MATLAB Primer
zur Vorlesung
Mathematische Methoden der Elektrotechnik

Version 2.2

Dipl. Math.-techn. Ingrid Reißel

7. September 2010

Lehrstuhl für Bildverarbeitung
Prof. Dr.-Ing. Til Aach
Sommerfeldstraße 24
D-52074 Aachen

RWTH AACHEN
RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

Tel.: +49 (0) 241-80-27860
Fax: +49 (0) 241-80-22200
lfb@lfb.rwth-aachen.de
www.lfb.rwth-aachen.de

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Was ist MATLAB?

MATLAB ist ein in der Industrie weit verbreitetes interaktives Programmsystem zum wissenschaftlichen, numerischen Rechnen. Es stellt eine einfache C-ähnliche Programmiersprache inklusive Entwicklungsumgebung zur Verfügung.

- Grundlegender Datentyp sind Matrizen.
- MATLAB steht dabei für **MAT**rix **LAB**oratory.
- Hersteller ist die Firma The MathWorks Inc.

1.2 Welche Vorteile hat MATLAB?

Neben der Berechnung bietet MATLAB auch die Möglichkeiten zur grafischen Darstellung und zum Programmieren eigener Skripte und Funktionen. Weitere Vorteile von MATLAB sind:

- es ist leicht erlernbar
- die Variablen (auch Felder) müssen nicht vereinbart (deklariert) werden
- vom Benutzer definierte Funktionen und Unterprogramme werden einfach in Textdateien (m-Files) abgespeichert und können dann wie eingebaute Funktionen benutzt werden
- auf dieser Basis gibt es zahlreiche Erweiterungen (Toolboxes), die eine Unmenge an Funktionalitäten zur Verfügung stellen
- ein einfaches Hilfe-System, in das (nach entsprechender Dokumentation) auch eigene Funktionen integriert werden können
- komfortable Fehlersuche durch den eingebauten, leicht bedienbaren Debugger

Diese MATLAB–Einführung dient als Ergänzung zur Vorlesung Mathematische Methoden der Elektrotechnik und wurde erstmals im WS 2007/2008 erstellt. Seitdem wurde der Primer stets weiterentwickelt und aktualisiert.

Studierende ohne größere Programmiererfahrung erhalten hiermit eine Anleitung zum Arbeiten mit MATLAB. Darüber hinaus soll der Primer die Studenten beim Bearbeiten der Übungsaufgaben zur Vorlesung unterstützen und den Teilnehmern des Projekts 'MATLAB meets LEGO Mindstorms' die notwendigen Grundkenntnisse in MATLAB vermitteln.

Der Primer bezieht sich auf die MATLAB Funktionalität ab Version 7.6.0 (R2008a).

An dieser Stelle möchte ich mich bei dem Institut für Regelungstechnik, Univ.-Prof. Dr.-Ing. Dirk Abel, für die Überlassung der 'Kurzeinführung in MATLAB/SIMULINK/STATEFLOW' bedanken. Ein besonderer Dank geht auch an das Institut für Nachrichtengeräte und Datenverarbeitung, Prof. Dr.-Ing Peter Vary, das mir die Unterlagen zum Praktikum 'MATLAB in der digitalen Signalverarbeitung' vom Wintersemester 2006/2007 inklusive LaTeX-Quellcode zur Verfügung gestellt hat.

Kapitel 2

Grundlagen

2.1 Beschreibung der Benutzeroberfläche

Nach dem Hochfahren von MATLAB sollte etwa folgendes Fenster zu sehen sein

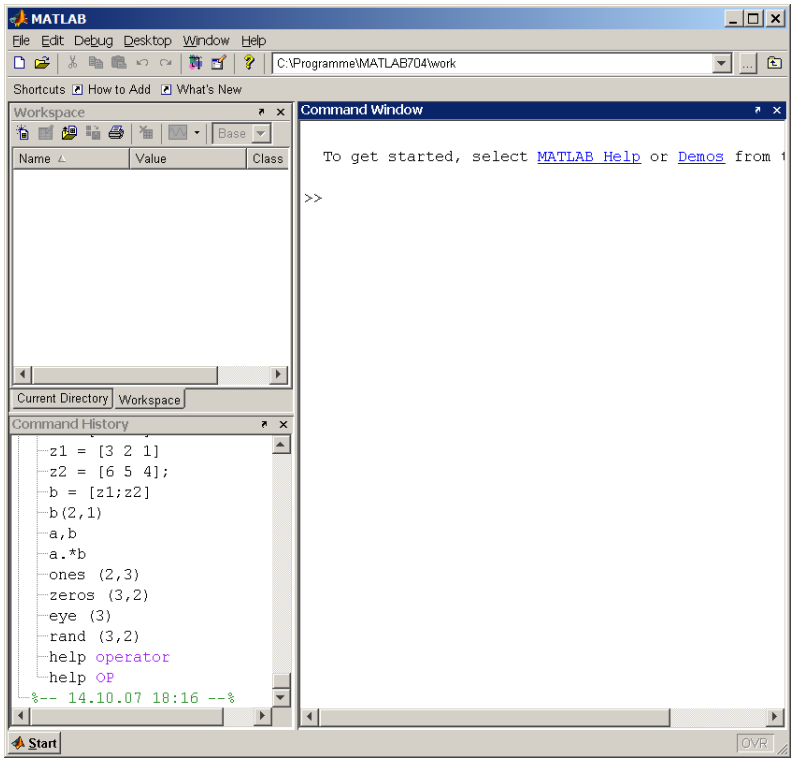


Abbildung 2.1: MATLAB Oberfläche

Das Fenster ist in 3 Bereiche geteilt

- rechts befindet sich das ***Command Window***. Hier gibt der Benutzer seine Befehle ein und hier werden auch die Ausgaben des Systems angezeigt.

- links oben werden (bei Einstellung ***Workspace***) alle augenblicklich vorhandenen Variablen mit Typ und Speicherbedarf oder (bei Einstellung ***Current Directory***) alle vom Benutzer in seinem Arbeitsverzeichnis abgelegten m-Files angezeigt

- links unten werden in der ***Command History*** alle vom Benutzer im Kommandofenster eingegebenen Befehle aufgelistet

Die Einstellung, ob links oben das *Current Directory* oder der *Workspace* dargestellt wird, kann durch Betätigen des jeweiligen Reiters oberhalb des Fensterauschnitts oder im Menü `Desktop-Desktop Layout` verändert werden. Dort findet man auch die Einstellung `Default`, die nach eventuell unbeabsichtigten Änderungen an den Fenstern den Ursprungszustand wieder herstellt.

Für jedes neue Projekt sollte ein eigenes Verzeichnis angelegt werden. Damit MATLAB in diesem Verzeichnis arbeitet, muss am oberen Rand des Fensters unter *Current Directory* das jeweilige Verzeichnis eingestellt werden.

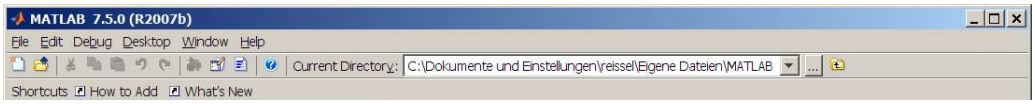


Abbildung 2.2: Current Directory

2.2 Erste Schritte

Zuweisungen von Werten und Variablen können ohne vorherige Vereinbarung im Kommandofenster vorgenommen werden, wie z.B.

```
>> x = 17
x =
     17
```

wobei x den Wert 17 zugewiesen bekommt. Dabei ist zu beachten:

- die Zuweisung wird nochmal im Kommandofenster ausgegeben

- im Fenster links oben (falls die Workspace-Ansicht eingestellt ist) taucht die Variable x auf

- x ist skalar und wird als 1×1-Matrix (Size ist 1×1 im linken oberen Fenster) gespeichert

- **Variablennamen** müssen immer mit einem Buchstaben beginnen und dürfen einige Sonderzeichen nicht enthalten

Wird die Eingabe mit einem ; beendet, so wird die Ausgabe im Kommandofenster (siehe Abb. 2.1) unterdrückt:

```
>> y = 22;
>>
```

Der Wert der Variablen x kann abgefragt werden, indem einfach x (ohne ;) eingegeben wird bzw. durch Doppelklick auf das x-Icon im Workspace-Fenster (linkes oberes Fenster).

```
>> x
x =
     17
```

Nach dem Doppelklick auf das x-Icon im Workspace Fenster erscheint der Array-Editor, indem der Wert der Variablen x auch verändert werden kann.

Mit dem Befehl `whos` können alle definierten Variablen abfragt werden:

```
>> whos
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  double
  y         1x1                 8  double
```

Mit der Cursor-nach-oben Taste können vorherige Eingaben erneut angezeigt und verändert werden, d.h. gibt man

```
x =
```

ein und betätigt die Cursor-Taste, dann werden nur alte Kommandos durchblättert, die mit `x =` beginnen.

Standardmäßig zeigt MATLAB bei Dezimalzahlen nur sehr wenige Dezimalstellen an, z.B.

```
>> x = 1/7
x =
    0.1429
```

Intern wird aber immer mit voller doppelter Genauigkeit (etwa 16 Dezimalziffern) gerechnet. Mit Hilfe des `format` Befehls kann man sich diese Ziffern auch anzeigen lassen.

```
>> x = 1/7
x =
    0.1429

>> format long

>> x
x =
   0.142857142857143

>> format short
```

```
>> x
x =
    0.1429
```

2.3 Benutzung der MATLAB Hilfe

MATLAB bietet ein sehr umfangreiches **Hilfesystem**. Man hat folgende Möglichkeiten die MATLAB Hilfe aufzurufen:

- Der Aufruf des Befehls

  ```
  >> help plot
  ```

  liefert im Kommandofenster nähere Angaben zur Funktion `plot`. Ein Nachteil hierbei ist, dass die Ausgabe sehr umfangreicher Informationen oft unübersichtlich ist.

  Außerdem wird im Hilfetext der abgefragte Befehl immer in Großbuchstaben dargestellt, was oftmals verwirrend ist, da MATLAB zwischen Groß- und Kleinschreibung unterscheidet (d.h. `plot` und `PLOT` sind nicht das selbe!).

- Mit

  ```
  >> doc plot
  ```

  öffnet sich der MATLAB Help Browser und zeigt eine ausführliche in HTML formatierte Beschreibung an. Besonders hilfreich sind die meist ausführlichen Beispiele und Verweise auf verwandte Themen.

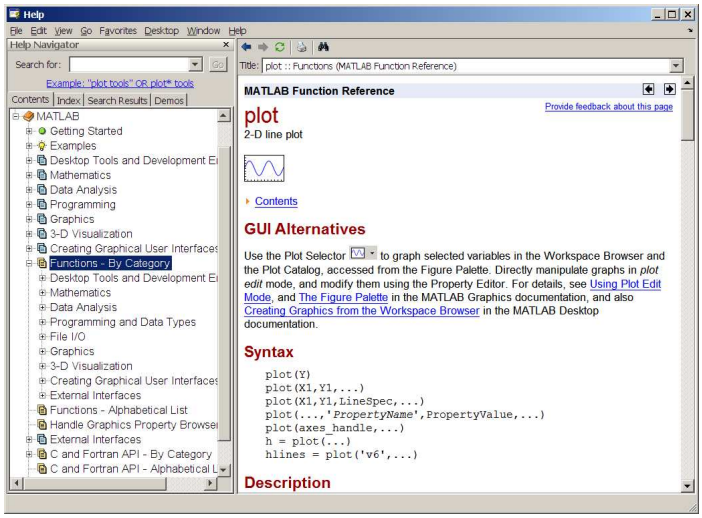


Abbildung 2.3: Hilfe

Den MATLAB **Help Browser** kann man auch über den Fragezeichen-Button in der Menüleiste erreichen. Im linken Teil des Browser-Fensters findet man den Navigationsbereich, mit dem man durch die gesamte lokal verfügbare Hilfe stöbern kann. Im Eingabefeld *Search* kann nach bestimmten Stichworten gesucht werden.

Kapitel 3

Datenstrukturen

3.1 Zahlen, Vektoren und Matrizen

3.1.1 Reelle Zahlen

MATLAB wurde primär zur Verarbeitung von Matrizen entwickelt. Konstanten (Skalare) und reelle Zahlen sind 1×1 Matrizen.
MATLAB besitzt einige **vordefinierte Systemvariablen** (die verändert werden können!):

- **ans** ('answer') enthält das Ergebnis des letzten Befehls, der in der Kommandozeile abgesetzt wurde ohne dass eine Zuweisung zu einer anderen Variablen erfolgt ist
- **pi** enthält die Zahl π
- **eps** enthält die Zahl $2^{-52} = 2.22e-16$ (Maschinengenauigkeit)
- **realmin, realmax** sind die kleinsten und größten positiven reellen Zahlen, die in MATLAB verfügbar sind
- $\pm$**Inf** (**Inf**inity) steht für $\pm$Unendlich, was man z.B. als Ergebnis von $\pm 1/0$ erhält
- **NaN** (**N**ot **a** **N**umber) erhält man bei mathematisch nicht definierten Operationen wie $0/0$
- **i,j** enthalten beide die imaginäre Einheit $\sqrt{-1}$ zur Eingabe von **komplexen Zahlen**

Auch hier sollte man beachten, dass MATLAB zwischen Groß- und Kleinschreibung unterscheidet.

3.1.2 Komplexe Zahlen

Neben reellen Zahlen, kann MATLAB auch mit komplexen Zahlen umgehen. **Komplexe Zahlen** können mit Hilfe der vordefinierten imaginären Einheit i oder j wie folgt eingegeben werden:

```
>> c = 1 + 2*i
c =
   1.0000 + 2.0000i

>> c2 = 3 + 4.5*j
c2 =
   3.0000 + 4.5000i
```

Das funktioniert nur, wenn die Variablen i bzw. j nicht schon anderweitig benutzt wurden:

```
>> i = 3
i =
     3

>> c = 1 + 2*i
c =
     7
```

In diesem Fall kann man die Funktion `complex` einsetzen:

```
>> c3 = complex(4,7)
c3 =
   4.0000 + 7.0000i
```

Mit `real` und `imag` kann man Real- und Imaginärteil einer komplexen Zahl ermitteln:

```
>> c = 2 + 3*j
c =
   2.0000 + 3.0000i

>> real(c)
ans =
     2

>> imag(c)
ans =
     3
```

Mit den Funktionen `abs` und `angle` erhält man die Polardarstellung einer komplexen Zahl

```
>> c = 2 + 3*j
c =
   2.0000 + 3.0000i
```

```
>> betrag = abs(c)
betrag =
    3.6056

>> winkel = angle(c)
winkel =
    0.9828

>> betrag * exp(j*winkel)
ans =
   2.0000 + 3.0000i
```

Beim Arbeiten mit komplexen Zahlen ist es wichtig, folgendes zu beachten: Anstelle der Funktion `angle` kann man auch `atan2` verwenden:

```
>> c =exp(j*3*pi/4)
c =
  -0.7071 + 0.7071i

>> atan2(imag(c),real(c))
ans =
    2.3562

>> 3*pi/4
ans =
    2.3562
```

Man sieht `atan2` liefert hier den richtigen Winkel.

Verwendet man jedoch `atan` muss man, je nachdem, in welchen Quadranten man sich befindet, entweder π oder -π dazu addieren.

```
>> atan(imag(c)/real(c))
ans =
   -0.7854

>> atan(imag(c)/real(c))+pi
ans =
    2.3562
```

3.1.3 Vektoren und Matrizen

Vektoren werden genau wie Skalare als spezielle Matrizen behandelt. Dabei wird strikt zwischen **Zeilenvektoren** ($1 \times n$ Matrizen) und **Spaltenvektoren** ($n \times 1$ Matrizen) unterschieden. Die Eingabe von Zeilenvektoren erfolgt durch

```
>> z = [1  2  5  7]
z =
     1     2     5     7
```

bzw.

```
>> z = [1,2,5,7]
z =
     1     2     5     7
```

Bei Spaltenvektoren müssen die Komponenten durch ; getrennt werden, d.h.

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

     1
     2
     3
     4
```

Spezielle Vektoren können auch schneller mit Hilfe des **Doppelpunktoperators** eingegeben werden:

```
vektor = [Anfang : Inkrement : Ende]
```

Der Vektor enthält anschließend äquidistante Einträge. Das Inkrement kann auch negativ sein und im Falle, dass es 1 ist, weggelassen werden.

```
>> z2 = [5:8]
z2 =
     5     6     7     8

>> z3 = [1:0.6:2.8]
z3 =
    1.0000    1.6000    2.2000    2.8000

>> z4 = [1:2:4]
z4 =
     1     3

>> z5 = [5:-2:1]
z5 =
     5     3     1
```

Die Länge eines Vektors kann mithilfe der Funktion `length` bestimmt werden.

```
>> length(z4),length(z5)
ans =
     2
ans =
     3
```

d.h. $z4$ ist ein Vektor der Länge 2 und $z5$ ist ein Vektor der Länge 3.

Mit dem Befehl `size` ermittelt man das Format einer Variablen. Für s und z aus dem obigen Beispiel erhalten wir

```
>> size(z), size(s)
ans =
     1     4

ans =
     4     1
```

d.h. z ist ein Zeilenvektor, s ein Spaltenvektor. Man beachte, dass das Ergebnis von `size` selbst ein Vektor mit zwei Komponenten ist.

Zeilenvektoren werden mit Hilfe des **Transpositionsoperators** $'$ in Spalten umgewandelt und umgekehrt:

```
>> z , s
z =
     1     2     5     7

s =
     1
     2
     3
     4

>> z' , s'
ans =

     1
     2
     5
     7

ans =
     1     2     3     4
```

Bei der Verwendung des Transpositionsoperators ist folgendes zu beachten: Bei komplexen Vektoren werden zusätzlich die einzelnen Komponenten komplex konjugiert:

```
>> z = [i, 1, i+1]
z =
   0 + 1.0000i   1.0000   1.0000 + 1.0000i
>> z'
ans =
        0 - 1.0000i
   1.0000
   1.0000 - 1.0000i
```

Auf ein einzelnes Element eines Vektors wird in Array-Notation zugegriffen:

```
>> z = [1,2,5,7]
z =
```

```
      1      2      5      7

>> z(3)
ans =
      5
```

Man beachte die runden Klammern! Analog kann man komplette Teilvektoren selektieren, z.B.

```
>> z = [1,2,5,7]
z =
      1      2      5      7

>> z(1:3)
ans =
      1      2      5
```

d.h. $z(1:3)$ selektiert die Komponenten mit Index $1, 2, 3$. Bei der Selektion von Komponenten kann die vordefinierte **Indexgrenze** `end` nützlich sein, die immer den größten Index des jeweiligen Vektors enthält:

```
>> z = [1,2,5,7]
z =
      1      2      5      7

>> z(2:end-1)
ans =
      2      5
```

Soll ein nicht zusammenhängender Bereich selektiert werden, so benutzt man einfach einen Indexvektor

```
>> z([1,3,4])
ans =
      1      5      7
```

Neben dem Auswählen von Teilvektoren können Vektoren auch erweitert werden:

```
>> z = [1,2,5,7]
z =
      1      2      5      7

>> z = [z,8,9]
z =
      1      2      5      7      8      9

>> z2 = [5:8]
z2 =
      5      6      7      8

>> z = [z,z2]
```

```
z =
     1     2     5     7     8     9     5     6     7     8
```

Für Matrizen gelten analoge Aussagen wie für Vektoren. Mit

```
>> a = [1 2 3; 4 5 6]
a =
     1     2     3
     4     5     6
```

definiert man also eine 2×3 Matrix. Matrizen können auch aus Zeilen- bzw. Spaltenvektoren zusammengesetzt werden:

```
>> z1 = [3 2 1]
z1 =
     3     2     1

>> z2 = [6 5 4];

>> b = [z1;z2]
b =
     3     2     1
     6     5     4
```

Man beachte, dass Zeilen mit ; aber Spalten mit , getrennt werden müssen.

Zur Selektion einzelner Elemente wird wieder Array-Notation benutzt, so dass für die Matrix b aus dem letzten Beispiel folgt

```
>> b(2,1)
ans =
     6
```

Möchte man **Untermatrizen** auswählen, so kann man wieder Bereiche angeben oder Indexvektoren verwenden:

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

>> a([1,3],[2,3])
ans =
     2     3
     8     9
```

d.h. es wird aus den Zeilen 1 und 3 und den Spalten 2 und 3 eine 2×2 Matrix selektiert. Eine andere Möglichkeit zur Selektion ist die Folgende:

```
>> ii = 1:2, jj = 2:3
ii =
```

```
     1     2
jj =
     2     3

>> a(ii,jj)
ans =
     2     3
     5     6
```

Hier werden aus den ersten beiden Zeilen die Spalten 2 bis 3 ausgewählt.
Analog zu Vektoren können auch Matrizen erweitert werden.

```
>> a = [1,2,3; 4,5,6]
a =
     1     2     3
     4     5     6

>> a = [a, [10;11]]
a =
     1     2     3    10
     4     5     6    11

>> a = [a; [21:24]]
a =
     1     2     3    10
     4     5     6    11
    21    22    23    24
```

Darüber hinaus kann jede Matrix durch Aneinanderhängen aller Matrixspalten in einen Spaltenvektor umgewandelt werden

```
>> a = [1,2,3; 4,5,6]
a =
     1     2     3
     4     5     6

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

Umgekehrt kann mit `reshape` aus einem Zeilen- bzw. Spaltenvektor geeigneter Länge eine Matrix erzeugt werden

```
>> z = 1:6
z =
```

```
     1     2     3     4     5     6

>> a = reshape(z, 2, 3)
a =
     1     3     5
     2     4     6

>> b = reshape(z, 3, 2)
b =
     1     4
     2     5
     3     6
```

Das Transponieren mit ′ sowie `size` und `end` sind bei Matrizen genauso einzusetzen wie bei Vektoren.

3.1.4 Operatoren und Standardfunktionen

Wie wir im letzten Abschnitt gesehen haben, sind Vektoren spezielle Matrizen, weshalb wir hier, bis auf wenige Ausnahmen, nicht zwischen ihnen unterscheiden müssen.

Matrizen gleichen Formats können addiert, subtrahiert und mit einer reellen Zahl multipliziert bzw. dividiert werden, wobei jeweils komponentenweise gerechnet wird:

```
>> z = [1,2,5,7],   z2 = [5:8]
z =
     1     2     5     7

z2 =
     5     6     7     8

>> z + z2
ans =
     6     8    12    15

>> z * 4
ans =
     4     8    20    28

>> a = [1,2; 3,4], a2 = [5,6; 7,8]
a =
     1     2
     3     4

a2 =
     5     6
     7     8
```

```
>> a2 - a
ans =
     4     4
     4     4
```

Außerdem kann auf eine Matrix eine Konstante addiert bzw. subtrahiert werden, wobei ebenfalls wieder komponentenweise gerechnet wird:

```
>> a = [1,2,3; 3,4,5]
a =
     1     2     3
     3     4     5

>> a + 5
ans =
     6     7     8
     8     9    10
```

Weiterhin steht die übliche **Matrizenmultiplikation** zur Verfügung.

```
>> a = [1,0,1; 2,1,2], a2 = [1,0; 0,1; 1,1]
a =
     1     0     1
     2     1     2

a2 =
     1     0
     0     1
     1     1

>> a * a2
ans =
     2     1
     4     3

>> a2 * a
ans =
     1     0     1
     2     1     2
     3     1     3
```

Als Spezialfall erhalten wir das **Skalarprodukt** als Matrixmultiplikation eines Zeilenvektors mit einem Spaltenvektor:

```
>> z = [1,2,5,7]
z =
     1     2     5     7

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

```
     1
     2
     3
     4

>> z * s
ans =
    48
```

d.h. das Ergebnis ist eine 1×1 Matrix. Alternativ kann man auch die Funktion `dot` benutzen

```
>> dot(s,z)
ans =
    48
```

Multipliziert man umgekehrt den Spaltenvektor mit dem Zeilenvektor, so erhalten wir eine Matrix, in unserem Beispiel also

```
>> s * z
ans =
     1     2     5     7
     2     4    10    14
     3     6    15    21
     4     8    20    28
```

Daneben gibt es die Möglichkeit, zwei Matrizen **komponentenweise** zu multiplizieren, indem statt $*$ der Operator $.*$ benutzt wird:

```
>> z = [1,2,5,7]
z =
     1     2     5     7

>> z2 = [5:8]
z2 =
     5     6     7     8

>> z .* z2
ans =
     5    12    35    56
```

Analog kann man mit ./ und .^ **komponentenweise dividieren** bzw. **potenzieren**. Eine vollständige Liste aller verfügbaren Operatoren kann man sich z.B. mit `help +` anzeigen lassen.

Bei allen Operationen mit Vektoren und Matrizen muss darauf geachtet werden, dass die Formate mathematisch korrekt gewählt sind.

In MATLAB sind zahlreiche **Standardfunktionen** wie z.B. `sin`, `cos`, `exp`, `sqrt` (Wurzel) implementiert, die auf reellen bzw. komplexen Zahlen im üblichen Sinn definiert sind

```
>> sin(pi), exp(1), sqrt(-1)
ans =
   1.2246e-016

ans =
    2.7183

ans =
          0 + 1.0000i
```

Wendet man eine Standardfunktion auf eine Matrix an, so wird die Funktion einfach für jede Komponente der Matrix ausgewertet und alle Ergebnisse in einer Matrix gleichen Formats zusammen gesammelt.

```
>> a = [1,4,9; 16,25,36]
a =
     1     4     9
    16    25    36

>> sqrt(a)
ans =
     1     2     3
     4     5     6
```

Darüber hinaus gibt es zahlreiche weitere spezielle Funktionen, die den Umgang mit Vektoren und Matrizen erleichtern. Einige dieser Funktionen werden hier exemplarisch aufgelistet. Weitere Details kann man der MATLAB Hilfe entnehmen.

Zum einfachen **Erzeugen spezieller Matrizen** kann man unter anderem die folgenden Funktionen benutzen

```
>> ones(2,3) % Vorbesetzen einer Matrix mit Einsen
ans =
     1     1     1
     1     1     1

>> zeros(3,2) % Vorbesetzen einer Matrix mit Nullen
ans =
     0     0
     0     0
     0     0

>> eye(3) % Erzeugen einer Einheitsmatrix
ans =
     1     0     0
     0     1     0
     0     0     1

>> rand(3,2) % Matrix mit Zufallszahlen
ans =
```

```
    0.9501    0.4860
    0.2311    0.8913
    0.6068    0.7621
```

Die **Norm** (Länge) eines Vektors z berechnet man folgendermaßen:

```
>> z = [1,2,5,7]
z =
     1     2     5     7

>> norm(z)
ans =
    8.8882
```

Zur Bestimmung des Maximums bzw. Minimums stehen die Funktionen `min` und `max` zur Verfügung. Betrachtet man folgende Matrix a

```
>> a = [10 1 3; 1 12 13; 9 200 1]
a =
    10     1     3
     1    12    13
     9   200     1
```

so liefert

```
>> max(a)
ans =
    10   200    13
```

einen Vektor, der die Maxima jeder Spalte von a enthält.

Es ist auch möglich, außer dem Vektor c mit den Maxima zusätzlich einen Vektor d mit den dazugehörigen Indizes der Maxima zu erhalten.

```
>> [c,d] = max(a)
c =
    10   200    13
d =
     1     3     2
```

Die Minimum-Berechnung erfolgt analog mit der Funktion `min`.

Zur Mittelwertbestimmung können die Funktionen `mean` und `median` verwendet werden:

```
>> a = [10 1 3; 1 12 13; 9 200 1]
a =
    10     1     3
     1    12    13
     9   200     1
>> mean(a)
ans =
    6.6667   71.0000    5.6667
```

`mean` liefert das arithmetische Mittel über jede Spalte.

Dagegen liefert `median` den Zentralwert der Spaltenvektoren von a:

```
>> a = [10 1 3; 1 12 13; 9 200 1]
a =
    10     1     3
     1    12    13
     9   200     1
>> median(a)
ans =
     9    12     3
```

Jede Spalte wird somit ihrer Größe nach sortiert und aus der sortierten Liste der sich in der Mitte befindende Wert zurückgegeben. Dies bewirkt, dass die Ergebnisse bei `median` unabhängiger von größeren statistischen Messfehlern (”Ausreißern”) sind.

Eine komfortable Möglichkeit zum Vervielfältigen vom Matrizen bietet die Funktion `repmat`

```
>> a = [1 2; 3 4]
a =
     1     2
     3     4
>> repmat(a, 2, 3)
ans =
     1     2     1     2     1     2
     3     4     3     4     3     4
     1     2     1     2     1     2
     3     4     3     4     3     4
```

Zur Ermittlung des größten gemeinsamen Teilers (Greatest common divisor) und des kleinsten gemeinsamen Vielfachen (Least common multiple) stehen in MATLAB die Funktionen `gcd lcm,` zur Verfügung.

```
>> a = [2 7;15 4]
a =
     2     7
    15     4

>> b = [4 210;5 2]
b =
     4   210
     5     2

>> gcd(a,b)

ans =
     2     7
     5     2

>> lcm(a,b)
```

```
ans =
     4   210
    15     4
```

Hier ist zu beachten, dass beide Funktionen auf Matrizen (Komponenten müssen natürliche Zahlen sein!) definiert sind. Die Auswertung erfolgt komponentenweise (daher müssen die Matrizen in ihren Dimensionen übereinstimmen) und die Eingabe ist auf zwei Matrizen beschränkt.

```
>> c = [1 1 1;1 1 1]
c =

     1     1     1
     1     1     1

>> lcm(a,b,c)
??? Error using ==> lcm
Too many input arguments.

>> lcm(a,c)
??? Error using ==> or
Matrix dimensions must agree.

>> c = [1.1 2.2; 3.0 4.0]
c =
    1.1000    2.2000
    3.0000    4.0000

>> lcm(a,c)
??? Error using ==> lcm at 16
Input arguments must contain positive integers.
```

Das **Kreuzprodukt** zweier dreidimensionaler Vektoren erhalten wir über

```
>> x = [1,2,3], y = [0,0,1]
x =
     1     2     3

y =
     0     0     1

>> cross(x,y)
ans =
     2    -1     0
```

Mit `det` kann man die Determinante einer quadratischen Matrix bestimmen:

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

```
     3     4
     1     2

>> det(a)
ans =
     2
```

Die **Inverse** einer quadratischen (invertierbaren) Matrix berechnet man mit `inv`

```
>> a = [3,1; 1,2]
a =
     3     1
     1     2

>> b = inv(a)
b =
    0.4000   -0.2000
   -0.2000    0.6000

>> a*b, b*a
ans =
     1     0
     0     1

ans =
     1     0
     0     1
```

Lineare Gleichungssysteme $Ax = b$ löst man mit Hilfe des Operators \

```
>> A = [4,3; 3,4], b = [10;11]
A =
     4     3
     3     4
b =
    10
    11

>> x = A \ b
x =
    1.0000
    2.0000
```

und die **Eigenwerte** von A bestimmt man über

```
>> eig(A)
ans =
     1
     7
```

3.2 Zeichenketten

Um einer Variablen eine Zeichenfolge zuzuordnen, muss die **Zeichenkette** in einfache Anführungsstriche (Hochkomma) gesetzt werden:

```
>> a = 'Dies ist eine Zeichenkette'
a =
Dies ist eine Zeichenkette

>> b = ' und hier gehts weiter'
b =
 und hier gehts weiter
```

Ein Zugriff auf die einzelnen Elemente ist wie bei Vektoren und Matrizen möglich:

```
>> b(5)
ans =
h

>> b(5:8)
ans =
hier
```

Die Zeichenketten können folgendermaßen zusammengesetzt werden:

```
>> z=[a,b]
z =
Dies ist eine Zeichenkette und hier gehts weiter
```

Soll die Zeichenkette selbst ein Hochkomma enthalten, so erreicht man das durch verdoppeln:

```
>> kette = 'vor dem Hochkomma-''-nach dem Hochkomma'
kette =
vor dem Hochkomma-'-nach dem Hochkomma
```

An dieser Stelle soll noch kurz die Verwendung der Funktion `sprintf` zur formatierten Ausgabe in eine Zeichenkette erläutert werden. Sie ist sehr nützlich, um Werte von Variablen (Zahlen) in Zeichenketten einzufügen (z.B. zur Beschriftung von Grafiken).

```
>> x=12;
>> text = sprintf('Das ist Bild Nummer %d !!!',x)
text =
Das ist Bild Nummer 12 !!!
```

Die Verwendung erfolgt analog zu *C*. Weitere Details zu den Formaten erhält man mit `help sprintf`.

3.3 Strukturen

Strukturen sind ein wichtiger Bestandteil von MATLAB. Sie ermöglichen es, Daten unterschiedlichen Typs in einer Variablen zusammenzufassen. Die Handhabung lässt sich am besten anhand des folgenden Beispiels erklären.

```
>> s.name = 'Meier'
s =
    name: 'Meier'

>> s.alter = 17
s =
     name: 'Meier'
    alter: 17

>> s
s =
     name: 'Meier'
    alter: 17

>> s.name
ans =
Meier

>> s.alter
ans =
    17

>> s.alter ^ 2
ans =
   289
```

Es wird eine Strukturvariable s angelegt, die als Komponenten eine Zeichenkette und eine Zahl aufnehmen soll. Jede Komponente wird durch einen eigenen Namen angesprochen, der zusammen mit einem '.' an den Namen der Strukturvariablen angehängt wird. Die einzelnen Komponenten können dann wie ganz normale Variablen benutzt werden.

Ein großer Vorteil von Strukturen ist ihre Erweiterbarkeit, d.h. es können einfach neue Komponenten hinzugefügt werden. Außerdem können mit Strukturen leicht komplexe Datenstrukturen wie verkettete Listen oder Bäume erzeugt werden.

Kapitel 4

Grafische Ausgabe

MATLAB bietet eine Reihe von Möglichkeiten zur grafischen Ausgabe. Wir werden hier nur zweidimensionale Grafiken betrachten. Mit dem Befehl `plot(t,y)` können Polygonzüge durch gegebene (t, y) Wertepaare gezeichnet werden. Um den Graphen von $\sin(t)$ im Intervall $[0, 2\pi]$ darzustellen, können folgende Kommandos benutzt werden:

```
>> t = [0:30]/30 * 2*pi;
>> y = sin(t);
>> plot(t,y)
```

Mit dem ersten Befehl werden die t-Werte zwischen 0 und 2π erzeugt. Der zweite Befehl generiert die zugehörigen Funktionswerte. Mit `plot(t,y)` öffnet sich ein neues Fenster und der Polygonzug wird blau in ein Koordinatensystem eingezeichnet.

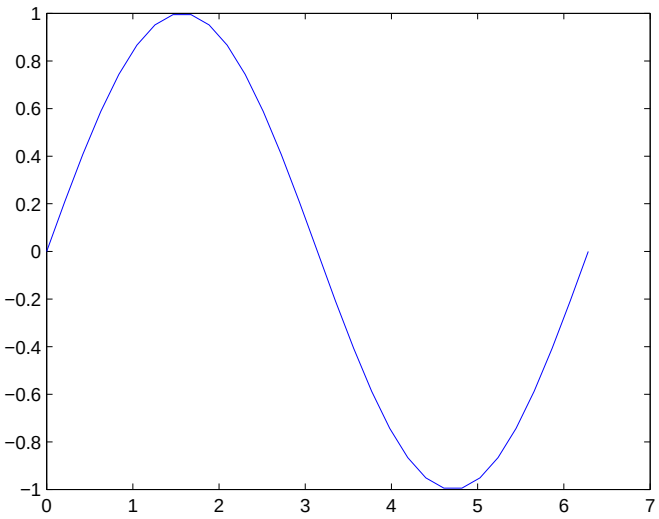


Abbildung 4.1: Sinusplot

Erzeugt man anschließend einen zweiten Satz von Funktionswerten z.B. für $\cos(t)$ und plottet den neuen Graphen, so erscheint dieser im gleichen Grafikfenster, aber die alte Kurve wird vorher gelöscht. Um beide Kurven in ein Grafikfenster zu zeichnen, muss man entweder den Befehl `hold on` oder `hold all` verwenden.

Nach `hold on` werden alle `plot` Ausgaben (in der gleichen Farbe) ins selbe Koordinatensystem im momentan aktiven Grafikfenster gezeichnet. Das kann durch `hold off` wieder rückgängig gemacht werden, d.h. nach `hold off` wird bei jedem `plot` Befehl der Fensterinhalt zuerst gelöscht. Um mehrere Kurven in einem Grafikfenster darzustellen kann man alternativ auch den Befehl `hold all` verwenden. Wenn also nacheinander erst der sin und dann der cos in ein Fenster gezeichnet werden soll, dann kann man folgendermaßen vorgehen:

```
>> t = [0:30]/30 * 2*pi;
>> y = sin(t);
>> plot(t,y);
>> z = cos(t);
>> hold all
>> plot(t,z);
```

Man erhält die beiden Kurven in unterschiedlicher Farbe.

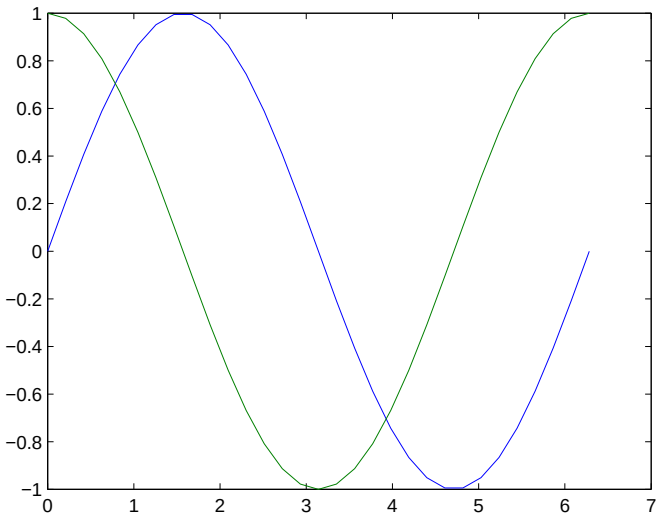


Abbildung 4.2: Sinus- und Cosinusplot

Möchte man selbst die Wahl der Farben übernehmen, so kann man dem `plot` Befehl einen Parameter für die Farbe hinzufügen:

```
>> plot(t,y,'r') %plottet rote Kurve
>> plot(t,y,'y') %plottet gelbe (yellow) Kurve
```

Neben den Farben kann man auch die Linienstärke und den Linientyp definieren. So erhält man mit

```
>> plot(t,y,'Color','g','LineStyle',':','LineWidth'
   ,4)
```

folgende Kurve:

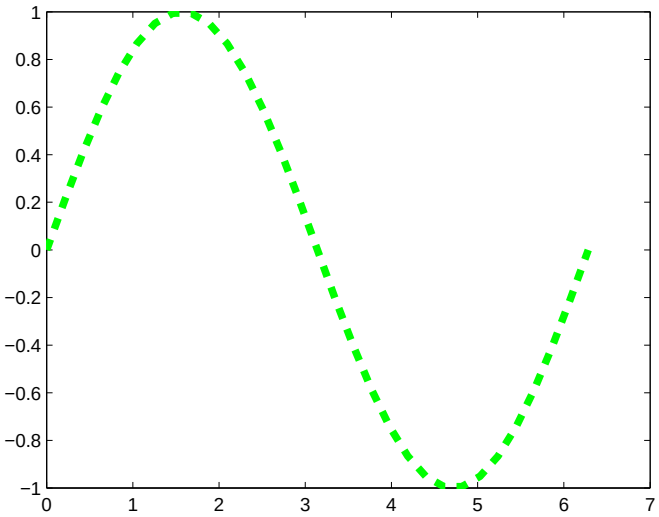


Abbildung 4.3: Sinus mit veränderter Strichstärke

Zu weiteren Optionen zum `plot` Befehl siehe

```
>> help plot
```

Wie wir in den obigen Grafiken gesehen haben, werden die Achsen von MATLAB automatisch skaliert. Mit dem Befehl `axis`

```
>> tmin = 0;
>> tmax = 2*pi;
>> ymin = -1.5
>> ymax = 1.5;
>> axis([tmin,tmax,ymin,ymax]);
```

kann man die Skalierung der t bzw. y-Achse ändern. Weiterhin kann man mit `title`

```
>> title('Sinuskurve');
```

die Grafik mit einen Titel und mit `xlabel`, `ylabel`

```
>> xlabel('t-Achse');
>> ylabel('y-Achse');
```

mit einer Achsenbeschriftung versehen.

Nach dem Zeichnen von Grafiken können diese auch noch mit dem `legend` Befehl beschriftet werden.

Neben `plot` kann man für Funktionen auch den Befehl `fplot` verwenden, mit dem man eine Funktion in einem Intervall plotten kann. Hierbei muss man die zu plottende Funktion an `fplot` als Funktionszeiger übergeben (siehe dazu auch das Kapitel Funktionen):

```
>> funk = @sin;
>> fplot(funk,[-2*pi 2*pi])
```

Neben `plot` und `fplot` stellt MATLAB noch eine Menge weiterer Funktionen zum erzeugen von Grafiken zur Verfügung. Für 3-dimensionale Ausgaben verwendet man die Funktion `plot3`. Details dazu sind mit der `help` Funktion zu erfahren.

Eine Ausgabefunktion, die für die digitale Signalverarbeitung wichtig ist, ist die Funktion `stem` mit der man sogenannte Lolli-Plots erzeugen kann.

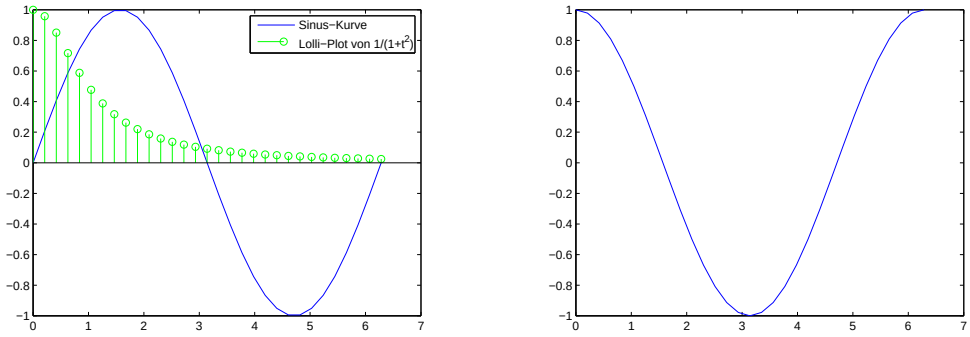
Zum Verwalten mehrerer Grafikfenster dient der `figure` Befehl. In MATLAB erfolgen Grafikausgaben in das augenblicklich aktive Grafikfenster. Ist keines vorhanden, so wird ein neues erzeugt und dieses als aktives Fenster angesehen. Verschiedene Fenster werden durch ihre Fenster-Nummer angesprochen. Der Aufruf von `figure(n)` bewirkt nun folgendes

- ist ein Fenster mit Nummer n noch nicht vorhanden, so wird ein neues, leeres Fenster erzeugt und als aktives Fenster für die folgenden Ausgaben benutzt
- existiert ein Fenster mit Nummer n schon, so wird dieses das aktive Fenster

Mit der Befehlsfolge

```
>> t = [0:30]/30 * 2*pi;
>> y = sin(t);
>> figure(1);
>> plot(t,y);
>> z = cos(t);
>> figure(2);
>> plot(t,z);
>> figure(1);
>> hold on;
>> u = 1 ./ (1+t.*t);
>> stem(t,u,'g');
>> legend('Sinus-Kurve','Lolli-Plot von 1/(1+t^2)');
```

erhalten wir folgende Ausgaben:



Fenster 1 Fenster 2

Abbildung 4.4: Benutzung mehrerer Plotfenster

Man beachte, dass die `hold on` und `hold off` Anweisungen für jedes Fenster getrennt gelten. Nicht mehr benötigte Grafikfenster werden mit dem Befehl `close` geschlossen.

Eine andere, sehr einfache Möglichkeit, Funktionen darzustellen liefert `ezplot`. Man übergibt einfach die Funktion in Hochkommas an `ezplot`. Es wird die Funktion im Intervall $[-2\,\pi, 2\,\pi]$ geplottet.

```
>> ezplot('1+2*cos(w)')
```

liefert:

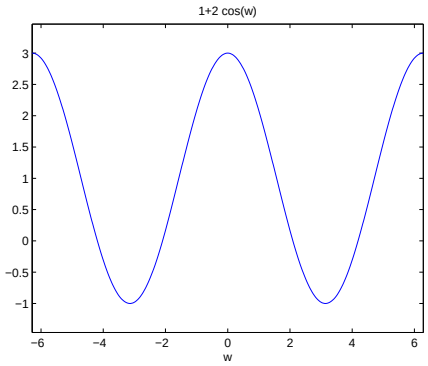


Abbildung 4.5: Darstellung mit ezplot

Bei der Verwendung der Funktion `ezplot` muss man beachten, dass man die Darstellungseigenschaften nicht wie bei `plot(t,y,'r')` in Hochkommas angeben kann. Möchte man nun z.B. die Farbe oder die Linienstärke ändern, dann muss man bei `ezplot` folgendermaßen vorgehen: Zunächst wird das zu plottende Objekt in einer Variablen h gespeichert ("handle"). Anschließend kann man dann die Grafikeigenschaften mithilfe von h ändern.

```
>> h=ezplot('1+2*cos(w)')
>> set(h,'Color','g')
>> set (h, 'LineWidth',4)
```

liefert folgende Darstellung

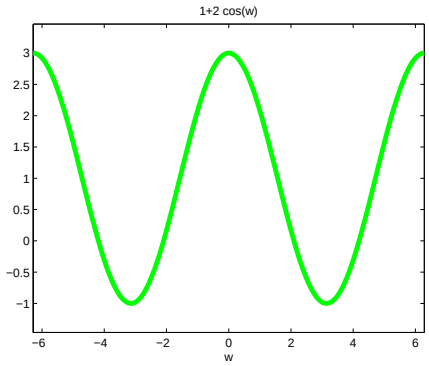


Abbildung 4.6: Änderung der Darstellungseigenschaften

Der Inhalt eines Grafikfensters kann in vielen gängigen Grafikformaten (ps, jpeg, eps etc.) abgespeichert werden.

Zur Darstellung der komplexen Zahlen kann man die Funktion `compass` verwenden. Mit

```
>> z=(0.9*exp(j*pi/6)).^[0:10];
>> compass(z)
```

kann man die Potenzen der komplexen Zahlen darstellen (vgl. Vorlesung Teil 2, Seite 50)

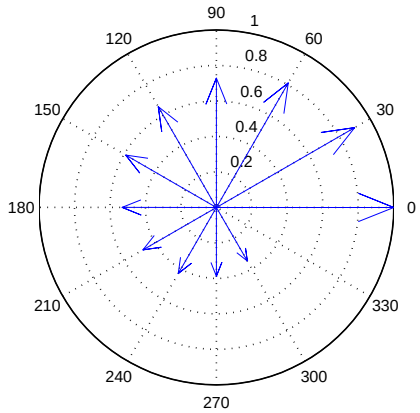


Abbildung 4.7: Potenzen komplexer Zahlen

Neben den hier beschriebenen zwei dimensionalen Darstellungen bietet MATLAB noch eine Vielzahl von weiteren Möglichkeiten zur grafischen Ausgabe.

Im Help Browser unter `MATLAB - User Guide - Graphics - Plots and Plotting Tools - Figures, Plots, and Graphs` findet man eine Menge zusätzlicher Informationen:

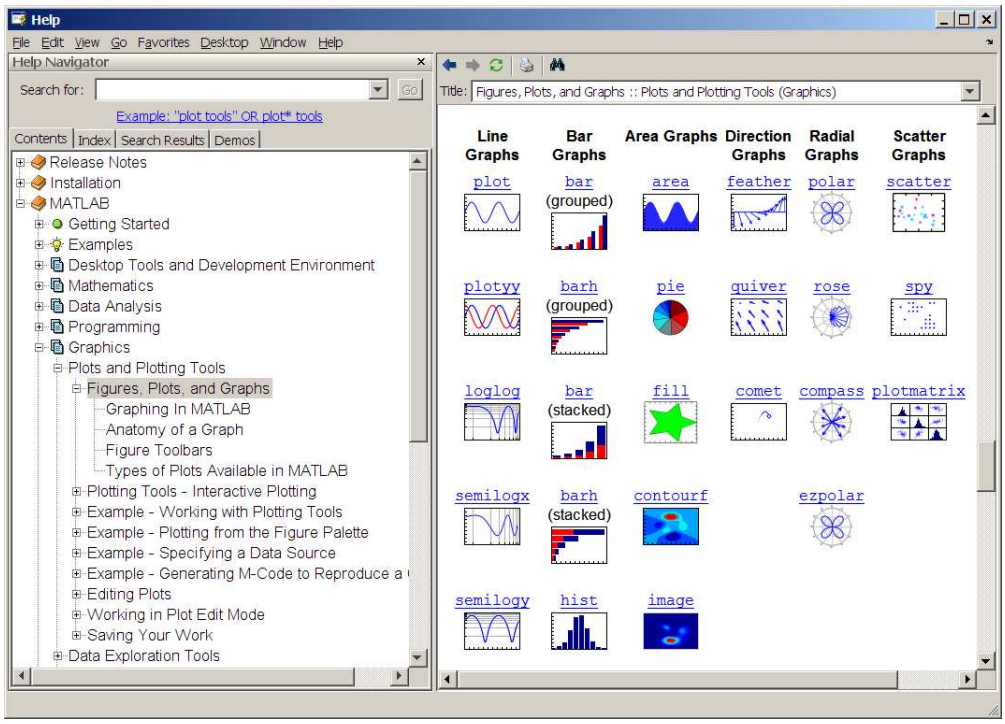


Abbildung 4.8: Figuren, Plots und Graphen

Kapitel 5

m-Files

5.1 Grundlagen

Sollen umfangreichere Berechnungen mehrfach auf verschiedene Datensätze angewandt werden, so ist es sinnvoll, die Befehle nicht jedesmal neu auf der Kommandozeile einzutippen sondern sie zur wiederholten Ausführung abzuspeichern. Dies geschieht in MATLAB durch einfaches anlegen so genannter m-Files, in die dann die einzelnen Befehle als Text eingetragen werden. Dabei sind zwei grundsätzlich verschiedene Varianten möglich:

Skripte. Skripte sind eine bloße Aneinanderreihung von Befehlen. Der Aufruf eines Skriptes hat den gleichen Effekt, als ob alle darin befindlichen Befehle erneut im Kommandofenster eingegeben würden.

Funktionen. Funktionen gruppieren ebenfalls mehrere Befehle zusammen, besitzen aber im Gegensatz zu Skripten noch Ein- und Ausgabeparameter sowie einen lokalen Datenbereich, um z.B. Zwischenergebnisse abzulegen.

Auf beide Konstrukte wird in den folgenden Abschnitten genauer eingegangen.

5.2 Skripte

Als Beispiel wollen wir ein **Skript** erstellen, das die Summe und die Differenz der Quadrate der Variablen a, b berechnet.

Dazu legen wir ein neues m-File mit dem Namen `summdiff.m` an, indem wir mit der Maus ins Fenster `Current Directory` gehen, die rechte Maustaste drücken und in dem dann erscheinenden Menü unter `New` die Option `M-File` wählen. Daraufhin erscheint im Fenster `Current Directory` ein neuer Dateieintrag mit dem Namen `Untitled1.m`. Dort tragen wir den von uns gewünschten Namen (also `summdiff.m`) ein.

Durch einen Doppelklick auf den Dateinamen öffnet sich die Datei im MATLAB Editor und kann somit bearbeitet werden.

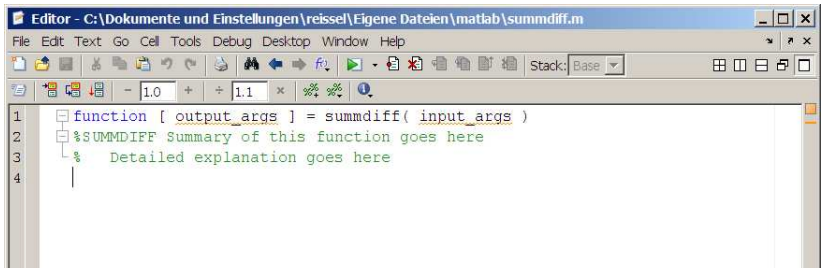


Abbildung 5.1: Default-Inhalt m-File

Standardmäßig ist der in der Abbildung gezeigte Funktionsrumpf in der Datei enthalten, den wir aber für unser Skript nicht benötigen und deshalb vollständig löschen.

Ein Skript sollte immer mit einer (oder mehreren) Kommentarzeilen beginnen, die eine kurze Dokumentation seiner Funktionalität enthalten. Anschließend tragen wir Zeile für Zeile unsere Befehle ein. Das fertige Skript sieht dann wie folgt aus:

```
% Summe und Differenz der Quadrate der Variablen a
    und b
a2 = a^2;
b2 = b^2;
s = a2 + b2
d = a2 - b2
```

Bevor wir es jetzt benutzen können, müssen wir es auf jeden Fall im Editor noch abspeichern (z.B. durch Klicken auf das Diskettensymbol). Wurde die Datei seit der letzten Änderung noch nicht gespeichert, so kann man das an einem $*$ hinter dem Dateinamen in der Titelleiste des Editorfensters erkennen.

Das fertige Skript kann nun im Kommandofenster über den Namen des Files in dem es steht gestartet werden, in unserem Fall also `summdiff`. Bevor das aber Sinn macht, müssen wir zuerst noch den Variablen a, b, die verarbeitet werden sollen, Werte zuweisen. Alles in allem erhalten wir dann:

```
>> a = 5, b= 3
a =
     5
b =
     3

>> summdiff
s =
    34
d =
    16

>> b = 7
b =
```

```
        7

>> summdiff
s =
      74
d =
     -24
```

Folgende Aspekte sind zu beachten:

- vor Aufruf des Skripts müssen alle vom Skript benötigten Daten, also a, b, im Workspace vorhanden sein
- die vom Skript angelegten Variablen $a2, b2, s, d$ sind auch nach Beendigung des Skripts noch im Workspace vorhanden
- werden die Befehle im Skript mit ; abgeschlossen, so erfolgt keine Anzeige der jeweiligen Ergebnisse im Kommandofenster
- mit dem Befehl `help summdiff` erhalten wir als Ausgabe

  ```
  >> help summdiff
    Summe und Differenz der Quadrate der Variablen a
        und b
  ```

  d.h. das MATLAB **Hilfesystem** durchsucht unsere Skripte und gibt die führenden Kommentarzeilen aus, so dass wir unsere Skript genauso komfortabel dokumentieren können wie die eingebauten MATLAB Funktionen.

5.3 Funktionen

Im letzten Abschnitt haben wir gesehen, dass wir mit Skripten komplexere Aufgaben automatisieren können. Betrachten wir nochmal unser Skript `summdiff` von oben, so werden direkt einige Nachteile offensichtlich:

- `summdiff` arbeitet immer nur auf den Variablen a, b, d.h. wollen wir damit Summe und Differenz der Variablen x, y berechnen, so müssen wir deren Werte erst in a, b umspeichern
- dasselbe gilt für die Ergebnisse, die immer in den Variablen s, d stehen
- außerdem sind die vom Skript benutzen Hilfsvariablen $a2, b2$ immer im aktuellen Workspace vorhanden, obwohl sie uns eigentlich nicht interessieren

Mit zunehmend komplexeren Aufgaben werden diese Eigenschaften immer störender.

Mit Funktionen können diese Probleme vermieden werden. **Funktionen** sind prinzipiell Skripte, die über spezielle **Ein- und Ausgabeparameter** verfügen. Darüber hinaus besitzen sie einen eigenen Workspace (der vom normalen Workspace

strikt getrennt ist) in dem **lokale Variablen** angelegt werden können, die nach Beendigung der Funktion wieder beseitigt werden.

Als Beispiel erstellen wir eine Funktion `fsummdiff`, die Summe und Differenz der Quadrate zweier beliebiger Zahlen berechnet. Dazu legen wir wie oben beschrieben eine neue Datei `fsummdiff.m` an und tragen dort die folgende Befehle ein:

```
function [s,d] = fsummdiff(a,b)
% [s,d] = fsummdiff(a,b)
%    berechnet Summe und Differenz zweier Zahlen
%
%    a,b  : Eingabegrößen
%    s    : a^2 + b^2
%    d    : a^2 - b^2
a2 = a^2;
b2 = b^2;

s = a2 + b2;
d = a2 - b2;
```

Die einzelnen Befehle haben dabei folgende Bedeutung:

- mit der ersten Zeile legen wir fest, welche Ein- und Ausgabeparameter unsere Funktion besitzt (Aufrufkopf, Schnittstelle)
- die (formalen) Parameter s, d, a, b sind lokale Variablen in der Funktion, d.h. sie haben nichts mit eventuell gleichnamigen Variablen im Workspace zu tun
- die Kommentare ab Zeile 2 werden wieder durch das Hilfesystem ausgewertet
- die Hilfsgrößen $a2, b2$ sind lokale Variablen, sie sind nur innerhalb der Funktion verwendbar und werden gelöscht, wenn die Funktion verlassen wird
- in den letzten beiden Zeilen werden den Ausgabeparametern s, d die gewünschten Werte zugewiesen

Die Funktion `fsummdiff` kann nun wie folgt benutzt werden:

```
>> x = 1; y = 2;
>> [u,v] = fsummdiff(x,y)
u =
     5
v =
    -3

>> [h,k] = fsummdiff(3,4)
h =
    25
k =
    -7
```

```
>> fsummdiff(1,1+2)
ans =
     10
```

Wir stellen also folgendes fest:

- als Ausgangs- bzw. Zielgrößen können beliebige Variablen benutzt werden
- die Hilfsvariablen für Zwischenrechnungen sammeln sich nicht im globalen Workspace an, da sie lokal im Workspace der Funktion liegen und dieser immer nach Beendigung der Funktion beseitigt wird
- die Funktion liefert zwar zwei Ergebnisgrößen ab, `ans` enthält aber immer nur den ersten Wert

Weiterhin sind folgende Aspekte wichtig:

- Funktionen verändern an sie übergebene Eingabeparameter nicht, sie arbeiten immer auf Kopien der übergebenen Daten (**Call-by-Value**)
- Lokale Variablen der Funktion sind von außerhalb nicht sichtbar, umgekehrt kann eine Funktion auch nicht auf Variablen zugreifen, die außerhalb definiert sind (Ausnahme: siehe `global` Befehl)

Neben Zahlen (Matrizen) kann man an Funktionen auch andere Funktionen als Parameter übergeben (**Funktionszeiger**, function handles). Als Beispiel betrachten wir eine Funktion `wtab`, die Wertetabellen für beliebige Standardfunktionen erstellen soll

```
function wert = wtab(funk)
% erstellt Wertetabelle

x = [0:4];

wert = feval(funk,x);
```

Der Aufruf

```
>> wtab(@cos)
ans =
    1.0000    0.5403   -0.4161   -0.9900   -0.6536

>> wtab(@sin)
ans =
         0    0.8415    0.9093    0.1411   -0.7568
```

liefert Funktionswerte von cos bzw. sin an $x = 0, 1, 2, 3, 4$. Mit `@cos` übergeben wir an `wtab` als Parameter `funk` einen Zeiger auf die Funktion cos, nicht einen einzelnen Funktionswert. Innerhalb `wtab` wird die Funktion über den Parameter `funk` angesprochen und mit Hilfe des `feval` Befehls dann an der Stelle x ausgewertet.

Kapitel 6

Strukturierte Programmierung

6.1 Grundlagen

Häufig erfordert die Praxis die Implementierung relativ komplexer Algorithmen.

Neben der Aufteilung umfangreicher Aufgabenstellungen auf mehrere (sinnvoll gewählte) Teil-Funktionen stehen in höheren Programmiersprachen drei prinzipielle Konstrukte zur Algorithmendefinition zur Verfügung:

Sequenz. Bei einer **Sequenz** handelt es sich um eine einfache Aneinanderreihung von Einzelbefehlen. Sequenzen haben wir bereits zur Definition der Skripte und Funktionen im letzten Abschnitt benutzt.

Verzweigung. In einer Sequenz wird eine bestimmte Anzahl von Befehlen immer in der gleichen Reihenfolge abgearbeitet. Oft ist es aber erforderlich in Abhängigkeit von gewissen Voraussetzungen (z.B. Wert einer bestimmten Variablen) unterschiedliche Befehle auszuführen. **Verzweigungen** stellen solche "Weichen" dar.

Schleifen. **Schleifen** dienen dazu, gewisse Befehle mehrfach zu wiederholen.

Auf die in MATLAB zur Verfügung stehenden Verzweigungen und Schleifen wird in den folgenden Abschnitten genauer eingegangen.

6.2 Verzweigungen

Als erstes Beispiel wollen wir einen einfachen Taschenrechner implementieren, Er soll folgende Funktionalität haben:

- wir übergeben zwei Zahlen x, y sowie eine string o
- enthält o das Zeichen '+', dann liefert die Funktion $x + y$ als Ergebnis
- enthält o irgend ein anderes Zeichen, dann liefert die Funktion `NaN` als Ergebnis

In der Funktion müssen wir also in Abhängigkeit vom Wert von o unterschiedliche Befehle ausführen, d.h. das Programm muss verzweigen. Eine Implementierung der zugehörigen Funktion `rechner1` sieht wie folgt aus:

```
function z = rechner1(x,y,o)

if (o == '+')
    z = x + y;
else
    z = NaN;
end
```

Mit der **vollständigen Verzweigung** `if...else...end` erreichen wir folgendes:

- falls die Bedingung nach dem `if` zutrifft, also `o` den Wert '+', dann wird die Anweisung `z = x + y` ausgeführt
- ist dies nicht der Fall, dann wird die Anweisung nach `else` ausgeführt

Zwischen `if...else` bzw. `else...end` können jeweils auch mehrere Anweisungen stehen.

Um die Bedingungen in der `if` Anweisung zu formulieren, können folgende **Vergleichsoperatoren** benutzt werden

`==`	gleich
`~=`	ungleich
`<`	kleiner
`>`	größer
`<=`	kleiner oder gleich
`>=`	größer oder gleich
`&`	und
`|`	oder
`~`	nicht

Weitere Details erhält man z.B. mit `help ==`.

Die **unvollständige Verzweigung** erhalten wir aus der vollständigen durch weglassen des `else` Teils. Auch damit können wir unseren Taschenrechner implementieren

```
function z = rechner2(x,y,o)

z = NaN;

if (o == '+')
    z = x + y;
end
```

Nun wollen wir den Taschenrechner erweitern und auch die Subtraktion von x, y berücksichtigen. Dazu kann man z.B. zwei vollständige Verzweigungen schachteln

```
function z = rechner3(x,y,o)

if (o == '+')
    z = x + y;
else
    if (o == '-')
        z = x - y;
    else
        z = NaN;
    end
end
```

was aber sehr unübersichtlich ist. Für diese Situation existiert in MATLAB der `elseif` Befehl

```
function z = rechner4(x,y,o)

if (o == '+')
    z = x + y;
elseif (o == '-')
    z = x - y;
else
    z = NaN;
end
```

Wenn wir nun weitere Grundrechenarten hinzufügen, so stoßen wir auch damit an die Grenzen der Übersichtlichkeit. Abhilfe schafft hier die **Mehrfachverzweigung**: `switch`:

```
function z = rechner5(x,y,o)

switch o
    case '+'
        z = x+y;
    case '-'
        z = x-y;
    case '*'
        z = x*y;
    case '/'
        z = x/y;
    case {'^','**'}
        z = x^y;
    otherwise
        z = NaN;
end
```

Die Mehrfachverzweigung arbeitet folgendermaßen:

- hat `o` einen Wert, der nach einem `case` Statement auftritt, so werden die dem `case` Statement folgenden Anweisungen ausgeführt

- taucht der Wert von `o` in keinem `case` Statement auf, dann werden die Anweisungen nach `otherwise` ausgeführt

Als letzte Verzweigungsstruktur existiert in MATLAB `try...catch...end`. Sie dient dazu, Fehler während der Programmausführung abzufangen. Berechnen wir mit Hilfe der Standardfunktion `dot` das Skalarprodukt zweier Vektoren unterschiedlicher Länge, so liefert MATLAB eine Fehlermeldung

```
>> dot(1,[1,2])
??? Error using ==> dot at 30
A and B must be same size.
```

Tritt solch ein Fehler in einer Funktion auf, dann wird die Programmausführung sofort abgebrochen, was oft unerwünscht ist.

Wollen wir solche Fehler abfangen und in diesem Fall als Ergebnis 0 zurück übergeben, dann können wir das wie folgt erreichen:

```
function z = trycatch(x,y)

try
    z = dot(x,y);
catch
    z = 0;
end
```

Der kritische Befehl wird zwischen `try` und `catch` eingeschlossen. Es wird zunächst versucht, diesen Befehl auszuführen. Tritt dabei ein Fehler auf, dann wird die Ausführung des Befehls abgebrochen und mit der Ausführung der Anweisungen zwischen `catch` und `end` das Programm fortgesetzt.

6.3 Schleifen

Zur Wiederholung von Befehlssequenzen gibt es in MATLAB zwei Schleifentypen, die `for`-Schleife (Zählschleife) und die `while`-Schleife (abweisende Wiederholung).

Als Beispiel für eine Zählschleife betrachten wir

```
for k = 1:3
    disp(k^2);
end
```

Im Schleifenkopf wird die Schleifenvariable `k` definiert (Laufvariable), sowie der Bereich über den sie laufen soll (`1:3` entspricht `1,2,3`). Es finden also `3` Schleifendurchläufe statt, wobei in den einzelnen Durchläufen die Variable `k` nacheinander die Werte `1,2,3` hat, d.h. als Ausgabe erhalten wir über den Befehl `disp` die Zahlen `1,4,9`. Analog liefern

```
for k = [2,4,-1]
    disp(k^2);
end
```

die Ausgabe `4,16,1` bzw.

```
for k = 2:-1:0
    disp(k^2);
end
```

die Ausgabe `2,1,0`. Ist der angegebene Bereich leer, wie z.B. bei

```
for k = 1:0
    disp(k^2);
end
```

so werden Befehle im Schleifenrumpf gar nicht ausgeführt, d.h. die `for`-Schleife ist abweisend.

Im Gegensatz dazu gibt es bei der `while`-Schleife keine gesonderte Schleifenvariable:

```
k = 0;

while(k<3)
    disp(k^2);
    k = k + 2;
end
```

Als Ausgabe erhält man 0 und 4. Solange die Bedingung nach `while` erfüllt ist, werden die Befehle im Schleifenrumpf ausgeführt. Man muss also darauf achten, dass die in der Bedingung auftretenden Variablen innerhalb der Schleife verändert werden, weil man ansonsten leicht Endlosschleifen erhalten kann.

Bei beiden Schleifentypen kann man mit `continue` direkt zum nächsten Schleifendurchgang springen bzw. mit `break` die Schleife augenblicklich beenden und die Programmausführung mit dem nächsten Befehl nach der Schleife fortsetzen. Die Befehle `continue` und `break` sollten nicht zu häufig verwendet werden, da sie die Programme sehr unübersichtlich machen können.

Kapitel 7

Verschiedenes

7.1 Datenhaltung

Benutzerdefinierte Skripte und Funktionen werden wie oben beschrieben grundsätzlich in externen Textdateien (m-Files) im *Current Directory* abgespeichert und sind somit auch zu einem späteren Zeitpunkt verfügbar.

Möchte man auch Funktionen benutzen, die in einem anderen Directory liegen (eventuell auch Toolboxen, die ein anderer Benutzer geschrieben hat), so muss unter `File-SetPath` das entsprechende Verzeichnis mit den zugehörigen Unterverzeichnissen eingetragen werden.

Daten werden im Allgemeinen in binären mat-Files abgelegt. Sie können mit dem Befehl `save` im Arbeitsverzeichnis angelegt werden. Durch `save` ohne zusätzliches Argument wird eine Datei `matlab.mat` erzeugt, die alle augenblicklich vorhandenen Daten enthält. Zu einem späteren Zeitpunkt können mat-Files mit dem Befehl `load` wieder geladen werden. Beide Befehle können auch gezielt auf einzelne Datensätze angewendet werden. Details dazu sind per `help` zu erfahren.

Variablen, die im *Workspace* angelegt sind, können mit `clear` beseitigt werden.

7.2 Debugger

MATLAB besitzt zur Fehlersuche einen sehr komfortablen **Debugger**. Die einfachste Möglichkeit, den Debugger zu benutzen, ist die folgende:

Befindet man sich im MATLAB Editor in dem File, dass man debuggen möchte, so kann man durch Anklicken des Strichs hinter der Zeilennummer einen **Break-Point** setzen. Optisch zu erkennen ist der Break-Point durch einen roten Punkt. Wird nun die Funktion aufgerufen, so wird sie an der markierten Stelle angehalten und im *Command Window* erscheint folgendes

```
8     a2 = a^2;
K>>
```

Die 8 gibt die Zeilennummer von der Zeile an, in der sich der Break-Point befindet und der Prompt `K>>` signalisiert, dass man sich im Debug Modus befindet. Dort kann man wie gewohnt Variablen abfragen bzw. ihre Werte ändern.

```
K>> a , b
a =
     10
b =
      2

K>> a=3
a =
      3

K>> a , b
a =
      3
b =
      2
```

Im Editor erscheint in der Zeile 8, also dort wo die Funktion angehalten wurde, ein grüner Pfeil

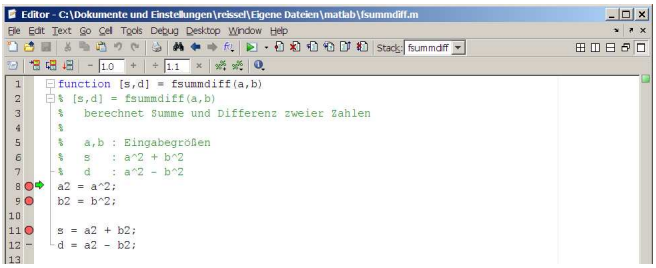


Abbildung 7.1: File mit Debugger

Über den Menüpunkt `Debug-Step` kann man nun das Programm fortsetzen, wobei jeweils nur der nächste Befehl ausgeführt wird. Handelt es sich dabei um den Aufruf einer weiteren Funktion, so stoppt der Debugger die Programmausführung erst, nachdem die aufgerufene Funktion vollständig beendet wurde.

Möchte man dagegen in die aufgerufene Funktion hineinspringen, so muss man `Debug-StepIn` verwenden.

Mit `Debug-Continue` wird das Programm bis zum nächsten Break-Point fortgesetzt.

Einen gesetzten Break-Point entfernt man durch Anklicken des roten Punktes. Mit `Debug-ExitDebugMode` kann man den Debugger verlassen.

Neben der Verwendung des Debuggers über die Benutzeroberfläche des MATLAB–Editors, ist es auch möglich, direkt in das m-File die Befehle `dbstop`, `dbstep`, `dbquit` usw. einzugeben. Weitere Informationen zu `Debug` Befehlen erhält man mit

```
>> help debug
```

Bei der Fehlersuche unterstützt MATLAB den Benutzer durch sehr ausführliche Fehlermeldungen. Erhält man z.B. die Fehlermeldung

```
Error in ==> fsummdiff at 2
a2 = a^2;
```

so springt man durch Anklicken von `fsummdiff` direkt in die fehlerhafte Zeile der Funktion und kann dort die entsprechenden Korrekturen vornehmen.

7.3 Effizientes Programmieren

Wie in der Einleitung bereits erwähnt, arbeitet MATLAB auf Matrizen und besitzt eine Reihe von eingebauten Funktionen, die teilweise vom Benutzer auch leicht selbst programmiert werden könnten. Allerdings sind die eingebauten Funktionen extrem effizient implementiert, so dass sie in der Regel sehr viel schneller arbeiten als selbst programmierter Code. Deshalb sollte man (wo immer das möglich ist) auf sie zurückgreifen. Dies wird am folgenden Beispiel deutlich.

```
n = 500;
A = rand(n,n);
B = rand(n,n);
C = zeros(n,n);
tic;
for ii = 1:n
    for jj = 1:n
        for kk = 1:n
            C(ii,jj) = C(ii,jj) + A(ii,kk)*B(kk,jj);
        end
    end
end
toc
D = zeros(n,n);
tic;
D = A*B;
toc
```

Dieses Skript berechnet das Produkt der Matrizen A und B zunächst über drei `for`-Schleifen und dann mit dem eingebauten Befehl zur Multiplikation zweier Matrizen.

Zur Messung der Laufzeit werden die Funktionen `tic` und `toc` benutzt. Dabei stellt `tic` eine Stoppuhr auf Null und startet sie und `toc` hält die Stoppuhr an und gibt die benötigte Zeit in Sekunden aus.

Für obiges Beispiel erhalten wir:

```
>> matmul
Elapsed time is 3.145798 seconds.
Elapsed time is 0.054338 seconds.
```

Es ist offensichtlich, dass die eingebaute Matrixmultiplikation um einen Faktor von ungefähr 60 schneller ist.

Diese Tatsache sollte man, wenn man in MATLAB effektiv programmieren will, immer im Auge behalten. Ist es möglich, anstelle von Schleifen, die eingebauten MATLAB Funktionen zu benutzen, so sollte dies auch getan werden.

Kapitel 8

Anwendungen

8.1 Nullstellen von Polynomen

Nullstellen eines Polynoms können mit der Funktion `roots` bestimmt werden. Sollen die Nullstellen des Polynoms z^3-2z^2+2z-1 berechnet werden, muss man zunächst die Koeffizienten den Polynoms (angefangen mit dem Koeffizienten vor der höchsten Potenz) in einem Vektor abspeichern.

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

Anschließend kann man dann mit der Funktion `roots` die Nullstellen bestimmen.

```
>> roots(z)
ans =
   1.0000
   0.5000 + 0.8660i
   0.5000 - 0.8660i
```

8.2 Diskrete Faltung

Die diskrete Faltung lässt sich mit der Funktion `conv` bestimmen. Soll das diskrete Signal

$$s[n] = \delta[n] + \delta[n-1] + \delta[n-2]$$

mit sich selbst gefaltet werden, so muss man zunächst den Vektor mit den entsprechenden Werten erzeugen:

```
>> z = [1 1 1]
z =
     1     1     1
```

Die diskrete Faltung kann dann folgendendermaßen berechnet werden:

```
>> conv(z,z)
ans =
     1     2     3     2     1
```

d.h. das man erhält als Ergebnis das Signal

$$s[n] = \delta[n] + 2\delta[n-1] + 3\delta[n-2] + 2\delta[n-3] + \delta[n-4].$$

Das Ergebnis kann mithilfe der Funktion `stem` dargestellt werden. Mit

```
>> x = (1:5)
x =
     1     2     3     4     5
>> stem(x,conv(z,z))
```

erhält man folgende Grafik:

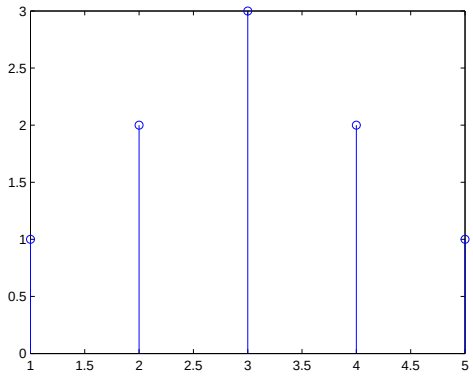


Abbildung 8.1: Faltung

8.3 Amplitudengang und Phasengang

In der Signalübertragung ist es oft hilfreich, wenn man von einer Übertragungsfunktion ihren Betrag und die dazugehörige Phase grafisch in einem Fenster darstellen kann.

Dazu muss man zunächst das Fenster mit `subplot` einteilen. Das erste Argument von `subplot` gibt die Anzahl der Zeilen an, in die das Fenster eingeteilt wird, das zweite Argument die Anzahl der Spalten und das dritte Argument enthält die Nummer der Stelle, in der die Grafik dargestellt werden soll.

So erhält man mit

```
>> subplot(2,1,1);
>> ezplot('abs(1-exp(-i*w))')
>> subplot(2,1,2);
>> ezplot('angle(1-exp(-i*w))')
```

folgende Darstellung

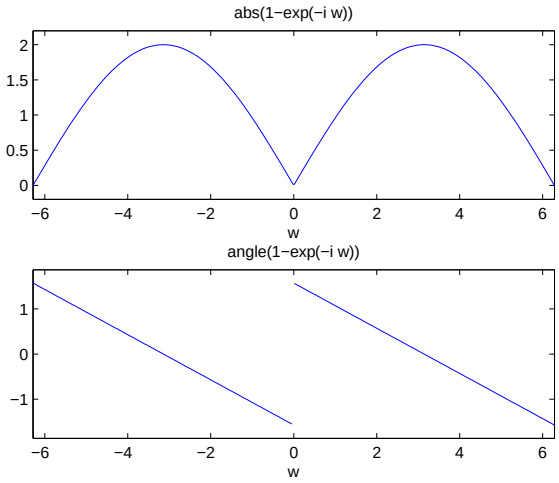


Abbildung 8.2: Amplituden- und Phasengang

Man sieht, dass man in der Funktion `ezplot` auch die Funktionen `abs` und `angle` benutzen kann. Statt `angle` kann man auch die Funktion `atan2` verwenden.

Abbildungsverzeichnis

Index