Eine kleine Einführung in Matlab

Maike Schulte

WS 06/07

Stand:

24. Oktober 2007

Inhaltsverzeichnis

Dieses "Skript" ist nur eine kleine Hilfestellung für diejenigen Hörer der Vorlesung Numerik I, die noch nie in irgendeiner Weise programmiert haben. Es wird einen Einblick in die Möglichkeiten der Programmierung mit MATLAB geben. Für die Bearbeitung der Programmieraufgaben zur Vorlesung ist folgende Literatur empfehlenswert:

- A. Biran: MATLAB *5 für Ingenieure* (1999)
- C. B. Moler: *Numerical Computing with* MATLAB (2004)
- W. Gander: *Solving problems in scientific computing using Maple and* MATLAB (2002)
- D. Higham: MATLAB *Guide* (2000)

Diese Bücher sind in der Bibliothek des Mathematischen Instituts vorhanden. Weitere MATLAB Literatur gibt es in der ULB, zahlreiche kurze Übersichten und Übungen gibt es im Internet, z.B.

`http://www.rz.fh-ulm.de/~gramlich/matlab_kap2.pdf`
`http://people.inf.ethz.ch/arbenz/MatlabKurs/matlabintro.html`
`http://mo.mathematik.uni-stuttgart.de/kurse/kurs4/`
`http://www.mathworks.com/moler/`

Über den letzten Link kann man sich das Buch *Numerical Computing with Matlab* vom MATLAB-Erfinder Cleve Moler kostenlos herunterladen! Es enthält viele Anwendungsbeispiele und MATLAB-Programmen. Sehr empfehlenswert!

Einige Eigenschaften vorab:

- Im Gegensatz zu z.B. Maple oder Mathematica arbeitet MATLAB auf dem diskreten Niveau. Alle Variablen in MATLAB werden durch Matrizen dargestellt (MATLAB steht für **Mat**rix **Lab**oratory).
- MATLAB unterscheidet bei der Bezeichnung von Variablen und Funktionen zwischen Groß- und Kleinbuchstaben!
- Ein Semikolon am Zeilenende unterdrückt die Ausgabe.
- Datenstrukturen erfordern minimale Beachtung (keine Deklarationen nötig)
- MATLAB verfügt über eine aufwendige Graphik
- MATLAB (und damit MATLAB-Programme) laufen auf allen gängigen Plattformen: Windows, Linux, SUN Solaris, HP-UX 11, MAC OS X

MATLAB ist eine sehr übersichtliche und einfach strukturierte Programmiersprache. Im Gegensatz zu anderen Sprachen ist die Syntax sehr eingängig. Es sind viele (mathematische) Funktionen und Standardalgorithmen bereits vorgefertigt, so dass nicht alles von Hand programmiert werden muss. Weiterhin ist die Visualisierung von Ergebnissen sehr leicht möglich. Allerdings sind Berechnungen mit Matlab im Gegensatz zu z.B. C++ sehr langsam, so dass diese Programmiersprache nicht sonderlich praxisrelevant ist.

1 Matlab starten

1.1 In der Uni

Die Kursteilnehmer legen bitte in ihrem Home-Verzeichnis einen Ordner `MatlabKurs` an! Dazu öffnet man ein x-term/Terminal und führt dort den Befehl `mkdir MatlabKurs` aus. Mit dem Befehl `cd MatlabKurs` kann man in dieses Verzeichnis wechseln. An den Unix- und Linux-Rechnern im Institut kann man MATLAB starten, in dem man im x-term folgende Befehle nacheinander eingibt:

- `. /etc/profile`
- `environ numeric`
- `matlab &`

Nach einigen Sekunden sollte MATLAB starten.
Achtung: Die Uni verfügt über 40 Lizenzen für Matlab, d.h. dass es nur 40mal gestartet werden kann. Sind alle Lizenzen vergeben, erscheint im x-term eine Fehlermeldung. Bei starker Nutzung von Matlab, z.B. während des Kurses im SRA, kann es etwas länger dauern, bis MATLAB gestartet ist. Bitte den Befehl nur einmal aufrufen!

1.2 Zu Hause - Zentrale Lizenz für Matlab

Am ZIV existieren so genannte "concurrent licenses" für die MATLAB-Suite, die aus dem Basisprodukt MATLAB, dem Simulationstool Simulink und der Symbolic Math Toolbox für symbolische Berechnungen besteht. Das ZIV beabsichtigt diese Produktfamilie zukünftig unter Wartung zu nehmen. Dabei ist es möglich, schon bestehende Lizenzen in der Universität kostengünstig mit einzubeziehen.

Die vom ZIV bereitgestellten Lizenzen sind an einen Lizenzmanager gebunden. Ein Rechner auf dem MATLAB installiert bzw. genutzt wird, muss daher an das Netz angeschlossen sein. Ausgangspunkt für die Installation bzw. Nutzung von MATLAB ist das zentrale Dateisystem des ZIV, das auch für eine Reihe anderer Produkte genutzt wird. Im Folgenden wird kurz beschrieben, welche Vorbereitungen zu treffen sind, um Zugriff auf die Verzeichnisse zu erhalten.

Voraussetzungen:
Auf dem Rechner muss eine Verbindung zur WWU per Funk-LAN, pLANet oder VPN eingerichtet und aktiviert sein (für Details siehe `https://zivwiki.uni-muenster.de/cgi-bin/view/Anleitungen/`).
Auf dem Rechner muss bei der Installation eine Verbindung zum Softwareverzeichnis der WWU eingerichtet sein (siehe `https://zivwiki.uni-muenster.de/cgi-bin/`

`view/Anleitungen/DfsMount`)

Installation:
Die Installationsdateien finden sich im Softwareverzeichnis im Verzeichnis
`a\urz\matlab\*Version*\DVD`.
Version ist die Versionsnummer, aktuell ist R2007b. Kopieren Sie die Dateien plp.txt und license.dat in diesem Verzeichnis auf Ihren lokalen Rechner. Starten Sie die Installation im Verzeichnis, das zu Ihrer Rechnerarchitektur gehört. Sie werden nach einem PLP gefragt, das ist der Inhalt der Datei `plp.txt`, die Sie kopiert haben. Anschließend werden Sie nach einer Lizenzdatei gefragt, das ist die Datei `license.dat`, die Sie kopiert haben.

2 Grundlagen

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

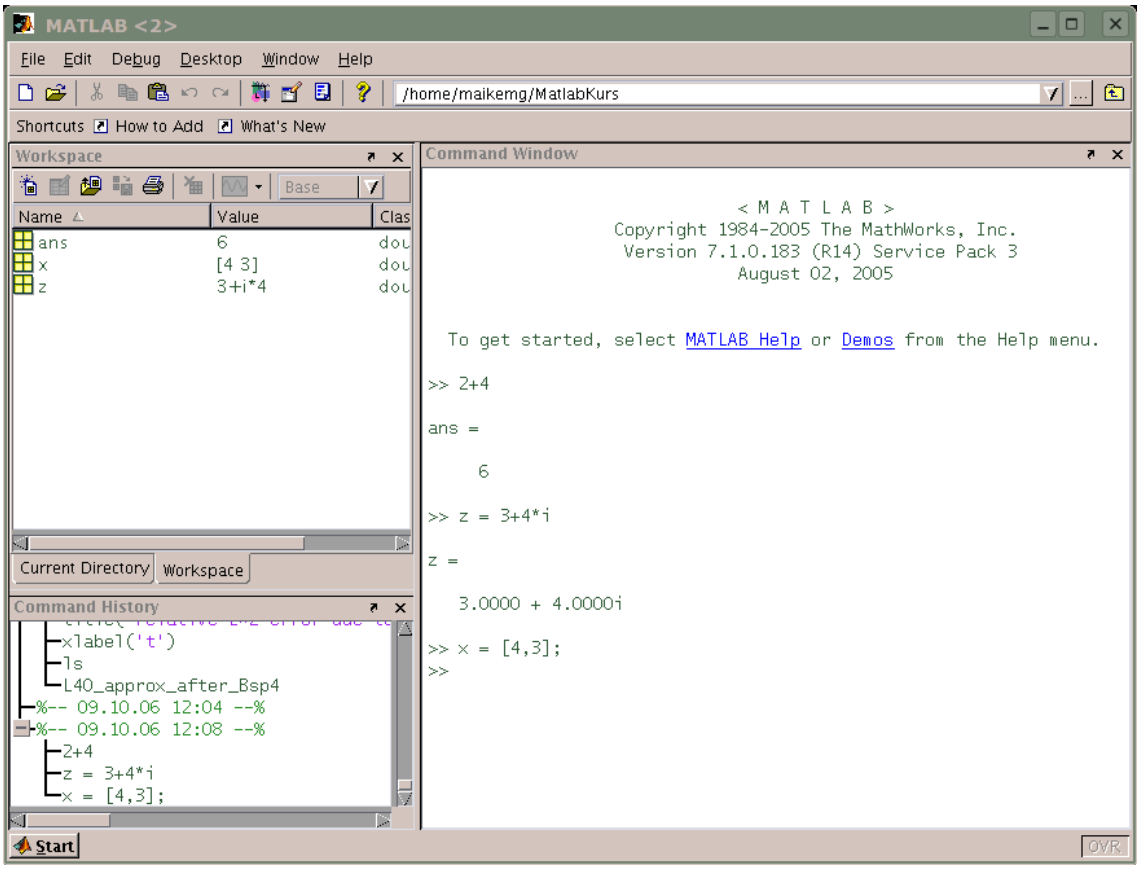


Abbildung 1: Bedienungsoberfläche von Matlab

Die Bedienungsoberfläche teilt sich in mehrere Bereiche auf. Das Befehls-Fenster (*Command Window*) befindet sich rechts und 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 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 directory*) befinden, welches in der Befehlsleiste oben ausgewählt wird. In unserem Fall ist dies das Verzeichnis `MatlabKurs`.

Im Teilfenster links unten (*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 Teilfenster links oben werden standardmäßig die Dateien des gegenwärtigen Verzeichnisses (*current directoy*) angezeigt. Unser Ordner `MatlabKurs` ist noch leer, so dass zu Beginn keine Dateien angezeigt werden. Man kann den Reiter auch von *current directoy* auf MATLABS Arbeitsspeicher (*workspace*) umstellen. Im Arbeitsspeicher befinden sich die Variablen, die Sie angelegt haben. Durch einen Doppelklick auf den Variablennamen wird ein *Arrayeditor* geöffnet, mit welchem man die Matrixelemente editieren kann.

Unter der Option *Desktop* kann man die Standardeinstellungen bei Bedarf ändern. Die MATLAB Hilfe kann mit der Maus an der oberen Befehlsleiste via *Help* → MATLAB *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!

3 Matrizen, Vektoren & Operatoren

3.1 Erste Eingaben

Für eine erste einfache Rechnung werde z.B.

```
>> 2-4
```

eingegeben. MATLAB antwortet mit dem 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

ans =
     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.

3.2 Matrizen

Werden Matrizen direkt eingegeben, ist folgendes zu beachten:

- Die einzelnen Matrixelemente werden durch Leerzeichen oder Kommas voneinander getrennt
- Das Zeilenende 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.

Zum Beispiel wird die 2×4-Matrix

$$\begin{pmatrix} 1 & -3 & 4 & 2 \\ -0.5 & 8 & 5 & 5 \end{pmatrix}$$

wird wie folgt in MATLAB eingegeben und der Variable `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

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

Grand total is 18 elements using 144 bytes
```

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 empfliehlt sich daher, Zeilen nicht zu lang werden zu lassen.

3.3 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 sonderlich gängige 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 $n_1 = m_2 = \tilde{n}$ und $n_2 = m_1 = \tilde{m}$ kann MATLAB eine gewöhnliche Matrixmultiplikation durchführen. `C=A*B` liefert dann die $(\tilde{n}, \tilde{m})$-Matrix

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

In allen anderen Fällen 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 $n_2 = 1$. 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.

Einige Beispiele:
Es seien die Matrizen, Vektoren und Skalare

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

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
```

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 =

    3.7083
```

berechnet werden. Prinzipiell MATLAB kann 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.

3.4 Der Doppelpunkt-Operator, Aufruf von einzelnen Matrixelementen und weitere Matrix-Spielereien

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 mit direkt via

```
>> B(2,3)

ans =

     6
```

ausgegeben. Matrizen können nahezu beliebig miteinander kombiniert werden. Dabei steht das , (oder das Leerzeichen) immer für eine neue Spalte, der ; dagegen 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, wie der Befehl

```
>> size(C)

ans =

     3     6
```

bestätigt. 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. 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: Mittelwaagerechten gespiegelt (*left/right* und *up/down*) werden:

```
>> 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 der Befehle `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
```

3.5 Zahldarstellung und Konstanten in Matlab

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.

3.6 Einige vorgefertigte Funktionen

Es sind sehr viele 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`	$\|\ \|$	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
`cumprod`	kummulatives Produkt aller Elemente
`cumsum`	kummulative Summe aller Elemente
`transpose`	transponieren
`det`, `inv`	Determinante, Inverse einer Matrix
`diag`	Diagnolen von Matrizen (s.o.)
`fliplr, flipud`	Spiegelungen von Matrizen (s.o.)

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.

4 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!

5 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 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!

6 Polynome

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.

7 Lineare Gleichungssysteme

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

$$\begin{aligned} x_1 + 2x_2 + 3x_3 &= 4 \\ 2x_1 + 3x_2 + 4x_3 &= 5 \\ 4x_1 + x_2 + 5x_3 &= 1 \\ \Leftrightarrow Ax &= b \end{aligned}$$

mit

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

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
```

8 Programmierung in Matlab

8.1 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.
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)
```

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 *C*ommand 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

A =

   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

>>
```

8.2 Funktionen

Wie schon in Kapitel 8.1 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. 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 Programm dienen, welches zwei Werte a und b einliest und dann Berechnungen zu dem Rechteck $a \cdot b$ durchführt. Dazu schreiben wir zunächst Funktionen `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. Sollen mehrere Werte innerhalb einer Funktion berechnet werden, schreib 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. Weiterhin soll die Diagonale des Rechtecks mit dem Programm `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 Programm `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 Diagonaledes 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)])
```

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.

Vorsicht bei der Vergabe von Funktionsnamen! Sehr viele Fehler entstehen 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

Beispiel:

```
>> exist d

ans =

     1
>> exist cos

ans =

     5
```

8.3 Funktionen als Argumente von Funktionen in Matlab – Verwendung von Zeigern

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

$$f'(x) \approx D^+ f(x) := \frac{f(x+h) - f(x)}{h}, \quad h > 0 \tag{1}$$

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 `Ableitung` programmiert, die eine vorgegebene Funktion f, einen Funktionswert x und eine Schrittweite h einliest und damit die näherungsweise Ableitung $f'(x)$ nach Gleichung (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 `Ableitung` 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 `Ableitung` 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 = Ableitung(f,x,h)
% Diese Routine berechnet die Ableitung 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 `Ableitung.m` abgespeichert. Nun muss noch das Hauptprogramm `Beispiel.m` geschrieben werden, in dem die Funktionen `f` und `Ableitung` aufgerufen werden. Bei dem Funktionsaufruf `Ableitung` im Hauptprogramm muss allerdings berücksichtigt werden, dass eines der zu übergebenden Argumente eine Funktion ist. Es wird ein *Zeiger* auf die Funktion (handle in Matlab) übergeben. Den Zeiger erhalten wir, indem vor dem Funktionsnamen das Zeichen @ angefügt wird.

```
% Programm, welches fuer gegebene Werte x eine Funktion f aufruft und dann
% mit Hilfe der Funktion Ableitung.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 = Ableitung(@f,x,h);
```

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

8.4 Schleifen und Kontrollstrukturen

In diesem Kapitel sollen kurz die wichtigsten Schleifen und Kontrollstrukturen `for`, `if-else` und `while` vorgestellt werden.

Die if-else-Schleife

Mit der `if-else`-Schleife können Fallunterscheidungen durchgeführt werden. Das folgende Beispiel zeigt, wie die `if-else`-Schleife 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.
% % % % % % % % % % % % % % % % % % %

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 for-Schleife

Die `for` - Schleife eignet sich, wenn Berechnungen wiederholt durchgeführt werden. Es sei vorab erwähnt, dass das Arbeiten mit Schleifen sehr einfach, jedoch sehr oft nicht effizient ist, wie wir später sehen werden. Sei als Beispiel eine (n, n)-Matrix A mit den Einträgen

$$a_{j,k} = \frac{1}{j+k-1}$$

für $j = 1, \ldots, n$ und $k = 1, \ldots, n$. Mit Hilfe der `for` Schleife kann man folgendes Programm `For_Schleife.m` schreiben:

```
% For_Schleife.m dient zum Lernen der for Schleife
% und berechnet eine Matrix.

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

% Initialisieren der Matrix:
A = zeros(n);

% Aufbau der Matrix:
for j=1:n
    for k=1:n
        A(j,k)=1/(j+k-1);
    end
end
```

Die Berechnung mit `for` - Schleifen kann sehr ineffizient sein. Die Vektorschreibweise ist oft viel nützlicher, wie uns das folgende Beispiel zeigt. Sie wird erreicht mit der Doppelpunktnotation (siehe Kapitel 3). Die Befehle `tic` und `toc` messen die Zeit, die das Programm ab dem Zeitpunkt `tic` bis zum Aufruf `toc` benötigt. Damit kann man überprüfen, welche der Varianten effizienter ist.

Beispiel:
Sei v ein Vektor der Länge $N = 50\,000$ mit $v_k = k^2$, $k = 1, \ldots, n$. Definiere einen Vektor w mit $w_k = v_{k+1} - v_{k-1}$, $k = 2, \ldots, N-1$. Man berechnet nun den Vektor w auf zwei verschiedene Weisen

1.) Berechne w mit einer gewöhnlichen `for`-Schleife und messe die benötigte Zeit mit den Befehlen `tic, toc`

2.) Berechne w mittels Vektorschreibweise und messe die benötigte Zeit

Dies mach das folgende Programm:

```
clear all
N=50000;

% Definiere v:
for k=1:N
    v(k)=k^2;
end

tic
for k=2:N-1
    w1(k)=v(k+1)-v(k-1);
end
toc

tic
w2(2:N-1)=v(3:N)-v(1:N-2);
toc
```

Die Ausgabe zeigt, dass die zweite Variante deutlich schneller ist:

```
Elapsed time is 3.530050 seconds.
Elapsed time is 0.001568 seconds.
```

Der erste Algorithmus benötigt ~ 2000 mal so viel Zeit wie der zweite. Man kann sich vorstellen, dass dieser Faktor bei Programmen, die eine Laufzeit von Stunden haben, sehr entscheiden sein kann!
Die Laufzeiten sind natürlich Computer- und auslastungsabhänging. Im Allgemeinen sind aber alle `for`-Schleifen sehr zeitaufwendig. Da MATLAB Matrix- und Vektormultiplikationen parallel ausführen kann, sind diese Berechnungen meistens und besonders für große n effektiver.

Die while - Schleife

Eine weitere Kontrollstruktur ist die `while`-Schleife. Auch sie soll wieder anhand eines Beispiels erläutert werden. Betrachten wir die Division zweier ganzer Zahlen mit Rest. Seien $x, y \in \mathbb{N}$ gegeben und $q, r \in \mathbb{N}$ 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
% x = qy + r 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;

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

% Speichere x fuer spaeter:
x_old=x;
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.

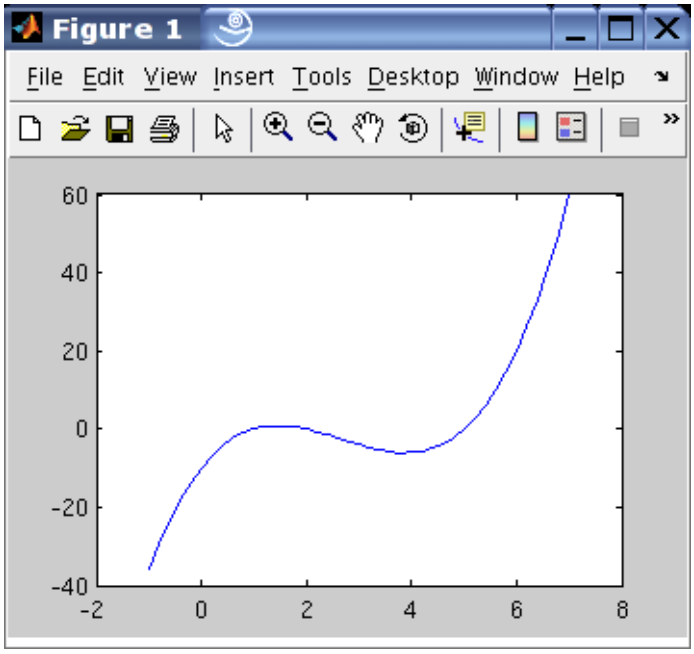
9 Graphische Ausgaben

Ein großer Vorteil von MATLAB liegt in der sehr einfachen graphischen Darstellung von Ergebnissen. Beginnen wir zunächst mit der zweidimensionalen graphischen Ausgabe. In Abschnitt 4 haben wir die Wertetabelle eines Polynoms `p=[1 -8 17 -10]` berechnet. Diese soll nun geplottet werden. Wir erinnern uns: die 17 x-Werte waren durch den Vektor `x=[-1:0.5:7]` gegeben, die passenden y-Werte lieferte der Befehl `y=polyval(p,x)`. `y` ist nun ebenfalls ein 1×17-Feld. Zum Öffnen eines Bildes, einer so genannten *figure*, muss zunächst der Befehl

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

aufgerufen werden, es öffnet sich ein leeres plot-Fenster. Die Nummer in den Klammern kann beliebig gewählt werden. Ruft man den Befehl `figure()` nicht auf, so werden alte Bilder überschrieben. Eine zweidimensionale wird mit dem Befehl `plot` generiert.

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



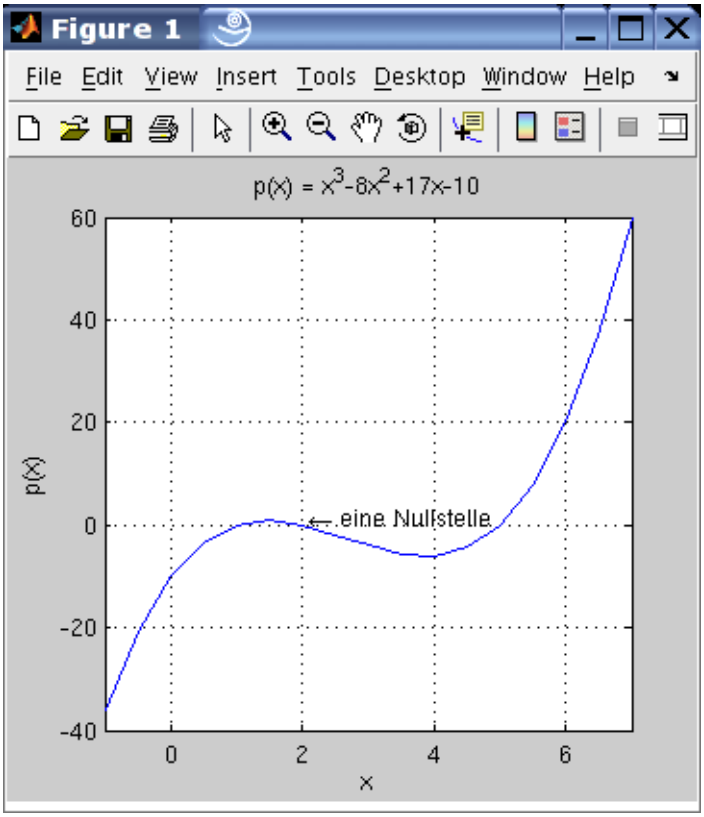
Es gibt verschiedene Möglichkeiten, diesen plot nun zu beschriften. Zum einen kann man sich in der *figure* durch die Menüs klicken und dort Achsenbeschriftungen, -skalen, -bezeichnungen, Titel, Legenden etc. eingeben. Da man diese Einstellungen aber nicht speichern kann und für jede Graphik neu erstellen muss, wird hier nur die (meines Erachtens) bessere Methode vorgestellt, wie man die *figure* direkt aus dem *M-File* heraus zu bearbeitet. Dies ist in der Programmierung wesentlich praktischer und weniger arbeitsintensiv, da die Einstellungen gespeichert werden können und nicht bei jedem Bild neu erarbeitet werden müssen.
Hier werden nur einige wesentliche Möglichkeiten der Graphik-Bearbeitung vorgestellt. MATLAB verfügt über viel mehr als die hier vorgestellten Möglichkeiten der graphischen Ausgabe. Weitere Informationen finden sich in den Literaturangaben oder in der Matlab-Hilfe!
Die folgende Tabelle zeigt einige Möglichkeiten zur Veränderung der graphischen Ausgabe:

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
`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','Punkt 1','Punkt2',...)`	Bennenung der x-Achsepunkte

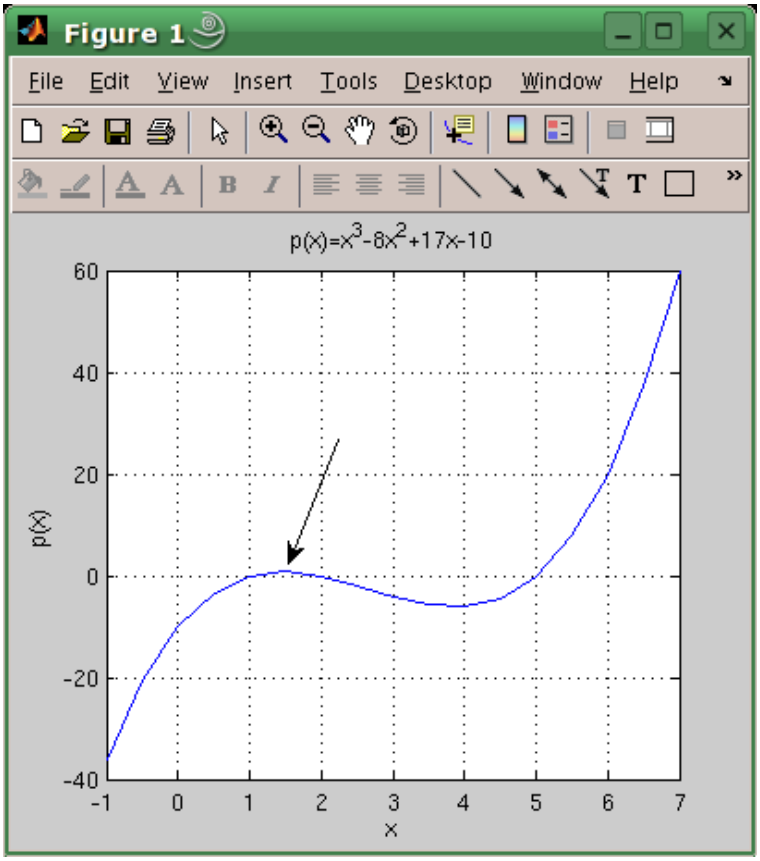
Die Bezeichnungen können LaTeX codes enthalten, MATLAB interpretiert z. B. die Zeichenfolge `\psi \rightarrow x^4 f(x_6) ||z||_{x\in\Omega}` wie LaTeX als $\psi \rightarrow x^4\ f(x_6)\ ||z||_{x\in\Omega}$. Der Aufruf eines plots in einem M-File könnte also folgendermaßen aussehen:

```
figure(1)
plot(x,y)
grid on
axis([-1,7,-40,60])
xlabel('x')
ylabel('p(x)')
title('p(x) = x^3-8x^2+17x-10')
text(2,0,' \leftarrow eine Nullstelle')
```

Nun sieht der plot wie folgt aus:

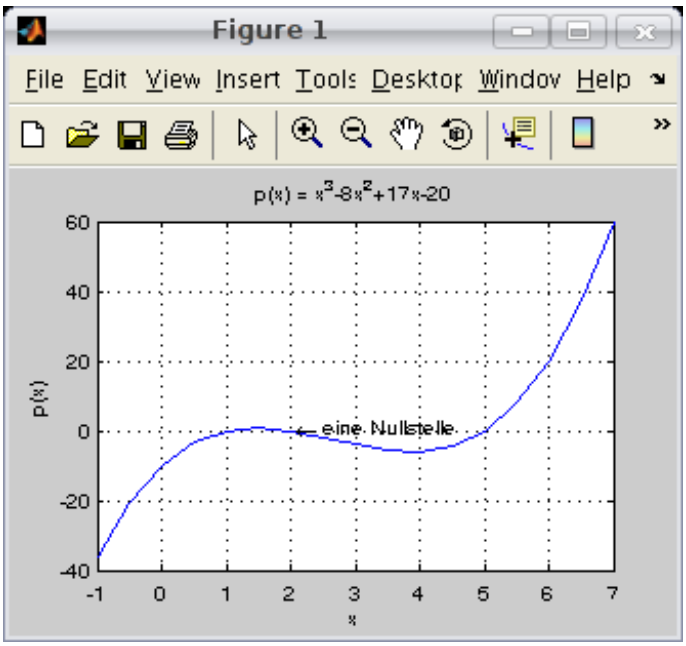


Das Einfügen von Text oder Pfeilen in einer Graphik kann auch über das Menü der *figure* realisiert werden. Unter *View* kann man in der Menüleiste der *figure* die Optionen *Plot Edit Toolbar* wählen. In einer neuen Zeile gibt es dann die Möglichkeit, durch "Klickerei" Pfeile und Text einzufügen und im plot auszurichten/zu drehen. Man hat auch die Möglichkeit, Kreise, Quadrate einzufügen, kann Texte farbig gestalten und ausrichten. Der folgende Screenshot zeigt die neue Menüleiste:



Das manuelle Verändern von Graphiken ist sinnvoll, wenn eine Graphik für einen Ausdruck (z.B. Diplomarbeit) erstellt werden soll. Die Einstellungen werden nämlich nicht gespeichert und müssen dann für jede Graphik neu vorgenommen werden. Daher sollte man sich angewöhnen, die grundlegenden Beschriftungen im M-file vorzunehmen.

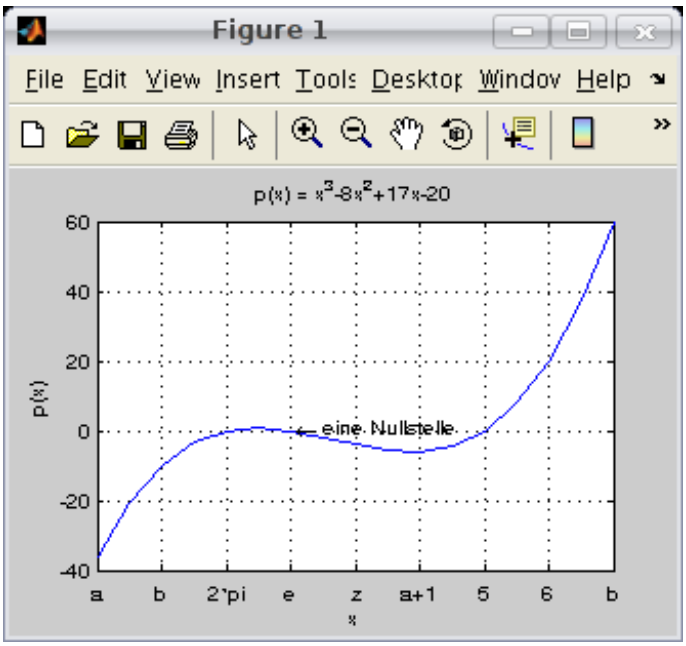
Die Achsenpunkte können via der Befehle
`set(gca,'XTick','Vektor')`
`set(gca,'XTickLabel','Punkt 1','Punkt2',...)`
geändert werden. Dabei steht `gca` für *get current axis*. Statt der Option `'XTick'`, `'XTickLabel'` können auch analog `'YTick'`, `'YTickLabel'` und analog für Z für die y-, bzw. z-Achse verwandt werden. Der Befehl `set(gca,'XTick','Vektor')` ändert die Punkte an der x-Achse, die explizit markiert sind. Bisher war dies jeder 2. ganzzahlige Wert von -1 bis 7. Soll nun jeder ganzzahlige Wert eine Markierung erhalten, kann das mit dem Befehl`set(gca,'XTick',-1:1:7)` realisiert werden:



Sollen nun auch andere Bezeichnungen der x-Achsenmarkierungen eingeführt werden, kann dies mit `set(gca,'XTickLabel','Punkt 1','Punkt2',...)` geschehen. Es ist wichtig, dass genauso viele Punkte angegeben werden, wie Markierungen existieren. In unserer Graphik bewirkt z.B. der Befehl

```
set(gca,'XTickLabel',{'a','b','2*pi','e','z','a+1','5','6','b'})
```

eine folgende Ausgabe:



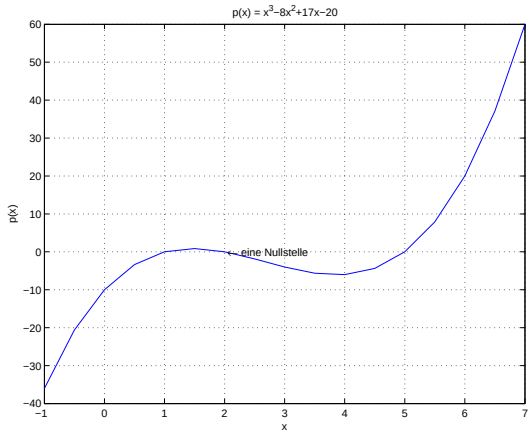
Leider werden in der Achsenmarkierung Latex-Fonts *nicht* berücksichtigt (z.B. würde eine Eingabe `\pi` die Bezeichnung `\pi` statt π hervorrufen).

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. Die resultierende Graphik sieht so aus:



Hier eine kleine Übersicht, welche Optionen der `print` Befehl unter anderem hat (es muss jeweils ein Ausgabename ohne Dateiendung hinter den Befehl gesetzt werden)

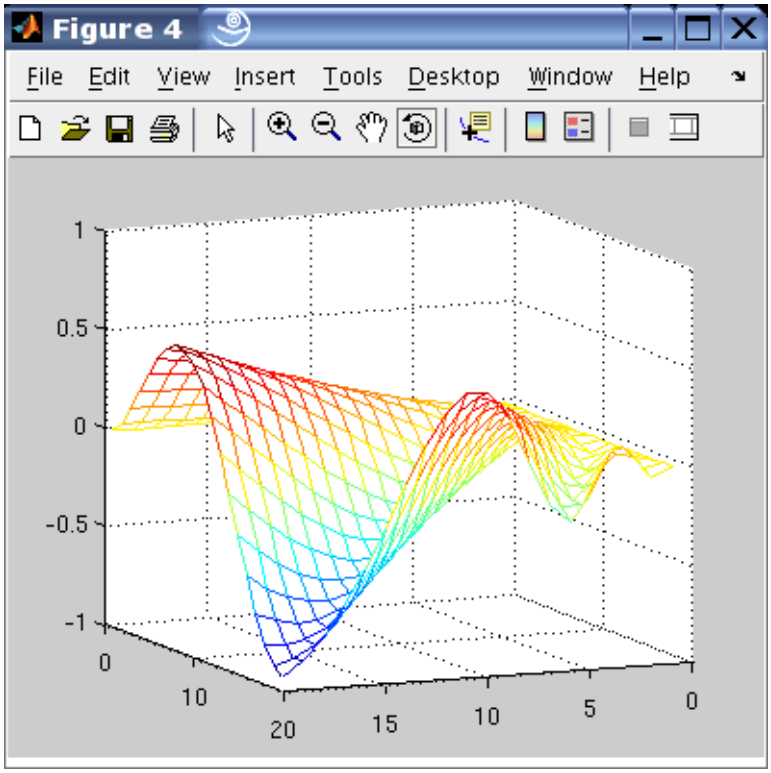
Befehl	Beschreibung
print -depsc	Export als farbige eps-Datei (Postskript, Vektordatei)
print -deps	Export als schwarz/weiße eps-Datei (Postskript, Vektordatei)
print -dill	Export als Adobe Illustrator Datei (Vektordatei)
print -djpeg	Export als jpg-Datei (Bitmapdatei)
print -dtiff	Export als tiff-Datei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

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.

3D Graphiken

Die bisher vorgestellten Graphik-Bearbeitungsbefehle sind auch für 3D Visualisierungen gültig. Statt des Befehls `plot(x,y)` wird in der 3D Visualisierung `mesh(y,x,f)` (Gitterplot) oder auch `surf(y,x,f)` (Oberflächenplot), `contour(y,x,f)` (Contour-

plot) benutzt. Dabei ist x ein Vektor der Dimension n, y ein Vektor der Dimension m und f ein array der Größe (n, m). Vorsicht: Bei diesen Befehlen wird wirklich erst das y-Feld angegeben! Haben x und y nicht die gleiche Dimension führt dies anderenfalls zu Fehlerausgaben! Das folgende Bild zeigt einen Plot, der mit dem `mesh(y,x,f)`-Befehl erzeugt wurde.

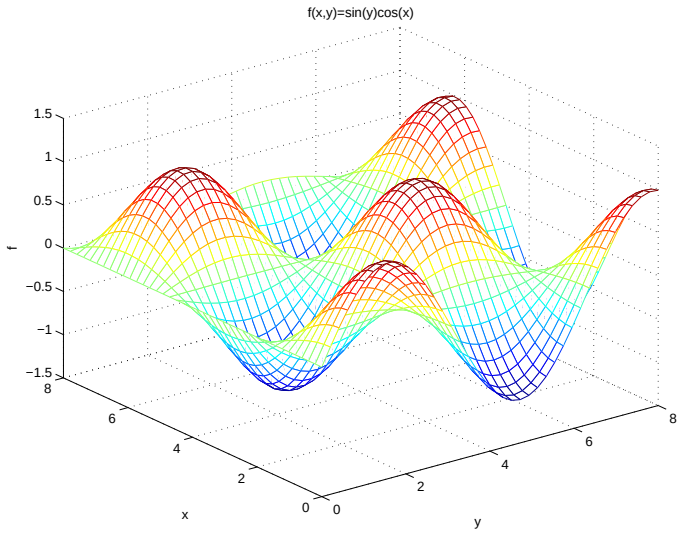


Eine komfortable Syntax zum Erstellen von 3D plots bietet der `meshgrid`-Befehl. Mit ihm können Gitter automatisch erzeugt werden, die direkt in Funktionen eingesetzt werden können.

Beispiel:

```
L = 8;                      % will auf Gebiet [0,L]x[0,L] plotten
N = 40;                     % Anzahl der Gitterpunkte
h = L/(N-1);                % Schrittweite
[x,y] = meshgrid(0:h:L);    % Erzeuge Gitter in xy-Ebene
f = cos(x).*sin(y);         % berechne f(x,y)
```

```
figure(2)
mesh(y,x,f)
xlabel('y')
ylabel('x')
zlabel('f')
title('f(x,y)=sin(y)cos(x)')
axis([0,8,0,8,-1.5,1.5])
```



Speziell für die 3D-Visualisierung gibt es noch viele weitere Optionen (z. B. Contour-Plots, Richtungsfelder, durchsichtige Plots, verschiedene Farbschemata, ...). Bei Interesse wird hier die Matlab-Hilfe als Literatur empfohlen.

9.1 Mehrere Plots in einer figure

Es gibt verschiedene Möglichkeiten, mehrere Plots in einer figure unterzubringen. Der Befehl `subplot` bildet mehrere Graphiken in einer figure nebeneinander ab. Der Aufruf

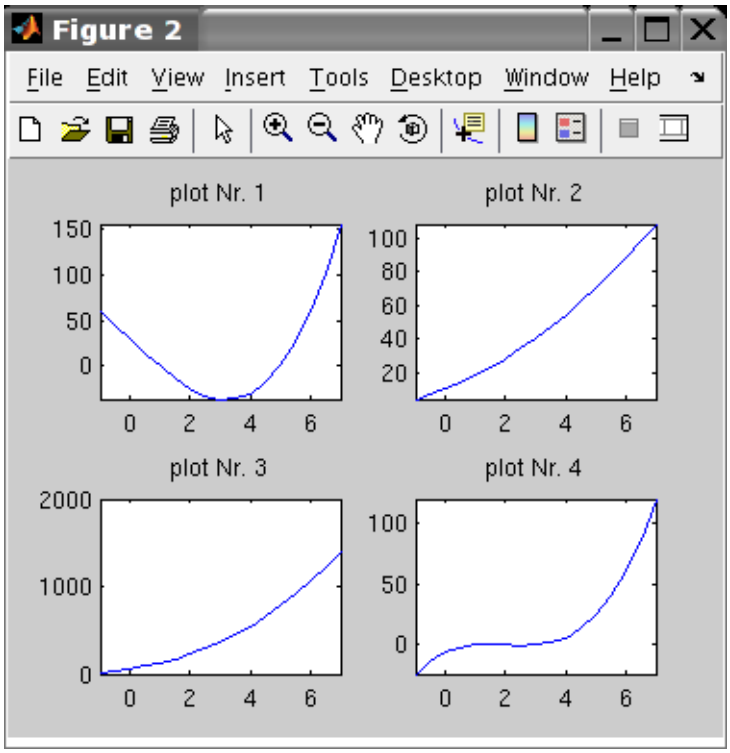
```
figure(2)
subplot(n,m,Bildnummer)
plot(x,y)
```

erzeugt eine figure 2, die n Bilder in einer Reihe und m Spalten erzeugt. Sollen zum Beispiel vier Datensätze $(x_1, y_1), \ldots, (x_4, y_4)$ in einer figure verglichen werden, so kann man dies mittels

```
figure(2)
subplot(2,2,1)
```

```
plot(x_1,y_1)
subplot(2,2,2)
plot(x_2,y_2)
subplot(2,2,3)
plot(x_3,y_3)
subplot(2,2,4)
plot(x_4,y_4)
```

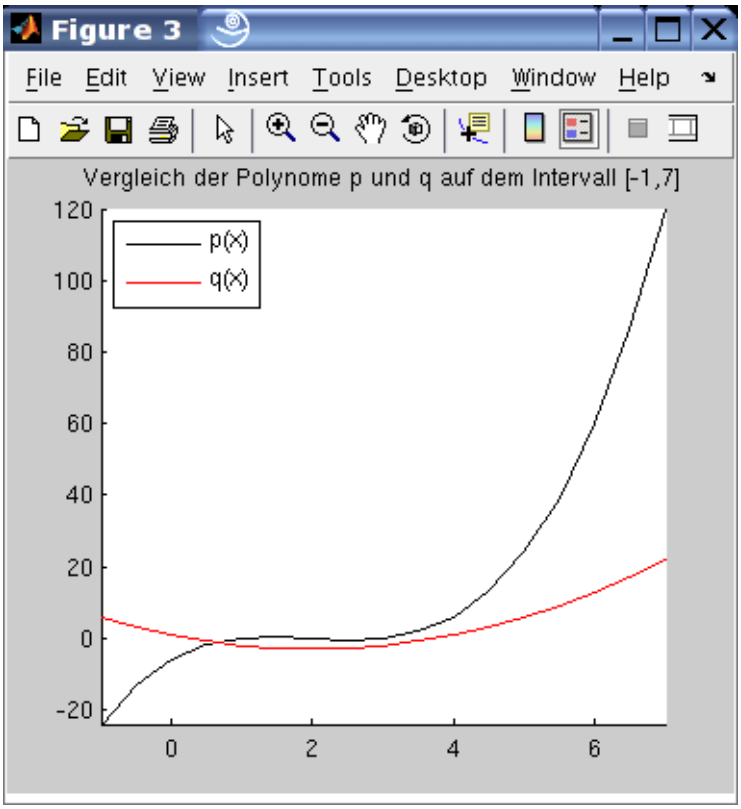
erreichen. Es werden alle vier Plots in einer figure dargestellt, in $n = 2$ Zeilen und $m = 2$ Spalten:



Manchmal kann es nützlich sein, Plots übereinander gelegt in einem Bild zu vergleichen. Dies kann mit den Befehlen `hold on; hold off` in MATLAB durchgeführt werden. Sei wiederum das Polynom $p(x) = x^3 - 8x^2 + 17x - 10$ gegeben und weiterhin ein Polynom $q(x) = x^2 - 4x + 1$, diese Polynome sollen in einem Bild für den oben gegebenen Wertebereich `x=[-1:0.5:7]` geplottet werden. Es sei also `y1=polyval(p,x)` und `y1=polyval(q,x)`.

```
figure(3)
plot(x,y1,'k')
hold on
```

```
plot(x,y1,'r')
hold off
legend('p(x)','q(x)',2)
title('Vergleich der Polynome p und q auf dem Intervall [-1,7]')
```



Der dritte Aufruf `'k'` und `'r'` bezeichnet die Farbwahl. Hier können auch verschiedene Strich- und Punktarten gewählt werden:

Befehl	Farbe	Befehl	Linenstil
`'y'`	gelb	`'-'`	durchgezogene Linie
`'m'`	magenta	`'--'`	gestrichelte Linie
`'c'`	cyan	`':'`	gepunktete Linie
`'r'`	red	`'-.'`	Strich-Punkt-Linie
`'g'`	green	`'none'`	keine Linie
`'b'`	blue		
`'w'`	white		
`'k'`	black		

Weitere so genannte *Line Properties* können unter diesem Suchbegriff in der Hilfe nachgelesen werden.

9.2 Matlab-Movies

Sollen evolutionäre Prozesse 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-eigenem 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. Es kann allerdings zu Problemen unter Unix/Linux kommen.

Matlab-Movies

MATLAB speichert die einzelnen Bilder ab 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 muss die Anzahl der Bilder im Film und der Name des Films mit dem Befehl `moviein` festgelegt werden. 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

$$f(x,y,t) = \cos\left(x - \frac{t\pi}{N}\right)\sin\left(y - \frac{t\pi}{N}\right) \tag{2}$$

berechnet und als Movie ausgegeben. Dabei ist N die Anzahl der Bilder.

```
close all
clear all

N = 10;
M = moviein(N);

[X,Y]=meshgrid([-pi:.1:pi]);
axis([-pi pi -pi pi -1 1])
for t=1:N
    f=cos(X-t*pi/N).*sin(Y-t*pi/N);
    meshc(X,Y,f);
    M(:,t)=getframe
end

movie(M,2);
```

In der Variablen `M` wird der Film gespeichert. In dem aktuellen Ordner entsteht mit der Programmdurchführung eine Datei `M.mov`. Mit dem Befehl `M = moviein(N)` wird festgelegt, dass der Film die Bezeichnung `M` haben wird und N Bilder enthält. 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.

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. In dem Beispiel unten sind die Parameter auf höchste Qualität eingestellt. Weitere Werte und Optionen kann man bei Bedarf der Matlab-Hilfe entnehmen. 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`.

```
close all
clear all
N = 40;

mov = avifile('Beispielfilm.avi','compression','none','quality',100)

figure(1)
[X,Y]=meshgrid([-pi:.1:pi]);
f = cos(X-0*pi/N).*sin(Y-0*pi/N);
meshc(X,Y,f);
axis([-pi pi -pi pi -1 1])
get(gcf,'CurrentAxes')
F=getframe(gcf);
mov=addframe(mov,F);

for t=1:N
    f = cos(X-t*pi/N).*sin(Y-t*pi/N);
    meshc(X,Y,f);
    axis([-pi pi -pi pi -1 1])
    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.