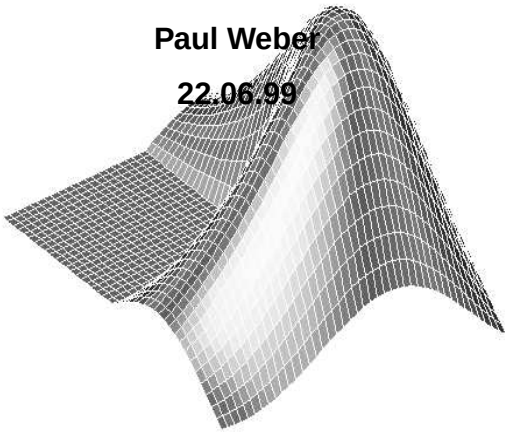
Einführung in MATLAB 5.3

Paul Weber

22.06.99



Was ist MATLAB?

MATLAB steht für **MAT**rix **LAB**oratory und ist ursprünglich als Interface zu LINPACK/EISPACK entwickelt worden:

- die Objekte, auf denen MATLAB arbeitet sind matrixorientiert, d.h.
  - Matrizen sind Felder mit 2 Indizes
  - Vektoren sind Felder mit 1 Index, also 1 x N - Matrizen
  - Skalare sind 1 x 1 - Matrizen
- MATLAB ist ein interaktives Programm mit einem Kommandointerpreter, die Kommandos können auch in einer Datei stehen und im Batch abgearbeitet werden = M-Files
- MATLAB rechnet rechnet numerisch, Schnittstellen zu Algebrasystemen wie MAPLE werden angeboten
- 2D und 3D Grafik; über ein *Handle Graphics* System kann man sich u.a. eigene grafische Oberflächen konstruieren
- Einbinden von eigenen FORTRAN- oder C-Routinen über sog. MEX-Files ist möglich.

Toolboxen und SIMULINK

- Sammlungen von M-Files, die für bestimmte Aufgabenstellungen gemacht sind, werden als *Toolboxen* angeboten, z.B. in Karlsruhe:
  - Control Toolbox für Regelsysteme
  - Neural Toolbox für neuronale Netze
  - Optimization Toolbox für Optimierungsprobleme
  - Signal Processing Toolbox für Signalverarbeitung
  - PDE Toolbox für partielle Differentialgleichungen
  - Fuzzy Logic Toolbox
  - Statistics Toolbox
  - Wavelet Toolbox
  - Mapping Toolbox
  - System Identification Toolbox
  - Financial Toolbox
- SIMULINK ist ein grafisches, blockorientiertes System zur Modellierung und Simulation dynamischer Systeme. SIMULINK-Modelle können auch aus MATLAB direkt oder über M-Files gestartet werden.

Verfügbarkeit

Unix-Version: Floating License,
im HP-Pool (Raum -122) und IBM-Pool (Raum -101)
und in allen vom RZ administrierten Pools verfügbar,
auf Institutsrechnern gegen Kostenbeteiligung

Aufruf

Unter Unix:	`matlab5` eingeben, es erscheint der Prompt `>>`
SIMULINK-Aufruf:	`>> simulink`
Toolbox-Aufrufe:	Eingabe der entsprechenden Kommandos
Beenden von MATLAB:	`>> quit` oder `>> exit`

Hilfe

- Nach Aufruf von MATLAB:
  - `>> intro` startet eine kurze Demo
  - `>> tour` Vorstellung der MATLAB Produktpalette
  - `>> demo` startet die MATLAB EXPO
  - `>> ver` zeigt alle installierten Toolboxen und die Versionen an
  - `>> help` übliches On-Line Helpsystem
  - `>> helpwin`
  - `>> lookfor`
  - `>> helpdesk` Hypertext-Dokumentationssystem unter Netscape
- Von UNIX aus: `matlabdoc`
- Handbücher zu MATLAB, SIMULINK und den Toolboxen beim Betreuer bzw. Online

Weitere Informationen

WWW:

MathWorks Homepage:	http://www.mathworks.com/
Scientific Computers:	http://www.scientific.de/
RZ Uni Karlsruhe:	http://www.uni-karlsruhe.de/~MATLAB/

Newsgroup: comp.soft_sys.matlab

Einfache Operationen

Arithmetische Operatoren:	`+, -, *, /`
Division von links:	`\`
Potenzieren:	`^`
Transposition und komplexe Konjugation:	`´`

Beispiel:

```
>> 3\6
 ans =
    2


>> 5^3
 ans =
   125
```

Priorität der Operationen und Regeln für Klammerung wie üblich.

Allgemein haben MATLAB-Kommandos die Form:

```
[variable=]expression[;]
```

- Falls kein Variablenname angegeben wird, wird das Ergebnis der rechten Seite in die Variable `ans` geschrieben.
- Falls ein `;` am Kommandoende eingegeben wird, wird das Ergebnis nicht angezeigt.

Beispiel:

```
>> x=12*3        >> 25/5             >> y=7^2.5;
    x =               ans =
      36                 5
```


Workspace

Variable werden im *Workspace* (Stack) gehalten.

- Lebensdauer der Variablen: bis Ende der MATLAB-Sitzung

```
>> who
Your variables are:
ans      x        y
```

```
>> whos
         Name        Size       Bytes    Class
          ans       1 by 1           8   double array
            x       1 by 1           8   double array
            y       1 by 1           8   double array
         Grand total is 3 elements using 24 bytes
```

- Abfrage der Variablen durch Eingabe des Namens:

```
>> y
y =
   129.6418
```

- Sicherung des Workspace

```
>> save [name][var1 var2 ...][-ascii][-double][-tabs]
   [-append]
```

sichert die angegebenen Variablen in der Datei *name*.mat. Falls keine Variablen angegeben werden, werden alle im Workspace befindlichen Variablen gesichert. Wird kein Dateiname angegeben, wird in der Datei **matlab.mat** gesichert. Wenn `-ascii` gesetzt ist, werden die Daten im ASCII-Format (8 Zeichen) gesichert, wenn zusätzlich `-double` angegeben wird, werden jeweils 16 Zeichen gesichert. Mit `-tabs` werden die Daten durch Tabulatoren getrennt. Mit `-append` wird an eine existierende Datei angehängt.

- Zurückladen geschieht durch

```
>> load [name[.extension]][var1 var2 ...]
```

- Löschen des Workspace oder einzelner Variable:

```
>> clear [var1 var2 ...]
```

- Kompaktifizieren des Workspace

```
>> pack
```

Beispiel:

```
>> x
x =
    36
>> clear x
>> x
??? Undefined function or variable ‘x‘.
```

Matrizen und Vektoren

Eingabe von Matrizen und Vektoren:

```
>> A=[a11 a12 ... a1n; a21 a22 ... a2n;...; an1 an2 ... ann]
```

oder

```
>> A=[a11 a12 ... a1n
      a21 a22 ... a2n
             .
             .
      an1 an2 ... ann]
```

Entsprechend wird ein Zeilenvektor durch

```
>> V=[b1 b2 ... bn]
```

und ein Spaltenvektor durch


```
>> VT=[b1
       b2
        .
        .
       bn]
```

beschrieben.

Die Matrixelemente bzw. Vektorkomponenten können
- reelle oder komplexe Zahlen
- Ausdrücke
- oder Matrizen bzw. Vektoren

sein.

Beispiel:

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

>> x=[-1.3 sqrt(3) 7/5*6]

x=
   1.3000    1.7321    8.4000

>> r=[10 11 12];
>> A=[A;r]

A =

    1   2   3
    4   5   6
    7   8   9
   10  11  12
```


Matrixoperationen

`+ - * / \ ^ ‘` beziehen sich auf die komplette Matrix bzw. Vektor

Beispiel:

```
>> A=[1 2 3; 4 5 6; 7 8 9];
>> B=A‘                                  Transposition
B =
    1  4  7
    2  5  8
    3  6  9
>> C=A+B                                 Addition bzw. Subtraktion
C =
    2  6 10
    6 10 14
   10 14 18
```

```
>> A*B
ans =
     14  32  50
     32  77 122
     50 122 194
```

Matrixmuliplikation
NxM- mit MxK-Matrix

```
>> x=[-1 0 1]; y=x-2;
>> x*y‘
ans =
     2

>> y*x‘
ans =
     2

>> x*y
??? Error using ==> *
Inner matrix dimensions must agree.
```

inneres Produkt


```
>> X=A\B
X =
  -0.3333   -7.3333  -14.3333
   0.6667   11.6667   22.6667
        0   -4.0000   -8.0000
```

X ist die Lösung von
A*X = B

```
>> X=C/B
X =
   5.6667   -9.3333    5.0000
 -11.3333   20.6667   -8.0000
 -15.3333   24.6667   -8.0000
```

X*B = C

```
>> x=A\y‘
x =
   9.6667
 -15.3333
   6.0000
```

x ist die Lösung des
Gleichungssystems
A*x = y‘

```
>> A^2
ans =
    30  36  42
    66  81  96
   102 126 150
```

Potenzieren von Matrizen A^p;
A muß quadratisch sein und
p ein Skalar

Falls p eine positive ganze Zahl ist, gibt die Potenz an, wie oft A mit sich multipliziert wird.

Falls p irgendeine andere Zahl ist, wird intern zunächst das Eigenwertproblem gelöst

```
A*v=v*D
```

wobei D die Diagonalmatrix der Eigenwerte ist und v der Eigenvektor.

$$D=\mathrm{diag}(\lambda_1\ \lambda_2\ \ldots\ \lambda_n)$$

Dann folgt

$$A^p = v * \mathrm{diag}(\lambda_1^p \lambda_2^p\ \ldots\ \lambda_n^p) / v$$

Operationen auf Feldelementen

in MATLAB-Bezeichnung: *array operations*

`.* ./ .\ .^` Array Operatoren

Diese Operatoren wirken bei Matrizen und Vektoren gleicher Dimension elementweise.

Beipiel:

```
>> x=[1 2 3]; y=[4 5 6];
>> z=x.*y
z =
   4 10 18

>> z=x.\y
z =
   4.0000 2.5000 2.0000
```

```
>> z=x.^y
z =
   1 32 729

>> z=x.^2
z =
   1 4 9

>>[x y]
ans =
     1 2 3 4 5 6

>>z=2.^[x y]
z =
   2 4 8 16 32 64
```


Falls ein Operand ein Skalar ist. erfolgt die Operationen ebenfalls elementweise. Hier unterscheiden sich einige der Array-Operatoren nicht von den Matrix-Operatoren:
{* / \} entspricht jeweils {.* ./ .\}

Beispiel:

```
>> A-5
ans =
     -4  -3  -2
     -1   0   1
      2   3   4
```

Addition bzw. Subtraktion einer Konstanten

```
>> B/2
ans =
    0.5000  2.0000  3.5000
    1.0000  2.5000  4.0000
    1.5000  3.0000  4.5000
```

Links- bzw. Rechtsdivision

```
>> 2\B
ans =
    0.5000  2.0000  3.5000

    1.0000  2.5000  4.0000

    1.5000  3.0000  4.5000
```

```
>> A*5
ans =
     5  10  15
    20  25  30
    35  40  45
```

Multiplikation mit einer Konstanten

Matrix-Erzeugung

`>> eye(n)`	erzeugt eine nxn - Einheitsmatrix
`>> eye(n,m)`	erzeugt eine nxm - Matrix mit 1 auf der Diagonalen und sonst 0
`>> zeros(n)` `>> zeros(n,m)`	erzeugt eine nxn - bzw. nxm - Matrix mit 0
`>> ones(n)` `>> ones(n,m)`	erzeugt eine nxn - bzw. nxm - Matrix mit 1
`>> rand(n)` `>> rand(n,m)`	erzeugt eine nxn - bzw. nxm - Matrix mit gleichförmig verteilten Zufallszahlen zwischen 0 und 1
`>> randn(n)` `>> randn(n,m)`	erzeugt eine nxn- bzw. nxm-Matrix mit Zufallszahlen aus einer Normalverteilung mit Mittelwert 0 und Varianz 1.

Sei A eine nxm - Matrix, dann wird durch

```
>> eye(size(A)); zeros(size(A));
>> ones(size(A)); rand(size(A));
```

jeweils ebenfalls eine nxm - Matrix erzeugt mit dem entsprechenden durch das Kommando gefordertem Aussehen.

Beispiel:

```
>> A=rand(3)
A =
    0.6711  0.0668  0.5890
    0.0077  0.4175  0.9304
    0.3834  0.6868  0.8462
```

```
>> B=inv(A)
B =
    1.1010 -1.3407  0.7078
   -1.349  -1.318   2.3887
    0.5964  1.677  -1.0777
```

Inverse Matrix


```
>> x=[1 2 3];C=diag(x)
C =
   1   0   0
   0   2   0
   0   0   3
```

Diagonalmatrix aus einem Vektor

```
>> triu(A)
ans =
   0.6711   0.0668   0.5890
        0   0.4175   0.9304
        0        0   0.8462
```

Obere Dreiecksmatrix von A

```
>> tril(A)
ans =
   0.6711        0        0
   0.0077   0.4175        0
   0.3834   0.6711   0.8462
```

Untere Dreicksmatrix von A

Darüberhinaus gibt es eine Reihe von weiteren Funktionen, die spezielle Matrizen erzeugen.

Matrixfunktionen

- Eigenwerte und -vektoren einer quadratischen Matrix A:

`>> [V,D]=eig(A)`	V ist eine Matrix, deren Spalten die Eigenvektoren enthalten, D ist eine Diagonalmatrix mit den Eigenwerten
`>> y=eig(A)`	erzeugt einen Spaltenvektor y mit den Eigenwerten

- Determinante, charakteristisches Polynom und Spur

`>> det(A)`	errechnet die Determinante
`>> trace(A)`	errechnet die Spur
`>> diag(A)`	erzeugt einen Spaltenvektor, der die Diagonale von A enthält

`>> poly(A)`	erzeugt einen Zeilenvektor, der die Koeffizienten des charakterischen Polynoms enthält

- Matrixeigenschaften

`>> D=size(A)` `>> [M,N]=size(A)`	D ist ein Zeilenvektor der Länge 2, dessen Komponenten die Anzahl der Zeilen und Spalten von A sind. Alternativ ist M die Anzahl der Zeilen und N die Anzahl der Spalten.
`>> length(x)`	gibt die Länge eines Vektors x aus
`>> rank(A)`	gibt den Rang von A an

- viele weitere Matrixfunktionen

Vektorfunktionen

Funktionen, die auf Vektoren operieren. Falls sie auf Matrizen angewendet werden, wird jede Spalte als Vektorargument genommen, das Ergebnis ist ein Zeilenvektor. Die wichtigsten sind:

`max, min, median`	Maximal-, Minimalwert bzw. Median der Komponenten
`mean, std`	Mittelwert, Standardabweichungen aus den Komponenten
`sum, prod`	Summe und Produkt der Komponenten
`cumsum, cumprod`	kumulierte Summe bzw. Produkt
`sort`	Sortieren in aufsteigender Reihenfolge
`diff`	Vektor aus den Differenzen benachbarter Komponenten: `[x(2)-x(1) x(3)-x(2) ... x(n)-x(n-1)]`

`hist`	Histogramm der Verteilung der Komponenten
`cov`	Varianz bei Vektoren, Kovarianzmatrix bei Matrizen
`corrcoef`	Matrix der Korrelationskoeffizienten

Die aufgelisteten Funktionen sind also dazu geeignet, in Vektor- bzw. Matrixform gelistete Daten zu analysieren. Dabei

- entspricht einem Vektor der Länge N, eine N-fache Messung einer Zufallsvariablen
- einer Matrix mit N Zeilen und M Spalten, eine N-fache Stichprobe von M Zufallsvariablen

Beispiel:

```
>> x=rand(1,6)
x =
    0.0009  0.7734  0.7273  0.3192  0.4177  0.6825
```

```
>> mean(x)
ans =
     0.4868
```
Mittelwert von 6 Meßwerten

```
>> median(x)
ans =
     0.5501
```
y = sort(x); (y(3)+y(4))/2

```
>> std(x)
ans =
     0.2992
```
Standardabweichung

```
>> cov(x)
ans =
     0.0895
```
Varianz = std(x)^2

Skalare Funktionen

Skalare Funktionen sind Funktionen, die auf einzelne skalare Variable wirken. Falls ein Vektor oder eine Matrix als Argument vorkommt, wirkt die Funktion auf jedes einzelne Element des Vektors oder der Matrix.

`sin, cos, tan`	Sinus, Kosinus, Tangens
`asin, acos, atan`	Arcsin, Arccos und Arctangens
`sinh, cosh, tanh`	hyperbolische Funktionen und
`asinh, acosh, atanh`	deren Umkehrfunktionen
`sqrt, exp, log, log10`	Wurzel, Exponentialfunktion, nat. bzw. dekadischer Logarithmus
`abs, real, imag, conj`	Absolutwert, Real- bzw. Imaginärteil, konjugiert komplexe einer Zahl

`gcd, lcm`	größter gemeinsamer Teiler, kleinstes gemeinsames Vielfaches
`fix, floor, ceil, round`	Rundungsfunktionen: `fix` kappt den Nachkommateil einer Dezimalzahl `floor` rundet auf die nächst niedere ganze Zahl `ceil` rundet auf die nächst höhere ganze Zahl `round` rundet entsprechend nach oben oder unten, je nach dem Nachkommateil einer Dezimalzahl
`sign, rem`	Signum-, Modulofunktion

Beispiel:

```
y =
  -2.1  -0.4  6.7  9.3  3.8
>> fix(y)
ans =
  -2   0   6   9   3
>> floor(y)
ans =
  -3  -1   6   9   3
>> ceil(y)
ans =
  -2   0   7  10   4
>> round(y)
ans =
  -2   0   7   9   4
>> sign(y)
ans =
  -1  -1   1   1   1
```

Vergleichsoperationen

In MATLAB gibt es 6 Vergleichsoperatoren:

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

Die Operatoren können Matrizen, Vektoren oder Skalare sein bzw. entsprechend zugelassene Ausdrücke davon. Der Vergleich findet elementweise statt. Das Ergebnis ist eine Matrix, ein Vektor oder Skalar mit Elementen 0 oder 1, je nach dem, ob das Ergebnis falsch oder wahr ist.

Bei Vergleichen einer Matrix oder eines Vektors mit einem Skalar, wird jedes Element mit dem Skalar verglichen.

Beispiel:

```
A =                          B =
  5   0   0                    4   9   0
  8   5   3                    6   8   6
  0   6   0                    5   5   4

A>=B
ans =
    1  0  1
    1  0  0
    0  1  0

A > 5
ans =
    0  0  0
    1  0  0
    0  1  0
```

Logische Operationen

Die logischen Operationen sind

`&`	and
`|`	or
`~`	not

`C=A` *`lop`* `B` A und B müssen Matrizen gleicher Dimension sein oder einer der beiden kann auch ein Skalar sein. Die logische Operation wird elementweise durchgeführt. C ist dann eine Matrix aus 0 und 1.

- `&` 1, wenn beide Elemente ungleich 0 sind
  0, wenn eines der beiden Elemente 0 ist
- `|` 1, wenn eines der beiden ungleich 0 ist
  0, wenn beide Elemente 0 sind

`B= ~A` Ein Element von B ist 1, wo das entsprechende Element von A gleich 0 ist, es ist 0, wo das Element von A ungleich 0 ist.

Logische Funktionen

Im Zusammenhang mit Vergleichs- und logischen Operationen stellt MATLAB einige Funktionen zur Verfügung.

`any, all` Falls das Argument ein Vektor (oder Skalar) ist, ist das Ergebnis bei
- `any` 1, falls mind. ein Element ungleich 0 ist
- `all` 1, falls alle Elemente ungleich 0 sind

In allen anderen Fällen ist das Ergebnis Null.
Falls das Argument eine Matrix ist, wird jede Spalte als Vektor behandelt und das Ergebnis ist eine Reihenvektor aus 0 und 1.

`find` falls das Argument ein Vektor ist, ist das Ergebnis ein Vektor mit den Indizes der Elemente, die ungleich Null sind. Falls das Argument eine Matrix ist, wird diese als Spaltenvektor aus den aneinandergehängten Spalten aufgefaßt.

`[i,j]=find(X)`	wobei X eine Matrix ist. i und j sind dann Spaltenvektoren, die den Reihen- bzw. Spaltenindex der Elemente von X enthalten, die ungleich Null sind.

Weitere Funktionen sind:

`exist`	Testet, ob eine Variable (im Workspace), ein M-File, eine SIMULINK- bzw. MATLAB-Funktion existiert.
`finite`	Entdeckt undefinierte Elemente (z.B. 1/0)
`isempty(X)` = 1 = 0	falls die Matrix X leer ist, sonst

Generierung von Vektoren

Zur Generierung von Vektoren gibt es
- die Doppelpunkt-Methode
- die `linspace` Funktion
- die `logspace` Funktion

`x=start:inkrement:end`	erzeugt einen n-komponentigen Vektor mit `n-1=(end-start)/inkrement`
`x=start:end`	das Inkrement ist 1
`y=linspace(x1,x2,n)`	erzeugt einen n-komponentigen Vektor mit den Elementen gleichverteilt zwischen x1 und x2. Falls n fehlt, wird n=100 angenommen.
`y=logspace(a,b,n)`	Elemente mit logarithmischen Abstand.

Beispiel:

```
>> x=1:5
x=
   1  2  3  4  5

>> y=0:pi/4:pi
y=
   0.0000   0.7854   1.5708   2.3562   3.1416

>> z=6:-1:1
z=
   6  5  4  3  2  1

>> k=linspace(-pi,pi,4)
k=
   -3.1416 -1.0472  1.0472   3.1416
```


Teilfelder und Indizierung

Die Doppelpunkt-Notation wird intensiv bei der Spezifikation von Teilfeldern von Vektoren oder Matrizen verwendet, da die Indizes von Vektoren und Matrizen selbst wieder als Vektoren aufgefaßt werden können.
Seien x und v Vektoren, dann ist x(v) ebenfalls ein Vektor, nämlich

```
[x(v(1)),x(v(2)),...,x(v(n))]
```

Die Elemente von v müssen jedoch positive ganze Zahlen sein.

`A(n1:n2,m1:m2)`	bezeichnet die Teilmatrix von A, die aus den Zeilen von n1 bis n2 und den Spalten von m1 bis m2 bestehen.
`A(:,3)`	ist die 3. Spalte von A. Ein " :" alleine bezeichnet den gesamten Bereich.
`A(:,[2 4])`	ist die 2. und 4. Spalte

`A(:,[2 5 10])=B(:,1:3)`	ersetzt die 2., 5. und 10. Spalte von A durch die erste 3 Spalten von B.
`A(:,n:-1:1)`	kehrt die Reihenfolge der Spalten um

Ein weiteres nützliches Konstrukt mit " : " ist A(:), wobei A eine Matrix ist.

- rechts von einer Zuweisung bedeutet A(:) ein Spaltenvektor, der alle Spalten von A hintereinander verkettet;
  `A=[1 2;3 4;5 6]`
  `b=A(:)`
  dann folgt `b=[1 3 5 2 4 6]'`
- wenn A schon existiert, dann bedeutet A(:) links von einer Zuweisung eine Umformung eines Vektors in die Form der Matrix A;
  `A(:)=11:16` ergibt dann
  `A=`
  ```
      11 14
      12 15
      13 16
  ```


Individuelle Elemente werden durch Angabe der Indizes ausgewählt. Also z.B.

`x(4), A(3,7)`

Grundsätzliches zu Zuweisungen:

- steht links von der Zuweisung eine Variable, die einen Skalar, Vektor oder eine Matrix bezeichnet und rechts ein Skalar, Vektor, eine Matrix oder ein Teilfeld, so nimmt die linke Seite den Typ und die Größe an, die die rechte Seite vorgibt
- steht dagegen links ein Teilfeld (einer Matrix oder eines Vektors), so muß die rechte Seite der Zuordnung die gleiche Größe haben.

Beispiel:

```
A =                          x=
   0.2727    0.7665              1  2  3  4  5  6
   0.4364    0.4777
```

```
>> B=x
B=
   1  2  3  4  5  6

>> B=A
B=
   0.2727    0.7665
   0.4364    0.4777

>> C=x(2:4)
C=
   2  3  4

>> A(1,:)=x(3:4)

A=
   3.0000    4.0000
   0.4364    0.4777
```

```
>> A(1,1)=x(6)
A=
   6.0000   4.0000
   0.4364   0.4777

>>A(2,:)=x

???  In an assignment
A(matrix,:) = B,
the number of columns
in A and B must be the
same.
```


Sei L ein Vektor der Länge m, der nur 0 und 1 als Komponenten enthält und A eine mxn Matrix, dann ist

`A(L,:)`	eine Matrix, die aus den Reihen von A besteht, die durch die Einsen in L bestimmt werden.
`v=(x<=3*std(x))`	v enthält dort 1, wo der Vergleich richtig ist
`x=x(v)`	x wird zu einem Vektor zusammengeschoben, der nur Elemente enthält, die kleiner als 3*std(x) sind. Beide Anweisungen können zu einer zusammengefaßt werden: `x=x(x<=3*std(x))`
`L=X(:,3)>100`	L ist ein Spaltenvektor mit 1 in der Reihe, deren Entsprechung in X eine Zahl größer als 100 in der 3. Spalte hat.
`X=X(L,:)`	entfernt die Zeilen aus X, an denen L eine 0 hat

`x=[]`	erzeugt eine *leere* Matrix. Eine leere Matrix ist eine 0x0 Matrix.

Beispiel:

`A(:,[2 4])` bezeichnet die 2. und die 4. Spalte von A. Die Anweisung

```
A(:,[2 4])=[]
```

eliminiert die 2. und 4. Spalte aus A.

Vektoren und Matrizen können zusammengesetzt werden, um dadurch größere Matrizen bzw. Vektoren zu erzeugen.

Beispiel:

A sei eine nxn Matrix. Dann ist

```
C=[A A';A.^2 one(size(A))]
```

eine 2nx2n Matrix.

Ausgabeformate

Zahlen in MATLAB werden im IEEE-Format dargestellt, standardmäßig auf 5 Ziffern genau. Mit dem `format` Kommando lassen sich verschiedene Formen und Genauigkeiten wählen:

`format short`	5 Ziffern, Festkomma (Default)
`format long`	15 Ziffern, Festkomma
`format short e`	5 Ziffern, Gleitkomma
`format long e`	15 Ziffern, Gleitkomma
`format short g`	beste Darstellung von `short` oder `short e`
`format long e`	beste Darstellung von `long` oder `long e`
`format hex`	Hex-Darstellung
`format bank`	auf 2 Dezimalstellen gerundet
`format rat`	Darstellung als rationale Zahl
`format +`	setzt für positive Zahlen ein +, für negative Zahlen ein -, für Nullen ein Blank ein.
`format compact`	Ausgabe erfolgt mit weniger Leerzeilen, also kompakter.

`format loose` Ausgabe erfolgt aufgelockerter, im Gegensatz zu format compact; Default

Beispiel:

```
x=[4/3 1.2345e-6]

short      1.3333          0.0000
short e    1.3333e+00      1.2345e-06
short g    1.3333          1.2345e-06
long       1.33333333333333   0.00000123450000
long e     1.333333333333333e+00
                  1.234500000000000e-06
long g     1.33333333333333   0.00000123450000
bank       1.33            0.00
rat        4/3             1/810045
hex        3ff5555555555555
                  3eb4b6231abfd271
```


MATLAB stellt Konstante zur Verfügung, die eine vorgegebene Bedeutung oder einen vorgegebenen Zahlenwert haben.

`ans` enthält immer das letzte Resultat, das nicht expilzit einer Variablen zugewiesen wurde

`eps` ist die Maschinengenauigkeit; der Abstand von 1 zur nächst größeren Gleikommazahl. Für IEEE-Format ist eps = 2.2204e-16

`realmin` kleinste bzw. größte darstellbare Gleitkommazahl:
`realmax` 2.225073858507201e-308
1.797693134862316e+308

`pi` 3.1416

`i,j` stehen bei komplexen Zahlen für $\sqrt{-1}$

`inf,NaN` IEEE Darstellung für ∞ und für Not-a-Number

`flops` kumulierte Anzahl der Floating Point Operationen

`clock,date`	Uhrzeit, Datum
`computer`	Computertyp (z.B. `HP700`)
`version`	MATLAB-Version: `5.3.0.10183(R11)`

Komplexe Zahlen

`z=a+b*i` sind alles gleichberechtigte Darstellung von komplexen
`z=a+b*j` Zahlen. a und b können Skalare oder auch Vektoren und
`z=a+bi` Matrizen gleicher Größe sein.
`z=a+bj`

`conj(z)` liefert die komplex Konjugierte von z

`real(z)` liefert den Realteil von z
`imag(z)` liefert den Imaginärteil von z

`angle(z)` ist atan(b/a), der Winkel bei der Polardarstellung von z
`abs(z)` liefert den Absolutwert, so daß

```
z=abs(z)*exp(i*angle(z))
```

ist.

Man beachte, daß nach einer Zuweisung an i, diese nicht mehr die Bedeutung der imaginären Einheit hat. Man kann sich eine eigene definieren, z.B. `ii=sqrt(-1); z=3+4*ii`

Polynome

Darstellung von Polynomen durch einen Reihenvektor, der die Koeffizienten in absteigender Reihenfolge der Potenzen der Variablen enthält. Z.B. der Ausdruck

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

entspricht dem Vektor

$$[a_n \; a_{n-1} \; \ldots \; a_1 \; a_0]$$

Sei `coeff` ein Zeilenvektor (der ein Polynom repräsentieren soll):

`r=roots(coeff)`	`r` ist ein Spaltenvektor, der die Nullstellen des Polynoms enthält
`p=poly(r)`	p ist ein Zeilenvektor, der aus `r` wieder die Polynomkoeffizienten berechnet (`p=coeff`)

Beispiel:

```
>> coeff=[1 2 1];
>> r=roots(coeff)
r  =
       -1
       -1
>> poly(r)
ans =
      1 2 1
```

p1 und p2 seien 2 Vektoren, die Polynome repräsentieren. Dann ist

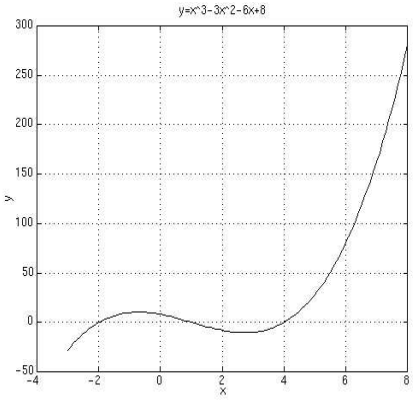
`p3=conv(p1,p2)`	das Produkt der beiden Polynome
`[p2,r]=deconv(p3,p1)`	der Quotient p2 und der Rest `r`
`q=polyder(p)`	Ableitung des Polynoms p
`p=polyfit(x,y,n)`	x und y sind Vektoren der x- bzw. y-Koordinaten von Punkten. `polyfit` erzeugt ein fittendes Polynom n-ter Ordnung p.

Sei p ein Polynom und x ein Vektor. Dessen Elemente sollen Auswertestellen des Polynoms sein:

`y=polyval(p,x)` y ist ein Vektor, der die Werte des Polynoms enthält. Durch plot(x,y) wird dann ein Plot des Polynoms erzeugt.

Beispiel:

```
>> p=[1 -3 -6 8];
>> x=-3:0.1:8;
>> y=polyval(p,x);
>> plot(x,y)
>> grid
>> xlabel('x')
>> ylabel('y')
>> title('y=x^3-3x^2-6x+8')
```



2D-Grafik

`plot` erzeugt xy-Plots. Die Argumente sind Vektoren gleicher Länge, deren Komponenten paarweise einen Datenpunkt in der xy-Ebene definieren. Zusätzlich kann ein Linientyp und die Farbe eingegeben werden.

`plot(x,y,'linetype')`

`x,y` sind Vektoren
`linetype` ist eine Kombination aus

-(Def)	y	yellow
--	m	magenta
:	c	cyan
-.	r	red
x	g	green
+	b	blue (Default)
o	w	white
*	k	black
.		
square		
diamond		

```
^
>
<
v
pentagram
hexagram
```

Um mehrere Kurven in einem Diagramm darzustellen, können mehrere Argumentsätze eingegeben werden.

Beispiel:

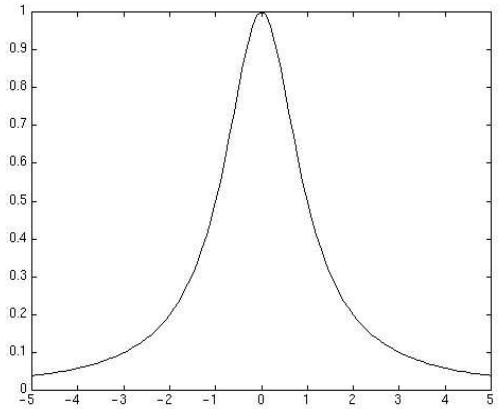
```
plot(x,y)
plot(x,y1,x,y2)
plot(x1,y1,'c+',x2,y2)
```

Bei verschiedenen Skalen wird eine gemeinsame Skala gewählt, in der alle Kurven dargestellt werden können.

`plot(y)` Falls nur ein Argument vorkommt, werden die Komponenten von y gegen den Index geplottet.

Beispiel: Die Funktion $y = 1/(1+x^2)$ soll im Intervall [-5,5] geplottet werden.

```
>> x=-5:0.1:5;
>> y=1./(1+x.^2);
>> plot(x,y)
```



Weitere Plotkommandos sind:

`loglog`	erzeugt xy-Plots mit logarithmischer x- und y-Achse
`semilogx` `semilogy`	erzeugt Plot mit logrithmischer *x*- bzw. *y-Achse*
`polar`	erzeugt Polarplots, wobei das erste Argument als Winkel und das zweite Argument als Radius interpretiert wird.

`title`
`xlabel`
`ylabel`
	versieht den Plot mit einem *Titel*
	einer *x-Achenbeschriftung* und einer
	y-Achsenbeschriftung. Das Argument dieser Kommandos besteht aus einem Text in Hochkomma, z.B.
	`title('Dies ist eine Überschrift')`

`legend`	fügt eine Legende in die Grafik ein
`text`	Fügt einen Text in den Plot an beliebiger Stelle ein. `text(x,y,'text')` (x,y) sind die Koordinaten im Diagramm

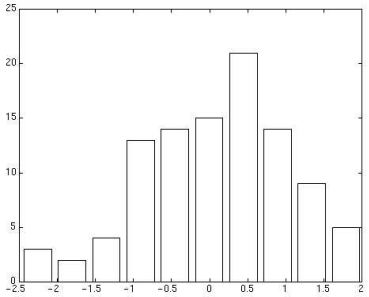
`gtext`	wie `text`, nur die Position wird durch Cursor festgelegt `gtext('text')`
`grid`	erzeugt ein Gitternetz; dieses Kommando hat kein Argument
`hold on` `hold off`	aufeinanderfolgende Plots werden überlagert

Histogramme

Sei y ein Vektor, dann wird durch

```
hist(y)
```

ein Histogramm erzeugt, in dem die Komponenten der Größe nach sortiert werden und 10 Balken ausgegeben werden.



`hist(y,n)`	es werden n Balken ausgegeben

`[n,x]=hist(y)`	es wird keinPlot ausgegeben, n enthält die Häufigkeiten, x die Positionen der Balken.

Falls `y` eine Matrix mit m Spalten ist, wird jede Spalte als eigene Balkengruppe geplottet.

`rose`	erzeugt ein Winkelhistogramm, das Argument wird als Winkelvariable aufgefaßt
`bar` `bar3`	erzeugt Balkendiagramme, 2D und 3D
`stairs`	erzeugt Balkendiagramme ohne interne Linien
`area`	XY-Plots, unterliegende Fläche wird eingefärbt
`pie`	Tortengrafik, ggf. mit explodierten Anteilen
`compass`	Polardarstellung von komplexen Zahlen

Funktionenplot

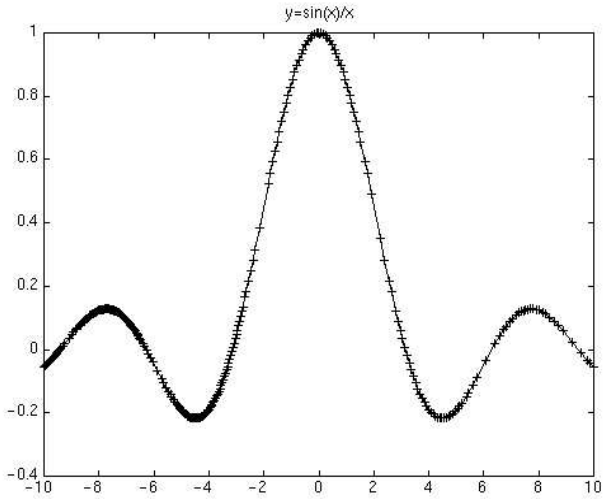
Sei *fun* eine Funktion, entweder intrinsisch oder vom Benutzer definiert.

```
fplot('fun',[xmin xmax ymin ymax],'marker',tol)
```

plottet die Funktion im angegebenen Intervall. Es werden Werte aus dem Definitionsbereich gewählt und von der Funktion ausgewertet. Die Anzahl hängt von der Fehlertoleranz `tol` ab (Default: 1e-3). Als Marker kann eine der Linientypen gewählt werden (Default: `'-'`). Die Auswahlpunkte können als Überlagerung geplottet werden, in dem zwei Linientypen kombiniert werden, z.B. `'-o'` .

Beispiel:

```
>> fplot('sin(x)/x',[-10 10],'-+')
>> title('y=sin(x)/x')
```



3D-Grafik

Linienplots

Seien x, y, z 3 Vektoren gleicher Länge. Dann erzeugt

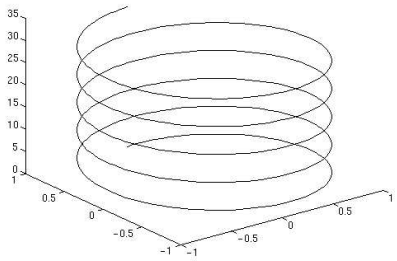
```
plot3(x,y,z,'linetype')
```

einen Linienplot im dreidimensionalen Raum, bei dem äquivalente Komponenten von x, y und z jeweils einen Punkt in 3D bedeuten.

Beispiel:

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

Überlagerungen von Plots sind möglich.



Gitternetze

Darstellung von z-Koordinaten oberhalb der Punkte eines rechteckigen Gitters in der xy-Ebene.

x und y seien 2 Vektoren, die die Werte entlang der x- bzw. y-Achse enthalten.

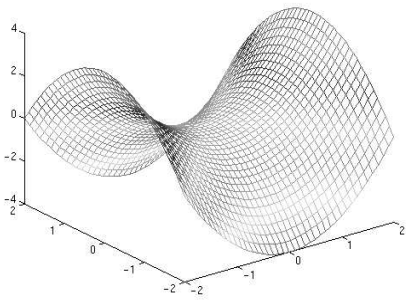
`[X,Y]=meshgrid(x,y)`	erzeugt 2 Matrizen X und Y, die die Werte der xy-Ebene enthalten
`Z=f(X,Y)`	Z ist eine Matrix; Funktion von X und Y
`mesh(X,Y,Z)`	erzeugt den Gitternetz-Plot

Man kann auch statt der Matrizen X und Y, die Vektoren x und y als Argumente verwenden. Es werden dann die Tripel (x_i, y_j, Z_{ij}) geplottet.

Bei `mesh(Z)` werden als x- und y-Achse die Vektorindizes verwendet. Der Farbverlauf der Fläche entspricht dem Z-Wert.

Beispiel:

```
>> x=[-2:0.1:2];y=x;
>> [X,Y]=meshgrid(x,y);
>> Z=(X.^2-Y.^2)
>> mesh(x,y,Z)
```



`meshc`	erzeugt zusätzlich noch Kontourplots auf der xy-Ebene
`meshz`	erzeugt zusätzlich Referenzebenen an den x- und y-Schnittebenen

Fabverläufe, Achsenskalierung und Ansichten können durch zusätzliche Kommandos kontrolliert werden (s. dazu `mesh, axis, view`).

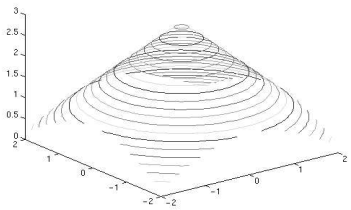
`surf, surfc und surfl` erzeugen schattierte Flächenplots

Kontourplots

`contour(x,y,Z,n)`	erzeugt Kontourplots mit n Kontourlinien; Defaultwert für n: 5
`contour(x,y,Z,v)`	erzeugt Kontourplot mit Kontourlinen an den Werten, die im Vektor v stehen
`contour3(X,Y,Z,n)`	erzeugt 3D Kontourplots

Beispiel:

```
>> Z=sqrt(X.^2 + Y.^2);
>> contour3(X,Y,Z,20)+3
```



Oberflächen-Plots

`pcolor(x,y,Z)`
`pcolor(X,Y,Z)`
`pcolor(Z)`
	Kontourplots mit Farbverläufen, jedes Z_{ij} wird entsprechend seinem Wert als farbiges Rechteck eingefärbt. Die Farbwerte werden aus einer Farbtabelle entnommen. Jede Zelle ist schwarz umrandet.

`shading flat`	Die Umrandung fällt weg.
`shading faceted`	Default
`shading interp`	Gouraud Shading

Die Standardansicht ist die Draufsicht auf die xy-Ebene. Sie kann durch `view` geändert werden:

`view(az,el)`	`az` ist der Azimuthwinkel um die z-Achse `el` ist der Höhenwinkel
`view(2)`	`az=0, el=90`
`view(3)`	`az=-37.5, el=30`
`view([x y z])`	Winkel in kartesischen Koordinaten

Ein *Image* ist eine Matrix, deren Elemente den Indizes einer (eigenen) Farbtabelle entsprechen.

`image(x,y,Z)`	Ansonsten ähnlich wie `pcolor`.
`image(X,Y,Z)`	
`image(Z)`	

Farbtabellen

Eine Farbtabelle (Color Lookup Table) ist eine mx3-Matrix, wobei die 3 Spalten in jeder Reihe den RGB-Anteil der Farbe festlegt. Die m Farben werden bei den Plots, die durch `mesh, surf, pcolor` und `image` erzeugt werden auf die Werte der Matrix abgebildet.

`colormap(`*`colm`*`)`	aktiviert die Farbtabelle *`colm`* , die entweder im Workspace bereitsteht oder eine von MATLAB bereitgestellte Farbtabelle ist.
`Map=colormap`	schreibt die aktive Farbtabelle in die Matrix `Map`

Von MATLAB werden 14 Farbtabellen zur Verfügung gestellt:

`hsv` Standardtabelle	`pink`	`spring`
`hot`	`flag`	`summer`
`cool`	`copper`	`autumn`
`jet`	`gray`	`winter`
`colorcube`	`lines`	

Farben

Standardmäßig verwendet MATLAB 256 Farben (indexed Colors, Pseudocolors). Die Farbzuweisung geschieht dann über den Index in der Farbtabelle.

Auf 24-Bit grafikfähigen Rechner kann MATLAB auch im Truecolor-Modus arbeiten, d.h. 2^{24} Farben ausgeben.

Dazu definiert man eine mxnx3 - Matrix C, wobei nxm die Dimension des Gitternetzes ist, über das geplottet werden soll. Die 3 Matrizen C(:,:,i) enthalten jeweils einen der 3 RGB-Anteile. Über

```
surf(Z,C)
```

wird dann die Truecolor-Matrix ausgewertet. Bei 8-Bit Grafik wird der Truecolor Modus simuliert.

Andere Aspekte der 3D Grafik wie Texturen, Beleuchtung, Kameraposition, Projektionen etc. findet man im Handbuch *Using MATLAB Graphics*.

Achsenskalierung

`axis([xmin xmax ymin ymax zmin zmax])`	skaliert die Achsen
`axis('auto')`	automatische Skalierung
`axis(axis)`	„friert" die aktuelle Skalierung ein
`axis('ij')`	skaliert im Matrixkoordinatensystem
`axis('xy')`	skaliert nach Matrixwerten
`axis('square')`	Plotbereich ist quadratisch
`axis('equal')`	Skalenfaktoren und Abstand der Skalenstriche sind gleich
`axis('off')`	schaltet die Achsen aus
`axis('on')`	schaltet die Achsen an

Farbskalierung

Üblicherweise wird das komplette Wertespektrum auf die Farbtabelle abgebildet. Man kann aber auch Teilbereiche davon abbilden. Die geschieht durch

`caxis([cmin cmax])`	wobei `cmin (cmax)` jeweils die untere (obere) Grenze des Wertebereichs ist

Verdeckungen

`hidden on` `hidden off`	Standard

mehrfache Fenster

`figure(n)`	aktiviert das n-te Fenster; falls dieses nicht existiert, wird es neu angelegt

Subplots

`subplot(m,n,i)`	teilt das aktive Fenster auf in eine mxn-Matrix aus Teilfenstern. Davon wird das i-te aktiviert.

Hardcopies

`print` *`plotdata -device`*

Das aktive Fenster wird in die Datei *plotdata.ext* geschrieben. Als Device gibt man das Grafikformat bzw. den Drucker an. Davon ist auch die Dateinamenserweiterung *ext* abhängig. Welche „Devices“ zulässig sind, kann man der Ausgabe von

```
help print
```

entnehmen.

Beispiel: `print meshdata -dps`

erzeugt ein PostScript-File **meshdata.dps** für schwarz-weiß Bilder

Programmieren von M-Files

Kontrollstrukturen

FOR Loops

```
for var=expression
   statements
end
```

`expression` ist im allgemeinen eine Matrix. Die FOR Loop wird durchlaufen, in dem nacheinenander jede Spalte von expression der Variable var zugewiesen wird und die *`statements`* ausgeführt werden.

Üblicherweise ist `expression` ein Zeilenvektor und die Spalten sind Skalare:

```
index=1:n;
for var=index
   statements;
end
```


Es gilt aber auch die bekannte Form der FOR Loop:

```
for i=1:n
     statements;
end
```

WHILE Loops

```
while expression
   statements
end
```

Die WHILE Loop wird solange durchlaufen, wie alle Komponenten der `expression` ungleich Null bzw. wahr sind. Meistens liegen Vergleichsoperationen vor:

```
while n>1
   n=n-1
      .
      .
end
```

IF

```
if expression-1
        statements
elseif expression-2
        statements

...
else
        statements
end
```

BREAK bricht FOR- oder WHILE-Loop ab:

```
while 1
        n=n-1
        if n<=0,break,end
        statements
end
```


SWITCH verzweigt in Abhängigkeit vom Wert eines Ausdrucks

```
switch expression
   case value1
        statements
   case value2
        statements
   .
   .
   otherwise
        statements
end
```

Skripts und Functions

M-Files sind Dateien, die MATLAB-Kommandos enthalten. Diese Dateien haben Namen der Form

filename.m

Gibt man

>> *filename*

ein, werden die einzelnen Anweisungen in dem M-File ausgeführt.

Skripts	sind M-Files, die mit Namen aufgerufen und dann ausgeführt werden.
Functions	sind M-Files, bei denen Parameter mitübergeben werden.
	Die 1. Zeile beginnt mit dem Wort *function*.


```
function y = sind(x)
% SIND(X) Sinuswerte der Elemente von X in Grad
% Autor: Helmut Huber
%        Karlsruhe, Stadtstrasse 11
%
y=sin(pi*x/180);
```

Aufruf:

```
>> alpha=[0:1:90];
>> beta=sind(alpha)
```

- die Variablen x und y in der Function sind lokal
- % leitet eine Kommentarzeile ein. Falls die 2. bis m-te Zeile Kommentarzeilen sind, werden diese nach Eingabe von
  `help sind`
  am Bildschirm ausgegeben, d.h. hier sollte die Function dokumentiert werden.
- die erste Kommentarzeile (H1-Zeile) wird von `lookfor` nach dem geforderten Stichwort durchsucht.

Subfunctions

Functions können selbst wieder Functions aufrufen. **Primary** (aufrufende) Function und **Subfunctions** stehen hintereinander im selben M-File. Subfunctions sind lokal. Help-Funktion und `lookup` durchsuchen nur Primary Functions.

Private Functions

Ein Unterverzeichnis des Arbeitsverzeichnisses mit dem Namen **`private`**, kann als Sammlung für eigene Funktionen dienen. Beim Aufruf von Funktionen und M-Files im Arbeitsverzeichnis wird auch das `private` Verzeichnis durchsucht.

Nützliche Features

`input` `n = input('text')`
gibt `text` am Bildschirm aus, wartet auf eine Eingabe und speichert diese in der Variablen n

`global` globalisiert Variable, Beispiel:

```
function yp=lotka(t,y)
%LOTKA Lotka-Volterra Modell
global ALPHA BETA
yp=[ y(1)-ALPHA*y(1)*y(2)
    -y(2)+BETA*y(1)*y(2)];
```

```
>> global ALPHA,BETA
>> ALPHA=0.01
>> BETA=0.02
>> [t,y)=ode23('lotka',0,10,[1;1]]
>> plot(t,y)
```

`pause` unterbricht die Abarbeitung eines M-Files bis der Benutzer eine Eingabetaste betätigt.
`pause(n)` unterbricht die Funktion für n Sekunden

`eval('string')` wertet das Textargument aus; z.B.

```
>> y='sin(x)'
>> eval(y)
```

Suchreihenfolge von M-Files

Gibt man den Namen eines M-Files, z.B. `sind`, ein, so arbeitet der Interpreter folgendermaßen:

1. Ist `sind` eine Variable?
2. Ist `sind` eine intrinsische MATLAB-Funktion?
3. Gibt es eine Datei `sind.m` im aktuelle Arbeitsverzeichnis?
4. Durchsucht alle Verzeichnisse, die im MATLAB Suchpfad eingetragen sind nach der Datei `sind.m`.

Erweitern des Suchpfades

`path`	zeigt den gesamten MATLAB Suchpfad an
`addpath` *`pfad`*	hängt ein Verzeichnis vor den MATLAB Suchpfad

Suchreihenfolge von Functions

Wird ein Function-Name aufgerufen, entweder direkt oder in einem M-File, ist die Reihenfolge

1. Ist der Name eine Variable?
2. Ist der Name eine Subfunktion?
3. Ist der Name im `private` Verzeichnis?
4. Ist der Name im MATLAB Suchpfad?

P-Code Files

MATLAB erzeugt beim Aufruf von M-Files und Functions einen Pseudocode (P-Code) , der im Hauptspeicher verbleibt bis er durch

`clear` *`function_name`*

gelöscht wird. P-Codes Files können gesichert werden durch

`pcode` *`mfile`*

`mfile``.m` wird dann auch als *`mfile`*`.p` gespeichert.

Performance-Aspekte

- wo immer möglich, Vektor- bzw. Matrixoperationen statt `for` oder `while` verwenden (Vektorisierung):

```
i=0;                    t=0:0.1:10;
for t=0:0.1:10          y=sin(t);
     i=i+1;
     y(i)=sin(t);
end
```

- Vektoren vorallokieren, dann braucht der Interpreter den Vektor, in den geschrieben wird, nicht jedesmal zu vergrößern:

```
y=zeroes(1,100);
for i=1:100
     y(i)=det(X^i);
end
```

- Präallokierung nutzt den Workspace besser aus und vermeidet Fragmentierung

Numerische Integration

MATLAB stellt 2 Funktionen zur numerischen Integration von analytischen Funktionen bereit:

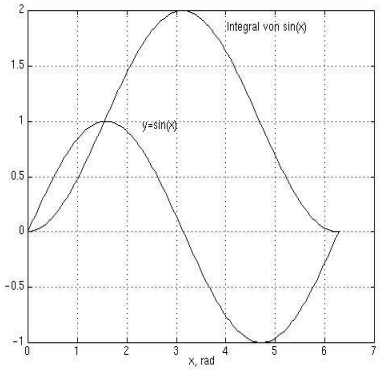
`q=` `quad('fun',a,b,tol,trace)`	Integration nach dem adaptiven Simpson-Verfahren `fun` Name der Funktion `a,b` Integrationsintervall `tol` Rel. Fehlertoleranz (1e-3) `trace` falls ≠ 0, wird ein Plot mit den Berechnungspunkten ausgegeben
`quad8`	Integration nach dem adaptiven Newton-Coates Verfahren

Falls die analytische Form der zu integrierenden Funktion nicht bekannt ist oder man die Stammfunktion plotten möchte, muß man andere Integrationsalgorithmen verwenden.

Euler-Verfahren

Hier bietet sich die MATLAB-Funktion `cumsum` an.

```
>> x=0:pi/60:2*pi;
>> y=sin(x);
>> yint=cumsum(y)*pi/60;
>> plot(x,y,x,yint)
>> grid
>> xlabel('x, rad'
>> gtext('y=sin(x)')
>> gtext('Integral von
sin(x)')
```



Trapezregel

MATLAB stellt dazu sie Funktion `trapz` zur Verfügung.

Sei x ein Vektor, y=fun(x) ein Vektor mit den Funktionswerten. Dann ist

```
q=trapz(x,y)
```

das Integral nach der Trapezregel.

Simpsonregel

Im Simpsonverfahren, wird die zu integrierende Funktion jeweils an drei Stützstellen durch Parabeln approximiert. Es folgt der Kern einer Funktion, der die Simpson-Integration durchführt. Man kann die Funktion noch komfortabler und funktionaler machen. Davon soll hier abgesehen werden.

```
function z=simp(x,y)
% SIMP(X,Y) Simpson Integration der Funktion y(x). X
% und Y sind Zeilenvektoren der gleichen Länge. Die
% Integrationsintervalle sind konstant.
[n,m]=size(y);
if length(x)~=n
   error('Die Eingabevektoren müssen die gleiche
          Länge haben')
end
if rem(n,2)==0
   error('Ungerade Anzahl an Intervallen')
end
dx=diff(x);z=0;
for i=1:2:(n-2)
   if(abs(dx(i)-dx(i+1)))>0.0001
        error('Ungleiche Intervalle')
   end
z=z+(y(i)+4*y(i+1)+y(i+2))*dx(i));
end
z=z/3;
```


Differentialgleichungen

MATLAB löst gew. Differentialgleichungen mit dem Runge-Kutta-Verfahren. Je nach Ordnung heißen die MATLAB-Files

`ode23` für Runge-Kutta 2. und 3. Ordnung

`ode45` für Runge-Kutta 4. und 5.Ordnung

`ode113` Adams-Bashfort-Moulton Solver

Für beide Verfahren ist die Syntax:

`[t,y]=ode`*`nn`*`('name',[t0 tf],y0)`

`name` ist der Name des M-Files, das die Funktion definiert, 1. Argument muß `t`, 2. Argument muß y sein
`t0,tf` Anfangs- und Endzeitpunkt der Integration
`y0` Spaltenvektor mit den Anfangsbedingungen

Ausgabe: y ist der Zustandsvektor, ausgewertet zu jedem Zeitpunkt t

Beispiel: Physikalisches Pendel

$\Theta'' = -g l \sin\Theta$

übliche Schulnäherung: $\sin\Theta \approx \Theta$

Auflösen in 2 DGLen 1. Ordnung:

```
Θ(1)‘= -glsinΘ(2)
Θ(2)‘= Θ(1)
```

Anfangsbedingungen:

```
Θ(1) = 0
Θ(2) = 1
```

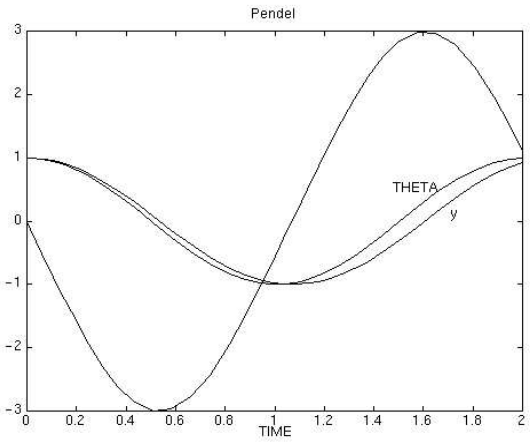
Definiere Funktion:

```
function dx=pendel(t,x)
dx=[-9.81*sin(x(2));x(1)];
```

```
>> theta0=[0;1];
>> [t,theta]=ode23(‘pendel‘,[0 2],theta0);
>> a=sqrt(9.81);
>> y=cos(a*t);
>> plot(t,y,t,theta)
```



```
>> xlabel(‘TIME‘);
>> gtext(‘THETA‘);
>> gtext(‘y‘);
>> title(‘Pendel‘);
```



Steife Systeme

```
function dy=stiff(t,y);
a=[998 1998;-999 -1999];
dy=a*y;
```

```
>> y0=[1;0];
>> [t,y]=ode23('stiff',[0 1.0],y0);
```

Das System ist ein sog. *steifes* System, d.h., die exakte Lösung

$$y_1 = 2e^{-t} - e^{-1000t}$$
$$y_2 = -e^{-t} + e^{-1000t}$$

zeigt sehr unterschiedliche Zeitskalen.

Berechnungen von steifen Systemen mit üblichen ODE Solvern können sehr lange dauern oder auch scheitern.

`ode15s` und `ode23s`

sind 2 Solver speziell für steife Systeme

Eine Zeitmessung des o.g. Problems zeigt, daß

`ode23s` etwa **dreimal** schneller ist als `ode23s`

Weitere Informationen speziell zur Steigerung der Performance von ODE Solver findet man im Handbuch *Using MATLAB*.

Was hier nicht behandelt wurde

- Debugger
- Fourier Analyse und FFT
- Vertiefte Behandlung der ODE Solver
- Sparse Matrizen
- M-File Programmierung
- Daten Typen, insbesondere Felder und Strukturen
- Definition von Klassen und Objekte
- Ein-/Ausgabe von Daten
- Einbindung von eigenen C- oder Fortran-Routinen

Im Grafikbereich

- 3D Grafik: Beleuchtung, Projektionen etc.
- Bildbearbeitung
- 3D Modellierung
- Handle Graphics System