Programmierpraktikum mit MATLAB

Elena Isaak

1 TEIL I

1.1 Einführung in Matlab-Kurs

MATLAB ist eine hoch-technische Berechnungssprache und interaktive Umgebung der Algorithmenentwicklung, Datenvisualisierung, Datenanalyse und numerischer Berechnung. Kurz gesagt: Matlab ist ein Softwarepaket das in vielen Bereichen so wie Signal- und Bildvearbeitung, Kommunikationen, Kontrolldesign, Testsmessungen, Finanzmodellierung und Computational Biology anwendbar. Es wurde in den 70-er Jahren an der University of Mexico und der Standford University entwickelt um Kurse aus lineare Algebra und Numerische Analysis zu unterstützen. Der Name MATLAB steht für "Matrix laboratory". Also, der Grundbaustein in Matlab ist eine Matrix, auch Skalare werden intern als 1×1 - Matrix gespeichert.

1.2 Erste Schritte mit Matlab

- **Einlogen**

  Für Studierende, die über das Teilnehmermanagement einen Platz in dieser Veranstaltung erhalten haben wurde, falls nicht bereits vorhanden, ein Account angelegt.

- **Password ändern** (falls nötig).

  Man verwendet den Befehl $ *passwd* der Linux-Kommandozeile (Shell). Das Dolarzeichen $ steht vor den Shell Befehlen, wird aber nicht eingegeben.

- **Matlab starten**

  Mit Hilfe der Shell können wir Matlab starten: $ *matlab*

  Nach dem Start von Matlab öffnet sich das Befehlfenster (Command Window). Es erscheint das Prompt und wenn Command Window aktiv ist, blinkt der Kursor. In das Befehlfenster werden Matlab-Anweisungen eingegeben und numerische Ergebnisse ausgegeben. Alle Anweisungen werden nach Prompt eingegeben und mit der Enter-Taste bestätigt. Matlab nennt das Ergebniss *ans*, das für *answer* steht. Standardmäßig rechts und links öffnen sich noch weitere Fenster: Current Folder (oben links), Details (unten links), Workspace (oben rechts) und Command History (unten rechts). Man kann die Position aller Fenster mit dem Klick auf "Layout" von der Matlab-Desktop Symbolleiste ändern. Im Teilfenster Current Folder werden alle aktuelle Ordner gezeigt und unten links erscheint zu jedem Ordner ein Komentar (falls es vorhanden). Im Teilfenster Workspace werden die momentan angelegten

Variablen aufgeführt und unten rechts werden Befehle gespeichert, die man bereits ausgeführt hat.

1.3 Gewöhnliche Mathematik mit Matlab

So wie ein Kalkulator kann Matlab auch die Grundrechenarten vornehmen. Wir betrachten ein folgendes Beispiel.

Beispiel 1.3.1. Firma Kramm hat für Büromitarbeiter elf Packungen Weisspapier, fünfzehn Marker und zehn Tacker bestellt. Eine Packung Weisspapier kostet 5.39 Euro, ein Marker kostet 2.69 Euro und ein Tacker kostet 3.95 Euro. Wie viel Gegenstände sind es insgesamt und wie viel Geld soll die Firma für diese Bestellung ausgeben?

Um diese Aufgabe zu lösen benutzen wir den Kalkulator und erhalten

$$11 + 15 + 10 = 36 \text{ Gegenstände}$$
$$11 \times 5.39 + 15 \times 2.69 + 10 \times 3.95 = 139.14 \text{ Euro}$$

In Matlab kann man dieses Problem auf verschiedenen Wegen lösen.

```
>>11 + 15 + 10
   ans =
      36
>>11*5.39 + 15*2.69 + 10*3.95
   ans =
      139.14
```

Als Alternative könnte man diese Aufgabe durch Speichern der Information über die Matlabvariablen lösen.

```
>>weisspapier = 11
   weisspapier =
                11
>>marker = 15
   marker =
          15
>>tacker = 10;
>>anzahl = weisspapier + marker + tacker
   anzahl =
          36
```

$$>> geldbetrag = weisspapier*5.39 + marker*2.69 + tacker*3.95$$
$$geldbetrag =$$
$$139.14$$

Hier haben wir drei Matlabvariablen erzeugt: *weisspapier*, *marker* und *tacker*. Nach dem *Enter* wird das Resultat angezeigt. Ein Semikolon am Ende der Eingabe der Variablen (wie im Fall *tacker*) befehlt dem Matlab diese auszuwertren, aber nicht das Ergebniss anzeigen. Schliesslich haben wir die Anzahl der Gegenstände und den gesammten Preis als *anzahl* und *geldbetrag* genannt. In jedem Schritt wird die entsprechende Information in Matlab gespeichert.

Jetzt sind alle Daten von unseren Variablen in Matlab bekannt. Also, wir können den Durchschnittspreis pro Gegenstand auswerten:

$$>> durchschnitt\_Preis = geldbetrag/anzahl$$
$$durchschnitt\_Preis =$$
$$3.87$$

Mit dem Strich in der Variable *durchschnitt_Preis* kann man zwei oder mehr Wörter verbinden. Matlab akzeptiert nicht das Leerzeichen in Variablennamen so wie z.B.

$>>$ *durchschnittlicher Preis* $= 3.87,$

aber erkennt große und kleine Buchstaben. Insgesamt bietet Matlab folgende Operationen:

Addition	$+$,
Subtraktion	$-$,
Multiplikation	$*$,
Division	$/$ oder $\backslash$,
Potenz	$\wedge$.

Die Ausdrücke werden von links nach rechts ausgewertet, wobei die Potenzoperation den höchsten Vorrang hat, dann folgen Multiplikation oder Division. Zum Schluss wertet Matlab die Operationen mit Addition oder Subtraktion aus.

1.4 Matlab Workspace (Arbeitsplatz)

Sobald fängt man im Befehlfenster (Command window) zu arbeiten, speichert Matlab alle Daten von Variablen und deren Werte im Workspace. Diese Variablen und Werte

kann man jeder Zeit zurückrufen

```
>>marcker
    marker =
          15
>>tacker
    tacker =
          10
>>weisspapier
    weisspapier =
          11
>>anzahl
    anzahl =
          36
```

Wenn man eine Variable ändert, zum Beispiel $tacker = 20$, wird dabei die Variable *anzahl* unverändert in Matlab gespeichert

```
>>tacker = 20
    tacker =
          20
>>anzahl
    anzahl =
          36
```

Für Neuberechnung der Anzahl der Gegenstände ist es notwendig noch mal die Operation für die Variable *anzahl* durchzuführen. Mit dem Befehl *who* kann man die Variablennamen zurückrufen

```
>>who
    Your variables are :
    ans                 geldbetrag   tacker      marker
    durchschnitt_Preis  anzahl       weisspapier
```

Mit diesem Befehl erhält man nicht die Werte der Variablen sondern nur die Namen. Weiter, mit dem Drücken der Schift-Taste ↑ kann man vorherige Befehle wieder aufrufen.

1.5 Variablen in Matlab

So wie anderen Computersprachen hat Matlab folgende Regeln für Variablennamen:

- Variablennamen können aus großen und kleinen Buchstaben bestehen.
- Variablennamen können bis 63 Zeichen enthalten.
- Variablennamen müssen mit Buchstaben anfangen, gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern und Unterstriche.
- Satzzeichen sind nicht erlaubt, da jeder von ihnen hat eine spezielle Bedeutung in Matlab.

Es gibt aber einige Ausnahmen, die man beachten soll: die Variablen dürfen nicht Namen von der reservierten Wort-Liste in Matlab ($function$, $case$, sum, exp, usw.) tragen.

1.6 Arbeiten mit Matrizen

Eine Matrix in der Matlab-Umgebung ist ein rechteckiges Feld von Zahlen. Besondere Bedeutung ist 1×1-Matrix, die einen Skalar in Matlab zuordnet, oder Matrizen, die nur eine Zeile oder eine Spalte enthalten, werden zu Vektoren zugeordnet. Matlab hat andere Wege numerische und nichtnumerische Angaben zu speichern, aber am Anfang ist es am besten alles als eine Matrix vorzustellen. Ein berühmtes Beispiel einer Matrix ist sogenanntes " Magisches Quadrat "

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

Im Matlab wird diese Matrix folgenderweise eingegeben:

$$>> MQ = [16 \quad 3 \quad 2 \quad 13; \quad 5 \quad 10 \quad 11 \quad 8; \quad 9 \quad 6 \quad 7 \quad 12; \quad 4 \quad 15 \quad 14 \quad 1];$$

Beim Eingeben einer Matrix in Matlab beachte folgende Regeln:

- Trenne die Elemente einer Zeile mit Leerzeichen oder Komma.
- Benutze Semikolon um das Ende jeder Zeile zu zeigen.
- Schließe alle Elemente einer Matrix in eckigen Klammern ein.

Solche Matrizen kann man auch mit der vorreservierten in Matlab Funktion *magic* angeben:

$$>>B = magic(4)$$

$$B =$$

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

$$>>sum(B)$$

$$ans =$$

$$\begin{matrix} 34 & 34 & 34 & 34 \end{matrix}$$

Hinweis: Es ist sehr hilfreich mit dem Befehl *help* alle vorreservierte Funktionen in Matlab abfragen, besonders wenn man solche Funktionen nicht kennt. Durch Eingeben *help magic* erhält man Information über diese Funktion:

```
>>help magic
  magic  Magic square.
  magic(N) is an N-by-N matrix constructed from the integers
  1 through N^2 with equal row, column, and diagonal sums.
  Produces valid magic squares for all N > 0 except N = 2.
  Reference page in Help browser
  doc magic
```

Die Matrix B ist fast gleich mit der oberen Matrix MQ. Nur zweite und dritte Spalte sind hier vertaucht. Aus der Matrix B kann man die Matrix MQ folgenderweise erzeugen:

$$>>MQ = B(:, [1, 3, 2, 4])$$

$$MQ =$$

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

Matlab Softwarepaket bietet vier Funktionen, die Basismatrizen erzeugen

- *zeros* Nullmatrix
- *ones* Einsmatrix
- *rand* Gleichmäßig verteilte Zufallselemente auf dem Interval $[0, 1]$.
- *randn* Normal verteilte Zufallselemente

Hier sind einige Beispiele:

$$>>Z = zeros(3,4)$$
$$Z =$$
$$\begin{matrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$
$$>>A = ones(3)$$
$$A =$$
$$\begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$
$$>>R = rand(2)$$
$$R =$$
$$\begin{matrix} 0.8147 & 0.1270 \\ 0.9058 & 0.9134 \end{matrix}$$
$$>>N = randn(4)$$
$$N =$$
$$\begin{matrix} 0.3188 & 3.5784 & 0.7254 & -0.1241 \\ -1.3077 & 2.7694 & -0.0631 & 1.4897 \\ -0.4336 & -1.3499 & 0.7147 & 1.4090 \\ 0.3426 & 3.0349 & -0.2050 & 1.4172 \end{matrix}$$

Wenn man nur eine Zahl in den Klamern angibt, wird in Matlab automatisch eine quadratische Matrix erzeugt. Mit dem Befehl *size* kann man Größe einer Matrix ansehen:

$$>>size(Z)$$
$$ans =$$
$$\begin{matrix} 3 & 4 \end{matrix}$$
$$>>size(N)$$
$$ans =$$
$$\begin{matrix} 4 & 4 \end{matrix}$$

So wie in anderen Computersprachen kann man mit dem % Zeichen einen Kommentar in Matlab schreiben. Um eine bestimmte Zeile oder Spalte zu löschen, benutzt man eckigen

Klammern

$$>>N(2,:) = [\ ]$$
$$N =$$
$$\begin{matrix} 0.3188 & 3.5784 & 0.7254 & -0.1241 \\ -0.4336 & -1.3499 & 0.7147 & 1.4090 \\ 0.3426 & 3.0349 & -0.2050 & 1.4172 \end{matrix}$$
$$>>N(:,2) = [\ ]$$
$$N =$$
$$\begin{matrix} 0.3188 & 0.7254 & -0.1241 \\ -1.3077 & -0.0631 & 1.4897 \\ -0.4336 & 0.7147 & 1.4090 \\ 0.3426 & -0.2050 & 1.4172 \end{matrix}$$

Mathematische Operationen auf Matrizen sind Subjekte linearer Algebra. Zum Beispliel, Addition einer Matrix mit ihrer transponierten Matrix ergibt eine symmetrische Matrix:

$$>>S = MQ + MQ'$$
$$S =$$
$$\begin{matrix} 32 & 8 & 11 & 17 \\ 8 & 20 & 17 & 23 \\ 11 & 17 & 14 & 26 \\ 17 & 23 & 26 & 2 \end{matrix}$$

Das Multiplikationszeichen $*$ bezeichnet Matrizenmultiplikation. Multiplikation einer transponierten Matrix mit ihrer ursprünglichen Matrix ergibt auch eine symmetrische Matrix:

$$>>M = MQ' * MQ$$
$$M =$$
$$\begin{matrix} 378 & 212 & 206 & 360 \\ 212 & 370 & 368 & 206 \\ 206 & 368 & 370 & 212 \\ 360 & 206 & 212 & 378 \end{matrix}$$

Wegen der Singularität MQ ist Determinante gleich Null:

$$>>det(MQ)$$
$$M =$$
$$0.0000$$

Matrix-Division in Matlab kann man am besten mit Hilfe von linearen Gleichungssystemen

$$Ax = y,$$

und auch als Multiplikation $x = A^{-1}y$ durchführen. In dem Beispliel oben hat die Matrix MQ keine Inverse, weil sie singulär ist.
Die Potenzoperation wird als Multiplikation $MQ^3 = MQ \times MQ \times MQ$ ausgerechnet

$>> MQ \wedge 3$

$ans =$

10306	9474	9410	10114
9602	9922	9986	9794
9858	9666	9730	10050
9538	10242	10178	9346

1.7 Arbeiten mit Arrays

Zuerst betrachten wir ein Beispiel zur Berechnung der Werte einer Sinusfunktion auf dem Intervall $[0, \pi]$. Da eine Berechnung von $\sin(x)$ in allen Punkten auf diesem Intervall (es sind unendlich viele) nicht möglich ist, müssen wir eine endliche Anzahl der Punkten wählen. Wir rechnen die Werte von Sinusfunktion in den Punkten $x = 0,\ 0.1\pi,\ 0.2\pi, \ldots, \pi$ aus. Wenn wir zum Taschenrechner greifen, um die Werte von $\sin(x)$ in jedem Punkt x auszurechnen, bekommen wir folgende Tabelle für $y = \sin(x)$:

x	0	0.1π	0.2π	0.3π	0.4π	0.5π	0.6π	0.7π	0.8π	0.9π	π
y	0	0.31	0.59	0.81	0.95	1	0.95	0.81	0.59	0.31	0

Hier sind x und y eine geordnete Liste von Zahlen, d.h., der erste Wert von y entspricht dem ersten Wert von x, der zweite Wert von y entspricht dem zweiten Wert von x und so weiter. Aber eine Durchführung wiederholten Skalaroperationen ist zeitaufwendig und umständlich. Matlab bietet für solches Rechnen Operationen auf Arrays.

Wir erzeugen folgende Arrays:

$>> x = [0 \ \ 0.1 * pi \ \ 0.2 * pi \ \ 0.3 * pi \ \ 0.4 * pi \ \ 0.5 * pi$
$0.6 * pi \ \ 0.7 * pi \ \ 0.8 * pi \ \ 0.9 \ \ pi]$

$x =$

Columns 1 *through* 9

0 0.3142 0.6283 0.9425 1.2566 1.5708 1.8850 2.1991 2.5133

Columns 10 *through* 11

2.8274 3.1416

In diesem Fall ist x ein Array, das nur eine Zeile und 11 Spalten hat. In Mathematischer Sprache ist es ein Zeilenvektor mit der Länge 11.

```
>>y = sin(x)
  y =
Columns 1 through 9
    0    0.3090    0.5878    0.8090    0.9511    1.0000    0.9511    0.8090    0.5878
Columns 10 through 11
    0.3090    0.0000
```

Bemerkung: Matlab versteht, dass man hier Sinus von jedem Element x sucht, und das Ergebniss in einem zugeordneten Array y bekommen will. Diese fundamentale Fähigkeit unterscheidet Matlab von anderen Computersprachen.

In Matlab werden einzelne Array-Elemente durch Indizes abgerufen:

```
>>x(1)  %das erste Element
ans =
     0
>>x(4)  %das vierte Element
ans =
    0.9425
```

Um auf einen Block von Array-Elemente zuzugreifen, bietet Matlab Doppelpunkt-Notation:

```
>>x(1:6)
ans =
     0    0.3142    0.6283    0.9425    1.2566    1.5708
>>x(6:11)
ans =
    1.5708    1.8850    2.1991    2.5133    2.8274    3.1416
>>x(3:-1:1)  %rueckwaerts
ans =
    0.6283    0.3142         0
>> x(2:2:8)
ans =
    0.3142    0.9425    1.5708    2.1991
```

Außer Eingabe von einzelten Elementen x im Command Window kann man noch in zwei

anderen Wegen die Arrays eingeben:

$>> x = (0 : 0.1 : 1) * pi$

$x =$

$Columns\ 1\ through\ 9$

0 0.3142 0.6283 0.9425 1.2566 1.5708 1.8850 2.1991 2.5133

$Columns\ 10\ through\ 11$

2.8274 3.1416

$>> x = linspace(0, pi, 11)$

$x =$

$Columns\ 1\ through\ 9$

0 0.3142 0.6283 0.9425 1.2566 1.5708 1.8850 2.1991 2.5133

$Columns\ 10\ through\ 11$

2.8274 3.1416

Im ersten Fall die Notation $(0 : 0.1 : 1)$ bedeutet: beginne mit Null mit dem Schritt 0.1 und ende mit Eins. Danach wird jedes Element mit π multipliziert. In dem zweiten Fall wird Matlab Funktion *linspace* benutzt, um das Array x zu erzeugen. Diese Funktion wird folgenderweise beschrieben:
$>> linspace(erstes\_element,\ letztes\_element,\ anzahl\_der\_elemente)$

Das Eingeben von Arrays im Matlab kann man durch folgende Regeln verallgemeinern:

- $x = [1 \quad pi \quad sqrt(3) \quad 1i{+}4]$ - Erzeuge einen Zeilenvektor mit beliebigen Elementen.
- $x = erstes\_element : letztes\_element$ - Erzeuge einen Zeilenvektor mit dem Startwert $erstes\_element$, zähle jedes Element mit dem Schritt 1 und ende mit dem Endwert $letztes\_element$.
- $x = erstes\_element : schritt : letztes\_element$ - Erzeuge einen Zeilenvektor mit dem Startwert $erstes\_element$, zähle jedes Element mit dem Schritt $schritt$ und beende mit dem $letztes\_element$.

Beispiel 1.7.1.

```
>>a = 1 : 7
  a =
       1   2   3   4   5   6   7
>>b = a '
  b =
       1
       2
       3
       4
       5
       6
       7
```

Mit dem Zeichenstrich wird ein transponierter Vektor beziehungsweise transponiertes Array erzeugt. Für Arrays mit komplexen Elementen ist dieser Strich bedeutet konjugierte Operation. Der Code

```
>>c = a. '
  c =
       1
       2
       3
       4
       5
       6
       7
```

zeigt, dass . ′ (elementweise Transponieren) und ′ sind identisch für Arrays mit reellen Elementen. Für ein komplexwertiges Array

```
>>d = complex(a, a)
  d =
Columns 1 through 4
   1.0000 + 1.0000i   2.0000 + 2.0000i   3.0000 + 3.0000i   4.0000 + 4.0000i
Columns 5 through 7
   5.0000 + 5.0000i   6.0000 + 6.0000i   7.0000 + 7.0000i
```

bekommt man mit $'$ eine komplexkonjugierte Matrix

$$
\begin{aligned}
&>> d\,' \\
&ans = \\
&\qquad 1.0000 - 1.0000i \\
&\qquad 2.0000 - 2.0000i \\
&\qquad 3.0000 - 3.0000i \\
&\qquad 4.0000 - 4.0000i \\
&\qquad 5.0000 - 5.0000i \\
&\qquad 6.0000 - 6.0000i \\
&\qquad 7.0000 - 7.0000i
\end{aligned}
$$

und mit $.'$ wird einfach eine transponierte Matrix erzeugt

$$
\begin{aligned}
&>> d.\,' \\
&ans = \\
&\qquad 1.0000 + 1.0000i \\
&\qquad 2.0000 + 2.0000i \\
&\qquad 3.0000 + 3.0000i \\
&\qquad 4.0000 + 4.0000i \\
&\qquad 5.0000 + 5.0000i \\
&\qquad 6.0000 + 6.0000i \\
&\qquad 7.0000 + 7.0000i
\end{aligned}
$$

Außerhalb linearer Algebra, werden Matrizen in Matlab zur zwei-dimensionalen numerischen Arrays. Mathematische Operationen auf Arrays werden elementweise durchgeführt. Das bedeutet, dass Addition und Subtraktion sowohl für Matrizen als auch für Arrays gleich sind. Andere Operationen so wie Multiplikation, Division und Potenz werden auf Arrays nur elementweise durchgeführt. In diesem Fall wird in Matlab ein Punkt vor der Operation eingesetzt:

- $.*$ elementweise Multiplikation
- $./$ elementweise Division
- $.\wedge$ elementweise Potenzoperation

Wenden wir eine elementweise Multiplikation auf die MQ Matrix:

$$
\begin{aligned}
&>> MQ.*MQ \\
&ans = \\
&\qquad \begin{matrix} 256 & 9 & 4 & 169 \\ 25 & 100 & 121 & 64 \\ 81 & 36 & 49 & 144 \\ 16 & 225 & 196 & 1 \end{matrix}
\end{aligned}
$$

Eine elementweise Division ergibt eine Einsmatrix:

$$>>MQ./MQ$$
$$ans =$$
$$\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{matrix}$$

In dem Beispiel

$$>>1./MQ$$
$$ans =$$
$$\begin{matrix} 0.0625 & 0.3333 & 0.5000 & 0.0769 \\ 0.2000 & 0.1000 & 0.0909 & 0.1250 \\ 0.1111 & 0.1667 & 0.1429 & 0.0833 \\ 0.2500 & 0.0667 & 0.0714 & 1.0000 \end{matrix}$$
$$>>1/MQ$$
$$Error\ using\ /$$
$$Matrix\ dimensions\ must\ agree.$$

sieht man den Unterschied zwischen beiden Divisionsoperationen. Im ersten Fall wurde 1 zu einem 4×4-Array erweitert und elementweise auf die Matrix MQ dividiert. In dem zweiten Fall meldet Matlab einen Fehler, weil / Zeichen eine Division aus linearer Algebra bedeutet.

1.8 Manipulationen auf Matrizen und Arrays

Da Matrizen beziehungsweise Arrays von grundlegender Bedeutung für MATLAB sind, gibt es viele Möglichkeiten sie in Matlab zu manipulieren. Um diese Manipulationen zu veranschaulichen, betrachten wir folgende Beispiele:

$$>>A = [1\ 2\ 3;\ 4\ 5\ 6;\ 7\ 8\ 9]$$
$$A =$$
$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

$>>A(3,3) = 0$

$A =$

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$$

Mit dieser Operation wird das Element in der dritten Zeile und dritten Spalte auf Null gesetzt. Der Code

$>>A(2,5) = 1$ %*setzt Element in die zweite Zeile und fuenfte Spalte*

$A =$

$$\begin{matrix} 1 & 2 & 3 & 0 & 0 \\ 4 & 5 & 6 & 0 & 1 \\ 7 & 8 & 0 & 0 & 0 \end{matrix}$$

setzt die Eins in die zweite Zeile und fünfte Spalte. Hier erweitert Matlab die Matrix A auf fünf Spalten und füllt die restlichen Einträge mit 0.

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

$>>B = A(3:-1:1, 1:3)$

$B =$

$$\begin{matrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{matrix}$$

Oder man kann die selbe Matrix B mit dem *end* Befehl erzeugen:

$>>B = A(end:-1:1, 1:3)$

$B =$

$$\begin{matrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{matrix}$$

Außerdem bitet Matlab einige Funktionen, die bestimmte Munipulationen mit Matrizen beziehungsweise mit Arrays ersetzen.

$>>flipud(A)$ %*Matrix von unten nach oben drehen*

$ans =$

$$\begin{matrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{matrix}$$

Oder

```
>>fliplr(A) %Matrix von links nach rechts drehen
ans =
     3   2   1
     6   5   4
     9   8   7
```

Mit dem Befehl *rot*90 kann man eine Matrix um 90 Grad nach links drehen:

```
>>rot90(A)
ans =
     3   6   9
     2   5   8
     1   4   7
>>rot90(A,2) %Matrix um 180 Grad nach links drehen
ans =
     9   8   7
     6   5   4
     3   2   1
```

2 TEIL II

2.1 Datentypen und Datenstrukturen

Standardmäßig interpretiert Matlab eine Zahl als *double*, das heißt als eine Gleitkommazahl von 64 Bit Länge. Gleitkommazahl oder Fleißkommazahl ist eine digitale Näherung für eine reelle Zahl die in Computer verwendet wird. Die Zahl wird dabei in eine Mantisse und einem Exponenten gespeichert wodurch ein größerer Wertebereich als bei Festkommadarstellung abgedeckt werden kann. Das heißt, man kann eine Zahl a durch zwei Zahlen m und b darstellen, so dass $a = m \cdot b^e$ gilt, wobei m eine Mantisse ist und b ist eine beliebige Basis. Die Mantisse m enthält die Ziffern der Gleitkommazahl. Speichert man mehr Ziffern ab, so erhöht sich die Genauigkeit.

Matlab kennt im Prinzip nur eine einzige Zahlenart. Sie erlaubt die Darstellung reeller und komplexer Zahlen. Reelle Zahlen mit einem Betrag aus dem Bereich von 10^{-308} bis 10^{308} werden mit einer Genauigkeit von etwa 16 Dezimalstellen dargestellt. In Matlabprogramm können wir diese Zahlen folgenderweise schreiben:
1) als ganze Dezimalzahlen mit oder ohne Vorzeichen $-1,\ 30,\ -400, ...$
2) als Dezimalbruch mit dem Dezimalpunkt $1.5,\ -2.0,\ 0.25,\ -3.1457, ...$
3) als Gleitkommazahl, wobei die Mantisse eine Zahl der Form 1) oder 2) ist und der Exponent eine ganze Zahl der Form 1) sein muss.

Mantisse und Exponent sind mittels eines e zu verknüpfen: $1.0e+3,\ .1e-5,\ 5.4000e+11$.
Zum Beispiel,
$20^6 = 64000000$,
$20^7 = 1.2800e+09 = 1.28 \cdot 10^9$, oder
$1/234576583 = 4.2630e-09 = 4.263 \cdot 10^{-9}$.

Es gibt aber weitere Datentypen wie

- *uint*8 8 Bit Länge ganze Zahl im Bereich von 0 bis 2^8
- *int*8 8 Bit Länge ganze Zahl im Bereich von -2^7 bis $2^7 - 1$
- *uint*16 16 Bit Länge ganze Zahl im Bereich von 0 bis 2^{16}
- *int*16 16 Bit Länge ganze Zahl im Bereich von -2^{15} bis $2^{15} - 1$
- *uint*32 32 Bit Länge ganze Zahl im Berich von 0 bis 2^{32}
- *int*32 32 Bit Länge ganze Zahl im Beriech von -2^{31} bis $2^{31} - 1$
- *uint*64 64 Bit Länge ganze Zahl im Beriech von 0 bis 2^{64}

- $int64$ 64 Bit Länge ganze Zahl im Bereich von -2^{63} bis $2^{63}-1$

Mit dem Befehl *intmax* oder *intmin* kann man obere und untere Grenze des entsprechendes Bereichs abfragen:

$$
\begin{aligned}
&>> intmax('\ uint8\ ')\\
&ans =\\
&\qquad 255\\
&>> intmin('\ int8\ ')\\
&ans =\\
&\qquad -128
\end{aligned}
$$

Beispiel 2.1.1.

$$>>\ A = randn(3,4);$$

Mit dem Befehl *whos* kann man nachschauen welche Klasse von Datentyp hat diese Matrix. Standard ist double. Durch Eingeben von $>> N = int8(A)$ bekommt man eine Matrix mit ganzen Zahlen und Datentyp $int8$.

Matlab hat noch weiteren Fließkommadatentyp als *single* mit geringerem Speicheraufwand, *char* als Strings, also Zeichenketten und *logical* als Booleschen Datentyp. Weitere Details zu den Datentypen kann man mit Hilfe des Befehls *helpwin* finden.

2.2 Zeichenketten (Strings)

Matlabs wahre Macht liegt in seiner Fähigkeit Zahlen berechnet zu können. Jedoch, braucht man manchmal in einem Programm einen Text zu bearbeiten. In Matlab wird Text als Zeichenkette bezeichnet. Der Wert einer Variablen vom Datentyp *char* steht zwischen Hochkommata. Die einfachste Methode einen String zu erzeugen ist demnach:

$$
\begin{aligned}
&>> s1 =' Hallo'\\
&s1 =\\
&\quad Hallo
\end{aligned}
$$

Um Strings aneinander zu hängen, kann man sie einfach in ein Array schreiben:

$$
\begin{aligned}
&>> text1 = ['Hallo',\ 'Leute']\\
&text1 =\\
&\quad Hallo\\
&\quad Leute
\end{aligned}
$$

Man beachte, dass Strings in Matlab nichts anderes als Matrizen beziehungsweise Arrays vom Datentyp *char* sind. Daher liefert

$$
\begin{aligned}
&>> text2 = ['Hallo';\ 'Leute.']\\
&text2 =\\
&\quad Error\ using\ vertcat\\
&\quad Dimensions\ of\ matrices\ being\ concatenated\ are\ not\ consistent.
\end{aligned}
$$

eine Fehlermeldung, denn die beiden Zeilen sind nicht gleich lang. In diesem Fall sollte man stattdessen

$$
\begin{aligned}
&>> text2 = char('Hallo',\ 'Leute.')\\
&text2 =\\
&\quad Hallo\\
&\quad Leute.
\end{aligned}
$$

verwenden, um automatisch Leerzeichen hinzuzufügen. Besonders wichtig sind Strings für die Ausgabe. Dazu dienen die Funktionen *disp*, $sprintf$ und $fprintf$. Um Zahlen mit *disp* darzustellen, muss man erst mit der Funktion $num2str$ in Strings umgewandelt werden:

$$
\begin{aligned}
&>> x = [12\ 23\ 34\ 45\ 56\ 67];\\
&>> disp(['Die\ Ergebnisse\ sind:\ '\ num2str(x)])\\
&\quad Die\ regebnisse\ sind:\ 12\ 23\ 34\ 45\ 56\ 67
\end{aligned}
$$

Die entsprechende Anweisung mit $fprintf$ sieht wie folgt aus (dabei wird für jeden Wert in x eine Zeile geschrieben):

$$
\begin{aligned}
&>> fprintf('Das\ Ergebniss\ ist:\ \%g\backslash n\ ', x)\\
&\quad Das\ Ergebniss\ ist:\ 12\\
&\quad Das\ Ergebniss\ ist:\ 23\\
&\quad Das\ Ergebniss\ ist:\ 34\\
&\quad Das\ Ergebniss\ ist:\ 45\\
&\quad Das\ Ergebniss\ ist:\ 56\\
&\quad Das\ Ergebniss\ ist:\ 67
\end{aligned}
$$

Hier ist $\%g$ ein Platzhalter, für den die Einträge aus x eingesetzt werden. Dabei ist aber zu beachten, dass $fprintf$ für jeden Platzhalter nur einen Eintrag aus x einsetzt. Die Verwendung von $\backslash n$ sorgt für einen Zeilenumbruch. Der ganz änlich funktionierende Befehl $sprintf$ erzeugt nur einen String und keine Ausgabe. Diese muss von *disp* übernommen werden.

2.3 Cell Arrays

Vektoren und Matrizen sind Arrays deren Elemente alle denselben Typ haben. Mehr Flexibilität bieten Cell Arrays. Der Typ jedes Elements kann darin verschieden sein. Wir betrachten ein einfaches Beispiel:

$$
\begin{aligned}
&>>A = \{'Welche'; \ 'Eintraege \ koennen'; \ 'in \ A \ sein?'\} \\
&\ \ A = \\
&\qquad 'Welche' \\
&\qquad 'Antraege \ koennen' \\
&\qquad 'in \ A \ sein?' \\
&>>size(A) \\
&ans = \\
&\qquad 3 \quad 1
\end{aligned}
$$

Hier hat das Cell-Array A drei Zeilen und eine Spalte. Jedoch, jedes Element von Cell enthält String mit verschiedenen Länge. Man beachte, dass geschweifte Klammer zum Erzeugen der Cell Arrays benutzt werden sollen. Mit runden Klamern greift man auf die Cells (Zellen) selbst.

$$
\begin{aligned}
&>>A(2:3) \\
&ans = \\
&\qquad 'Antraege \ koennen' \\
&\qquad 'in \ A \ sein?'
\end{aligned}
$$

Das Resultat ist hier immer noch ein Cell Array. Also, $A(Indizes)$ gibt wieder eine Zelle zurück, aber nicht den Inhalt von dieser Zelle. Um den Inhalt von Cell Arrays abzurufen benutzt man geschweifte Klammer:

$$
\begin{aligned}
&>>A\{3\} \\
&ans = \\
&\qquad in \ A \ sein?
\end{aligned}
$$

Man kann auch eine Liste von Cells mit Hilfe Matlabfunktion *deal* erstellen:

$$
\begin{aligned}
&>>[a, b, c] = deal(A\{:\}) \\
&\ \ a = \\
&\quad Welche \\
&\ \ b = \\
&\quad Antraege \ koennen
\end{aligned}
$$

$c =$

in A sein?

Die gleiche Ausgabe bekommt man auch durch:

$$>> [a, b, c] = deal(A\{1\},\ A\{2\},\ A\{3\});$$

Wir betrachten Beispiele mit Elementen von verschiedenen Typen in einem Cell Array:

```
>>B = {[1,  2,  3], 'r';  cell(2,2),  [4 5;  6 7;  8 9]}
  B =
        [1x3 double]    'r'
        {2x2 cell}      [3x2 double]
>>B(1,1)
ans =
        [1x3 double]
>>B{1,1}
ans =
        1   2   3
```

Als Nächstes versuchen wir ein Cell Array grafischisch darzustellen:

```
>>C = {[3 4 5;  6 7 8],    'g*'};
>>plot(C{:})
```

In diesem Fall *plot*(C) wird nicht funktionieren, da *plot* Funktion den Zugriff auf den Inhalt von Cell Array nicht hat.

2.4 Strukturen

Eine Datenstruktur, die sich, wie der Name schon sagt, zum Strukturieren von Daten eignet, ist die Struktur (*struct*). In ihr können verschiedene Daten, die zu einem Objekt gehören, zusammengefasst werden. In dem Beispiel

```
>>S.name ='LevinMaass';
>>S.punkte = 84;
>>S.note ='2+';
```

```
>>S
  S =
    name :  'Levin Mass'
    punkte :  84
    note :  '2+'
```

wird eine skalare Strukture mit drei Felder erzeugt. So wie alles andere in der MATLAB-Umgebung sind Strukturen Arrays. In diesem Fall ist jedes Element eine Strukture mit mehreren Feldern. Die Felder können zu bestimmten Zeitpunkt gegeben werden:

```
>>S(2).name ='Toni Mueller';
>>S(2).punkte = 92;
>>S(2).note ='1-';
```

Oder

```
>>S(2) = struct('name', 'Toni', 'punkte', 92, 'note','1-')
  S =
    1 × 2 struct array with fields :
    name
    punkte
    note
```

Der Code

```
>>S.punkte
ans =
          84
ans =
          92
```

kann man auch anders angeben:

```
>>S(1).punkte,   S(2).punkte
```

Mit den eckigen Klammern kann man diesen Ausdrück in ein Array einschließen:

```
>>Punkte = [S.punkte]
Punkte =
          84  92
```

$$>> durchschnitts\_punkte = sum(Punkte)/length(Punkte)$$
$$durchschnitts\_punkte =$$
$$88$$

Mit der Funktion *char* kann man eine Liste mit Namen erstellen:

$$>> Namen = char(S.name)$$
$$Namen =$$
$$Levin\ Maass$$
$$Toni\ Mueller$$

Genauso kann man mit geschweiften Klammern ein Cell-Array mit Namenfelder zu erstellen:

$$>> Namen = \{S.name\}$$
$$Namen =$$
$$'LevinMaass' \quad 'ToniMueller'$$

2.5 Logische- und Vergleichsoperationen

Zusätzlich zu den traditionellen mathematischen Operationen, bietet Matlab Vergleichsoperationen und logische Operationen. Ziel dieser Operationen ist eine Antwort *true* oder $false$ auf eine Frage zu geben. Als Eingabe zu allen Vergleichs- und logischen Ausdrücken werden alle von Null verschiedene Werte als *true* betrachten, sowie alle Werte, die gleich Null sind, werden für $false$ gehalten.

2.6 Vergleichsoperatoren

Matlab Vergleichsoperatoren umfassen alle gewöhnliche Vregleichszeichen:

- $<$ Kleiner als
- $<=$ Kleiner gleich
- $>$ Größer als
- $>=$ Größer gleich
- $==$ Gleich (nicht zu verwechseln mit $=$)
- $\sim=$ Ungleich

An den folgenden Beispielen kann man die Nutzung der Vergleichsoperatoren in Matlab sehen.

$$
\begin{aligned}
&>>A = 1:10,\ B = A + 1\\
&\quad A =\\
&\qquad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10\\
&\quad B =\\
&\qquad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11\\
&>>C = B > 5\\
&\quad C =\\
&\qquad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1
\end{aligned}
$$

Hier ist C ein logischer Vektor mit den Elementen, die Matlab als $false$ eine Null und $true$ als eine Eins interpretiert.

$$
\begin{aligned}
&>>x = (1 + 10 - 33) == (1 + 8 - 31)\ \%Gleich?\\
&\quad x =\\
&\qquad 1\\
&>>x = (1 + 10 - 33) \sim= (1 + 8 - 31)\ \%Ungleich?\\
&\quad x =\\
&\qquad 0
\end{aligned}
$$

Es ist möglich Vergleichsoperationen mit mathematischen Operationen zu kombinieren:

$$
\begin{aligned}
&>>C = B - (A < 4)\\
&\quad C =\\
&\qquad 1 \quad 2 \quad 3 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11
\end{aligned}
$$

In diesem Fall ist C ein numerischer Vektor.

2.7 Logische Operatoren

Logische Operatoren ermöglichen Kombinationen beziehungsweise Verneien bei Vergleichsoperationen durchzuführen. Matlab bietet folgende logische Operationen:

- & Elementweise UND bei Arrays
- | Elementweise ODER bei Arrays
- $\sim$ NICHT
- && Skalares UND
- || Skalares ODER

Die nächsten Beispiele veranschaulichen die Verwendung von logischen Operationen.

```
>>A = 1 : 10;
>>B = 1 + A;
>>C =∼ (A > 5)
  C =
        1   1   1   1   1   0   0   0   0   0
```

Dieser Code verneint den Ausdrück $A > 5$. Der Code

```
>>C = (A > 2) & (A < 7)
  C =
        0   0   1   1   1   1   0   0   0   0
```

liefert eine Eins für $A > 2$ und $A < 7$. Der Code

```
>>C = A < 3 | A > 6
  C =
        1   1   0   0   0   0   1   1   1   1
```

liefert einen logischen Vektor C, wobei Elemente von A kleiner als 3 oder größer als 6 sind.

```
>>a = 1; b = pi
>>a == 1 || b ∼= 1
ans =
        1
>>a > 1 && b == pi
ans =
        0
```

In den letzten zwei Beispielen sind a und b skalare Elemente. Weiter, betrachten wir noch einige Funktionen, die man an logischen Arrays anwenden kann.

```
>>A = rand(5)
  A =
    0.8147   0.0975   0.1576   0.1419   0.6557
    0.9058   0.2785   0.9706   0.4218   0.0357
    0.1270   0.5469   0.9572   0.9157   0.8491
    0.9134   0.9575   0.4854   0.7922   0.9340
    0.6324   0.9649   0.8003   0.9595   0.6787
```

Jetzt erzeugen wir ein logisches Array:

$$>>B = (A > 0.5)$$
$$B =$$
$$\begin{array}{ccccc} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{array}$$

Mit Funktionen *any* und *all* kann man entscheiden, ob ein oder alle Elemente die Bedingung erfüllen. Das Ergebniss ist vom Datentyp logical. Weiterhin, können wir alle Elemente in A, die größer als 0.5 ansehen:

$$>>A(B)$$
$$ans =$$
$$\begin{array}{c} 0.8147 \\ 0.9058 \\ 0.9134 \\ 0.6324 \\ 0.5469 \\ 0.9575 \\ 0.9649 \\ 0.9706 \\ 0.9572 \\ 0.8003 \\ 0.9157 \\ 0.7922 \\ 0.9595 \\ 0.6557 \\ 0.8491 \\ 0.9340 \\ 0.6787 \end{array}$$

Anzahl der Elemente mit $A > 0.5$ erhält man durch

$$>>sum(B(:)) \; \%das\ gleiche\ wie\ sum(sum(B))$$
$$ans =$$
$$17$$

2.8 Lineare Gleichungssysteme

Ein Problem, das am häufigsten in der linearen Algebra auftritt, ist die Lösung eines Gleichungssystem. Als Beispiel betrachten wir ein lineares Gleichungssystem der Form $Ax = y$:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 366 \\ 804 \\ 351 \end{pmatrix},$$

wobei das Multiplikationssymbol $(\cdot)$ in Matrixsinne definiert ist, so wie es schon am ersten Tag erwähnt wurde. Die Existenz der Lösungen dieses Gleichungssystems ist eine grundlegende Frage in der linearen Algebra. Außerdem, wenn eine Lösung existiert, es gibt zahlreiche Ansätze (Aproximationen) zur Lösung eines lineares Gleichungssystem: Gaußsche Elimination, LR-Verfahren oder auch eine direkte Lösung durch benutzen von A^{-1}. Klar, es sind natürlich mehrere Fragen werden hier gestellt, aber wir wollen nur zeigen wie Matlab solche Probleme wie oben lösen kann. Für die Lösung dieses Problems geben wir zuerst per Hand in Matlab A und y ein:

$$>> A = [1\ 2\ 3;\ 4\ 5\ 6;\ 7\ 8\ 0];$$
$$>> y = [366;\ 804;\ 351];$$

Wenn man sich schon etwas in der linearen Algebra auskennt, weis man, dass diese Aufgabe eine eindeutige Lösung hat, falls der Rang von A und der Rang von der erweiterten Matrix $[Ay]$ gleich sind.

(**Zur Erinnerung:**) Rang einer Matrix heißt maximale Anzahl linear unabhänigen Zeilen (Spalten).

$$>> rank(A)$$
$$ans = 3$$
$$>> rank([Ay])$$
$$ans = 3$$

Als Alternative kann man natürlich die Konditionszahl von einer Matrix überprüfen. Wenn diese Zahl nicht übermäßig groß ist, dann hat A eine Inverse mit guten numerischen Eigenschaften.

Das Gute in Matlab ist, dass wir es alles mit Hilfe von Matlab-Funktionen *rank* und *cond* überprüfen können:

$$>> cond(A)$$
$$ans = 35.1060$$

Jetzt bietet Matlab zwei Möglichkeiten um die Lösung von $Ax = y$ zu finden.

1) $x = A^{-1}y$ (ungünstig)

$$\begin{aligned} >> x =& inv(A) * y \\ x =& 25.0000 \\ & 22.0000 \\ & 99.0000 \end{aligned}$$

Hier ist $inv(A)$ eine Matlab-Funktion, die A^{-1} ausrechnet. Eine bessere Möglichkeit die Lösung zu finden ist die Matrix-links-Division:
2)

$$\begin{aligned} >> x =& A \setminus y \\ x =& 25.0000 \\ & 22.0000 \\ & 99.0000 \end{aligned}$$

Bemerkung: Wenn A ist eine $n \times n$-Matrix und y ist ein Spaltenvektor mit n-Elementen, dann ist $x = A \setminus y$ die Lösung der Gleichung $Ax = y$, die durch das Gaußsches Eliminationsverfahren ausgerechnet wird.

Weiterhin, falls die Anzahl der Gleichungen und die Anzahl der Unbekannten verschieden sind, so existiert die eindeutige Lösung nicht. Jedoch, mit einigen Einschränkungen kann normalerweise eine praktische Lösung gefunden werden. In Matlab, wenn $Rang(A) = min(m, n)$ mit m-Zeilen und n-Spalten, und die Anzahl der Gleichungen größer als Anzahl der Unbekanten ($m > n$), dann findet der Divisionsoperator / oder \ automatisch eine Lösung, die eine Lösung der kleinsten Quadrate heißt.

Beispiel 2.8.1.

$$\begin{aligned} >> & A = [1\ 2\ 3;\ 4\ 5\ 6;\ 7\ 8\ 0;\ 2\ 5\ 8]\ \%\ \textit{vier Gleichungen und drei Unbekannte} \\ >> y =& [366\ 804\ 351\ 514]'; \\ >> x =& A \setminus y\ \ \%\ \textit{Loesung der kleinsten Quadrate} \\ x =& 247.9818 \\ & -173.1091 \\ & 114.9273 \end{aligned}$$

Außerdem, Matlab bittet hilfreiche numerische Matrix-Funktionen zur Lösung linearen Gleichungssysteme:

- $det(A)$ - Determinante
- $eig(A)$ - Eigenwerte
- $[V, D] = eig(A)$ - Matrix mit Eigenvektoren und eine Diagonalmatrix enthalten Eigenwerte

- $norm(A, type)$ - Matrix- und Vektornorm
- $rank(A)$ - Rang einer Matrix
- $trace(A)$ - Summe der Diagonalelemente einer Matrix

2.9 Matrixnorm

Die Matlab-Funktion *norm* bittet verschiede Typen einer Matrixnorm:

- $\|A\|_\infty = \max_{i=1,\ldots,n} \sum_{j=1}^{n} |a_{ij}|$ - Zeilensummennorm

$$>> norm(A, inf)$$
$$ans = 15$$

- $\|A\|_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{m} |a_{ij}|$ - Spaltensummennorm

$$>> norm(A, 1)$$
$$ans = 20$$

- $\|A\|_2 = \sqrt{\lambda_{max}(A^H A))}$ - Spektralnorm

$$>> norm(A)$$
$$ans = 15.5501$$

- $\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2}$ - Frobeniusnorm

$$>> norm(A,' fro')$$
$$ans = 17.2337$$

2.10 Dünnbesetzte Matrizen

Wenn man mit Matrizen rechnen möchte, die nur wenige Eintäge ($< 5\%$) ungleich Null haben, bietet es sich an, den Datentyp einer Sparsematrix zu benutzen. In diesem Datentyp werden nur die Nicht-Nullelemente zusammen mit ihren Indizes gespeichert. Solche Matrizen entstehen zum Beispiel bei der numerischen Berechnung von Lösungen partieller Differentialgleichungen. Die Speicherung in einer Sparsematrix bedeutet zum einen eine große Einsparung an Hauptspeicher, zum anderen werden so Matrixoperationen nur für die Nicht-Nullelemente durchgeführt und sind damit wesentlich schneller. Außerdem

stellt Matlab effiziente Algorithmen zur Lösung linearer Gleichungssysteme mit Sparsematrizen zur Verfügung. In Matlab werden Sparsematrizen genauso wie alle anderen vollbesetzte Matrizen in Variablen gespseichert. Außerdem laufen die meisten Berechnungen für dünnbesetzten Matrizen mit gleicher Syntax wie für vollbesetzte Matrizen. Im Allgemeinen, Operationen auf vollbesetzten Matrizen erzeugen vollbesetzte Matrizen und Operationen auf dünnbesetzten Matrizen erzeugen dünnbesetze Matrizen. Insbesondere, Operationen auf gemischten Matrizen erzeugen in der Regel dünnbesetze Matrizen. Eine dünnbesetze Matrix wird in Matlab durch Funktion *sparse* erzeugt:

$$>> S = sparse(1:5,\ 5:-1:1,\ rand(5,1)$$

$S =$

$(5,1)$	0.9133
$(4,2)$	0.2290
$(3,3)$	0.0838
$(2,4)$	0.4505
$(1,5)$	0.7482

Hier wird in $S(i,j)$ zur jeder i-te Zeile und j-te Spalte ein Wert zugeordnet. Mit der Funktion $full$ kann man die ganze Matrix ansehen

$>> full(S)$

$ans =$

0	0	0	0	0.7482
0	0	0	0.4505	0
0	0	0.0838	0	0
0	0.2290	0	0	0
0.9133	0	0	0	0

Man kann eine Identitätsmatrix mit der Funktion *eye* erstellen:

$>> I = sparse(eye(5))$

$I =$

$(1,1)$	1
$(2,2)$	1
$(3,3)$	1
$(4,4)$	1
$(5,5)$	1

$$>> full(I)$$
$$ans =$$
$$\begin{matrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{matrix}$$

In dem folgenden Beispiel sieht man den Unterschied vom Speicherplatz zwischen den vollbesetzten und dünnbesetzen Matrizen:

$$>> I\_voll = eye(100);$$
$$>> I\_duenn = sparse(I\_voll);$$
$$>> whos$$

$Name$	$Size$	$Bytes$	$Class$	$Attributes$
I_duenn	100×100	2408	$double$	$sparse$
I_voll	100×100	80000	$double$	

Der Speicherverbrauch einer Sparsematrix in Matlab ist wie folgt:

$$\underbrace{nnz \times 8Byte}_{Eintraege} + \underbrace{nnz \times 4Byte}_{Position\ jeder\ Spalte} + \underbrace{cols \times 4Byte}_{erster\ nicht\ Null\ Eintrag\ jeder\ Spalte} + \underbrace{4Byte}_{Datenstruktur}$$

Die Datenstruktur der Sparsematrizen ist demnach für dünnbesetze Matrizen optimiert und sollte keinesfalls blindlings für alle Matrizen benutzt werden.

Es gibt spizielle Funktionen zum Erzeugen von Sparsematrizen, dies sind *spones*, *spdiags*, *sprand* und *speye* (sehe *help spones*, ... in Matlab). In der Numerik wird Funktion *spdiags* häufig benutzt um Sparsematrizen durch Angabe der Verktoren auf den Diagonalen zu erzeugen:

$$>> n = 10;\ e = ones(n, 1);$$
$$>> D = spdiags([e,\ -2 * e,\ e], -1 : 1,\ n,\ n);$$
$$>> spy(D)$$

Mit dem Befehl *spy* visualisiert man eine Sparsematrix. Der Zugriff auf Elemente einer Sparsematrix S erfolgt wie üblich mit $S(i, j)$ oder $S(k)$. Das Ergebniss eines Zugriffs auf einen Bereich ist wieder eine Sparsematrix.

$$>> S(:, 1 : 3);\ \%Das\ Ergebniss\ ist\ wieder\ eine\ Sparsematrix.$$

2.11 Skripte

Für leichte Aufgaben in Matlab braucht man nur seine Anforderungen im Command Window hinter dem Promt stellen und das funktioniert schnell und effektiv. Jedoch, wenn die Anzahl der Befehle steigt oder man möchte einige Werte von Variablen ändern wird es langsam aufwendig und langweilig das selbe hinter dem Matlab-Promt zu tippen. Matlab bietet eine logische Lösung zu diesem Problem und zwar, dass man alle Angaben und Befehle in einem einfachen Text-Datei aufschreibt. Danach befehlet man Matlab diese Datei zu öffnen und alle Angaben und Befehle auszuführen, genauso wie man es davor hinter dem Promt getippt hat. Solche Dateien heißen **Skript-Dateien** oder **m-Dateien**. Um ein Skript zu erzeugen, klickt man "New Script" oder "New" auf dem Matlab-Desktop Symbolleiste. Danach erscheint ein Text-Editor wo man Angaben und Matlab-Befehle schreiben kann. Eine Script-Datei kann folgende Form haben:

```
%Skript m-Datei Beispiel1
 weisspapier =11;
 marker = 15;
 tacker = 10;
 anzahl = weisspapier + marker + tacker
 geldbetrag = weisspapier*5.39 + marker*2.69 + tacker*3.95
 durchschnitt_Preis = geldbetrag/anzahl
```

Diese m-Datei muss als $Beispiel1.m$ gespeichert werden. Nach der Abspeicherung erscheint diese Datei in dem Current Folder (aktuelle Dateien) Fenster. Nach der Eingabe im Command Window $Beispiel1$ (ohne Endung .m) erhält man

$$
\begin{aligned}
&>> Beispiel1\\
&anzahl =\\
&\qquad 36\\
&geldbetrag =\\
&\qquad 139.1400\\
&durchschnitt_Preis =\\
&\qquad 3.8650
\end{aligned}
$$

Alle Variablen von einem Skript erscheinen sofort im Workspace Fenster. Aber als Ausgabe im Kommand-Fenster haben wir nur drei Variablen bekommen, und zwar, nur die Variablen, die ohne Semikolon im Skript stehen. Man könnte diese Variablen auch mit Semikolon aufschreiben und mit dem Befehl $disp(anzahl)$, $disp(geldbetrag)$, ... und so weiter, die gewünschte Variablen herausbekommen. Die rote und orange Linien, die links im Skript erscheinen, stehen für Fehlermeldung. Durch Klicken auf diesen Linien kann man herausfinden, wo man Fehler gemacht hat.

Bemerkung: Man kann ein Skript auch durch Klicken von "Run" auf der Editor-Leiste aufrufen.

Ein Kommentar, den man in der ersten Zeile eines Skripts hinterlässt, erscheint in dem Matlab-Fenster Details, unten links. Für lange Kommentare, die man nicht in einer Zeile unterbringen kann, braucht man % Zeichen für die weiteren Zeilen.

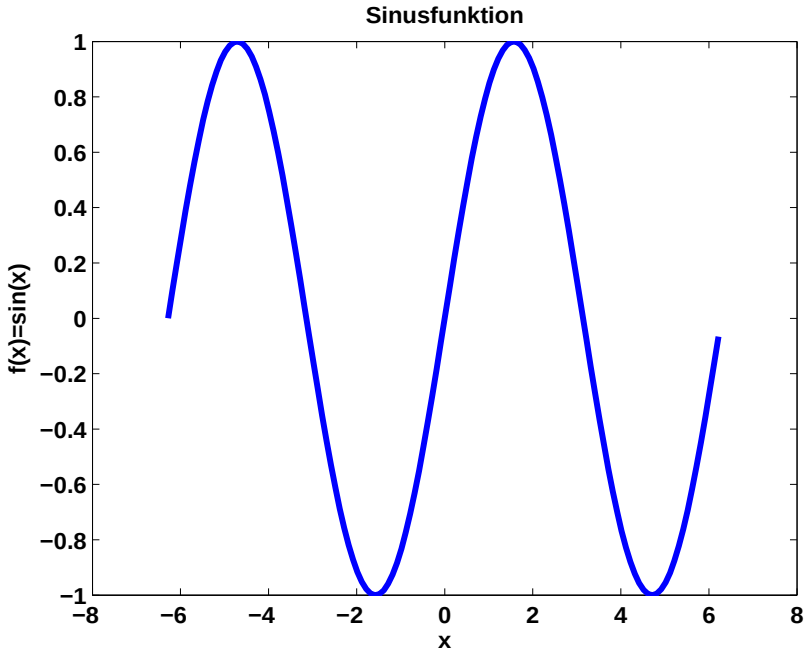
Aufgabe1: *Schreibe die Befehle zum Plotten einer Sinusfunktion, sowie zur Beschriftung der Achsen und eines Titels in ein Skript und führe es aus.*

Lösung: m-Datei $Aufgabe1.m$

```
%Sinusfunktion plotten
x = -2*pi:0.1:2*pi;
y = sin(x);
plot(x,y);
titel(' Sinusfunktion ')
xlabel(' x ')
ylabel(' f(x)=sin(x) ')
```

Der Aufruf dieses Skriptes liefert

>> A



3 TEIL III

3.1 Kontrollstrukturen

In einem Programm wird in weiten Teilen nichts anderes beschrieben als die Berechnung von Werten, die dann einem Namen zugeordnet sind. Dabei treten bestimmte Strukturen auf. Manchmal wiederholen sich dieselben oder ähnliche Anweisungen und man führt sie wieder und wieder aus. Manchmal sollen solche Anweisungen nur unter bestimmten Bedingungen durchgeführt werden. Wir bezeichnen diese Strukturen als *Kontrollstrukturen*. Die Anweisungen, die im Hinblick auf eine solche Struktur zusammengefasst werden können (alle Anweisungen, die nur dann ausgeführt werden sollen, falls eine bestimmte Bedingung erfüllt ist), bezeichnen wir als Anweisungsblock. Zusammen mit der Beschreibung der Kontrollstruktur bilden diese Anweisungsblöcke eine Verbundeinweisung. Solange man nur mit skalaren Größen arbeitet, unterscheidet sich die in Matlab verfügbaren Mitteln zu Berechnung der Kontrollstrukturen kaum von den anderen Programmiersprachen.
Matlab bietet die Standardkontrollstrukturen: Verzweigung und Schleife.

3.2 *For* Schleife

Die for-Schleifen erlauben einer Gruppe von Anweisungen für festgesetzte Zeit wiederholt zu werden. Zu dem Schleifendurchlauf wird mit Hilfe einer Schleifenvariablen ein bestimmter Wert der Schleifenindex zugeordnet, der in der Schleife abgefragt werden kann. Die allgemeine Syntax von for-Schleifen lautet wie folgt:

$$\begin{aligned} &>> for\ n = Array \\ &\qquad (Anweisungen) \\ &\quad end \end{aligned}$$

Die Anweisungen zwischen for und end werden einmal für jede Spalte in Array ausgeführt. Bei jeder Iteration wird n in die nächste Spalte zugewiesen, das heißt, in der N-te Zeit hat man einen Schleifendurchlauf $n = Array(:, N)$. Zum Beispiel, in dem Code

$$\begin{aligned} &>> for\ n = 1:10 \\ &\qquad x(n) = sin(n * pi/10); \\ &\quad end \end{aligned}$$

```
>>x
  x =
    Columns 1 through 8
    0.3090  0.5878  0.8090  0.9511  1.0000  0.9511  0.8090  0.5878
    Columns 9 through 10
    0.3090  0.0000
```

wird die erste Anweisung als "Berechne alle Anweisungen für n gleich 1, 2, ..., 10" interpretiert. Nach dem $n = 10$ wird die for-Schleife beendet. Wird die Schrittweite nicht angegeben, erfolgt also die Angabe des Laufbereichs in der Form: Anfangswert : Endwert. So wird implizit die Schrittweite 1 verwendet. Anfangs- und Endwert sowie die Schrittweite werden bei for-Schleife vor dem ersten Schleifendurchlauf bestimmt und koennen dabei nicht mehr verändert werden. Man kann auch in der Schleife die Anzahl der Iterationen nachzählen:

```
>>i = 0; %Zaehlen Schleifeniterationen
>>for n = 1 : 10
   i = i + 1;
  x(n) = sin(n * pi/10);
   end
>>i
   i =
        10
>>x
  x =
    Columns 1 through 8
    0.3090  0.5878  0.8090  0.9511  1.0000  0.9511  0.8090  0.5878
    Columns 9 through 10
    0.3090  0.0000
```

Man könnte die Anweisungen auch in einer Zeile schreiben, dann muss man nach jede Anweisung Semikolon setzen.
In dem Code

```
>>i = 0; %Zaehlen Schleifeniterationen
>>for n = (1 : 10)'
   i = i + 1;
  x(n) = sin(n * pi/10);
   end
```

```
>>i
  i =
       1
>>x
  x =
  Columns 1 through 8
  0.3090  0.5878  0.8090  0.9511  1.0000  0.9511  0.8090  0.5878
  Columns 9 through 10
  0.3090  0.0000
```

hat die for-Schleife nur einen Durchlauf ausgeführt. Der Ausdrück $(1:10)'$ ist ein Spaltenvektor, und n wird als gesammtes Array nur durch eine Iteration ausgeführt. Da es keine zusätzlichen Spalten auf der rechten Seite von Array gibt, wird die for-Schleife beendet. In dem Code

```
>>i = 1;
>>for x = rand(4, 5)
    y(i) = sum(x);
    end
>>y
  y =
       3.0892
```

haben wir die Summen von x nur für eine Iteration ausgerechnet. Um alle Summen von x-Array zu erhalten, kann man den folgenden Ausdrück benutzen:

```
>>i = 1;
>>for x = rand(4, 5)
    y(i) = sum(x);
    i = i + 1;
    end
>>y
  y =
    2.4746   2.5718   1.5645   1.2437   1.9966
```

Bis jetzt haben wir nur die Vektoren angeschaut. Natürlich kann man die for-Schleife für Matrizen anwenden. Also, wir verschachteln diese Schleife:

$$
\begin{array}{l}
>> for\ n = 1:5 \\
\quad >>\ for\ m = 5:-1:1 \\
\quad A(n,m) = n \wedge 2 + m \wedge 2; \\
\quad\quad end \\
\quad disp(n) \\
\quad end \\
\quad\quad 1 \\
\quad\quad 2 \\
\quad\quad 3 \\
\quad\quad 4 \\
\quad\quad 5
\end{array}
$$

$$
\begin{array}{l}
>> A \\
A =
\end{array}
$$

$$
\begin{array}{ccccc}
2 & 5 & 10 & 17 & 26 \\
5 & 8 & 13 & 20 & 29 \\
10 & 13 & 18 & 25 & 34 \\
17 & 20 & 25 & 32 & 41 \\
26 & 29 & 34 & 41 & 50
\end{array}
$$

Hier kann man die m und n beliebig tauschen. Noch ein Beispiel:

```
clear
m=3; n=m;
for i=1:m
   for j=1:n
       v(j)=1+j;
   end
      w(i)=v(i)/100;
end
A=[v; w]
```

Nach allen diesen Beispielen heißt es lange nicht, dass das Programmieren mit for-Schleifen in Matlab sehr effektiv ist. Früher, bedeuteten for-Schleifen eine schlechte Pogrammierungspraxis, sobald ein äquivalenter Ansatz existierte. Dieser Ansatz heißt **vektorisierte Lösung** und meistens ist um Größenordnungen schneller als skalare An-

sätze, die wir oben gezeigt haben. Zum Beispiel, in dem Code

$>>n = 1 : 10;$
$>>x = sin(n * pi/10)$
$x =$
$Columns\ 1\ through\ 8$
0.3090 0.5878 0.8090 0.9511 1.0000 0.9511 0.8090 0.5878
$Columns\ 9\ through\ 10$
0.3090 0.0000

kann man nur in zwei Zeilen die Sinusfunktion in 10 Punkten ausrechnen. Vor allem, vektorisierte Lösung ist schneller beim Ausrechen in Matlab und einfacher zu verstehen. Der Code mit verschachtelten for-Schleifen kann man auch umschreiben:

$>>[N, M] = meshgrid(1 : 5, 1 : 5);$
$>>A = N. \wedge 2 + M. \wedge 2$
$A =$

2	5	10	17	26
5	8	13	20	29
10	13	18	25	34
17	20	25	32	41
26	29	34	41	50

Aufgabe2. *Bestimme für eine zufällige $n \times n$-Matrix die Anzahl der Einträge im Intervall* $[0.3,\ 0.7]$.

Lösung: (mit for-Schleifen) m-Datei $Aufgabe2.m$

```
%Anzahl der Eintraege in  A
n = 10;
A = rand(n);
anzahl = 0;  %Initialisierung
  for  i = 1:n
     for  j = 1:n
       anzahl = anzahl + (A(i,j)>=0.3 && A(i,j)<=0.7);
     end
   end
 disp(anzahl)
```

Lösung: (ohne for-Schleifen) m-Datei $Aufgabe2\_vek$

```
%Anzahl der Eintraege in A
n = 10;
A = rand(n);
```

```
anzahl = 0;
anzahl = anzahl + sum(sum(A>=0.3 & A<=0.7));
```

Vektorisierung wird häufig benutzt, um den Code übersichtlicher zu machen. Manchmal kann man auch die Ausführungsgeschwindigkeit erhöhen. Im folgenden Beispiel wird Geschwindigkeit von einer Lösung mit *for*-Schleifen und vektorisierten Lösung gemessen. Das kann man mit der Hilfe der Funktionen *tic* und *toc* durchführen:

m-Datei *Geschwind_mess*1.*m*

```
clear    %Nach jeder Ausfuehrung werden alle Variablen geloescht
 tic;
 x = 0.99;  y = 0.05;
 a(1) = 1;  b(1) = 0;
  for  k = 2:8000
        a(k) = x*a(k - 1) - y*b(k - 1);
        b(k) = y*a(k - 1) + x*b(k - 1);
  end
toc;
```

Die Ausführungszeit dieses Codes dauert etwa 0.288738 Sekunden.
m-Datei *Geschwind_mess*2.*m*

```
clear
%Vektorisierte Loesung
tic;
x = 0.99;   y = 0.05;
n = 1:8000;
a(1) = 1;  b(1) = 0;
a = x.*a.*n - y.*b.*n;
b = y.*a.*n + x.*b.*n;
toc;
```

In diesem Fall dauert die Ausführungszeit etwa 0.154170 Sekunden. Der Code mit der *for*-Schleife braucht auch mehr Speicherplatz in Matlab. Arrays sollten vorbelegt werden, bevor eine for-Schleife ausgeführt wird. Dabei wird die Menge an Speicherbelegung notwendig minimisiert. In dem ersten Beispiel dises Kapitels wurden alle Befehle innerhalb der *for*-Schleife ausgeführt und die Größe der Variable x wurden um 1 erhöht. Das alles erzwingt Matlab sich die Zeit nehmen, um mehr Speicher für x zuzuweisen, jedes mal, wenn es durch die Schleife läuft. Um diesen Schritt zu beseitigen, das Beispiel mit

der for-Schleife soll umgeschrieben werden:

$$
\begin{aligned}
&>> x = zeros(1, 10);\ \%vorreservirter\ Speicher\ fuer\ x \\
&>> for\ n = 1 : 10 \\
&\qquad x(n) = sin(n * pi/10); \\
&\quad end
\end{aligned}
$$

In diesem Fall nur die Werte von x brauchen in der Schleife jeweils geändert werden. Vorreservierung von dem Speicherplatz kommt einmal außerhalb der Schleife vor.

Wichtig: Falls eine Berechnung nicht zum Ende kommt, kann man sie in Matlab mit der Tastenkombination *Ctrl-c* abbrechen!

3.3 Der Profiler

Der Matlab-Profiler hilft, Stellen im Programmcode zu finden, für die Matlab besonders lange braucht. Er wird mit $profile\ on$ in der Kommandozeile gestartet. Danach wird der zu alnalysiernde Code ausgeführt. Mit $profile\ viewer$ wird eine detalierte Übersicht über die Laufzeiten angezeigt. Der Profiler hat noch viele weitere Optionen, siehe dazu $help\ profile$ und $doc\ profile$. Wir analysieren die Laufzeit einzelner Operationen in dem folgenden Beispiel:

m-Datei $Laufzeit\_Analyse.m$

```
%Laufzeit einzelner Operationen analysieren
N = 100; max_n = 100;
a = 100*rand(N); b = randn(N);
for n = 1:max_n
     a + b;
     a - b;
     a. *b;
     a. /b;
     sqrt(a);
     exp(a);
     sin(a);
     tan(a);
end
```

Es folgt mit dem Profiler durch Eingabe von:

$$
\begin{aligned}
&>> profile\ on \\
&>> Laufzeit\_Analyse \\
&>> profile\ viewer
\end{aligned}
$$

Nach dem Anklicken von $Laufzeit\_Analyse$ im sich öffnenden Fenster kann man sehen welche Operationen in diesem Code am meisten verbrauchen Zeit.

3.4 *While* Schleife

Im Gegensatz zur *for*-Schleife, die eine Gruppe von Anweisungen mit bestimmter Anzahl der Iterationen ausrechnet, rechnet die *while*-Schleife eine Gruppe von Anweisungen mit unbestimmter Zeit aus. Die allgemeine Form der *while*-Schleife lautet:

$$
\begin{aligned}
&>>while\ Angabe\\
&\quad (Anweisungen)\\
&\quad end
\end{aligned}
$$

Die Befehle zwischen *while* und *end* werden **so lange** durchgeführt bis alle Elemente in *Angabe* wahr sind. Normalerweise Auswertung von den Angaben ergibt ein skalares Ergebniss, aber Array-Ergebnisse sind auch möglich. In dem Code

$$
\begin{aligned}
&>>summe = 0;\\
&>>N = 1;\\
&>>while\ (N+1) > 1\\
&\quad N = N/2;\\
&\quad summe = summe + 1;\\
&\quad end\\
&>>summe\\
&\quad summe =\\
&\qquad\quad 53
\end{aligned}
$$

N startet bei Eins. Solange $(N+1) > 1$ ist wahr, werden alle Anweisungen innnerhalbder *while*-Schleife ausgerechnet. Da N ständig durch zwei geteilt wird, wird die Addition $N+1$ irgendwann nicht mehr größer als Eins. In diesem Moment ist diese Angabe falsch und die *while*-Schleife wird beendet. Es muss immer darauf geachtet werden, dass auch tatsächlich einer der Elemente einer Matrix oder Vektors bzw. eines Skalars irgendwann den Wert Null annimmt oder dass die logische Bedingung irgendwann "falsch" ergibt und damit die Bedingung für das Beenden der Schleife erreichen wird. Wird dieses nicht beachtet kann die *while*-Schleife theoretisch unendlich lange laufen. Ein Beispiel dazu:

```
m=5;
while m
m=m-2
end
```

Dadurch, dass von ungeradenen Zahl m immer wieder 2 abgezogen wird, wird m nie eine Null, sondern sehr schnell negativ. Das Gegenbeispiel ist:

```
m=5;
while m
m=m-1
```

```
end
```

Im Gegensatz zur *for*-Schleife muss man für die *while*-Schleife mit begränzten Durchgängen die Endbedingung sehr genau überlegen, damit die Schleife auch wirklich beenden werden kann. Matlab gibt dazu keine Hilfestellung oder Wahrnung aus. Noch ein Beispiel:

```
i=1;
 s=0;
 while i<4
 s=s+i;
 i=i+1;
 endedisp([s])
```

3.5 Verzweigung mit $If - Else$

Oft müssen die Anweisungsfolgen auf Grundlage eines Vegleichstests bewertet werden. In Programmiersprachen wird solches Vergleichen durch Verzweigung $if - else - end$ bereitgestellt. Die einfachste Form dieser Verzweigung ist:

$$\begin{aligned} &>> if \quad Angabe \\ &\qquad Anweisungen \\ &\quad end \end{aligned}$$

Die Befehle zwischen if und *end* werden ausgewertet, **wenn** alle Ellemente in der Angabe wahr sind. Wir betrachten ein Beispiel:

$$\begin{aligned} &>> Aepfel = 10; \\ &>> Preis = Aepfel * 25 \\ &\quad Preis = \\ &\qquad 250 \\ &>> if\ Aepfel > 5 \\ &\qquad Preis = (1 - 20/100) * Preis; \\ &\quad end \\ &>> Preis \\ &\quad Preis = \\ &\qquad 200 \end{aligned}$$

In diesem Beispiel wollen wir denn Preis für größere Einkäufe (wenn man mehr als fünf Äpfel kauft) mit dem 20% Rabat ausrechnen. In dem weiteren Beispiel ist auch eine

Alternative mit *else* möglich:

$$
\begin{aligned}
&>>a = [2\ 4\ 6\ 8\ 4];\\
&>>b = [1\ 3\ 5\ 8\ 9];\\
&>>if\ sum(a) < sum(b)\\
&\qquad Norm = sum(a);\\
&\quad else\\
&\qquad Norm = sum(b);\\
&\quad end\\
&>>Norm\\
&\quad Norm =\\
&\qquad 24
\end{aligned}
$$

Hier wird der erste Teil ausgewertet, wenn die Angabe wahr ist. Der zweite Teil wird nur dann ausgewertet, wenn die Angabe falsch ist. Man kann auch mehrere Bedingungen mit Hilfe von *elseif* abprüfen lassen und jeder Bedingung einen Anweisungsblock zuordnen:

$$
\begin{aligned}
&>>x = 2457;\\
&>>if\ x\ >=\ 2 \wedge 14\\
&\qquad ['Antwort\ A :'\ num2str(2 \wedge -14 * x)]\\
&\quad elseif\ x\ >=\ 2 \wedge 12\\
&\qquad ['Antwort\ B :'\ num2str(2 \wedge -12 * x)]\\
&\quad elseif\ x\ >=\ 2 \wedge 10\\
&\qquad ['Antwort\ C :'\ num2str(2 \wedge -10 * x)]\\
&\quad else\\
&\qquad x\\
&\quad end
\end{aligned}
$$

Das Ergebniss dieses Codes ergibt:

$$
\begin{aligned}
&>>ans =\\
&\qquad Antwort\ C : 2.3994
\end{aligned}
$$

Im folgenden Beispiel sind die *elseif*-Bedingungen unsinnig gewählt, da alle Werte von i, die jeweils die Bedingungen unter *elseif* erfüllen würden, bereits die vorrangegangene Bedingung unter $if\ i > 0$ erfüllt haben müssen.

```
i=12;
if i > 0
   k=12;
elseif i > 10
```

```
   k=20;
else
   k=0;
end
```

Die Kontrollstrukturen kann man auch miteinander verbinden. In dem Code

$$
\begin{aligned}
&>> a = [2\ 4\ 6\ 8\ 12];\\
&>> b = [1\ 3\ 5\ 8\ 9];\\
&>> for\ n = 1:5\\
&\qquad if\ a(n) > b(n)\\
&\qquad\quad Norm = sum(a);\\
&\qquad end\\
&\quad end\\
&>> Norm\\
&\quad Norm =\\
&\qquad\quad 32
\end{aligned}
$$

wird jedes Element jeweils aus a und b miteinander verglichen und die entsprechende $Norm$ ausgewertet. Die kombinationen mit Schleifen und Verzweigungen erlauben oft mehr Bedingungen in einem Programm zu schreiben.

```
clear
for n = 1:6
    z = ceil(6*rand);
  if z < 1
     continue
  end
  if z == 5
     break;
  end
   disp(z)
end
```

Mit dem Befehl *continue* wir setzen die Iteration in der for-Schleife vor. Der Befehl *break* beendet die Schleife. Ein weiteres Beispiel:

```
Zaehler=1;
Zahl=2;
while Zaehler < 10
disp([Zaehler Zahl])
Zahl=Zahl*3;
Zaehler=Zaehler+1;
   if Zahl >= 7
```

```
      break
    end
  end
```

3.6 Verzweigung mit $Switch - Case$

Mit der *switch−case−otherwise*-Verzweigung kann der Status, Wert oder Zustand einer Variablen über verschiedenen Alternativen abgefragt werden und je nach zutreffenden Fall (*case*) in die entsprechende Anweisung oder Folge von Anweisungen verzweigt werden.

Wenn die Anweisungsfolgen auf Wiederholungsbasis ausgewertet werden müssen unter Verwendung eines Gleichheitstest mit einem gemeinsamen Argument, ist $switch - case$ Verzweigung häufig einfacher. Diese Verzweigung hat die Form:

$$
\begin{aligned}
&>>switch \;\; Angabe\\
&\quad case \;\; Ausdrueck\_1\\
&\quad\; Anweisungen\\
&\quad case \;\; Ausdrueck\_2\\
&\quad\;\; Anweisungen\\
&\quad\; \ldots\\
&\quad otherwise\\
&\quad\;\; Anweisungen\\
&end
\end{aligned}
$$

Die Angabe in diesem Fall kann entweder ein skalarer String oder ein Zeichenstring sein. In dem folgenden Beispiel wird je nach Wert von A ein anderer Text ausgegeben.

```
A=input('Wie gross ist A?')
b=3;
switch A
      case (3+4)*5/7
      disp('A istgleich 5')
      case 2
      disp( 'A ist gleich 2')
      case (b+3)/3^2
      disp('A ist gleich 4')
      otherweise
      disp('A ist nicht erkennbar')
 end
```

Mehrere mögliche Alternativwerte, die die gleichen Anweisungen nach sich ziehen, können in geschweiften Klammern durch Hochkommata getrennt aufgelistet werden. Die

demonstriert das folgende Beispiel.

```
>>x = 2.7
>>einheiten =' m';
>>switch einheiten
      case{'meter', 'm'}
       y = x/100;
      case{'millimeter', 'mm'}
       y = x * 10;
      case{'zentimeter', 'cm'}
       y = x;
      otherweise
       disp(['Unbekante Einheiten: ' einheiten])
       y = NaN;
   end
```

Nach der Eingabe dieses Codes in Matlab erhält man: $y = 0.027$.

4 TEIL IV

4.1 Funktionen

Bei Verwendung Matlab-Funktionen wie zum Beispiel, *inv*, *abs* und *sqrt* nimmt Matlab die Variablen, die man angegeben hat, rechnet sie aus und gibt Resultate wieder zurück. Anweisungen, die in einer Funktion ausgerechnet werden, sowie alle Zwischenvariablen, die durch diese Anweisungen erzeugt wurden, werden ausgeblendet. Alles was man sehen kann ist das, was in die Funktion hinein geht und was heraus kommt. In anderen Worten, eine Funktion ist eine "Black Box". Solche Eigenschaften sind sehr effektiv, die bei einer gewissen Eingabe eine bestimmte, wohldefinierte Ausgabe liefert ohne den Workspace zu beeinflussen.

4.2 Struktur einer Funktion

Es ist oft sehr sinnvoll eigene Funktionen zu Programmieren, die in anderen Programmen ausgeführt werden können. Funktionen werden in gewöhnlichen m-Dateien geschrieben. Eine Funktion mit einem Eingabe (*in*)- und einem Ausgabenparameter (*out*) ist wie folgt strukturiert:

```
function out = Funktions_Name(in)
%Kommentar
Anweisung1
Anweisung2
...
out = ...
end
```

Eine Funktion mit m Eingabe- und n Ausgabeparametern sieht wie folgt aus:

```
function [out_1, ..., out_n] = Funktions_Name(in_1, ..., in_m)
%Kommentar
Anweisung1
Anweisung2
...
out_1 = ...
out_2 = ...
```

```
...
out_n = ...
end
```

Die Funktion $Funktions\_Name$ wird erzeugt, indem die Funktionsdefinition in einer Datei $Funktios\_Name.m$ gespeichert wird. Der Aufruf erfolgt mit

$$>>[out\_1, ..., \ out\_n] = Funktions\_Name(in\_1, ..., in_m)$$

Wir betrachten folgende Beispiele

```
function summe = test1(A)
%Summe aller Eintraege von Matrix  A
[m,n] = size(A); % m Zeilen  und n  Spalten  von   A
summe = 0;      %Initialisierung
   for i = 1:m        %laeuft von  1  bis  m
       for  j = 1:n       %laeuft von  1  bis  n
           summe = summe + A(i,j);
       end
   end
 end
```

Alle Kommentare in einem Skript oder in einer Funktion sind sehr hilfreich für s Verstehen, was man im Programm gemacht hat. Die erste Kommentarzeile ist besonders wichtig. Der Aufruf der Funktion $test1$ im Command Window liefert

$$>>A = round(randn(3,4))$$
$$A =$$
$$\begin{matrix} 1 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 2 \end{matrix}$$
$$>>test1(A)$$
$$ans =$$
$$8$$

Es ist wichtig den Eingabeparameter vor dem Aufruf einer Funktion zu definieren.

```
function [summe, prod] = test2(A)
%Rechne Summe und Produkt  einer   Matrix A
[m,n] = size(A);    % m Zeilen und n  Spalten von  A
summe = 0; prod = 1; %Initialisierung
```

```
 for  i = 1:m          %laeuft von  1  bis  m
    for  j = 1:n        %laeuft von  1  bis  n
         summe = summe + A(i,j);
         prod = prod*A(i,j);
     end
  end
end
```

Der Aufruf dieser Funktion erfolgt mit

$$>>[summe,\ prod] = test2(A)$$
$$summe =$$
$$8$$
$$prod =$$
$$0$$

Man beachte, dass der Aufruf $test2(A)$ ebenfalls möglich ist, in dem Fall erhält die Variable *ans* den Wert der ersten Ausgabevariablen.
Damit Matlab auf eine Datei zugreifen kann, muss sich diese im Suchpfad befinden. Dieser umfasst das aktuelle Verzeichniss, sowie alle Verzeichnisse, die bei
$>> path$
angezeigt werden. Ob eine Funktion existiert beziehungsweise gefunden wird, kann man mit dem Befehl *exist* herausfinden.

Einige Regeln für die Erzeugung von Funktions m-Dateien:

- Der Name der Funktion und der Datei sollte identisch sein.
- Der erste Kommentarblock nach der Funktionsdefinition wird als Hilfetext der Funktion bei $help\ Funktions\_Name$ angezeigt. Er sollte stets die Funktionsdefinition beinhalten, da man sich nicht immer alles merken kann und oft auch die Reihenfolge der Parameter nach einiger Zeit nicht mehr kennt.
- Die allererste Kommentarzeile wird bei $lookfor$ durchsucht, daher sollte sie den Funktionsnamen, sowie eine stichwortartige Beschreibung der Funktion entchalten.
- Eine Funktion endet nach der letzten Zeile.
- Zu viele Kommentare sind besser als zu wenige!

Aufgabe 1. Erzeugen Sie eine Funktion die eine Matrix als Eingabe hat und die Anzahl der Elemente zurückgibt.
Lösung:

```
function anzahl = test3(A)
% Anzahl der Elemente von A
[m,n] = size(A);
  for i = 1:m
    for j = 1:n
         anzahl = m*n;
    end
  end
end
```

4.3 Haupt und Unterfunktionen

Es können auch mehrere Funktionen in eine Datei geschrieben werden. Nach außen sichtbar ist aber nur die erste Funktion. Die anderen Funktionen dienen nur als unsichtbare Hilfsfunktionen. Wir betrachten folgendes Beispiel:

```
function out = Funk_Haupt(in)
%Diese Funktion ist nach aussen sichtbar
N = 5;
y = Funk_Hilf(N, in); %Berechne Funk_Haupt mit der Hilfe von Funk_Hilf
out = 2*y + in;
end

function x = Funk_Hilf(z_1, z_2)
%Diese Funktion ist von aussen unsichtbar
x = z_1.^2 + sin(z_2);
end
```

Der Aufruf der Funktion $Funk\_Haupt(in)$ mit dem Eingabeparameter in liefert:

$>> in = [2\ 4\ 3\ 1\ 8];$
$>> Funk\_Haupt(in)$
$ans =$

$$5.3819e+01\ \ 5.2486e+01\ \ 5.3282e+01\ \ 5.2683e+01\ \ 5.9979e+01$$

Beim Aufrufen der Funktion $Funk\_Hilf(z\_1, z\_2)$ bekommt man eine Fehlermeldung:

$>> z\_1 = linspace(0, pi, 11);\ \ z\_2 = z\_1;$
$>> Funk\_Hilf(z\_1, z\_2)$
$???\ Undefined\ function\ or\ method\ 'Funk\_Hilf'\ for\ input\ arguments\ of\ type\ 'double'.$

Matlab erkennt diese Funktion nicht, weil sie von aussen unsichtbar ist.

4.4 Ananyme Funktionen

Anonyme Funktionen erlauben es innerhalb einer Zeile Funktionen zu definieren, die intern keine Variablen benötigen. Besonders nützlich sind anonyme Funktionen im Zusammenhang mit Funktionen, die Funktionen als Parameter benötigen (z.B. Integrieren, Differenzieren, ...), weil es so möglich ist bei Funktionen einige Parameter festzulegen und andere Variablen zu lassen.

Allgemein sieht die Definition einer anonymen Funktion wie folgt aus:

$$Funktions\_Name = @(Eingabeparameter)\ \ Ausdruck$$

Zum Beispiel die Funktion $f(x, y) = e^x - y$ lässt sich so definieren:

$$\begin{aligned} >> & f = @\ (x, y)\ exp(x) - y \\ & f = \\ & \quad @(x, y) exp(x) - y \end{aligned}$$

Hier braucht man nicht x und y vorher bestimmen. Im Falle mehrerer Ausgabeparameter sollte man beachten, dass ein Ausdruck der Form

$$>> [out1,\ out2,\ out3] = @(x) sin(x), cos(x), exp(x)$$

in Matlab nicht mglich ist, da nicht klar ist, wie die Elemente zugewiesen werden sollen. Eine elementweise Zuordnung erhält man aber mittels

$$>> [out1,\ out2,\ out3] = deal(sin(x),\ cos(x),\ exp(x))$$

Das kann man dann in der Definition einer anonymen Funktion wie folgt benutzen:

$$\begin{aligned} >> & f = @\ (x)\ deal(sin(x),\ x \wedge 2 + 4); \\ >> & [out1,\ out2] = f(1) \\ & out1 = \\ & \quad 8.4147e - 01 \\ & out2 = \\ & \quad 5 \end{aligned}$$

Hier wird die anonyme Funktion f mit dem Eingabeparameter $x = 1$ ausgerechnet. Um Parameter in anderen Funktionen festzulegen (oder einen Schnitt zu betrachten), kann man wie folgt vorgehen:

$$\begin{aligned} >> & f = @\ (x, y)\ exp(x) - y; \\ >> & g = @\ (x)\ f(x, 1); \\ >> & fplot(g,\ [-1, 1])\ \ \%Zeichne\ die\ Funktion\ g = f(x, 1)\ mit\ fplot \end{aligned}$$

Wichtig: Sobald eine anonyme Funktion definiert wurde, verändert die Variation eines

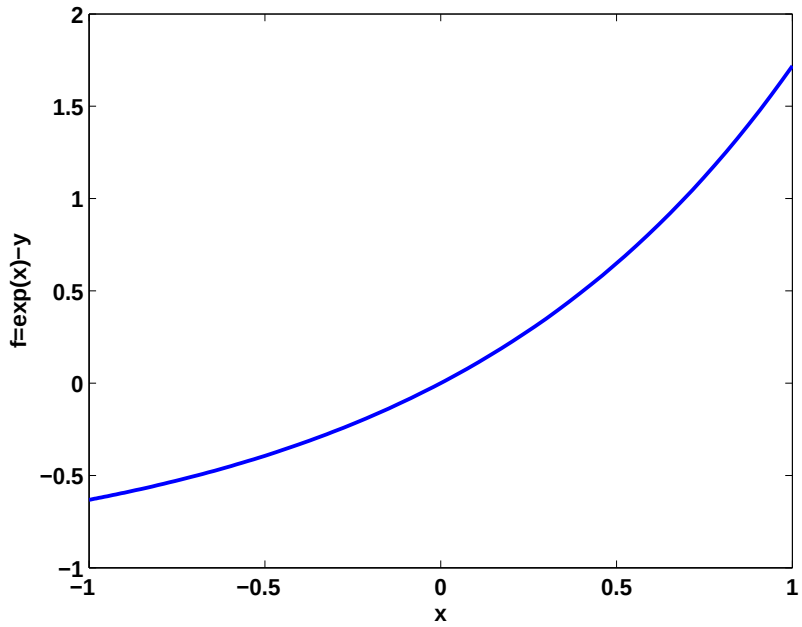


Abbildung 4.1: Plotten einer anonymen Funktion mit $fplot$.

Parameters die anonyme Funktion nicht mehr.

Beispiel 4.4.1.

$$
\begin{aligned}
&>> alpha = 2;\\
&>> f = @\ (x,\ y,\ z)\ x \wedge 2 + y \wedge 2 - alpha * z \wedge 2;\\
&>> f(1,\ 2,\ 1)\\
&ans =\\
&\qquad 3\\
&>> alpha = 0;\\
&>> f(1,\ 2,\ 1)\\
&ans =\\
&\qquad 3\\
&>> f = @\ (x,\ y,\ z)\ x \wedge 2 + y \wedge 2 - alpha * z \wedge 2;\\
&>> f(1,\ 2,\ 1)\\
&ans =\\
&\qquad 5
\end{aligned}
$$

Um den Parameter *alpha* zu verändern, muss man nach der Eingabe von *alpha* die Funktion f neu definieren.

4.5 Inline Funktionen

Die letzte Möglichkeit Funktionen in Matlab zu definieren, ist mit Hilfe des Befehls *inline*. Dieser macht aus einem String eine Funktion. Wir betrachten folgende Beispiele:

$$
\begin{aligned}
&>> f = inline(' \, exp(x) - 1 \, ') \quad \%Definition\ der\ Inline\ Funktion \\
&\quad f = \\
&\qquad Inline\ function: \\
&\qquad f(x) = exp(x) - 1 \\
&>> f(2) \\
&ans = \\
&\qquad 6.3891e + 00
\end{aligned}
$$

Man kann natürlich eine Inline Funktion mit mehreren Variablen definieren:

$$
\begin{aligned}
&>> g = inline(' \, log(x)/(1 + y \wedge 2) \, ') \\
&\quad g = \\
&\qquad Inline\ function: \\
&\qquad g(x, y) = log(x)/(1 + y^2) \\
&>> g(2,\ 3) \\
&ans = \\
&\qquad 6.9315e - 02
\end{aligned}
$$

Wir betrachten ein Beispiel:

```
ftext = input( ' Gib eine Funktion R->R ein: ' );
f = inline(ftext);
ezplot(f);
title( ' ftext ');
xlabel( ' x ');
ylabel( ' f(x) ');
```

Nach dem Aufruf dieses Codes muss man eine Funktion als einen String eingeben.
Ferner kann man aus einer Inline Funktion sehr einfach eine anonyme Funktion erstellen.

$$
\begin{aligned}
&>> f = inline(' \, sin(x) + cos(y) \wedge 2 \, '); \quad \%Definition\ der\ Inline\ Funktion \\
&>> g = @\ (a,\ b) f(a,\ b); \qquad \%Definition\ der\ anonymen\ Funktion
\end{aligned}
$$

4.6 Funktionshandels - Funktionen als Übergabeparameter

Durch die Definition einer anonymen Funktion erhält man ein Funktionshandle, das auf die definierte Funktion zeigt. Es können auch Funktionshandles erzeugt werden, die auf bereits existierende Funktionen zeigen:

$>> f = @\ Funktions\_Name$ $\%definiert\ das\ Handle\ auf\ die\ Funktion\ Funktions\_Name$
$\%dazu\ sollte\ allerdings\ Funktions\_Name.m\ existieren$
$>> g = @\ (y,\ x)\ y(x)$ $\%definiert\ eine\ anonyme\ Funktion\ g,$
$>> g = (f,\ 4)$ $\%die\ eine\ Funktion\ y\ bei\ x\ auswertet$

Man kann auch Handles auf eingebaute Matlab-Funktionen definieren, etwa $f = @\ sin$. Auf diese Weise lassen sich bequem Funktionen als Parameter an andere Funktionen übergeben. Typische Beispiele für Funktionen, die Funktionen als Eingabeparameter benötigen, sind unter anderem die Differentiation und Integration. Wir betrachten folgende Beispiele:

Beispiel 4.6.1.

```
function out = auswerte1 (funk, x)
%Funktion auswerte1 wertet funk bei x aus
%macht dasselbe wie oben g
  out = funk(x);
end
```

Die Funktion $auswert1.m$ kann nur mit Handles oder Funktonsobjekten umgehen:

$>> funk = @\ test1;$
$>> x = rand(3);$
$>> auswerte1(funk, x)$
$ans =$
$4.2057e+00$

Beispiel 4.6.2.

```
function out = auswerte2 (funk, x)
%Funktion auswerte2 wertet funk bei x aus
%macht dasselbe wie oben g
  out = feval(funk, x);
end
```

Die Funktion $auswerte2.m$ kann auch mit Funktionsnamen umgehen:

$>>auswerte2(funk, x)$ $\quad$ $\%macht\ das\ gleiche\ wie\ auswerte1$

$ans =$

$4.2057e+00$

$>>auswerte2(\ 'sin\ ', 5)$

$ans =$

$-9.5892e-01$

$>>auswerte1('\ sin\ ', 5)$

??? *Attempted to access* $funk(5)$; *index out of bounds because* $numel(funk) = 3$.

Error in $==>$ $auswerte1$ *at* 4

$out = funk(x);$

Aufgabe 2. Schreiben Sie eine Funktion, die eine Funktion sowie ein Intervall $[a,\ b]$ als Eingabe hat, welche die Funktion im Intervall mit *plot* zeichnet.

Lösung:

```
function out = funkplot(f, a, b)
%Plotten einer Funktion
x = a:0.1:b;
out = f(x);
plot(x,out)
end
```

Der Aufruf dieser Funktion kann z.B. mit

$$>>a = -2 * pi;\ b = 2 * pi;$$
$$>>f = @\ cos;$$
$$>>funkplot(f,\ a,\ b)$$

folgen.

Wichtig: Die Reihenfolge der Argumente beim Aufruf der Funktion darf nicht vertauscht werden.

4.7 Rekursive Funktionen

Wie in anderen Programmiersprachen können die Aufrufe der Funktionen auch rekursiv sein. Rekursive Funktionen sind Funktionen, die sich selbst aufrufen. Das bringt natürlich die Gefahr unendliche Schleife im Programm zu haben. Eine typische rekursive Funktion sieht wie folgt aus:

```
function out = Funktions_Name(in)
%Kommentar
```

```
Anweisungen
. . .
out = Funktions_Name(in - 1)
end
```

Diese Funktion wird folgenderweise interpretiert: rufe $Funktions\_Name$ in der Definition von $Funktions\_Name$ mit einem kleineren Eingabeparameter auf. Die rekursive Funktionen kann man durch das Beispiel der Berechnung der Finbonacci-Zahlen presentieren. Also, wir erzeugen eine m-Datei, die $fibonacci.m$ heißt:

```
function f=fibonacci(n)
 if n==0
    f=1;
 elseif n==1
    f=1;
 else
    f=fibonacci(n-1)+fibonacci(n-2);
  end
end
```

Rekusive Funktionen sind sinnvoll in Situationen, in denen man ein Problem in gleichartige kleinere Teilprobleme zerlegen kann. Bei der obigen Implementation der Funktion $fibonacci$ sind bei Berechnung von f_n die kleinere Teilprobleme durch die Berechnung von f_{n-1} und f_{n-2} gegeben. Somit kommt eine rekursive Implementation potentiel in Frage. Es wird für die Berechnung von größeren Zahlen f_n auffällig sein, dass die Wartezeit beim Lauf dieses Programms extrem lang ist. In der Tat kann man die Berechnung von f_n effizienter programmieren. Ein weiteres Beispiel:

Beispiel 4.7.1.

Der Fakultätswert einer natürlichen Zahl n wird durch

$$n! = n(n-1)(n-2)\cdots 3\cdot 2\cdot 1$$

definiert. Dann lässt sich die Funktion $rfakult.m$ als rekursive Funktion definieren:

```
function out = rfakult(n)
%Implementiere die Fakultaet rekursiv
   if n == 1
       out = 1;
   else
       out = n*rfakult(n-1);
   end
end
```

$$>> rfakult(10)$$
$$ans =$$
$$3628800$$

Eine iterative Funktion für die Berechnung des Fakultätswertes lässt sich mit Hilfe von for-Schleife zu erzeugen:

```
function out = ifakult(n)
%Implementiere die Fakultet iterativ
 out = 1;
   for i = 1:n
       out = out*i;
   end
end
```

Als Antwort erhält man hier die gleiche Zahl 3628800. Jedoch, der Vergleich von Laufzeit der beiden Programmen mit *tic* und *toc* liefert für die rekursive Funktion

$$>> rekfakult(10)$$
$$Elapsed\ time\ is\ 0.000005\ seconds.$$
$$Elapsed\ time\ is\ 0.000038\ seconds.$$
$$Elapsed\ time\ is\ 0.000049\ seconds.$$
$$Elapsed\ time\ is\ 0.000058\ seconds.$$
$$Elapsed\ time\ is\ 0.000066\ seconds.$$
$$Elapsed\ time\ is\ 0.000075\ seconds.$$
$$Elapsed\ time\ is\ 0.000084\ seconds.$$
$$Elapsed\ time\ is\ 0.000092\ seconds.$$
$$Elapsed\ time\ is\ 0.000101\ seconds.$$
$$Elapsed\ time\ is\ 0.000110\ seconds.$$
$$ans =$$
$$3628800$$

und für die iterative Funktion

$$>> ifakult(10)$$
$$Elapsed\ time\ is\ 0.000003\ seconds.$$
$$ans =$$
$$3628800$$

4.8 Programmieren mit Funktionen

Bisher wurde nur herkömliche Funktionen besprochen. In einer gesonderten m-Datei wird eine Funktion gespeichert, welche dann in einem Hauptprogramm ausgeführt werden kann. Wir haben über die Mögkichkeiten gesprochen, durch die wir eine Funktion programmieren können, deren Argumente Funktionen sind. Zum Beispiel, wir wollen eine numerische Approximation der Ableitung programmieren. Es seien mehrere Funktionen, z.B., $f_1(x) = sin(x),\ f_2(x) = x^2,\ f_3(x) = cos(x)$, gegeben und wir wollen die Ableitungen dieser Funktionen in einem festem Punkt $x = 1$ mit Hilfe der Approximation

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

für eine fest vorgegebene Schrittweite h berechnen. Dann könnte man für jede der Funktionen $f_1,\ f_2,\ f_3$ den Wert $D^+ f(x)$ berechnen. 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 $f'(x)$ berechnet. Die Routine muss dann nur einmal programmiert werden und könnte für alle Funktionen genutzt werden. Das bedeutet auch, dass man die Funktion f als Argument der Funktion *Ableitung.m* übergeben muss. Zunächst muss die Funktion f programmiert werden:

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

Diese Funktion ist als $f.m$ Datei gespeichert. Weiter schreiben die Funktion *Ableitung.m*:

```
function Df=Ableitung(f,x,h)
%Berechnen die Ableitung einer Funktion f
Df=(f(x+h)-f(x))/h;
end
```

Jetzt erzeugen wir ein Hauptprogramm-Skript in dem die Funktion f und Ableitung aufgerufenwerden. Beim Funktionslauf Ableitung im Hauptprogramm muss allerdings berücksichtigt werden, dass einer der übergebene Argumente eine Funktion ist, und es geht um den Funktionshandle.

```
%Rufe die Funktion f auf und
%mit Hilfe der Funktion Ableitung.m die Ableitungen bestimme
h=0.02;
x=0:h:4;
y=f(x)
Dy=Ableitung(@f,x,h)
```

Dieses Programm kann nur zur näherungsweisen Ableitung aller differenzirbaren Funktionen $f : \mathbb{R}^n \to \mathbb{R}$ verwendet werden.

Man kann die Funktionen definieren, die weder Ausgabe- noch Eingabeparameter enthalten. Z.B.,

```
function test4
x=rand;
sqrt(x)+x^2;
end
```

Es ist auch möglich die Funktionen zu definieren, die einen Ausgabeparameter ohne den Eingangsparameter haben:

```
function out=test5
x=rand;
out=sqrt(x)+x^2;
end
```

Im Prinzip ist das das Selbe wie im obigen Beispiel. Es ist erlaubt in Matlab gleichnamige Ein- und Ausgabeparameter zu verwenden:

```
function x=test6(x)
x=x+1;
end
```

Man kann mehrere eigene Funktionen erstellen und sie alle mit einem Skript auszuführen. Wir betrachten ein Beispiel, in dem man zwei eigene Funktionen erstellt und dann mit einem Programm aufruft:

```
function out = funk1(x)
%Werte zuerst die Funktion funk1 aus
out = x^2+1;
end
```

```
function out = funk2(f,a,b)
%Werte die naechste Funktion funk2 aus
out = (f(a) - f(b))/2;
end
```

Nach Erzeugung Funktionen $funk1$ und $funk2$ kann man ein entsprechendes Programm $prog1.m$ erstellen:

```
%Finde einen Wert S unter Benutzung von funk1 und funk2
x = 3;
a = -pi/2; b = pi/3;
f = @sin;
S = funk1(x)*funk2(f,a,b)
```

Der Aufruf dieses Programms liefert: -1.8740e+01.

4.9 Der Debugger

Der Matlab Debugger ist äußerst hilfreich, um die Funktionswise von Programmen zu verstehen und um Programmierfehler zu suchen. Er ermöglicht das Setzen von Haltepunkten an gewünschten Stellen im Code, um diesen schrittweise zu durchlaufen. Variablen können dann an der Deburger-Kommandozeile $K >>$ ausgewertet und sogar verändert werden.
Im Matlab Editor werden Haltepunkte durch Klicken rechts neben der Zeilennummer gesetzt beziehungsweise gelöscht und alle weiteren Kommandos sind per Mausklick auf die Debugger-Toolbar ausführbar.

Wichtige Debuggerbefehle für das Kommandofenster lauten wie folgt:

$>>dbstop$ $\quad \%Setze\ Stoppunkt\ (z.B.>>dbstop\ if\ error)$
$K>>dbclear$ $\quad \%Deaktiviert\ Stoppunkt$
$K>>dbcont$ $\quad \%Setzt\ Berechnung\ bis\ zum\ nchsten\ Stoppunkt\ fort$
$K>>dbstatus$ $\quad \%Listet\ alle\ Stoppunkte\ auf$
$K>>dbstep$ $\quad \%Macht\ einen\ Schritt\ im\ Code$
$K>>dbtype$ $\quad \%Listet\ den\ Quellcode\ mit\ Zeilennummern$
$K>>dbquit$ $\quad \%Beendet\ den\ Debugger$

Weitere Informationen über den Debugger findet man mit *help debug* oder *doc debug*.

5 TEIL V

Grafiken

5.1 Zeichnen in 2D

In den bisherigen Sitzungen wurden bereits einige Methoden zum Zeichnen von Funktionen $\mathbb{R} \to \mathbb{R}$ vorgestellt. Die einfachsten sind *ezplot* und *plot*, ein weiterer Befehl ist *fplot*. Zunächst schauen wir uns einmal den Befehl *ezplot* an:

```
>>f = @ (x) abs(x). * sin(x);      %Vektorisierte, anonyme Funktion erstellen
>>ezplot(f)                        %und zeichnen
>>ezplot(' abs(x) * sin(x) ')      %Formelstringdirekteingeben
>>ezplot(@ (x) abs(x). * sin(x))  %AnonymeFunktiondirekteingeben
```

Es ist wichtig in der ersten und dritten Version die Funktion zu vektorisieren, da die Funktion mit *ezplot* wesentlich schneller gezeichnet werden kann.
Mit *ezplot* ist es auch möglich parametrische Kurven zu zeichnen, das heißt, Funktionen von der Form $t \to\ (x(t),\ y(t)) \in \mathbb{R}^2$.

```
>>ezplot(@sin, @cos, [0, 2 * pi])
>>ezplot(' sin(x) ', ' cos(x) ', [0, 2 * pi])
```

Diese Kurven können auch implizit durch eine Gleichung, etwa $x^2 + y^2 - 1 = 0$ gegeben sein.

```
>>ezplot(' x ∧ 2 + y ∧ 2 - 1 ', [-1, 1, -1, 1])
>>ezplot(@ (x, y) x. ∧ 2 + y. ∧ 2 - 1, [-1, 1, -1, 1])
```

Die Benutzung von *ezplot* hat den Nachteil, dass sich nur wenige Einstellungen vornehmen lassen: die Einteilung in x-Richtung liegt fest und der Linienstil sowie die Farbe kann man nur nachträglich ändern.
Wesentlich flexibler ist die Funktion *plot*. Sie zeichnet Punkte (x, y) und verbindet sie (gegebenfalls) der Reihe nach mit Linien. Um Funktionen mit *plot* visualisieren zu können,

müssen also Wertetabellen erzeugt werden.

$$>>x = [0\ 0\ 1\ 0.5\ 0\ 1\ 1\ 0\ 1];$$
$$>>y = [0\ 1\ 1\ 1.5\ 1\ 0\ 1\ 0\ 0];$$
$$>>plot(x, y,'\ ro-\ ','\ LineWidth\ ',\ 4);$$
$$>>title('\ Haus\ von\ Nikolaus\ ')$$

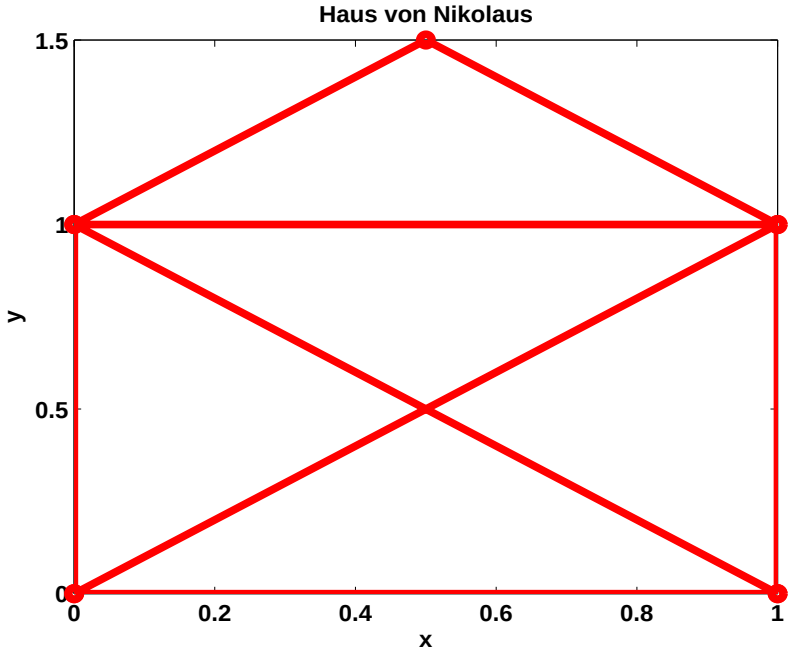


Abbildung 5.1:

5.2 Achsenskalierungen

Manche Beziehungen lassen sich besser mit logarithmischen Skalen erkennen, dazu kann man entweder die Skalierung der Achsen in einem Plot nachträglich über die Eigneschaften *xscale*, *yscale*, *zscale* ändern, oder die Grafik gleich mit *semilogx*, *semilogy* oder *semilogz* erstellen. Wir betrachten ein Beispliel:

```
x = linspace(1, 20, 39);
y = 2*exp(x);
plot(x, y)
```

In der Abbildung 6.2 haben wir eine Exponentialfunktion im Intervall [1, 20] geplottet. Bei den Größen, deren Werttebereich der dargestellten Daten viele Größenordnungen

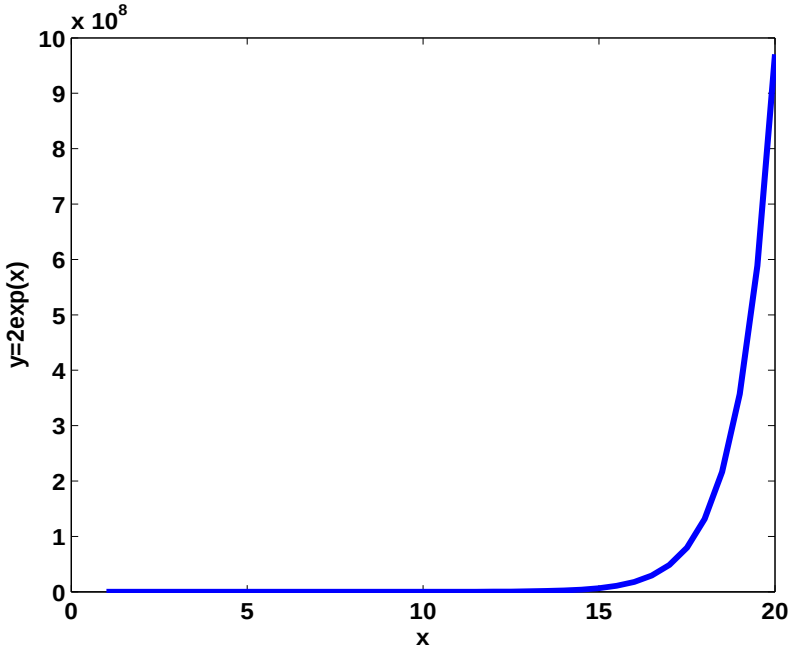


Abbildung 5.2:

umfasst, ist es sinnfoll die logarithmische Skala zu benutzen, die nicht den Zahlenwert einer Größe verwendet, sondern den Logarithmus ihres Zahlenwerts. Dann sieht die Exponentialfunktion auf logarithmischen Skala wie folgt aus (siehe die Abbildung 6.3):

```
x = linspace(1, 20, 39);
y = 2*exp(x);
loglog(x,y)
```

Man kann auch noch folgende Befehle für die Achsenskalierungen benutzen:
$>> set(gca, 'yscale','log')$ % y-Achse logarithmisch setzen
$>> get(gca, 'xscale')$ % zeigt Skalierung der x-Achse an
$>> semilogy(x, y, 'r')$ % zeichnet gleich mit dieser Skalierung

5.3 Plots beschriften

Mit den Funktionen *title*, *xlabel*, *ylabel*, *zlabel* und *legends* können Beschriftungen (nachträglich) zum Plot hinzugefügt werden.

```
x = linspace(1, 20, 39);
y = [2*sqrt(x); sin(x)];
plot(x, y, 'r*-')
title( ' Ueberschrift ' )
```

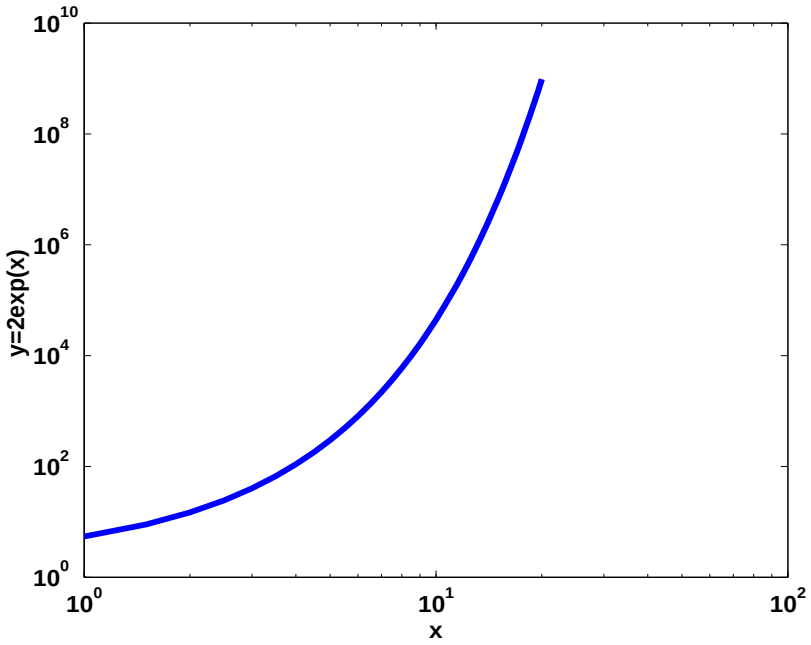


Abbildung 5.3:

```
xlabel( ' x-Achse ' )
ylabel( 'y-Achse ' )
legend( { ' 2\cdot x^{0.5} ', sin(x) '} )
```

Dasselbe ist auch per Maus über die Menüs im Grafikfenster möglich. Mit dem Befehl *shg* (show graph window) kann übrigens das aktive Grafikfenster in den Vordergrund geholt werden, wenn es verborgen ist.

5.4 Mehrere Plots in einer Figure

Wie bereits oben gezeigt wurde, lassen sich mehrere Funktionen direkt mit *plot* in ein Fenster zeichnen. Außerdem, kann man mehrere Plots mit der Funktion *hold on* nacheinander in ein Grafikfenster zeichnen ohne die vorherigen zu löschen. Wir betrachten folgendes Beispiel:

```
x = linspace(1, 20, 39);
y = [2*sqrt(x); sin(x)];
plot(x, y)
hold on
plot(x, sin(x).*cos(x), ' r: ')
```

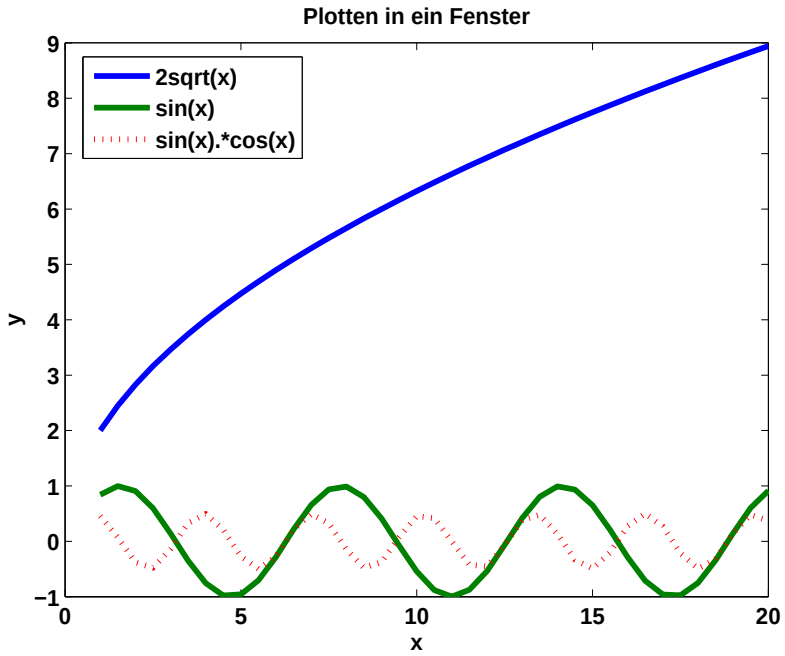


Abbildung 5.4:

Um mehrere Plots nebeneinander oder untereinander in ein Fenster zu zeichnen, gibt es die Funktion *subplot*:

```
x = 0:0.2:5;
y = 2*x.^3;
s1 = subplot(2, 1, 1)      %1.Subplot von 2 Plots in 2 Zeilen, 1 Spalte
s2 = subplot(2, 1, 2)      %2.Subplot von 2 Plots in 2 Zeilen, 1 Spalte
plot(x, y, ' c+-' )        % auf aktuelle Achse plotten
title('2x^3')
xlabel('x')
ylabel('y')
axes(s1)                   % waehle s1 aus
plot(y, x.^2)              % plotte in s1
title('x^2')
xlabel('x')
ylabel('y')
```

Dieser Code liefert folgendes Bild:

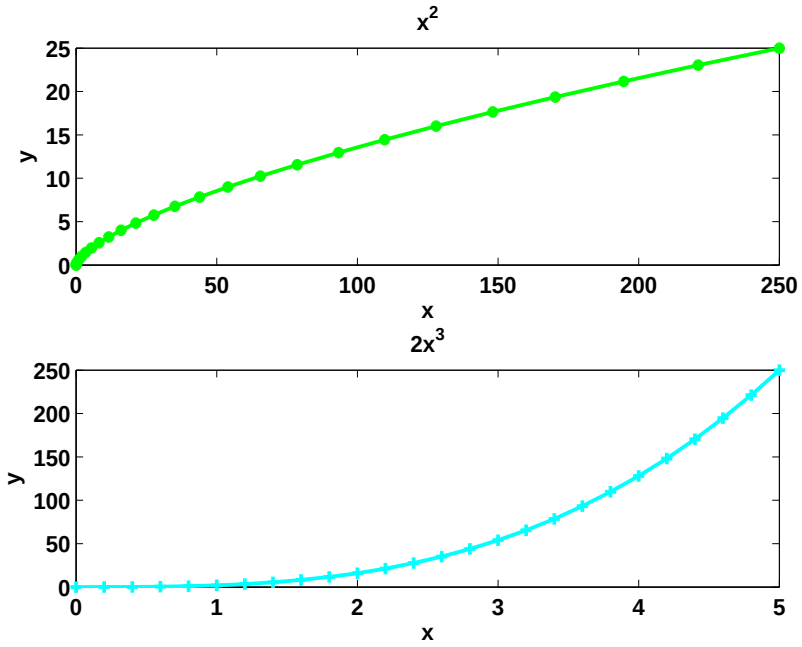


Abbildung 5.5:

5.5 Komplexe Werte in 2D mit *plot* darstellen

Falls man die Funktion *plot* nur mit einem Argument aufruft, also $plot(x)$, so hängt das Verhalten der Funktion davon ab, was für Daten übergeben wurden. Sind die Daten reellwertig, so zeichnet *plot* den Wert gegen den Index $1 : length(x)$. Falls es sich bei x um eine Matrix handelt, so wird jede Spalte gegen den Spaltenindex gezeichnet.
Handelt es sich dagegen um komplexwertige Daten, so wird $\mathbb{R}^2$ als $\mathbb{C}$ interpretiert und der Befehl entspricht $plot(Re(x),\ Im(x))$, wie das folgende Beispiel zeigt.

```
x = exp(1i*(1:10).*pi/5);
plot(x, ' m: ')
title( ' Komplexe 10-te Einheitswurzeln ' );
xlabel( ' Re(x) ' )
ylabel( ' Im(x) ')
```

Hier haben wir alle 10-ten Einheitswurzeln gezeichnet, die der Formel

$$x = \exp(\frac{2\pi ik}{n}), \quad k = 0, \quad 1, ..., n-1$$

entsprechen.

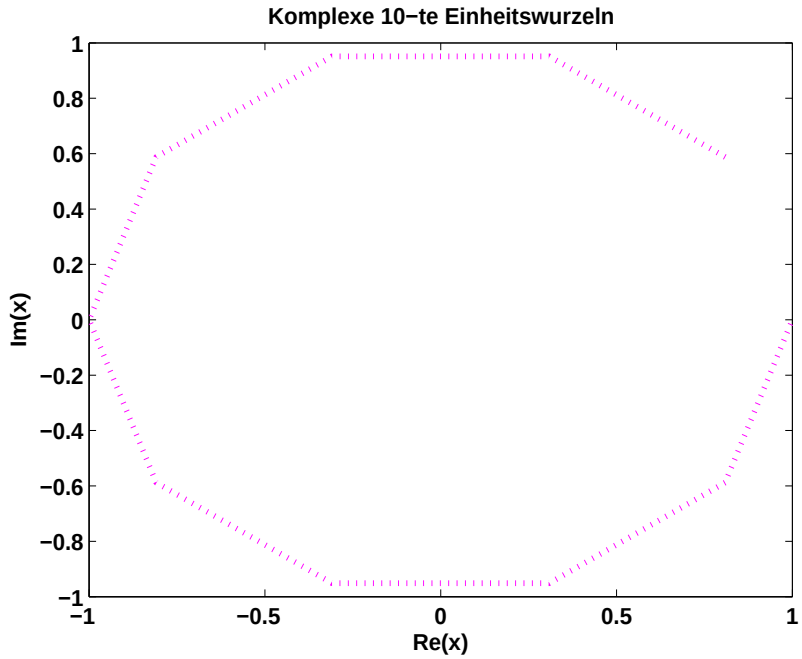


Abbildung 5.6:

5.6 Histogramme, Flächen und Polardarstellung

Neben dem normalen 2D Plot gibt es noch weitere *plot*-Funktionen wie *bar*, *barh*, *hist*, *pie* und *stem*, die insbesonderer für die Analyse von Daten interessant sind. Ein einfaches Beispiel ist:

```
y = [7 6 5; 8 4 6; 1 2 3; 9 7 8];
subplot(2, 2, 1);
bar(y)
title('bar(y)')
subplot(2, 2, 2);
bar( -10:2:-4, y )
title('Im Intervall [-10:-4]')
subplot(2, 2, 3);
bar(y, 'stacked')
title('bar stacked')
subplot(2, 2, 4);
barh(y)
title('barh(y)')
```

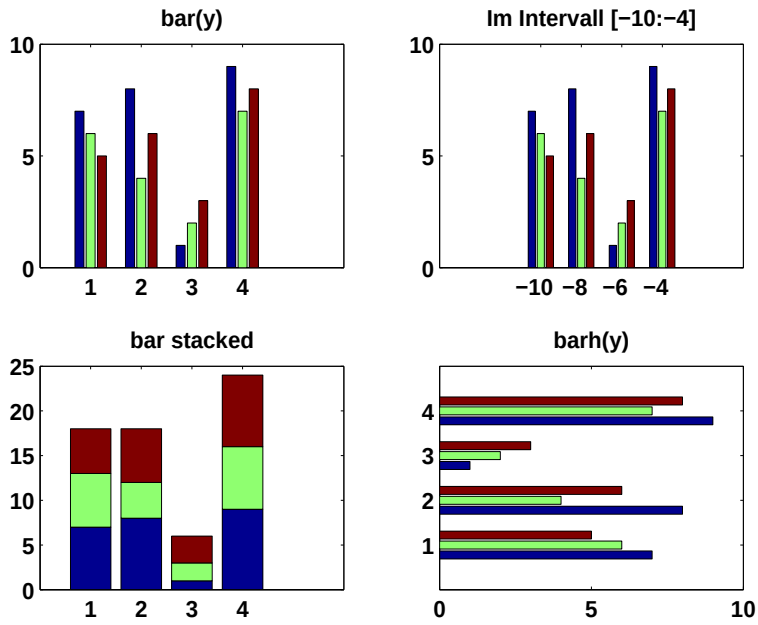


Abbildung 5.7:

Um Flächen zu zeichnen kann man die Befehle *area* und *fill* benutzen. Wir betrachten ein Beispiel:

```
t = (1:2 :15) ' *pi/8;
x = sin(t); y = cos(t);
fill(x, y, 'r');
axis equal;
hold on
text(0, 0, 'Stop', 'Color', [1 1 1 ], 'HorizontalAlignment', . . .
' center', 'FontSize', 80, 'FontWeight', 'bold');
```

Mit der Funktion *polar* lassen sich Funktionen $f = f(\theta,\ r)$, die in Polarkoordinaten gegeben sind, einfach zeichnen.

```
r=0:0.0001:1;
theta=4*pi*r;
polar(theta, r, 'g');
hold on
t=linspace(0, 2*pi);
polar(t, sin(2*t).*cos(2*t));
```

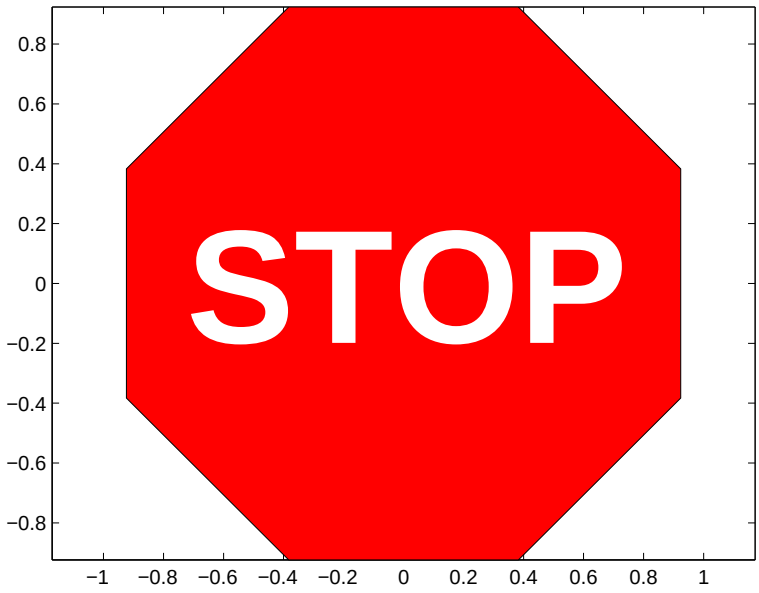


Abbildung 5.8:

Beispiel 5.6.1. Zeichnen Sie zwei Kreise, einen mit Radius 1 und einen mit Radius 2 mit *polar* in eine Achse.

Ein entsprechendes Programm kann wie folgend aussehen:

```
% Ploten von zwei Kreise
R=2;
t=linspace(0,2*pi,100);
theta=ones(1,100*R);
polar(t,theta,'r')
hold on
R=1;
t=linspace(0,2*pi,100);
theta=ones(1,100*R);
polar(t,theta,'g')
```

5.7 Vektorfelder

Mit der Funktion *quiver* lassen sich Vektorfelder $f : \mathbb{R}^2 \to \mathbb{R}^2$ durch kleine Pfeile visualisieren. Dies ist im Zusammenhang mit Differentialgleichungen besonders nützlich.
Zur Erinnerung: Ein Vektorfeld ist eine Funktion, die jedem Punkt eines Raumes einen Vektor zuordnet.

Das Vektorfeld $f : \mathbb{R}^2 \to \mathbb{R}^2,\ (x,\ y) \to (-y,\ x)$ kann wie folgt visualisiert werden:

```
x=[0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3];
y=[0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2];
quiver(x, y, -y, x)
```

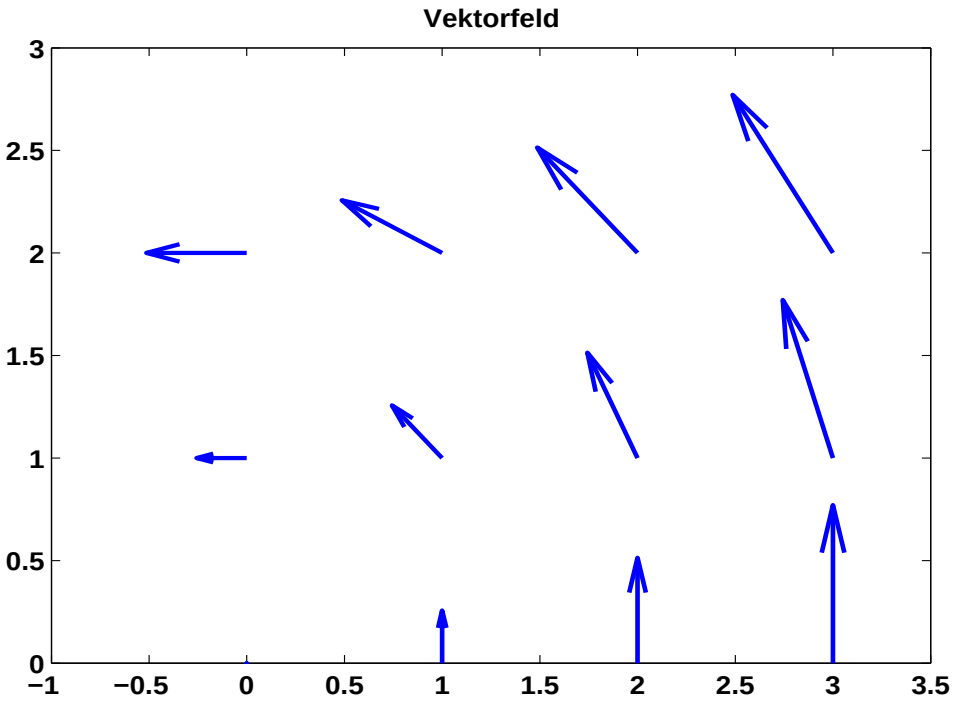


Abbildung 5.9:

Einfacher lassen sich Koordinaten eines rechteckigen Gitters mit der Funktion *meshgrid* erstellen.

```
[x, y]=meshgrid(0:3, 0:2);
quiver(x, y,-y,x)
```

6 TEIL VI

Zeichnen in 3D

6.1 Kurven im $\mathbb{R}^3$

Auch für Zeichnen in drei Dimensionen gibt es einen einfachen Zeichenbefehl: *ezplot*3. Dieser funktioniert ganz ähnlich wie der entsprechende Befehl *ezplot*. Er ist jedoch nur für das Zeichen parametrischer Kurven geeignet.

```
f_x = @(t) 2*t;
f_y = @(t) sin(t);
f_z = @(t) t.^2;
ezplot3(f_x, f_y, f_z);        % Standardmaessig 0<t<2pi
```

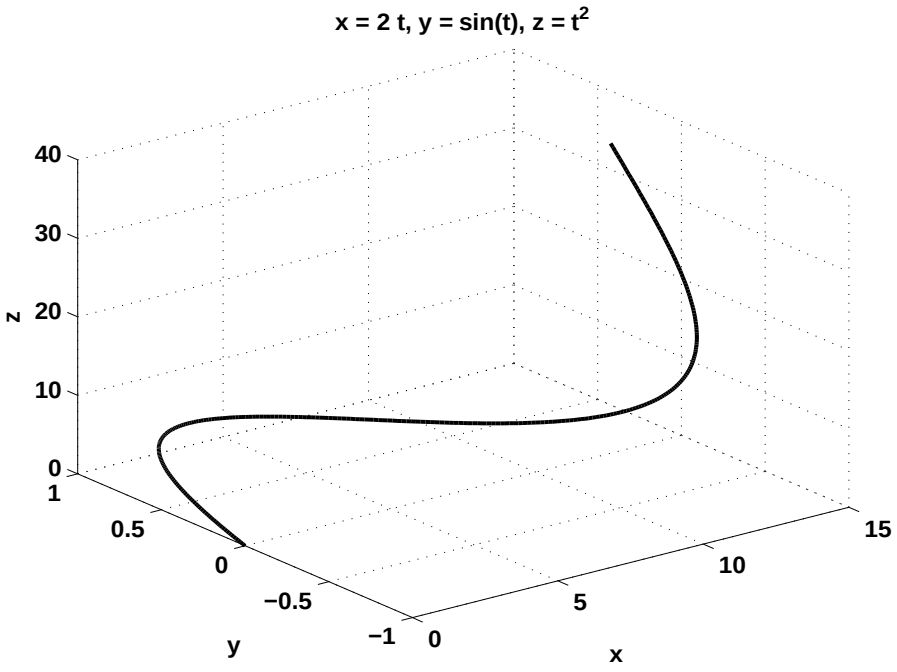


Abbildung 6.1:

Wie auch schon im 2D Fall ist dieser Befehl sehr einfach, man kann aber bei weitem nicht soviele Einstellungen vornehmen, wie mit dem komplizierteren *plot*3 Befehl. Dieser ist

das 3D-Äquivalent zum *plot* Befehl und wird auch genau so aufgerufen. Es werden also Datenpunkte gezeichnet und (eventuell) mit Linien verbunden. Wir betrachten folgendes Beispiel:

```
figure;
x = rand(5, 1);
y = rand(5, 1);
z = rand(5, 1);
plot3(x, y, z, 'r-p')
grid
```

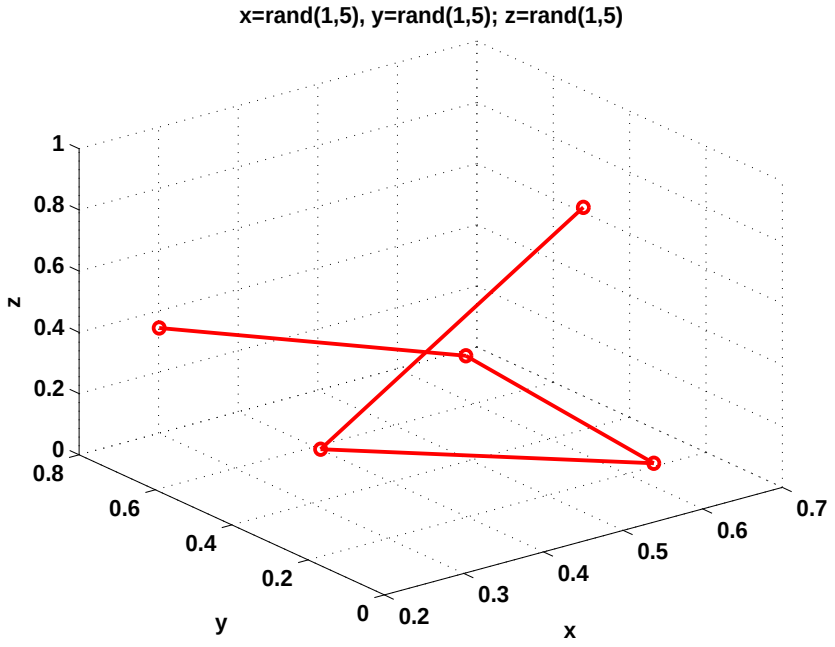


Abbildung 6.2:

Entsprechend einfach ist es Kurven in 3D mit *plot*3 zu zeichnen, die in Parameterdarstellung

$$t \to (x(t),\ y(t),\ z(t)) \in \mathbb{R}^3$$

vorliegen.

Aufgabe1: Zeichne die folgenden in Parameterdarstellung gegebenen Raumkurven:

1. $t \to (t,\ t^2,\ t^3), \quad t \in [-1,\ 1],$
2. $t \to ((t/\pi)^2 \sin(20t),\ t^2,\ t), \quad t \in [0,\ \pi],$
3. $t \to (\sin(t),\ \sin(t)\cos(t),\ t), \quad t \in [0,\ 2\pi].$

Lösung:

```
t = linspace(-1,1,100);              % das Intervall bestimmen
subplot(3,1,1);
plot3(t, t.^2,t.^3);
title('x=t, y=t^2, z=t^3')
t = linspace(0,pi,100);               % ein anderes Intervall bestimmen
subplot(3,1,2);
plot3((t/pi).^2.*sin(20*t), t.^2, t)
title('x=(t/pi)^2sin(20t), y=t^2, z=t')
t = linspace(0,2*pi,100);
subplot(3,1,3);
plot3(sin(t), sin(t).*cos(t),t)
title('x=sin(t), y=sin(t)cos(t), z=t')
```

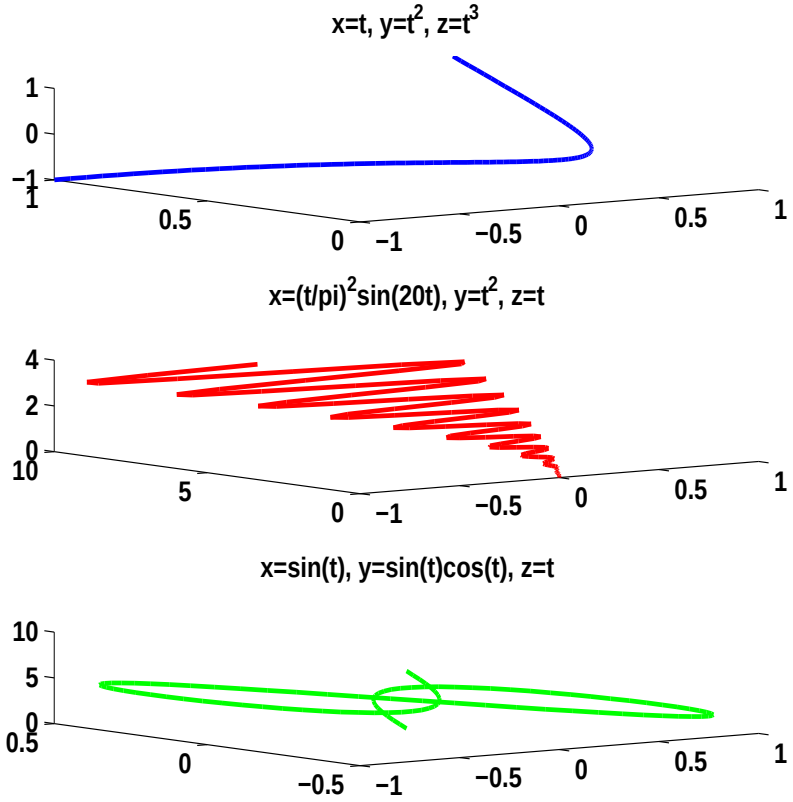


Abbildung 6.3:

Als Alternative zum *subplot* könnte man vor jedem Plotten die Funktion *figure* benutzen:

```
figure(1)
t = linspace(-1,1,100);              % das Intervall bestimmen
plot3(t, t.^2,t.^3);
title('x=t, y=t^2, z=t^3')
figure(2)
t = linspace(0,pi,100);               % ein anderes Intervall bestimmen
plot3((t/pi).^2.*sin(20*t), t.^2, t)
title('x=(t/pi)^2sin(20t), y=t^2, z=t')
figure(3)
t = linspace(0,2*pi,100);
plot3(sin(t), sin(t).*cos(t),t)
title('x=sin(t), y=sin(t)cos(t), z=t')
```

So erhält man drei einzelne Bilder.

6.2 Animation von Kurven

Eine nette Fähigkeit von *ezplot*3 ist die Möglichkeit Kurven zu animieren. Zum Beispiel, die Kurve

$$x(t) = (1+t^2)\sin(2t),\ y(t) = (1-t^2)\cos(2t),\ z(t) = t,\ \ t \in [-10,\ 10]$$

kann einfach durch

$>> ezplot3('\ (1+t.\wedge 2)*\sin(2*t)\ ',\ '\ (1-t.\wedge 2)*\cos(2*t)\ ',\ 't\ ',\ [-10,10],\ '\ animate')$

mit einer Animation des Kurvendurchlaufs gezeichnet werden. Mittels *Repeat* kann man die Animation wiederholen.
Ein anderer Befehl um Animationen zu erzeugen ist *comet* in 2D und *comet*3 in 3D. Wie *plot* zeichnet auch *comet* nicht Funktionen sondern Datenpaare (oder Datenvektoren). Hier ist ein Beispiel vom Zeichnen eines Kreises mit *comet*:

```
t = 0:0.01:2*pi;
x = sin(t);
y = cos(t);
comet(x, y)
```

Eine Animation wie mit der Option *animate* von *ezplot*3 kann auch selbst erstellt werden, indem man die einzelnen Bilder (frames) in einer Schleife hintereinander plottet. Wichtig ist der Befehl *drawnow*, der veranlasst, dass das erstellte Bild angezeigt wird.

Eine sehr effiziente Methode besteht darin, nicht die ganze Grafik, sondern nur die Position des Markers zu verändern, indem man Eigenschaften *xdata* und *ydata* des Linienobjektes, dass den Marker beschreibt, veändert, so wie es in dem folgenden Code

gezeigt wird.

```
N = input(' Wieviele Stuetzstellen? ');
figure;
x = linspace(0, 2*pi, N);
y = 1./x.*sin(x.^2);
plot(x, y, 'k');
hold on;
f = plot(x(1), y(1), 'ro', 'Markerfacecolor', 'red');
for j = 2:N
    set(f, 'xdata', x(j), 'ydata', y(j));
    drawnow;
end
```

6.3 Flächen in $\mathbb{R}^3$

Flächen, die in Parameterdarstellung gegeben sind, das heißt die als Abbildung $f : \mathbb{R}^2 \to \mathbb{R}$ vorliegen, kann man mit *mesh*, *surf* und manchen Fällen auch besser mit *trimesh* zeichnen.
Das Vorgehen ist dabei stets dasselbe, man muss zunächst eine Wertetabelle erzeugen, also jedem Paar von (x, y)-Werten einen z-Wert zuordnen. Da man häufig die x-Achse und die y-Achse in gewisse Abschnitte eingeteilt hat, muss zunächst das Gitter der $(x,\ y)$-Paare erzeugt werden. Dieses liefert ein strukturiertes Gitter. Erst dann kann man die Werttabelle ausfüllen, dafür ist es sinnvoll eine Funktion in vektorisierter Form vorliegen zu haben, da man dann sehr einfach auswerten kann. Wir betrachten ein Beispiel dazu (siehe Abbildung 6.4):

```
f = @(x, y) sin(x) - 4*cos(x).*sin(y);
x = -pi:0.2:pi;
y = x;
[X, Y] = meshgrid(x, y);
Z = f(X, Y);
surf( X, Y, Z);
shading interp
```

Man kann auch andere Befehle zum Flächeplotten in 3D verwenden: *mesh*, *meshc* und *meshz*. Die Konturen, die mit *meshc* zusätzlich zum Gitter erzeugt werden, können auch mit *contour*, *contour*3 oder *contourf* einzeln erstellt werden.

Falls man unstrukturiertes Gitter hat, das heißt die Punkte, an denen die Funktion f ausgewertet wird, liegen nicht in einem Rechteckgitter vor, so muss man vor dem Zeichnen noch eine Gitterstruktur erzeugen. Dies geht mit der Funktion *delaunay*, die

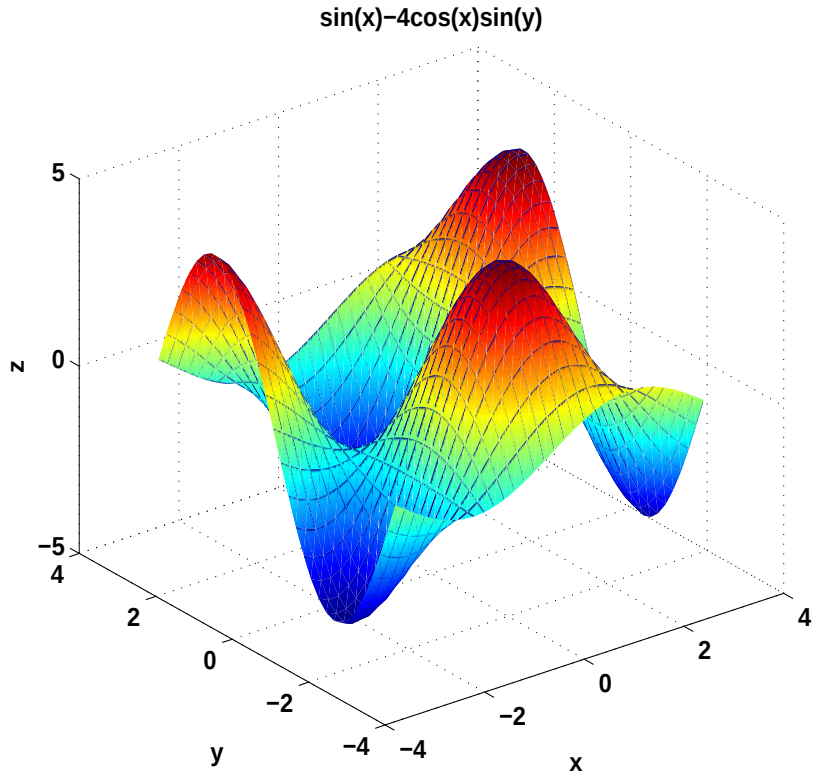


Abbildung 6.4:

aus den ungeordneten Punkten ein Dreiecksgitter erzeugt. Wir betrachten ein Beispiel (siehe Abbildung 6.5):

```
x = 2*rand(100, 1)-1;   % Zufaellige Stuetzstellen x und y erzeugen
y = 2*rand(100, 1)-1;
z = x.^3 - y.^2;
g = delaunay(x, y);      % erzeugt das Dreiecksgitter
trisurf(g, x, y, z);
```

Mann kann auch für das Plotten die Befehle *triplot* und *trimesh* benutzen.

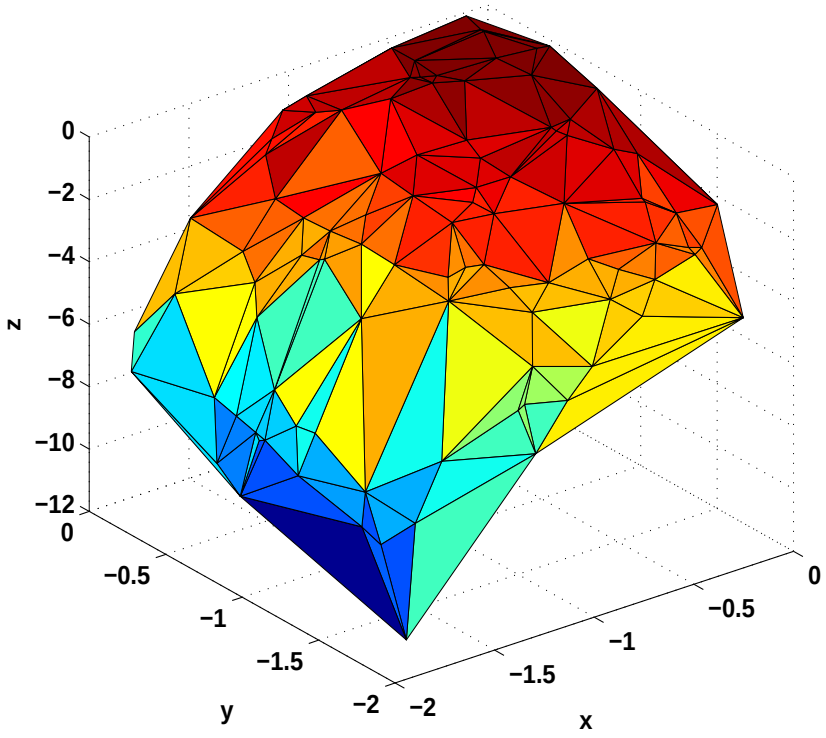


Abbildung 6.5:

6.4 Rotationsköper

Ein Rotationsköper um die z-Achse mit dem Profil $r(z)$ wird durch die Parameterdarstellung

$$\begin{aligned} x(z,\ \phi) &= r(z)\sin(\phi), \\ y(z,\ \phi) &= r(z)\cos(\phi), \\ z &= z \end{aligned}$$

beschrieben. Der zugehörige Matlab-Code lautet:

```
phi = linspace(0, 2*pi, 180);  % Winkel
z = -1:0.1:1;
r = @(z)cos(z);          % Profil
x = @(z, phi) r(z).*sin(phi);
y = @(z, phi) r(z).*cos(phi);
[Z, Phi] = meshgrid(z, phi);
surf(x(Z, Phi), y(Z, Phi), Z)
```

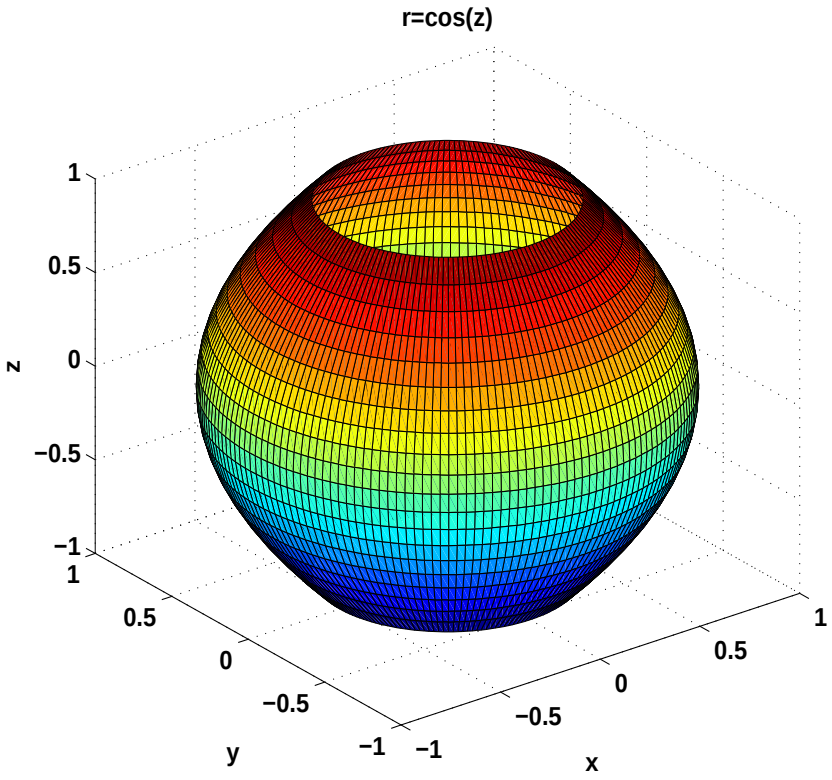


Abbildung 6.6:

Wenn man als dritte Koordinate nicht den Wert z, sondern den Winkel ϕ zeichnet, so erhält man statt Rotationsköpern Wendelflächen (siehe Abbildung 6.7)
$>> surf(Z. * sin(Phi),\ Z. * cos(Phi),\ Phi)$
Es ist auch nicht schwer ein Möbiusband zu zeichnen:

```
x = @(u, v)(1+1/2.*v.*cos(1/2*u)).*cos(u);
y =  @(u, v)(1+1/2.*v.*cos(1/2*u)).*sin(u);
z = @(u, v)1/2*v.*sin(1/2*u);
u = linspace(0, 2*pi, 180);
v=-0.5:0.05:0.5;
[u, v] = meshgrid(u, v);
surf(x(u, v), y(u, v), z(u, v));
```

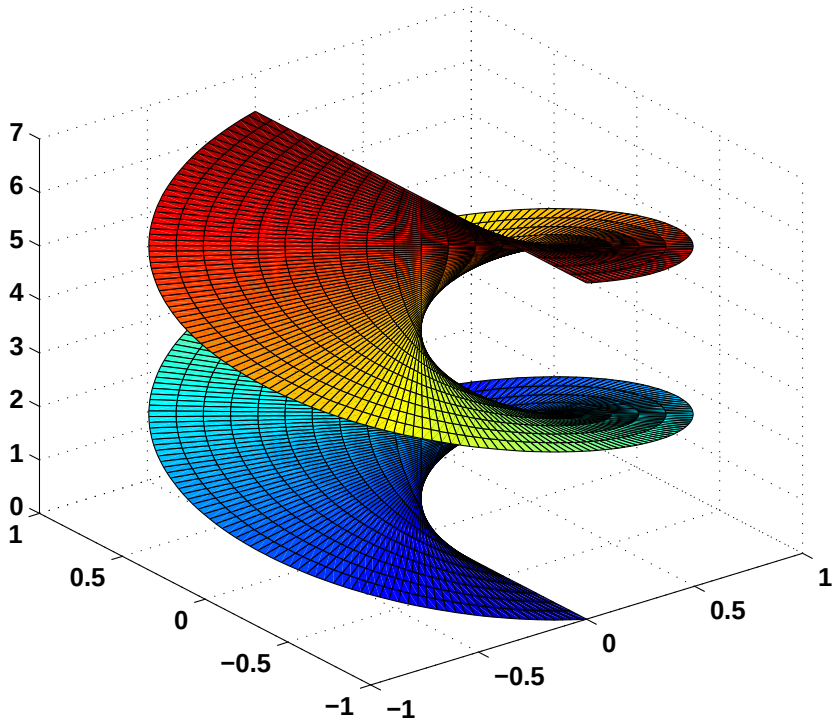


Abbildung 6.7:

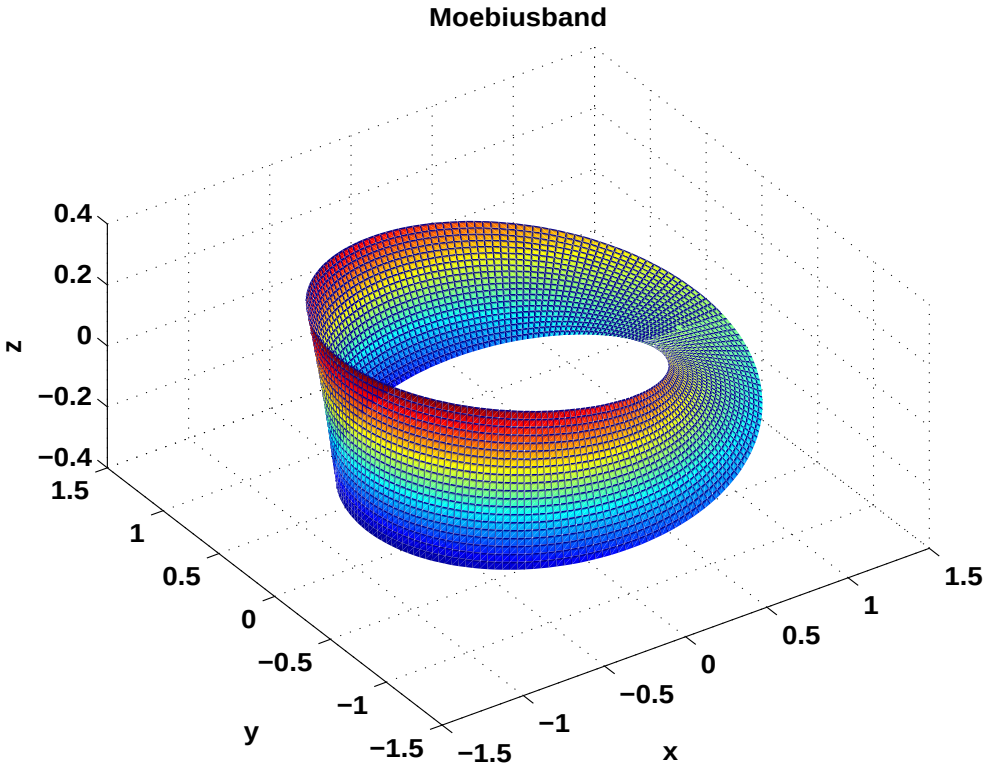


Abbildung 6.8:

7 TEIL VII

Tabellen mit Matlab erzeugen

Tabellen sind ganz oft ein Teil unserer schriftlichen Arbeiten. Man kann auch mit Matlab Tabellen erstellen und sie in eine Tex-Datei mit dem Befehl *input* einfügen.

Zunächst betrachten wir ein Beispiel, in dem wir eine approximative Lösung für die Differentialgleichung $y' = f(t, y),\ t \in [0, T]$ mit dem Anfangswert $y(t_0) = y_0$ suchen. Danach wollen wir den Fehler zwischen den numerischen und exakten Lösungen für fünf verschiedenen Schrittweiten bestimmen. Dabei erstellen wir eine Tabelle, die entsprechenden Daten enthält.

Die einfachste Methode zur numerischen Lösung eines Anfangswertproblem ist das explizite Euler Verfahren in der Form:

$$y_{i+1} = y_i + hf(t_i, y_i),\ i = 0, 1, 2, ...,$$

mit der Schrittweite $h > 0$ und $t_i = t_0 + ih$. In unserem Beispiel definieren wir $f(t, y) = -2.3y$. Dann haben wir die exakte Lösung in der Form:

$$y(t) = \exp(-2.3t).$$

Als erstes erstellen wir eine Matlab-Funktion *euler_det.m*, die exakte Lösung, Euler Approximation und den Fehler berechnet:

```
 function [error,heul] = euler_det(T,N,X0,k)
%Implimentire eine Approximation (Euler Methode) fuer:
% y'=-2.3y;

%Exakte Loesung:

h=T/N;
t=h*(0:1:N);
Xexact=X0*exp(-2.3*t);

%Approximation (Euler Methode)

R=2^k;
heul=R*h; n=N/R;
Xeul(1)=X0;
for i=1:n
```

```
    Xeul=Xeul+heul*(-2.3*Xeul);
end

% Berechne den Fehler

error=sqrt(abs(Xeul-Xexact(end)).^2);
end
```

Jetzt können wir ein Matlab-Programm erstellen, das die Werte von obigen Funktion ausrechnet und eine Tex-Datei *tab_euler.tex* liefert.

```
clear;
T=1;
N=2^(12);
X0=1;
k=7;
hvec=zeros(1,k);
error=zeros(1,k);

%Berechne die Fehler fuer k Schritte

for i=1:k
   [err,hvec(k-i+1)]=euler_det(T,N,X0,i);
   error(k-i+1)=error(k-i+1)+err;
end


%Berechne die Konvergenz Ordnung

EOC=zeros(1,k);
for j=1:k-1
    EOC(j+1)=(log(error(j))-log(error(j+1)))/(log(hvec(j))-log(hvec(j+1)));
end

%Erstelle eine Tabelle

A=[hvec;error;EOC];
 fileID=fopen('tab_euler.tex','w');
 fprintf(fileID,'\\begin{table}\n');
 fprintf(fileID,' \\caption{Fehler von numerischen Approximation}\n');
 fprintf(fileID,' \\label{tab:matprog}\n');
 fprintf(fileID,' \\begin{tabular}{p{1.1cm}p{1.3cm}p{1.2cm}}\n');
 fprintf(fileID,'     &  %5s  &   %7s   &   \n','Euler','Methode');
 fprintf(fileID,'\\\\ \\noalign{\\smallskip}\\hline\\noalign{\\smallskip}\n');
 fprintf(fileID,' $%1s$  &   %5s  &  %3s   \n','h','error','EOC');
```

```
fprintf(fileID,'\\\\ \\noalign{\\smallskip}\\hline\\noalign{\\smallskip}\n');
fprintf(fileID,'    %6.3f & %12.5f & ',A([1,2],1));
fprintf(fileID,'\\\\\n %6.3f & %12.5f & %12.2f ',A(:,2:end));
fprintf(fileID,'\n \\end{tabular}\n');
fprintf(fileID,'\\end{table}');
fclose(fileID);

% Zeichne den Plott von Konvergenzordnung

loglog(hvec,error,'k-*');
axis equal
```

Bemerkung: Man beachte, dass alle Befehle, die im LaTeX verwendet werden, mit dem doppelten backslasch im Matlab-Programm bezeichnet müssen.

Mit dem Befehl $fileID$ haben wir die Tex-Datei *tab_euler.tex* erzeugt, die man im Matlab-Editor anschauen kann:

```
 \begin{table}
  \caption{Fehler von numerischen Approximation}
  \label{tab:matprog}
  \begin{tabular}{p{1.1cm}p{1.3cm}p{1.2cm}}
       &  Euler &   Methode  &
\\   \noalign{\smallskip}\hline\noalign{\smallskip}
   $h$   &   error  &  EOC
\\   \noalign{\smallskip}\hline\noalign{\smallskip}
       0.250 &     0.06763 & \\
     0.125 &     0.03384 &         1.00 \\
     0.062 &     0.01677 &         1.01 \\
     0.031 &     0.00834 &         1.01 \\
     0.016 &     0.00416 &         1.00
  \end{tabular}
\end{table}
```

Hier kann man schon eine Tabelle erkennen, die drei Spalten mit den entsprechenden Werten enthält. Jetzt kann man die Datei *tab_euler.tex* mit dem LaTeX-Befehl input{*tab_euler.tex*} in die Hauptdatei einfügen. Man muss natürlich aufpassen, in welchem Ordner die Beiden Dateien sich befinden. Am besten wäre sie in einem Ordner zu speichern. Sonst muss man beim einfügen den $Pfad$ angeben.

Tabelle 7.1: Fehler von numerischen Approximation

Euler	Methode	
h	error	EOC
0.0078	0.00208	
0.0039	0.00104	1.00
0.0020	0.00052	1.00
0.0010	0.00026	1.00
0.0005	0.00013	1.00

Literaturverzeichnis

[1] Hanselman, D., Littlefield, B.: Mastering MATLAB 7, 2005 Pearson Education, Inc.

[2] MATLAB Getting Started Guide, 1984-2011 by The MathWorks, Inc.

[3] Moler, Cleve B.: Numerical Computing with MATLAB, 2004 The MathWorks, Inc.

[4] Quarteroni, A., Fausto, S.: Wissenschaftliches Rechnen mit MATLAB, Springer-Verlag Berlin Heidelberg 2006.

[5] Quarteroni, A., Fausto, S.: Scientific Computing with MATLAB and Octave, Second Edition, Springer-Verlag Berlin Heidelberg 2003, 2006.