Matlab-Einführung

Pierre Bayerl

21. Oktober 1999

Einleitung

Die Programmierung von Matlab besteht überwiegend aus der Manipulation von Matrizen und deren Visualisierung.
Es werden im folgenden drei Möglichkeiten vorgestellt:

- Scriptprogramme
  Ein solches Script besteht aus Textzeilen, die bei der Ausführung des Scriptprogramms den selben Effekt haben, als wenn man sie interaktiv eingegeben hätte.
- Scriptfunktionen
  Funktionen sind dem Scriptprogramm ähnlich, mit der Ausnahme, daß alle verwendeten Variablen lokal sind. Über Rückgabewerte kommuniziert man mit dem Aufrufer.
- MEX-Programme
  MEX-Programme sind C-Funktionen, die von Matlab aufgerufen werden können

Ausführlichere Informationen und eine Befehlsreferenz erhält man, indem man "`helpdesk`" in der Kommandozeile eingibt oder manuell mit Netscape die Datei .../matlab5/help/helpdesk.html öffnet. Einen ersten Eindruck von Matlab kann man mit "`intro`" bekommen.

in Matlab:
```
helpdesk
intro
```

1 Scriptprogramme

Scriptprogramme sind Textdateien und werden unter "Programmname.m" gespeichert. Um ein solches Programm zu starten, gibt man in der Matlab-Shell "Programmname" ein.
Matlab selbst muß im selben Verzeichnis gestartet werden, in dem das Programm gespeichert ist. (man kann auch einen Pfad setzen, siehe Onlinehilfe)
Das Bearbeiten der Programmdateien muß in einem externen Editor geschehen, z.B. nedit, xemacs, vi.

Beispiel einer Matlabsitzung (Das "&" startet ein Programm im Hintergrund)

```
~> cd ordner
~/ordner> nedit test.m &
~/ordner> matlab
```

Kommentare werden in Matlabscriptdateien mit einem "%" eingeleitet und gelten bis zum Zeilenende.

% Kommentar

1.1 Variablen

In Matlab werden Variablen nicht deklariert, sondern bei ihrer ersten Benutzung implizit definiert:

keine Variablendeklaration

```
x=5;
```

Das Beispiel zeigt eine Variable x, die den skalaren Wert 5 erhält. Nach jedem Matlabbefehl kommt ein Semikolon. Wird dieses weggelassen, erfolgt eine Ausgabe (das Ergebnis des Befehls) auf der Konsole (hier: 5).
Um z.B. den Wert der Variablen A anzuzeigen, gibt man einfach in der Kommandozeile "A" ein.

Ohne Semikolon ⇒ Ausgabe am Bildschirm

Mittels der Rechenzeichen $+, -, *, /, \hat{}$ (Potenz), den Klammern () und vielen mathematischen Funktionen $(\sin(x), \cos(x), ...)$ kann man mit skalaren Variablen rechnen.

Matlab ist in allen Fällen case-sensitiv!

Neben dem skalaren Datentyp gibt es Matrizen; diese stellen die eigentliche Stärke von Matlab dar.

1.2 Matrizen anlegen

- Gefüllte Matrizen erzeugen:

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

  Hier werden die Matrixdaten, in eckige Klammern gefaßt, *zeilenweise* eingegeben. Die Zeilen werden mit Semikolons getrennt. Innerhalb der Zeilen werden die Zahlenwerte mit Leerzeichen getrennt.

  A=[zeile1; zeile2];

- Einheitlich gefüllte Matrizen erzeugen:
  Es können mit Nullen oder Einsen gefüllte Matrizen mit dem *ones-* oder *zeros*-Befehl erzeugt werden:

  ```
  B=ones(2,4);
  C=zeros(1,3);
  ```
  $$B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix};$$
  $$C = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix};$$

  - Die Matrixdimensionen werden in Fortran-Notation angegegeben: Zuerst die Höhe (y), dann die Breite(x).

    A=ones(y,x);

  - Wird nur eine Dimension angegeben: *ones(n)*, so wird eine quadratische Matrix erzeugt.

    A=ones(n); ⇒ $n \times n$ -Matrix

- Zahlenreihen erstellen:
  Man kann $1 \times n$-Matrizen erstellen, die mit einer Zahlenreihe gefüllt sind:

```
D=1:7;
E=5:-0.5:4;
```

$$D = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix};$$
$$E = \begin{pmatrix} 5 & 4.5 & 4 \end{pmatrix};$$

Im ersten Fall gibt man einen Start- und Endwert an. Dabei wird eine Matrix erzeugt, die den Startwert selbst, und in Einserschritten folgend alle Werte bis hin zum Endwert enthält. A=start:end;
Im zweiten Fall wird zwischen dem Start- und dem Endwert noch die Schrittweite angegeben, mit der gezählt wird. A=start:step:end;

- Matrizen von Matrizen erstellen: Man hat die Möglichkeit Matrizen von Matrizen zu erstellen, d.h. Matrizen die keine Zahlenwerte, sondern wiederum Matrizen enthalten:

```
M=cell(1,3);
M{1,1}=ones(2);
```

$$M = \begin{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} \ddots \end{pmatrix} \begin{pmatrix} \ddots \end{pmatrix} \end{pmatrix};$$

Der Zugriff auf die inneren Matrizen erfolgt über die Indizes, die in geschweiften Klammern dem Variablennamen angehängt werden.

1.3 Zugriff auf Matrixelemente

Einzelne Elemente können mittels ihrer Indizes angesprochen werden. Diese werden in runden Klammern der Matrixvariable angehängt.
Die Indizes beginnen immer bei 1.
Auch hier gilt: zuerst den y-Index (Zeile), dann den x-Index (Spalte). A(y,x)=*wert*

```
B=zeros(2,4);
B(2,3)=5;
```

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{pmatrix};$$

Übersteigt man bei einer solchen Anweisung die Matrixgrenzen, so wird die Matrix soweit mit Nullen erweitert, daß sich die angegebene Zeile/Spalte innerhalb der Matrix befindet. dynamische Anpassung der Matrixgröße

```
C=zeros(2,4);
C(4,3)=5;
```

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{pmatrix};$$

Um ganze Bereiche einer Matrix anzusprechen, benutzt man dieselbe Notation, die man für Zahlenreihen benutzt: A(von:bis,von:bis)=...

```
D=zeros(4,4);
D(1:2,2:3)=ones(2,2);
```

$$D = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix};$$

Man kann auf diese Weise auch Matrixbereiche an eine andere Stelle kopieren.
Wenn man den Zahlenbereich nicht mit einer Zahl, sondern mit dem Schlüsselwort *end* beendet, dann geht der Bereich bis zur Matrixgrenze: **end** $\hat{=}$ letzte Zeile/Spalte

```
E=zeros(4,4);
E(1:end,2:3)=ones(4,2);
```

$$E = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix};$$

Bemerkung: "`1:end`" (also der gesamte Bereich) kann mit "`:`" abgekürtzt werden.
Will man z.B. die gesamte letzte Zeile einer Matrix A ansprechen, so kann man dies mit "`A(end,:)`" erreichen.

1:end $\hat{=}$:

Verwendet man nur einen Index, so können alle Matrixelemente linear durchnumeriert angesprochen werden. Die Nummerierung beginnt links oben und geht spaltenweise nach rechts unten.

$A_{n\times m}(i) = wert$
$1 \le i \le n \cdot m$

```
F=zeros(2,4);
F(3)=1;
```

$$F = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix};$$

$$\begin{pmatrix} \downarrow_1 & \downarrow_4 & \downarrow_7 \\ \downarrow_2 & \downarrow_5 & \downarrow_8 \\ \downarrow_3 & \downarrow_6 & \downarrow_9 \end{pmatrix}$$

1.4 Matrixoperationen und Funktionen (Auszug)

+	`A=B+C`	Matrixaddition
−	`A=B-C`	Matrixsubtraktion
*	`A=A*B`	Matrixmultiplikation
.*	`A=A.*B`	Multiplikation der einzelnen Elemente: $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} .* \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}$
^	`A=B^3`	Potenzieren
.^	`A=B.^2`	elementweise Potenzieren
/	`A=B/C`	"Matrixdivision" `A=B*inv(C)`
inv	`A=B*inv(C)`	Bildung der Inversen
./	`A=B./C`	Division der einzelnen Elemente
'	`A=A'`	Bildung der Transponierten
pi	`x=pi*r^2`	π
or,and,xor	`a=or(b,c)`	Logisches Oder, Und, exklusives Oder
mod	`a=mod(b,c)`	Modulo
sum	`X=sum(A)`	Summen der Spalten der Matrix $\texttt{sum}(\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}) = \begin{pmatrix} 3 & 7 \end{pmatrix}$
fft	`Y=fft(X)`	Eindimensionale diskrete Fouriertransformation des Vektors X
ifft	`X=ifft(Y)`	Inverse Fouriertransformation
fft2, ifft2	`A=fft2(B)`	Zweidimensionale FFT
conv,conv2	`A=conv2(B,F)`	Faltung (ein- und zweidimensional) Das Beispiel faltet das Bild (eine Matrix) $B_{m\times n}$ mit dem Filter (eine andere Matrix) $F_{k\times l}$. Im erzeugten Bild $A_{m+k-1\times n+l-1}$ entsteht ein (u.U. ungewünschter) dunkler Rand. Dieser kann durch Angabe eines zusätzlichen Parameters unterdrückt werden: `A=conv2(B,F,'same');` dann hat das Bild A die gleiche Größe wie B.

real	`A=real(B)`	Realteil

imag	`A=imag(B)`	Imaginärteil
		`x=2+3i;` `A=[2+3i 4-i];`
		`real(x)` `imag(A)`
		Ergibt: 2; Ergibt: [3 -1];

abs	`A=abs(B)`	$A = \|B\|$
atan2	`w=atan2(x)`	Berechnung des Winkels, der durch die komplexe Zahl x dargestellt wird. (z.B.: `atan2(1+i)`$= \frac{\pi}{4}$)
round	`A=round(B)`	Runden (zur nächsten Ganzzahl) `round(`$\begin{pmatrix} -1.9 & -0.2 & 3.4 & 5.6 \end{pmatrix}$`)` $= \begin{pmatrix} -2 & 0 & 3 & 6 \end{pmatrix}$
ceil,fix,floor	A=fix(B)	Runden in Richtung $+\infty$ (ceil), 0 (fix) und $-\infty$ (floor).
size	`A=size(B)`	Gibt einen Vector mit y- und x-Groesse der Matrix B zurück. `size(ones(3,4))`$\Rightarrow$ [3 4]

Mathematische Funktionen (wie `exp`, `sin`,...) können sowohl für skalare als auch für Matrixvariablen verwendet werden. Bei Letzeren werden sie elementweise ausgeführt:

```
A=[0 pi/2 pi 3/2*pi];
B=sin(A);
```

$A = \begin{pmatrix} 0 & \frac{\pi}{2} & \pi & \frac{3\pi}{2} \end{pmatrix};$
$B = \begin{pmatrix} 0 & 1 & 0 & -1 \end{pmatrix};$

1.5 Programmflußsteuerung

if Bedingte Ausführung eines Programmblockes:

```
if Bedingung
  ...
elseif Bedingung
  ...
else
  ...
end
```

```
if i<5
  ...
elseif i==5
  ...
else
  ...
end
```

while Bedingte wiederholte Ausführung eines Programmblockes:

```
while Bedingung
  ...
end
```

```
i=0;
while i<5
  ...
  i=i+1;
end
```

for Wiederholte Ausführung eines Programmblockes:

```
for i=Zeilenvektor
  ...
end
```

Es werden alle Elemente des Zeilenvektors mit der Indexvariable (hier i) durchlaufen.

```
for i=1:5
  ...
end
```

```
for i=[8 5 9 3]
  ...
end
```

Mit dem Befehl `break` kann man eine while- oder for-Schleife vorzeitig verlassen.

Vergleichsoperatoren:

<,>	kleiner, größer	==	gleich
<=,>=	kleinergleich, größergleich	~=	ungleich

Matrizen werden elementweise bearbeitet: Das Ergebnis ist wiederum eine Matrix mit Nullen (falsch) und Einsen (wahr), die das Resultat der einzelnen Vergleiche darstellen.

```
if or(a<2,a>4)
  ...
end
```

Logische Verknüpfungen (and, or) werden mit Hilfe der entsprechenden Funktionen realisiert. (siehe 1.4)

1.6 Scriptfunktionen

Umfangreichere Aufgaben sollten in Unterprogramme aufgeteilt werden: Ein Unterprogramm steht immer in einer eigenen Datei ("Unterprogrammname.m") und kann wie die vordefinierten Funktionen verwendet werden:

```
...
Rückgabe = Unterprogrammname(Parameter);
...
```

Der Kopf der Unterprogrammdatei muß immer folgenden Aufbau haben:

```
function Rückgabe = Unterprogrammname(Parameter);
  ...
```

Im Unterprogramm sind alle Variabeln lokal.
Folgende Beispiele zeigen den Aufbau einer Unterprogrammsdatei "test.m":

Ein Rückgabewert, ein Parameter

```
function A=test(n)
% es wird eine mit Einsen gefüllte nxn-Matrix(A) zurückgegeben
  A=ones(n,n);
```

Aufruf:
X=test(3);

Ein Rückgabewert, zwei Parameter

```
function A=test(m,n)
% es wird eine mit Einsen gefüllte mxn-Matrix(A) zurückgegeben
  A=ones(m,n);
```

Aufruf:
X=test(3,2);

Zwei Rückgabewerte, ein Parameter

```
function [A,B]=test(n)
% es wird eine mit Einsen gefüllte nxn-Matrix(A) und eine mit Nullen
% gefüllte nxn-Matrix(B) zurückgegeben
  A=ones(n,n);
  B=zeros(n,n);
```

Aufruf:
[X,Y]=test(3);

Weiterhin kann man innerhalb einer Unterprogrammsdatei weitere Unterprogramme definieren. Diese sind dann aber privat, also nur innerhalb der Datei zugänglich und nach außen unsichtbar.

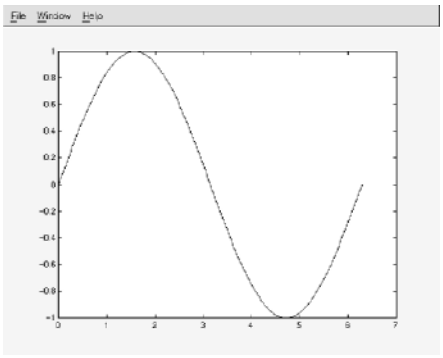
```
function A=test(n)
  A=inner(n);

function A=inner(n)
  A=ones(n,n);
```

1.7 Visualisierung

Matlab kann eine Vektormatrix (z.B. y) in einem zweidimensionalen Koordinatensystem darstellen. Dazu wird jedoch eine zweite Vektormatrix (z.B. t) benötigt, die für jedes Element aus y einen Punkt auf der x- oder Zeitachse beschreibt.

x=[...*x-werte*...];
y=[...*y-werte*...];
plot(x,y)

```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
```

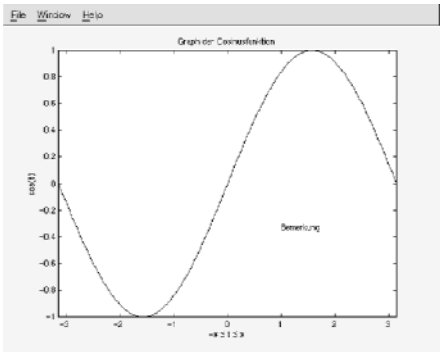


Im Beispiel beschreibt t Punkte auf der Zeitachse im Abstand von $\frac{\pi}{100}$ von 0 bis 2π. Die Matrix y enthält für jeden Wert in t den Zugehörigen Sinuswert.
Der Befehl `plot(t,y)` zeigt einen zusammenhängenden Linienzug, der einer Sinuskurve nahe kommt.
In diesem Beispiel beträgt der Abstand der t-Werte $\frac{\pi}{100}$. Dieser bestimmt die Anzahl der Punkte und damit die Genauigkeit der Darstellung.

kein Semikolon nach plot, da das Ergebnis des Befehls "angezeigt" werden soll.

Folgendes Beispiel demonstriert, wie man die Achsenbeschriftung (xlabel, ylabel) und den Titel (title) eingeben, den sichtbaren Bereich (axis) steuern und Text in die Grafik setzen kann.

```
t = -pi:pi/100:pi;
y = sin(t);
plot(t,y)
axis([-pi pi -1 1])
xlabel('-\pi \leq t \leq \pi')
ylabel('cos(t)')
title('Graph der Cosinusfunktion')
text(1,-1/3,'Bemerkung')
```



Die Textangaben sind LaTeX-ähnlich. Beim Befehl axis wird ein Zeilenvektor übergeben, der zuerst das x- und dann das y-Achsenintervall enthält.

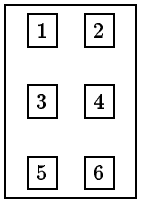
mehrere Ausgabefenster: figure

Mit der ersten Grafikausgabe wird automatisch ein Ausgabefenster geöffnet. Bei erneuten Ausgaben wird dieses wiederverwendet. Mit dem Befehl "`figure`" kann man ein neues Ausgabefenster erstellen.

Will man mehrere Darstellungen zusammen anzeigen, muß das Darstellungsfenster in *subplots* aufgeteilt werden: Folgendes Beispiel zeigt, wie das Darstellungsfenster in eine 3×2-Matrix aufgeteilt wird und in jeden Bereich ein *plot* gezeichnet wird.

```
t = 0:pi/100:2*pi;

y1 = sin(t);
y2 = t^2;
...
y6 = cos(t);

subplot(3,2,1); plot(t,y1)
subplot(3,2,2); plot(t,y2)
...
subplot(3,2,6); plot(t,y6)
```

mehrere Ausgaben in ein Fenster: subplot(y,x,index);



`subplot` selektiert die Region im Darstellungsfenster in die gezeichnet werden soll. Die ersten beiden Parameter bestimmen die Aufteilung des Fensters in y- und x-Richtung. Der dritte Parameter bestimmt die Region. Die Regionen sind von 1 (links oben) an aufsteigend, *zeilenweise* durchnumeriert.

1.8 Bilddateien laden und anzeigen

Dieser Reader beschränkt sich auf das Einlesen und Bearbeiten von nichtkomprimierten TIF-Bildern mit 256 Graustufen. (Matlab beherrscht noch andere Formate und Farbemodelle, siehe Online-Dokumentation: imread)
Mit dem Befehl `imread` wird eine Bilddatei in eine Matrix, der Größe des Bildes, eingelesen. Das Zahlenformat der Matrix ist `uint8` mit Werten zwischen 0 (schwarz) und 255 (weiß). Um mit der Matrix zu arbeiten, ist es notwendig, diese in ein Fließkommazahlenformat mit Werten zwischen 0 und 1 umzuwandeln.

```
B=imread('bild.tif');
B=(1/255) * double(B);
```

Bildmatrix anzeigen: imshow(Bildmatrix)

Angezeigt wird das Bild (die Matrix B) mit dem Befehl imshow:

```
imshow(B)
```

Auch hier kann mit dem `subplot`-Befehl das Ausgabefenster aufgeteilt werden:

```
subplot(2,1,1); imshow(Bild1)
subplot(2,1,2); imshow(Bild2)
```

Mit Hilfe eines Bildbearbeitungsprogramms, eines Bildbetrachters oder eines Konverters kann man beliebige Bildfomate ins unkomprimierte TIF-Format mit 256 Graustufen bringen. (z.B.: xv, gimp)

1.9 Bemerkungen

- Variableninhalte können auch komplex sein: `x=3+2i;`
- VORSICHT: Funktionen und Konstanten sind nicht geschützt und können vom Programmierer überschrieben werden. (mit dem Befehl "`clear`" kann die Ursprüngliche Belegung wiederhergestellt werden).

```
pi=4.5;
...
clear pi;
```

- Viele Matlabbefehle sind in Matlab selbst implementiert. Man kann sich mit "`type funktionsname`" deren Quelltext ansehen.
- `whos` zeigt alle Variablen und deren Dimension an.

2 MEX-Programme (Funktionen in C)

MEX-Programme verhalten sich nach außen wie Matlabfunktionen, sind jedoch in C implementiert. MEX-Programme sind weniger übersichtlich als Matlabfunktionen, bieten jedoch einen gewissen Geschwindigkeitsvorteil. Es bietet sich daher an, kleine, zeitkritische Programmteile in C zu schreiben und den Rest in der Matlab-Scriptsprache. (z.B. sind umfangreiche, verschachtelte Schleifen, in denen etwas berechnet wird, in Matlab etwas zeitkritisch...)

2.1 Struktur eines MEX-files

MEX-Programme: funktionsname.c

MEX-Programme kommen in eine .c-Datei mit dem Namen der Funktion (z.B. "test.c" für die Funktion "test"). In der .c-Datei muß wie im folgenden Codeabschnitt eine c-Funktion "mexFunction" definiert sein. Dieser Funktion werden vier Parameter übergeben:

Programmparameter

1. int nl: Anzahl der Rückgabevariablen (linke Seite).
2. mxArray* ml[]: ein Array von Pointern auf die Rückgabevariablen (mxArray). Die Rückgabevariablen sind "undefiniert" und müssen noch erzeugt werden.
3. int nr: Anzahl der Funktionsparameter (rechte Seite).
4. mxArray* mr[]: ein Array von Pointern auf die Parametervariablen (mxArray).

Als erstes sollte dann die Anzahl der Parameter der linken und rechten Seite überprüft werden. Stimmt die Anzahl der Parameter, so wird die eigentliche Funktion ausgeführt. Ansonsten wird mittels `mexErrMsg(...)` eine Fehlermeldung ausgegeben.

Übersetzen eines MEX-Programms: mex name.c

Übersetzt wird ein solches Programm (z.B.: test.c) mit dem mex-Compiler: (außerhalb von Matlab)

```
mex test.c
```

Aufgerufen wird es in Matlab wie eine normale Funktion: z.B. A=test(3);

MEX-Programm-Rahmen

```
#include"mex.h"

void mexFunction(int nl, mxArray* ml[], int nr, const mxArray* mr[])
{
  if (nl== ... && nr== ... )
  {                                    /* stimmen die Parameter? */
    /* code */
  }
  else
  {                                    /* Fehlerausgabe */
    mexErrMsgTxt("Fehlermeldung.\n");
  }
}
```

2.2 Zugriff auf Matrixdaten via "util.c"

"util.c" enthält ein paar Hilfsfunktionen, um den Zugriff auf Matrizen in MEX-Dateien zu erleichtern.

www.uni-ulm.de/~s_pbayer/util.html

Die Direktive `#include "util.c"` bindet die im folgenden beschrieben Hilfsfunktionen in ein Programm ein. Die Funktionen sollen den Zugriff auf den Typ mxArray erleichtern.
Die Datei "util.c" muß im selben Ordner wie das geschriebene MEX-Programm liegen.

mxArray $\hat{=}$ Matlabvariable

Hilfsfunktionen

- Lesen aus einer Matrix:
  - `double getr(const mxArray* ma,int y,int x);`
    Lesen des Realteils eines Wertes aus einer Matrix
  - `double geti(const mxArray* ma,int y,int x);`
    Lesen des Imaginärteils eines Wertes aus einer Matrix
- Schreiben in eine Matrix:
  - `void setr(const mxArray* ma,int y,int x,double value);`
    Schreiben in eine Matrix (Realteil)
  - `void seti(const mxArray* ma,int y,int x,double value);`
    Schreiben in eine Matrix (Imaginärteil)

Steuerung des Verhaltens der obigen Hilfsfunktionen

- Indextest (ungültige Zugriffe abfangen)
  Es wird bei den get- und set-Funktionen getestet, ob die Indizes x und y im gül-

Index-Test ein-/ausschalten

tigen Bereich liegen. Wenn man die Direktive `#define NOTEST` vor die Includedirektive `#include "util.c"` setzt, unterdrückt man den Test und beschleunigt dadurch das Programm (jedoch nur um einen konstaten Faktor).
Vorsicht: Ein ungültiger Zugriff kann einen Segmentation-fault erzeugen und evtl. Matlab zum Absturz bringen...

- Ungültige Zugriffe auf den Rand zurückführen
  Mit der Direktive `#define BORDERDEFAULT` wird, wenn man bei den get-Funktionen außerhalb des gültigen Bereichs ist, der Wert am Rand zurückgeliefert.

Bei ungültigem Index den Randwert der Matrix benutzen

2.3 Erstes MEX-Programm: first.c

Folgendes MEX-Programm bedient sich der Hilfsfunktionen aus "util.c". Desweiteren wird eine Ausgabematrix mit `mxCreateMatrix` erzeugt (siehe unten). In einer doppelten Schleife wird diese dann mit Werten gefüllt: Jede Zeile wird mit der jeweiligen Zeilenummer gefüllt:

```
A=first(3);
```

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix};$$

Vorsicht: In MEX-Programmen beginnen die Indizes bei 0 und nicht bei 1.

```
#include"mex.h"
#include"util.c"

void mexFunction(int nl, mxArray* ml[], int nr, const mxArray* mr[])
{
  int x,y;
  int n;
  double value;
  if (nr==1 && nl==1) {                    /* stimmen die Parameter? */
    n=mxGetScalar(mr[0]);                  /* Parmeter n auslesen */
    ml[0]=mxCreateDoubleMatrix(n,n,mxREAL); /* Rueckgabematrix erstellen */
    for(y=0;y<n;y++)
      for(x=0;x<n;x++) {
      value=y;
      setr(ml[0],y,x,value);          /* zeilen mit zeilennummer fuellen */
    }
  }
  else {
    mexErrMsgTxt("usage: A=first(n)");    /* Fehlerausgabe */
  }
}
```

2.4 Zugriff auf Matrixdaten via mx-Funktionen

Eine Befehlsreferenz der c-Funktionen, die in MEX-Programmen zur Verfügung stehen gibt es über die Haupthilfeseite (helpdesk) ⇒ "Application Program Interface".

Onlinehilfe: "Application Program Interface"

mxGetScalar	Den Wert aus einer skalaren Matlabvariable auslesen. `double mxGetScalar(const mxArray *array_ptr);`
mxGetM, mxGetN	Anzahl der Reihen und Spalten einer Matrix ermitteln. `int mxGetM(const mxArray *array_ptr);`
mxGetDimensions	Anzahl der Reihen und Spalten einer Matrix ermitteln. Das Ergebnis ist ein Pointer auf ein int-Array mit `mxGetNumberOfDimensions(...)` Elementen. (normalerweise 2: Höhe und Breite) `const int *mxGetDimensions(const mxArray *array_ptr);`
mxCreateDoubleMatrix	Matrix erstellen. Temporäre Matrizen (keine Rückgabematrizen) müssen mit `mxDestroyArray` wieder freigegeben werden. `mxArray *mxCreateDoubleMatrix(int m, int n, mxComplexity c);` *m,n : Höhe und Breite der neuen Matrix* $c \in \{mxREAL, mxCOMPLEX\}$*: reelle/komplexe Zahlen*
mxDestroyArray	Matrix freigeben. `void mxDestroyArray(mxArray *array_ptr);`
mxGetClass	Herausfinden, um was für einen Typ von Parameter es sich handelt. (siehe Onlinehilfe) `mxClassID mxGetClassID(const mxArray *array_ptr);`
mxGetPi, mxGetPr	Gibt Pointer auf die imaginären/reelen Matrixdaten zurück. (Die Daten liegen in einem eindimensionalen Array, die Position eines Matrixelements muß berechnet werden ⇒ util.c) mxGetPi liefert NULL, wenn es sich um eine reelle Matrix handelt. `double *mxGetPr(const mxArray *array_ptr);`

2.5 "Persistente" MEX-Variablen

Alle Variablen die außerhalb der MEX-Funktion deklariert wurden, bleiben während einer MATLAB-Sitzung zwischen Funktionsaufrufen erhalten. (Bsp.: siehe weiter unten, das erste octave-Beispiel: Dort werden die Funktionsaufrufe mitgezählt und ausgegeben).

MATLAB linkt die MEX-Programme zur Laufzeit dynamisch zu sich selbst dazu. Beim ersten Aufruf werden die Variablen initialisiert (direkte Zuweisung bei der Deklaration, `int i=0`). Wenn die MEX-Funktion dann ein weiters Mal aktiviert wird, also schon dazugelinkt wurde, behält die Variable ihren alten Zustand bei.

3 Tips

3.1 Matrizen statt Schleifen

Die MATLAB-Scriptsprache ist Matrizenorientiert und -optimiert. Sie geht höchst unperfomant mit Schleifen um.

3.1.1 Anwendung von skalaren Rechenoperationen auf ganzen Matrizen

Matlab bietet die Möglichkeit nahezu alle skalaren Rechenoperationen auch global auf allen Elementen einer Matrix auszuführen (vergleiche Einführung):
Bsp: Statt `for k=1:n; A(k)=B(k)*B(k); end;` kann man `A=B.*B` schreiben (A,B Vektoren). Oder statt `for k=1:n; A(k)=sin(B(k)); end;` kann man `A=sin(B)` schreiben.

sin(A)

3.1.2 Matrixvergleiche

1. Elementweise Vergleichen (A,B: Matrizen gleicher Dimension):
   `A == B` (allg. A *vergleichsoperator* B) ergibt eine 0-1-Matrix mit einer (logischen) Eins an den Stellen der Matrix an denen der Vergleich der Elemente aus A,B positiv war.
   $\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} < \begin{pmatrix} 1 & 2 \\ 8 & 9 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$

   $A == B$

2. Vergleich ob *alle* bzw.*mind ein* Element der Spalten einer Matrix eine logische Eins enthählt: `all(A)` bzw. `any(A)`. Ist A ein Vektor, so ist der Rückgabewert skalar.
   $\text{any}\left(\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} < \begin{pmatrix} 1 & 2 \\ 8 & 9 \end{pmatrix}\right) = \text{any}\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = 1;$
   $\text{all}\left(\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} < \begin{pmatrix} 1 & 2 \\ 8 & 9 \end{pmatrix}\right) = \text{all}\begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = 0;$

   any, all

3.1.3 Benutzung von Matrixausschnitten

1. Alle Elemente einer Matrix als einen Spaltenvektor ansprechen:

   `A=[1 2; 3 4];` $\qquad A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \Rightarrow A(:) = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$

   A(:)

2. Nichtzusammenhängende Matrixausschnitte:
   Neben rechteckigen Matrixauschnitten (Matrix B, siehe Einführung) können auch nichtzusammenhängende Matrixausschnitte (Matrix C,D) angesprochen werden:

   A(1:2,1:2)

   A(1:2:end,[1 4])

```
A=[1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16];
B=A(1:2,1:2);
C=A(1:2:end, 1:2:end);
D=A(2:2:end, 2:2:end);
```

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

$$\Rightarrow B = \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 3 \\ 9 & 11 \end{pmatrix} \quad D = \begin{pmatrix} 6 & 8 \\ 14 & 16 \end{pmatrix}$$

Es können auch Vektoren direkt als Bereichsbeschreibung angegeben werden:
$A([1\ 4],:) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 13 & 14 & 15 & 16 \end{pmatrix}$ (Hier wird die erste und die vierte Zeile ausgewählt)

3.2 MEX-Programme

1. Verwendung von mathematischen Funktionen wie `sqrt, sin`:
   Hier darf die Anweisung `#include“math.h”` nicht vergessen werden. (Es gab ohne diese Anweisung *komische* Ergebnisse.)

4 octave, ein MATLAB-clone

“octave” ist ein GNU MATLAB-clone. “.m”-Dateien können wie bei MATLAB angelegt werden. Der Funktionsumfang deckt nicht alle MATLAB-Befehle ab (z.B. fehlt conv2, imread,... Vieles kann man jedoch leicht selbst substituieren, oder eine Lösung aus dem Internet holen).
Die grundlegenden (Matrizen-) Operationen sind identisch. Eine umfangreiche Einführung in octave liegt im Postscriptformat vor: octave.ps.

4.1 octave's MEX-Programme: oct-Programme

octave kann, genau wie MATLAB, externe Funktionen dynamisch dazulinken. Diese werden bei octave in C++ geschrieben. Es steht eine umfangreiche C++-Bibliothek zur Verfügung. Eine Referenz dazu liegt ebenfalls im Poststcriptformat vor: liboctave.ps. Da diese etwas mager ist, kommt man hin und wieder um einen Blick in die Headerdateien nicht herum.

Aufbau einer externen C++-Funktion

```
#include<octave/oct.h>
...
DEFUN_DLD(name_der_funktion,
          eingabeparameter,
          erwartete_anzahl_ausgabeparameter,
          "Beschreibung der Funktion")
{
  ...
  return octave_value_list();
}
```

- Der *name_der_funktion* ist der Name der octave-Funktion, mit der man die Funktion in octave aufrufen will, und sollte mit dem cc-Dateinamen *name_der_funktion.cc* übereinstimmen.
- Die *eingabeparameter* sind ein Objekt vom Typ *octave_value_list*, welches wiederum von *Array* abgeleitet ist.
  - `args.length()` liefert die Anzahl der Argumente.
  - `args(x).is_string()` ist wahr, wenn ein string übergeben wurde.
  - `args(x).is_matrix()`, `args(x).is_real_matrix()`, `args(x).is_complex_matrix()` ist wahr, wenn eine Matrix (bzw. eine complexe oder reelle Matrix) übergeben wurde.
  - `args(x).is_scalar()`, `args(x).is_complex_scalar()`, `args(x).is_real_scalar()` ist wahr, wenn ein Skalar übergeben wurde.
  - `args(x).string_value()` liefert einen Stringwert.
  - `args(x).matrix_value()`,`args(x).complex_matrix_value()` liefert eine Matrix bzw. eine komlpexe Matrix.
- Die *erwartete_anzahl_ausgabeparameter* ist eine int-Variable, die die Anzahl der erwarteten Rückgabewerte enthält.
  Bei [A,B]=myfunction; ist dieser Wert gleich 2.
- Die *"Beschreibung der Funktion"* ist der Text der bei `help myfunction` angezeigt wird.
- Als Rückgabewert gibt es zwei Möglichkeiten:
  - octave_value_list: Eine Liste von Objekten des Typs octave_value.
  - octave_value: Ein Objekt des Typs octave_value. (Viele Objekte, wie Matrix, int, double, Complex) kann man mit `octave_value(variable)` konvertieren.
    (In diesem Fall wird das eine Objekt in eine Liste mit diesem einen Objekt umgewandelt.)

Ein kleines Beispiel:

Hier werden in einer "persistenten" Variable (count) die Funktionsaufrufe mitgezählt und zusammen mit der Anzahl der übergebenen Funktionsparameter ausgegeben. Am Ende wird eine Liste erstellt, mit Werten gefüllt und zurückgegeben.
Die Grösse der Liste passt sich den erwarteten Rückgabeparametern an. wird z.B. `[a,b,c]=myfunction;` eingegeben, so werden a=0, b=1, c=2 zugewiesen.

```
#include <octave/oct.h>
#include <iostream>

int count=0; // (statische) Variable: bleibt zwischen Aufrufen erhalten

DEFUN_DLD(myfunction,args,nargsout,"test me!") {

  int i,j;

  // Anzahl der Eingabeparameter und 'count' ausgeben:
  cout << "  #Aufruf = "<< count++ << endl;
  cout << "args.length()== " << args.length() << endl;
  cout << "nargsout== " << nargsout << endl;

  // Rueckgabeliste initialisieren:
  octave_value_list list;
  list.resize(nargsout);

  // liste fuellen:
  for (i=0;i<nargsout;i++) {
    list(i)=octave_value(i); //  Zahlen 0..nargsout
  }

  return list;
}
```

Übersetzt wird eine solche Datei mit `mkoctfile myfunction.cc`. mkoctfile name.cc

Anmerkung: Auf den Sun Workstations gab es beim compileren Probleme, weil der dortige g++ den *-shared* Parameter ignoriert: Mit `mkoctfile -v datei.cc` kann man sich die zwei g++-Aufrufe anzeigen lassen, die mkoctfile benutzt. Im zweiten Befehl (dem Linken) substituiert man den Parameter "-shared" mit "-Xlinker -G" oder man benutzt gleich Sun-Linker mit dem -G Parameter. (-G erzeugt shared Objects, wie sie von octave oder auch matlab verwendet werden) Es könnte jedoch sein, daß dieser Fehler in zukünfitgen Verisonen oder anderen Installationen nicht (mehr) auftritt. Fehler im Sunpool

Wie bei MATLAB muß die (erzeugte) Datei im octave-Such-Pfad liegen, damit sie beim Aufruf innerhalb von octave gefunden wird. Suchpfad
Der Suchpfad steht *innerhalb von octave* in der octave-Variable `LOADPATH`. Diese Variable entspricht der Umgebungsvariable `OCTAVE_PATH`. Es gibt auch eine Initialisierungsdatei "`~/.octaverc`". Diese könnte wie folgt aussehen:

```
LOADPATH = [LOADPATH ":/home/MaxMeier/MeineTools//"];
```

(Alternativ kann man auch die Umgebungsvariable setzen. Dann muß aber darauf geacghtet werden, auch die octave-Standardpfade einzuschließen.)
Suchpfade werden mit einem ":" getrennt. Der Doppelslash "//" am Ende eines Suchpfades bewirkt, daß dieser rekursiv durchsucht wird.

Ein weiteres Beispiel: Naive conv2-Implementierung:

conv2(b,f) := $\sum_{i,j=-\infty}^{\infty} \sum_{k,l=-\infty}^{\infty} f(k,l) \cdot b(i-k, j-l)$

```
#include <octave/oct.h>
#include <iostream>

DEFUN_DLD(conv2,args,,"conv2, naiv (1999, Pierre Bayerl).") {
  /* Stimmen die Parameter ?? */

  if (args.length()!=2 || !args(0).is_matrix_type() || !args(1).is_matrix_type()) {
    cout << "usage: A=conv2(B,Filter)" << endl;
    return octave_value_list();     // Keine Rueckgabe (leere Liste)
  }


  /* Variablen anmelden und initialisieren: */

  Matrix a;                         // Rueckgabematrix
  Matrix b,f;                       // Eingabematrizen

  int i,j,k,l,x,y;                  // Indizes
  double g;                         // Hilfsvariable

  b=args(0).matrix_value();         // Matrizen aus der
  f=args(1).matrix_value();         // Argumentliste kopieren

  a.resize(b.rows(),b.cols());      // Rueckg.-Matrixgroesse anpassen


  /* Berechnung */

  for (i=0;i<b.rows();i++) {
    for (j=0;j<b.cols();j++) {
      a(i,j)=0;                     // initialisieren
      for (k=0;k<f.rows();k++) {
        for (l=0;l<f.cols();l++) {
          y=f.rows()/2+i-k;         // Indizes y und x berechnen
          x=f.cols()/2+j-l;
          if (x>=0 && y>=0 && x<b.cols() && y<b.rows())
            g=b(y,x);               // Index (y,x) ist ok
          else
            g=0;                    // sont: am Rand 0 annehmen
          a(i,j)+=f(k,l)*g;         // addieren
  } } } }


  /* Rueckgabe */
                           // einzelnes octave_value wird automatisch
  return octave_value(a);  // in eine octave_value_list umgewandelt.
}
```

4.2 Plotten mit gnuplot

Ein mächtiger und einfacher Plotbefehl ist gplot, der das Programm gnuplot benutzt. Folgende Beispiele demonstrieren das plotten einer Sinuskurve und eines Histogramms:

gnuplot = gplot

Sinuskurve:

```
x=0:0.1:2*pi;
data=[x',sin(x)'];
gplot(data)
```

Histogramm: (h enthält z.B. 256 Häufigkeiten)

```
gplot(h')
```

Die Werte werden hier in *Spaltenvektoren* angegeben.

Drucken, Postscript

Um einen Plot z.B. als Postscript zu exportieren, muß man folgende gnuplot-Optionen, mittels `gset` angeben. (Mit `gshow` kann man die Einstellungen anzeigen)

Ausgabe am Bildschirm (X11)

```
gset term x11
```

Postscriptausgabe in "ausgabe.ps":

```
gset term postscript
gset output "ausagbe.ps"
```

Anschließend wiederholt man die plotbefehle oder benutzt `replot`, um den letzten Plot neu zu zeichnen.
Des weiteren kann man den Titel und den Zeichenstil durch hinzufügen der Befehle `title` *"Titel"* und `with` *Stil* ändern:

```
gplot data title "Hallo" with point
```

Stilmöglichkeiten:

- lines: Verbundene Linien
- point: Punkte
- linespoint : Punkte und Linien
- impulses: Senkrechte Linien

Weitere Optionen kann man dem octave-manual entnehmen.

Inhaltsverzeichnis