MATLAB im Selbststudium

Eine Einführung

Christof Büskens

Zentrum für Technomathematik
Fachbereich Mathematik
Universität Bremen
28359 Bremen, Germany

Vorlesungsbegleitende Ausarbeitung
Sommersemester 2004

(Unkorrigierte Fassung)

Vorwort

Die vorliegende Ausarbeitung entstand während meiner Tätigkeit als Privatdozent am Lehrstuhl für Ingenieurmathematik der Universität Bayreuth. Sie entstand im Rahmen einer vorlesungsbegleitenden Lehrveranstaltung, die ich im Sommersemster 2003 gehalten habe. Als Vorlage dienten die Aufzeichnungen von Prof. Dr. Lars Grüne, dem ich auf diesem Wege herzlich danken möchte.

Bayreuth, April 2004 Christof Büskens

Inhaltsverzeichnis

Kapitel 1

Einführung in Matlab

1.1 Matrizen und Vektoren

Matrizen kann man einfach zeilenweise 'per Hand' eingeben:

```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
```

Oder, wenn man eine Matrix 'leer' definieren möchte, z.B. um darin ein Rechenergebnis abzulegen

```
B = zeros(4,3)
```

Spaltenvektoren gibt man einfach als einspaltige Matrizen ein

```
b = [3; 4; 5]
```

und Zeilenvektoren als einzeilige Matrizen

```
c = [3 4 5]
```

MATLAB kann Matrizen transponieren

```
A'
```

und Matrizen mit Matrizen oder Vektoren multiplizieren

```
C = [6 5; 4 3; 2 1]
A*C
A*b
```

Man kann auf die Elemente einer Matrix oder eines Vektors einzeln zugreifen

```
A(1,1)
A(2,3)
b(3)
```

und ebenso auf einzelne Zeilen

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

oder Spalten

```
A(:,3)
```

Diese kann man nicht nur auslesen, sondern auch zuweisen

```
A(:,2) = [1; 2; 3; 4]
```

Man kann (Zeilen)-Vektoren mit abgezählten Einträgen auch mit einer Abkürzung definieren. Hierbei gibt man den ersten Eintrag, die Schrittweite und den letzten Eintrag, jeweils durch Doppelpunkt getrennt an. Z.B.: Erzeuge den Vektor [1; 2; 3; 4; 5]

```
t =  [1:1:5]
```

Schließlich kann man nicht nur Matrizenrechnung direkt ausführen, sondern auf Vektoren und Matrizen auch komponentenweise rechnen Hierzu setzt man einen '.' vor den entsprechenden mathematischen Operator. Um z.B. jedes Element des obigen Vektors t mit sich selbst zu multiplizieren, berechnet man

```
t.*t
```

Beachte, dass t*t aus Dimensionsgründen eine Fehlermeldung ergibt.

Wir illustrieren einige Matrix-Operationen, die aus der Vorlesung bzw. aus den grundlegenden Mathematikvorlesungen bekannt sind: Seien

```
A = [ 1 5 6; 7 9 6; 2 3 4]
b = [29; 43; 20]
```

Berechnung der Determinante:

```
det(A)
```

Berechnung der Inversen

```
inv(A)
```

Berechnung von Vektornormen $\|.\|_p$ für p=1,2,unendlich (inf)

```
norm(b,1)
norm(b,2)
norm(b,inf)
```

Berechnung der zugehörigen induzierten Matrixnormen

```
norm(A,1)
norm(A,2)
norm(A,inf)
```

Berechnung der Kondition einer Matrix für p=1,2,unendlich (inf)

```
cond(A,1)
cond(A,2)
cond(A,inf)
```

1.2 'FOR', 'IF', 'WHILE' und 'BREAK' Anweisungen

MATLAB's Programmiersprache bietet eine Reihe von Möglichkeiten, den Programmablauf in einem M-File zu steuern. Wir werden hier einige kennen lernen:

Die 'for' Schleife ermöglicht es, eine oder mehrere Operationen nacheinander für verschiedene Werte einer Variablen auszuführen Der mehrfach auszuführende Block von Operationen wird mit 'end' beendet.

Beispiel: Ausgabe der ersten 10 Quadratzahlen:

```
for i = 1:10
  i^2
end
input('Druecke RETURN');
```

Schleifen können auch rückwärts zählen, dabei muss zwischen Anfangs- und Endwert die Schrittweite (hier '-1') angegeben werden

```
for i= 10:-1:1
  sqrt(i)
end
input('Druecke RETURN');
```

Desweiteren kann man Schleifen verschachteln

```
for i = 1:10
  i
  for j = 10:-1:i
    j
  end
end
input('Druecke RETURN');
```

Mit solchen Schleifen kann man z.B. eine Matrizenmultiplikation programmieren

```
A = [1 2 3; 4 5 6; 7 8 9]
B = [5 4 6; 1 7 5; 3 9 6]
AB = zeros(3,3)
for i=1:3
  for j=1:3
    for k=1:3
      AB(i,j) = AB(i,j) + A(i,k)*B(k,j);
    end
  end
end
AB
input('Druecke RETURN');
```

Oft will man abhängig von Werten von Variablen unterschiedliche Anweisungen ausführen. Dazu dient die 'if' Anweisung. Nach 'if' steht eine logische Aussage. Ist diese wahr, werden die Anweisungen im nächsten Block (bis zum nächsten 'end') ausgeführt

```
for i=1:10
  i
  if i>5
    'groesser als 5'
  end
end
input('Druecke RETURN');
```

Man kann auch einen Block für den Fall, dass die Aussage falsch ist, angeben. Dazu dient 'else'

```
for i=1:10
  i
  if i>5
```

```
    'groesser als 5'
  else
    'kleiner oder gleich 5'
  end
end
input('Druecke RETURN');
```

Tatsächlich kann man eine ganze Menge verschiedener Fälle abarbeiten

```
for i=1:10
  i
  if i>5
    'groesser als 5'
  elseif i==5
    'gleich 5'
  else
    'kleiner als 5'
  end
end
```

Beachte: Um auf Gleichheit zu testen, muss man '==' schreiben. Das einfache Gleichheitszeichen '=' bedeutet Zuweisung

Die 'while' Schleife ermöglicht es, eine oder mehrere Operationen so oft auszuführen, bis die Bedingung am Anfang der Schleife nicht mehr erfüllt ist. Der mehrfach auszuführende Block von Operationen wird mit 'end' beendet.

Beispiel: Verdoppeln einer Zahl bis eine Obergrenze erreicht ist:

```
i = 1
while i<1000
  i = i*2
end
input('Druecke RETURN');
```

Manchmal muss man eine Bedingung jeweils nach Ablauf des Anweisungsblocks testen, z.B. weil die zu überpüfende Größe erst in dem Block berechnet wird. In diesem Fall muss man sich mit einem Trick behelfen.

Beispiel: Division einer Zahl, bis die Differenz zum vorherigen Ergebnis kleiner als eine vorgegebene Schranke ist

```
i = 1000
diff = 10
while diff > 5
```

```
   i_alt = i;
   i = i/3
   diff = abs(i - i_alt)
end
input('Druecke RETURN');
```

Eine andere Art, diese Schleife zu programmieren, bietet die 'break' Anweisung. Diese bewirkt, dass die Ausführung der Schleife auf der Stelle abgebrochen wird. Eine Anwendung macht daher nur in einer 'if' Anweisung Sinn.

```
i = 1000
while 1==1
   i_alt = i;
   i = i/3
   diff = abs(i - i_alt)
   if diff <= 5
      break
   end
end
input('Druecke RETURN');
```

Hier haben wir in der 'while' Anweisung eine Bedingung eingesetzt, die immer wahr ist. Die Schleife kann also nur durch die 'break' Anweisung verlassen werden. Selbstverständlich kann es sinnvoll sein, mehrere Kriterien an den Abbruch der Schleife zu stellen.

Die 'break' Anweisung kann auch in der 'for' Schleife eingesetzt werden.

```
for i=1:10
   i
   j = i^2;
   if j>50
      break
   end
   j
end
input('Druecke RETURN');
```

Bei verschachtelten Schleifen beendet 'break' nur die innerste Schleife:

```
for i=1:10
   for j=1:i
      if j>5
```

```
         break
      end
      j
   end
end
```

Noch ein Nachtrag zu den logischen Aussagen, wie sie in der 'while' oder 'if' Anweisung auftreten: Wir hatten bereits erwähnt, dass Gleichheit mittels '==' getestet wird. Ungleichheit wird mit ' =' getestet, und kleiner gleich bzw. größer gleich mit '¡=' bzw. '¿='

1.3 Plotten von Daten und Funktionen

1.3.1 Grundlagen

Beispiel 1: Plotten von Daten

Wir definieren einen kleinen Datensatz mittels zweier Vektoren:

```
t  = [1; 2; 3; 4; 5]
m  = [0.9; 3.8; 7.9; 15; 26.7]
```

Mit der 'plot' Anweisung kann man nun die Daten gegeneinander grafisch darstellen

```
plot(t,m,'.')
input('Druecke RETURN');
```

Statt dem Punkt '.' kann man auch viele andere Symbole verwenden, z.B. ein Kreuz 'x':

```
plot(t,m,'x')
input('Druecke RETURN');
```

Beispiel 2: Plotten von Funktionen

Funktionswerte, die grafisch dargestellt werden sollen, müssen in einen Vektor umgewandelt werden. Dazu definiert man zunächst einen Vektor mit den Stützstellen, an denen die Funktion ausgewertet werden soll, hier das Intervall von 1 bis 5 mit einem Abstand von 0.1 zwischen je zwei Stützstellen

```
tt = [1:0.1:5];
```

Dann weist man einem weiteren Vektor die Werte der Funktion (hier $f(t) = t^2$) zu. Beachte den '.' vor dem mathematischen Operator, der bewirkt, dass die Operation komponentenweise im Vektor tt ausgeführt wird.

```
y  = tt.^2;
```

Jetzt können wir plotten. Der Strich '-' bewirkt, dass der Graph als Linie dargestellt wird

```
plot(tt,y,'-')
input('Druecke RETURN');
```

Beispiel 3: Gemeinsames Plotten von Daten und Funktionen

Hierzu ist nichts weiter zun tun, als die einzelnen Argumente nacheinander in den plot Befehl zu schreiben

```
plot(t,m,'x',tt,y,'-')
```

1.3.2 Erweiterte Grundlagen

Das elementare Plotten von Funktionen haben wir bereits kennen gelernt. In diesem M-File wollen wir einige weitere Möglichkeiten von MATLAB ausprobieren.

Zunächst wollen wir eine Möglichkeit kennen lernen, um verschiedene mit 'plot' erzeugt Grafiken in ein Bild zu zeichnen. Wir bereiten dazu zunächst zwei Funktionen zum Plotten vor.

```
t=[0:0.01:2*pi];
x=sin(t).*cos(2.*t).^2;
y=cos(t).*sin(2.*t).^2;
```

Wenn wir nun zuerst

```
plot(t,x,'r-')
input('Druecke RETURN')
```

und dann

```
plot(t,y,'g-')
input('Druecke RETURN')
```

aufrufen, löscht die zweite Grafik die erste.

Nebenbemerkung: Mit den Buchstaben vor der Formatanweisung wählt man Farben aus. Zum Beispiel stehen zur Verfügung:
k = schwarz (blacK)
w = weiss (White); (auf weissem Hintergrund sinnlos :-))
r = rot (Red)
g = grün (Green)
b = blau (Blue)

y = gelb (Yellow)
m = magenta (wie die TelekoM)
c = türkis (Cyan)

Um dieses Löschen zu vermeiden dient die Anweisung 'hold on': Die bereits dargestellte Grafik wird gehalten.

```
plot(t,x,'r-')
hold on
input('Druecke RETURN')
plot(t,y,'g-')
input('Druecke RETURN')
```

Der 'hold on' Befehl wirkt bereits auf die letzte dargestellte Grafik (falls vorhanden). Um den normalen Lösch-Modus wieder einzuschalten, dient 'hold off'

```
hold off
plot(t,y,'g-')
input('Druecke RETURN')
```

Es kann auch sinnvoll sein, verschiedene Grafiken gleichzeitig in verschiedenen Fenstern auszugeben. Mit der 'figure' Anweisung erzeugt man weitere Grafikfenster. Die 'plot' Anweisung wirkt immer auf das letzte geöffnete Fenster.

```
figure
input('Druecke RETURN')
plot(t,x,'r-')
```

Alternativ kann man weitere Grafik-Fenster auch aus dem Menu eines bereits geöffneten Grafik-Fensters öffnen.

Als nächstes wollen wir 2d Kurven plotten. Mathematisch ist 2d Kurve eine Funktion von R nach R^2. In MATLAB kann man diese als vektorwertige Funktion oder einfach mittels zweier reellwertiger Funktionen darstellen. In diesem Sinne bilden die zwei oben definierten Funktionen bereits eine Kurve. Das Argument einer Kurve wird oft mit 't' bezeichnet.

Wenn man Kurven grafisch darstellen möchte, gibt es im Wesentlichen zwei Möglichkeiten: Entweder man stellt sie - so wie oben - koordinatenweise in Anhängigkeit von t dar (entweder mit zwei plot Anweisungen und 'hold on', oder in einer plot Anweisung):

```
plot(t,x,'r-',t,y,'g-')
input('Druecke RETURN')
```

Meistens jedoch ist man an einer Darstellung der Kurve in der (x,y)-Ebene interessiert. Hierbei bleibt das Argment 't' in der Grafik unsichtbar; man verliert also die Information über die Abhängigkeit von 't', sieht dafür aber den Zusammenhang zwischen 'x' und 'y'. In MATLAB geht dies ganz einfach, indem man den Vektor 'x' gegen den Vektor 'y' plottet.

```
plot(x,y,'b-')
input('Druecke RETURN')
```

(Diese spezielle Kurve ist übrigens eine sogenannte 'Lissajou-Kurve')

Zum Abschluss wollen wir noch einige Möglichkeiten erläutern, mit denen man Grafiken schöner gestalten kann.

Angabe eines Titels:

```
title('Eine Lissajou-Kurve', 'FontSize', 16)
input('Druecke RETURN')
```

Beschriftung der Achsen:

```
xlabel('x=sin(t)cos(2t)^2')
ylabel('y=cos(t)sin(2t)^2');
input('Druecke RETURN')
```

Eine Legende hinzufügen

```
legend('Beispielkurve')
input('Druecke RETURN')
```

Texte an beliebigen Stellen im Bild anbringen

```
text(-1,-0.1,'\uparrow Hier ist ein Knick', 'FontSize', 12)
text(0.2,0.6,'\leftarrow Hier ist die Kurve glatt', 'FontSize', 12)
input('Druecke RETURN')
```

Achtung: MATLAB kann nur einen Teil der LaTeX-Symbole darstellen. Taucht in einer Anweisung ein unbekanntes Symbol auf, so werden alle Symbole in dieser Anweisung ignoriert!

Schliesslich kann man Text in der Grafik noch mit der Maus positionieren, was zum Beispiel sinnvoll ist, wenn man die Grafik danach abspeichern oder ausdrucken will. Dies geht mit

```
gtext('Ein Text mit der Maus')
```

Nach der Anweisung kann man den Text mit der Maus im Bild einfügen.

1.3.3 3D Grafik

Nachfolgend werden verschiedene Möglichkeiten zur Erzeugung von dreidimensionaler Grafik vorgestellt.

Wir haben zuvor bereits das Plotten von zweidimensionalen Kurven betrachtet; als erste 3d Anwendung erweitern wir dies auf dreidimensionale Kurven. Wir betrachten die Kurve $t \rightarrow (sin(t), cos(t), t)$ für $t \in [0, 10\pi]$

```
t = [0:pi/50:10*pi];
x = sin(t);
y = cos(t);
z = t;
```

Ganz analog zum 'plot' Befehl funktioniert der 'plot3'-Befehl:

```
plot3(x,y,z,'r-')
input('Druecke RETURN')
```

Wir können erzwingen, dass das Koordinatensystem mit gleichen Kantenlängen dargestellt wird

```
axis square;
input('Druecke RETURN')
```

und wir können ein Gitter einblenden, das die 3d Sicht der Kurve erleichtert

```
grid on;
input('Druecke RETURN')
```

Eine weitere wichtige Anwendung dreidimensionaler Grafik ist die Darstellung von Flächen im R^3. Solche Flächen können über eine Funktion $f : R^2 \rightarrow R$ definiert werden. Hier betrachten wir als Beispiel die Funktion $f(x, y]) = 1 - \|(x, y)\|^2$

Analog zur üblichen Darstellung eindimensionaler Funktionen muss man die zur Darstellung verwendeten Punkte definieren. Statt eines Vektors braucht man jetzt aber eine Matrix von Punkten, die das darzustellende Gebiet abdecken. Genauer brauchen wir zwei Matrizen: Eine für die x-Komponenten der Punkte und eine für die y-Komponenten.

Diese können effizient mit der 'meshgrid'-Anweisung erzeugt werden.

```
[X,Y] = meshgrid(-8:.5:8,-8:.5:8);
```

Dann werten wir die Funktion aus und speichern die Punkte in Z

```
Z = 1-(X.^2+Y.^2);
```

und stellen das Ganze mit 'mesh' dar.

```
mesh(X,Y,Z);
input('Druecke RETURN')
```

Falls gewünscht, können die verdeckten Linien sichtbar gemacht werden

```
hidden off;
input('Druecke RETURN')
```

Statt als Gitter kann die Funktion als Fläche dargestellt werden. Dazu benutzt man den 'surf'-Befehl.

```
surf(X,Y,Z);
input('Druecke RETURN')
```

Die Farben werden dabei standardmässig durch die Z-Werte definiert. Man kann aber auch eine weitere Matrix von der Grösse der Z-Matrix als viertes Argument übergeben, wobei dann diese als Farbwert benutzt wird.

Als Beispiel verwenden wir die Norm der Ableitung Df der Funktion f, die gegeben ist durch $Df(x, y) = -2(x, y)$

```
C = 2.*sqrt(X.^2 + Y.^2);
surf(X,Y,Z,C);
input('Druecke RETURN')
```

Die Verteilung der Farben wird durch das Farbschema ('colormap') gesteuert. Man kann sich die Verteilung als Farbbalken anzeigen lassen.

```
colorbar
input('Druecke RETURN')
```

MATLAB stelle verschiedene Standard-Farbschemen zur Verfügung, z.B. hsv, hot, cool, summer, gray (siehe 'help graph3d' für eine vollständige Liste, Voreinstellung ist 'jet'). Bei der Auswahl eines Farbschemas kann als optionaler Parameter die Anzahl der verwendeten Farben übergeben werden, z.B. 'hot(10)'. Voreinstellung ist '64'.

```
colormap(hot);
input('Druecke RETURN')
colormap(jet);
```

Funktionen von R^2 nach R kann man auch als Höhenlienien darstellen Wir definieren dazu zunächst eine etwas interessantere Funktion

```
Z2 = sin(X./2) + sin(Y./3);
surf(X,Y,Z2);
input('Druecke RETURN')
```

Mit 'contour' kann man die Höhenlienien plotten. Der vierte Parameter gibt die Anzahl der dargestellten Niveaus an

```
contour(X,Y,Z2,20);
input('Druecke RETURN')
```

Alternativ kann man die Höhenlienien auch dreidimensional plotten.

```
contour3(X,Y,Z2,20);
input('Druecke RETURN')
```

Zurück zu den 2d Höhenlinien Hier kann man die Höhenlienien wie folgt beschriften

```
[L,h] = contour(X,Y,Z2,20);
clabel(L,h);
input('Druecke RETURN')
```

Alternativ kann man die Funktionswerte auch mit Farben kennzeichnen

```
contourf(X,Y,Z2,20);
colorbar;
input('Druecke RETURN')
```

Schliesslich kann man auch nur eine einzelne Höhenlinie darstellen, indem man als viertes Argument einen 2d Vektor mit 2 mal diesem Wert übergibt.

```
contour(X,Y,Z2,[0.5 0.5]);
input('Druecke RETURN')
```

Wir kommen nun zurück zu den Flächendarstellungen und betrachten wieder unsere erste Beispielfunktion

```
surf(X,Y,Z,C);
colormap(hot);
input('Druecke RETURN')
```

Wir wollen nun Lichteffekte hinzufügen. Um diese effektiv einzusetzen, empfiehlt es sich, die Oberflächeneigenschaften unserer Fläche zuerst geeignet anzupassen. Dazu kann man mit dem 'findobj'-Befehl eine Datenstruktur (hier 'h') anlegen, in der diese Eigenschaften festgelegt sind.

```
h = findobj('Type','surface');
```

Diese Datenstruktur 'h' kann mit Hilfe des 'set'-Befehls verändert werden. Wir ändern zunächst die Farben der Kanten ('Edges') und Facetten ('Faces') so, dass ein kontinuierliches Farbverlauf erreicht wird.

```
set(h,'EdgeColor','interp');
set(h,'FaceColor','interp');
input('Druecke RETURN');
```

Mit der 'light'-Anweisung können wir nun Lichtquellen platzieren.

```
light('Position',[ 1  3  2]);
light('Position',[-3 -1  3]);
input('Druecke RETURN')
```

Jetzt ist das Gitter wieder sichtbar; der Grund liegt darin, dass die Voreinstellung zur Behandlung von Lichtreflexen für die Facetten und die Kanten unterschiedlich sind. Wir konnen das Reflex-Verhalten wiederum mit dem 'set'-Befehl einstellen. Hier zwei verschiedene Möglichkeiten:

```
set(h,'FaceLighting','flat',...
      'EdgeLighting','flat');
input('Druecke RETURN')
set(h,'FaceLighting','phong',...
      'EdgeLighting','phong');
```

Ein Überblick über die vielen weiteren Möglichkeiten, die Oberflächeneigenschaften einzustellen findet sich unter

```
http://www.mathworks.com/access/helpdesk/help/techdoc/ref/surface_props.shtml
```

Schliesslich kann man die Achsen komplett abschalten, um die Fläche ganz allein darzustellen

```
axis vis3d off
input('Druecke RETURN')
```

Als letzten Punkt betrachten wir noch die Einstellung des Beobachtungspunktes ('viewpoint'). Wir nehmen dazu noch einmal unsere zweite Funktion

```
surf(X,Y,Z2);
input('Druecke RETURN')
```

Der Viewpoint wird über zwei Winkel definiert:

Azimuth = Rotation in der (x,y)-Ebene
Elevation = Neigewinkel bzgl. der (x,y)-Ebene

Am einfachsten kann man diese mit der Maus direkt in der Grafik einstellen. Dazu klickt man auf das Rotations-Symbol (ganz rechts in der Symvbolleiste), bewegt die Maus in die Grafik, hält die rechte Taste gedrückt und kann nun den Viewpoint verdrehens.

Während des Drehens werden die Azimuth und Elevation-Werte im Grafik-Fenster angezeigt.

Will man diese Werte (zu Beispiel aus einem M-File) über eine Anweisung angeben, so kann man das mit 'view' machen. Dabei muss ein Vektor mit den zwei Werten [Azimuth, Elevation] übergeben werden.

```
view([-12 52])
```

1.3.4 (*) Handle-Graphics

Die Handle-Graphics bieten die Möglichkeit, Eigenschaften von Grafiken in MATLAB über die Standard-Befehle hinaus direkt zu manipulieren. Wir wollen hier zunächst das allgemeine Konzept kurz erläutern und dann mit einigen Anwendungsbeispielen illustrieren. Natürlich kann dieses M-File nur einen kleinen Einblick in die Möglichkeiten der Handle-Graphics geben.

Jedes grafische Objekt besteht aus einer Reihe von Komponenten, wie z.B. dem Grafikfenster ('figure'), den Koordinatenachsen ('axes'), einzelnen grafischen Objekten (z.B. Linien 'line', Flächen 'surface') und weiteren Objekten wie z.B. Lichtquellen 'light'. Zudem besitzt jede Grafik die Komponente 'root', die den gesamten Bildschirm symbolisiert. Der 'Handle' bietet nun die Möglichkeit, jedes dieser Objekte einzeln anzusprechen und zu manipulieren.

Um dies zu illustrieren, beginnen wir mit einem kleinen Beispiel:

```
t = [0:10];
plot(t,t.^2,'o-')
input('Druecke RETURN')
```

Den Handle auf diese Grafik erhalten wir mit dem Befehl

```
h = findobj
```

der sich immer auf die zuletzt erzeugte Grafik bezieht.

Die Werte in h sind Zeiger auf die Komponenten der Grafik.

Die Bedeutung der Komponenten erhält man mit dem Befehl

```
get(h,'type')
input('Druecke RETURN')
```

Wir haben in dieser Grafik also die Komponenten

h(1) = root = Bildschirm
h(2) = figure = Grafikfenster
h(3) = axes = Koordinatenachsen
h(4) = line = Linie (inklusive der Markierungen)

Mit dem 'set'-Befehl kann man diese Komponenten nun manipulieren. Gibt man nur

```
set(h(3))
```

ein, so erhält man eine Liste der Eigenschaften mitsamt der möglichen Einstellungen. Die Voreinstellungen sind in geschweiften Klammern angegeben

```
input('Druecke RETURN')
set(h(4))
```

Alle Eigenschaften in dieser ziemliech grossen und unübersichtlichen Liste können nun direkr manipuliert werden. Wir illustrieren dies anhand der Linie und wollen dort zunächst die Markierungen 'Marker' verändern. Mit

```
input('Druecke RETURN')
set(h(4),'Marker')
```

erhält man eine Liste der möglichen Einstellungen. Wir wählen nun z.B. Quadrate statt der vorhandenen Kreise

```
set(h(4),'Marker', 'square')
```

und ändern auch noch ihre Grösse

```
set(h(4),'MarkerSize', 16)
```

Diese beiden Anweisungen könnte man auch mittels

```
 set(h(4),'Marker', 'square', 'MarkerSize', 16)
```

in einer Anweisung zusammenfassen.

Das Gegenstück zu 'set' ist die 'get'-Anweisung, mit der man Eigenschaften ermitteln kann:

```
get(h(4),'MarkerSize')
input('Druecke RETURN')
```

Handles können nicht nur über 'findobj' erhalten werden. Jede Grafik-Anweisung gibt als Funktionswert einen Handle auf das erzeugte Objekt zurück:

```
h1 = plot(t,sqrt(t),'-')
```

Allerdings erhält man hier keinen Vektor von Objekten, sondern nur ein einzelnes Objekt, nämlich gerade das mit dem Befehl erzeugt; in diesem Beispiel also eine Linie.

```
get(h1,'Type')
input('Druecke RETURN')
```

Neben der Einstellung verschiedenster Parameter können Handles auch dazu benutzt werden, Grafikobjekte wieder zu entfernen. Als Beispiel betrachten wir die 3d Grafik mit Lichteffekten aus der Beschreibung zuvor, die wir hier noch einmal erzeugen.

```
[X,Y] = meshgrid(-8:.5:8,-8:.5:8);
Z = 1-(X.^2+Y.^2);
C = 2.*sqrt(X.^2 + Y.^2);
surf(X,Y,Z,C);
colormap(hot);
hs = findobj('Type','surface');
set(hs, 'EdgeColor','interp',...
        'FaceColor','interp',...
        'FaceLighting','phong',...
        'EdgeLighting','phong');
input('Druecke RETURN');
```

Jetzt können wir auch die Bedeutung dieser Befehle besser verstehen; ausserdem sieht man hier eine Variante der 'findobj'-Anweisung, die durch Angabe des Typs 'surface' nur den Handle des Surface-Objekts liefert.

Mit der 'light'-Anweisung können wir nun Lichtquellen platzieren. Wir speichern beim Einschalten den Handle in der Variable lh

```
lh = light('Position',[-3 -1  3]);
```

lh ist - wie zu erwarten - ein Handle vom Typ 'light'

```
get(lh, 'Type')
input('Druecke RETURN');
```

Jedes grafische Objekt kann mit dem Befehl 'delete' aus der Grafik gelöscht werden. Zum Abschalten sagt man also einfach

```
delete(lh)
input('Druecke RETURN');
```

Wenn h ein Handle auf ein Grafikfenster (Typ 'figure') ist, kann das zugehörige Fenster mit 'close(h)' geschlossen werden.

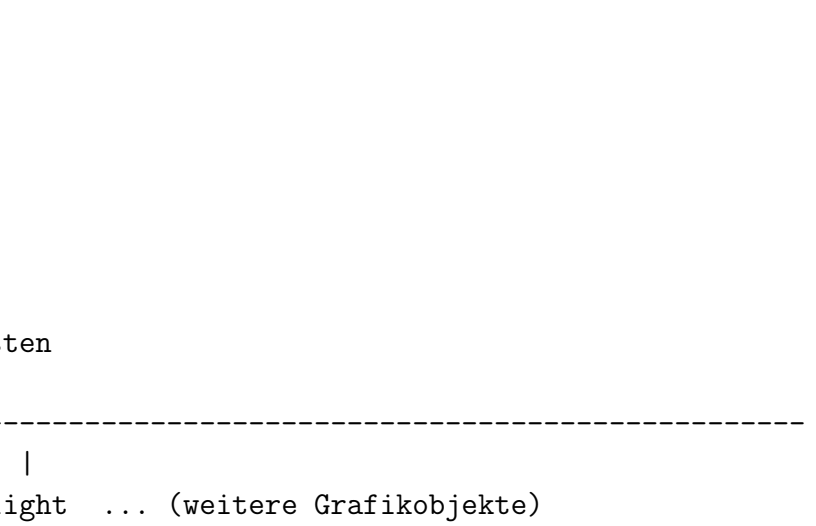
```
close(h(2))
```

Wenn man die Handle-Graphics verwenden will, empfiehlt es sich, immer gleich nach Erzeugen der Grafik einen Handle auf die Grafik zu speichern.

Es gibt allerdings einige globale Variablen, die (falls eine Grafik vorhanden ist) Handles gewisser Objekte enthalten:

gca = Handle der aktuellen Achsen
gcf = Handle des aktuellen Grafikfensters
gco = Handle des aktuellen Grafikobjektes (Linie, Fläche etc.)

Die einzelnen Objekte sind nicht einfach ungeordnet gespeichert, sondern nach einer Ordnung, der sogenannten 'Hierarchie':



Die tiefer liegenden Objekte werden dabei als Kinder 'children' der darüberliegenden Objekte bezeichnet.

Diese Hierarchie wird von einigen Eigneschaften verwendet. Als Beispiel betrachten wir die 'nextplot' Eigenschaft des 'axes'-Objektes

```
surf(X,Y,Z,C);
set(gca,'nextplot')
```

Hier gibt es die Einstellungen add, replace (Voerinstellung) und replacechildren

'add' entspricht dem 'hold on' Modus während 'replace' dem 'hold off' Modus entspricht.

Bei 'replacechildren' wird beim Zeichnen eines neuen Grafikobjektes jeweils das Alte gelöscht, wobei die Achsen aber erhalten bleiben.

Beispiel:

```
set(gca,'nextplot','replacechildren');
surf(X,Y,Z./2,C);
input('Druecke RETURN')
```

Hier bleiben die Achsen erhalten, statt - wie üblich - automatisch angepasst zu werden.

Eine schöne Anwendung dieses Modus ist die Erstellung von Animationen. Hierzu bietet MATLAB die 'movie'-Anweisung, die einen Vektor von Grafiken in einer vorgegebenen Reihenfolge abspielt. Die einzelnen Grafiken können dabei mit der Anweisung 'getframe' in eine Bild-Datenstruktur umgewandelt werden und in den Komponenten von F abgelegt werden. Der 'replacechildren' Modus ist dabei wichtig, damit die Achsen von Bild zu Bild gleich bleiben.

Erzeugen eines 'Testbildes'

```
surf(X,Y,Z,C)
```

Einstellen der Achsen auf einen sinnvollen Bereich

```
axis([-10 10 -10 10 -140 140])
input('Druecke RETURN')
```

Einstellen des 'replacechildren' Modus

```
set(gca,'nextplot','replacechildren')
```

Erzeugen der Bilder

```
for j = 1:21
    surf(X,Y,cos(2*pi*(j-1)/20).*Z,Z)
    F(j)=getframe;
end
input('Druecke RETURN')
```

Abspielen des Films

```
movie(F)
input('Druecke RETURN')
```

Abspielen des Films mit 9 Wiederholungen (also 10 mal, Voreinstellung 1 Wiederholung) und 24 Bildern pro Sekunde (Voreinstellung 12)

```
movie(F,9,24)
input('Druecke RETURN')
```

Abspielen des Films mit 9 Wiederholungen (Voreinstellung 1) und 24 Bildern pro Sekunde (Voreinstellung 12) sowie der in dem angegebenen Vektor eingestellten Bildfolge

```
movie(F,[9 1 2 3 4 5 6 6 6 6 5 4 3 2 1], 24)
```

1.4 Ausgabe

MATLABs Standardausgabe von Rechenergebnissen ist weder besonders schön noch besonders übersichtlich. Eine sehr leistungsfähige Abhilfe bietet die 'fprintf'-Anweisung. Mit ihr kann man Text und Werte von Variablen in beliebiger Form ausgeben lassen. Die folgenden Beispiele erläutern einige wesentliche Anwendungsmöglichkeiten.

Zunächst einmal ist es sehr einfach, mit 'fprintf' Text auszugeben Beachte: Mit dem Ende der 'fprintf' wird nicht automatisch in die nächste Zeile gewechselt. Ein Zeilenende muss explizit durch die Zeichenfolge '$\backslash n$' erzeugt werden. Leerzeilen erzeugt man mit 'fprintf('$\backslash n$')'

```
fprintf('Dies ist ein Text')
fprintf('...und hier geht es in der selben Zeile weiter\n')
fprintf('Jetzt beginnt eine neue Zeile\n')
fprintf('\n')
input('Druecke RETURN')
```

Um Variablenwerte auszugeben, muss im Text ein Platzhalter eingegeben werden, der das Format der Ausgabe bestimmt. Für Gleitkommazahlen gibt es die Format-Platzhalter '%e', '%f' oder '%g'.
'%e' gibt die Zahl immer in Exponenten-Darstellung aus
'%f' gibt die Zahl immer ohne Exponenten aus
'%g' wechselt zwischen diesen Formaten, je nach dem, in welchem Zahlenbereich die auszugebende Zahl liegt. Außerdem werden unwichtige Nachkommastellen (Nullen) abgeschnitten.

Diese Platzhalter legen nur das Format und die Position der Ausgabe im laufenden Text fest. Die eigentlich auszugebende Variable(n) muss (müssen) dann als weitere Argumente an die 'fprintf' Anweisung übergeben werden.

```
x = 123456.789876
fprintf('Verschiedene Formate\n')
fprintf('e: %e\n',x)
fprintf('f: %f\n',x)
fprintf('g: %g\n\n',x)
input('Druecke RETURN')
```

Zusätzlich zum Format kann man auch eine Mindestanzahl von Stellen angeben, die für die Zahl bereitgestellt werden sollen. Dies ist für Tabellen sehr nützlich. Wir drucken eine Tabelle zunächst ohne Stellenangabe.

```
for i = 1:10
   wurzel = sqrt(i);
   fprintf('Die Zahl %g ist Quadratwurzel von %g\n',wurzel,i)
end
fprintf('\n')
input('Druecke RETURN')
```

Jetzt formatieren wir dies schöner indem wir die Stellen festlegen. Man kann die Anzahl der Gesamtstellen und (wahlweise) zusätzlich die Anzahl der Nachkommastellen angeben, indem man diese zwischen '%' und Formatzeichen schreibt. Für das Format '%f' mit 10 Stellen schreibt man z.B. '%10f'. Um von den 10 Stellen 5 für Nachkommastellen zu reservieren, schreibt man '%10.5f'. Wir benutzen dies für die schönere Ausgabe der Quadratwurzeltabelle

```
for i = 1:10
   wurzel = sqrt(i);
   fprintf('Die Zahl %10.8g ist Quadratwurzel von %2g\n',wurzel,i)
end
fprintf('\n')
input('Druecke RETURN')
```

Eine weitere Anwendung der Stellenangabe ist es, die Genauigkeit der Ausgabe zu erhöhen

```
fprintf('Die Wurzel von 2 ist %f\n',sqrt(2))
fprintf('Die Wurzel von 2 ist %20.18f\n',sqrt(2))
input('Druecke RETURN')
```

'fprintf' kann nicht nur einfache Zahlen, sondern auch Vektoren und Matrizen ausgeben. Hierbei wird die entsprechende Formatanweisung für jedes Element der Matrix wiederholt. Hierbei wird spaltenweise vorgegangen, so dass die gewohnte Reihenfolge (zeilenweise) durch Transponieren der auszugebenden Matrix erzeugt werden muss.

```
A = [1.5 2.7 3.456; 7.6 4.765 1.234; 3 4 5]
```

Ein erster Versuch, der aber nicht so besonders schön aussieht

```
fprintf('Eine Matrix %f\n',A')
fprintf('\n')
input('Druecke RETURN')
```

Eine schönere Variante

```
fprintf('Eine Matrix\n')
fprintf('%f %f %f\n',A')
```

Kapitel 2

Matlab, ein mathematisches Labor

2.1 Lineare Gleichungssysteme

Man kann lineare Gleichungssysteme $Ax = b$ mittels der Inversen der Matrix A lösen:

```
x=inv(A)*b
```

Dies ist aber numerisch ineffizient im Vergleich zu anderen Vorgehensweisen.

2.1.1 Direkte Verfahren

MATLAB hat eine Reihe von Algorithmen zur Lösung von linearen Gleichungssystemen und verwandten Problemen eingebaut. Sei

```
A = [ 1 5 6; 7 9 6; 2 3 4]
b = [29; 43; 20]
```

Der Grundbefehl zur Lösung eines linearen Gleichungssystems ist der umgekehrte Schrägstrich

```
x=A\b
```

Die Rechnung wird mit Gauß-Elimination (mit geschickter Pivotierung) durchgeführt Wenn A in obererer oder unterer Dreiecksform ist, wird Rückwärts bzw. Vorwärtseinsetzen durchgeführt. Anmerkung: Wenn A nicht quadratisch ist, wird keine Fehlermeldung ausgegeben, sondern automatisch das zugehörige lineare Ausgleichsproblem gelöst.

Neben der direkten Lösung von linearen GS stehen Algorithmen zur Zerlegung von Matrizen zur Verfügung.

'lu' zerlegt per Gauß-Elimination eine Matrix in eine rechte untere und eine linke obere Dreiecksmatrix - bis auf Zeilenvertauschungen durch Pivotierung

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

'chol' zerlegt symmetrische, positiv definite Matrizen mittels des Choleski-Verfahrens. Achtung: Im Gegensatz zur Vorlesung wird hier 'R' mit R'*R=A berechnet, nicht 'L' mit L*L'=A

```
AA = A'*A        Hinweis: erzeugt eine symm, pos. def. Matrix
R = chol(AA)
```

zuletzt gibt es noch die QR Zerlegung, die wir in der Vorlesung nicht besprochen haben. Hier wird A=Q*R zerlegt, wobei R eine obere Dreiecksmatrix ist und Q eine orthogonale Matrix ist, d.h. es gilt $Q^{-1} = Q^T$

```
[Q,R] = qr(A)
```

Die 2 Norm der Matrix Q ist immer 1:

```
norm(Q,2)
```

2.1.2 Iterative Verfahren

MATLAB stellt eine Reihe von Verfahren zur iterativen Lösung linearer Gleichungssysteme zur Verfügung

Zunächst definieren wir uns ein Beispiel-Gleichungssystem und zwar eins, das sehr schlecht konditioniert ist

```
A = [0.780 0.563; 0.913 0.659];
b = [0.217; 0.254];
cond(A,2)
```

Die exakte Lösung dieses Systems ist [1; -1]. Wir werden nun einige wichtige Verfahren durchgehen, die alle - im Gegensatz zum Jacobi oder Gauss-Seidel Verfahren - für allgemeine quadratische Matrizen funktionieren. Im Einzelnen sind dies
bicgstab - 'stabilisiertes bikonjugiertes Gradientenverfahren'
cgs - 'quadriertes konjugiertes Gradientenverfahren'
bicg - 'bikonjugiertes Gradientenverfahren'
gmres - 'verallgemeinertes Minimalresiduumsverfahren'

Bei dem ersten Beispiel, dem 'bicgstab' Verfahren, betrachten wir genauer, welche Parameter angegeben werden können Alle Verfahren haben die gemeinsame Eigenschaft, dass sie verschiedene Abbruchkriterien verwenden. Nach Abbruch des Verfahrens wird ausgegeben, aus welchem Grund die Iteration beeendet wurde.
1) Abbruch bei erreichter gewünschter Genauigkeit: Das Abbruchkriterium ist bei den Verfahren in MATLAB anders als die einfache Kriterium aus der Vorlesung. Hier wird für eine vorgegebene Toleranz 'tol' so lange iteriert, bis das relative Residuum $\|Ax - b\|/\|b\| < tol$ (für die 2-Norm) ist. Der Vorteil dieses Verfahrens liegt darin, dass sich der Fehler in der Lösung dann aus der Kondition der Matrix A mittels des Satzes 2.32 (Fehleranalyse) der Vorlesung abschätzen lässt. Die gewünschte Genauigkeit ist mit 1e-6 voreingestellt.
2) Abbruch nach maximaler Anzahl an Iterationen: Wenn eine maximale Anzahl von Iterationen erreicht ist, wird das Verfahren in jedem Fall abgebrochen, auch wenn die gewünschte Genaugkeit noch nicht erreicht ist. Die maximale Anzahl der Iterationen ist mit n (=Dimension des Problems) voreingestellt.
3) Abbruch bei Stagnation: Wenn sich das Residuum nicht mehr verbessert, wird ebenfalls abgebrochen, auch wenn die gewünschte Genaugkeit noch nicht erreicht ist.

Wir illustrieren nun die verschiedenen Abbruchkriterien: Beachte: Die Anzahl der Iterationen, die nach dem Abbruch angegeben wird, kann ein nichtganzzahliger Wert sein, da viele Verfahren mehrstufige Iterationsschritte durchlaufen, die nach einem Teildurchlauf abgebrochen werden können.
Wir rufen das Verfahren zunächst ohne weitere Paremeter auf.

```
bicgstab(A,b)
input('druecke RETURN')
```

Hier tritt nun ein Abbruch wegen erreichter maximaler Anzahl der Iterationen auf. Als optionale Paramater können nun die Toleranz als drittes Argument und die maximale Anzahl der Iterationen als viertes Argument angegeben werden. Wir erhöhen nun die Anzahl der Iterationen und belassen die Toleranz bei dem voreingestellten Wert 1e-6

```
bicgstab(A,b,1e-6,20)
input('druecke RETURN')
```

Hier tritt ein Abbruch bei erreichter Genauigkeit ein, das Ergebnis ist aber noch nicht so ganz zufriedenstellend, da A schlecht konditioniert ist. Wir erhöhen also die gewünschte Toleranz:

```
bicgstab(A,b,1e-10,20)
input('Druecke RETURN')
```

Jetzt kann man mit dem Ergebnis zufrieden sein.

Wir verwenden jetzt den gleichen Aufruf für andere Verfahren

```
cgs(A,b,1e-10,20)
input('Druecke RETURN')

bicg(A,b,1e-10,20)
input('Druecke RETURN')

qmr(A,b,1e-10,20)
input('Druecke RETURN')
```

Eine Ausnahme macht das gmres-Verfahren, das eine verschachtelte Iteration verwendet. Hier muss als dritter Parameter die maximale Anzahl der Schritte der inneren Iteration angegeben werden.

```
gmres(A,b,3,1e-10,20)
input('Druecke RETURN')
```

Welches Verfahren für welche Problemklasse am angemessensten ist, hängt stark vom betrachteten Problem ab.

2.2 Definition von Funktionen

2.2.1 Funktionen von Zahlen

Jede Funktion muss in ein eigenes M-File geschrieben werden.

Der Dateiname bestimmt den Funktionsnamen

Die erste Programmzeile des M-Files muss die Vereinbarung 'function a = name(b)' enthalten. Hierbei ist 'b' der Eingabeparameter und 'a' der Ausgabeparameter (die Namen sind beliebig). 'name' sollte zur besseren Übersichtlichkeit mit dem Funktionsnamen, also dem Dateinamen des M-Files übereinstimmen.

Beispiel: Berechne den Absolutbetrag einer reellen Zahl:

```
function y = testfunktion(x)
if x>0
   y=x;
else
   y=-x;
end
```

Beispielaufruf: 'testfunktion(-10)' oder 'a=testfunktion(-10)'

Funktionen können mehrere Ein- und Ausgabeparameter besitzen. Die Eingabeparameter werden dabei einfach durch Komma getrennt Die Ausgabeparameter werden durch Komma getrennt in eckige Klammern geschrieben

Beispiel: Berechne die zwei Lösungen einer quadratischen Gleichung $x^2 + px + q = 0$

```
function [x1,x2] = testfunktion2(p,q)
x1 = -p/2 + sqrt(p^2-4*q)/2;
x2 = -p/2 - sqrt(p^2-4*q)/2;
```

Achtung: Bei einem Aufruf 'testfunktion2(0,-1)' wird nur der erste Ausgabeparameter am Bildschirm ausgegeben. Um beide zu bekommen, muss die Funktion mittels '[a,b]=testfunktion2(0,-1)' aufgerufen werden.

2.2.2 Funktionen von Vektoren und Matrizen

Wenn eine reellwertige Funktion mit einem reellwertigen Argument mit einem Vektor oder einer Matrix aufgerufen wird, wird die Funktion elementweise fr jeden Eintrag des Vektors oder der Matrix ausgeführt.

Dies sollte man auch bei selbstgeschriebenen Funktionen beachten. Wenn man dabei nur bereits (korrekt!) definierte mathematische Operationen verwendet, genügt es, alle Multiplikationen, Divisionen und Potenzierungen mit dem '.' zu versehen

```
function y = vektorfunktion1(x)
  y = x.^2+2.*x+1;
```

Wenn eine reellwertige Funktion mit einem reellwertigen Argument mit einem Vektor oder einer Matrix aufgerufen wird, wird die Funktion elementweise fr jeden Eintrag des Vektors oder der Matrix ausgeführt.

Dies sollte man auch bei selbstgeschriebenen Funktionen beachten. Wenn man fr in der Funktion 'if'-Anweisungen verwendet, muss man dann etwas komplizierter programmieren. Die bereits bekannte (reelle) Betragsfunktion kann man wie folgt modifizieren, damit sie auch für Matrizen korrekt funktioniert.

```
function y = vektorfunktion2(x)
```

'size' liefert die Anzahl von Zeilen und Spalten von 'x' in einem 2-dimensionalen Vektor zurück

```
s = size(x);
```

Jetzt kann man den reellen Betrag komponentenweise berechnen

```
for i=1:s(1)
  for j=1:s(2)
    if x(i,j)>0
      y(i,j) =  x(i,j);
    else
      y(i,j) = -x(i,j);
    end
  end
end
```

2.2.3 Funktionen von Funktionen

Bisher haben wir Funktionen benutzt, deren Parameter Zahlen bzw. Vektoren oder Matrizen waren. Es gibt in MATLAB aber auch die Möglichkeit, ganze Funktionen an andere Funktionen als Parameter zu übergeben.

Als Beispiel für eine Anwendung programmieren wir hier eine Funktion, die eine gegebene Funktion plottet. Ein Beispielaufruf lautet z.B.

```
funplot('sin(x)',1,5,100)
```

Der erste Parameter bestimmt die zu plottende Funktion, der zweite und dritte die untere bzw. obere Intervallgrenze des zu plottenden Bereichs. Der vierte Parameter bestimmt die Anzahl der Stützstellen, die in der Grafik verwendet werden sollen.

```
function funplot(fun,a,b,n,format)
```

inline wandelt den übergebenen Funktionsnamen in eine ausführbare MATLAB Funktion um.

```
  g = inline(fun);
```

Wie üblich erzeugen wir den Vektor der Stützstellen

```
  t = [a:(b-a)/(n-1):b];
```

dazu den Vektor von Funktionswerten

```
  y = g(t);
```

und plotten das Ganze

```
  plot(t,y,format)
```

Diese Beispielfunktion 'funplot' plottet eine Funktion mit abhängiger Variable 'x'. Beispielaufruf:

```
funplot('sin(x)',0,2*pi,50,'-')
input('Druecke RETURN')
```

Zur Erläuterung der Parameter siehe oben. Die hier übergebene Funktion kann übrigens auch eine selbstdefininerte Funktion sein. Z.B.

```
funplot('vektorfunktion2(x)',-1,1,51,'-')
input('Druecke RETURN')
```

MATLAB bietet aber auch eine ganze Reihe eigene Funktionen, die in verschiedenster Weise auf anderen Funktionen arbeiten

Wir werden hier eine Reihe solcher Funktionen kennen lernen.

Zunächst einmal bietet MATLAB selbst mit 'fplot' eine eingebaute Grafik-Routine zur Darstellung von Funktionen. Der Funktion wird der Name der darzustellenden Funktion übergeben sowie der darzustellende Bereich als Vektor:

```
fplot('sin(x)',[0,2*pi])
input('Druecke RETURN')
```

'fplot' ist intern aufwändiger programmiert als unsere einfache 'funplot' Routine:

- das Argument '(x)' kann bei der Funktionsübergabe weggelassen werden.
- 'fplot' errechnet selbst eine brauchbare Anzahl von Stützstellen zur Darstellung der Funktion. Will man diese per Hand einstellen, kann die gewünschte Minimalanzahl N¿1 von Stützstellen als weiterer Parameter angegeben werden. Alternativ kann statt der Anzahl der Stützstellen ein Parameter 'tol' angegeben werden, der die Genauigkeit der Grafik steuert. Voreinstellung ist 2e-3.
- Auch der Bereich, den die y-Achse abdecken soll, kann eingestellt werden

```
fplot('sin(x)',[0,2*pi,-0.5,0.5])
input('Druecke RETURN')
```

Wir kommen nun zu Funktionen, die mathematische Operationen auf anderen Funktionen durchführen. Eine klassische Anwendung ist die Minimierung von Funktionen auf einem vorgegebenen Intervall. Dies macht 'fminbnd'. Hier wird die Funktion sowie die untere und obere Intervallgrenze des Intervalls angegeben, auf dem minimiert werden soll. Ausgegeben wird eine Stelle 'x', an der die Funktion ihr Minimum annimmt.

```
x = fminbnd('sin',0,2*pi)
input('Druecke RETURN')
```

Siehe die MATLAB Hilfe für die Optionen, die der Funktion 'fminbnd' übergeben werden können. Beachte: Die zugrundeliegenden Algorithmen garantieren im Allgemeinen nur, dass ein lokales Minimum gefunden wird.

Eine weitere Anwendung ist die Nullstellensuche. Wir werden einige numerische Algorithmen dazu in der Vorlesung noch besprechen. Hier aber bereits die MATLAB Anweisung dazu. Ausgegeben wird eine Stelle 'x', an der die Funktion Null wird.

```
x = fzero('sin',3)
input('Druecke RETURN')
```

Neben der Funktion, für die eine Nullstelle gesucht werden soll muss ein Anfangswert für die Nullstellensuche angegeben werden. Es wird üblicherweise eine Nullstelle gefunden, die nahe bei diesem Anfangswert liegt.

Siehe die MATLAB Hilfe für die Optionen, die der Funktion fzero übergeben werden können.

Eine weitere wichtige Operation auf Funktionen ist die Integration. Auch hierzu werden wir in der Vorlesung noch entsprechende Algorithmen behandeln. Die MATLAB Routine zur Integration lautet 'quad', da numerische Integration üblicherweise als 'Quadratur' bezeichnet wird.

```
x = quad('sin',0,2*pi)
input('Druecke RETURN')
```

Der Grundaufruf entspricht dem 'fminbnd' Befehl. Als viertes Argument kann zusätzlich die gewünschte Genauigkeit angegeben werden. Beachte: Hier wird numerisch integriert, d.h. man muss mit numerischen Fehlern rechnen.

2.3 Nullstellen nichtlinearer Gleichungen

Nachfolgend eine Demonstration der MATLABRoutine 'fzero' zur Nullstellenbestimmung

Informationen zum Algorithmus:
'fzero' sucht zunächst ein Intervall [a,b], für dessen Grenzen f(a) und f(b) unterschiedliche Vorzeichen besitzen. Danach startet ein ein iteratives Verfahren, wobei in jedem Schritt falls möglich in Schritt des Sekantenverfahrens (in MATLAB Ínterpolation' genannt) durchgeführt wird. Führt dieser nicht zum Erfolg, wird statt dessen ein Schritt des Bisektionsverfahrens durchgeführt.

Der Grundaufruf lautet

```
fzero('sin',4)
```

wobei das zweite Argument ein Startwert für die Iteration ist.

Statt eines einzelnen Startwertes kann auch direkt ein Intervall [a,b] angegeben werden.

```
fzero('sin',[2,4])
```

Aber Achtung: Falls ein Intervall [a,b] übergeben wird, müssen f(a) und f(b) unterschiedliche Vorzeichen besitzen: Der folgende Aufruf liefert eine Fehlermeldung:

```
fzero('sin',[1,2])
```

Zusätzlich zu der Nullstelle können weitere Informationen ausgegeben werden:

Der Funktionswert an der ermittelten Nullstelle:

```
[x,fval] =  fzero('sin',4)
```

Ein Flag, das einen positiven Wert bei erfolgreicher Suche und einen negativen Wert bei Misserfolg ausgibt. Dies ist nützlich, falls 'fzero' als Teil eines anderen Algorithmus verwendet wird und Fehler abgefangen werden sollen.

```
[x,fval,flag] =  fzero('sin',4)
[x,fval,flag] =  fzero('sin(x)+2',4)
```

Schliesslich können noch die Anzahl der Iterationen, die Anzahl der Funktionsauswertungen und der tatsächlich verwendete Algorithmus ausgegeben werden.

```
[x,fval,flag,output] =  fzero('sin',4)
```

'fzero' hat zwei einstellbare Parameter, die aber nicht direkt, sondern über eine Verbundvariable 'options' eingestellt werden, die mit der Anweisung 'optimset' gesetzt wird. Die Verwendung von 'optimset' ist unten an Beispielen illustriert.

Die zwei Parameter von 'fzero' sind:

TolX : Gewünschte Genauigkeit
- voreingestellt auf die Maschinengenauigkeit eps
- mögliche Werte: jede reelle Zahl >=eps
Display: Steuerung der Ausgabe während der Iteration
- voreingestellt auf den Wert 'final'
- mögliche Werte: 'off' : Keine Ausgabe, - 'iter' : Ausgabe jedes Schrittes, - 'final': Nur Endergebnis

Beispiele:

```
format long
```

Änderung der Genauigkeit:

```
options = optimset('TolX',1e-4);
fzero('sin',4,options)
options = optimset('TolX',1e-15);
fzero('sin',4,options)
fprintf('(Zum Vergleich: pi = %16.14f)\n\n',pi)
```

Änderung von Ausgabe und Genauigkeit

```
options = optimset('TolX',1e-4,'Display','iter');
fzero('sin',4,options)
```

Mit dieser Ausgabe kann man auch schön erkennen, dass die Angabe eines Intervalls den numerischen Aufwand deutlich reduzieren kann

```
fzero('sin',[2,4],options)
```

2.4 Polynome und Interpolation

Polynome sind eine wichtige Klasse von Funktionen. In der Vorlesung haben wir sie zur Interpolation und im Ausgleichsproblem verwendet, sie tauchen aber auch in vielen anderen Anwendungen auf.

Hier wollen wir die wichtigsten Polynom-Routinen von MATLAB besprechen.

In mathematischer Schreibweise ist ein Polynom vom Grad n gegeben durch

$$p(x) = a0 + a1x + a2x^2 + ... + anx^n$$

In MATLAB werden die Koeffizienten anders numeriert; es gilt die Konvention

$$p(x) = P(n+1) + P(n) * x + P(n-1) * x^2 + ... + P(1) * x^n$$

Ein Polynom vom Grad n wird also als n+1-dimensionaler Vektor dargestellt.

Beispiel: Das Polynom $p1(x) = 1 + 2x - 3x^2$ wird in MATLAB als

```
P1 = [-3 2 1]
```

dargestellt.

Zur Auswertung eines Polynoms dient die Routine 'polyval':

```
polyval(P1,0)
polyval(P1,1)
input('Druecke RETURN')
```

```
 Diese kann selbstverst\"andlich auch zum Plotten verwendet werden.
\begin{verbatim}
t = [-2:0.05:2];
y = polyval(P1,t);
plot(t,y,'-')
input('Druecke RETURN')
```

'polyval' rechnet immer komponentenweise, wenn der zweite Parameter ein Vektor oder eine Matrix ist. Wenn man möchte, dass die Multiplikationen in der Polynomauswertung als Matrix-Multiplikationen ausgeführt werden, muss man die Routine 'polyvalm' verwenden. Achtung: Dies ist nur für quadratische Matrizen sinnvoll.

```
A = [0 1; 2 3]
polyval(P1,A)
polyvalm(P1,A)
input('Druecke RETURN')
```

Die Routine 'roots' findet die Nullstellen eines Polynoms

```
roots(P1)
input('Druecke RETURN')
```

Dies funktioniert auch, wenn, wie z.B. beim Polynom $p2(x) = 1 + x + 2x^2$, nur komplexe Nullstellen existieren

```
P2 = [2 1 1]
roots(P2)
input('Druecke RETURN')
```

Umgekehrt kann man sich das zu einer vorgegebenen Menge von Nullstellen gehörige Polynom mit eben diesen Nullstellen ausrechnen lassen. Hierbei wird der Koeffizient P(1) immer gleich 1 gesetzt.

```
P3 = poly([1 -1])
input('Druecke RETURN')
```

Dies liefert den Vektor P3 = [1 0 -1], also das Polynom $p3(x) = -1 + x^2$.

Mit 'conv' kann man zwei Polynome miteinander multiplizieren.

```
P4 = conv(P1,P2)
input('Druecke RETURN')
```

und mit 'deconv' kann man ein Polynom durch ein anderes Teilen. Falls ein Rest bleibt, wird dieser als zweite Variable zurückgegeben. Mit dem Aufruf [Q,R] = deconv(P1,P2) erfüllen die zugehörigen Polynome q, r, p1 und p2 also für alle x die Gleichung $p1(x) = p2(x)q(x) + r(x)$

```
[Q,R] = deconv(P4,P1)
[Q,R] = deconv(P4,P3)
input('Druecke RETURN')
```

Mit 'polyder' kann man Polynome mit MATLAB ableiten.

```
P5 = polyder(P1)
```

MATLAB hat mit 'polyfit' einen eingebauten Befehl, um Interpolationspolynome zu berechnen. Als Beispiel definieren wir die Daten

```
nn = 4            % Anzahl der Stuetzstellen - 1
xx = [1   2   3    4    5  ]
ff = [2.2 5.4 9.2 16.0 26.2]
```

und berechnen

```
P6 = polyfit(xx,ff,nn)
```

Grafisch sieht man am einfachsten, dass das wirklich stimmt

```
t = [1:0.05:5];
y = polyval(P6,t);
plot(t,y,'r-',xx,ff,'bx')
input('Druecke RETURN')
```

Es ist hierbei erlaubt, den dritten Parameter in 'polyfit' kleiner als die Anzahl der Stützstellen minus 1 zu wählen. Dann wird ein Polynom des angegebenen Grades berechnet, das - im Sinne der kleinsten Quadrate - am nächsten an den vorgegebenen Datenpunkten liegt.

```
P7 = polyfit(xx,ff,2)
hold on;
y = polyval(P7,t);
plot(t,y,'g-')
hold off;
input('Druecke RETURN')
```

MATLAB hat auch Routinen zur Erzeugung und Auswertung von Funktionen, die stückweise aus Polynomen bestehen. Wir werden in der Vorlesung noch sehen, dass diese zur Lösung von Interpolationsproblemen sehr nützlich sein können.

Nehmen wir an, dass wir eine Funktion definieren wollen, die durch die folgenden Formel beschrieben wird:

$$\begin{array}{lll} f(x) = (x+10)^2/81 - 2 & \text{falls } x >= -10 & \text{und } x <= -1 \\ f(x) = (x+1) - 1 & \text{falls } x >= -1 & \text{und } x <= 1 \\ f(x) = (x-1)^3 + (x-1)^2 + 1 & \text{falls } x >= 1 & \text{und } x <= 10 \end{array}$$

Die MATLAB Konvention sieht hierbei vor, dass jede Teilfunktion ein Polynom in der Variablen (x-xi) ist, wobei xi die linke Intervallgrenze des betreffenden Definitionsabschnittes ist.

Um diese Funktion in MATLAB zu realisieren, definieren wir zunächst einen Vektor mit den 'Nahtstellen' (inklusive der Randpunkte):

```
xx = [-10 -1 1 10]
```

Dann definieren wir die drei Polynome, die die Funktion auf den entsprechenden Abschnitten definieren. Beachte: Im Allgemeinen ist es egal, ob die Polynom- Koeffizienten in Zeilen oder Spaltenvektoren gespeichert werden. Für diese Anwendung müssen es aber zwingend Zeilenvektoren sein. Ausserdem müssen alle Vektoren die gleiche Dimension haben, weswegen wir PP2 mit einer Null 'auffüllen' müssen.

```
PP1 = [1/81 0 -2]
PP2 = [0    1 -1]
PP3 = [1    1  1]
```

Mit der Anweisung 'mkpp' erzeugen wir nun die gewünschte Funktion

```
PP = mkpp(xx,[PP1; PP2; PP3])
```

Beachte: PP ist eine sogenannte 'Verbundvariable' (engl. 'struct'), in der alle nötigen Werte gespeichert sind.

Mit 'ppval' können wir diese Funktion nun an beliebigen Stellen auswerten.

```
ppval(PP,0)
ppval(PP,-5)
```

und auch plotten

```
t = [-3:0.01:3];
y = ppval(PP,t);
plot(t,y,'-')
```

2.5 Integration

In diesem Abschnitt wollen wir MATLABs Integrationsroutinen kurz besprechen.

ACHTUNG: Diese Version ist für MATLAB 6, für MATLAB 5.3 gibt es leichte Abänderungen.

Wir wollen das Integral

$$\int_a^b f(x)\,dx$$

numerisch berechnen. Als Beispielintegranden betrachten wir die Funktion

$$f(x) = 1/((x-0.3)^2+0.01) + 1/((x-0.9)^2+0.04) - 6,$$

die in MATLAB unter dam Namen 'humps' vordefiniert ist.

```
fplot('humps',[0,1])
```

Das Integral von 0 bis 1 über diese Funktion ist etwa 29.858325395

MATLAB hat zwei Newton-Cotes Formeln eingebaut, die aber unterschiedlich aufgerufen werden.

Zunächst gibt es die Trapez-Regel, die mittels 'trapz' zur Verfügung steht. Hier wird allerdings keine Funktion sondern einfach eine Liste von Datenpunkten f(i) zu äquidistanten Stützstellen übergeben. Um das richtige Integrationsergebnis zu erhalten, muss das Ergebnis durch die Anzahl der Stützstellen -1 geteilt werden

```
n = 100;
t = [0:1/n:1];
f = humps(t);
trapz(f)/n
```

Wegen der expliziten Übergabe von Daten eigent sich diese Routine insbesondere zur Integration von Funktionen, die nur als Messwerte oder Wertetabellen vorliegen.

Zur Integration von 'echten' Funktionen dienen die Routinen 'quad' und 'quadl'. 'quad' benutzt die Simpson-Regel während 'quadl' eine sogenannte '(4,7) Gauss-Lobatto-Kronrod Regel' erwendet. Diese ist bei hinreichend oft differenzierbaren Funktionen i.A. effizienter.

Der Aufruf dieser Funktionen lautet 'quad('fun',a,b,tol,trace)', mit:

fun : zu integrierende Funktion
a,b : Integrationsgrenzen
tol : gewünschte Genauigkeit (Voreinstellung 1e-6)
trace: =0: Keine Ausgabe von Zwischenergebnissen (Voreinstellung), =1: Ausgabe von Zwischenergebnissen

tol und trace können weggelassen werden, dann wird die Voreinstellung gewählt. Bei 'trace=1' werden als Zwischenergebnisse jeweils die Anzahl der Funktionsauswertungen, der Anfangspunkt und die Länge der verwendeten Teilintervalle und der Integralwert auf diesem Teilinterval ausgegeben.

```
quad('humps',0,1)
quad('humps',0,1,1e-15)
quad('humps',0,1,1e-6,1)
input('Druecke RETURN')
```

Beide Routinen verwenden eine sogenannte adaptive Stützstellenwahl, d.h. die Stützstellen werden abhängig von der vorgegebenen Genauigkeit iterativ ausgewählt und sind im Allgemeinen nicht äquidistant.

Mit dem Aufruf

```
[I,p] = quad('humps',0,1)
input('Druecke RETURN')
```

wird in der Variablen 'p' die Anzahl der Funktionsauswertungen zurückgegeben.

Leider liefert 'quad' keine explizite Liste der verwendeten Stützstellen. Mit einem Trick können wir diese trotzdem grafisch darstellen: Wir definieren eine Funktion 'myhumps'

```
function y = myhumps(x)
  y = humps(x);
  plot(x,y,'ro')
  plot(x,zeros(size(x)),'k+')
```

die bei jedem Aufruf den Wert von 'humps' zurückgibt und zugleich einen kleinen Kreis auf dem Graphen und ein '+' auf der x-Achse an der entsprechenden Stelle zeichnet.

```
hold on
quad('myhumps',0,1,1e-4)
hold off
input('Druecke RETURN')
```

MATLAB kann auch Doppelintegrale für Funktionen mit zwei Variablen berechnen, also

$$\int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x,y)\,dydx$$

Die Anweisung hierfür lautet 'dblquad('fun',a1,b1,a2,b2,tol,method)' mit

fun : Zu integrierende Funktion in x und y
a1,b1 : Integrationsgrenzen in x
a2,b2 : Integrationsgrenzen in y
tol : gewünschte Genauigkeit (Voreinstellung 1e-6)
method: @quad (Voreinstellung) oder @quadl

Als Methode kann auch der Name einer selbstgeschriebenen Integrationsroutine mit führendem '@' eingegeben werden, sofern diese den Parameterkonventionen der eingabuten Routinen entspricht.

```
dblquad('sin(x).*y',0,pi,0,1)
```

2.6 Differentialgleichungen

Wir betrachten einige Beispieldifferentialgleichungen

1. Radioaktiver Zerfall. Hier kennen wir die exakte Lösung, die wir einfach plotten können

```
t = 0:0.1:10;
plot(t,exp(-t.*0.5),'k-');
xlabel('t');
ylabel('x_1(t)');
input('Druecke RETURN')
clear
```

2. Restringiertes Drei-Körper-Problem

Hierbei müssen wir die voreingestellte Genauigkeit etwas erhöhen, um eine schöne Lösung zu erhalten

Die rechte Seite der DGL ist in 'satellitf' definiert.

```
function y = satellitf(t,x)
 y = zeros(4,1);
 mu1 = 0.012277471;
 mu2 = 1 - mu1;
 D1 = ((x(1) + mu1).^2 + x(3).^2).^(3/2);
 D2 = ((x(1) - mu2).^2 + x(3).^2).^(3/2);
 y(1) = x(2);
 y(2) = x(1) + 2.*x(4) - mu2.*(x(1) + mu1)./ D1 ...
         - mu1.*(x(1) - mu2)./ D2;
 y(3) = x(4);
 y(4) = x(3) - 2.*x(2) - mu2.*x(3) ./ D1 - mu1.*x(3)./ D2;
```

```
options = odeset('RelTol',1e-10);
[t,y] = ode45('satellitf',[0,17.066], [0.994,0,0,-2.0015851063790825],...
              options);
```

Darstellung in Abhängigkeit von t.

```
plot(t,y(:,1),'k-',t,y(:,3),'k:');
xlabel('t');
ylabel('x_1(t) (-), x_3(t) (\cdot\cdot\cdot)');
input('Druecke RETURN')
```

Darstellung als Kurve

```
figure
plot(y(:,1),y(:,3),'k');
axis([-1.5,1.5,-1.5,1.5]);
xlabel('x_1(t)');
ylabel('x_3(t)');
input('Druecke RETURN')
```

Animierte Darstellung der Kurve

```
hold off
plot(-0.0123, 0, 'ko', 1-0.0123, 0, 'k.')
axis([-1.5,1.5,-1.5,1.5]);
set(gca,'nextplot','replacechildren')
s = size(t);
j = 1;
for i=1:69
  while (t(j)<i/4)
      j=j+1;
      if (j==s(1))
          break
      end
  end
  plot(0, 0, 'ko', 1, 0, 'k.',y(1:j,1),y(1:j,3))
  F1(i) = getframe;
end
input('Druecke RETURN')
hold off
clear
```

3. Lorenz-Gleichung
Die rechte Seite der DGL ist in 'lorenzf' definiert.

```
function y = lorenzf(t,x)
 y = zeros(3,1);
 y(1) = 10.*(x(2)-x(1));
 y(2) = -x(1).*x(3)+28.*x(1)-x(2);
 y(3) = x(1).*x(2)-(8/3).*x(3);

[t,y] = ode45('lorenzf',[0 40], [0.1 0 0]);
```

Darstellung in Abhängigkeit von t

```
plot(t,y(:,1),'k-',t,y(:,2),'k:',t,y(:,3),'k--');
xlabel('t');
ylabel('x_1(t) (-), x_2(t) (\cdot\cdot\cdot), x_3(t) (--)');
input('Druecke RETURN')
clear
[t,y] = ode45('lorenzf',[0 100], [0.1 0 0]);
```

Darstellung als Kurve

```
figure
plot3(y(:,1),y(:,2),y(:,3),'k');
xlabel('x_1(t)');
ylabel('x_2(t)');
zlabel('x_3(t)');
view(-110,-28);
input('Druecke RETURN')
```

Animierte Darstellung der Kurve

```
hold off
plot3(0, 0, 0)
axis([-20 20 -30 30 0 50])
view(-110,-28)
set(gca,'nextplot','replacechildren')
s = size(t);
j = 1;
for i=1:400
  while (t(j)<i/4)
      j=j+1;
      if (j==s(1))
```

```
            break
        end
    end
    plot3(y(1:j,1),y(1:j,2),y(1:j,3),y(j:j,1),y(j:j,2),y(j:j,3),'r.')
    F2(i) = getframe;
end
hold off
```

Nachfolgend illustrieren wir MATLABs Differentialgleichungslöser.

MATLAB stellt eine ganze Familie von Lösern zur Verfügung:

ode45 ode23 ode113 ode15s ode23s ode23tb ode23t

Wir werden hier die Löser ode45, ode23, ode15s und ode23s und ihre Anwendungsgebiete genauer untersuchen.

Als Standardlöser bietet sich für allgemeine gewöhnliche Differentialgleichungen die Routine ode45 an.

```
[t,x] = ode45('dgl_f1', [0 4], 1);
plot(t,x);
```

Dies ist ein eingebettetes Runge-Kutta Verfahren nach Dormand-Prince mit Konsistenzordnung 4 und 5. Für den Aufruf muss die rechte Seite der Differentialgleichung angegeben werden sowie das Lösungs-Zeitintervall und der Anfangswert. Eine Schrittweite muss nicht angegeben werden, da diese bei allen Verfahren automatisch gesteürt wird.

Auch wenn wir uns hier mit den Lösungsroutinen beschäftigen wollen, soll kurz die grafische Darstellung von Differentialgleichungen als Richtungsfeld erwähnt werden, die in MATLAB mit 'quiver' erzeugt wird.

```
hold on
clear
[t,x] = meshgrid(0:0.2:4,-0.6:0.1:1);
z = dgl_f1(t,x);
quiver(t,x,ones(size(t)),z,'r-');
axis([0 4 -0.6 1]);
hold off
input('Druecke RETURN')
```

Den DGL-Lösern können eine Reihe von Optionen übergeben werden, die durch die Verbundvariable 'options' gesteürt werden, welche wiederum durch die Anweisung 'odeset' gesetzt wird. Einige der wichtigen Optionen sind

RelTol: Gewünschte Relative Genauigkeit Voreinstellung 1e-3 (1 Prozent)

AbsTol: Gewünschte Absolute Genauigkeit Voreinstellung 1e-6

Die Gesamtgenauigkeit wird aus diesen Werten mittels der Formel: Fehler ¡ max(RelTol*abs(x(i)),AbsTol) berechnet, wobei 'Fehler' den in jedem Schritt hinzukommenden lokalen Fehler bezeichnet. Ist die Bedingung verletzt, wird über eine Schrittweitensteuerung die Schrittweite h verkleinert (Details dazu in der Numerik-Vorlesung am 15.7.)

Refine: Feinheit der Ausgabe, steuert die Anzahl der Punkte, die ausgegeben wird. Voreinstellung: 4 für 'ode45' 1 für alle anderen Löser

Stats: Ausgabe des numerischen Aufwands eines Verfahrens 'on' oder 'off' (Voreinstellung)

Vectorized: Angabe, ob das M-File der rechten Seite der DGL Vektoreingaben korrekt verarbeiten kann 'on' oder 'off' (Voreinstellung)

MaxStep: Maximal erlaubte Schrittweite Voreinstellung: 1/10 der Rechenintervallbreite

InitialStep: Vorschlag für Anfangsschrittweite Voreinstellung: Automatische Auswahl

NormControl: 'on': Die Genauigkeit der Lösung wird über die 2-Norm gemessen 'off': Die Genauigkeit wird komponentenweise gemessen (Voreinstellung)

Im Folgenden werden wir eine Reihe von Optionen und ihre Auswirkungen illustrieren. Hier verwenden wir neben ode45 auch ode23, was ein ähnliches Verfahren mit (niedrigerer) Konsistenzordnung 2 und 3 ist.

```
clear
options = odeset('Stats', 'on', 'AbsTol', 1e-10);
fprintf('\node45, Absolute Genauigkeit 1e-10\n');
[t,x] = ode45('dgl_f1', [0 4], 1, options);
fprintf('\node23, Absolute Genauigkeit 1e-10\n');
[t,x] = ode23('dgl_f1', [0 4], 1, options);
options = odeset('Stats', 'on', 'AbsTol', 1e-4);
fprintf('\node45, Absolute Genauigkeit 1e-4\n');
[t,x] = ode45('dgl_f1', [0 4], 1, options);
fprintf('\node23, Absolute Genauigkeit 1e-4\n');
[t,x] = ode23('dgl_f1', [0 4], 1, options);
input('Druecke RETURN')
```

Bei dieser Gleichung ist ode45 offenbar klar effizienter, was an der höheren Konsistenzordnung liegt. Es kann aber sinnvoll sein, Verfahren niedrigerer Konsistenzordnung zu verwenden, wenn die rechte Seite der Differentialgleichung nicht hinreichend oft differenzierbar ist.

Der 'refine' Parameter ist insbesondere für eine schönere grafische Ausgabe nützlich

```
clear
[t,x] = ode45('dgl_f1', [0 4], 1);
plot(t,x);
input('Druecke RETURN')
figure
clear
options = odeset('Refine', 20);
[t,x] = ode45('dgl_f1', [0 4], 1, options);
plot(t,x);
input('Druecke RETURN')
```

Mit refine=1 erhält man genau die 'Rechenpunkte' (t_i, x_i), was auch für die grafische Ausgabe sinnvoll verwendet werden kann.

```
hold on
clear
options = odeset('Refine', 1);
[t,x] = ode45('dgl_f1', [0 4], 1, options);
plot(t,x,'rx');
hold off
input('Druecke RETURN')
```

Neben den 'options' können noch weitere Parameter an die DGL-Löser übergeben werden. Diese beinflussen allerdings nicht die numerische Routine, sondern werden direkt an die rechte Seite der Differentialgleichung weitergegeben. Auf diese Weise kann man Differentialgleichungen für verschiedene Parameter lösen, ohne das M-File der DGL zu ändern.

Wir veranschaulichen das an einer 3d Differentialgleichung, die als 'Rössler-System' bekannt ist.

```
clear
options = []; % Ein Trick, weil hier eine 'options'-Variable
              % uebergeben werden muss, auch wenn keine
              % Optionen geaendert werden sollen
[t,x]=ode45('dgl_f2',[0,100],[1; 1; 1], options, 0.2, 0.2, 2.5);
plot3(x(:,1),x(:,2),x(:,3))
title('c=2.5')
grid
figure
[t,x]=ode45('dgl_f2',[0,100],[1; 1; 1], options, 0.2, 0.2, 5);
plot3(x(:,1),x(:,2),x(:,3))
title('c=5')
```

```
grid
input('Druecke RETURN')
```

Die bisher betrachteten Verfahren sind sogenannte 'explizite' Einschrittverfahren, bei denen die Lösung x(i+1) direkt aus der vorhergehenden Lösung x(i) berechnet wird. Für gewisse Klassen von Differentialgleichungen, die sogenannten 'steifen' Differentialgleichungen sind diese Verfahren sehr ineffizient, auch wenn sie nicht unbedingt ein falsches Ergebnis liefern. Steife Differentialgleichungen zeichnen sich dadurch aus, dass die Konstante 'K(T)' im Konvergenzsatz sehr gross ist, was sich typischerweise durch eine grosse Lipschitz-Konstante für die rechte Seite der Differentialgleichung f ausdrückt. Geometrisch betrachtet haben wir es mit Lösungen zu tun, die sich zunächst sehr schnell verändern können, sich dann aber lange Zeit fast konstant verhalten.

Für diese Gleichungen sind die sogenannten impliziten Verfahren deutlich besser geeignet. Hierbei erhält man keine explizite Formel für x(i+1), sondern - etwas vereinfacht gesagt - ein nichtlineares Gleichungssystem der Form x(i+1) = F(x(i),x(i+1)), dass in jedem Schritt (z.B. mit dem Newton-Verfahren) gelöst werden muss.

MATLAB stellt mit ode15s und ode23s zwei solcher Verfahren zur Verfügung (ode23s ist ein Einschrittverfahren, ode15s ein sogenantes Mehrschrittverfahren, bei dem der Wert x(i+1) aus mehreren vorhergehenden Werten x(i-s), x(i-s+1),...,x(i) berechnet wird). Wir vergleichen die Ergebnisse zunächst an einer einfachen 1d DGL und dann an einer komplizierteren 3d DGL.

```
clear
options = odeset('Stats', 'on', 'Refine', 1);
tt = [0 0.5];
x0 = 1;
fprintf('\node45:\n')
[t1,x1] = ode45('dgl_f3', tt, x0, options);
plot(t1,x1,'rx',t1,x1);
title('ode45')
axis([0 0.5 -1 1])
figure

fprintf('\node15s:\n')
[t2,x2] = ode15s('dgl_f3', tt, x0, options);
plot(t2,x2,'rx',t2,x2);
title('ode15s')
axis([0 0.5 -1 1])
figure
```

```
fprintf('\node23s:\n')
[t3,x3] = ode23s('dgl_f3', tt, x0, options);
plot(t3,x3,'rx',t3,x3);
title('ode23s')
axis([0 0.5 -1 1])
input('Druecke RETURN')
clear
options = odeset('Stats', 'on', 'Refine', 1);

fprintf('\node45:\n')
[t1,x1] = ode45('dgl_f4', [0 3], [1;0;0],options);
plot(t1,x1(:,2),'rx',t1,x1(:,2));
title('ode45')
figure

fprintf('\node15s:\n')
[t2,x2] = ode15s('dgl_f4', [0 3], [1;0;0], options);
plot(t2,x2(:,2),'rx',t2,x2(:,2));
title('ode15s')
figure

fprintf('\node23s:\n')
[t3,x3] = ode23s('dgl_f4', [0 3], [1;0;0], options);
plot(t3,x3(:,2),'rx',t3,x3(:,2));
title('ode23s')
```

Man sieht im zweiten Beispiel: Die impliziten Verfahren ode15s und ode23s sind nicht nur viel effizienter, sondern liefern auch 'glattere' und damit schönere und exaktere Lösungen.

Zum Schluss noch eine kurze Beschreibung weiterer MATLAB DGL-Löser:

ode113: explizites Mehrschrittverfahren, d.h. x(i+1) wird mittels einer expliziten Formel aus meheren vorhergehenden Werten x(i-s), x(i-s+1), ..., x(i) berechnet. Konsistenzordnung p = 1 bis 13 Nicht geeignet für steife DGL.

ode23tb: Implizites eingebettetes Runge-Kutta-Verfahren Konsistenzordnung p = 2 und 3 Geeignet für steife DGL.

ode23t: Auf der Trapezregel basierendes implizites Verfahren. Konsistenzordnung p = 2 und 3 Geeignet für moderat steife DGL.

Eindimensionale zeitabhängige Differentialgleichung als Beispiel für MATLABs DGL-Löser

(Beispiel ohne praktische Bedeutung)

```
function y = dgl_f1(t,x)
y = -x - 5.*exp(-t).*sin(5.*t);
```

Dreidimensionale Differentialgleichung ('Rössler-System') als Beispiel für MATLABs DGL-Löser

Enthält weitere Parameter, die vor der Lösung spezifiziert werden müssen

```
function y = dgl_f2(t,x,flag,a,b,c)
y = [-x(2)-x(3); x(1)+a*x(2); b+x(3)*(x(1)-c)];
```

Eindimensionale zeitabhängige Differentialgleichung als Beispiel für MATLABs DGL-Löser

(Beispiel für eine sehr einfache 'steife' Differentialgleichung)

```
function y = dgl_f3(t,x)
y = -1000.*x;
```

Dreidimensionale Differentialgleichung als Beispiel für MATLABs DGL-Löser

Die Gleichung modelliert eine chemische Reaktion ('Robertsons Modell') und ist ein Beispiel für eine sogenannte 'steife' DGL

```
function y = dgl_f4(t,x)
y = [-0.04*x(1) + 1e4*x(2)*x(3); ...
      0.04*x(1) - 1e4*x(2)*x(3) - 3e7*x(2)^2; ...
      3e7*x(2)^2 ];
```

Kapitel 3

Sonstiges

3.1 Effizienzsteigerung in Matlab

3.1.1 Grundlagen

In diesem Abschnitt wollen wir eine Reihe von Tricks besprechen, mit denen man Operationen schneller ausführen kann

Zuerst stellen wir die Standard-Ausgabe passend ein.

```
format long
format compact
```

Um dis Auswirkungen verschiedener Programmiertechniken zu sehen, sollte man die Zeit messen, mit der eine Anweisung oder Anweisungsfolge ausgeführt wird. Dazu gibt es die MATLAB Befehle 'tic' und 'toc'

```
fprintf('\n\ntic und toc:\n\n')
tic
for i=1:10000
  x = i.^2;
end;
toc
for i=1:10000
  x = i.^2;
end;
z=toc
input('Druecke RETURN')
```

Mit der Eingabe von 'tic' beginnt die Zeitmessung. Bei jeder folgenden Eingabe von 'toc' wird die seit dem letzten 'tic' vergangene Zeit ausgegeben oder kann

einer Variablen zugewiesen werden. Gemessen wird die CPU-Zeit, d.h. nur die Zeit, in der die Zentralrecheneinheit wirklich aktiv war. Beachte, dass andere parallel zu MATLAB laufende Programme diese Zeit beeinflussen können.

MATLAB ist sehr genügsam bei der Definition von Matrizen bzw. Vektoren (wir werden im Allgemeinen von 'Feldern' sprechen), in dem Sinne, dass Felder komponentenweise belegt werden können, ohne dass man vorher die Grösse festlegen muss:

```
fprintf('\n\nFelder mit und ohne vorheriger Definition:\n\n')
clear;
tic
for i=1:10000
  z(i) = i.^2;
end
toc
```

Dies ermöglicht zwar sehr flexible Programmierung, führt aber zu sehr langsamer Ausführung der Zuweisungen, da intern bei jeder Zuweisung erst der nötige Speicherplatz reserviert werden muss.

Viel schneller ist die folgende Variante, bei der zuerst ein Vektor der benötigten Grösse definiert wird.

```
tic
y = zeros(1,10000);
for i=1:10000
  y(i) = i.^2;
end
toc
input('Druecke RETURN')
```

Dieses Verfahren hat nicht nur bei der Zuweisung Vorteile sondern auch beim Auslesen der Werte (wenngleich der Unterschied hier nicht so drastisch ist und stark von dem verwendeten Rechner abhängt).

```
fprintf('\n\nAuslesen von Feldern:\n\n')
tic
for i=1:10000
  x = z(i);
end
toc
tic
for i=1:10000
```

```
  x = y(i);
end
toc
input('Druecke RETURN')
```

Darüberhinaus belastet die Zuweisung von Feldern ohne vorherige Definition auch die Gesamtleistungsfähigkeit, da im Voraus definierte Felder als Block im Speicher abgelegt werden können und so eine viel effizientere Speicherverwaltung ermöglichen.

Zur Definition eines Feldes, das später mit Werten belegt werden soll, empfiehlt sich der 'zeros' Befehl, siehe oben. Hier wird jeder Eintrag mit Null vorbelegt. Aber auch wenn man Felder mit anderen Werten als 0 vorbelegen will, gibt es geeignete MATLAB Routinen. Der Befehl 'ones' funktioniert völlig analog zu 'zeroes' und erzeugt Einser als Vorbelegung. ACHTUNG: 'zeros(N)' bzw. 'ones(N)' erzeugen keinen N-dimensionalen Vektor sondern eine N x N Matrix!

'ones' kann im Prinzip auch benutzt werden, um beliebige (gleiche) Einträge zu erzeugen, z.B. eine 100 x 100 Matrix mit lauter Zweiern.

```
fprintf('\n\nZuweisung von Matrizen:\n\n')
clear;
tic
A = zeros(1000);
A = 2.*ones(1000);
toc
```

Für sehr grosse Felder (nicht für kleine) geht dies schneller mit dem 'repmat' Befehl, der eine Matrix mit dem Eintrag aus dem ersten Argument erzeugt:

```
clear;
tic
B = zeros(1000);
B = repmat(2,1000);
toc
```

Erklärung: Im ersten Fall wird 1000 x 1000 = 1 000 000 mal eine Multiplikation durchgeführt; im zweiten Fall nur zugewiesen. Noch schneller als 'repmat' ist allerdings die folgenden Zuweisung, bei der mit A(:) jedes Element der Matrix ausgewählt wird.

```
clear;
tic
C = zeros(1000);
C(:) = 2;
toc
```

'repmat' ist deswegen langsamer, weil es nicht explizit ausnutzt, dass die '2' hier ein skalarer Wert ist. Tatsächlich kann das erste Argument von 'repmat' selbst ein Feld sein, dass dann mittels 'repmat' entsprechend oft aneinandergefügt wird.

MATLAB ist eine Vektor- bzw. Matrix-orientierte Programmierumgebung mit vielen eingebauten Routinen, die direkt auf Vektoren bzw. Matrizen arbeiten. Diese Routinen sind in MATLAB als hochoptimierte C-Routinen implementiert und in fast jedem Fall deutlich schneller als die Verwendung von 'for' Schleifen.

Die Zuweisung

```
fprintf('\n\nSchleife vs. Vektoroperation:\n\n')
clear;
tic
y = zeros(1,10000);
for i=1:10000
  y(i) = i.^2;
end
toc
```

kann man z.B. auch so programmieren:

```
clear;
tic
y = zeros(1,10000);
t = [1:10000];
y = t.^2;
toc
```

Die gemessenen Zeiten sprechen für sich...

Neben vektorieller Zuweisung sind auch Summierung, Produktbildung und Maximierung bzw. Minimierung von Feldern möglich.

Einige Beispiele:

Berechnung der Summe der Quadrate der Elemente eines Vektors

```
fprintf('\n\nSummenberechnung:\n\n')
clear;
v = [0:0.1:100];
tic
x = 0;
for i=1:1000
  x = x + v(i).^2;
end
toc
```

```
tic
y = sum(v.^2);
toc
x
y
input('Druecke RETURN')
```

Berechnung der Fakultät

```
fprintf('\n\nBerechnung der Fakult\"at 100!:\n\n')
clear
tic
p = 1;
for i=1:100
  p = p*i;
end
toc
tic
q = prod(1:100);
toc
p
q
input('Druecke RETURN')
```

Zum Abschluss ein Beispiel aus dem Buch 'MATLAB Guide' von D. & N. Higham (http://www.ma.man.ac.uk/ higham/mg/), in dem ein Bifurkationsdiagramm einer Differenzengleichung berechnet wird. Die zugehörigen M-Files 'bif1.m' (Implementierung mit Schleifen) und 'bif2.m' (Implementierung mit Vektor- und Matrix-Funktionen wo immer möglich) sind im WWW unter http://www.ma.man.ac.uk/ higham/mg/mfiles.html (Chapter 20) abrufbar.

```
fprintf('\n\nBifurkationsdiagramm (bitte etwas Geduld!):\n\n')
tic
bif1
toc
figure
tic
bif2
toc
```

3.1.2 Profiler

In diesem Abschnitt veranschaulichen wir einige Möglichkeiten des MATLAB Profilers, mit dem man detaillierte Auskunft über die Ausführungsgeschwindigkeit von Programmen erhalten kann

Die einfachste Version des Profilers liefert bereits eine Menge von Informationen. Wir veranschaulichen dies anhand der numerischen Integration.

Einschalten des Profilers

```
profile on
```

Ausführen des M-Files, das untersucht werden soll

```
blatt7(1000);
```

Ausgabe des Profiler-Reports

```
profile report
input('Druecke RETURN')
```

Der Report wird jetzt in einem Web-Browser angezeigt

Grafische Darstellung einiger Profiler-Daten

```
profile plot
```

Die vielleicht etwas überraschende Erkenntnis hier: Alle drei Integrationsformeln benötigen fast die gleiche Zeit; obwohl die Formeln für Milne und Simpson komplizierter sind. Der mit Abstand zeitaufwändigste Teil ist die Auswertung der Funktion f(x)=sin(x), die sich in 'inline/subsref' 'versteckt'.

Abschalten des Profilers

```
profile off
```

Löschen der bisher gesammelten Profiler-Daten

```
profile clear
input('Druecke RETURN')
```

Mit der '-detail'-Optionen kann man steuern, wie viele Informationen ausgegeben werden.

Mögliche Optionen:

'mmex' : Nur Funktionen in M- oder MEX-Files werden ausgegeben (Voreinstellung). Beachte: Viele MATLAB Routinen liegen MATLAB-intern als M-Files vor, und werden mit dieser Option auch ausgegeben
'builtin' : Auch fest einprogrammierte MATLAB Routinen werden ausgegeben
'operator': Auch Operatorauswertungen (+,*,=,==...) werden ausgegeben

Diese Optionen beeinflussen nur den Report, nicht die Grafik Beispiele:

```
profile on -detail builtin
blatt7(1000);
profile report
profile off
profile clear
input('Druecke RETURN')

profile on -detail operator
blatt7(1000);
profile report
profile off
profile clear
input('Druecke RETURN')
```

Die Daten, die Profile sammelt, können auch direkt ausgelesen werden.

```
profile on
blatt7(1000);
stats = profile('info')
```

Die so erhaltene Variable 'stats' ist eine sogenannte 'Verbundvariable' oder 'Struct', in der viele Variablen zusammengefasst sind. Die einzelnen Komponenten können über stats.Komponentenname angesprochen werden. Wir betrachten nun den Eintrag 'FunctionTable', in dem die Informationen über die einzelnen Funktionen gespeichert sind.

```
stats.FunctionTable
```

FunctionTable ist ein Feld von Structs, dessen Einträge wie üblich als Komponenten angesprochen werden können

```
stats.FunctionTable(1)
stats.FunctionTable(2)
stats.FunctionTable(3)
stats.FunctionTable(4)
input('Druecke RETURN')
```

Betrachten wir nun den Eintrag 2 (der zur Funktion trapez.m gehört) genauer. Die Hauptinformation steckt hier in der Komponente 'ExecutedLines', in der aufgelistet ist, wie viel Zeit pro Programmzeile verwendet wurde.

```
stats.FunctionTable(2).ExecutedLines
```

Die erste Zahl ist die Nummer der Zeile, die zweite gibt an, wie oft diese Zeile abgearbeitet wurde und die dritte, wieviel Zeit dafür (insgesamt) benötigt wurde.

Das Ganze lässt sich natürlich auch grafisch ausgeben

```
x = stats.FunctionTable(2).ExecutedLines;
plot(x(:,1),x(:,3),'--ks','MarkerSize',20,'MarkerFaceColor','k')
input('Druecke RETURN')

profile off
profile clear
```

Der Profiler kann manchmal überraschende Informationen liefern. Als Beispiel betrachten wir Ausgleichsprobleme. Fazit: Durch Betrachten der 'Function Details' sieht man, dass das Aufstellen der Normalengleichungen die meiste Zeit in Anspruch nimmt, und nicht (wie man vielleicht erwarten würde) die Lösung des linearen Gleichungssystems mit dem Choleski- Verfahren.

Diese Informationen liefern wichtige Hinweise darauf, wo man bei einer Optimierung von M-Files ansetzen sollte.

```
profile off
profile clear
```