MATLAB
Eine freundliche Einführung

JÖRN BEHRENS und ARMIN ISKE

{behrens,iske}@mathematik.tu-muenchen.de

Lehrstuhl für Numerische Mathematik
und Wissenschaftliches Rechnen
Technische Universität München

http://www-m3.mathematik.tu-muenchen.de/m3/

Version **1.1**

München, 26. Februar 1999

Adresse der Autoren:

Jörn Behrens und Armin Iske
Technische Universität München
Zentrum Mathematik
D-80290 München
{behrens,iske}@mathematik.tu-muenchen.de

Elektronische Version verfügbar unter

http://www-m3.mathematik.tu-muenchen.de/m3/ftp/matlab.ps

Vorwort

Das vorliegende Material entsteht momentan aus einem Kurs, der sich begleitend zur Vorlesung *Numerische Mathematik 1* und zum *Numerischen Praktikum 1* an der Technischen Universität München im Wintersemester 1998/99 entwickelt. Insbesondere wird der Inhalt in der Anfangsphase sehr dynamisch umfangreicheren Änderungen unterworfen sein.

Der Kurs eignet sich besonders für Studierende aller naturwissenschaftlichen und technischen Fachrichtungen im Grundstudium. Es werden *sehr geringe* Vorkenntnisse im Umgang mit Rechnern vorausgesetzt, jedoch keinerlei Erfahrungen mit Betriebssystemen und Programmiersprachen.

Was ist anders an diesem Kurs im Vergleich zu zahlreichen Kursen über MATLAB bzw. warum erhebt ausgerechnet dieser Kurs den Anspruch, *freundlich* zu sein?

- Der Kurs ist praxisorientiert und interaktiv. Die Inhalte des Kurses werden durch Beispiele erarbeitet. Idee dieses Kurses ist es, daß die Teilnehmer das vorliegende Skript neben dem Rechner lesen, und die einzelnen Beispiele sofort ausprobieren. Diese Vorgehensweise bietet sich bei einem interaktiven System wie MATLAB an.
- Die gewählte Sprache ist informell und leicht verständlich. Darüber hinaus wird der Kurs in deutscher Sprache gehalten. Die Teilnehmer werden nicht mit englischen Computer-Fachbegriffen überladen. Unser Ziel ist es, daß selbst die blutigsten Anfänger dieses Dokument vom Anfang bis zum Ende verstehen werden, und nicht etwa nach ein paar Seiten aus Verzweiflung diesen Kurs verlassen.
- Der Kurs ist sehr direkt. Erklärtes Ziel ist es, ohne große Umwege zum Wesentlichen zu kommen. Die Teilnehmer sollen effektiv lernen.

Wir hoffen, daß wir den obigen selbstauferlegten Ansprüchen im Verlaufe des Kurses gerecht werden können. Kritik sowie Anregungen seitens der Teilnehmer und Tutoren werden für uns dabei von essentieller Bedeutung sein. Wir bitten daher ausdrücklich um kritische Kommentierung des zugrundeliegenden Dokumentes.

An dieser Stelle danken wir Professor Hans-Georg Feichtinger (Universität Wien) für seine hilfreichen Anmerkungen zu einer Vorversion dieses Manuskriptes sowie unseren Kollegen Dipl.-Math. Susanne Ertel und Dipl.-Math. Michael Lintner, die sich freiwillig als “Betatester” der ersten Version zur Verfügung gestellt haben.

München, im Februar 1999 Jörn Behrens und Armin Iske.

Inhaltsverzeichnis

Kapitel 1

Einführung

Dieses Dokument beinhaltet eine Einführung in MATLAB (für *MATrix LABoratory*). MATLAB ist ein interaktives System, das sich für wissenschaftliche numerische Berechnungen und zur Visualisierung in fast allen Bereichen technischer und naturwissenschaftlicher Disziplinen gewinnbringend einsetzen läßt.

Wie der Name MATLAB schon vermuten läßt, sind die Objekte, mit denen MATLAB arbeitet, Matrizen. Skalare werden folglich als Matrizen der Dimension 1×1 behandelt, Vektoren als Matrizen der Dimension $1 \times n$ bzw. $m \times 1$. Wir setzen daher in den folgenden Kapiteln voraus, daß unsere Leserinnen und Leser wissen, was eine Matrix ist.

Um die Darstellung zu vereinfachen, werden wir folgende Notationen verwenden: Eine Eingabe in der MATLAB-Kommandozeile wird so dargestellt:
`>> a=[1 2 3];`
Dabei soll `>>` die Eingabeaufforderung (*prompt*) des MATLAB Kommandozeilen-Interpreters symbolisieren. Wenn wir eine Eingabe in einem Unix-Fenster vornehmen, so schreiben wir z.B.:
`unix> ls`
Dabei bezeichnet `unix>` die Eingabeaufforderung des Unix-Kommandointerpreters (*shell*). Wenn Tastenfolgen als Eingabe verlangt werden, schreiben wir:
$\boxed{\texttt{Taste1}} + \boxed{\texttt{Taste2}}$

Der vorliegende Kurs gibt mit dem folgenden Kapitel eine kurze Einführung in das Betriebssystem Unix. Ziel dieses Kapitels ist es, den Leserinnen und Lesern eine Hilfestellung für die Verwendung von MATLAB unter Unix zu liefern. Diejenigen, die sich hinreichend sicher im Umgang mit Unix fühlen, sollten sofort mit Kapitel 3 starten. Dort werden wesentliche Merkmale von MATLAB anhand einer *Beispielsitzung* aufgezeigt. Darüber hinaus werden hier Begriffe eingeführt, und es wird ein erster Eindruck über die zahlreichen Möglichkeiten von MATLAB vermittelt. Diese Beispielsitzung soll im Dialog stattfinden. Die einzelnen Grundlagen werden in den folgenden Kapiteln vertieft.

Wir empfehlen für den Umgang mit dem Skript folgende Vorgehensweise:

- Wenn Sie noch nicht mit Unix gearbeitet haben, nehmen Sie sich etwa 30 Minuten Zeit, die Unix-Beispielsitzung nachzuvollziehen und die vorgestellten Unix-Befehle auszuprobieren. Haben Sie schon Erfahrungen mit der Arbeit an der Unix-Workstation gesammelt, überspringen Sie dieses Kapitel.
- Die MATLAB-Beispielsitzung wird etwa 40 Minuten in Anspruch nehmen. Sie sollten diese Zeit investieren, um ein Gefühl für den Umgang mit MATLAB zu erwerben.
- Für die vertiefenden Kapitel (Schleifen, Skript-Dateien, Funktionen) sollten Sie jeweils zwi-

schen 20 und 40 Minuten am Rechner aufwenden. Allerdings lohnt es sich, diese Teile auch ohne einen Rechner schon einmal zu lesen und vorzubereiten.

- Die Einführung in die Graphikfunktionen von MATLAB kann man wieder sehr gut interaktiv erarbeiten. Sie werden etwa 30 Minuten dafür benötigen.

Insgesamt werden Sie also in etwa drei Stunden eine Einführung in Matlab auf einem Unix-Rechner erhalten. Wir wünschen Ihnen viel Spaß und Erfolg dabei.

Kapitel 2

Der Umgang mit Unix

In diesem Kapitel sollen die wichtigsten Unix-Befehle eingeführt werden. Es ist nicht geplant, eine umfassende Beschreibung des Betriebssystems Unix abzuliefern. Es soll lediglich die Grundlage für die Arbeit mit MATLAB geschaffen werden.

Wir gehen davon aus, daß die Leserin oder der Leser schon einmal an einem Computer gesessen hat und mit einer Windows-ähnlichen fensterorientierten Benutzungsoberfläche umgehen kann. Die im Praktikum verwendeten Workstations arbeiten unter dem Unix-Betriebssystem *Solaris* und auf dieses beziehen sich die folgenden Hinweise. Sie sollten aber im wesentlichen auch auf andere Unix-Derivate (z.B. *Irix*, *Linux*, etc.) übertragbar sein.

> **Ein wichtiger Hinweis am Anfang:** Eine Unix-Workstation wird **nicht** einfach ausgeschaltet, wenn man die Arbeit beendet hat. Sie wird auch **nicht** per *Reset*-Knopf neu gestartet, wenn ein Programm "hängt". Da auf einer Unix-Workstation mehrere Benutzer gleichzeitig arbeiten können, würde man mit so einer Maßnahme möglicherweise wertvolle Daten zerstören. Wie ein nicht mehr laufendes Programm geschlossen wird, wird im Abschnitt 2.5 diskutiert. Wenn gar nichts mehr geht, dann sollte man jemanden fragen, der sich mit Unix besser auskennt.

2.1 Einloggen

Da eine Unix-Workstation mehrere Benutzer gleichzeitig zuläßt, muß man sich zunächst anmelden (einloggen), wenn man mit der Arbeit beginnen möchte. Am Ende meldet man sich entsprechend ab (ausloggen).

Die unter Solaris laufende Workstation zeigt zunächst ein Anmeldefenster (login-Menü). Geben Sie Ihre Benutzerkennung und Ihr Passwort ein:

```
unix> benutzerkennung
unix> passwort
```

> Achten Sie darauf, daß im Menü *Options* die Sprache auf *En_US* eingestellt ist, weil MATLAB sonst fehlerhafte Ergebnisse zeigt.

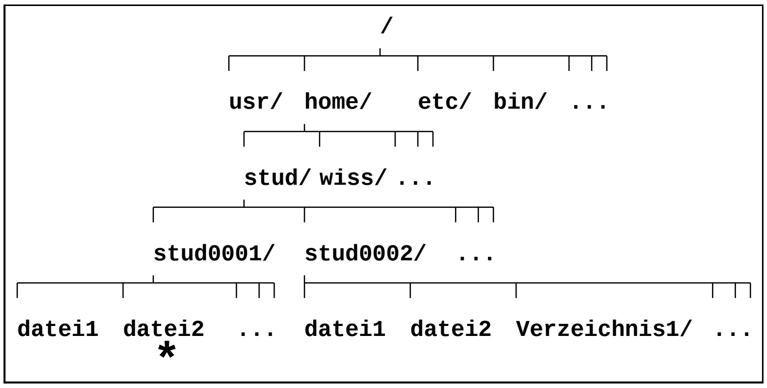

Abbildung 2.1: Unix-Verzeichnisbaum

2.2 Unix Terminal, das Arbeitsfenster

Um unter Unix arbeiten zu können benötigen Sie in den meisten Fällen ein Fenster, in dem Sie die Kommandos eingeben können. Auch MATLAB wird in einem solchen Fenster (Terminal) gestartet.

Sie öffnen ein Terminal (unter Solaris CDE), indem Sie mit der rechte Maustaste auf den Bildschirm-Hintergrund klicken und aus dem dort erscheinenden Menü den Schalter `Programs...` auswählen. Es öffnet sich ein neues Menü und dort finden Sie den Schalter `Terminal`, auf den Sie klicken, um ein Terminal zu öffnen. Klicken Sie nun mit der Maus in das Terminal, um es zu aktivieren.

2.3 Das Unix Dateisystem

Das Unix-Dateisystem ist als Baumstruktur organisiert. Es beginnt mit der Wurzel, dem Verzeichnis “/” (root). Dabei ist es unerheblich, ob einzelne Teile des Baums auf unterschiedliche Festplatten verteilt sind, der Anwender “sieht” immer einen einheitlichen Baum. Ein Ausschnitt eines typischen Verzeichnisbaums ist in der Abbildung 2.1 dargestellt.

> **Wichtig:** Alle Befehle und Namen unter Unix unterscheiden Groß- und Kleinschreibung . Man kann (fast) beliebig lange Dateinamen verwenden, allerdings sollten die Dateinamen keine Leerzeichen und keine Sonderzeichen enthalten.

Jedem Benutzer des Unix-Systems ist ein eigenes “Heimatverzeichnis” zugeordnet. Das Heimatverzeichnis wir auch mit dem Symbol “∼” bezeichnet. Dies ist der Startpunkt für eine eigene Verzeichnisstruktur (einen Unterbaum), die sich jeder anlegen kann. Wenn man sich einloggt, dann landet man automatisch an dieser Stelle des Unix-Verzeichnisbaums.

Die Position einer Datei im Verzeichnisbaum wird auch *Pfad* genannt. Eine Datei-Position kann mit einem *absoluten Pfad* angegeben werden, d.h. ausgehend vom root-Verzeichnis. In der Abbildung 2.1 kann man die Position der mit dem Sternchen gekennzeichneten Datei angeben mit `/home/stud/stud0001/datei2`. Befindet man sich schon im Verzeichnis `/home/stud`, dann könnte man die Position auch mit einem *relativen Pfad* (der auf die jetzige Position bezogenen ist) angeben: `stud0001/datei2`. Schließlich nehmen wir an, Ihr Heimatverzeichnis sei `/home/stud/stud0001`. Dann könnten Sie die Datei auch mit dem Pfad ∼`/datei2` bezeichnen, weil die Tilde, das Symbol für das Heimatverzeichnis ist. Die einzelnen Ebenen des Verzeichnisbaums werden im Pfad durch das Zeichen “/” (slash) getrennt.

Navigieren im Verzeichnisbaum

Hat man die grundsätzliche Struktur des Verzeichnissystems verstanden, ist das Navigieren recht einfach. Das aktuelle Verzeichnis (d.h. die aktuelle Position im Verzeichnisbaum) wird angezeigt mit dem Kommando `pwd` (print working directory):

```
unix> pwd
```

Der Befehl `cd` (change directory) wechselt in ein anderes Verzeichnis. Ohne weitere Angabe wird ins Heimatverzeichnis gewechselt. Wechseln Sie zunächst eine Ebene höher im Verzeichnisbaum:

```
unix> cd ..
```

Nun geben wir einen Pfadnamen mit an, z.B.:

```
unix> cd /home
```

Schließlich können Sie wieder in Ihr Heimatverzeichnis zurückgelangen mit:

```
cd
```

Wenn Sie jeweils zwischendurch den Befehl `pwd` eingeben, sehen Sie die Veränderung Ihrer Position.

Dateien und Verzeichnisse

Um einen eigenen Unterbaum anzulegen, muß man Verzeichnisse einrichten und löschen können. Die beiden Befehle `mkdir` (make directory) und `rmdir` (remove directory) erledigen dies:

```
unix> mkdir neues-verzeichnis
```

Sie können das Verzeichnis mit dem Befehl

```
unix> rmdir neues-verzeichnis
```

wieder löschen, tun Sie es aber jetzt noch nicht, wir wollen das Verzeichnis noch verwenden.

Datei kopieren `cp` (copy), löschen `rm` (remove), umbenennen/verschieben `mv` (move). Kopieren Sie die Datei `.profile` (achten Sie auf den vorangestellten Punkt!):

```
unix> cp .profile meine-datei
```

Kopieren Sie die Datei ins eben erstellte Verzeichnis:

```
unix> cp meine-datei neues-verzeichnis
```

Verschieben Sie die Datei (bzw. benennen Sie sie um):

```
unix> mv meine-datei neue-datei
```

Löschen Sie die Datei mit dem Befehl:

```
unix> rm neue-datei
```

Das Ziel der Befehle `cp` und `mv` kann jeweils ein Verzeichnis sein. Dann wird die Datei unter dem alten Namen in das angegebene Verzeichnis kopiert bzw. verschoben.

> **Achtung:** Falls die Ziel-Datei schon existiert, wird sie mit den obigen Befehlen von der ersten Datei überschrieben. Eine Datei, die unter Unix gelöscht oder überschrieben wird, ist **nie** wieder herstellbar!

Dateien können unter Unix (fast) beliebige Namen tragen. Allerdings werden oft bestimmte Vereinbarungen bei der Namensgebung getroffen. Die Namensteile werden oft mit einem “.” (Punkt) unterteilt. So würde eine typische Datei aussehen:

```
name.suf
```

Dabei bezeichnet `name` den eigentlichen Dateinamen, `suf` einen Suffix, der häufig Aufschluß über die Dateiart gibt (so werden im Kapitel über MATLAB-Dateien die Suffixe "m" und "mat" von besonderer Bedeutung sein). Dateien, die einen Punkt vorangestellt haben, sind versteckte Dateien.

Um einen Überblick über den Inhalt von Verzeichnissen und die Struktur von Unterbäumen zu erhalten, verwendet man die Befehle `ls` (list) und `du` (disk usage). `ls` dient dabei auch zur Auflistung einzelner Dateien oder von Dateien mit ähnlichen Namen. `ls -l` zeigt ausführliche Informationen wie Erstellungsdatum, Besitzer und Größe der Dateien. Probieren Sie

```
unix> ls -l
```

oder

```
unix> ls neues-verzeichnis
```

Wenn Sie die Verzeichnisstruktur und den verbrauchten Speicherplatz (in 512 kB Blöcken) sehen möchten, verwenden Sie:

```
unix> du
```

Man verwendet das Zeichen "*" als Platzhalter (wildcard), um alle Dateien/Verzeichnisse angezeigt zu bekommen, die gleiche Namensteile enthalten. Möchte man beispielsweise alle Dateien mit dem Suffix "doc" in einem Verzeichnis aufgelistet bekommen, so verwendet man den Befehl:

```
unix> ls *.doc
```

alle Dateien, die mit "neu" beginnen, bekommt man mit folgendem Befehl gelistet:

```
unix> ls neu*
```

Den Inhalt einer Datei kann man sich mit den Befehlen `more` (seitenweise), `head` (Anfang), `tail` (Ende) und `cat` (alles ausgeben), ansehen. Zum Beispiel können wir uns jetzt ansehen, was eigentlich in der Datei `.profile` steht:

```
unix> more .profile
```

Mit der Leertaste wird eine Seite weitergeblättert.

Achtung: Die obigen Befehle sollten nur für Text-Dateien verwendet werden.

Den Dateityp kann man mit dem Befehl `file` feststellen:

```
unix> file .profile
```

Text- oder Programmdateien können mit leistungsfähigen Programmen (Editoren) bearbeitet werden. Die gebräuchlichsten Editoren unter Unix sind `emacs`, `nedit` und `vi`, wobei letzterer kein eigenes Fenster öffnet. Eine Einführung in diese Programme kann an dieser Stelle nicht gegeben werden, wer jedoch das Windows Wordpad oder den MacOS SimpleText kennt, wird sich mit `nedit` schnell zurechtfinden.

```
unix> nedit neue-datei
```

2.4 Hilfe

Unix stellt schon seit jeher eine umfangreiche Online-Hilfe zur Verfügung. Sie ist jederzeit von der Kommandozeile mit dem Befehl `man` (manual) aufzurufen. Man muß den Unix-Befehlsnamen angeben, zu dem man Hilfe erwartet. Weiß man den Befehl nicht, sondern nur ein Stichwort, so kann man den Befehl `apropos` verwenden, um alle Unix-Befehle zu diesem Begriff aufgelistet zu bekommen.

```
unix> apropos editor
unix> man nedit
```

Solaris verfügt zusätzlich zu den standardmäßig vorhandenen Unix-Hilfe-Funktionen über interaktive Hilfeprogramme. Sie werden aus dem CDE-Menü am unteren Bildrand gestartet. Klicken Sie dazu mit der linken Maustaste auf das Hilfe Symbol im Menü. Wenn Sie auf den kleinen Pfeil über dem Hilfe Symbol klicken, erhalten sie weitere Hilfsangebote.

2.5 Programme und Prozesse

Programme startet man unter Unix, indem man ihren Namen auf der Kommandozeile angibt. Einige Programme sind auch per Mausklick aufrufbar, wenn Sie im CDE-Menü am unteren Bildrand als Symbole angezeigt werden. Startet man ein Programm, so werden im Unix-System ein oder mehrere *Prozesse* gestartet (daher werden diese Begriffe hier häufig synonym verwendet). Auf einem Unix-System laufen ständig viele Prozesse nebeneinander, die bestimmte Betriebssystemfunktionen erledigen. Diese Prozesse laufen *im Hintergrund*, d.h. als Benutzer merkt man in der Regel nichts davon. Startet man ein Programm von der Kommandozeile aus, so läuft es *im Vordergrund*, d.h. die Kommandozeile ist vom laufenden Programm belegt.

Programme starten

MATLAB und Octave sind Programme, die Benutzereingaben von der Kommandozeile aus erwarten. Sie starten also im Vordergrund von der Kommandozeile aus:

```
unix> matlab
```

Beenden Sie MATLAB mit dem Befehl `quit`.

Programme, die eigene Fenster öffnen, wie z.B. nedit, sollten im Hintergrund gestartet werden, damit die Kommandozeile frei bleibt für Eingaben. Man startet ein Programm im Hintergrund, indem man den Programmnamen angibt und danach das Zeichen "&" (ampersand). `nedit` würde man also mit dem folgenden Befehl starten:

```
unix> nedit &
```

Prozessmanagement

Laufende Prozesse kann man stoppen und wieder weiterlaufen lassen oder beenden. Diese Funktionen sind wichtig, wenn ein Programm hängt und man es beenden möchte.

Ein im Vordergrund gestartetes Programm, das hängt, oder aus anderen Gründen vorzeitig abgebrochen werden soll, kann mit der folgenden Tastenkombination beendet werden:

`Crtl` + `c`

Hat man ein Programm versehentlich im Vordergrund gestartet, das man gerne im Hintergrund ablaufen lassen möchte, so kann man es mit:

`Crtl` + `z`

anhalten und dann mit dem Befehl `bg` (background) in den Hintergrund schicken. Mit dem Befehl `fg` (foreground) holt man es wieder in den Vordergrund:

```
unix> nedit
Crtl + z
unix> bg
```

Sie beenden `nedit`, indem Sie aus dem Menü "File" den Menübefehl "Exit" wählen

Soll ein Programm, das im Hintergrund läuft, vorzeitig beendet werden, so kann der Befehl `kill` verwendet werden. Dazu muß jedoch zunächst die Prozessnummer *pid* (process identification) ermittelt werden. Eine Liste der laufenden Prozesse und deren pid erhält man mit dem Befehl `ps` (process status). Verschiedene Optionen erlauben die Auflistung der eigenen Prozesse, oder auch eine Liste aller Prozesse auf dem System. Ein Programm läßt sich nun beenden mit dem Befehl `kill`. Wir wollen ein kurzes Beispiel demonstrieren. Wenn Sie es nachvollziehen wollen, müessen Sie die kursiv gesetzte *pid* durch die in Ihrer Sitzung ermittelte Zahl ersetzen:

```
unix> nedit &
unix> ps
```

Sie erhalten jetzt eine Antwort vom System, die etwa so aussehen könnte:

```
  PID TTY       TIME CMD
18606 pts/1     0:01 nedit
18577 pts/1     0:00 tcsh
```

Jetzt können Sie `nedit` beenden, indem Sie eingeben:

`unix> kill` *18606*

Falls `nedit` damit noch nicht beendet sein sollte, so kann `kill` mit der folgenden Option aufgerufen werden, das sollte dann auch den hartnäckigsten Prozess überzeugen:

`unix> kill -9` *18606*

2.6 Zum Schluß: Ausloggen

Am Ende einer jeden Unix-Sitzung müssen Sie sich wieder abmelden. Zum Abmelden (ausloggen) gibt es unter Solaris zwei Möglichkeiten:

1. Sie können den *Exit*-Knopf benutzen, der sich im CDE-Menü am unteren Bildschirmrand befindet: klicken Sie mit der linken Maustaste darauf.
2. Sie können auch mit der rechten Maustaste auf den Bildschirm-Hintergrund klicken. Es öffnet sich ein Menü, in dem Sie auf `Log Out...` klicken.

Kapitel 3

Eine Beispielsitzung

3.1 Präliminarien

Im folgenden wird davon ausgegangen, daß Sie bereits wissen, wie man MATLAB unter Unix (Linux, Windows etc.) startet. Außerdem sollten Sie folgende Zeichen vorab schon kennen:

- `>>` Der *MATLAB-Prompt* (die Eingabeaufforderung).
- `%` Kommentar. Alles, was hinter diesem Zeichen in einer Befehlszeile steht, wird vom MATLAB-Interpreter ignoriert.
- `;` Schließt man eine Befehlszeile mit einem Semikolon ab, so wird das ausgewertete Ergebnis der MATLAB-Anweisung nicht angezeigt.
- `=` Der Zuweisungsoperator.

Setzen Sie sich nun an den Rechner, starten Sie MATLAB, und nehmen Sie *aktiv* an der folgenden Beispielsitzung teil, indem Sie die einzelnen MATLAB-Anweisungen selbst eingeben.

3.2 Eingabe von Matrizen, Vektoren und Skalaren

3.2.1 Explizite Eingabe

Eingabe eines Skalars

```
>> n = 8
```

Eingabe eines Zeilenvektors

```
>> x = [ 1 2 3 4 5 6 7 8]
```

Dieser Befehl läßt sich ebenso wie folgt abkürzen.

```
>> x = 1:n
```

Will man nur die ungeraden Ziffern zwischen 1 und 8 aufnehmen, so gibt man

```
>> x = 1:2:n
```

ein. Die Ziffer 2 in der obigen Anweisung fungiert als Schrittweite. Selbstverständlich darf man auch größere Schrittweiten, oder nichtganzzahlige, wie beispielsweise in

```
>> x = 1:0.5:n
```

verwenden. Darüber hinaus sind negative Schrittweiten zulässig und häufig sinnvoll: Nach der Anweisung

```
>> x=n:-1:1
```

enthält der Zeilenvektor `x` die ganzzahligen Werte aus dem Intervall $[1, n]$ in absteigender Reihenfolge.

Eingabe eines Spaltenvektors

```
>> y = [1
        2
        3
        4]
```

Äquivalent dazu ist die folgende Eingabe dieses Spaltenvektors

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

Zeichenketten gibt man wie folgt ein.

```
>> s = 'Hello World!'
```

Mehrere Zuweisungen innerhalb einer Zeile trennt man durch Kommata

```
>> u = 3, v = [1 2 3], w = [ 4; 5; 6; 7]
```

Will man das Anzeigen der ersten Zuweisung unterdrücken, so schließt man diese Zuweisung mit einem Semikolon ab.

```
>> u = 3; v = [1 2 3], w = [ 4; 5; 6; 7]
```

Folgerichtig werden mit der folgenden Anweisungszeile drei Zuweisungen ausgeführt, keine davon wird angezeigt.

```
>> u = 3; v = [1 2 3]; w = [ 4; 5; 6; 7];
```

Eingabe einer Matrix (man beachte die Wirkung des Semikolons am Zeilenende, s.o.)

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

Zeige den Inhalt der Matrix A an

```
>> A
```

Beachte, daß MATLAB zwischen Groß- und Kleinschreibung unterscheidet: Die Variable 'a' ist noch nicht definiert. Die folgende Eingabe führt daher zu einem Fehler.

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

Ändere den Eintrag a_{11} in der Matrix A. Beachte, daß die Indizierung bei 1 (nicht bei 0) beginnt!

```
>> A(1,1) = 5;
```

Ebenso kann man so auf Zeichen einer Zeichenkette zugreifen.

```
>> s(2) = 'a';
>> s
```

3.2.2 Laden von Matrizen

Angenommen, die einzelnen Komponenten der obigen Matrix A liegen in einer Datei namens `matrix.dat` in der folgenden Form vor.

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

Die einzelnen Zeilen von A sind zeilenweise in `matrix.dat` aufgelistet. Um den Inhalt aus dieser Datei in eine andere Matrixvariable, sagen wir D, einzulesen, verwendet man den Befehl `load`:

```
>> D = load('matrix.dat');
```

Mit der obigen Anweisung wird eine Variable D der Dimension 3×3 erzeugt, die anschließend mit dem Inhalt der Datei `matrix.dat` gefüllt wird. Die Werte aller Komponenten von A und D stimmen somit überein. Gibt man lediglich

```
>> load('matrix.dat')
```

ein, so wird eine Variable namens `matrix` erzeugt, die die Komponenten aus der Datei `matrix.dat` enthält. Weitere Optionen des Befehls `load` bekommt man durch dessen Dokumentation, die man sich mit

```
>> help load
```

anzeigen läßt.

3.2.3 Der MATLAB-Editor

Wie erzeugt und editiert man eine solche Datei wie `matrix.dat`, deren Inhalte man ggf. wie oben anschließend in eine Variable unter Verwendung von `load` einlesen möchte? MATLAB stellt einen Editor zur Verfügung, den man mit dem Befehl

```
>> edit
```

aufruft. Allerdings ist es hier nicht unser Ziel, diesen Editor ausführlich zu beschreiben. Selbstverständlich darf man genauso einen anderen Editor, etwa `emacs`, `nedit` oder `vi`, für solche Zwecke verwenden.

3.2.4 Eingebaute MATLAB-Variablen

In MATLAB bezeichnen die Variablen i und j die imaginäre Einheit. Diese Variablen werden beim Starten der MATLAB-Sitzung automatisch zur Verfügung gestellt.

```
>> i
```

Man wird üblicherweise höchstens eine der beiden Variablen i oder j verwenden. Wir verwenden i, und überschreiben nun j.

```
>> j = 3
```

Man darf natürlich auch eine andere Variable, sagen wir k, als Bezeichner der imaginären Einheit verwenden. Die Anweisung

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

erzeugt die Variable k und weist ihr den Wert $\sqrt{-1}$ zu. Eine weitere eingebaute MATLAB-Variable ist die Kreiszahl π.

```
>> pi
```

3.2.5 Eingebaute MATLAB-Funktionen zur Matrixgenerierung

Die Identität wird wie folgt erzeugt. Das Argument legt die Dimension fest.

```
>> E = eye(3)
```

Eine (3×3)-Matrix M, deren Komponenten Zufallszahlen aus dem Intervall $[0, 1]$ enthalten.

```
>> M = rand(3)
```

Eine Nullmatrix Z der Dimension 3×2

```
>> Z = zeros(3,2);
```

So setzt man eine Matrix aus Blöcken (Untermatrizen) zusammen.

```
>> C = [A, M ; Z, E, rand(3,1); [6 5 4 3 2 1]]
```

3.3 Matrixoperationen

Welche Operationen gibt es überhaupt bzw. wie bekommt man Hilfe?

```
>> help
```

Wähle eine Option aus der angezeigten Liste, z.B. `matfun` (Matrizenfunktionen)

```
>> help matfun
```

Hier findet man eine ausgewählte Liste mit weiteren Befehlen, u.a. `det` (Determinante einer Matrix). Das folgende Kommando zeigt die Beschreibung der Funktion `det` sowie deren Syntax an.

```
>> help det
```

Und so bekommt man *Online-Hilfe* unter Verwendung eines geeigneten Browsers (z.B. Netscape)

```
>> doc
```

Nun berechnen wir die Determinante der quadratischen Matrix A, die oben definiert wurde.

```
>> det(A)
```

MATLAB rechnet übrigens mit *doppelter Genauigkeit*. Das Format der Ausgabe verändert man mit dem Befehl `format`.

```
>> help format
```

listet die verfügbaren Ausgabeformate auf. Entscheidet man sich beispielsweise für ein Ausgabeformat mit Festpunktdarstellung und 14 Ziffern hinter dem Komma, so wird dies durch die Anweisung

```
>> format long
```

realisiert. Dieses Ausgabeformat ist wirksam bis zu dessen nächster Änderung. Als Standard ist Festpunktformat mit 4 Ziffern hinter dem Komma voreingestellt, auf das man wieder zurückkommt mit der folgenden Anweisung.

```
>> format short
```

Nun weitere elementare Matrix-Funktionen:

```
>> A^2
```

$A \cdot A$, das Quadrat der Matrix A

```
>> A+E
```

die Summe der Matrizen A und E

```
>> A*M
```

das Produkt von A und M.

Diese Operationen können ebenso komponentenweise ausgeführt werden. Dazu hat man dem entsprechenden Operator einen Punkt `.` voranzustellen:

```
>> A.^2
```

generiert die Matrix mit den Komponenten $a_{ij} \cdot a_{ij}$

```
>> A.*E
```

generiert die Matrix $(a_{ij} * e_{ij})_{ij}$.

Operationen zwischen Skalaren und Matrizen.

```
>> n*A
```

jede Komponente der Matrix A wird mit dem Skalar n multipliziert

```
>> n+A
```

zu jeder Komponente der Matrix A wird n addiert

Eine reelle Zufallsmatrix der Dimension 3×3.

```
>> A = rand(3)
```

Die Transponierte von A bildet man wie folgt.

```
>> A'
```

Eine komplexe Zufallsmatrix.

```
>> A = rand(3) + i*rand(3)
```

Schauen Sie sich nun das Ergebnis der folgenden Zuweisung an.

```
>> B = A'
```

Der Wert von B entsteht durch Bilden der konjugiert komplexen Werte der Komponenten von A und anschliessender Transposition.

Berechne die Eigenwerte von A

```
>> eig(A)
```

Berechne die Koeffizienten des charakteristischen Polynoms von A

```
>> poly(A)
```

Berechne die LR-Zerlegung von A. Rückgabewert sind zwei Matrizen, die hier mit L und R

bezeichnet werden.

```
>> [L R] = lu(A);
```

Beachte, daß L nicht notwendigerweise eine untere Dreiecksmatrix ist. Details dazu findet man durch Aufruf von ...

```
>> help lu
```

... nachdem man verstehen wird, was das folgende Kommando bewirkt.

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

Man bekommt nämlich eine untere Dreiecksmatrix L, eine obere Dreiecksmatrix U, und eine Permutationsmatrix P, so daß $PA = LU$ gilt.

So kann man die Dimension von A abfragen.

```
>> size(A)
```

... oder direkt verwenden als Eingabewert eingebauter MATLAB-Funktionen

```
>> Z = zeros(size(A))
```

3.4 Untermatrizen und die Colon-Notation

Wir lassen uns zunächst die Matrix A anzeigen.

```
>> A
```

Mit der Anweisung

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

expandiert man die Untermatrix von A, die aus den Zeilen 1 und 2, hier einfach abgekürzt mit `1:2`, sowie den Spalten 2 und 3, abgekürzt mit `2:3`, zusammengesetzt ist. Zum Expandieren von einer kompletten Zeile bzw. Spalte geht man folgendermaßen vor. Die zweite Zeile von A läßt man sich so anzeigen:

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

Das erste Argument, die Ziffer 2, legt den Zeilenindex fest, während der Doppelpunkt `:` (engl. colon) als Argument für den Spaltenindex bewirkt, daß die gesamte Zeile expandiert wird. Analog zeigt man die erste Spalte von A wie folgt an.

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

Die gesamte Matrix expandiert man folgerichtig mit

```
>> A(:,:)
```

was das gleiche bewirkt wie die bereits bekannte Anweisung

```
>> A
```

Mit Hilfe dieser Notation kann man bestimmte Untermatrizen abkürzend darstellen. Dies kann man beispielsweise zum Überschreiben von Untermatrizen verwenden. Beispielsweise wird mit der Anweisung

```
>> A(1:2,1:2) = eye(2)
```

die Untermatrix von A, die aus den Komponenten der ersten und zweiten Zeile sowie der ersten und zweiten Spalte von A besteht, durch die Identität der Dimension 2×2 ersetzt. Beachte: A hat sich durch diesen Aufruf tatsächlich geändert!

Umgekehrt kann man Untermatrizen extrahieren und an andere Variablen zuweisen. Ein mögliches Beispiel ist gegeben durch den Aufruf

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

durch den die Variable S erzeugt wird und mit derjenigen Untermatrix von A gefüllt wird, deren Zeilenindex zwischen 1 und 3 liegt und deren Spaltenindex 1 oder 2 ist.

So erzeugt man einen Spaltenvektor aus A, der durch Aneinanderreihen der einzelnen Spalten von A entsteht.

```
>> a = A(:)
```

Damit dürfte klar sein, auf welchen Eintrag von A man mit

```
>> A(4)
```

zugreift, nämlich auf A_{12}.

Umgekehrt kann man einen Spaltenvektor verwenden, um die Einträge einer bereits dimensionierten Matrix zu überschreiben:

```
>> B = zeros(size(A))
>> B(:) = a
```

Nun stimmen die Einträge der beiden Matrizen A und B überein.

Mit Zeichenketten läßt sich ähnlich wie mit Matrizen verfahren. Die Zeichenkette `s`, die den Text `Hello World!` enthält, wird von MATLAB als Zeilenvektor behandelt.

```
>> s(1,:)
```

die Zeichenkette `Hello World!`; `s(1,:)` ist somit gleich `s`.

```
>> s(:,1)
```

das erste Zeichen, `H`; `s(:,1)` ist in diesem Fall somit gleich `s(1)`;

```
>> s(:)
```

die Zeichen in einem Spaltenvektor

```
>> s(8:12) = 'elt !'
```

`Hello Welt !`; `s(8:12)` ist hier gleichbedeutend mit `s(1,8:12)`. Nun korrigieren wir den zweiten Buchstaben auch noch mit

```
>> s(2) = 'a'
```

`Hallo Welt !`, und entfernen das Leerzeichen vor dem Ausrufezeichen mit

```
>> s = [s(1:10),s(12)]
```

`Hallo Welt!`

3.5 Wiederverwendung einzelner Anweisungen

Im allgemeinen ist man ja tippfaul. MATLAB unterstützt diese Art von Faulheit. Bereits eingegebene Befehle kann man mit den Cursortasten

`UP` (Pfeil nach oben) und
`DOWN` (Pfeil nach unten)

in die Kommandozeile holen und anschliessend editieren. Geben Sie `UP` doch einfach mal solange ein, bis Text der Anweisung

```
>> B(:) = a
```

erscheint. Diese Anweisung kann man nun mit Hilfe der Cursortasten `LEFT` (Pfeil nach links) und `RIGHT` (Pfeil nach rechts) sowie der anderen Tasten, die ständig verwendet werden, editieren:

```
>> B(1,2)
```

Hat man den Editiervorgang abgeschlossen, so kann man die entsprechende Anweisung durch Eingabe von `RETURN` ausführen lassen.

Im speziellen ist man ja noch tippfauler als tippfaul. MATLAB unterstützt Tippfaulheit weiterhin dadurch, daß man nach Eingabe eines oder mehrerer Anfangszeichen eines vorherigen Befehls die Cursortasten `UP` und `DOWN` verwenden kann, um Anweisungszeilen, die mit dieser Zeichenkette beginnen, in die Kommandozeile (editierbar) zu holen. Durch diese Selektionsmöglichkeit lassen sich in Spezialfällen Eingaben von `UP` sparen. Gibt man etwa die Zeichenfolge

```
>> si
```

ein, so wird diese Zeichenkette nach Eingabe von `UP` durch die Zeichenkette

```
>> size(A)
```

von oben vervollständigt. Diese Anweisung kann man nun im Bedarfsfall editieren, das entsprechende Kommando wird nach Eingabe von `RETURN` ausgeführt.

3.6 Die Behandlung von Variablen

Zeige die aktuellen Variablen an.

```
>> who
```

Beachte, daß es eine Variable `ans` in der Liste gibt. Diese Variable hat fuer MATLAB eine besondere Bedeutung. Sie enthält den Rückgabewert der letzten Berechnung, in der keine Zuweisung an eine andere Variable stattgefunden hat. Noch mal die Liste der Variablen, aber etwas detaillierter:

```
>> whos
```

Entferne die Variable `M` aus der Liste

```
>> clear M
```

Die Variable `M` ist tatsächlich rausgeflogen, wie man gleich sehen wird.

```
>> whos
```

3.7 Abspeichern und Laden, Beenden und Starten einer Sitzung

Mit dem folgenden Befehl werden alle aktuellen Variablen und ihre momentanen Werte in einer Datei namens `session1.mat` abgespeichert.

```
>> save session1
```

Lösche alle aktuellen Variablen

```
>> clear
```

Beenden der Sitzung

```
>> quit
```

Beachte, daß die Anzahl der *flops* (floating point operations), die während der Sitzung verbraucht wurden, angezeigt wird. Das folgende Promptzeichen zeigt an, daß wir es wieder mit dem Betriebssystem UNIX zu tun haben.

```
unix>
```

Und so fahren wir MATLAB wieder hoch. Das sollten Sie bereits wissen (s.o.).

```
unix> matlab
...
>>
```

Lade nun die abgespeicherten Variablen und deren Inhalte von der vorigen Sitzung.

```
>> load session1
```

3.8 Zählen von Operationen und Zeitmessungen

Mit dem Kommando ...

```
>> flops(0)
```

...initialisiert man den Zählmechanismus, und das Kommando

```
>> flops
```

liefert die Anzahl der flops seit der letzten Initialisierung.

> **ACHTUNG:** Man sollte nie die Anweisung `flops = 0` verwenden, denn in diesem Fall wird eine *Variable* `flops` erzeugt, die die Information, die man sonst duch den Aufruf an die *Funktion* `flops` bekommt, überschattet.

Beispiel: Wie teuer ist eigentlich eine Matrix-Vektor-Multiplikation? Erzeugen wir zunächst einen Spaltenvektor x der Länge n und eine Matrix A der Dimension $n \times n$.

```
>> n=10; x = rand(n,1)
>> A = rand(n)
```

Nun: Ausführen der Multiplikation und Anzeige der Anzahl der benötigten flops.

```
>> flops(0); A*x; flops
```

Eigentlich hätten es exakt $2n^2 - n$ flops sein müssen, MATLAB liefert allerdings nur den führenden Term $2n^2$ als Ergebnis. Dies liegt daran, dass MATLAB nicht tatsächlich mitzählt (das wäre viel zu aufwendig!), sondern MATLAB ermittelt stattdessen den Aufwand *größenordnungsmäßig* anhand der Dimensionen der beteiligten Objekte.

Ein Vergleich: Die beiden folgenden Auswertungen liefern das gleiche Resultat, aber mit unterschiedlicher Anzahl an flops. Welche Schreibweise ist effizienter?

```
>> flops(0); (x*x')*x; flops
>> flops(0); x*(x'*x); flops
```

Vergleiche ebenso die benötigte Zeit.

```
>> tic; (x*x')*x; toc;
>> tic; x*(x'*x); toc;
```

Bemerkung: Die obigen Informationen hängen von der aktuellen Auslastung der Maschine ab.

Kapitel 4

Bedingte Verzweigungen und Schleifen

In diesem Kapitel werden wichtige Kontrollmechanismen zur Steuerung des Ablaufes von MATLAB-Anweisungen vorgestellt. Von diesen Mechanismen wird innerhalb von *Funktionen*, die im nächsten Kapitel behandelt werden, massiv gebraucht gemacht. Wir geben die einzelnen Kontrollstrukturen mit Hilfe *elementarer* Beispiele an, anhand deren Schema die einzelnen Konstrukte hinreichend deutlich werden sollten.

4.1 If-Abfragen

If-Abfragen führen zu bedingten Verzweigungen innerhalb eines Programmablaufes. Die Verzweigung erfolgt in Abhängigkeit eines bestimmten Testresultats. Diese If-Abfragen tauchen typischerweise in Situationen auf, in denen fehlerhafte Eingaben behandelt werden müssen, um größerem Unheil vorzubeugen bzw. die Korrektheit des zugrundeliegenden Algorithmus zu garantieren. Ein klassisches Beispiel für den ersten Fall ist die Operation der Division a/b einer Zahl a dividiert duch b. Man will natürlich nur durch b dividieren, falls b nicht die Null ist. Anderenfalls verzichten wir auf die Division. Die entsprechende Folge von MATLAB-Anweisungen sieht für diesen Fall folgendes vor:

```
>> if b ~= 0
     c = a/b;
   end
```

Dabei haben wir stillschweigend angenommen, daß die Variablen `a` und `b` dem MATLAB-Interpreter bereits bekannt sind. Der Test in dieser If-Abfrage lautet `b ~= 0`, es wird geprüft, ob b ungleich Null ist. Fällt dieser Test positiv aus (d.h. b ist nicht Null), so wird in die Zeile `c = a/b;` verzweigt. Jede If-Abfrage endet generell mit der Anweisung `end`!

Mit dem obigen Beispiel haben wir bereits den *Relationsoperator* `~=` (ungleich) kennengelernt. Dieser liefert im obigen Fall das Resultat `1`, falls $b \neq 0$, ansonsten das Resultat `0`. Man überzeuge sich davon, indem man einfach die Zeile

```
>> b ~= 0
```

oder allgemeiner

```
>> b ~= a
```

für mehrere verschiedene Werte der Variablen `a` und `b` eingibt. Mit anderen Worten: Der Test `b ~= 0` fällt positiv aus, falls die Operation `b ~= 0` den Wert `1` liefert; der Test `b ~= 0` fällt negativ aus, falls die Operation `b ~= 0` den Wert `0` liefert.

Weitere Relationsoperatoren in MATLAB sind die folgenden.

$<$	kleiner
$>$	größer
$<=$	kleiner oder gleich
$>=$	größer oder gleich
$==$	gleich
$\tilde{}=$	ungleich

Ein weiteres Beispiel, das an das obige anknüpft: Wir wollen der Variablen `c` den Wert der Division `a/b` zuweisen, falls `b` nicht Null ist, anderenfalls wollen wir `c` den Wert Null zuweisen. Mit MATLAB sieht die entsprechende Anweisungsfolge wie folgt aus.

```
>> if b ~= 0
     c = a/b;
   else
     c = 0;
   end
```

Ein weiteres Beispiel: Wir wollen die *Vorzeichenfunktion* implementieren: Wir weisen der Variablen `s` den Wert `-1` zu, falls die Variable `x` einen negativen Wert besitzt, `s` bekommt den Wert `1`, falls der Wert von `x` positiv ist; anderenfalls (d.h. falls `x` den Wert Null besitzt), bekommt `s` den Wert Null zugewiesen.

```
>> if x < 0
     s = -1;
   elseif x > 0
     s = 1;
   else
     s = 0;
   end
```

4.2 For-Schleifen

Wollen wir eine bestimmte Folge von Anweisungen n-mal hintereinander ausführen, so bietet sich die Verwendung einer sogenannten *For-Schleife* an. Nehmen wir an, wir wollen *elf* ganzzahlige Zufallszahlen aus dem Intervall $[0, 2]$ erzeugen (um etwa am Totowettbewerb teilzunehmen). Eine mögliche Implementation mit einer For-Schleife sieht so aus.

Beipiel 1: Elf Totozahlen zufällig generieren. Der Vektor x enthält nach Ablauf der For-Schleife die elf Zufallszahlen.

```
>> x = [];
>> for k=1:11
```

```
    x = [x; round(2*rand)];
  end
>> x
```

Beipiel 2: Wir berechnen die Zahl $12! = 1 \cdot 2 \cdot \ldots \cdot 12$. Die Hilfsvariable `fac` wird das Ergebnis enthalten.

```
>> n = 12;
>> fac = 1;
>> for i=2:n
    fac = i*fac;
  end
```

Schauen wir uns das Ergebnis an.

```
>> fac
```

Nun brauchen wir die Variable `fac` nicht mehr.

```
>> clear fac;
```

Beipiel 3: Der Satz von Caley-Hamilton. Wir nehmen eine spezielle Matrix her.

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

kennen wir schon von oben

```
>> n = length(p);
```

liefert die Länge des Vektors p

```
>> B = zeros(size(A))
```

erzeugt eine Nullmatrix B der Dimension 3×3.

Mit dem Satz von Caley-Hamilton erwarten wir, daß die Matrix B nach Verlassen der folgenden for-Schleife den Wert 'Null' (d.h. Nullmatrix) enthält.

```
>> for i=0:n-1
    B = B + A^i * p(n-i);
  end
```

Schaun'mer mal.

```
>> B
```

Bemerkung: MATLAB stellt eine effizientere Auswertung von Polynomen mit Matrixargumenten durch die Funktion `polyvalm` zur Verfügung.

```
>> polyvalm(p,A)
```

4.3 While-Schleifen

Wir haben im vorigen Unterabschnitt For-Schleifen kennengelernt. *While-Schleifen* sind so ähnlich: Während man bei einer For-Schleife bereits vor deren Ablauf weiß, wie oft man den Schleifenkörper durchlaufen möchte, koppelt man dies bei While-Schleifen an eine Bedingung: Vor jedem Durchlaufen des Schleifenkörpers findet ein Test statt. Fällt der Test positiv aus, so wird der Schleifenkörper durchlaufen. Anderenfalls wird die While-Schleife verlassen (oder erst gar nicht betreten), und die Anweisungen nach der While-Schleife werden ausgeführt.

Ein Negativbeispiel: Wir produzieren absichtlich eine *Endlosschleife* (wenn Sie noch nicht wissen, was eine Endlosschleife ist, dann werden Sie es gleich wissen) mit der folgenden Anweisungssequenz.

```
>> n = 1;
>> while n > 0
     n = n + 1;
   end
```

Wie kommen wir dort bloß wieder raus? Mit der Tastenkombination

`CTRL`+`C`

kann man die aktuelle Berechnung abbrechen, und bekommt den vertrauten MATLAB-Prompt zurück, d.h. man kann anschließend die begonnene MATLAB-Sitzung fortführen.

Ein weniger destruktives Beispiel: Wir wollen zu einer positiven Zahl x die kleinste ganze Zahl n mit der Eigenschaft $n \leq x < n+1$ berechnen. Zur Information: Üblicherweise kürzt man diese Zahl mit $\lfloor x \rfloor$ ab, und das Symbol $\lfloor \cdot \rfloor$ wird als *Gaußklammer* bezeichnet.

```
>> n=0;
>> while n+1 <= x
     n = n+1;
   end
>> n
```

Selbstverständlich ist $\lfloor x \rfloor$ auch für negative reelle Zahlen definiert. Diese mögliche Erweiterung der obigen Implementation von $\lfloor x \rfloor$ sieht beispielsweise so aus:

```
>> n=0;
>> if x < 0
      while n > x
        n = n-1;
      end
   elseif x > 0
     while n+1 <= x
       n = n+1;
     end
   end
>> n
```

Sie werden es zurecht als umständlich empfunden haben, die einzelnen Anweisungen, die mit If-Anweisungen sowie For- und While-Schleifen verbunden sind, Zeile für Zeile an den Interpreter zu übergeben. Wie bereits eingangs erwähnt, tauchen solche Kontrollstrukturen üblicherweise innerhalb von Funktionen auf. Wir werden MATLAB-Funktionen im folgenden Kapitel kennenlernen, und das Material aus diesem Kapitel diente lediglich vorbereitend zu Übungszwecken. Nach Studium des folgenden Kapitels werden Sie in der Lage sein, aus jedem der obigen Beispiele eine MATLAB-Funktion zu schreiben; etwa eine Funktion, die zu einer gegebenen natürlichen Zahl n den Wert von $n!$ berechnet, oder n ganzzahlige Zufallszahlen aus $[0, 2]$ erzeugt usw. Behalten Sie dies als kleine Übung im Hinterkopf.

4.4 Abbruch von Schleifen

Für den Fall, daß man den Ablauf einer For- oder einer While-Schleife aus irgendwelchen Gründen terminieren lassen möchte, steht in MATLAB der Befehl `break` zur Verfügung. Diese Art von Abbruch kann sinnvoll sein in Anwendungen, die mit Suchen oder Zählen in Verbindung stehen. Hierzu jeweils ein Beispiel.

Beipiel 4: Ein Suchbeispiel mit einer For-Schleife. Man möchte zu einem Zeilenvektor x, dessen Komponenten Zufallszahlen aus dem Intervall $[0, 1]$ sind, dessen kleinsten Index i ermitteln, für den gilt $x(i) < 0.5$. Falls es keinen solchen Index gibt, so soll i den Wert -1 besitzen.

```
>> n = 10;
>> x = rand(1,n);
>> i = -1;
>> for k=1:n
     if x(k) < 0.5
       i = k;
       break;
     end
   end
```

Beipiel 5: Ein Zählbeispiel mit einer While-Schleife. Man möchte solange Zufallszahlen aus [0,1] erzeugen, bis eine der Zufallszahlen größer als 0.99 ist. Die Anzahl n der *Versuche* sollen mitgezählt und anschließend anzeigt werden; Falls die Anzahl der Versuche 10 übersteigt, so möchte man aufhören.

```
>> x = rand;
>> n = 1;
>> while x <= 0.99
     if n > 10
       n = - 1;
       break;
     else
       x = rand;
       n = n+1;
     end;
   end;
>> n
```

Kapitel 5

MATLAB-Dateien

Dateien mit dem Suffix `.m` spielen eine wichtige Rolle für MATLAB. Solche Dateien werden als *M-files* bezeichnet. Im folgenden werden wir zwei verschiedene Klassen von M-files vorstellen.

5.1 Skriptdateien

Eine Skriptdatei enthält eine Folge von MATLAB-Anweisungen. Nehmen wir an, wir haben eine Datei namens `dreizeiler.m` erzeugt, die die folgenden drei Zeilen enthält.

```
A = [1 2; 3 4];
d = det(A)
B = A'*A
```

Dann werden nach Aufruf von

```
>> dreizeiler
```

die drei MATLAB-Anweisungen aus `dreizeiler.m` Zeile für Zeile an den MATLAB-Interpreter übergeben und vom Interpreter ausgewertet. Es macht somit keinen (semantischen) Unterschied, ob man diese drei Zeilen wie oben durch Laden von `dreizeiler.m` ausführen läßt, oder ob man die einzelnen Anweisungen während der Sitzung nacheinander eingibt:

```
>> A = [1 2; 3 4];
>> d = det(A)
>> B = A'*A
```

Damit sollte die Bedeutung von Skriptdateien hinreichend klar sein: In Skriptdateien speichert man typischerweise eine Abfolge von MATLAB-Anweisungen ab, die man während einer MATLAB-Sitzung häufiger verwendet, und die man aus Gründen der Zeitersparnis nicht wiederholt eingeben möchte.

5.2 Funktionsdateien

Mit Hilfe von Funktionsdateien kann man neue, eigene Funktionen definieren, die MATLAB während einer Sitzung verwenden kann. Ebenso kann man mit eigenen Funktionen bestehende Implementierungen von MATLAB-Funktionen überlagern.

Angenommen, wir wollen eine Funktion definieren, die uns zu einer gegebenen Seitenlänge q den Flächeninhalt $q*q$ eines Quadrats dieser Seitenlänge ausgibt. Die folgende Funktion `quadratflaeche`, abgespeichert in der Datei `quadratflaeche.m`, leistet das Gewünschte:

```
function I = quadratflaeche(q)

% QUADRATFLAECHE. Flaeche eines Quadrats.
%   QUADRATFLAECHE(Q) ist die Flaeche eines Quadrats mit Seitenlaenge Q.

I = q*q;

% Ende der Funktion quadratflaeche
```

Anhand der ersten Zeile in `quadratflaeche.m` erkennt der MATLAB-Interpreter, daß es sich um eine Funktion namens `quadratflaeche` handelt. Ohne das führende Schlüsselwort `function` würde der Interpreter die Datei `quadratflaeche.m` wie eine Skriptdatei behandeln und dessen Inhalte zeilenweise als einzelne MATLAB-Anweisungen abarbeiten.

Der Parameter `I` wird als *Ausgabeparameter* der Funktion `quadratflaeche` bezeichnet, während `q` *Eingabeparameter* heißt. Mit den Ein- und Ausgabeparametern und dem Funktionsnamen ist die Schnittstelle zur MATLAB-Sitzung festgelegt. Damit ist klar, wie die neue Funktion während der MATLAB-Sitzung aufzurufen ist: Mit der Anweisung

```
>> flaeche = quadratflaeche(2.0);
```

wird die Funktion `quadratflaeche` ausgeführt, und die Variable `flaeche` bekommt nach der Auswertung den *Rückgabewert* 4 von `quadratflaeche` zugewiesen.

Die selbstdefinierte Funktion `quadratflaeche` wird im Prinzip genauso wie eine eingebaute MATLAB-Funktion behandelt. Beispielsweise läßt sich mit

```
>> help quadratflaeche
```

sogar die Hilfefunktion auf `quadratflaeche` anwenden. Ausgegeben werden in diesem Fall die in der Datei `quadratflaeche.m` auskommentierten Zeilen zwei und drei. Dies sollte man bei der Beschreibung der neuen Funktion nutzen. Zu einer sinnvollen Beschreibung gehört neben der *Semantik* ebenso die *Syntax*.

Damit der beschriebene Mechanismus funktioniert, muß sich die Datei `quadratflaeche.m` im *aktuellen Arbeitsverzeichnis* befinden. Ansonsten kann das System die neue Funktion nicht finden. Zur Erinnerung: Das aktuelle Arbeitsverzeichnis kann man sich während einer Sitzung mit dem Befehl

```
>> pwd
```

anzeigen lassen. In ein anderes Verzeichnis, etwa `/home/users/myaccount`, kann man während einer MATLAB-Sitzung mit der Anweisung

```
>> cd /home/users/myaccount
```

wechseln.

Man kann neue Funktionen selbstverständlich genauso wie eingebaute MATLAB-Funktionen mit mehreren Eingabeparametern und mehreren Ausgabeparametern versehen. Um ein Beispiel zu geben, nehmen wir an, daß wir die obige Funktion `quadratflaeche` folgendermaßen erweitern wollen: Wir wollen zu einer gegebenen Kantenlänge q und einer Dimension d das Volumen und die Länge der Diagonalen eines d-dimensionalen Würfels ausgeben lassen. Wir implementieren dies durch die Funktion `wuerfeldaten`, die in der folgenden Datei `wuerfeldaten.m` abgespeichert ist:

```
function [vol, diag] = wuerfeldaten(q,d)

% WUERFELDATEN. Volumen und Laenge der Diagonalen eines Wuerfels.
%   WUERFELDATEN(Q,D) liefert das Volumen VOL und die Laenge DIAG der
%   Diagonalen eines D-dimensionalen Wuerfels mit Kantenlaenge Q.

vol = q^d;
diag = q*sqrt(d);

% Ende der Funktion wuerfeldaten
```

Anhand der Deklaration der neuen Funktion `wuerfeldaten` ist ersichtlich, daß `wuerfeldaten` als Ausgabe einen Zeilenvektor bestehend aus zwei Komponenten liefert, und daß `wuerfeldaten` zwei Eingabewerte erwartet. Somit ruft man die neue Funktion beispielsweise wie folgt auf.

```
>> [v,dd] = wuerfeldaten(2.0,3);
```

Mit der obigen Anweisung werden die beiden Variablen `v` und `dd` erzeugt (sofern diese nicht schon bereits im Verlauf der Sitzung erzeugt worden sind), und ihre Inhalte werden mit den Werten `v = 8` bzw. dd = 3.4641 gefüllt.

Weiterhin ist der Aufruf

```
>> w = wuerfeldaten(2.0,3);
```

erlaubt und sinnvoll. In diesem Fall bekommt die Variable `w` als Wert das Volumen `w = 8` zugewiesen, während die in der Funktion `wuerfeldaten` berechnete Länge der Würfeldiagonalen verloren geht.

Ruft man schließlich die Funktion `wuerfeldaten` durch die Anweisung

```
>> wuerfeldaten(2.0,3)
```

auf, so wird der Wert der Variablen `ans`, die bekanntermaßen in MATLAB eine besondere Rolle spielt, mit dem Würfelvolumen `ans = 8` überschrieben.

Dagegen führt der folgende Aufruf zu einem Fehler.

```
>> [x, y, z] = wuerfeldaten(2.0,3)
```

Hier übersteigt nämlich die Anzahl der Ausgabeargumente `x, y` und `z` die Anzahl der Ausgabeparameter von `wuerfeldaten`.

Nun zur Anzahl der Eingabeargumente: Diese darf ebenso die Anzahl der Eingabeparameter nicht überschreiten, sehr wohl aber unterschreiten. Für das spezielle Beispiel der Funktion `wuerfeldaten` führt ein Aufruf der Form

```
>> [vv, dd] = wuerfeldaten(2.0)
```

allerdings zu einem Fehler, wovon man sich an dieser Stelle durch Ausprobieren überzeugen sollte. Hier wird der eingegebene Wert `2.0` dem Eingabeparameter `q` zugewiesen, während der Wert von `d` undefiniert ist. Somit ergibt die Zuweisung `vol = q^d;` in `wuerfeldaten` keinen Sinn, und dies erklärt die Fehlermeldung. Man kann die Funktion `wuerfeldaten` jedoch so modifizieren, daß Anweisungen der obigen Form sinnvoll und ohne Fehler ausgeführt werden. Die folgende Erweiterung der Funktion `wuerfeldaten` durch eine neue Funktion `wuerfeldaten2` wird die entsprechenden Berechnungen für $d = 2$ ausführen, falls die Dimension durch Eingaben der obigen Form nicht spezifiziert ist. Dabei wird innerhalb von `wuerfeldaten2` die Anzahl der Eingabeargumente mit der Variablen `nargin` abgefragt.

```
function [vol, diag] = wuerfeldaten2(q,d)

% WUERFELDATEN2. Volumen und Laenge der Diagonalen eines Wuerfels.
%   WUERFELDATEN2(Q,D) liefert das Volumen VOL und die Laenge DIAG der
%   Diagonalen eines D-dimensionalen Wuerfels mit Kantenlaenge Q.
%   WUERFELDATEN2(Q) liefert die Flaeche VOL und die Laenge DIAG der
%   Diagonalen eines Quadrats.

if nargin < 2
  d=2;
end
vol = q^d;
diag = q*sqrt(d);

% Ende der Funktion wuerfeldaten2
```

Die folgende Anweisungssequenz läuft nun fehlerfrei durch.

```
>> [vv, dd] = wuerfeldaten2(1.0,3)
>> vv = wuerfeldaten2(1.0,3)
>> wuerfeldaten2(1.0,3)
>> [vv, dd] = wuerfeldaten2(1.0)
>> vv = wuerfeldaten2(1.0)
>> wuerfeldaten2(1.0)
```

Die Eingabe von

```
>> wuerfeldaten2;
```

führt dagegen erneut zu einem Fehler. Man lokalisiere diesen Fehler und überlege sich, wie man durch Erweitern der Funktion `wuerfeldaten2` einen sinnvollen und fehlerfreien Ablauf der erweiterten Funktion `wuerfeldaten3` garantieren kann.

5.2.1 Funktionen und Funktionsdateien

Anhand der obigen Beispiele sollte aufgefallen sein, daß ...

- ... sich der jeweilige Dateiname von dem Namen der Funktion, die in dieser Datei definiert wird, lediglich um das Suffix `.m` unterscheidet. Die Funktion `quadratflaeche` wird beispielsweise in der Datei `quadratfleache.m` definiert. Dies ist wichtig! Auf diese Art und Weise nimmt der MATLAB-Interpreter die Zuordung zwischen Datei und Funktion vor. Also nochmal: Eine Funktion namens `myfunction` ist in der Datei `myfunction.m` zu definieren, nirgends sonst!
- ... sich in einer Funktionsdatei genau eine Funktion befindet. Man darf nicht mehr als eine Funktion innerhalb einer Funktionsdatei definieren. Alles andere führt allein schon wegen des vorherigen Punktes zu einem Widerspruch. Bemerkung: Es gibt dennoch Ausnahmesituationen, in denen mehr als eine Funktion pro Datei definiert werden darf. Mehr dazu später.

Kapitel 6

MATLAB-Funktionen

6.1 Eingebaute Funktionen

Nicht jede MATLAB-Funktion korrespondiert zu einem M-file. Die meisten Funktionen, die von MATLAB zur Verfügung gestellt werden, sind in MATLAB *eingebaut*. Die Funktion `flops` ist eine solche Funktion. Gibt man die Anweisung

```
>> type flops
```

ein, so wird man dies feststellen. Dagegen ist die Funktion `rank`, mit der man den Rang einer Matrix berechnen lassen kann, keine eingebaute Funktion, sondern eine *öffentliche*. Mit der Anweisung

```
>> type rank
```

bekommt man den Inhalt des zugehörigen M-files angezeigt. Dies bietet die Möglichkeit, sich die Implementierung dieser Funktion anzuschauen.

6.2 Lokale und globale Variablen

Alle Variablen, die zu einem bestimmten Zeitpunkt während einer MATLAB-Sitzung dem MATLAB-Interpreter bekannt sind, werden als *globale Variablen* bezeichnet. Zu den globalen Variablen gehören sämtliche Variablen, die man sich mit der Anweisung

```
>> who
```

bzw.

```
>> whos
```

anzeigen lassen kann. Darüber hinaus gibt es bestimmte Variablen, die eine Sonderbedeutung für MATLAB haben. Beispielweise steht die Variable `i` für die imaginäre Einheit, sofern sie nicht während der Sitzung mit einem anderen Wert überlagert wurde. Generell gilt somit: Die Variable `x` ist eine globale Variable, falls die Eingabe von

```
>> x
```

vom MATLAB-Interpreter ausgewertet werden kann. Anderenfalls führt die obige Eingabe zur folgenden Fehlermeldung

```
??? Undefined function or variable 'x'.
```

Variablen, die innerhalb von Funktionen verwendet werden, sind üblicherweise *lokale Variablen*. Als Beispiele für lokale Variablen haben wir bereits die Ein- und Ausgabeparameter einer Funktion kennengelernt. Ebenso ist die in der Implementierung von `wuerfeldaten2` verwendete Variable `nargin` eine lokale Variable.

Operationen auf lokalen Variablen und Zuweisung an lokale Variablen innerhalb einer Funktion haben keine *Seiteneffekte*, d.h. sie haben keinen Einfluß auf die Werte von globalen Variablen.

In Skriptdateien gibt es dagegen keine lokalen Variablen. Alle Variablen, die in einer Skriptdatei erzeugt werden, sind dem MATLAB-Interpreter nach Laden der Skriptdatei bekannt. Weiterhin können innerhalb einer Skriptdatei die Werte von globalen Variablen verändert werden. Letzteres kann man unter Verwendung einer Funktionsdatei ebenso erreichen, wie wir weiter unten sehen werden.

Im folgenden wird die Bedeutung von lokalen und globalen Variablen sowie deren Unterschiede durch eine Reihe von Beispielen erläutert. Dabei wird jeweils zunächst der Inhalt eines M-files namens `schatten.m` abgedruckt, bevor eine entsprechende Folge von MATLAB-Anweisungen angezeigt wird. Auf die Kommentierung der jeweiligen Funktion `schatten` wird verzichtet. Man überlege sich mit Hilfe der folgenden Beispiele,

- welche Variablen während des Ablaufs der jeweiligen Sitzung global sind;
- welche Werte die einzelnen Variablen während des Ablaufs der Sitzung haben;
- ob der Ablauf, der durch die einzelnen Beispiele beschrieben wird, ggf. zu Fehlern führt.

Man überprüfe die jeweilige Einschätzung durch Rekonstruktion der einzelnen Beispiele unter Verwendung von MATLAB.

Beipiel 1: Dieses Beispiel zeigt, daß man ungestraft Funktionen definieren darf, die weder Ausgabe- noch Eingabeparameter enthalten.

```
function schatten()
x = 1;
disp('In schatten ist '); x
% Ende der Funktion schatten
```

```
>> clear;
>> schatten;
>> x
>> x = 0
>> schatten;
>> x
```

Beipiel 2: Ein ähnliches Beispiel wie oben.

```
function x = schatten()
x = 1;
disp('In schatten ist '); x
% Ende der Funktion schatten
```

```
>> clear;
>> x = 0
>> y = schatten
>> x
>> y
>> x = schatten
>> x
```

Beipiel 3: Es ist sogar erlaubt, gleichnamige Ein- und Ausgabeparameter zu verwenden.

```
function x = schatten(x)
x = x+1;
% Ende der Funktion schatten
```

```
>> clear;
>> x = schatten(x)
>> x = 0
>> schatten(x)
>> x
>> x = schatten(x)
>> x
```

Beipiel 4: Was ist mit globalen Variablen, etwa `pi`, die man innerhalb der Berechnungen einer Funktion gelegentlich braucht? Darf man diese ohne weiteres innerhalb der Funktion verwenden, oder muß man sie jedesmal neu definieren? Was passiert, wenn man den Wert von `pi` ändert?

```
function U = schatten(r)
U = 2.0*pi*r;
% Ende der Funktion schatten
```

```
>> clear;
>> U = schatten(1.0)
>> pi = 5.0
>> U = schatten(1.0)
```

Beipiel 5: Darf man *eigene* globale Variablen erzeugen, deren Werte (genauso wie oben für `pi`) innerhalb von Funktionen bekannt sind? So geht's nicht:

```
function U = schatten(r)
disp('In schatten ist '); diam
U = diam*r;
disp('In schatten ist '); U
% Ende der Funktion schatten
```

```
>> clear;
>> diam = 2*pi
>> U = schatten(1.0)
```

Beipiel 6: Aber so geht's:

```
function U = schatten(r)
global diam;
disp('In schatten ist '); diam
U = diam*r;
disp('In schatten ist '); U
% Ende der Funktion schatten
```

```
>> clear;
>> global diam;
>> diam = 2*pi
>> U = schatten(1.0)
>> diam = 2.0
>> U = schatten(1.0)
```

Erklärung: Nach der Anweisung

```
>> global diam
```

darf die Variable `diam` innerhalb von Funktionen als globale Variable verwendet werden, sofern die einzelnen Funktionen (so wie `schatten`) die Deklaration `global diam` enthalten. Tauchte diese Deklaration in `schatten` allerdings nicht auf, so würde die Variable `diam` wie bisher als eine lokale Variable behandelt.

Beipiel 7: Man beachte, daß die Funktion `schatten` die (globale) Variable `diam` global ändern darf.

```
function U = schatten(r)
global diam;
disp('In schatten ist '); diam
diam = 4.0;
disp('In schatten ist '); diam
U = diam*r;
disp('In schatten ist '); U
% Ende der Funktion schatten
```

```
>> clear;
>> global diam;
>> diam = 2*pi
>> U = schatten(1.0)
>> diam
```

Welche Variablen auf die obige Art und Weise als *global* markiert worden sind, bekommt man mit dem Aufruf

```
>> whos
```

angezeigt. Eliminiert man allerdings die Variable mit der Anweisung

```
>> clear diam;
```

und erzeugt sie anschließend erneut, etwa mit

```
>> diam = 5.0;
```

dann funktioniert der obige Mechanismus nicht mehr. Man überzeuge sich durch Ausprobieren von der Wirkung dessen und finde eine schlüssige Begründung für dieses Verhalten.

Beipiel 8: Man muß die Variable `diam` nicht notwendigerweise auf der Ebene des MATLAB-Interpreters als global definieren.

```
function schatten1(r)
global diam;
diam = 3.0;
disp('In schatten1 ist diam '); diam
% Ende der Funktion schatten1
```

```
function U = schatten2(r)
global diam;
disp('In schatten2 ist diam '); diam
U = diam*r;
% Ende der Funktion schatten2
```

```
>> clear;
>> diam
>> schatten1(1.0);
>> diam;
>> schatten2(1.0);
>> global diam;
>> diam
```

Beipiel 9: Aufruf einer Funktion innerhalb einer Funktion: Wir fassen `diam` nun als Funktion auf.

```
function d = diam()
disp('Funktion diam laeuft!');
d = 2*pi;
% Ende der Funktion diam
```

```
function U = schatten(r)
U = r*diam;
% Ende der Funktion schatten
```

```
>> clear;
>> diam
>> y = schatten(1.0)
>> diam
```

6.3 Rekursive Aufrufe innerhalb von Funktionen

MATLAB erlaubt es (genauso wie viele Programmiersprachen), daß sich Funktionen selbst aufrufen dürfen. So etwas nennt man *Rekursion* bzw. *rekursiver Funktionsaufruf.*

Ein Beispiel: Die Fibonacci-Zahlen sind *rekursiv* wie folgt definiert: Die n-te Fibonacci-Zahl f_n ist die Summe $f_{n-1} + f_{n-2}$ der $(n-1)$-ten Fibonacci-Zahl f_{n-1} und der $(n-2)$-ten Fibonacci-Zahl f_{n-2}. Für $n = 0, 1$ setzt man $f_0 = f_1 = 1$. Ein typisch mathematisches Beispiel! Rechnen wir doch vorab mal ein paar Fibonacci-Zahlen f_n aus: $f_2 = f_1 + f_0 = 2$, $f_3 = f_2 + f_1 = 3$, $f_4 = f_3 + f_2 = 5$ etc. Alles klar? Wir wollen nun eine Funktion namens `fibonacci` schreiben, die nach dem Aufruf

```
>> fibonacci(n)
```

die Fibonacci-Zahl f_n ausspuckt. Und so geht's: Wir erzeugen eine Datei `fibonacci.m` mit dem folgenden Inhalt:

```
function fn = fibonacci(n)
if n == 0
  fn = 1;
elseif n == 1
  fn = 1;
else
  fn = fibonacci(n-1) + fibonacci(n-2);
% Ende der Funktion fibonacci
```

Man überzeuge sich durch mehrere Aufrufe von

```
>> fibonacci(n)
```

für verschiedene (natürliche) Zahlen `n`, daß die obige Implementation das Gewünschte leistet.

Rekursive Funktionsaufrufe sind sinnvoll in Situationen, in denen man ein Problem in gleichartige kleinere Teilprobleme zerlegen kann. Bei der obigen Implementation der Funktion `fibonacci` sind bei der Berechnung von f_n die kleineren Teilprobleme gegeben durch die Berechnung von f_{n-1} und f_{n-2}. Somit kommt eine rekursive Implementation potentiell in Frage. Es wird aufgefallen sein, daß Berechnungen von f_n für große Zahlen n mit der obigen Implementation extrem lange Wartezeiten mit sich bringen. In der Tat kann man die Berechnung von f_n effizienter programmieren. Hier geht es uns aber ausschließlich um das Verständnis von rekursiven Funktionsaufrufen, und daher unterdrücken wir eine mögliche Verbesserung von `fibonacci` an dieser Stelle.

Sicherlich wissen Sie, daß man den Binomialkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)!k!}$ folgendermaßen zerlegen kann:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Wie wärs mit einer Implementation unter Verwendung von Rekursionen nach dem Vorbild der Implementation von `fibonacci`? Man überlege sich geeignete *Anfangsbedingungen.*

6.4 Funktionen als Eingabeargumente

Will man die Werte der Sinus-Funktion in einem bestimmten Intervall $I = [a, b]$ an n äquidistanten Stützstellen ermitteln, so kommt man für den Spezialfall $I = [-3, 1]$, $n = 10$ beispielsweise durch die folgende MATLAB-Anweisungssequenz zum Ziel.

```
>> a = -3;
>> b =  1;
>> n = 10;
>> y = [];
>> for k=0:n-1
     y = [y; sin(a+k/(n-1)*(b-a))];
   end
>> y
```

In vielen Anwendungen will man aber nicht nur die Parameter a, b und n, sondern auch die auszuwertende Funktion (hier sin) variieren. MATLAB bietet die Möglichkeit, nicht nur Variablen, sondern auch Funktionen als Eingabeparameter zu übergeben. Die Auswertung der eingegebenen Funktion geschieht mit der Anweisung `feval`. Für die Anwendung des obigen Beispiels sieht die

Implementation einer entsprechenden Funktion `auswerter`, die in der Datei `auswerter.m` abzuspeichern ist, wie folgt aus.

```
function y = auswerter(fun,a,b,n)
% AUSWERTER. Auswertung einer Funktion an aequidistanten Stuetzstellen
%   AUSWERTER(fun,a,b,n) wertet die Funktion fun in dem Intervall
%   I=[a,b] an n aequidistanten Stuetzstellen aus.
y = [];
for k=0:n-1
y = [y; feval(fun,a+k/(n-1)*(b-a))];
end;
% Ende der Funktion auswerter
```

Man beachte, daß die auszuwertende Funktion als Zeichenkette übergeben werden muß. Der Aufruf der Funktion `auswerter` sieht beispielsweise so aus.

```
>> y = auswerter('sin',-3,1,10);
```

Oder so.

```
>> y = auswerter('log',1,2,100);
```

Aber nicht so.

```
>> y = auswerter(log,1,2,100);
```

Man darf genauso gut selbstdefinierte Funktionen an `auswerter` übergeben. Hat man etwa eine Funktion `eins` der Form

```
function y = eins(x)
% EINS. Die Einsfunktion.
%   EINS(X) liefert den Wert Eins.
y = 1;
% Ende der Funktion eins.
```

geschrieben, so führt der Aufruf von

```
>> y = auswerter('eins',0,1,10);
```

dazu, daß `y` ein Spaltenvektor der Dimension 10×1 ist, deren Komponenten alle den Wert 1 haben.

6.5 Warnungen und Fehlermeldungen

Sehr häufig möchte man fehlerhaften Eingaben innerhalb von Funktionen mit entsprechenden Warnungen oder Fehlermeldungen und/oder Abbruch der entsprechenden Funktion entgegnen. Für folgendes wird hiermit vereinbart, daß eine *Warnung* zu einer Ausgabe eines Textes führt, der den Benutzer warnen soll. Dagegen zieht ein *Fehler* stets eine entsprechende Fehlermeldung sowie einen Abbruch der betroffenen Funktion nach sich.

In MATLAB erfolgt die Ausgabe einer Zeichenkette durch die Anweisung `disp`. Will man beispielsweise den Text *'Hello World!'* anzeigen lassen, so erreicht man dies durch die folgende Anweisung.

```
>> disp('Hello World!');
```

Innerhalb von Funktionen kann man genauso verfahren. Will man etwa in der Funktion `auswerter` die fehlerhafte Eingabe

```
>> auswerter('sin',1,-1,10);
```

des Intervalls $[1,-1]$ ersetzen durch $[-1,1]$, so kann man die Funktion `auswerter` wie folgt modifizieren.

```
function y = auswerter(fun,a,b,n)
% AUSWERTER. Auswertung einer Funktion an aequidistanten Stuetzstellen
%   AUSWERTER(fun,a,b,n) wertet die Funktion fun in dem Intervall
%   I=[a,b] an n aequidistanten Stuetzstellen aus.
if b < a
  disp('Fehlerhafte Eingabe des Intervalls wird korrigiert.');
  c = b;
  b = a;
  a = c;
end
y = [];
for k=0:n-1
  y = [y; feval(fun,a+k/(n-1)*(b-a))];
end;
% Ende der Funktion auswerter
```

Dagegen kann/will man die folgende unsinnige Eingabe nicht mehr retten.

```
>> y = auswerter('sin',0,1,-10);
```

Die Eingabe einer nichtpositiven Anzahl an Auswertungen kann man mit der folgenden Implementation von `auswerter` als Fehler behandeln.

```
function y = auswerter(fun,a,b,n)
% AUSWERTER. Auswertung einer Funktion an aequidistanten Stuetzstellen
%   AUSWERTER(fun,a,b,n) wertet die Funktion fun in dem Intervall
%   I=[a,b] an n aequidistanten Stuetzstellen aus.
if n <= 0
  error('Bitte positive Anzahl an Auswertungen eingeben.');
if b < a
  disp('Fehlerhafte Eingabe des Intervalls wird korrigiert.');
  c = b;
  b = a;
  a = c;
end
y = [];
for k=0:n-1
  y = [y; feval(fun,a+k/(n-1)*(b-a))];
end;
% Ende der Funktion auswerter
```

Nach Aufruf der Funktion `error` wird die Fehlermeldung ausgegeben, und die Funktion `auswerter` terminiert, d.h. es werden keine Auswertungen vorgenommen.

6.6 Interaktive Eingabe von Parameterwerten

Eine interaktive Eingabe von Parameterwerten ist durch den Befehl `input` etabliert, den man innerhalb von Funktionsdateien verwenden kann.

Man kann mit diesem Befehl beispielsweise Eingabeparameter ersetzen, etwa durch die folgende Implementation der Funktion `fac`, die $n!$ berechnen soll.

```
function y = fac
% FAC. Fakultaetsfunktion.
%   FAC liefert nach Eingabe einer positiven Zahl n
%   den Wert n! = 1*...*n.
n = 0;
while n <= 0
  n = input('Bitte eine positive Zahl n eingeben. ');
end
y = 1;
for i=2:n
  y = i*y;
end
% Ende der Funktion fac
```

Typischerweise wird man `input` jedoch verwenden für Nachfragen nach einer fehlerhaften Eingabe von Parametern.

Kapitel 7

Einfache Befehle zur Visualisierung

In diesem Kapitel sollen einige einfache Befehle zur Visualisierung mit MATLAB eingeführt werden. Wir werden uns im wesentlichen mit Beispielen an die Materie herantasten. Zunächst soll der Befehl `plot` ausführlich dargestellt werden, weil alle anderen grundlegenden Visualisierungsbefehle in MATLAB etwa der gleichen Logik folgen.

7.1 Die Funktion `plot`

Die grundlegende Funktion zur Darstellung von Funktionsgraphen oder 2D-Daten heißt `plot`. Die Syntax von `plot` lautet:

```
plot([x],y,[s][,x,y,s,...]).
```

Die eckigen Klammern sollen andeuten, daß die darin enthaltenen Teile ausgelassen werden können. `x` sei dabei ein Vektor mit x-Koordinaten, `y` ist der Datenvektor, `s` ist ein String, der die Kurvendarstellung beschreibt. Ein einfaches Beispiel:

```
>> y=sin(0:0.1:2*pi);
>> plot(y)
```

stellt die Sinus-Kurve im Intervall $[0, 2\pi]$ dar. Man beachte, daß auf der x-Achse nicht das Intervall dargestellt ist, sondern die Anzahl der Daten im Vektor y.

Soll auch die x-Achse die richtigen Werte tragen, kommt man um eine Eingabe des Vektors x nicht herum:

```
>> x=0:0.1:2*pi;
>> y=sin(x);
>> plot(x,y)
```

Wenn zwei Graphen, die durch verschiedene `plot`-Befehle erzeugt werden, in der selben Abbildung dargestellt werden sollen, muß der Befehl `hold` verwendet werden. Wir plotten die Sinus- und die Kosinus-Kurve:

```
>> plot(sin(0:0.1:2*pi))
>> hold on
>> plot(cos(0:0.1:2*pi))
>> hold off
```

Zunächst wird die Sinus-Kurve erzeugt. Dann wird mit `hold on` die Darstellung fixiert, so daß die Kosinus-Kurve darübergelegt werden kann. Nach dem Plotten der Kosinus-Kurve schalten wir mit `hold off` wieder in den einfachen Darstellungsmodus zurück.

Um die beiden Kurven unterscheiden zu können, sollen sie unterschiedliche Farben erhalten. Außerdem soll die Kosinus-Kurve als gestrichelte Kurve dargestellt werden:

```
>> plot(sin(0:0.1:2*pi),'r')
>> hold on
>> plot(cos(0:0.1:2*pi),'b--')
>> hold off
```

Die vielfältigen Möglichkeiten der Einfärbung und "Strichelung" schaut man sich am besten in der MATLAB-Online-Hilfe an.

Beliebige Datenpaare können mit dem Befehl `plot` ebenso einfach dargestellt werden:

```
>> x=[1 2 3 4 5 6 7 8 9 10];
>> y=[95 23 60 48 89 76 45 2 82 44];
>> plot(x,y)
```

Hat man mehrere (x, y)-Datensätze, können sie sogar mit einer `plot`-Anweisung (ohne `hold`) dargestellt werden:

```
>> x=[1 2 3 4 5];
>> y=[76 45 2 82 44];
>> z=[95 23 60 48 89];
>> plot(x,y,'r',x,z,'g')
```

Will man bei diskreten Datensätzen einerseits die Meßpunkte darstellen, andererseits aber auch eine Kurve erhalten, dann plottet man die Kurve einfach zweimal mit unterschiedlichen Optionen für die Darstellungsart:

```
>> x=[1 2 3 4 5 6 7 8 9 10];
>> y=[95 23 60 48 89 76 45 2 82 44];
>> plot(x,y,'r-',x,y,'bo')
```

Wenn $c \in \mathbb{C}^n$ ein komplexwertiger Vektor ist, dann wird der `plot`-Befehl die Elemente des Vektors c in der komplexen Zahlenebene darstellen. Dabei wird der Realteil als x-Koordinate, der Imaginärteil als y-Koordinate interpretiert:

```
>> c=rand(1,20)+i.*rand(1,20);
>> plot(c,'r.')
```

7.2 Titel, Achsenbeschriftungen, Erscheinungsbild

Es ist zwar im allgemeinen schon schön, überhaupt Funktionsgraphen darstellen zu können, aber um Darstellungen in die Examensarbeit zu übernehmen, oder auf Folien zu präsentieren, bedarf es doch ein wenig mehr Gestaltungsmittel. In diesem Abschnitt werden die Gestaltungsmittel behandelt.

Um Gitterlinien in die Graphik einzufügen, verwendet man den Befehl `grid`:

```
>> x=[1 2 3 4 5 6 7 8 9 10];
>> y=[95 23 60 48 89 76 45 2 82 44];
>> plot(x,y,'r-',x,y,'bo')
>> grid on
```

Man kann eine MATLAB-Graphik mit einem Titel versehen (wir verwenden das obige Beispiel weiter):

```
>> title('gezackte Kurve')
```

Auch die Achsen können beschriftet werden:

```
>> xlabel('x-Achse')
>> ylabel('y-Achse')
```

Mit dem Befehl `axis` läßt sich die Achsenskalierung verändern. Dieser Befehl ist recht mächtig und die einzelnen Möglichkeiten sollte man sich mittels der MATLAB-Online-Hilfe erschließen. Mit dem folgenden Befehl wird aus einer gezackten Kurve fast eine Gerade (jedenfalls optisch):

```
>> axis([0 10 -2000 2000])
```

Innerhalb der Graphik kann man Texte plazieren. Der Befehl dazu heißt `text`. Er erwartet die Koordinaten und einen Text-String:

```
>> text(1,200,'hier ist das Maximum')
```

Manchmal ist auch der Befehl `gtext` hilfreich. Er erlaubt die Positionierung eines Textes durch den Benutzer. Um es auszuprobieren, geben wir ein:

```
>> gtext('das Minimum wird per Hand gesucht')
```

Im Graphikfenster erscheint (wenn man die Maus hineinbewegt) ein Fadenkreuz, das die Position des Textstrings markiert. Wird nun eine Maustaste gedrückt, erscheint der Text an der markierten Stelle.

Schließlich kann man eine Legende erstellen mit dem Befehl `legend`:

```
>> legend('Interpolierte Linie','Datenpunkte')
```

Wir wollen diesen Abschnitt beenden mit der Bemerkung, daß es eine weitere Vielzahl an Möglichkeiten zur schönen Gestaltung von Graphiken in MATLAB gibt. Man sollte sich der Online-Hilfe bedienen und ein wenig herumspielen, um sich dieses mächtige Werkzeug anzueignen.

7.3 Einfache 3D-Graphiken

Dieser Abschnitt soll lediglich einige nützliche 3D-Plot-Routinen vorstellen. Die Möglichkeiten mit MATLAB zu visualisieren sind sehr mächtig, alle Einzelheiten zu erwähnen, würde ein eigenes Buch füllen.

Zunächst erzeugen wir uns einen Datensatz, der visualisiert werden kann:

```
>> s=peaks(50);
```

Nun können wir diese Funktion als Gitterlinien darstellen:

```
>> mesh(s)
```

Zur Darstellung von Konturlinien verwenden wir:

```
>> contour(s)
```

Für gefüllte Konturflächen kann der Befehl

```
>> contourf(s)
```

verwendet werden. Wenn Oberflächen dargestellt werden sollen:

```
>> surf(s)
```

Die Farben können mit dem Befehl `colormap` verändert werden:

```
>> colormap(hot)
```

Es gibt gut ein Duzend fertig konfigurierte Farbtabellen. Probieren Sie den Befehl mit den Namen `flag`, `gray`, `hsv`, `jet`, `prism` oder `winter` aus.

Wenn die Flächen glatt dargestellt werden sollen:

```
>> shading interp
```

Mit diesem Befehl können zwei weitere Einstellungen vorgenommen werden: `flat` für eine weniger glatte Einfärbung und `faceted` für eingeblendete Gitternetzlinien.

Eine einfache Methode, dreidimensionale Graphiken zu drehen, wird durch den Befehl `rotate3d` bereitgestellt:

```
>> rotate3d on
```

Nun kann die Graphik mit der Maus gedreht und gewendet werden.

Auch bei dreidimensionalen Graphiken können natürlich die im Abschnitt 7.2 besprochenen Befehle zur Achsenbeschriftung, etc. verwendet werden. Zusätzlich gibt es den Befehl `zlabel`, um die z-Achse zu beschriften. Befehle, die Koordinateneingaben erwarten, müssen nun mit jeweils drei Koordinaten "versorgt" werden (dies betrifft die Befehle `text` und `axis`).

7.4 Weitere Routinen zur Visualisierung

Logarithmische Achsenskalierungen kann man einfach erzeugen durch die Befehle `loglog`, `semilogx` und `semilogy`. So kann beispielsweise aus der Exponentialfunktion

```
>> plot(exp(0:0.1:5))
```

mit dem Befehl

```
>> semilogy(exp(0:0.1:5))
```

eine Gerade erzeugt werden.

Der Befehl `subplot` plaziert mehrere kleine Graphiken in einem Fenster. Die Syntax des Befehls ist:

```
subplot(m,n,p)
```

Dabei wird eine $(m \times n)$-Matrix aus m Zeilen und n Spalten von Teilgraphiken definiert. p wählt die p-te Untergraphik aus. Damit lassen sich Graphiken ausdrucken, die mehrere Koordinatensysteme enthalten. Als Beispiel drucken wir die Sinus- und die Kosinus-Kurve in zwei verschiedenen Koordinatensystemen untereinander in ein Fenster:

```
>> subplot(2,1,1), plot(sin(0:0.1:2*pi))
>> subplot(2,1,2), plot(cos(0:0.1:2*pi))
```

7.5 Graphiken ausdrucken

Der Befehl `print` druckt eine Graphik aus. Dabei kann man mit Hilfe von Optionen die Ausgabe des Befehls beeinflussen.

```
>> print
```

ohne irgendwelche Zusätze druckt die Graphik auf dem Standarddrucker aus. Man kann sich ansehen, welchen Druckbefehl und welche Optionen MATLAB verwendet. Das geschieht mit dem Befehl:

```
>> printopt
```

Meist möchte man aber eine PostScript-Datei erzeugen, die man dann in einen Text einfügen kann. Oder man benötigt eine JPEG-Datei, die man in HTML-Seiten für die Internet-Präsentation einbinden möchte. Auch solche Dateien können mit dem `print`-Befehl erzeugt werden. Dazu muß man den *Device*-Namen angeben und einen Dateinamen:

```
>> print -deps myplot.eps
```

erzeugt eine Datei namens "myplot.eps". Das Format dieser Datei ist EPS (encapsulated PostScript). Eine JPEG Datei erzeugt man mit:

```
>> print -djpeg80 myplot.jpg
```

Dabei sagt die Zahl "80" nach dem Device-Namen `-djpeg`, daß die Qualitätsstufe für die JPEG-Komprimierung 80% beträgt. Die folgende Tabelle führt einige übliche Devices auf:

`-dps`	PostScript-Datei
`-dpsc`	farbige PostScript-Datei
`-deps`	encapsulated PostScript-Datei
`-depsc`	farbige encapsulated PostScript-Datei
`-dmfile`	Matlab M-file und MAT-file
`-djpeg`*nn*	JPEG-Datei der Qualität *nn*
`-dtiff`	TIFF-Graphik-Datei

Die Orientierung der Seite kann schließlich mit dem Befehl `orient` gesteuert werden:

```
>> orient portrait
```

wählt das Seitenformat “hochkant”, während

```
>> orient landscape
```

ein querformatiges Seitenformat wählt.

Index