl an Lizenzen für Matlab, d.h. dass es kann nicht gleichzeitig beliebig oft gestartet werden kann. Sind alle Lizenzen vergeben, erscheint im x-term eine Fehlermeldung. Bei starker Nutzung von Matlab, z.B. während dieses Kurses, kann es etwas länger dauern, bis Matlab gestartet ist. Bitte den Befehl nur einmal aufrufen!

1.4.2 Zu Hause - Kostenloser Download für alle Studenten der WWU

Das ZIV stellt in seinem Softwarearchiv ZIVSoft Matlab zum kostenlosen Download für Studenten und Angestellte der WWU zur Verfügung. Es liegen Versionen sowohl für Microsoft Windows, Linux als auch MacOS bereit. Sie finden alle nötigen Informationen, die Installationsdateien und eine ausführliche, bebilderte Installationsanleitung auf folgender Webseite:

https://zivdav.uni-muenster.de/ddfs/Soft.ZIV/TheMathWorks/Matlab/

Möglicherweise werden Sie zunächst mit etwas einschüchternden Meldungen konfrontiert, dass das Zertifikat des entsprechenden ZIV-Servers nicht akzeptiert wird. Das müssen Sie dann ggfs. gemäß der Anweisungen des gewählten Browsers selbst nachholen. Anschließend werden Sie vor dem Zugriff auf die Dateien nach Ihrem Nutzernamen und Ihrem Standardpasswort gefragt. Bitte arbeiten Sie die Installationsanleitung `matlab.pdf` durch, die Sie im obigen Verzeichnis finden. Sie ist sehr ausführlich und mit zahlreichen Screenshots illustriert.
Während der Matlab -Nutzung muss Ihr Rechner an das Uninetz angeschlossen sein. Falls Sie von zu Hause aus Matlab nutzen möchten, muss Ihr Rechner daher online sein und über VPN mit dem Uninetz verbunden sein. Zur Einrichtung einer VPN-Verbindung gibt es ebenfalls eine Anleitung vom ZIV:

http://www.uni-muenster.de/ZIV/Zugang/VPN.html

Die aktuelle Version (Oktober 2011) ist R2011a.
Achtung: Der Umfang der Installationsdateien ist erheblich (mehrere GB!). Daher kann es hilfreich sein, sich die Medien innerhalb der Universität, z.B. im CIP-Pool, auf mobile Datenträger oder Geräte zu laden, anstatt direkt von zu Hause aus das Herunterladen zu initiieren.

1.4.3 Octave - Die freie Open-Source Alternative

Das Programm GNU Octave ist eine freie Alternative zu MATLAB . Es kann für Linux, Windows und Mac auf folgender Webseite heruntergeladen werden:

http://www.gnu.org/software/octave

Octave bildet fast den gesamten Befehlssatz von MATLAB ab und enthält sogar einige zusätzliche Operatoren und Befehle. Als grafische Oberfläche für Octave kann man zum Beispiel qtoctave verwenden.

2 Grundlagen

Zunächst soll die Benutzeroberfläche, der sogenannte MATLAB-Desktop, beschrieben werden.

2.1 Der MATLAB-Desktop

Nach dem Aufruf von MATLAB öffnet sich ein Fenster ähnlich dem folgenden:

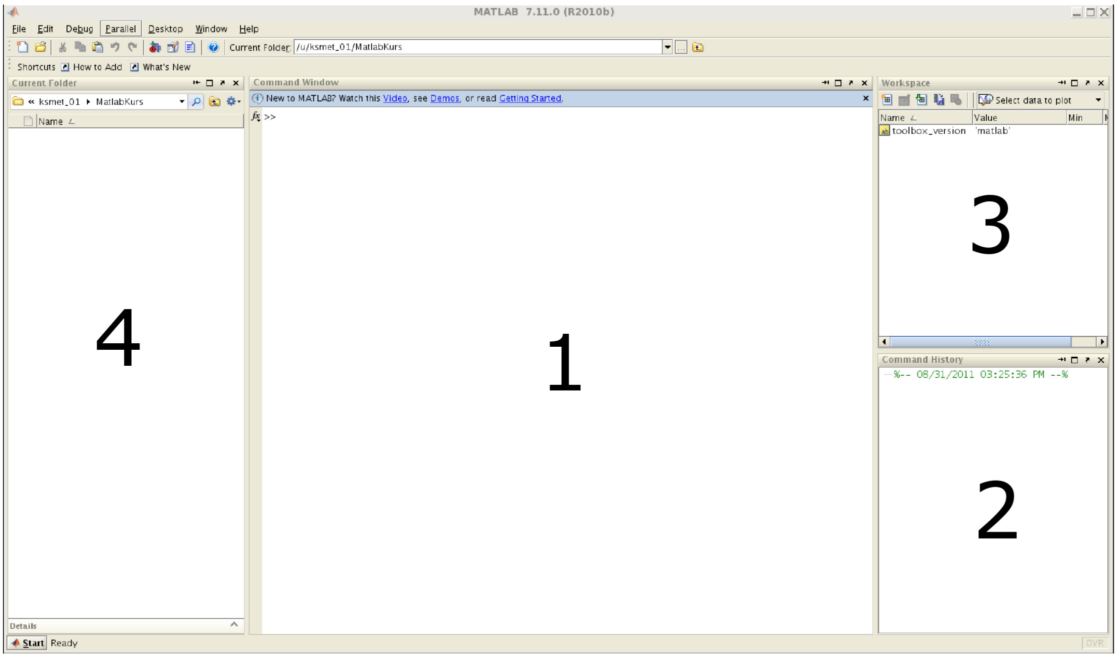

Abbildung 2.1: Der MATLAB-Desktop

Der MATLAB-Desktop besteht aus dem *Command Window* (Bereich 1), der *Command History* (Bereich 2), dem *Workspace* (Bereich 3) und der Angaben zum *Current Folder* (Bereich 4). Der Pfad des *Current Folder* ist auch oberhalb des *Command Windows* nochmals zu sehen. Das Aussehen und die genaue Positionierung dieser Bereiche kann von Version zu Version leicht variieren.

Das *Command Window* ist der Ort, an dem Sie MATLAB-Befehle eingeben können. Ein Befehl wird rechts vom Doppelpfeil eingetippt und mit der <Enter>-Taste abgeschlossen. Er wird dann von MATLAB ausgeführt. Eine eventuell Ausgabe des Befehls lässt sich durch ein angehängtes Semikolon unterdrücken. Eine ganze Reihe von Befehlen können zusammen in einer Datei mit der Endung `.m` abgespeichert werden. Solche Dateien werden M-Files genannt. Dazu muss sich die Datei im aktuellen Verzeichnis (*Current Folder*) befinden, welches entweder links oder in der Befehlsleiste über dem *Command Window* ausgewählt wird. In unserem Fall ist dies das Verzeichnis `MatlabKurs`. Zunächst werden wir unsere Befehle und Berechnungen nur in das *Command Window* eingeben. Erläuterungen zu den M-Files finden Sie in Kapiel 4.

Im Teilfenster rechts unten, der *Command History*, werden Befehle gespeichert, die Sie bereits ausgeführt haben. Durch einen Doppelklick auf einen Befehl in diesem Fenster wird der Befehl ins *Command Window* kopiert und noch einmal ausgeführt. Weiterhin kann man im *Command Window* mit den Pfeil-hoch/runter - Tasten alte Befehle durchblättern. Tippen der ersten Zeichen vorangehender Befehle gefolgt von der Pfeil-hoch-Taste liefert den nächsten zurückliegenden Befehl, der mit diesen Zeichen beginnt.

Im *Workspace* rechts oben werden die momentan angelegten Variablen aufgeführt. Durch einen Doppelklick auf den Variablennamen wird ein *Arrayeditor* geöffnet, mit welchem man die Matrixelemente editieren kann. Über *File* → *Save Workspace As* lassen sich die im *Workspace* vorhandenen Variablen, Funktionen,... als `Name.mat`-Datei speichern. Mit *File* → *Import Data* kann man alle Variablen, Funktionen aus der `Name.mat`-Datei wieder in den *Workspace* laden.

In *Current Folder* werden die Dateien des aktuellen Verzeichnisses angezeigt. Unser Ordner `MatlabKurs` ist noch leer, so dass zu Beginn keine Dateien angezeigt werden.

Unter der Option *Desktop* kann man die Standardeinstellungen bei Bedarf ändern. Die MATLAB-Hilfe kann mit der Maus an der oberen Befehlsleiste via *Help* → *Product Help* aufgerufen werden. Sie enthält die Dokumentation einzelner Befehle, einen MATLAB - Einführungskurs (*Getting Started*), ein Benutzer-Handbuch (*User Guide*), Demos, pdf-Files der Dokumentation und vieles mehr. Die MATLAB -Hilfe ist sehr ausführlich und empfehlenswert!

2.2 Erste Schritte: Matlab als „Taschenrechner“

Für erste einfache Rechnung nutzen wir den interaktiven Modus und geben Befehle direkt in das Command Windows hinter dem Prompt-Zeichen >> ein. So liefert z.B.

```
>> 2-4
```

das Ergebnis

```
ans =
       -2
```

wobei `ans` eine Hilfsvariable ist und für *answer* steht. Mittels der Eingabe

```
>> a = 5.6;
```

wird der Variablen a der Wert 5.6 zugewiesen. Lässt man das Semikolon am Zeilenende weg, erscheint im Command Window die Ausgabe

```
>> a = 5.6

a =
     5.600
```

Ruft man die Variable a auf, wird sie ausgegeben:

```
>> a

a =
    5.600
```

Nun kann mit a weitergerechnet werden:

```
>> a + 2

ans =

    7.6000
```

Das Semikolon am Zeilenende unterdrückt nur die Ausgabe im Command Window, die Berechnung wird dennoch durchgeführt. Mit dem Befehl

```
>> size(a)

ans =
      1    1
```

lässt sich die Dimension einer Variable bestimmen, in diesem Fall ist a eine 1×1-Matrix. Die relative Genauigkeit liegt bei $\approx 2.2 \cdot 10^{-16}$. Standardmäßig gibt MATLAB nur die ersten fünf Dezimalstellen an, gespeichert werden jedoch immer 16. Das Ausgabeformat kann mittels des `format` Befehls geändert werden:

```
>> format long;

>> a + 2

ans =

   7.60000000000000
```

Das Zeichen `i` steht bei MATLAB für die Imaginäre Einheit:

```
>> a + 2*i

ans =

   7.60000000000000 + 2.00000000000000i
```

Das Zeichen `*` kann bei der Multiplikation mit `i` weg gelassen werden, der Aufruf `a+2i` liefert das gleiche Ergebnis. Vorsicht bei der Variablenbezeichnung! Obwohl `i` standardmäßig für die Imaginäre Einheit steht, kann man `i` einen neuen Wert zuweisen. Dies kann zu Fehlern im Programm führen! Daher empfiehlt es sich, eine Variable nie mit `i` zu deklarieren. Namen von Variablen und Funktionen beginnen mit einem Buchstaben gefolgt von einer beliebigen Anzahl von Buchstaben, Zahlen oder Unterstrichen. Matlab unterscheidet zwischen Groß- und Kleinbuchstaben. Parallel zur Variablen `a` kann also eine Variable `A` definiert und genutzt werden. Das Minus-Zeichen `-` darf in Variablen- oder Funktionsnamen nicht auftauchen.

MATLAB rechnet nach dem IEEE Standard für Fließkommazahlen. Die Eingabe

```
>> b = 1/1111101 * 1/3
```

liefert die Ausgabe

```
b =

   3.000027300248433e-07
```

Dies ist gleichbedeutend mit dem Wert $3.000027300248433 \cdot 10^{-7}$. Die Endung `e^` gibt immer den Wert der Zehnerpotenzen an. Wechseln wir wieder mittels `format short` zur Standard-Ausgabe in MATLAB , so würde der Befehl

```
>> b = 1/1111101 * 1/3
```

die Ausgabe

```
b =

   3.0000e-07
```

liefern. Intern wird jedoch der auf 16 Stellen genaue Wert $3.000027300248433 \cdot 10^{-7}$ verwendet.

2.3 Die MATLAB-Hilfe

MATLAB verfügt über ein äußerst umfangreiches Hilfesystem. Sämtliche MATLAB-Befehle, Operatoren und Programmierstrukturen wie Schleifen, Fallunterscheidungen etc. sind ausführlich dokumentiert. Die wohl am meisten benutzte Möglichkeit, die MATLAB-Hilfe in Anspruch zu nehmen, ist durch den Befehl `help Befehlsname` gegeben. Ist man sich beispielsweise unsicher, wie die Syntax oder Funktionsweise des Kommandos `max` zur Suche des Maximums aussieht, so liefert die Eingabe `help max` die folgende Beschreibung:

```
>> help max
 MAX    Largest component.
    For vectors, MAX(X) is the largest element in X. For matrices,
    MAX(X) is a row vector containing the maximum element from each
    column.
    .
    .
    .
```

Das Kommando `help` ohne Argument listet alle verfügbaren Hilfethemen im Command Window auf. Etwas ausführlicher ist die Ausgabe des Befehls `doc Befehlsname`. Hier öffnet sich ein seperates Fenster mit dem Hilfe-Browser, in welchem Hinweise zur Nutzung des entsprechenden Befehls aufgeführt sind. Der Hilfe-Browser lässt sich auch manuell durch das Kommando `helpdesk` starten, anschließend kann man durch eine integrierte Suchfunktion nach MATLAB-Befehlen suchen.

Man kann sich außerdem Demos zu unterschiedlichen Themen anschauen. Das Kommando `demo` öffnet das MATLAB Demo-Fenster mit einer Vielzahl von Demonstrationsbeispielen. Mit `help matlab/demos` erhält man eine Liste aller Demos zu MATLAB.

Die MATLAB-Hilfe sollte bei allen Problemen erste Anlaufstelle sein. So lernt man effektiv, sich selbst weiterzuhelfen. Auch langjährige, erfahrene MATLAB-User verwenden regelmäßig die MATLAB-Hilfe, um sich die genaue Syntax eines Kommandos schnell ansehen zu können.

3 Matrizen, Arrays, Operatoren & Funktionen

3.1 Matrizen & Lineare Algebra

3.1.1 Rechnungen mit Matrizen

Werden Matrizen direkt eingegeben, ist folgendes zu beachten:

- Die einzelnen Matrixelemente werden durch Leerzeichen oder Kommas voneinander getrennt
- Das Zeilenende in einer Matrix wird durch ein Semikolon markiert.
- Die gesamte Matrix wird von eckigen Klammern [ ] umschlossen.
- Skalare Größen sind 1×1-Matrizen, bei ihrer Eingabe sind keine eckigen Klammern nötig.
- Vektoren sind ebenfalls Matrizen. Ein Zeilenvektor ist eine $1 \times n$-Matrix, ein Spaltenvektor eine $n \times 1$-Matrix.

Zum Beispiel wird die 2×4-Matrix

$$\begin{pmatrix} 1 & -3 & 4 & 2 \\ -5 & 8 & 2 & 5 \end{pmatrix}$$

wird wie folgt in Matlab eingegeben und der Variablen `A` zugewiesen:

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

A =
      1  -3   4   2
     -5   8   2   5
```

Man hätte auch `>> A = [1,-3,4,2; -5,8,2,5]` schreiben können. Die Dimension von Matrizen kann mit

```
>> size(A)

ans =

     2     4
```

überprüft werden. Wie in der Mathematik üblich steht zu erst die Anzahl der Zeilen, dann die der Spalten. Vektoren werden in MATLAB wie $(1, n)$ bzw. $(n, 1)$-Matrizen behandelt. 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 ergibt die folgende Eingabe z.B.

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

    w =
         3
         1
         4

    v =
         2     0    -1

    s =
         7
```

Einen Überblick über die definierten Variablen verschafft der so genannte *Workspace* (s. Kapitel 2) oder der Befehl `who` :

```
>> who

Your variables are:

A    a    ans    v    w    s
```

Mit dem Befehl `whos` werden genauere Angaben zu den Variablen gemacht:

```
>> whos
  Name       Size                    Bytes  Class     Attributes

    A          2x4                       64  double
    a          1x1                        8  double
    ans        1x2                       16  double
    s          1x1                        8  double
    v          1x3                       24  double
    w          3x1                       24  double
```

Ein Besipiel für `Attributes` ist zum Beispiel die Eigenschaft `sparse` bei dünnbesetzten Matrizen. Gelöscht werden Variablen mit dem Befehl `clear`.

```
>> clear a;
```

löscht nur die Variable `a`,

```
>> clear all
```

löscht alle Variablen im *Workspace*. Soll ein *Workspace* komplett gespeichert werden, so legt der Aufruf

```
>> save dateiname
```

im aktuellen Verzeichnis (bei uns: `MatlabKurs`) eine Datei `dateiname.mat` an, in der alle Variablen aus dem *Workspace* gespeichert sind. Über *File* → *Import Data* in der oberen Befehlsleiste können die in der Datei `dateiname.mat` gespeicherten Werte wieder hergestellt werden. Ebenso kann man den Befehl

```
>> load dateiname
```

ausführen, um die Daten wieder zu erhalten.

Ist eine Eingabezeile zu lang, kann man sie durch ... trennen und in der folgenden Zeile fortfahren:

```
>> a = 2.5 + 2*i + 3.434562 - 4.2*(2.345 - ...
           1.23) + 1

a =

   2.2516 + 2.0000i
```

Gerade in längeren Programmen ist es sehr wichtig, das Layout übersichtlich zu gestalten. Es empfiehlt sich daher, Zeilen nicht zu lang werden zu lassen.

3.1.2 Arithmetische Operationen mit Skalaren, Vektoren und Matrizen

Es seien `A, B` Matrizen und `c, d` skalare Größen. In MATLAB sind unter gewissen Dimensionsbedingungen an `A, B` folgende **arithmetische Operationen** zwischen Matrizen und Skalaren definiert (hier zunächst nur eine Übersicht, Erläuterungen zu den einzelnen Operationen folgen im Text):

Symbol	Operation	MATLAB -Syntax	math. Syntax
`+`	skalare Addition	`c+d`	$c+d$
`+`	Matrizenaddition	`A+B`	$A+B$
`+`	Addition Skalar - Matrix	`c+A`	
`-`	Subtraktion	`c-d`	$c-d$
`-`	Matrizensubtraktion	`A-B`	$A-B$
`-`	Subtraktion Skalar - Matrix	`A-c`	
`*`	skalare Multiplikation	`c*d`	cd
`*`	Multiplikation Skalar - Matrix	`c*A`	cA
`*`	Matrixmultiplikation	`A*B`	AB
`.*`	punktweise Multiplikation	`A.*B`	
`/`	rechte skalare Division	`c/d`	$\frac{c}{d}$
`\`	linke skalare Division	`c\d`	$\frac{d}{c}$
`/`	rechte Division Skalar - Matrix	`A/c`	$\frac{1}{c}A$
`/`	rechte Matrixdivision	`A/B`	AB^{-1}
`\`	linke Matrixdivision	`A\B`	$A^{-1}B$
`./`	punktweise rechte Division	`A./B`	
`.\`	punktweise linke Division	`A.\B`	
`^`	Potenzieren	`A^c`	A^c
`.^`	punktweise Potenzieren	`A.^B`	
`.'`	transponieren	`A.'`	A^t
`'`	konjugiert komplex transponiert	`A'`	$\bar{A}^t$
`:`	Doppelpunktoperation		

Die Rechenregeln sind analog zu den mathematisch bekannten - auch in MATLAB gilt die Punkt-vor-Strich Regel. Für Klammerausdrücke können die runden Klammern (und) genutzt werden. Die eckigen Klammern sind für die Erzeugung von Matrizen und Vektoren und für Ergebnisse von Funktionsaufrufen reserviert. Geschwungene Klammern werden in MATLAB für die Erzeugung und Indizierung von Zellen verwendet. Zellen sind Felder, die an jeder Stelle beliebige Elemente (Felder, Zeichenketten, Strukturen) und nicht nur Skalare enthalten können.

Erläuterungen zu den arithmetischen Operationen

Seien in mathematischer Notation die Matrizen

$$A = (a_{jk})_{n_1,m_1} = \begin{pmatrix} a_{11} & \dots & a_{1m_1} \\ \vdots & \ddots & \vdots \\ a_{n_1 1} & \dots & a_{n_1 m_1}, \end{pmatrix}, \quad B = (b_{jk})_{n_2,m_2} = \begin{pmatrix} b_{11} & \dots & b_{1m_2} \\ \vdots & \ddots & \vdots \\ b_{n_2 1} & \dots & b_{n_2 m_2}, \end{pmatrix}$$

und der Skalar s gegeben. Im folgenden werden einige Fälle je nach Dimension der Matrizen unterschieden.

1.) Die Multiplikation eines Skalaren mit einer Matrix erfolgt wie gewohnt, der Befehl `C=s*A` ergibt die Matrix

$$C = (s \cdot a_{jk})_{n_1,m_1}.$$

Ebenso kann in MATLAB die Addition/Subtraktion von Skalar und Matrix genutzt werden. Die mathematisch nicht gebräuchliche Schreibweise `C = s + A` erzeugt in MATLAB die Matrix

$$C = (s + a_{jk})_{n_1,m_1},$$

zu jedem Element der Matrix A wird der Skalar s addiert. Analoges gilt für die Subtraktion. Dagegen bewirkt der Befehl `A^s` das s-fache Potenzieren der Matrix A mit Hilfe des Matrixproduktes, wie man es aus der Notation der linearen Algebra kennt, z.B ist `A^2` gleichbedeutend mit `A*A`.

Die MATLAB -Notation `A/c` ist gleichbedeutend mit der Notation `1/c*A`, was der skalaren Multiplikation der Matrix A mit dem Skalar $\frac{1}{c}$ entspricht.
Vorsicht: die Befehle `A\c` und `s^A` sind in diesem Fall nicht definiert!

2.) Für den Fall $n_1 = n_2 = n$ und $m_1 = m_2 = m$ lassen sich die gewöhnlichen Matrixadditionen und -subtraktionen berechnen. Der MATLAB Befehl `A+B` erzeugt die Matrix

$$A + B = (a_{jk} + b_{jk})_{n,m},$$

`A-B` erzeugt dann natürlich

$$A - B = (a_{jk} - b_{jk})_{n,m}.$$

In diesem Fall können weiterhin die punktweisen Operationen `.*`, `./` und `.^` durchgeführt werden, hier werden die einzelnen Matrixelemente punktweise multipliziert/dividiert/potenziert. So ergibt z.B. der Befehl `C=A.*B` das Ergebnis

$$C = (a_{jk} \cdot b_{jk})_{n,m},$$

welches natürlich nicht mit der gewöhnlichen Matrixmultiplikation übereinstimmt! Die punktweise Division von rechts, `C=A./B`, ergibt

$$C = \left(\frac{a_{jk}}{b_{jk}}\right)_{n,m},$$

für `C=A.\B` folgt

$$C = \left(\frac{b_{jk}}{a_{jk}}\right)_{n,m}.$$

Der Vollständigkeit halber soll auch das punktweise Potenzieren aufgeführt werden, `C=A.^B`, ergibt im Fall gleicher Dimensionen die Matrix

$$C = (a_{jk}^{b_{jk}})_{n,m}.$$

3.) In dem Fall $n_2 = m_1 = \tilde{n}$ kann MATLAB eine gewöhnliche Matrixmultiplikation (oder Matrix-Vektor-Multiplikation) durchführen. `C=A*B` liefert dann die (n_1, m_2)-Matrix

$$C = (c_{jk})_{\tilde{n},\tilde{m}} \text{ mit } c_{jk} = \sum_{l=1}^{\tilde{n}} a_{jl} \cdot b_{lk}$$

In dem Fall $n_2 \neq m_1$ gibt `C=A*B` eine Fehlermeldung aus.

4.) Bei der rechten/linken Division von Matrizen muss man sehr vorsichtig sein. Diese macht Sinn, wenn lineare Gleichungssysteme gelöst werden sollen. Sei also A eine (n, n) Matrix, d.h. $n_1 = n$, $m_1 = n$ und B ein $(n, 1)$-Spaltenvektor, d.h. $n_2 = n$ und $m_2 = 1$. Hat die Matrix A vollen Rang, so ist das Gleichungssystem $Ax = B$ eindeutig lösbar. Der MATLAB -Befehl `x=A\B` berechnet in diesem Fall die gesuchte Lösung $x = A^{-1}B$. Ist B ein $(1, n)$-Zeilenvektor, löst der Befehl `x=B/A` das lineare Gleichungssystem $xA = B$ und für das Ergebnis gilt $x = BA^{-1}$. Sind weiterhin A, B quadratische, reguläre (n, n)-Matrizen, so kann mit dem Befehl `C=A/B` die Matrix $C = AB^{-1}$ berechnet werden, `C=A\B` ergibt $C = A^{-1}B$.

Bemerkung: Vorsicht mit dem Gebrauch der Matrixdivisonen `\` und `/`. Ist A eine (n, m) Matrix mit $n \neq m$ und B ein Spaltenvektor mit n Komponenten, ist das LGS $Ax = B$ nicht eindeutig lösbar! Aber der Befehl `x = A\B` ist ausfürbar und liefert eine Approximation des LGS $Ax = B$! Gleiches gilt für die linke Division. In der Vorlesung Numerik 1 werden Sie diese Approximation als so genannte *kleinste Quadrate-Lösung* von überbestimmten Gleichungssystemen kennen lernen.

Einige Beispiele:
Es seien die Matrizen, Vektoren und Skalare

$$a = 5, \quad b = \begin{pmatrix} 4 \\ 2 \\ 1 \end{pmatrix}, \quad c = \begin{pmatrix} 5 \\ 7 \\ -4 \end{pmatrix}, \quad A = \begin{pmatrix} 8 & 7 & 3 \\ 2 & 5 & 1 \\ 5 & 2 & -2 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & -7 & 2 \\ 3 & 2 & 6 \\ 1 & 2\mathrm{i} & 0 \end{pmatrix}$$

gegeben. Dann berechnet MATLAB das gewöhnliche Matrix-Produkt

```
>> A*B

ans =

  24.0000             -42.0000 + 6.0000i  58.0000
  16.0000              -4.0000 + 2.0000i  34.0000
   4.0000             -31.0000 - 4.0000i  22.0000
```

Die punktweise Multiplikation ergibt dagegen

```
>> A.*B

ans =

        0             -49.0000             6.0000
   6.0000              10.0000             6.0000
   5.0000                    0 + 4.0000i        0
```

Das Gleichungssystem $Ax = b$ kann gelöst werden (falls lösbar) mit

```
>> x=A\b

x =

    0.1667
    0.2917
    0.2083
```

In der Tat ergibt

```
>> A*x

ans =

    4.0000
    2.0000
    1.0000
```

Weiterhin ergibt

```
>> B'

ans =

        0            3.0000            1.0000
  -7.0000            2.0000                 0 - 2.0000i
   2.0000            6.0000                 0
```

Das Resultat von `B'+a` errechnet sich zu

```
>> B'+a

ans =

        5             8.0000           6.0000
  -2.0000             7.0000                5 - 2.0000i
   7.0000            11.0000                5
```

Das Skalarprodukt $\langle b, c \rangle$ zwischen den Vektoren b und c kann schnell mittels der Multiplikation

```
>> b'*c

ans =

    30
```

berechnet werden. Prinzipiell kann MATLAB mit Matrizen parallel rechnen, es sollte **immer** auf aufwendige Schleifen verzichtet werden. Fast alles kann mittels Matrix-Operationen effizient berechnet werden. Dies sei nur vorab erwähnt – später dazu mehr.

Für weitere Beispiele zu den Matrix-Operationen sei hier auf die ausführliche Matlab-Hilfe verwiesen.

3.1.3 Lineare Gleichungssysteme

Ist ein Gleichungssystem exakt lösbar, so kann die Lösung auf verschiedene Weisen berechnet werden. Sei:

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ 2x_1 + 3x_2 + 4x_3 &= 5 \\ 4x_1 + x_2 + 5x_3 &= -0.8 \\ \\ \Leftrightarrow Ax &= b \end{aligned}$$

mit

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 4 & 1 & 5 \end{pmatrix}, \qquad b = \begin{pmatrix} 4 \\ 5 \\ -0.8 \end{pmatrix}.$$

Direkt löst Matlab

```
>> A\b

ans =

   -1.4000
    1.8000
    0.6000
```

Dies ist gleichbedeutend mit

```
>> inv(A)*b

ans =

   -1.4000
    1.8000
    0.6000
```

3.1.4 Zugriff auf einzelne Matrixelemente

Wie man konkret mit Matrixelementen in Matlab arbeiten kann, ist am besten an Beispielen ersichtlich. Daher seien in diesem Kapitel wieder die oben definierten Matrizen A und B gegeben. Zur Erinnerung:

$$A = \begin{pmatrix} 8 & 7 & 3 \\ 2 & 5 & 1 \\ 5 & 2 & -2 \end{pmatrix}, \quad B = \begin{pmatrix} 0 & -7 & 2 \\ 3 & 2 & 6 \\ 1 & 2\mathrm{i} & 0 \end{pmatrix}$$

Einzelne Matrixelemente werden direkt via

```
>> B(2,3)

ans =

     6
```

ausgegeben. Der Befehl `B(j,k)` gibt das Matrixelement der Matrix B aus, das in der j. Zeile in der k. Spalte steht.
Matrizen können nahezu beliebig miteinander kombiniert werden. Dabei steht das Zeichen `,` (oder das Leerzeichen) immer für eine neue Spalte. Das Zeichen `;` steht dagegen für eine neue Zeile. So definiert z.B.

```
>>C=[A;B]

C =

   8.0000         7.0000             3.0000
   2.0000         5.0000             1.0000
   5.0000         2.0000            -2.0000
        0        -7.0000             2.0000
   3.0000         2.0000             6.0000
   1.0000              0 + 2.0000i  0
```

die 6×3-Matrix C, welche zunächst die Matrix A enthält und in den nächsten Zeilen die Matrix B, da sich zwischen A und B ein Semikolon, das für Zeilenende steht, befindet. Dagegen bildet der Befehl `C=[A,B]`, welcher gleichbedeutend zu `C=[A B]` ist,

```
>> C=[A B]

C =

   8.0000   7.0000   3.0000        0  -7.0000            2.0000
   2.0000   5.0000   1.0000   3.0000   2.0000            6.0000
   5.0000   2.0000  -2.0000   1.0000        0 + 2.0000i  0
```

eine 3×6-Matrix C. Dies bestätigt der Befehl

```
>> size(C)
```

```
ans =

     3     6
```

Die Matrix B wird in neue Spalten neben die Matrix A geschrieben. Ebenso können bei richtiger Dimension Spalten und Zeilen an eine bestehende Matrix angefügt werden:

```
>> D = [A; [1 0 3]]

D =

     8     7     3
     2     5     1
     5     2    -2
     1     0     3
```

fügt eine neue Zeile an die Matrix A an und speichert das Resultat in eine neue Matrix D,

```
>> D = [A [1 0 3]']

D =

     8     7     3     1
     2     5     1     0
     5     2    -2     3
```

fügt eine neue Spalte an die Matrix A an und überschreibt die bestehende Matrix D damit.
Ferner ist auch eine lineare Indizierung möglich. MATLAB interpretiert dabei die Matrix als einen einzigen langen Spaltenvektor. Z.B. ergibt

```
>> A(5)

      ans =

           5
>>
```

Eine besondere Rolle spielt der Operator **:**. Mit ihm kann man z. B. Teile einer Matrix extrahieren. Der folgende Befehl gibt die erste Zeile von A aus:

```
>> A(1,:)

ans =

     8     7     3
```

Die Nummer 1 innerhalb der Klammern bedeutet 'die erste Zeile' und der Doppelpunkt steht für 'alle Spalten' der Matrix A.

```
>> A(:,2)

ans =

     7
     5
     2
```

dagegen gibt zunächst alle Zeilen aus, aber nur die Elemente, die in der 2. Spalte stehen. Der **:** bedeutet eigentlich 'von … bis'. Die gewöhnliche und ausführliche Syntax für den Doppelpunktoperator ist

Startpunkt **:** Schrittweite **:** Endpunkt

Bei der Wahl

Startpunkt **:** Endpunkt

wird die Schrittweite auf 1 gesetzt. Der **:** alleine gibt alle Elemente spaltenweise nacheinander aus. Das letzte Element kann auch mit `end` beschrieben werden. Schrittweiten können auch negativ sein!
Sei beispielsweise der 1×11-Zeilenvektor $x = (9 \;\; 2 \;\; 4 \;\; 8 \;\; 2 \;\; 0 \;\; 1 \;\; 4 \;\; 6 \;\; 3 \;\; 7)$ gegeben. Dann ergibt

```
>> x(1:2:end)

ans =

     9     4     2     1     6     7
```

die Ausgabe jedes zweiten Elements von x, angefangen beim ersten Element. Negative Schrittweiten bewirken ein 'Abwärtszählen':

```
>> x(end-2:-3:1)

ans =

     6     0     4
```

Das Weglassen der Schrittweite bewirkt wie erwähnt die automatische Schrittweite 1:

```
>> x(1:7)

ans =

     9     2     4     8     2     0     1
```

ist gleichbedeutend mit `x(1:1:7)`.
Ebenso kann der `:` für Matrizen genutzt werden. Mit

```
>> A(1:2,2:3)

ans =

     7     3
     5     1
```

kann man sich z. B. Teile der Matrix A ausgeben lassen, `A(1:2:,2:3)` ist gleichbedeutend mit `A(1:1:2:,2:1:3)` . Hier sind es Zeilen 1 bis 2, davon dann Spalten 2 bis 3. Spalten und Zeilen können auch getauscht werden:

```
>> A(1:3,[3 2 1])

ans =

     3     7     8
     1     5     2
    -2     2     5
```

vertauscht die Spalten 1 und 3.

Es ist auch möglich, Zeilen und Spalten aus einer Matrix heraus zu löschen. Dies ist mit Hilfe eines leeren Vektors `[]` möglich:

```
>> A(:,1)=[]

A =

     7     3
     5     1
     2    -2
```

Die erste Spalte wird gelöscht.

Der Befehl `diag` gibt Diagonalen einer Matrix aus. Ohne weiteres Argument wird die Hauptdiagnoale ausgegeben:

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

     8     7     3
     2     5     1
     5     2    -2

>> diag(A)

ans =

     8
     5
    -2
```

Nebendiagonalen lassen sich durch ein weiteres Argument `n` im Aufruf `diag(A,n)` ausgeben. Positive Zahlen `n` stehen für die n-te obere Nebendiagnale, negative für die n-te untere Nebendiagnale:

```
>> diag(A,1)
ans =

     7
     1

>> diag(A,-1)
ans =

     2
     2
```

Mittels der Befehle `fliplr`, `flipud` können Matrizen an der Mittelsenkrechten oder Mittelwaagerechten gespiegelt (*left/right* und *up/down*) werden:

```
>> fliplr(A)
```

```
ans =

     3     7     8
     1     5     2
    -2     2     5

>> flipud(A)

ans =

     5     2    -2
     2     5     1
     8     7     3
```

Manchmal ist es nützlich, einen Teil der Matrix mit Nullen zu überschreiben. Mit Hilfe des Befehls `tril` wird der Teil *linke untere* Teil der Matrix ausgewählt und der Rest mit Nullen aufgefüllt, mit `triu` wird der Teil über der Hauptdiagonalen, also der *rechte obere* Bereich, ausgewählt und der Rest mit Nullen ausgefüllt:

```
>> tril(A)

ans =

     8     0     0
     2     5     0
     5     2    -2

>> triu(A)

ans =

     8     7     3
     0     5     1
     0     0    -2
```

Abschließend werden noch einige übliche Befehle zum Erzeugen von Matrizen angegeben:

Matlab-Bezeichnung	Beschreibung
`zero(n,m)`	Nullmatrix mit n Zeilen und m Spalten
`ones(n,m)`	Matrix mit n Zeilen und m spalten besetzt mit Einsen
`eye(n)`	Einheitsmatrix mit n Zeilen und Spalten
`rand(n,m)`	Matrix mit n Zeilen und m Spalten besetzt mit gleichverteilten Zufallswerten aus dem Interval [0,1]
`magic(n)`	Matrix mit n Zeilen und m Spalten dessen Zeilen- und Spaltensummen übereinstimmen

3.2 Zahlen und mathematische Funktionen

3.2.1 Darstellung(sbereiche) von Zahlen; definierte Konstanten

An vordefinierten Konstanten seien hier die folgenden angegeben:

Matlab-Bezeichnung	math. Bezeichnung	Erläuterung
`pi`	π	
`i`, `j`	i	Imaginäre Einheit
`eps`		Maschinengenauigkeit
`inf`	∞	
`NaN`		not a number, z.B. `inf-inf`
`realmin`		kleinste positive Maschinenzahl
`realmax`		größte positive Maschinenzahl
`intmax`		größte ganze Zahl (int32).
`intmin`		kleinste ganze Zahl

Die Genauigkeit der Rundung $rd(x)$ einer reellen Zahl x ist durch die Konstante `eps` gegeben, es gilt

$$\left|\frac{x - rd(x)}{x}\right| \leq \texttt{eps} \approx 2.2 \cdot 10^{-16},$$

diese Zahl entspricht der Maschinengenauigkeit, sie wird mit `eps` bezeichnet. Intern rechnet MATLAB mit doppelt genauen (64 Bit) Gleitkommazahlen (gemäß IEEE 754). Standardmäßig gibt Matlab Zahlen fünfstellig aus. Die Genauigkeit von 16 Stellen ist jedoch unabhängig von der Ausgabe. Sehr große/kleine Zahlen werden in Exponentialdarstellung ausgegeben, z.B. entspricht die Ausgabe `-4.3258e+17` der Zahl $-4.3258 \cdot 10^{17}$. Die kleinsten und größten darstellbaren Zahlen sind `realmin` $\sim 2.2251 \cdot 10^{-308}$ und `realmax` $\sim 1.7977 \cdot 10^{+308}$. Reelle Zahlen mit einem Betrag aus dem Bereich von 10^{-308} bis 10^{+308} werden also mit einer Genauigkeit von etwa 16 Dezimalstellen dargestellt.

Vorsicht bei der Vergabe von Variablen! Viele Fehler entstehen z. B. durch die Vergabe der Variablen i und der gleichzeitigen Nutzung von $i = \sqrt{-1}$! Um nachzuprüfen ob eine Variable bereits vergeben ist, gibt es die Funktion `exist`. Sie gibt Eins zurück wenn die Variable vorhanden ist und Null wenn sie es nicht ist. Der Befehl `exist` kann auch auf Funktionsnamen etc. angewandt werden, dazu später mehr.

3.2.2 Wichtige vordefinierte mathematische Funktionen

Es sind sehr viele mathematische Funktionen in MATLAB vorgefertigt, hier wird nur ein kleiner Überblick über einige dieser Funktionen gegeben. Prinzipiell sind Funktionen sowohl auf Skalare als auch auf Matrizen anwendbar. Es gibt jedoch dennoch die Klasse der skalaren Funktionen und die der array-Funktionen. Letztere machen nur Sinn im Gebrauch mit Feldern (Vektoren, Matrizen). Die folgende Tabelle zeigt nur einen kleinen Teil der skalaren Funktionen:

Matlab-Bezeichnung	math. Syntax	Erläuterung
`exp`	$exp()$	Exponentialfunktion
`log`, `log10`	$ln()$, $log_{10}()$	Logarithmusfunktionen
`sqrt`	$\sqrt{}$	Wurzelfunktion
`mod`	$mod(\ ,\ )$	Modulo-Funktion
`sin`, `cos`, `tan`	$sin()$, $cos()$, $tan()$	trig. Funktionen
`sinh`, `cosh`, `tanh`	$sinh()$, $cosh()$, $tanh()$	trig. Funktionen
`asin`, `acos`, `atan`	$arcsin()$, $arcos()$, $arctan()$	trig. Funktionen
`abs`	$\mid\ \mid$	Absolutbetrag
`imag`	$\Im()$	Imaginärteil
`real`	$\Re()$	Realteil
`conj`		konjugieren
`sign`		Vorzeichen
`round,floor,ceil`		Runden (zur nächsten ganzen Zahl, nach unten, nach oben)

Die MATLAB Hilfe enthält eine alphabetische Liste aller Funktionen!

Funktionen können sowohl auf skalare Größen als auch auf Matrizen angewandt werden:

```
>> exp(0)

ans =

     1
>> exp(-inf)

ans =

     0
>> x =  [ 1 2 4];
>> exp(x)

ans =
```

```
   2.7183    7.3891   54.5982
```

Die Funktion wird dann komponentenweise angewandt.

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 werden in der folgenden Tabelle erläutert:

Matlab-Bezeichnung	Beschreibung
`max`	größte Komponente
`mean`	Durchschnittswert, Mittelwert
`min`	kleinste Komponente
`prod`	Produkt aller Elemente
`sort`	Sortieren der Elemente eines Feldes in ab- oder aufsteigender Ordnung
`sortrows`	Sortieren der Zeilen in aufsteigend Reihenfolge
`std`	Standardabweichung
`sum`	Summe aller Elemente
`trapz`	numerische Integration mit der Trapezregel
`transpose`	transponieren
`det`, `inv`	Determinante, Inverse einer Matrix
`diag`	Diagonalen von Matrizen (s.o.)
`fliplr, flipud`	Spiegelungen von Matrizen (s.o.)
`any, all`	Testet ob mindestens ein oder alle Komopnenten ungleich Null sind

Weitere Funktionen findet man in der MATLAB -Hilfe. Die meisten Funktionen können auch z.B. zeilenweise, nur auf bestimmte Felder der Matrix oder auch auf mehrere Matrizen angewandt werden. Details dazu findet man ebenfalls in der Hilfe.

3.3 Relationsoperatoren und Logische Operatoren

3.3.1 Relationsoperatoren

Die Relationsoperatoren sind wie folgt definiert:

Matlab-Syntax	mathematische Syntax
`A > B`	$A > B$
`A < B`	$A < B$
`A >= B`	$A \geq B$
`A <= B`	$A \leq B$
`A == B`	$A = B$
`A ~= B`	$A \neq B$

Die Relationsoperatoren sind auf skalare Größen, aber auch auf Matrizen und Vektoren anwendbar. Bei Matrizen und Vektoren vergleichen die Relationsoperatoren die einzelnen Komponenten. A und B müssen demnach die gleiche Dimension haben. MATLAB antwortet komponentenweise mit den *booleschen Operatoren*, d.h. mit 1 (*true*), falls eine Relation stimmt und mit 0 (*false*), falls nicht. Beispiel (skalar):

```
>> 5>2

ans =

     1

>> 4 ~= 4

ans =

     0

>> 5==abs(5)

ans =

     1
```

Beispiel (vektoriell):

```
>> x = [ 1 2 3];
>> y = [-1 4 6];
```

```
>> z = [1 0 -7];
>> y > x

ans =

     0     1     1

>> z == x

ans =

     1     0     0
```

Die Relationsoperatoren können auch auf Felder angewandt werden. Seien Vektoren `x=[1 2 3 4 5]` und `y=[-5 3 2 4 1]` gegeben, so liefert der Befehl

```
>> x(y>=3)

ans =

     2     4
```

Der Befehl `y>=3` liefert das Ergebnis `0 1 0 1 0` und `x(y>=3)` gibt dann die Stellen von `x` aus, an denen `y>=3` den Wert 1 hat, also *true* ist.

Bemerkung: Vorsicht bei der Verwendung von Relationsoperatoren auf die komplexen Zahlen. Die Operatoren `>`, `>=`, `<` und `<=` vergleichen nur den Realteil! Dagegen werten `==` und `~=` Real- und Imaginärteil aus!

3.3.2 Logische Operatoren

Es sind die logische Operatoren *und* (`&`), *oder* (`|`), *nicht* (`~`) und das *ausschließende oder* (`xor`) in MATLAB integriert. Die Wahrheitstafel für diese Operatoren sieht wie folgt aus:

		und	oder	nicht	ausschließendes oder
`A`	`B`	`A & B`	`A|B`	`~A`	`xor(A,B)`
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

Wie die Relationsoperatoren sind die logischen Operatoren auf Vektoren, Matrizen und skalare Größen anwendbar, sie können ebenfalls nur auf Matrizen und Vektoren gleicher Dimension angewandt werden. Die logischen Operatoren schauen komponentenweise nach, welche Einträge der Matrizen 0 und ungleich 0 sind. Die 1 steht wiederum für 'true', die 0 für 'false'. Am besten lassen sich die logischen Operatoren anhand von Beispielen erläutern. Seien z.B. die Vektoren

```
>> u = [0 0 1 -3 0 1];
>> v = [0 1 7 0 0 -1];
```

gegeben. Der Befehl `~u` gibt dann an den Stellen, an denen u gleich 0 ist, eine 1 aus (aus false wird true) und an den Stellen, an denen u ungleich 0 ist, eine 0 (aus true wird false):

```
>> ~u

ans =

     1     1     0     0     1     0
```

Ebenso funktionieren die Vergleiche. `u&v` liefert komponentenweise eine 1, also true, falls sowohl u als auch v in der Komponente beide $\neq 0$ sind und anderenfalls eine 0. Welchen Wert die Komponenten, die $\neq 0$ sind, genau annehmen, ist hierbei irrelevant.

```
>> u & v

ans =
```

```
     0     0     1     0     0     1
```

`u|v` liefert komponentenweise eine 1, falls u oder v in einer Komponente $\neq 0$ sind und eine 0, falls die Komponente in beiden Vektoren gleich 0 ist. Welchen Wert die Komponenten, die $\neq 0$ sind, genau annehmen, ist hierbei wiederum irrelevant. Das ausschließende oder `xor` steht für den Ausdruck 'entweder ... oder'. `xor(u,v)` liefert eine 1 in den Komponenten, in denen entweder u oder v (nicht aber beides zugleich!) ungleich 0 ist und eine 0, falls die Komponente in beiden Vektoren 0 oder $\neq 0$ ist.

Bemerkung: MATLAB wertet Befehlsketten von links nach rechts aus: `~u|v|w` ist gleichbedeutend mit `((~u)|v)|w`. Der Operator `&` hat jedoch oberste Priorität. `u|v&w` ist gleichbedeutend mit `u|(v&w)`.

Diese Erläuterung der logischen Operatoren ist sehr formell. Nützlich werden die logischen Operatoren im Gebrauch mit den Relationsoperatoren, um später z.B. Fallunterscheidungen programmieren zu können. Möchte man überprüfen, ob zwei Fälle gleichzeitig erfüllt werden, z.B. ob eine Zahl $x \geq 0$ und eine weitere Zahl $y \leq 0$ ist, kann dies der Befehl `x>=0 & y<=0` prüfen. Nur wenn beides wahr ist, wird dieser Befehl die Ausgabe 'true' hervorrufen.

Short-circuit Operatoren:

Es gibt die so genannten *Short-circuit Operatoren* (Kurzschluss-Operatoren) `&&` und `||` für skalare Größen, diese entsprechen zunächst dem logischen *und* `&` und dem logischen *oder* `|`, sie sind jedoch effizienter. Bei einem Ausdruck

```
expr_1 & expr_2 & expr_3 & expr_4 & expr_5 & expr_6
```

testet MATLAB alle Ausdrücke und entscheidet dann, ob er 1 (= true) oder 0 (= false) ausgibt. Schreibt man hingegen

```
expr_1 && expr_2 && expr_3 && expr_4 && expr_5 && expr_6
```

so untersucht MATLAB zuerst `expr_1`. Ist dies schon *false*, so gibt MATLAB sofort ein *false*, ohne die weiteren Ausdrücke zu untersuchen. Analoges gilt für `||`. Vorsicht, die Short-circuit Operatoren sind nur auf skalare Größen anwendbar! Gerade bei längeren Rechnungen können sie aber Zeit einsparen!

3.4 Polynome (Beispiel für die Anwendung von Arrays)

Polynome können in MATLAB durch Arrays dargestellt werden, allgemein kann ein Polynom $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x^1 + a_0$ durch

```
>> p = [a_n a_n-1 a_n-2 ... a_1 a_0];
```

dargestellt werden. Das Feld mit dem Koeffizienten muss mit dem Koeffizienten der höchsten Ordnung beginnen. Fehlende Potenzen werden durch 0 deklariert. MATLAB kann mit dieser Schreibweise Nullstellen von Polynomen berechnen und auch Polynome miteinander multiplizieren und dividieren. Das Polynom $p(x) = x^3 - 15x - 4$ hat die Darstellung

```
>> p = [1 0 -15 -4];
>> r = roots(p)

ans =

    4.0000
   -3.7321
   -0.2679
```

gibt die Nullstellen von p aus. Sind umgekehrt nur die Nullstellen gegeben, kann mit dem Befehl `poly` das Polynom zu den Nullstellen berechnet werden.

```
>> r= [ 2 5 1]

r =

     2     5     1

>> p=poly(r)

p =

     1    -8    17   -10
```

Dies entspricht dem Polynom $p(x) = x^3 - 8x^2 + 17x - 10$. Weiterhin kann man sich einfach Funktionswerte eines Polynoms ausgeben lassen. Man wählt einen Bereich aus, für den man die Wertetabelle des Polynoms berechnen will, z. B. für das Polynom $p(x) = x^3 - 8x^2 + 17x - 10$ den Bereich $[-1, 7]$, in dem die Nullstellen liegen. Hier wählen wir beispielsweise die Schrittweite 0.5 und erhalten mit dem Befehl `ployval` eine Wertetabelle mit 17 Einträgen.

```
>> x=-1:0.5:7;
>> y=polyval(p,x)

y =

  Columns 1 through 6

  -36.0000  -20.6250  -10.0000   -3.3750         0    0.8750

  Columns 7 through 12

         0   -1.8750   -4.0000   -5.6250   -6.0000   -4.3750

  Columns 13 through 17

         0    7.8750   20.0000   37.1250   60.0000
```

Polynommultiplikation und -division werden mit dem Befehlen `conv( , )` und `deconv( , )` berechnet.

4 Programmierung in Matlab

4.1 Das M-File

Es ist sehr unübersichtlich, alle Berechnungen im *Command Window* durchzuführen. Matlab kann Dateien (so genannte *M-Files*) erstellen, in denen man seine Befehle speichern kann. Über das Menü **File** → **New** → **M-File** kann eine neue Datei geöffnet werden. Die Befehle werden nun in diese Datei geschrieben, die Datei unter einem Namen abgespeichert. Sie erhält die Matlab-Endung `.m`. Der Aufruf der Datei geschieht im *Command Window* durch Aufruf des Dateinamens. Die Endung `.m` muss dabei nicht angefügt werden. Alternativ kann man auch im *Editor* auf den grünen Pfeil  klicken um das Programm auszuführen. Im Folgenden wird ein einfaches Programm besprochen. Zunächst wird der gesamte Quellcode angegeben, anschließend werden die einzelnen Befehlszeilen erläutert. Das Programm `Einheitsmatrix.m`, welches zu einer gegebenen Zahl n die $n \times n$-Einheitsmatrix ausgibt, könnte z.B. so aussehen:

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% Dieses Programm mit dem Namen Einheitsmatrix.m liest
% eine Zahl n ein und berechnet dann die (n,n)
% Einheitsmatrix.
% % % % % % % % % % % % % % % % % % % % % % % % % % % % %

disp('Dieses Programm liest eine Zahl n ein ')
disp('und berechnet dann die (n,n) Einheitsmatrix.');
disp(' ');
n = input('Bitte nun eine natuerliche Zahl n eingeben: ');
disp(' ');
disp(['Die von Ihnen eingegebene Zahl war n = ',num2str(n)]);

% Berechnung der n. Einheitsmatrix:

A = eye(n);

disp('Die zugehoerige Einheitsmartix ist:')
disp(' ')
disp(A)
```

Kommentare werden mit `%` versehen. Alles, was in einer Zeile nach einem `%` folgt, ist ein Kommentar. Es ist wichtig, jedes Programm zu kommentieren. Speichert man das oben geschriebene Programm unter dem (noch nicht in der Matlab-Funktionen-Datenbank vorhandenen) Titel `Einheitsmatrix.m`, so gibt der Befehl

```
>> help Einheitsmatrix
```

das Kommentar

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % %
Dieses Programm mit dem Namen Einheitsmatrix.m liest
eine Zahl n ein und berechnet dann die (n,n)
Einheitsmatrix.
% % % % % % % % % % % % % % % % % % % % % % % % % % % % %
```

aus. Es empfiehlt sich, zu jedem Programm, dass man schreibt, eine solche Erläuterung zu schreiben und wirklich jeden Schritt im Programm zu kommentieren. Wenn die ersten Zeilen eines M-Files aus einem Kommentar bestehen, gibt der Befehl `help Dateiname.m` diese Zeilen aus. Alle in Matlab enthaltenen Funktionen haben eine kurze Erläuterung als Kommentar in den ersten Zeilen stehen.

Textausgaben mit dem Text ... im *Command Window* werden mit Hilfe des Befehls `disp(' ... ')` erreicht. `disp(' ')` verursacht eine Leerzeile, was das Programm manchmal übersichtlicher macht. Das Programm kann auch Zahlen und Berechnungen des Programms im Fließtext ausgeben, in dem man den Befehl `disp([' '])` wie oben angewendet in Kombination mit `num2str()` benutzt:

```
disp(['Die von Ihnen eingegebene Zahl war n = ',num2str(n)]);
```

Der Befehl `num2str` verwandelt eine Zahl oder eine Matrix in eine Zeichenkette, die ausgegeben werden kann.

Parameter können entweder direkt im M-File definiert werden, oder wie oben mittels `input(' ')` eingelesen werden:

```
n = input('Bitte nun eine natuerliche Zahl n eingeben: ');
```

Der Text `Bitte nun eine natuerliche Zahl n eingeben`: erscheint im *Command Prompt*, der Variablen `n` wird der Wert der eingegebenen Zahl zugewiesen.

Führt man nun das Programm `Einheitsmartix.m` aus und gibt die Zahl $n = 5$ ein, so sieht der Programmablauf wie folgt aus:

```
>> Einheitsmatrix
Dieses Programm liest eine Zahl n ein
und berechnet dann die (n,n) Einheitsmatrix.

Bitte nun eine natuerliche Zahl n eingeben: 5

Die von Ihnen eingegebene Zahl war n = 5
Die zugehoerige Einheitsmartix ist:

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

>>
```

Weiterin ist es nicht nur möglich den Benutzer mit `disp` zu informieren sondern auch für Ausnahmesituation mit dem Befehl `warning` welches in der Ausgabe durch Farbe oder Schriftbild hervorgehoben wird.

```
warning('Warnung: Es wurde keine naturliche Zahl eingegben, verwende Rundung');
```

Sollte es nötig sein ein Programm aufgrund eines Fehlers abzubrechen so kann man dies mit `error` erreicht werden. Durch den `error` Befehl wird eine deutlich kenntlich gemachte Nachricht in der Ausgabe angezeigt, die Funktion abgebrochen und kein Rčukgabewert übergeben.

```
error('Fehler: Es wurde keine naturliche Zahl eingegeben!');
```

Es ist guter Programmierstil Warn- und Fehlermeldungen mit dem Wort “Warnung” bzw “Fehler” einzuleiten.

4.2 Funktionen

Wie schon in Kapitel 4.1 angedeutet, sollten längere Berechnungen oder Sequenzen von MATLAB Kommandos in M-Files durchgeführt werden. Oft ist es sinnvoll, eigene Funktionen zu programmieren, die dann in einem Programmdurchlauf ausgeführt werden. Es gibt dann ein Hauptprogramm, in welchem mehrere Funktionen aufgerufen werden. Funktionen werden genauso wie gewöhnliche M-Files geschrieben. Der einzige Unterschied besteht darin, dass das erste Wort `function` sein muss. In der ersten Zeile wird dann der Name der Funktion definiert und es werden die Variablen eingelesen. Als Beispiel soll nun ein kleines Programm dienen, welches zwei Werte a und b einliest und dann Berechnungen zu dem Rechteck $a \cdot b$ durchführt. Dazu schreiben wir zunächst die Funktion `Flaecheninhalt_ Rechteck.m`.

```
function A = Flaecheninhalt_Rechteck(a,b)

% Flaecheninhalt_Rechteck(a,b) berechnet den Flaecheninhalt
% des Rechtecks mit den Seiten a und b

A = a * b;
```

Die Variablen unmittelbar hinter dem `function`, hier also `A`, bezeichnen die Werte, die berechnet und ausgegeben werden. Die Variablen hinter dem Funktionsnamen in den runden Klammern bezeichnen die Werte, die eingelesen werden. Weiterhin soll die Diagonale des Rechtecks mit der Funktion `Diagonale_ Rechteck.m` berechnet werden.

```
function d = Diagonale_Rechteck(a,b)

% Diagonale_Rechteck(a,b) berechnet die Diagonale des Rechtecks
% a*b mit Hilfe des Satzes von Pythagoras.

d = sqrt((a^2) + (b^2));
```

Das Hauptprogramm `Rechteck.m` könnte nun so aussehen:

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % %
% Das Programm Rechteck.m liest zwei Werte a und b ein
% und berechnet den Flaecheninhalt und die Diagonale
% des Rechtecks a*b
% % % % % % % % % % % % % % % % % % % % % % % % % % % %

disp('Dieses Programm liest zwei Werte a und b ein und berechnet')
disp('den Flaecheninhalt und die Diagonale des Rechtecks a*b')

disp(' ')

a = input('Bitte a eingeben: ');
b = input('Bitte b eingeben: ');

A = Flaecheninhalt_Rechteck(a,b);
d = Diagonale_Rechteck(a,b);

disp(['Der Flaecheninhalt des Rechtecks ist A = ',num2str(A)])
disp(['und die Diagonale betraegt d = ',num2str(d)])
```

Es ist sehr wichtig, dass das Hauptprogramm und die darin aufgerufenen Funktionen im selben Ordner liegen. Liegt zum Beispiel die Funktion `Diagonale_ Rechteck.m` nicht im selben Ordner wie das Hauptprogramm `Rechteck.m`, so gibt MATLAB nach der Eingabe von `a` und `b` folgenden Fehler aus:

```
??? Undefined function or method 'Diagonale_Rechteck' for input arguments of
type 'double'.

Error in ==> Rechteck at 16
 d = Diagonale_Rechteck(a,b);
```

Das heißt, dass MATLAB uns mitteilt, dass die Funktion `Diagonale_ Rechteck.m` nicht definiert ist. Bei so kleinen Programmen erscheint es noch nicht wirklich sinnvoll, Unterroutinen als Funktionen zu schreiben. Dies ist aber bei komplizierteren Programmen und besonders wenn eine Routine häufig benutzt werden muss, sehr hilfreich. Sollen mehrere Werte innerhalb einer Funktion berechnet werden, schreibt man

```
function [A, B, C] = Funktions_Name(a,b,c,d,e);
```

Die Variablen hinter dem Funktionsnamen in den runden Klammern bezeichnen die Werte, die eingelesen werden. In den eckigen Klammern nach dem Wort `function` stehen die Variablen, die von der Funktion nach dem Funktionsdurchlauf wieder zurückgegeben werden. Also die Werte, die in der Funktion berechnet werden sollen.

Sollte es nötig sein eine Funktion vor der letzten Zeile des Funktionskörpers abzubrechen, so benutzt man dazu den Befehl `return`. Damit wird der aktuelle Wert im Funktionskopf deklarierten Rückgabewert an den Aufrufer zurückgegeben.

Vorsicht bei der Vergabe von Funktionsnamen! Der Name des M-Files darf keine Sonderzeichen, sprich Zeichen, die entweder bereits MATLAB -Operatoren sind ("+", "(", ")", "-"), oder Umlaute (MATLAB basiert auf der englischen Sprache) enthalten. Möchte man mehrere Wörter verwenden und diese trennen, so kann man dies mit den Zeichen "_" oder "." tun. Ein Beispiel: Dies_wird_ein.korrekt.bezeichnetes.MFile . Sehr viele Fehler entstehen auch durch eine doppelte Vergabe eines Funktionsnamens. Ob der von mir gewählte Name bereits vergeben ist, kann ich mit Hilfe der Funktion `exist` überprüfen.

Ausgabe	Bedeutung
0	Name existiert noch nicht
1	Name ist bereits für eine Variable im *Workspace* vergeben
2	Name ist ein bereits bestehendes M-file oder eine Datei unbekannten Typs
3	Es existiert ein Mex-File mit diesem Namen
4	Es existiert ein MDL-file mit diesem Namen
5	Name ist an eine Matlab Funktion vergeben (z.B. `sin`)
6	Es existiert ein P-file mit diesem Namen
7	Es existiert ein Verzeichnis mit diesem Namen

Beispiele:

```
>> exist d              >> exist cos            >> exist hallo

ans =                   ans =                   ans =

     1                       5                       0
```

4.3 Function handles und anonyme Funktionen

4.3.1 Function handles

Bisher wurde nur der herkömmliche Funktionsaufruf besprochen. In einem gesonderten M-File wird eine Funktion gespeichert, welche dann in einem Hauptprogramm ausgeführt werden kann. Die Funktion enthielt Variablen als Argumente.

Es ist auch möglich, Funktionen zu programmieren, deren Argumente andere Funktionen sind. Dies kann zum Beispiel nötig sein, wenn eine numerische Approximation einer Ableitung programmieren werden soll. Es seien mehrere Funktionen, z.B. $f_1(x) = \sin(x)$, $f_2(x) = x^2$, $f_3 = \cos(x)$ gegeben und wir wollen die Ableitungen dieser Funktionen in einem festen Punkt $x = 1$ mit Hilfe der Approximation

$$f'(x) \approx D^+ f(x) := \frac{f(x+h) - f(x)}{h}, \quad h > 0 \tag{4.1}$$

für eine fest vorgegebene Schrittweite h berechnen. Dann könnte man für jede der Funktionen f_1, f_2, f_3 den Wert $D^+ f(x)$ berechnen:

```
x=1;
h=0.001;
```

```
Df_1 = (sin(x+h)-sin(x))/h;
Df_2 = ((x+h)^2-x^2)/h;
Df_3 = (cos(x+h)-cos(x))/h;
```

Übersichtlicher ist es, wenn man eine Routine `numAbleitung` programmiert, die eine vorgegebene Funktion f, einen Funktionswert x und eine Schrittweite h einliest und damit die näherungsweise Ableitung $f'(x)$ nach Gleichung (4.1) berechnet. Die Routine müsste dann nur einmal programmiert werden und könnte für alle Funktionen genutzt werden. Dies bedeutet aber, dass man die Funktion f als Argument der Funktion `numAbleitung` übergeben muss. Das kann Matlab mit Hilfe von *Zeigern* realisieren.

Zunächst muss die Funktion f programmiert werden (im Beispiel betrachten wir nur die oben genannte Funktion f_1), welche eine Zahl x einliest und den Funktionswert $f(x)$ zurück gibt:

```
function y = f(x)
y = sin(x);
```

Diese speichern wir als M-File unter dem Namen `f.m`. Dann wird eine Funktion `numAbleitung` geschrieben, die wie gefordert f, x und h einliest und die Ableitung von f im Punkt x approximiert. In dieser Funktion kann das Argument f wie eine Variable normal eingesetzt werden:

```
function Df = numAbleitung(f,x,h)
% Diese Routine berechnet den Differenzenquotienten einer Funktion f im Punkt x
% mit Hilfe von Gleichung (1). Dabei ist f ein Zeiger auf diese Funktion
% (was jedoch an der herkoemmlichen Syntax hier nichts aendert).

Df = (f(x+h)-f(x))/h;
```

Die Routine wird als M-File `numAbleitung.m` abgespeichert. Nun muss noch das Hauptprogramm `Abl.m` geschrieben werden, in dem die Funktionen `f` und `numAbleitung` aufgerufen werden. Bei dem Funktionsaufruf `numAbleitung` im Hauptprogramm muss allerdings berücksichtigt werden, dass eines der zu übergebenden Argumente eine Funktion ist. Es wird nicht die Funktion selbst, sondern ein *Zeiger* auf die Funktion übergeben. Dieser Zeiger ist vom Datentyp *Function handle*. Wir erzeugen ihn, indem wir entweder `fhandle = @f` definieren oder vor dem Funktionsnamen das Zeichen `@` angefügen.

```
% Programm, welches fuer gegebene Werte x eine Funktion f aufruft und dann
% mit Hilfe der Funktion numAbleitung.m die Ableitungen von f in diesen x
% bestimmt

% Festlegung der Werte x aus dem Intervall [0,4]
```

```
h=0.02;
x = 0:h:4;
% Berechne f(x) mit der Funktion f.m:
y = f(x);
% Berechne die Ableitung von f auf dem Intervall [0,4]
% D.h. Dy ist ein Vektor der gleichen Laenge wie x.
Dy = numAbleitung(@f,x,h);
```

Wird die Funktion `f` als Argument der Funktion `numAbleitung` aufgerufen, so setzen wir das Zeichen `@` davor. Die Routine `numAbleitung` kann nun zur näherungsweisen Ableitung aller differenzierbaren Funktionen $f : \mathbb{R} \to \mathbb{R}$ verwendet werden.

4.3.2 Anonyme Funktionen

Insbesondere bei größeren Projekten kann dieses Vorgehen jedoch leicht unübersichtlich werden. Damit nicht für jede mathematische Funktion ein neues M-File geschrieben werden muss, erlaubt MATLAB die Definition sogenannter *anonymer Funktionen*, die bereits vom Datentyp Function handle sind. Um wie im obigen Beispiel die numerische Ableitung von $sin(x)$ zu berechnen, schreiben wir nicht ein eigenes M-File `f.m`, sondern definieren im Hauptprogramm

```
f = @(x) sin(x);
```

Unser neues Hauptprogramm `Ableitung.m` sieht nun so aus:

```
% Programm, welches fuer gegebene Werte x und eine gegebene Funktion f
% mit Hilfe der Routine/Funktion numAbleitung.m die Ableitungen von f
% in diesem Punkt x bestimmt

% Definition von f als anonyme Funktion

f = @(x) sin(x);

% Festlegung der Werte x aus dem Intervall [0,4]

h = 0.02;
x = 0:h:4;

% Berechne die Ableitung von f auf dem Intervall [0,4]
% D.h. Dy ist ein Vektor der gleichen Laenge wie x.

Dy = numAbleitung(f,x,h);
```

Da `f` nun schon vom Datentyp Function handle ist, müssen wir keinen Zeiger mehr verwenden, um `numAbleitung` aufzurufen. Allerdings muss bei der Verwendung von anonymen Funktionen außer bei Funktionsaufrufen immer die jeweilige Variable mitgeführt werden. Definieren wir zum Beispiel die anonymen Funktionen `f = @(x) sin(x).^2` und `g = @(x) cos(x).^2`, so schreiben wir bei deren Addition `h = @(x) f(x) + g(x)`. Schreiben wir nur `h = @(x) f(x) + g` und werten `h` anschließend aus, so gibt MATLAB die folgende Fehlermeldung aus:

```
??? Undefined function or method 'plus' for input arguments of type
'function_handle'.

Error in ==> @(x)f(x)+g
```

Analog muss natürlich auch bei der Multiplikation, der Hintereinanderausführung von anonymen Funktionen,... immer die Variable mitgeführt werden. Falls wir anonyme Funktionen definieren wollen, die von $\mathbb{R}^n$ nach $\mathbb{R}$ abbilden, so geht das im Fall von $\mathbb{R}^2 \to \mathbb{R}$ wie folgt:

```
f = @(x,y) x^2 + y^2;
```

Falls man eine Funktion definieren möchte, die zum Beispiel von $\mathbb{R}^2 \to \mathbb{R}^2$ abbildet, so definiert man

```
f = @(x,y) [x+y;x-y];
```

4.3.3 Abschnittsweise definierte Funktionen

Häufig begegnen wir Funktionen, die abschnittsweise definiert sind. Mit Hilfe der logischen Operatoren, die wir in Kapitel 3 kennengelernt haben, können wir solche Funktionen nun entweder als Funktion in einem neuen M-File oder vorzugsweise als anonyme Funktion definieren. Da die erste Variante beim Programmieren mit MATLAB nicht verwendet werden sollte, werden wir hier nur die zweite Variante anhand eines Beispiels vorstellen. Wir möchten folgende Funktion abschnittsweise definieren und anschließend plotten:

$$f(x) = \begin{cases} x-2 & 2 \leq x \leq 3, \\ 4-x & 3 < x \leq 4, \\ 0 & \text{sonst.} \end{cases}$$

Ein entsprechendes M-File (abschnittsweiseFunk.m) könnte zum Beispiel so aussehen:

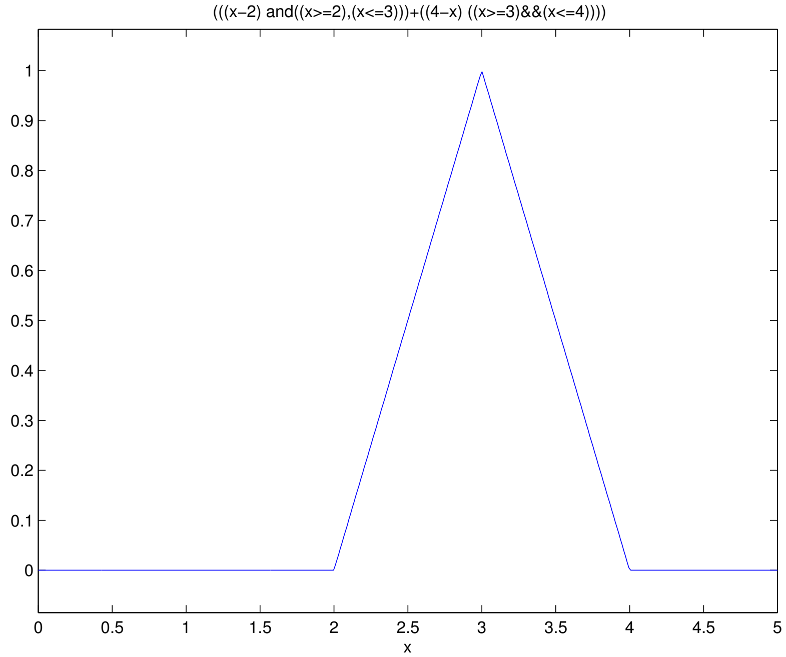

Abbildung 4.1: Plot der Hütchenfunktion

```
% In diesem M-File wird eine sogenannte Huetchenfunktion abschnittsweise
% definiert. Dazu verwenden wir logische Operatoren.

% Wir definieren eine abschnittsweise definierte Huetchenfunktion f mit
% Hilfe von logischen Operatoren

f = @(x) (((x - 2) .* and((x >= 2),(x <= 3))) ...
        + ((4 - x) .* ((x > 3)&&(x <= 4))));

% Um zu schauen, ob wir alles richtig gemacht haben, plotten wir die
% Funktion. Dazu verwenden wir ezplot. ezplot ist ein sehr einfacher Befehl
% um Funktionen zu plotten. Wir geben in Klammern die Funktion an
% (und wenn gewuenscht die Intervallgrenzen a und b ueber die die
% Funktion geplottet werden soll).

% ezplot(funktion,[a,b])

ezplot(f,[0,5]);
```

Das Ergebnis sieht dann wie in Abbildung 4.1 aus. Wie man graphische Ausgaben beschriftet lernen wir in Kapitel 5.

4.4 Entscheidungen und Schleifen

In diesem Kapitel sollen kurz die wichtigsten Entscheidungen `if` und `switch` und Schleifen `for` und `while` vorgestellt werden.

4.4.1 Entscheidung: `if`

Die (bedingte) Anweisung `if` wertet einen logischen Ausdruck aus und verzweigt zu einer Gruppe von Anweisungen, sofern der Ausdruck wahr ist. Die Syntax lautet wie folgt

```
if Ausdruck 1
  Anweisungen 1
elseif Ausdruck 2
  Anweisungen 2
else
  Anweisungen 3
end
```

Ist der Ausdruck 1 wahr, so werden unmittelbar die folgenden Anweisungen 1 ausgeführt. Andernfalls wird der Ausdruck 2 der nachfolgenden elseif-Anweisung geprüft und, falls dieser wahr ist, die Anweisungen 2 ausgeführt. Sind alle logischen Ausdrücke falsch, werden die Anweisungen 3 des else-Zweigs ausgeführt. Die Anzahl der elseif-Zweige ist beliebig. Deren Angabe kann ebenso wie der else-Zweig entfallen.

Das folgende Beispiel zeigt, wie die `if`-Anweisung angewandt werden kann. Für eine eingegebene Zahl n soll ausgegeben werden, ob n gerade oder ungerade ist. Als Hilfe dient hierzu die MATLAB Funktion `mod(x,y)`, welche den mathematischen Ausdruck x mod y berechnet.

```
% % % % % % % % % % % % % % % % % %
% gerade.m gibt aus, ob eine
% eingegebene Zahl gerade oder
% ungerade ist.
% % % % % % % % % % % % % % % % % %

clear all

n = input('Bitte eine ganze Zahl eingeben: ')
```

```
if mod(n,2)==0
   disp('Die eingegebene Zahl ist gerade!')
elseif mod(n,2)==1
   disp('Die eingegebene Zahl ist ungerade!')
else
   disp('Die eingegebene Zahl ist keine ganze Zahl!')
end
```

Die `if`-Anweisung eignet sich auch gut um größere Codeteile auszukommentieren. Insbesondere bei längeren M-Files kann dies sehr hilfreich sein. Das M-File `auskommentieren_mit_if.m` zeigt wie man einfach Teile eines Codes aus- und einschalten (also auskommentieren und dann die Kommentare wieder entfernen) kann.

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% auskommentieren_mit_if.m zeigt wie man mit Hilfe der
% if-Entscheidung groessere Code-Teile einfach auskommentieren
% kann.
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

% Zuerst definieren wir eine Variable a

a = 5;

% Soll nun ein Teil eines M-Files auskommentiert werden, so weist man
% zunaechst einer Hilfsvariable aux den Wert 0 zu.

aux = 0;

% Der auszukommentierende Codeteil wird dann mit der if-Anweisung
% if aux == 1 auskommentiert.

if aux == 1

    a = a + 2; % Dieser Teil soll auskommentiert werden!

end

% Wir geben a aus und stellen fest, dass der Codeteil a = a + 2 nicht
% ausgeführt wurde.

a
```

```
% Wollen wir nun diesen Codeteil "anschalten", setzen wir aux einfach auf 1

aux = 1;

if aux == 1

    a = a + 2; % Dieser Teil wird jetzt ausgeführt!

end

% Zum Test lassen wir uns wieder a ausgeben und stellen fest, dass der
% Codeteil a = a + 2 dieses Mal ausgeführt wurde.

a
```

4.4.2 Fallunterscheidung: `switch`

Stellen wir uns die folgende Problemstellung vor: Je nach Wert einer Variablen soll eine bestimmte Gruppe von Anweisungen ausgeführt werden. Das entsprechende Konstrukt heißt `switch`. Die unterschiedlichen Fälle werden durch `case` festgelegt. Dabei wird nur die erste Übereinstimmung ausgeführt. Wird keiner der durch `case` definierten Fälle erfüllt, wird ähnlich dem Schlüsselwort `else` unter `if` durch `otherwise` eine alternative Gruppe von Anweisungen definiert. Die Syntax lautet:

```
switch Variable

case Fall 1
  Anweisungen 1
case Fall 2
  Anweisungen 2
      ⋮
otherwise
  sonstige Anweisungen
end
```

Da die vereinfachte `switch`-Anweisung rascher abgearbeitet wird als `if` – `elseif` – `else`-Abfragen, sollte, wenn möglich, die `switch`-Anweisung verwendet werden. So können und sollten wir auch unser Programm `gerade.m` mit Hilfe der `switch`-Anweisung schreiben. Wir erhalten das M-File `geradeswitch.m`.

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% geradeswitch.m gibt aus, ob eine eingegebene Zahl gerade
% oder ungerade ist und verwendet dazu aus
% Effizienzgruenden die switch-Anweisung.
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % %

clear all

n = input('Bitte eine ganze Zahl eingeben: ')

switch mod(n,2)

case 0
   disp('Die eingegebene Zahl ist gerade!')
case 1
   disp('Die eingegebene Zahl ist ungerade!')
otherwise
   disp('Die eingegebene Zahl ist keine ganze Zahl!')
end
```

4.4.3 Die `for`-Schleife

Die `for` - Schleife eignet sich, wenn Berechnungen wiederholt durchgeführt werden. Es sei vorab erwähnt, dass das Arbeiten mit Schleifen sehr einfach, jedoch auch oft sehr ineffizient ist, wie wir später sehen werden. Die for-Schleife zur n-fachen Ausführung einer Befehlssequenz besitzt die folgende Syntax:

> `for` Variable = Matrix/Zelle/Array
>   Anweisungen
> `end`

Der Variable werden nacheinander die Spalten der Matrix bzw. der Zelle oder dem Array zugewiesen. Im Falle mehrdimensionaler Arrays werden analog die Spalten aller Teilmatrizen durchlaufen. Für jede Spalte werden die Befehle einmal ausgeführt. Ein n-facher Schleifendurchlauf kann mittels `for index=1:n` realisiert werden, wie das folgende einfache Beispiel zeigt.

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
% for_hoelder_mittel.m berechnet das Hoelder-Mittel der
% Stufe k der Zahlen 1 bis n. Das Hoelder-Mittel der Stufe k
% der Zahlen x_1 bis x_n ist wie folgt definiert:
% Hoelder-Mittel = ((1/n)*sum_{i = 1}^{n} x_i^k)^(1/k)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Waehle die Zahl n

n = 10;

% Waehle die Stufe k

k = 1;

% Initialisiere summe

summe = 0;

% Berechne die Summe der Potenzen x_i^k fuer i = 1:n

for index = 1:n

    summe = summe + index.^k;

end

% Definiere das Hoelder Mittel

hoelder = ((1/n).*summe)^(1/k)
```

Wichtig ist hierbei, dass die Variable `summe` vor der Schleife initialisiert wird (d.h., dass sie vorher definiert und einen Wert zugewiesen bekommt), da sonst in der Schleife beim ersten Durchlauf auf eine bis dahin unbekannte Variable zugegriffen wird, was natürlich zu einer Fehlermeldung führt. Als weiteres Beispiel (`Ableitungfor.m`) wollen wir uns nochmal die Berechnung der numerischen Ableitung anschauen. Dies können wir nun mittels einer `for`-Schleife wesentlich effizienter realisieren. Um die Zeiten vergleichen zu können, die der Computer benötigt um mit den unterschiedlichen Methoden die numerische Ableitung zu berechnen, verwenden wir die Befehle `tic` und `toc`.

```
% Programm, welches fuer gegebene Werte x und eine gegebene Funktion f
% einerseits mit Hilfe der Routine/Funktion numAbleitung.m die Ableitungen
% von f in diesem Punkt x bestimmt und zum Vergleich das Ganze auch
% mit einer Schleife berechnet

% Definition von f als anonyme Funktion
```

```
f = @(x) sin(x);

% Festlegung der Werte x aus dem Intervall [0,4]

h=0.02;
x=0:h:4;

% Berechne die Ableitung von f auf dem Intervall [0,4]
% D.h. Dy ist ein Vektor der gleichen Laenge wie x. Die Zeit messen wir mit
% den Befehlen tic und toc.

tic;

Dy = numAbleitung(f,x,h);

toc;
t1 = toc;

% Zur Kontrolle plotten wir die Ableitung

%plot(Dy);

% Bei der Berechnung von Dy wird fuer jeden Eintrag die Funktion
% numAbleitung(f,x,h) aufgerufen. Dies ist ineffizient. In der Funktion
% numAbleitung(f,x,h) wird dann f einmal in x und einmal in x + h
% ausgewertet, also fuer jedes x zweimal. Wesentlich effizienter ist es f
% einmal in jedem x auszuwerten, das in einem Vektor y zu speichern und
% dann mittels einer Schleife die numerische Ableitung zu bestimmen.

% Wir werten f in jedem x aus und speichern diese Werte im Vektor y.
% Ausserdem berechnen wir die Anzahl der Eintraege. Da die Division einen
% Wert vom Typ double liefert, runden wir mittels dem Befehl round auf die
% naechste natuerliche Zahl, da Indizes immer vom Typ integer sein muessen

tstart = tic;
y = f(x);
n_help = 4/h;
n = round(n_help);

% Belegt man Matrizen mittels einer Schleife, so sollte man vorher die
% Matrix initialisieren, d.h. entsprechenden Speicher bereitstellen.
```

```
Dffor = zeros(n,1);

% Berechne die numerische Ableitung mit Hilfe des Differenzenquotienten.

for i = 1:n

    Dffor(i) = (y(i + 1) - y(i))/h;

end

toc(tstart);
t2=toc(tstart);

plot(Dffor);
```

Vergleicht man die Zeiten $t1$ und $t2$, so ergibt sich ungefähr $t1 = 3{*}t2$. Dies scheint auf den ersten Blick kein großer Unterschied zu sein. Man sollte sich aber vor Augen führen, dass in komplexeren Programmen relativ häufig Ableitungen oder ähnliches berechnet werden müssen und dass wir im Allgemeinen auch kompliziertere Funktionen zum Auswerten haben. Zusammengefasst sollte man Funktionen nur so oft aufrufen und auswerten wie absolut notwendig. Da dies in einem Programm später nicht immer leicht zu korrigieren ist, sollte von vornherein beim Programmieren auf diesen Grundsatz geachtet werden. Eine weitere Verbesserung der Effizienz des Codes kann durch die Vektorisierung desselbigen erreicht werden, da `for`-Schleifen im Allgemeinen auch relativ ineffizient sind. Dies werden wir im nächsten Unterkapitel besprechen. In den wenigen Fällen, in denen eine Vektorisierung nicht möglich sein sollte, empfiehlt es sich, vor der Abarbeitung der Schleife den entsprechenden Array oder die Matrix beispielsweise durch `Dffor = zeros(n,1)` zu initialisieren, d.h. entsprechenden Speicher bereitzustellen. Ohne diese Initialisierung erzeugt MATLAB beim ersten Schleifendurchlauf zunächst einen Skalar `Dffor`. Im zweiten Durchgang wird dieser Skalar in einen 2-dimensionalen Vektor umkopiert, beim dritten Durchlauf in einen 3-dimensionalen Vektor und so fort. Durch die Speicher-Vorbelegung wird zumindest das zeitraubende Umkopieren in größer und größer werdende Arrays vermieden.

4.4.4 Die `while`-Schleife

Die `while`-Schleife zur mehrfachen Ausführung einer Befehlssequenz besitzt die folgende Syntax:

> `while` logischer Ausdruck
> Anweisungen
> `end`

Die Befehle des Schleifenrumpfes werden ausgeführt, solange der logische Ausdruck wahr ist. Die `while`-Schleife soll wieder anhand eines Beispiels erläutert werden. Betrachten wir die Division zweier ganzer Zahlen mit Rest. Seien $x, y \in \mathbb{N}$ gegeben und $q, r \in \mathbb{N}$ mit $r < y$ gesucht, so dass

$$x = qy + r$$

gelte. Das folgende Programm `Division_mit_Rest.m` berechnet q, r für gegebene x, y.

```
% % % % % % % % % % % % % % % % % % % % % % % % % % % %
% Division_mit_Rest.m berechnet fuer gegebene x, y aus
% N die Zahlen q,r aus N, derart dass r < y und
% x = qy + r gilt!
% % % % % % % % % % % % % % % % % % % % % % % % % % % %

clear all
disp('Division_mit_Rest.m berechnet fuer gegebene x, y aus ')
disp('N die Zahlen q,r aus N, derart dass x = qy + r mit r<y gilt!')

disp(' ')
x = input('Bitte eine Zahl x eingeben, die groesser als 0 ist: ');
disp(' ')
y = input('Bitte eine Zahl y eingeben, die groesser als 0 und kleiner als x ist: ');
disp(' ')

% Initialisierungen:
q = 0;

% Speichere den eingegebenen Wert von x fuer die Ausgabe
x_old=x;

% Abbruchbedingungen:
if x<0 || y<0
    error('x oder y kleiner 0!')
end
```

```
if x<y
    error('y ist nicht kleiner als x!')
end

if x == 0
    q = 0;
    r = x;
end

if y == 0
    error('Division durch 0!')
end

% Ziehe so oft y von x ab, bis ein Rest bleibt, der kleiner als y ist:

while x>=y
    x = x - y;
    q = q+1;
end

% Rest:
r = x;

disp(['q = ',num2str(q), ' und r = ',num2str(r)])
disp([num2str(x_old),'=',num2str(q),'*',num2str(y),'+',num2str(r)])
```

Der Algorithmus arbeitet nur mit positiven Zahlen, deswegen führen $x < 0$ oder $y < 0$ sofort zum Abbruch. Abbrüche werden mit dem Befehl

```
error('Text')
```

realisiert. Als Grund für den Abbruch erscheint `Text` im *Command Prompt*. Weiterhin darf y nicht gleich Null sein und x nicht kleiner als y. Da wir den Wert von x später noch brauchen, aber überschreiben werden, sichern wir ihn in der neuen Variablen `x_old`. q wird mit 0 initialisiert. Die Anweisungen der `while`-Schleife, also die Anweisungen zwischen `while` und `end`, werden so lange ausgeführt, bis die Bedingung `x>=y` nicht mehr erfüllt ist. In jedem Durchlauf der Schleife wird der Wert von x mit dem Wert $x - y$ überschrieben und der Wert von q um eins erhöht. Anhand der Ausgaben

```
disp(['q = ',num2str(q), ' und r = ',num2str(r)])
```

kann man erkennen, dass man innerhalb einer Zeile auch mehrere Ergebnisse des Programms ausgeben kann.

4.4.5 Die Befehle `break` und `continue`

Verwendet man die `if`-Anweisung innerhalb einer Schleife, so kann durch den Befehl `break` die Schleife abgebrochen und verlassen werden. Bei geschachtelten Schleifen wird nur die innerste Schleife beendet in der sich der Befehl `break` befindet. Durch die Verwendung des Befehls `continue` werden die restlichen Befehle des Schleifenrumpfes übersprungen und mit der nächsten Iteration begonnen. Das folgende kleine Beispiel (im Skriptbeispiele-Ordner `bspbreak.m` und `bspcontinue.m`) soll die Verwendung dieser beiden Befehle illustrieren.

```
% Dies ist ein Beispielprogramm fuer die Verwendung von break und continue

a = 2;
b = 10;

while (a < b)

    % Indem wir auf das Semikolon verzichten, wird b im Command Window
    % ausgegeben

    b = b/2

    if (a < 2)

        break %continue
    end

    a = a - 1

end
```

Die Ausgabe lautet: $b = 5$, $a = 1$, $b = 2.5$. Beim zweiten Durchlauf der Schleife ist die Bedingung `a < 2` wahr, so dass mit `break` die gesamte Schleife abgebrochen wird. Ersetzen wir `break` durch `continue`, so erhalten wir die folgende Ausgabe: $b = 5$, $a = 1$, $b = 2.5$, $b = 1.25$, $b = 0.625$. Beim ersten Schleifendurchlauf ist die `if`-Bedingung nicht erfüllt, folglich wird a um 1 erniedrigt. Bei allen folgenden Schleifen ist $a < 2$, die `if`-Bedingung wahr. Daher wird die Anweisung `continue` ausgeführt, d.h. die Kontrolle sofort an die `while`-Schleife übergeben und a nicht um 1 erniedrigt.

4.5 Ein paar (weitere) Hinweise zum effizienten Programmieren mit Matlab

4.5.1 Dünnbesetzte Matrizen

In der Numerischen Mathematik spielen Matrizen bei denen viele Elemente 0 sind eine große Rolle. In MATLAB lassen sich so genannte dünnbesetzte Matrizen effizient im Sparse-Format speichern. In dieser Darstellung werden nur die von 0 verschiedenen Elemente und deren Indizes in Listen abgelegt. Alle Matrixoperationen lassen sich auch auf Matrizen im Sparse-Format anwenden. Die Resultate sind dabei stets Sparse-Matrizen, sofern alle Argumente der Operation Sparse-Matrizen sind. Werden Rechenoperationen mit vollbesetzten Matrizen durchgeführt, so erhält man vollbesetzte Matrizen. Eine vollbesetzte Matrix M kann mit Hilfe des Befehls `zeros` initialisieren. Als Beispiel soll eine 100×100-Matrix gebildet werden:

```
M1 = zeros(100);
```

liefert eine Matrix M, die 100 Zeilen und 100 Spalten besitzt, alle Elemente sind 0. Mit den Schleifen

```
for j=1:100
    M1(j,j) = 2;
end
for j=1:99
    M1(j,j+1) = -1;
    M1(j+1,j) = -1;
end
```

wird eine Matrix gebaut, die auf der Hauptdiagonalen den Wert 2 annimmt und auf den Nebendiagonalen den Wert -1. Somit sind lediglich ca. 3 % der 10 000 Matrixelemente von 0 verschieden. Soll nun beispielsweise ein Gleichungssystem $Mx = b$ gelöst werden, kann MATLAB spezielle Algorithmen für Sparse-Matrizen nutzen, wenn M im Sparse-Modus gespeichert ist. Dies ginge z.B. durch die Initialisierung

```
M2 = spalloc(100,100,300);
```

Dabei geben die ersten zwei Einträge in den Klammern die Dimension der Matrix an. Der dritte Eintrag 300 steht für die maximale Anzahl der Elemente von M, die von 0 verschieden sind. Entweder können nun die gleichen Schleifen für das Setzen der Elemente genutzt werden. Es kann aber auch der schnelle Befehl `spdiags` verwendet werden, welcher eine Sparse-Matrix erstellt, die durch Diagonalen gebildet wird:

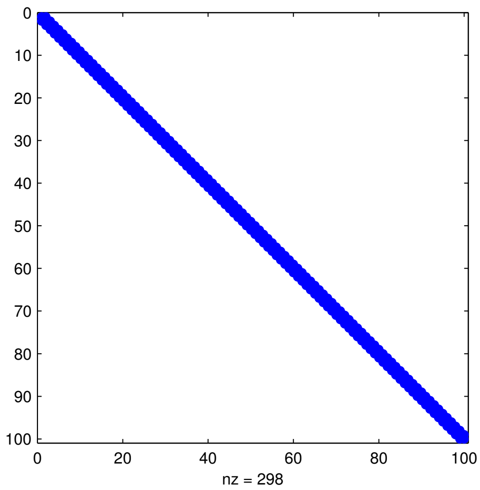

Abbildung 4.2: Plot der von 0 verschiedenen Matrixelemente

```
e = ones(100,1)
M2 = spdiags([-e 2*e -e], -1:1, 100, 100)
```

In den eckigen Klammern stehen die Diagonalen, der folgende Eintrag `-1:1` gibt an, welche Diagonalen besetzt werden. `0` steht für die Hauptdiagonale, `1` für die erste, `2` für die zweite rechte Nebendiagonale, usw.. `-1` steht für die erste, `-2` für die zweite linke Nebendiagonale, usw.. Allgemein kreiert der Befehl `M = spdiags(B,d,m,n)` eine $m \times n$ Sparse Matrix, in dem die Spalten von `B` entlang der Diagonalen platziert werden, wie der Vektor `d` es vorgibt. Im Beispiel ist `B` die Matrix `[-e 2*e -e]`, `d` der Vektor $(-1, 0, 1)$ definiert durch `-1:1`. Der Befehl `M2(1,:)` gibt wie bereits bekannt, alle Elemente der ersten Zeile aus. Bei der Sparse-Matrix `M2` sind dies nur die Elemente in der ersten und zweiten Spalte. Die Darstellung ist dann wie folgt:

```
>> M2(1,:)

ans =

   (1,1)          2
   (1,2)         -1
```

Dies bedeutet, dass das Element $(1, 1)$ den Wert 2 hat, das Element $(1, 2)$ den Wert -1 und alle anderen Elemente der ersten Zeile 0 sind. Der Befehl `spy(M2)` gibt eine Graphik aus, die anzeigt, welche Elemente von `M2` von Null verschieden sind (Abbildung 4.2). Um eine dünnbesetzte Diagonalmatrix zu erhalten kann man den Befehl `speye` benutzen.

4.5.2 Vektorisieren und Zeitmessung

Wie bereits erwähnt, ist die Verwendung von Schleifen meist ineffizient. Daher sollte man wann immer dies möglich ist den Code vektorisieren. Wie das geht, zeigen wir anhand des Beispiels der numerischen Ableitung. Die Differenz wird nun nicht mit Hilfe einer `for`-Schleife gebildet, sondern dadurch, dass wir verschiedene Bereiche des Vektors y ansprechen und diese voneinander subtrahieren.

```
% Programm, welches fuer gegebene Werte x und eine gegebene Funktion f
% einerseits mit Hilfe mit einer Schleife die Ableitungen
% von f in diesem Punkt x bestimmt und zum Vergleich das Ganze auch
% vektorisiert berechnet

% Clear Workspace

clear all

% Definition von f als anonyme Funktion

f = @(x) sin(x);

% Festlegung der Werte x aus dem Intervall [0,4]

h=0.00002;
x=0:h:4;

% Wir werten f in jedem x aus und speichern diese Werte im Vektor y.

y = f(x);

% Ausserdem berechnen wir die Anzahl der Eintrage. Da die Division einen
% Wert vom Typ double liefert, runden wir mittels dem Befehl round auf die
% naechste natuerliche Zahl, da Indizes immer vom Typ integer sein muessen


n_help = 4/h;
n = round(n_help);

% Belegt man Matrizen mittels einer Schleife, so sollte man vorher die
% Matrix initialisieren, d.h. entsprechenden Speicher bereitstellen.

tic;
```

```
Dffor = zeros(n,1);

% Berechne die numerische Ableitung mit Hilfe des Differenzenquotienten in
% einer for-Schleife.

for i = 1:n

    Dffor(i) = (y(i + 1) - y(i))/h;

end

toc;
t1=toc;

% Plotte zur Kontrolle die Funtkion

%plot(Dffor);

% Da die Verwendung von Schleifen i. Allg. ineffizient ist, vektorisieren
% wir nun obigen Code und messen die Zeit wieder mit tic und toc.
% Wir berechnen die Ableitung indem wir verschiedene Bereiche des Vektors y
% subtrahieren und anschliessend durch h teilen.

tstart = tic;

Dfvect = (y(2:n+1)-y(1:n))/h;

toc(tstart);
t2=toc(tstart);

plot(Dfvect);
```

Da ein Code der Schleifen enthält im Allgemeinen relativ leicht vektorisiert werden kann, ist es beim Programmieren eines größeren Projektes manchmal sinnvoll erst die intuitiveren Schleifen zu verwenden, welche weniger Potential für Fehler haben, und erst in einem zweiten Schritt das Ganze zu vektorisieren. Generell sollte man erst dafür sorgen, dass das Programm läuft und richtige Ergebnisse liefert. Vor allem für Programmieranfänger ist es daher ratsam das Vektorisieren erst in einem zweiten Schritt durchzuführen. In diesem Zusammenhang soll noch erwähnt werden, dass sich die Befehle `tic` und `toc` auch sehr gut eignen um festzustellen welche Abschnitte im Code viel Rechnerzeit benötigen und daher wenn möglich verbessert werden sollten.

5 Graphische Ausgaben

Ein großer Vorteil von MATLAB liegt in der sehr einfachen graphischen Darstellung von Ergebnissen. Hier wollen wir die Grundlagen der 2- und 3D Plots darstellen. Im Anschluss wird erläutert, wie Filme mit MATLAB erstellt werden können.

5.1 2D Plots

Beginnen wir zunächst mit der zweidimensionalen graphischen Ausgabe. Zum Öffnen eines Bildes, einer so genannten *figure*, muss zunächst der Befehl

```
>> figure(1)
```

aufgerufen werden. Es öffnet sich ein leeres Graphik-Fenster. Die Nummer in den Klammern kann beliebig gewählt werden. Ruft man den Befehl `figure()` nicht auf, so werden alte Bilder überschrieben. Der Befehl `plot(x,y)` erzeugt eine Graphik, in der die Werte eines Vektors x gegen die des Vektors y aufgetragen sind. Die Punkte werden durch eine Gerade verbunden. Seien zum Beispiel die Vektoren

```
x = [ 0 0.5 1 1.5 2 2.5 3 3.5 4]
y = [ 1.0000  0.8776  0.5403  0.0707 -0.4161 -0.8011 -0.9900 -0.9365 -0.6536]
```

gegeben. Dies sind die Werte $y = \cos(x)$. Eine zweidimensionale Graphik wird mit dem Befehl `plot` generiert:

```
>> plot(x,y)
```

Der resultierende Plot ist in Abbildung 5.1 zu sehen. Wählen wir einen feiner abgestuften Vektor x, z.B. `x=[0:0.2:4]`, werten `y=cos(x)` und `plot(x,y)` erneut aus, so ergibt sich ein glatterer Plot. Oftmals wird der Vektor x für Plots automatisch mit dem Befehl

```
x = linspace(a,b,N)
```

gebildet. Dabei gibt `a` die linke Intervallgrenze und `b` die rechte Intervallgrenze an. Das Intervall $[a, b]$ wird in N Punkte äquidistrante Punkte aufgeteilt. Wählen wir $a = 0$, $b = 4$ und $N = 20$, so erhält man den Plot in Abbildung 5.2. Es gibt verschiedene Möglichkeiten, diesen Plot nun zu beschriften. Zum einen kann man sich in der *figure* durch die Menüs klicken und dort Achsenbeschriftungen, -skalen, -bezeichnungen, Titel, Legenden etc. eingeben. Da man diese Einstellungen aber nicht speichern kann und für

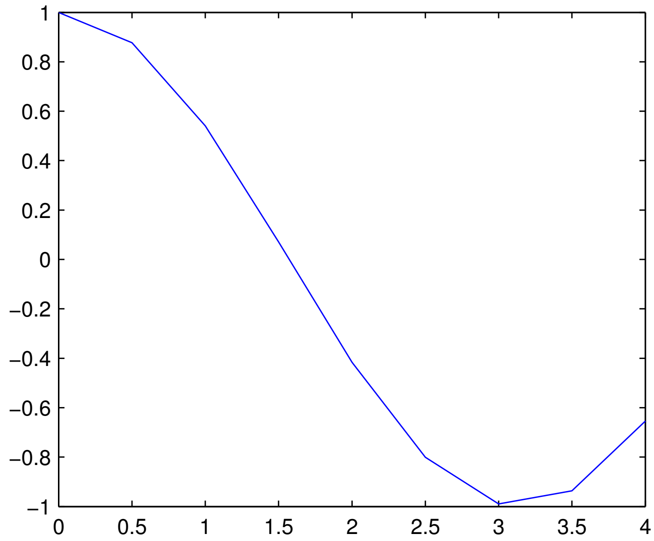

Abbildung 5.1: Plot zu den Vektoren $x = (0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4)^t$ und $y = (1.0000, 0.8776, 0.5403, 0.0707, -0.4161, -0.8011, -0.9900, -0.9365, -0.6536)^t$

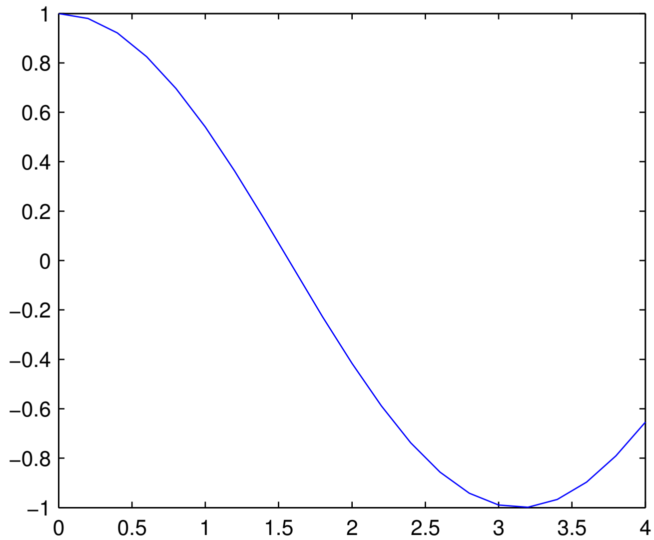

Abbildung 5.2: Plot mit Hilfe von `x = linspace(0,4,20)` und `y = cos(x)`

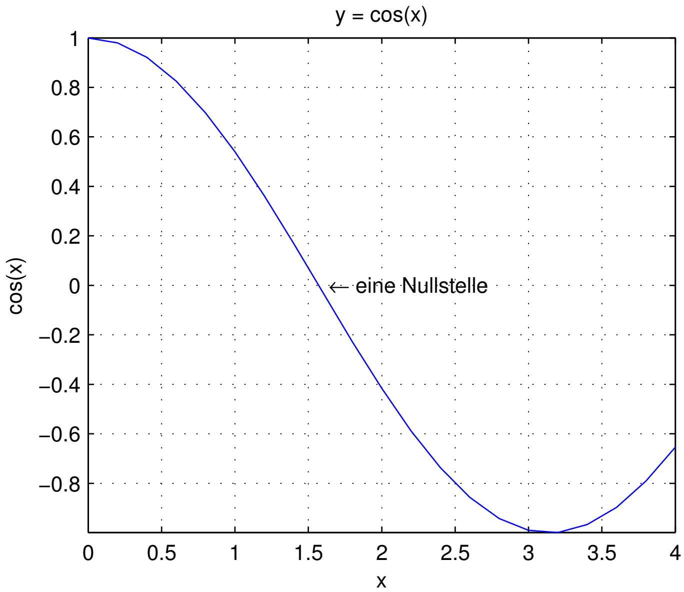

Abbildung 5.3: Direkte Beschriftung des Punktes (1.6, 0) mit "← eine Nullstelle "

jede Graphik neu erstellen muss, wird hier nur die Methode vorgestellt, wie man die *figure* direkt aus dem *M-File* heraus bearbeitet. Dabei beschränken wir uns auf einige wesentliche Möglichkeiten der Graphik-Bearbeitung. Weitere Informationen zu Graphik-Ausgaben finden sich in den Literaturangaben oder in der Matlab-Hilfe. Tabelle 5.1 zeigt einige Möglichkeiten zur Veränderung der graphischen Ausgabe.
Die Bezeichnungen können LaTeX codes enthalten, MATLAB interpretiert z. B. die Zeichenfolge `\psi \rightarrow x^4 f(x_6) ||z||_{x\in\Omega}` wie LaTeX als $\psi \rightarrow x^4$ $f(x_6)$ $||z||_{x\in\Omega}$. Der Aufruf eines Plots in einem M-File (`figure_nullstelle.m`) könnte also folgendermaßen aussehen:

```
x = linspace(0,4,20);
y = cos(x);

figure(1)
plot(x,y)
grid on
axis tight
xlabel('x')
ylabel('cos(x)')
title('y=cos(x)')
text(1.6,0,' \leftarrow eine Nullstelle')
```

Matlab-Befehl	Beschreibung
`axis([xmin xmax ymin ymax])`	setzen der x-Achse auf das Intervall [xmin,xmax], y-Achse auf [ymin,ymax]
`axis manual`	Einfrieren der Achsen in einer figure für folgende plots
`axis tight`	automatische Anpassung der Achsen auf die Daten
`axis xy`	Ausrichtung des Ursprungs (wichtiger in 3D Visualisierungen)
`axis equal`	Gleiche Wahl der Skalierung auf allen Achsen
`axis square`	Quadratischer Plot
`grid on`	Gitter anzeigen
`grid off`	Gitter nicht anzeigen
`xlabel('Name der x-Achse')`	x-Achsen Beschriftung
`ylabel('Name der y-Achse')`	y-Achsen Beschriftung
`zlabel('Name der z-Achse')`	z-Achsen Beschriftung (bei 3D Visualisierungen)
`title('Titel der Figure')`	Überschrift der Figure
`legend('Legende des Plots')`	L̈egende des Plots
`text(x,y,'string')`	direkte Beschriftung des Punktes (x,y) in der Graphik
`set(gca,'XTick','Vektor')`	Setzen der zu beschriftenden x-Achsen Punkte
`set(gca,'XTickLabel',{'P1','P2',...})`	Bennenung der x-Achsenpunkte
`colormap(Farbskala)`	Auswahl einer Farbskala
`caxis([cmin cmax])`	Setzen der Farbskala auf das Intervall [cmin cmax]
`caxis auto`	automatisches Setzen der Farbskala

Tabelle 5.1: Möglichkeiten zur Veränderung der graphischen Ausgabe

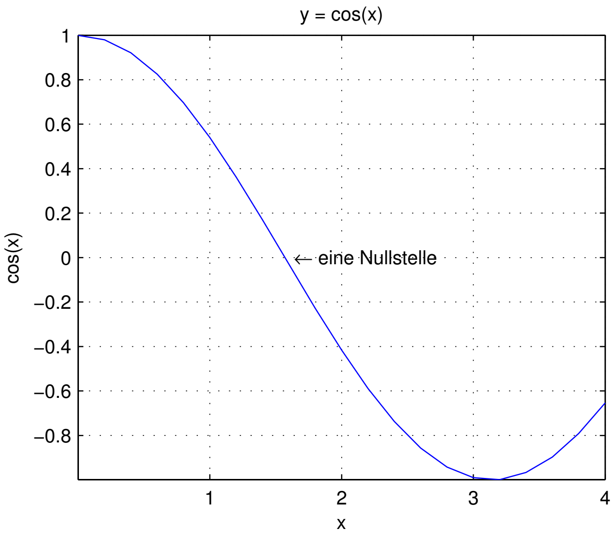

Abbildung 5.4: Änderung der x-Achsenpunkte mit Hilfe des Befehls `set(gca,'XTick',1:4)`

Den resultierenden Plot sieht man in Abbildung 5.3. Die Achsenpunkte können via

```
set(gca,'XTick',Vektor)
set(gca,'XTickLabel',{'Punkt 1','Punkt 2',...})
```

geändert werden. Dabei steht `gca` für *get current axis*. `Vektor` ist hierbei ein vorher definierter Vektor und `Punkt 1`, `Punkt 2`, .. die Namen die man den neuen Achsenpunkten, welche durch den `Vektor` definiert wurden, geben möchte. Statt der Option `'XTick'`, `'XTickLabel'` können auch analog `'YTick'`, `'YTickLabel'` und analog für z für die y-, bzw. z-Achse verwandt werden. Der Befehl `set(gca,'XTick',Vektor)` ändert die Punkte an der x-Achse, die explizit markiert sind. Bisher war dies jeder halbzahlige Wert von 0 bis 4. Soll nun jeder ganzzahlige Wert eine Markierung erhalten, kann das mit dem Befehl `set(gca,'XTick',1:4)` realisiert werden. `figure(1)` aus dem M-File `Achsenbeschriftung.m` ergibt dann den Plot in Abbildung 5.4. Sollen nun auch andere Bezeichnungen der x-Achsenmarkierungen eingeführt werden, kann dies mit `set(gca,'XTickLabel',{'Punkt 1','Punkt 2',...})` geschehen. Es ist wichtig, dass genauso viele Namen angegeben werden, wie Markierungen existieren. In unserer Graphik bewirkt z.B. der Befehl `set(gca,'XTickLabel',{'a','b','c','d'})` (= `figure(2)` in `Achsenbeschriftung.m`) die Ausgabe in Abbildung 5.5. Leider werden in der Achsenmarkierung Latex-Fonts *nicht* berücksichtigt (z.B. würde eine Eingabe `\pi` die Bezeichnung `\pi` statt π hervorrufen).

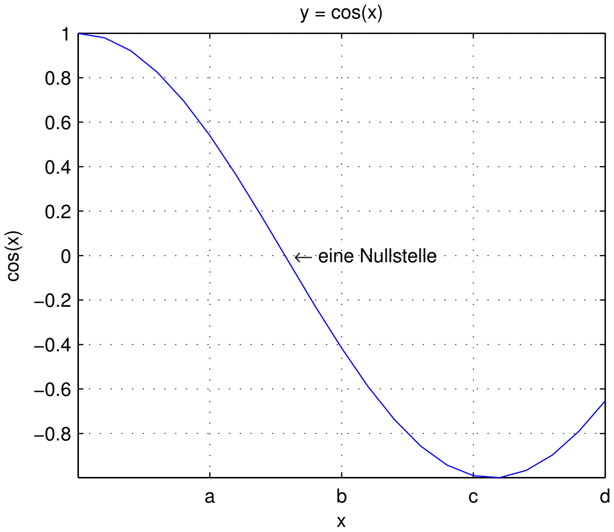

Abbildung 5.5: Umbenennung der neu definierten x-Achsenpunkte durch den Befehl `set(gca,'XTickLabel',{'a','b','c','d'})`

5.1.1 Logarithmische Skalierung

Um einen Plot mit logarithmischer Skalierung zu erzeugen benutzt man anstatt des `plot` Befehls die Befehle `semilogx`, `semilogy` oder `loglog` für logarithmische Skalierung der x-,y-Achse oder beider Achsen.

5.2 Graphikexport

Mittels des Menüs *File→Save as* oder *Export Setup* können Graphiken in verschiedenen Formaten (z.B. `fig`, `jpg`, `eps`, `pdf`, `tif`, ...) gespeichert werden. Das Matlab Standard-Format ist `.fig`. Es empfiehlt sich, Dateien, die exportiert werden sollen, auch immer im `fig`-Format abzuspeichern. Sehr oft fallen kleine Fehler in der Achsenbeschriftung etc. erst verspätet auf. Ist die Graphik im `fig`-Format gespeichert, kann sie (ohne neue Berechnungen!) geöffnet und geändert werden. Gerade bei langwierigen Rechnung macht dies Änderungen viel einfacher. Möchte man eine Vielzahl von Graphiken speichern, kann es sehr umständlich werden, alle Bilder einzeln über das Menü zu exportieren. Es ist möglich, Speicherbefehle via `print` direkt in das M-File zu schreiben. Der Aufruf

```
print -depsc Name
```

erzeugt z.B. eine (bunte) Postscript Datei `Name.eps`, welche in dem lokalen Ordner, in dem sich das aktuelle M-File befindet, abgelegt wird. Die Menüleisten etc. werden natürlich nicht exportiert. Unten ist eine kleine Übersicht über die Optionen des `print`

Befehls zu sehen. Mit Hilfe der Option `-rAufloesung` kann die Druckauflösung erhöht werden. Dies ist besonders für aufwendige 3D Plots notwendig! Für den Ausdruck einer Graphik auf Papier eignen sich aufgrund ihrer guten Qualität die `eps`-Formate. Für Beamer-Vorträge sind komprimierte Dateien von Vorteil, daher eignet sich hier eher das `jpg`-Format.

Befehl	Beschreibung
print -depsc	Export als farbige eps-Datei (Postskript, Vektordatei)
print -deps	Export als schwarz/weiße eps-Datei (Postskript, Vektordatei)
print -dill	Export als Adobe Illustrator Datei (Vektordatei)
print -djpeg	Export als jpg-Datei (Bitmapdatei)
print -dtiff	Export als tiff-Datei (Bitmapdatei)
print -depsc -r600	Export als farbige eps-Datei mit der Auflösung 600 dpi
print -dpng	Export als Webseiten konforme png-Datei

5.3 Mehrere Plots in einer figure

5.3.1 Gruppierung von mehreren Plots in einer figure mittels `subplot`

Es gibt verschiedene Möglichkeiten, mehrere Plots in einer figure unterzubringen. Der Befehl `subplot` bietet die Möglichkeit mehrere Graphiken beliebig in einer figure zu gruppieren. Der Aufruf

```
figure(2)
subplot(n,m,1)
plot(x,y)
subplot(n,m,2)
plot(x,y)
subplot(n,m,3)
plot(x,y)
   .
   .
   .
subplot(n,m,nm)
plot(x,y)
```

erzeugt eine `figure 2`, die n Bilder in einer Reihe und m Spalten erzeugt. Die Bildnummer muss von 1 bis nm laufen. Sollen zum Beispiel wie im M-File `Beispiel_2d_subplot.m` vier Datensätze $(x, f), (x, g), (x, h), (x, j)$ in einer figure verglichen werden, so kann dies mit dem Befehl `subplot` realisiert werden. Wir wollen 4 Bilder, 2 pro Zeile und 2 pro Spalte erzeugen. Also ist $n = 2$, $m = 2$ und die Bildnummer läuft von 1 bis 4. Das M-File sieht dann wie folgt aus:

```
x = linspace(0,6,100);

f = x.^2;
g = sqrt(x);
h = x.^3;
j = 3.*x-5;

figure(2)
subplot(2,2,1)
plot(x,f)
title('f(x)=x^2')
xlabel('x')

subplot(2,2,2)
plot(x,g)
title('g(x)=sqrt(x)')
xlabel('x')

subplot(2,2,3)
plot(x,h)
title('h(x)=x^3')
xlabel('x')

subplot(2,2,4)
plot(x,j)
title('j(x)=3x-5')
xlabel('x')
```

Wie in Abbildung 5.6 zu sehen, werden alle vier Plots in einer figure dargestellt. Der Befehl `subplot` funktioniert analog für 3D Graphiken, welche wir im nächsten Unterkapitel kennenlernen werden.

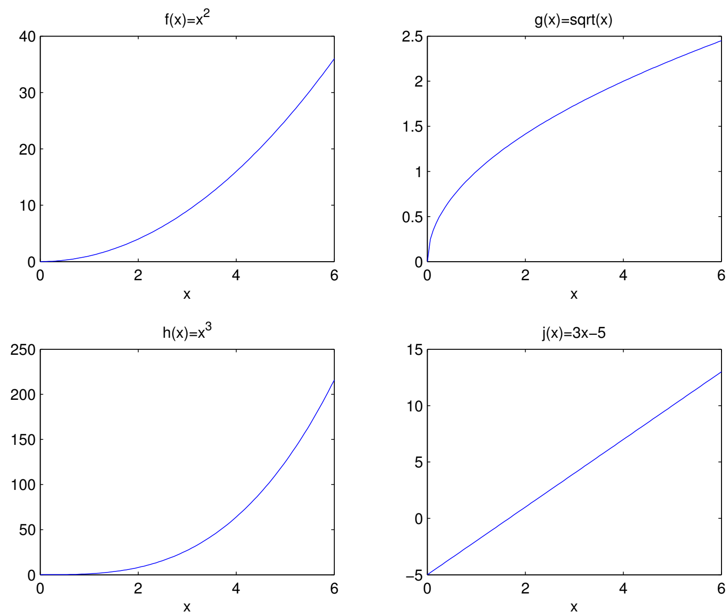

Abbildung 5.6: Zusammenfügen mehrerer plots zu einer figure mit dem Befehl `subplot`

5.3.2 Zusammenfügen mehrerer Plots zu einem Bild mit `hold on` und `hold off`

Manchmal kann es nützlich sein, mehrere (Funktions-)Plots in einem Bild zu vergleichen. Dafür verwenden wir die Befehle `hold on` und `hold off`. Es sollen die beiden Funktionen $g(x) = \sqrt{x}$ und $j(x) = x$ für den Definitionsbereich $[0, 2]$ verglichen werden. Dazu betrachten wir das M-File `Beispiel_2d_plot_holdon.m`:

```
x = linspace(0,2,20);
g = sqrt(x);
j = x;

figure(1)
plot(x,g,'Color','k','Linestyle','--')
hold on
plot(x,j,'Color','r','Marker','d','MarkerSize',8)
hold off
legend('g(x)=sqrt(x)','j(x)=x','Location','Northwest')
xlabel('x')
axis square
title('Vergleich der Funktionen g(x)=sqrt(x) und j(x)=x ueber [0,2]')
```

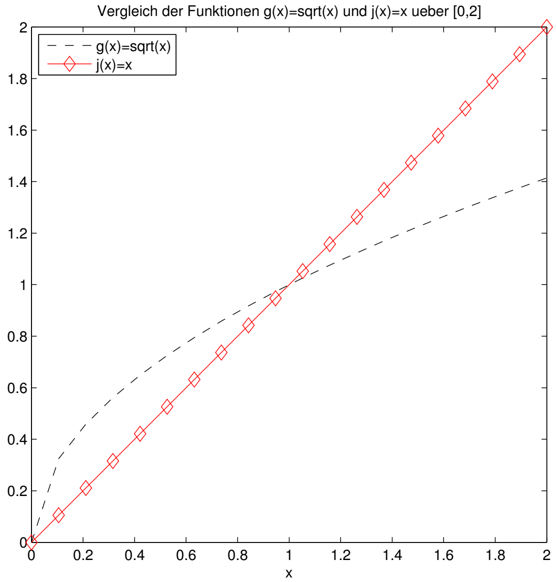

Abbildung 5.7: Vergleich verschiedener Plots in einem mittels `hold on` und `hold off`

Um mehrere Plots in ein Bild zu legen, schreibt man nach dem ersten Plot `hold on`. Dann folgen die Plots, welche zu dem ersten hinzugefügt werden sollen. Nach dem letzten hinzuzufügenden Plot folgt `hold off`. Danach fügt man mit dem Befehl `legend` eine Legende hinzu. Der erste Eintrag entspricht dabei dem ersten Plot, der zweite dem zweiten Plot, usw. Die Anzahl der Plots, welche in die Legende eingetragen werden, muss hierbei mit der Anzahl der dem Bild hinzugefügten Plots übereinstimmen. Bei den Einträgen in die Legende kann LaTeX-Code verwendet werden. Die Platzierung der Legende in der figure lässt sich verändern. Standardgemäß erscheint sie rechts oben. In unserem Bild wurde sie mit dem Befehl `'Northwest'` nach links oben gesetzt. Die weiteren möglichen Platzierungen listet die MATLAB -Hilfe zu `legend`. Da wir nun verschiedene Plots in einer figure haben ist es notwendig diese zu unterscheiden. Am einfachsten geht dies durch das Verwenden unterschiedlicher Farben. Soll das Bild nachher in schwarz-weiß gedruckt werden, so kann man entweder den Linienstil verändern (1. Plot) oder Marker auf die Linie setzen (2. Plot). Um die Plots anzupassen geben wir dem `plot`-Befehl weitere Argumente mit: `plot(x,y,'PropertyName',PropertyValue)`. Hierbei steht immer zunächst der `PropertyName`, die Eigenschaft welche man ändern möchte, z.B. `Color` oder `Marker`. Dann folgt der `Propertyvalue`, also der Wert auf welchen man die jeweilige Eigenschaft setzen möchte. Alle `PropertyValues` außer Zahlen werden in Anführungszeichen gefasst. Es folgt eine Tabelle mit den gängigen Farben, Markern und Linienstilen. Weitere so genannte *Lineseries Properties* können unter diesem Suchbegriff in der Hilfe nachgelesen werden. Den Plot zum M-File `Beispiel_2d_plot_holdon.m` zeigt Abbildung 5.7.

Befehl	Farbe	Befehl	Marker	Befehl	Linienstil
'y'	gelb	'+'	Pluszeichen	'-'	durchgezogene Linie
'm'	magenta	'*'	Stern	'--'	gestrichelte Linie
'c'	cyan	'x'	Kreuz	':'	gepunktete Linie
'r'	rot	's'	Quadrat	'-.'	Strich-Punkt-Linie
'g'	grün	'd'	Diamand	'none'	keine Linie
'b'	blau	'p'	Pentagramm		
'w'	weiß	'v'	Dreieck unten		
'k'	schwarz	'<'	Dreieck links		

5.4 3D Graphiken

Die bisher vorgestellten Graphik-Befehle sind auch für 3D Visualisierungen gültig. Statt des Befehls `plot(x,y)` wird in der 3D Visualisierung `mesh(x,y,f)` (Gitterplot) oder auch `surf(x,y,f)` (Oberflächenplot), `contour(x,y,f)` (Contourplot) benutzt. Dabei ist x ein Vektor der Dimension n, y ein Vektor der Dimension m und f ein array der Größe (m, n). Das Einhalten der Reihenfolge von x und y ist wichtig, denn haben x und y nicht die gleiche Dimension, führt dies zu Fehlern. Um uns die Verwendung einiger Befehle im Zusammenhang mit 3D Graphiken klar zu machen, arbeiten wir uns durch das M-File `Beispiel_3d_plot.m`. Hierbei stellt jede `figure` ein besprochenes Thema dar. Wir beginnen mit dem Erzeugen des diskreten Analogons der zu plottenden Funktion. Es soll die Funktion $f(x, y) = \cos(x)\sin(y)$ auf dem Gebiet $[0, 8] \times [0, 8]$ geplottet werden. Zunächst erzeugen wir Gitter `xgrid` und `ygrid` mit $N = 40$ Punkten in x- und y-Richtung. Die Distanz zwischen zwei Gitterpunkten ist hierbei $h = 8/(N - 1)$. Mittels `[x,y] = meshgrid(xgrid,ygrid)` wird das äquidistante 2D Gitter erzeugt. Der `meshgrid`-Befehl funktioniert analog falls die Gitter in x- und y-Richtung nicht dieselbe Dimension haben. Da mit dem `meshgrid`-Befehl erzeugte Gitter direkt in Funktionen eingesetzt werden können, sieht der erste Teil des M-Files wie folgt aus:

```
f = @(x,y) sin(x).*cos(y);         % Definiere Funktion f
L = 8;                             % will auf Gebiet [0,L]x[0,L] plotten
N = 40;                            % Anzahl der Gitterpunkte
h = L/(N-1);                       % Schrittweite
xgrid = 0:h:8;                     % Erzeuge Gitter in x-Richtung
ygrid = 0:h:8;                     % Erzeuge Gitter in y-Richtung
[X,Y] = meshgrid(xgrid,ygrid);     % Erzeuge Gitter in xy-Ebene
fd = f(X,Y);                       % Erzeuge diskrete Version von f
```

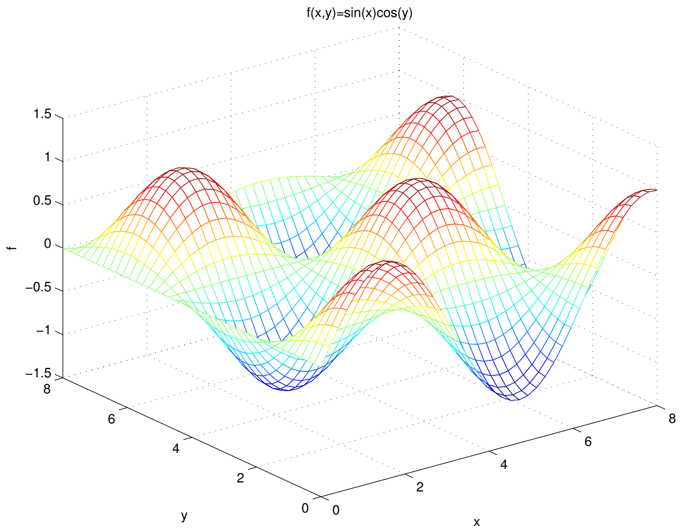

Abbildung 5.8: Ein mit `mesh(X,Y,fd)` erzeugter 3D Plot zeigt das zugrundeliegende Gitter

Nachdem wir nun eine diskrete Version unserer Funktion f erzeugt haben, wollen wir uns zunächst mit verschieden Befehlen zum Erzeugen einer 3D Graphik befassen. Wir beginnen mit dem `mesh`-Befehl und `figure(1)`.

```
figure(1)
mesh(X,Y,fd)
xlabel('x')
ylabel('y')
zlabel('f')
title('f(x,y)=sin(x)cos(y)')
axis([0,8,0,8,-1.5,1.5])
```

Der Befehl `mesh` zeigt die Funktion als farbiges Gitter und bietet sich insbesondere dann an, wenn auch Informationen über das zugrundeliegende Gitter gezeigt werden sollen. `figure(1)` ist in Abbildung 5.8 zu sehen. Mit Hilfe des Befehls `colorbar` kann eine Säule hinzugefügt werden, welche die Farbskala des Plots den Wertebereich der geplotteten Funktion zuordnet.

```
figure(2)
mesh(X,Y,fd)
xlabel('x')
ylabel('y')
```

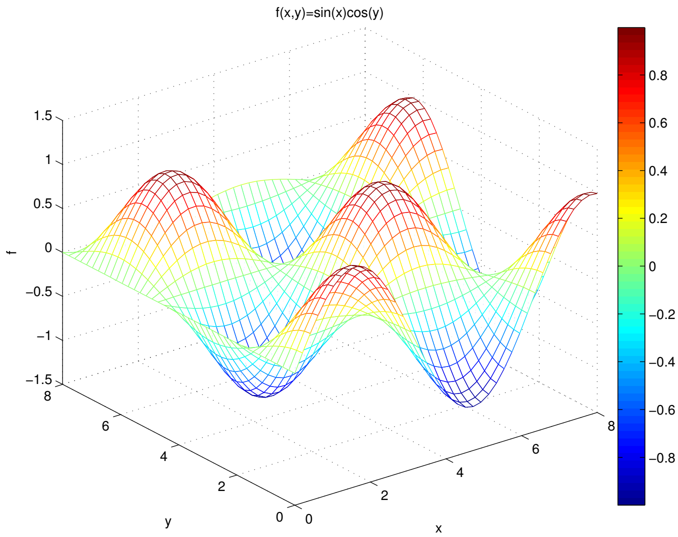

Abbildung 5.9: Ein mit `mesh(X,Y,fd)` erzeugter 3D Plot mit `colorbar`

```
zlabel('f')
title('f(x,y)=sin(x)cos(y)')
axis([0,8,0,8,-1.5,1.5])
colorbar
```

Eine 3D Graphik sollte immer eine `colorbar` enthalten, da nur so dem Bild wirklich Informationen über die geplottete Funktion entnommen werden können. Die graphische Ausgabe zu `figure(2)` ist in Abbildung 5.9 zu sehen. Oberflächen-Plots können z.B. mit `surf(X,Y,fd)` erzeugt werden (`figure(3)` und Abbildung 5.10). Möchte man sich ferner noch die Contourlinien der Funktion anzeigen lassen, so verwendet man den Befehl `surfc(X,Y,fd)` (`figure(4)` und Abbildung 5.11).

```
figure(3)
surf(X,Y,fd)
xlabel('x')
ylabel('y')
zlabel('f')
title('f(x,y)=sin(x)cos(y)')
axis([0,8,0,8,-1.5,1.5])
colorbar
```

```
figure(4)
surfc(X,Y,fd)
xlabel('x')
ylabel('y')
zlabel('f')
title('f(x,y)=sin(x)cos(y)')
axis([0,8,0,8,-1.5,1.5])
colorbar
```

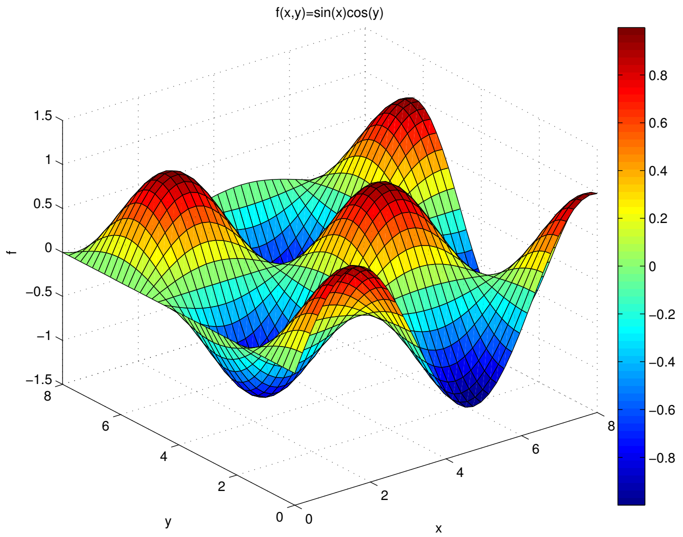

Abbildung 5.10: Ein mit `surf(X,Y,fd)` erzeugter 3D Plot plottet die Oberfläche

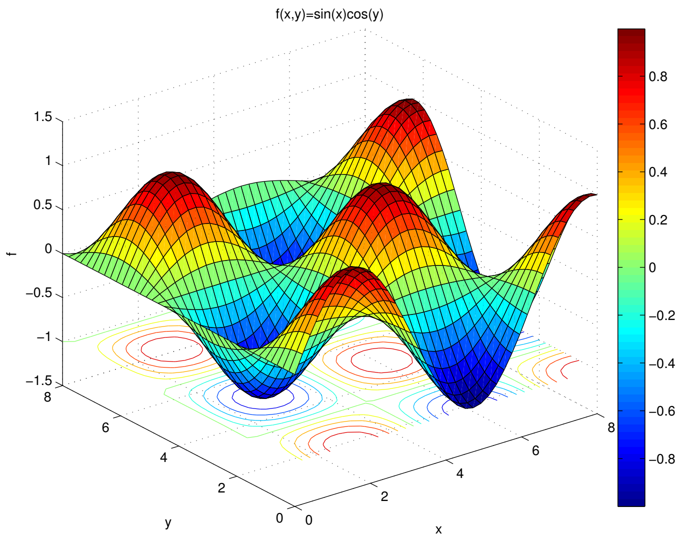

Abbildung 5.11: Ein mit `surfc(X,Y,fd)` erzeugter 3D Plot plottet die Oberfläche und zeigt die Contourlinien

Verwendet man ein feines Gitter und möchte mit `surf` oder `surfc` plotten, so sollte man auf den Befehl `shading interp` zurückgreifen. Warum man dies tun sollte, illustriert

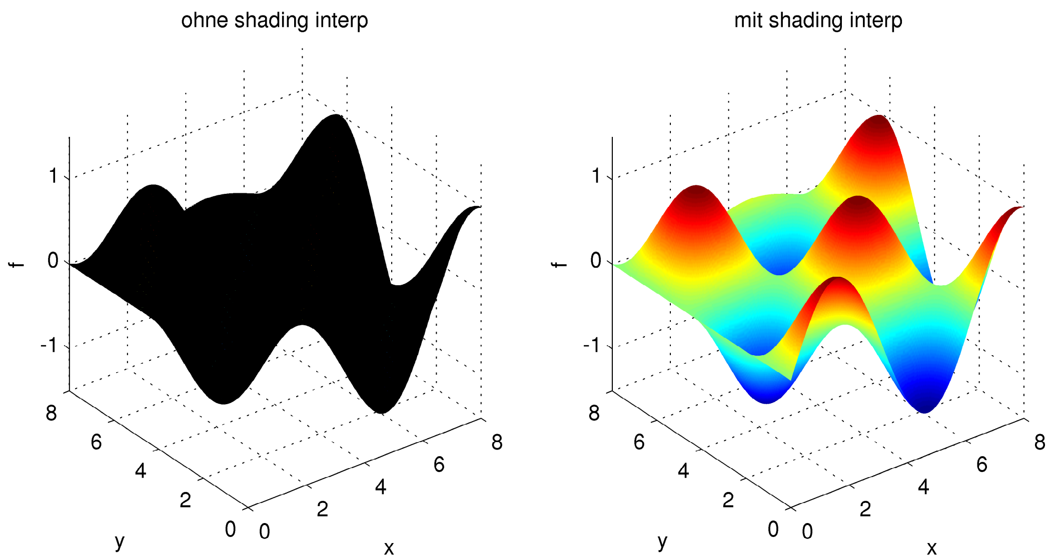

Abbildung 5.12: Ein mit `surf(y,x,f)` erzeugter 3D Plot ohne (links) und mit (rechts) dem Befehl `shading interp`

Abbildung 5.12. Diese Plots erhält man, wenn man das M-File `Beispiel_3d_plot.m` statt mit `N = 40` Gitterpunkten mit `N = 400` Gitterpunkten ausführt. Aufgrund der Feinheit des Gitters sind im linken Bild nur noch die Begrenzungen der Gitterelemente und keine Farbverläufe mehr zu sehen. Durch den Befehl `shading interp` wird kontinuierlich über die Farbfläche interpoliert und die Begrenzungslinien entfallen (rechtes Bild). Das M-File für die `figure(5)` sieht wie folgt aus:

```
figure(5)
subplot(1,2,1)
surf(X,Y,fd)
xlabel('x')
ylabel('y')
zlabel('f')
title('ohne shading interp')
axis([0,8,0,8,-1.5,1.5])
axis square

subplot(1,2,2)
surf(X,Y,fd)
shading interp;
xlabel('x')
```

```
ylabel('y')
zlabel('f')
title('mit shading interp')
axis([0,8,0,8,-1.5,1.5])
axis square
```

Abgesehen von dem `shading` des Plots können natürlich noch andere Eigenschaften verändert werden. Zum Beispiel lässt sich das Farbschema mit dem Befehl `colormap` einstellen. Dabei können unter anderem folgende Farbschemata gewählt werden: `default`, `hot`, `jet`, `hsv`, `winter`, `spring`, `autumn`, `summer`, `bone`, `cool`, `copper`, usw. Im M-File `Beispiel_3d_plot.m` haben wir uns für das Farbschema `hot` entschieden:

```
figure(6)
surf(X,Y,fd)
xlabel('x')
ylabel('y')
zlabel('f')
title('f(x,y)=sin(x)cos(y)')
axis([0,8,0,8,-1.5,1.5])
colorbar
colormap('hot')
```

Das Ergebnis ist hier (Abb. 5.13) zu sehen.

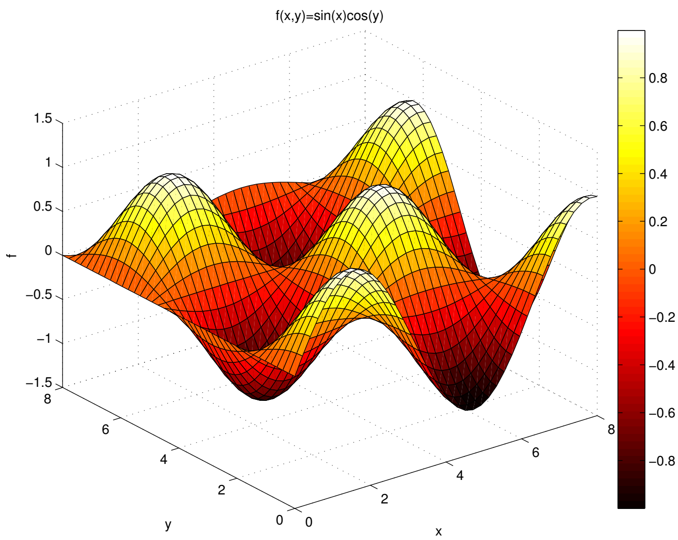

Abbildung 5.13: Ändern des Farbschemas mittels `colormap`; hier: `'hot'`

Weiterhin kann zum Beispiel auch das Material und die Beleuchtung eingestellt werden, wie das M-File für `figure(7)` zeigt:

```
figure(7)
surf(X,Y,fd)
shading interp;
xlabel('x')
ylabel('y')
zlabel('f')
title('f(x,y)=sin(x)cos(y)')
axis([0,8,0,8,-1.5,1.5])
colorbar
colormap('autumn')
material metal
camlight('left')
camlight('headlight')
lighting phong
```

Die Graphikausgabe von `figure(7)` (Abbildung 5.14) sieht wie dann so aus. Für Details

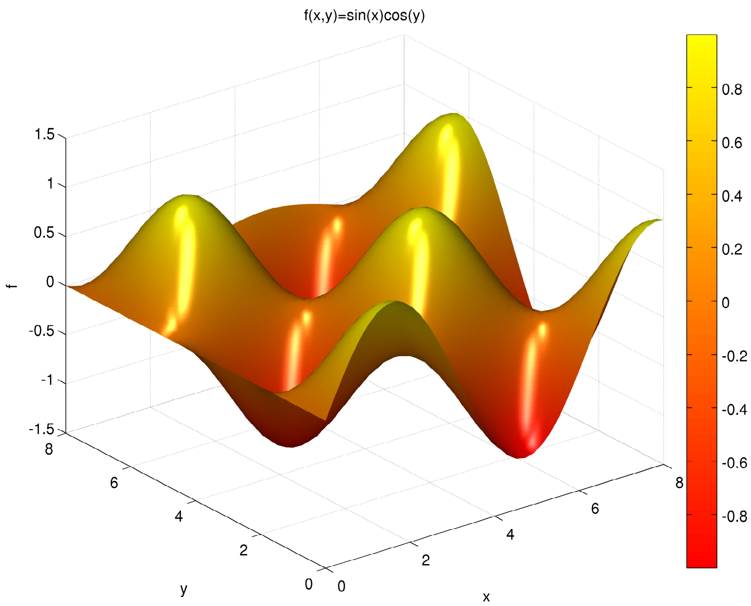

Abbildung 5.14: Änderung des Aussehens des Plots unter Beleuchtung mittels `lighting`, des Materials mit `material`, der Herkunft des Lichts mit `camlight`

zu diesen Themen sei auf die Hilfe (siehe beispielsweise `lighting`, `material`, `camlight`) verwiesen. Dort findet man sowohl weitere Befehle für 3D Plots (z.B. `contour3`, `contour`,..) als auch Informationen über weitere Eigenschaften, die an Plots eingestellt werden können.

5.5 Matlab-Movies

Sollen Evolutionsprozesse untersucht werden, kann es manchmal sehr hilfreich sein, Filme (z.B. über die Entwicklung einer Funktion) zu erstellen. Dafür stellt MATLAB zwei Möglichkeiten zur Verfügung. Zum einen können qualitativ hochwertige Filme erstellt werden, die in einem MATLAB -eigenen Format speicherbar sind. Diese Dateien können sehr groß werden. Weiterhin benötigt man zum Abspielen des Films das Programm MATLAB . Unter Umständen können diese beiden Aspekte problematisch werden. Daher gibt es noch die Möglichkeit, einen Film zu erstellen und direkt mit MATLAB in ein so genanntes `avi`-File zu konvertieren. Dieses Format wird von nahezu jeder gängigen Video-Software unterstützt, so dass für das Abspielen des Films MATLAB nicht installiert sein muss. Die Filme, die auf diese Weise erstellt werden, benötigen wenig Speicherplatz, sie sind jedoch qualitativ nicht so hochwertig wie die MATLAB -eigenen Filme.

5.5.1 Matlab-Movies

MATLAB speichert die einzelnen Bilder und spielt sie nacheinander in einem Film ab. Daher ist es notwendig, die jeweiligen Bilder mit gleicher Achsenskalierung und Beschriftung zu wählen. Es bietet sich daher an, vorab Achsen, Title etc. festzulegen und dann die Plots, die zu einem Film zusammengefasst werden sollen, immer in der gleichen Figure aufzurufen. Vorab wird die Anzahl der Bilder im Film und der Name des Films mit dem Befehl `moviein` festgelegt. Nachdem die Bilder mit dem Befehl `getframe` zu einem Film zusammengefügt werden, kann der Film mittels `movie` abgespielt werden. Als Beispiel wird die zeitliche Entwicklung der trigonometrischen Funktion

$$f(x,y,t) = \cos\left(x - \frac{t\pi}{N}\right)\sin\left(y - \frac{t\pi}{N}\right)$$

berechnet und als Movie ausgegeben. Dabei ist N die Anzahl der Bilder. Das zugehörige M-File `matlab_movie.m` sieht wie folgt aus:

```
N = 10;                                 % Anzahl der Bilder im Film
[X,Y]=meshgrid(-pi:.1:pi);              % Gitter erzeugen
M = moviein(N);                         % Initialisierung des Films
```

```
for t=1:N
    f=cos(X-t*pi/N).*sin(Y-t*pi/N);   % aktualisieren der Funktion f
                                      % in jedem Schritt
    meshc(X,Y,f);                     % Plotten, in jedem Schritt
    xlim([-pi,pi])
    ylim([-pi,pi])
    xlabel('x')
    ylabel('y')
    zlabel('f( . ,t_{fix})')
    title('Evolution der Funktion f(x,y,t)=cos(x-(t\pi)/N)sin(y-(t\pi)/N)')
    colorbar
    M(:,t)=getframe;                  % Speichern des aktuellen Bildes
                                      % in der t. Spalte von M
end

movie(M,2);                           % Wiederholen des Films
```

In der Variablen `M` wird der Film gespeichert. In jedem Schritt $t = 1, \ldots, N$ wird das aktuelle Bild in die t. Spalte von M gespeichert (`M(:,t)=getframe`). Mit dem Befehl `movie(M,2)` wird der Film mit dem Titel `M` abgespielt. Die zweite Komponente gibt an, wie oft der Film wiederholt werden soll, in diesem Fall zweimal. Ist ein aufwendiger Film erstellt worden, der anschließend ohne neue Berechnung vorgeführt werden soll, so kann man nach dem Berechnungsdurchlauf die Daten mittels des Befehls `save` speichern:

```
save Daten_zum_Film;
```

Im aktuellen Verzeichnis wird die Datei `Daten_zum_Film.mat` gebildet, die ohne Programmdurchlauf mit

```
load Daten_zum_Film;
```

dazu führt, dass die Variablen (incl. des Films `M`) wieder hergestellt werden. Der Befehl

```
movie(M,3)
```

bewirkt dann das Abspielen des Films ohne neue Berechnungen.

5.5.2 Filme im avi-Format

Zunächst muss im MATLAB File eine `avi`-Datei erstellt werden, in die der Film gespeichert wird. Dies geht mit dem Befehl `avifile`. Das erste Argument im Befehl `avifile` gibt den Namen der `avi`-Datei an. Dann werden Qualitätsparameter aufgerufen. Details zu diesem Thema können bei Bedarf der Matlab-Hilfe entnommen werden. Nachdem die Filmdatei erstellt worden ist, müssen nun in einer Schleife die einzelnen Bilder zu dem Film hinzugefügt werden. Dazu werden die Befehle `getframe` und `addframe` genutzt. Auch hier ist es wieder wichtig, immer die gleichen Achsen zu wählen! Damit die Achsen in jedem Bild des Filmes gleich sind, werden sie mit dem Befehl `get(gcf,'CurrentAxes')` beim ersten Bildaufruf gespeichert. Mit dem Befehl `getframe(gcf)` werden dann die Achsen übergeben. Nachdem die Bilder mit dem Befehl `addframe` zu einem Film zusammengesetzt wurden, muss der Film mit dem Befehl `close` geschlossen werden. Im folgenden Beispiel wird wieder die Evolution der Funktion $f(x, y, t)$ aus dem vorherigen Kapitel in einem Film gespeichert. Der Film trägt den Namen `Beispielfilm.avi`.

```
N = 40;

mov = avifile('Beispielfilm.avi','compression','none')

figure(1)
[X,Y]=meshgrid(-pi:.1:pi);

for t = 0:N

  f = cos(X - (t*pi)/N).*sin(Y-(t*pi)/N);
  meshc(X,Y,f);
  axis([-pi pi -pi pi -1 1])
  xlabel('x')
  ylabel('y')
  zlabel('f( . ,t_{fix})')
  title('Evolution der Funktion f(x,y,t)=cos(x-(t\pi)/N)sin(y-(t\pi)/N)')
  colorbar
  F = getframe(gcf);
  mov = addframe(mov,F);

end

mov=close(mov);
```

Nach dem Programmdurchlauf sollte im aktuellen Verzeichnis eine Datei namens Beispielfilm.avi angelegt worden sein, die nun mit beliebiger Videosoftware unabhängig von Matlab abgespielt werden kann. Bei dem Erstellen des Films ist zu beachten, dass Matlab wirklich Screenshots der Figure zu einem Film zusammenfügt. Wird das Figure-Fenster von einem anderen Fenster überdeckt, so wird dieses im Film gespeichert! Unter Windows ist weiterhin zu beachten, dass bestehende Filmdateien nicht immer überschrieben werden können. Dann muss die Filmdatei vor einem erneuten Programmdurchlauf gelöscht werden.

6 Codeoptimierung

Um MATLAB so effizient wie möglich zu nutzen empfiehlt es sich einige Regeln zu kennen und zu befolgen.

6.1 Funktionsargumente

Damit unötige Speicherkopien von Funktionsargumenten verhindert werden können, sollte man innerhalb einer Funktion nur lesend auf ein Argument zugreifen, da solange es nicht verändert wird keine Kopie erzeugt werden muss.

6.2 for-Schleifen

Sollte man auf die Benutzung von for-Schleifen angewiesen sein, und keine Vektorisierung möglich sein, so ist es wichtig den Schleifenkörper so einfach wie möglich zu halten und nach Möglichkeit nur MATLAB eigene Funktionen zu benutzen.

6.3 Komponentenweise Operationen

Um Komponentenweise Operationen zwischen zwei ungleich dimensionierten Operanden auszufhren kann man den Befehl `bsxfun` benutzen. Dieser ermöglicht zum Beispiel eine Subtraktion eines Vektors von allen Spalten einer Matrix. Mit bsxfun sehr viele binäre Operationen auf allgemeine Matrizen anwenden, beispielhaft seien hier Addition, Subtraktion, Multiplikation, sowie Vergleichsoperationen erwähnt.

6.4 Frühzeitige Speicherreservierung

Insbesondere wenn man sehr große dichtbesetzte Matrizen benötigt, ist es sinnvoll diese so früh wie möglich in der Funktion zu reservieren; zum Beispiel durch den `zeros` Befehl. Auf jeden Fall ist das wiederholte Erzeugen solcher Matrizen innerhalb eines Schleifenkörpers zu verhindern.