MATLAB - Eine Einführung

Christian Kufner Harald Schmid Konstantin Schorp
Bettina Tögel Matthias Vestner Konstantin Pieper
Lorenz Pfeifroth

13. April 2008

Vorwort

Das vorliegende Skript gibt dem Leser eine Einführung in den Umgang mit MATLAB. Die Bedienung von MATLAB und die Themen Visualisierung und lineare Algebra mit MATLAB werden erklärt. Auf weiterführende Funktionen MATLAB wie Toolboxen und das Einbinden von Java- und C-Programmen wird eingegangen.

Dieses Handbuch ist besonders für Studenten der Mathematik geeignet, die im Studium MATLAB benötigen. Grundkenntnisse der linearen Algebra werden vorausgesetzt, jedoch sind keine Programmierkenntnisse notwendig. Dieses Skript ist im Internet unter

`http://www.fs.tum.de/fsmpi/kurse/matlab/matlabeinfuehrung.pdf`

verfügbar. Sie erreichen den verantwortlichen Mitarbeiter der TU München unter der Email-Adresse `brandenb@ma.tum.de`.

Zweite Auflage Dieses Skript wurde für den Ferienkurs „MATLAB Grundlagen“ im Wintersemester 07/08 überarbeitet und um die Abschnitte „Eigenwerte“ 5.4, „Ein- und Ausgabe“ 7 und „Troubleshooting“ erweitert.

Die Webadresse des Kurses ist:
`http://ferienkurse.ma.tum.de/twiki/bin/view/Allgemein/WS0708/Matlab`

Inhaltsverzeichnis

Kapitel 1

Einleitung

MATLAB ist eine mächtige Anwendung für Mathematiker und Ingenieure, die von *The Mathworks* entwickelt und vertrieben wird. Vielfältigste Funktionen zur Lösung numerischer Probleme, der Visualisierung von Daten und die Erweiterbarkeit des Basispakets über Toolboxen machen MATLAB zu einem guten Werkzeug für Studenten der Mathematik, die sich im Rahmen ihres Studiums auch mit dem Lösen numerischer Probleme beschäftigen.

MATLAB verfügt über drei besondere Funktionen, die es von anderen modernen Entwicklungsumgebungen unterscheidet.

1.1 Automatische Speicherverwaltung

In MATLAB müssen Variablen nicht deklariert werden, bevor sie zugewiesen werden. Darüber hinaus erweitert MATLAB Arrays automatisch, um Zuweisungen sinnvoll zu machen.

```
>> x(3) = 0

x =
    0     0     0

>> x([5,6]) = [1,2]

x =
    0     0     0     0     1     2
```

Diese automatische Speicherzuweisung ist äußerst bequem und eine der herausragenden Eigenschaften von MATLAB.

1.2 Variable Argumentlisten

MATLAB verfügt eine große Menge an Funktionen, die darüber hinaus vom Benutzer leicht erweitert werden kann. Diese verlangen keine oder mehr Eingabearguemente und liefern keine oder mehr Ausgabeargumente zurück. Es gibt

eine klare Trennung zwischen Ein- und Ausgabe: Eingabeargumente erscheinen in runden Klammern rechts, Ausgabeargumente in eckigen Klammern links vom Funktionennamen. Funktionen können eine veränderliche Zahl an Eingabe- und Ausgabeargumenten unterstützen, so dass bei einem Aufruf nicht zwingend alle Argumente übergeben werden müssen. Funktionen können sich aufgrund von Anzahl und Art der Argumente unterschiedlich verhalten.

```
>> x = [3 4];
>> norm(x)
ans =
     5

>> norm(x,1)
ans =
     7

>> m = max(x)
m =
     4

>> [m,k] = max(x)
m =
     4
k =
     2
```

1.3 Komplexe Arrays und Rechnerarithmetik

Der allgemeine Datentyp in MATLAB ist ein mehrdimensionales Array komplexer Zahlen, deren Real- und Imaginärteile mit doppelter Genauigkeit als Gleitkommazahlen gespeichert werden. Wichtige Spezialfälle sind Matrizen (zweidimensionale Arrays), Vektoren und Skalare. Sämtliche Berechnungen in MATLAB werden in Gleitkommaarithmetik durchgeführt, wobei komplexe Arithmetik genau da verwendet wird, wo komplexe Zahlen vorkommen. Es gibt keinen ausgezeichneten reellen Datentyp (obwohl für reelle Zahlen der Imaginärteil nicht gespeichert wird). Dies unterscheidet MATLAB von FORTRAN, C, Java und anderen Programmiersprachen, die nur relle Datentypen unterstützen. Es gibt noch weitere Datentypen, die aber vor allem für die effiziente Speicherung und weniger für die Berechnung verwendet werden.

Kapitel 2

Matrizen

Eine mxn Matrix ist ein zweidimensionales Array von Zahlen, das aus m Zeilen und n Spalten besteht. Sonderfälle sind Spaltenvektoren (n=1) und Zeilenvektoren (m=1). Praktisch alles in MATLAB basiert auf Matrizen. Selbst wenn Sie nicht planen Lineare Algebra zu benutzen, ist es notwendig, sich mit dem Erzeugen und Manipulieren von Matrizen zu beschäftigen.

2.1 Matrizen erzeugen

Matrizen können auf mehrere Arten erzeugt werden. Viele spezielle Matrizen können direkt über eine MATLAB Funktion generiert werden. Die Nullmatrix, die Einheitsmatrix und Einsmatrizen können über die Funktionen `zeros()`, `eye()` und `ones()` erzeugt werden. Alle haben die gleiche Syntax. Zum Beispiel erzeugt `zeros(m,n)` oder `zeros([m,n])` eine mxn Nullmatrix, während `zeros(n)` eine nxn Nullmatrix erzeugt.

```
>> zeros(2)

ans =

     0     0
     0     0

>> ones(2,3)

ans =

     1     1     1
     1     1     1

>> eye(3,2)

ans =

     1     0
     0     1
```

```
     0     0
```

Oft braucht man eine Einheitsmatrix deren Dimension mit der einer bereits existierenden Matrix übereinstimmt. Dies kann mit `eye(size(A))` erreicht werden.
Die Funktion `length` ist mit der Funktion `size` verwandt: `length(A)` ist die größere der beiden Dimensionen von `A`. Für einen nx1 oder 1xn Vektor `x` würde `length(x)` also n zurückgeben.

Zwei andere, sehr wichtige, Matrizen erzeugende Funktionen sind `rand` und `randn`, die Matrizen mit Pseudozufallszahlen als Einträgen generieren. Die Syntax ist die gleiche wie bei `eye`. Die Funktion `rand` erzeugt eine Matrix mit Zahlen aus der Gleichverteilung über dem Einheitsintervall $[0, 1]$. Für diese Verteilung ist der Anteil der Zahlen aus einem Intervall $[a, b]$ mit $0 < a < b < 1$ stets $b - a$. Die Funktion `randn` erzeugt eine Matrix deren Einträge standardnormalverteilt sind. Ohne Argumente geben beide Funktionen eine einzelne Zufallszahl zurück.

```
>> rand

ans =

    0.9501

>> rand(3)

ans =

    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
    0.4860    0.4565    0.2835
```

Bei Experimenten mit Zufallszahlen ist es oft wichtig, die gleichen Zufallszahlen erneut zu erzeugen. Die Zahlen, die von `rand` erzeugt werden hängen vom Zählerstand des Generators ab.
Dieser Zähler kann über den Befehl `rand('state',j)` gesetzt werden. `j=0` ist der Anfangszustand des Generators (diesen Zustand hat der Generator, wenn MATLAB startet). Für eine nichtnegative ganze Zahl `j` wird der Generator auf den Zustand `j` gesetzt. Der Zustand von `randn` wird auf ähnliche Art und Weise gesetzt. Die Periode von `rand` und `randn`, also die Anzahl Terme die erzeugt werden, bevor sich Teilfolgen wiederholen, übersteigt $2^{1492} \approx 10^{449}$.

Matrizen können explizit über die Klammernotation (square bracket notation) erzeugt werden. Zum Beispiel kann eine 3x3-Matrix mit den ersten neun Primzahlen mit folgendem Befehl erzeugt werden:

```
>> A = [2 3 5
        7 11 13
        17 19 23]

A =

     2     3     5
```

```
     7    11    13
    17    19    23
```

Das Ende einer Zeile kann über den Strichpunkt angegeben werden anstatt eines Zeilenumbruchs. Ein kürzerer Befehl wäre also:

```
>> A = [2 3 5; 7 11 13; 17 19 23]
```

Innerhalb einer Zeile können einzelne Elemente über ein Leerzeichen oder Komma getrennt werden. In ersterem Fall sollte beachtet werden, dass bei Angabe von Vorzeichen für die einzelnen Einträge kein Leerzeichen zwischen Vorzeichen und Element gelassen wird. MATLAB interpretiert das Vorzeichen sonst als Plus oder Minus Operator.

```
>> v = [-1 2 -3 4]

v =

    -1     2    -3     4

>> w = [-1,2,-3,4]

w =

    -1     2    -3     4

>> x = [-1 2 - 3 4]

x =

    -1    -1     4
```

Matrizen können auch über Blockform erzeugt werden. Sei `B` gegeben als `B = [1 2; 3 4]`:

```
>> C = [B zeros(2); ones(2) eye(2)]

C =

     1     2     0     0
     3     4     0     0
     1     1     1     0
     1     1     0     1
```

Blockdiagonalmatrizen können über die Funktion `blkdiag` erzeugt werden, die einfacher zu benutzen ist als die Klammernotation.

```
>> A = blkdiag(2*eye(2),ones(2))

A =
```

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

Für "getäfelte" Blockmatrizen eignet sich `repmat`: `repmat(A,m,n)` erzeugt eine Block-$m \times n$-Matrix, in der jeder Block eine Kopie von `A` ist. Wird `n` weggelassen, wird der Wert standardmäßig auf `m` gesetzt.

```
>> A = repmat(eye(2),2)

A =

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

MATLAB verfügt über Funktionen die Matrizen einer bestimmten Strukture erzeugen können. Ein Beispiel hierfür ist die Hilbertmatrix, deren Elemente (i, j) den Wert $1/(i + j - 1)$ haben. Die Matrix wird durch den Befehl `hilb` erzeugt, und ihre Inverse (die nur ganzzahlige Komponenten hat) über `invhilb`. Zu Erwähnen sei dann noch die Funktion `magic` zum Erzeugen magischer Quadrate, und `gallery`, eine Funktion die Zugang zu einer großen Anzahl spezieller Matrizen bietet. Mehr Informationen bezüglich einer Matrix aus der `gallery` erhält man durch die Eingabe des Befehls `help private/matrix_name`. Manche der Matrizen aus der Gallery werden im sparse-Format zurückgegeben.

2.2 Subindizes und Doppelpunktnotation

Um Zugriff und Zuweisung auf Teilmatrizen zu ermöglichen, verfügt MATLAB über eine mächtige, auf dem Doppelpunkt basierende Notation. Der Doppelpunkt fungiert als Definition von Vektoren die als Subindizes benutzt werden können. Für ganze Zahlen `i` und `j` definiert `i:j` den Zeilenvektor bestehend aus den ganzen Zahlen von `i` bis `j`. Mit `i:s:j` kann eine andere Schrittweite `s` festgelegt werden. Diese Notation gilt sogar, wenn `i`,`j` und `s` nicht ganzzahlig sind.

```
>> 1:5

ans =

     1     2     3     4     5

>> 4:-1:-2

ans =

     4     3     2     1     0    -1    -2
```

```
>> 0:.75:3

ans =

         0    0.7500    1.5000    2.2500    3
```

Einzelne Elemente einer Matrix werden über `A(i,j)` angesprochen, wobei $i, j \geq 1$ (Null- oder negative Indizes werden von MATLAB nicht unterstützt). Die Teilmatrix bestehend aus der Schnittmenge der Zeilen `p` bis `q` und den Spalten `r` bis `s` wird mit `A(p:q,r:s)` bezeichnet. Als Sonderfall bezeichnet ein Doppelpunkt an sich die sämtliche Zeilen oder Spalten; `A(:,j)` bezeichnet also die j-te Spalte, und `A(i,:)` die i-te Zeile. Das Schlüsselwort `end` bedeutet in diesem bezeichnet den letzten Index in der angegebenen Dimension; `A(end,:)` bezeichnet also die letzte Zeile von `A`. Schließlich kann auch eine beliebige Teilmatrix indiziert werden, indem spezifische Zeilen- und Spaltenindizes gewählt werden. Zum Beispiel erzeugt `A([i j k],[p q])` die Teilmatrix bestehend aus der Schnittmenge der Zeilen `i`,`j` und `k` und den Spalten `p` und `q`.

```
>> A = [2 3 5; 7 11 13; 17 19 23]

A =

     2     3     5
     7    11    13
    17    19    23

>> A(2,1)

ans =

     7

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

ans =

    11    13
    19    23

>> A(:,1)

ans =

     2
     7
    17

>> A(2,:)

ans =
```

```
     7    11    13

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

ans =

     3     5
    19    23
```

Ein weiterer Spezialfall ist `A(:)`, welches einen Vektor aus allen Elementen von `A` zurückgibt, spaltenweise von der ersten zur letzten Spalte.

```
>> B = A(:)

B =

     2
     7
    17
     3
    11
    19
     5
    13
    23
```

Wird `A(:)` auf der linken Seite einer Zuweisung benutzt, so wird damit `A` gefüllt, unter Beibehaltung der Struktur von `A`. Mit dieser Notation ergibt sich eine weitere Möglichkeit unsere 3x3 - Matrix der ersten 9 Primzahlen zu erzeugen:

```
>> A = zeros(3); A(:) = primes(23); A = A'

A =

     2     3     5
     7    11    13
    17    19    23
```

Die Funktion `primes` gibt einen Vektor von Primzahlen zurück die kleiner oder gleich dem Eingabeargument sind. Die Transposition `A = A'` ist nötig um die Prinzahlen entlang der Zeilen anstatt entlang der Spalten anzuordnen.
Der Doppelpunktnotation verwandt ist die Funktion `linspace`, die als Eingabe die Anzahl der zu erzeugenden Punkte verlangt anstatt des Abstandes. `linspace(a,b,n)` erzeugt `n` Punkte gleichen Abstands zwischen `a` und `b`. Der Standardwert für `n` ist 100.

```
>> linspace(-1,1,9)
```

```
ans =

  Columns 1 through 7

   -1.0000   -0.7500   -0.5000   -0.2500         0    0.2500    0.5000

  Columns 8 through 9

    0.7500    1
```

Die Notation `[]` bezeichnet eine leere 0×0-Matrix. Weist man einer Zeile oder Spalte einer Matrix den Wert `[]` zu, so wird sie aus der Matrix gelöscht.

```
>> A(2,:) = []

A =

     2     3     5
    17    19    23
```

In diesem Beispiel würde der gleiche Effekt durch
`A = A([1 3],:)` erzielt. Die leere Matrix ist auch als Platzhalter in Argumentlisten nützlich.

2.3 Matrizen manipulieren

Es gibt mehrere Befehle für die Manipulation von Matrizen (weiterführende Befehle werden in einem späteren Kapitel behandelt). Die Funktion `reshape` ändert die Dimensionen einer Matrix: `reshape(A,m,n)` erzeugt eine $m \times n$ Matrix, deren Elemente spaltenweise aus `A` entnommen werden.

```
>> A = [1 4 9; 16 25 36], B = reshape(A,3,2)

A =

     1     4     9
    16    25    36


B =

     1    25
    16     9
     4    36
```

Die Funktion `diag` behandelt die Diagonale einer Matrix und kann sowohl Matrizen als auch Vektoren als Argument enthalten. Für einen Vektor `x` erzeugt `diag(x)` eine Diagonalmatrix mit der Diagonale `x`.

```
>> diag([1 2 3])
```

```
ans =

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

Allgemeiner legt `diag(x,k)` `x` auf die k-te Diagonale, wobei $k > 0$ Diagonalen über der Hauptdiagonalen beschreibt, und $k < 0$ die darunter ($k = 0$ bezeichnet die Hauptdiagonale).

```
>> diag([1 2],1)

ans =

     0     1     0
     0     0     2
     0     0     0

>> diag([3 4],-2)

ans =

     0     0     0     0
     0     0     0     0
     3     0     0     0
     0     4     0     0
```

Für eine Matrix `A` ist `diag(A)` der Spaltenvektor, der die Hauptdiagonale von `A` repräsentiert. Um eine Diagonalmatrix mit der gleichen Diagonale wie `A` zu erzeugen muss man also diag zweimal aufrufen: `diag(diag(A))`. Analog zur Vektornotiation gibt `diag(A,k)` die k-te Diagonale von `A` zurück.

```
>> A = [2 3 5; 7 11 13; 17 19 23]

A =

     2     3     5
     7    11    13
    17    19    23

>> diag(A)

ans =

     2
    11
    23

>> diag(A,-1)
```

```
ans =

     7
    19
```

Trianguläre Teile einer Matrix können mit `tril` und `triu` extrahiert werden. Der untere Dreiecksgehalt von `A` (die Elemente auf und unter der Hauptdiagonale) wird mit `tril(A)` bezeichnet, und der obere Dreiecksgehalt (die Elemente auf und über der Hauptdiagonale) mit `triu(A)`. Allgemeiner gibt `tril(A,k)` die Elemente auf und unter der k-ten Diagonale zurück und `triu(A,k)` die Elemente darauf und drüber.

```
>> tril(A)

ans =

     2     0     0
     7    11     0
    17    19    23

>> triu(A,1)

ans =

     0     3     5
     0     0    13
     0     0     0

>> triu(A,-1)

ans =

     2     3     5
     7    11    13
     0    19    23
```

Kapitel 3

Operatoren und Flusskontrolle

3.1 Relationale Operatoren

Die relationalen Operatoren von MATLAB sind

==	gleich
~=	ungleich
<	weniger als
>	größer als
<=	kleiner oder gleich
>=	größer oder gleich

Tabelle 3.1: Die relationalen Operatoren in MATLAB

Beachten Sie, dass in MATLAB ein einzelnes Gleichheitszeichen = eine Zuweisung angibt und nie auf Gleichheit testet. Vergleiche zwischen Skalaren erzeugen 1 wenn die Relation wahr und 0 wenn sie falsch ist. Vergleiche sind auch zwischen Matrizen gleicher Dimension definiert und zwischen Matrizen und Skalaren, deren Ergebnis in beiden Fällen eine Matrix mit Nullen und Einsen ist. Für Matrix-Matrix Vergleiche werden die entsprechenden Paare von Elementen verglichen, während für Matrix-Skalar Vergleiche der Skalar mit jedem Matrixelement verglichen wird.

```
>> A = [1 2; 3 4]; B = 2*ones(2);
>> A==B

ans =

     0     1
     0     0

>> A>2
```

```
ans =

     0     0
     1     1
```

Um zu testen ob zwei Matrizen A und B gleich sind, kann der Ausdruck `isequal(A,B)` verwendet werden. Die Funktion `isequal` is eine der vielen nützlichen logischen Funktionen, deren Namen mit `is` beginnt. Für eine Liste aller dieser Funktionen rufen sie in MATLAB `doc is` auf. Die Funktion `isnan` ist besonders wichtig, da der Test `x == NaN` immer das Ergebnis 0 (falsch) liefert, selbst wenn `x = NaN`! (Ein NaN ist gerade so definiert, dass er zu allem ungleich und ungeordnet ist).

3.2 Logische Operatoren

Die logischen Operatoren von MATLAB sind

&	logisches und
|	logisches oder
$\sim$	logisches nicht
Xor	logisches exklusives oder
All	wahr wenn *alle Elemente* eines Vektors von Null verschieden sind
Any	wahr wenn *wenigstens ein Element* eines Vektors von Null verschieden ist

Tabelle 3.2: Die logischen Operatoren in MATLAB

Wie die relationalen Operatoren produzieren &,| und $\sim$ Matrizen von Nullen und Einsen, falls eines der Argumente eine Matrix ist. Die Funktion `all` gibt, wenn sie auf einen Vektor angewendet wird, 1 zurück wenn alle Elemente des Vektors verschieden von 0 sind, und 0 sonst. Die `any` Funktion ist ebenfalls so definiert, wobei 'irgendeiner' hier 'alle' ersetzt. Beispiele:

```
>> x = [-1 1 1]; y = [1 2 -3];
>> x>0 & y>0

ans =

     0     1     0

>> x>0 | y>0

ans =

     1     1     1

>> xor(x>0,y>0)
```

```
ans =

     1     0     1

>> any(x>0)

ans =

     1

>> all(x>0)

ans =

     0
```

Beachte, dass `xor` als Funktion `xor(a,b)` aufgerufen werden muss. Die Operatoren `and`, `or`, `not` und die relationalen Operatoren können ebenfalls in funktionaler Form aufgerufen werden, `and(a,b)` ...(mehr dazu unter
`help ops`).

Bei Matrizen gibt `all` einen Zeilenvektor zurück der die Ergebnise von `all` angewendet auf die Spaltenvektoren enthält. Also ist `all(all(A==B))` eine weitere Möglichkeit um die beiden Matrizen A und B auf Gleichheit zu testen. Die Funktion `any` arbeitet entsprechend. Zum Beispiel hat der Ausdruck `any(any(A==B))` den Wert 1, wenn `A` und `B` in wenigstens einem Element übereinstimmen, und 0 sonst.

Der Befehl `find` gibt die Indizes der von Null verschiedenen Einträge eines Vektors zurück.

```
>> x = [-3 1 0 -inf 0];
>> f=find(x)

f =

     1     2     4
```

Das Ergebnis von `find` kann dann benutzt werden, um genau diese Elemente des Vektors auszugeben.

```
>> x(f)

ans =

    -3     1   -Inf
```

Zu dem gegebenen x kann also `find` benutzt werden, um die endlichen Einträge des Vektors x zu finden

```
>> x(find(isfinite(x)))

ans =

    -3     1     0     0
```

Und die negativen Elemente auf 0 zu setzen:

```
>> x(find(x<0))=0

x =

     0     1     0     0     0
```

Wenn `find` auf eine Matrix angewendet wird, bezieht sich der Indexvektor auf `A` als Vektor seiner Spaltenvektoren (also `A(:)`), und dieser Vektor kann wiederum verwendet werden, um Elemente in `A` anzusprechen. Im folgenden Beispiel benutzen wir `find` um alle die Elemente von `A` auf 0 zu setzen, die kleiner sind als die entsprechenden Elemente von `B`:

```
>> A=[4 2 16; 12 4 3], B = [12 3 1; 10 -1 7]

A =

     4     2    16
    12     4     3

B =

    12     3     1
    10    -1     7

>> f = find(A<B)

f =

     1
     3
     6

>> A(f) = 0

A =

     0     0    16
    12     4     0
```

Eine andere Methode um `find` zu benutzen ist, die Funktion mit `[i,j] = find(A)` zu benutzen. Dann werden in i die Zeilen- und in j die Spaltenindizes der von 0 verschiedenen Elemente abgespeichert.

Die Rückgabewerte von MATLABs relationalen und logischen Operatoren sind Arrays bestehend aus Nullen und Einsen. Diese sind Beispiele für logische Arrays. Logische Arrays können ebenfalls erzeugt werden, indem man den Befehl `logical` auf ein numerisches Array anwendet. Logische Arrays können zur Indizierung verwendet werden.

```
>> y=[1 2 0 -3 0]
```

```
y =

     1     2     0    -3     0

>> i1 = logical(y)

i1 =

     1     1     0     1     0

>> i2 = (y~=0)

i2 =

     1     1     0     1     0

>> i3 = [1 1 0 1 0]

i3 =

     1     1     0     1     0

>> whos
  Name      Size         Bytes  Class

  i1        1x5             40  double array (logical)
  i2        1x5             40  double array (logical)
  i3        1x5             40  double array
  y         1x5             40  double array

Grand total is 20 elements using 160 bytes

>> y(i1)

ans =

     1     2    -3

>> y(i2)

ans =

     1     2    -3

>> isequal(i2,i3)

ans =

     1
```

```
>> y(i3) ??? Index into matrix is negative or zero.  See release
notes on changes to logical indices.
```

Dieses Beispiel illustriert die Regel, dass `A(M)`, wenn `M` ein logisches Array ist, die Elemente von `A` zurückgibt, die zu den von Einträgen von `M` gehören, deren Realteil von 0 verschieden ist. Merke, dass obwohl `i2` die gleichen Elemente wie `i3` hat (und in MATLAB auch als identisch angesehen wird), nur `i2` zur Indizierung verwendet werden kann.

Ein Aufruf von `find` kann manchmal vermieden werden, wenn das Argument ein logisches Array ist. Im vorigen Beispiel kann `x(find(isfinite(x))` durch `x(isfinite(x))` ersetzt werden. Der Übersicht halber bietet sich allerdings `find` an.

3.3 Flusskontrolle

MATLAB hat vier Strukturen für die Flußkontrolle: die `if`-Abfrage, die `for`-Schleife, die `while`-Schleife und den `switch`-Befehl. Die einfachste Form der `if`-Abfrage lautet:

```
if expression
     statements
end
```

Statements werden ausgeführt, wenn die Realteile der Elemente von expression alle verschieden von 0 sind. Der folgende Code ersetzt zum Beispiel x und y wenn x größer ist als y:

```
if x > y
     temp = y;
     y = x;
     x = temp;
end
```

Folgen auf einen `if`-Befehl auf der gleichen Zeile weitere Befehle, so müssen diese über ein Komma getrennt werden, um den `if`-Befehl vom nächsten Befehl zu trennen.

```
>> if x > 0, sqrt(x);end
```

Befehle die nur ausgeführt werden sollen, wenn `expression` falsch ist, können nach `else` eingefügt werden.

```
>> e = exp(1); >> if 2^e > e^2
      disp('2^e is bigger')
  else
      disp('e^2 is bigger')
```

Schließlich kann mit `elseif` ein weiterer Test hinzugefügt werden mit (beachte, dass zwischen `else` und `if` kein Leerzeichen stehen darf):

```
if isnan(x)
   disp('Not a Number')
elseif isinf(x)
   disp('Plus or minus infinity')
else
   disp('A ''regular'' floating point number')
```

Bei einer `if`-Abfrage der Form " `if` Bedingung 1 & Bedingung 2" wird `Bedingung 2` nicht ausgewertet, wenn `Bedingung 1` falsch ist (dies wird "early return" `if` Auswertung genannt). Dies ist nützlich, wenn die Auswertung von `Bedingung 2` ansonsten einen Fehler produzieren könnte, zum Beispiel durch einen Index außerhalb des zulässigen Bereichs oder eine nicht definierte Variable.

Die `for`-Schleife ist einerseits eine der wichtigsten Strukturen in MATLAB, andererseits vermeiden viele Programmierer, die kurzen und schnellen Code produzieren wollen, dessen Verwendung. Die Syntax lautet:

```
for variable = ausdruck
     statements
end
```

Normalerweise ist `ausdruck` ein Vektor der Form `i:s:j`. Die Befehle werden für jedes Element von `ausdruck` ausgeführt, wobei `variable` dem entsprechenden Element von `ausdruck` zugeordnet ist. Zum Beispiel wird die Summe der ersten 25 Elemente der harmonischen Folge 1/i erzeugt durch:

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

s =

    3.8160
```

Eine weitere Möglichkeit um `ausdruck` zu definieren ist, die Klammernotation zu verwenden.

```
>> for x = [pi/6 pi/4 pi/3], disp([x,sin(x)]), end
    0.5236    0.5000

    0.7854    0.7071

    1.0472    0.8660
```

Mehrere `for`-Schleifen können verschachtelt werden. In diesem Fall hilft einrücken, um die Lesbarkeit des Codes zu erhöhen. Der folgende Code bildet die symmetrische 5x5-Matrix $A = (a_{ij})$ mit $a_{ij} = i/j$ für $j >= i$:

```
>> n = 5; A = eye(n);
>> for j=2:n
      for i = 1:j-1
          A(i,j) = i/j;
          A(j,i) = i/j;
      end
   end
```

Der `ausdruck` in der `for`-Schleife kann eine Matrix sein, in diesem Fall wird `variable` der Spaltenvektor zugewiesen, vom Ersten zum Letzten. Zum Beispiel, um x in der Reihenfolge auf jeden Einheitsvektor zu setzen, kann man
`for x = eye(n),...,end` schreiben.
Die `while`-Schleife hat die Form

```
while ausdruck
     statements
end
```

Die Befehle werden ausgeführt, so lange `ausdruck` wahr ist. Das folgende Beispiel nähert die kleinste von Null verschiedene Gleitkommazahl an:

```
>> x=1; while x>0, xmin = x; x = x/2; end, xmin

xmin =

  4.9407e-324
```

Eine `while`-Schleife kann mit dem Befehl `break` beendet werden, der die Kontrolle an den ersten Befehl nach dem entsprechenden end zurückgibt. Eine unendliche Schleife kann mit `while 1, ..., end` erzeugt werden. Dies kann nützlich sein, wenn es ungünstig ist, den Abbruchtest an den Anfang der Schleife zu setzen. (Merke, dass MATLAB im Gegensatz zu manch anderen Sprachen keine "repeat-until" Schleife hat.)
Das vorige Beispiel lässt sich weniger verständlich folgendermaßen notieren:

```
x = 1;
while 1
   xmin = x;
   x = x/2;
   if x == 0, break, end
end
xmin
```

Der Befehl `break` kann auch verwendet werden, um eine `for`-Schleife zu verlassen. In einer verschachtelten Schleife führt ein `break` zum Verlassen der Schleife auf der nächsthöheren Ebene. Der Befehl `continue` setzt die Ausführung sofort in der nächsten Iteration der `for`- oder `while`-Schleife fort, ohne die verbleibenden Befehle in der Schleife auszuführen.

```
for i=1:10
   if i < 5, continue, end
   disp(i)
end
```

In komplexeren Schleifen kann `continue` verwendet werden um umfangreiche `if`-Abfragen zu vermeiden.
Zuletzt bleibt noch der Kontrollbefehl `switch`. Er besteht aus "`switch Ausdruck`", gefolgt von einer Liste von
"`case Ausdruck Befehl`", die optional mit

"`otherwise Ausdruck`" enden und mit einem `end` beendet werden. Der Ausdruck bei `switch` wird ausgewertet und die Befehle nach dem ersten passenden `case` Ausdruck werden ausgeführt. Falls keiner der Fälle passt, werden die Befehle nach `otherwise` ausgeführt. Das folgende Beispiel wertet die p-Norm eines Vektors x aus (also `norm(x,p)`):

```
switch p
    case 1
    y = sum(abs(x));
    case 2
    y = sqrt(x'*x);
    case inf
    y = max(abs(x));
    otherwise
    error('p must be 1, 2 or inf.')
end
```

Der Ausdruck nach `case` kann eine Liste von Werten sein, die in geschweiften Klammern eingeschlossen ist (ein Cell-Array). In diesem Fall kann der `switch`-Ausdruck zu jedem Element der Liste passen.

```
x = input('Enter a real number: '); switch x
    case {inf,-inf}
    disp('Plus or minus infinity')
    case 0
    disp('Zero')
    otherwise
    disp('Nonzero and finite')
end
```

C Programmierer seien darauf hingewiesen, dass das `switch`-Konstrukt von MATLAB sich anders verhält als das von C: sobald ein MATLAB case Ausdruck zur Variable passt und die Befehle ausgeführt wurden, wird die Kontrolle an den ersten Befehl nach dem `switch`-Block übergeben, ohne dass weitere `break` Befehle nötig sind.

Kapitel 4

M-Dateien

Viele nützliche Berechnungen lassen sich über die Kommandozeile von MATLAB durchführen. Nichtsdestotrotz wird man früher oder später M-Dateien schreiben müssen. Diese sind das Pendant zu Programmen, Funktionen, Subroutinen und Prozeduren in anderen Programmiersprachen. Fügt man eine Folge von Befehlen zu einer M-Datei zusammen, ergeben sich vielfältige Möglichkeiten, wie etwa

- an einem Algorithmus herzumzuexperimentieren indem man eine Datei bearbeitet, anstatt eine lange Liste von Befehlen wieder und wieder zu tippen
- einen dauerhaften Beleg für ein numerisches Experiment schaffen
- nützliche Funktionen aufzubauen die zu einem späteren Zeitpunkt erneut verwendet werden können
- M-Dateien mit anderen Kollegen austauschen

Im Internet findet man eine Vielzahl nützlicher M-Dateien, die von Benutzern geschrieben wurden. Eine M-Datei ist eine Textdatei mit der Dateiendung `.m`, die MATLAB Befehle enthält. Es gibt zwei Arten:

- **Skriptdateien** (oder Kommandodateien) haben keine Ein- oder Ausgabeargumente und operieren auf Variablen im Workspace.
- **Funktionsdateien** enthalten eine function Definitionszeile und akzeptieren Eingabeargumente und geben Ausgabeargumente zurück, und ihre internen Variable sind lokal auf die Funktion beschränkt (sofern sie nicht als global deklariert wurden).

4.1 Skriptdateien

Ein Skript erlaubt es, eine Folge von Befehlen die wiederholt verwendet werden sollen oder in Zukunft noch gebraucht werden. Ein Beispiel für ein Skript ist folgendes "Eigenwertroulette":

```
%SPIN
% Counts number of real eigenvalues of random matrix.
A = randn(8); sum(abs(imag(eig(A)))<.0001)
```

Das Beispiel erzeugt eine zufällige normalverteilte 8x8-Matrix und zählt die Eigenwerte, deren Imaginärteile unter einer willkürlich gewählten Grenze liegen.

Die ersten beiden Zeilen des Skripts beginnen mit dem % Symbol und sind daher Kommentare. Sobald MATLAB auf ein % trifft ignoriert es den Rest der Zeile. Dies erlaubt das Einfügen von Text, der das Skript für Menschen leichter verständlich macht. Angenommen dieses Skript wurde unter dem Namen `spin.m` gespeichert, dann ist die Eingabe von `spin` an der Kommandozeile gleichwertig zur Eingabe der zwei Befehle `A = randn(8);` und `sum(abs(imag(eig(A)))<0.0001)`. Dies „dreht das Roulette-Rad“, und produziert eine der fünf Antworten 0,2,4,6 oder 8. Jeder Aufruf von `spin` erzeugt eine andere zufällige Matrix und kann daher eine andere Antwort erzeugen.

```
>> spin

ans =

     2

>> spin

ans =

     4
```

4.2 Funktionsdateien

Selbst geschriebene Funktionsdateien erweitern den Umfang von MATLAB. Sie werden auf die gleiche Weise verwendet wie die bereits existierenden MATLAB Funktionen wie `sin`, `eye`, `size` usw.

Hier ist ein Beispiel für eine Funktionsdatei:

```
function y = maxentry(A)
%MAXENTRY   Largest absolute value of matrix entries.
%           MAXENTRY(A) is the maximum of the absolute values
%           of the entries of A.

y = max(max(abs(A)));
```

Dieses Beispiel präsentiert mehrere Features. Die erste Zeile fängt mit dem Schlüsselwort `function` an, gefolgt vom Ausgabeargument y, und dem Gleichheitszeichen. Rechts von `=` kommt der Funktionsname `maxentry`, gefolgt vom Eingabeargument A in Klammern. (Im Allgemeinen kann es beliebig viele Ein- und Ausgabeargumente geben.) Der Funktionsname muss der gleiche sein wie der Name der M-Datei, die die Funktion enthält - in diesem Fall muss die Datei `maxentry` heißen.

Die zweite Zeile der Funktionsdatei heißt H1-Zeile (help 1). Sie sollte eine Kommentarzeile sein: Eine Zeile die mit einem %-Zeichen beginnt und danach ohne Leerzeichen dem Funktionsname in Großbuchstaben – gefolgt von einem oder mehr Leerzeichen und dann einer kurzen Beschreibung. Die Beschreibung sollte mit einem Großbuchstaben anfangen und mit einem Punkt enden und dabei auf die Worte „der“, „die“, „das“ und „ein“, „eine“ verzichten. Alle Kommandozeilen – beginnend mit der ersten bis zur ersten Nichtkommentarzeile (in der Regel eine Leerzeile, um die Lesbarkeit des Codes zu erhöhen) – werden beim Aufruf von `help function_name` angezeigt. Darum sollten diese Zeilen die Funktion und ihre Argumente beschreiben. Funktionenname und -argumente groß zu schreiben ist eine allgemein akzeptierte Konvention. Im Falle von `maxentry` sieht die Ausgabe folgendermaßen aus

```
>> help maxentry

 MAXENTRY   Largest absolute value of matrix entries.
            MAXENTRY(A) is the maximum of the absolute values
            of the entries of A.
```

An dieser Stelle sei noch einmal mit Nachdruck darauf hingewiesen, dass es sich lohnt, alle Funktionsdateien auf diese Art und Weise zu dokumentieren. Oft ist es nützlich, in Kommentaren anzugeben, wann die Funktion zuerst geschrieben wurde, und ob Veränderungen hinzugefügt wurden.

Der Befehl `help` funktioniert auf ähnliche Art und Weise mit Skriptdateien, er zeigt die Anfangsfolge an Kommentaren an.

Die Funktion `maxentry` wird wie jede andere MATLAB Funktion aufgerufen:

```
>> maxentry(1:10)

ans =

    10

>> maxentry(magic(4))

ans =

    16
```

4.3 M-Dateien bearbeiten

Es gibt zwei Möglichkeiten, M-Dateien zu bearbeiten. Jeder beliebige ASCII-Editor kann verwendet werden (benutzt man ein Textverarbeitungsprogramm, so ist sicherzustellen, dass das Programm die Datei auch wieder im ASCII-Format speichert). Oder man benutzt den internen Editor/Debugger von MATLAB. Dieser Editor verfügt über Funktionen, die das Bearbeiten von M-Dateien unterstützen. Zum Beispiel werden Schleifen und if-Abfragen automatisch eingerückt, es gibt mehrfarbiges Syntax Highlighting, und einen automatischen Vergleich von Klammern und Doppelpunkten. Diese und andere Features können im Menü Tools-Optionen deaktiviert oder modifizert werden.

4.4 Mit M-Dateien und der Pfadvariable von MATLAB arbeiten

Viele MATLAB-Funktionen sind M-Dateien die auf der Festplatte gespeichert werden, während andere in den MATLAB-Interpreter integriert sind. Der MATLAB Suchpfad ist eine Liste von Ordnern, die angibt, wo MATLAB nach M-Dateien sucht. Eine M-Datei ist nur dann verfügbar, wenn sie sich im Suchpfad befindet. Geben Sie `path` ein, um den aktuellen Suchpfad angezeigt zu bekommen. Die Pfadvariable kann über `path` gesetzt werden, und über `addpath` können Befehle hinzugefügt werden. Eine weitere Möglichkeit zur Bearbeitung des Suchpfads ist der Pfad-Browser, der über Datei-Pfad Einstellen oder die Eingabe von pathtool aufgerufen wird.

Der Suchpfad kann mit mehreren Befehlen durchsucht werden. Der Befehl `what` listet alle MATLAB Dateien im aktuellen Verzeichnis auf;
`what verzeichnisname` listet alle MATLAB-Dateien im Ordner `verzeichnisname` im Pfad. Der Befehl `lookfor keyword` durchsucht den Pfad nach M-Dateien, die keyword in ihrer H1 Zeile enthalten (die erste Zeile des Hilfetextes). Um alle Kommentarzeilen, die bei Eingabe von help angezeigt werden, zu durchsuchen, kann man `lookfor` mit dem Parameter `-all` aufrufen: `lookfor keyword -all`

Manche MATLAB-Funktionen benutzen Kommentarzeilen nach dem ersten Block von Kommentaren, um weitere Informationen zu liefern, wie etwa bibliographische Angaben (zum Beispiel `fminsearch`). Auf diese Information kann über `type` zugegriffen werden. Sie wird beim Aufruf von help nicht angezeigt.

Die Eingabe von `which foo` zeigt den Pfad der Funktion `foo` an, oder gibt an, dass die Funktion built-in ist, oder nicht vorhanden. Diese Funktion ist nützlich falls man wissen möchte, in welchem Ordner sich eine bestimmte Funktion befindet. Falls mehr als eine Funktion mit dem Namen `foo` im Suchpfad existiert, zeigt `which foo -all` alle Funktionen an.

Ein Skript, welches sich nicht auf dem Suchpfad befindet kann über den Befehl `run` gefolgt vom vollständigen Pfad des Skriptes aufgerufen werden. Eine M-Datei `foo.m` kann über `type foo` oder `type foo.m` am Bildschirm ausgegeben werden. (Falls es eine ASCII-Datei mit dem Namen `foo` gibt, so würde der erste Befehl `foo` ausgeben anstatt `foo.m`) Folgt `type` auf den Befehl `more on`, so wird die Datei seitenweise angezeigt (more off schaltet die seitenweise Darstellung wieder aus).

Erstellt man eine neue M-Datei, sollte man vermeiden, einen Namen zu wählen, der im Suchpfad bereits vorhanden ist. Dies kann man mit Hilfe der Befehle `which`, `type`, `help` oder `exist` überprüfen. Der Befehl `exist('name')` überprüft, ob `name` eine Variable im Workspace, eine Datei (mit mehreren möglichen Endungen, unter Anderem `.m`) im Suchpfad, oder ein Ordner ist. Das Ergebnis 0 bedeutet, dass keine Übereinstimmungen gefunden wurden, währen die Zahlen 1-7 bedeuten, dass es welche gibt. Für die Bedeutung der Zahlen siehe `help exist`.

Wird eine Funktion, die sich im Suchpfad befindet, zum ersten Mal aufgerufen, wird sie in den Speicher geladen. MATLAB kann im Allgemeinen feststellen, ob eine Funktionsdatei sich geändert hat, und kompiliert sie automatisch erneut, wenn sie aufgerufen wird. Um die Funktion `fun` aus dem Speicher zu löschen, benutzt man den Befehl `clear fun`. `clear functions` löscht alle Funktionen.

4.5 Dualität von Befehl und Funktion

Normalerweise werden vom Benutzer geschriebene Funktionen so aufgerufen: der Funktionenname gefolgt von einer Liste von Argumenten in Klammern. Doch manche MATLAB-internen Funktionen, wie zum Beispiel `type` und `what`, die im vorhergehenden Abschnitt behandelt wurden, werden normalerweise so aufgerufen, dass die Argumente vom Funktionennamen durch ein Leerzeichen getrennt werden. Diese ist keine Inkonsistenz von MATLAB sondern ein Beispiel für die Gleichheit von Befehl und Funktion.

```
function comfun(x,y,z)
%COMFUN Illustrative function with three string arguments.
disp(x), disp(y), disp(z)
```

Die Funktion `comfun` kann mit den Stringargumenten in Klammern (funktionale Form) oder mit den Stringargumenten getrennt durch Leerzeichen aufgerufen werden (Befehlsform):

```
>> comfun('ab','cd','ef')
ab
cd
ef
>> comfun ab cd ef
ab
cd
ef
```

Die beiden Funktionsaufrufe sind gleichwertig. Andere Beispiele für die Dualität von Befehl und Funktion sind

```
format long, format('long')
disp('Hello'), disp Hello
diary mydiary, diary('mydiary')
warning off, warning('off')
```

Beachte, dass die Befehlsform nur dann verwendet werden sollte, wenn die Eingabeparameter der Funktion Strings sind.

Kapitel 5

Lineare Algebra mit MATLAB

5.0.1 Norm und Kondition

Befehl:

```
norm(A,p)
```

wobei A eine m x n Matrix und $p = 1, 2, inf$ oder 'fro' ist.

p	Bedeutung
1	1-Norm, d.h. max. Spaltensumme
2	Spektralnorm, max. Singulärwert, (default)
inf	Unendlichnorm, d.h. max. Zeilensumme
fro	Frobeniusnorm, definiert als $\sum_{i=1}^{m} \sum_{i=1}^{m} a_{i,j}$

Für höherdimensionale Matrizen sind Normberechnung für $p = 2$ relativ aufwendig und kostenintensiv – daher werden oft Abschätzungen verwendet.

```
normest(A,tol)
```

Mit Hilfe von `normest` wird die 2-Norm von A bis auf die durch `tol` gegebene Genauigkeit berechnet. Defaultwert für `tol=1e-16`.

Oft werden lineare Gleichungssystem auf das Verhalten von Störungen untersucht (so genannten Stabilitätsanalyse). Ein Indikator hierfür ist die Kondition einer Matrix A, definiert als

$$\kappa_p(A) = \|A\|_p \cdot \|A^{-1}\|_p$$

Je nachdem ob κ groß bzw. klein ist, ist das System schlecht oder gut konditioniert.

Befehl:

```
cond(A,p)
```

p ist wie für `norm` definiert.

Anmerkung:

Normalerweise wird die Kondition nur für quadratische, nicht singuläre Matrizen berechnet. Für $p = 2$ kann auch die Kondition von rechteckigen Matrizen berechnet werden, wobei dann $A^(- 1)$ durch die Pseudoinverse ersetzt wird.

Auch für die Kondition existieren Abschätzungsfunktionen, wie `condest` und `rcond`:

```
[c,v]=condest(A)
```

c	Skalar
v	Vektor

mit:

$$c \leq \kappa_1(A)$$

und $\|A \cdot v\|_1 = \frac{\|A\|_1 * \|v\|_1}{c}$

Befehl:

```
rcond(A)
```

$$\frac{1}{\kappa_1}(A) \approx rcond(A)$$

wobei für A singulär `rcond(A)=0`.

5.1 Lineare Gleichungssysteme

Ein lineares Gleichungssystem sieht grundsätzlich folgendermaßen aus:

$$Ax = b$$

mit A eine Matrix und b Vektor. Gesucht wird ein Vektor x der das System löst.

5.1.1 A $n \times n$ Matrix

Falls A eine quadratische, nicht singuläre Matrix ist, existiert eine eindeutige Lösung. In Matlab wird diese wie folgt mit Hilfe des '\' berechnet.

```
x=A \ b
```

5.1.2 A $m \times n$ Matrix mit $m > n$

Wenn A eine rechteckige Matrix ist, die mehr Zeilen als Spalten hat, spricht man von einem überbestimmten System, da mehr Bedingungen (Zeilen) als Unbekannte (Spalten) vorhanden sind. Eine eindeutige Lösung, so dass $A \cdot x = b$, ist im Allgemeinen nicht möglich. Es kann nur eine Minimallösung angegeben werden, so dass $\|A * x - b\|_2 = min!$ ist. (vgl. Numerik 1. „Lineare Ausgleichsrechnung“)

In diesem Fall wird durch den '\' Operator das lineare Ausgleichsproblem gelöst, mit Hilfe der „least square solution“. Ist man nur an den nicht negativen Lösungen interessiert, ist der Befehl `lsqnonneg` zu empfehlen.

```
x=lsqnonneg(A,b)
```

wobei x nur Einträge größer, gleich null enthält.

5.1.3 A $m \times n$ Matrix mit $m < n$

Falls A eine Matrix mit weniger Zeilen als Spalten ist, so ist das LGS unterbestimmt und es existieren entweder keine oder unendlich viele Lösungen. Falls das System unendlich viele Lösungen besitzt, gibt '`A\b`' eine Lösung aus, die k nichtnull Elemente besitzt, wobei k der Rang der Matrix A ist. Sollte das System keine Lösungen haben gibt '`A\b`' eine Minimallösung aus, wie bei oben genannten überbestimmten Systemen.

Falls man mehrere Gleichungssysteme, mit der selben Matrix A aber mit verschiedenen Seiten $b^{(1)}, \ldots, b^{(n)}$ auf einmal lösen möchte, geschieht das am einfachsten dadurch, dass man die $b^{(i)}$s als Spalten einer Matrix B anordnet und den '\' Operator mit dieser Matrix aufruft. Das Ergebnis ist ein Matrix X mit $x^{(1)}, \ldots x^{(n)}$ als Spalten, die das jeweilige System zu gegebenem $b^{(i)}$ lösen.

5.2 Inverse, Pseudoinverse und Determinante

In manchen Fällen – nicht oft, aber hin und wieder, vor allem in Hausaufgaben – muss man die Inverse einer Matrix berechnen. In Matlab geht das mit Hilfe des 'inv' Befehls:

```
X=inv(A)
```

mit $A * X = X * A = I$, wobei A eine nicht singuläre, quadratische $n \times n$ Matrix sein muss.

Eine Matrix A ist singulär, falls die Determinante von $A = 0$ ist.

Befehl:

```
det(A)
```

mit A einer $n \times n$ Matrix.

Für gewisse Zwecke, z.B. zur Lösung eines least squares Problem ist die so genannte Pseudoinverse einer Matrix A von Interesse. Die Pseudoinverse X von A erfüllt folgende Bedingungen:

$$XAX = X$$
$$AXA = A$$

$$(XA)^T = XA$$
$$(AX)^T = AX$$

Befehl:

```
pinv(A)
```

wobei A hier nun auch eine rechteckige $m \times n$ Matrix sein kann.

5.3 Zerlegungen

Oben wurde bereits erwähnt, dass der '\' Operator lineare Gleichungssysteme löst, bzw. Näherungslösungen dazu angibt. Hierbei wird die Matrix A in eine untere Dreiecksmatrix L und eine obere Dreiecksmatrix R zerlegt ,sodass gilt $L \cdot R = A$.

5.3.1 LR-Zerlegung

Befehl:

```
[L,R]=lu(A) \\ [L,R,P]=lu(A)
```

L	untere Dreiecksmatrix mit 1 auf der Hauptdiagonale
R	obere Dreiecksmatrix

wobei A eine quadratische $n \times n$ Matrix ist.

Nicht jede Matrix A kann auf diese Weise zerlegt werden, es existieren Matrizen bei denen 'Pivoting' notwendig ist. Hierbei gibt der `lu` Befehl eine zusätzliche Permutationsmatrix P aus, sodass gilt: $P * A = L * R$. Wird P bei der Rückgabe von `lu` nicht aufgefangen so wird $L = P^T \cdot L$.

Die Verwendung des '\' Operators ist äquivalent zu:

```
[L,R]=lu(A);
x=R\(L \ b);
```

5.3.2 Cholesky-Zerlegung

Falls A eine symmetrische, positiv definite (s.p.d.) Matrix ist, d.h. sie genügt:

1. $A^T = A$
2. $\forall x : x^T A x \geq 0$

existiert, die Cholesky Zerlegung von A in eine obere Dreiecksmatrix R, so dass gilt:

$$A = R^T \cdot R$$

Anmerkung:

In vielen Lehrbüchern wird die Choleskyzerlegung mit unteren Dreicksmatrizen L beschrieben. Beide Formulierungen sind äquivalent, da für $L = R^T$ gilt:

$$A = L^T \cdot L$$

Befehl:

```
R=chol(A)
[R,p]=chol(A)
```

wobei A eine quadratische $n \times n$ Matrix ist. Falls A nicht positiv definit ist, ist p eine ganze Zahl größer 0, andernfalls gleich 0.

5.3.3 QR-Zerlegung

Die oben genannten Zerlegungen sind jeweils nur für quadratische Matrizen geeignet. Die QR Zerlegung jedoch arbeitet auch auf rechteckigen m x n Matrizen A , wobei A in eine orthogonale m x m Matrix Q und eine obere m x n Dreiecksmatrix R zerlegt wird.

```
[Q,R]=qr(A) \\ [Q,R]=qr(A,0) \\ [Q,R,P]=qr(A)
```

Falls m>n ist und die Variante `qr(A,0)` benutzt wird, ist Q eine spaltenorthogonale m x n und R eine n x n Matrix. Falls das Pivoting genutzt wird (`[Q,R,P]=qr(A)`) gilt: $AP = QR$ wobei dann die Diagonalelemente von R dem Betrag nach, abfallend geordnet sind.

Die QR-Zerlegung findet z.B. in der linearen Ausgleichsrechnung oder bei der Konstruktion von Orthonormalbasen Verwendung.

5.3.4 Singulärwertzerlegung

Ähnlich den Eigenwerten, charakterisieren die so genannten Singulärwerte unterschiedliche Eigenschaften einer Matrix. Jedoch sind für Eigenwerte quadratische Matrizen notwendig, die Singulärwertzerlegung hingegen arbeitet auch mit rechteckigen m x n Matrizen A. Es gilt:

$$A = U \cdot \Sigma \cdot V$$

mit U m x m und V n x n unitären Matrizen Σ m x n diagonal Matrix mit σ_i (Singulärwerte) auf der Hauptdiagonalen, wobei gilt:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{min(m,n)}$$

Befehl:

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

Anmerkung:

Mit Hilfe der Singulärwertzerlegung ist es möglich, verschiedene Dinge über die Matrix A auszusagen. So ist zum Beispiel die Anzahl der σ_i ungleich Null der Rang der Matrix A. Auch ist eine Orthonormalbasis des durch die Spalten von A aufgespannten Raumes in der Singulärwertzerlegung enthalten. Es sind die ersten k Spalten der Matrix U, wobei k der Rang von A ist.

5.4 Eigenwertpropleme

5.4.1 Eigenwerte

Sei A eine quadratische n x n Matrix. Dann heißt λ ein Eigenwert von A, wenn es einen vom Nullvektor verschiedenen Vektor x gibt mit der Eigenschaft $Ax = \lambda x$. Um diese Eigenwerte auszurechnen, berrechnet man normalerweise das charackteristische Polynom $p = det(I\lambda - A)$. Somit gilt:

$$p = p_n\lambda^n + \cdots + p_1\lambda + p_0$$

Die Koeffizenten $p_n, \cdots, p_0$ des charakteristischen Polynomes erhält man mit dem Befehl `poly(A)`.
Die Eigenwerte bekommt man mittels `e=eig(A)`. e ist ein Vektor der Länge n, wobei die Eigenwerte der Größe nach geordnet werden. Mit `[V,D]=eig(A)` bekommt man eine n x n Diagonalmatrix, die die Eigenwerte enthält, und eine n x n Matrix V, deren Spaltenvektoren, wenn möglich, eine Basis aus Eigenvektoren bilden, so dass $V^{-1}AV = D$.
Falls man nicht n linear unabhängige Eigenvektoren erhält, ist die Matrix V nicht regulär. Dies verdeutlichen wir mit dem folgenden Beispiel:

```
>> A=[2 -1;1 0]

A =
     2    -1
     1     0

>> [V,D]=eig(A)

V =
    0.7071    0.7071
    0.7071    0.7071

D =
     1     0
     0     1
```

Wie man an diesem Beispiel auch sieht, normiert MATLAB die Eigenvektoren immer bzgl. der 2-Norm.

Falls A eine hermitesche Matrix ist, d.h. $A = \overline{A}^T$, gibt es nur relle Eigenwerte und eine Orthonormalbasis aus Eigenvektoren. So eine Basis ist in V gespeichert.

```
>> [V,D]=eig([2 -1;-1 1])

V =
   -0.5257   -0.8507
   -0.8507    0.5257

D =
    0.3820         0
         0    2.6180
```

Um die Kondition der Berrechung der einzelnen Eigenwerte zu berechnen, ist der Befehl `[V,D,s]=condeig(A)` implementiert. Die i-te Stelle im Vektor s entspricht der Kondition des i-ten Eigenwert.

```
>> [V,D,s]=condeig([2 0.9;200 5]);
>> [diag(D)';s']

ans =
  -10.0000   17.0000
    7.4416    7.4416
```

Wenn man eine sehr große quadratische Matrix A hat und sich nur für die 6 größten Eigenwerte intressiert, kann man sich durch `eigs(A)` eine Approximation an diese Eigenwerte anzeigen lassen. Für weitere Informationen `help eigs`.

5.5 Matrixfunktionen

Für partielle Differentialgleichungen wird oft die Exponentialfunktion mit einer Matrix als Argument benötigt. Diese ist auch wie folgt definiert:

$$\exp(A) = \sum_{i=1}^{\infty} \frac{A^i}{i}$$

Von Matlab wird dies direkt unterstützt:

`expm(A)`

wobei A eine quadratische n x n Matrix ist

Anmerkung: Im Gegensatz zu der Funktion expm operiert exp nur auf den einzelnen Komponenten der Matrix A!

Weiterhin sind in Matlab folgenden Matrixfunktionen definiert:

- `logm`: Der Logarithmus einer Matrix als Umkehrfunktion der Exponentialfunktion, d.h. `logm(expm(A))=A`

- `sqrtm`: Quadratwurzel aus A, d.h. `B=sqtrm(A)` mit $B \cdot B = A$

- `funm`: Mit Hilfe von funm können Funktionen als Argument Matrizen übergeben werden. Zum Beispiel `funm(A,@sin)` berechnet den sinus der Matrix A.

Kapitel 6

Eigene Funktionen

6.1 Referenzen

Wie im vorhergehende Kapitel besprochen wurde, können mit Hilfe von `funm` Funktionen mit Matrizen als Argument ausgewertet werden. Hierzu wird eine so genannte Referenz auf eine Funktion übergeben. Diese Referenz ist nichts anderes als ein Verweis auf eine Funktion, die dann im Laufe des Aufrufs von funm mit feval ausgewertet wird.

`feval(@Funktionsname,`x_1`,...,`x_n`)`

`@Funktionsname` erzeugt den Verweis auf die Funktionen $x_1, \ldots, x_n$ sind die Argumente die an „Funktionsname“ zur Auswertung übergeben werden sollen.

In neueren Matlabversionen ist es möglich. Einer Referenz direkt die Argument zu übergeben, dies sollte allerdings aus Gründen der Abwärtskompatibilität zu älteren Matlabversionen unterlassen werden.

Beispiel:

- **neuere Matlabversion** :
  `func_handle=@test_function; func_handle(4)`
  erzeugt eine Funktionsreferenz auf `test_function` und wertet dann die Referenz an der Stelle 4 aus.
- **ältere Matlabversionen** :
  `func_handle=@test_function; feval(func_handle,4)`
  in älteren Versionen ist diese Vorgehensweise zwingend.

Diese Funktionsreferenzen können auch beim Schreiben eigener Funktionen, die z.B. auf Abbildungen operieren, verwendet werden. Meistens werden diese Verweise bei komplexeren Funktionen verwendet. Für einfache ist es auch möglich, sie ïnlineßu definieren.

```
f=inline('exp(a+log(x))/42','x','a')
```

Erzeugt eine Funktion, die mit $f(x_0, a_0)$ ausgewertet werden kann. Die Abfolge hinter der Funktionsvorschrift definiert die Reihenfolge der Argumente beim Funktionsaufruf.

Meistens geht man bei der Entwicklung von Funktionen von einem einfachen, meistens eindimensionalen Beispiel aus und versucht an Hand von diesem zu einer vektorwertigen Funktion zu gelangen. Dies geschieht in Matlab mit Hilfe des '`vectorize`' Befehls.

```
vectorize(FUNKTION)
```

Der Befehl erzeugt aus einer gegebenen Funktion eine neue, die auf komponentenweise auf den Elementen von Vektoren bzw. Matrizen operiert.

6.2 Subfunktionen

Zu einem 'sauberen' Programmierstil gehört, dass man sich wiederholende Operationen ausgliedert in so genannte Subfunktionen. Da diese in der Regel nur im Kontext des entsprechenden M-Files Sinn machen, werden sie einfach an die Hauptfunktion angefügt. Sie werden wie jede andere Funktion definiert durch:

```
function [RÜCKGABEWERTE]=FUNKTIONSNAME[ EINGABEPARAMETER]
```

Sie sind jedoch nur sichtbar für alle anderen Unterroutinen und die Hauptfunktion.

```
function y=Hauptfunktion(x)

% Hier wird etwas gemacht, wozu man immer wieder die Wurzel
% aus u+u^2 durch pi braucht. Der Aufruf erfolgt mit Subfunktion(u).

function t=Subfunktion(u)

t=sqrt(u+u^2)/pi;
```

6.3 Globale Variablen und Rekursion

Oft stellt sich das Problem, dass man Konstanten oder Verweise, in allen Funktionen zur Verfügung haben muss. Anstatt diese nun von Funktion zu Funktion immer und immer wieder zu übergeben werden sie als 'global' deklariert.

Falls man nun in all seinen Funktionen auf den Wert der Variable `GLOB_VAR` zugreifen will, deklariert man sie im Command Window mit '`global GLOB_VAR`' und verweist beim schreiben seiner Funktionen mit '`global GLOB_VAR`' darauf dass die verwendete Variable schon vorhanden ist. Allerdings ist auch Vorsicht geboten beim Umgang mit globalen Variablen, da sie nicht schreibgeschützt sind, d.h. es ist möglich ihren Wert in Funktionen zu verändern.

Rekursion bezeichnet einen Aufruf einer Funktion durch sich selbst. Am besten erklärt sich das an Hand eines Beispiels. Die Fakultät einer natürlichen Zahl n ist definert durch:

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

In Matlab mit Hilfe der Rekursion führt das zu folgendem:

```
function ret_val=fak(n)

if (n<=1)
     ret_val=1
else
    ret_val=n*(fak(n-1))
end
```

In Zeile 6 ruft sich die Funktion '`fak`' selbst auf, allerdings mit einem um 1 verminderten Argument.

Rekursive Programmierung ist allerdings auch mit Bedacht zu wählen, da sie zum einen speicheraufwendiger und zum anderen auch langsamer als iterative Methoden sein kann.

6.4 Programmierstil

Ein Beispiel:

```
function a=bla(b) blubb=ftt(pi)*sqrt(42);
i=f(b);a=ft(24)/i*exp(blubb)-sinh(43)/fun(16);

function c=f(b) c=(b/43)^2-norm(eye,inf);

function d=ftt(x) d=norm(exp(sqrt(-1)*x));

function x=ft(y) x=y;

function irgendwas=fun(nochwas) if nochwas<=1
irgendwas=(norm(eye(nochwas))*4)/sqrt(16); else
irgendwas=nochwas*(fun(nochwas-1)); end;
```

Das funktioniert – nach einem Monat allerdings kennt sich kein Mensch mehr ohne weiteres aus. Deswegen ein paar grundlegende Regeln, die es auch noch möglich machen, nach mehr als einer Woche seinen eigenen Code zu verstehen:

6.4.1 Kommentieren und Strukturieren

Kommentare können mit Hilfe von '%' in M-Files eingefügt werden. Es sollten am Anfang einer Funktion ein paar Informationen stehen, z.B. was für Eingabeparameter erwartet werden, was eigentlich berechnet werden soll, und was für eine Ausgabe erfolgen kann und wird. Weiterhin sollten wichtige Codeabschnitte kurz beschrieben werden. Auch hat es sich als sinnvoll erwiesen, eine gewisse Form zu wahren. Statt:

```
if a<=b, c=d; elseif a>=0, c=42; else, c=a; end
```

Besser:

```
if a<=b
    c=d;
elseif
```

```
    c=42;
else
    c=a;
end
```

6.4.2 Variablen und Funktionen sinnvoll benennen

Statt der dritten Funktion mit Namen `fun` (oder Abwandlungen wie `myfun`, `funky`, `fun3`, usw.) ist es besser, aussagekräftige Namen verwenden.

6.4.3 Planung

Nicht einfach drauf losprogrammieren – sondern davor kurz überlegen und sich ein Konzept zurechtlegen.

Kapitel 7

Ein- und Ausgabe

Im letzten Kapitel haben wir gelernt wie man eigene Funktionen korrekt in MATLAB schreibt – aber diese konnten weder Daten einlesen noch Daten ausgeben. Dieses Manko wollen wir in diesem Kapitel beseitigen, da insbesondere viele Funktion direkte Eingaben vom Benutzer erwarten.

7.1 Direkte Eingabe

Direkte Eingabe heißt, dass der Nutzer des Programmes aufgefordert wird, Daten einzugeben.

```
x=input('Beschriftungstext')
x=input('Beschriftungstext','s')
```

Numerische Werte werden mit Hilfe der ersten Variante eingelesen.

```
>> x=input('Bitte eine Zahl eingeben: ');
Bitte eine Zahl eingeben: 9.987
```

Falls eine Benutzereingabe in Form einer Zeichenkette gespeichert werden soll, muss die zweite Variante verwendet werden. Zur Abfrage von Koordinaten in einer graphischen Oberfläche wird die Funktion `ginput` verwendet.

```
[x,y] = ginput(n)
```

x	x Koordinaten relativ zu dem geklickten Fenster
y	y Koordinaten relativ zu dem geklickten Fenster
n	Anzahl der Klicks die abgefangen werden sollen

Im Vektor x werden folglich die x-Koordinaten der nächsten n Mausklicks gespeichert. Für den Vektor y gilt das entsprechende. Um zwischen zwei Befehlen n Sekunden zu warten, gibt es den MATLAB-Befehl `pause(n)`. Dies wird üblicherweise bei Anzeigen von verschiedenen Plots angewandt.

7.2 Bildschirmausgabe

Einfache Ausgaben können über den Befehl `disp` erfolgen:

```
disp('Zeichenfolge')
disp(a)
```

Bei Ausgabe einer Zeichenfolge muss disp verwendet werden, bei Aufruf von disp mit einer Variablen als Argument wird der Inhalt der Variablen formatiert ausgegeben (formatiert wie am interaktiven Prompt). Für komplexere und formatierte Ausgaben kann man den Befehl `fprintf` verwenden:

```
fprintf('FORMATSTRING',Wert)
```

FORMATSTRING	Ein Formatierungsstring der Form `%a.b(e/f)\n`.
`a`	Anzahl der Stellen vor dem Komma
`b`	Anzahl der Stellen hinter dem Komma
`e/f`	Exponential oder Gleitkommadarstellung
`\`	Nach Ausgabe eine neue Zeile beginnen.
Wert	Werte die ausgegeben werden sollen.

Ein Minus hinter dem % Zeichen erzwingt eine linksbündige Ausgabe:

```
fprintf('%-3.5f\n%3.5f\n', 9, 103)
    9.00000
    103.00000
```

Für die Ausgabe von mehreren Werten müssen diese in dem Formatstring gekennzeichnet werden. Es können auch Beschriftungen eingefügt werden z.B.:

```
fprintf('Skalarprodukt von [1 1] und [1 2]: %3.5f\n',[1,1]*[1;2])
    Skalarprodukt von [1 1] und [1 2]: 3.00000
```

Des weiteren ist es möglich, auch komplette Matrizen zeilenweise ausgeben zu lassen:

```
A=[1 2 3; 4 5 6; 7 8 9];
fprintf('a1=%g, a2=%g, a3=%g\n', A);

    a1=1, a2=4, a3=7
    a1=2, a2=5, a3=8
    a1=3, a2=6, a3=9
```

Um `'`, `%` und `\` in einem Formatstring auszugeben, müssen `''`, `\%` und `\\` verwendet werden. Für die kombinierte Ausgabe von Zahlen und Zeichenketten ist es auch möglich, die beiden zu verbinden, indem man die Zahlen in Strings umwandelt mit Hilfe von `num2str` bzw. `int2str`:

```
disp(['a', num2str(4)])
    a4
```

7.3 Indirekte Eingabe und Ausgabe

Indirekte Eingabe und Ausgabe heißt, dass man aus einer `*.txt`-Datei Daten einliest bzw. in diese Datei Daten, die z.B. eine Funktion geliefert hat, reinschreibt.

Zuerst muss man Matlab sagen, in welcher Datei es nach Daten suchen oder Daten schreiben soll. Dies wird durch den Befehl `fid=fopen('myoutput.txt','r/w')` erledigt, wobei `r` für das Einlesen steht bzw. `w` für das Eingeben von Daten. Mit `fprintf` kann man dann Daten in `myoutput.txt` schreiben.

```
>> fid=fopen('myoutput.txt','w');
>> a=[1 2 3 4];
>> for i=1:4
   fprintf(fid,'Das %g. Element des Vektors a: %g\n',i,a(i));
   end
>> fclose(fid)
```

Nun enthält `myoutput.txt` erwartungsgemäß folgendes:

```
Das 1. Element des Vektors a: 1
Das 2. Element des Vektors a: 2
Das 3. Element des Vektors a: 3
Das 4. Element des Vektors a: 4
```

Diese Datei kann folgendermaßen eingelesen werden:

```
>> fid=fopen('myoutput.txt','r');
>> X=fscanf(fid,'Das %g Element des Vektors a: %g\n')

X =

     1
     1
     2
     2
     3
     3
     4
     4

>> fclose(fid);
```

`scanf` funktioniert also folgendermaßen: Überspringe `'Das ` ; lese irgendeine floating point Zahl (`%g`) ein, überspringe wieder ` Element des Vektors a: ` und lese die zweite floating point Zahl ein. Schreibe alle Zahlen der Reihe nach in den Vektor X.

Mit `reshape` formt man X in eine Matrix um:

```
>> X=reshape(X,2,4)

X =

    1     2     3     4
    1     2     3     4
```

Wenn man nur eine Matrix mit lauter reelen Zahlen einlesen will, macht man dies folgendermaßen. In `Matrix.txt` steht z.B.

```
1 4 6 7
7 9 10 11
99 1 5 6
1 103 4 89
```

Nun muss man nur die Dimensionen der Matrix `fscanf` bekannt machen.

```
>> fid=fopen('Matrix.txt','r');
>> X=fscanf(fid,'%g',[4,4]);
>> X=X'

X =

     1     4     6     7
     7     9    10    11
    99     1     5     6
     1   103     4    89

>> fclose(fid)
```

Außerdem können Dateien, die nur Zahlen erhalten, einfacher mit `fread` und `fwrite` bearbeitet werden. Informieren Sie sich dazu in der Hilfe.

Kapitel 8

Optimierung von M-Files

Bei großen Problemen ist es sinnvoll und notwendig, Algorithmen effizient zu implementieren. Durch die Beachtung einiger Regeln kann man enorme Zeitgewinne erreichen (vor allem in älteren MATLAB-Versionen ohne JIT-Compiler).

8.1 Vektorisierung

In MATLAB sind Matrix- und Vektoroperationen annähernd optimal implementiert. Deshalb sollte man so oft wie möglich auf sie zurückgreifen, insbesondere for-Schleifen sollten möglichst ersetzt werden. Dazu ein Beispiel:

```
>> n = 5e5; x = randn(n,1);
>> tic, s = 0; for i=1:n, s = s+x(i)^2; end, toc
elapsed time is 1.532000 seconds
>> tic t=sum(x.^2); toc
elapsed time is 0.046000 seconds
>>s-t
ans = 0
>> 0.046/1.532
ans = 0.0300
```

Offensichtlich wird in beiden Fällen die Summe der Quadrate der Elemente von x berechnet, beide Male mit dem selben Ergebnis. Der Zeitgewinn ist allerdings phänomenal: Durch die vektorielle Implementierung benötigt man lediglich 3% der Zeit der intuitiven Version mittels einer for-Schleife.

Weitere Beispiele:

- Berechnung der Fakultät:

```
>> n=1e7;
>> p1=1; for i=1:n, p=p*i; end %9.625 sec
>> p2=prod(1:n) %0.531 sec
```

- Ersetzen zweier Zeilen einer Matrix durch Linearkombinationen:

$$\begin{pmatrix} \longleftarrow A_1 \longrightarrow \\ \vdots \\ \longleftarrow A_j \longrightarrow \\ \vdots \\ \longleftarrow A_k \longrightarrow \\ \vdots \\ \longleftarrow A_n \longrightarrow \end{pmatrix} \hookrightarrow \begin{pmatrix} \longleftarrow A_1 \longrightarrow \\ \vdots \\ \longleftarrow c * A_j - s * A_k \longrightarrow \\ \vdots \\ \longleftarrow s * A_j + c * A_k \longrightarrow \\ \vdots \\ \longleftarrow A_n \longrightarrow \end{pmatrix}$$

```
>> temp = A(j,:);
>> A(j,:) = c*A(j,:) - s*A(k,:);
>> A(k,:) = s*temp + c*A(k,:);
>> A([j k],:) = [c -s;s c]*A([j k],:);
```

8.2 Preallokierung von Speicher

Eine bequeme Eigenschaft von MATLAB besteht darin, dass Arrays (Vektoren) vor ihrer Benutzung nicht deklariert werden müssen, und man sich ins Besondere die Festlegung auf eine Größe (Höchstzahl der Elemente) spart. Werden an ein vorhandenes Array neue Elemente angehängt, wird die Dimension automatisch angepasst. Dafür muss immer neuer Speicher belegt werden – in Extremfällen kann dieses Vorgehen deshalb zu Ineffizienz führen:

```
>> clear;
>> n=1e4;
>> tic, x(1:2) = 1; for i=3:n, x(i)=0.25*x(i-1)^2-x(i-2); end, toc
elapsed time is 5.547000 sec
>> clear
>> n=1e4
>> tic, x=ones(n,1); for i=3:n, x(i)=0.25*x(i-1)^2-x(i-2); end, toc
elapsed time is 0.062000 sec
```

Es ist zu erkennen, dass die arithmetischen Operationen im Vergleich zur Speicher-Allokierung einen verschwindend geringen Anteil an der Rechenzeit haben.

Kapitel 9

Tipps und Tricks

9.1 Leere Arrays

Wir haben die Möglichkeik Arrays der Dimensionen (4,0), (0,7) oder auch (0,0) zu erzeugen, im Folgenden „leere Arrays“ genannt. Die Regeln für Operationen mit leeren Arrays werden aus denen für „normale“ Arrays hergeleitet. So zum Beispiel bei der Matrix-Multiplikation:

```
>> k = 5; A = ones(2,k); B = ones(k,3); A*B
ans=
     5   5   5
     5   5   5
>> k = 0; A = ones(2,k); B = ones(k,3); A*B
ans=
     0   0   0
     0   0   0
```

Und wofür das Ganze? Es unterstützt uns bei unserem Vorhaben, for-Schleifen zu vermeiden – wie das folgende Beispiel zeigt:

```
for i=j-1:-1:1
   s=0;
   for k=i+1:j-1
       s=s+R(i,k)*R(k,j);
   end
end

for i=j-1:-1:1
   s=R(i,i+1:j-1)*R(i+1:j-1,j);
end
```

Im Fall i=j-1 wird R(i,i+1:j-1) zu einer 1×0 - Matrix, R(i+1:j-1,j) zu einer 0×1 - Matrix und wir erhalten als Produkt das die 1×1-Matrix.

9.2 ∞ und $-\infty$

Über den Nutzen von ∞ und $-\infty$ im Zusammenhang mit der Achsenbeschriftung bei plots wird unten noch etwas gesagt. Ein weiteres Gebiet, an dem uns damit geholfen wird, sind die p-Normen. Möchte man nämlich die zu p gehörige q-Norm berechnen, so dass $p^{-1}+q^{-1}=1$ gilt, muss man sich um die Spezialfälle $p=1$ ($\Rightarrow q=\infty$) bzw. p=∞ keine Gedanken machen, sondern kann ruhigen Gewissens `q=1/(1-1/p)` berechnen:

```
>> x=1:10; p=1;
>> norm(x,p)
ans = 55

>> norm(x,1/(1-1/p)
ans = 10;
```

9.3 Matrizen sind auch nur Vektoren

Es ist möglich, ein zweidimensionales Array zu behandeln, als wäre es ein Vektor. Mit A(j) kann man auf das j-te Element des Vektors zugreifen, der entsteht, wenn man die Spalten von A „stapelt“.

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

ans = 2
```

Kapitel 10

Grafiken mit MATLAB erstellen

10.1 Grafiken

In MATLAB ist es relativ leicht verschiedenste Arten von Grafiken zu erzeugen. Das Erscheinungsbild kann sehr individuell gestaltet werden. Beispielsweise kann man Farben, Achsenskalierungen und Beschriftungen den eigenenen Vorstellungen anpassen. Grundsätzlich bestehen zur Modifikation von Grafiken zwei Möglichkeiten, einerseits direkt über die Kommandozeile oder alternativ interaktiv mit der Maus in der vorhandenen Grafik. Wir werden uns auf die erste Variante beschränken, aber es ist durchaus ratsam, auch mal mit den "fertigen" Grafiken rumzuspielen.

10.2 2-dimensionale Plots

Die einfachste Variante zweidimensionale Plots zu erzeugen, besteht darin zwei Vektoren mit gleichen Dimensionen zu koppeln. Hat man zwei solcher Vektoren (x und y) gegeben, werden mit dem Befehl `plot(x,y)` die jeweiligen $x(i)$ und $y(i)$ als zusammengehörig erkannt und ein dementsprechender Plot erzeugt.

Beispiel:

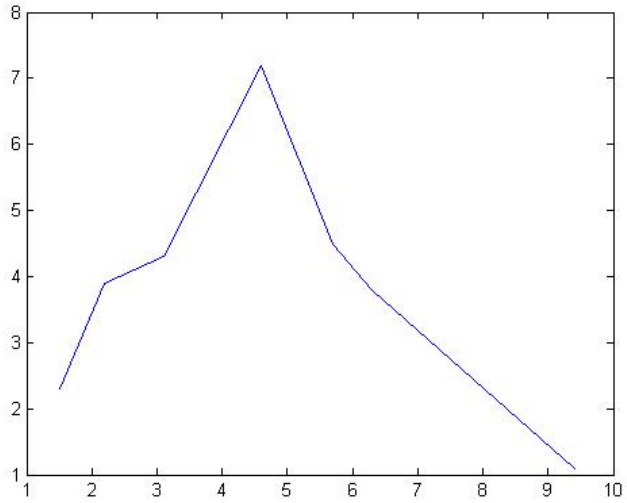
```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> plot(x,y)
```

Farbe	
r	Red
g	Green
b	Blue
c	Cyan
m	Magenta
y	Yellow
k	Black
w	White

Marker	
∘	Kreis
∗	Stern
.	Punkt
+	Plus
x	Kreuz
s	Quadrat
d	Diamant
^	Dreieck nach oben
v	Dreieck nach unten
>	Dreieck nach rechts
<	Dreieck nach links
p	Fünfpunktestern
h	Sechspunktestern

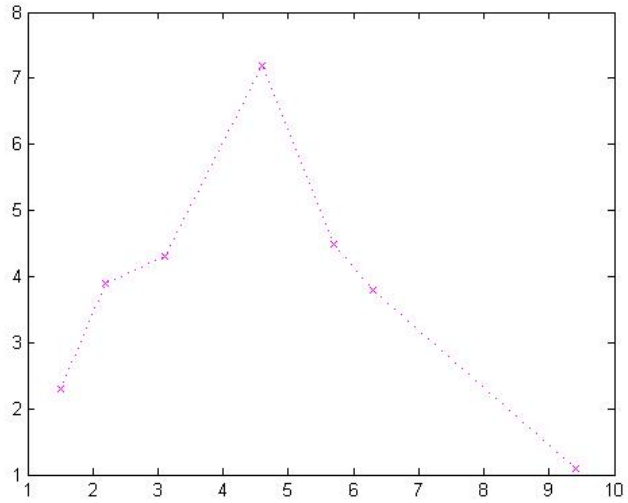
Linienart	
-	durchgezogene Linie
–	gestrichelte Linie
:	gepunktete Linie
-.	gepunktet und gestrichelte Linie

Tabelle 10.1: Parameter für das Aussehen eines Plots



Mit zusätzlichen Angaben kann man das Aussehen modifizieren. Ein allgemeinerer Aufruf der Funktion plot hat die Form `plot(x,y,String)`, wobei sich der String aus bis zu drei Angaben (Farbe, Knoten, Linienart) zusammensetzt. Die möglichen Eingaben können den folgenden Tabellen entnommen werden:

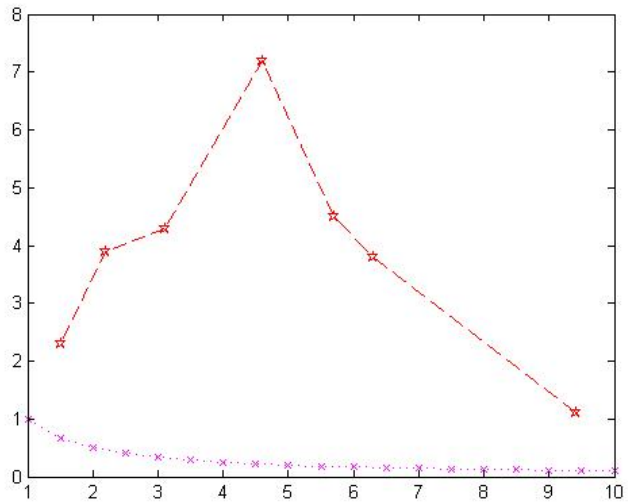
Ein typischer Aufruf hätte die Form $>> plot(x, y,' mx :')$, das gleiche Resultat erhält man auch mit $>> plot(x, y,' m : x')$, woran man sieht, dass die Reihenfolge der Angabe irrelevant ist.



Es ist auch möglich mehr als einen Graphen in das Koordinatensystem zu legen.

Beispiel:

```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> a = 1:.1:10;
>> b = 1./a;
>> plot(x,y,'rp--',a,b,'mx:)
```

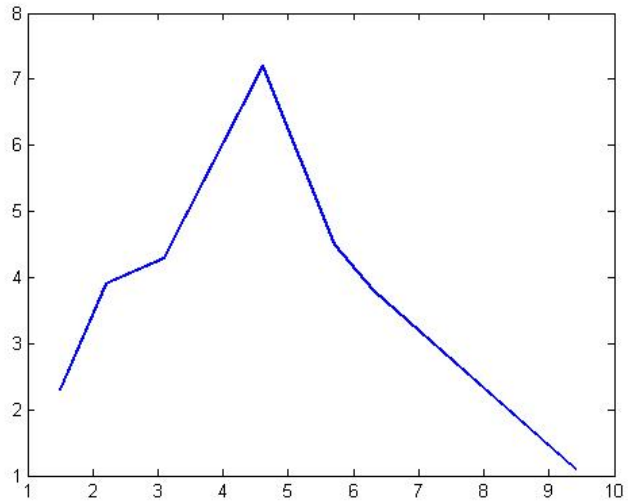


Es werden allerdings nicht nur Vektoren als „Datenquellen“ akzeptiert, sondern auch Matrizen. Wenn x ein m-dimensionaler Vektor und Y eine m x n Ma-

trix ist, bewirkt `plot(x,Y)` das gleiche wie `plot(x,Y(:1),x,Y(:2),...,x,Y(:,n))`. Wenn zusätzlich X auch eine m x n Matrix ist, kann man mit `plot(X,Y)` die jeweils korrespondierenden Spalten der beiden Matrizen zu einem Graph verbinden, es handelt sich also um eine verkürzte Schreibweise von `plot(X(:1),Y(:1),...,X(:,n),Y(:,n))`. Nun stellt sich die Frage, was passiert, wenn die Einträge in den beteiligten Vektoren und Matrizen nicht reellwertig sondern komplex sind. In diesem Fall wird der imaginäre Teil ignoriert. Allerdings gibt es eine Ausnahme - übergibt man plot als einziges Argument eine komlexwertige Matrix Z, erhält man das identische Resultat wie mit dem Kommando `plot(real(Z),imag(Z))`. plot können weitere Attribute übergeben werden, beispielsweise LineWidht:

```
>> x = [1.5 2.2 3.1 4.6 5.7 6.3 9.4];
>> y = [2.3 3.9 4.3 7.2 4.5 3.8 1.1];
>> plot(x,y,'LineWidth',2)
```

Wobei die numerische Angabe jeweils in Punkt geschieht. Dias Resultat sieht dann folgendermaßen aus:



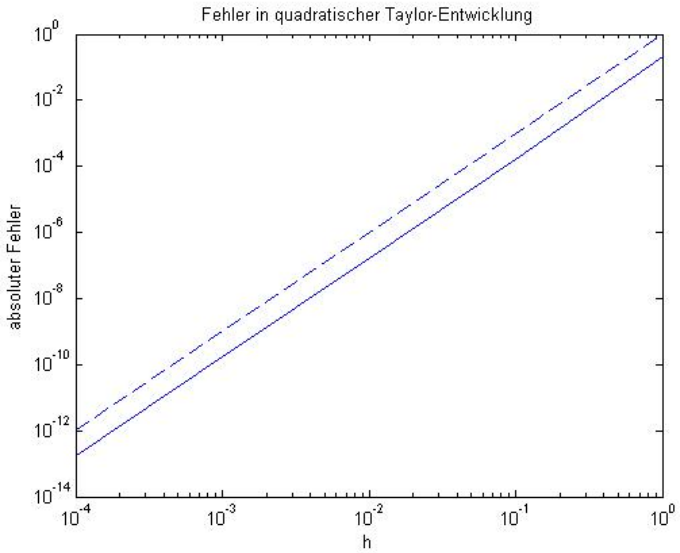
Für manche Anwendungen ist es sinnvoller, die Achsen logarithmisch zu skalieren. Dazu verwendet man statt plot eine der folgenden Anweisungen:

- `semilogx(x,y)`, die x-Achse wird logarithmisch skaliert
- `semilogy(x,y)`, die y-Achse wird logarithmisch skaliert
- `loglog(x,y)`, beide Achsen werden logarithmisch skaliert

Beispiel:
Wir wollen den Fehler bei der quadratischen Taylor-Approximation bei kleinen Zahlen visualisieren und zeigen, dass er sich proportional zu h^3 verhält. Dazu betrachten wir $|1 + h + \frac{h^2}{2} - exp(h)|$ und zum Vergleich h^3 jeweils ausgewertet für $h = 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1$ und mit logarithmischer Skalierung:

```
>> h = 10.^[0:-1:-4];
>> taylor = abs((1+h+h.^2/2)-exp(h));
>> loglog(h,taylor)
>> hold on
>> xlabel('h')
>> ylabel('absoluter Fehler')
>> title('Fehler in quadratischer Taylor-Entwicklung')
>> plot(h,h.^3,'--')
```

Das Ergebnis bestätigt unsere Theorie:



Neben loglog haben wir noch einige weitere neue Befehle benutzt, die allerdings mit Ausnahme von `hold on` selbserklärend sein sollten. Was macht nun aber dieses ominöse `hold on`? Nun, es sorgt dafür, dass beim Plotten von weiteren Funktionen die bisherigen erhalten bleiben und nicht - wie default eingestellt bzw. bei `hold off` - zunächst das figure-Fenster „geleert“ und erst dann der neue Plot eingefügt wird. Das ist in unserem Beispiel durchaus sinnvoll, da wir so überprüfen können, ob die beiden Geraden tatsächlich annähernd parallel sind. Trotzdem ist es komisch, dass MATLAB grundsätzlich in das figure-Fenster mit der Nummer 1 zeichnet. Das muss doch auch anders gehen?! Tut es auch! Mit dem Befehl figure wird ein neues figure-Fenster geöffnet, mit figure(n) kann man ein beliebiges figure-Fenster aktivieren, so dass sich alle folgenden (Grafik-) Befehle auf dieses beziehen.

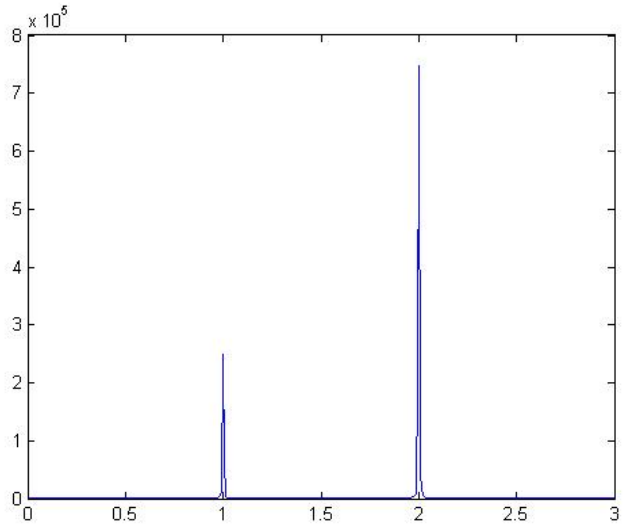
Viele Eigenschaften der Achsen können mit dem `axis`-Kommando beeinlusst werden. Mögiche Einstellungen wären:

- `axis [xmin xmax ymin ymax]`
- `axis auto`
- `axis equal`

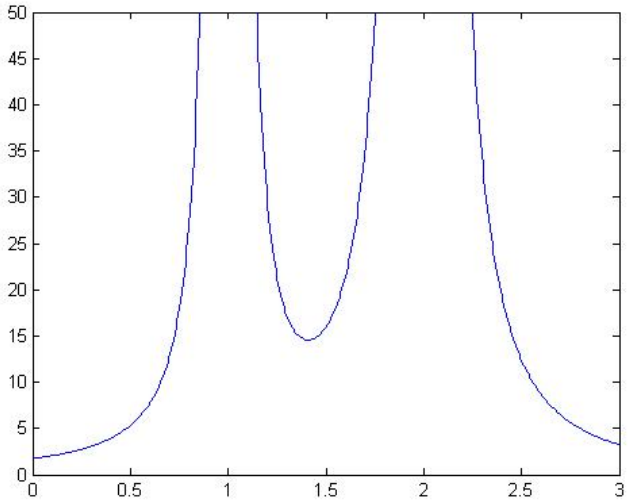
- `axis off`
- `axis square`
- `axis tight`

Für den Fall, dass man nur die Grenzen einer Achse festlegen möchte, stehen die Befehle `xlim([xmin,xmax]` und `ylim([ymin,ymax]` zur Verfügung. Sollte man sich an einem Ende des Intervalls nicht festlegen wollen oder können, sorgt die Angabe von `-inf` bzw. `inf` dafür, dass sich MATLAB darum kümmert. Warum es an manchen Stellen sinnvoll ist, die Achseneinstellungen selbst vorzunehmen, zeigt folgendes Beispiel - wir betrachten beide Male die Funktion $\frac{1}{(x-1)^2} + \frac{3}{(x-2)^2}$, zunächst übernimmt MATLAB selbst die Begrenzung der Achsen, dann greifen wir manuell ein.

```
>> x = linspace(0,3,500);
>> plot(x,1./(x-1).^2+3./(x-2).^2)
>> grid on
```




```
>> ylim([0 50])
```



Weiter oben haben wir ja schon gesehen, wie man seinem Plot einen Titel verpasst, doch man kann auch Text an einer beliebigen Stelle einfügen und sogar einer automatisch erstellten Legende kann man sich bedienen. Um Text einzufügen, reicht der Befehl `text(x,y,'string)`, wobei (x,y) die Koordinate festlegt, an der der Text beginnen soll. Zum Einfügen der Legende lautet das Kommando `legend('string1', ... ,'stringn', Position)`, die Reihenfolge und die Anzahl der übergebenen Strings sollte mit der im plot- bzw loglog-Befehl übereinstimmen. Mögiche Werte für Position:

- -1: rechts des Plots
- 0: MATLAB entscheidet
- 1: rechts oben
- 2: links oben
- 3: links unten
- 4: rechts unten

Es ist möglich (und teilweise auch notwendig) in Strings TEX-Code zu benutzen. Um zum Beispiel folgenden Text am Punkt (-0.6,0.7) in einer Grafik einzufügen: $(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$ schreibt man idealerweise:

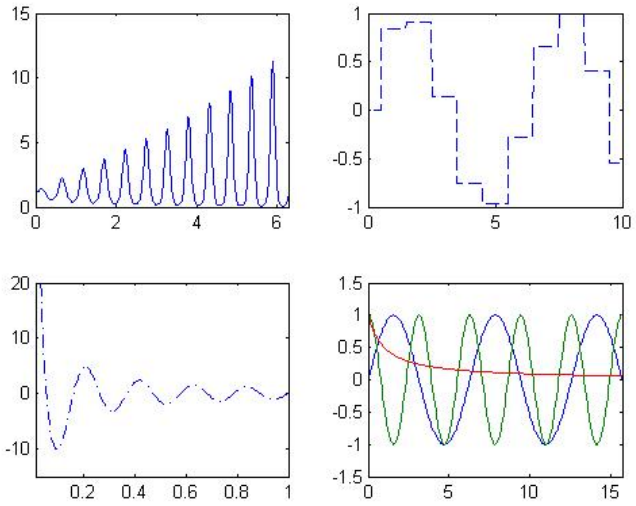
```
>> text(-.6,.7,'(n+1)P_{n+1}(x)=(2n+1)xP_n(x)-nP_{n-1}(x)')
```

10.3 Mehrere Plots in einem Fenster

Der Befehl `subplot(m,n,p)` teilt das figure-Fenster in m Zeilen und n Spalten auf. p ist ein Skalar und bezeichnet das aktive Feld.

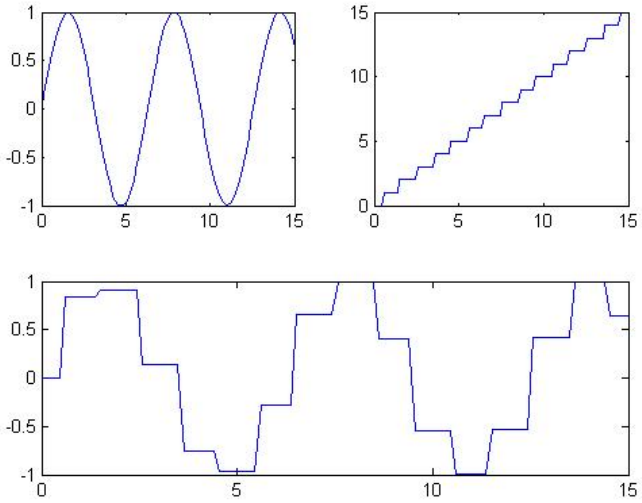
Beispiel

```
>> subplot(2,2,1), fplot('exp(sqrt(x)*(sin(12*x))',[0 2*pi])
>> subplot(2,2,2), fplot('sin(round(x))',[0 10], '--')
>> subplot(2,2,3), fplot('cos(30*x)/x',[0.01 1 -15 20], '-.')
>> subplot(2,2,4), fplot('[sin(x), cos(2*x),1/(1+x)',[0 5*pi -1.5 1.5])
```



Hier haben wir eine neue Funktion genutzt: `fplot`. MATLAB wertet dabei die Funktion an „genug“ Stellen aus, um einen realitätsnahen Graph zu erhalten. Natürlich können auch fplot weitere Parameter übergeben werden. Mehr dazu findet man - wie zu jeder anderen Funktion auch - unter `>> help fplot`. Auch eine unregelmäßige Aufteilung des figure-Feldes ist möglich:

```
>> x = linspace(0,15,100);
>> subplot(2,2,1), plot(x,sin(x))
>> subplot(2,2,2), plot(x,round(x))
>> subplot(2,2,3:4), plot(x, sin(round(x)))
```

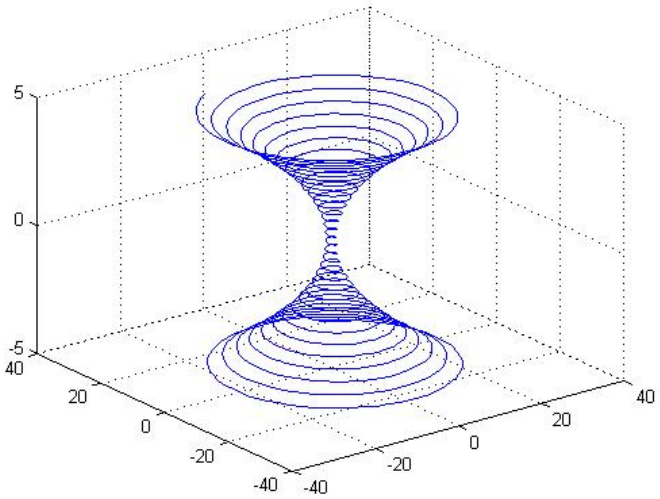


10.4 3-dimensionale Plots

Analog zum Befehl plot im zweidimensionalen exisitert auch der Befehl plot3. Im Grunde funktioniert er genauso.

Beispiel

```
>> t = -5:.005:5;
>> x = (1+t.^2).*sin(20*t);
>> y = (1+t.^2).*cos(20*t);
>> z = t;
>> plot3(x,y,z)
>> grid on
```



plot3 ist zwar ganz nett, kann aber nur zur Visualisierung von Funktionen $f : R \to R^2$ genutzt werden, nicht aber Funktionen $g : R^2 \to R$, anschaulich gesprochen werden ausschließlich Linien und nie Flächen dargestellt.
Einen ersten Ausweg aus diesem Dilemma bietet der Befehl ezcontour, der für eine Funktion $f : [xmin, xmax] \times [ymin, ymax] \to R$ eine Höhenkarte erzeugt, indem man `>>ezcontour('f',[xmin xmax ymin ymax]);` ausführt.
Ähnliches liefert auch die Funktion contour, ihr müssen allerdings zwei Vektoren x und y, sowie eine Matrix $Z = (z_{i,j})$ mit $z_{i,j} = f(x_i, y_i)$ übergeben werden:

```
>> subplot(211)
>> ezcontour('sin(3*y-x^2+1)+cos(2*y^2-2*x)',[-2 2 -1 1]);
>> x = -2:.01:2; y = -1:.01:1;
>> [X,Y] = meshgrid(x,y);
>> Z = sin(3*Y-X.^2+1)+cos(2*Y.^2-2*X);
>> subplot(212)
>> contour(x,y,Z,20)
```

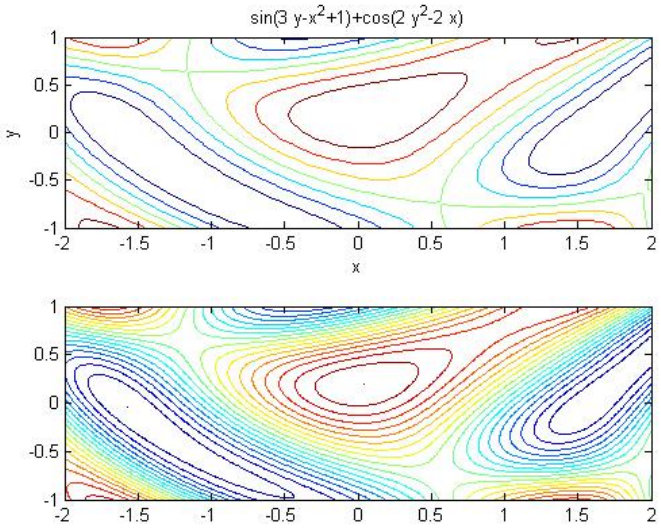
Um Z zu berechnen haben wir uns hier komponentenweisen Operationen auf Matrizen bedient. Es ist leicht einzusehen, dass x und y dazu in zwei Matrizen X und Y konvertiert werden müssen, deren Zeilen bzw. Spalten Kopien der Ursprungsvektoren sind. Genau das erreichen wir durch den Befehl `>> [X,Y] = meshgrid(x,y);` in der vierten Zeile. Beispielsweise liefert `[X,Y]=meshgrid(1:4,5:7)` die beiden Matrizen

```
X =

     1     2     3     4
     1     2     3     4
     1     2     3     4


Y =
```

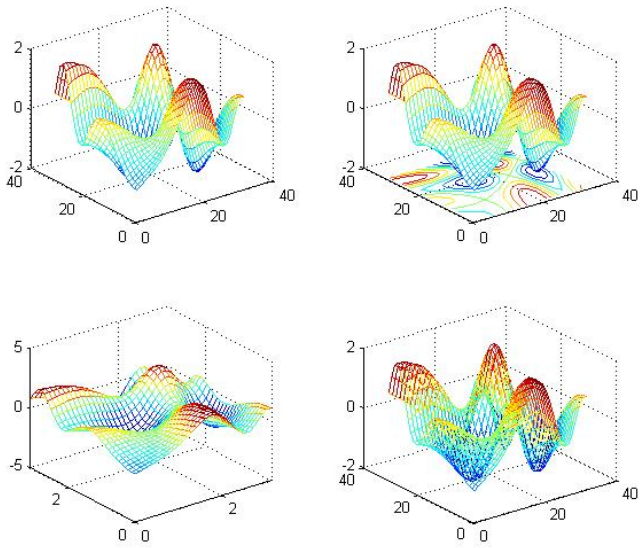
```
5     5     5     5
6     6     6     6
7     7     7     7
```



Natürlich können contour auch mehr als drei Argumente übergeben werden, so dass zum Beispiel die Anzahl der unterschieldichen Höhen beeinflusst werden kann. Für Einzelheiten in der Hilfe nachschauen, bei der Gelegeneheit auch mal `clabel` nachschlagen.

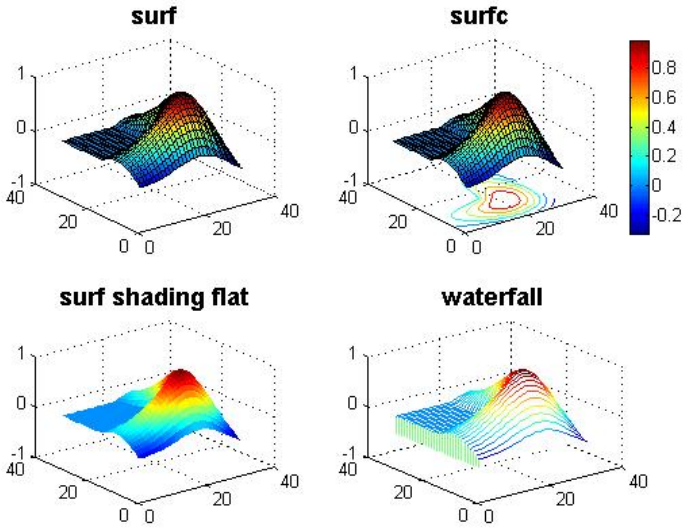
Die Befehle `mesh` und `meshc` erwarten als Eingabe die gleichen Datentypen wie `contour`, sie liefern uns endlich dreidimensionale Plots, wie wir sie uns vorstellen, `meshc` legt in die x-y-Ebene zusätzlich eine Höhenkarte.

```
>> x = 0:.1:pi; y = 0:.1:pi;
>> [X,Y] = meshgrid(x,y);
>> Z = sin(Y.^2+X)-cos(Y-X.^2);
>> subplot(221)
>> mesh(Z)
>> subplot(222)
>> meshc(Z)
>> subplot(223)
>> mesh(x,y,Z)
>> axis([0 pi 0 pi -5 5])
>> subplot(224)
>> mesh(Z)
hidden off
```



Es fällt auf, dass nur netzartige Strukturen entstanden sind. Ausgefüllt bekommt man die Graphen durch die Befehle `surf` bzw. `surfc`. Ein ähnliches Resultat liefert auch `waterfall`.

```
>> Z = membrane; FS = 'FontSize';
>> subplot(221), surf(Z), title('\bf{surf}',FS,14)
>> subplot(222), surfc(Z), title('\bf{surfc}',FS,14), colorbar
>> subplot(223), surf(Z), shading flat
>> title('\bf{surf} shading flat',FS,14)
>> subplot(224), waterfall(Z), title('\bf{waterfall}',FS,14)
```



10.4.1 Animationen

Es gibt zwei Möglichkeiten, animierte Plots zu erzeugen.

- Durch Erzeugen einer Sequenz von figures, die gespeichert und in Form eines Films abgespielt wird:

```
>> Z = peaks; surf(Z)
>> axis tight
>> set(gca,'nextplot','replacechildren')
>> for j = 1:11
>> surf(cos(2*pi*(j-1)/10).*Z,Z)
>> F(j) = getframe;
>> end
>> movie(F)
```

- Durch Manipulation der Objektattributte XData, YData und ZData, beispielsweise durch Aufruf der Funktion `comet` oder `comet3`:

```
>> x = linspace(-2,2,500);
>> y = exp(x).*sin(1./x);
>> comet(x,y)
>> t = -pi:pi/500:pi;
>> comet3(sin(5*t),cos(3*t),t)
```

10.5 Diagramme

10.5.1 Balkendiagramme

Zum Erzeugen von Balkendiagrammen stehen uns im zweidimensionalen die Funktionen `bar` und `barh`, im dreidimensionalen die Funktionen `bar3` und `barh3` zur Verfügung, das 'h' steht dabei für horizontal, wir erhalten also waagerechte Balken. Balkendiagramme aggieren auf Matrizen. Deshalb erzeugen wir uns zunächst eine Beispiel-Matrix, die uns durch dieses Kapitel begleitet:

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

MATLAB gruppiert jeweils die Elemente einer Zeile

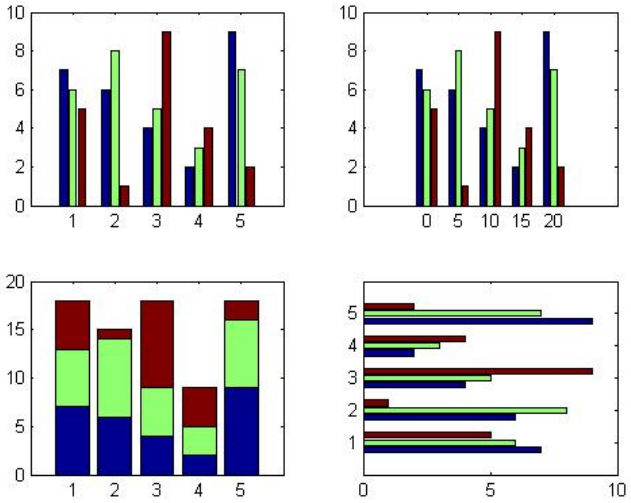
```
>> subplot(2,2,1), bar(Y)
```

Durch zusätzliche Übergabe eines Vektors x kann man die Bescriftung der x-Achse ändern.

```
>> x = 0:5:20;
>> subplot(2,2,2), bar(x,Y)
```

Bisher wurde das Attribut `grouped` verschwiegen, eigentlich müsste der Aufruf `bar(x,Y,'grouped')` heißen, allerdings wird `grouped` als Standard angenommen. Die Alternative dazu lautet `stacked`.

```
>> subplot(2,2,3), bar(Y,'stacked')
>> subplot(2,2,4), barh(Y)
```

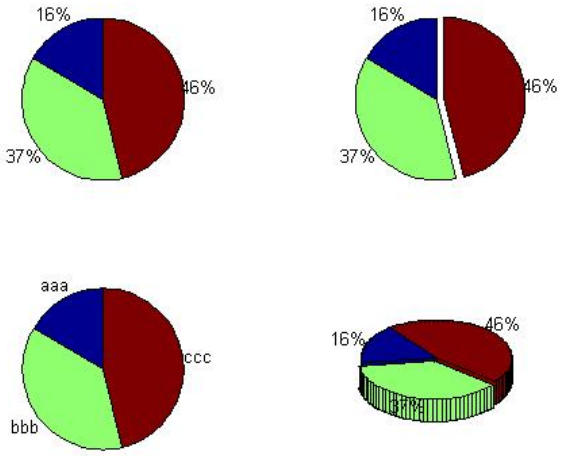


Im Dreidimesionalen funktionieren diese Befehle natürlich auch.

10.5.2 Tortendiagramme

Tortendiagramme werden mit `pie` bzw. `pie3` erzeugt und erwarten als Parameter einen Vektor. Es besteht die Möglichkeit einige Tortenteile „herauszutrennen“ bzw. die Torte alternativ zu beschriften:

```
>> x = [1.5 3.4 4.2];
>> subplot(2,2,1), pie(x)
>> subplot(2,2,2), pie(x,[0,0,1])
>> subplot(2,2,3), pie(x,{'aaa','bbb','ccc'})
>> subplot(2,2,1), pie3(x,[0,1,0])
```



Kapitel 11

Troubleshooting

11.1 Errors und Warnings

Es ist bestimmt schon jedem ein Tipp- oder Programmierfehler unterlaufen, der von Matlab durch einen Error quittiert wird:

```
>> mod(3,sqrt(-2))
??? Error using ==> mod
Arguments must be real.
```

Zusätzlich zum Wort 'Error' besteht die Fehlermeldung aus einem erläuternden Text - wie aussagekräftig auch immer der sein mag...
Auf jeden Fall ist es auch möglich sich selbst solche Fehlermeldungen ausgeben zu lassen, das folgende Beispiel sollte selbserklärend sein:

```
if ~isreal(arg2), error('Arguments must be real.'), end
```

Die Funktion `warning` zeigt, wie `error`, sein Argument an, stoppt die Ausführung des Programms aber nicht. Warum dann nicht einfach `disp` benutzen? Das hat vor allem zwei Gründe - zum einen kann man mit `warning('off')`, die Warnmeldungen einfach unterdrücken, zum anderen ist es auch möglich alle aufgetretenen Fehler in einer Variable zu speichern: `warns=warning`.

11.2 Debugging

Die einfachste Möglichkeit, sich wichtige Zwischenergebnisse bei der Berechnung anzeigen zu lassen, besteht schlicht darin, die gewünschte Variable ohne Semikolon in den Code zu schreiben.
Ein mächtiges Werkzeug von Matlab ist aber auch der so genannte Keyboard-Modus. Er kann an einer bestimmten Stelle durch die Eingabe von `keyboard` oder alternativ mit zum Beispiel `dbstop at 5 in fak`. Auf diese Art und Weise lässt sich auch die nützliche Version `dbstop if error` implementieren.
Im Keyboaard-Modus steht eine Kommandozeile zur Verfüegung, so dass man sowohl Werte von Variablen abfragen als auch verändern kann.

Kapitel 12

Weiterführende MATLAB-Funktionen

Im abschließenden Kapitel sollen noch kurz einige, der vielenweiteren Möglichkeiten mit MATLAB zu programmieren vorgestellt werden. Dies erfolgt beispielhaft an einigen interessanten Toolkits, der Möglichkeit GUIs zu erstellen und MATLAB mit anderen Programmiersprachen zu kombinieren.

12.1 Toolkits

Die Funktionalität von MATLAB wurde von einigen Toolboxen erweitert.Um herauszufinden, ob eine Toolbox bereits installiert ist, schaut man die Liste durch, die MATLAB nach dem Kommando `ver` ausgibt. Wenn der Name der gewünschten Toolbox enthalten ist, so ist diese auch vorhanden.

Es gibt eine Vielzahl solcher Toolboxen z.B. zur Optimierung, für Neuronale Netzwerke, Splines,Partielle Differentialgleichungen oder für Wavelets. Eine aktuelle Liste solcher Toolboxen findet sich unter

`http://www.mathworks.com/products/product_listing/index.html`

wo es auch entsprechende Dokumentationen gibt. Außerdem gibt es noch eine Vielzahl von Toolboxen, die von Privatpersonen oder Organisationen zur Verfügung gestellt werden. Diese haben meist sehr spezielle Zielsetzungen (z.B. INTLAB, Hidden Markow Modell Toolbox, Bioelectromagnetism Toolbox). Wir werden uns nun mit einer kleinen Auswahl beschäftigen.

12.1.1 Symbolic Math Toolbox

Um auch symbolische Rechnungen (wie im Maple) in MATLAB zu machen, benötig man die Symbolic Math Toolbox. Diese basiert auf MAPLE und liefert einen Teil der dort vorhandenen Funktionen in MATLAB-Syntax mit. Um symbolische Variablen einzuführen, verwendet man den Befehl `sym` oder `syms`:

```
>> b = sym(sqrt(2))
>> syms x y n a
>> f = sin(x*y)/(x*y)
```

Um diese Variablen wieder in numerische umzuwandeln, verwendet man `vpa` und um die Genauigkeit festzulegen `digits`:

```
>> digits(25)
>> vpa(b)
```

Außerdem kann man den symbolischen Variablen auch Werte zuweisen:

```
>> solve(a*x^2+b*x+c)
>> subs(y,{a,b,c},{1,-1,-1})
```

Differential- und Integralrechnung

Nachdem man eine Funktion `f` mit symbolischen Variablen erzeugt hat, kann man diese nun auch ableiten mit `diff`oder integrieren mit `int`:

```
>> diff(f)
>> int(f)
```

Außerdem lassen sich auch Grenzwerte symbolisch berechnen mit `limit`:

```
>> limit((1 + x/n)^n,n,inf)}
```

Vereinfachung

Um die Ausgabe leichter lesbar zu machen gibt es das `pretty`- Kommando:

```
>> pretty(f)
```

Um Ausdrücke zu vereinfachen, kann man entweder `simplify` oder `simple` verwenden, wobei `simple` den Ausdruck sucht, der am kürzesten ist:

```
>> simplify((1/a^3+6/a^2+12/a+8)^(1/3))
>> simple((1/a^3+6/a^2+12/a+8)^(1/3))
```

Gleichungssysteme lösen

Desweiteren kann man auch Gleichungen mit `solve` und Differentialgleichungen mit `dsolve` lösen:

```
>> solve('cos(n*x)+sin(x)=1')
>> dsolve('Dy=1+y^2')
```

Graphische Ausgabe

Es gibt außerdem Kommandos zum plotten, z.B. `ezplot`:

```
>> ezplot(x^2-y^4)
```

12.1.2 Image Processing Toolbox

Die Image Processing Toolbox bietet die Möglichkeit auf Bilder zuzugreifen, sie zu verändern und zu speichern.

Lesen und Schreiben von Bildern

Dazu lädt man zuerst ein Bild mit dem Befehl `imread`. Dieser unterstützt unter anderm die Formate JPEG, TIFF, GIF, BMP oder PNG. Mit `imwrite` kann man Bilder speichern. Der Befehl `imfinfo` gibt Informationen:

```
>> A = imread('test.jpg');
>> imwrite(A,'test2.jpg');
>> imfinfo('test.jpg')
```

Anzeigen und Bearbeiten von Bildern

Das Bild kann man sich mit dem Befehl `imshow` anzeigen lassen:

```
>> imshow(A)
```

Den Kontrast eines Graustufenbildes kann man mit dem Befehl `imadjust` verbessern. Mit dem Befehl `imfilter` kann man beliebige Filter auf ein Bild anwenden. Eine Auswahl solcher Filter stellt `fspecial` zur Verfügung.

```
>> imadjust(A)
>> h=fspecial('sobel')
>> imfilter(A,h)
```

12.2 Der MATLAB C-Compiler

Mit dem MATLAB Compiler kann man auch Standalone Anwendungen erstellen oder C-, C++ Shared Libraries. Dazu muss auf dem System erstmal die MATLAB Component Runtime installiert sein. (im MATLAB-Verzeichnis unter root\toolbox\compiler\deploy\win32). Anschließend muss man den Compiler konfigurieren. Dazu ruft man in MATLAB `mbuild -setup` auf und wählt dort das Verzeichnis, in dem der C-Compiler liegt. Danach kann man mit `mcc` die eigenen m-Dateien kompilieren. Wenn man eine Standalone Anwendung erstellen möchte verwendet man:

```
>> mcc -m eigenefun.m
```

Für ein C Shared Library:

```
>> mcc -l eigenefun.m
```

Für ein C++ Shared Library:

```
>> mcc -l eigenefun.m -W cpplib -T link:lib
```

Um die so erzeugte Anwendung oder Bibliothek weiterzugeben, muss man natürlich die compilierte Datei weitergeben, außerdem die `eigenefun.ctf` und das MATLAB Component Runtime (unter Windows MCRInstaller.exe, unter Linux MCRInstaller.zip). Nun kann man auf jedem System über die Kommandozeile das Programm starten, dabei werden die evtl. vorhandenen Parameter einfach nach einem Leerzeichen angehängt.

12.3 Andere Programmiersprachen mit MATLAB verwenden

MATLAB ist sehr nützlich für mathematische Programmierung, bietet aber auch die Möglichkeit andere Programmiersprachen zu nutzen. So kann man z.B. bereits bestehende Programme in C oder Fortran einbinden oder Java-Klassen nutzen.

12.3.1 Fortran und C

Viele mathematische Programme sind in FORTRAN oder C geschrieben sind und man kann sie auch in MATAB nutzen, indem man sie etwas ändert und compiliert. Daher testen wir zuerst, ob unser Systm dafür passend konfigueriert ist. Dazu kompieren wir `yprime.c` ins aus \extern\examples\mex Arbeitsverzeichnis. Nun rufen wir `mex` auf:
`>> mex yprime.c`
Kann man nun yprime wie eine normale M-Funktion verwenden, funktioniert alles. `>> yprime(1,1:4)`

Ansonsten sollte man mit `mex -setup` sicherstellen, dass ein geeigneter Compiler ausgewählt ist. Ebenso verfährt man mit Fortran-Programmen. Beim Aufruf des Compilers `mex` kann man auch noch eine Optionsdatei
(z.B. aus \bin\win32\mexopts) mitgeben:
`>> mex yprime.c -f lccopts.bat`

Solche Optionsdateien kann man auch selbst erstellen, wenn man zum Beispiel einen nicht supporteten Compiler verwenden will, oder mehr Einfluss auf den Compilationsprozess nehmen will.

Um ein C bzw. Fortran-Programm MATLAB kompatibel anzupassen, benötigt man eine Gatewayroutine. Außerdem muss im C-Header auch `# include „mex.h"` stehen. Die Gatewayroutine stellt die Schnittstelle zwischen MATLAB und C dar:

```
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const
mxArray*prhs[] )}
```

`nlhs` gibt die Anzahl der Ausgabeparameter,`nrhs` sie Anzahl der Eingabeparameter an. Die Parameter `plhs` und `prhs` enthalten die Pointer auf die entsprechenden Eingabe-/Ausgabevektoren. Danach kommt C Code, der die Eingaben prüft und die Datenstrukturen (natürlich MATLAB kompatibel, z.B. `mxArray`) für die Ausgabe anlegt. Die eigentliche C Routine ist meist eine eigene Funktion, die von hier aus aufgerufen wird und deren Ergebnis dann durch die Gatewayroutine an MATLAB zurückgegeben wird.

Ruft man nun die Funktion in Matlab auf, findet MATLAB die MEX-Datei vor M-Dateien mit gleichem Namen. Dies kann man sich für die MATLAB Hilfe zunutze machen. Da die MEX-Datei keine für MATLAB lesbare Hilfe hat, sucht MATLAB weiter und gibt die Hilfe einer gleichnamigen M-Datei aus, falls vorhanden.

Auch Fortran benötigt eine Gatewayroutine, die der in C sehr ähnlich sieht:
`subroutine mexFunction(nlhs, plhs, nrhs, prhs)`

`integer plhs(*), prhs(*)`
`integer nlhs, nrhs`,
die anschließend die Eingaben prüft, den Ausgabevektor initialisiert und die eigentliche Funktion aufruft.

Außerdem kann man auch Matlab Routinen in C oder Fortran verwenden. Dazu läuft eine seperate MATLAB Engine, die das C bzw. Fortran Programm ansteuern kann. Dazu muss `engine.h` eingebunden sein. Mit `engOpen("\0")` startet man die MATLAB-Engine. Man erhält dabei einen Pointer auf die Engine. Nun kann man Matlab kompatible Variablen deklarieren und mit `engPutVariable` an Matlab geben. Die Auswertung eines Ausdrucks erfolt mit `engEvalString`. Wenn man alle Auswertungen ausgeführt hat, gibt man den Speicher wieder frei und schließt die MATLAB Engine mit `engClose`, z.B.:

```
Engine *ep;

ep = engOpen("'\0"'); T = mxCreateDoubleMatrix(1, 10, mxREAL);

engPutVariable(ep, "'T"', T);
engEvalString(ep,"'D=.5.*(-9.8).*T.^2;"');

mxDestroyArray(result);
engClose(ep);
```

Matlab in Fortran einzubinden funktioniert analog.

12.3.2 Java

MATLAB liefert außerdem bei seiner Installation eine JVM (Java Virtual Machine) mit und bietet die Möglichkeit auf Java-Objekte zuzugreifen. Dabei kann man den Pfad entweder statisch in der classpath.txt (unter \toolbox\local) oder dynamisch mit `javaaddpath C:\Java` einbinden. Wenn man jar-Archives einbinden will, muss man ihren vollständigen Pfad inklusiv ihrem Namen angeben, bei normalen Javaklassen reicht der Pfad zu einem Elternverzeichnis um alle Klassen darin in MATLAB verfügbar zu machen. Nun kann man wie man es aus Java gewohnt ist, mit dem Konstruktor Instanzen einer Java-Klasse erzeugen und bearbeiten: `>> frame = java.awt.Frame('Frame A');`
`>> frame.setSize(100,60);`
`>> frame.setVisible(true);`

Mittels Java-Klassen kann man zum Beispiel Daten aus dem Internet holen:

```
>> url = java.net.URL('http://www.fs.tum.de/FSMPI/Kurse/Matlab/test.html')
>> is = openStream(url);
>> isr = java.io.InputStreamReader(is);
>> br = java.io.BufferedReader(isr);
>> for k = 1:85
s = readLine(br);
end
>> s = readLine(br);
```

```
>> disp(s)
```

12.4 GUIs programmieren

Mit dem MATLAB Compiler haben wir eine Möglichkeit kennengelernt, Standalone Anwendungen zu erstellen, allerdings ist die Parameter-Übergabe wenig intuitiv. Daher ist es sinnvoll für solche Anwendungen eine GUI zu erstellen. MATLAB bietet dazu direkt Funktionen an oder ein graphisches Tool, mit dem man die Steuerelemente anordnen kann.

12.4.1 GUI Kommandos

Um Steuerelemente direkt zu programmieren, kann man das Kommando `uicontrol` verwenden. Dem Kommando werden verschiedene Werte mitgegeben, z.B. gibt 'Style' an, was für ein Element man möchte (z.B. checkbox, edit, listbox, popupmenu pushbutton, radiobutton, slider, text). Man kann zum Beipiel einer `Figure` einen Button zum Schließen des Fensters mitgeben:

```
>> fh = figure('NumberTitle','off','Name','GUI');
>> ah = axes('Position',[0.1 0.1 0.65 0.85]);
>> membrane;
>> ph = uicontrol(fh, 'Style', 'pushbutton', 'String', 'Ende', 'Units',
'normalized', 'Position', [0.8 0.08 0.18 0.12], 'ToolTipString','Schließt
das Fenster','Callback', 'close(gcbf)');
```

12.4.2 GUIDE

Guide ist ein graphisches Interface um GUIs zu erzeugen. Um es zu starten tippt man den Befehl `guide`. Nun kann man auf einem leeren GUI die Komponenten per Drag'n'Drop anordnen. Und mit dem Property Manager Eigenschaften zuordnen (z.B. Farbe, Text, Closefunction). Wenn man die fertige GUI gespeichert hat, findet man eine .fig und .m Datei desselben Namens. Bis jetzt haben wir nur eine Gui, die aber noch keine Funktionalität besitzt, wenn wir sie starten. Wenn wir die .m-Datei editieren, weisen wir nun die entsprechenden Funktionen zu. In der generierten M-Datei findet sich am Anfang eine Initialisierung, die man nicht änern sollte. Anschließend kommt die eigene Opening Function. Hier schreiben wir alle Funktionen hinein, die nur einmal und zwar beim Öffnen der Gui erledigt werden sollen (z.B. Beispielplot). Mit `setappdata` stellen wir dann die gebrechneten Daten der GUI zur Verfügung. Die Output Funktion verwenden wir, wenn wir Werte z.B. an eine weitere GUI weitergeben möchten (so muss man sie nicht öffentlich deklarieren). In der Callback Funktion implementieren wir die Funktionalität der einzelnen Elemente, z.B. Auslesen eines Wertes von einem Slider (hObject ist das Handle des Sliders):

```
>> f = get(hObject,'Value');
```