Hochschule München

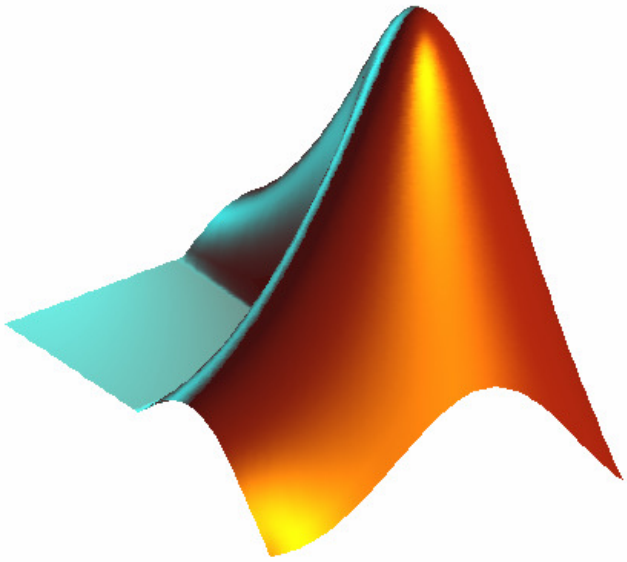
Fakultät 03

Numerische Methoden

EINFÜHRUNG

IN

MATLAB



Prof. Dr.-Ing. Katina Warendorf

Prof. Dr.-Ing. Peter Wolfsteiner

Florian Löser

Inhaltsangabe

1. Einführung

Der Name MATLAB leitet sich ursprünglich von Matrix Laboratory ab. Heute steht MATLAB für ein umfangreiches Softwarepaket für numerische Berechnungen mit leistungsfähiger grafischer Darstellung. Es ist verbreitet in den Mathematik- und Ingenieurswissenschaften sowie allen anderen wissenschaftlichen Disziplinen. Man findet es an Hochschulen, in Forschung und Industrie.

MATLAB verfügt über eine eigene Hochsprache. Eine Konvertierung von/in andere Sprachen (C/C++, FORTRAN, JAVA) ist über das sog. "Application Program Interface" (API) möglich. Während andere Sprachen nur skalar rechnen können, d.h. eine Variable bekommt einen Wert zugewiesen, welcher dem Algorithmus entsprechend verarbeitet wird, rechnet MATLAB auf Basis von Matrizen bzw. sog. Arrays, d.h. eine Variable beinhaltet eine Vielzahl von Werten, die in einem Schema abgelegt sind (z.B. Anzahl Zeilen/Spalten im Falle einer zweidimensionalen Matrix). Ferner müssen Matrixoperationen in anderen Sprachen relativ aufwendig mittels Schleifen programmiert werden, wohingegen in MATLAB eine einfache Eingabe in mathematischer Schreibweise genügt.

Zentraler Begriff ist das sog. Array, ein n-dimensionales Datenfeld, in das sich alle anderen Datenstrukturen einordnen lassen. Neben Zahlenwerten kann es ebenso einzelne Zeichen und Zeichenketten enthalten. Die klassische Matrix als ein numerisches, zweidimensionales Feld stellt nur einen Spezialfall eines solchen Arrays dar. Ein Vektor wiederum ist bekannterweise eine eindimensionale Matrix, angeordnet als Spalten- oder Zeilenvektor, und ein Skalar eine 1x1-Matrix. Allgemein gesprochen handelt es sich also immer um Arrays.
In MATLAB gibt es neben den aus der Mathematik bekannten Rechengesetzen für Matrizen und Vektoren (Matrizenalgebra) noch weitere Rechenvorschriften. Diese sog. Array-operationen erfolgen elementweise, das bedeutet, dass jedes einzelne Element eines Arrays bzw. einer Matrix angesprochen wird und bei einer Operation mit zwei Arrays jeweils die Elemente gleicher Position (i,j und m,n) miteinander verrechnet werden. Da dies ein zentrales Merkmal ist, sei zur Begriffsabgrenzung noch einmal verdeutlicht: einerseits die bekannten Operationen der Matrizenalgebra bzw. der linearen Algebra, andererseits die MATLAB-eigenen sog. Array-Operationen, die stets elementweise erfolgen.

Schließlich sind zahlreiche Programmerweiterungen erhältlich, z. B. für die Signalver-arbeitung, Bildverarbeitung, Statistik, Mechanik, Hydraulik, Optimierung.

Dieses Skript ist an vielen Stellen knapp gefaßt und erfordert, dass Sie an manchen Stellen die eingebauten Hilfefunktionen benützen (siehe Kapitel "Hilfe") und das hier Abgedruckte nachvollziehen, indem Sie es in MATLAB eingeben. Im Aufgaben- und Lösungsteil finden Sie nach Inhalten geordnete Übungen zur Vertiefung Ihres Verständnisses.

2. Die Arbeitsoberfläche

Hier sei nur ein grober Überblick über die Arbeitsoberfläche gegeben. Durch Ausprobieren erklärt sich vieles von selbst.

Der Desktop von MATLAB besteht aus:
- *Command Window* (Befehlseingabe)
- *Command History* (Aufzeichnung der eingegebenen Befehle)

und standardmäßig in einem gemeinsamen Fenster untergebracht:
- *Current Directory Browser* (Liste sämtlicher Dateien im aktuellen Verzeichnis)
- *Workspace* (Liste der existierenden Variablen)

Im Menüpunkt *Help* findet man interessante Beispiele und Anleitungen zu diesem Thema (je nachdem welches Fenster beim Aufruf aktiv ist: *Using the Desktop, Using the Current Directory Browser, Using Directory Reports, Using the Command Window).* Im Menüpunkt *Desktop* lassen sich diese und weitere Programmteile ein- und ausblenden.

Die einzelnen Komponenten können in ihrer Positionierung und Größe mit der Maus verändert werden. Mit der Dock-Eigenschaft (ein schräger Pfeil in jedem Fenster oben rechts) lassen sich Komponenten abkoppeln und als unabhängige Fenster handhaben. Figure Windows, Editor/Debugger und Arrayeditor können zusätzlich an das Command Window angekoppelt werden. Dabei bestehen unterschiedliche Möglichkeiten der Anordnung, beispielsweise nebeneinander oder hintereinander. Probieren Sie es aus!

Unten links ist der Startbutton angeordnet. Von hier aus lassen sich ähnlich wie in der WINDOWS-Oberfläche diverse Funktionen und Unterprogramme aufrufen und verwalten. Für sich häufig wiederholende Aufgaben lassen sich eigene Shortcuts definieren und in die Liste der Shortcuts (Unterpunkt des Startbutton) einfügen.

Command Window
Hier werden die Befehle eingegeben.

Command History
In der Command History werden die eingegebenen Befehle zur Wiederverwendung abgespeichert. Durch Doppelklick oder Taste F9 können diese Befehle direkt ausgeführt werden und mit der linken Maustaste in das Command Window zur erneuten Bearbeitung verschoben werden. Die linke Maustaste gemeinsam mit Shift oder Strg erlaubt eine Auswahl mehrerer Kommandos, Strg+A funktioniert ebenso. Mit der rechten Maustaste öffnet sich das Kontextmenü, welches weitere Optionen anbietet.

Current Directory
Der Current Directory Browser funktioniert wie der WINDOWS-Explorer: Verzeichniswechsel, Datei-Infos, Öffnen von Dateien durch Doppelklick, Suchfunktion, Dateien umbenennen etc. Html-Dateien werden übrigens im integrierten eigenständigen Webbrowser geöffnet.
Das rechte Symbol in der Menüleiste des Current Directory Browser stellt weitere Funktionen bereit wie Optimierung des Quellcode eines m-Files, Aufzeigen von Abhängigkeiten der m-Files untereinander u.a.
Mit der Eingabe *filebrowser* im Command Window wird der Current Directory Browser aktiviert bzw. geöffnet, ebenso über den Menüpunkt *Desktop*.

Workspace
Die Variablen lassen sich wie im WINDOWS-Explorer je nach angeklickter Spalteneigenschaft sortieren. Scrollen Sie nach rechts, um alle Spalten zu sehen. Machen Sie sich auch mit den Icons der Menüzeile vertraut.

Weitere Programmsegmente sind:

Help Browser
Der Help Browser wird mit dem Befehl *doc* oder aus dem Menü heraus aufgerufen. Mehr hierzu im folgenden Kapitel *Verwendung der Hilfe.*

Array Editor
Er läßt sich öffnen über den Befehl *openvar Variablenname,* über den Menüpunkt *Desktop* oder durch Doppelklick auf eine Variable im Workspace. Hier lassen sich Arrays und Matrizen bequem und übersichtlich einsehen, erstellen und verändern. Wie in Microsoft EXCEL sind Spalten teilbar. Daten aus EXCEL können u.a. mit Copy/Paste in den Array Editor kopiert werden.

Editor

Der Editor ist ein integrierter ASCII-Editor mit Debugger-Funktion für m-Files. Er ist aufrufbar durch den Befehl *edit Dateiname*, über den Menüpunkt *Desktop* oder durch Doppelklick auf eine Datei im Current Directory Browser. Von hier aus werden bequem Programme und Funktionen geschrieben und ausgeführt, bei Bedarf mit der integrierten Debug-Funktion.

3. Verwendung der Hilfefunktionen

MATLAB bietet umfangreiche Hilfefunktionen an. Der nicht weiter spezifizierte Befehl *help* ruft die Hilfe im Command Window auf. Es erscheint ein Überblick über alle verfügbaren MATLAB-Funktionsgruppen. Durch Klicken auf die blau hinterlegten Einträge kann der gesuchte Befehl gefunden werden. Mit dem Kommando *help Befehl* (z. B. *help sqrt*) erhalten Sie Hilfe zu jedem Befehl. Dabei erscheint dieser zum Zwecke der Hervorhebung in der Erklärung in Großbuchstaben (z.B. *SQRT(X)*). Die richtige Eingabe erfolgt in MATLAB jedoch immer in Kleinbuchstaben.

Eine komfortablere Hilfe erhalten Sie mit dem Kommando *doc*. Dieser öffnet den Help Browser. Analog erhalten Sie mit dem Kommando *doc Befehl* Hilfe zu einem Befehl im Help Browser. Im Help Browser können Sie sich im Verzeichnisbaum systematisch durchklicken, unter *Index* alphabetisch nach Kommandos suchen oder unter *Demos* aus einer Vielzahl von abspielbaren Demonstrationen auswählen. Letztere können Schritt für Schritt im Command Window ausgeführt (*Run in the Command Window*) oder im Editor geöffnet (*Open … in the Editor)* und auf die gleiche Weise dargestellt werden (das jeweilige Feld markieren und auf das Icon *Evaluate Cell* klicken).
Unter S*earch for:…* im Help Browser finden Sie u.a. auch ausführliche Hilfe zu den Programmoberflächen wie Editor oder Debugger.

Das Kommando *lookfor* ermöglicht es, mittels Schlüsselwörtern nach MATLAB-Funktionen zu suchen, wenn deren Syntax nicht genau bekannt ist. Dabei wird die erste Zeile des Help-Textes jeder Funktion zurückgegeben, die das entsprechende Schlüsselwort enthält. Zum Beispiel gibt es in MATLAB keine Funktion mit dem Namen *inverse*. Geben Sie *lookfor inverse* ein um ein Suchresultat zu erhalten. Dabei zeigt MATLAB mit der Angabe "*Busy"* rechts neben dem Startbutton an, dass es noch sucht. Dauert es zu lange, unterbrechen Sie mit der Tastenkombination *STRG-C*. Will man, dass alle Help-Zeilen durchsucht werden, so gibt man zusätzlich die Option *–all* an: *lookfor inverse –all*. Das Suchergebnis fällt dann entsprechend umfangreicher aus.

Auf der Homepage von MATLAB unter http://www.mathworks.com/ ist ebenfalls die gesamte MATLAB-Dokumentation nutzbar. Gehen Sie auf *Support*, wählen das Produkt *MATLAB* und dann *Documentation.*

Ferner gibt es im Internet eine Unzahl von MATLAB-Dokumentationen. Die meisten sind in Englisch verfaßt, viele gibt es auch in Deutsch.

4. Grundlagen

Jede Zahl wird in MATLAB defaultmäßig als Gleitkommazahl des Datentyps (=*class*) *double* gespeichert und belegt damit einen Speicherplatz von 8 Byte = 64 Bit. So ergibt sich ein Zahlenbereich von ca. 10^{-308} bis 10^{+308} und eine relative Rechengenauigkeit von $2^{-53} \approx 1.11 \cdot 10^{-16}$. Grob gesagt: MATLAB führt elementare Rechenoperationen mit einer Genauigkeit von ungefähr 16 Dezimalstellen durch und speichert Zahlen ebenso ab. Genauere Werte

erhalten Sie durch Eingeben der Kommandos *realmin* (kleinste positive Gleitkommazahl), *realmax* (größte positive Gleitkommazahl) und *eps* (Genauigkeit). Die Funktion *eps* gibt den Abstand von 1.0 zur nächst größeren Gleitkommazahl an. Dieser Abstand beträgt die doppelte relative Rechengenauigkeit.

```
>> realmin, realmax, eps
ans =
  2.2251e-308
ans =
  1.7977e+308
ans =
  2.2204e-016
```

Werden die Grenzen des Zahlenbereichs überschritten (Overflow), dann antwortet MATLAB:

```
>> 1.1*realmax, -1.1*realmax
ans =
   Inf
ans =
  -Inf
```

Ist das Ergebnis einer Rechenoperation kleiner als *realmin*eps*, dann stellt dies einen Underflow dar und das Ergebnis wird Null gesetzt.

```
>> realmin*eps/2
ans =
     0
```

Ist eine Rechnung mathematisch nicht definiert, so ist das Resultat *NaN*, was für Not a Number steht.

```
>> 0/0, inf/inf, 0*inf
ans =
   NaN
ans =
   NaN
ans =
   NaN
```

V.a. um Speicherplatz und Rechenzeit einzusparen, gibt es noch weitere numerische Datentypen. *Single* benötigt 4 Bytes, die Hälfte des Speicherplatzes von *double*. Beispielsweise benötigt eine 4x4-Matrix des Typs *double* 128 Bytes, im *single*-Modus nur 64 Bytes.
Daneben gibt es noch die ganzzahligen Datentypen ohne Vorzeichen *uint8, uint16, uint 32, uint64* (u steht für "unsigned") sowie mit Vorzeichen analog *int8* bis *int64*. Finden Sie im Help Browser die jeweiligen Wertebereiche heraus (z.B. rechnet uint8 mit 8 Bit, das ergibt einen Wertebreich von 0 bis 2^8-1, also 0 bis 255).

Die Namen aller Datentypen lassen sich auch als Funktion zum Konvertieren von einem in einen anderen Datentyp verwenden:

```
>> single(7), uint16(13.4)
ans =
     7
ans =
     13
```

Mit *format* formatieren Sie die Bildschirmausgabe: *format short* gibt Zahlen mit vier Nachkommastellen aus, *format long* mit 14-15 Stellen. Die Voreinstellung *format loose* erzeugt Leerzeilen in der Ausgabe, wogegen *format compact* diese wegläßt und Platz spart. Alle Beispiele in diesem Skript wurden mit *format compact* und *format short* erzeugt. Weitere Optionen bieten z.B. *format hex* (hexadezimale Ausgabe), *format rat* (Ausgabe in Form eines Bruches) oder *format +* (Ausgabe der Vorzeichen). Weitere Möglichkeiten finden Sie mit *help format* oder *doc format*. Sie können die Voreinstellungen bezüglich der Ausgabe ändern

unter P*references → Fonts → Command Window → Numeric display.* Das Kommando *get(0)* zeigt alle aktuellen Einstellungen des Command Window an, darunter auch *Format* und *FormatSpacing* (Zeilen 8 und 9) mit den Einstellungen *short/long* und *loose/compact*.

Ähnlich wie in MICROSOFT DOS sind auch in MATLAB einige Tasten mit Zusatzfunktionen belegt: Die TAB-Taste vervollständigt unvollständige Eingaben indem eine Auflistung der möglichen Befehle angezeigt wird (z.B. bietet die Eingabe *>> re* und dann TAB-Taste u.a. die bereits bekannten Befehle *realmin* und *realmax* als mögliche Auswahl an).
Mit den Cursertasten Up/Down kann man alle bereits eingegebenen Befehle durchscrollen und erspart sich damit Tipparbeit.

Einige Basiskommandos:

clear Variablenname	Löscht die angegebenen Variablen (z.B. *clear x y z*, *doc clear*)
clear, clear all	Löscht kompletten Inhalt des Workspace
clc	Löscht Bildschirm der Command Window, keine Variablen
who	Zeigt Variablen an
whos	Zeigt mehr Informationen an als *who*
what	Zeigt eine Liste der m-Files im Current Directory
CTRL-C	Unterbricht MATLAB, z. B. im Fall "Busy"
quit/exit	Beendet MATLAB per Eingabe

4.1 Erste Schritte in der Dateneingabe

MATLAB unterscheidet zwischen Groß- und Kleinschreibung, Leerzeichen können beliebig gesetzt werden. Üblicherweise werden Matrizen mit Großbuchstaben, Vektoren mit Kleinbuchstaben bezeichnet.

```
>> x=3
x =
     3
>> X = 4
X =
   4
```

Ein Semikolon am Zeilenende unterdrückt die Ausgabe, mit Komma lassen sich mehrere Befehle in einer Zeile unterbringen:

```
>> x = 5;
>> a = 1, b = 2;
a =
     1
>> who

Your variables are:

a  b  x
>> whos
  Name      Size             Bytes  Class     Attributes

  a         1x1                  8  double
  b         1x1                  8  double
  x         1x1                  8  double
>> clear a x, who
Your variables are:

b
```

Der Befehl *whos* zeigt: Obige Variablen haben die Größe/Ordnung 1x1, d.h. sie sind ein Array mit einer Zeile und einer Spalte, also ein Skalar. Der Speicherplatz je Variable vom Typ *double* beträgt wie gehabt 8 Bytes.

Vektoren, Matrizen und Arrays werden in eckigen Klammern erzeugt. Die Trennung der einzelnen Elemente erfolgt durch Kommata oder Leerzeichen, Zeilen werden durch Semikolon oder *Return* abgeschlossen:

```
>> A = [1,2,3; 4,5,6];
>> A = [1 2 3; 4 5 6];
>> A = [1 2 3   <Return>
4 5 6]
A =
     1     2     3
     4     5     6
```

Eingabe eines Zeilenvektors:

```
>> z = [1 2 3]
z =
     1     2     3
```

Eingabe eines Spaltenvektors:

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

oder durch Transposition des Zeilenvektors:

```
>> t = z'
t =
     1
     2
     3
```

Einzelne Elemente werden mit runden Klammern angesprochen. Dabei bezieht sich der erste Index wie gewohnt auf die Zeile, der zweite auf die Spalte: *A(2,1)* spricht in diesem Fall die Zahl 4 an. Mit *A(2)* erhält man das gleiche Ergebnis, da MATLAB die Elemente einer Matrix spaltenweise durchnummeriert, beispielsweise sieht die Reihenfolge der Elemente in einer 3x4-Matrix so aus:

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

Sie haben immer die Möglichkeit, Ihre Eingabezeile mit drei Punkten … zu beenden und dann in der nächsten Zeile fortzufahren:

```
>> s = 1+1/2+1/3+1/4+1/5+1/6+...
1/7+1/8+1/9+1/10
```

Die mathematischen Operanden sind in MATLAB wie üblich: + - * / ^
Es gelten die üblichen Vorrangsregeln, Klammern können wie gewohnt eingesetzt werden.

4.2 Mathematische Funktionen

MATLAB beinhaltet eine Vielzahl mathematischer Funktionen. *doc elfun* (von "elementary functions"), *help datafun* und *doc specfun* ("special functions") verschaffen einen Überblick. Einige wesentliche Funktionen sind:

sqrt()	Quadratwurzel
exp()	Exponentialfunktion
log()	Natürlicher Logarithmus (ln)
log2()	Logarithmus mit Basis 2 (lb)
log10()	Dekadischer Logarithmus (lg)
sin()	Sinus mit Argument in Bogenmaß
cos()	Cosinus mit Argument in Bogenmaß
tan()	Tangens mit Argument in Bogenmaß
atan()	Arcus-Tangens im Bereich $\pm\pi/2$
abs()	Betrag
sign()	Vorzeichen ("Signum")
mod(x,y)	Modulo – gibt den Rest nach einer Division an

Das Argument in Klammern kann ein Skalar, ein Vektor oder eine Matrix sein. Im Falle eines Vektors oder einer Matrix werden diese Funktionen elementweise berechnet! Sie werden später noch die entsprechenden Funktionen kennenlernen, die nicht elementweise, sondern gemäß der Matrizenalgebra vorgehen.

Beispiele:
```
>> sin(0)
ans =
     0
>> sin([pi/4 pi/2 pi])   % mit Vektor als Argument
ans =
    0.7071    1.0000    0.0000
```

Vergleichen Sie die Ergebnisse: sin(0) und sin(pi) betragen beide Null – allerdings mit unterschiedlicher Rundung, da der Ausdruck *pi* nur eine Gleitkomma-Näherung des exakten Wertes von π darstellt.
Wird wie im obigen Beispiel nur der Ausdruck ohne eine Variable eingegeben, dann schreibt MATLAB das Ergebnis immer in die Systemvariable *ans* ("answer"). Diese wird bei nochmaliger Verwendung überschrieben. Den aktuellen Wert sehen Sie im Workspace.
Veranschaulichung des elementweisen Operierens, mit und ohne Variable:
```
>> mod([1 2 3; 4 5 6], 3)
ans =
     1     2     0
     1     2     0
>> x = pi*[0 1/6 1/4 1/3 1/2 1]; sin(x), sqrt(x)
ans =
         0    0.5000    0.7071    0.8660    1.0000    0.0000
ans =
         0    0.7236    0.8862    1.0233    1.2533    1.7725
```

4.3 Komplexe Zahlen

Im Grunde genommen rechnet MATLAB ausschließlich auf Basis komplexer Zahlen. Die Zahl 3 ist somit eine komplexe Zahl mit Imaginärteil 0. Real- und Imaginärteil werden getrennt gespeichert, weshalb eine "echte" komplexe Zahl den doppelten Speicherbedarf beansprucht. Geben Sie eine beliebige komplexe Zahl ein und vergewissern Sie sich im Workspace über den Speicherbedarf. Die imaginäre Einheit kann mit i oder mit j angegeben werden.

Einige komplexe Funktionen, die ebenfalls unter *elfun* aufgenommen sind:

real()	Realteil einer komplexen Zahl
imag()	Imaginärteil einer komplexen Zahl
conj()	Konjugiert komplexe Zahl
angle()	Phasenwinkel einer komplexen Zahl
abs()	Betrag, auch einer komplexen Zahl

Mit *lookfor complex* finden Sie zahlreiche weitere Funktionen zum Thema komplexe Zahlen.

Beispiele:

```
>> abs([-4 3+4i -j])
ans =
     4     5     1
>> conj(-j)
ans =
        0 + 1.0000i
```

4.4 Rundungsfunktionen

floor()	Rundet zur nächsten ganzen Zahl ab (= Runden nach –inf)
ceil()	Rundet zur nächsten ganzen Zahl auf (= Runden nach +inf)
round()	Rundet zur nächsten ganzen Zahl auf bzw. ab
fix()	Gibt nur den ganzzahligen Teil wider ohne zu Runden (= Runden nach Null)

Überlegen Sie sich anhand der Zahlen -2.8, -2.5, -2.3, 2.3, 2.5, 2.8 und 3.2+5,5i die Ergebnisse dieser vier Funktionen und vergleichen Sie dann mit der unten stehenden Lösung.

Lösung:

```
>> x = [-2.8 -2.5 -2.3 2.3 2.5 2.8];
>> round(x)
ans =
    -3    -3    -2     2     3     3
>> floor(x)
ans =
    -3    -3    -3     2     2     2
>> ceil(x)
ans =
    -2    -2    -2     3     3     3
>> fix(x)
ans =
    -2    -2    -2     2     2     2
>> round(3.2 + 5.5i))
ans =
    3.0000 + 6.0000i
```

4.5 Sonstige Kommandos

cd Verzeichnis, cd ..	wechselt das Verzeichnis (mit Leerzeichen/blank nach *cd!)*
cd oder *pwd*	Zeigt den Verzeichnispfad
dir oder *ls*	Zeigt das Directory
delete Dateiname	Löscht die Datei *Dateiname*
load	Daten laden
save	Daten speichern

Beispiel:
Erzeugen Sie eine beliebige Matrix A und speichern diese als ASCII-Datei mit dem Namen data.out (die Dateiendung *.out* ist beliebig wählbar):

```
>> save data.out A -ASCII
```

Im Current Directory erscheint jetzt die Datei data.out. Durch Doppelklick hierauf oder durch Eingabe von *edit data.out* wird sie im Editor geöffnet. Während einer späteren MATLAB-Sitzung möchten Sie wieder auf die Daten von A zugreifen. Sie können dies z. B. tun mit:

```
>> A = load data.out        - oder -
>> load data.out
```

Beobachten Sie dabei den Workspace.

5. Variablen

Alle Variablen in MATLAB sind Arrays (wie auch immer geartet) mit möglicherweise komplexen Elementen und müssen nicht wie in anderen Sprachen deklariert werden. Ein Variablenname muss mit einem Buchstaben beginnen. Zahlen und Unterstriche sind erlaubt, Umlaute und alle weiteren Sonderzeichen (auch Minus -) sind nicht erlaubt. Es wird unterschieden zwischen Groß- und Kleinschreibung.
Die maximal zulässige Länge für einen Variablennamen erhalten Sie mit

```
>> namelengthmax
ans =
    63
```

Mit *isvarname('Variablenname')* finden Sie heraus, ob ein Variablenname zulässig ist. Die Antwort *1* steht für "wahr" und bedeutet "ja", *0* steht für "falsch" und bedeutet "nein".

Beachten Sie die bereits vergebenen Systemvariablen *pi, ans, inf, NaN, i, j*. Sie können zwar überschrieben werden, allerdings ist das nicht empfehlenswert.

6. Vektoren und Matrizen

In Kapitel 4 Grundlagen haben Sie Matrizen und Vektoren bereits auf einfache Weise erstellt. Es ist auch möglich, sie durch Verschachtelungen zu formulieren:

```
>> v1 = [1 2]; v2 = [5 7];

>> z = [v1 v2]
z =
     1     2     5     7

>> M = [v1; v2]
M =
     1     2
     5     7

>> M = [v1' v2']
M =
     1     5
     2     7
```

```
>> s = [v1'; v2']
s =
     1
     2
     5
     7

>> M = [v1' [3;4]]
M =
     1     3
     2     4
```

6.1 Der Doppelpunkt-Operator

Ein wichtiger Operator ist der Doppelpunkt. Er kann auf verschiedene Arten verwendet werden. Zum Bilden von Elementfolgen lautet die Syntax S*tart : Schrittweite : Ziel*. Gibt man nur zwei Werte an, dann bedeuten diese *Start : Ziel* und die Schrittweite wird +1 gesetzt. Die Resultate sind Vektoren.

```
>> 1:5
ans =
     1     2     3     4     5
>> v = 1:2:5
v =
     1     3     5
>> 0.5:-0.2:-0.6
ans =
    0.5000    0.3000    0.1000   -0.1000   -0.3000   -0.5000
>> A = [1:2:5; 7:9]
A =
     1     3     5
     7     8     9
```

Des weiteren können mit Hilfe des Doppelpunkt-Operators ganze Zeilen oder Spalten angesprochen werden. Sie haben bereits die runden Klammern hinter einer Variablen kennengelernt. Am Beispiel obiger Matrix A ergibt sich *A(2,3) = 9*. Erweitern Sie A nun um eine weitere Zeile und ziehen dann einzelne Spalten/Zeilen heraus:

```
>> A(3,:) = 6:-2:2
A =
     1     3     5
     7     8     9
     6     4     2
>> A(2,:)
ans =
     7     8     9
>> A(:,3)
ans =
     5
     9
     2
```

Bildung einer Untermatrix:

```
>> B = A([1 2], [2 3])
B =
     3     5
     8     9
>> C = A(:, [1 3])
C =
     1     5
     7     9
     6     2
```

Die zweite Zeile streicht man mit:

```
>> A(2,:)=[]
A =
     1     3     5
     6     4     2
```

Der Doppelpunkt wandelt eine Matrix auch in einen Spaltenvektor um. Beachten Sie die Reihenfolge der Elemente (das Ergebnis wurde hier in einen Zeilenvektor umgewandelt):

```
>> A(:)'
ans =
     1     6     3     4     5     2
```

end kennzeichnet das letzte Element, entweder einer Zeile, einer Spalte, oder der gesamten Matrix:

```
>> A(1,end)
ans =
     5
>> A(end,2)
ans =
     4
>> A(end)
ans =
     2
```

Ein einzelnes Element kann gezielt geändert werden:

```
>> B(2,1) = -7
B =
     3     5
    -7     9
```

Alternativ kann man im Workspace per Doppelklick auf B die Matrix bequem über den Array-Editor einsehen und bearbeiten.
Bereiche innerhalb der Reihenfolge der Elemente werden angesprochen mit:

```
>> A(2:5)
ans =
     6     3     4     5
```

Beispiel: Struktur einer 3D-Matrix: *A(Zeile, Spalte, Seite)*

```
>> D3(1,1,1) = 1
D3 =
     1

>> D3(2,2,2) = 2
D3(:,:,1) =
     1     0
     0     0
D3(:,:,2) =
     0     0
     0     2

>> D3(3,3,3) = 3
D3(:,:,1) =   % erste Seite
     1     0     0
     0     0     0
     0     0     0
D3(:,:,2) =   % zweite Seite
     0     0     0
     0     2     0
     0     0     0
D3(:,:,3) =   % dritte Seite
     0     0     0
     0     0     0
     0     0     3
```

Weitere Beispiele und Erklärungen finden Sie unter *doc colon*.

6.2 Funktionen zum Erzeugen von Vektoren – linspace und logspace

Mit der Funktion *linspace* können Vektoren erzeugt werden, indem man den Wert des ersten (a) und des letzten Elements (b), sowie die Anzahl n der Elemente vorgibt. Im Gegensatz zum Doppelpunktoperator wird hier also anstelle der Schrittweite die Anzahl der Elemente

vorgegeben. Sind a und b reelle Zahlen, dann erzeugt *x = linspace(a,b,n)* einen Zeilenvektor x der Länge n mit den Elementen

$x_k = a + (b-a) \cdot \frac{k-1}{n-1}$ wobei k = 1,… n ist.

Gibt man nur die ersten beiden Argumente a und b ein, dann ist n = 100.

```
>> x = linspace(1,2,3)
x =
    1.0000    1.5000    2.0000
```

logspace(a,b,n) funktioniert identisch zu *linspace*, erzeugt aber Elemente in logarithmischem Abstand. a und b sind hier Zehnerpotenzen, d.h. a = 0 liefert $10^0 = 1$ als Startwert. Ohne die Angabe von n setzt *logspace* n = 50.

```
>> logspace(0,3,4)
ans =
           1          10         100        1000
>> x = logspace(1,3,3)
x =
          10         100        1000
```

6.3 Funktionen zum Erzeugen von Matrizen

zeros()	erzeugt eine Matrix bestehend aus Nullen
ones()	erzeugt eine Matrix bestehend aus Einsen
eye()	erzeugt eine Einheitsmatrix
rand()	erzeugt eine Matrix mit gleichverteilten Zufalls-Elementen zwischen 0 und +1
diag()	erzeugt u.a. eine Diagonalmatrix

Wird bei den Befehlen *zeros, ones, eye, rand* nur eine Eingangsvariable angegeben, so ist das Ergebnis eine quadratische Matrix.

```
>> ones(2)
ans =
     1     1
     1     1
>> ones(2,3)
ans =
     1     1     1
     1     1     1
>> zeros(1,3)
ans =
     0     0     0
>> eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

Eine gleichverteilte 5x5-Zufallsmatrix im Bereich [-1,+1] erhält man mit: `>> rand(5)*2-1`

Beispiel: Erzeugen Sie eine symmetrische 4x4-Matrix bestehend aus ganzzahligen Zufallselementen im Bereich [0,10]!

Lösung:

```
>> S = round(5*rand(4)); S = S+S'
S =
     6     4     4     8
     4    10     5     7
     4     5     6     5
     8     7     5     0
```

diag verfügt über mehrere Funktionen. Wird nur eine Eingangsvariable angegeben, so wird immer die Hauptdiagonale angesprochen. Bildung einer Diagonalmatrix aus einem Vektor:

```
>> diag([1 2 3])
ans =
     1     0     0
     0     2     0
     0     0     3
```

Isolieren der Hauptdiagonalen obiger Zufallsmatrix S in einen (Spalten-)Vektor:

```
>> diag(S)
ans =
    6
   10
    6
    0
```

Über eine zweite Eingangsvariable lassen sich die Nebendiagonalen mit *diag(A,x)* ansprechen, wobei positives x die oberen Nebendiagonalen anspricht, negatives x die unteren:

```
>> diag(S,1)'
ans =
     4     5     5
>> diag(S,-2)'
ans =
     4     7
>> diag([3 4 5],-1)   % ebenso: diag(3:5,-1)
ans =
     0     0     0     0
     3     0     0     0
     0     4     0     0
     0     0     5     0
```

6.4 Matrizenfunktionen

Viele der folgenden Befehle haben in Abhängigkeit ihrer Anwendung sowie der Anzahl der Eingangs- und Ausgangsvariablen verschiedene Funktionen, die in den entsprechenden Hilfeseiten bei Bedarf nachlesbar sind.

inv(A)	Inverse Matrix, identisch mit *A^-1*
det()	Determinante
eig()	Eigenwerte, Eigenvektoren
rank()	Rang einer Matrix
size()	Matrixdimension in Zeilen, Spalten
length()	größter Wert aus Spalten-, Zeilenindex, entspricht *max(size())*
sum()	spalten-/zeilenweise Summe der Elemente der Matrix
prod()	spalten-/zeilenweise Produkte der Elemente der Matrix
min()	spaltenweise kleinste Elemente der Matrix
max()	spaltenweises größte Elemente der Matrix
sort()	sortiert Elemente spaltenweise in aufsteigender Reihenfolge
find()	findet alle von Null verschiedene Elemente
norm()	Euklidische Norm eines Vektors (= seine natürliche Länge)

Darüber hinaus gibt es eine Vielzahl von Funktionen, mit deren Hilfe sich Matrizen umformen/verändern oder statistisch auswerten lassen u.v.a. Sie sind nicht Gegenstand dieses Skripts.
Die meisten MATLAB-Funktionen erkennen Art (Skalar, Matrix oder Array) und Anzahl der übergebenen Eingangsvariablen und verhalten sich dementsprechend. Das gleiche gilt für

die Ausgangsvariablen. Im Hilfeaufruf *doc Funktion* sind alle Möglichkeiten einer Funktion aufgeführt. Das folgende Beispiel soll dies anhand der Funktion *eig()* in Bezug auf die Ausgangsvariablen verdeutlichen:

```
>> A = [1 2 3;4 5 6;7 8 9];
>> eig(A)
ans =
   16.1168
   -1.1168
   -0.0000
>> [V,D] = eig(A)
V =
   -0.2320   -0.7858    0.4082
   -0.5253   -0.0868   -0.8165
   -0.8187    0.6123    0.4082
D =
   16.1168         0         0
         0   -1.1168         0
         0         0   -0.0000
```

Der erste Aufruf mit nur einer Ausgangsvariablen liefert die Eigenwerte von A – in Form eines Spaltenvektors.

Der zweite Aufruf mit zwei Ausgangsvariablen ordnet die Eigenwerte in der Hauptdiagonalen der Matrix D an und schreibt die Eigenvektoren spaltenweise in die Matrix V so dass gilt: A*V = V*D.

Ein Eigenvektor ist beispielsweise [-0.2320 -0.5253 -0.8187], der dazugehörige Eigenwert beträgt 16.1168 (spaltenweise Zuordnung!) usw.

Beispiel: Geg.: $A = \begin{bmatrix} 2 & 4 \\ 7 & 5 \end{bmatrix}$, $B = \begin{bmatrix} 4 & 2 \\ 5 & 7 \end{bmatrix}$ Ges.: $Q = \begin{bmatrix} 0_{2x2} & E_{2x2} \\ -A^{-1}B & BA^T \end{bmatrix}$

```
>> A = [2 4;7 5];
>> B = A(:,[2 1]); % Definition von B durch Vertauschen der Spalten!
>> Q = [zeros(2)  eye(2); -inv(A)*B  B*A']
Q =
         0         0    1.0000         0
         0         0         0    1.0000
   -0.0000   -1.0000   16.0000   38.0000
   -1.0000         0   38.0000   70.0000
```

6.5 Matrizen- und Array-Operationen

In der Einführung wurde der Unterschied zwischen den Rechenvorschriften der Matrizenalgebra und den elementweisen Operationen der Arrays angesprochen. Folgende fünf Operanden existieren sowohl in der Matrizenalgebra als auch jeweils mit vorangestelltem Punkt für Array-Operationen (*doc ops, doc arithmeticoperators*):

*	.*	Multiplikation
/	./	rechtsseitige Division
\	.\	linksseitige Division
^	.^	Potenz
'	.'	Transposition

+ und – sind bereits elementweise definiert, es gibt deshalb keine .+ und .- Operatoren.

Führen Sie folgende Operationen von Hand aus und überprüfen Sie dann in MATLAB!

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $B = \begin{bmatrix} 2 & 0 \\ 1 & 5 \end{bmatrix}$

→ A+1, A+B, A-B, A*B, A.*B, A/B, A./B, A\B, A.\B, A^2, A.^2 , A', A.'

Hinweis:
X = A/B entspricht X = A*B^{-1} und ist die Lösung der Gleichung X*B = A
X = A\B entspricht X = A^{-1}*B und ist die Lösung der Gleichung A*X = B
A.\B ist gleichbedeutend mit B/A (funktioniert auch bei Skalaren: 10/2=5, 10\2=10^{-1}*2=0.2)

Lösung:

```
>> A + 1  % bereits elementweise!
ans =
     2     3
     4     5
>> A + B
ans =
     3     2
     4     9
>> A - B
ans =
    -1     2
     2    -1

>> A * B
ans =
     4    10
    10    20
>> A .* B
ans =
     2     0
     3    20

>> A / B
ans =
    0.3000    0.4000
    1.1000    0.8000
>> A ./ B
ans =
    0.5000       Inf
    3.0000    0.8000
```

```
>> A \ B
ans =
   -3.0000    5.0000
    2.5000   -2.5000
>> A .\ B
ans =
    2.0000         0
    0.3333    1.2500

>> A^2
ans =
     7    10
    15    22
>> A.^2
ans =
     1     4
     9    16

>> A'   % Erklärung unten!
ans =
     1     3
     2     4
>> A.'  % Erklärung unten!
ans =
     1     3
     2     4
```

Die Transpositionsvorschriften ' und .' sind bei reellen Matrizen identisch. Kommen komplexe Zahlen vor, dann erzeugt ' zusätzlich zur Transposition komplex konjugierte Zahlen, wohingegen .' nur transponiert. vk' ist gleichbedeutend mit *ctranspose(vk*), vk.' mit *transpose(vk)*.

```
>> vk = [1+i, 2+2i]
vk =
   1.0000 + 1.0000i   2.0000 + 2.0000i
>> vk'
ans =
   1.0000 - 1.0000i
   2.0000 - 2.0000i
>> vk.'
ans =
   1.0000 + 1.0000i
   2.0000 + 2.0000i
```

Neben den Operanden gibt es auch Befehle, die gemäß der Matrizenalgebra oder aber elementweise operieren. Sie lernten bereits Funktionen kennen, die elementweise definiert sind, z.B. *sqrt(), exp(), log()*. Äquivalente Funktionen im Sinne der Matrizenalgebra sind: *sqrtm(), expm(), logm().*

Das nachgestellte *m* zeigt an, dass es sich um Matrizenfunktionen handelt.

```
>> A = [0 1; 4 9];
>> sqrt(A)
ans =
     0     1
     2     3
>> sqrtm(A)
ans =
   0.1323 + 0.6234i   0.3117 - 0.0661i
   1.2468 - 0.2646i   2.9376 + 0.0281i

>> A.^2
ans =
     0     1
    16    81
>> A^2       % identisch mit A*A
ans =
     4     9
    36    85
```

Abschließend bleibt noch das Kreuzprodukt oder Vektorprodukt $\vec{a} \times \vec{b}$ mit der Syntax *cross(Vektor1,Vektor2)* als weitere Rechenvorschrift zu erwähnen. Beide Vektoren müssen über drei Elemente verfügen.
Rechnen Sie auch diese Vektor-Operationen zuerst von Hand und dann mit MATLAB anhand der Vektoren a = [1 2 3] und b = [4 5 6]:

```
>> a*b'  % Skalarprodukt - Ergebnis ist ein Skalar
ans =
    32
>> a'*b  % Dyadisches Produkt - Ergebnis ist eine 3x3-Matrix
ans =
     4     5     6
     8    10    12
    12    15    18
>> cross(a,b)   % Vektorprodukt
ans =
    -3     6    -3
>> cross(a',b)  % Vektorprodukt - mit Zeilen- oder Spaltenvektoren
ans =
    -3     6    -3
>> a.*b  % elementweise Multiplikation
ans =
     4    10    18
>> a.*b' % elementweise Multiplikation
??? Error using ==> times
Matrix dimensions must agree.
```

7. m-Files

Es gibt zwei Arten von m-Files: **Scripts** und **Functions** (es werden die englischen Begriffe verwendet, die auch die MATLAB-Hilfedateien verwenden). Ein neues, leeres m-File öffnet man am einfachsten mit dem ersten Icon links unterhalb der Menüzeile, oder über das Menü: *File → New → M-File.*

7.1 Scripts

Ein Script ist eine Datei mit MATLAB-Anweisungen, die die Dateiendung .m aufweist. Ein Script kann auf Daten des Workspace zugreifen und erzeugte Daten dort ablegen. Es kann ein anderes Scripts aufrufen und ausführen. Es enthält i. G. zu MATLAB-Functions (siehe nächstes Kapitel) keine Ein- und Ausgabevariablen und keinen *end*-Befehl sowie i. G. zu

anderen Programmiersprachen keinen Deklarationsanteil (wie z.B. in C: *double a, int b*). Kommentare sind zeilenweise durch vorangestelltes % zu kennzeichnen, blockweise (über mehrere Zeilen) durch *%{* am Anfang und *%}* am Ende des Blocks. Ein Zeilenumbruch innerhalb eines Ausdrucks wird wie im Command Window mit drei Punkten … erreicht:

```
1 -    x = 11+3-7+...
2 -        2;
```

Der Semikolon am Zeilenende unterbindet die Zeilenausgabe beim Programmablauf. Die Ausführung eines Scripts erfolgt durch Eingabe des Programmnamens ohne (!) Endung im Command Window, im Editor entweder durch Klicken des Icons mit dem grünen Pfeil ("*Run*") oder durch Drücken der Taste F5 (speichert und führt aus).
Zur Bearbeitung kann ein Script natürlich aus dem Menü im Editor heraus geöffnet werden, oder wiederum im Command Window mit *edit Scriptname*.

Beim Programmieren unterstützt Sie der integrierte Debugger, der aus dem Editor gestartet werden kann. Dieser umfaßt u.a. das Setzen von Haltepunkten / Breakpoints durch Klick auf die Spiegelstriche hinter den Zeilennummern oder das Icon in der Menüzeile. Die zum Zeitpunkt eines Breakpoints aktuellen Variablenwerte können während des Script-Ablaufs im Workspace eingesehen oder bequemer durch einfaches Platzieren des Mauszeigers auf der betreffenden Variablen im Editor angezeigt werden. Die Icons rechts der Breakpoint-Icons bieten verschiedene Möglichkeiten zum weiteren Abarbeiten des Programms an (*Step, Step in, Step out, Exit Debug Mode*).

Der Befehl *echo on / echo off* zeigt alle abgearbeiteten Befehle eines Scripts während dessen Ausführung. Er kann nützlich sein bei der Fehlersuche oder zu Zwecken der Demonstration und entweder im Programmcode selbst oder im Command Window eingegeben werden.

7.2 Functions

Die zweite Art eines m-Files ist die sog. Function (*doc function).* Eine Function ist eine Unterprogrammstruktur in MATLAB analog zur sog. Funktion in C/C++, die nicht mit der mathematischen Funktion zu verwechseln ist. Zur besseren Unterscheidung wird für eine MATLAB-Funktion in diesem Skript der englische Begriff Function benutzt (siehe auch *doc function)*. Sämtliche Befehle, die Sie bisher kennengelernt haben, sind solche Functions, die bereits in der MATLAB-Software integriert sind. Sie können diesen (wenn auch umfangreichen) Grundstock jederzeit mit selbst geschriebenen Functions ergänzen und dadurch die Funktionalität von MATLAB erweitern. Außerdem finden Sie unter http://www.mathworks.com/matlabcentral/ viele weitere Functions bzw. allgemein m-Files von anderen Usern, die zum Download bereit stehen.

Damit ein m-File eine Function ist, muss es mit dem Schlüsselwort *function* beginnen, dann folgen die Ausgangsvariablen (in eckigen Klammern), der Name der Function und schließlich die Eingangsvariablen (in runden Klammern). Die Kopfzeile einer Function hat somit den Aufbau:

```
function [Aus1, Aus2, … ] = functionname (Ein1, Ein2, … )
```

Die Anzahl der Ein- und Ausgangsvariablen kann unabhängig voneinander Null bis beliebig viele betragen, und ist sogar jeweils variabel: Mithilfe des Befehls *nargin* kann man eine Function so programmieren, dass sie die Anzahl der Eingangsvariablen selbständig ermittelt und daraufhin die Anzahl der Ausgangsvariablen mit *nargout* festlegt (siehe *doc nargin*). Gibt es weder Ein- noch Ausgangsvariablen, dann lautet die Kopfzeile:

```
function[] = functionname
```

Der Name ist frei wählbar, muss allerdings identisch sein zum Dateinamen, unter welchem die Function gespeichert wird (ebenfalls mit Dateiendung .m). Ferner gelten bezüglich der Namenswahl die gleichen Regeln wie für Variablennamen (siehe Kapitel 5: Variablen).

Es gibt drei Arten von Variablen: **lokale, globale und persistente**. Bis auf die Ausgangsvariablen sind alle vorkommenden Variablen in Functions lokale Variablen und existieren nicht im Workspace. Functions haben demnach ihren eigenen, nach außen hin nicht sichtbaren Workspace. Zusätzlicher Variablenaustausch kann erfolgen durch die Deklaration:

```
global Variable1 Variable 2 …
```

Diese Deklaration muss auch in demjenigen Script stehen, von welchem aus die Function aufgerufen wird. Dadurch können an die Function zusätzlich zu den Ein- und Ausgangsgangsvariablen noch mehr Werte übergeben, verarbeitet und verändert werden. Persistente Variablen werden deklariert durch:

```
persistent Variable1 Variable2 …
```

Persistente Variablen sind ebenso wie die lokalen Variablen nur in der Function bekannt, in der sie deklariert werden – mit dem Unterschied, dass sie bis zum nächsten Aufruf derselben Function fortbestehen. Sie sind anderen m-Files nicht zugänglich und werden nur gelöscht, wenn die Function mit *clear Functionname* aus dem Speicher entfernt wird (siehe auch *doc variables, doc persistent, doc global*).
MATLAB-Functions können auch am Ende eines Scripts definiert werden.

7.3 Die Struktur von m-Files

Beide Arten von m-Files, Scripts wie Functions, sollten gut dokumentiert sein. Folgende Struktur ist anzustreben:

```
function [Aus1, Aus2, ...] = Funktionsname (Ein1, Ein2, ...)
% H1-Zeile
% Help Text
% Help Text usw.
Anweisungen
```

Im Falle eines Scripts entfällt natürlich die erste Zeile. Die zweite Zeile ist die sogenannte H1-Zeile. In ihr wird in einer Zeile das Programm beschrieben. Der Befehl *lookfor* sucht nur in dieser H1-Zeile und gibt nur diese aus. Sie sollte deshalb kurz und prägnant sein. In anschließenden Kommentarzeilen können Sie das m-File weiter dokumentieren.

Übrigens können Sie z. B. die Function *linspace* mit *edit linspace* im Editor wie ein selbst geschriebenes m-File öffnen und den Programmcode sowie die Erklärungen in den Kommentarzeilen einsehen, die ja auch beim Aufruf der Hilfe erscheinen.

Beispiel:
Definieren Sie zwei Vektoren zur Bildung des Vektorprodukts. Öffnen Sie die Function cross() mit *edit cross.* Setzen Sie im Editor an sinnvollen Stellen Breakpoints und starten die Function aus dem Command Window mit *cross(a,b).* Nun können Sie den Abarbeitungsprozess von cross() im Editor und anhand der aktuellen Variablenwerte mitverfolgen (siehe Editor oder Workspace). Ein einfacheres Beispiel erhalten Sie mit *edit linspace.*

8. Weitere Datentypen

Neben den bereits bekannten numerischen Datentypen wie single, double, int8 etc. gibt es noch nicht-numerische Typen. Den Datentyp einer Variablen erhält man mit *class().* Jeder Datentyp besitzt ein eigenes grafisches Icon im Workspace.

8.1 Zeichenketten

Zeichenketten, sog. Strings, sind in MATLAB vektoriell angeordnete einzelne character-Elemente. Ein character-Element (*class = char)* belegt 2 Byte Speicherplatz. Ihm liegt die ASCII-Tabelle zugrunde. Strings werden in Hochkommata eingegeben:

```
>> str = 'Das ist ein String'
str =
Das ist ein String
```

doc strfun listet Functions zu diesem Thema auf. Die Function *length()* funktioniert auch hier:

```
>> length(str)
ans =
    19
```

Beachten Sie das Symbol der Variablen *str* im Workspace.

8.2 Cell-Array (*doc cell)*

Mit Hilfe einer *Cell* können Daten unterschiedlichen Typs wie Strings, Arrays unterschiedlicher Dimension, Skalare usw. in einer Variablen verwaltet werden. Einzelne Cell-Elemente werden über ihre Indizes angesprochen. Cell-Elemente werden mit geschweiften Klammern gekennzeichnet.

```
>> Z = {'Test'  3+2i; -5  (0:10)'}
Z =
       'Test'    [3.0000 + 2.0000i]
       [  -5]         [11x1 double]
```

Beachten Sie auch hier das Symbol der Cell-Variablen Z im Workspace. Analog zu einer Matrix ist die Eingabe auch über einzelne Elemente möglich:

```
>> Z(1,1) = {[1 2 3; 4 5 6; 7 8 9]};
>> Z(1,2) = {'Testmatrix = 2'};
>> Z(2,1) = {3+7i};
>> Z(2,2) = {0 : pi/100 : 2*pi};
>> Z
Z =
             [3x3 double]    'Testmatrix = 2'
    [3.0000 + 7.0000i]         [1x201 double]
>> Z(1,1)       % erste Art, ein Cell-Element aufzurufen
ans =
    [3x3 double]

>> Z{1,1}       % zweite Art, ein Cell-Element aufzurufen
ans =
     1     2     3
     4     5     6
     7     8     9

>> class(Z)
ans =
cell
```

8.3 Structure (*doc struct*)

Structures dienen ebenfalls der Verwaltung unterschiedlicher Datentypen. Um den Aufbau der Structure leichter zu verstehen, können Sie die Structure-Variable *Bsp* (siehe unten) im Array Editor öffnen. Dort erhalten Sie Einblick in den Inhalt einzelner Elemente mit Doppelklick und können dann wieder Ebene für Ebene nach oben gehen.

```
>> A=[1 2;3 4];
>> Bsp = struct('Bezeichnung', 'Teil 1', 'Daten', A)
Bsp =
    Bezeichnung: 'Teil 1'
          Daten: [2x2 double]
>> class(Bsp)
ans =
struct
```

Analog zu einer Cell kann auch eine Structure über einzelne Elemente definiert werden. Das Ergebnis ist identisch mit obiger Eingabe:

```
>> Bsp.Bezeichnung = 'Teil 1';
>> Bsp.Daten = A
Bsp =
    Bezeichnung: 'Teil 1'
          Daten: [2x2 double]
```

Die Structure wird erweitert zu einer 2x1-Structure:

```
>> Bsp(2,1).Bezeichnung = 'Teil 2'
Bsp =
2x1 struct array with fields:
    Bezeichnung
    Daten
>> Bsp(2,1).Daten = inv(A);
```

Aufrufen der gespeicherten Informationen:

```
>> Bsp.Bezeichnung
ans =
Teil 1
ans =
Teil 2
>> Bsp(2).Daten
ans =
   -2.0000    1.0000
    1.5000   -0.5000
>> Bsp(2).Daten(2,2)
ans =
   -0.5000
```

Beachten Sie wieder das Symbol der Structure-Variablen *Bsp* im Workspace.

8.4 Function Handle

Function Handles werden durch den @-Operator definiert. Sie werden im Umgang mit den sog. *Anonymous Functions* benötigt. Eine Anonymous Function ermöglicht es, eine Function nicht nur als m-File zu erstellen, sondern sie direkt im Command Window einzugeben. Anonymous Functions werden erstellt mit der Syntax:
Name des Function Handle = @ (Eingangsvariablen) Funktionsausdruck

Beispiel:

```
>> fu = @(x) x.^2;
>> fu(5)
ans =
    25
```

```
>> class(fu)
ans =
function_handle
```

Enthält die Anonymous Function zusätzliche Variablen, so müssen diese bereits existieren, wenn sie erstellt wird:

```
>> a = 2; b = 6; c = 30;
>> parabel = @(x) a*x.^2 + b*x + c;
>> parabel(3)
ans =
    66
```

Achtung: Werden die Parameter a,b und c danach geändert, so benutzt *parabel()* die neuen Werte erst, wenn sie danach nochmals definiert wird mit *>> parabel = @(x) ....*
Eine Anonymous Function kann Null bis beliebig viele Eingangsvariablen besitzen. Sie ist dann dementsprechend aufzurufen: *>> bsp() >> bsp(3/2*pi) >> bsp(3,2,6)*

Sie können den Function Handle auch als eine Art Abkürzung einer m-File-Function (Kapitel 7.2) verwenden und diese indirekt über den Function Handle aufrufen. Die Syntax hierfür lautet:
Name des Function Handle = @ Functionname

Beispiel:

```
>> A = [1 2 3;4 5 6;7 8 9];
>> d = @diag;
>> d(d(A))  % verschachtelter Aufruf = diag(diag(A))
ans =
     1     0     0
     0     5     0
     0     0     9
```

Beachten Sie das Symbol eines Function Handle im Workspace. Im Help Browser erhalten Sie mit den Suchbegriffen *Anonymous Functions* und *Function Handle* weitere Informationen.

9. Vergleichsoperatoren und logische Operatoren *(doc ops, doc relop, doc logical)*

- Vergleichsoperatoren:
  == gleich, ~= ungleich, < kleiner, <= kleiner gleich, > größer, >= größer gleich

- Logische Operatoren:
  & oder *and*, | oder *or*, ~ oder *not*, *xor()* für Exklusiv-Oder

Logisch falsch wird durch 0, logisch wahr durch 1 oder einen Wert ungleich Null ausgedrückt. Wie Sie gleich sehen werden, erhalten Wahrheitswerte im Workspace eigene Symbole. Matrizen und Vektoren werden bei allen Operatoren elementweise verknüpft.

```
>> rand(2,3) >= rand(2,3)
ans =
     1     0     0
     0     1     0

>> 0.5 < rand(2,3)
ans =
     1     0     1
     0     0     0
```

```
>> a = [0 0 1 1], b = [0 1 0 1]
a =
     0     0     1     1
b =
     0     1     0     1
>> Und = a&b, Oder = a|b, XOR = xor(a,b), a_negiert = ~a
Und =
     0     0     0     1
Oder =
     0     1     1     1
XOR =
     0     1     1     0
a_negiert =
     1     1     0     0
```

Höchste Priorität besitzt in MATLAB *not*, gefolgt von *and* und schließlich *or*. Klammern beeinflussen natürlich auch hier die Vorrangsregeln:

```
>> ~a & (a|b)
ans =
     0     1     0     0
```

Die Operatoren && und || werden als Kurzschlußoperatoren (short-circuit operators) bezeichnet und haben die gleiche Funktion wie & bzw. |, allerdings werden bei ihnen Skalare nicht weiter evaluiert als zur Ermittlung des Wahrheitswertes erforderlich. (Für dieses Beispiel müssen Sie vorher die Warn-Funktion aktivieren mit `>> warning on MATLAB:divideByZero`. Ausschalten: `>> warning off MATLAB:divideByZero`)

```
>> a = 5; b = 0;
>> (b ~= 0) && (a/b > 18.5)
ans =
     0
>> (b ~= 0) & (a/b > 18.5)
Warning: Divide by zero.
ans =
     0
```

Es wird deutlich, dass im ersten Fall nur der Ausdruck b ~= 0 auf seinen Wahrheitswert untersucht wird, da das Ergebnis dann bereits feststeht, im zweiten aber beide Ausdrücke.

10. Ablaufstrukturen

Programmiersprachen erlauben es, den Ablauf eines Programms zu steuern. Man spricht von Ablaufstrukturen. MATLAB bietet vier Möglichkeiten, den sequentiellen Ablauf durch Verzweigungen und Schleifen zu ändern:

- FOR-Schleifen
- WHILE-Schleifen
- Verzweigungen mit IF
- Verzweigungen mit SWITCH

Ablaufstrukturen werden in m-Files (Scripts und Functions*)* benützt, können aber ebenso im Command Window eingegeben werden. Vorsicht ist geboten bei Verwendung des Buchstabens i als beliebte Zählervariable in Schleifen. Nach Beenden der Schleife bleibt er mit einem Zahlenwert im Workspace hinterlegt und erzeugt dann Komplikationen im Umgang mit komplexen Zahlen.

10.1 FOR-Schleife *(doc for)*

FOR-Schleifen sind Zählschleifen, bei denen die Zählervariable x bei *Startwert* beginnt, bei jedem Durchlauf um *Schrittweite* erhöht wird und beim Erreichen von *Endwert* die Schleife verlassen wird. Bei jedem Schleifendurchlauf werden eine oder mehrere Anweisungen durchlaufen. Die Syntax lautet:

- `for x =` *Startwert:Endwert,* *Anweisungen,* `end`
- `for x =` *Startwert:Schrittweite:Endwert,* *Anweisungen*, `end`

Die Formulierung wie oben mit Kommata wird eher im Command Window angewendet. Im Editor ersetzt man diese aus Gründen der Lesbarkeit durch einen Zeilenumbruch.

Beispiel:
```
>> for t = 0:0.5:2, t, end
t =
     0
t =
    0.5000
t =
     1
t =
    1.5000
t =
     2
```

Anstelle von *Startwert:(Schrittweite:)Endwert* können in MATLAB auch beliebige Vektoren und sogar Matrizen eingesetzt werden. Matrizen werden dann spaltenweise abgearbeitet:

```
>> sum = 0;
>> for s = [1 4 1+2i 78 -3.5]
sum = sum + s
end
sum =
     1
sum =
     5
sum =
   6.0000 + 2.0000i
sum =
  84.0000 + 2.0000i
sum =
  80.5000 + 2.0000i
```

```
>> M = [1 2 3; 4 5 6; 7 8 9];
>> for v = M, v, end
v =
     1
     4
     7
v =
     2
     5
     8
v =
     3
     6
     9
```

10.2 WHILE-Schleife *(doc while)*

Bei einer WHILE-Schleife wird vor jedem Schleifendurchlauf der *Ausdruck* auf seinen Wahrheitsgehalt überprüft. Die *Anweisungen* werden nur ausgeführt, wenn dieser wahr ist.

- `while` *Ausdruck, Anweisungen,* `end`

Beispiel: Programm zur Berechnung der Systemvariablen *eps*, vgl. Eingabe von *>> eps*
```
1 -   x = 1;
2 -   while (1+x) > 1
3 -       x = x / 2;
4 -   end
5 -   x = x * 2
```

Wie in Kapitel 9 erläutert, gilt jeder numerische Wert außer Null als wahr, so auch in der WHILE-Bedingung. Ebenso ist *Inf* wahr. *[ ]* wird als falsch gewertet, *NaN* und komplexe Zahlen führen zu einer Fehlermeldung. Im Gegensatz zu anderen Programmiersprachen ist das Ergebnis der logischen Überprüfung immer eine Matrix - solange alle Elemente dieser Matrix ungleich Null sind, wird die Schleife durchlaufen. Damit ist z. B. folgende Schleife möglich:

```
>> A = [1 2; 3 4];
>> while A, A(2) = A(2)-1, end
A =
     1     2
     2     4
A =
     1     2
     1     4
A =
     1     2
     0     4
```

10.3 IF-Verzweigung *(doc if)*

In einer IF-Verzweigung werden die *Anweisungen* nur dann ausgeführt, wenn die logische Auswertung von *Ausdruck* das Resultat "wahr" liefert. Die Grundform lautet:

- `if` *Ausdruck, Anweisungen,* `end`

Beispiel:

```
>> if 1 < 2, disp Wahr!, end
Wahr!
```

Mit *elseif* können noch weitere Ausdrücke auf ihren Wahrheitsgehalt geprüft werden. Sobald ein Ausdruck das Resultat "wahr" liefert, werden die zugehörigen Anweisungen ausgeführt - die anderen Ausdrücke werden dann allerdings nicht mehr überprüft! Eine zusätzliche *else*-Konstruktion am Ende (rechtes Beispiel) ist fakultativ:

- `if` *Ausdruck1*
  Anweisungen1
  `elseif` *Ausdruck2*
  Anweisungen2
  `end`

- `if` *Ausdruck1*
  Anweisungen1
  `elseif` *Ausdruck2*
  Anweisungen2
  `else`
  Anweisungen3
  `end`

Das folgende Beispiel zeigt, dass die Reihenfolge innerhalb der Verzweigung wesentlich ist. Denken Sie das folgende Beispiel für den Fall durch, es würde zuerst *x < 4* abgefragt, und danach *x < 5*. Ist das Ergebnis identisch?

```
1 -  x = 3;
2 -  if x < 5
3 -      disp ('kleiner 5')
4 -  elseif x < 4
5 -      disp ('kleiner 4')
6 -  else
7 -      disp ('groesser gleich 5')
8 -  end
```

Programmausgabe: `kleiner 5`

10.4 SWITCH-Verzweigung *(doc switch)*

Mit der SWITCH-Verzweigung lassen sich Fallunterscheidungen realisieren. Der *switch-Ausdruck* kann sich auf ein Skalar oder eine Zeichenkette beziehen (hier sind keine Matrizen erlaubt). Mehrere Fälle innerhalb eines *case* kann man zu Cells zusammenfassen. Falls ein Cell-Element die Auswahlbedingung erfüllt, so gilt der zugehörige *case* als erfüllt.
Ähnlich wie bei der *if-elseif-else-end*-Verzweigung gibt es auch bei SWITCH die Möglichkeit, mit *otherwise* Anweisungen anzugeben, die dann auszuführen sind, wenn alle anderen Bedingungen zuvor nicht zugetroffen haben. *otherwise* muss ebenso wie *else* als letzte Möglichkeit aufgeführt sein.
Mehrere Fälle kann man zu Cells zusammenfassen. Erfüllt ein einziges Cell-Element die Auswahlbedingung, so gilt der zugehörige *case* als erfüllt.

- `switch` *switch-Ausdruck*
  `case` *case-Ausdruck*
  Anweisungen
  `case` *{ case-Ausdruck1, case-Ausdruck2, case-Ausdruck3, ... }*
  Anweisungen
  `(otherwise`
  Anweisungen)
  `end`

Beispiel:

```
 1 -  switch x
 2 -      case -1
 3 -          disp ('x ist -1');
 4 -      case  {4, 5, 6, 7}
 5 -          disp ('x liegt im Intervall [4,7]')
 6 -      case 2+i
 7 -          disp('x ist 2+i')
 8 -      case 'Text'
 9 -          disp('x ist ein string')
10 -      otherwise
11 -          disp('x ist ein anderer Wert')
12 -  end
```

```
>> x = 'Text';
x ist ein string
>> x = 5;
x liegt im Intervall [4,7]
>> x = 100;
x ist ein anderer Wert
```

10.5 BREAK *(doc break)*

Break erzeugt den vorzeitigen Schleifenabbruch innerhalb von FOR- und WHILE-Schleifen. Bei verschachtelten Schleifen gilt der Abbruch nur für diejenige Schleife, in der der *break*-Befehl steht, und nicht für die nachfolgenden, siehe folgendes Programm:

```
1 -   clc, clear A
2 -   for i = 1:10
3 -       for j = 1:10
4 -           A(i,j) = i+j-1
5 -           if A(i,j) == 13
6 -               break
7 -           end
8 -       end
9 -   end
```

Scrollen Sie das Command Window nach durchlaufenem Programm nach oben, um u.a. zu sehen, dass MATLAB beim Beginn einer neuen Zeile (also A(2,1), A(3,1) usw.) die restlichen Elemente dieser Zeile immer mit Nullen auffüllt.

11. Ein- und Ausgabe

disp(** String **)	Textausgabe
disp(** Variable **)	Datenausgabe, unformatiert
fprintf(** Format, Variable **)	Datenausgabe, formatiert, in File oder an serielle Schnittstelle
sprintf(** Format, Variable **)	Ausgabe als String oder Char, formatiert
num2str(** Variable **)	Konvertiert Variable mit reellen Zahlen in einen String
int2str(** Variable **)	Konvertiert Variable mit Integer-Zahlen in einen String
input(** Variable **)	Einlesen einer Variablen
input(** String**,'s')	Einlesen eines Strings
eval(** String **)	Auswertung eines Strings
\n	Zeilenumbruch innerhalb einer Funktion
pause** / **pause(** Sekunden **)	Warten bis beliebige Taste betätigt wird / Zeit abgelaufen ist

Beispiele:

```
>> a = 5/3; disp('a')
a
>> disp(a)
    1.6667

>> fprintf('%10.2f',sqrt(5)) % rechtsbündig, Breite Ausgabefeld = 10
      2.24>>                  % Anzahl Nachkommastellen = 2
>> fprintf('%-10.2f',sqrt(5)) % linksbündig
2.24      >>
>> fprintf('%s%.2f','a = ', a)
a = 1.67>>

>> b = sprintf('%.2f', a)     % b ist ein String, siehe Workspace
b =
1.67
>> b = sprintf('%.2e', a)     % Ausgabe im Format e
b =
1.67e+000

>> num2str(5/3, 2)            % ans = String
ans =
1.7
>> int2str(3)                 % ans = String
ans =
3

>> v = [1 2 3]; disp(['v = ' num2str(v,'%7.1f\n')])
v = 1.0                     % ohne \n erfolgt Ausgabe in einer Zeile
    2.0
    3.0
>> disp(['v = ' sprintf('%-3.0f',v)])
v = 1  2  3

>> age = input('Alter : ')      % Einlesen einer Variablen
Alter : 22
age =
    22
>> name = input('Name : ', 's') % Einlesen eines Strings
Name : Hans
name =
Hans
```

```
>> str = 'sin(x)'; x = pi/3; eval(str)
ans =
    0.8660

>> pause
>> pause(2)

>> 'a'*1,  'A'*1,  'a'-'A'      % Rechnen mit dem ASCII-Zeichencode
ans =
    97
ans =
    65
ans =
    32
```

12. 2D-Grafik

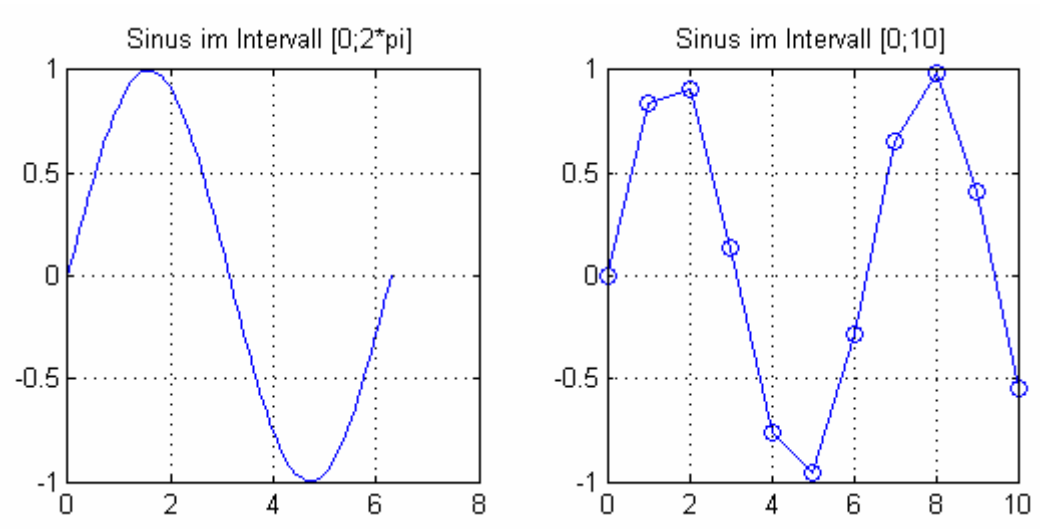
Die folgenden Beispiele zeigen das Grundprinzip einer Grafik in MATLAB. Wir wollen den Graphen der Sinus-Funktion darstellen. Dazu erzeugen wir zuerst einen beliebigen Vektor für die x-Werte. Aus diesen errechnen wir die y-Werte gemäß y = sin(x) und verbinden dann mit dem Befehl *plot(x,y)* die Wertepaare (x_1/y_1), (x_2/y_2), ... (x_n/y_n) untereinander mittels Geraden. Das Ergebnis erscheint im Grafikfenster, dem sog. *Figure Window,* kurz: *Figure*.

```
>> x = linspace(0, 2*pi);      % erzeugt einen 1x100-Zeilenvektor
>> y = sin(x);                 % erzeugt einen 1x100-Zeilenvektor
>> plot(x,y)    % zeichnet die Elemente von y gegen diejenigen von x
```

Wir hinterlegen ein Gitter und vergeben einen Titel:

```
>> grid
>> title('Sinus im Intervall [0;2*pi]')
```

Nochmalige Eingabe von *grid* oder *grid off* blendet das Gitter wieder aus. Der folgende linke Graph stellt das Ergebnis dar. Zum Vergleich: Der rechte Graph wurde mit dem 1x11-Zeilenvektor *x = 0:10* für die x-Achse gezeichnet. Hier wird nochmals deutlich, dass MATLAB nichts anderes macht, als die einzelnen Punkte mit Geraden zu verbinden.



Grundlegende Befehle im Umgang mit 2D-Grafiken:

figure* / *figure(n)	Erzeugen / Ansprechen einer Figure (mit der Id.-Nr. n)
gcf	**g**et **c**urrent **f**igure handle - Id.-Nr. der aktuellen *Figure*
gca	**g**et **c**urrent **a**xis handle - Id.-Nr. des aktuellen KoSys
clf	**cl**ear current **f**igure – Inhalt der aktuellen Figure löschen
cla	**cl**ear current **a**xes - Inhalt des aktuellen KoSys löschen
close, close (n), close all	schließt Figure (aktuelle / n-te / alle)
shg	**sh**ow **g**raph window - bringt Figure in den Vordergrund
plot	Zeichenbefehl (linear)
subplot(m, n, p)	Teilt die Figure in eine mxn-Matrix und wählt Element p aus
grid, grid on / off	Gitter, Gitter an/aus
title('Überschrift')	Grafiktitel erzeugen
semilogx / semilogy	Zeichenbefehl, x- bzw. y-Achse logarithmisch
loglog	Zeichenbefehl, doppel- logarithmisch
hold on / off	Zeichnet mehrere Graphen in einer Figure übereinander
xlabel('Text') / ylabel('Text')	Beschriftung von x- bzw. y-Achse
axis([xmin xmax ymin ymax])	manuelle Achsenskalierung
drawnow	aktualisiert Figure

Ein *figure handle* ist eine Identifikationsnummer für Figures. Gibt es mehrere Figures, dann werden diese durchnummeriert beginnend mit der Zahl 1. Der handle 0 ist für das Command Window reserviert. Analog haben Koordinatensysteme (*axes*) sog. *axes handles*, allerdings sind diese Identifikationsnummern komplexer. Im Fall des obigen Beispiels liefert *gcf* den aktuellen figure handle 1, *gca* den axes handle 161.0010... (Gleitkommazahl).
Das Kommando *>> get(gcf)* oder z. B. *>> get(2)* liefert alle Einstellungen der betreffenden Figure, >> get(gca) alle Einstellungen des aktuellen Koordinatensystems und *>> get(0)* wie bereits erwähnt alle Einstellungen des Command Window. Alle diese Einstellungen sind manuell sowie über diverse Menüs manipulierbar.
Nur für weiteres Interesse: Mit *doc figure* und Klick auf *properties* erhalten Sie einen Überblick über alle möglichen Einstellungen einer Figure und deren Optionen. Diese können geändert werden mit dem *set*-Kommando, z.B. *>> set(gcf, 'Units', 'points').* Ebenso kann man einzelne Eigenschaften direkt abfragen: *>> get(gcf, 'Units').* Sie können diese *properties* auch in einer Variablen abspeichern: *>> prop1 = get(1).* Im Workspace sehen Sie, dass es sich hierbei um eine Structure handelt, die Sie mit Doppelklick komfortabel einsehen und bearbeiten können.

Beispiel:
Es liegt ein Meßergebnis mit 50 Zufalls-Meßwerten vor. Die Meßwerte sollen sortiert werden und über ihrem Index (also 1 bis 50) aufgetragen werden.

```
>> Erg = sort(rand(1,50));
>> plot(Erg)  % ohne zweite Eingangsvariable benutzt den Index!
```

Beispiel:
Erzeugen Sie mit einem einzigen plot-Befehl die Graphen der Funktionsterme sin(kx) über dem Intervall [0; 2π] für k = 1 : 5.

```
 1      % Methode 1:
 2 -    x = linspace(0,2*pi);
 3 -    plot(x,sin(x),x,sin(2*x),x,sin(3*x),x,sin(4*x),x,sin(5*x))
 4
 5      % Methode 2:
 6 -    figure   % öffnet ein neues Figure Window
 7 -    x = linspace(0,2*pi);
 8 -    y = [sin(x);sin(2*x);sin(3*x);sin(4*x);sin(5*x)];
 9 -    plot(x,y)
10
```

```
11      % Methode 3:
12 -    figure   % öffnet ein neues Figure Window
13 -    x = linspace(0,2*pi);
14 -    plot(x,sin(x))
15 -    hold on
16 -    for k=2:5
17 -    plot(x,sin(k*x))
18 -    end
19 -    hold off
```

Die Linien der Graphen sind vielfach veränderbar in Farbe, Dicke, Linienart u.v.m. Man kann dies manuell durch Befehle oder interaktiv über die *Plot Tools* erreichen. Darüber hinaus gibt es eine Vielzahl an weiteren Grafikoptionen. Unter *doc plot* und die entsprechenden Verweise dort finden Sie viele weiterführende Informationen.

13. Sonstiges

tic … toc	Start und Ende einer Zeitmessung → Zeitmessung in m-Files
cputime	gibt die CPU-Zeit in Sekunden aus → Zeitmessung in m-Files
bench	führt einen Benchmark-Vergleich auf Ihrem Rechner aus
diary on / off	speichert Ihre Eingaben im Command Window in einer Datei
datestr(now)	liefert aktuelles Datum und Uhrzeit in einem String

Siehe auch *doc tic, doc cputime, doc bench, doc diary, doc datestr.*

AUFGABEN

Kapitel 4.2

Aufgabe 1: Berechnen Sie den Ausdruck $2^2 + \ln\pi \cdot \sin(0.75 \cdot \frac{\pi}{2}) + \sqrt{e^{\frac{2}{3}\pi}}$

Kapitel 4.3

Aufgabe 2: Geg.: $C = \begin{bmatrix} -1+i & -5i \\ 7 & 3-4i \end{bmatrix}$

2.1 Bilden Sie eine Matrix mit den Realwerten von C.
2.2 Bilden Sie einen Spaltenvektor, der aus den Imaginärteilen der zweiten Zeile von C besteht.
2.3 Die Matrix C2 soll aus den konjugiert komplexen Elementen von C bestehen.
2.4 Berechnen Sie Betrag und Phasenwinkel des Elements in der ersten Zeile und zweiten Spalte von C.

Kapitel 6

Aufgabe 3

3.1 Erzeugen Sie eine 3x3-Matrix, deren Zeilen alle aus dem Zeilenvektor [1 2 3] bestehen!
3.2 Fügen Sie den Vektor u = [4 4 4] als letzte Spalte an A an.
3.3 Fügen Sie nun den Vektor v = [1 2 3 4] als letzte Zeile an A an.
3.4 Vertauschen Sie das erste mit dem vierten Element von v, indem Sie die Reihenfolge der Elemente innerhalb von v neu definieren.

Aufgabe 4: Geg.: $A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 1 \end{bmatrix}$, $B = \begin{bmatrix} -1 & 2 \\ 4 & -2 \\ 7 & -1 \end{bmatrix}$, $C = \begin{bmatrix} 1 & 5 \\ -5 & 3 \end{bmatrix}$, $D = \begin{bmatrix} 4 & 3 & -2 \\ 1 & 0 & 5 \\ 2 & -1 & 6 \end{bmatrix}$

Führen Sie die folgenden Matrixoperationen zuerst von Hand aus und überprüfen Sie die Ergebnisse dann in MATLAB. Überlegen Sie sich, warum manche nicht definiert sind.

A+B, B+C, DA, 2A-3B, A^T, C^2, det(A), det(C), det(D)

Aufgabe 5

5.1 Erzeugen Sie eine 4x4-Matrix mit Namen "Comp", deren Hauptdiagonalelemente alle den Wert 2-i haben. Alle anderen Elemente sollen 0 sein.
5.2 Weisen Sie dem Element der 3. Zeile und 1. Spalte den Wert 3+4i zu.
5.3 Bilden Sie mit Comp2 die transponierte Matrix von Comp (vgl. Aufgabe 12).
5.4 Bilden Sie mit Comp3 die komplex konjugierte Matrix von Comp2.

Aufgabe 6

Ermitteln Sie mit Hilfe einer Function die Dimension der Matrix Z = *zeros(4,3)* und erzeugen Sie unter Verwendung des Ergebnisses eine Einheitsmatrix der gleichen Dimension.

Aufgabe 7

7.1 Erzeugen Sie mit dem Befehl *A = magic(10)* eine quadratische Matrix. Überlegen Sie sich die folgenden Ausdrücke und überprüfen Sie sie dann in MATLAB:

```
A(:,2), A(3,:)
A(2:8,3:7), A(2:2:8,3:2:7), A(5:8, [4 9])
A(1:3,1:3), A(3:-1:1,3:-1:1)
A([1,5]), A(1:5)
```

7.2 Isolieren Sie die Hauptdiagonale von A in einen Zeilenvektor.
7.3 Erstellen Sie eine Diagonalmatrix, in der nur die Hauptdiagonale von A existiert, alle anderen Elemente aber Nullen sind.
7.4 Isolieren Sie die erste Nebendiagonale unterhalb der Hauptdiagonalen von A in einen Zeilenvektor.
7.5 Erstellen Sie eine Diagonalmatrix, deren Hauptdiagonale aus den Elementen der zweiten Nebendiagonalen oberhalb der Hauptdiagonalen von A besteht (alle anderen Elemente sind Nullen).
7.6 Berechnen Sie die Summe der Elemente je Spalte von A.
7.7 Berechnen Sie die Summe der Elemente je Zeile von A.
7.8 Berechnen Sie die Summe der Elemente in der Hauptdiagonalen von A.
7.9 Berechnen Sie die Summe aller Elemente in A.
Wie Sie sehen, erzeugt der Befehl *magic()* eine ganz spezielle Matrix (siehe *doc magic,* diese Function ist aber für die Vorlesung nicht relevant).
7.11 Ändern Sie A derart, dass die Elemente spaltenweise sortiert sind.
7.12 Ermitteln Sie spaltenweise die minimalen und die maximalen Werte.
7.13 Ermitteln Sie das Element mit dem minimalen und dem maximalen Wert der gesamten Matrix.
7.14 Geben sie dem letzten Element der fünften Zeile von A den Wert sin(π/3).
7.15 Weisen Sie den auf eine ganze Zahl aufgerundeten Wert $e^{\frac{2}{3}\pi}$ dem letzten Element der dritten Spalte von A zu.
7.16 Geben sie dem letzten Element von A den Wert 67^2.
7.17 Löschen Sie die Zeilen 3, 5 und 7 sowie die Spalten 2 und 9.

Kapitel 6.3

Aufgabe 8

8.1 Erzeugen Sie eine Diagonalmatrix mit den Zahlen 1, 2, 3, 4 und 5 in der Hauptdiagonalen!
8.2 Erzeugen Sie eine Diagonalmatrix mit den Zahlen 9, 7, 5, 3 und 1 in der Hauptdiagonalen, dem Vektor [4 5] in der unteren Nebendiagonalen und dem Vektor [6 7 8] in der oberen Nebendiagonalen.

Aufgabe 9

Erzeugen Sie eine 2x3-Matrix, deren Elemente alle den Wert $\sqrt{2}$ besitzen!

Aufgabe 10

10.1 Erzeugen Sie einen Zeilenvektor der Länge 10, dessen Elemente zufällig die Werte 0 oder 1 haben.
10.2 Erzeugen Sie eine 3x6-Matrix, deren Elemente zufällig die Werte 0,+1 oder -1 haben.

Kapitel 6.4

Aufgabe 11: Geben Sie die reale Länge des Vektor v = [1 1 1] an.

Kapitel 6.5

Aufgabe 12: Geg.: $C = \begin{bmatrix} -1+i & -5i \\ 7 & 3-4i \end{bmatrix}$

12.1 Die Matrix C2 soll aus den konjugiert komplexen Elementen von C bestehen.
12.2 Transponieren Sie nun diese Matrix C2 und speichern Sie sie unter C3.
12.3 Wie können Sie Matrix C3 mit einem Befehl aus Matric C erhalten?
12.4 Bilden Sie elementweise das Produkt von C und C2.

Aufgabe 13: Lösen Sie folgende Gleichungssysteme zuerst von Hand und dann in MATLAB:

13.1: $X \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

13.2: $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot x = \begin{bmatrix} 2 \\ 5 \end{bmatrix}$

13.3: $X \cdot \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$

Aufgabe 14: Geg.: $A = \begin{bmatrix} 1 & 2 & 4 \\ 2 & 6 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 4 & 1 & 4 & 3 \\ 0 & -1 & 3 & 1 \\ 2 & 7 & 5 & 2 \end{bmatrix}$

Berechnen Sie A*B von Hand, dann mit MATLAB.

Aufgabe 15: Geg.: a = [1 2 3], b = [2 4 6]

Führen Sie die folgenden Vektoroperationen in MATLAB durch. Überlegen Sie sich, warum manche nicht definiert sind. Erklären Sie die Resultate.
(Schnelle Eingabe mit: `>> a=1:3, b=2*a)`

a+b	a-b	a+b'	a*b	a*b'	a'*b	b'*a	b/a	1\2	b/a'	a^2	sin(a)
a.+b	a.-b	a'+b'	a.*b	a.*b'	a'.*b	b'*.a	b./a	a.\b	b./a'	a.^2	

Kapitel 8.2

Aufgabe 16:

16.1 Geben Sie folgende CELL mit dem Namen Ce_2 ein:

	Spalte 1	*Spalte 2*
Zeile 1	"Zylinderkopf"	3x3-Einheitsmatrix
Zeile 2	3+2i	Vektor von 0 bis 2π mit 100 Elementen

16.2 Rufen Sie die Einheitsmatrix der Cell derart auf, dass Sie in diese hinein sehen können.
16.3 Wie können Sie die Einheitsmatrix noch aufrufen, ohne dabei ihren Inhalt zu sehen?

Kapitel 8.4

Aufgabe 17

Definieren Sie im Command Window direkt und ohne den Umweg über den Editor die Funktion $y = e^x + \sin(x^2)$ über einen Function Handle und berechnen Sie das Ergebnis für y=0 und y=π.

Kapitel 10.1

Aufgabe 18

Erzeugen Sie eine gleichverteilte 10x10-Zufallsmatrix aus ganzen Zahlen im Wertebereich von 1 bis 10. Weisen Sie unter Verwendung einer FOR-Schleife jedem dritten Element den Wert 0 zu, angefangen mit Element eins.

Kapitel 7.2 und 10

Aufgabe 19

Schreiben Sie im Editor eine Function Fu1 als m-File, die bei Aufruf folgende Matrizen liefert:

```
>> Fu1(2)
ans =
     1     2
     3     4

>> Fu1(3)
ans =
     1     2     3
     4     5     6
     7     8     9

>> Fu1(2,3)
ans =
     1     2     3
     4     5     6
```

```
>> Fu1(2,4,10,5)
ans =
    10    15    20    25
    30    35    40    45

>> Fu1(3,3,100,-15)
ans =
   100    85    70
    55    40    25
    10    -5   -20
```

Die Syntax lautet also: *Fu1(Anzahl Zeilen, Anzahl Spalten, Startwert, Schrittweite)*. Verwenden Sie die Function *nargin (siehe 7.2* und *doc nargin)* derart, dass die Anzahl der Eingangsargumente variabel ist von 1 bis 4:

- Werden weniger als 4 Argumente angegeben, dann soll die Schrittweite 1 betragen.
- Werden weniger als 3 Argumente angegeben, dann soll der Startwert 1 betragen.
- Wird nur 1 Argument angegeben, dann soll die Anzahl der Zeilen gleich der Anzahl der Spalten betragen.

Zusätzlich soll bei Eingabe des Befehls *doc Fu1* im Help Browser eine kurze aber stichhaltige Erklärung der Function erscheinen.

Kapitel 7.1, 10 und 12

Aufgabe 20

Ihnen liegen Messergebnisse in einer gleichverteilten Zufallsmatrix der Dimension 5x5 vor: *M = 20*rand(5)*. Schreiben Sie ein Script, das diese Messergebnisse in einen Spaltenvektor anordnet und sie in aufsteigender Reihenfolge sortiert. Es sollen alle Messwerte gelöscht werden, die kleiner als 5 oder größer als 12 sind. Bilden Sie das Ergebnis in einem Graphen ab.

Kapitel 7.1 und 11

Aufgabe 21

Schreiben Sie ein Script, das 21 Werte der Exponentialfunktion im Bereich von 0 bis 1 im Command Window auf folgende Weise ausgibt:

```
Werte der exp-Funktion

  k          exp(x(k))
----------------------
  1           1.0000
  2           1.0513
  3           1.1052

  .              .
  .              .
  .              .

 21           2.7183
```

Kapitel 7.1 und 12

Aufgabe 22

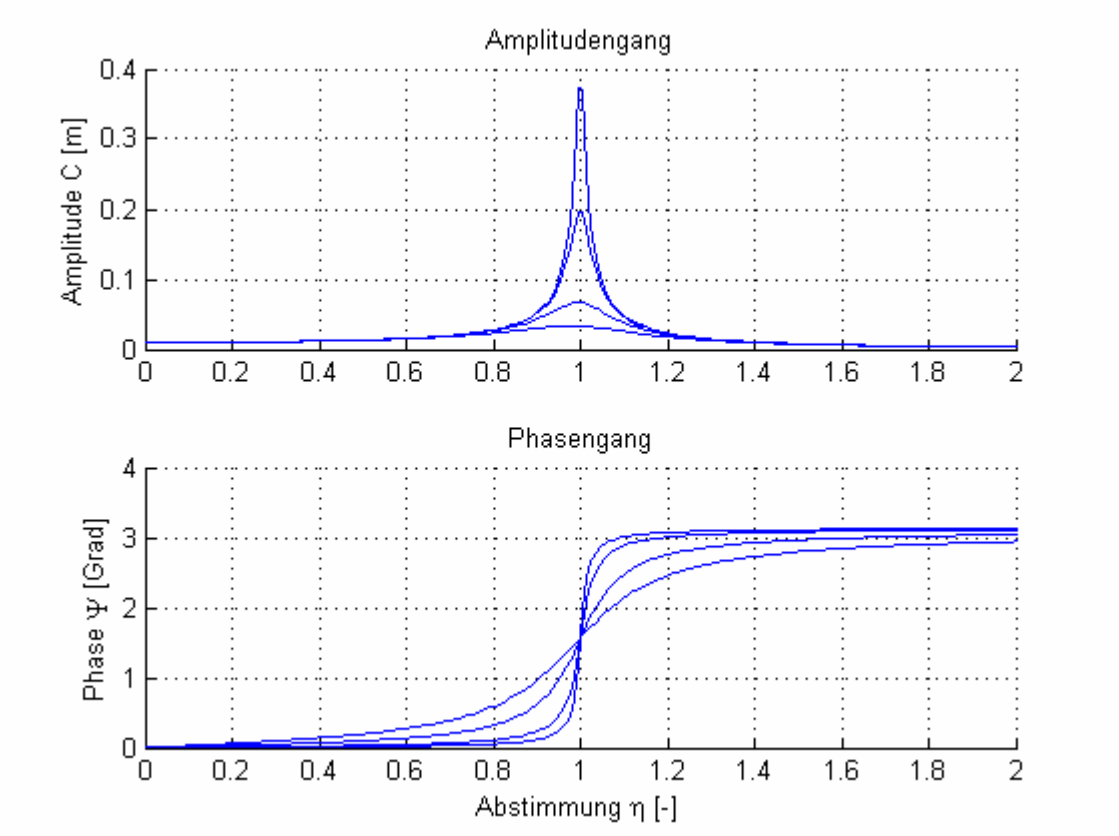
Schreiben Sie ein Script, das Amplituden- und Phasengang eines gedämpften Einmasse-Schwingers grafisch darstellt.

- Der Amplitude C sei gegeben durch: $$C = \frac{q}{\omega_0^2} \cdot \frac{1}{\sqrt{\left(1-\eta^2\right)^2 + 4D^2\eta^2}}$$

- Die Phase ψ sei im vorliegenden Fall: $$\psi = \arctan\left(\frac{2D\eta}{1-\eta^2}\right)$$

- Dämpfungsgrade D = 1.25%, 2.5%, 7.5% und 15%
- Eigenkreisfrquenz ω_0 = 10.0 rad/s
- Bezogene Erregeramplitude q = 1.0 m/s^2

Verwenden Sie für die Berechnung von ψ die in MATLAB integrierte Function *atan2()* . Zeichnen Sie Amplituden- und Phasengang für die vier Dämpfungswerte jeweils in ein Koordinatensystem übereinander. Das Ergebnis soll so ausehen:



LÖSUNGEN

Aufgabe 1

```
>> 2^2+log(pi)*sin(0.75*pi/2)+sqrt(exp(2/3*pi))
```

Aufgabe 2

2.1
```
>> real(C)
```
2.2
```
>> imag(C(2,:))'
```
2.3
```
>> C2 = conj(C)
```
2.4
```
>> abs(C(3)), angle(C(3))
```

Aufgabe 3

3.1
```
>> x = 1:3;
>> A = x([1 1 1],:)
```
3.2
```
>> u = [4 4 4];
>> A = [A u']  -oder-  >> A = [A [4;4;4]]
```
3.3
```
>> v = 1:4;
>> A = [A;v]   -oder-  >> A = [A; 1:4]
```
3.4
```
>> v([4 2 3 1])
```

Aufgabe 5

5.1
```
>> Comp = (2-i)*eye(4)
```
5.2
```
>> Comp(3,1) = 3+4i
```
5.3
```
>> Comp2 = Comp.'  -oder-  >> Comp2 = transpose(Comp)
```
5.4
```
>> Comp3 = conj(Comp2)
```

Aufgabe 6

```
>> Z = zeros(4,3)
>> dim = size(Z)
>> eye(dim)   -Abkürzung:-   >> eye(size(zeros(4,3)))
```

Aufgabe 7

7.2
```
>> z = diag(A)'
```
7.3
```
>> diag(diag(A))  -oder-  >> diag(z)
```
7.4
```
>> z = diag(A, -1)'
```
7.5
```
>> diag(diag(A, 2))
```
7.6
```
>> sum(A)
```
7.7
```
>> sum(A')
```
7.8
```
>> sum(diag(A))
```
7.9
```
>> sum(sum(A))
```
7.11
```
>> A = sort(A)
```
7.12
```
>> min(A), max(A)
```
7.13
```
>> min(min(A)), max(max(A))
```

```
7.14 >> A(5,end) = sin(pi/3)
7.15 >> A(end,3) = ceil(exp(2/3*pi))
7.16 >> A(end) = 67^2
7.17 >> A(3:2:7,:) = []
     >> A(:,[2 9]) = []
```

Aufgabe 8

```
8.1  >> D = diag([1:5])
8.2  >> D = diag([9:-2:1]) + diag([4 5],-3) + diag(6:8,2)
```

Aufgabe 9

```
>> sqrt(2)*ones(2,3)
```

Aufgabe 10

```
10.1 >> round(rand(1,10))
10.2 >> round(rand(3,6)*2-1)
```

Aufgabe 11

```
>> norm([1 1 1])
```

Aufgabe 12

```
12.1 >> C = [-1+i -5i; 7 3-4i];
     >> C2 = conj(C)
12.2 >> C3 = transpose(C2)  -oder-   >> C3 = C2.'
12.3 >> C3 = ctranspose(C)  -oder-   >> C3 = C'
12.4 >> C.*C2
```

Aufgabe 13

```
13.1 >> X = [5 6;7 8]/[1 2;3 4]        -oder-
     >> X = [5 6;7 8]*inv([1 2;3 4])   -oder-
     >> X = [5 6;7 8]*([1 2;3 4])^-1
13.2 >> x = [1 2;3 4]\[2;5]            -oder-
     >> x = inv(A)*[2;5]
13.3 >> X = [3;4]/[1;2]
```

Aufgabe 16

```
16.1 >> Ce_2{1,1} = 'Zylinderkopf';
     >> Ce_2{1,2} = eye(3);
     >> Ce_2{2,1} = 3+2i;
     >> Ce_2{2,2} = linspace(0,2*pi,100)
16.2 >> Ce_2{1,2}
16.3 >> Ce_2(1,2)
```

Aufgabe 17

```
>> y = @(x) exp(x) + sin(x^2);
>> y(0)
>> y(pi)
```

Aufgabe 18

```
>> A = ceil(10*rand(10))
>> for ii = 1:3:100, A(ii) = 0, end; % nicht i als Zähler verwenden!
```

Aufgabe 19

```
 1    function[A] = Fu1(m, n, start, schritt)
 2    % erstellt eine Matrix A = Fu1(Anz. Zeilen, Anz. Spalten,
        Start, Schrittweite)
 3    % Variable Anzahl der Eingabeargumente!
 4
 5 -  if nargin < 4, schritt = 1; end
 6 -  if nargin < 3, start = 1; end
 7 -  if nargin < 2, n = m; end
 8 -  z = start;
 9 -  for ii = 1:m
10 -      for jj = 1:n
11 -          A(ii,jj) = z;
12 -          z = z + schritt;
13 -      end
14 -  end
```

Aufgabe 20

```
 1 -   M = 20*rand(5);
 2 -   M = sort(M(:));
 3 -   z = 1;
 4 -   while z <= length(M)
 5 -       if M(z)<5 || M(z)>12
 6 -           M(z) = [];
 7 -       else z = z + 1;
 8 -       end
 9 -   end
10 -   plot(M), grid
```

Aufgabe 21

```
 1 -  clc
 2 -  x = linspace(0,1,21);
 3 -  ausg = [1:21; exp(x)];
 4 -  disp('Werte der exp-Funktion')
 5 -  disp(' ')
 6 -  disp('  k          exp(x(k))')
 7 -  disp('----------------------')
 8 -  disp(sprintf('%3.0f %16.4f \n', ausg))
```

alternativ zu Zeile 8 geht auch:

```
 8 -  for n=1:21
 9 -      fprintf('%3.0f %16.4f \n', ausg(:,n))
10 -  end
```

Aufgabe 22

```
 1 -  om_0 = 10.0;               % Eigenkreisfrequenz [rad/s]
 2 -  q    = 1.0;                % bezogene Erregeramplitude [m/s^2]
 3 -  eta  = linspace(0,2,200);  % bezogene Erregerfrequenz
 4 -  D    = [0.0125 0.025 0.075 0.15];% Dämpfungswerte
 5
 6 -  for z = 1:4
 7        % Amplitude C:
 8 -      C = q/om_0^2 ./ sqrt((1.0-eta.^2).^2 + 4.0*D(z)^2*eta.^2);
 9        % Phasenverschiebungswinkel Psi:
10 -      Psi = atan2((2.0*D(z)*eta), (1.0-eta.^2));
11
12 -      subplot(2,1,1)
13 -          hold on
14 -          plot(eta, C)
15 -          if z == 4
16 -              title('Amplitudengang')
17 -              ylabel('Amplitude C [m]')
18 -              grid
19 -          end
20 -          hold off
21
22 -      subplot(2,1,2)
23 -          hold on
24 -          plot(eta, Psi)
25 -          if z == 4
26 -              title('Phasengang')
27 -              xlabel('Abstimmung \eta [-]')
28 -              ylabel('Phase \Psi [Grad]')
29 -              grid
30 -          end
31 -          hold off
32 -  end
```