

TECHNISCHE UNIVERSITÄT  
CHEMNITZ

Fakultät für Elektrotechnik und  
Informationstechnik

Lehrstuhl für Systemtheorie

Prof. Dr. sc. techn. S. F. Bocklich

# **Matlab und Alternativen in der Systemtheorie**

Marcus Obst



# Inhaltsverzeichnis

1 Funktionalität von Matlab.....	1
1.1 Grundlegende Arbeitsweise.....	2
1.1.1 Umgang mit Variablen.....	2
1.1.2 Spezielle Werte.....	2
1.1.3 Desktop und Hilfe.....	3
1.2 Einfache Berechnungen.....	3
1.2.1 Logische Operationen.....	3
1.2.2 Komplexe Zahlen.....	5
1.2.3 Weitere mathematische Funktionen.....	5
1.3 Vektoren und Matrizen.....	6
1.4 Arbeiten mit Gleichungssystemen.....	12
1.5 Polynome.....	13
1.6 Graphische Darstellung.....	16
1.6.1 Grundlagen.....	16
1.6.2 Eigenschaften von Koordinatensystemen und Graphen.....	18
1.6.3 Weitere Plotbefehle.....	20
1.7 Programmierung.....	23
1.7.1 Kontrollstrukturen.....	23
1.7.2 Funktionen.....	25
1.7.3 Script-File vs. Function-File.....	27
1.8 Matlab in der Systemtheorie.....	28
1.8.1 Differentialgleichung.....	28
1.8.2 Systemmodelle mit der Control System Toolbox.....	29
1.8.3 Systemeigenschaften – Verhalten im Zeit- und Frequenzbereich.....	31
1.8.4 Regelkreisstrukturen.....	33
1.8.5 Signalgenerierung.....	37
1.8.6 Statistik.....	38
1.8.7 Fourieranalyse – Spektren.....	42
1.8.8 Filterung.....	43
1.8.9 Interpolation und Regression.....	45
2 Gegenüberstellung.....	48
2.1 Grundlegende Arbeitsweise.....	49
2.1.1 Umgang mit Variablen.....	49
2.1.2 Spezielle Werte.....	49
2.1.3 Desktop und Hilfe.....	50
2.2 Einfache Berechnungen.....	51
2.2.1 Komplexe Zahlen.....	51
2.2.2 Weitere mathematische Funktionen.....	52
2.3 Vektoren und Matrizen.....	53
2.4 Arbeiten mit Gleichungssystemen.....	55
2.5 Polynome.....	56
2.6 Graphische Darstellungen.....	59
2.7 Programmierung.....	62
2.8 Systemtheorie.....	64
2.8.1 Systemmodelle.....	64
2.8.2 Untersuchung von Systemeigenschaften.....	66
2.8.3 Regelkreisstrukturen.....	68
2.8.4 Signalgenerierung.....	69
2.8.5 Statistik.....	70
2.8.6 Fourieranalyse.....	72

2.8.7 Filterung.....	72
2.8.8 Interpolation und Regression.....	73

# 1 Funktionalität von Matlab

Nach dem Start von **Matlab** befindet man sich im sogenannten Desktop (siehe Abbildung 1). Dieser gliedert sich standardmäßig in das *Command Window*, die *History* und den *Workspace*. Die Eingabe der eigentlichen Befehle geschieht dabei im *Command Window* und wird in der *History* fortlaufend protokolliert. Der *Workspace* stellt die aktuell verwendeten Variablen mit ihren Werten übersichtlich da. Über das Menü lassen sich weitere Elemente des Desktops einblenden. Hervorzuheben ist dabei die sehr umfangreiche und ausführliche Hilfe.

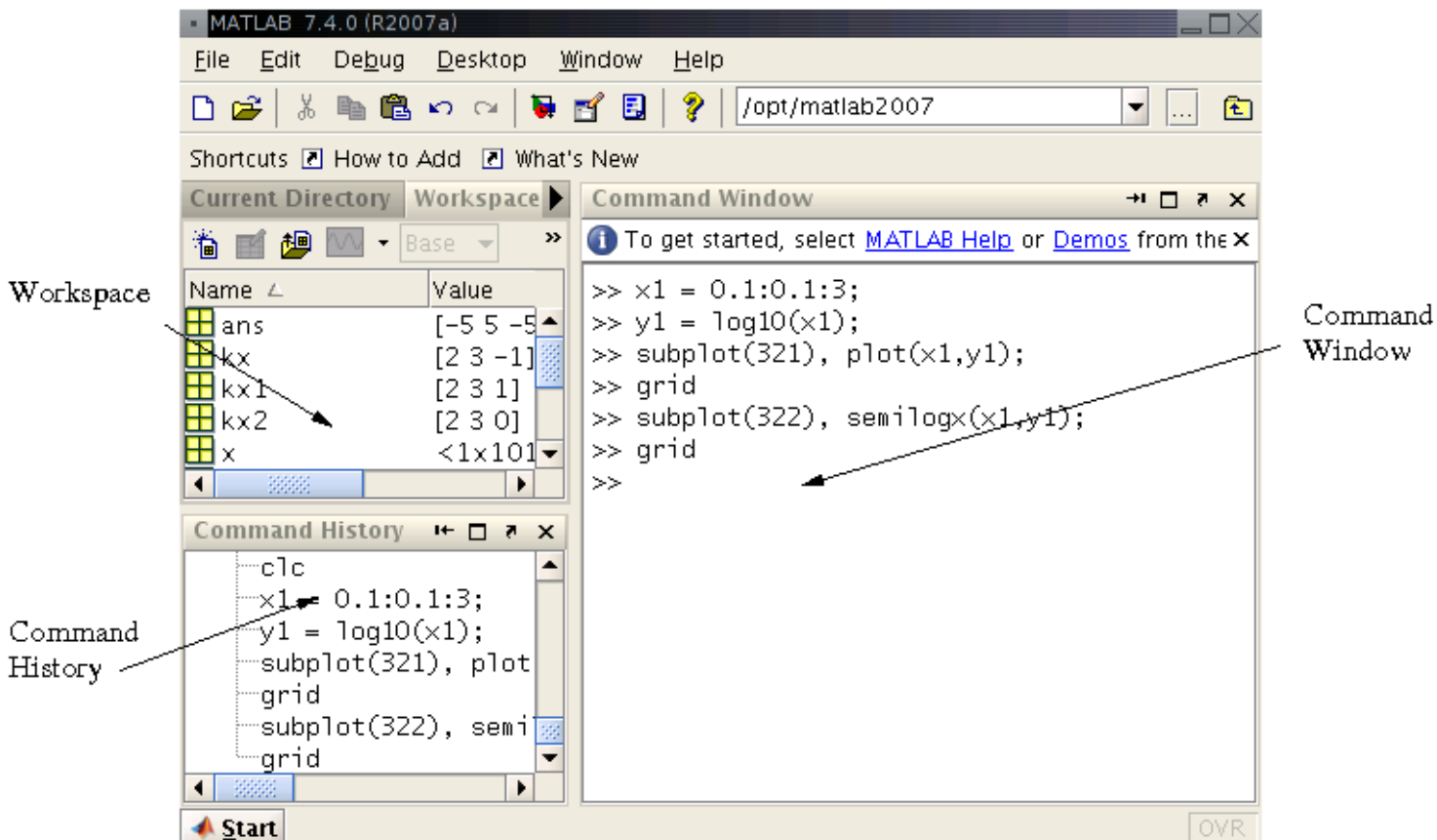


Abbildung 1: Matlab-Desktop und seine einzelnen Elemente

Die meisten der in **Matlab** enthaltenen Befehle funktionieren nach dem aus anderen Programmiersprachen bekannten Prinzip, d.h. man kann einem Befehl bzw. einer Funktion Werte übergeben und bekommt i.d.R. unmittelbar ein Ergebnis geliefert. Zusätzlich gibt es weitere Befehle, die nicht direkt der Verarbeitung dienen, sondern vielmehr den Arbeitsablauf unterstützen. Das könnte z.B. das Anzeigen der Hilfe zu einem Befehl bzw. das Löschen des *Command Window* sein.

Normalerweise arbeitet **Matlab** standardmäßig in einem interaktiven Modus, d.h. das Ergebnis bzw. die Wirkung der eingegebenen Befehle wird sofort sichtbar.

Dezimalzahlen werden in **Matlab** durch einen Punkt abgetrennt. Das Komma hingegen dient als Trennzeichen zwischen Vektor- bzw. Matrixelementen.

## 1.1 Grundlegende Arbeitsweise

### 1.1.1 Umgang mit Variablen

In **Matlab** ist eine explizite Deklaration von Variablen nicht notwendig. Bei der erstmaligen Zuweisung (Definition) einer Variable wird diese auch implizit angelegt. Greift man dagegen auf eine vorher nicht definierte Variable zu, gibt **Matlab** eine Fehlermeldung aus. Eine Liste der aktuellen verwendeten Variablen (alternativ auch im *Workspace* sichtbar) liefern die Befehle `whos` und `who`. `whos` ist dabei die ausführlichere Variante. Zum Löschen aller definierten Variablen benutzt man das Kommando `clear`. Ein optionaler Parameter legt fest, was gelöscht werden soll. Somit lassen sich z.B. nur Variablen mit einem bestimmten Namen oder auch Funktionen löschen.

Befehl	Beschreibung	Beispiel
<code>whos</code>	Ausführliches Anzeigen von Variablen	<code>&gt;&gt; whos</code>
<code>who</code>	Kompaktes Anzeigen von Variablen	<code>&gt;&gt; who</code>
<code>clear</code>	Variablen löschen	<code>&gt;&gt; clear</code>
<code>save Datei</code>	Variablen in Datei speichern	<code>&gt;&gt; save variablen.mat</code>
<code>load Datei</code>	Variablen aus Datei einlesen	<code>&gt;&gt; load variablen.mat</code>

Tabelle 1: Grundlegende Befehle für den Umgang mit Variablen

Um alle definierten Variablen des *Workspace* für eine spätere Sitzung zu speichern, ruft man den Befehl `save` auf. Dieser erwartet als Parameter einen Dateinamen. Das Laden eines gespeicherten *Workspace* funktioniert analog mit `load`.

### 1.1.2 Spezielle Werte

**Matlab** unterstützt eine Reihe spezieller Werte, die sich wie schreibgeschützte Variablen bzw. Funktionen verwenden lassen. Eine Übersicht ist in Tabelle 2 zu finden.

Name	Beschreibung
<code>ans</code>	gespeichertes Ergebnis der letzten Rechnung
<code>eps</code>	Genauigkeit zwischen zwei benachbarten Realzahlen
<code>pi</code>	die Kreiszahl $\pi$
<code>i/j</code>	imaginäre Einheit für komplexe Zahlen
<code>realmin</code>	kleinste positive Realzahl
<code>realmax</code>	größte positive Realzahl
<code>Inf</code>	Repräsentation der Unendlichkeit
<code>NaN</code>	keine Zahl, z.B. das Ergebnis einer Division durch 0

Tabelle 2: Übersicht der in Matlab standardmäßig vorhandenen Werte

### 1.1.3 Desktop und Hilfe

Übergibt man dem Befehl `help` als Parameter eine Funktion so liefert dieser die dazu vorhandene Hilfeseite. Zum Suchen aller Vorkommen eines bestimmten Begriffs, ist der Befehl `lookfor` mit entsprechendem Parameter gedacht.

Der Befehl `diary` schreibt alle folgenden Anweisungen und deren Ausgaben in eine Art Protokoll (standardmäßig ist der Name der Ausgabedatei `diary`). Er funktioniert wie ein Schalter und erlaubt somit auch das zeitweise Aus- bzw. wieder Einschalten (`diary on` bzw. `diary off`).

Um den Inhalt des *Command Window* zu löschen gibt es den Befehl `clc`.

Befehl	Beschreibung	Beispiel
<code>help</code> Funktion	Hilfetext zur angegebenen Funktion ausgeben	<code>&gt;&gt; help sin</code>
<code>lookfor</code> Begriff	Nach Zeichenkette in M-Files suchen	<code>&gt;&gt; lookfor inverse</code>
<code>diary</code> Datei	Sitzung in Datei protokollieren	<code>&gt;&gt; diary protokoll.txt</code> <code>&gt;&gt; diary off</code> <code>&gt;&gt; diary on</code>
<code>clc</code>	Eingabefenster löschen	<code>&gt;&gt; clc</code>

Tabelle 3: Befehlsübersicht Hilfe und Desktop

## 1.2 Einfache Berechnungen

Einfache Rechenoperationen lassen sich in **Matlab** intuitive durchführen. Hier ein Beispiel:

```
Beispiel 1.1.:
>> 3 + 8           % Addition
ans =
    11
>> 12 / 4;        % Division
>> ans - 2        % das letzte Ergebnis mit -2 subtrahieren
>>
ans =
     1
>> x = 12 / 3;    % Ergebnis in Variable x speichern
>> y = x^2 * (12 - ans) % Berechnung mit Variablen
ans =
    176
```

In **Matlab** lassen sich Klammern zum Ändern der Reihenfolge von Operationen nutzen. Wird an eine Anweisung ein Semikolon angehängt, so unterdrückt **Matlab** die Ausgabe des Ergebnisses. Kommentare werden mit einem Prozentzeichen eingeleitet, der Rest der Zeile wird anschließend ignoriert.

### 1.2.1 Logische Operationen

Logische Vergleiche bzw. Verknüpfungen basieren auf der Aussage eines Wahrheitswerts von 0 oder 1. Trifft eine Aussage zu, liefert **Matlab** eine 1 zurück, ist sie falsch, bekommt man eine 0 geliefert. Im Gegensatz zu anderen Programmiersprachen kennt **Matlab** keinen extra Datentyp (Boolean)

für logische Operationen. Eine Übersicht der vorhandenen Vergleichsoperatoren und logischen Operationen ist in Tabelle 4 bzw. 5 zu finden.

Operator	Beschreibung
==	ist gleich
~=	ist ungleich
<	ist kleiner als
>	ist größer als
<=	ist kleiner-gleich
>=	ist größer gleich

Tabelle 4: Vergleichsoperatoren

Operator	Beschreibung
&&	logisches UND
	logisches ODER
~	logisches NICHT
&	logisches UND (elementweise)
	logisches ODER (elementweise)
xor	logisches Exklusiv-ODER

Tabelle 5: Logische Operatoren

*Beispiel 1.2.:*

```
>> 1 < 2           % ist 1 kleiner als 2
ans =
     1           % ja, Ergebnis ist eine 1 -> Wahr
>> 2 == 0          % ist 2 gleich 0
ans =
     0           % nein, Ergebnis ist eine 0 -> Falsch
```

Die Verknüpfungsoperatoren (siehe Tabelle 5) verbinden den Wahrheitswert zweier Ausdrücke und liefern wiederum einen neuen, der Verknüpfungsoperation entsprechenden, Wahrheitswert.

*Beispiel 1.3.:*

```
>> 1 || 0          % wahr ODER unwahr
ans =
     1           % -> wahr
>> 1 && 0           % wahr UND unwahr
ans =
     0           % -> unwahr
```

Eine Anwendung logischer Ausdrücke auf Matrizen bzw. Vektoren ist ebenfalls möglich. Haben die zu vergleichenden Ausdrücke, die gleiche Dimension, findet eine elementweise Operation statt. Ist ein Ausdruck ein Skalar, wird er implizit mit Hilfe der skalaren Expansion, auf die Dimension des anderen Ausdrucks erweitert.

**Beispiel 1.4.:**

```
>> 7 > [ 2 10 1 7 ]           % Vergleich von Skalar mit Vektor
                                % skalare Expansion
ans =                            % elementweiser Vergleich
     1     0     1     0
>> [ 1 1 1 ] & [ 1 0 1 ]      % elementweise UND-Verknüpfung
ans =
     1     0     1
```

**1.2.2 Komplexe Zahlen**

Die Verwendung von Komplexen Zahlen soll ebenfalls an einem Beispiel demonstriert werden. In **Matlab** wird die imaginäre Einheit sowohl durch *i* als auch *j* repräsentiert.

**Beispiel 1.5.:**

```
>> z = 3 + j*4                 % Komplexe Zahl erzeugen
z = 3 + 4i
>> re = real(z)                % Realteil
re = 3
>> im = imag(z)                % Imaginärteil
im = 4
>> a = abs(z)                  % Betrag
a = 5
>> phi = angle(z)              % Phase
phi = 0.92730
>> z2 = a*exp(j*phi)           % Komplexe Zahl erzeugen
z2 = 3.0000 + 4.0000i
```

Befehl	Beschreibung	Beispiel
<code>x = real(z)</code>	Realteil von z bestimmen	<code>&gt;&gt;real(2+3j)</code> <code>ans = 2</code>
<code>y = imag(z)</code>	Imaginärteil von z bestimmen	<code>&gt;&gt; imag(2+3j)</code> <code>ans = 3</code>
<code>a = abs(z)</code>	Betrag von z bestimmen	<code>&gt;&gt; abs(-5)</code> <code>ans = 5</code>
<code>phi = angle(z)</code>	Phase von z bestimmen	<code>&gt;&gt; angle(1-j)</code> <code>ans = -0.78540</code>

*Tabelle 6: Befehle für komplexe Zahlen*

**1.2.3 Weitere mathematische Funktionen**

Ein Überblick oft verwendeter mathematischer Funktionen (Winkelfunktionen, Logarithmus, Runden) ist in Tabelle 7 enthalten.

Befehl	Beschreibung	Beispiel
exp	Exponentialfunktion	>> exp(2) ans = 7.3891
log	natürlicher Logarithmus	>> log(10) ans = 2.3026
sqrt	Quadratwurzel	>> sqrt(25) ans = 5
fix	Ganzzahlanteil	>> fix(7.389) ans = 7
floor	nächstkleinere Ganzzahl	>> floor(7.389) ans = 7
ceil	nächstgrößere Ganzzahl	>> ceil(7.389) ans = 8
round	mathematisches Runden	>> round(7.5) ans = 8
mod	Modulodivision	>> mod(7,3) ans = 1
rem	Rest einer Ganzzahldivision	>> rem(12,9) ans = 3
sign	Signumfunktion	>> sign(-7) ans = -1

Tabelle 7: Weitere mathematische Funktionen

### 1.3 Vektoren und Matrizen

Da **Matlab** intern vollständig auf Matrizenoperationen beruht, lassen sich alle im letzten Abschnitt genannten Funktionen natürlich auch auf Matrizen anwenden (bisher wurden nur Skalare betrachtet, die quasi ein Sonderfall, eine Matrix mit der Dimension 1x1, sind). Der Rückgabewert ist dann wiederum eine Matrix mit identischen Dimensionen, in der die entsprechende Funktion auf jedes Element angewendet wurde.

Die prinzipielle Erzeugung und Verwendung von Matrizen bzw. Vektoren soll in einem kleinen Beispiel demonstriert werden:

*Beispiel 1.6.:*

```
>> a = [ 1 2 3 4 5 ]; % erzeugt einen Zeilenvektor (1x4)
>> b = [ 1 2 3 4 5; 6 7 8 9 10 ]; % erzeugt eine Matrix (2x4)
>> b_1 = b(1, :) % nur die erste Zeile aus Matrix b auslesen
b_1 =
    1    2    3    4    5
>> c = [ 1:4 ] % Zeilenvektor mit Schrittweite 1
c =
    1    2    3    4
>> d = [ 1:0.5:2 ] % Zeilenvektor mit Schrittweite 0,5
d =
    1.0000    1.5000    2.0000
>> d(1,2) % Zugriff auf ein einzelnes Element
ans = 1.5000
```

Zum Erzeugen von Matrizen bzw. Vektoren werden die eckigen Klammern verwendet (sind allerdings nicht unbedingt notwendig). Um einzelne Elemente voneinander abzutrennen kann ein Komma oder ein Leerzeichen verwendet werden. Der Doppelpunkt füllt den Vektor automatisch mit der angegebenen Schrittweite (Standard ist 1) auf. Der Zugriff auf ein einzelnes Element geschieht durch die Angabe des Index<sup>c</sup> in runden Klammern. Möchte man auf eine ganze Zeile bzw. Spalte einer Matrix zugreifen, ersetzt man den variablen Index einfach durch einen : (Doppelpunkt). Die automatische Erzeugung von Werten mit Hilfe des :-Operators ist sehr hilfreich, manchmal aber nicht zweckmäßig. Benötigt man z.B. 100 äquidistante Werte im Bereich [5, 10], müsste man erst die sich daraus ergebenden Schrittweite berechnen (also 0.05) und könnten anschließend den :-Operator benutzen. Die Funktion `linspace` wäre für diesen Anwendungsfall geeigneter:

*Beispiel 1.7.:*

```
>> x = linspace(5,10,100);
>> x(1:5)
ans =
    5.0000    5.0505    5.1010    5.1515    5.2020
>>x1= logspace(1, 2,5)
x1 =
    10.000    17.783    31.623    56.234    100.000
```

`linspace` erzeugt einen Zeilenvektor mit 100 Elementen. Im Beispiel werden die ersten 5 Element ausgegeben. Zusätzlich wird der Befehl `logspace` vorgestellt, der analog zu `linspace` arbeitet, allerdings die Werte logarithmisch verteilt und die Grenzen als Angabe von Zehnerpotenzen erwartet.

Das Löschen von Zeilen bzw. Spalten einer Matrix funktioniert über die Zuweisung eines leeren Klammerpaares an der entsprechenden Stelle. Hier ein Beispiel:

*Beispiel 1.8.:*

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

>> A(1,:) = []           % erste Zeile löschen
A =
     4     5     6
     7     8     9

>> A(:,2) = []          % nun von der verbleibenden Matrix A
A =                      % noch die zwei Spalte löschen
     4     6
     7     9
```

Das Rechnen mit Matrizen/Vektoren funktioniert analog zu Abschnitt 1.2, vorausgesetzt, die Matrizen haben die selben Dimensionen. Allerdings ist zu beachten, dass der \*-Operator jetzt eine Matrixmultiplikation durchführt (Spaltenzahl der ersten Matrix gleich Zeilenzahl der zweiten Matrix)!

Möchte man Operationen auf Matrizen elementweise durchführen (auch hier müssen die Matrizen gleiche Dimensionen haben), so ist im Falle der Multiplikation/Division ein `.` (Punkt) voranzustellen.

*Beispiel 1.9.:*

```
>> a = [ 1 3 5 ];
>> b = [ 2 6 4 ];
>> c = a .* b           % elementweise Multiplikation
c =
     2    18    20
>> c ./ b              % elementweise Division
ans =
     1     3     5
```

Oft sieht man in **Matlab** die Verknüpfung eines Skalars und eines Vektors respektive einer Matrix. Normalerweise würde dies aufgrund der verschiedenen Dimensionen nicht funktionieren. In diesem Fall führt **Matlab** intern allerdings die sogenannte **skalare Expansion** durch. Dabei wird der Skalar (eigentlich ja auch eine Matrix der Dimension 1x1) auf eine Matrix der selben Dimension wie der zweite Operand erweitert. An jeder Position der Matrix steht nun der Wert des Skalars. Anschließend wird die eigentliche Operation durchgeführt.

*Beispiel 1.10.:*

```
>> 1 - [1 2; 3 4]
ans =
     0    -1
    -2    -3
```

Das spezielle Schlüsselwort `end` entspricht dem Index des letzten Elements innerhalb einer Matrix. Verwendet wird es z.B. für relative Adressierung oder dazu, die Elemente einer Matrix rückwärts auszulesen.

*Beispiel 1.11.:*

```
>> a = [1 2 3 4];
>> x = a(end-1)           % vorletztes Element auslesen
x = 3
> a(end:-1:1)           % Vektor rückwärts auslesen
ans =
     4     3     2     1
```

Manchmal ist es notwendig Matrizen oder Vektoren vor einer Operation umzuformen. So lassen sich zwei Matrizen zu einer neuen erweitern, indem man sie beispielsweise stapelt oder aneinanderreihet. Wichtig ist dabei, dass sie jeweils die gleiche Anzahl von Spalten bzw. Zeilen besitzen. Intern ist das Stapeln bzw. Aneinanderreihen über die beiden **Matlab**-Befehle `horzcat` und `vertcat`, die jeweils Spezialfälle von `cat` sind, realisiert.

*Beispiel 1.12.:*

```
>> A = [ 1 2; 3 4 ];           % Matrix vom Typ 2x2
>> B = [ 5 5 ];               % Matrix vom Typ 1x2
>> C = [ A; B ]               % Matrizen stapeln
C =
     1     2
     3     4
     5     5
>> D = [ A B' ]               % Matrizen aneinanderreihen
D =
     1     2     5
     3     4     5
```

Um Eigenschaften einer Matrix abzufragen gibt es die Befehle `size` und `length`. `size` gibt die Größe aller Dimensionen einer Matrix (Zeilen, Spalten für zweidimensionalen Fall) zurück, `length` nur die jeweils größte (also das Maximum von `size`). Die Funktion `find` kann zum Auffinden von Elementen, die einen Werte ungleich 0 haben, genutzt werden. Rückgabewert ist ein Vektor, der die Indizes der gefundenen Elemente angibt.

*Beispiel 1.13.:*

```
>> i = eye(3)           % Einheitsmatrix vom Typ 3x3 erzeugen
i =
     1     0     0
     0     1     0
     0     0     1

>> find(i)            % welche Elemente sind ungleich 0?
ans =
     1
     5
     9

>> m = magic(3)       % Magisches Quadrat erzeugen
m =
     8     1     6
     3     5     7
     4     9     2

>> m > 4              % logischen Vergleich der Matrix m
ans =                 % mit dem Wert 4
     1     0     1
     0     1     1
     0     1     0

>> find(m > 4)       % welche Elemente von m sind größer als 4?
ans =
     1
     5
     6
     7
     8
```

Dieses etwas umfangreichere Beispiel soll noch einmal die Verwendung der eben genannten Funktionen verdeutlichen. Anfangs wird mit `eye` eine Einheitsmatrix der Dimension  $3 \times 3$  angelegt. Die Einheitsmatrix enthält auf der Hauptdiagonale Einsen ansonsten Nullen. `find` liefert nun die Position der Elemente, die nicht 0 sind. Interessanterweise wird der Index dabei nicht in Form eines Wertepaares `ij` sondern als fortlaufende Zahl angegeben. **Matlab** verwendet diese Form der Darstellung, **lineare Indizierung** genannt, intern für seine Matrizendarstellung. Dabei wird der Index von links oben, die aktuelle Spalte abwärts und dann wieder von oben in der nächsten Spalte beginnend gezählt. Ein Zugriff auf Matrixelemente mittels linearer Indizierung ist natürlich ebenfalls möglich. Nur auf Werte ungleich 0 zu prüfen, ist noch nicht sonderlich hilfreich, deswegen wurde das Beispiel noch etwas erweitert. Die Funktion `magic` erzeugt eine Matrix mit unterschiedlichen Werten.

Nun wird ein logischer Vergleich zwischen der Matrix  $m$  und der Zahl 4 (diese wird durch skalare Expansion intern auf eine Matrix der Dimension  $3 \times 3$  erweitert) durchgeführt. Ergebnis ist eine Matrix mit Wahrheitswerten (logische Matrix), die an die `find`-Funktion übergeben werden kann. Kombiniert man diese beiden Schritte zu `find(m > 4)`, erhält man einen kompakten Ausdruck, der die Indizes aller Elemente von  $m$ , die größer als 4 sind, liefert. Nachfolgend ein weiteres Beispiel, das alle Elemente innerhalb einer Matrix auf den Wert 5 begrenzt.

*Beispiel 1.14.:*

```
>> A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

>> A(A > 5) = 5
A =
     5     1     5
     3     5     5
     4     5     2
```

Zum Erzeugen einer Diagonalmatrix, ist der Befehl `diag` zu verwenden. Er erwartet als Argument einen Vektor, der die Elemente der Hauptdiagonalen darstellt. Auch das Erzeugen von Matrizen mit zufälligen Werten ist kein Problem, `rand` und `rand` (Normalverteilung) stehen dafür zur Verfügung. Die Nullmatrix lässt sich mit dem Befehl `zeros` generieren.

Befehl	Beschreibung	Beispiel
<code>x = [ ... ... ]</code>	Zeilenvektor erzeugen	<pre>&gt;&gt; x = [ 1 2 3 ] x =     1    2    3</pre>
<code>x = [ ... ; ... ]</code>	Spaltenvektor erzeugen	<pre>&gt;&gt; x = [ 1; 2] x =     1     2</pre>
<code>A = [ ... ; ... ; ... ]</code>	Matrix erzeugen	<pre>&gt;&gt; A = [ 1 2; 3 4 ] A =     1    2     3    4</pre>
<code>a = x(...)</code>	Zugriff auf Vektorelement	<pre>&gt;&gt; a = x(2) a = 3</pre>
<code>y = linspace(a,b,n)</code>	Spaltenvektor im Bereich von a bis b mit n Elementen	<pre>&gt;&gt; y = linspace(1, 2, 3) y =     1.0000    1.5000    2.0000</pre>
<code>y = logspace(a,b,n)</code>	Spaltenvektor im Bereich von $10^a$ bis $10^b$ mit n logarithmisch geteilten Elementen	<pre>&gt;&gt; y = logspace(1, 2, 3) y =     10.000    31.623    100.000</pre>
<code>B = A'</code>	Matrix transponieren	<pre>&gt;&gt; B = A' B =     1    3     2    4</pre>
<code>B = inv(A)</code>	Inverse zur Matrix A berechnen	<pre>&gt;&gt; B = inv(A) B =    -2.00000    1.00000     1.50000   -0.50000</pre>
<code>B = pinv(A)</code>	Pseudoinverse berechnen	<pre>&gt;&gt; C = [ 1 2 ; 3 4 ; 5 6 ]; &gt;&gt; D = pinv(C) D =    -1.33333   -0.33333    0.66667     1.08333    0.33333   -0.41667</pre>
<code>A = eye(n)</code>	Einheitsmatrix vom Typ n x n	<pre>&gt;&gt; A = eye(3) A =     1    0    0     0    1    0     0    0    1</pre>
<code>i = find(A)</code>	Liefert Spaltenvektor mit Indizes der Element die ungleich 0 sind	<pre>&gt;&gt; A = eye(3); &gt;&gt; find(A) ans =     1     5     9</pre>
<code>[m,n] = size(X)</code>	Liefert Dimension einer Matrix	<pre>&gt;&gt; [ m,n ] =size(A) m = 3 n = 3</pre>
<code>n =length(X)</code>	Länge eines Vektors	<pre>&gt;&gt; length(A) ans = 3</pre>
<code>end</code>	Index des letzten Elements	<pre>&gt;&gt; a = [1 2 3 4]; &gt;&gt; a(end) ans = 4</pre>

Tabelle 8: Befehle für Vektoren und Matrizen

## 1.4 Arbeiten mit Gleichungssystemen

Übliche lineare Gleichungssysteme sind nach der Form

$$A \cdot x = b$$

aufgebaut. Wobei  $A$  hier die quadratische Matrix der Koeffizienten und  $b$  der Ergebnisvektor ist. Um den Lösungsvektor  $x$  zu erhalten muss  $b$  mit  $A^{-1}$  multipliziert werden:

$$x = A^{-1} \cdot b$$

$A^{-1}$  entspricht dabei der Inversen zu  $A$ . Die Inverse zu einer (quadratischen) Matrix lässt in **Matlab** sehr einfach mit dem Befehle `inv` ermitteln. Zu beachten ist, dass `inv` nur eine gültige Lösung liefert, wenn die Matrix  $A$  regulär ist. Ob eine Matrix regulär ist, lässt sich z.B. über die Determinante bestimmen. Ist diese ungleich 0, ist die Matrix regulär. **Matlab** bietet zum Bestimmen der Determinante die Funktion `det` an.

Eine Besonderheit von **Matlab** ist der `\`-Operator. Es handelt sich dabei um eine direkte Matrixdivision. Man könnte zum Lösen des o.g. Linearen Gleichungssystems als auch folgendes schreiben:

*Beispiel 1.15.:*

```
>> x = A \ b
```

Des weiteren wird im Falle eines überbestimmten Gleichungssystems automatisch eine Lösung im Sinne der kleinsten Quadrate durchgeführt.

Ein weiteres Kriterium bei der Berechnung von linearen Gleichungssystemen ist der sogenannte Rang. Er gibt für eine symmetrische Matrix an, wie viele voneinander unabhängige Zeilen- bzw. Spaltenvektoren diese enthält. Für ein Matrix vom Typ (3x3) würde ein Rang von 3 z.B. auf eine eindeutige Lösung hinweisen. Die Bestimmung des Ranges geschieht in **Matlab** mit dem Befehl `rank`.

Benötigt man die Eigenwerte bzw. Eigenvektoren einer Matrix, ist der `eig`-Befehl zu verwenden. Er gibt entweder einen Vektor mit den Eigenwerten oder wenn gewünscht, eine Matrix der Eigenvektoren und eine Diagonalmatrix mit den Eigenwerten zurück.

Hier nun ein etwas umfangreicheres Beispiel, dass die Lösungsmöglichkeiten eines linearen Gleichungssystems in **Matlab** zeigt:

*Beispiel 1.16.:*

```
>> A = [ 2 1 ; 4 3]; % quadratische Matrix A vom Typ (2x2) erzeugen
>> b = [8 ; 10]; % Ergebnisvektor erzeugen
>> rank(A) % Rang von A überprüfen -> nicht unterbestimmt!
ans = 2
>> det(A) % Determinanten ungleich 0 -> reguläre Matrix
ans = 2
>> x = A \ b % Lösung des Gleichungssystems mit Divisionoperator
x =
    7
   -6
>> x2 = inv(A) * b % Lösung der Gleichungssystems durch Multiplikation
x2 = % mit der Inversen von A
    7
   -6
```

Wie zu erwarten liefert sowohl die Matrixdivision als auch die Multiplikation mit der Inversen von  $A$  das Ergebnis des linearen Gleichungssystems.

Befehl	Beschreibung	Beispiel
$x = A \setminus b$	Direkte Matrixdivision (Lösen eines linearen Gleichungssystems)	<pre>&gt;&gt; A = [ 2 1; 4 3 ]; &gt;&gt; b = [ 8 ; 10 ]; &gt;&gt; x = A \ b  x =      7    -6</pre>
$k = \text{rank}(A)$	Rang der Matrix A	<pre>&gt;&gt; rank(A)  ans =      2</pre>
$d = \text{det}(X)$	Determinanten der Matrix X	<pre>&gt;&gt; det(A)  ans =      2</pre>

Tabelle 9: Befehle für den Umgang mit linearen Gleichungssystemen

## 1.5 Polynome

Da **Matlab** ohne spezielle Toolboxes keine symbolischen Rechnungen wie z.B. **Maple** unterstützt, müssen Polynome in einer etwas anderen Form angegeben werden. **Matlab** wählt hier die Darstellung über die Koeffizienten. Die Polynome lassen sich dabei bequem als Vektoren darstellen. Hierzu werden die Koeffizient des Polynoms in absteigender Reihenfolge in einem Vektor angeordnet. Ist ein Koeffizient bestimmter Ordnung nicht vorhanden, wird er mit 0 angegeben. Die Funktion

$$y = 3x^3 - 2x + 0.5$$

würde in **Matlab** dem Koeffizientenvektor

$$[3 \ 0 \ -2 \ 0.5]$$

entsprechen. Hier nun ein kleines Beispiel. Mit Hilfe der Funktion `roots` sollen die Nullstellen des Polynoms  $y = x^2 + 0.5x$  bestimmt werden.

*Beispiel 1.17.:*

```
>> pA = [ 1 0.5 0 ];           % Polynome erzeugen
>> x = roots(pA)              % Nullstellen berechnen
x =

   -0.50000
    0.00000

>> poly(x)                    % Polynomkoeffizienten aus
ans =                          % Nullstellen bestimmen

    1.00000    0.50000    0.00000

>> y = polyval(pA, 2)        % Wert des Polynoms an der Stelle x = 2
y = 5
```

Im Beispiel 1.17 wurde zusätzlich der umgekehrte Weg angewendet, d.h. aus den mit `roots` berechneten Nullstellen wurde mittels `poly`<sup>1</sup> wieder das Ursprungspolynom zurückgewonnen ( $p_A(x) = \prod_{i=1}^n (x - x_i)$ ). Mit dem Befehl `polyval` wurde der Wert des Polynoms  $p_A$  an der Stelle  $x=2$  berechnet. Für die graphische Darstellung eines Polynoms benötigt man den Funktionswerte natürlich an mehr als einer Stelle. Für diesen Fall übergibt man `polyval` als zweites Argument einfach einen Vektor. Hier die Fortsetzung des vorherigen Beispiels:

*Beispiel 1.18.:*

```
>> y2 = polyval(pA, [ 1:0.1:5 ]); % Funktionswerte des Polynoms berechnen
>> size(y2)
ans =
     1     41
```

Für den Wertebereich von  $[1,5]$  mit der Schrittweite von 0.1 werden die Funktionswerte des Polynoms  $p_A$  berechnet und in einem Zeilenvektor mit 41 Elementen zurückgegeben.

Manchmal ist es notwendig, zwei Polynome miteinander zu multiplizieren bzw. auch den umgekehrten Weg, der Polynomdivision zu gehen. **Matlab** bietet hierfür die Befehle `conv` bzw. `deconv`.

*Beispiel 1.19.:*

```
>> p1 = [ 1 2 3 ];
>> p2 = [ 2 1 -5 ];
>> p3 = conv(p1,p2) % Multiplikation der beiden Polynome
p3 =
     2     5     3    -7   -15

>> p4 = deconv(p3, p2) % Division
p4 =
     1     2     3

>> polyder(p3) % Ableitung des Polynoms bilden
ans =
     8    15     6    -7

>> polyder(p1,p2) % erst beide Polynome multiplizieren
ans = % und anschließend ableiten
     8    15     6    -7
```

In obigem Beispiel wurden die Polynome  $p_1 = x^2 + 2x + 3$  und  $p_2 = 2x^2 + x - 5$  miteinander multipliziert. Ergebnis der Polynommultiplikation war  $p_3 = 2x^4 + 5x^3 + 3x^2 - 7x - 15$ . Anschließend wurde  $p_3$  mittels `deconv` durch  $p_2$  dividiert. In diesem Zusammenhang wird auch gleich noch der Befehl `polyder` zur Ableitung von Polynomen vorgestellt. Wird er mit nur einem Polynomvektor als Argument aufgerufen, wird ausschließlich die Ableitung gebildet. Ruft man ihn hingegen mit zwei Vektoren auf, findet vor der Ableitung implizit eine Polynommultiplikation mittels `conv` statt.

Die Partialbruchzerlegung, d.h. das Aufspalten einer gebrochenrationalen Funktion in eine Summe von einfachen Brüchen, wird in **Matlab** mit der Funktion `residue` realisiert.

---

1 Übergibt man `poly` eine symmetrische Matrix wird stattdessen das charakteristische Polynom berechnet

$$\frac{b(x)}{a(x)} = k(x) + \sum_i \frac{r_i}{x - p_i}$$

Die Funktion erwartet als Parameter zwei Vektoren, die jeweils das Zähler- bzw. Nennerpolynom repräsentieren. Die Rückgabe enthält wiederum min. zwei Vektoren, die Koeffizienten der Partialbrüche und die dazugehörigen Polstellen. Ist der Grad des Zählers größer oder gleich dem Nennergrad entsteht zusätzlich ein weiterer Vektor, das Ergebnis der Polynomdivision  $k(x)$

*Beispiel 1.20.:*

```
>> b = [ 4 12];           % Zählerpolynom
>> a = [1 3 2 0];       % Nennerpolynom
>> [r, p, k] = residue(b, a) % Partialbruchzerlegung
r =
     2
    -8
     6
     % Residuen
p =
    -2
    -1
     0
     % Pole
k =
    []
     % Restterm der Polynomdivision
```

Im Beispiel wurde der Quotient  $\frac{4x-12}{x^3+3x^2+2x}$  in eine Summe partieller Brüche zerlegt:  $\frac{6}{x} - \frac{8}{x+1} + \frac{2}{x+2}$ . Da der Grad des Zählers kleiner dem des Nenners ist, entsteht kein Restterm  $k(x)$ .

Befehl	Beschreibung	Beispiel
<code>pA = [ ... .. ]</code>	Koeffizienten des Polynoms pA als Zeilenvektor angeben	<pre>&gt;&gt; pA = [ 2 3 1 ] pA =      2     3     1</pre>
<code>x = roots(pA)</code>	Nullstellen des Polynoms pA berechnen	<pre>&gt;&gt; x = roots(pA) x =  -1.00000  -0.50000</pre>
<code>pA = poly(x)</code>	Polynomkoeffizienten aus Nullstellenvektor bestimmen	<pre>&gt;&gt; pA = poly([ -1 -0.5 ]) pA =   1.00000  1.50000  0.50000</pre>
<code>y = polyval(pA, x)</code>	Funktionswert des Polynoms pA an der Stelle x berechnen	<pre>&gt;&gt; y = polyval(pA, 3) y = 14</pre>
<code>p3 = conv(p1, p2)</code>	Multiplikation der beiden Polynome p1 und p2	<pre>&gt;&gt; p3 = conv([ 3 1 0 ], [ 2 1 ]) p3 =      6     5     1     0</pre>
<code>p3 = deconv(p1, p2)</code>	Polynomdivision von p1 durch p2	<pre>&gt;&gt; p4 = deconv([ 6 5 1 0 ], [ 2 1 ]) p4 =      3     1     0</pre>
<code>p1 = polyder(p)</code>	Ableitung von Polynom p bilden	<pre>&gt;&gt; p1 = polyder([ 2 3 1 ]) p1 =      4     3</pre>
<code>[r,p,k] = residue(b,a)</code>	Partialbruchzerlegung des Polynomquotienten	<pre>&gt;&gt; b = [ 5 3 -2 7 ]; &gt;&gt; a = [-4 0 8 3 ]; &gt;&gt; [r, p, k] = residue(b,a) r =  -1.4167  -0.6653   1.3320 p =   1.5737  -1.1644  -0.4093 k =  -1.2500</pre>

Tabelle 10: Übersicht Umgang mit Polynomen

## 1.6 Graphische Darstellung

**Matlab** verfügt über ein sehr leistungsfähiges System zur Ausgabe von graphischen Plots. Es werden sowohl 2D- als auch 3D-Darstellung unterstützt. Diese wiederum lassen sich in verschiedenen Koordinatensystemen mit Achsenbeschriftung und Legende sowie als Höhenlinien oder als Netz darstellen.

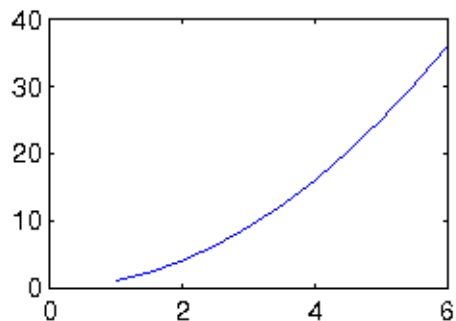
### 1.6.1 Grundlagen

Wie schon an verschiedenen Stellen erwähnt, basiert **Matlab** ausschließlich auf den Datentypen Vektoren bzw. Matrizen. Eine symbolische Darstellung einer Funktion ist somit nicht möglich. Stattdessen müssen die Wertepaare (im einfachsten Falle x und y) für einen Plot in Form von Stützstellen (Vektoren) vorgegeben werden. Folgendes Beispiel soll dies anschaulich verdeutlichen.

*Beispiel 1.21.:*

```
>> x = [1:0.5:6];           % Vektor x mit den Werten 1 bis 6,  
                             % Schrittweite 0.5 erzeugen  
>> y = x.^2;               % y-Werte berechnen  
>> plot(x,y);              % y über x ausgeben
```

Zuerst wird der x-Vektor erzeugt. Er enthält die Werte  $x=1;1.5;2;\dots;6$ , insgesamt also 11 Element. Die darzustellende analytische Funktion lautet  $y=x^2$ . Der zu jedem x-Werte zugehörige y-Wert wird durch eine elementweise (siehe Abschnitt 1.3) Quadrierung erzeugt und im Vektor  $y$  hinterlegt. Dieser hat nun ebenfalls die Länge 11. Anschließend kann die graphische Ausgabe mit dem Befehl `plot` erfolgen. Es werden jeweils die Elemente mit gleichem Index aus den Vektoren  $x$  und  $y$  dargestellt und mit einer durchgezogenen Linie verbunden. Das Ergebnis ist in Abbildung 2 zu sehen.



*Abbildung 2: Darstellung einer quadratischen Funktion in Matlab*

Übergibt man dem Befehl `plot` nur einen Vektor als Argument, so wird dieser als Vektor der y-Werte interpretiert. Die Indizes dieses Vektors stellen dann zugleich die x-Werte der Abszisse da.

Eine Besonderheit von **Matlab** ist, dass jeder neue Plotbefehl die vorherigen Ausgaben überschreibt. Oft kommt es jedoch vor, dass man mehrere Graphen in ein Koordinatensystem einzeichnen möchte. Hierfür gibt es prinzipiell zwei Vorgehensweisen. Der als Schalter wirkende Befehl `hold` weist **Matlab** an, alle nachfolgenden Plots in ein eventuell schon existierendes Koordinatensystem einzuzeichnen. Soll der nächste Plot wieder in ein eigenständiges Koordinatensystem eingezeichnet werden, so ist die Anweisung `hold off` notwendig. Alternativ kann man dem Plotbefehl auch eine Matrix übergeben. Im Folgenden Beispiel sollen insgesamt drei Winkelfunktionen in ein Koordinatensystem eingezeichnet werden:

*Beispiel 1.22.:*

```
>> x = [0:0.1:20];           % x-Vektor erzeugen, geringe Schrittweite  
>> plot(x, 0.5*sin(x))      % ersten Graph ausgeben, skaliertes Sinus  
>> hold on                  % Koordinatensystem ,,festhalten''  
Current plot held  
>> plot(x, [ cos(x) + 2;   sin(x) - 2 ] ); % zwei weitere Graphen
```

Das Ergebnis ist in Abbildung 3 zu sehen.

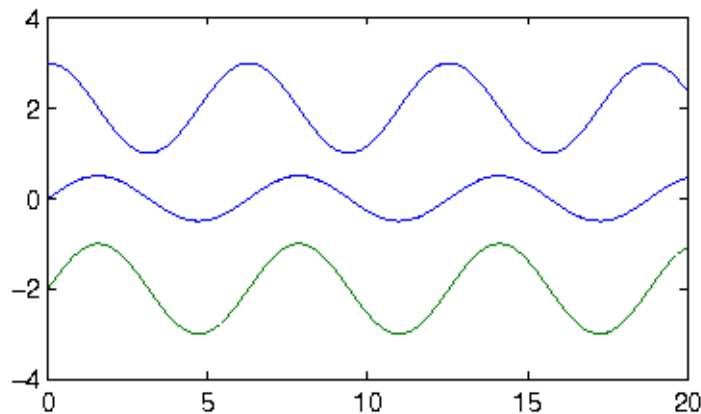


Abbildung 3: Mehrere Plots in einem Koordinatensystem

Zuerst wird wieder ein  $x$ -Vektor mit passender Einteilung erzeugt. Anschließend wird die Sinusfunktion auf diesen Vektor angewendet (für jedes Element wird der Sinuswert berechnet) und jedes Element auf die Hälfte skaliert, dann folgt der Plot. Der `hold` Befehl hält nun das aktuelle Koordinatensystem für weitere Ausgaben fest. Der nächste Plotbefehl, der als zweites Argument nun eine Matrix übergeben bekommt, gibt zusätzlich noch eine Kosinus- und Sinusfunktion (jeweils um 2 nach unten bzw. oben verschoben) aus. Um bereits existierende Graphen aus einem Koordinatensystem zu löschen, steht der Befehl `clf` zur Verfügung.

## 1.6.2 Eigenschaften von Koordinatensystemen und Graphen

Die einfache Darstellung von Wertepaaren ist allein oftmals nicht ausreichend, meistens ist es notwendig, eine definierte Achseneinteilung vorzugeben, die Graphen entsprechend zu beschriften bzw. zu kennzeichnen oder eine Legende einzutragen.

Hier nun wieder ein kurzes Beispiel. Es werden zwei Graphen  $y_1=x^2$  und  $y_2=3\sin(x)$  in ein Koordinatensystem eingezeichnet. Es soll sowohl eine Legende erstellt als auch eine Beschriftung der  $x$ - und  $y$ -Achse durchgeführt werden. Das Ergebnis ist in Abbildung 4 zu sehen.

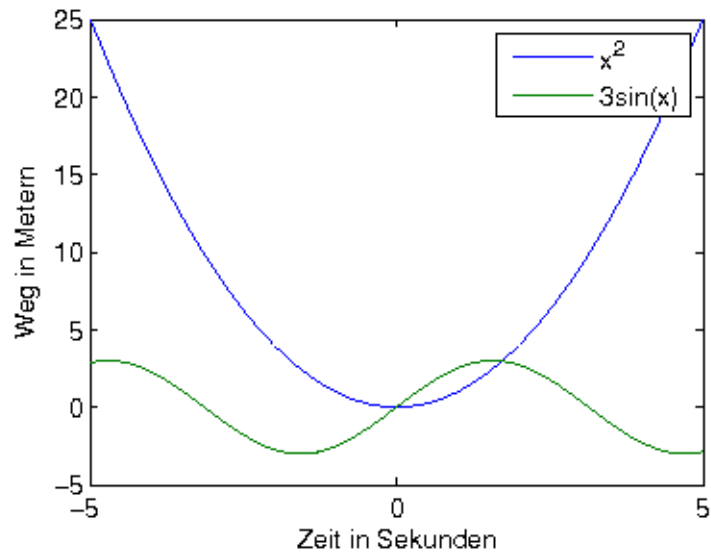


Abbildung 4: Plot mit Legende

**Beispiel 1.23:**

```
>> x = [-5:0.1:5];           % x-Vektor erzeugen
>> y1 = x.^2;                % y-Werte der quadratischen Funktion
>> y2 = 3*sin(x);           % y-Werte der Sinusfunktion
>> plot(x, [ y1; y2 ]);     % beide Funktionen ausgeben
>> legend('x^2', '3sin(x)'); % Legende anzeigen
>> xlabel('Zeit in Sekunden'); % x-Achse bezeichnen
>> ylabel('Weg in Metern');  % y-Achse bezeichnen
```

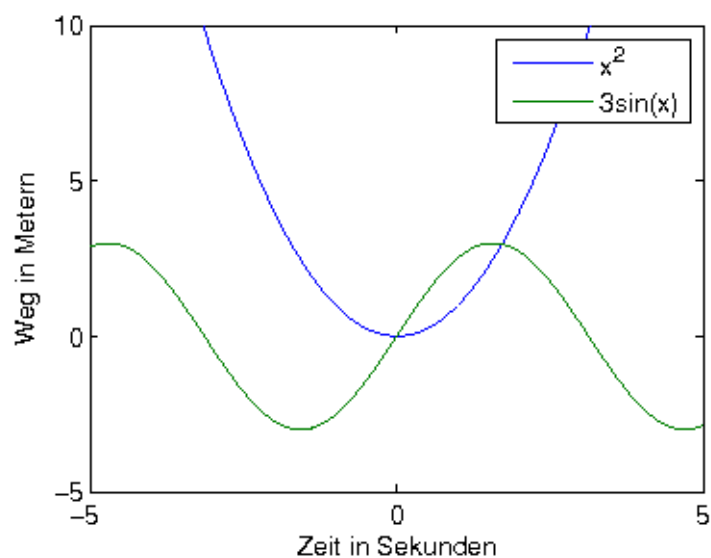


Abbildung 5: Plot mit manueller Achseneinteilung

Die ersten vier Zeilen bis zur Ausgabe des Plots sollten nun bekannt sein. Anschließend wird **Matlab** durch den Befehl `legend` angewiesen, eine Legende (verschiebbare Box) zu erstellen. Die Zuordnung zu den vorher erstellten Graphen geschieht automatisch, d.h. der zuerst erzeugte Graph bekommt auch die erste Zeichenkette des `legend`-Befehls zugeordnet. Das Bezeichnen der Achsen funktioniert ähnlich, `xlabel` erwartet eine Zeichenkette und benennt damit die x-Achse, `ylabel` funktioniert analog.

Manchmal ist die von **Matlab** automatisch vorgenommene Achseneinteilung unzweckmäßig. So sollte im vorherigen Beispiel die y-Achse nur im Bereich von -5 bis 10 dargestellt werden. Fügt man folgende Anweisung an das vorherige Beispiel an, so entsteht das in Abbildung 5 dargestellte Bild.

*Beispiel 1.24:*

```
>> ylim([-5 10]);
```

`ylim` erwartet einen Vektor mit zwei Elementen, der die Unter- und Obergrenze der Achse angibt. Analog zu `ylim` existieren noch die Befehle `xlim` und `zlim`. Allgemeiner als die gerade vorgestellten Befehle zum Setzen der Grenzen pro Achse ist der Befehl `axis`. Er bestimmt die Skalierung und Beschriftung aller Achsen. Wird er ohne Argument aufgerufen, gibt er einen Vektor mit den Unter- und Obergrenzen für alle Achsen zurück. Neben dem Setzen von definierten Grenzwerten unterstützt er auch spezielle Schlüsselworte wie z.B. `AUTO`, der automatische Anpassung der Achsen in Abhängigkeit von den vorhandenen Wertepaaren.

### 1.6.3 Weitere Plotbefehle

Bisher wurde ausschließlich die Ausgabe von einfachen zweidimensionalen Graphen gezeigt. **Matlab** unterstützt allerdings noch eine Vielzahl weiterer Plotbefehle. `loglog`, `semilogx` und `semilogy` sind Spezialfälle von `plot`, sie funktionieren genauso wie der anfangs vorgestellte `plot`-Befehl, nur mit dem Unterschied, dass sie eine doppelt logarithmische (zur Basis 10) Teilung bzw. eine einfach logarithmische Teilung der x-/y-Achse vornehmen.

Für die Darstellung mehrerer Graphen neben- bzw. untereinander ist der Befehl `subplot` gedacht. Er ist jeweils vor dem eigentlich Graphikbefehl aufzurufen und gestattet die Einteilung in ein tabellarisches Layout. Er erwartet drei Argumente, die Gesamtanzahl der Zeilen und Spalten, sowie die Position (dabei wird im Gegensatz zur linearen Indizierung nicht spaltenweise sondern zeilenweise gezählt) des nachfolgenden Plots.

3D-Darstellung können mit folgenden Befehlen durchgeführt werden:

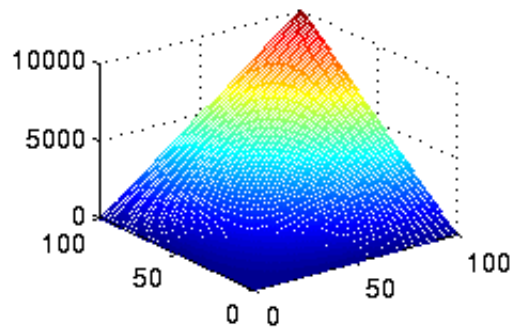
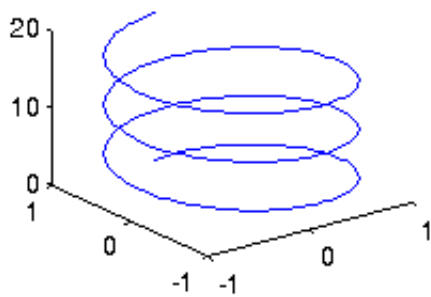
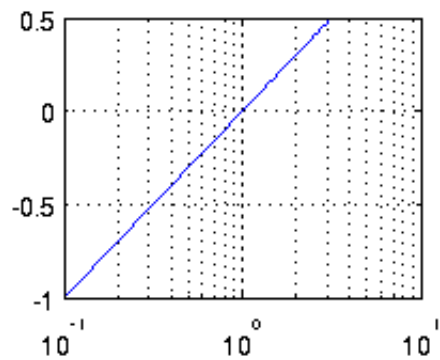
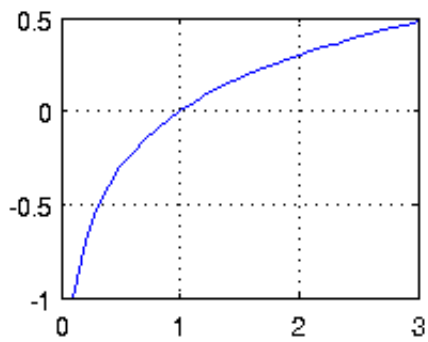
- `plot3` Erweiterung des `plot`-Befehls auf eine dritte Dimension (parametrisch)
- `mesh` 3D-Darstellung mittels Netzoberfläche (dritter Parameter ist eine Matrix)
- `surf` ähnlich zu `mesh`, Darstellung mittels Schattierung

Diese Befehle erwarten im Vergleich zu `plot` als Argument nun noch zusätzlich jeweils einen dritten Vektor (im Falle von `mesh` und `surf` eine Matrix), der die Werte für die z-Achse enthält. Befehle wie `xlabel` die bisher nur für zwei Dimensionen zur Verfügung standen, können nun auch in Form von `zlabel` auf die dritte Dimension angewendet werden.

Nachfolgend ein Beispiel und das dazugehörige Ergebnis (Abbildung 6) zu den eben vorgestellten Befehlen.

*Beispiel 1.25.:*

```
>> x1 = 0.1:0.1:3;           % lineare Teilung
>> y1 = log10(x1);          % logarithmische Teilung
>> subplot(2,2,1), plot(x1,y1); % linearer Plot
>> grid                     % Gitternetzlinien einschalten
>> subplot(2,2,2), semilogx(x1,y1); % einfach logarithmische Teilung
>> grid                     % Gitternetzlinien einschalten
>> subplot(2,2,3), plot3(sin(t),cos(t),t) % parametrischer 3D-Plot
>> x = 0:100;
>> y = x;
>> Z = x' * y;              % Höheninformation, Z-Matrix
>> subplot(2,2,4), mesh(x,y,Z) % Gitternetzlinienplot
```



*Abbildung 6: weitere Plotbefehle*

<b>Befehl</b>	<b>Beschreibung</b>	<b>Beispiel</b>
<code>plot(x, y)</code>	2D-Plot der Vektoren y über x	<pre>&gt;&gt; x = [0:0.2:10]; &gt;&gt; y = sin(x); &gt;&gt; plot(x, y)</pre>
<code>hold</code>	nachfolgende Plot-Befehle in das selbe Koordinatensystem (Schalter)	<pre>&gt;&gt; plot(x, y); &gt;&gt; hold on; &gt;&gt; plot(x, 2 * y)</pre>
<code>legend('s1', 's2', ...)</code>	Legende für Plots einfügen	<pre>&gt;&gt; plot(x, [y; 2*y]) &gt;&gt; legend('Amplitude = 1', 'Amplitude = 2');</pre>
<code>xlabel('...')</code>	Achsenbezeichnung x-Achse	<pre>&gt;&gt; xlabel('Zeit in Sekunden')</pre>
<code>ylabel('...')</code>	Achsenbezeichnung y-Achse	<pre>&gt;&gt; ylabel('Weg in Metern')</pre>
<code>xlim([xmin xmax])</code> <code>ylim([ymin ymax])</code> <code>zlim([zmin zmax])</code>	Grenzen der x-Achsen Einteilung	<pre>&gt;&gt; xlim([ 0 5 ]);</pre>
<code>axis</code>	Achseneinteilung festlegen	<pre>&gt;&gt; axis square &gt;&gt; axis auto</pre>
<code>loglog(x, y)</code>	ähnlich wie plot, allerdings doppelt logarithmisch	<pre>&gt;&gt; x = logspace(-1, 2); &gt;&gt; loglog(x, exp(x));</pre>
<code>semilogx(x, y)</code> <code>semilogy(x, y)</code>	ähnlich wie plot, allerdings x-Achse logarithmisch	<pre>&gt;&gt; x = 0:0.1:10; &gt;&gt; semilogy(x, 10.^x);</pre>
<code>plot3(x, y, z)</code>	3D-Plot	<pre>&gt;&gt; t = 0:pi/50:10*pi; &gt;&gt; plot3(sin(t), cos(t), t) &gt;&gt; axis square</pre>
<code>mesh(x, y, Z)</code>	3D-Gitternetzlinien-Plot	<pre>&gt;&gt; x = 0:100; &gt;&gt; y = x; &gt;&gt; z = x' * y; &gt;&gt; mesh(x, y, z) &gt;&gt; surf(x, y, z)</pre>
<code>surf(x, y, Z)</code>	3D-Oberflächen-Plot	
<code>grid</code>	Gitternetzlinie ein-/ausschalten	<pre>&gt;&gt; grid on &gt;&gt; grid off</pre>
<code>subplot(r, c, i)</code>	Mehrere Graphen in ein Koordinatensystem	<pre>&gt;&gt; x= 0:0.2:10; &gt;&gt; subplot(1,2,1), plot(x, sin(x)) &gt;&gt; subplot(1,2,2), plot(x, cos(x))</pre>
<code>polar(theta, rho)</code>	Plot in Polarkoordinaten	<pre>&gt;&gt; t = 0:.01:2*pi; &gt;&gt; polar(t, sin(2*t) .* cos(2*t))</pre>
<code>compass(Z)</code>	Vektoren als Pfeile vom Koordinatenursprung ausgeben	<pre>Z = eig(randn(20,20)); compass(Z)</pre>
<code>title 'string'</code>	Titel des Plots setzen	<pre>&gt;&gt; plot(sin(0:0.2:10)) &gt;&gt; title('Sinusfunktion')</pre>
<code>figure</code>	Leeres Koordinatensystem erzeugen	<pre>&gt;&gt; figure</pre>
<code>close</code>	Koordinatensystem schließen	<pre>&gt;&gt; close all</pre>
<code>text(x, y, 'string')</code>	beliebigen Text einfügen	<pre>&gt;&gt; t = 0:0.1:20; &gt;&gt; plot(t, sin(t)); &gt;&gt; text(pi/2, 1 'erstes Maximum')</pre>

*Tabelle 11: Übersicht Plotbefehle*

## 1.7 Programmierung

Bisher wurde **Matlab** ausschließlich interaktiv über den Desktop bzw. das *Command Window* gesteuert. Für sich wiederholende Aufgaben bzw. die Auslagerungen von Funktionalität bietet **Matlab** sogenannte M-Files. Dies sind einfache Textdateien, die auf die Endung *.m* enden. Diese lassen sich mit jedem beliebigen Texteditor erstellen. Alternativ steht auch ein in **Matlab** integrierter Editor (*Menu -> File -> New ->M-File*) zur Verfügung. Da ein M-File prinzipiell nichts anderes als eine Ansammlung von **Matlab**-Befehlen ist, wird auch die gleiche Syntax wie bisher verwendet. Das *%*-Zeichen dient weiterhin als Einleitung zu einem Kommentar. Eine kleine Besonderheit ist, dass die ersten zusammenhängenden Kommentarzeilen am Anfang eines M-Files gleichzeitig als Beschreibung (Hilfetext) dienen. Legt man beispielsweise ein M-File mit dem Namen *meinErstesMScript.m* und Inhalt

*Beispiel 1.26.:*

```
% Die Beschreibung zu meinem ersten Matlab-Script
% ...
```

an, kann man sich anschließend diesen Text in **Matlab** mittels

*Beispiel 1.27.:*

```
>> help meinErstesMScript
Die Beschreibung zu meinem ersten Matlab-Script
```

anzeigen lassen.

Prinzipiell kann man M-Files in zwei Gruppen unterteilen:

- Script-Files: Ansammlung von verschiedenen **Matlab**-Befehlen. Alle im M-File definierten Variablen sind anschließend auch im *Workspace* sichtbar. Genauso sind vorher im *Workspace* definierte Variablen auch innerhalb des M-Files verfügbar.
- Function-Files: Ebenfalls eine Ansammlung von **Matlab**-Befehlen allerdings jetzt mit der Besonderheit, dass man hier eigene Funktionen definieren kann. Alle innerhalb einer Funktionsdeklaration verwendeten Variablen sind nun nur noch lokal im M-File sichtbar. Eine Veränderung des globalen *Workspace* ist nicht möglich, stattdessen können Funktionen einen Wert (Rückgabewert) zurückgeben.

### 1.7.1 Kontrollstrukturen

Typische Vertreter von Kontrollstrukturen sind z.B. Schleifen oder Verzweigungen in Abhängigkeit von bestimmten Bedingungen. **Matlab** bietet hier wieder eine Reihe von Möglichkeiten, die in den beiden nachfolgenden Beispielen demonstriert werden sollen.

*Beispiel 1.28.:*

```
% Beispiel der in Matlab vorhandenen Schleifen
%
% for-Schleife
for i=0:2:6
    fprintf('%d\n',i);
end
% while-Schleife
while i > 2
    i=i - 1;
end
fprintf('i ist nun %d\n', i);
```

Hier wurden die zwei typischen Schleifen, die `for`-Schleife (auch Zählerschleife) und die `while`-Schleife (bedingte Schleife) vorgestellt. Nach der Verwendung des Schlüsselwortes `for` folgt die Initialisierung der Schleifenvariable `i`. Diese soll im Beispiel von 0 an starten und in Zwischenschritten bis zum Werte 6 hochzählen. Dabei wird in jedem Durchlauf der Inhalt der Variable `i` über die Funktion `fprintf` auf dem Bildschirm ausgegeben.

**Hinweis:** Die Initialisierung der Schleifenvariable ist eigentlich nichts weiter als eine Vektorzuweisung. Die Zählervariable `i` bekommt einen Zeilenvektor mit den Werten `[0 2 6 4]` zugewiesen. Dieser Vektor wird implizit durch den `:` erzeugt. Bei der Einführung von Vektoren/Matrizen wurden zur Verbesserung des Verständnisses immer explizit eckige Klammern um den Doppelpunkt-Ausdruck gesetzt. Streng genommen sind diese aber nicht nötig und bei der Schleifeninitialisierung unüblich.

Für all diese Werte wird der Schleifenrumpf (alle Anweisungen zwischen `for` und `end`), hier die Ausgabe der Variable `i`, durchlaufen.

Der zweite Schleifentyp, die `while`-Schleife führt die Anweisungen im Schleifenrumpf solange aus, solange die logische Eintrittsbedingung, Ausdruck nach dem Schlüsselwort `while`, zutrifft. Im Beispiel wird die Variable `i` (hatte vorher den Wert 6) solange heruntergezählt, solange sie größer als 2 ist. Anschließend wird der Schleifenrumpf verlassen und die Abarbeitung hinter der Schleife fortgesetzt.

*Beispiel 1.29.:*

```
% Beispiel der in Matlab vorhanden Verzweigungen

% if-then-else-Verzweigung, Signumfunktion
x = -19;

if x > 0
    disp('1');
elseif x < 0
    disp('-1');
else
    disp('0')
end

% switch-case-Anweisung
hochschule = 'tuc';

switch hochschule
    case 'tuc'
        disp('Technische Universität Chemnitz')
    case 'tud'
        disp('Technische Universität Dresden')
    otherwise
        disp('unbekannte Hochschule')
end
```

Die sequentielle Abarbeitung bzw. Wiederholung in Form von Schleifen reicht für eine variable Programmgestaltung meistens nicht aus. Oft ist es notwendig, bestimmte Anweisung nur in Abhän-

gigkeit einer Bedingung auszuführen. Genau für diesen Zweck sind die beiden Verzweigungsmöglichkeiten `if-then-else` und `switch-case` gedacht. Im ersten Teils des obigen Beispiels, soll in Abhängigkeit der Variable  $x$  der Wert der Signumfunktion ausgegeben werden. Die Signumfunktion ist so definiert, dass sie für einen negativen Ausdruck  $-1$ , für einen positiven  $+1$  und ansonsten  $0$  liefert. Für die `if`-Verzweigung sind nur die Schlüsselworte `if` und `end` zwingend. Der `elseif`- und der `else`-Block sind nur bei Bedarf notwendig. Zu erwähnen ist noch, dass mehrere `elseif`-Blöcke durchaus erlaubt sind, der `else`-Block hingegen maximal ein mal auftreten darf.

Bei der `switch-case`-Anweisung wird der Wert einer Variable nacheinander mit verschiedenen Vorgaben verglichen. Ist eine Bedingung erfüllt, so werden die nachfolgenden Anweisungen ausgeführt und der `switch-case`-Block anschließend verlassen. Trifft keine der vorhandenen `case`-Bedingungen zu, so wird ein eventuell vorhandener `otherwise`-Block ausgeführt. Die Werte in der `case`-Bedingung können entweder Zeichenketten oder Skalare sein.

## 1.7.2 Funktionen

Wie bereits erwähnt können innerhalb von M-Files auch eigene Funktionen definiert werden, in diesem Fall nennt man sie Function-Files. Da die Variablen innerhalb einer Funktion alle lokal sind (d.h. kein Zugriff auf *Workspace* möglich), kann eine Funktion ausschließlich über Parameter bzw. ihren Rückgabewert mit **Matlab** kommunizieren. Weiterhin ist zu beachten, dass der Name des M-Files (abzüglich der Endung `.m`) identisch mit der zu definierenden Funktion sein muss, ansonsten wird die Funktion von **Matlab** nicht gefunden. Nachfolgendes Beispiel zeigt den prinzipiellen Aufbau einer Funktionsdeklaration:

```
signumfkt.m:
% Beispiel: einfache Matlab-Funktion
% Signumfunktion
function[s]=signumfkt(x)
% Eingangsvariable: x Variable deren Signumfunktion bestimmt werden soll
if x > 0
    s = 1;
elseif x < 0
    s = -1;
else
    s = 0;
end
```

Genau wie bei Script-Files sind die ersten zusammenhängenden Kommentarzeilen gleichzeitig auch der Hilfetext, der bei `help signumfkt` ausgegeben wird. Anschließend folgt die Definition der Funktion. Diese besagt, dass die neue Funktion `signumfkt` einen Parameter, der anschließend unter dem Namen  $x$  im Funktionsrumpf bekannt ist, erwartet und auch genau einen Wert zurückgibt. Die letzte Zuweisung an die Variable  $s$  innerhalb der Funktion bestimmt somit den Rückgabewert von `signumfkt`. Der Funktionsrumpf selbst ist aus dem vorherigen Beispiel kopiert. Dabei wurde lediglich die Ausgabe mittels `disp` (im Gegensatz zu `fprintf` keine Formatierung möglich) durch die Zuweisung an  $s$  ersetzt.

Ist das M-File ordnungsgemäß (Dateiname ohne Endung = Funktionsname!) gespeichert, kann man die neue Funktion wie folgt aus **Matlab** benutzen:

*Beispiel 1.30.:*

```
>> signumfkt(3)

ans =

     1

>> signumfkt(0)

ans =

     0
```

Bis jetzt wurde die Funktion lediglich rudimentär implementiert. Was passiert, wenn `signumfkt` ohne Parameter aufgerufen wird?

*Beispiel 1.31.:*

```
>> signumfkt
??? Input argument "x" is undefined.

Error in ==> signumfkt at 5
if x > 0
```

Da `x` nicht definiert ist, liefert **Matlab** einen Fehler. Schöner wäre es allerdings, wenn man genau diesen Fall innerhalb der Funktion `signumfkt` abfangen könnte und gegebenenfalls einen Hinweis ausgibt. Hier nun eine verbesserte Version:

*signumfkt2.m:*

```
% Beispiel: einfache Matlab-Funktion
% Signumfunktion
function[s]=signumfkt2(x)
% Eingangsvariable: x Variable deren Signumfunktion bestimmt werden soll
if nargin ~= 1
    error('signumfkt2 erwartet genau 1 Argument')
end

if x > 0
    s = 1;
elseif x < 0
    s = -1;
else
    s = 0;
end
```

Direkt nach der Deklaration des Funktionskopfes folgt eine Überprüfung der **Matlab**-internen Variable `nargin`. Diese enthält innerhalb einer selbstdefinierten Funktion die Anzahl der Parameter mit denen die Funktion aufgerufen wurde. Im obigen Fall wäre das also normalerweise eine 1. Ist `nargin` also ungleich 1 wird der Befehl `error` ausgeführt. Dieser erwartet als Parameter einfach eine Zeichenkette, die ausgegeben wird. Zusätzlich gibt `error` noch den Namen der Funktion und die Zeilennummer, in der die Meldung ausgegeben wurde, aus, anschließend wird die Abarbeitung der Funktion beendet. Ein Aufruf von `signumfkt2` ohne Parameter führt nun zu:

*Beispiel 1.32.:*

```
>> signumfkt2
??? Error using ==> signumfkt2 at 6
signumfkt2 erwartet genau 1 Argument
```

Die ungültige Anzahl von Parameter wurde also innerhalb der Funktion abgefangen und der selbst-definierte Fehlertext ausgegeben.

Analog zu `nargin` gibt es die Variable `nargout`. Sie gibt an, wie viele Rückgabewerte der Aufrufer erwartet. Viele **Matlab**-interne Funktionen verhalten sich abhängig von der Anzahl der gewünschten Rückgabewerte unterschiedlich. Intern ist dies durch eine Auswertung von `nargout` realisiert. Zu erwähnen ist, dass es nie mehr Parameter sein können, als in der Funktionsdeklaration vorgegeben. `nargout` kann also immer nur gleich bzw. kleiner der ursprünglich definierten Rückgabewerte sein. Ein möglicher Anwendungsfall könnte z.B. das Vermeiden aufwändiger Berechnungen sein, wenn das Ergebnis gar nicht weiterverarbeitet wird:

*compute.m:*

```
% Beispiel: Demonstration von nargout, return, warning

function[a,b]=compute(x)

% Erste Berechnung, nur ein Näherung
a = ...

if nargout < 2
    warning('Achtung Ergebnis möglicherweise nicht genau genug')
    return
end

% wenn gewünscht, jetzt die aufwändige Berechnung
b = ...
```

Die Berechnung für den ersten Rückgabewert wird im Beispiel in jedem Falle durchgeführt, anschließend wird überprüft, ob `nargout` größer als 1 ist, also zwei Rückgabewerte gefordert sind. Ist dies nicht der Fall, gibt **Matlab** mittels der Funktion `warning` eine Warnung aus und beendet anschließend durch die `return`-Anweisung die Abarbeitung des Scripts. Ansonsten wird die genauere Berechnung ausgeführt und das Ergebnis bereitgestellt.

### 1.7.3 Script-File vs. Function-File

Abschließend soll der Unterschied zwischen beiden Arten von M-Files noch einmal betont werden.

- Function-Files enthalten am Anfang die Deklaration der Funktion. Der Dateiname (ausschließlich der Endung `.m`) stimmt mit dem Funktionsnamen überein.
- Function-Files kommunizieren mittels Argument und Rückgabewerten mit **Matlab**
- In einem Function-File sind die verwendeten Variablen lokal. Im Script-File dagegen global.
- Script-Files werden einfach über ihren Namen oder das Menü aufgerufen

Befehl	Beschreibung	Beispiel
for ... end	Zählerschleife	<pre>&gt;&gt; for i=1:10     disp(i) end</pre>
while ... end	Bedingungschleife	<pre>&gt;&gt; i = 3; &gt;&gt; while i &gt; 0     disp(i)     i = i -1; end</pre>
fprintf('...', ... )	Formatierte Ausgabe	<pre>&gt;&gt; i = 2; &gt;&gt; fprintf('Inhalt von i = %d\n', i);</pre>
if ... else ... end	Bedingte Anweisung	<pre>&gt;&gt; x = -3; &gt;&gt; if x &lt; 0     s = -1 elseif x &gt; 0     s = 1 else     s = 0 end</pre>
switch ... case ... end	Fallunterscheidung	<pre>&gt;&gt; a = 1; &gt;&gt; a case 0     disp('a ist null'); case 1     disp('a ist eins'); otherwise     disp('a ist keine Binärzahl'); end a ist eins</pre>
function[ret]=name(para m)	Definiert die Funktion name.	<pre>function[a,b]=beispielfunktion(c,d) if nargin ~= 2     error('zu wenig Argumente') end  if nargin &lt; 2     warning('nur ein Rückgabewerte angefordert') return end  a = c; b = d;</pre>
nargin	Anzahl der übergebenen Parameter	
nargout	Anzahl der gewünschten Rückgabewerte	
return	Abarbeitung der Funktion beenden	
warning('...')	Warnung ausgeben	
error('...')	Warnung ausgeben und Abarbeitung beenden	

Tabelle 12: Programmierung in Matlab

## 1.8 Matlab in der Systemtheorie

Um kontinuierliche Systeme mit **Matlab** zu beschreiben, analysieren oder entwerfen ist die sogenannte *Control System Toolbox* notwendig. Die Systeme können dabei in verschiedener Art und Weise (Zeitbereich, Bildbereich oder Zustandsraumdarstellung) angegeben werden.

### 1.8.1 Differentialgleichung

Die mathematische Modellbildung kontinuierlicher Systeme im Zeitbereich wird durch eine Differentialgleichung beschrieben. Die Differentialgleichung vermittelt dabei einen dynamischen Zusammenhang zwischen den Eingangsgröße  $u(t)$  und der Ausgangsgröße  $y(t)$ . Eine lineare Differenti-

gleichung n-ter Ordnung besteht aus den reellen Koeffizienten  $a_i$  und  $b_i$ , diese spiegeln die physikalischen Parameter des Systems wieder. Die allgemeine Form (Koeffizient  $a_n=1$ ) lautet

$$\frac{d^n y}{dt^n} + a_{n-1} \frac{d^{n-1} y}{dt^{n-1}} + \dots + a_1 \frac{dy}{dt} + a_0 y(t) = b_q \frac{d^q u}{dt^q} + \dots + b_1 \frac{du}{dt} + b_0 u(t)$$

Die gesuchte Lösung einer Differentialgleichung ist nun das Verhalten der Ausgangsgröße  $y(t)$  für  $t > 0$ .

**Matlab** stellt für die numerische Lösung von Differentialgleichungen die Funktionen `ode23` und `ode45` bereit. Beide Funktionen beruhen auf dem erweiterten Runge-Kutta-Verfahren mit automatischer Schrittweitenberechnung und liefern zu einem Satz von Zeitpunkten den entsprechenden Satz von Funktionswerten. Die Befehle erwarten als ersten Parameter den Namen einer Funktion, die die Berechnungsvorschrift der Differentialgleichung darstellt sowie den Start- und Endzeitpunkt und Anfangswert. Nachfolgend soll die Lösung der Sprungantwort für ein  $PT_1$ -System vorgestellt werden. Gegeben ist die Differentialgleichung  $\dot{y}(t) = -\frac{1}{2}y(t) + \frac{5}{2}u(t)$  mit dem Anfangswert  $y_0=0$  für den Zeitbereich  $0 \leq t \leq 15$ .

*Beispiel 1.33.:*

```
>> pt1dgl=inline('-(1/2)*y+(5/2)*1', 't', 'y');  
>> t0 = 0;  
>> te = 15;  
>> y0 = 0;  
>> [t,y] = ode23(pt1dgl,[t0 te], y0);
```

Da die Berechnungsvorschrift für die DGL in einer Zeile ausgedrückt werden kann, bietet es sich an, die Funktion `pt1dgl` `inline`, das heißt direkt innerhalb des *Command Windows*, zu definieren. Alternativ ließe sich diese auch in ein M-File (Function-File) auslagern. Bevor die Differentialgleichung numerisch gelöst wird, müssen die Zeitpunkte und der Anfangswert definiert werden. Die Vektoren `t` und `y` enthalten anschließend die Sprungantwort des  $PT_1$ -Systems.

## 1.8.2 Systemmodelle mit der Control System Toolbox

Alternative Darstellungen von LTI-Systemen<sup>2</sup> sind in **Matlab** mit der *Control System Toolbox* möglich. Eine häufig genutzte Form ist die Darstellung im Bild- bzw. Laplacebereich. Differentialgleichungen n-ter Ordnung werden dabei in einfache algebraische Gleichungen überführt. Die Differentiation entspricht dabei z.B. der Multiplikation mit dem Operator  $p$ , die Integration mit  $\frac{1}{p}$ . Bei der Laplacetransformation wird vom reellen Zeitbereich ( $t$ ) in den komplexen Bildbereich<sup>3</sup> ( $p$ ) übergegangen. Für den Regelungstechniker ist insbesondere die Übertragungsfunktion  $G(p)$  interessant. Sie liefert im Bildbereich einen Zusammenhang zwischen Ein- und Ausgangsgröße.

$$G(p) = \frac{Y(p)}{U(p)}$$

Wobei  $U(p)$  die Laplace-Transformierte des Eingangssignals und  $Y(p)$  die Laplace-Transformierte des Ausgangssignals darstellt. Wie oben beschrieben, sind Laplace-Transformierte nichts anderes als eine algebraische Summe mit konstanten Koeffizienten. Sie lassen sich in **Matlab** damit

<sup>2</sup> LTI = linear time invariant = lineares zeitinvariantes System

<sup>3</sup> Matlab verwendet  $s$  anstelle von  $p$  zur Darstellung des Bildbereichs

genauso wie ein Polynom, als Zeilenvektor mit Koeffizienten absteigender Ordnung, darstellen. Folgende Übertragungsfunktion soll in **Matlab** realisiert werden:

$$G(p) = \frac{Y(p)}{U(p)} = \frac{5}{p^2 + 6p + 8}$$

*Beispiel 1.34.:*

```
>> Y = [ 5 ];           % Koeffizienten des Ausgangssignals im Bildbereich
>> U = [ 1 6 8 ];      % Koeffizienten des Eingangssignals im Bildbereich
>> sys1 = tf(Y, U)     % Übertragungsfunktion
```

```
Transfer function:
      5
-----
s^2 + 6 s + 8
```

(*U*) und (*Y*) sind die Vektoren, die die Koeffizienten des Eingangs- bzw. Ausgangssignals enthalten. `tf` liefert eine Variable, die das kontinuierliche System mit den übergebenen Parametern enthält. (**Matlab** erlaubt auch eine Betrachtung von diskreten Systemen. In der digitalen Signalverarbeitung wird mit der *z*-Transformation gearbeitet. Die Eingabe einer diskreten Übertragungsfunktion kann durch `filt` erfolgen. Dabei werden wieder die Koeffizienten des Zähler- und Nennerpolynoms, jetzt allerdings von  $z^{-1}$ , als Parameter übergeben)

Sind die Nullstellen und Pole sowie die Verstärkung eines Systems bekannt, kann man die Funktion `zpk` nutzen. `zpk` erwartet drei Vektoren als Parameter, die genau diese Werte enthalten. Das obige System, hat keine Nullstellen, zwei Pole bei -2 und -4 und eine Verstärkung von 5.

*Beispiel 1.35.:*

```
>> z = zero(sys1)

z =

Empty matrix: 0-by-1

>> p = pole(sys1)

p =

-4
-2

>> z = zero(sys1)
>> sys2 = zpk( z , p, 5 ) % Pole-Nullstellen-Darstellung
```

```
Zero/pole/gain:
      5
-----
(s+2) (s+4)
```

```
>> sys3 = tf(sys2)           % Übertragungsfunktion erzeugen
                             % sys3 ist identisch zu sys1
```

```
Transfer function:
      5
-----
s^2 + 6 s + 8
```

Zuerst wurden aus der Übertragungsfunktion die Pole und Nullstellen mittels `pole` und `zero` ermittelt (eine Darstellung des Pol-Nullstellen-Plans ist mit dem Befehl `pzmap(sys)` möglich). Diese wurden dann zusammen mit der Verstärkung als Parameter an `zpk` übergeben und eine äquivalente Übertragungsfunktion in Pol-Nullstellendarstellung erzeugt. Beide Darstellungsformen lassen sich auch ineinander überführen. Übergibt man `tf` ein kontinuierliches System, in unserem Falle `sys2`, liefert es ebenfalls die Übertragungsfunktion.

Eine weitere in der Systemtheorie übliche Darstellung von kontinuierlichen Systemen ist die Zustandsraumdarstellung (im Zeitbereich). Dabei geht man prinzipiell davon aus, dass man eine Differentialgleichung n-ter Ordnung auch als System von Differentialgleichungen 1. Ordnung darstellen kann. Die Zustandsgleichungen haben folgende Form:

$$\begin{array}{llll} \mathbf{x} & \dots & \text{Zustandsvektor} \\ \mathbf{u} & \dots & \text{Eingangsgrößen} \\ \dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} & \mathbf{y} & \dots & \text{Ausgangsgrößen} \\ \mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} & \mathbf{A} & \dots & \text{Systemmatrix} \\ & \mathbf{B} & \dots & \text{Steuermatrix} \\ & \mathbf{C} & \dots & \text{Beobachtungsmatrix} \\ & \mathbf{D} & \dots & \text{Durchgangsmatrix} \end{array}$$

Systeme in Form der Zustandsraumdarstellung lassen sich mit der Funktion `sys = ss(A,B,C,D)` eingeben. Als Parameter werden dabei einfach die entsprechenden Matrizen übergeben.

### 1.8.3 Systemeigenschaften – Verhalten im Zeit- und Frequenzbereich

Zur Analyse eines linearen kontinuierlichen Systems werden oft die sogenannte Impuls- und Sprungantwort verwendet. Die Impulsfunktion  $\delta(t)$  stellt dabei einen sehr kurzen theoretisch unendlich starken Anstieg da. Man beobachtet das System nach dem Aufschalten des Impuls' eine gewisse Zeit und erhält dadurch die Impulsantwort. Die Sprungfunktion arbeitet ähnlich, nur das diesmal zum Zeitpunkt  $t=0$  das Eingangssignal von  $x_e=0 \rightarrow x_{e0}$  übergeht und diesen Wert für die Zeit der Beobachtung beibehält. Beide Funktionen sind in **Matlab** verfügbar und werden im Folgenden an einer periodischen (d.h. Dämpfung kleiner 1)  $PT_2$ -Strecke demonstriert. Die dabei verwendeten Kennwerte sind  $K=2$ ,  $D=0.2$  und  $T=2$  und ergeben somit die Übertragungsfunktion

$$G(p) = \frac{2}{4p^2 + 0.8p + 1}$$

```
Beispiel 1.36.:
>> sys = tf( [ 2 ], [ 4 0.8 1 ] )      % PT2-System

Transfer function:
      2
-----
4 s^2 + 0.8 s + 1

>> impulse(sys)                        % Impulsantwort darstellen
>> step(sys)                           % Sprungantwort darstellen
```

Zuerst wurde ein  $PT_2$ -System in Form einer Übertragungsfunktion angegeben. Anschließend folgte die Simulation des Impuls' mit `impulse` und der Sprungantwort durch `step`. Beide Befehle er-

warten als Argument ein LTI-System. Zum Testen des Übertragungsverhaltens für ein beliebiges Eingangssignal eignet sich die Funktion `lsim`. Sie erwartet im Gegensatz zu den gerade vorgestellten Testfunktionen als zusätzliches Argument noch einen Vektor, der das Eingangssignal sowie die gewünschte Zeitbasis vorgibt. Das Ergebnis der Simulation wird wiederum als Vektor zurückgegeben und kann anschließend z.B. geplottet werden.

*Beispiel 1.37.:*

```
>> t = (0:0.4:20)'; % Zeitbasis festlegen, Spaltenvektor
>> xe = sin(t); % Eingangssignal
>> xa = lsim(sys, xe, t); % Ausgangssignal simulieren
>> plot(t, [xe xa]) % beide Signale graphisch darstellen
```

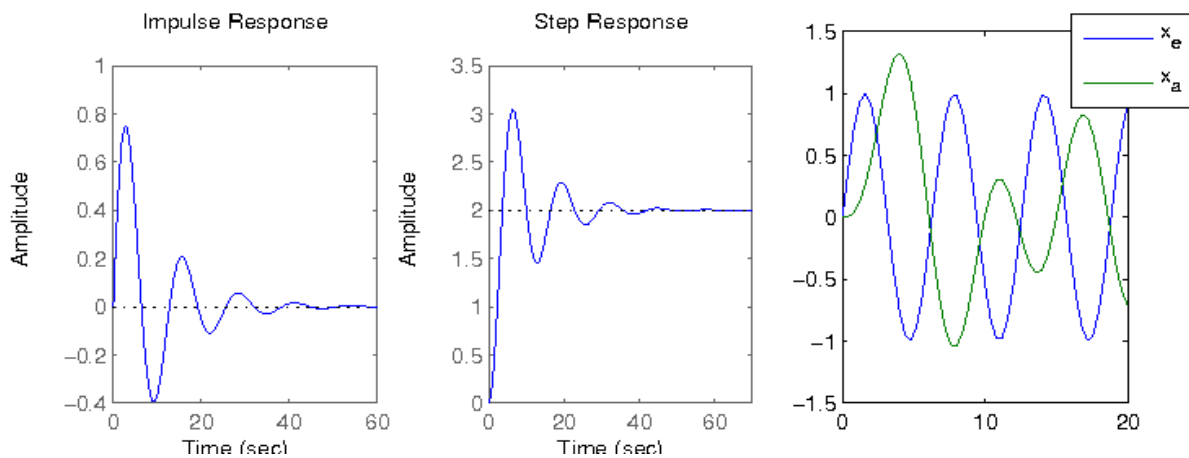


Abbildung 7: Impuls- und Sprungantwort, sowie Reaktion auf ein sinusförmiges Eingangssignal

In Abbildung 7 sind sowohl die Impuls- und Sprungantwort als auch die Reaktion des  $PT_2$ -Systems auf eine sinusförmige Eingangsgröße dargestellt. An der Sprungantwort (Step Response) kann man den Verstärkungsfaktor  $K = 2$  ablesen.

Um weitere Parameter zu bestimmen bzw. Stabilitätsaussagen über eine Strecke zu treffen ist es oftmals notwendig, eine Analyse im Frequenzbereich durchzuführen. wichtige Hilfsmittel sind dabei der Frequenzgang im Bodediagramm, die Wurzelortskurve und das Nyquist-Diagramm. Die  $PT_2$ -Strecke aus dem vorherigen Beispiel wird nun wieder aufgegriffen und weiter untersucht.

*Beispiel 1.38.:*

```
>> bode(sys) % Phasen- und Amplitudengang
>> rlocus(sys) % Wurzelortskurve
>> nyquist(sys) % Ortskurve
```

In den Diagrammen (Abbildung 8) sind einige zusätzliche Kennwerte wiederzufinden.

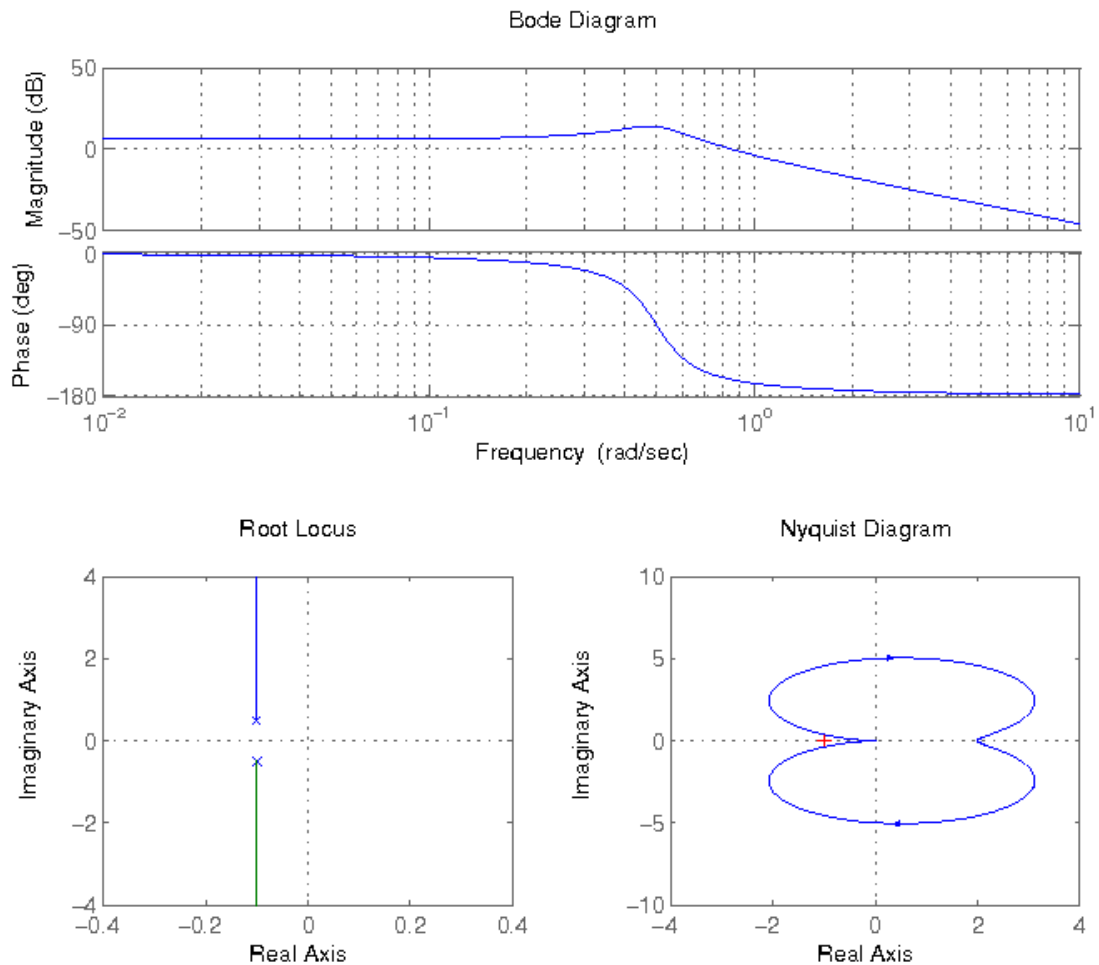


Abbildung 8: Bodediagramm, Wurzelortskurve und Nyquistdiagramm

An der Überhöhung des Amplitudengangs im Bodediagramm kann man erkennen, dass es sich um ein System mit einer Dämpfung  $< 1$  handelt. Die typische Phasenverschiebung von  $180^\circ$  ist im Phasengang sichtbar. In der Darstellung der Wurzelortskurve erkennt man, dass das System zwei imaginäre Pole besitzt. Die Wurzelortskurve eignet sich u.a. auch zur Bestimmung der Stabilität im geschlossenen Regelkreis. Dabei gilt prinzipiell, dass ein Kreis stabil ist, solange sich alle Nullstellen und Pole in der linken Halbebene befinden. Die Regel, dass ein  $PT_n$ -System auch immer durch  $n$  Quadranten verläuft, wird im Nyquist-Diagramm bestätigt. Die Ortskurve startet im vierten Quadranten auf der reellen Achse beim Verstärkungsfaktor  $K = 2$  und läuft dann über den 3. Quadranten in den Ursprung zurück.

### 1.8.4 Regelkreisstrukturen

Regelkreise bestehen üblicherweise aus einer Verschaltung mehrerer Blöcke bzw. Übertragungsfunktionen. Dabei können Reihen- und Parallelschaltungen oder eine Kreisstruktur vorkommen. Die Zusammenfassung und Vereinfachung der verschiedenen Übertragungsfunktionen zu einer einzelnen Übertragungsfunktion ist für eine weitere Betrachtung des Regelkreises oft notwendig. Innerhalb der *Control System Toolbox* von **Matlab** gibt es einige Funktionen, die dieses Vorgehen un-

terstützen. Reihenschaltungen sind im Bildbereich äquivalent zu einer Multiplikation, Parallelschaltungen entsprechen einer Addition mit Berücksichtigung des Vorzeichens an der Summationsstelle.

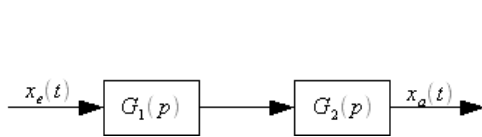


Abbildung 9: Reihenschaltung

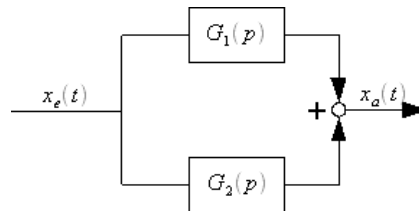


Abbildung 10: Parallelschaltung

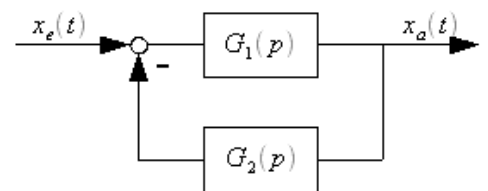


Abbildung 11: Rückkopplung

**Beispiel 1.39.:**

```
>> g1 = tf( [ 1 1 ], [ 1 4 ] );           % erste Übertragungsfunktion
>> g2 = tf( [ 1 -2 ], [ 1 3 2 ] );      % zweite Übertragungsfunktion
>> g3 = g1 * g2                          % Reihenschaltung
                                         % mittels Multiplikation

Transfer function:
      s^2 - s - 2
-----
s^3 + 7 s^2 + 14 s + 8

>> g4 = series(g1, g2)                  % Reihenschaltung über series

Transfer function:
      s^2 - s - 2
-----
s^3 + 7 s^2 + 14 s + 8

>> g5 = g1 + g2                         % Parallelstruktur mit Addition

Transfer function:
s^3 + 5 s^2 + 7 s - 6
-----
s^3 + 7 s^2 + 14 s + 8

>> g6 = parallel(g1, g2)               % Parallelstruktur mit paralle

Transfer function:
s^3 + 5 s^2 + 7 s - 6
-----
s^3 + 7 s^2 + 14 s + 8
```

Im obigen Beispiel wurde für zwei Übertragungsfunktionen  $G_1(p) = \frac{p+1}{p+4}$  und  $G_2(p) = \frac{p-2}{p^2+3p+2}$  jeweils das Ergebnis der Reihen- und Parallelschaltung berechnet. Zusätzlich zur einfachen Multiplikation bzw. Addition lassen sich Reihen- und Parallelschaltungen auch mit den Befehlen `series` und `parallel` umformen. Dass beide Verfahren jeweils das gleiche Ergebnis liefern, sieht man im Beispiel.

Eine Kreisstruktur, d.h. Struktur mit Gegenkopplung, zweier Übertragungsfunktionen ergibt im Bildbereich einen Quotienten aus Vorwärtszweig dividiert durch  $1 +$  gesamter Kreis. Für die in Abbildung 11 dargestellte Schaltung ergibt das folgenden algebraischen Ausdruck

$$G(p) = \frac{G_1(p)}{1 + G_1(p) \cdot G_2(p)}$$

Nachfolgend ein Beispiel für  $G_1(p) = \frac{p+1}{p+4}$  und  $G_2(p) = \frac{3}{p+6}$

*Beispiel 1.40.:*

```
>> g1 = tf([ 1 1 ], [ 1 4 ])           % Übertragungsfunktion im
                                       % Vorwärtszweig
Transfer function:
s + 1
-----
s + 4

>> g2 = tf([ 3 ], [ 1 6 ])           % Übertragungsfunktion in der
                                       % Rückkopplung
Transfer function:
3
-----
s + 6

>> feedback(g1,g2)                   % Kreisstruktur berechnen

Transfer function:
s^2 + 7 s + 6
-----
s^2 + 13 s + 27
```

Die **Matlab**-Funktion `feedback` ermittelt die zusammengefasste Übertragungsfunktion. Handelt es sich um eine direkte Gegenkopplung, wird für die zweite Übertragungsfunktion (Gegenkopplung) einfach eine 1 eingesetzt. Möchte man statt einer Gegenkopplung eine Mitkopplung modellieren kann man `feedback` als drittes Argument noch die Art der Rückführung übergeben. Standardmäßig wird dabei ein Wert von -1, was einer Gegenkopplung entspricht, angenommen. Für den Wert 1 hingegen handelt es sich um eine Mitkopplung.

Befehl	Beschreibung	Beispiel
sys = tf([num], [den])	Übertragungsfunktion erzeugen	<pre>&gt;&gt; sys1 = tf([1 0.5], [2 3])  Transfer function: s + 0.5 ----- 2 s + 3</pre>
sys = ss(A, B, C, D)	Übertragungsfunktion in Zustandsraumdarstellung erzeugen	<pre>&gt;&gt; A = [ 0 -0.4; 1 -1.3]; &gt;&gt; B = [ 0.8; 1 ]; &gt;&gt; C = [ 0 1 ]; &gt;&gt; D = [ 0 ]; &gt;&gt; sys_ss = ss(A,B,C,D);</pre>
[t,y] = ode23(function, [t0 te], y0)	Differentialgleichung numerisch lösen	<pre>&gt;&gt; ptldgl=inline('-(1/2)*y+(5/2)*1', 't', 'y'); &gt;&gt; t0 = 0; &gt;&gt; te = 15; &gt;&gt; y0 = 0; &gt;&gt; [t,y] = ode23(ptldgl,[t0 te], y0);</pre>
sys = filt([num], [den])	diskrete Übertragungsfunktion	<pre>&gt;&gt; sys_d = filt([1 0.5], [2 3])  Transfer function: 1 + 0.5 z^-1 ----- 2 + 3 z^-1</pre>
sys = zpk([p], [n], [k])	Übertragungsfunktion in Pol-Nullstellen-Darstellung	<pre>&gt;&gt; sys3 = zpk([-0.5], [-1.5], 2)  Zero/pole/gain: 2 (s+0.5) ----- (s+1.5)</pre>
sys3 = series(sys1, sys2)	Reihenschaltung zweier Übertragungsfunktionen	<pre>&gt;&gt; series(tf([1],[2 3]), tf([0.5], [4]))  Transfer function: 0.5 ----- 8 s + 12</pre>
sys3 = parallel(sys1, sys2)	Parallelschaltung zweier Übertragungsfunktionen	<pre>&gt;&gt; parallel(tf([1],[2 3]), tf([0.5], [4]))  Transfer function: s + 5.5 ----- 8 s + 12</pre>
sys3 = feedback(sys1, sys2, sign)	Regelkreis schließen, Rückkopplung	<pre>&gt;&gt; feedback(tf([1], [3 2]), 1)  Transfer function: 1 ----- 3 s + 3</pre>
p = pole(sys)	Pole der Übertragungsfunktion ermitteln	<pre>&gt;&gt; pole(sys1)  ans =  -1.5000</pre>
z = zero(sys)	Nullstellen der Übertragungsfunktion ermitteln	<pre>&gt;&gt; zero(sys1)  ans =  -0.5000</pre>
ns = order(sys)	Ordnung des LTI Systems	<pre>&gt;&gt; order(sys1)  ans =  1</pre>

<code>impulse(sys)</code>	Impulsantwort simulieren	<code>&gt;&gt; impulse(sys1)</code>
<code>step(sys)</code>	Sprungantwort simulieren	<code>&gt;&gt; step(sys1)</code>
<code>xa = lsim(sys,xe,t)</code>	Simulation beliebiges Eingangssignal	<code>&gt;&gt; sys = tf([2], [4 0.8 1]); &gt;&gt; t = (0:0.4:20)'; &gt;&gt; xe = sin(t); &gt;&gt; xa = lsim(sys, xe, t); &gt;&gt; plot(t, [xe xa]);&gt;&gt;</code>
<code>bode(sys)</code>	Bodediagramm	<code>&gt;&gt; bode(sys1)</code>
<code>nyquist(sys)</code>	Ortskurve	<code>&gt;&gt; nyquist(sys1)</code>
<code>pzmap(sys)</code>	Pol-Nullstellen-Plan	<code>&gt;&gt; pzmap(sys1)</code>
<code>rlocus(sys)</code>	Wurzelortskurve	<code>&gt;&gt; rlocus(sys1)</code>
<code>k = dcgain(sys)</code>	Statischer Verstärkungsfaktor des Systems	<code>&gt;&gt; dcgain(sys1)</code>  <code>ans =</code>  <code>0.1667</code>
<code>[w0, d] = damp(sys)</code>	Dämpfung des Systems berechnen	<code>&gt;&gt; % PT2 mit Dämpfung 0.2 &gt;&gt; sys = tf( [ 2 ], [4 0.8 1] ); &gt;&gt; [w0,d] = damp(sys) w0 = 0.5000 0.5000 d = 0.2000 0.2000</code>
<code>[y,x,t] = initial(sys, x0)</code>	Anfangswertantwort des Systems bei fehlendem Eingangssignal (nur für Zustandsraumdarstellung)	<code>&gt;&gt; [y,x,t] = initial(sys_ss, [3 2]);</code>
<code>sys = minreal(sys)</code>	Kürzen von Polen und Nullstellen in Übertragungsfunktion	<code>&gt;&gt; sys = zpk([0 1], [2 1 1], 1)</code>  <code>Zero/pole/gain:</code> <code>s (s-1)</code> <code>-----</code> <code>(s-2) (s-1)^2</code>  <code>&gt;&gt; minreal(sys)</code>  <code>Zero/pole/gain:</code> <code>s</code> <code>-----</code> <code>(s-2) (s-1)</code>
<code>[g, p] = margin(sys)</code>	Phasen- und Amplitudenreserve	<code>&gt;&gt; margin(sys1)</code>

Tabelle 13: Befehle für die Analyse linearer zeitinvarianter Systeme

### 1.8.5 Signalgenerierung

Oftmals ist es für die Analyse linearer zeitinvarianter Systeme notwendig, ein bestimmtes Eingangssignal vorzugeben. Neben den als Befehlen vorhandenen Signalen des Einheitssprungs und des Dirac-Impuls‘ ist oft auch ein kontinuierliches harmonisches Signal von Interesse. Üblicherweise bieten sich hier Sinus- bzw. Kosinusfunktionen an. Für einen Zeitbereich von  $0 \leq t \leq 20$  Sekunden soll sowohl ein reines Sinussignal als auch ein mit zufälligen Störungen beaufschlagter Kosinus generiert werden.

**Beispiel 1.41.:**

```
>> t = [0:0.1:20];
>> y_sin = sin(t);
>> y_cos = cos(t) + 0.1 * randn(size(t));
>> plot([t], [y_cos; y_sin])
>> legend('glatter Sinus', 'verrauschter Kosinus');
```

Die beiden Signale sind in Abbildung 12 zu sehen.

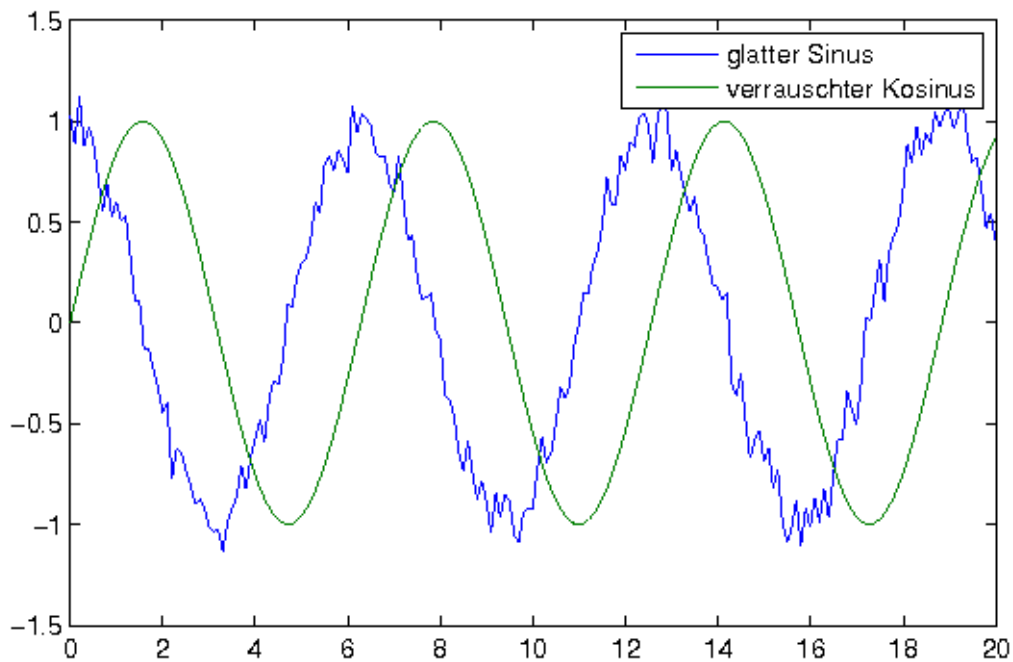


Abbildung 12: Reine und verrauschte harmonische Signale

Befehl	Beschreibung	Beispiel
<code>y = sin(x)</code>	Sinus für Argument berechnen	<code>y = sin( [ 1 2 3 4 ] )</code>
<code>y = cos(x)</code>	Kosinus für Argument berechnen	<code>y = cos( [ 0:0.1:10 ] )</code>
<code>R = rand(n, m)</code>	Matrix der Dimension n x m mit positiven Zufallszahlen erzeugen	<pre>&gt;&gt; rand(1,2); ans =     0.83187    0.57217r</pre>
<code>R = randn(n, m)</code>	Matrix der Dimension n x m mit normalverteilten Zufallszahlen erzeugen	<pre>&gt;&gt; randn(1,2) ans =    -0.82250    0.49130</pre>

Tabelle 14: Übersicht Signalgenerierung

### 1.8.6 Statistik

Statistische Verfahren sind z.B innerhalb der Prozessanalyse und Identifikation ein wichtiges Hilfsmittel um Eigenschaften von Systemen anhand gemessener Werte zu ermitteln. Im nachfolgenden Beispiel werden einige grundlegenden Funktionen vorgestellt:

*Beispiel 1.42.:*

```
>> d = [1 4 6 2 5 8 3];      % Vektor mit willkürlichen Werten
>> sort(d)                  % Vektor aufsteigenden sortieren
ans =

     1     2     3     4     5     6     8

>> min(d)                   % kleinstes Element innerhalb des Vektors
ans = 1
>> max(d)                   % größtes Element innerhalb des Vektors
ans = 8
>> sum(d)                   % aufsummieren aller Elemente
ans = 29
>> prod(d)                  % Produkt aller Vektorelemente
ans = 5760
```

Nachdem der Vektor `d` mit einigen willkürlichen Werten erzeugt wurde, wird er mit der Funktion `sort` aufsteigenden sortiert. Das kleinste Element innerhalb eines Vektors liefert der Befehl `min`. Analog dazu wird `max` benutzt, um das größte Element zu ermitteln. `sum` und `prod` summieren bzw. multiplizieren alle Elemente des Vektors der Reihe nach auf und liefern einen Skalar als Ergebnis.

*Beispiel 1.43.:*

```
>> diff(d)                  % Differenz der benachbarten Elemente
ans =

     3     2    -4     3     3    -5

>> cumsum(d)                % aufsummieren und Teilergebnisse liefern
ans =

     1     5    11    13    18    26    29

>> cumprod(d)              % aufmultiplizieren und Teilergebnisse liefern
ans =

     1     4    24    48    240   1920   5760
```

`diff` und `cumsum` entsprechen in etwa einer diskreten Differentiation bzw. Integration. Im Falle von `diff` werden die Differenzen der benachbarten Elemente gebildet und als Zeilenvektor zurückgegeben. `cumsum` summiert ähnlich zu `sum` die einzelnen Elemente des Vektors auf, gibt aber in jedem Teilschritt auch das Zwischenergebnis zurück. `cumprod` verhält sich wie `cumsum` nur mit dem Unterschied, dass multipliziert wird.

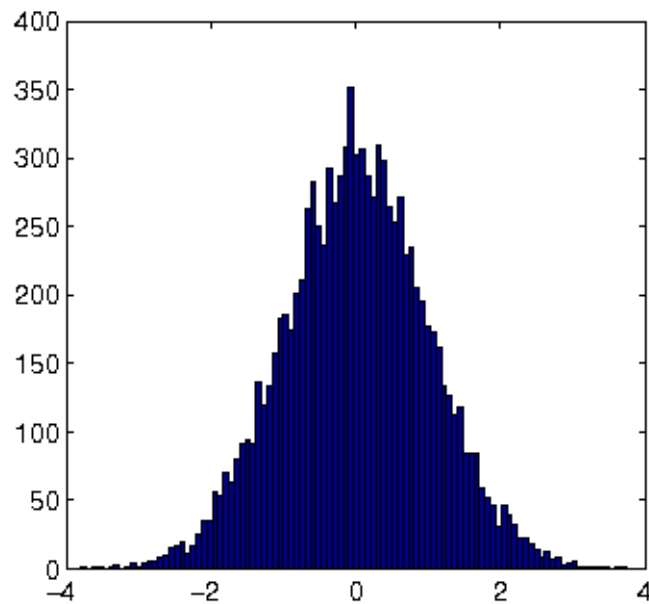


Abbildung 13: Histogramm einer Normalverteilung

Das in Abbildung 13 dargestellte Histogramm zeigt die Normalverteilung (auch Gaußlocke) eines mit `randn` erstellten Zufallsvektors (10000 Elemente). Es wurde mit folgenden **Matlab**-Anweisungen erzeugt.

*Beispiel 1.44.:*

```
>> y = randn(10000, 1);  
>> hist(y, 100)
```

`hist` erstellt aus den Werten des Vektors  $y$  ein Histogramm, eine Darstellung der Häufigkeitsverteilung. Der zweite Parameter gibt an, auf wie viele Klassen die Werte verteilt werden sollen. Standardmäßig ist dieser Wert in **Matlab** auf 10 voreingestellt, wurde hier aber mit 100 überschrieben. Kennwerte von Häufigkeitsverteilungen sind u.a. der Mittelwert bzw. Erwartungswert  $\mu$ , die Standardabweichung  $\sigma$  und Varianz  $\sigma^2$  sowie der Median. Laut Theorie liefert der Mittelwert für eine gleichverteilte Menge den Wert 0, die Standardabweichung beträgt 1. Da die Varianz sich aus dem Quadrat der Standardabweichung berechnet, erhält man für diese ebenfalls den Wert 1. Der Median  $m$ , ein ähnliches Maß wie der Mittelwert, ist der Wert, für den gilt, dass maximal die Hälfte der zu untersuchenden Wert  $< m$  sowie maximal die Hälfte  $> m$  ist. Im Falle einer Normalverteilung liefert er ebenfalls den Wert 0. Nachfolgend die Überprüfung in **Matlab**.

**Beispiel 1.45.:**

```
>> mean(y)           % Mittelwert berechnen
ans = 0.0081407
>> std(y)            % Standardabweichung berechnen
ans = 0.99767
>> var(y)            % Varianz berechnen
ans = 0.99535
>> median(y)         % Median berechnen
ans = -0.0010450
```

Wie zu erwarten stimmen die Ergebnisse überein. Die geringe Abweichung von 0 bzw. 1 ergeben sich aus der zu niedrigen Anzahl von Messwerten. Umso mehr Werte der Vektor *y* enthält umso besser nähern sich die Kennzahlen den theoretischen Werten an.

Befehl	Beschreibung	Beispiel
<code>c = min(x)</code>	kleinstes Element des Vektors	<pre>&gt;&gt; min([ 3 2 10 ]) ans = 2</pre>
<code>c = max(x)</code>	größtes Element des Vektors	<pre>&gt;&gt; max([ 3 2 10 ]) ans = 10</pre>
<code>m = mean(x)</code>	Mittelwert	<pre>&gt;&gt; mean([ 3 2 10 ]) ans = 5</pre>
<code>m = median(x)</code>	Median	<pre>&gt;&gt; r = randn(1, 10000); &gt;&gt; median(r) ans =     0.0229</pre>
<code>s = std(x)</code>	Standardabweichung	<pre>&gt;&gt; std(randn(1, 10000)) ans =     1.0036</pre>
<code>v = var(x)</code>	Varianz	<pre>&gt;&gt; var(randn(1, 10000)) ans =     1.0098</pre>
<code>xs = sort(x)</code>	Vektor sortieren	<pre>&gt;&gt; sort([ 3 2 10]) ans =      2     3    10</pre>
<code>s = sum(x)</code>	Vektorelemente aufsummieren	<pre>&gt;&gt; sum([ 3 2 10 ]) ans = 15</pre>
<code>p = prod(x)</code>	Vektorelemente multiplizieren	<pre>&gt;&gt; prod([ 3 2 10 ]) ans = 60</pre>
<code>[p] = cumprod(x)</code>	kumulatives Produkt	<pre>&gt;&gt; cumprod([1 2 3 4]) ans =      1     2     6    24</pre>
<code>[d] = diff(x)</code>	Differenz benachbarter Elemente	<pre>&gt;&gt; diff([1 3 6 10]) ans =      2     3     4</pre>
<code>[s] = cumsum(x)</code>	kumulative Summe	<pre>&gt;&gt; cumsum([1 2 3 4]) ans =      1     3     6    10</pre>
<code>hist(y, k)</code>	Histogramm des Vektors <i>y</i> mit <i>k</i> Klassen	<pre>&gt; hist(randn(1,10000), 10)&gt;</pre>

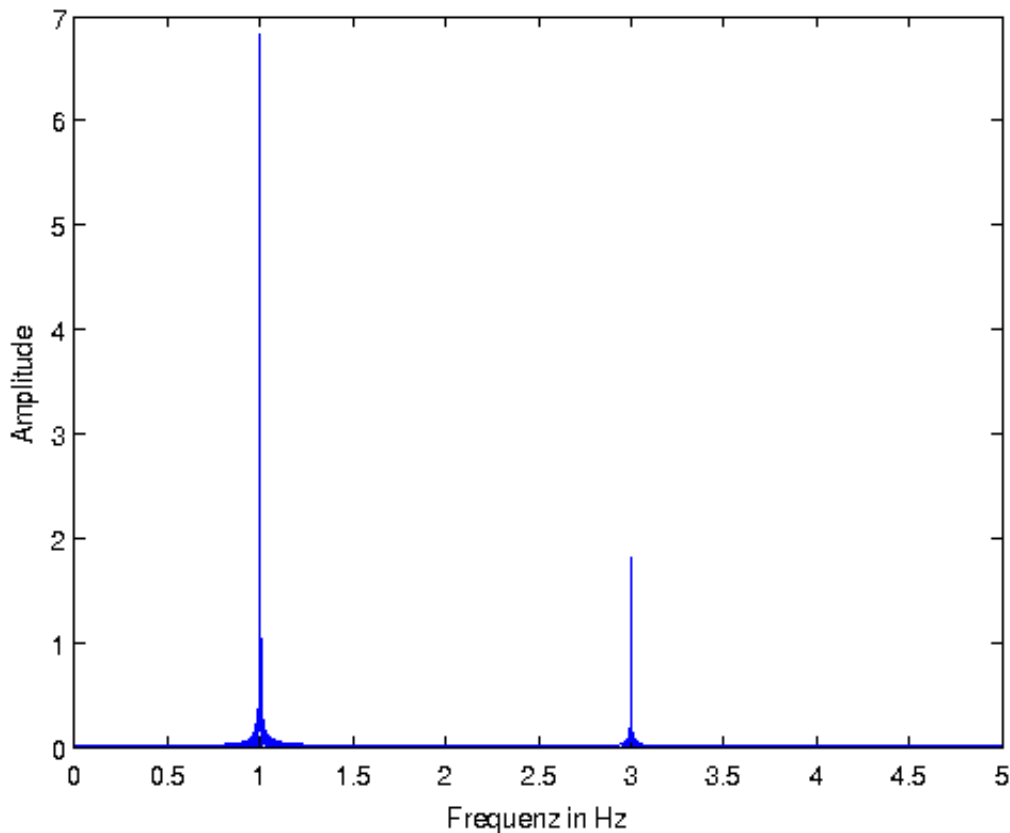
*Tabelle 15: Übersicht Statistikbefehle*

## 1.8.7 Fourieranalyse – Spektren

*Beispiel 1.46.:*

```
>> Fs = 10; % Abtastfrequenz festlegen, 10Hz
>> t = 0:1/Fs:500; % Abtastzeitraum
>> x = 2*cos(3*2*pi*t) + 7*cos(1*2*pi*t); % Abtastwerte erzeugen
>> f = fft(x,8192) / length(t);
>> N = length(f);
>> freq = [0:N/2-1] * Fs / N; % Frequenzachse, nur pos. Frequenzen
>> plot(freq, 2 * abs(f(1:N/2))); % Spektrum ausgeben
>> xlabel('Frequenz in Hz');
>> ylabel('Amplitude');
```

Hat man diskrete Messwerte eines Prozesses aufgenommen und interessiert sich nun für die in der Messung vorkommenden Frequenzen, kann man eine Fourieranalyse durchführen. Aufgrund der erhöhten Effizienz hat sich bei der Berechnung auf Computern die sogenannte Fast Fourier Transformation = FFT durchgesetzt. Diese arbeitet mit einem sogenannten Fenster von Messwerten. Um wirklich schnell zu sein, sollte dieses Fenster immer einer Zweierpotenz entsprechen, die größer als die Anzahl der abgetasteten Werte ist. Nachfolgendes Beispiel überlagert zwei Kosinussignale mit einer Frequenz und Amplitude von  $f_1=3, a_1=2$  und  $f_2=1, a_2=7$  für eine Zeitspanne von 500 Sekunden. Dabei soll mit einer Abtastrate von 10 Hz gearbeitet werden.



*Abbildung 14: Spektrum der beiden überlagerten Kosinusfunktionen*

Das entstandene Frequenzspektrum ist in Abbildung 14 zu sehen. Die Frequenz und Amplitude beider Kosinusfunktion wurde richtig erkannt. Die Fouriertransformatin wird in **Matlab** mit `fft`

durchgeführt. Erstes Argument ist der Vektor mit Messwerten des zu analysierenden Signals. Der zweite Parameter gibt die Größe des Fensters für die Fast Fourier Transformation an. Da die FFT Werte im Bereich von  $-\frac{f_s}{2}$  bis  $\frac{f_s}{2}$  liefert, wird nur die erste Hälfte (positive Frequenzen) dargestellt. Um die Frequenzen symmetrisch zum Null-Punkt auszugeben, würde man das Ergebnis der FFT mit `fftshift` zentriert.

Befehl	Beschreibung	Beispiel
<code>f = fft(x,b)</code>	Diskrete Fast Fourier Transformation aus x mit Fensterbreite b berechnen	<pre>&gt;&gt; t = 0:0.1:500; &gt;&gt; x = 2*cos(3*2*pi*t) + 7*cos(1*2*pi*t); &gt;&gt; f = fft(x, 8192); &gt;&gt; freq=-5:10/8192:5-1/8192; &gt;&gt; plot(freq, 2 * abs(fftshift(f)))</pre>
<code>fz = fftshift(f)</code>	Ergebnis der FFT um Nullpunkt zentrieren	

Tabelle 16: Befehle Fourieranalyse

### 1.8.8 Filterung

Die Glättung von Messwerten einer Zeitreihe ist ein Verfahren um Ausreißer bzw. zu starke Abweichungen über einen bestimmten Zeitraum herauszurechnen. Ein typisches Glättungsverfahren ist der sogenannte gleitende Mittelwert. Er wirkt wie ein Tiefpass-Filter und führt letztendlich zur Glättung der Zeitreihe. Für die Berechnung des gleitenden Mittelwerts wird immer nur ein Teil (Fenster) der vorhandenen Werte berücksichtigt. Im Prinzip berechnet man mehrmals den arithmetischen Mittelwert aus aufeinander folgenden Werten der Zeitreihe und erhält somit den gleitenden Mittelwert. Folgende Gleichung gibt die Berechnungsvorschrift für den gleitenden Mittelwert 3. Ordnung an (er bedient sich nur aus Werten der Vergangenheit bzw. Gegenwart):

$$\bar{y}_t = \frac{y_t + y_{t-1} + y_{t-2}}{3}$$

Die Ordnung gibt dabei die „Breite“ des Fensters an. Somit kann man nun für die Werte aus der Zeitreihe den entsprechend geglätteten gleitenden Mittelwertverlauf berechnen. Wie man sieht, liefert der gleitende Mittelwert keine (verlässliche) Aussage über die Randbereiche. Eine Berechnung in **Matlab** könnte wie folgt aussehen:

```

Beispiel 1.47.:
>> x_orig = [ 408 372 480 444 447 492 429 411 486 525 495 ]; % Ausgangssignal
>>
>> m = 3; % gl. Mittelwert 3. Ordnung
>> x = [zeros(1,m-1) x_orig]; % Anfang mit Nullen auffüllen
>>
>> for i=m:length(x)
>> xg(i-m+1) = ( x(i) + x(i-1) + x(i-2) ) / m;
>> end
>> xg
xg =
    136    260    420    432    457    461    456    444    442    474    502

>> plot([1:length(x_orig)], [x_orig; xg])
>> legend('ursprüngliche Zeitreihe', 'geglättete Zeitreihe')

```

Im Vektor  $x_{orig}$  werden die einzelnen Elemente der Zeitreihe hinterlegt. Da die Berechnungsvorschrift Werte in der Vergangenheit berücksichtigt, müssen dem Ursprungsvektor noch zwei Nullelemente vorangestellt werden, ansonsten wäre der Index ungültig. Anschließend wird mit Hilfe einer `for`-Schleife die oben genannte Berechnungsvorschrift des gleitenden Mittelwerts implementiert. Im Vektor  $x_g$  befinden sich nun die Werte der geglätteten Zeitreihe. Um beiden Zeitreihen optisch miteinander zu vergleichen, werden sie mit dem `plot`-Befehl ausgegeben (siehe Abbildung 15) und einer Legende versehen.

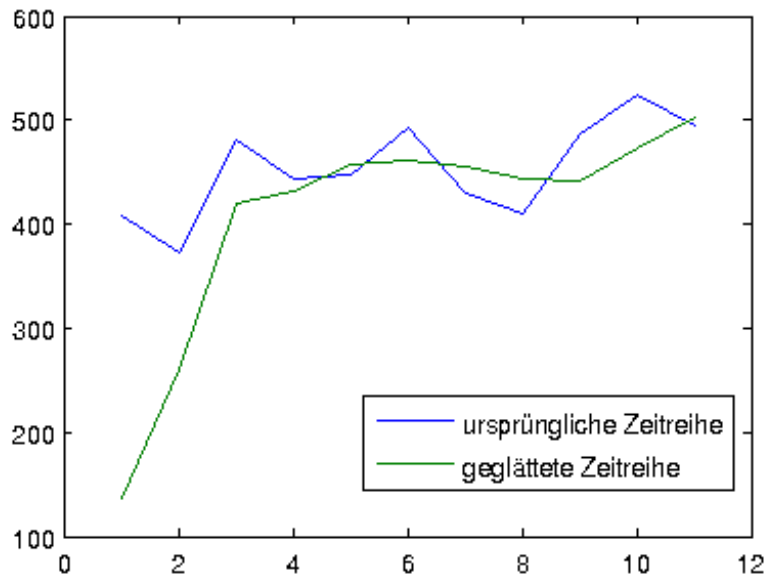


Abbildung 15: Glättung einer Zeitreihe

Gleiches ließe sich auch kürzer mit der **Matlab**-Funktion `filter` realisieren. Für die Berechnung eines gleitenden Mittelwerts erwartet die Funktion drei Argumente: Zähler und Nenner des Filters sowie einen Vektor, der die Werte der Zeitreihe enthält. Hier das Beispiel für die selbe Zeitreihe wie oben mit einem gleitenden Mittelwert 3. Ordnung:

*Beispiel 1.48.:*

```
>> m = 3; % Breite des Fenster = 3. Ordnung
>> b = [ 1/m 1/m 1/m ]; % Koeffizienten für Filter der Eingangsgrößen
>> a = 1; % Filter für Ausgangsgrößen
>> filter(b, a, x_orig) % gl. Mittelwert berechnen
ans =
    136    260    420    432    457    461    456    444    442    474    502
```

Die Filterfunktion ist innerhalb von **Matlab** durch folgende Berechnungsvorschrift realisiert:

$$[b_0 \ b_1 \ \dots] \cdot \begin{bmatrix} x(n) \\ x(n-1) \\ \dots \end{bmatrix} - [1 \ a_1 \ a_2 \ \dots] \cdot \begin{bmatrix} y(n) \\ y(n-1) \\ \dots \end{bmatrix} = 0$$

Ist der Index für den Vektor  $x$  kleiner als 1, wird ein Wert von 0 zurückgegeben. Dadurch entsteht das selbe Ergebnis wie im Beispiel 1.47.

Befehl	Beschreibung	Beispiel
<code>y = filter(b, a, x)</code>	Lineare zeitinvariante Differenzgleichung	<pre>% gl. Mittelwert 3. Ordnung &gt;&gt; b = [ 1/3 1/3 1/3 ]; &gt;&gt; a = [ 1 ]; &gt;&gt; y = filter(b, a, x);</pre>

Tabelle 17: Filterbefehle

### 1.8.9 Interpolation und Regression

Den Verlauf eines Prozesses aus diskreten Messwerten zu ermitteln, d.h einen funktionalen Zusammenhang der Ein- und Ausgangsgrößen anzugeben, wird als Identifikation bezeichnet und gehört u.a. zu den Aufgabengebieten der Prozessanalyse bzw. Systemtheorie. Im Abschnitt 1.8.8 wurde gezeigt, wie man für die Messwerte einer Zeitreihe eine geglättete Kurve erhält. Dabei war es allerdings nicht möglich, potentielle Werte zwischen zwei Messpunkten anzugeben. Durch eine Interpolation bzw. Regression von Messwerten kann dies erreicht werden und zusätzlich eine kompakte Darstellung des Prozesses erfolgen. Eine Interpolation unterscheidet sich von einer Regression dadurch, dass sie die vorgegebenen Punkte (Stützstellen) möglichst exakt trifft, während eine Regression eher eine Art Ausgleichsgerade darstellt die versucht, die Abweichung von allen Punkten zu berücksichtigen.

Die einfachste Form der Interpolation ist die lineare Interpolation (nach *Newton*). Durch Verbinden benachbarter Punkte mit einer Strecke entsteht die Interpolationsfunktion. Bereits in der Schulmathematik wurde ein weiteres Interpolationsverfahren, die polynomiale Interpolation, vorgestellt. Laut dem Fundamentalsatz der Algebra lassen sich  $n+1$  verschiedenen Wertepaare durch ein Polynom  $n$ -ten Grades beschreiben. Die Bestimmung des Interpolationspolynoms läuft schlussendlich auf die Lösung eines linearen Gleichungssystems hinaus. Obwohl dieses Verfahren die vorgegebenen Punkte exakt wiedergibt, hat es einige Probleme. Neben dem überproportional steigenden Rechenaufwand des Verfahrens für weitere Messwerte, neigen Polynome höheren Grades ( $> 5$ ) zum Oszillieren. Oszillation bedeutet hierbei, dass sich der Anstieg des Polynoms zwischen zwei Messwerten stark ändert. Da man in der Praxis allerdings meist davon ausgeht, dass die Interpolation zwischen zwei Punkten verhältnismäßig glatt ist, ist dieses Verfahren ungeeignet. Abhilfe schafft hier u.a. eine stückweise Interpolation. Im linearen Fall ist das ein Polygonzug, für 2. und 3. Grad spricht man von einer Spline-Interpolation. Wichtig ist bei abschnittsweise definierten Funktionen die Frage der Stetigkeit und Differenzierbarkeit an den Übergangspunkten.

Hat man eine große Anzahl von Wertepaaren, würde eine genaue Beschreibung relativ komplexe Interpolationspolynome hervorbringen. Durch Regression lassen sich einfachere funktionale Zusammenhänge finden, die die vorgegebenen Werte zwar nicht exakt treffen, zumindest aber versuchen, die Abweichung aller Punkte gering zu halten. Die Vorgehensweise besteht i.d.R. aus der Vorgabe eines Modells und der anschließenden Schätzung der Koeffizienten über die Methode der kleinsten Quadrate.

Innerhalb von **Matlab** lassen sich die Koeffizienten für lineare Interpolations- und Regressionspolynome mit die Funktion `polyfit` bestimmen. `polyfit` erwartet als Argumente die Wertepaare in Form zweier Vektoren sowie den Grad des gewünschten Polynoms. Ist der Grad des Polynoms kleiner als die Anzahl der vorgegebenen Stützstellen führt **Matlab** eine Regression im Sinne der kleinsten Quadrate durch. Ansonsten ist das Ergebnis eine in den Stützstellen exakte polynomiale Interpolation.

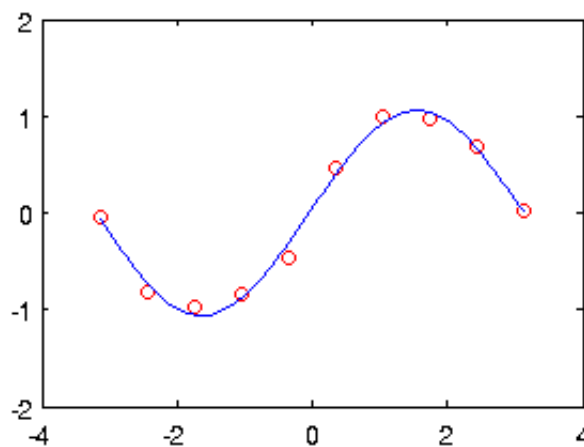
Nachfolgend ein kleines Beispiel. Es existieren 10 Wertepaare (ursprünglich aus einer Sinusfunktion mit Störung). Diese sollen als Basis für eine Regression 5. Grades dienen.

*Beispiel 1.49:*

```
>> x = linspace(-pi, pi, 10);           % 10 x-Werte
>> y = sin(x) + 0.1 * randn(size(x));   % Funktionswerte mit Störung

>> p = polyfit(x, y, 5);                 % lineare Regression (5. Grades)
>> x2 = linspace(-pi, pi, 100);        % genauere x-Auflösung
>> y2 = polyval(p, x2);                 % y-Werte aus Regressionspolynom
>> plot(x,y, 'ro', x2, y2)             % vergleichende Darstellung
```

Das Ergebnis ist in Abbildung 16 zu sehen. Nachdem die Stützstellen erzeugt worden, werden aus diesen die Koeffizienten des Ausgleichspolynoms 5. Grades berechnet. Um zu sehen, wie gut die Regression das ursprüngliche Signal beschreibt, wird der Funktionswert des Regressionspolynoms



*Abbildung 16: Ergebnis der polynomialen Regression*

in kleinen Schritten mit `polyval` berechnet und ausgegeben. Natürlich ist eine polynomiale Regression für eine Schwingung nicht sonderlich geeignet (außerhalb der vorgegebenen Punkte liefert sie keine sinnvollen Werte mehr), dennoch sollte die prinzipielle Vorgehensweise klar geworden sein.

Für die Interpolation mittels Splines muss Beispiel 1.49 nur geringfügig abgeändert werden.

*Beispiel 1.50:*

```
>> x = linspace(-pi, pi, 10);           % 10 x-Werte
>> y = sin(x) + 0.1 * randn(size(x));   % Funktionswerte mit Störung

>> pp = spline(x,y);                    % Spline Interpolation
>> x2 = linspace(-pi, pi, 100);        % genauere x-Auflösung
>> y2 = ppval(pp,x2);                   % y-Werte des stückweisen Spline
>> plot(x,y, 'ro', x2, y2)             % vergleichende Darstellung
```

Zwei Unterschiede fallen auf. Zum einen erfolgt die Interpolation nun durch die Funktion `spline`, die als Argumente die Stützstellen in Vektorform entgegennimmt und zum anderen kann der Funktionswert von Splines nicht mit `polyval` berechnet werden. Stattdessen kommt die Funktion `ppval` zum Einsatz. Die in `pp` hinterlegten Spline-Parameter werden dabei in Abhängigkeit vom `x`-Wert ausgewertet und der Funktionswert berechnet.

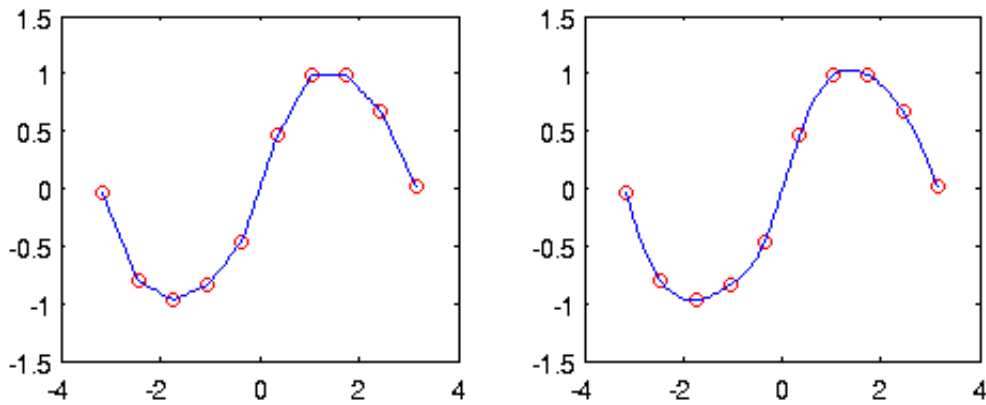
Zusätzlich existiert in **Matlab** noch die Funktion `interp1` bzw. die Varianten `interp2` und `interp3` für zwei- bzw. dreidimensionale Interpolation. Die Besonderheit dieser Befehle liegt darin, dass sie keine Parameter oder Koeffizienten zurückliefern, sondern sofort die mittels der Interpolation

berechneten Funktionswerte. Ein Beispiel soll dies verdeutlichen. Es werden wieder die Stützstellen aus dem vorherigen Beispiel genutzt.

*Beispiel 1.51:*

```
>> x2 = linspace(-pi, pi, 100);           % genauere x-Auflösung
>> y_polygon = interp1(x,y, x2);         % Interpolation durch Polygonzug
>> y_spline = interp1(x,y, x2, 'spline'); % Spline-Interpolation
>> subplot(1,2,1), plot(x,y, 'ro', x2, y_polygon)
>> subplot(1,2,2), plot(x,y, 'ro', x2, y_spline)
```

`interp1` erwartet wie zuvor zuerst die Stützstellen als Vektoren. Das dritte Argument ist der Vektor mit x-Werten, für den die Funktionswerte der Interpolation berechnet werden sollen. Ein optionaler vierter Parameter bestimmt die Form der Interpolation. Im Beispiel wurden sowohl eine Interpolation in Form eines Polygonzuges als auch eine Spline-Interpolation durchgeführt. Das Ergebnis ist in Abbildung 17 zu sehen.



*Abbildung 17: Interpolation mittels Polygonzug und Spline*

Befehl	Beschreibung	Beispiel
<code>pA = polyfit(x, y, n)</code>	Berechnet die Koeffizienten eines linearen Interpolations-/Regressionspolynoms n-ten Grades für die gegebenen Stützstellen	<pre>&gt;&gt; x = 1:5; &gt;&gt; y = x + randn(size(x)) * 0.3; &gt;&gt; pA = polyfit(x, y, 1); &gt;&gt; y2 = polyval(pA, x); &gt;&gt; plot(x, y, 'ro', x, y2)</pre>
<code>pp = spline(x, y)</code>	Spline-Interpolation anhand der Stützstellen x und y. Ergebnis ist ein spezielle Spline-Objekt	<pre>&gt;&gt; x = [0:10]; &gt;&gt; y = sin(x); &gt;&gt; xx = [0:0.1:10]; &gt;&gt; pp = spline(x, y); &gt;&gt; yy = ppval(pp, xx); &gt;&gt; plot(x, y, '+', xx, yy);</pre>
<code>y = ppval(pp, x)</code>	Funktionswert der Spline-Interpolation berechnen	
<code>interp1(x, y, xi, '..')</code>	Liefert die Funktionswerte der Interpolation an den Stellen xi.	<pre>&gt;&gt; x = [0:10]; &gt;&gt; y = sin(x); &gt;&gt; xx = [0:0.1:10]; &gt;&gt; y_spline = interp1(x, y, xx, 'spline'); &gt;&gt; y_nearest = interp1(x, y, xx, 'nearest'); &gt;&gt; plot(x, y, '+', xx, y_nearest, xx, y_spline)</pre>

*Tabelle 18: Befehle für Interpolation und Regression*

## 2 Gegenüberstellung

Im letzten Abschnitt wurden Lösungswege typischer Probleme der Systemtheorie in **Matlab** vorgestellt. Um gleiches in den beiden ausgewählten alternativen Programmsystemen **Octave** und **Scilab** zu realisieren, ist es notwendig deren syntaktische Unterschiede bzw. Eigenheiten zu untersuchen.

Dazu wurde die Form einer tabellarischen Gegenüberstellung gewählt. Gegliedert nach Themengebiet werden Befehle und Beispiellösungen gegenübergestellt. Dabei werden in den Beispielen, sofern möglich, gleiche Variablenbezeichnungen gewählt und dadurch ein schneller Vergleich ermöglicht. Sind die syntaktischen oder auch konzeptionellen Unterschiede zu stark, als dass man sie durch einen kurzen Blick auf die Beispiele erkennen könnte, folgt anschließend gegebenenfalls eine ausführlichere Erläuterung.

## 2.1 Grundlegende Arbeitsweise

### 2.1.1 Umgang mit Variablen

Matlab	Octave	Scilab
Ausführliches Anzeigen von Variablen		
whos	whos	whos
>> whos	> whos	-->whos
Kompaktes Anzeigen von Variablen		
who	who	who
>> who	> who	-->who
Variablen löschen		
clear	clear	clear
>> clear	> clear	-->clear
Variablen in Datei speichern		
save Datei	save Datei	save Datei
>> save variablen.txt	> save variablen.txt	-->save variablen.txt
Variablen aus Datei einlesen		
load Datei	load Datei	load Datei
>> load variablen.txt	> load variablen.txt	-->load variablen.txt

Tabelle 19: Vergleich Umgang mit Variablen

### 2.1.2 Spezielle Werte

Beschreibung	Matlab	Octave	Scilab
gespeichertes Ergebnis der letzten Rechnung	ans	ans	ans
Genauigkeit zwischen zwei benachbarten Realzahlen	eps	eps	%eps
die Kreiszahl $\pi$	pi	pi	%pi
imaginäre Einheit für komplexe Zahlen	i, j	i, j	%i
kleinste positive Realzahl	realmin	realmin	number_properties('tiny')
größte positive Realzahl	realmax	realmax	number_properties('huge')
Repräsentation der Unendlichkeit	Inf	Inf	%inf
keine Zahl, z.B. das Ergebnis einer Division durch 0	NaN	NaN	%nan

Tabell 20: Vergleich spezielle Werte

Eine Besonderheit von **Scilab** im Vergleich zu **Matlab** und **Octave** ist, dass es boolesche Ausdrücke in Form von %T für Wahr und %F für Unwahr unterstützt.

### 2.1.3 Desktop und Hilfe

Matlab	Octave	Scilab
Hilfetext zur angegebenen Funktion ausgeben		
help Funktion	help Funktion	help Funktion
>> help sin	> help sin	-->help sin
nach Schlüsselwort in Hilfetexten suchen		
lookfor Begriff	lookfor Begriff	apropos Begriff
>> lookfor inverse	> lookfor inverse	-->apropos inverse
Sitzung in Datei protokollieren		
diary Datei	diary Datei	diary
>> diary protokoll.txt >> diary off	> diary protokoll.txt > diary off	-->diary protokoll.txt -->diary(0)
Eingabefenster löschen		
clc	clc	clc
>> clc	> clc	-->clc
Kommentar einleiten		
% Matlab Kommentar	% Octave Kommentar	// Scilab Kommentar

*Tabelle 21: Vergleich Desktop und Hilfe*

Während in **Matlab** und **Octave** ein Prozentzeichen zur Einleitung von Kommentaren benutzt wird, findet in **Scilab** die C++-Syntax, zwei Vorwärtsschrägstriche, Anwendung.

## 2.2 Einfache Berechnungen

### 2.2.1 Komplexe Zahlen

Matlab	Octave	Scilab
<b>Realteil von z bestimmen</b>		
<code>x = real(z)</code>	<code>x = real(z)</code>	<code>x = real(z)</code>
<code>&gt;&gt; real(2+3j)</code> <code>ans = 2</code>	<code>&gt; real(2+3j)</code> <code>&gt;ans = 2</code>	<code>--&gt;real(2+3*%i)</code> <code>ans =</code>  <code>2.</code>
<b>Imaginärteil von z bestimmen</b>		
<code>y = imag(z)</code>	<code>y = imag(z)</code>	<code>y = imag(z)</code>
<code>&gt;&gt; imag(2+3j)</code> <code>ans = 3</code>	<code>&gt; imag(2+3j)</code> <code>ans = 3</code>	<code>--&gt;imag(2+3*%i)</code> <code>ans =</code>  <code>3</code>
<b>Betrag von z bestimmen</b>		
<code>a = abs(z)</code>	<code>a = abs(z)</code>	<code>a = abs(z)</code>
<code>&gt;&gt; abs(3+4j)</code> <code>ans = 5</code>	<code>&gt; abs(3+4j)</code> <code>ans = 5</code>	<code>--&gt;abs(3+4*%i)</code> <code>ans =</code>  <code>5.</code>
<b>Phase von z bestimmen</b>		
<code>phi = angle(z)</code>	<code>phi = angle(z)</code>	<code>phi =</code> <code>atan(imag(z), real(z))</code>
<code>&gt;&gt; angle(1-j)</code> <code>ans = -0.78540</code>	<code>&gt; angle(1-j)</code> <code>ans = -0.78540</code>	<code>--&gt;z = 1-%i;</code>  <code>--&gt;atan(imag(z), real(z))</code> <code>ans =</code>  <code>- 0.7853982</code>

*Tabelle 22: Vergleich komplexe Zahlen*

## 2.2.2 Weitere mathematische Funktionen

Matlab	Octave	Scilab
<b>Exponentialfunktion</b>		
exp	exp	exp
>> exp(2) ans = 7.3891	> exp(2) ans = 7.3891	-->exp(2) ans = 7.3890561
<b>natürlicher Logarithmus</b>		
log	log	log
>> log(10) ans = 2.3026	> log(10) ans = 2.3026	-->log(10) ans = 2.3025851
<b>Quadratwurzel</b>		
sqrt	sqrt	sqrt
>> sqrt(25) ans = 5	> sqrt(25) ans = 5	-->sqrt(25) ans = 5.
<b>Ganzzahlanteil</b>		
fix	fix	fix
>> fix(7.389) ans = 7	> fix(7.389) ans = 7	-->fix(7.389) ans = 7.
<b>nächstkleinere Ganzzahl</b>		
floor	floor	floor
>> floor(7.389) ans = 7	> floor(7.389) ans = 7	-->floor(7.389) ans = 7.
<b>nächstgrößere Ganzzahl</b>		
ceil	ceil	ceil
>> ceil(7.389) ans = 8	> ceil(7.389) ans = 8	-->ceil(7.389) ans = 8.
<b>mathematisches Runden</b>		
round	round	round
>> round(7.5) ans = 8	> round(7.5) ans = 8	-->round(7.5) ans = 8.
<b>Modulodivision</b>		
mod	mod	modulo
>> mod(7,3) ans = 1	> mod(7,3) ans =	-->modulo(7,3) ans = 1.

Rest einer Ganzzahldivision		
rem	rem	pmodulo
>> rem(12,9) ans = 3	> rem(12,9) ans = 3	-->pmodulo(12,9) ans =  3.
Signumfunktion		
sign	sign	sign
>> sign(-7) ans = -1	> sign(-7) ans = -1	-->sign(-7) ans =  - 1.

## 2.3 Vektoren und Matrizen

Matlab	Octave	Scilab
Zeilenvektor erzeugen		
$x = [ \dots \dots ]$	$x = [ \dots \dots ]$	$x = [ \dots \dots ]$
>> x = [ 1 2 3 ] x =  1 2 3	> x = [ 1 2 3 ] x =  1 2 3	-->x = [ 1 2 3 ] x =  1. 2. 3.
Spaltenvektor erzeugen		
$x = [ \dots ; \dots ]$	$x = [ \dots ; \dots ]$	$x = [ \dots ; \dots ]$
>> x = [ 1; 2 ] x =  1 2	> x = [ 1; 2 ] x =  1 2	-->x = [ 1; 2 ] x =  1. 2.
Matrix erzeugen		
$A = [ ..; ...; \dots ]$	$A = [ ..; ...; \dots ]$	$A = [ ..; ...; \dots ]$
>> A = [ 1 2; 3 4 ] A =  1 2 3 4	> A = [ 1 2; 3 4 ] A =  1 2 3 4	-->A = [ 1 2; 3 4 ] A =  1. 2. 3. 4.
Zugriff auf Vektorelement		
$a = x(\dots)$	$a = x(\dots)$	$a = x(\dots)$
>> a = x(2) a = 2	> a = x(2) a = 2	-->a = x(2) a =  2.
Spaltenvektor im Bereich von a bis b mit n Elementen		
$y = \text{linspace}(a, b, n)$	$y = \text{linspace}(a, b, n)$	$y = \text{linspace}(a, b, n)$
>> y = linspace(1, 2, 3) y =  1.0000 1.5000 2.0000	> y = linspace(1, 2, 3) y =  1.0000 1.5000 2.0000	-->y = linspace(1, 2, 3) y =  1. 1.5 2.

Spaltenvektor im Bereich von $10^a$ bis $10^b$ mit n logarithmisch geteilten Elementen		
<code>y = logspace(a, b, n)</code>	<code>y = logspace(a, b, n)</code>	<code>y = logspace(a, b, n)</code>
<pre>&gt;&gt; y = logspace(1, 2, 4) y =     10.0000    21.5443    46.4159   100.0000</pre>	<pre>&gt;&gt; y = logspace(1,2,4) y =     10.0000    21.544    46.416   100.00</pre>	<pre>--&gt;y = logspace(1,2,4) y =     10.    21.544347   46.415888    100.</pre>
Matrix transponieren		
<code>B = A'</code>	<code>B = A'</code>	<code>B = A'</code>
<pre>&gt;&gt; B = A' B =      1     3      2     4</pre>	<pre>&gt; B = A' B =      1     3      2     4</pre>	<pre>--&gt;B = A' B =      1.     3.      2.     4.</pre>
Inverse zur Matrix A berechnen		
<code>B = inv(A)</code>	<code>B = inv(A)</code>	<code>B = inv(A)</code>
<pre>&gt;&gt; B = inv(A) B =    -2.00000    1.00000     1.50000   -0.50000</pre>	<pre>&gt; B = inv(A) B =    -2.00000    1.00000     1.50000   -0.50000</pre>	<pre>--&gt;B = inv(A) B =    - 2.     1.     1.5   - 0.5</pre>
Pseudoinverse berechnen		
<code>B = pinv(A)</code>	<code>B = pinv(A)</code>	<code>B = pinv(A)</code>
<pre>&gt;&gt; C = [ 1 2 ; 3 4 ; 5 6 ]; &gt;&gt; D = pinv(C) D =    -1.33333   -0.33333    0.66667     1.08333    0.33333   -0.41667</pre>	<pre>&gt; C = [ 1 2; 3 4; 5 6]; &gt; D = pinv(C) D =    -1.33333   -0.33333    0.66667     1.08333    0.33333   -0.41667</pre>	<pre>--&gt;C = [ 1 2; 3 4; 5 6 ]; --&gt;D = pinv(C) D =    - 1.3333333   - 0.3333333     0.6666667     1.0833333    0.3333333   -     0.4166667</pre>
Einheitsmatrix vom Typ n x n		
<code>A = eye(n)</code>	<code>A = eye(n)</code>	<code>A = eye(n, n)</code>
<pre>&gt;&gt; A = eye(3) A =      1     0     0      0     1     0      0     0     1</pre>	<pre>&gt; A = eye(3) A =      1     0     0      0     1     0      0     0     1</pre>	<pre>--&gt;A = eye(3, 3) A =      1.     0.     0.      0.     1.     0.      0.     0.     1.</pre>
Liefert Spaltenvektor mit Indizes der Elemente die verschieden von 0 sind		
<code>i = find(A)</code>	<code>i = find(A)</code>	<code>i = find(A)</code>
<pre>&gt;&gt; A = eye(3); &gt;&gt; find(A) ans =      1      5      9</pre>	<pre>&gt; A = eye(3); &gt; find(A) ans =      1      5      9</pre>	<pre>--&gt;A = eye(3,3); --&gt;find(A) ans =      1.     5.     9.</pre>
Liefert Dimension einer Matrix		
<code>[m, n] = size(X)</code>	<code>[m, n] = size(X)</code>	<code>[m, n] = size(X)</code>
<pre>&gt;&gt; [ m, n ] =size(A) m = 3 n = 3</pre>	<pre>&gt; [ m,n ] = size(A) m = 3 n = 3</pre>	<pre>--&gt;[m,n] = size(A) n =      3.</pre>

		m = 3.
<b>Länge eines Vektors</b>		
n = length(x)	n = length(x)	n = length(x)
>> length(A) ans = 3	> length(A) ans = 3	-->length(A) ans = 9.
<b>Index des letzten Elements innerhalb einer Matrix oder eines Vektors</b>		
end	end	\$
>> a = [1 2 3 4]; >> a(end) ans = 4	> a = [1 2 3 4]; > a(end) ans = 4	-->a = [1 2 3 4]; -->a(\$) ans = 4.

Tabelle 23: Vergleich Vektoren und Matrizen

Um eine Einheitsmatrix in **Scilab** zu erzeugen, ist es notwendig, die vollständige Dimension, also n und m anzugeben. **Matlab** und **Octave** benötigen für eine quadratische Matrix nur einen Parameter.

length ist in **Scilab** trotz gleicher Syntax anders implementiert. Während in **Matlab** und **Octave** nur die größte Dimension einer Matrix zurückgegeben wird, liefert **Scilab** die Anzahl der Elemente einer Matrix (der entsprechende Befehl in **Matlab** lautet numel).

## 2.4 Arbeiten mit Gleichungssystemen

Matlab	Octave	Scilab
<b>Direkte Matrixdivision (Lösen eines linearen Gleichungssystems)</b>		
x = A \ b	x = A \ b	x = A \ b
>> A = [ 2 1; 4 3 ]; >> b = [ 8 ; 10 ]; >> x = A \ b  x =  7 -6	> A = [ 2 1; 4 3 ]; > b = [ 8 ; 10 ]; > x = A \ b x =  7 -6	-->A = [ 2 1; 4 3 ]; -->b = [8 ; 10 ]; -->x = A \ b x =  7. - 6.
<b>Rang der Matrix A berechnen</b>		
k = rank(A)	k = rank(A)	k = rank(A)
>> rank(A) ans = 2	>> rank(A) ans = 2	-->rank(A) ans = 2.
<b>Determinante der Matrix X</b>		
d = det(X)	d = det(X)	d = det(X)
>> det(A) ans = 2	> det(A) ans = 2	-->det(A) ans = 2.

Tabelle 24: Vergleich Arbeiten mit Gleichungssystemen

## 2.5 Polynome

Matlab	Octave	Scilab
<b>Polynom angeben</b>		
<code>pA = [ ... ... ... ]</code>	<code>pA = [ ... ... ... ]</code>	<code>pA = poly([ ... ], 'x', 'coeff')</code>
<pre>&gt;&gt; pA = [ 2 3 1 ] pA =     2    3    1</pre>	<pre>&gt; pA = [ 2 3 1 ] pA =     2    3    1</pre>	<pre>--&gt;pA = poly([ 1 3 2 ], "x", "coeff") pA =           2     1 + 3x + 2x</pre>
<b>Nullstellen des Polynoms pA berechnen</b>		
<code>x = roots (pA)</code>	<code>x = roots (pA)</code>	<code>x = roots (pA)</code>
<pre>&gt;&gt; x = roots (pA) x =    -1.00000    -0.50000</pre>	<pre>&gt; x = roots (pA) x =    -1.00000    -0.50000</pre>	<pre>--&gt;roots (pA) ans =    - 0.5    - 1.</pre>
<b>Polynom aus Nullstellen bestimmen</b>		
<code>pA = poly(x)</code>	<code>pA = poly(x)</code>	<code>pA = poly(x, 'x', 'roots')</code>
<pre>&gt;&gt; pA = poly([ -1 -0.5 ]) pA =     1.00000    1.50000    0.50000</pre>	<pre>&gt; pA = poly([-1 -0.5]) pA =     1.00000    1.50000    0.50000</pre>	<pre>--&gt;pA = poly([ -0.5 -1 ], "x", "roots") pA =           2     0.5 + 1.5x + x</pre>
<b>Funktionswert des Polynoms pA n der Stelle x berechnen</b>		
<code>y = polyval (pA, x)</code>	<code>y = polyval (pA, x)</code>	<code>y = horner (pA, x)</code>
<pre>&gt;&gt; y = polyval (pA, 3) y = 14</pre>	<pre>&gt; y = polyval (pA, 3) y = 14</pre>	<pre>--&gt;y = horner (pA, 3) y =     14.</pre>
<b>Multiplikation der beiden Polynome p1 und p2</b>		
<code>p3 = conv (p1, p2)</code>	<code>p3 = conv (p1, p2)</code>	<code>p3 = p1 * p2</code>
<pre>&gt;&gt; p3 = conv([ 3 1 0 ], [ 2 1 ]) p3 =     6    5    1    0</pre>	<pre>&gt; p3 = conv([ 3 1 0 ], [ 2 1 ]) p3 =     6    5    1    0</pre>	<pre>--&gt;p1 = poly([ 0 1 3 ], "x", "coeff"); --&gt;p2 = poly([ 1 2 ], "x", "coeff"); --&gt;p3 = p1 * p2 p3 =           2    3     x + 5x + 6x</pre>

Polynomdivision von p1 durch p2		
<code>p3 = deconv(p1, p2)</code>	<code>p3 = deconv(p1, p2)</code>	<code>p3 = p1 / p2</code>
<pre>&gt;&gt; p4 = deconv([ 6 5 1 0 ], [ 2 1 ]) p4 =      3    1    0</pre>	<pre>&gt; p4 = deconv([ 6 5 1 0 ], [ 2 1 ]) p4 =      3    1    0</pre>	<pre>--&gt;p1 = poly([0 1 5 6], "x", "coeff"); --&gt;p2 = poly([1 2], "x", "coeff"); --&gt;p4 = p1 / p2 p4 =            2         x + 3x         -----              1</pre>
Ableitung des Polynoms pA bilden		
<code>p1 = polyder(pA)</code>	<code>p1 = polyder(pA)</code>	<code>p1 = derivat(pA)</code>
<pre>&gt;&gt; p1 = polyder([ 2 3 1 ]) p1 =      4    3</pre>	<pre>&gt; p1 = polyder([ 2 3 1 ]) p1 =      4    3</pre>	<pre>--&gt;pA = poly([1 3 2], "x", "coeff"); --&gt;p1 = derivat(pA) p1 =      3 + 4x</pre>
Partialbruchzerlegung		
<code>[r,p,k] = residue(b,a)</code>	<code>[k ,p] = residue(b,a)</code>	<code>elts = pfss(b/a)</code>
<pre>&gt;&gt; b = [ 5 3 -2 7 ]; &gt;&gt; a = [-4 0 8 3 ]; &gt;&gt; [r, p, k] = residue(b,a) r =    -1.4167    -0.6653     1.3320 p =     1.5737    -1.1644    -0.4093 k =    -1.2500</pre>	<pre>&gt; b = [ 5 3 -2 7 ]; &gt; a = [-4 0 8 3 ]; &gt; [r, p, k] = residue(b,a) r =    -1.41671     1.33196    -0.66525 p =     1.57372    -0.40928    -1.16444 k = -1.2500</pre>	<pre>--&gt;b = poly([7 -2 3 5], 'x', 'coeff'); --&gt;a = poly([3 8 0 -4], 'x', 'coeff'); --&gt;elts = pfss(b/a) elts =          elts(1)         - 1.4167102         -----         - 1.5737151 + s          elts(2)         1.3319607         -----         0.4092790 + s          elts(3)         - 0.6652506         -----         1.1644361 + s          elts(4)         - 1.25</pre>

Tabelle 25: Vergleich Polynome

Das in **Matlab** und **Octave** verwendete Konzept, Polynome als Zeilenvektoren ausschließlich über ihre Koeffizienten darzustellen, findet in **Scilab** keine Anwendung. Stattdessen ist dort eine Art symbolische Angabe von Polynomen vorgesehen. Die Erzeugung eines neuen Polynoms geschieht immer über die Funktion `poly`. Einfachste Form ist der Aufruf `x = poly(0, 'x')`. Damit erzeugt man sich die symbolische Variable `x`, die dann verwenden kann, um Polynome anzugeben.

*Beispiel 2.1:*

```
-->x = poly(0, 'x');  
-->p1 = x^2 + x - 1  
p1 =  
      2  
- 1 + x + x
```

Alternativ kann mit `poly` eine Erzeugung durch die Angabe von Koeffizienten (dritter Parameter gleich 'coeff') oder Nullstellen (dritter Parameter gleich 'roots') erfolgen. Bei der Angabe von Koeffizienten ist zu beachten, dass **Scilab** diese anders ordnet (beginnend mit der kleinsten Potenz).

## 2.6 Graphische Darstellungen

Matlab	Octave	Scilab
<b>2D-Plot der Vektoren y über x</b>		
<code>plot(x, y)</code>	<code>plot(x, y)</code>	<code>plot(x, y)</code>
<pre>&gt;&gt; x = [0:0.2:10]; &gt;&gt; y = sin(x); &gt;&gt; plot(x, y)</pre>	<pre>&gt; x = [0:0.2:10]; &gt; y = sin(x); &gt; plot(x, y)</pre>	<pre>--&gt;x = 0:0.2:10; --&gt;y = sin(x); --&gt;plot(x, y)</pre>
<b>nachfolgende Plot-Befehle in das selbe Koordinatensystem</b>		
<code>hold</code>	<code>hold</code>	<code>set(gca(), 'auto_clear', 'off')</code>
<pre>&gt;&gt; plot(x, y); &gt;&gt; hold on; &gt;&gt; plot(x, 2 * y)</pre>	<pre>&gt; plot(x, y); &gt; hold on; &gt; plot(x, 2 * y)</pre>	<pre>--&gt;plot(x, y); --&gt;set(gca(), "auto_clear", "off") --&gt;plot(x, 2 * y)</pre>
<b>Legende für Plots einfügen</b>		
<code>legend('s1', 's2', ...)</code>	<code>legend('s1', 's2', ...)</code>	<code>legend('s1', 's2', ...)</code>
<pre>&gt;&gt; plot(x, [y; 2*y]) &gt;&gt; legend('Amplitude = 1', 'Amplitude = 2');</pre>	<pre>&gt; plot(x, [y; 2*y]) &gt; legend('Amplitude = 1', 'Amplitude = 2');</pre>	<pre>--&gt;plot(x, [y; 2*y]); --&gt;legend("Amplitude = 1", "Amplitude = 2")</pre>
<b>Achsenbezeichnung (x, y, z) festlegen</b>		
<code>xlabel('...')</code> <code>ylabel('...')</code> <code>zlabel('...')</code>	<code>xlabel('...')</code> <code>ylabel('...')</code> <code>zlabel('...')</code>	<code>xlabel('...')</code> <code>ylabel('...')</code> <code>zlabel('...')</code>
<pre>&gt;&gt; xlabel('Zeit in Sekunden') &gt;&gt; ylabel('Weg in Metern')</pre>	<pre>&gt; xlabel('Zeit in Sekunden') &gt; ylabel('Weg in Metern')</pre>	<pre>--&gt;xlabel("Zeit in Sekunden") --&gt;ylabel("Weg in Metern")</pre>
<b>Grenzen der Achsen festlegen</b>		
<code>xlim([xmin xmax])</code> <code>ylim([ymin ymax])</code> <code>zlim([zmin zmax])</code>	<code>xlim([xmin xmax])</code> <code>ylim([ymin ymax])</code> <code>zlim([zmin zmax])</code>	<code>a = gca();</code> <code>a.data_bounds=[xmin</code> <code>xmax; ymin ymax; zmin</code> <code>zmax];</code>
<pre>&gt;&gt; xlim([ 0 5 ]);</pre>	<pre>&gt; xlim([ 0 5 ]);</pre>	<pre>--&gt;a = gca(); --&gt;a.data_bounds = [0 -1; 5 1]</pre>
<b>Achseneinteilung festlegen</b>		
<code>axis</code>	<code>axis</code>	<code>mtlb_axis</code>
<pre>&gt;&gt; axis square &gt;&gt; axis auto</pre>	<pre>&gt; axis square &gt; axis auto</pre>	<pre>--&gt;mtlb_axis square; --&gt;mtlb_axis auto;</pre>
<b>Titel des Plots setzen</b>		
<code>title Titel</code>	<code>title Titel</code>	<code>xtitle Titel</code>
<pre>&gt;&gt; plot(sin(0:0.2:10)) &gt;&gt; title('Sinusfunktion')</pre>	<pre>&gt; plot(sin(0:0.2:10)) &gt; title('Sinusfunktion')</pre>	<pre>--&gt;plot(sin(0:0.2:10)): --&gt;xtitle "Sinusfunktion"</pre>
<b>Leeres Koordinatensystem erzeugen</b>		
<code>figure</code>	<code>figure</code>	<code>xinit</code>
<pre>&gt;&gt; figure</pre>	<pre>&gt; figure</pre>	<pre>--&gt;xinit</pre>

Koordinatensystem löschen		
<code>clf</code>	<code>clf</code>	<code>xclear</code>
<code>&gt;&gt; clf</code>	<code>&gt; clf</code>	<code>--&gt;xclear</code>
Koordinatensystem schließen		
<code>close</code>	<code>close</code>	<code>xdel</code>
<code>&gt;&gt; close all</code>	<code>&gt; close all</code>	<code>--&gt;xdel</code>
Gitternetzlinien einschalten		
<code>grid</code>	<code>grid</code>	<code>xgrid(mode)</code>
<code>&gt;&gt; grid on</code> <code>&gt;&gt; grid off</code>	<code>&gt; grid on</code> <code>&gt; grid off</code>	<code>--&gt;xgrid(1)</code> <code>--&gt;xgrid(-1)</code> <code>--&gt;mtlb_grid on</code> <code>--&gt;mtlb_grid off</code>
beliebigen Text einfügen		
<code>text(x,y, 'string')</code>	<code>text(x,y, 'string')</code>	<code>xstring(x,y, 'string')</code>
<code>&gt;&gt; t = 0:0.1:20;</code> <code>&gt;&gt; plot(t, sin(t));</code> <code>&gt;&gt; text(pi/2,1 'erstes Maximum')</code>	<code>&gt; t = 0:0.1:20;</code> <code>&gt; plot(t, sin(t));</code> <code>&gt; text(pi/2,1 'erstes Maximum')</code>	<code>--&gt;t=0:0.1:20;</code> <code>--&gt;plot(t, sin(t))</code> <code>--&gt;xstring(%pi/2, 1, "erstes Maximum")</code>

*Tabelle 26: Vergleich Graphische Darstellung Grundlagen*

Matlab	Octave	Scilab
<b>doppelt logarithmischer Plot</b>		
loglog(x, y)	loglog(x, y)	a = gca(); a.log_flags="ll"
>> x = logspace(1, 2); >> loglog(x, exp(x));	> x = logspace(1, 2); > loglog(x, exp(x));	-->x = logspace(1, 2); -->plot(x, exp(x)); -->a = gca(); a.log_flags="ll"; -->mtlb_loglog(x, exp(x))
<b>einfach logarithmischer Plot</b>		
semilogx(x, y) semilogy(x, y)	semilogx(x, y) semilogy(x, y)	a = gca(); a.log_flags="nl"
>> x = 0:0.1:10; >> semilogy(x, 10.^x);	> x = 0:0.1:10; > semilogy(x, 10.^x);	-->x = 0:0.1:10; -->plot(x, 10.^x); -->a = gca(); a.log_flags="nl"; -->mtlb_semilogy(x, 10.^x)
<b>3D-Plot (parametrisch)</b>		
plot3(x, y, z)	plot3(x, y, z)	param3d(x, y, z)
>> t = 0:pi/50:10*pi; >> plot3(sin(t), cos(t), t) >> axis square	> t = 0:pi/50:10*pi; > plot3(sin(t), cos(t), t) > axis square	-->t = 0:%pi/50:%pi*10; -->param3d(sin(t), cos(t), t)
<b>mehrere Graphen in ein Koordinatensystem</b>		
subplot(r, c, i)	subplot(r, c, i)	subplot(r, c, i)
>> x= 0:0.2:10; >> subplot(1,2,1), plot(x, sin(x)) >> subplot(1,2,2), plot(x, cos(x))	> x= 0:0.2:10; > subplot(1,2,1), plot(x, sin(x)) > subplot(1,2,2), plot(x, cos(x))	-->x = 0:0.2:10; -->subplot(1,2,1); -->plot(x, sin(x)); -->subplot(1,2,2); -->plot(x, cos(x));
<b>3D-Oberflächen-Gitternetzlinie-Plot</b>		
surf(x, y, Z) mesh(x, y, Z)	surf(x, y, Z) mesh(x, y, Z)	surf(x, y, Z) mesh(x, y, Z)
>> x = 0:100; >> y = x; >> Z = x' * y; >> mesh(x, y, Z) >> surf(x, y, Z)	> x = 0:100; > y = x; > Z = x' * y; > mesh(x, y, Z) > surf(x, y, Z)	-->x = 0:100; -->y = x; -->Z = x' * y; -->mesh(x, y, Z) -->surf(x, y, Z)
<b>Plot in Polarkoordinaten</b>		
polar(theta, rho)	polar(theta, rho)	polarplot(theta, rho)
>> t = 0:.01:2*pi; >> polar(t, sin(2*t).*cos(2*t))	> t = 0:.01:2*pi; > polar(t, sin(2*t).*cos(2*t))	-->t = 0:0.01:2*pi -- >polarplot(t, sin(2*t).*cos(2*t))

Tabelle 27: Vergleich Graphische Darstellung weitere Plotbefehle

Die graphische Darstellung ist in **Scilab**, genau wie in **Matlab**, intern über sogenannte Handles realisiert. Diese Handles haben jeweils spezielle Eigenschaften, die sich auslesen und setzen lassen. Die Ausgabe eines doppelt logarithmischen Plots geschieht also wie gewohnt durch den Befehl `plot` und das anschließende Anpassen der Achseneinteilung.

## 2.7 Programmierung

Matlab	Octave	Scilab
<b>Zählerschleife</b>		
for ... end	for ... end	for ... end
for i=1:10 disp(i) end	for i=1:10 disp(i) end	for i=1:10 disp(i) end
<b>Bedingungsschleife</b>		
while ... end	while ... end	while ... end
i = 3; while i > 0 disp(i) i = i -1; end	i = 3; while i > 0 disp(i) i = i -1; end	while i > 0 disp(i) i = i -1; end
<b>Formatierte Ausgabe</b>		
fprintf('...', ...)	fprintf('...', ...)	printf('...', ...)
i = 2; fprintf('Inhalt von i = %d\n', i); Inhatlv von i = 2	i = 2; fprintf('Inhalt von i = %d\n', i); Inhalt von i = 2	i = 2; printf('Inhalt von i = %d\n', i) Inhalt von i = 2
<b>Bedingte Anweisung</b>		
if ...elseif ... else	if ... elseif ... else	if ... then ... else
x = -3; if x < 0 s = -1 elseif x > 0 s = 1 else s = 0 end	x = -3; if x < 0 s = -1 elseif x > 0 s = 1 else s = 0 end	x = -3; if x < 0 then s = -1; elseif x > 0 then s = 1; else s = 0; end
<b>Fallunterscheidung</b>		
switch ... case ... end	switch ... case ... end	select ... case .. end
>> a = 1; >> switch a case 0 disp('a ist null'); case 1 disp('a ist eins'); otherwise disp('a ist keine Binärzahl'); end a ist eins	> a = 1; > switch a > case 0 >  disp('a ist null'); > case 1 >  disp('a ist eins'); > otherwise >  disp('a ist keine Binärzahl'); > end a ist eins	-->a = 1; -->select a case 0 disp('a ist null'); case 1 disp('a ist eins'); else disp('a ist keine Binaerzahl'); end a ist eins
<b>Funktionsdefinition</b>		
function [ret]=name (para )	function [ret]=name (para meter)	function [ret]=name (param)     ... endfunction

Anzahl der übergebenen Parameter innerhalb einer Funktion		
nargin	nargin	argn(2)
Anzahl der gewünschten Rückgabewerte		
nargout	nargout	argn(1)
Abarbeitung der Funktion beenden		
return	return	return
Warnung ausgeben		
warning('...')	warning('...')	warning('...')
Warnung ausgeben und Abarbeitung beenden		
error('...')	error('...')	error('...')
<pre>function[a,b]=beispielfunktion(c,d) if nargin ~= 2     error('zu wenig Argumente') end  if nargout &lt; 2     warning('nur ein Rückgabewerte angefordert')     return end  a = c; b =d;</pre>	<pre>function[a,b]=beispielfunktion(c,d) if nargin ~= 2     error('zu wenig Argumente') end  if nargout &lt; 2     warning('nur ein Rückgabewerte angefordert')     return end  a = c; b =d;</pre>	<pre>function [a,b]=beispielfunktion(c,d)     if argn(2) ~= 2 then         error("zu wenig Argumente");     end      if argn(1) &lt; 2 then         warning("nur ein Rückgabewert angefordert");     end     return; end  a = c; b = d; endfunction</pre>

*Tabelle 28: Vergleich Programmierung*

Wie **Matlab** und **Octave** unterscheidet **Scilab** auch in Script- und Function-Files. Allerdings ist es nicht zwingend notwendig, dass der Dateiname der darin definierten Funktion entspricht. Es können sogar mehrere Funktionen in einer Datei implementiert werden. Während **Matlab** und **Octave** durchgängig die Endung `.m` verwenden, hat es sich in **Scilab** eingebürgert, Script-Files mit `.sce` und Function-Files mit `.sci` zu kennzeichnen. Um ein Script-File in der laufenden Sitzung auszuführen, ist der Aufruf von `exec('dateiname')` notwendig. Function-Files und die darin enthaltenen Funktionen müssen hingegen mit `getf('dateiname')` bekannt gemacht werden. Anschließend können sie wie eingebaute Funktionen verwendet werden.

## 2.8 Systemtheorie

### 2.8.1 Systemmodelle

Matlab	Octave	Scilab
<b>Kontinuierliche Übertragungsfunktion erzeugen</b>		
<code>sys = tf([num], [den])</code>	<code>sys = tf([num], [den])</code>	<code>sys = syslin('c', num, den)</code>
<pre>&gt;&gt; sys1 = tf([1 0.5], [2 3])  Transfer function: s + 0.5 ----- 2 s + 3</pre>	<pre>&gt; sys1 = tf([1 0.5], [2 3]); &gt; sysout(sys1) Input(s)     1: u_1  Output(s):     1: y_1  transfer function form: 1*s^1 + 0.5 ----- 2*s^1 + 3</pre>	<pre>--&gt;num = poly([0.5 1], 's', 'c'); --&gt;den = poly([3 2], 's', 'c'); --&gt;sys1 = syslin('c', num, den) sys1 =     0.5 + s -----     3 + 2s</pre>
<b>Differentialgleichung numerisch lösen</b>		
<code>[t,y] = ode23(func, [t0 te], y0)</code>	<code>y = lsode(func, y0, t)</code>	<code>y = ode(y0,t0,t,func)</code>
<pre>&gt;&gt; pt1dgl=inline('(1/2)*y+(5/2)*1','t','y'); &gt;&gt; t0 = 0; &gt;&gt; te = 15; &gt;&gt; y0 = 0; &gt;&gt; [t,y] = ode23(pt1dgl,[t0 te],y0);</pre>	<pre>&gt; pt1dgl=inline('(1/2)*y+(5/2)*1','t','y'); &gt; t = 0:0.1:15; &gt; y0 = 0; &gt; y = lsode(pt1dgl, y0, t);</pre>	<pre>--&gt;deff('dy=pt1dgl(t,y)','dy=(1/2)*y+(5/2)*1'); --&gt;y0 = 0; --&gt;t0 = 0; --&gt;t = 0:0.1:15; --&gt;y = ode(y0,t0,t,pt1dgl);</pre>
<b>Diskrete Übertragungsfunktion (mittels z-Transformation) erzeugen</b>		
<code>sys = filt([num], [den])</code>	<code>sys = tf([num], [den], T)</code>	<code>sys = syslin('d', num, den)</code>
<pre>&gt;&gt; sys_d = filt([1 0.5], [2 3])  Transfer function: 1 + 0.5 z^-1 ----- 2 + 3 z^-1</pre>	<pre>&gt; sys_d = tf([1 0.5], [2 3], 0.1); &gt; sysout(sys_d) Input(s)     1: u_1  Output(s):     1: y_1 (discrete)  Sampling interval: 0.1 transfer function form: 1*z^1 + 0.5 ----- 2*z^1 + 3</pre>	<pre>--&gt;num = poly([0.5 1], 's', 'c'); --&gt;den = poly([3 2], 's', 'c'); --&gt;sys_d = syslin('d', num, den) sys_d =     0.5 + z -----     3 + 2z</pre>

Systemmodell in Zustandsraumdarstellung		
<code>sys = ss(A, B, C, D)</code>	<code>sys = ss(A, B, C, D)</code>	<code>sys = syslin('c', A, B, C, D)</code>
<pre>&gt;&gt; A = [ 0 -0.4; 1 -1.3]; &gt;&gt; B = [ 0.8; 1 ]; &gt;&gt; C = [ 0 1 ]; &gt;&gt; D = [ 0 ]; &gt;&gt; sys_ss = ss(A,B,C,D);</pre>	<pre>&gt; A = [ 0 -0.4; 1 -1.3]; &gt; B = [ 0.8; 1 ]; &gt; C = [ 0 1 ]; &gt; D = [ 0 ]; &gt; sys_ss = ss(A,B,C,D);</pre>	<pre>--&gt;A = [ 0 -0.4; 1 -1.3]; --&gt;B = [ 0.8; 1 ]; --&gt;C = [ 0 1 ]; --&gt;D = [ 0 ]; --&gt;sys_ss = syslin('c', A,B,C,D);</pre>
Übertragungsfunktion in Pol-Nullstellen Darstellung		
<code>sys = zpk([p], [n], k)</code>	<code>sys = zp([p], [n], k)</code>	<code>sys = syslin('c', p_ns, p_pole) * k</code>
<pre>&gt;&gt; sys3 = zpk([-0.5], [-1.5], 2)  Zero/pole/gain: 2 (s+0.5) ----- (s+1.5)</pre>	<pre>&gt; sys3 = zp([-0.5], [-1.5], 2); &gt; sysout(sys3) Input(s)     1: u_1  Output(s):     1: y_1  zero-pole form: 2 (s + 0.5) ----- (s + 1.5)</pre>	<pre>--&gt;p_ns = poly([-0.5], 's', 'r'); --&gt;p_pole = poly([-1.5], 's'); --&gt;sys3 = syslin('c', p_ns, p_pole) * 2 sys3 =      1 + 2s -----     1.5 + s</pre>

Tabelle 29: Vergleich Systemmodelle

## 2.8.2 Untersuchung von Systemeigenschaften

Matlab	Octave	Scilab
<b>Pole des Systems ermitteln</b>		
<code>p = pole(sys)</code>	nicht vorhanden	<code>p = roots(denom(sys))</code>
<pre>&gt;&gt; pole(sys1) ans = -1.5000</pre>		<pre>--&gt;p_poly=denom(sys1) p_poly = 3 + 2s --&gt;roots(p_poly) ans = - 1.5</pre>
<b>Nullstellen des Systems ermitteln</b>		
<code>z = zero(sys)</code>	<code>z = tzero(sys)</code>	<code>z = roots( numer(sys) )</code>
<pre>&gt;&gt; zero(sys1) ans = -0.5000</pre>	<pre>&gt; tzero(sys1) ans = -0.50000</pre>	<pre>--&gt;ns_poly = numer(sys1) ns_poly = 0.5 + s --&gt;roots(ns_poly) ans = - 0.5</pre>
<b>Ordnung des Systems ermitteln</b>		
<code>ns = order(sys)</code>	nicht vorhanden	nicht vorhanden
<pre>&gt;&gt; order(sys1) ans = 1</pre>		
<b>Impulsantwort simulieren</b>		
<code>impulse(sys)</code>	<code>impulse(sys)</code>	<code>y = csim('impulse', t, sys)</code>
<pre>&gt;&gt; impulse(sys1)</pre>	<pre>&gt; impulse(sys1)</pre>	<pre>--&gt;t=0:0.1:4; --&gt;y = csim('impulse', t, sys1); --&gt;plot(t, y)</pre>
<b>Sprungantwort simulieren</b>		
<code>step(sys)</code>	<code>step(sys)</code>	<code>y = csim('step', t, sys)</code>
<pre>&gt;&gt; step(sys1)</pre>	<pre>&gt; step(sys1)</pre>	<pre>--&gt;t=0:0.1:4; --&gt;y = csim('step', t, sys1); --&gt;plot(t, y)</pre>
<b>Simulation beliebiges Eingangssignal</b>		
<code>xa = lsim(sys, xe, t)</code>	<code>xa = lsim(sys, xe, t)</code>	<code>xa = csim(f, t, sys)</code>
<pre>&gt;&gt; sys = tf([2], [4 0.8 1]); &gt;&gt; t = (0:0.4:20)'; &gt;&gt; xe = sin(t); &gt;&gt; xa = lsim(sys, xe, t); &gt;&gt; plot(t, [xe xa]);</pre>	<pre>&gt; sys = tf([2], [4 0.8 1]); &gt; t = (0:0.4:20)'; &gt; xe = sin(t); &gt; xa = lsim(sys, xe, t); &gt; plot(t, [xe xa])</pre>	<pre>--&gt;p_poly = poly([1 0.8 4], 's', 'c'); --&gt;ns_poly = poly([2], 's', 'c'); --&gt;sys = syslin('c', ns_poly, p_poly); --&gt;t = 0:0.4:20; --&gt;deff('u=fsin(t)', 'u=sin(t)'); --&gt;xa = csim(fsin, t, sys); --&gt;plot(t, [sin(t); xa])</pre>

Bodediagram (Frequenz- und Phasengang)		
bode (sys)	bode (sys)	bode (sys)
>> bode (sys1)	> bode (sys1)	-->bode (sys1)
Ortskurve (Nyquist-Diagramm)		
nyquist (sys)	nyquist (sys)	nyquist (sys)
>> nyquist (sys1)	> nyquist (sys1)	-->nyquist (sys1)
Pol-Nullstellen-Plan		
pzmap (sys)	pzmap (sys)	plzr (sys)
>> pzmap (sys1)	> pzmap (sys1)	-->plzr (sys1)
Wurzelorskurve		
rlocus (sys)	rlocus (sys)	evans (sys)
>> rlocus (sys1)	> rlocus (sys1)	-->evans (sys1)
Phasen- und Amplitudenreserve		
[gm pm] = margin (sys)	nicht vorhanden	gm = g_margin (sys) pm = p_margin (sys)
>> sys = tf(1, [1 6 11 6]); >> [gm pm] = margin(G) gm = 60.0000 pm = Inf		-->sys = syslin('c', 1, %s^3 + 6*%s^2 + 11*%s + 6); -->gm = g_margin(sys) gm = 35.563025 -->pm = p_margin(sys) pm = []
Dämpfung bestimmen		
[w0, d] = damp (sys)	damp (sys)	nicht vorhanden
>> % PT2 mit Dämpfung 0.2 >> sys = tf( [ 2 ], [4 0.8 1] ); >> [w0,d] = damp(sys) w0 = 0.5000 0.5000 d = 0.2000 0.2000	> sys = tf( [ 2 ], [4 0.8 1] ); ..... Eigenvalue ..... Damping Frequency -----[re]----- [im]----- [abs]-----[Hz] -0.100000 0.489898 0.500000 0.200000 0.079577 -0.100000 -0.489898 0.500000 0.200000 0.079577	

Tabelle 30: Vergleich Untersuchung von Systemeigenschaften

## 2.8.3 Regelkreisstrukturen

Matlab	Octave	Scilab
<b>Reihenschaltung zweier Übertragungsfunktionen</b>		
<pre>sys3 = series(sys1, sys2)</pre>	<pre>sys3 = sysmult(sys1,sys2)</pre>	<pre>sys3 = sys1 * sys2</pre>
<pre>&gt;&gt; series(tf([1],[2 3]), tf([0.5],[4])) Transfer function:   0.5 -----  8 s + 12</pre>	<pre>&gt; sys = sysmult(tf([1],[2 3]),tf([0.5],[4])); &gt; sysout(sys, 'tf') Input(s)   1: u_1 Output(s):   1: y_1 transfer function form: 0.0625 ----- 1*s^1 + 1.5</pre>	<pre>--&gt;s = poly(0, 's'); --&gt;sys1 = syslin('c', 1, 2*s + 3); --&gt;sys2 = syslin('c', 0.5, 4); --&gt;sys1*sys2 ans =   0.5 -----  12 + 8s</pre>
<b>Parallelschaltung zweier Übertragungsfunktionen</b>		
<pre>sys3 = parallel(sys1, sys2)</pre>	<pre>sys3 = sysadd(sys1, sys2)</pre>	<pre>sys3 = sys1 + sys2</pre>
<pre>&gt;&gt; parallel(tf([1],[2 3]), tf([0.5],[4])) Transfer function: s + 5.5 -----  8 s + 12</pre>	<pre>&gt; sys = sysadd(tf([1],[2 3]),tf([0.5],[4])); &gt; sysout(sys, 'tf') Input(s)   1: u_1 Output(s):   1: y_1 transfer function form: 0.125*s^1 + 0.6875 ----- 1*s^1 + 1.5</pre>	<pre>--&gt;s = poly(0, 's'); --&gt;sys1 = syslin('c', 1, 2*s + 3); --&gt;sys2 = syslin('c', 0.5, 4); --&gt;sys1 + sys2 ans =   5.5 + s -----  12 + 8s</pre>
<b>Regelkreis schließen, Rückkopplung</b>		
<pre>sys3 = feedback(sys1, sys2, sign)</pre>	<pre>sys3 = feedback(sys1, sys2)</pre>	<pre>sys3 = sys1 /. sys2</pre>
<pre>&gt;&gt; feedback(tf([1],[3 2]), 1) Transfer function:   1 -----  3 s + 3</pre>	<pre>&gt; sys = feedback(tf([1],[3 2]), tf(1, 1)); &gt; sysout(sys, 'tf') Input(s)   1: u_1* Output(s):   1: y_1 transfer function form: 0.33333 ----- 1*s^1 + 1</pre>	<pre>--&gt;s = poly(0, 's'); --&gt;sys1 = syslin('c', 1, 3*s + 2); --&gt;sys1 /. 1 ans =   1 -----  3 + 3s</pre>

Tabelle 31: Vergleich Regelkreisstrukturen

## 2.8.4 Signalgenerierung

Matlab	Octave	Scilab
Sinus für Argument berechnen		
<code>y = sin(x)</code>	<code>y = sin(x)</code>	<code>y = sin(x)</code>
<code>&gt;&gt; y = sin( [ 1 2 3 4 ] )</code>	<code>&gt; y = sin( [ 1 2 3 4 ] )</code>	<code>--&gt;y = sin( [ 1 2 3 4 ] )</code>
Matrix der Dimension n x m mit positiven Zufallszahlen erzeugen		
<code>R = rand(n, m)</code>	<code>R = rand(n, m)</code>	<code>R = rand(n,m)</code>
<code>&gt;&gt; rand(1,2)</code> ans = 0.83187 0.57217r	<code>&gt; rand(1,2)</code> ans = 0.89196 0.33110	<code>--&gt;rand(1,2)</code> ans = 0.2113249 0.7560439
Matrix der Dimension n x m mit normalverteilten Zufallszahlen erzeugen		
<code>R = randn(n, m)</code>	<code>R = randn(n, m)</code>	<code>R = rand(n,m, 'normal')</code>
<code>&gt;&gt; randn(1,2)</code> ans = -0.82250 0.49130	<code>&gt; randn(1,2)</code> ans = -0.26993 0.66507	<code>--&gt;rand(1,2,'normal')</code> ans = 0.6380837 - 0.8498895
Signale mit Störung beaufschlagen		
<code>y = sin(x) + randn(size(x)) * v</code>	<code>y = sin(x) + randn(size(x)) * v</code>	<code>y = sin(x) + rand(x,'normal') *v</code>
<code>&gt;&gt; x = 1:5;</code> <code>&gt;&gt; y = sin(x);</code> <code>&gt;&gt; e = randn(size(x)) * 0.1;</code> <code>&gt;&gt; ye = y + e</code> ye = 0.7982 0.7427 0.1537 -0.7280 -1.0736	<code>&gt; x = 1:5;</code> <code>&gt; y = sin(x);</code> <code>&gt; e = randn(size(x)) * 0.1;</code> <code>&gt; ye = y + e</code> ye = 0.850749 0.831497 -0.060257 -0.729970 -0.880171	<code>--&gt;x = 1:5;</code> <code>--&gt;y = sin(x);</code> <code>--&gt;e = rand(x,'normal') * 0.1;</code> <code>--&gt;ye = y + e</code> ye = 0.6693162 0.9694393 0.0387797 - 0.6889345 - 0.7823647

Tabelle 32: Vergleich Signalgenerierung

Um ein Signal mit einer Störung zu beaufschlagen addiert man im einfachsten Fall einen Zufallsvektor mit gleichen Dimensionen zum ursprünglichen ungestörten Signal hinzu. Während **Matlab** und **Octave** als Dimension für `rand` bzw. `randn` auch den Rückgabewert von `size` akzeptieren funktioniert dies in **Scilab** nicht. Stattdessen genügt es, `rand` einfach nur einen Vektor zu übergeben, **Scilab** ermittelt die Dimension dann implizit.

## 2.8.5 Statistik

Matlab	Octave	Scilab
<b>Minimum bestimmen</b>		
<code>c = min(x)</code>	<code>c = min(x)</code>	<code>c = min(x)</code>
<code>&gt;&gt; min([ 3 2 10 ])</code> ans = 2	<code>&gt; min([ 3 2 10 ])</code> ans = 2	<code>--&gt;min([ 3 2 10 ])</code> ans = 2.
<b>Maximum bestimmen</b>		
<code>c = max(x)</code>	<code>c = max(x)</code>	<code>c = max(x)</code>
<code>&gt;&gt; max([ 3 2 10 ])</code> ans = 10	<code>&gt; max([ 3 2 10 ])</code> ans = 10	<code>--&gt;max([ 2 3 10 ])</code> ans = 10.
<b>Mittelwert bestimmen</b>		
<code>m = mean(x)</code>	<code>m = mean(x)</code>	<code>m = mean(x)</code>
<code>&gt;&gt; mean([ 3 2 10 ])</code> ans = 5	<code>&gt; mean([ 3 2 10 ])</code> ans = 5	<code>--&gt;mean([ 3 2 10 ])</code> ans = 5.
<b>Median bestimmen</b>		
<code>m = median(x)</code>	<code>m = median(x)</code>	<code>m = median(x)</code>
<code>&gt;&gt; r = randn(1, 10000);</code> <code>&gt;&gt; median(r)</code> ans = 0.0096	<code>&gt; r = randn(1, 10000);</code> <code>&gt; median(r)</code> ans = 0.026994	<code>--&gt;r = rand(1, 10000, 'normal');</code> <code>--&gt;median(r)</code> ans = - 0.0021775
<b>Standardabweichung bestimmen</b>		
<code>s = std(x)</code>	<code>s = std(x)</code>	<code>s = st_deviation(x)</code>
<code>&gt;&gt; std(randn(1, 10000))</code> ans = 1.0052	<code>&gt; std(randn(1, 10000))</code> ans = 0.99411	<code>--&gt;r = rand(1, 10000, 'normal');</code> <code>--&gt;st_deviation(r)</code> ans = 1.000718
<b>Varianz bestimmen</b>		
<code>v =var(x)</code>	<code>v =var(x)</code>	<code>v = variance(x)</code>
<code>&gt;&gt; var(randn(1, 10000))</code> ans = 1.0120	<code>&gt; var(randn(1, 10000))</code> ans = 1.0149	<code>--&gt;r = rand(1, 10000, 'normal');</code> <code>--&gt;variance(r)</code> ans = 1.0086744
<b>Vektor sortieren</b>		
<code>xs = sort(x)</code>	<code>xs = sort(x)</code>	<code>xs = qsort(x)</code>
<code>&gt;&gt; sort([ 3 2 10])</code> ans = 2 3 10	<code>&gt; sort([ 3 2 10 ])</code> ans = 2 3 10	<code>--&gt;qsort([ 3 2 10 ], 'g', 'i')</code> ans = 2. 3. 10.

Vektorelemente aufsummieren		
<code>s = sum(x)</code>	<code>s = sum(x)</code>	<code>s = sum(x)</code>
<code>&gt;&gt; sum([ 3 2 10 ])</code> <code>ans = 15</code>	<code>&gt; sum([ 3 2 10])</code> <code>ans = 15</code>	<code>--&gt;sum([ 3 2 10 ])</code> <code>ans =</code> <code>15.</code>
Vektorelemente multiplizieren		
<code>p = prod(x)</code>	<code>p = prod(x)</code>	<code>p = prod(x)</code>
<code>&gt;&gt; prod([ 3 2 10 ])</code> <code>ans = 60</code>	<code>&gt; prod([ 3 2 10 ])</code> <code>ans = 60</code>	<code>--&gt;prod([ 3 2 10 ])</code> <code>ans =</code> <code>60.</code>
Kumulatives Produkt des Vektors bilden		
<code>[p] = cumprod(x)</code>	<code>[p] = cumprod(x)</code>	<code>[p] = cumprod(x)</code>
<code>&gt;&gt; cumprod([ 1 2 3 4])</code> <code>ans =</code> <code>1 2 6 24</code>	<code>&gt; cumprod([ 1 2 3 4])</code> <code>ans =</code> <code>1 2 6 24</code>	<code>--&gt;cumprod([1 2 3 4])</code> <code>ans =</code> <code>1. 2. 6. 24.</code>
Kumulative Summe berechnen		
<code>[s] = cumsum(x)</code>	<code>[s] = cumsum(x)</code>	<code>[s] = cumsum(x)</code>
<code>&gt;&gt; cumsum([ 1 2 3 4])</code> <code>ans =</code> <code>1 3 6 10</code>	<code>&gt; cumsum([ 1 2 3 4])</code> <code>ans =</code> <code>1 3 6 10</code>	<code>--&gt;cumsum([1 2 3 4])</code> <code>ans =</code> <code>1. 3. 6. 10.</code>
Differenz benachbarter Elemente		
<code>[d] = diff(x)</code>	<code>[d] = diff(x)</code>	<code>[d] =diff(x)</code>
<code>&gt;&gt; diff([1 3 6 10])</code> <code>ans =</code> <code>2 3 4</code>	<code>&gt; diff([ 1 3 6 10 ])</code> <code>ans =</code> <code>2 3 4</code>	<code>--&gt;diff([1 3 6 10])</code> <code>ans =</code> <code>2. 3. 4.</code>
Histogramm des Vektors y mit k Klassen darstellen		
<code>hist(y, k)</code>	<code>hist(y, k)</code>	<code>histplot(k, y)</code>
<code>&gt;&gt; hist(randn(1,10000), 10)</code>	<code>&gt; hist(randn(1, 10000), 10)</code>	<code>--&gt;histplot(10, rand(1, 10000, 'normal'))</code>

*Tabelle 33: Vergleich Statistik*

## 2.8.6 Fourieranalyse

Matlab	Octave	Scilab
Diskrete Fast Fourier Transformation aus x mit Fensterbreite b berechnen		
<code>f = fft(x,b)</code>	<code>f = fft(x,b)</code>	<code>f = fft(x)</code>
Ergebnis der FFT um Zentrum des Spektrums verschieben		
<code>fz = fftshift(f)</code>	<code>fz = fftshift(f)</code>	<code>fz = fftshift(f)</code>
<pre>&gt;&gt; t = 0:0.1:500; &gt;&gt; x = 2*cos(3*2*pi*t) + 7*cos(1*2*pi*t); &gt;&gt; f = fft(x, 8192); &gt;&gt; freq=-5:10/8192:5-1/8192; &gt;&gt; plot(freq, 2 * abs(fftshift(f)))</pre>	<pre>&gt; t = 0:0.1:500; &gt; x = 2*cos(3*2*pi*t) + 7*cos(1*2*pi*t); &gt; f = fft(x, 8192); &gt; freq=-5:10/8192:5-1/8192; &gt; plot(freq, 2*abs(fftshift(f)))</pre>	<pre>--&gt;t = 0:0.1:500; --&gt;x = 2*cos(2*3*pi*t) + 7 * cos(1*2*pi*t); --&gt;f = fft(x); --&gt;freq = -5:0.002:5; --&gt;plot(freq, 2 * abs(fftshift(f)))</pre>

Tabelle 34: Vergleich Fourieranalyse

## 2.8.7 Filterung

Matlab	Octave	Scilab
Lineare zeitinvariante Differenzgleichung		
<code>y = filter(b, a, x)</code>	<code>y = filter(b, a, x)</code>	<code>y = flts(x, b/a)</code>
<pre>% gl. Mittelwert 3. Ordnung &gt;&gt; b = [ 1/3 1/3 1/3 ]; &gt;&gt; a = [ 1 ]; &gt;&gt; y = filter(b, a, x);</pre>	<pre>% gl. Mittelwert 3. Ordnung &gt; b = [ 1/3 1/3 1/3 ]; &gt; a = [ 1 ]; &gt; y = filter(b, a, x);</pre>	<pre>// gl. Mittelwert 3. Ordnung --&gt;b = poly([1/3 1/3 1/3], 'z', 'coeff') / z^(2); --&gt;a = poly([1], 'z', 'coeff'); --&gt;y = flts(x,b/a)</pre>

Tabelle 35: Vergleich Filterung

## 2.8.8 Interpolation und Regression

Matlab	Octave	Scilab
Berechnung der Koeffizienten eines linearen Interpolations-/Regressionspolynoms n-ten Grades für die gegebenen Stützstellen		
pA = polyfit(x, y, n)	pA = polyfit(x, y, n)	nicht vorhanden
<pre>&gt;&gt; x = 1:5; &gt;&gt; y = x + randn(size(x)) * 0.3; &gt;&gt; pA = polyfit(x, y, 1); &gt;&gt; y2 = polyval(pA, x); &gt;&gt; plot(x, y, 'ro', x, y2)</pre>	<pre>&gt; x = 1:5; &gt; y = x + randn(size(x)) * 0.3; &gt; pA = polyfit(x, y, 1); &gt; y2 = polyval(pA, x); &gt; plot(x, y, 'ro', x, y2)</pre>	<pre>function [p]=polyfit(x, y, n, s) x = x(:); y = y(:) m = length(x) v = ones(m, n+1) for i=2:n+1, v(:, i) = x.*v(:, i-1), end p = (v\y)'</pre> <p>endfunction</p>
Spline-Interpolation anhand der Stützstellen x und y.		
pp = spline(x, y)	pp = spline(x, y)	pp = splin(x, y)
<pre>&gt;&gt; x = [0:10]; &gt;&gt; y = sin(x); &gt;&gt; xx = [0:0.1:10]; &gt;&gt; pp = spline(x, y); &gt;&gt; yy = ppval(pp, xx); &gt;&gt; plot(x, y, '+', xx, yy);</pre>	<pre>&gt; x = [0:10]; &gt; y = sin(x); &gt; xx = [0:0.1:10]; &gt; pp = spline(x, y); &gt; yy = ppval(pp, xx); &gt; plot(x, y, '+', xx, yy);</pre>	<pre>--&gt;x = [0:10]; --&gt;y = sin(x); --&gt;xx = [0:0.1:10]; --&gt;pp = splin(x, y); --&gt;yy = interp(xx, x, y, pp); --&gt;plot(x, y, '+', xx, yy)</pre>
Funktionswert der Spline-Interpolation berechnen		
y = ppval(pp, x)	y = ppval(pp, x)	y = interp(x, x_orig, y_orig, pp)
s.o.	s.o.	s.o.
Liefert die Funktionswerte der Interpolation an den Stellen xi.		
interp1(x, y, xi, '...')	interp1(x, y, xi, '...')	interp1(x, y, xi, '...')
<pre>&gt;&gt; x = [0:10]; &gt;&gt; y = sin(x); &gt;&gt; xx = [0:0.1:10]; &gt;&gt; y_spline = interp1(x, y, xx, 'spline'); &gt;&gt; y_nearest = interp1(x, y, xx, 'nearest'); &gt;&gt; plot(x, y, '+', xx, y_nearest, xx, y_spline)</pre>	<pre>&gt; x = [0:10]; &gt; y = sin(x); &gt; xx = [0:0.1:10]; &gt; y_spline = interp1(x, y, xx, 'spline'); &gt; y_nearest = interp1(x, y, xx, 'nearest'); &gt; plot(x, y, '+', xx, y_nearest, xx, y_spline)</pre>	<pre>--&gt;x = [0:10]; --&gt;y = sin(x); --&gt;xx = [0:0.1:10]; --&gt;y_spline = interp1(x, y, xx, 'spline'); --&gt;y_nearest = interp1(x, y, xx, 'nearest'); --&gt;plot(x, y, '+', xx, y_nearest, xx, y_spline)</pre>

Tabelle 36: Vergleich Interpolation Regression

**Scilab** kennt keine Funktion, die **Matlabs** `polyfit` entspricht. Da die polynomiale Regression sich mit der Methode der kleinsten Quadrate lösen lässt, kann man `polyfit` durch die Lösung eines überbestimmten Gleichungssystems nachbilden. Der Backslash-Operator führt dies bei einer Matrixdivision implizit durch und liefert eine Lösung im Sinne der kleinsten Quadrate.

## Literatur

- 1: Eaton, John W, GNU Octave Manual, 2002, Network Theory Limited, Bristol
- 2: Alfio Quarteroni and Fausto Saleri, Scientific Computing with MATLAB and Octave, 2006, Springer, Milano
- 3: Bruno Pincon, Einführung in Scilab, 2004, Institut Elie Cartan Nancy, Nancy
- 4: J.M. Zogg, Arbeiten mit Scilab und Scicos, 2007, HTW Chur, Chur
- 5: Stephen Campbell and Jean-Philippe Chancelier and Ramine Nikoukhah, Modeling and Simulation in Scilab/Scicos, 2005, Springer, Berlin
- 6: C. Bunks and J.-P. Chancelier, Engineering and Scientific Computing with Scilab, 1999, Birkhäuser, Boston