Matlab Tutorial

David Engster, 3/04

1 Was ist Matlab?

Matlab ist im Grunde genommen zwei Sachen:

- Zum Ersten ist es eine sehr einfach zu erlernende Skript-Sprache, die ganz auf die Verarbeitung von Matrizen ausgelegt ist.
- Zum Zweiten ist es eine riesige Sammlung an numerischen und grafischen Funktionen, die entweder in der eigenen Matlab-Skript-Sprache oder in einer "tieferen" Sprache wie C oder FORTRAN geschrieben sind.

Desweiteren:

- *Matlab* steht für "Matrix Laboratory".
- Läuft unter: Windows, Linux, MacOS X, Solaris.
- Es ist eine *numerische* Software und daher nicht zu vergleichen mit *algebraischer/symbolischer* Software wie Mathematica oder Maple.
- Ursprünglich in FORTRAN geschrieben. Grafische Entwicklungsumgebung in Java (Swing), daher auf allen Plattformen gleich (langsam...) zu bedienen.
- Verwendet seit Version 6.0 als Kernkomponente die extrem optimierte ATLAS-Library (bestehend aus BLAS/LAPACK, siehe http://math-atlas.sf.net). Grundlegende Funktionen der linearen Algebra werden daher extrem schnell ausgeführt.
- Interpretierte Skript-Sprache. Vorteil: Komfort, einfaches Debugging, Absturzsicherheit und dadurch verkürzte Entwicklungszeit. Nachteil: langsamer als kompilierter Code, insb. bei Schleifen.
- Matlab-Code kann trotzdem sehr schnell sein, solange einige Tipps beachtet werden (siehe letztes Kapitel). Zudem können zeitkritische Funktionen auch in C,C++ oder Fortran geschrieben und in Matlab-Code eingebunden werden (MEX-Files).
- Es existieren zahlreiche (kostenpflichtige) Toolboxen für verschiedenste Aufgaben (Bildverarbeitung, Signalanalyse, DSP, etc.). Es gibt aber auch zahlreiche Pakete umsonst im Internet (z.B. im Matlab File Exchange, TSTOOL).
- Mächtige Library für C,C++ und Fortran zur Erstellung von eigenen Anwendungen, die unabhängig von Matlab laufen.
- WWW: http://www.mathworks.com , Newsgroup: comp.soft-sys.matlab

Kosten für die Studentenversion: 87Euro (enthält Win/Linux/MacOS-Versionen, voll funktionsfähig, enthält Symbolic Toolbox und ein etwas abgespecktes Simulink).

Kostenlose Alternativen:

- Octave (www.octave.org). Bemüht sich um Kompatibilität zu Matlab, unterstützt jedoch manche Features nicht (deshalb läuft z.B. TSTOOL nicht unter Octave). Benutzt GNUPlot für Grafik.
- Scilab (www.scilab.org). Verfolgt ein ähnliches Konzept wie Matlab inkl. Simulink, ist aber nicht kompatibel. Es existiert ein Konverter, der zumindest einfache M-Files in Scilab lauffähig machen kann.
- R (www.r-project.org), eigentlich spezialisiert auf Statistik aber auch sehr gut für allgemeine Daten- und Zeitreihenanalyse zu gebrauchen. Gänzlich unkompatibel zu Matlab.

2 Hilfe

Wichtigster Befehl in Matlab ist `help`. Mit `help <FUNKTIONSNAME>` bekommt man eine Hilfe zu jeder Matlab-Funktion angezeigt:

```
>> help eye

 EYE Identity matrix.
    EYE(N) is the N-by-N identity matrix.

    EYE(M,N) or EYE([M,N]) is an M-by-N matrix with 1's on
    the diagonal and zeros elsewhere.

    EYE(SIZE(A)) is the same size as A.

    See also ONES, ZEROS, RAND, RANDN.
```

Falls man für ein Problem eine passende Matlab-Funktion sucht, kann man mit `lookfor` in den Kurzbeschreibungen der Befehle nach Stichwörtern suchen:

```
>> lookfor eigenvalues
CONDEIG Condition number with respect to eigenvalues.
EIG    Eigenvalues and eigenvectors.
QZ     QZ factorization for generalized eigenvalues.
[... und noch einige mehr ...]
```

Die Dokumentation von Matlab befindet sich in Form von zahlreichen pdf- und HTML-Dateien in `/dpi/matlab/6.5/help`. Die eigentliche Dokumentation zur Sprache und zur Oberfläche findet sich in *Using Matlab*, welche mit

```
picasso:~$ acroread5 /dpi/matlab/6.5/help/pdf_doc/matlab/using_ml.pdf &
```

gelesen werden kann.

3 Datentypen und Container

Aufgemerkt: **In Matlab ist fast alles eine Matrix mit double-Zahlen als Komponenten!**

So wird mit

```
>> a=5

a =

     5
```

eine 1x1-Matrix erstellt mit der 5 als Inhalt. Mit dem Befehl `whos` lässt sich eine Liste der aktuell im sog. `Workspace` vorhandenen Variablen anzeigen:

```
>> whos
  Name      Size                    Bytes  Class

  a         1x1                         8  double array

Grand total is 1 element using 8 bytes
```

Matlab speichert üblicherweise alle Zahlenwerte als double-Werte, auch wenn eigentlich nur ein Integer angegeben wurde - deshalb auch die 8 Byte Speicherplatz. Das mag verschwenderisch erscheinen, aber Integer-Zahlen spielen bei numerischer Software eher eine Außenseiterrolle. Die gerade definierte 1x1-Matrix lässt sich nun auch beliebig erweitern:

```
>> a(1,2)=1

a =

     5     1

>> a(2,1)=pi

a =

    5.0000    1.0000
    3.1416         0
```

Hier sieht man nun gleich mehrere Dinge: In Matlab werden Indizes in normalen Klammern angegeben in der Form `(Zeile,Spalte)`. Matlab erlaubt nur reguläre $n \times m$ Matrizen und fügt deshalb bei der letzten Erweiterung automatisch eine '0' ein. Konstanten werden in Matlab immer klein geschrieben (`pi` oder auch `i` für $\sqrt{-1}$). Weiterhin ist nun auch sichtbar geworden, dass Matlab wirklich immer mit double-Werten rechnet, nur ist Matlab so schlau, erst dann auf die Fließkomma-Darstellung umzuschalten, wenn es nötig wird. Es werden vier Nachkommastellen angegeben, aber die interne Rechengenauigkeit ist natürlich viel höher (ca. 10^{-16}, siehe `help eps`). Möchte man eine genauere Darstellung haben, kann man z.B. `format long` angeben, der Standard ist `format short` (siehe `help format` für eine Auflistung aller Anzeige-Modi).

Komfortabler gibt man eine Matrix so ein:

```
>> b=[5 1 ; pi 0]

b =

    5.0000    1.0000
    3.1416         0
```

d.h. in **eckigen Klammern** mit einem **Semikolon** als Zeilentrenner.

Auch **Strings** sind Matrizen. So sind

```
>> string='hallo'

string =

hallo
```

und

```
>> string=['h' 'a' 'l' 'l' 'o']

string =

hallo
```

in ihrer Funktion identisch. Strings werden in einfache Hochkommata eingeschlossen. Es gibt noch einige andere Datentypen in Matlab, die hier nur kurz erwähnt werden sollen:

Komplex:

```
>> a=5+2i

a =

   5.0000 + 2.0000i

>> real(a)

ans =

     5

>> imag(a)

ans =

     2
```

Strukturen:

```
>> a.name='schubi';
>> a.zahl=23;
>> a

a =

    name: 'schubi'
    zahl: 23
```

Strukturen können insb. bei der Übergabe vieler Parameter an eine Funktion sehr praktisch sein und viel Schreibarbeit sparen. Das **Semikolon** am Ende der ersten beiden Anweisungen verhindert die sofortige Ausgabe des Ergebnisses. Die Angabe von `a` alleine zeigt den Inhalt der Variablen an.

Cell Arrays:

```
>> a={3.14 'schubi' [1 2 ; 3 4] }

a =

    [3.1400]    'schubi'    [2x2 double]
```

```
>> a{3}

ans =

     1     2
     3     4
```

Cell Arrays sind Container, die verschiedene Datentypen aufnehmen können. Sie werden mit geschweiften Klammern indexiert. Man braucht sie allerdings eher selten (d.h. einfach erstmal vergessen...).

4 Umgang mit Matrizen

Zum Indexieren von Matrizen leistet der sog. Colon-Operator gute Dienste (siehe `help colon`). Mit dem Doppelpunkt lassen sich auf einfache Weise Zahlenfolgen in der Form `Anfang:Schrittweite:Ende` erstellen:

```
>> 1:2:10

ans =

     1     3     5     7     9
```

Gegeben sei nun eine Matrix:

```
>> M=[1:3 ; 4:6 ; 7:9]

M =

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

Die Dimension einer Matrix ermittelt man mit `size`. Die Länge eines Vektors mit `length`. Die einfachste Form der Indexierung einer Matrix ist:

```
>> M(1),M(2),M(3)

ans =

     1


ans =

     4


ans =

     7
```

Der C-Programmierer mag sich nun verwundert die Augen reiben, aber Matlab wurde ursprünglich in FORTRAN geschrieben und indexiert Arrays daher *spaltenweise*!

Aber es geht auch komfortabler. So ergibt

```
>> M(:,1)

ans =

     1
     4
     7
```

die erste Spalte, denn der Doppelpunkt für sich alleine steht für „alle Elemente“. Die erste und dritte Zeile kriegt man über

```
>> M([1 3],:)

ans =

     1     2     3
     7     8     9
```

Der Begriff `end` steht für den maximalen Index:

```
>> M([1 3],end-1:end)

ans =

     2   3

     8   9
```

Man kann auch eine Matrix zur Indexierung einer anderen Matrix verwenden:

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

s =

     1     1     2
     3     3     4

>> M(s)

ans =

     1     1     4
     7     7     2
```

Besonders praktisch ist dies, wenn man Matrix-Elemente anhand bestimmter Kriterien extrahieren will. Angenommen, man möchte alle Elemente von `M`, die kleiner als 4 sind:

```
>> C=M<4

C =

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

Diese neue Matrix `C` ist vom Typ **logical**:

```
>> whos C
  Name      Size                    Bytes  Class

  C         3x3                         9  logical array

Grand total is 9 elements using 9 bytes
```

Der Datentyp **logical** kennt nur die Werte 0 und 1 und kann nun zum Indexieren der Matrix `M` verwendet werden:

```
>> M(C)

ans =

     1
     2
     3
```

Man hätte auch direkt `M(M<4)` schreiben können.

5 Elementare Matrix-Operationen

Da in Matlab fast alles eine Matrix ist, muss man sich keine Gedanken bei den elementaren Rechen-Operationen (wie +,-,*,/) machen. So muss man keine speziellen Funktionen für eine **Matrix-Multiplikation** aufrufen:

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

A =

     1     2
     3     4


B =

     5     6
     7     8

>> A*B
```

```
ans =

    19    22
    43    50
```

Möchte man eine Operation aber **elementweise** anwenden, so muss man einen Punkt vor den Operator setzen:

```
>> A.*B

ans =

     5    12
    21    32
```

Noch ein wichtiges Beispiel:

```
>> A^2

ans =

     7    10
    15    22

>> A.^2

ans =

     1     4
     9    16
```

Die **Inverse** einer Matrix erhält man mit `inv(A)`. Der Ausdruck `A/B` entspricht `A*inv(B)` und `A\B` entspricht `inv(A)*B`. Letzteres lässt sich zum Lösen von linearen Gleichungssystemen verwenden:

```
>> b=[13;17]

b =

    13
    17

>> A\b

ans =

    -9
    11
```

Hier wurde das LGS x+2y=13,3x+4y=17 gelöst.

Die **Transponierte** einer Matrix erhält man so:

```
>> A'

ans =

     1     3
     2     4
```

6 Löschen, Laden und Speichern von Variablen

Um eine Matrix (z.B. `A`) zu löschen gibt man

```
>> clear A
```

ein. Um den gesamten Workspace (d.h. alle Variablen) zu löschen, verwendet man `clear all`.

Um den Inhalt der Matrix `A` sichern:

```
>> save dateiname A
```

Matlab speichert dann die Matrix `A` in der Datei `dateiname.mat`. Laden kann man diese Matrix wieder durch

```
>> load dateiname
```

Nun existiert die Matrix `A` wieder mit dem alten Inhalt im Workspace. Das mat-Format von Matlab ist ein Binärformat; möchte man den Inhalt einer Matrix als ASCII speichern (z.B. um sie mit anderer Software weiter zu verarbeiten), so verwendet man

```
>> save dateiname.asc A -ascii
```

Besitzt man eine ASCII-Datei mit Zahlenwerten (z.B. aus einer Messung), so verwendet man zum Einlesen in die Matrix `A`

```
>> A=load('dateiname.asc');
```

Tippt man einfach nur `save` ohne jegliche Parameter ein, so speichert Matlab den gesamten momentanen Workspace in der Datei `matlab.mat`. Dieser Workspace kann durch ein einfaches `load` jederzeit wieder geladen werden.

7 Skripte und Funktionen

Befehle lassen sich auch in einfache Text-Dateien schreiben. Diese müssen dann mit der Endung „.m“ gespeichert werden. In Matlab kann so ein Skript dann einfach durch Eintippen des Dateinamens aufgerufen werden. Hierbei ist zu beachten, dass die Endung „.m“ *nicht* mit eingetippt werden darf, da der Punkt bereits für Felder reserviert ist. Die Befehle in der Skript Datei werden ausgeführt als würden sie direkt in den Workspace eingetippt (d.h. alle Variablen sind global!).

Damit Matlab die Datei findet, muss sie aber im aktuellen Verzeichnis oder im Pfad von Matlab sein. Den aktuellen Pfad kann man sich mit `path` anschauen, hinzufügen kann man ein Verzeichnis mit `addpath <DIRNAME>`. Zum Navigieren im Verzeichnisbaum existieren die üblichen Unix-Befehle: `pwd, cd, ls, dir`.

In Matlab lassen sich auch Funktionen erstellen. Hierzu erzeugt man ebenfalls eine Datei mit der Endung „.m“, die folgenden Aufbau hat:

```
function [erg1,erg2]=tuwastolles(par1,par2,par3)

% TUWASTOLLES  Tut etwas tolles
%    TUWASTOLLES(par1,par2,par3) akzeptiert drei Parameter und liefert
%    als Ergebnis zwei tolle Ergebnisse zurück
%
%    Sie dient nur zu Demonstrationszwecken.

[... hier kommt die eigentliche Funktion ...]
```

Der Dateiname muss identisch mit dem Funktionsnamen sein, d.h. in diesem Fall `tuwastolles.m`. Diese Funktion akzeptiert drei Parameter (par1,par2,par3) und liefert zwei Ergebnisse (erg1,erg2) zurück. In Matlab selbst wird eine Funktion dann auch genau so aufgerufen, also z.B.

```
>> [a,b]=tuwastolles(1,2,3)
```

Man kann auch nur `a=tuwastolles(1,2,3)` schreiben, dann fällt der zweite Wert einfach stillschweigend weg.

Alle verwendeten Variablen im Funktionsblock sind lokal. Im Laufe der Funktion muss den Rückgabevariablen `erg1` und `erg2` ein Wert zugewiesen werden, ansonsten gibt Matlab eine Warnung aus. Es ist aber erlaubt, *weniger* Parameter an die Funktion zu übergeben, solange die Funktion diese Parameter dann auch nicht verwendet. Die Anzahl der übergebenen Parameter steht in der Variablen `nargin`.

Auch bei Skripten und Funktionen gibt Matlab nach dem Ende jeder Zeile das Ergebnis aus. Es ist deshalb wichtig, das **Semikolon** am Ende der Zeile nicht zu vergessen! Evtl. ist eine Ausgabe aber auch erwünscht (zu Debugging-Zwecken), siehe dazu auch den nächsten Abschnitt. Möchte man eine Anweisung über mehrere Zeilen fortsetzen, so müssen am Ende der Zeile jeweils drei Punkte stehen:

```
A = [ 1 3 6; 2 4 1; 12 53 25; ...
      3 3 1; 55 23 1; 434 23 1; ...
      4 23 1; 23 13 2; 1 3 5];
```

Alles nach einem Prozent-Zeichen ist für Matlab ein Kommentar. Der erste zusammenhängende Kommentarblock, der auf die `function` Deklaration folgt, hat allerdings eine besondere Funktion: er wird bei Aufruf von `help tuwastolles` angezeigt. Die erste Zeile dieses Kommentarblockes wird zudem von dem Befehl `lookfor` durchsucht, er sollte somit die wesentlichen Stichwörter zur Funktion enthalten.

Vorsicht: Matlab kennt keine reservierten Namen, d.h. niemand hindert einen daran, sowas wie `sin=5;` einzutippen. Dann passieren aber auch gerne Fehler wie diese:

```
>> sin=5;

>> sin(pi)

??? Subscript indices must either be real positive integers or logicals.
```

8 Ausgeben von Variablen

Häufig möchte man während der Implementierung eines Algorithmus dessen Ablauf verfolgen. Um das Ergebnis einer Zeile in der Funktion auszugeben, lässt man einfach das Semikolon weg. Weiterhin gibt es den Befehl `disp`:

```
>> disp('Oops...')
Oops...
```

Um auch Variablen ausgeben zu können, gibt es `fprintf`, mit dem C-Programmierer sich sehr schnell anfreunden werden:

```
>> a=1;b=pi;
>> fprintf('\nWert von a: %d \nWert von b: %f\n',a,b);

Wert von a: 1
Wert von b: 3.141593
```

Mit `fprintf` können auch Dateien geschrieben werden, siehe hierzu `help fprintf` sowie die Befehle `fopen` und `fclose` (im Prinzip funktioniert es genau so wie mit C).

9 Kontrollstrukturen

Als Kontrollstrukturen existieren die üblichen Verdächtigen:

if:

```
if(a==5)
  disp('a ist 5')
elseif(a==6)
  disp('a ist 6')
else
  disp('a ist weder 5 noch 6')
end
```

Man beachte die doppelten Gleichheitszeichen!

for:

Die for-Schleife iteriert über den Inhalt einer Matrix. So kann sehr einfach eine Funktion nacheinander mit verschiedenen Parametern aufgerufen werden:

```
parameter=[0.13 0.23 0.15 0.73 1.11 2.13]
for i=parameter
  einefunktion(i)
end
```

Für die „klassische“ for-Schleife wird der Colon-Operator verwendet:

```
for i=1:10
  einefunktion(i)
end
```

while:

```
a=0;
while(a<10)
  a=a+1
end
```

switch:

```
methode='linear';

switch methode
  case {'linear','bilinear'}
    disp('Methode ist linear')
  case 'constant'
    disp('Methode ist konstant')
  otherwise
    disp('Unbekannte Methode')
end
```

10 (Numerisches) Lösen von DGLs

In Matlab gibt es verschiedene Funktionen zum numerischen Lösen von DGLs und DGL-Systemen: `ode15s, ode23s, ode23t, ode23, ode34, ode113` etc. Es würde hier zu weit führen, die Unterschiede dieser DGL-Löser zu erklären. Ganz gut fährt man normalerweise mit `ode45`, ein Runge-Kutta-Löser 4. Ordnung für nichtsteife DGLs. Möchte man nun das Lorenz-System berechnen, so erstellt man ein File `lorenzdgl.m`

```
function dx=lorenzdgl(t,x)

% LORENZDGL Lorenz DGL

% Parameter
sigma=-10;
b=8/3;
r=28;

dx = [ sigma*(x(1)-x(2)) ;
       ... Rest zur ...  ;
       ... Übung    ...   ]
```

Dieser Funktion wird der aktuelle Zeitschritt `t` und der momentane Punkt als 3dim. Vektor `x` übergeben. Der Vektor der Ableitungen `dx` muss von der Funktion zurückgegeben werden. Der DGL-Löser wird nun folgendermaßen aufgerufen:

```
>> start=rand(3,1);
>> range=[0:0.03:100];
>> ts = ode45(@lorenzdgl,range,start)

ts =

         x: [1x1398 double]
         y: [3x1398 double]
    solver: 'ode45'
     idata: [1x1 struct]
```

Matlab gibt ein Feld `ts` zurück mit den Zeitwerten `ts.x` und der Lösung der DGL in `ts.y`. Wie an der Dimension von `ts.y` sieht stehen die einzelnen Komponenten der Lösung als Zeilenvektoren in dieser Matrix, die erste Komponenten erhält man somit durch `ts.y(1,:)`.

Die Eigenschaften des DGL-Lösers können mit `odeset` festgesetzt werden. Meist ist es sinnvoll, die Genauigkeit des DGL-Lösers etwas höher zu setzen (was dann natürlich auch die Dauer der Berechnung erhöht):

```
>> options = odeset('AbsTol',1e-9,'RelTol',1e-6);
>> ts = ode45(@lorenzdgl,range,start,options);
```

Es können zahlreiche weitere Parameter mit `odeset` gesetzt werden (z.B. Jacobi-Matrix, Schrittweitensteuerung). Für Details schaue man sich `help odeset` an.

11 Plots

Wichtigster Befehl ist `plot`. Hier ein einfaches Beispiel:

```
range=[0:0.1:2*pi];
A=sin(range);
plot(range,A,'k-');
```

Die Angabe `'k-'` bedeutet, dass die Farbe des Plots schwarz sein soll (`k`) und die Punkte mit Linien verbunden werden sollen (`-`). Alternativ würde `'b*'` blaue Sternchen plotten.

Der dargestellte Plot kann nun noch beschriftet werden:

```
title('Der Sinus','FontSize',16)
xlabel('Die X-Achse')
ylabel('Die Y-Achse')
```

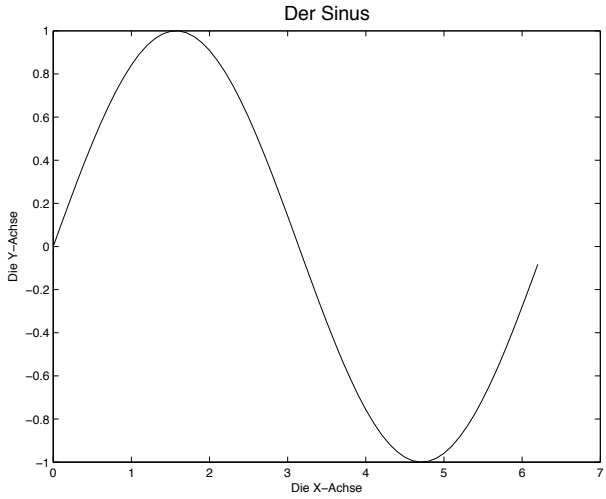

Abbildung 1: Demonstration von `plot`

Wie man sieht, kann man hier noch zusätzlich Attribute wie `FontSize` angeben. Falls man mehrere Plots offen hat, kann man den geraden aktiven Plot durch den Befehl `figure(PLOTNUMMER)` festlegen. Aufruf von `figure` ohne Plot-Nummer öffnet ein neues, leeres Plot-Fenster.

Jeder neue Plot-Befehl überschreibt das gerade aktive Plot-Fenster. Möchte man dies verhindern um mehrere Plots zu überlagern, so muss man nach dem ersten Plot-Befehl `hold on` eingeben.

Dreidimensionale Plots erstellt man mit `plot3`. Z.B. eine Spirale:

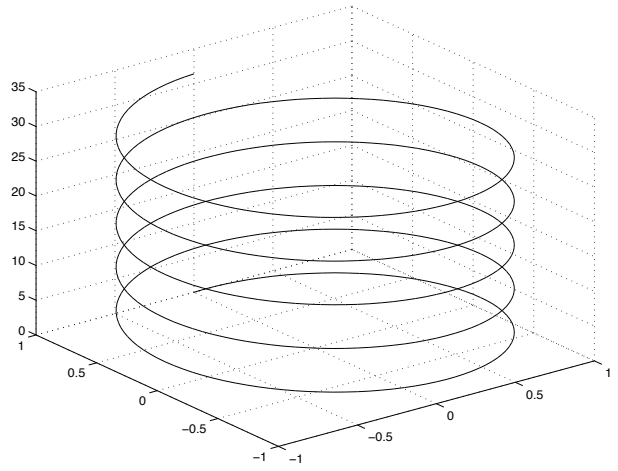

Abbildung 2: Demonstration von `plot3`

```
>> t = 0:pi/50:10*pi;
>> plot3(sin(t),cos(t),t);
>> grid
```

Das im vorigen Abschnitt integrierte Lorenz-System plottet man so:

```
>> handle=plot3(ts.y(1,:),ts.y(2,:),ts.y(3,:));
>> rotate(handle,[45 45 ],90);
>> grid
```

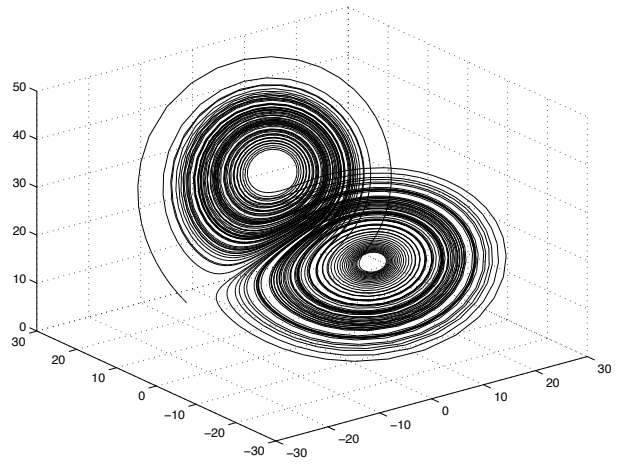

Abbildung 3: Ui... Chaos

Jeder Plot bekommt von Matlab ein sog. Handle zugewiesen, mit dem man Einzelheiten des Plots verändern kann oder das von anderen Befehlen gebraucht wird (wie hier `rotate`). Mit `get(handle)` bekommt man eine Liste von Attributen, die man mit `set(handle,'ATTRIBUT',WERT)` setzen kann.

Eine Oberfläche kann man mit `surf` erstellen. Angenommen, ich habe eine Funktion `z=ramphill([x y])`, der ich die `(x,y)` Koordinaten übergebe (in $[-1,1]^2$) und die den Funktionswert zurückgibt, so kann ich dies folgendermaßen plotten:

```
points=100;
g=2/points;
A=zeros(points+1,points+1); % Speicher reservieren
for i=0:points
  for j=0:points
    A(i+1,j+1)=ramphill([-1+i*g,-1+j*g]);
 end
end
handle=surf([-1:g:1],[-1:g:1],A)
colormap(gray)
caxis([-2 1])
rotate(handle,[0 90],90)
```

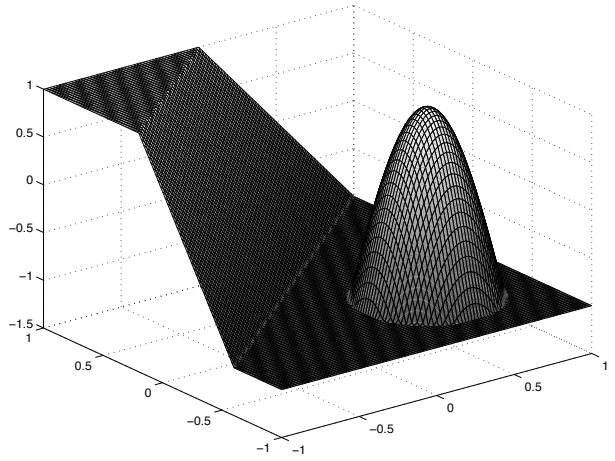

Abbildung 4: Demonstration von `surf`

Matlab färbt den Plot automatisch ein. Die Färbung kann mit den Befehlen `colormap` und `caxis` verändert werden.

Der Befehl `imagesc` stellt eine Matrix graphisch dar:

```
>> imagesc(A)
>> colormap(gray)
```

Weitere nützliche Befehle zur graphischen Darstellung: `loglog, semilogx, semilogy, subplot, pcolor, patch, line, surface, bar, stairs, errorbar, fill, legend`

12 Debugging

Das Debuggen von M-Files ist unter Matlab sehr einfach. Wenn die Funktion mit einem Fehler abbricht und man aus der Fehlermeldung nicht schlau wird, kann man mit

```
>> dbstop if error
```

dafür sorgen, dass Matlab die Funktion bei Erreichen des Fehlers anhält. Man kann dann durch Betrachten der Variablen näher untersuchen, woran der Fehler liegen könnte. Möchte man auch Warnungen von Matlab

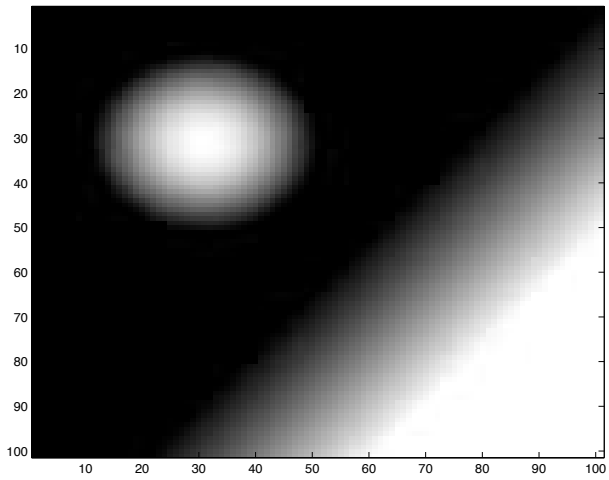

Abbildung 5: Demonstration von `imagesc`

näher auf den Grund gehen, so kann man `dbstop if warning` verwenden. Um den alten Zustand herzustellen, verwendet man `dbstop clear`.

Um einen Breakpoint in der Funktion `tuwastolles` in Zeile 79 zu setzen, tippt man

```
>> dbstop in tuwastolles.m at 79
```

Nach erreichen des Breakpoints kann man mit `dbstep` die nachfolgenden Befehle schrittweise ausführen. Selbstverständlich kann man nun auch beliebig die Werte von Variablen ändern. Mit `dbcont` wird die Ausführung des Codes fortgesetzt.

Ein großer Schwachpunkt ist, dass es leider nicht möglich ist, Breakpoints mit gewissen Bedingungen zu setzen, z.B. dass die Variable `a` den Wert 79 haben muss. Hier muss man sich damit behelfen, die Bedingung in den Code selbst einzubauen, also

```
if(a==79)
  disp('Hier halten')
end
```

und dann den Breakpoint auf die Zeile mit dem `disp` setzen.

Mit Hilfe der graphischen Matlab-Oberfläche und dem integrierten Matlab-Editor kann man sehr viel komfortabler Breakpoints einfach durch Klicken mit der Maus setzen.

13 Weitere wichtige Matlab-Funktionen

Hier eine kurze Liste der wesentlichen Funktionen von Matlab, insofern sie nicht schon behandelt wurden:

Elementare mathematische Operationen: `sin, cos, tan, sqrt, exp, log, log10, log2, abs, imag, real, round, floor, ceil, mod, rem, sign, mean, median`

Matrix-Numerik: `det` (Determinante), `eig` (Eigenwerte), `poly` (Charakteristisches Polynom), `rank` (Rang), `svd` (Singulärwertzerlegung), `qr, lu, chol` (QR-, LU- bzw. Cholesky-Zerlegung), `pinv` (Pseudoinverse)

Spezielle Funktionen: `airy, besselj, erf, gamma`

Sonstiges: `polyfit` (Fittet Polynom an Daten), `polyval` (Schnelle Berechnung von Polynomen), `filter` (Filterung)

14 Für Fortgeschrittene

In diesem Abschnitt sollen kurz einige fortgeschrittene Aspekte von Matlab vorgestellt werden. Es richtet sich an die Leute, die nach dem Praktikum vielleicht noch etwas tiefer in diese Software einsteigen wollen.

Matlab-Code beschleunigen

Besonders wichtig für die Performance von M-Code sind die sog. elementaren Built-In Funktionen, die entweder aus der LAPACK/BLAS-Library stammen oder von The Mathworks in C oder FORTRAN erstellt wurden. Diese Funktionen sind sehr schnell und sollten so viel wie möglich anstelle von eigenem M-Code verwendet werden. Ob eine Funktion built-in ist, kann man mit `which` feststellen:

```
>> which svd
svd is a built-in function.
```

Da Matlab gerade bei den Kontrollstrukturen recht viele Aspekte von C und FORTRAN übernommen hat, machen viele Leute den Fehler, in Matlab ebenso zu programmieren wie sie es aus C oder FORTRAN kennen. Dies führt jedoch meist zu sehr schlechter Performance! Um Matlab-Code schneller zu machen, sollte man sich im wesentlichen eine Regel merken: **Vermeide for-Schleifen!**

Häufig ist es möglich, eine for-Schleife durch Code zu ersetzen, der eine built-in Funktion von Matlab verwendet. Der Code muss hierzu **vektorisiert** werden. Ein einfaches Beispiel:

Langsam:

```
for i=1:10
  i^2
end
```

Schnell:

```
[1:10].^2
```

Gerade die grundlegenden mathematischen Routinen wie `sin,cos,exp` usw. akzeptieren auch Matrizen als Eingabe, es ist daher nicht nötig, die Komponenten einzeln zu durchlaufen.

Das Vermeiden von for-Schleifen ist allerdings seit Matlab 6.5 nicht mehr unbedingt nötig, denn hier kann der sog. JIT-Accelerator in Aktion treten, der Schleifen (wie auch das obige Beispiel) unter bestimmten Bedingungen automatisch kompilieren und dadurch weit schneller ausführen kann - dies gilt aber nur innerhalb von Funktionen, nicht in Skripten oder auf der Kommandozeile. Zudem können bei weitem nicht alle for-Schleifen beschleunigt werden.

In Matlab existiert eine Fülle sehr schneller Routinen, die zur Manipulation und Erstellung von Matrizen dienen und mit denen sich sehr oft for-Schleifen vermeiden lassen. Leider verführt gerade die Einfachheit der Sprache von Matlab dazu, sich solche Routinen lieber schnell mit for-Schleifen selbst zu stricken, als nach einer Lösung mit Matlab-Funktionen zu suchen. Daher hier eine kleine Auswahl von nützlichen Befehlen, die man sich in einer ruhigen Minute mal mit `help` näher anschauen kann:

`all, any, find, repmat, reshape, squeeze, shiftdim, diag, sum, cumsum, diff, prod, filter, permute, zeros, ones, eye, norm`

Übrigens: Für alle Funktionen, die keine built-in Funktionen sind, kann man sich auch den M-Code mit dem Befehl `type` anschauen. So liefert `type pinv` den Code von Matlab zum Berechnen einer Pseudo-Inversen (über SVD).

Ein anderes wichtiges Thema ist Speicherverwaltung. Matlab kümmert sich im Prinzip selbst darum, es gibt jedoch Probleme, wenn große Matrizen schrittweise aufgebaut werden, da Matlab sich dann ständig um die Reservierung (Allokation) von zusätzlichem Speicher kümmern muss, was evtl. auch eine zeitaufwändige Kopie der Matrix in einen neuen (größeren) Speicherbereich nötig macht. Dies kann man verhindern, indem man gleich zu Beginn die Matrix auf die passende endgültige Größe setzt und diese anschließend schrittweise mit Werten füllt. Am einfachsten und schnellsten geht dies mit dem Befehl `zeros`, also z.B.

```
>> A=zeros(1000,5); % erstelle 1000x5 Matrix
```

Auch sollten große Matrizen sobald sie nicht mehr gebraucht werden mit dem Befehl `clear` gelöscht werden.

Profiling

Eine alte Regel lautet, dass 90% der Rechenzeit von 10% des Codes verbraucht wird. Und wenn's mal wieder länger dauert, fragt man sich häufig, welche 10% denn nun genau verantwortlich sind. Hierfür kann man die Laufzeit des Codes von Matlab ausmessen lassen (**Profiling**). Dies ist unter Matlab sehr einfach und komfortabel: Man gibt vor Start der eigenen Funktion einfach

```
>> profile on
```

ein. Hierdurch werden alle M-Funktionen und MEX-Files (siehe nächsten Abschnitt) ausgewertet; wer auch built-in Funktionen ausmessen will, kann `-DETAIL builtin` angeben. Nachdem die Funktion durchgelaufen ist beendet man das Profiling mit `profile off`. Mit `profile viewer` kann man sich dann das Ergebnis im Detail anschauen. Mit dem Profiler lässt sich z.B. auch feststellen, ob for-Schleifen mit dem JIT-Accelerator beschleunigt werden können.

MEX-Files

Wenn man nun die Performance-relevanten Teile des Codes ausfindig gemacht hat, sollte man versuchen, diese mit den bereits erläuterten Mitteln zu beschleunigen. Doch manchmal lassen sich Algorithmen einfach nicht vektorisieren. Hier sollte man dann überlegen, ob man diese Teile nicht besser in C schreibt (auch C++ und FORTRAN ist möglich). Diese kompilierten Funktionen können dann aus Matlab heraus aufgerufen werden wie normale M-Funktionen. Diese Technik nennt sich **MEX** (= Matlab Executable). Nähere Informationen hierzu finden sich in der Matlab-Dokumentation.

Stand-alone Anwendungen

Es ist durchaus möglich, seine Anwendung gänzlich in C,C++ oder FORTRAN zu entwickeln und trotzdem auf Matlab-Funktionen zugreifen zu können. Hierzu liefert Matlab Libraries mit, die mit eigenem C,C++ oder FORTRAN-Code verlinkt werden können. Insbesondere in C++ lassen sich dank OOP mit Hilfe der Libraries sehr komfortabel Anwendungen entwickeln, wobei man Zugriff auf zahlreiche Funktionen von Matlab hat (leider nicht alle).

Abschalten der Java-Oberfläche

Wer keine Lust auf die etwas träge Java-Oberfläche hat, kann auch in der Shell oder in einem Editor wie Emacs mit Matlab arbeiten. Matlab muss hierzu mit dem Schalter `-nodesktop` aufgerufen werden (grafische Funktionen wie Plots, Profiling u.ä. funktionieren natürlich weiterhin!). Nähere Informationen zum Matlab-Mode des Emacs finden sich in der Datei `matlab.el`, die man vom Matlab-File-Exchange ziehen kann. Besonders praktisch ist in Verbindung mit der reinen Matlab-Kommandozeile unter Unix das Programm `screen`, mit welchem man Matlab auch nach einem Ausloggen weiter rechnen lassen kann.