

Ingenieurinformatik – Teilmodul II – MATLAB-Simulink

Vorlesung

Einleitung

Kap1 Einführung in MATLAB

Kap2 Felder und Arrays

Kap3 M-Files : Skripte und Funktionen

Kap4 Kontrollstrukturen, Bedingungen

Kap5 Datentypen

Kap6 Anwendungen aus der Analysis

Kap7 Lineare Algebra

Kap8 Numerische Lösung von DGLn

Kap9 Einführung in Simulink

Praktikum

Prakt-1 MATLAB Entwicklungsumgebung

Prakt-2 Ein-Ausgabe

Funktionen zeichnen

Prakt-3 Debugger - Kontrollstrukturen

Prakt-4 Gleichungssysteme

Eigenwerte - Eigenvektoren

Prakt-5 Numerische Lösung von DGLn

Prakt-6 Simulink - Übungsbeispiele

Ingenieurinformatik
Teilmodul II
Numerik für Ingenieure
(Einführung in MATLAB/Simulink)

Ingenieurinformatik - Teilmodul II

Lernziele :

- Einführung in eine numerische Simulationsumgebung

VDI-Richtlinie 3633: „Simulation ist das Nachbilden eines Systems mit seinen dynamischen Prozessen in einem experimentierfähigem Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind.“

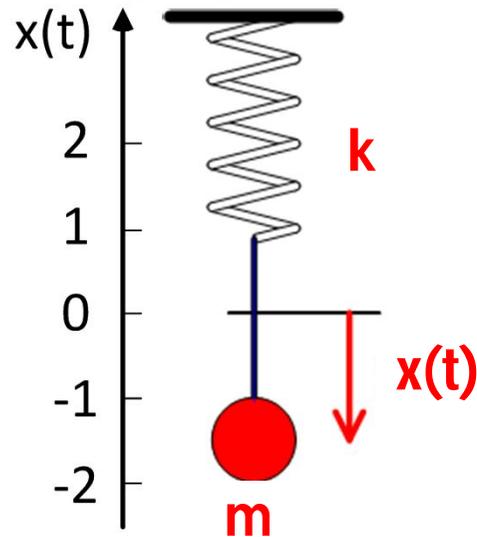
- Umgang mit MATLAB und Simulink
- Numerische Verfahren zur Lösung technischer Probleme
- Ausgewählte Probleme aus Maschinenbau, Fahrzeugtechnik, Luft- und Raumfahrttechnik mit MATLAB und Simulink lösen

Voraussetzungen :

- Beherrschung einer Programmiersprache (Ingenieurinformatik - Teilmodul I)
- Ingenieurmathematik
 - Lineare Algebra : Vektoren, Matrizen, Eigenwerte und Eigenvektoren
 - Analysis : Differential- und Integralrechnung
 - Gewöhnliche Differentialgleichungen
- Technische Mechanik : Schwingungen

Typisches Problem, das einfach mit MATLAB oder Simulink gelöst werden kann :

Schwingung einer Masse mit Dämpfung (siehe Wikipedia)



Bewegungsgleichungen (δ Abklingkonstante) :

$$m \cdot \ddot{x}(t) = -k \cdot x(t) - 2 \cdot \delta \cdot m \cdot \dot{x}(t)$$

$$\ddot{x}(t) + 2 \cdot \delta \cdot \dot{x}(t) + \omega_0^2 \cdot x(t) = 0$$

$$\omega_0^2 = \frac{k}{m}$$

Anfangsbedingungen :

$$x(t=0) = x_0 \quad \dot{x}(t=0) = v_0$$

Für den Fall $\delta=0$ (keine Dämpfung/Reibung) lautet die allgemeine Lösung der DGL:

$$x(t) = A \cdot \sin(\omega_0 \cdot t) + B \cdot \cos(\omega_0 \cdot t)$$

Die Konstanten A und B ergeben sich aus den beiden Anfangsbedingungen. Die Lösung des Anfangswertproblems lautet damit :

$$x(t) = x_0 \cdot \cos(\omega_0 \cdot t) + \frac{v_0}{\omega_0} \cdot \sin(\omega_0 \cdot t)$$

MATLAB-Skript: `schwingungScr.m`

```
% Parameter der DGL
global w0 delta;
T = 2;           % Periodendauer
w0 = 2*pi/T;    % Eigenfrequenz
delta = 0.25;   % Abklingkonstante

% Anfangsbedingungen
y0 = [-2; 0.0]; % Vektor mit 2 Zeilen

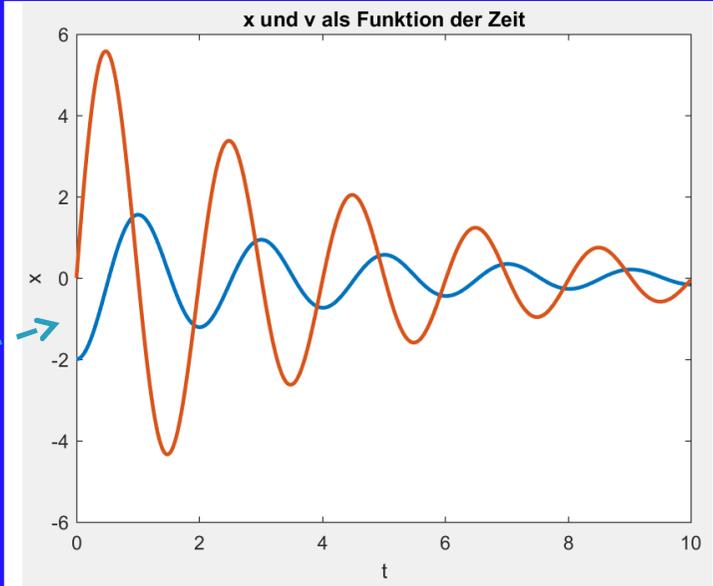
% Differentialgleichung mit ode45 lösen
[t, erg] =
    ode45(@schwingungDgl,[0:0.02:10],y0);

% Lösung graphisch ausgeben
plot(t, erg,'LINEWIDTH',2.0)
xlabel('t'); ylabel('x');
title('x und v als Funktion der Zeit');

% Lösung tabellarisch ausgeben
[t, erg] % Ausgabe von t, x und v
```

```
function dy_dt = schwingungDgl(t,y)
global w0 delta;
dy_dt(1,1)= y(2);
dy_dt(2,1)=-2*delta*y(2)-w0^2*y(1);
end
```

MATLAB-Funktion: `schwingungDgl.m`



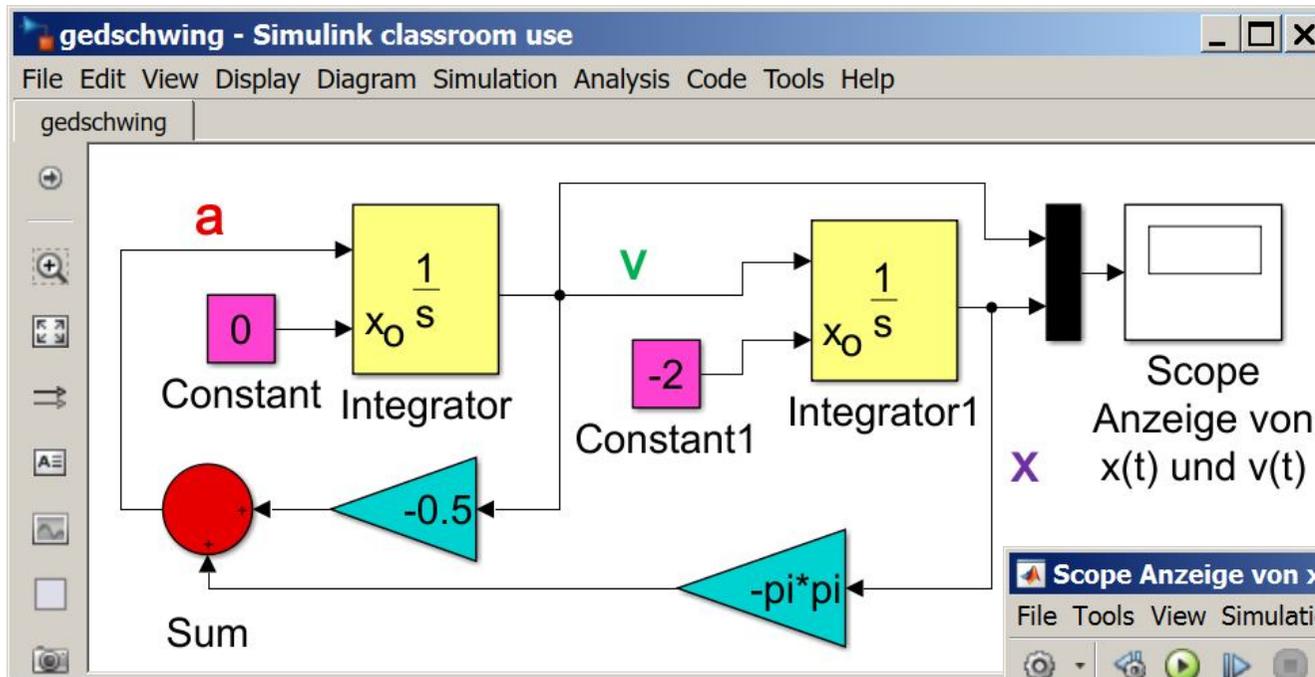
t	x(t)	v(t)
0.000	-2.0000	0.0000
0.020	-1.9961	0.3926
0.040	-1.9843	0.7797
0.060	-1.9649	1.1599
...

Variable `t` ist ein Vektor - Zeitpunkte

Variable `erg` ist eine **Matrix** mit 2 Spalten (x und v)!

`t` und `erg` besitzen gleich viele Zeilen

E_04 Was ist Simulink ? - Lösung graphisch erstellen



$$\ddot{x}(t) = -2 \cdot \delta \cdot \dot{x}(t) - \omega_0^2 \cdot x(t)$$

$$\text{mit } \omega_0 = \pi \text{ und } \delta = 0.25$$

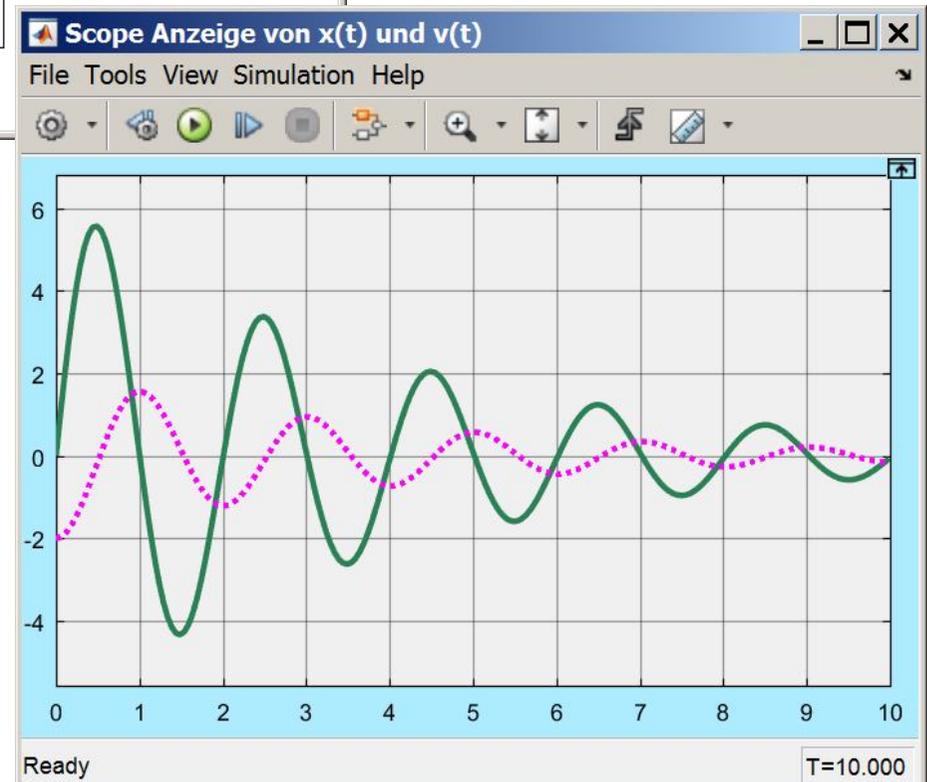
$$\ddot{x}(t) = -0.5 \cdot \dot{x}(t) - \pi^2 \cdot x(t)$$

$$x(t=0) = -2 \quad v(t=0) = 0$$

Simulink-Modell: **gedschwing.slx**

3 Signale : **a**, **v** und **x**

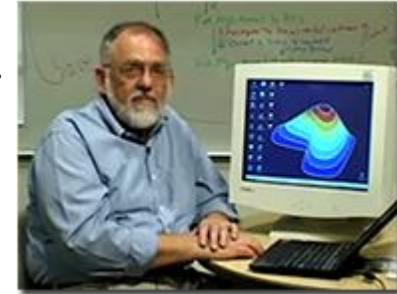
- Integration der Beschleunigung **a** ergibt die Geschwindigkeit **v**
- Integration der Geschwindigkeit **v** ergibt die Auslenkung **x**
- Bei jeder Integration wird jeweils eine Anfangsbedingung verwendet
- Beschleunigung **a** aus **x** und **v** berechnen



MATLAB - Abkürzung für **MAT**rix **LAB**oratory

- Programmierumgebung, die in den 1970er Jahren entwickelt worden ist, um auf einfache und komfortable Weise die numerischen Funktionsbibliotheken LINPACK und EISPACK zu nutzen (Softwarebibliotheken um Probleme aus der Linearen Algebra zu lösen, z.B. Gleichungssysteme, Eigenwerte und Eigenvektoren)

"**Cleve Moler** is chief mathematician, chairman, and cofounder of MathWorks. Moler was a professor of math and computer science for almost 20 years at the University of Michigan, Stanford University and the University of New Mexico."



"In addition to being the author of the first version of MATLAB, Moler is one of the authors of the LINPACK and EISPACK scientific subroutine libraries."

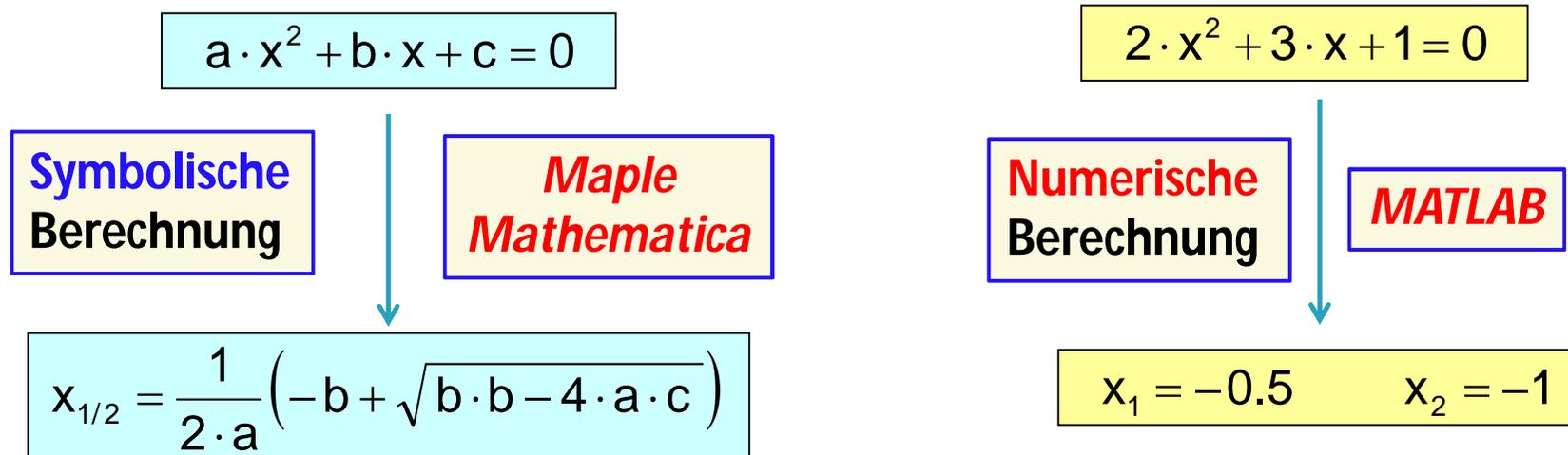
- zunächst in der Programmiersprache FORTRAN geschrieben, später in C
- **The MathWorks, Inc.** entwickelt und vertreibt MATLAB
www.mathworks.com www.mathworks.de
- Kostenlose Alternativen (?) zu MATLAB
Scilab – www.scilab.org Octave – www.octave.org
Python

MATLAB/Simulink besonders geeignet für

- Numerische Berechnungen
- Visualisierung von Daten
- Modellierung und Simulation technischer Probleme

Im Gegensatz zu **Maple** oder **Mathematica** arbeitet **MATLAB** nicht symbolisch sondern **numerisch**. Z.B. die Lösung einer quadratischen Gleichung wird mit MATLAB nicht analytisch berechnet sondern numerisch. D.h. das **Ergebnis der Berechnung sind Zahlen und keine geschlossene Formeln** (Symbolische Berechnungen können aber trotzdem mit MATLAB durchgeführt werden).

Beispiel : Lösung einer quadratischen Gleichung



MATLAB kann auf verschiedene Weise genutzt werden

- **Interaktiv** (als „Taschenrechner“)
direkte Eingabe von Anweisungen/Befehlen und sofortige Ausführung
nur für sehr einfache Probleme geeignet
- **Programmiersprache**
Anweisungen/Befehle werden in einer Datei (**M-File**) gespeichert –
MATLAB-Programm. Dieses Programm wird anschließend ausgeführt.

MATLAB/Simulink bietet bereits eine Vielzahl von fertigen Lösungen für verschiedenste Probleme aus den Ingenieur-, Natur- und Finanzwissenschaften.

Toolboxen (Programmbibliotheken) für

- Entwurf und Analyse von Steuerungs- und Regelungssystemen
- Bildverarbeitung und Computer Vision
- Signalverarbeitung und Kommunikation
- Test- und Messtechnik
- Rapid Prototyping und HIL-Simulation
- Codegenerierung und Verifikation
- ...

Zeitaufwand

Sie müssen **jede Vorlesung mindestens zwei Stunden nachbereiten**. D.h. den Vorlesungsstoff noch einmal durcharbeiten und die Beispiele aus der Vorlesung am Rechner ausprobieren. Ebenso sollten Sie kleinere Änderungen an den Beispielen vornehmen um diese besser zu verstehen. Sie müssen **alle Hausaufgaben selbst bearbeiten!** Für jeden **Praktikumstermin gilt das gleiche**. Wenn Sie das nicht machen, werden Sie den Inhalt der Vorlesung nicht verstehen.

Wenn Sie meinen mit weniger eigenem Aufwand auszukommen, dann täuschen Sie sich meistens selbst oder merken gar nicht, dass Sie den Stoff nicht verstanden haben und nicht anwenden können. Typischerweise können Sie dann beim dritten Praktikumstermin selbst einfachste Aufgaben nicht mehr bearbeiten und sitzen nur herum, ohne etwas zu verstehen.

Wenn Sie Inhalte aus der Mathematik bereits wieder vergessen haben oder nicht anwenden können, werden Sie noch erheblich mehr Zeit verwenden müssen, da Sie Inhalte aus der Mathematik nachholen müssen.

Die C-Programmierung (Teilmodul I, 2. Semester) wird vorausgesetzt.

Skript: Teil 1 - **Vorlesung**, Teil 2 – **Praktikum** , Teil 3 - **Klausuren**

Lehrbücher zur Vorlesung:

- Beucher** MATLAB und Simulink, Verlag mitp, 1. Auflage, 2013
- Hagl** Informatik für Ingenieure, Hanser, 1. Auflage, 2017
- Stein** Programmieren mit MATLAB, Hanser, 6. Auflage, 2017
- Scherf** Modellbildung und Simulation dynamischer Systeme
Oldenbourg Verlag, 4. Auflage, 2010
- Pietruszka** MATLAB und Simulink in der Ingenieurpraxis
Vieweg + Teubner Verlag, 4. Auflage, 2014
- Angermann, Beuschel, Rau, Wohlfahrt**
MATLAB – Simulink - Stateflow
Oldenbourg Verlag, 9. Auflage, 2016

Homepage: <http://reichl.userweb.mwn.de/>
<http://kuepper.userweb.mwn.de/>

Moodle: Numerik für Ingenieure (kurzer Kursname)
Ingenieurinformatik Teil2: Numerik für Ingenieure (Reichl)

MATLAB – Simulink Dokumente (R2016b) **Seitenanzahl**

MATLAB Primer 206

MATLAB Programming Fundamentals 1222

MATLAB Mathematics 642

MATLAB Function Reference 12488

...

Simulink Getting Started Guide 92

Simulink User's Guide 4022

Simulink Reference Guide 4164

...

Stateflow Getting Started Guide 110

Stateflow User's Guide 1418

...

Transferlaufwerk : I:\Dozenten\Ingenieurinformatik\MatLabDocu

Online : www.mathworks.de/support -> Documentation

http://www.rz.hm.edu/startseite_1/index.de.html

HOME HM | UNTERNEHMEN | PRESSE | ALUMNI | INTRANET | IMPRESSUM ENGLISH VERSION

ACCOUNT > WLAN / EDUROAM > WEBMAIL-CLIENT >

Startseite

ZENTRALE IT DAS RECHENZENTRUM DER HOCHSCHULE MÜNCHEN

- > **Startseite**
- > Infos für Beschäftigte
- > Infos für Studierende
- > Infos für Gäste
- > Sicherheitsthemen
- > Wir über uns
- > Sitemap

Datenschutz & Webtrack

Übersicht

Benutzeraccount

E-Mail / Exchange

Internet-Zugang (WLAN / eduroam)

LRZ Sync+Share

HM-Cloud

Softwarebezug

IT-Labore / Rechenräume

Studierenden-Services

Die Abteilung Zentrale IT ist eine zentrale Einrichtung der Hochschule München. Unsere Aufgabe besteht in der Bereitstellung, dem Betrieb und der laufenden Wartung der technischen Infrastruktur. Darüber hinaus sind wir der zentrale Ansprechpartner für die Schulverwaltung.

Das Supportangebot der Abteilung Zentrale IT gestaffelt nach den verschiedenen Fachbereichen der Hochschule München.

Die Zentrale IT ist nicht der alleinige IT-Dienstleister an der Hochschule München. Die IT-Dienste und Services werden dezentral von den Fakultäten jeweils für die eigenen Bereiche bereitgestellt bzw. administrativ verwaltet.

Die **Übersicht** zu diesen Diensten und Services gibt auch Informationen zu Art und Aufgabenspektrum der jeweiligen IT-Beauftragten.

Wir benötigen nur MATLAB und Simulink. Installiert man alle angebotenen Toolboxen braucht man ca. 17 GB Speicherplatz.

- 1 Einführung in MATLAB**
- 2 Felder – Arrays**
- 3 M-Files : Skripte und Funktionen**
- 4 Kontrollstrukturen und Bedingungen**
- 5 Datentypen**
- 6 Anwendungen aus der Analysis**
- 7 Lineare Algebra**
- 8 Numerische Lösung von Differentialgleichungen**
- 9 Einführung in Simulink**

Kapitel 1 – Einführung in MATLAB

1.1 MATLAB - Entwicklungsumgebung

1.2 MATLAB – Interaktiver Modus (Taschenrechner)

1.3 MATLAB - Variablen

1.4 MATLAB - Matrizen

1.5 MATLAB - M-Files

1.6 Hausaufgaben

The screenshot shows the MATLAB R2016b - classroom use interface. The top menu bar includes HOME, PLOTS, and APPS. The ribbon contains various toolbars: FILE (New Script, New, Open, Compare), VARIABLE (Import Data, Save Workspace, Clear Workspace), CODE (Analyze Code, Run and Time, Clear Commands), SIMULINK (Simulink), and ENVIRONMENT (Preferences, Set Path, Parallel). A yellow box labeled "Toolstrip" highlights the ribbon area.

The interface is divided into several panes:

- Current Folder:** Shows a list of files: gedschwing.slx, schwingungDgl.m, and schwingungScr.m. A yellow box labeled "Current Folder" lists its functions: Arbeitsverzeichnis, Dateiverwaltung, and Zugriff auf M-Files (Skripte, Funktionen).
- Command Window:** Displays the execution of MATLAB commands: `>> x=2*4`, `x = 8`, `>> z=sind(30)`, `z = 0.5000`, and `fx >>`. A yellow box labeled "Command Window" lists its functions: Befehle eingeben, Ergebnisse anzeigen, Fehlermeldungen anzeigen, Skripte ausführen, and Funktionen aufrufen.
- Workspace:** Shows a table of variables: x (8) and z (0.5000). A yellow box labeled "Workspace" lists its function: Liste der aktuellen Variablen und ihrer Eigenschaften.
- Command History:** Shows a list of executed commands, including "CIC" and "schwingungScr". A yellow box labeled "Command History" lists its functions: Liste der bisher eingegebenen Befehle and Befehle wählen und erneut ausführen.

```
Command Window
>> 3+4+3*3
ans =
    16
>> 3^4+5
ans =
    86
>> 7/4
ans =
    1.7500
```

```
Command Window
>> 3+4.0
ans =
     7
>> 3.1+4.2
ans =
    7.3000
>> 3.1+3.9
ans =
     7
```

```
Command Window
>> x=50
x =
    50
>> y=sqrt(x)
y =
    7.0711
>> cos(0)
ans =
     1
>> z=sind(30)
z =
    0.5000
>> z=x*sqrt(y)
z =
   132.9574
```

```
Command Window
>> sinus(0)
Undefined function 'sinus' for input
arguments of type 'double'.

>> cos(o)
Undefined function or variable 'o'.
```

Für einfachste Probleme kann MATLAB wie eine Art Taschenrechner verwendet werden. Der Anwender gibt einen Befehl (Ausdruck) im Command-Window ein, MATLAB führt diesen Befehl aus und zeigt das Ergebnis an, ggf. auch eine **Fehlermeldung**. Das Ergebnis wird in der Standardvariablen **ans** (answer) gespeichert oder in einer Variablen mit vorgegebenem Namen (hier : x, y, z).

Help

Documentation

CONTENTS

sin

Sine of argument in radians

Syntax

$Y = \sin(X)$

Description

$Y = \sin(X)$ returns the sine of the elements of X. The `sin` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval $[-\infty, \infty]$, `sin` returns real values in the interval $[-1, 1]$. For complex values of X , `sin` returns complex values. All angles are in radians.

Examples

Plot Sine Function

Plot the sine function over the domain $-\pi \leq x \leq \pi$. [Open Script](#)

```
x = -pi:0.01:pi;
plot(x,sin(x)), grid on
```



Command Window

```
>> help sin
sin    Sine of argument in radians.
       sin(X) is the sine of the elements of X.

       See also asin, sind.

       Reference page for sin
       Other functions named sin
```

sin - Sinus

radians – Bogenmaß

Was bedeutet „elements“ ?

X darf Skalar, Vektor oder Matrix sein!

Command Window

```
>> sin(pi/6)
ans =
    0.5000
[pi/6 pi/3 pi/2]
>> sin([pi/6 pi/3 pi/2])
ans =
    0.5000    0.8660    1.0000
```

Das Ergebnis eines Ausdrucks wird, falls keine linke Seite (z.B. $y = \dots$) angegeben ist, in der **Standardvariablen ans** (answer) gespeichert.

```
Command Window
>> help ans
ans Most recent answer.
ans is the variable created automatically when expressions
are not assigned to anything else. ANSwer.
```

Grundrechenarten: + - * / ^ (Potenzierung) ~~%~~

Elementare Funktionen:

```
Command Window
//
>> help matlab\elfun
Elementary math functions.

Trigonometric.
  sin          - Sine.
  sind         - Sine of argument in degrees.
  sinh        - Hyperbolic sine.
  asin        - Inverse sine.
  asind       - Inverse sine, result in degrees.
  asinh       - Inverse hyperbolic sine.
  cos         - Cosine.
```

tan
exp
log
log10
sqrt
abs
mod
sign

In der **Programmiersprache C** und auch in den meisten anderen Programmiersprachen muss eine Variable vor der Verwendung **explizit definiert** werden. Bei der Definition wird der **Datentyp** festgelegt und es wird ein **Variablenname** gewählt.

Beispiel :

```
int i;
```

```
i = 5;
```

```
i[3] = 7;
```

```
i = "Hallo";
```

// Zuweisung ist **korrekt**

// Anweisung erzeugt beim Compilieren einen **Fehler**

// Anweisung erzeugt beim Compilieren einen **Fehler**

Durch die explizite Deklaration werden Programme sicherer, da eine Reihe von Fehlern bereits durch den Compiler erkannt werden. Durch implizite Typumwandlungen können sich aber dennoch Fehler einschleichen.

```
i = 5.6;
```

// Anweisung erzeugt beim Compilieren eine **Warnung**

Vektoren und Matrizen in C :

```
double x[3];
```

// Vektor mit 3 Elementen - **x[0], x[1], x[2]**

```
double mat[3][4];
```

// Matrix mit 3 Zeilen und 4 Spalten

```
mat[0][0] ... mat[2][3]
```

Die Größe eines Vektors oder einer Matrix wird bei der Definition festgelegt und kann danach nicht mehr verändert werden!

Bei MATLAB gilt (etwas vereinfacht, später genauer) folgendes :

- a) Eine MATLAB-Variable ist immer eine **Matrix** mit **n** Zeilen und **m** Spalten – jede MATLAB-Variable ist eine **n*m-Matrix** (ein **2d-Array**).
- b) Variablen müssen **nicht explizit definiert** werden – häufig wird eine Variable bei der ersten Verwendung erzeugt z.B. durch eine Wertzuweisung; MATLAB legt dabei die Anzahl der Zeilen und Spalten automatisch fest.
- c) Alle Elemente einer Matrix sind vom Typ **double**, d.h. der Wert jedes Elements einer Matrix ist vom Typ double (64-Bit, ca. 15-16 führende Stellen).
- d) Es gibt drei wichtige **Spezialfälle**:
 - 1*1** – Matrix - **Skalar** - Matrix, die genau ein Element besitzt
 - 1*n** – Matrix - **Zeilenvektor** - Matrix mit genau einer Zeile
 - n*1** – Matrix - **Spaltenvektor** - Matrix mit genau einer Spalte
- e) Es gibt viele Operatoren und Funktionen, mit denen Matrizen einfach definiert, initialisiert, verändert oder verarbeitet werden können. Die meisten **Operatoren sind bereits für Matrizen definiert**.
- f) Die **Anzahl der Zeilen und Spalten** einer Matrix kann sich während des Programmablaufs **ändern** (beabsichtigt oder unbeabsichtigt aufgrund eines Fehlers). Die Dimension einer Matrix kann vergrößert (Zeilen oder Spalten einfügen) oder verkleinert (Zeilen oder Spalten löschen) werden.
- g) **Zeilen- und Spaltenindizes beginnen bei 1 (nicht bei 0 wie in C)**.

```

Command Window
>> A = [2,3; 4,5]
A =
     2     3
     4     5
>> x = [6; 7]
x =
     6
     7
>> y = A * x
y =
    33
    59
>> whos
  Name  Size      Bytes  Class
  A      2x2         32  double
  x      2x1         16  double
  y      2x1         16  double

```

Alle MATLAB-Variablen sind Matrizen/Arrays !

Die Variable **A** ist hier eine **2*2** Matrix. Diese Variable wird durch eine Zuweisung erzeugt.

x und **y** sind Spaltenvektoren (**2*1**-Matrizen)

[] ist der sogenannte **Concatenation-Operator**

; erzeugt neue Zeile in einem Array

, oder **Leerzeichen** trennen Spalten in einer Zeile

A = [2 3; 4 5] identisch mit **A = [2, 3; 4, 5]**

Vektor **x** wird bei der Initialisierung erzeugt.

Vektor **y** wird als Ergebnis einer Zuweisung erzeugt.

Der Operator ***** (Multiplikationsoperator) ist auch für Vektoren und Matrizen definiert (Matrix-Vektormultiplikation aus der Mathematik). Zeilen- und Spaltenzahl der Operanden müssen zueinander passen. Bei einer **1*1** Matrix (Skalar) bewirkt der Operator ***** eine „ganz normale Multiplikation“.

```

Command Window
>> A = [2,3; 4,5]
A =
     2     3
     4     5
>> A(1,2)
ans =
     3
>> z = A(2,1)
z =
     4
>> A(2,3)
Index exceeds matrix dimension

```

```

Command Window
>> z = [5]
z =
     5
>> z(1,1)
ans =
     5
>> z(1)
ans =
     5
>> z
z =
     5

```

```

Command Window
>> x = [10; 11 ; 12 ]
x =
    10
    11
    12
>> x(2,1)
ans =
    11
>> x(2)
ans =
    11
>> y = [20, 21, 22]
y =
    20    21    22
>> y(1,2)
ans =
    21
>> y(2)
ans =
    21

```

Zugriff auf die Elemente einer Matrix :

$A(\text{zeile}, \text{spalte})$

Bei einer 1×1 -Matrix (**Skalar**) gilt :

$z(1,1) \equiv z(1) \equiv z$ und $z=[5] \equiv z=5$

Bei **Spaltenvektoren** ($n \times 1$ -Matrix) gilt :

$x(\text{zeile}, 1) \equiv x(\text{zeile})$

Bei **Zeilenvektoren** ($1 \times n$ -Matrix) gilt :

$y(1, \text{spalte}) \equiv y(\text{spalte})$

linear Indexing !

```

>> x = 2
x =
    2
>> whos x
  Name  Size  Bytes  Class
  x     1x1     8  double
>> x(1)
ans =
    2
>> x(2) = 12
x =
    2    12
>> whos x
  Name  Size  Bytes  Class
  x     1x2    16  double
>> x(2,3) = 17
x =
    2    12     0
    0     0    17
>> whos x
  Name  Size  Bytes  Class
  x     2x3    48  double

```

Die Größe oder Dimension (Anzahl der **Zeilen** und **Spalten**) der Variablen **x** wird hier durch geeignete Wertzuweisungen von MATLAB **automatisch** verändert.

- zuerst ist **x** eine **1*1** Matrix (**Skalar**)
- dann wird **x** zu einer **1*2** Matrix (**Zeilenvektor**)
- danach wird **x** zu einer **2*3** Matrix ; bei der Erweiterung von **x** zu einer **2*3** Matrix werden nicht explizit initialisierte Matrixelemente automatisch auf **0** gesetzt

Die automatische Erweiterung von Zeilen oder Spalten kann leicht zu Fehlern führen; Schreibfehler sind oftmals schwer zu finden!

M-Files sind Dateien, die MATLAB-Befehle enthalten. M-Files müssen auf `.m` enden. Es handelt sich um einfache ASCII-Dateien (Text-Dateien), die mit einem beliebigen ASCII-Editor geschrieben werden können. Die MATLAB-IDE enthält bereits einen komfortablen Editor zum Schreiben von M-Files.

Man unterscheidet zwei Typen von M-Files:

- **Skripte**

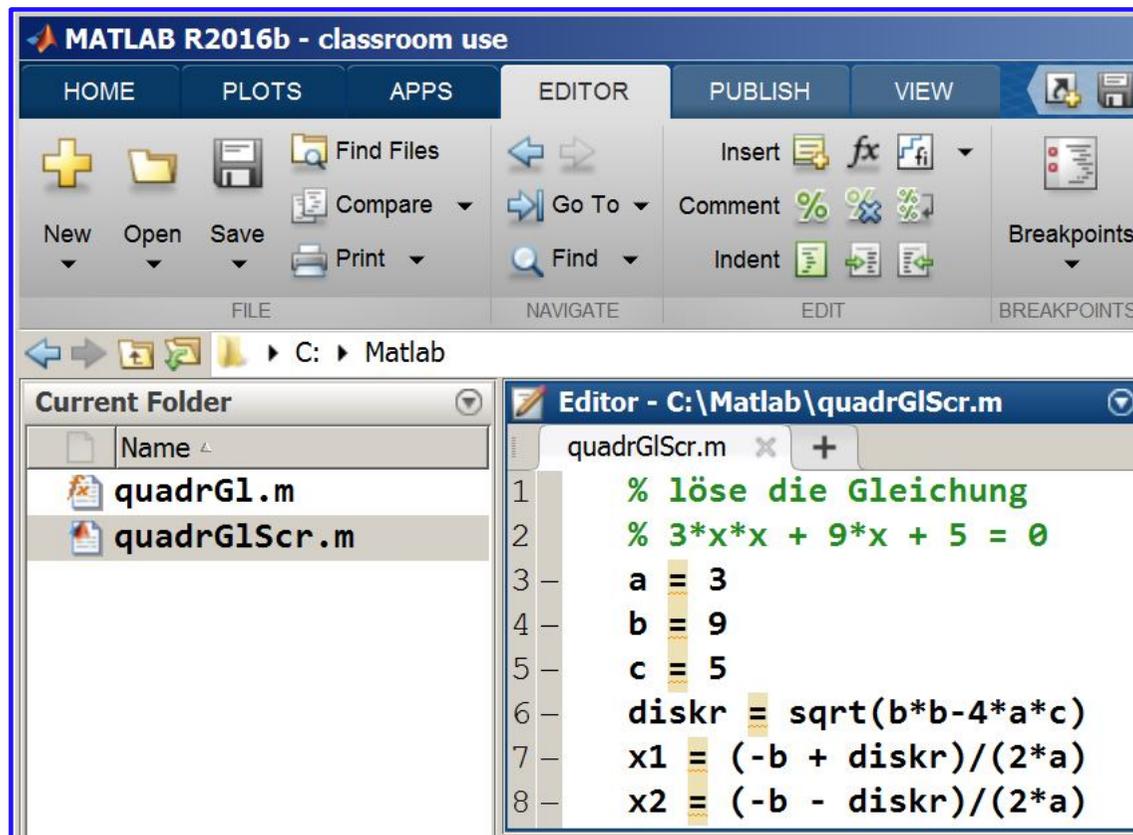
Ein MATLAB-Skript enthält eine Folge von Befehlen, die bei der Ausführung des Skripts (Run) nacheinander bearbeitet werden; gibt man den Dateinamen ohne die Endung im Command-Fenster ein, dann werden die Befehle nacheinander ausgeführt, genauso als hätte man die Befehle direkt in das Command-Fenster eingegeben.

Siehe Beispiel [quadrGlScr.m](#)

- **Funktionen**

Eine MATLAB-Funktion erfüllt den gleichen Zweck wie eine Funktion in der Programmiersprache C. Im Normalfall bearbeitet eine Funktion eine ganz bestimmte Aufgabe. Eine Funktion besitzt typischerweise Übergabeparameter und gibt ein oder mehrere Ergebnisse zurück. Der Aufruf einer Funktion erfolgt im Command-Fenster oder innerhalb von anderen Funktionen oder Skripten.

Siehe Beispiel [quadrGl.m](#)

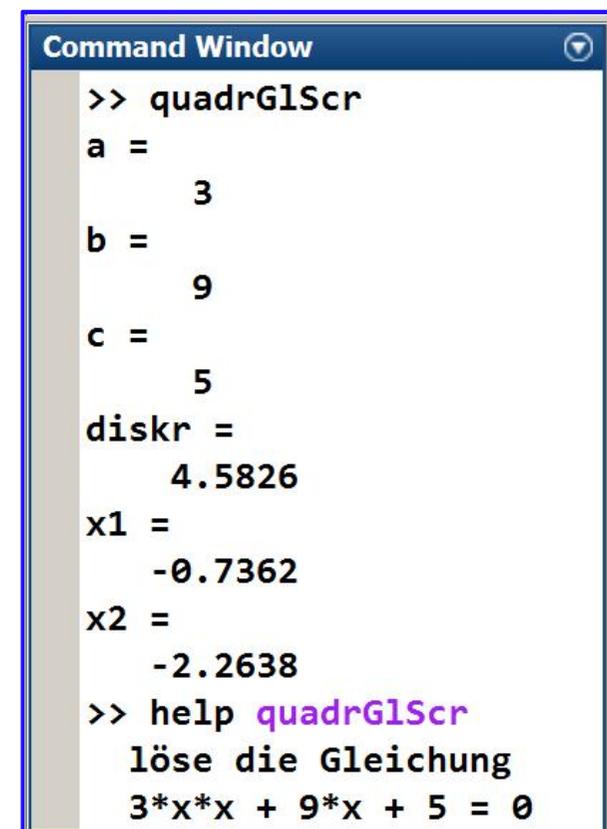


The screenshot shows the MATLAB R2016b interface. The Editor window displays the script `quadrG1Scr.m` with the following content:

```

1  % löse die Gleichung
2  % 3*x*x + 9*x + 5 = 0
3  a = 3
4  b = 9
5  c = 5
6  disk = sqrt(b*b-4*a*c)
7  x1 = (-b + disk)/(2*a)
8  x2 = (-b - disk)/(2*a)

```



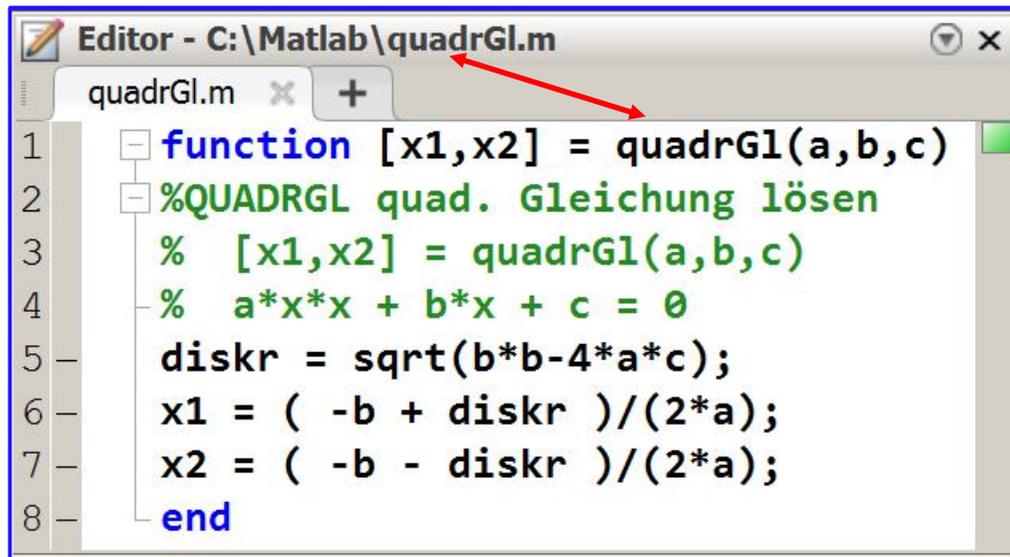
The screenshot shows the MATLAB Command Window with the following output:

```

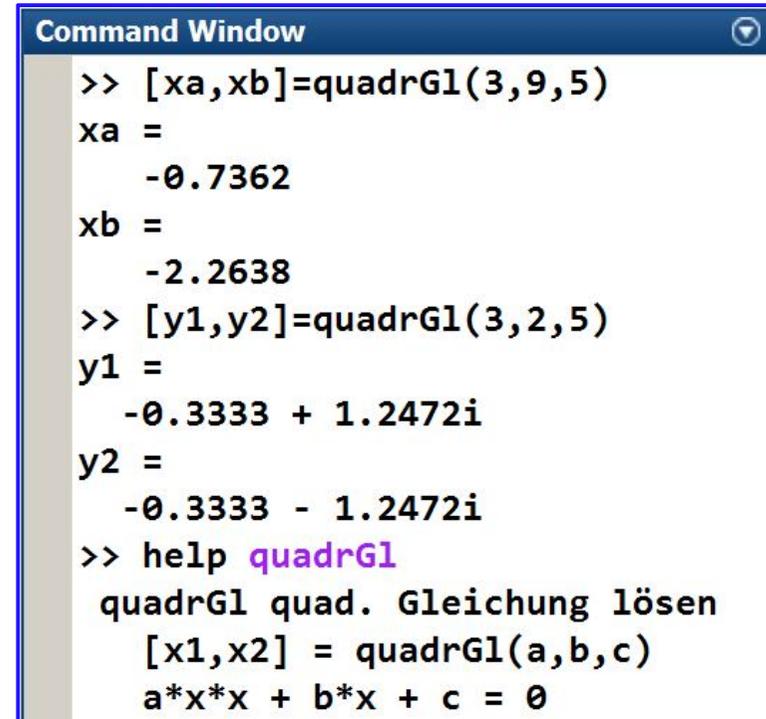
>> quadrG1Scr
a =
    3
b =
    9
c =
    5
disk =
    4.5826
x1 =
   -0.7362
x2 =
   -2.2638
>> help quadrG1Scr
löse die Gleichung
3*x*x + 9*x + 5 = 0

```

Die Datei mit dem Namen `quadrG1Scr.m` ist ein **MATLAB-Skript**. Es enthält Anweisungen zur Lösung einer quadratischen Gleichung. Der Name der Datei ist frei wählbar, muss aber auf `.m` enden. Das gezeigte Skript besteht aus zwei Kommentarzeilen und sechs Anweisungen. Die Anweisungen werden nacheinander ausgeführt, wenn man den Namen des Skripts - `quadrG1Scr` - im Command-Window eingibt. Das Zeichen `%` leitet einen Kommentar ein. Der Befehl `help M-Filename` zeigt die ersten Kommentarzeilen an.



```
Editor - C:\Matlab\quadrGl.m
quadrGl.m x +
1 function [x1,x2] = quadrGl(a,b,c)
2 %QUADRGL quad. Gleichung lösen
3 % [x1,x2] = quadrGl(a,b,c)
4 % a*x*x + b*x + c = 0
5 disk = sqrt(b*b-4*a*c);
6 x1 = ( -b + disk )/(2*a);
7 x2 = ( -b - disk )/(2*a);
8 end
```



```
Command Window
>> [xa,xb]=quadrGl(3,9,5)
xa =
    -0.7362
xb =
    -2.2638
>> [y1,y2]=quadrGl(3,2,5)
y1 =
    -0.3333 + 1.2472i
y2 =
    -0.3333 - 1.2472i
>> help quadrGl
quadrGl quad. Gleichung lösen
[x1,x2] = quadrGl(a,b,c)
a*x*x + b*x + c = 0
```

Die Datei `quadrGl.m` enthält die Funktion mit dem Namen `quadrGl`. Datei- und Funktionsname (siehe Zeile 1 der Funktion) müssen übereinstimmen (bis auf die Endung `.m`).

Die Funktion berechnet die Lösungen einer quadratischen Gleichung und gibt diese zurück. Beim Aufruf der Funktion werden 3 Parameter übergeben. Auf der linken Seite des Funktionsaufrufes wird festgelegt, in welchen Variablen die beiden Lösungen gespeichert werden. Mit der Funktion lassen sich beliebige quadratische Gleichungen lösen, auch solche, die komplexe Lösungen besitzen.

1. Installieren Sie MATLAB und Simulink auf ihrem Rechner.
2. Probieren Sie alle Beispiele aus dem ersten Kapitel aus, d.h. geben Sie die Befehle am Rechner ein.
3. Schreiben Sie das Skript `quadrGlScr.m` zur Berechnung der Lösung einer quadratischen Gleichung. Führen Sie das Skript aus.
4. Schreiben Sie die MATLAB-Funktion `quadrGl` zur Berechnung der Lösung einer quadratischen Gleichung. An die Funktion werden die 3 Koeffizienten der Gleichung übergeben. Die Funktion gibt die beiden (reellen oder komplexen) Lösungen zurück.

Anschließend berechnen Sie mit dieser Funktion die Lösungen der beiden Gleichungen :

a) $3 \cdot y^2 + 9 \cdot y + 5 = 0$

b) $3 \cdot y^2 + 2 \cdot y + 5 = 0$

Wie lautet der Aufruf bei Teilaufgabe a), wenn die Ergebnisse in den beiden Variablen `y1` und `y2` gespeichert werden sollen. Wie lautet der Aufruf bei Teilaufgabe b), wenn die Ergebnisse in `z1` und `z2` gespeichert werden sollen.

c) Was passiert, wenn beide Lösungen zusammenfallen? Überlegen Sie sich ein entsprechendes Beispiel. Berechnen Sie dann die Lösungen.

5. Berechnen Sie folgende Matrix-Vektormultiplikation auf zwei Arten :

- handschriftlich
- mit Hilfe von MATLAB

$$\begin{pmatrix} 3 & -2 \\ 2 & 4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

6. Berechnen Sie folgende Matrix-Vektormultiplikation auf zwei Arten :

- handschriftlich
- mit Hilfe eines MATLAB-Skripts

$$\begin{bmatrix} 3 & -2 & 2 \\ 2 & 3 & 4 \\ 1 & -2 & 2 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 3 \end{bmatrix}$$

7. Was passiert, wenn Sie die C-Anweisungen (siehe unten) nach MATLAB „übersetzen“ und ausführen? Erstellen Sie ein MATLAB-Skript mit den Befehlen. Führen Sie das Skript aus.

```
int i;  
i = 5;  
i[3] = 7;  
i = "Hallo";
```

C-Anweisungen

```
i = 5  
i(3) = 7  
i = 'Hallo'  
i(2)
```

MATLAB-Anweisungen

Kapitel 2 Felder - Arrays

2.1 Begriffe : Array – Matrix – Vektor – Skalar

2.2 Arrays erzeugen

2.3 Operatoren und Arrays

2.4 Multiplikation von Vektoren und Matrizen

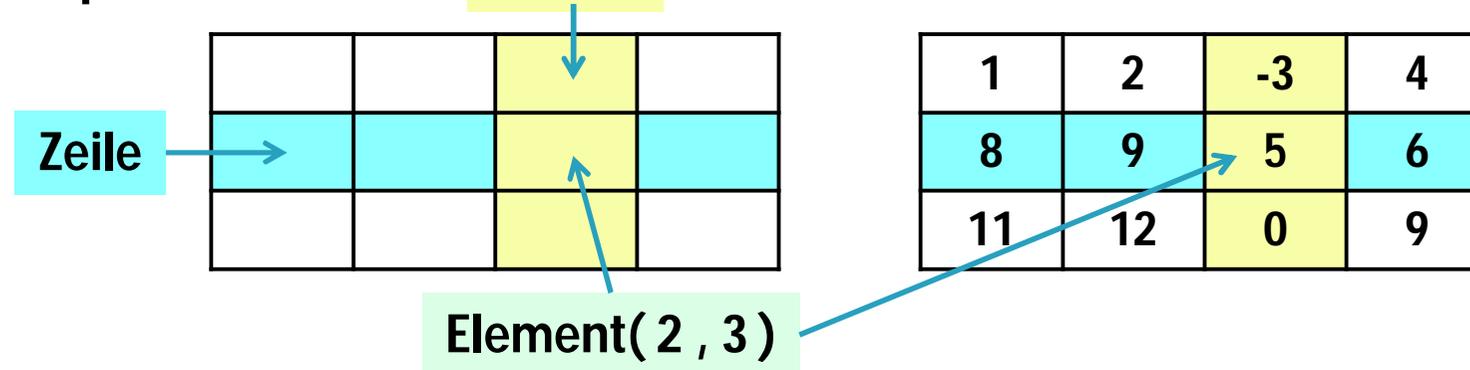
2.5 Vektoren für die Funktion plot erzeugen

2.6 Skalarprodukt

2.7 Hausaufgaben

Eine $n*m$ -**Matrix** (ein 2d-Array) ist der zentrale Datentyp bei MATLAB.

Beispiel : 3*4-Matrix **Spalte**



- Alle Elemente einer Matrix besitzen den **gleichen Datentyp**, d.h. alle Elemente sind vom Typ `double` (oder vom Typ `uint8` , `float` ...)
- Die Elemente einer Matrix sind vom Typ `double`, außer der Typ der Elemente der Matrix wird explizit anders festgelegt.
- Der **Zugriff** auf ein einzelnes Element einer Matrix erfolgt normalerweise durch Angabe von **Zeile** (erster Index) und **Spalte** (zweiter Index) z.B. `A(2,3)`.
- Der Zugriff auf ein Element kann auch mit **einem einzigen (linear indexing)** Index erfolgen. Hierbei werden die Elemente der Matrix **spaltenweise durchnummeriert**, beginnend bei 1. Daher gilt im obigen Beispiel :

$$A(2,3) \equiv A(8)$$

$$A(3,3) \equiv A(9)$$

$$A(1,4) \equiv A(10)$$

Array oder Feld:

Beide Begriffe bezeichnen ganz allgemein ein **mehrdimensionales Objekt**.
 Ein Element eines n-dimensionalen Arrays wird durch n Indizes
 ($k_1, k_2, k_3, \dots, k_n$) eindeutig festgelegt ($n \geq 2$).

$$A(k_1, k_2, k_3, \dots, k_n)$$

Spezielle Arrays:• **Matrix**

2-dimensionales Array – 2d-Array – $n \times m$ -Array oder $n \times m$ -Matrix

$$A(\text{zeile}, \text{spalte}) \quad A(2, 3) \quad A(n) \quad (A(n) : \text{linear indexing})$$

• **Vektor**

2d-Array mit genau einer Zeile ($1 \times n$ -Array – **Zeilenvektor** – **row vector**) oder

2d-Array mit genau einer Spalte ($n \times 1$ -Array – **Spaltenvektor** – **column vector**)

$$A(1, k) \equiv A(k) \quad A(k, 1) \equiv A(k) \quad (A(k) : \text{linear indexing})$$

• **Skalar**

2d-Array mit genau einer Zeile und einer Spalte (1×1 -Array)

$$A(1, 1) \equiv A(1) \equiv A$$

Der Begriff Matrix wird oft in zwei unterschiedlichen Bedeutungen verwendet:

1. Darstellung mathematischer oder physikalischer Objekte

Drehung in der Ebene : Darstellung durch eine **2*2**-Matrix

Drehung im Raum : Darstellung durch eine **3*3**-Matrix

Drehstreckung : Darstellung durch eine **3*3**-Matrix

Trägheitsmatrix : Darstellung durch eine **3*3**-Matrix

n Gleichungen mit **m** Unbekannten : **n*m**-Matrix – Koeffizientenmatrix

$$\begin{array}{l} 3 \cdot x + 4 \cdot y - 5 \cdot z = 2 \\ 2 \cdot x + 5 \cdot y + 6 \cdot z = 4 \\ 3 \cdot x + 4 \cdot y - 7 \cdot z = 3 \end{array} \quad \begin{pmatrix} 3 & 4 & -5 \\ 2 & 5 & 6 \\ 3 & 4 & -7 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}$$

Bei diesen Matrizen ist es von Bedeutung, ob sie symmetrisch sind, schiefssymmetrisch, diagonal, quadratisch, Für diese Matrizen kann man den Rang bestimmen, Eigenwerte und Eigenvektoren berechnen oder die Determinante. Diese Eigenschaften einer Matrix haben auch immer eine bestimmte mathematische oder physikalische Bedeutung (z.B. Eigenwert \Leftrightarrow Eigenfrequenz).

2. Zusammenfassung verschiedener Daten, damit diese einfach und bequem gespeichert und verarbeitet werden können. Die Begriffe Rang, Determinante oder Eigenvektor machen bei solchen Matrizen keinen Sinn.

Beispiel : Speichere die **Zeit**, die **Auslenkung**, die **Geschwindigkeit** und die **Beschleunigung** bei einer gedämpften Schwingung

- **Spalte 1 (Zeile 1)** : speichert die Zeitpunkte
- **Spalte 2 (Zeile 2)** : speichert die zugehörige Auslenkung
- **Spalte 3 (Zeile 3)** : speichert die zugehörige Geschwindigkeit
- **Spalte 4 (Zeile 4)** : speichert die zugehörige Beschleunigung

t	x(t)	v(t)	a(t)
0	-2.0	0.0	19.74
0.3	-1.21	4.72	9.62
0.7	0.87	4.3	-10.7
0.9	1.47	1.60	-15.3
...

0	0.3	0.7	0.9	...	t
-2.0	-1.21	0.87	1.47	...	x(t)
0.0	4.72	4.30	1.60	...	v(t)
19.74	9.62	-10.7	-15.3	...	a(t)

Bei n Zeitpunkten ergibt sich eine $n \times 4$ -Matrix oder eine $4 \times n$ -Matrix. Hinter dieser Matrix steckt kein mathematisches oder physikalisches Objekt.

Man könnte diese Informationen auch in 4 einzelnen Vektoren abspeichern. Das ist oft unpraktisch, wenn man z.B. die Werte an eine Funktion übergeben muss oder die Werte als Ergebnis eines Funktionsaufrufes zurückgeben möchte.

Die Matrix speichert nur Daten.

Kapitel 2 Felder - Arrays

2.1 Begriffe : Array – Matrix – Vektor – Skalar

2.2 Arrays erzeugen

2.2.1 Explizite Angabe der Elemente

2.2.2 Zugriff auf nicht existierende Elemente

2.2.3 Neues Array aus vorhandenen Arrays erzeugen

2.2.4 Colon-Operator

2.2.5 Zugriff auf Teile eines Arrays

2.2.6 Transponieren einer Matrix oder eines Vektors

2.3 Operatoren und Arrays

2.4 Multiplikation von Vektoren und Matrizen

2.5 Vektoren für die Funktion plot erzeugen

2.6 Skalarprodukt

2.7 Hausaufgaben

2.2.1 Explizite Angabe der Elemente

Matrizen mit wenigen Elementen werden **zeilenweise** durch explizite Angabe der Werte der einzelnen Elemente erzeugt. Dabei wird der **Concatenation – Operator []** verwendet (to concatenate – aneinanderhängen, verbinden).

- Zeilen werden durch einen Semikolon **;** voneinander getrennt
- Elemente in einer Zeile werden durch **,** oder **Space (Leerzeichen)** getrennt

```
>> A = [ 1,2,-3,4; 8,9,5,6; 11 12 0 9 ]
```

```
A =
```

```
    1     2    -3     4
    8     9     5     6
   11    12     0     9
```

```
>> A = [ 1, 2, -3, 4;
         8, 9, 5, 6;
         11 12 0 9 ]
```

```
A =
```

```
    1     2    -3     4
    8     9     5     6
   11    12     0     9
```

```
>> x = [ 1 4 8 ];
```

```
>> y = [ 2; 5; 9 ];
```

Matrix :

A : 3*4-Matrix

Vektoren :

x : Zeilenvektor - 1*3-Vektor

y : Spaltenvektor - 3*1-Vektor

K2_08 2.2.2 Zugriff auf nicht-existierende Elemente

Es muss zwischen **lesendem** und **schreibendem** Zugriff unterschieden werden.

Der **lesende Zugriff** auf ein Element eines Arrays, das nicht existiert, erzeugt einen Fehler. Lesender Zugriff bedeutet, dass das (nicht existierende) Element auf der **rechten Seite** des Zuweisungsoperators steht.

```
>> B = [ 1, 2, -3, 4; 8, 9, 5, 6 ]
B = 1      2      -3      4
     8      9      5      6
>> x = 1 + B(3,2)
Attempted to access B(3,2); index out of
bounds because size(B)=[2,4].
```

Steht das nicht existierende Element dagegen **links vom Zuweisungsoperator** (**schreibender Zugriff**), wird das **Array automatisch erweitert**. Es wird der Wert der rechten Seite zugewiesen. Eingefügte Elemente ohne explizite Wertzuweisung werden mit 0 belegt. **Die Größe der Matrix ändert sich damit!!**

```
>> B(3,2) = 11
B =
     1     2    -3     4
     8     9     5     6
     0    11     0     0
```



1	2	-3	4
8	9	5	6
1	2	-3	4
8	9	5	6
0	11	0	0

K2_09 2.2.3 Neues Array aus vorhandenen Arrays erzeugen

Mit dem **Concatenation-Operator** `[]` können aus bereits vorhandenen Arrays (meist Vektoren oder Matrizen) neue Arrays gebildet werden. Es gibt „**horizontal concatenation** `[,]`“ und „**vertical concatenation** `[;]`“.

```
>> A = [ 1, 2; 3, 4 ]
```

```
A =
```

```
    1    2
    3    4
```

```
>> B = [ A, A ]
```

```
B =
```

```
    1    2    1    2
    3    4    3    4
```

```
>> C = [ A; A ]
```

```
C =
```

```
    1    2
    3    4
    1    2
    3    4
```

```
>> D = [ A, 2*A; 3*A, 4*A ]
```

```
D =
```

```
    1    2    2    4
    3    4    6    8
    3    6    4    8
    9   12   12   16
```

Beachte :

im Ausdruck `2*A` werden alle Elemente der Matrix `A` mit 2 multipliziert !

```
>> a1 = [ 1, 2 ]
```

```
>> a2 = [ 3, 4 ]
```

```
>> x = [ a1; a2 ]
```

```
>> y = [ a1 a2 ]
```

Frage :

Was ist `x` ?

Was ist `y` ?

Colon-Operator : `start:step:end` (`start:step:end`)
 `start:end` (`start:end`)

Der **Colon-Operator (Doppelpunkt-Operator)** erzeugt einen **Zeilenvektor** „ von `start` bis `end` mit einer Schrittweite von `step` (Default ist 1) “ – einen Vektor, dessen Elemente einen festen Abstand besitzen.

```
>> x = 1:5
x =
     1     2     3     4     5
>> y = 1:2:10
y =
     1     3     5     7     9
>> z = 0:pi/4:pi
z =
     0  0.7854  1.5708  2.3562  3.1416
>> x = 3:5
x =
     3     4     5
>> x = 5:-2:1
x =
     5     3     1
```

Beispiele :

- a) Erzeuge einen Vektor x , der die Zahlen 100, 99, ... , 1 speichert.

- b) Erzeuge einen Vektor x mit den Zahlen von 50 bis -50 bei einer Schrittweite von 5. Geben Sie die Anzahl der Elemente von x aus.

- c) Speichere alle Winkel von 0 bis 180° (Abstand 1°) in einem Vektor. Die Winkel sollen im Bogenmaß gespeichert werden.

- d) Geben Sie das letzte Elemente des Vektors x aus.

- e) Fügen Sie dem Vektor x noch ein weiteres Element mit dem Wert 120 hinzu.

Der **Colon-Operator** wird auch verwendet, um auf Teile einer Matrix zuzugreifen.

```
>> A=[ 11,12,13,14; 21,22,23,24; 31,32,33,34 ]
```

```
A =
```

```
    11    12    13    14
    21    22    23    24
    31    32    33    34
```

: (ohne **start** und **end**) steht für „**alle Elemente**“ einer Zeile oder Spalte

```
>> x = A(2, :)
```

```
x =
```

```
    21    22    23    24
```

Der **Zeilenvektor** **x** wird aus der zweiten Zeile der Matrix **A** gebildet

```
>> y = A(:, 3)
```

```
y =
```

```
    13
    23
    33
```

Der **Spaltenvektor** **y** wird aus der dritten Spalte der Matrix **A** gebildet.

```
>> B = A(1:2, 2:4)
```

```
B =
```

```
    12    13    14
    22    23    24
```

Teilmatrix **B** wird aus den Zeilen 1 bis 2 und den Spalten 2 bis 4 der Matrix **A** gebildet.

Weitere nützliche Funktionen und Operatoren für Vektoren

`x(3:7) = 0` `x([3 4 5 6 7]) = 0` `x([3 16 23]) = 0`

Die Elemente `x(3)`, `x(4)` ... `x(7)` auf 0 setzen bzw. die Elemente `x(3)`, `x(16)` und `x(23)` gleich 0 setzen.

`x(:) = 3`

Setze alle Elemente des Vektors `x` auf 3. Der **Colon-Operator** ohne Angabe von `anfang` und `ende` bezeichnet alle Elemente. Größe von `x` unverändert.

`x = 2`

Erzeuge eine 1*1-Matrix und setze den Wert auf 2. Größe von `x` ändert sich.

`x(3:8) = []`

Die Elemente `x(3)`, `x(4)` ... `x(8)` aus dem Vektor entfernen - löschen

`x(:) = []` oder `x = []`

Alle Elemente aus dem Vektor löschen – `x` ist dann eine 0*0-Matrix

`x = linspace(anfang, ende, n)`

Erzeugt einen Zeilenvektor mit `n` Elementen. `x(1)=anfang`, `x(n)=ende`
Die anderen Werte liegen äquidistant zwischen `anfang` und `ende`.

`x = linspace(1, 2, 2)`

`x = linspace(0, pi, 181)`

`x = linspace(1, 2, 10)`

`x = linspace(1, 2, 11)`

Die **Transponierte** einer Matrix bilden : **Vertausche Zeilen und Spalten der Matrix**

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

```
A =
```

```

  1   2   3
  4   5   6
  7   8   9
```

```
>> B = A'
```

```
B =
```

```

  1   4   7
  2   5   8
  3   6   9
```

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

```
A =
```

```

  1   2   3
  4   5   6
```

```
>> B = A'
```

```
B =
```

```

  1   4
  2   5
  3   6
```

```
>> B = transpose(A)
```

$A' \equiv \text{transpose}(A)$

Der Operator **'** und die Funktion **transpose** transponieren ein 2d-Array, d.h. spiegeln die Elemente an der Hauptdiagonalen und wandeln eine **n*m**-Matrix in eine **m*n**-Matrix um.

Beachte : Bei Vektoren werden **Zeilenvektoren** in **Spaltenvektoren** umgewandelt und umgekehrt.

```
>> x=[1, 2]
```

```
x =
```

```

  1   2
```

```
>> x'
```

```
ans =
```

```

  1
  2
```

```
>> y=[2; 3]
```

```
y =
```

```

  2
  3
```

```
>> y = y'
```

```

y =
  2   3
```

Die meisten Operatoren sind sowohl für **Skalare** als auch **Arrays** definiert. Ebenso können viele Funktionen auch auf Arrays angewendet werden. Operatoren und Funktionen wirken dabei **elementweise** (**Ausnahme: Multiplikation und Division**).

```
>> A = [1,2; 3,4]
```

```
A =
```

```
    1    2
    3    4
```

```
>> sind(A)
```

```
ans =
```

```
    0.0175    0.0349
    0.0523    0.0698
```

```
>> B = 5*A + 2
```

```
B =
```

```
    7    12
   17    22
```

```
>> C = A + B
```

```
C =
```

```
    8    14
   20    26
```

```
[ sind(1)  sind(2) ]
[ sind(3)  sind(4) ]
```

Die Multiplikation eines Arrays mit einem Skalar erfolgt elementweise, ebenso die Addition eines Arrays mit einem Skalar.

$B(i,j) = 5*A(i,j) + 2$ für alle i,j

$C(i,j) = A(i,j) + B(i,j)$ für alle i,j

```
> x = [ 0 pi/6 pi/4 pi/3 ];
> y = sin(x)
```

Frage: Welche Werte besitzen die Elemente von y ?

Arrays können nur dann addiert (analoges gilt für andere Operationen, die elementweise wirken) werden, wenn die Dimensionen der Operanden übereinstimmen, außer man addiert einen **Skalar**. Skalare Größen werden stets elementweise auf Arrays angewendet.

```
>> A = [ 1, 2; 3, 4 ]
```

```
A =
```

```
    1    2
```

```
    3    4
```

```
>> B = A + 2
```

```
B =
```

```
    3    4
```

```
    5    6
```

```
>> x = [ 2, 7 ]
```

```
x =
```

```
    2    7
```

```
>> y = 3 + x
```

```
y =
```

```
    5   10
```

```
>> B = A + x(1)
```

```
B =
```

```
    3    4
```

```
    5    6
```

```
>> A + x
```

```
Error using +  
Matrix dimensions must agree.
```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}$$

$$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

$$y = [4 \ 5]$$

$$c = 5$$

Es gibt verschiedenen Arten der Multiplikation :

1) $c * A =$

2) $A * B =$

3) $A . * B =$

4) $A * x =$

5) $x * y =$

6) $y * x =$

7) $A ^ 3 =$

8) $A . ^ 3 =$

Die Multiplikation von zwei Skalaren ist einfach und eindeutig. Multipliziert man dagegen Arrays miteinander oder multipliziert man ein Array mit einem Skalar muss genau definiert werden, was man darunter versteht. MATLAB kennt verschiedene Arten der Multiplikation, die durch unterschiedliche Operatoren gekennzeichnet werden :

- **Matrixmultiplikation** : $C = A * B$ mit dem Operator `*`

„bekannte Matrixmultiplikation aus der Linearen Algebra“

Die Matrixmultiplikation ist nur für 2d-Arrays definiert und auch nur, wenn Zeilenzahl und Spaltenzahl bestimmte Bedingungen erfüllen. Das Ergebnis der Multiplikation ist wieder eine Matrix.

Ebenso : Potenz (`^`) und Division (`/` - Multiplikation mit der Inversen)

- **Elementweise Multiplikation** : $C = A .* B$ mit dem Operator `.*`

Die elementweise Multiplikation ist für beliebige Arrays definiert, aber nur, wenn die Dimensionen der beiden Arrays übereinstimmen. Es werden jeweils die entsprechenden Elemente der beiden Arrays miteinander multipliziert.

Ebenso : elementweise Potenz (`.^`) und elementweise Division (`./`)

- **Multiplikation eines Arrays mit einem Skalar**

Jedes Element des Arrays wird mit dem Skalar multipliziert. Es gibt in diesem Fall keinen Unterschied zwischen den Operatoren `*` und `.*` .

Multiplikation von Matrizen * :

Das Produkt $A*B$ der beiden Matrizen A und B ist nur dann definiert, wenn A **genauso viele Spalten besitzt wie B Zeilen, also**

- ist A eine $n * k$ -Matrix und B eine $k * m$ -Matrix, dann ist das Produkt der beiden Matrizen $C = A * B$ definiert und das Ergebnis ist eine $n * m$ -Matrix.

- die Elemente von C berechnen sich wie folgt :

$$C(p,q) = \sum_{m=1}^k A(p,m) * B(m,q)$$

d.h. das Element $C(p, q)$ ist das Skalarprodukt aus dem p -ten Zeilenvektor von A mit dem q -ten Spaltenvektor von B . Beachte : Der Zeilenvektor und der Spaltenvektor müssen die gleiche Länge besitzen (hier jeweils k Elemente) !

In dem Ausdruck $C = A * B$ führt MATLAB die Matrixmultiplikation automatisch durch. Bei der Programmierung in C sind hierfür 3 geschachtelte Schleifen notwendig.

Beachte :

- Ist das Matrixprodukt $A * B$ definiert, dann ist im allgemeinen das Produkt $B * A$ nicht definiert, außer wenn $n == m$.
- Die Matrixmultiplikation ist nicht kommutativ, d.h. im allgemeinen gilt

$$A * B \neq B * A$$

Beispiel Matrixmultiplikation :

$$C(p,q) = \sum_{m=1}^k A(p,m) * B(m,q)$$

$$C(p,q) = A(p, :) * B(:, q)$$

Eine $3 * 2$ -Matrix **A** wird mit einer $2 * 4$ -Matrix **B** multipliziert.
Das Ergebnis ist eine $3 * 4$ -Matrix **C** !

B(:, 3)

	A(1,1)	A(1,2)	B(1,1)	B(1,2)	B(1,3)	B(1,4)	C(1,1)	C(1,2)	C(1,3)	C(1,4)
A(2, :)	A(2,1)	A(2,2)	B(2,1)	B(2,2)	B(2,3)	B(2,4)	C(2,1)	C(2,2)	C(2,3)	C(2,4)
	A(3,1)	A(3,2)					C(3,1)	C(3,2)	C(3,3)	C(3,4)
	1	2	10	11	12	13	50	53		
	3	4	20	21	22	23	110			
	5	6					170			

$$C(2,3) =$$

Elementweise Multiplikation von Matrizen mit dem Operator `.*`

Zwei Arrays A und B können **elementweise** miteinander multipliziert werden, wenn die beiden Arrays die gleiche **Dimension** besitzen. Der zugehörige MATLAB-Operator hierfür ist `.*`.

Weiterhin kann jedes beliebige Array mit einer Zahl (Skalar) multipliziert werden. Hierfür werden die Operatoren `.*` oder `*` verwendet – beide Operatoren haben die gleiche Wirkung.

Elementweise Multiplikation :

$$C = A .* B$$

Beispiele :

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 10 & 11 \\ 20 & 21 \end{bmatrix}$$

$$C = A .* B = \begin{bmatrix} 10 & 22 \\ 60 & 84 \end{bmatrix}$$

$$D = 5 * A$$

$$D = 5 .* A$$

$$D = A * 5$$

$$D = A .* 5$$

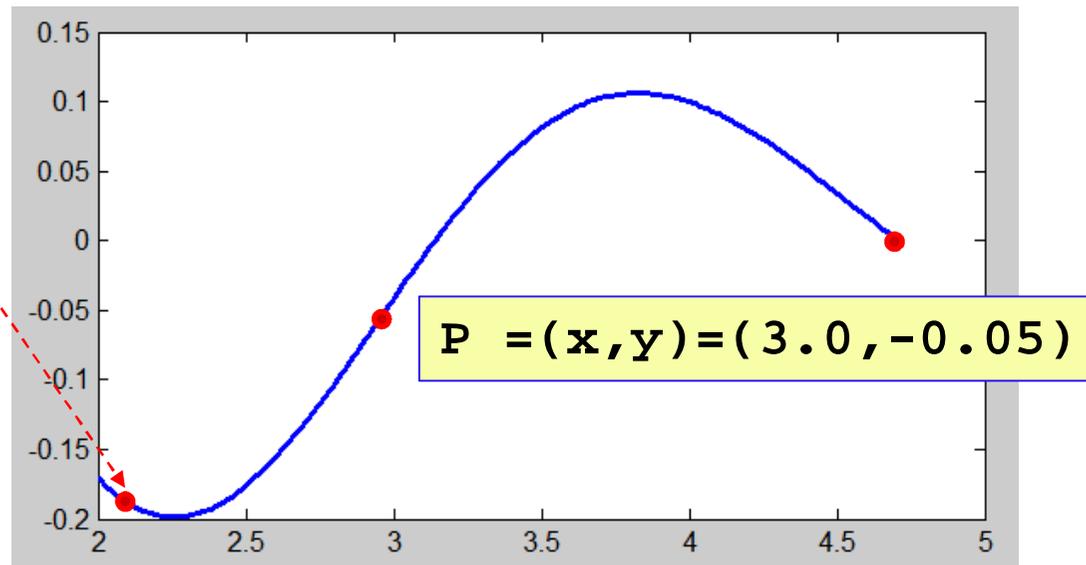
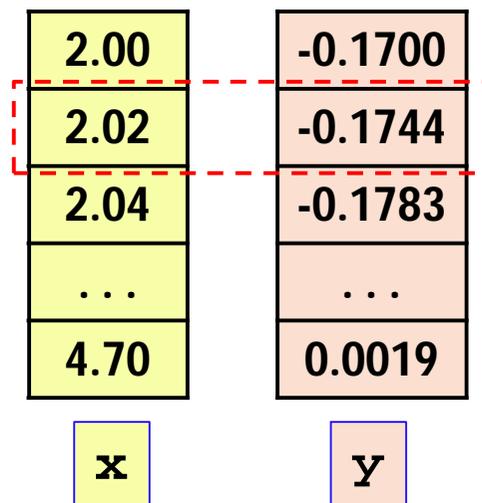


$$D = \begin{bmatrix} 5 & 10 \\ 15 & 20 \end{bmatrix}$$

Aufgabe : Zeichne eine Funktion $y = f(x)$ in einem vorgegebenen Bereich.

`plot(x, y)`

Der Funktion `plot` werden im einfachsten Fall zwei Vektoren (Zeilenvektoren oder Spaltenvektoren) mit je n Elementen als Parameter übergeben. Die Funktion `plot` zeichnet dann eine Kurve durch die Punkte $(x(k), y(k))$ von $k = 1 \dots n$.



Frage :

Wie kann man auf einfache Weise solche Vektoren erzeugen und verarbeiten?

- Teilaufgaben**
- 1) Vektor x erzeugen – alle Elemente besitzen den gleichen Abstand
 - 2) Funktionswerte $y(i)$ in Abhängigkeit von $x(i)$ berechnen

Aufgaben :

- 1. Erzeuge einen Vektor x mit äquidistanten Elementen im Bereich zwischen 2.0 und 4.7. Der Abstand aufeinanderfolgender Werte des Vektors soll 0.02 sein. (2.00, 2.02, 2.04, ... , 4.68, 4.70). Wie viele Elemente besitzt der Vektor.**
- 2. Berechne die zugehörigen Funktionswerte für folgende Funktionen :**

$$y = 2 \cdot x \quad y = x^2 \quad y = \sin(x) \quad y = e^{-x} \quad y = e^{-x^2} \quad y = x^2 \cdot \sin(1/x)$$

- 3. Hausaufgabe : Zeichnen Sie die Funktionen.**

Mathematik :

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad y = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}$$

Skalarprodukt (**dot product, inner product**) von x und y

$$s = x \cdot y = x(1) \cdot y(1) + x(2) \cdot y(2) + x(3) \cdot y(3) \\ = |x| \cdot |y| \cdot \cos(\varphi)$$

MATLAB :**x** = [1; 2; 3] 3 * 1 Array**a** = [1, 2, 3] 1 * 3 Array**y** = [4; 5; 6] 3 * 1 Array**b** = [4, 5, 6] 1 * 3 Array**Das Skalarprodukt berechnen :**

1) Mit Hilfe der Transponierten eines Vektors (Matrixmultiplikation)

$$s = x' * y \quad \text{oder} \quad s = y' * x$$

Die Transponierte eines Spaltenvektors ist ein Zeilenvektor. Multiplikation einer 1*3 Matrix mit einer 3*1 Matrix ergibt eine 1*1 Matrix (Skalar).

Sind a und b **Zeilenvektoren**, dann lautet die Formel : **s =**

2) Die Funktion **dot** verwenden

$$s = \text{dot}(x, y) \quad \text{oder} \quad s = \text{dot}(y, x)$$

Command Window

>> help dot

dot Vector dot product.

C = dot(A,B) returns the scalar product of the vectors A and B.

A and B must be vectors of the same length. When A and B are both column vectors, dot(A,B) is the same as A'*B.

Aufgabe 1

Gegeben ist der Vektor $\mathbf{x} = [1, 2, 3, 4, 5]$

Welche Wirkung haben die folgenden Befehle ?

1. $\mathbf{x}(2:3) = -1$

2. $\mathbf{x}(2:3) = 2:3$

3. $\mathbf{x}(2:3) = []$

4. $\mathbf{x}([2,4]) = [20,30]$

5. $\mathbf{x}(:) = -1$

6. $\mathbf{x}(\text{end}) = -1$

7. $\mathbf{x} = -1$

8. $\mathbf{x}(1) = -1$

9. $\mathbf{x}(1,1) = -1$

10. $\mathbf{x} = \mathbf{x}(2)$

11. $\mathbf{x} = 2:3$

12. $\mathbf{x} = \text{linspace}(0,5,5)$

13. $\mathbf{z} = \mathbf{x}' * \mathbf{x}$

14. $\mathbf{z} = \mathbf{x} * \mathbf{x}'$

15. $\mathbf{z} = \mathbf{x} * \mathbf{x}$

16. $\mathbf{z} = \mathbf{x} . * \mathbf{x}$

17. $\mathbf{z} = 1 ./ \mathbf{x}$

18. $\mathbf{z} = \mathbf{x} . ^ 2 ./ \mathbf{x}$

19. $\mathbf{z} = \mathbf{x} ./ \mathbf{x}$

20. $\mathbf{z} = (1:5) + (5:-1:1)$

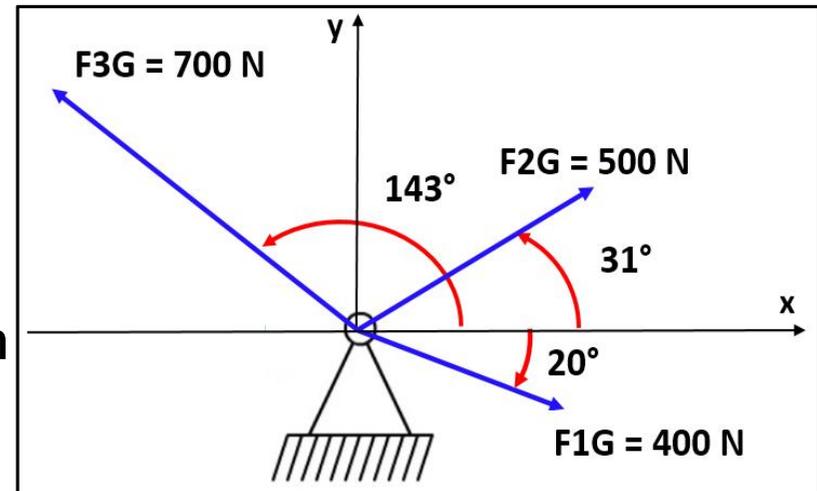
21. $\mathbf{z} = 1:5+5:1$

Überlegen Sie sich zuerst die Lösung. Geben Sie danach die Befehle ein.
Schreiben Sie dazu ein geeignetes Skript.

Aufgabe 2 :

Auf ein Festlager wirken drei Kräfte wie im Bild rechts dargestellt.

Schreiben Sie ein MATLAB-Skript zur Berechnung der Gesamtkraft F . Geben Sie den Betrag der Gesamtkraft aus. Berechnen Sie den Winkel, den die Gesamtkraft mit der x -Achse einschließt und geben Sie diesen in Grad aus.



Ein Beispiel, wie die Ausgabe aussehen könnte, finden Sie hier:

Gesamtkraft: 594.956198

Winkel: 65.638402

Ergänzen Sie das MATLAB-Skript rechts. Verwenden Sie zur Berechnung der Lösung die Variablen, die bereits definiert sind.

```
F1G = 400;  
F2G = 500;  
F3G = 700;  
w1 = -20;  
w2 = 31;  
w3 = 143;  
. . .
```

Aufgabe 3 : MATLAB besitzt eine Vielzahl von Funktionen, um spezielle Arrays (häufig Matrizen) zu erzeugen :

zeros Erzeugt ein Array, bei dem alle Elemente auf 0 gesetzt werden.

ones Erzeugt ein Array, bei dem alle Elemente auf 1 gesetzt werden.

eye Einheitsmatrix : Elemente auf der Diagonalen sind 1, alle anderen 0

diag Diagonalmatrix erzeugen – Werte auf der Diagonalen vorgeben.

rand Array mit zufälligen Werten erzeugen; die Werte der Elemente sind Zufallszahlen zwischen 0 und 1 (gleichverteilt, ohne die Grenzen 0 und 1)

randn Die Werte aller Elemente sind Zufallszahlen einer Normalverteilung

Es wird erwartet, dass Sie diese Funktionen in der Prüfung anwenden können.

```
> A = zeros(2,3)
```

```
A =
```

```
0 0 0
```

```
0 0 0
```

```
> B = ones(2)
```

```
B =
```

```
1 1
```

```
1 1
```

```
> A = eye(3)
```

```
A =
```

```
1 0 0
```

```
0 1 0
```

```
0 0 1
```

```
> B = eye(2,3)
```

```
B =
```

```
1 0 0
```

```
0 1 0
```

```
> A = diag([3,2,5])
```

```
A =
```

```
3 0 0
```

```
0 2 0
```

```
0 0 5
```

```
> B = rand(2,2)
```

```
B =
```

```
0.2695 0.4469
```

```
0.9963 0.1528
```

Erzeugen Sie die folgende Matrizen mit jeweils einem Befehl. Geben Sie die Werte der Elemente nicht explizit ein. Hinweis : Verwenden Sie die Funktionen **ones**, **zeros**, ..., den Concatentation-Operator **[]**, den Colon-Operator.

$$A = \begin{bmatrix} 6 & 8 \\ 6 & 8 \\ 6 & 8 \\ 6 & 8 \\ 6 & 8 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 8 & 6 & 4 & 2 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

```
>> help zeros
```

```
zeros Zeros array.
```

```
zeros(N) is an N-by-N matrix of zeros.
```

```
zeros(M,N) or zeros([M,N]) is an M-by-N matrix of zeros.
```

```
zeros(M,N,P,...) or zeros([M N P ...]) is an  
M-by-N-by-P-by-... array of zeros.
```

```
zeros(SIZE(A)) is the same size as A and all zeros.
```

```
zeros with no arguments is the scalar 0.
```

```
zeros(..., CLASSNAME) is an array of zeros of class  
specified by CLASSNAME.
```

```
Note: The size inputs M, N, and P... should be non-  
negative integers. Negative integers are treated as 0.
```

```
Example:
```

```
x = zeros(2,3,'int8');
```

Aufgabe 4 : Machen Sie sich mit den folgenden Funktionen vertraut.

<code>length(x)</code>	Die Anzahl der Elemente eines Vektors („Länge“) zurückgeben. Gilt für Zeilen- und Spaltenvektoren - vergleiche end
<code>[n, m] = size(A)</code> <code>k = size(A)</code> <code>p = size(x)</code>	Anzahl der Zeilen (n) und Spalten (m) der Matrix A zurückgeben Was ist die Größe k ? Welchen Wert besitzt k, wenn A eine 3*4-Matrix ist? Welchen Wert hat die Größe p, wenn x ein Zeilenvektor mit 5 Elementen ist? Was ändert sich, wenn x ein Spaltenvektor mit 5 Elementen ist?
<code>mean(x)</code>	Ist x ein Vektor (Zeilen oder Spaltenvektor), dann wird der Mittelwert (der Summe der Elemente) des Vektors x zurückgegeben.
<code>mean(A)</code> <code>mean(A, 1)</code>	Ist A eine Matrix , dann werden die Mittelwerte der einzelnen Spalten von A berechnet und in einem Zeilenvektor gespeichert. Der Zeilenvektor wird als Ergebnis zurückgegeben.
<code>mean(A, 2)</code>	Die Mittelwerte der einzelnen Zeilen der Matrix A werden berechnet und in einem Spaltenvektor gespeichert. Der Spaltenvektor wird als Ergebnis zurückgegeben.

Frage : Was ergeben folgende Anweisungen?

```
x = [ 2 ; 4 ; 9 ] ;
mean(x)
mean(x, 2)
```

<code>sum(x)</code>	Ist x ein Vektor (Zeilen oder Spaltenvektor), dann wird die Summe der Elemente von x berechnet und zurückgegeben.
<code>sum(A)</code> <code>sum(A, 1)</code>	Die Summe der einzelnen Spalten einer Matrix A berechnen und das Ergebnis als Zeilenvektor zurückgeben.
<code>sum(A, 2)</code>	Summe der Zeilen einer Matrix A zurückgeben - Spaltenvektor
<code>max(x)</code>	Ist x ein Vektor , dann den Wert des größten Elements zurückgeben. Ist x eine Matrix , dann werden die Maxima der einzelnen Spalten berechnet und das Ergebnis wird als Zeilenvektor zurückgegeben.
<code>max(x, y)</code>	Eine Matrix mit den Maxima von x und y zurückgeben (Maxima jeweils elementweise berechnen) - x oder y können auch Skalare sein
<code>max(x, [], 2)</code>	Ist x eine Matrix, dann gibt max einen Spaltenvektor zurück, der die Maxima der Elemente der einzelnen Zeilen der Matrix enthält
<code>sort(x)</code>	Die Elemente eines Vektors x aufsteigend sortieren; bei Matrizen werden die Spalten aufsteigend sortiert.
<code>sort(x, DIM, MODE)</code>	MODE = 'ascend' (aufsteigend) 'descend' (absteigend sortiert) DIM = 1 spaltenweise, DIM = 2 zeilenweise sortieren

Frage : Wie wird das größte Element einer Matrix A berechnet ?

Was berechnet der nebenstehende Ausdruck?

Wie kann der Ausdruck vereinfacht werden?

`max(max(A, [], 2))`

Kapitel 3 – M-Files : Skripte und Funktionen

3.1 MATLAB-Skripte

3.2 Funktionen

3.3 Beispiele für Funktionen

3.4 Subfunctions

3.5 Hausaufgaben

Um in MATLAB zu programmieren gibt zwei Möglichkeiten.

- **MATLAB-Skript**
- **MATLAB-Funktion**

In beiden Fällen werden die MATLAB-Befehle in einem sogenannten **M-File** gespeichert - eine Text-Datei (ASCII-Datei) mit der Endung **.m**. Diese Befehle können danach ausgeführt werden.

MATLAB-Skript

- Für ein Skript gibt es keine Entsprechung in der Programmiersprache C
- Ein Skript besitzt keine Parameter und keine Rückgabewerte; wird nicht durch das Schlüsselwort **function** eingeleitet
- Ein Skript kann direkt auf Variablen des MATLAB-Workspace zugreifen

MATLAB-Funktion

- Besitzt ähnliche Eigenschaften wie eine Funktion in der Programmiersprache C
- Besitzt Parameter und Rückgabewerte; wird durch das Schlüsselwort **function** eingeleitet
- Besitzt eigene lokale Variablen und kann nicht direkt auf die Variablen des MATLAB-Workspace zugreifen

Ein MATLAB-Skript enthält eine Folge von MATLAB-Befehlen.

Wird ein Skript ausgeführt, dann werden die Befehle im Skript nacheinander abgearbeitet, genauso als würde man die Befehle direkt im Command-Window nacheinander eingeben. Ein Skript speichert also MATLAB-Befehle, die hintereinander ausgeführt werden.

Ein Skript wird in einer ASCII-Datei gespeichert, deren Name auf **.m** endet.

Ein Skript kann auf verschiedene Arten gestartet/aufgerufen/ausgeführt werden :

- Name des Skripts im Command-Window eingeben
- Skript im Current-Folder-Window selektieren und in das Command-Window ziehen oder Kontext-Menü öffnen und Menüpunkt **Run** wählen
- Über das Menü oder die Toolbar des Editors – **Run** wählen
- Aufruf eines Skripts in einem anderen Skript

Wichtig : Ein Skript hat Zugriff auf den aktuellen MATLAB-Workspace !

Ein MATLAB-Skript darf alle Variablen verwenden, die im aktuellen Workspace vorhanden sind (d.h. während des Ablaufs des Skripts). Ein Skript kann Variablen aus dem Workspace lesen, verändern oder löschen oder auch neue Variablen erzeugen. Wird ein Skript zweimal hintereinander ausgeführt, können die Ergebnisse unterschiedlich sein, da sich der Workspace verändert haben kann.

The screenshot shows the MATLAB R2016b interface. The Editor window displays the script `quadrG1Scr.m` with the following code:

```
1 clear
2 % löse die Gleichung
3 % 3*x*x + 9*x + 5 = 0
4 a = 3
5 b = 9
6 c = 5
7 diskr = sqrt(b*b-4*a*c)
8 x1 = (-b + diskr)/(2*a)
9 x2 = (-b - diskr)/(2*a)
```

The Workspace window shows the following variables and their values:

Name	Value
x2	-2.2638
x1	-0.7362
diskr	4.5826
c	5
b	9
a	3

The Command Window shows the output of running the script:

```
>> quadrG1Scr
a =
    3
b =
    9
c =
    5
diskr =
    4.5826
x1 =
   -0.7362
x2 =
   -2.2638
```

Annotations in the image:

- m-File**: Points to the `quadrG1Scr.m` file in the Editor.
- Aufruf des Skripts**: Points to the `>> quadrG1Scr` command in the Command Window.
- Ausgaben bei der Ausführung des Skripts ein ; am Ende jeder Anweisung würde die Ausgaben unterdrücken**: Points to the semicolon at the end of each line in the script.
- Diese Variablen werden vom Skript bei der Ausführung erzeugt und können danach weiter verwendet werden.**: Points to the variables in the Workspace.
- Der Befehl `clear` löscht alle Variablen aus dem Workspace -> Skript läuft immer gleich ab.**: Points to the `clear` command in the script.

Kapitel 3 – M-Files : Skripte und Funktionen

3.1 MATLAB-Skripte

3.2 Funktionen

3.2.1 Funktionen in C – Funktionen in MATLAB

3.2.2 Aufbau einer Funktion

3.3.3 Funktionsnamen

3.2.4 Dokumentation einer Funktion

3.3 Beispiele für Funktionen

3.4 Subfunctions

3.5 Hausaufgaben

Schreibe eine Funktion zur Berechnung des Kreisumfangs – Rufe die Funktion auf

```
#include <stdio.h>
#include <math.h>

double umfang(double r);

int main(void)
{
    double umf, r=3.0;
    umf = umfang(r);
    printf("Umfang=%f\n", umf);
    return 0;
}

double umfang(double r)
{
    double erg;
    erg = 2 * M_PI * r;
    return erg;
}
```

C-Programm

- Die Funktion **umfang** besitzt einen Parameter vom Typ `double` und gibt einen Wert vom Typ `double` zurück.
- Die Typen der **Parameter** und der Typ des **Rückgabewertes** werden im Funktionskopf festgelegt.
- Eine Funktion wird von der `main`-Funktion oder einer anderen Funktion aufgerufen.
- Variablen, die innerhalb einer Funktion oder im Funktionskopf definiert werden, sind **lokale** Variablen. Diese Variablen sind nur innerhalb der Funktion sichtbar.
- Die Variablen `r` in den Funktionen `main` und `umfang` sind zwei verschiedene Variablen, die nur den gleichen Namen besitzen.

Mehrere Rückgabewerte in einem **C-Programm** - Umfang und Fläche
In C müssen hierzu Zeiger oder globale Variablen verwendet werden !

```
#include <stdio.h>
#include <math.h>
void kreis(double r, double *zumf, double *zfl);
int main(void)
{
    double umf, fl;
    kreis(3.0, &umf, &fl);
    printf("Umfang = %f\n", umf);
    printf("Fläche = %f\n", &fl);
    return 0;
}
void kreis(double r, double *zumf, double *zfl)
{
    *zumf = 2 * M_PI * r;
    *zfl = M_PI * r * r;
}
```



```
function [out1, out2, ...] = Funktionsname(in1, in2, ...)  
% Beschreibung - Hilfetext  
Anweisungen  
end
```

Eine Funktion wird in einer Datei gespeichert, die den gleichen Namen besitzt wie die Funktion selbst. Als Dateiendung muss `.m` verwendet werden.

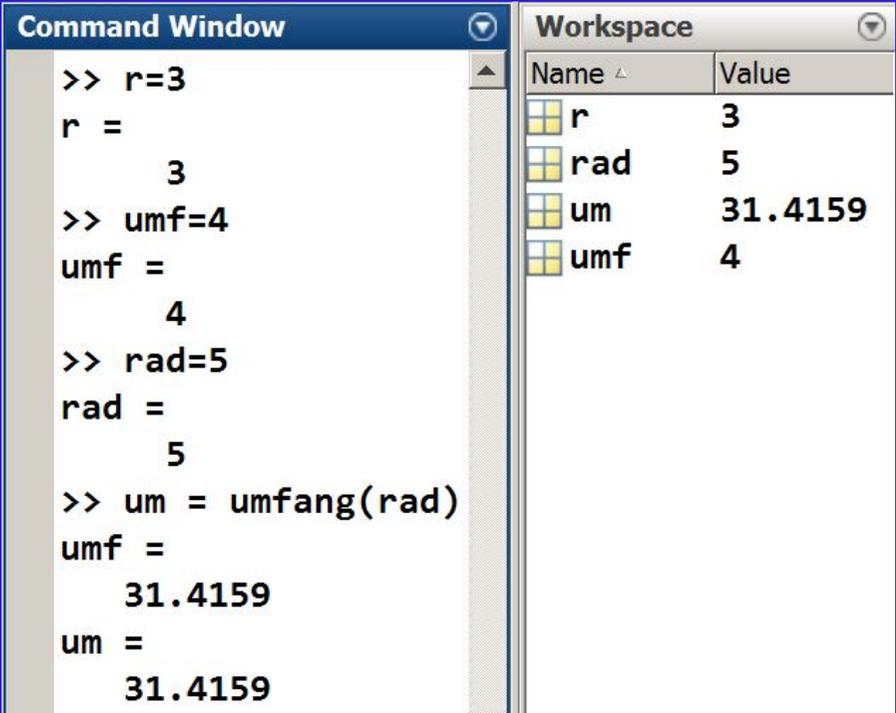
Eine Funktion `euler` muss also in der Datei `euler.m` gespeichert werden.

Eine Funktion kann

- keinen, einen oder mehrere Parameter besitzen
(hier `in1, in2, . . .` genannt)
- keinen, einen oder mehrere Rückgabewerte oder Ergebnisse besitzen
(hier `out1, out2, . . .` genannt)

Wichtig : Parameter und Rückgabewerte haben keine Typinformationen
Eine Funktion hat keinen direkten Zugriff auf den Workspace

Eine MATLAB-Funktion kann deshalb keine Variablen aus dem Workspace direkt verändern. Eine Funktion besitzt einen „eigenen Workspace“.



The screenshot shows the MATLAB Command Window and Workspace. The Command Window contains the following commands and outputs:

```
>> r=3
r =
    3
>> umf=4
umf =
    4
>> rad=5
rad =
    5
>> um = umfang(rad)
umf =
    31.4159
um =
    31.4159
```

The Workspace window shows the following variables and their values:

Name	Value
r	3
rad	5
um	31.4159
umf	4

```
function umf = umfang( r )
    umf = 2*pi*r
end
```

**Beide Funktionen arbeiten
vollkommen identisch!**

```
function a = umfang( b )
    a = 2*pi*b
end
```

Die Funktion **umfang** besitzt eigene **lokale** Variablen (eigenen Workspace). Diese Variablen haben nichts mit den Variablen aus dem ‚normalen‘ MATLAB-Workspace zu tun. Nur über Funktionsparameter und Rückgabewerte können Werte zwischen den beiden Workspaces ausgetauscht werden.

Werden die Variablen in der Funktion umfang umbenannt (hier **r -> b** und **umf->a**), ändert sich an der Funktion umfang überhaupt nichts!

Parameter, die bei der Definition einer Funktion in einem M-File im Funktionskopf angegeben werden, bezeichnet man als „Formalparameter“. Diese Parameter sind zunächst nur Platzhalter und haben keine definierten Werte. Die Parameter beschreiben, wie eine Funktion aufgerufen werden muss. Erst wenn eine Funktion aufgerufen wird, werden an diese Variablen konkrete Werte zugewiesen (Aktualparameter).

Funktionen haben eine eigene Arbeitsumgebung und besitzen keinen **direkten Zugriff auf den Workspace, der im Command-Window verwendet wird**. Insbesondere sind die Variablen aus dem Workspace nicht sichtbar. Variablen aus dem MATLAB-Workspace können von einer Funktion aus nicht direkt verändert werden.

Parameter, Rückgabewerte und Variablen innerhalb einer Funktion sind **lokale Variablen**, die nur innerhalb der Funktion bekannt sind. Variablen mit gleichen Namen, die in anderen Funktionen oder im Workspace definiert sind, haben mit nichts miteinander zu tun.

Beim Aufruf einer Funktion werden keine Variablen übergeben - es werden nur die **Werte kopiert**. Bei einem Funktionsaufruf werden den Formalparametern aktuelle Werte zugewiesen. Mit diesen Werten wird die Funktion dann abgearbeitet.

Für **Funktionsnamen** gelten die gleichen Regeln wie für **Variablennamen**

- Ein Name beginnt mit einem Buchstaben; danach können **Buchstaben**, **Ziffern** oder der **Unterstrich** `_` folgen; keine Umlaute verwenden !
- Bis zu einer Länge von 31 Zeichen sind Namen eindeutig
- MATLAB unterscheidet Groß- und Kleinschreibung

Konvention : für Funktionsnamen werden nur Kleinbuchstaben verwendet

Beachte :

Eine Funktion kann mehrmals (unterschiedlich) definiert sein; ebenso kann es eine Variable im Workspace geben, die den gleichen Namen wie eine Funktion besitzt (siehe Calling Functions) -> **Auflösungsproblematik**

"Before assuming that a name should match a function, MATLAB checks the current workspace to see if it matches a variable name. If MATLAB finds a match, it stops the search."

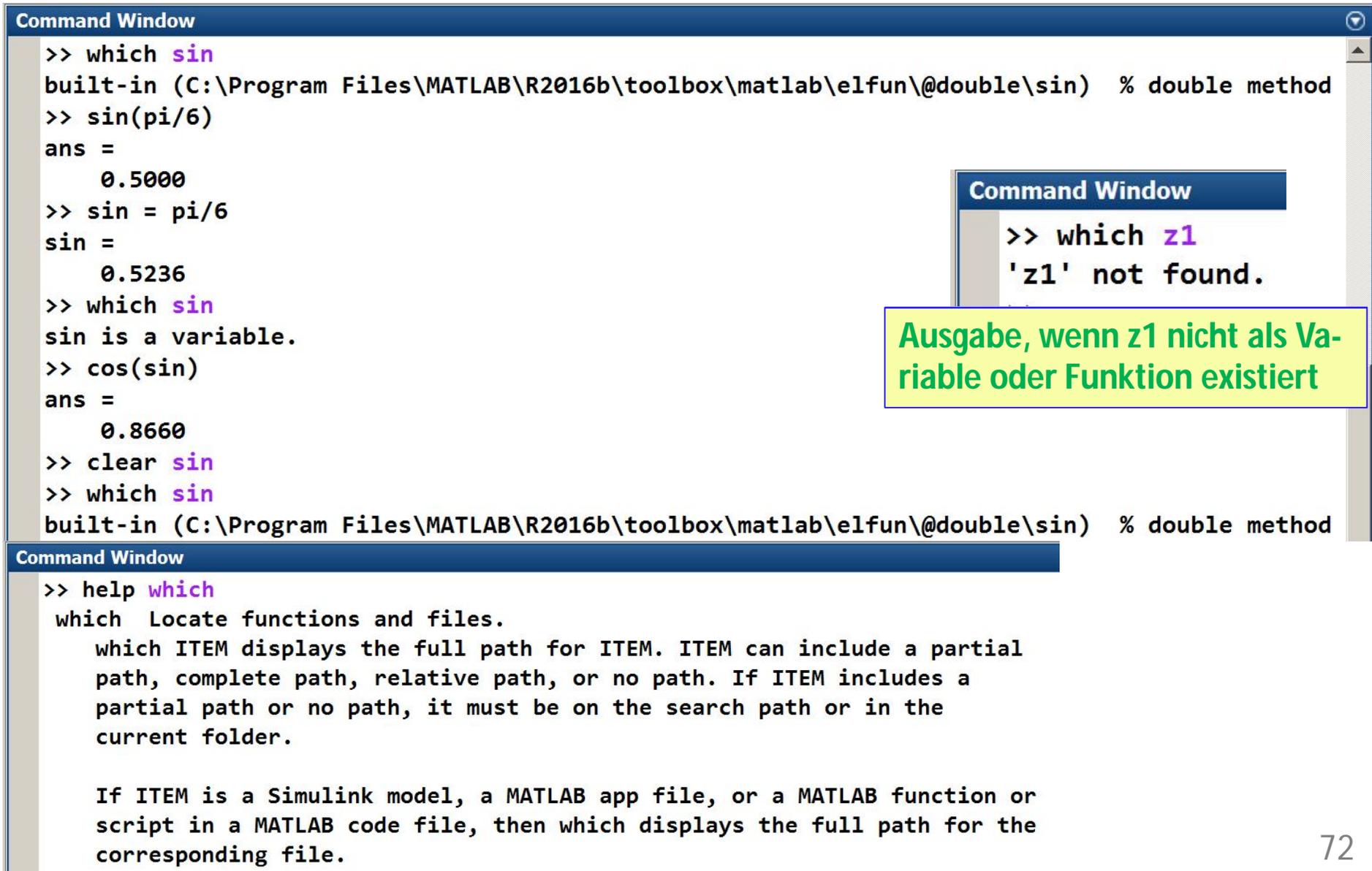
Auflösungsreihenfolge :

Variable -> Nested Functions -> Sub-Functions -> Private Functions

Namen von Funktionen müssen sehr sorgfältig gewählt werden!

Frage : Warum sollte man keine Variable mit dem Namen `sin` definieren?
Warum sollte man einem m-File nicht den Namen `exp.m` geben ?

Der Befehl **which** zeigt an, ob ein Name bekannt ist (verwendet wird) und ob es sich um eine Variable oder eine Funktion handelt.



```
Command Window
>> which sin
built-in (C:\Program Files\MATLAB\R2016b\toolbox\matlab\elfun\@double\sin) % double method
>> sin(pi/6)
ans =
    0.5000
>> sin = pi/6
sin =
    0.5236
>> which sin
sin is a variable.
>> cos(sin)
ans =
    0.8660
>> clear sin
>> which sin
built-in (C:\Program Files\MATLAB\R2016b\toolbox\matlab\elfun\@double\sin) % double method

Command Window
>> which z1
'z1' not found.

Command Window
>> help which
which Locate functions and files.
which ITEM displays the full path for ITEM. ITEM can include a partial
path, complete path, relative path, or no path. If ITEM includes a
partial path or no path, it must be on the search path or in the
current folder.

If ITEM is a Simulink model, a MATLAB app file, or a MATLAB function or
script in a MATLAB code file, then which displays the full path for the
corresponding file.
```

Ausgabe, wenn z1 nicht als Variable oder Funktion existiert

Die Beschreibung (Hilfe-Text) einer Funktion wird entweder in den Zeilen vor oder nach dem Schlüsselwort `function` angegeben. Mehrere Zeilen sind erlaubt. Alle Zeilen müssen mit `%` beginnen.

```
help funktionsname
```

```
% Die Funktion ...  
%  
function
```

```
function  
% Die Funktion ...  
%
```

Der Befehl `help` zeigt die Beschreibung einer Funktion an, genauer alle Zeilen, die mit `%` beginnen, bis zur ersten Leerzeile oder der ersten Anweisung.

Die Beschreibung einer Funktion bei MATLAB ist sehr wichtig und sollte mindestens folgende Dinge beinhalten

- Kurzbeschreibung der Funktion
- Beschreibung aller Parameter und Rückgabewerte
- Beispiel für den Aufruf der Funktion

Funktionen, die nicht ausführlich und eindeutig und klar dokumentiert sind, sind völlig wertlos. Bei MATLAB ist das wichtiger als bei der Programmiersprache C, weil die Parameter und Rückgabewerte nur Namen aber keine Typinformationen besitzen. Insbesondere erkennt man nicht, ob ein Parameter ein Skalar oder ein Vektor oder eine Matrix sein muss.

Schreibe eine Funktion, die den Umfang eines Kreises mit Radius r berechnet.

```
function [umf] = umfang( r )
%UMFANG : Kreisumfang berechnen
% umf = umfang( r ) , r:Radius
    umf = 2*pi*r;
end
```

```
function umf = umfang1( r )
    u = 2*pi*r;
end
```

Command Window

```
>> r = 5
r =
     5
>> u = umfang(r)
u =
  31.4159
>> help umfang
umfang : Kreisumfang berechnen
    umf = umfang(r),r:Radius

>> u = umfang([ 2 3 4 ])
u =
  12.5664  18.8496  25.1327
fx >> u = umfang([1 2; 3 4])
```

Command Window

```
>> r = 5
r =
     5
>> u = umfang1(r)
Output argument "umf" (and maybe others)
not assigned during call to "umfang1".
```

Beachte : Funktionen sollten, wenn möglich immer so geschrieben werden, dass beliebige Arrays als Parameter verwendet werden können!

Schreibe eine Funktion, die die Fläche eines Kreises berechnet.

```
function [ fl ] = flaeche( r )  
%FLAECHE : Kreisfläche berechnen  
% [ fl ] = flaeche( r ), r: Radius fl: Kreisfläche  
    fl = pi*r*r;  
end
```

```
Command Window  
>> r = 5  
r =  
    5  
>> fl=flaeche(r)  
fl =  
    78.5398  
>> R = [2 3; 4 2]  
R =  
     2     3  
     4     2  
>> fl=flaeche(R)  
fl =  
    50.2655    37.6991  
    50.2655    50.2655  
fx >> fl=flaeche([2 3])
```

Wenn an eine Funktion Parameter übergeben werden, dann werden nur **Kopien der Variablen** (Parameter) an die Funktion übergeben (**Call by Value**). **Die Funktion arbeitet nur mit den Kopien**. Diese Tatsache hat eine Reihe von Konsequenzen. Beispiel : Funktion f1 ändert nur die Werte in der Kopie des Aufrufparameters. Funktion f2 ändert die Werte der Kopie und gibt die Kopie aber zurück.

```
function f1( x )
    x = 2*x
end
```

```
function y = f2( x )
    x = 2*x;
    y = x;
end
```

```
a = [ 1 2 ]
f1( a )
a
```

f1scr.m

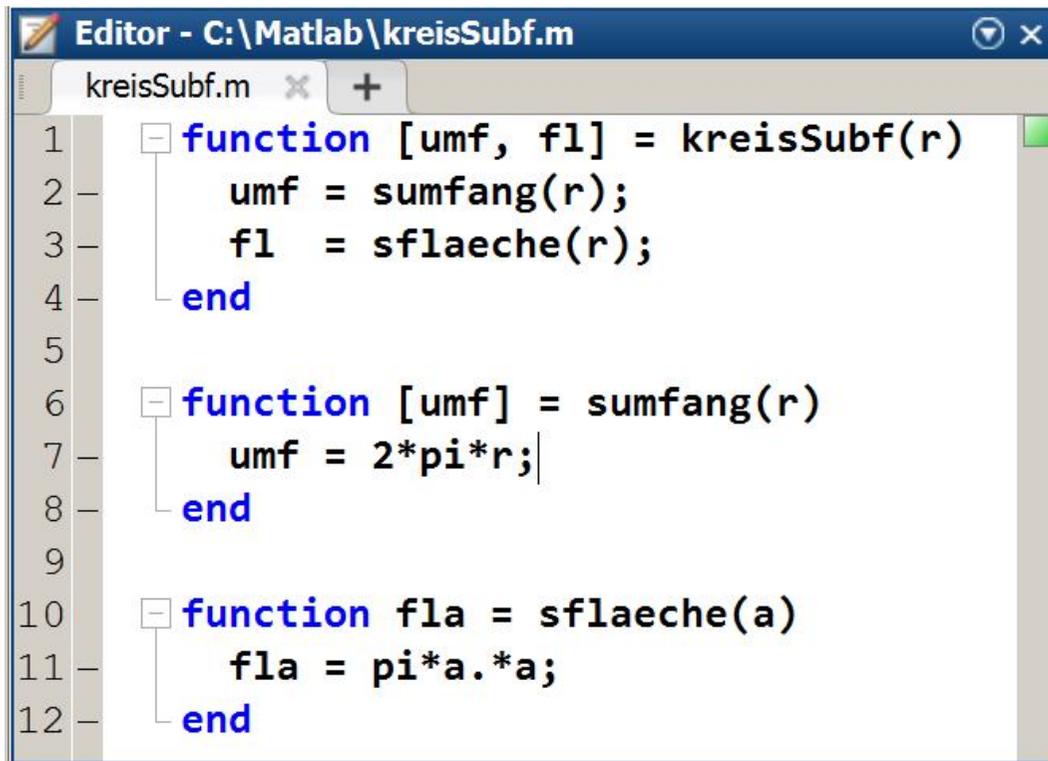
```
>> f1scr
a =
     1     2
x =
     2     4
a =
     1     2
```

```
a = [ 1 2 ]
b = f2( a )
a
c = [ 3 4 ]
c = f2( c )
```

f2scr.m

```
>> f2scr
a =
     1     2
b =
     2     4
a =
     1     2
c =
     3     4
c =
     6     8
```

Ein M-File kann mehrere Funktionen enthalten. Aber nur die Funktion, die mit dem Dateinamen übereinstimmt (**Primary Function**), kann von außerhalb aufgerufen werden. Die anderen Funktionen (**Subfunctions**) können nicht von außen aufgerufen werden. Subfunctions werden nur von der Primary Function oder anderen Subfunctions des gleichen M-Files aufgerufen.



```
Editor - C:\Matlab\kreisSubf.m
kreisSubf.m x +
1 function [umf, fl] = kreisSubf(r)
2     umf = sumfang(r);
3     fl = sflaeche(r);
4 end
5
6 function [umf] = sumfang(r)
7     umf = 2*pi*r;
8 end
9
10 function fla = sflaeche(a)
11     fla = pi*a.*a;
12 end
```

Primary Function **kreisSubf**

Funktions- und Dateiname sind
identisch!!

Diese Funktion kann im Command-
Window aufgerufen werden.

Subfunction **sumfang**

Diese Funktion kann nicht im Com-
mand-Window aufgerufen werden.

Subfunction **sflaeche**

Aufgabe 1

- a) Schreiben Sie eine Funktion **kugel**, die das Volumen einer Kugel mit Radius r und deren Oberfläche berechnet und zurückgibt. Schreiben Sie die Funktion so, dass man beliebige Arrays übergeben kann.
Erstellen Sie ein Skript und rufen Sie die Funktion **kugel** auf. Übergeben Sie einmal einen Skalar, einen Zeilenvektor, einen Spaltenvektor und ein 2×3 Array an die Funktion.
- b) Schreiben Sie eine Funktion **radius**, an die eine Kreisfläche übergeben wird und die den zugehörigen Radius berechnet und zurückgibt. Schreiben Sie die Funktion so, dass man beliebige Arrays übergeben kann.
Testen Sie Ihre Funktion mit einem geeigneten Skript und prüfen Sie, ob die berechneten Radien tatsächlich korrekt sind. **Wann ist ein berechneter Radius korrekt und wann nicht?** Versuchen Sie ein geeignetes Kriterium zu finden. Wie kann man einfach testen, ob alle Radien korrekt sind, wenn man z.B. ein 2×3 Array mit Flächen übergibt oder ein 200×300 Array.

K3_A2 3.5 Hausaufgaben

Aufgabe 2

Versuchen Sie die folgenden MATLAB-Anweisungen zu verstehen. Überlegen Sie sich was das Ergebnis der jeweils letzten Anweisung ist. Geben Sie dann die Befehle ein.

Geben Sie am Ende den Befehle **which sin** und **which sind** ein. Was muss man machen, damit man wieder auf die sin-Funktion zugreifen kann?

```
Command Window
>> x1 = sin(pi/6)
x1 =
    0.5000
>> sin = 1
sin =
    1
>> sin(2) = sin(1)
sin =
    1    1
>> x2 = sin(pi/4)
```

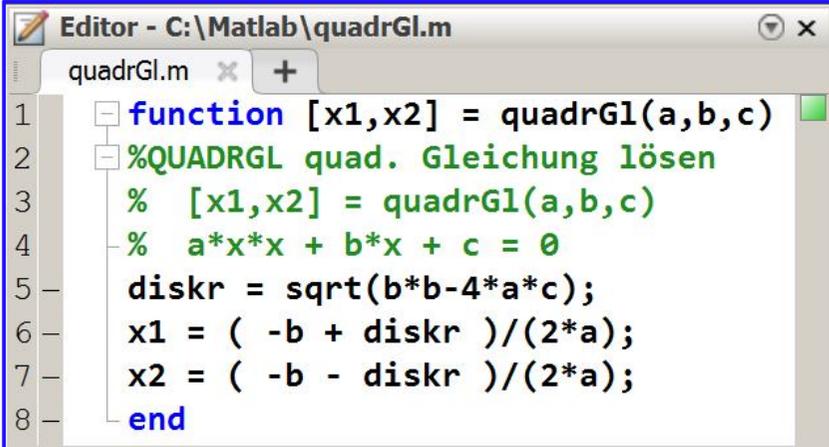
```
Command Window
>> x1 = sind(30)
x1 =
    0.5000
>> sind = 5
sind =
    5
>> sind(sind) = 3
```

Aufgabe 3

Die nebenstehende Funktion `quadrG1` berechnet die beiden Nullstellen einer quadratischen Funktion. Schreiben Sie die Funktion so um, dass man beliebige Arrays übergeben kann, d.h. a , b und c dürfen auch $n \times m$ -Matrizen sein.

Es sollen zwei Arrays zurückgegeben werden, die jeweils die $+$ (x_1) und die $-$ (x_2) Lösungen enthalten.

- Schreiben Sie ein Skript, das die Funktion `quadrG1` aufruft. Testen Sie ihr Programm mit Skalaren, Vektoren und einem 2×3 -Array.
- Überprüfen Sie für jedes ihrer Beispiele, ob die Nullstellen tatsächlich Nullstellen, d.h. setzen Sie alle Nullstellen in die entsprechenden quadratischen Funktionen ein.
- Testen Sie Ihr Programm auch für sehr viele Gleichungen, z.B. 1000 000 oder 100 000 000. Verwenden Sie hierzu die Funktion `rand` um große Arrays zu erzeugen.
- Mit den MATLAB-Befehlen `tic` und `toc` können Sie ausgeben, wie lange eine Berechnung dauert.



```

Editor - C:\Matlab\quadrG1.m
quadrG1.m x +
1 function [x1,x2] = quadrG1(a,b,c)
2 %QUADRGL quad. Gleichung lösen
3 % [x1,x2] = quadrG1(a,b,c)
4 % a*x*x + b*x + c = 0
5   disk = sqrt(b*b-4*a*c);
6   x1 = ( -b + disk )/(2*a);
7   x2 = ( -b - disk )/(2*a);
8   end
  
```

```

tic
[x1, x2] = quadrG1(a,b,c);
toc
  
```

Ausgabe, wenn a , b und c jeweils 10000×10000 Matrizen sind – 100 000 000 Gleichungen!

Elapsed time is 0.869365 seconds.

Aufgabe 4

Eine gedämpfte Schwingung kann in einfachen Fällen durch folgende DGL beschrieben werden (siehe Wikipedia).

$$\ddot{x}(t) + 2 \cdot \delta \cdot \dot{x}(t) + \omega_0^2 \cdot x(t) = 0$$

Die Abklingkonstante δ bestimmt die Stärke der Dämpfung. ω_0 ist dabei die Kreisfrequenz des ungedämpften Systems. Durch die Dämpfung ändert sich die Kreisfrequenz zu ω_δ . Es gilt :

$$\omega_\delta = \sqrt{\omega_0^2 - \delta^2}$$

Die Lösung der DGL für das Anfangswertproblem $x(t=0)=x_0$ und $v(t=0)=v_0$ lautet:

$$x(t) = e^{-\delta \cdot t} \cdot \left(\frac{v_0 + \delta \cdot x_0}{\omega_\delta} \cdot \sin(\omega_\delta \cdot t) + x_0 \cdot \cos(\omega_\delta \cdot t) \right)$$

Schreiben Sie eine Funktion **gedschwingung**, die die Auslenkung, Geschwindigkeit und Beschleunigung für eine gedämpfte Schwingung berechnet und zurückgibt. An die Funktion werden die beiden Intervallgrenzen für den Zeitraum, die Anzahl der Zeitpunkte, die Anfangsbedingungen, die Frequenz der ungedämpften Schwingung und die Abklingkonstante als Parameter übergeben. Die Funktion gibt vier Vektoren t , x , v und a zurück.

function [t, x, v, a] = **gedschw**ingung(time, n, xv, w0, delta)

Schreiben Sie ein Skript **gedschwScr**, das die Funktion **gedschwingung** aufruft und anschließend x , v und a als Funktion der Zeit zeichnet.

```
[t, x, v, a] = gedschwingung( [0, 30], 4000, [-2, 0.0], 1.2, 0.2 );
plot(t, x, t, v, t, a)
```

Kapitel 4 – Kontrollstrukturen und Bedingungen

4.1 Bedingungen

4.2 Logische Verknüpfungen

4.3 Kontrollstrukturen

4.3.1 Alternativen

4.3.2 Schleifen

4.3.3 Sprunganweisungen

4.4 Hausaufgaben

In MATLAB gibt es im wesentlichen die gleichen Kontrollstrukturen wie in der Programmiersprache C.

- Sequenz - Folge
- Verzweigung – Alternative – Auswahl
- Schleife – Iteration – Wiederholung
- Funktionen

Weiteres wichtiges Element bei der Anwendung von Kontrollstrukturen sind **Bedingungen** (**Vergleiche**) und deren Verknüpfung durch **logische Operatoren** (**logische oder boolsche Ausdrücke**).

Beachte :

- Bei MATLAB sind viele Operatoren und Funktionen auch für Arrays definiert.
- Es gibt spezielle Operatoren, um auf Zeilen oder Spalten oder Teilbereiche eines Arrays zuzugreifen.
- Aufgaben, die in der Programmiersprache C eine Schleife erfordern, können in MATLAB durch einen „einfachen Ausdruck“ programmiert werden.

Skalarprodukt : $\mathbf{x}' * \mathbf{x}$ $\mathbf{y}(1:3)' * \mathbf{x}(4:6)$

Matrix-Vektormultiplikation : $\mathbf{y} = \mathbf{A} * \mathbf{x}$

Ein logischer Ausdruck (z.B. ein Vergleich von zwei Werten des Typs `double`) erlaubt Entscheidungen (Ja/Nein). Das Ergebnis nach der Auswertung eines logischen Ausdrucks ist ein Wert vom Typ `logical`. Eine Variable dieses Datentyps besitzt entweder den Wert `true` oder `false`.

Vergleichsoperatoren bei MATLAB

<code><</code>	kleiner	<code><=</code>	kleiner gleich
<code>></code>	größer	<code>>=</code>	größer gleich
<code>==</code>	gleich	<code>~=</code>	ungleich (in C <code>!=</code>)

Beispiele :

<code>a < b</code>	Ist a kleiner als b ?
<code>a == b</code>	Besitzen a und b den gleichen Wert?
<code>a ~= b</code>	Besitzen a und b unterschiedliche Werte?

Beachte :

Die Prüfung auf Gleichheit oder Ungleichheit macht bei Gleitkommazahlen häufig keinen Sinn (Rundungsfehler) und führt manchmal zu unerwarteten Ergebnissen oder Fehlern !

```
Command Window
>> a = 3
a =
    3
>> b = 5
b =
    5
>> x = a < b
x =
    1
>> whos
  Name      Size      Bytes      Class
  a         1x1         8         double
  b         1x1         8         double
  x         1x1         1         logical
>> x = false
x =
    0
fx >> x = 1
```

Die Variable x ist das Ergebnis eines logischen Ausdrucks (hier ein Vergleich). Variable x ist daher vom Typ **logical**.
true wird als **1** ausgegeben
false wird als **0** ausgegeben
Bei der direkten Zuweisung an eine Variable vom Typ **logical** muss **true** oder **false** verwendet werden.

Welchen Typ besitzt die Variable nach dieser Anweisung?

Beachte : **Vergleichsoperatoren sind auch für Arrays definiert!**

Beispiel :

A und B sind zwei 2*2-Matrizen.

Was bedeutet $A < B$? Was ist das Ergebnis von $A < B$?

```
Command Window
>> A = [ 1 2; 3 4]
A =
     1     2
     3     4
>> B = [ 1 3; 4 -1]
B =
     1     3
     4    -1
>> C = A < B
C =
     0     1
     1     0
>> whos C
Name Size Bytes Class
C      2x2     4 logical
```

Sind die Operanden bei einem Vergleich Arrays gleicher Größe, dann wird der Ausdruck **elementweise** ausgewertet. Das Ergebnis ist ein Array und es besitzt die gleiche Größe wie die Operanden. Die Elemente des Arrays sind vom Typ **logical**.

In der Programmiersprache C müsste man für diese Aufgabe zwei geschachtelte Schleifen programmieren.

Zur Verknüpfung boolescher Werte existieren fünf **logische Operatoren**

&&	UND	&	UND
 	ODER	 	ODER
~	NICHT – Negation (in C !)		

Beispiele :

$(a > b) \ \&\& \ (b \neq 5)$	a größer als b UND b ungleich 5
$(0 < a) \ \&\& \ (a < 1)$	liegt a zwischen 0 UND 1, ohne die Grenzen
$(0 \leq a) \ \&\& \ (a \leq 1)$	liegt a zwischen 0 UND 1, mit den Grenzen
$(a < 0) \ \ (a > 1)$	a ist kleiner als 0 ODER größer als 1
$\sim (a < 0)$	NICHT a kleiner als 0

Beachte :

&& und **||** werden von links nach rechts ausgewertet und sind **short-circuit**
 Die Operatoren **&** und **|** werten einen Ausdruck vollständig aus, d.h. es wird immer die linke und die rechte Seite ausgewertet.

Beispiele :

$(a > 3) \ \&\& \ (1 == starte())$	$(1 == starte()) \ \&\& \ (a > 3)$
$(a > 3) \ \& \ (1 == starte())$	$(1 == starte()) \ \& \ (a > 3)$

In MATLAB gibt es im Wesentlichen die gleichen Kontrollstrukturen wie in der Programmiersprache C. Die **Syntax** (Schreibweise) der Kontrollstrukturen in MATLAB unterscheidet sich aber von der Syntax in der Programmiersprache C. Die **Semantik** (Bedeutung) ist praktisch identisch.

4.3.1 Alternativen

if-else-Anweisung

bedingte Anweisung – einfaches **if**

verschachtelte **if-else**-Anweisung - Mehrfachauswahl

switch-Anweisung

4.3.2 Schleifen - Wiederholungen

for-Schleife

while-Schleife

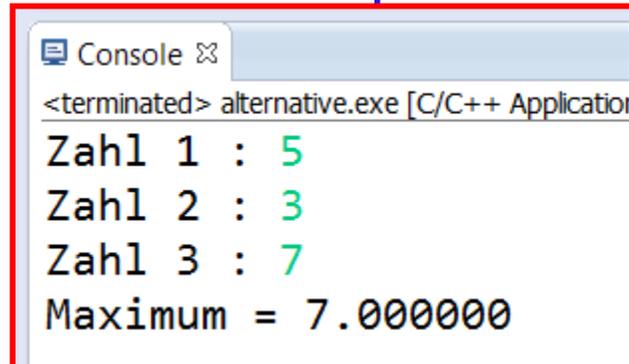
4.3.3 Sprunganweisungen

break

continue

return

```
#include <stdio.h>
int main(void) {
    double z1, z2, z3, max;
    printf("Zahl 1 :"); scanf("%lf", &z1);
    printf("Zahl 2 :"); scanf("%lf", &z2);
    printf("Zahl 3 :"); scanf("%lf", &z3);
    if ( z1 > z2 )
    {
        if ( z1 > z3 )
            max = z1;
        else
            max = z3;
    }
    else
    {
        if ( z2 > z3 )
            max = z2;
        else
            max = z3;
    }
    printf("Maximum = %f\n", max);
    return 0;
}
```



```
Console
<terminated> alternative.exe [C/C++ Application]
Zahl 1 : 5
Zahl 2 : 3
Zahl 3 : 7
Maximum = 7.000000
```

4.3.1 Alternativen – Funktion input

Command Window

```
>> help input
```

input Prompt for user input.

RESULT = input(PROMPT) displays the PROMPT string on the screen, waits for input from the keyboard, evaluates any expressions in the input, and returns the value in RESULT. To evaluate expressions, input accesses variables in the current workspace. If you press the return key without entering anything, input returns an empty matrix.

STR = input(PROMPT,'s') returns the entered text as a MATLAB string, without evaluating expressions.

To create a prompt that spans several lines, use '\n' to indicate each new line. To include a backslash ('\') in the prompt, use '\\'.

Example:

```
reply = input('Do you want more? Y/N [Y]:','s');  
if isempty(reply)  
    reply = 'Y';  
end
```

See also [keyboard](#).

[Reference page for input](#)

Command Window

```
>> x=input('Zahl x : ')  
Zahl x : 5  
x =  
    5  
>> y = input('Zahl y : ')  
Zahl y : 4*x  
y =  
    20
```

Command Window

```
>> z=input('Eingabe : ')  
Eingabe : sin(pi/6)  
z =  
    0.5000
```

Wie in C gibt es verschiedene Möglichkeiten Alternativen zu programmieren.

a) *if-else*-Anweisung

```
if logischerAusdruck  
    Anweisungen-1  
else  
    Anweisungen-2  
end
```

```
if a < 0  
    b = -a;  
else  
    b = a;  
end  
c = sqrt(b);
```

```
if ( a < 0 )  
    b = -a;  
else  
    b = a;  
end  
c = sqrt(b);
```

b) Bedingte Anweisung - Einfaches *if*

```
if logischerAusdruck  
    Anweisungen  
end
```

```
if a < 0  
    a = 0;  
end  
b = a;
```

```
if ( a < 0 )  
    a = 0;  
end  
b = a;
```

Die runden Klammern um die Bedingung sind optional. Der Anweisungsblock muss stets durch das Schlüsselwort *end* abgeschlossen werden.

Merke : Zu jedem *if* gehört genau ein *end* .

c) Verschachtelte `if-else`-Anweisungen

```
if logischerAusdruck-1
  Anweisungen-1
else
  if logischerAusdruck-2
    Anweisungen-2
  else
    if logischerAusdruck-3
      Anweisungen-3
    else
      ...
    end
  end
end
end
```

Verschachtelung :

- Jedes `if` wird durch ein `end` abgeschlossen.
- Ohne saubere Formatierung wird es unübersichtlich und unverständlich!

d) Mehrfachauswahl

```
if logischerAusdruck-1
  Anweisungen-1
elseif logischerAusdruck-2
  Anweisungen-2
elseif logischerAusdruck-3
  Anweisungen-3
elseif
  ...
else
  ...
end
```

Ein `elseif` wird durch ein anderes `elseif`, ein `else` oder ein `end` abgeschlossen.

e) Fallunterscheidung `switch - case`

```
switch Auswahlausdruck
  case Wert-1
    Anweisungen-1
  case Wert-2
    Anweisungen-2
  . . .
  otherwise
    Anweisungen-1
end
```

```
switch ( note )
  case 1
    disp('sehr gut')
  case 2
    disp('gut')
  otherwise
    disp('mehr üben')
end
```

```
switch ( tag )
  case 'Montag'
    disp('Arbeiten')
  case 'Dienstag'
    disp('Arbeiten')
  case 'Mittwoch'
    disp('Frei')
  case 'Donnerstag'
    disp('Arbeiten')
  case 'Freitag'
    disp('Arbeiten')
  otherwise
    disp('Wochenende')
end
```

Die `switch`-Anweisung wertet einen Ausdruck aus, der verschiedene **diskrete** Werte (nicht notwendig ganze Zahlen wie in der Programmiersprache C) annehmen kann. Im Gegensatz zu C entfällt die `break`-Anweisung. Der **otherwise**-Teil ist optional.

Beispiele für einfache Schleifen in der Programmiersprache C

for - Schleife

```
#include <stdio.h>
int main(void)
{
    int i,q;
    for (i=1; i<=15; i=i+1)
    {
        q = i*i;
        printf("i=%d Quadrat=%d\n",i,q);
    }
    return 0;
}
```

while - Schleife

```
#include <stdio.h>
int main(void)
{
    int i,q;
    i=1;
    while ( i <= 15 )
    {
        q = i*i;
        printf("i=%d ...
        i = i +1;
    }
    return 0;
}
```

```
Console
<terminated> quadratfor.exe [C/C++ App]
i=1 Quadrat=1
i=2 Quadrat=4
i=3 Quadrat=9
i=4 Quadrat=16
i=5 Quadrat=25
```

Bei Schleifen unterscheidet man zwischen **Zählschleifen** (`for`-Schleife) und **Wiederholschleifen** (`while`-Schleifen). Im Gegensatz zu C gibt es bei MATLAB keine `do-while`-Schleife.

Zähl-Schleife :

```
for index = startwert : endwert  
    Anweisungen  
end
```

```
for ( index = startwert : endwert )  
    Anweisungen  
end
```

```
for index = startwert : inkrement : endwert  
    Anweisungen  
end
```

Wiederhol-Schleife :

```
while Bedingung  
    Anweisungen  
end
```

```
while ( Bedingung )  
    Anweisungen  
end
```

index : Zählvariable (kein Vektor!) - typischerweise eine ganze Zahl
bei ersten Durchlauf wird ***index*** auf den Startwert gesetzt

inkrement : Wert, um den die Zählvariable ***index*** nach einem Schleifen-
durchlauf erhöht wird , Defaultwert für ***inkrement*** ist 1

Ein Schleifendurchlauf kann mit Sprunganweisungen unterbrochen werden :

- **break**
die innerste umgebende Schleife wird verlassen
- **continue**
der aktuelle Schleifendurchlauf der innersten umgebenden Schleife wird beendet und es wird ein neuer Schleifendurchlauf geprüft und ggf. ausgeführt
- **return**
Rücksprung aus einer Funktion zur aufrufenden Funktion
der Rücksprung erfolgt auch über Schleifenebenen

```
while ( Bedingung-a )  
  Anweisungen-1  
  if ( Bedingung-b )  
    break;  
  end  
  Anweisungen-2  
end
```

```
while ( Bedingung-a )  
  Anweisungen-1  
  if ( Bedingung-b )  
    continue;  
  end  
  Anweisungen-2  
end
```

Aufgabe 1

Welche Anweisungen sind korrekt, welche falsch? Welche Ausdrücke sind gut formatiert, welche schlecht?

```
if i == 1
    disp('i ist 1')
elseif i == 2
    disp('i ist 2')
else
    disp('i ist 3')
end
```

```
if i == 1
    disp('i ist 1')
else if i == 2
    disp('i ist 2')
else
    disp('i ist 3')
end
end
```

```
if i == 1
    disp('i ist 1')
else
    if i == 2
        disp('i ist 2')
    else
        disp('i ist 3')
    end
end
end
```

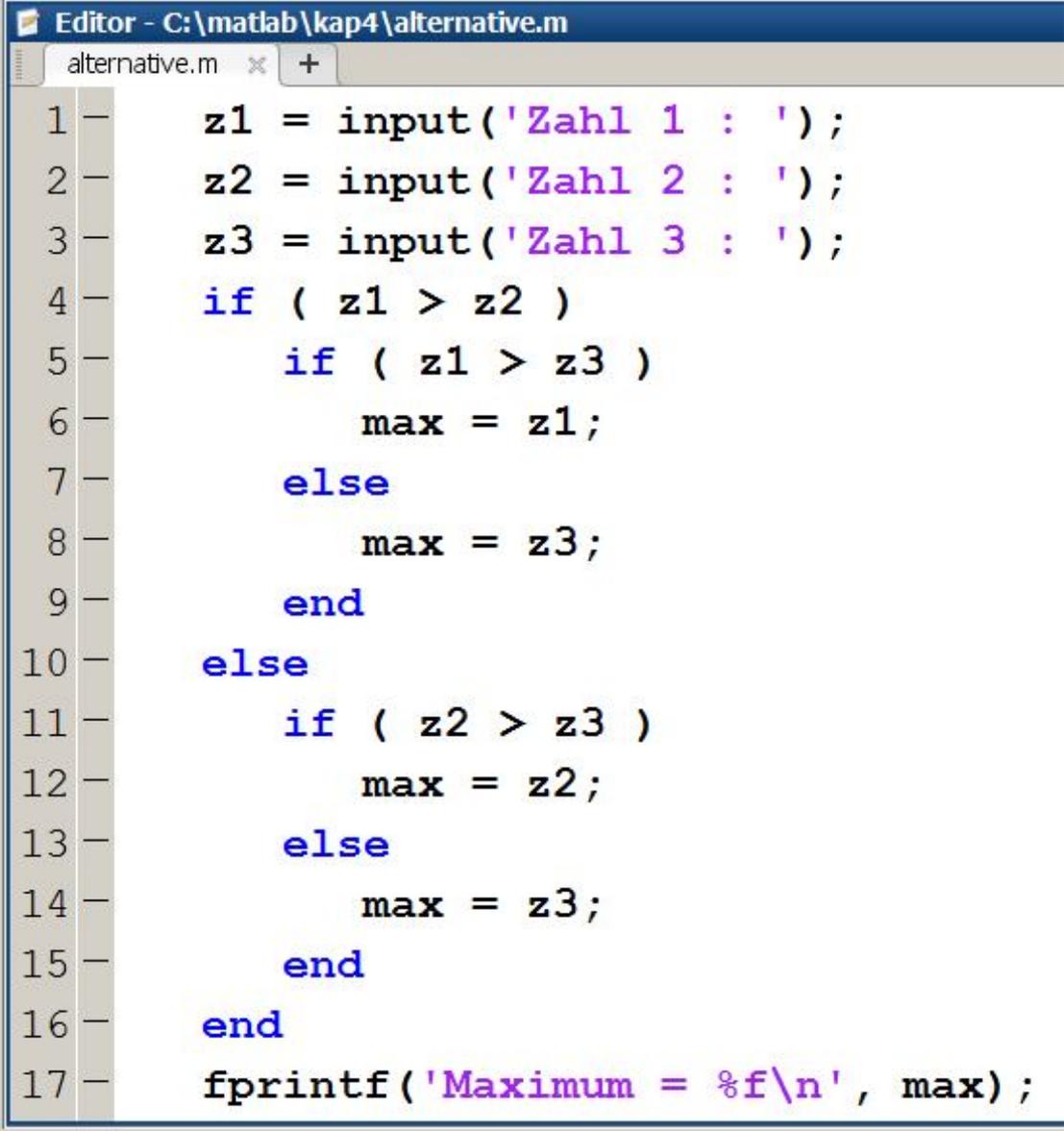
```
if i == 1
    disp('i ist 1')
else if i == 2
    disp('i ist 2')
else
    disp('i ist 3')
end
```

```
if i == 1
    disp('i ist 1')
else
    if i == 2
        disp('i ist 2')
    else
        disp('i ist 3')
    end
end
```

Aufgabe 2

Schreiben Sie das Skript `alternative.m` zur Berechnung des Maximums von drei Zahlen um. Es sollen keine verschachtelten `if-else`-Anweisungen verwendet werden sondern nur `if-elseif-else`-Anweisungen und logische Verknüpfungen.

Muss man bei den logischen Verknüpfungen `&` oder `&&` verwenden oder ist es egal? Was ist der genaue Unterschied?



```
Editor - C:\matlab\kap4\alternative.m
alternative.m x +
1 - z1 = input('Zahl 1 : ');
2 - z2 = input('Zahl 2 : ');
3 - z3 = input('Zahl 3 : ');
4 - if ( z1 > z2 )
5 -     if ( z1 > z3 )
6 -         max = z1;
7 -     else
8 -         max = z3;
9 -     end
10 - else
11 -     if ( z2 > z3 )
12 -         max = z2;
13 -     else
14 -         max = z3;
15 -     end
16 - end
17 - fprintf('Maximum = %f\n', max);
```

a) Schreibe eine MATLAB-Funktion `wurzel.m`, die die Quadratwurzel einer positiven Zahl mit Hilfe des Heron-Verfahrens berechnet.

Heron-Verfahren (Iteration): $x_{n+1} = \frac{1}{2} \cdot (x_n + x/x_n)$ $\lim_{n \rightarrow \infty} x_n = \sqrt{x}$

Für jeden positiven Startwert x_1 konvergiert x_n gegen die Quadratwurzel von x .

Beispiel :

$x = 5.0$ Startwert $x_1 = 10$

$$x_2 = \frac{1}{2} \cdot (x_1 + x/x_1) = 5.25$$

$$x_3 = \frac{1}{2} \cdot (x_2 + x/x_2) = \frac{1}{2} \cdot (5.25 + 5/5.25) = 3.101$$

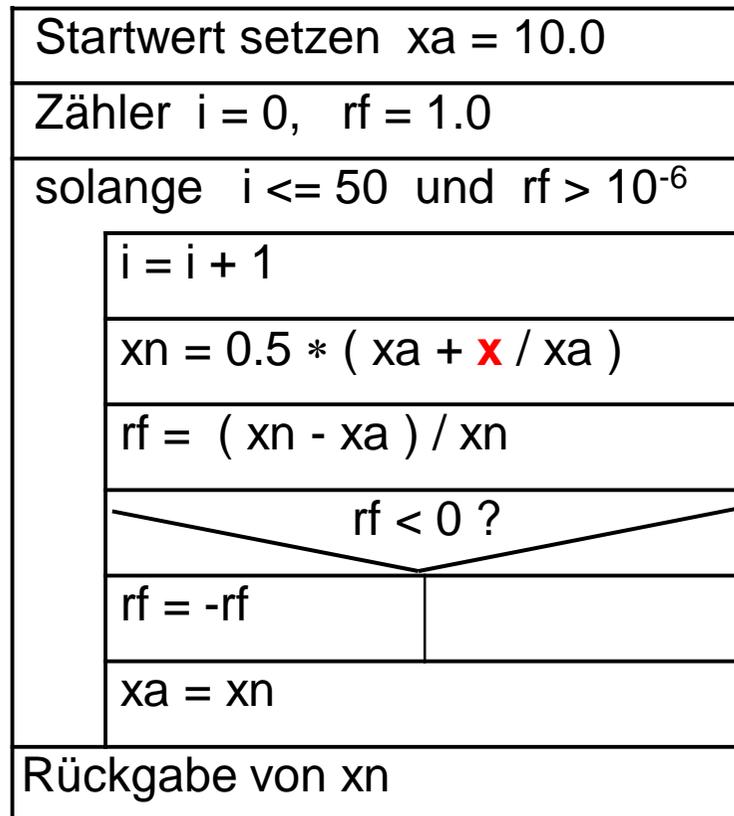
$x_4 = 2.356\ 737\ 272\ 644\ 183$

$x_5 = 2.239\ 157\ 222\ 737\ 191$

$x_6 = 2.236\ 070\ 108\ 532\ 850$

$x_7 = 2.236\ 067\ 977\ 500\ 805$

$x_8 = 2.236\ 067\ 977\ 499\ 790$



b) Schreiben Sie die Funktion so, dass man anstelle eines Skalars auch ein Array übergeben kann. Testen Sie ihre beiden Funktionen mit einem geeigneten Skript – Skalar, Matrix, mehrdimensionale Arrays (30*40*20*50), kleine Zahlen.

Gegeben ist die C-Funktion **sortiere**, die die Elemente eines Vektors der Größe nach ordnet (Bubble-Sort-Algorithmus). Der **Vektor x** wird als Parameter übergeben. Die Anzahl der Elemente des Vektors wird im Parameter **n** übergeben.

Schreiben Sie die C-Funktion in eine MATLAB-Funktion um. Der Ablauf der MATLAB-Funktion soll der C-Funktion entsprechen. Die MATLAB-Funktion besitzt als Aufrufparameter nur den Vektor **x** und gibt den **sortierten Vektor zurück**.

Verwenden Sie nur Variablen vom Typ **double**.

Schreiben Sie ein Skript **sortScr**, das mit Hilfe der Funktion **rand** Vektoren erzeugt, die anschließend sortiert werden.

Vergleichen Sie die Rechenzeit, die die Funktion **sortiere** braucht, mit der Rechenzeit, die die MATLAB-Funktion **sort** braucht, für Vektoren mit sehr vielen Elementen.

```
void sortiere(double x[], int n)
{
    int i, sortiert;
    double hilf;
    while (1 == 1)
    {
        sortiert = 1; /*Annahme: x sortiert*/
        for (i=1; i<n; i++)
        {
            if ( x[i-1] > x[i] )
            {
                hilf=x[i]; /*tausche x[i],x[i-1]*/
                x[i]=x[i-1];
                x[i-1]=hilf;
                sortiert = 0; /*nicht sortiert */
            }
        }
        if (sortiert == 1)
        {
            break; /* Vektor ist sortiert */
        }
    }
}
```

Das Skript zur Bestimmung der Rechenzeit, könnte z.B. die folgende Ausgabe erzeugen. Die erste Tabelle zeigt die Rechenzeit der Funktion `sortiere` (Bubble-Sort-Algorithmus) und der MATLAB-Funktion `sort` für verschiedene Werte von n (n ist die Anzahl der Elemente des zu sortierenden Vektors). Die Länge wird jeweils verdoppelt.

Bei $n=25600$ ist die MATLAB-Funktion um den Faktor 5000 schneller.

Die zweite Tabelle zeigt die Rechenzeit der MATLAB-Funktion `sort` für große Werte von n .

```
Command Window
>> sortScr
      n      sortiere      sort
    100      0.000571      0.000065
    200      0.000320      0.000042
    400      0.001270      0.000064
    800      0.003968      0.000069
   1600      0.016805      0.000450
   3200      0.071315      0.000165
   6400      0.309427      0.000335
  12800      1.311671      0.000708
 25600      5.334858      0.001003

Funktion sort
      n      Zeit in Sek
 10000      0.001983
100000      0.005802
1000000      0.048427
10000000      0.363769
100000000      4.884484
```

Aufgabe 5

Ein Sack Reis mit einer Masse von 40kg wird von einer Person mit der Kraft F unter dem Winkel w gezogen. Der Betrag der Kraft F ist durch folgenden Ausdruck gegeben:

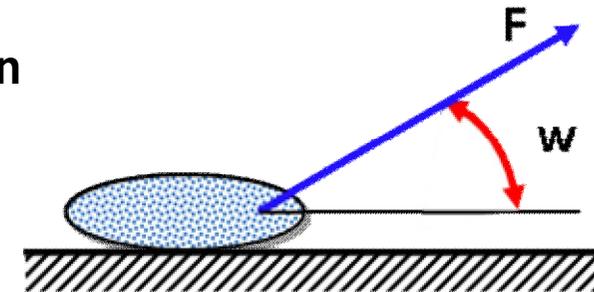
$$F(w) = \frac{m \cdot \mu}{\mu \cdot \sin(w) + \cos(w)}$$

wobei m die Masse und μ der Reibungskoeffizient ist. Dieser wird mit 0.35 angenommen.

Schreiben Sie ein MATLAB-Skript zur Ausgabe einer Tabelle (siehe rechts). In jeder Zeile der Tabelle wird ein Winkel und die zugehörige Kraft angegeben. Der Winkelbereich reicht von 5° bis 35° mit einer Schrittweite von einem Grad. Die Kraft wird über eine Funktion bestimmt. Am Ende der Tabelle wird der Winkel ausgegeben, bei dem die Kraft ein Minimum besitzt. Das Minimum wird aus den berechneten Tabellenwerten ermittelt. Bei mehrfachen Minima geben Sie den kleinsten Winkel aus. Die zugehörige Kraft wird ebenfalls ausgegeben. Danach geben Sie den Mittelwert aller Kräfte aus.

Beachten Sie: Die Tabelle besitzt eine Überschrift. Winkel werden ohne Nachkommastellen ausgegeben, Kräfte mit je drei Nachkommastellen. Die Werte für Winkel und Kraft sollen in der Tabelle rechtsbündig untereinander ausgegeben werden.

Die Kraft in Abhängigkeit von Reibungskoeffizient μ , Masse m und Winkel w wird in einer MATLAB-Funktion `kraft` berechnet und zurückgegeben. Erstellen Sie diese Funktion.



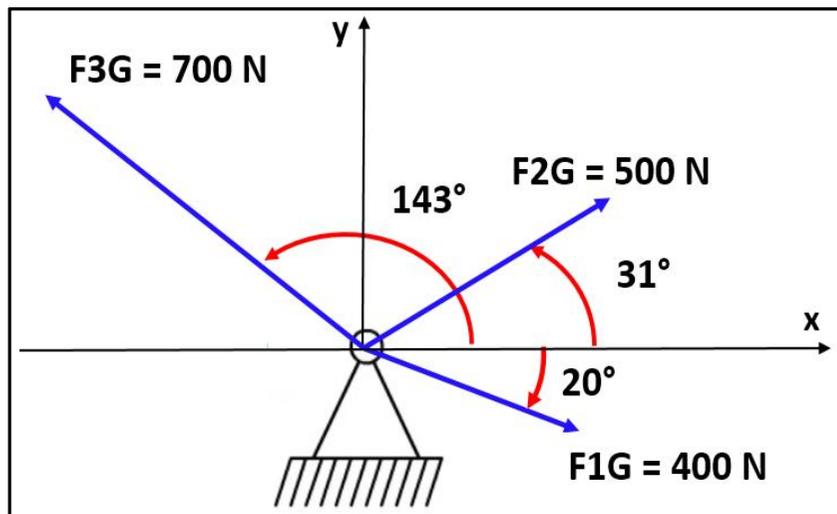
Winkel	Kraft
5	13.636
6	13.578
•	•
34	13.662
35	13.727
Minimum: $w = 19$ Kraft = 13.214	
Mittelwert der Kräfte 13.379	

Aufgabe 6

Ändern Sie die Aufgabe 2 aus Kapitel 2 so ab, das beliebig viele Kräfte am Lager angreifen können. Der Anwender wird aufgefordert die Kräfte und die Richtungen einzugeben. Ist der Betrag der Kraft negativ wird die Eingabe beendet. Anschließend werden die Kräfte und die zugehörigen Richtungen zur Kontrolle ausgegeben.

Danach wird die Gesamtkraft berechnet. Geben Sie diese einmal in kartesischen Koordinaten und dann mit Betrag und Richtung aus. Für die Kräfte aus dem Beispiel von Kap2/Aufgabe2 sieht die Ausgabe wie rechts dargestellt aus .

Berücksichtigen Sie auch den Fall, dass die Gesamtkraft 0 ist und dass die Gesamtkraft in y-Richtung zeigt.



```
Kraft (Ende: -1): 400
Winkel in Grad : -20
Kraft (Ende: -1): 500
Winkel in Grad : 31
Kraft (Ende: -1): 700
Winkel in Grad : 143
Kraft (Ende: -1): -1
```

Nr.	Kraft	Winkel
1	400.0000	-20.00
2	500.0000	31.00
3	700.0000	143.00
Gesamtkraft		: (245.4158, 541.9815)
Betrag der Kraft:		594.9562
Richtung in Grad:		65.638

Kapitel 5 – Datentypen

5.1 Datentypen bei MATLAB

5.2 Zahlenformate – Literale

5.3 Mathematische Konstanten

5.4 Komplexe Zahlen

Der grundlegende Datentyp in MATLAB ist der Datentyp `double`.

Wenn es nicht explizit anders vereinbart wird, dann sind die Elemente eines Arrays vom Typ `double`.

Neben dem Datentyp `double` gibt es eine Reihe von weiteren Datentypen, die für spezielle Anwendungen gebraucht werden :

- Darstellung von Text/Strings `'Hallo'`
- logische Werte – `true` und `false`
- Variablen, die nur ganzzahlige Werte annehmen dürfen
- Speicherplatzeffiziente Darstellung von Werten
- Datentypen für komplexe Datenstrukturen

Wichtige Unterschiede zur Programmiersprache C :

- **Der Datentyp einer Variablen kann sich durch Zuweisungen ändern.**
- **Datentypen dürfen nicht ohne weiteres in einem Ausdruck gemischt werden**
- **Das Verhalten bei Overflow und Underflow ist völlig anders !**
- **Es gelten andere Regeln bei Typkonvertierung**

Die Programmiersprache MATLAB definiert folgende Datentypen (Classes) :

double
single
int8 **int16** **int32** **int64**
uint8 **uint16** **uint32** **uint64**
logical
char
function_handle function handle array
cell cell array
struct structure array
string String (seit Matlab 2017a)

Daneben gibt es noch spezielle Datentypen, die aber selten gebraucht werden.

Typ	Bytes	Wertebereich	Führende Stellen
double	8	$\pm 1.8 \cdot 10^{308}$ $\pm 4.9 \cdot 10^{-308}$	15 - 16
char	2	0 ... $2^{16}-1$	Unicode
logical	1 Bit	true (1), false (0)	

Zahlenformate

- ganze Zahlen in der Dezimaldarstellung mit oder ohne Vorzeichen

13 -20 +300

- Gleit- oder Fließkommazahlen mit Dezimalpunkt

1.5 -2.1 3.0 .26 +.25 3.1415

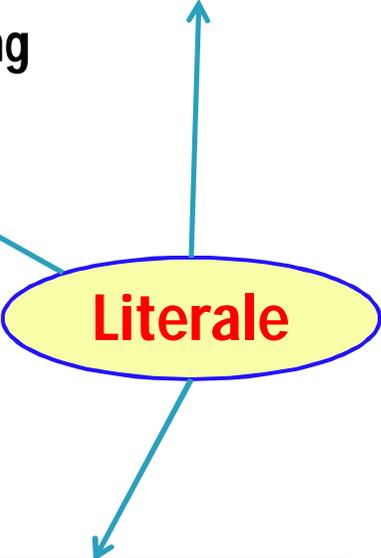
- Gleit- oder Fließkommazahlen in der Exponentialdarstellung

2.1e+3 .1e-5 2e3 0.2E-5

- Komplexe Zahlen – **imaginäre Einheit i oder j**

3 + 4i 3 + 4j nicht 3 + 4*i

Literale



Zeichen – Strings – Character-Arrays

- Character 'T'
- Zeichenketten 'Das ist ein Text'
(in C: "Das ist ein Text")
- Keine binäre Null am Ende

```
Command Window
>> 'Das ist ein Text'
ans =
Das ist ein Text
```

Mathematische Konstanten :

eps	Floating-point relative accuracy - $2.2204e-016$ $1 \sim= 1+eps$ ist false! $1 == (1+0.5*eps)$ ist true !
Inf	Infinity z.B. $1/0$
i	Imaginary unit - $3.5i$, $1 + 5i$
j	Imaginary unit - $2.4 + 5j$
NaN	Not-a-Number z.B. $0/0$
pi	Ratio of circle's circumference to its diameter - $3.1415\dots$
realmax	Largest positive floating-point number - $1.7977e+308$
realmin	Smallest positive normalized floating-point number $2.2251e-308$

Beachte : Konstanten können auch überschrieben werden, z.B.

pi = 3.17

MATLAB kann mit komplexen Zahlen umgehen. Sehr viele Funktionen arbeiten auch mit komplexen Zahlen.

Für die **imaginäre Einheit** werden die Symbole **i** oder **j** verwendet.

Um eine Verwechslung mit der imaginären Einheit zu vermeiden, sollte man bei MATLAB die Namen i und j nicht als Variablennamen verwenden.

```
>> x = sqrt(-2)
x =
    0 + 1.4142i
>> whos x
  Name Size Bytes   Class Attributes
  x     1x1    16   double   complex
>> x = 3 + 5i
x =
    3 + 5.0000i
>> y = x*6j
y =
   -30 + 18.0000i
>> x*y
ans =
 -1.8000e+002 -9.6000e+001i
```

Funktionen für komplexe Zahlen :

imag
conj
angle
abs
real
isreal

Was ist der Unterschied zwischen $x = 5i$ und $x = 5*i$?

Kapitel 6 Anwendungen aus der Analysis

6.1 Polynome

6.1.1 Funktionswert

6.1.2 Nullstellen

6.1.3 Ableitung

6.1.4 Integration

6.2 Function Handle

6.3 Quadratur – Bestimmtes Integral berechnen

6.4 Anonymous Functions

6.5 Globale Variablen

6.6 Hausaufgaben

Polynom n-ten Grades :

$$y(x) = a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \dots + a_2 \cdot x^2 + a_1 \cdot x + a_0$$

Mathematik

$$y(x) = a_1 \cdot x^n + a_2 \cdot x^{n-1} + \dots + a_{n-1} \cdot x^2 + a_n \cdot x + a_{n+1}$$

MATLAB

Beispiel : $y(x) = x^3 - 4 \cdot x^2 + 7$

$$a_1 = 1 \quad a_2 = -4 \quad a_3 = 0 \quad a_4 = 7$$

Ein Polynom n-ten Grades wird durch $n+1$ Koeffizienten a_i eindeutig festgelegt.

Die Werte der **Koeffizienten** eines Polynoms n-ten Grades werden in einem **Vektor** der Länge $n+1$ gespeichert (**Zeilen- oder Spaltenvektor**). Das erste Element des Vektors ist der Koeffizient des Terms x^n . Das letzte Element des Vektors ist der konstante Term des Polynoms.

$$a = [1, -4, 0, 7] \quad a = [1; -4; 0; 7]$$

Beim Arbeiten mit einem Polynom wird der Vektor mit den Werten der Koeffizienten als Parameter an MATLAB-Funktionen übergeben. Mit Hilfe des Vektors kann der Funktionswert des Polynoms an einer beliebigen Stelle x berechnet werden, ebenso Ableitungen, das Integral oder die Nullstellen.

Funktionswert an einer vorgegebenen Stelle x berechnen:

$$y(x) = a_1 \cdot x^n + a_2 \cdot x^{n-1} + \dots + a_{n-1} \cdot x^2 + a_n \cdot x + a_{n+1}$$

$$y(x) = x^3 - 4 \cdot x^2 + 7$$

$$a_1 = 1 \quad a_2 = -4 \quad a_3 = 0 \quad a_4 = 7$$

```
function y = pvalue(
```

```
end
```

```
> a = [ 1, -4, 0, 7 ];  
> x = 2.3;  
> y = pvalue(a, x);
```

```
> a1 = [ 1; -4; 0; 7 ];  
> x = [ 1.2 2.4 3.7 ];  
> y = pvalue(a1, x);
```

Die Funktion `pvalue` muss nicht selbst programmiert werden (ein Polynom kann wesentlich besser und effektiver ausgewertet werden als in `pvalue` gezeigt – **Hornerschema**). Für die Auswertung eines Polynoms und für viele andere Aufgaben gibt es bereits fertige Funktionen – diese Funktionen sollte man verwenden!

Aufgabe : Funktionswerte bestimmen

Die MATLAB-Funktion `polyval` berechnet die Funktionswerte eines Polynoms an der „Stelle x “, wobei x ein Skalar oder ein beliebiges Array sein kann.

```
> a = [ 1,-4,0,7 ];  
> x = 4;  
> y = polyval(a, x);
```

```
> a = [ 1; -4; 0; 7 ];  
> x = [-2:0.1:5];  
> y = polyval(a, x);
```

`polyvalScr.m`

`polyval` Evaluate polynomial.

`Y = polyval(P,X)` returns the value of a polynomial P evaluated at X . P is a vector of length $N+1$ whose elements are the coefficients of the polynomial in descending powers.

$$Y = P(1)*X^N + P(2)*X^{(N-1)} + \dots + P(N)*X + P(N+1)$$

If X is a matrix or vector, the polynomial is evaluated at all points in X .

Aufgabe : Nullstellen eines Polynoms bestimmen

Die Funktion `roots` berechnet die Nullstellen eines **Polynoms** und gibt diese in einem Spaltenvektor zurück.

`roots` Find polynomial roots.

`roots(C)` computes the roots of the polynomial whose coefficients are the elements of the **vector C**. If C has N+1 components, the polynomial is $C(1)*X^N + \dots + C(N)*X + C(N+1)$.

Bemerkung :

- Ein Polynom n-ten Grades besitzt genau n Nullstellen.
- Die Nullstellen eines Polynoms können reell oder komplex sein.
- Die Nullstellen für Polynome 5-ten und höheren Grades müssen meist numerisch bestimmt werden. Es gibt kein allgemein anwendbares analytisches Verfahren.

Verifikation, dass tatsächlich Nullstellen vorliegen – o.k.

```
> a = [ 1 -4 0 7 ];  
> nst = roots(a)  
      nst = 3.3914  
           1.7729  
          -1.1642  
> polyval(a,nst)  
ans = 1.0e-014 *  
      0.1776  
     -0.1776  
      0.5329
```

`rootsScr.m`

Aufgabe : Die Ableitung eines Polynoms bestimmen

Die Funktion `polyder` (**der** – derivative - Ableitung) berechnet die Ableitung eines Polynoms. Das Ergebnis ist ein Zeilenvektor, der die Koeffizienten des abgeleiteten Polynoms enthält. Die Ableitung wird analytisch berechnet.

Beispiel :

Berechne die Ableitung des Polynoms :

$$y(x) = x^3 - 4 \cdot x^2 + 7$$

Das Ergebnis ist das Polynom :

$$y'(x) = 3 \cdot x^2 - 8 \cdot x$$

Anwendung :

Bestimme die lokalen Extrema eines Polynoms.

- Berechne die erste Ableitung
- Bestimme die Nullstellen der Ableitung
- Berechne die Funktionswerte an den Nullstellen der ersten Ableitung

Skizzieren Sie das Polynom! Verwenden Sie `nst` und `extrema`.

```
> a = [ 1 -4 0 7 ];
> da = polyder(a)
da =
      3      -8      0
> xextr = roots(da)
xextr =
           0
      2.6667
> yextr=polyval(a,xextr)
yextr = 7.0000
      -2.4815
```

`polyderScr.m`

```
>> help polyint
polyint Integrate polynomial analytically.
polyint(P,K) returns a polynomial representing the
integral of polynomial P, using a scalar constant of
integration K.

polyint(P) assumes a constant of integration K=0.

Class support for inputs p, k:
    float: double, single

See also polyder, polyval, polyvalm, polyfit.

Reference page in Help browser
    doc polyint
```

Beachte :

Die Funktion **polyint** integriert ein Polynom analytisch. Das ist immer möglich, weil die Stammfunktion eines Polynoms wieder ein Polynom ist. **polyint** gibt die Koeffizienten der Stammfunktion als Vektor zurück.

Aufgabe :

Berechne den Wert des bestimmten Integrals unter Verwendung der Funktion **polyint**.

$$\int_2^4 y(x) \cdot dx = \int_2^4 (x^3 - 4 \cdot x^2 + 7) \cdot dx$$

Aufgabe :

Gegeben sind eine Reihe von Funktionen, die als M-Files definiert sind :

sin, **cos**, **exp** und die selbst geschriebenen Funktionen **xsin**, **gauss** und **poly**

```
function y=xsin(x)
    y = x.*sin(x);
end
```

```
function y=gauss(x)
    y = exp(-0.2*x.*x);
end
```

```
function y=poly(x)
    a = [1,-4,0,7];
    y = polyval(a,x);
end
```

Diese und weitere Funktionen, die die gleiche Struktur haben, nämlich $y=f(x)$, sollen mit einer Funktion **plotFktn** im Intervall $[-5, +5]$ gezeichnet werden. Als **Parameter** werden die **Funktion** und die Anzahl der Punkte **n** übergeben. Der Kopf der Funktion **plotFktn** sieht also wie folgt aus :

```
function [ ] = plotFktn( fun, n )
```

Probleme :

Wie kann man eine **Funktion**, die in einem M-File definiert worden ist, als **Parameter** an eine andere Funktion übergeben?

Wie können Funktionswerte einer Funktion, die als Parameter übergeben worden ist, berechnet werden? Wie verwendet man **fun** im Innern der Funktion **plotFktn** um Funktionswerte von **fun** zu berechnen ?

Antwort : verwende ein **Function-Handle**

Function Handle :*handle (engl.) : Henkel oder Griff*

Ein function handle ist ein **eigener Datentyp**, der einen Verweis (Funktionszeiger) auf eine Funktion speichert. In einer Variablen vom Typ **function_handle** wird gespeichert, um welche Funktion es sich konkret handelt. Ein function handle wird häufig an eine andere Funktionen übergeben, damit diese die Funktion, die durch das function handle festgelegt wird, aufrufen kann.

```
> hd1 = @xsin
> hd2 = @exp
> plotFktn(hd1,100)
> plotFktn(hd2,200)
> plotFktn(@gauss,300)
```

Zwei Variablen vom Typ **function_handle** erzeugen und mit Werten belegen

An die `plotFktn` wird jeweils ein function handle als Parameter übergeben. `plotFktn` kann die entsprechende Funktion aufrufen, Funktionswerte an geeigneten Stellen bestimmen und dann die Funktion zeichnen.

```
> help function_handle
```

```
FUNHANDLE = @FUNCTION_NAME returns a handle to the named function,
FUNCTION_NAME. A function handle is a MATLAB value that provides a
means of calling a function indirectly. You can pass function handles
in calls to other functions (often called function functions). You
can also store function handles in data structures for later use (for
example, as Handle Graphics callbacks). A function handle is one of
the standard MATLAB data types. Its class is 'function_handle'.
```

```
function plotFktn( fun, n )
```

```
end
```

plotFktn.m

```
> hd1 = @gauss
```

```
> hd2 = @sin
```

```
> plotFktn(hd1, 150)
```

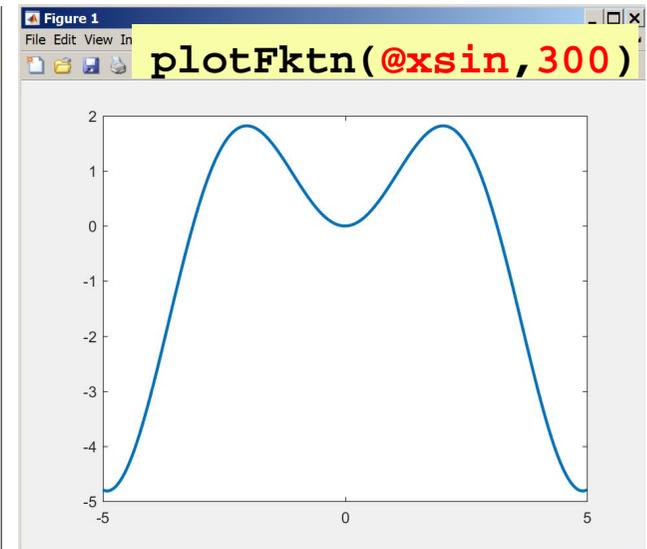
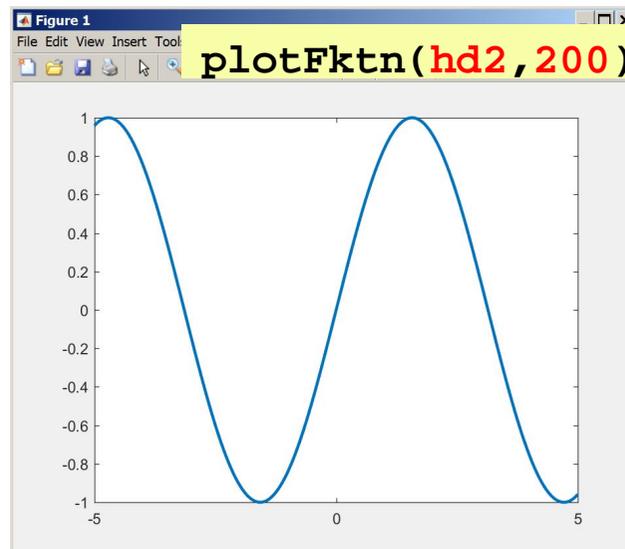
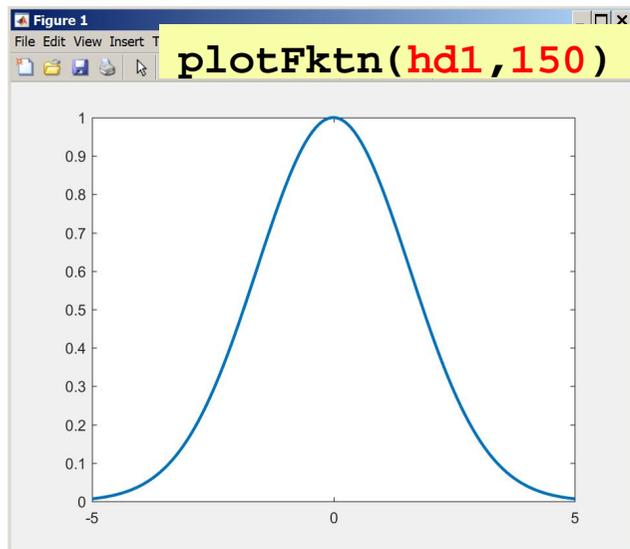
```
> plotFktn(hd2, 200)
```

```
> plotFktn(@xsin, 300)
```

handleScr.m

```
function y=xsin(x)
    y = x.*sin(x);
end
```

xsin.m



Frage :

Wie kann man einfach sichtbar machen, dass der Aufruf `y=fun(x)` tatsächlich die übergebene Funktion aufruft?

Aufgabe :

Berechne den Wert eines bestimmten Integrals bei vorgegebenen Integranden.

$$\int_2^4 y(x) \cdot dx = \int_2^4 (x^3 - 4 \cdot x^2 + 7) \cdot dx$$

$$\int_2^4 y(x) \cdot dx = \int_2^4 e^{-x^2} \cdot dx$$

$$\int_0^4 y(x) \cdot dx = \int_0^4 \frac{\sin(x)}{x} dx$$

Das **linke Integral** kann analytisch berechnet werden, da man eine Stammfunktion aus elementaren Funktionen angeben kann.

$$\frac{1}{4} \cdot x^4 - \frac{4}{3} \cdot x^3 + 7 \cdot x + c$$

Die Funktion **polyint** berechnet ein Integral analytisch – das Ergebnis ist wieder ein Polynom. Die Funktion **polyint** kann aber nur Polynome integrieren.

Das **beiden anderen Integrale** können nicht analytisch berechnet werden. Die Werte dieser Integrale müssen numerisch bestimmt werden. Hierzu gibt es verschiedene Verfahren, z.B. das Trapezverfahren oder das Simpsonverfahren.

Wichtige Forderung an ein **Verfahren**, die das **Integral einer Funktion berechnet** :

An das Verfahren wird neben den Integrationsgrenzen auch der **Integrand als Parameter** (d.h. eine MATLAB-Funktion) übergeben. Das Verfahren kann dann **Integrale von beliebigen Funktionen** berechnen.

```
> help integral
```

```
integral Numerically evaluate integral.
```

```
Q = integral(FUN,A,B) approximates the integral of function FUN from A to B using global adaptive quadrature and default error tolerances.
```

```
FUN must be a function handle. A and B can be -Inf or Inf. If both are finite, they can be complex. If at least one is complex, integral approximates the path integral from A to B over a straight line path.
```

```
For scalar-valued problems the function Y = FUN(X) must accept a vector argument X and return a vector result Y, the integrand function evaluated at each element of X. ...
```

$$\int_2^4 y(x) \cdot dx = \int_2^4 (x^3 - 4 \cdot x^2 + 7) \cdot dx$$

```
function y = poly( x )
    a = [ 1, -4, 0, 7 ];
    y = polyval(a, x);
end
```

poly.m

$$\int_2^4 y(x) \cdot dx = \int_2^4 e^{-x^2} \cdot dx$$

```
function y = gauss( x )
    y = exp( -x.*x );
end
```

gauss.m

```
> z = integral( @poly, 2, 4 );
```

```
> z = integral( @gauss, 2, 4 );
```

Der Aufruf der Funktionen `poly` und `gauss` durch die Funktion `integral` ist nicht direkt sichtbar! Beide Funktionen müssen aber in Form von M-Files vorhanden sein. Beide Funktionen müssen die Übergabe von Arrays unterstützen.

```
function z = integral( fun, a, b )
    . . .
    fun(x)
    . . .
    y
end
```

integral.m

```
function y = fun( x )
    . . .
    . . .
end
```

m-File : fun.m

Oft ist es umständlich für jede Funktion, die man als Parameter übergeben muss, ein eigenes M-File zu schreiben besonders, wenn es sich um einfache Funktionen handelt. In diesem Fall kann man „**anonymous Functions**“ verwenden.

Beispiele :

```
function y=xsin(x)
    y = x.*sin(x);
end
```

```
function y=gauss(x)
    y = exp(-0.2*x.*x);
end
```

```
function y=poly(x)
    y = x.*x+.2*x+3.1;
end
```

Wie wird die Funktion `integral` aufgerufen, wenn man anonymous Function-Handles oder temporäre Function-Handles verwendet ?

Welche Werte ergeben sich bei folgenden Aufrufen?

```
a = integral(@(x) 2*x,1,10)    b = integral(@(x) 3*x.*x,1,10)
```

What Are Anonymous Functions?

An **anonymous function** is a function that is **not stored in a program file**, but is **associated with a variable whose data type is function_handle**. Anonymous functions can accept inputs and return outputs, just as standard functions do. However, they can contain only a single executable statement. For example, create a handle to an anonymous function that finds the square of a number:

```
sqr = @(x) x.^2;
```

Variable `sqr` is a function handle. The `@` operator creates the handle, and the parentheses `()` immediately after the `@` operator include the function input arguments. This anonymous function accepts a single input `x`, and implicitly returns a single output, an array the same size as `x` that contains the squared values. Find the square of a particular value (5) by passing the value to the function handle, just as you would pass an input argument to a standard function.

```
a = sqr(5)
```

```
a =
```

```
25
```

Many MATLAB functions accept function handles as inputs so that you can evaluate functions over a range of values. You can create handles either for anonymous functions or for functions in program files. The benefit of using anonymous functions is that you do not have to edit and maintain a file for a function that requires only a brief definition. For example, find the integral of the `sqr` function from 0 to 1 by passing the function handle to the `integral` function:

```
q = integral(sqr,0,1);
```

You do not need to create a variable in the workspace to store an anonymous function. Instead, you can create a **temporary function handle** within an expression, such as this call to the `integral` function:

```
q = integral(@(x) x.^2,0,1);
```

Aufgabe :

Berechne das Integral für verschiedene Werte von **a**.

$$\int_2^4 y(x) \cdot dx = \int_2^4 e^{-a \cdot x^2} \cdot dx$$

```
function y = gauss( x )  
    y = exp( -x.*x );  
end
```

```
function y = gneu( x , a )  
    y = exp( -a*x.*x );  
end
```

```
> z = integral( @gauss , 2 , 4 );
```

```
> z = integral( @gneu , 2 , 4 );
```

Man kann in der Funktion `gauss` einen weiteren Parameter hinzufügen – siehe Funktion `gneu`. Diese Lösungsvariante funktioniert aber nicht, da die Funktion `integral` ein Function Handle einer Funktion der Form $y = \text{fun}(x)$ erwartet. Die Koeffizienten des Polynoms können daher nicht (so einfach) als Parameter übergeben werden.

Eine Lösung für dieses und eine Reihe ähnlicher Probleme bieten **globale Variablen**. Eine globale Variable kann von mehreren Funktionen und im Workspace verwendet werden. Alle Funktionen und auch der Workspace dürfen die globale Variable verändern. Jede Veränderung des Wertes ist für die anderen Funktionen sichtbar. Eine **globale Variable** wird **nur einmal im Speicher** gehalten.

global Define global variable.

global X Y Z defines X, Y, and Z as **global in scope**.

Ordinarily, each MATLAB function, defined by an M-file, has its own **local variables**, which are separate from those of other functions, and from those of the base workspace. However, **if several functions, and possibly the base workspace, all declare a particular name as global, then they all share a single copy of that variable.**

Any assignment to that variable, in any function, is available to all the other functions declaring it global.

If the global variable doesn't exist the first time you issue the global statement, it will be initialized to the empty matrix.

Integration der Gauss-Funktion für beliebige Werte des Parameters a.

```
function y = gaussG1( x )  
    global a;  
    y = exp( -a*x.*x );  
end
```

M-File der Funktion **gaussG1**

Diese Funktion definiert die Variable **a** als **globale Variable**. Der Wert von a wird in der Funktion nur gelesen aber nicht gesetzt.

```
> global a;  
> a = 0.2;  
> z1 = integral(@gaussG1, 2, 4)  
z1 =  
    0.3854  
> a = 0.4;  
> z2 = integral(@gaussG1, 2, 4)  
z2 =  
    0.1027
```

integralScr.m

Im Workspace wird die Variable **a** ebenfalls als **globale Variable** definiert, danach erfolgt eine Wertzuweisung. Später wird der Wert von **a** geändert.

workspace

Die Variable **a** muss **zweimal als globale Variable definiert** werden, in der Funktion **gaussG1** und im **Workspace** oder im **aufrufenden Skript** (siehe Skript **integralScr.m**). Der Wert von **a** wird im Workspace gesetzt und verändert, in **gaussG1** nur gelesen und an die Funktion **exp** übergeben.

Aufgabe 1

Schreiben Sie in ein Skript `analysis.m`, das die Anweisungen zur Lösung der folgenden Aufgaben enthält

1. Stellen Sie die Funktion $y(x) = x^3 - 4 \cdot x^2 + 7$ sowie deren erste und zweite Ableitung graphisch dar.
2. Geben Sie eine Wertetabelle für das x-Intervall $[0, 3]$ aus, Schrittweite 0.1.
3. Bestimmen Sie die Nullstellen der Funktion und die Lage der Extrema.
4. Geben Sie die Koeffizienten der Stammfunktion aus.
5. Berechnen Sie das Integral der Funktion zwischen den Grenzen 1 und 3. Lösen Sie die Aufgabe auf zwei verschiedene Arten :
 - mit Hilfe der Funktion `polyint`
 - mit Hilfe der Funktion `integral`.

Aufgabe 2

Schreiben Sie ein Skript, das eine Wertetabelle für das Integral

$$z = \int_0^y \frac{\sin(x)}{x} dx$$

im Intervall $[0, 2\pi]$ ausgibt. Der Abstand der y-Werte soll jeweils 10° betragen. Rufen Sie die Funktion **integral** auf 3 verschiedene Arten auf :

- Ein Function-Handle eines geeigneten M-Files übergeben
- Anonymous Function beim Aufruf definieren
- Variable vom Typ Function-Handle beim Aufruf verwenden

Command Window		
Winkel	Bogenmaß	Integral
0	0.0000	0.00000
10	0.1745	0.17424
20	0.3491	0.34671
30	0.5236	0.51569
40	0.6981	0.67950
50	0.8727	0.83658
...		
330	5.7596	1.44073
340	5.9341	1.42812
350	6.1087	1.42062
360	6.2832	1.41815

Warum gilt für kleine Werte von y ?

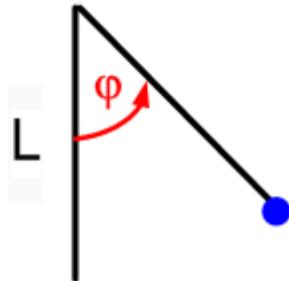
$$z = \int_0^y \frac{\sin(x)}{x} dx \approx y$$

Hierbei ist y ein kleiner Winkel im Bogenmaß.

Aufgabe 3

Berechnen Sie die Schwingungsdauer eines Pendels mit Hilfe einer Funktion **schwingungsdauer**. An diese Funktion werden die Länge des Pendels [in Metern] und die Anfangsauslenkung [in Grad] übergeben. Die Funktion gibt die Schwingungsdauer in Sekunden zurück.

Die Schwingungsdauer ist durch folgende Formel gegeben :



$$T = 4 \cdot \sqrt{\frac{L}{g}} \cdot \int_0^{\pi/2} \frac{d\psi}{\sqrt{1 - k^2 \sin^2 \psi}}$$

Anfangsauslenkung φ_0

$$k = \sin \frac{\varphi_0}{2} \quad k \sin(\psi) = \sin(\varphi/2)$$

Wie lautet der Integrand $f(x)$? Testen Sie ihr Programm! Für kleine Werte von k (d.h. für kleine Anfangsauslenkungen) ist die Periodendauer näherungsweise $2 \cdot \pi$ Sekunden, wenn für die Länge $L = 9.81\text{m}$ gewählt wird. Beweisen Sie das!

Berechnen Sie die Schwingungsdauer für eine Anfangsauslenkung von 30° .

$$T = \text{schwingungsdauer}(9.81, 30) \quad (\text{Ergebnis: } T = 6.39257)$$

Geben Sie danach eine Tabelle mit den Werten der Schwingungsdauer für Anfangsauslenkungen von 0° bis 180° aus, Schrittweite 10° .

Kapitel 7 - Lineare Algebra

7.1 Gleichungssysteme und Quadratische Matrizen

7.2 Linksdivision

7.3 Gewöhnliches Eigenwertproblem

7.4 Anwendung : Lineare Kette

7.5 Verallgemeinertes Eigenwertproblem

7.6 Hausaufgaben

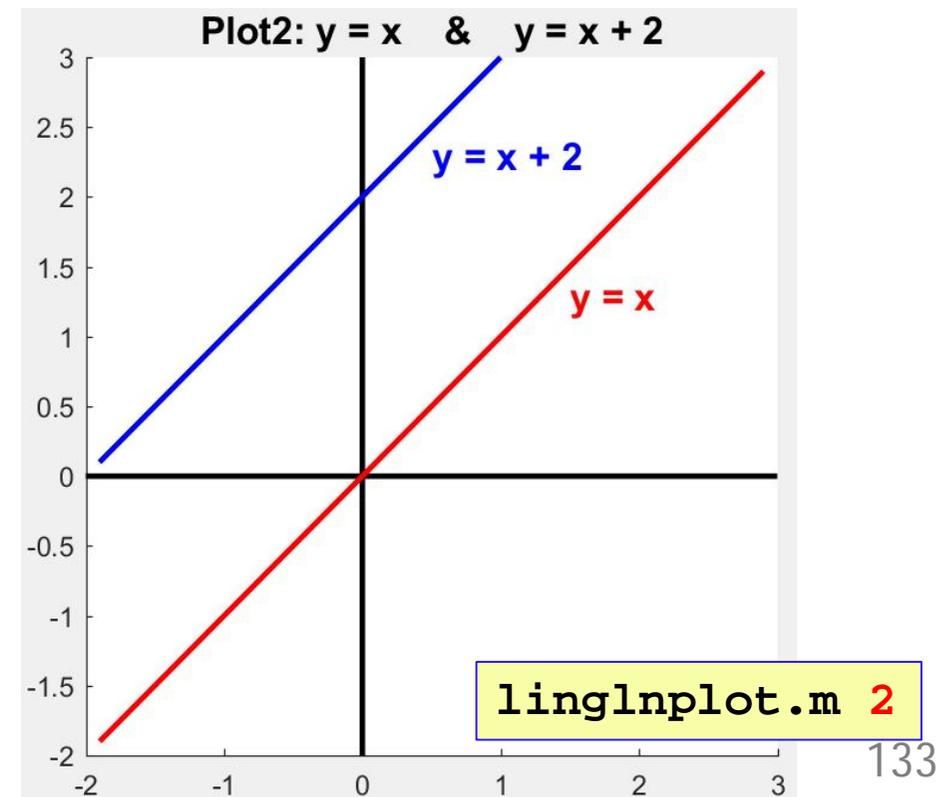
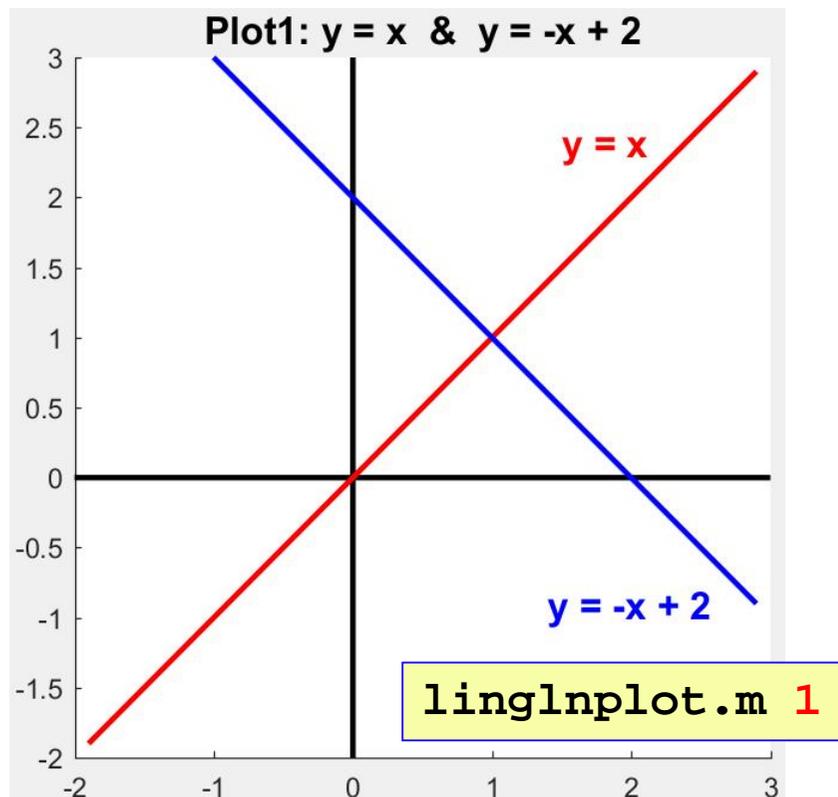
Lineare Gleichungssysteme :

Beispiel 1 : Zwei Gleichungen mit zwei Unbekannten x und y

$$\begin{array}{l} 1: -x + y = 0 \\ 2: x + y = 2 \end{array} \quad \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

$$\begin{array}{l} 1: -x + y = 0 \\ 2: -2x + 2y = 4 \end{array} \quad \begin{bmatrix} -1 & 1 \\ -2 & 2 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

Lineare Gleichungssysteme löst man meist mit Hilfe eines Eliminationsverfahrens. Jede dieser Gleichungen stellt eine Geradengleichung dar. Die Lösung von zwei Gleichungen mit zwei Unbekannten ist somit der Schnittpunkt von zwei Geraden.



Beispiel 2: Drei Gleichungen mit drei Unbekannten x , y und z .

$$\begin{array}{l} -x + y + z = 0 \\ x + 2y + z = 2 \\ 3x + y + z = 3 \end{array} \quad \begin{bmatrix} -1 & 1 & 1 \\ 1 & 2 & 1 \\ 3 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 3 \end{bmatrix}$$

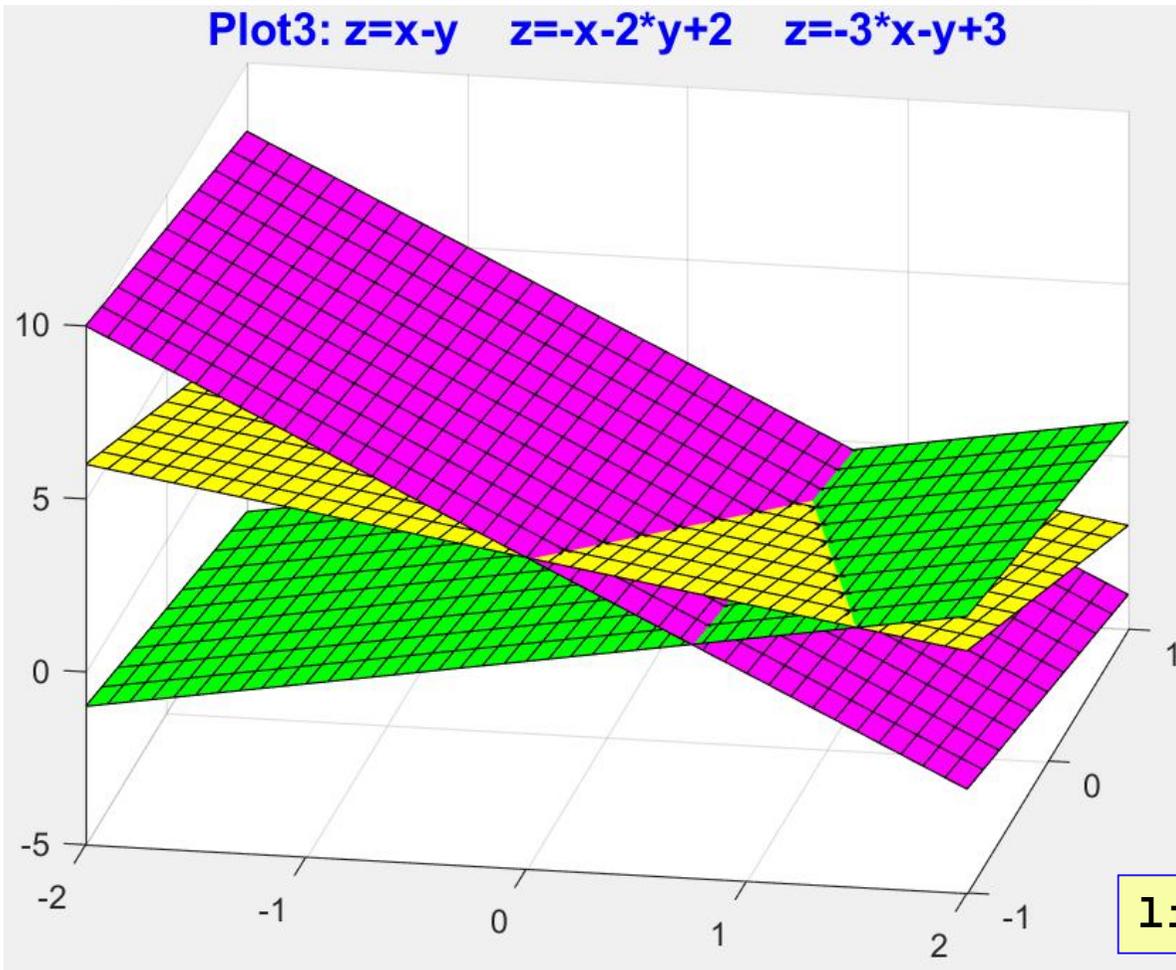
Jede der drei Gleichungen beschreibt eine Ebene im Raum (z als Funktion von x und y).

$$z = x - y \quad (\text{Gl. 1})$$

$$z = -x - 2y + 2 \quad (\text{Gl. 2})$$

$$z = -3x - y + 3 \quad (\text{Gl. 3})$$

Die Lösung des Gleichungssystems ist somit der Schnittpunkt von drei Ebenen im Raum. Normalerweise gibt es genau einen Schnittpunkt der drei Ebenen, aber es können auch Spezialfälle eintreten (Ebenen liegen parallel zueinander oder Ebenen fallen zusammen).



linglnplot.m 3

Beispiel 3: Überbestimmte Gleichungssysteme

$$\begin{array}{l}
 1: \quad -x + y = 0 \\
 2: \quad x + y = 2 \\
 3: \quad -0.5x + y = 0.5
 \end{array}
 \quad
 \begin{bmatrix}
 -1 & 1 \\
 1 & 1 \\
 -0.5 & 1
 \end{bmatrix}
 \cdot
 \begin{bmatrix}
 x \\
 y
 \end{bmatrix}
 =
 \begin{bmatrix}
 0 \\
 2 \\
 0.5
 \end{bmatrix}$$

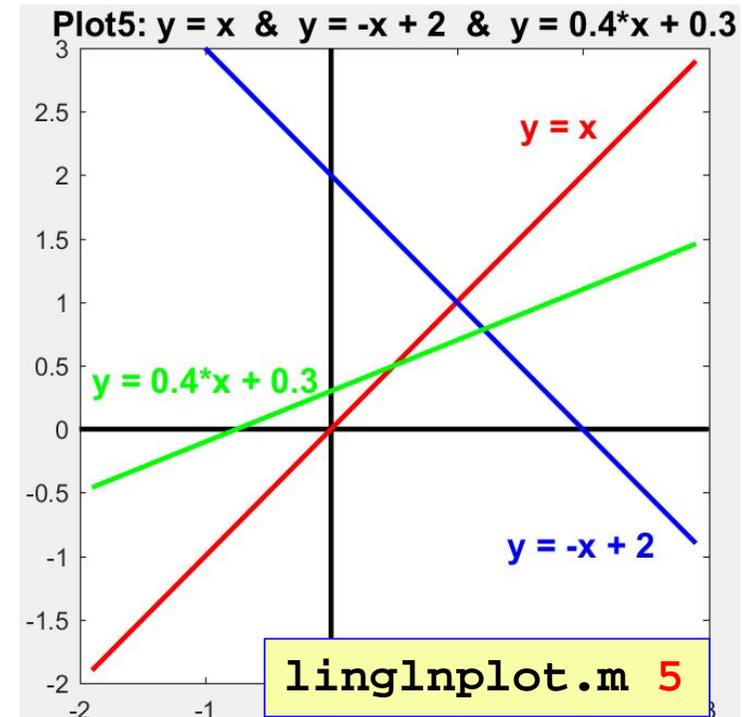
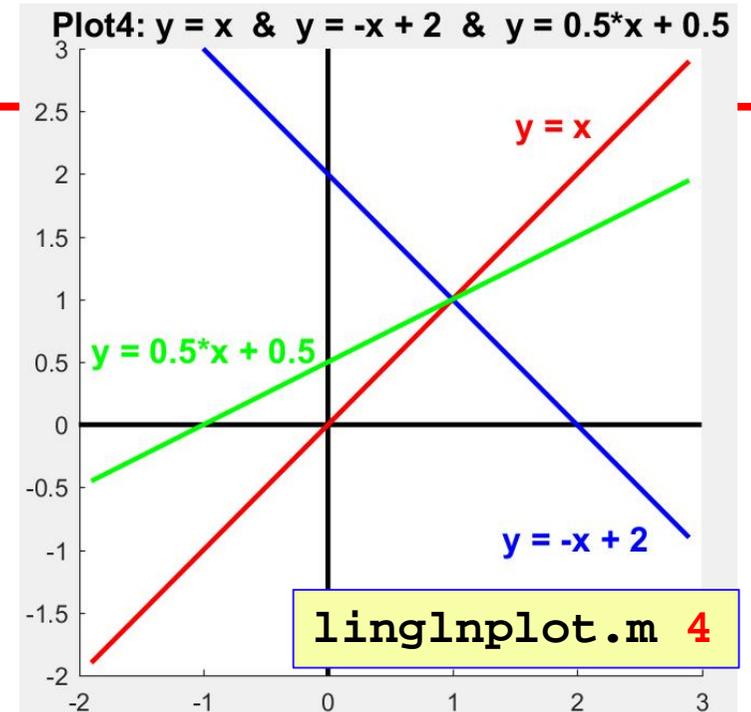
Diese 3 Gleichungen mit 2 Unbekannten besitzen eine **exakte Lösung**, nämlich $x = y = 1$. Dies ist aber ein Sonderfall.

$$\begin{array}{l}
 1: \quad -x + y = 0 \\
 2: \quad x + y = 2 \\
 3: \quad -0.4x + y = 0.3
 \end{array}
 \quad
 \begin{bmatrix}
 -1 & 1 \\
 1 & 1 \\
 -0.4 & 1
 \end{bmatrix}
 \cdot
 \begin{bmatrix}
 x \\
 y
 \end{bmatrix}
 =
 \begin{bmatrix}
 0 \\
 2 \\
 0.3
 \end{bmatrix}$$

Diese 3 Gleichungen mit 2 Unbekannten besitzen **keine (exakte) Lösung**. Das ist der Normalfall.

Beide Gleichungssysteme haben die Struktur :

$$\mathbf{A} * \mathbf{x} = \mathbf{b} \quad \mathbf{A} \text{ ist eine nicht-quadratische Matrix}$$



Allgemeine Problemstellungen :

1) $A * x = b$ „ n Gleichungen mit n Unbekannten “

A und b sind gegeben, x ist gesucht

A ist eine $n * n$ - Matrix (d.h. A ist **quadratisch**)

x und b sind Spaltenvektoren mit jeweils n Zeilen

Wichtige mathematische Größen für quadratische Matrizen :

Determinante - Inverse Matrix - Eigenwerte und Eigenvektoren - Rang

2) $A * x = b$ „ m Gleichungen mit n Unbekannten “

A und b sind gegeben, x ist gesucht

$m > n$: Gleichungssystem ist **überbestimmt**

$m < n$: Gleichungssystem ist **unterbestimmt**

A ist eine $m * n$ - Matrix (d.h. A ist **nicht quadratisch**)

x ist ein Spaltenvektor mit n Zeilen

b ist ein Spaltenvektor mit m Zeilen

A sei eine $n \times n$ -Matrix

Satz : Ist die **Determinante** der Matrix A ungleich 0, dann besitzt die Gleichung $A * x = b$ für jedes vorgegebene b eine eindeutige Lösung x .

Satz : Ist die **Determinante** der Matrix A ungleich 0, dann existiert die **inverse Matrix** von A , d.h. es gibt eine Matrix $B = A^{-1}$ mit der Eigenschaft

$$A^{-1} * A = A * A^{-1} = E \quad , \quad \text{wobei } E \text{ die Einheitsmatrix ist.}$$
$$B * A = A * B = E$$

Wichtige MATLAB-Funktionen für quadratische Matrizen :

$d = \text{det}(A)$: berechnet die Determinante der Matrix A

$B = \text{inv}(A)$: berechnet die inverse Matrix von A

Lösung des Gleichungssystems ?

$$A * x = b$$

```

disp('Gleichungssystem: A*x = b')
A = [-1 1; 1 1]
b = [2; 5]

disp('Inverse von A berechnen')
IA = inv(A)

disp('Gleichungssystem lösen')
x = IA*b
x = inv(A)*b

disp('Probe: A*x = b')
b1 = A*x

disp('Probe: ist IA Inverse?')
E1 = A*IA
E2 = IA*A

disp('Determinante von A und IA')
D1 = det(A)
D2 = det(IA)
D3 = det(A)*det(IA)

```

lingln1.m

```

A = -1    1
     1    1
b =  2
     5

Inverse von A berechnen
IA = -0.500...  0.500...
      0.500...  0.500...

Gleichungssystem lösen
x = 1.500
     3.500

Probe: A*x = b
b1 =  2
     5

Probe: ist IA Inverse?
E1 =  1    0
     0    1
E2 =  1    0
     0    1

Determinante von A und IA
D1 = -2
D2 = -0.5000...
D3 = 1

```

K7_8 7.1 Gleichungssysteme und Quadratische Matrizen

```
disp('Gleichungssystem A*x = b')
A = [-1 1; -2 2]
b = [0; 4]

disp('Determinante von A')
d = det(A)

disp('Inverse von A berechnen')
IA = inv(A)

disp('Gleichungssystem lösen')
x = IA*b
```

lingln2.m

```
A = -1    1
     -2    2
b =  0
     4
Determinante von A
d = 0
Inverse von A berechnen
Warning: Matrix is singular
to working precision.
> In lingln2 at 10
   In run at 57
IA =
     Inf     Inf
     Inf     Inf
Gleichungssystem lösen
x =
     NaN
     NaN
     ...
NaN Not a Number
```

Ein lineares Gleichungssystem mit Hilfe der inversen Matrix zu lösen ist zwar sehr einfach zu programmieren und funktioniert auch meistens für n Gleichungen mit n Unbekannten. Trotzdem wird dieses Verfahren nur sehr selten angewendet, da es eine Reihe von Problemen gibt :

- Die Berechnung der **inversen Matrix** ist für großes n sehr **aufwendig**. Es gibt eine Reihe von Verfahren zur Lösung von Gleichungssystemen, die wesentlich effektiver (weniger Rechenzeit, weniger Speicherplatz) arbeiten.
- Ist die Determinante einer Matrix 0, dann existiert gar **keine inverse Matrix**. Das Gleichungssystem kann in diesem Fall Lösungen besitzen oder auch nicht.
- Man möchte auch Gleichungssysteme der Art „ **m Gleichungen mit n Unbekannten**“ lösen – **überbestimmte** ($m > n$) und **unterbestimmte** ($m < n$) Gleichungssysteme. In diesem Fall gibt es gar keine inverse Matrix.

Um diese Probleme ganz allgemein und effektiv zu lösen hat man bei MATLAB die sogenannte **Linksdivision** eingeführt :

- die Funktion **mldivide**
- den Operator **** (Backslash)

Der Operator ist nur eine Kurzschreibweise für den Funktionsaufruf.

Aufgabe :

A sei eine $m \times n$ -Matrix und **b** ein Spaltenvektor mit m Zeilen. Gesucht ist ein Vektor **x** mit n Zeilen, der folgende Gleichung löst : $A \cdot x = b$

Lösung :

Die MATLAB-Funktion `mldivide` berechnet die Lösung der obigen Aufgabe.

$$x = \text{mldivide}(A, b)$$

Um den Funktionsaufruf kompakter zu schreiben, wurde ein neuer Operator eingeführt, nämlich die Linksddivision – **Operatorsymbol **

$$x = A \setminus b$$

Beachte : $x = \text{mldivide}(A, b)$ und $x = A \setminus b$ sind vollkommen identisch!

Je nach dem, welche Eigenschaften die Matrix **A** besitzt, wendet MATLAB im Hintergrund unterschiedliche Verfahren an, um das Gleichungssystem $A \cdot x = b$ möglichst effektiv zu lösen. Die Details sind in der MATLAB-Dokumentation beschrieben (siehe `mldivide` algorithm).

Frage : Welche Lösung berechnet MATLAB eigentlich ?

Manchmal gibt es ja gar keine Lösung ! MATLAB kann aber trotzdem eine Lösung zurückgeben!

$$A*x = b \quad : \quad x = A \setminus b \quad \quad x = \text{mldivide}(A, b)$$

Welche Lösung berechnet MATLAB ?

- $m = n$: MATLAB sucht eine **exakte Lösung**
MATLAB gibt eine Warnung aus , wenn die **Lösung „problematisch“** ist, z.B. wenn die Determinante der Matrix A fast 0 ist – siehe Beispiel
- $m > n$: MATLAB sucht eine **least-square solution**. Gibt es eine exakte Lösung, dann wird diese zurückgegeben. Gibt es keine exakte Lösung, dann wird eine Lösung x des Gleichungssystems $A*x = b'$ zurückgegeben, so dass die Differenz $b' - b$ „so klein wie möglich“ wird. D.h. man sucht ein b' in der Nähe von b , für das die Gleichung gelöst werden kann und gibt diese Lösung zurück.
- $m < n$: MATLAB sucht eine Lösung, bei der höchstens m Komponenten von 0 verschieden sind.

Bemerkung :

Ist A eine $n*n$ -Matrix und $\det(A) \neq 0$, dann ist die Matrix $A \setminus$ „identisch mit der inversen Matrix“ von A . Es gilt : $A \setminus A = E = A^{-1} * A$.

MATLAB berechnet aber die Inverse der Matrix A bei der Lösung von Gleichungssystemen gar nicht sondern löst das Gleichungssystem wesentlich effektiver.

$$A \cdot x = b \quad x = A \setminus b$$

Zwei Beispiele mit problematischen Lösungen

```
> A = [ 1 2; 2 4 ]
A =
     1     2
     2     4
> det(A)
ans =
     0
> b = [ 3; 4]
b =
     3
     4
> x = A\b %y=mldivide(A,b)
Warning: Matrix is singular
to working precision.
x =
     Inf
    -Inf
```

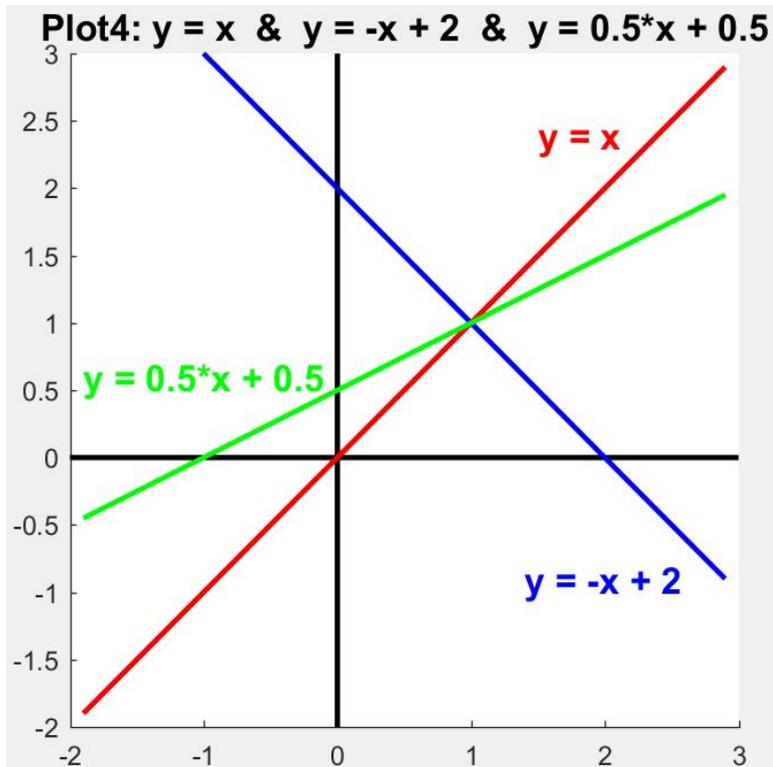
```
> A(2,2) = 3.99999999999999995
A =
     1.0000     2.0000
     2.0000     4.0000
> d=det(A)
d =
    -8.8818e-16
> x = A\b % y= mldivide(A,b)
Warning: Matrix is close to singular
or badly scaled. Results maybe
inaccurate. RCOND = 2.467162e-17.
x =
    1.0e+15 *
    -4.5036
     2.2518
> b1 = A*x % Probe
b1 =
     3.0000
     5.0000
```

3 Gleichungen mit 2 Unbekannten – hier gibt es eine **exakte Lösung**

```

A = [-1 1; 1 1; -0.5 1]
b = [0; 2; 0.5]
disp('Gleichungssystem lösen')
x = A\b      %x = mldivide(A,b)
disp('Probe: b1=A*x')
b1 = A*x

```



```

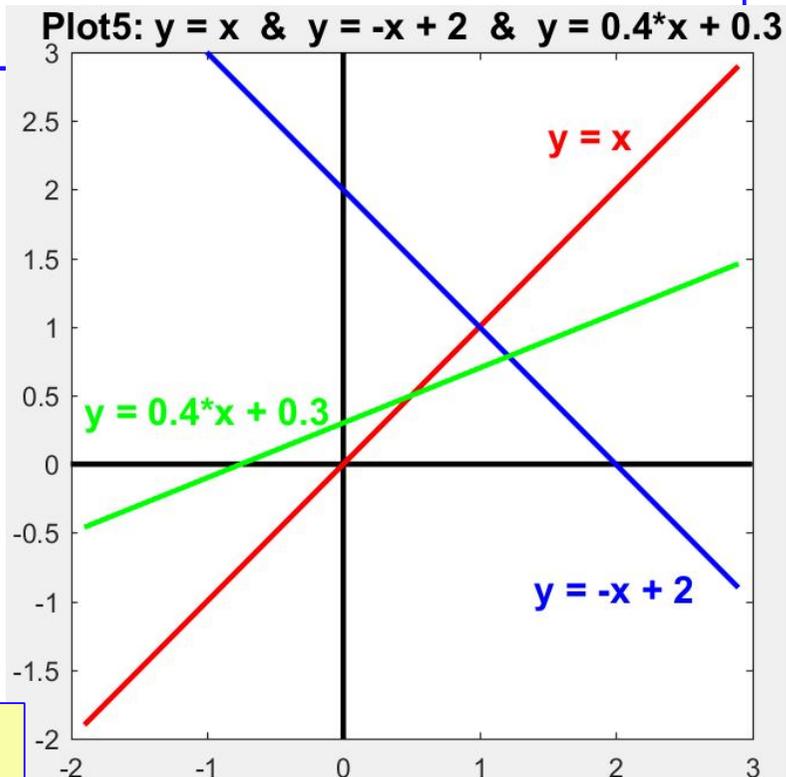
A =
    -1.0000    1.0000
     1.0000    1.0000
    -0.5000    1.0000
b =
     0
     2.0000
     0.5000
Gleichungssystem lösen
x =
     1.0000
     1.0000
Probe: b1=A*x
b1 =
    -0.0000
     2.0000
     0.5000

```

3 Gleichungen mit 2 Unbekannten – hier existiert **keine exakte Lösung**

MATLAB berechnet eine least-square-solution (eine bestmögliche Lösung)

```
A = [-1 1; 1 1; -0.4 1]
b = [0; 2; 0.3]
disp('Gleichungssystem lösen')
x = A\b
%x = mldivide(A,b)
disp('Probe: A*x=b')
b1 = A*x
```



lingln4.m

```
A =
    -1.0000    1.0000
     1.0000    1.0000
    -0.4000    1.0000
b =
         0
     2.0000
     0.3000
Gleichungssystem lösen
x =
     1.0380
     0.9051
Probe: b1=A*x
b1 =
    -0.1329
     1.9430
     0.4899
```

Definition :

A sei eine $n \times n$ -Matrix. Ein Vektor **x** heißt **Eigenvektor** der Matrix **A**, wenn es eine (reelle oder komplexe) Zahl λ gibt, für die gilt : $A * x = \lambda * x$
Die Zahl λ heißt **Eigenwert** zum Eigenvektor **x**.

Bemerkungen :

- Die Bestimmung der Eigenwerte und Eigenvektoren der Gleichung $A * x = \lambda * x$ nennt man **spezielles** oder **gewöhnliches Eigenwertproblem**.
- Satz : Ist die Matrix **A** **symmetrisch**, dann gibt es **n reelle** Eigenwerte und **n linear unabhängige Eigenvektoren**.
- Ist die Matrix **A** nicht symmetrisch, können Eigenwerte auch komplex sein. Es muss kein vollständiges System von n linear unabhängigen Eigenvektoren geben.

MATLAB :

Die Funktion **eig** berechnet Eigenwerte und Eigenvektoren einer Matrix.

$EW = eig(A)$ is a **vector** containing the **eigenvalues** of a square matrix **A**.

$[V,D] = eig(A)$ produces a **diagonal matrix D** of eigenvalues and a full **matrix V** whose **columns** are the corresponding **eigenvectors** so that

$A * V = V * D$ d. h. es gilt $A * V(:,k) = D(k,k) * V(:,k)$

$V(:,k)$ k-ter **Eigenvektor** der Matrix **A**

$D(k,k)$ k-ter **Eigenwert** der Matrix **A**

Beispiel :

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \end{bmatrix} \quad A \cdot \vec{x} = \lambda \cdot \vec{x} \quad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \lambda \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

```
> EW = eig(A)
EW =
   -0.4203
    0.2336
   10.1867
> [ V D ] = eig(A);
> V
V =
  -0.9269    0.0856    0.3653
   0.1307   -0.8389    0.5283
   0.3517    0.5374    0.7665
> D
D =
  -0.4203         0         0
         0    0.2336         0
         0         0   10.1867
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} -0.9269 \\ 0.1307 \\ 0.3517 \end{bmatrix} = -0.4203 \cdot \begin{bmatrix} -0.9269 \\ 0.1307 \\ 0.3517 \end{bmatrix}$$

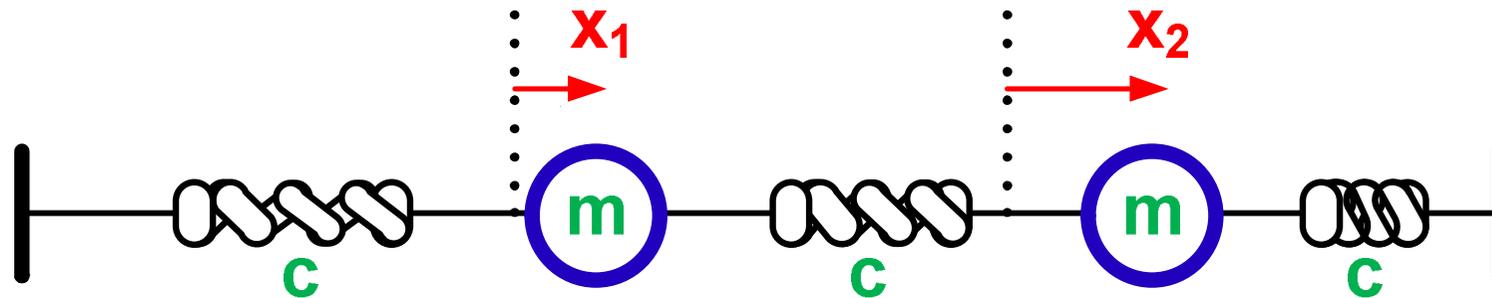
$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 0.0856 \\ -0.8389 \\ 0.5374 \end{bmatrix} = 0.2336 \cdot \begin{bmatrix} 0.0856 \\ -0.8389 \\ 0.5374 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 0.3653 \\ 0.5283 \\ 0.7665 \end{bmatrix} = 10.1867 \cdot \begin{bmatrix} 0.3653 \\ 0.5283 \\ 0.7665 \end{bmatrix}$$

$$A * V(:,3) = D(3,3) * V(:,3)$$

eigenwertproblem.m

System von zwei Massen, die durch **lineare (!!)** Federn miteinander gekoppelt sind.



Bewegungsgleichungen :

$$m \cdot \ddot{x}_1(t) = -c \cdot x_1(t) - c \cdot [x_1(t) - x_2(t)]$$

$$m \cdot \ddot{x}_2(t) = -c \cdot [x_2(t) - x_1(t)] - c \cdot x_2(t)$$

Anfangsbedingungen :

$$x_1(t=0) = x_{10} \quad \dot{x}_1(t=0) = v_{10}$$

$$x_2(t=0) = x_{20} \quad \dot{x}_2(t=0) = v_{20}$$

Gesucht ist die Lösung für das Anfangswertproblem, d.h. die Werte x_{10} und x_{20} und v_{10} und v_{20} sind zur Zeit $t=0$ vorgegeben; meist ist $v_{10}=v_{20} = 0$

Lösungsschritte :

- Schreibe das Problem (DGL lösen) in ein Eigenwert- Eigenvektorproblem um
- Berechne die **Eigenwerte** und die **Eigenvektoren** mit MATLAB
- Berechne den Anteil der jeweiligen Eigenvektoren an der Gesamtlösung aus den **Anfangsbedingungen** mit MATLAB – **lineare Gleichungssysteme** lösen
- Lösung ergibt sich durch Überlagerung der Eigenvektoren (Eigenschwingungen)

Bei **linearen Problemen** kann dieses Verfahren immer angewendet werden.

**Spezielles Eigenwert-
Eigenvektor-Problem :**

$$\begin{bmatrix} 2 \cdot c/m & -c/m \\ -c/m & 2 \cdot c/m \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \omega^2 \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$A \cdot x = \lambda \cdot x$$

Analytische Lösung – ohne MATLAB als Hausaufgabe :

Zwei Eigenwerte:

$$\lambda_1 = c/m \quad \lambda_2 = 3 \cdot c/m$$

Zwei Eigenfrequenzen:

$$\omega_1 = \sqrt{c/m} \quad \omega_2 = \sqrt{3 \cdot c/m}$$

Zwei Eigenvektoren:

$$x_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad x_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Normierte Eigenvektoren:

$$x_1 = \begin{bmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \end{bmatrix} \quad x_2 = \begin{bmatrix} \sqrt{2}/2 \\ -\sqrt{2}/2 \end{bmatrix}$$

$$\vec{x}(t) = a_1 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \cos(\sqrt{c/m} \cdot t) + a_2 \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} \cdot \cos(\sqrt{3c/m} \cdot t) + b_1 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \sin(\sqrt{c/m} \cdot t) + b_2 \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} \cdot \sin(\sqrt{3c/m} \cdot t)$$

Allgemeine Lösung

**a_1, a_2, b_1 und b_2 aus den Anfangsbedingungen bestimmen
Löse lineare Gleichungssysteme**

$$\vec{x}(t=0) = \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = a_1 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + a_2 \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

$$\begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

$$\vec{v}(t=0) = \begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix} = \omega_1 \cdot b_1 \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \omega_2 \cdot b_2 \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} \omega_1 \cdot b_1 \\ \omega_2 \cdot b_2 \end{bmatrix}$$

$$\begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} \omega_1 \cdot b_1 \\ \omega_2 \cdot b_2 \end{bmatrix}$$

Lösung mit Hilfe von MATLAB

$$\begin{bmatrix} 2 \cdot c/m & -c/m \\ -c/m & 2 \cdot c/m \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \omega^2 \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$A \cdot x = \lambda \cdot x$$

Löse das Eigenwert- Eigenvektorprobleme : $[V, D] = \text{eig}(A)$

$$V = \begin{bmatrix} v_{1,1} & v_{2,1} \\ v_{2,1} & v_{2,2} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \quad D = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} = \begin{bmatrix} \omega_1^2 & 0 \\ 0 & \omega_2^2 \end{bmatrix}$$

Für die Eigenwerte und die zugehörigen Eigenvektoren gilt :

$$A \cdot V(:,1) = A \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \lambda_1 \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} \quad A \cdot V(:,2) = A \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} = \lambda_2 \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix}$$

$$\vec{x}(t) = a_1 \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} \cdot \cos(\omega_1 \cdot t) + a_2 \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} \cdot \cos(\omega_2 \cdot t) + b_1 \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} \cdot \sin(\omega_1 \cdot t) + b_2 \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} \cdot \sin(\omega_2 \cdot t)$$

$$\vec{x}(t=0) = \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = a_1 \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} + a_2 \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

$$\begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

$$\vec{v}(0) = \begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix} = b_1 \omega_1 \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} + b_2 \omega_2 \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \cdot \begin{bmatrix} b_1 \omega_1 \\ b_2 \omega_2 \end{bmatrix}$$

$$\begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \cdot \begin{bmatrix} b_1 \omega_1 \\ b_2 \omega_2 \end{bmatrix}$$

Lösungsschritte mit MATLAB

1. 'Steifigkeitsmatrix' definieren

$$A = \begin{bmatrix} 2 \cdot c/m & -c/m \\ -c/m & 2 \cdot c/m \end{bmatrix}$$

2. Eigenwert- Eigenvektorproblem lösen $[V, D] = \text{eig}(A)$

$$V = \begin{bmatrix} v_{1,1} & v_{2,1} \\ v_{2,1} & v_{2,2} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix}$$

$$\begin{bmatrix} \omega_1 & 0 \\ 0 & \omega_2 \end{bmatrix} = \text{sqrt}(\text{diag}(D))$$

3. Anteile der Eigenschwingungen aus den Anfangsbedingungen berechnen
Anfangsauslenkung und Anfangsgeschwindigkeit nach Eigenvektoren zerlegen

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = V \setminus \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1/\omega_1 \\ 1/\omega_2 \end{bmatrix} .* \text{mldivide}(V, \begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix})$$

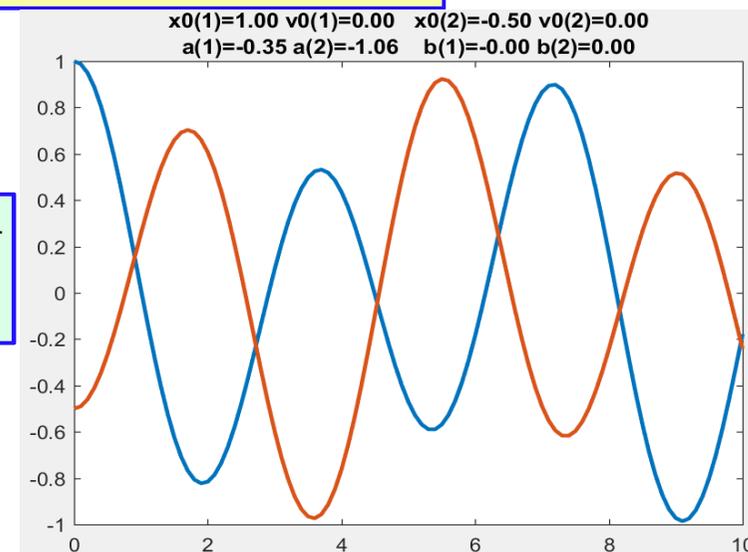
4. Lösung für das Anfangswertproblem :

$$t = 0:0.1:10$$

$$x = V(:,1) * (a_1 * \cos(\omega_1 * t) + b_1 * \sin(\omega_1 * t)) + \\ V(:,2) * (a_2 * \cos(\omega_2 * t) + b_2 * \sin(\omega_2 * t))$$

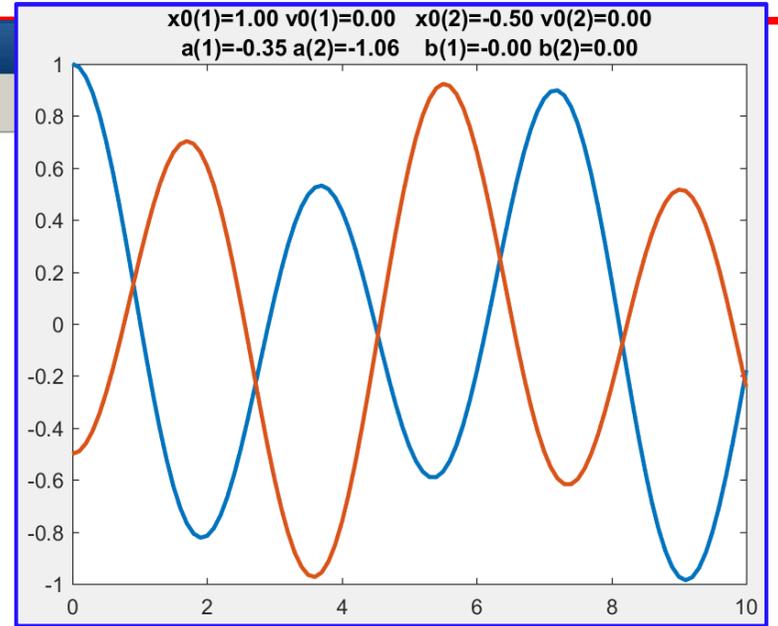
$$\text{plot}(t,x)$$

Beachte: x ist eine 2*101-Matrix!

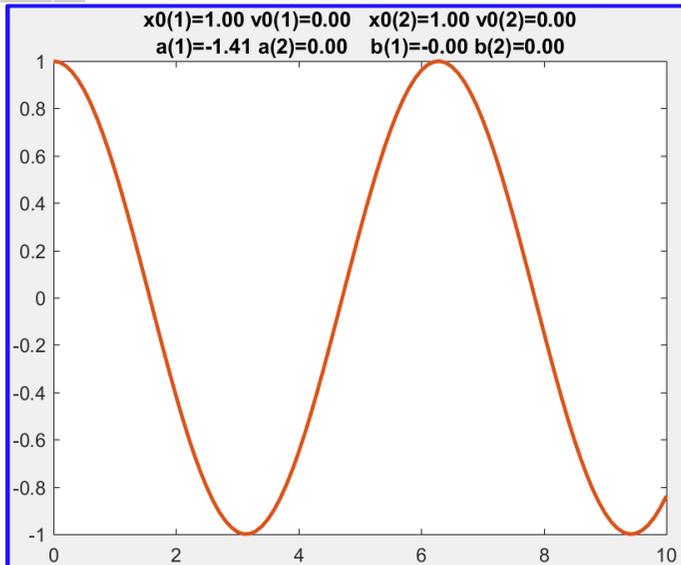


```

Editor - U:\Matlab\kette2mScr.m
kette2mScr.m x +
1 cm = 1
2 x0 = [1; -0.5]
3 v0 = [0; 0]
4 A = [2*cm -cm; -cm 2*cm]
5 [V,D] = eig(A);
6 w = sqrt(diag(D)); ← Frequenzen
7 a = V\x0
8 b = (1./w) .* (V\v0)
9 t=0:0.1:10;
10 x = V(:,1)*( a(1)*cos(w(1)*t) + b(1)*sin(w(1)*t) )+...
11      V(:,2)*( a(2)*cos(w(2)*t) + b(2)*sin(w(2)*t) );
12 plot(t,x,'linewidth',2)
    
```

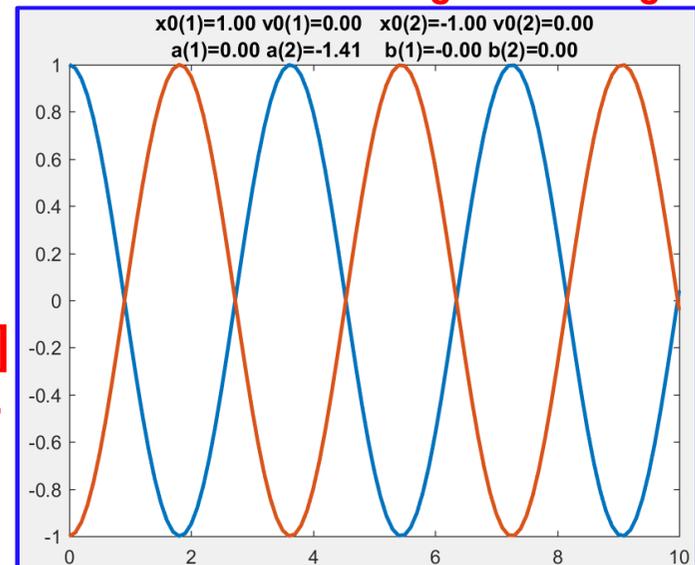


Überlagerung zweier Eigenschwingungen



**x0=[1; 1]
Erste Eigen-
schwingung**

**x0=[1; -1]
Zweite Eigen-
schwingung**



Definition : verallgemeinertes Eigenwertproblem

A, B seien $n \times n$ -Matrizen. Ein Vektor x heißt **Eigenvektor** des verallgemeinerten Eigenwertproblems, wenn es eine (reelle oder komplexe) Zahl λ gibt, für die gilt : $A * x = \lambda * B * x$. Die Zahl λ heißt Eigenwert zum Eigenvektor x .

Bemerkung :

Existiert die inverse Matrix von B , dann lässt sich das verallgemeinerte Eigenwertproblem in ein spezielles Eigenwertproblem umschreiben

$$\begin{aligned}
 B^{-1} * \quad & | \quad A * x = \lambda * B * x \\
 & B^{-1} * A * x = \lambda * B^{-1} * B * x = \lambda * E * x \\
 & (B^{-1} * A) * x = \lambda * x
 \end{aligned}$$

MATLAB :

Die Funktion **eig** berechnet auch Eigenwerte und Eigenvektoren des verallgemeinerten Eigenwertproblems. Zwei Matrizen werden als Parameter übergeben .

$EV = \text{eig}(A, B)$ is a vector containing the generalized **eigenvalues** of square matrices A and B .

$[V, D] = \text{eig}(A, B)$ produces a **diagonal matrix D** of **generalized eigenvalues** and a **full matrix V** whose **columns** are the corresponding **eigenvectors** so that

$$A * V = B * V * D \quad \text{d. h.} \quad A * V(:, k) = D(k, k) * B * V(:, k)$$

Aufgabe 1

Bestimmen Sie die Lösung des folgenden Gleichungssystems :

$$-x + y + z = 0$$

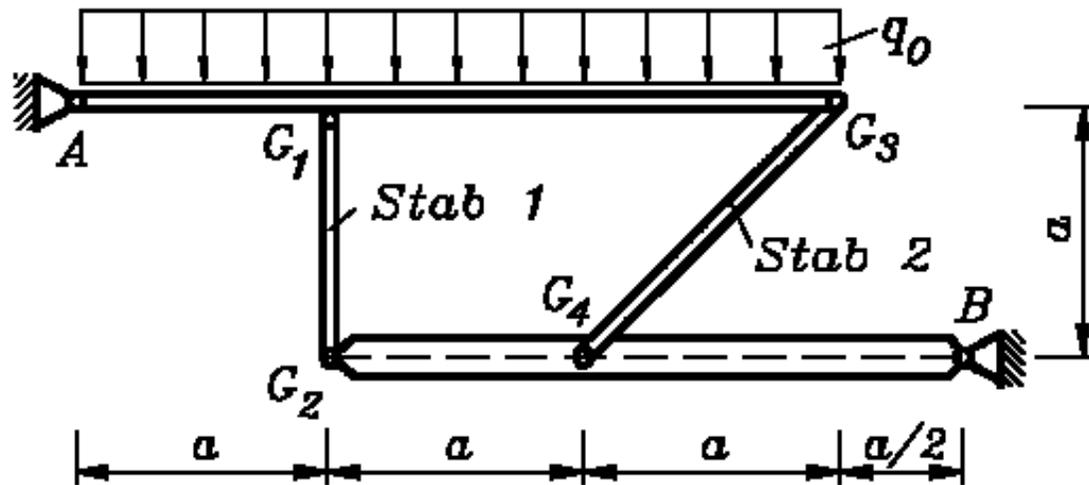
$$x + 2y + z = 2$$

$$3x + y + z = 3$$

Verwenden Sie einmal die Funktion `mldivide` und einmal den Operator für die Linksdivision.

Zeigen Sie, dass ihr Ergebnis das Gleichungssystem tatsächlich löst. Wie kann man ganz einfach überprüfen, ob zwei Vektoren gleich sind (bis auf Rundungsfehler)?

Warum ist die Lösung eindeutig? Wie kann man das mit MATLAB überprüfen? Schreiben Sie alle Befehle zur Lösung dieser Aufgaben in ein geeignetes Skript.

Aufgabe 2 : Dankert/Dankert: Technische Mechanik www.tm-aktuell.de


Für die Berechnung der Lagerreaktionen und Stabkräfte im nebenstehenden System ergibt sich auf Basis der Gleichgewichtsbedingungen folgendes Gleichungssystem in Matrixform :

$$\begin{bmatrix}
 1 & 0 & 0 & 0 & 0 & -\cos(\alpha) \\
 0 & 1 & 0 & 0 & -1 & -\sin(\alpha) \\
 0 & 3 & 0 & 0 & -2 & 0 \\
 0 & 0 & 1 & 0 & 0 & \cos(\alpha) \\
 0 & 0 & 0 & 1 & 1 & \sin(\alpha) \\
 0 & 0 & 0 & -1.5 & 1 & 0
 \end{bmatrix}
 * \begin{bmatrix}
 F_{AH} \\
 F_{AV} \\
 F_{BH} \\
 F_{BV} \\
 F_{S1} \\
 F_{S2}
 \end{bmatrix}
 = \begin{bmatrix}
 0 \\
 3 \cdot q_0 \cdot a \\
 4.5 \cdot q_0 \cdot a \\
 0 \\
 0 \\
 0
 \end{bmatrix}$$

Schreiben Sie ein Skript, das die Kräfte F für das obige Gleichungssystem berechnet. Wählen Sie für den Winkel 15° und für $q_0 \cdot a$ den Wert 0.1 .

Aufgabe 3

Laden Sie die Datei `lager.m` in den Editor und versuchen Sie das Programm zu verstehen. Es werden die Gleichungen aus Aufgabe 2 gelöst. Starten Sie das Programm für verschiedene Parameter.

Welche Funktion hat die Variable `ausgabe`? Versuchen Sie die Ausgabeanweisung für die Matrix `M` zu verstehen. Warum muss man die transponierte Matrix von `M` in der `fprintf`-Anweisung verwenden? Was ergibt sich, wenn man die Matrix `M` verwendet?

Für welche Winkel ist die Matrix `M` singulär? Wie reagiert das Programm darauf? Zeigen Sie, dass die Berechnungen mit Hilfe der Linksdivision und die Berechnungen mit der inversen Matrix von `M` die gleichen Ergebnisse liefern.

Aufgabe 4

Laden Sie die Datei `lagerkraft.m` in den Editor und versuchen Sie das Programm zu verstehen. Welche Vorteile und Nachteile besitzt diese Funktion gegenüber der Funktion `lager.m`? Starten Sie das Programm.

Aufgabe 5

Laden Sie die Datei **eigenwertproblem.m** in den Editor und versuchen Sie das Programm zu verstehen. Es werden die Eigenwerte und die Eigenvektoren einer **symmetrischen Matrix** berechnet. Weiterhin wird gezeigt, dass die berechneten Eigenvektoren tatsächlich Eigenvektoren sind. Führen Sie das Skript aus und versuchen Sie die Ausgabe zu verstehen.

Kapitel 8 Numerische Lösung von Differentialgleichungen

8.1 Differentialgleichungen – Analytische und Numerische Lösung

8.2 Begriffe

8.3 Einfache Beispiele

8.4 Standardform einer DGL für numerische Lösungsverfahren

8.5 Eulersches Polygonzugverfahren

8.6 Runge-Kutta-Verfahren

8.7 Runge-Kutta-Verfahren mit MATLAB

8.8 Hausaufgaben

K8_2 8.1 Differentialgleichungen – Analytische und Numerische Lösung

Gleichungen :

$$y(x) = x^3 - 4 \cdot x^2 + 7 = 0$$

$$7 \cdot x_1 + 8 \cdot x_2 = 7$$

$$8 \cdot x_1 + 3 \cdot x_2 = 6$$

Gleichungen dieser Form zu lösen bedeutet „suche die Nullstellen eines Polynoms“ bzw. „suche die Lösungen für ein System von linearen Gleichungen“. Als Ergebnis erhält man reelle oder komplexe Zahlen oder auch keine Lösung, falls die Gleichungen nicht lösbar sind.

Differentialgleichung (DGL) :

$$\frac{d^2 y}{dt^2} = -\frac{k}{m} \cdot y(t)$$

„Schwingung“

Eine DGL stellt eine Beziehung zwischen den Ableitungen einer Funktion und der Funktion selbst dar. Im obigen Beispiel werden Funktion gesucht, bei denen die zweite Ableitung proportional zum Funktionswert ist. Welche Funktionen haben diese Eigenschaft? Die Lösung einer DGL ist keine Zahl sondern eine Menge von Funktionen mit gewissen Eigenschaften. Diese Eigenschaften werden durch die DGL festgelegt.

Die obige DGL kann **analytisch** gelöst werden, d.h. man kann eine Lösung der DGL mit Hilfe von bekannten Funktionen angeben. Die allgemeine Lösung lautet :

$$y(t) = a \cdot \sin(\omega \cdot t) + b \cdot \cos(\omega \cdot t) \quad \omega^2 = k/m$$

$$y(t=0) = y_0 \quad \dot{y}(t=0) = v_0$$

Die Größen **a** und **b** müssen durch zusätzliche Bedingungen festgelegt werden, z.B. durch Anfangs- oder Randbedingungen.

Numerische Lösung von Differentialgleichungen :

Eine analytische Lösung einer DGL (d.h. die Angabe der Lösungsfunktion durch bekannte Funktionen) ist häufig nicht möglich, insbesondere bei nichtlinearen DGLn. Selbst wenn eine DGL analytisch lösbar ist, kann die Lösung so komplex sein, dass man mit der Lösung nur wenig anfangen kann.

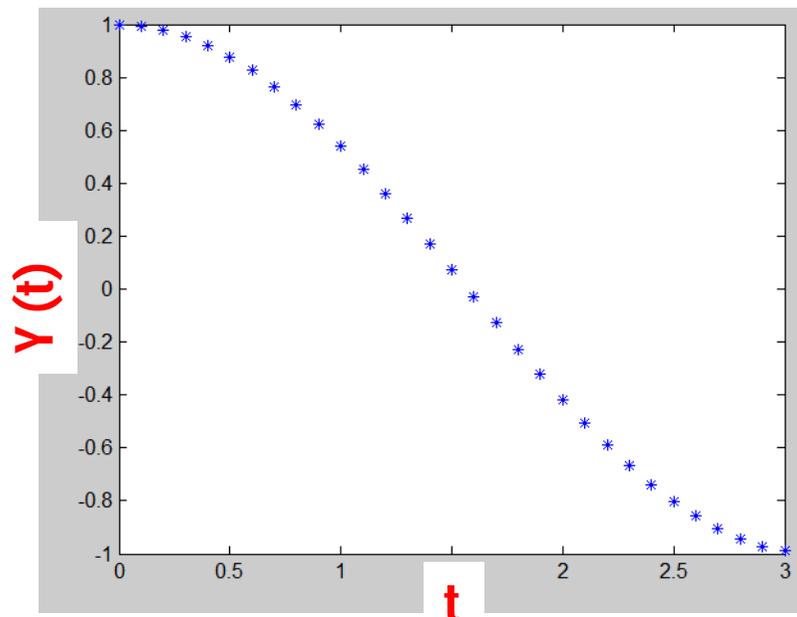
In der Praxis werden DGLn daher oft **numerisch** gelöst. Als Ergebnis erhält man **keine Funktion sondern Zahlenwerte**, die die Funktionswerte der „unbekannten Lösung“ näherungsweise darstellen. Diese Zahlenwerte werden meist in Vektoren oder Matrizen abgespeichert. Mit diesen Zahlenwerten lässt sich die Lösung graph. darstellen.

$$\frac{d^2y}{dt^2} = -\frac{k}{m} \cdot y(t)$$

DGL

	0.00		1.00
	0.10		0.955
	0.20		0.825
t =	0.30	y =	0.621
	0.40		0.362
	0.50		0.070

Numerische Lösung:
Vektoren t und y



Gewöhnliche Differentialgleichungen n-ter Ordnung

- **Gewöhnliche** Differentialgleichungen (**Ordinary Differential Equations - ODEs**)

In der DGL kommt nur **eine unabhängige Variable** vor, sowie eine unbekannte Funktion und deren Ableitungen. Die unabhängige Variable ist meist der Ort x oder die Zeit t . Neben der unabhängigen Variablen können in der DGL auch noch Parameter vorkommen und bekannte Funktionen, z.B. eine äußere Kraft. Die DGL für eine Schwingung enthält z.B. den Parameter $k/m = \omega^2$.

Bei **partiellen DGL (PDE)** hat man **mehrere unabhängige Variablen** z.B. x und t oder x, y, z und t .

Beispiel : Strömungsmechanik – PDEs werden hier nicht behandelt.

- **Ordnung** einer DGL

In einer **DGL n-ter Ordnung** kommen Ableitungen nach der unabhängigen Variablen bis zur Ordnung n vor.

Die DGL einer Schwingung ist eine DGL 2-ter Ordnung.

In der Praxis ist die Ordnung von DGLn häufig kleiner gleich 4.

Bei der numerischen Lösung von DGLn werden DGLn **n-ter Ordnung** meistens in ein **System von n DGLn erster Ordnung** umgeschrieben (**Standardform**).

Festlegung der freien Parameter in der allgemeinen Lösung einer DGL

Eine DGL kann mehrere unabhängige Lösungen besitzen. Beispiel :

$$\frac{d^2y}{dt^2} = -\frac{k}{m} \cdot y(t) \quad y_1(t) = a \cdot \sin(\omega \cdot t) \quad , \quad y_2(t) = b \cdot \cos(\omega \cdot t) \quad , \quad \omega^2 = \frac{k}{m}$$

Da es sich um eine **lineare DGL** handelt, ist auch die Summe der beiden Lösungen wieder eine Lösung. Die **allgemeine Lösung** lautet deshalb :

$$y(t) = y_1(t) + y_2(t) = a \cdot \sin(\omega \cdot t) + b \cdot \cos(\omega \cdot t)$$

Um die Größen **a** und **b** festzulegen sind weitere Bedingungen notwendig. In der Praxis treten meist zwei Arten von Bedingungen auf :

- **Anfangswertproblem**

Der Funktionswert und die Werte der Ableitungen sind zu einem bestimmten Zeitpunkt oder an einem bestimmten Ort vorgegeben. Bei einer Schwingung z.B.

$$y(t=0) = y_0 \quad \frac{dy}{dt}(t=0) = v_0 \quad \text{oder} \quad y(t=0) = y_0 \quad \frac{dy}{dt}(t=0) = 0$$

- **Randwertprobleme**

Funktionswerte und/oder Werte der Ableitungen sind zu unterschiedlichen Zeitpunkten oder an verschiedenen Orten vorgegeben, z.B.

$$y(x=0) = 0 \quad \text{und} \quad y(x=a) = 0$$

Beispiele :

a) $\frac{dy}{dx} = k$

b) $\frac{dy}{dx} = k \cdot y(x)$

c) $\frac{d^2y}{dt^2} = -\frac{k}{m} \cdot y(t)$

Lösungen :

Die ersten zwei Beispiele sind DGLn **erster Ordnung**. Das 3-te Beispiel ist eine DGL **2-ter Ordnung**. Alle Lösungen enthalten noch **freie Konstanten – a** bzw. **b**. Diese Konstanten werden durch Anfangsbedingungen festgelegt, d.h. durch Bedingungen bei $x=0$ bzw. $t=0$. Bei einer DGL erster Ordnung reicht eine Anfangsbedingung, bei einer DGL 2-ter Ordnung sind zwei Anfangsbedingungen notwendig.

a) $y(x = 0) = 3$

b) $y(x = 0) = 4$

c) $y(t = 0) = 3 \quad \dot{y}(t = 0) = 0$

Lösungen zu den vorgegebenen Anfangswerten :

Die Größe k in den Beispielen a) und b) ist ein **Parameter**, ebenso ω im Beispiel c). Für konkrete Werte von k und ω ergeben sich konkrete Lösungen.

Die numerische Lösung für das Anfangswertproblem einer gewöhnlichen Differentialgleichung erfolgt meist in folgenden Schritten:

1. DGL umschreiben in ein **System von DGLn erster Ordnung**
d.h. man bringt die DGL in eine Standardform
2. Geeignetes numerisches Lösungsverfahren auswählen
Es gibt verschiedene Verfahren zur Lösung eines Systems von DGLn erster Ordnung. Meist verwendet man sogenannte Runge-Kutta-Verfahren.
3. Numerische Lösungsverfahren programmieren oder ein bereits implementiertes Verfahren wählen (MATLAB liefert eine Vielzahl von Verfahren, die von Experten programmiert worden sind)
4. Setzen der Anfangsbedingungen und Parameter und dann das Numerische Lösungsverfahren aufrufen. Das Lösungsverfahren gibt einen Vektor zurück (dieser enthält die Werte der abhängigen Variablen) und eine Matrix mit n Spalten (n ist die Anzahl der DGLn erster Ordnung). Die Matrix enthält die entsprechenden Funktionswerte.
5. Lösung darstellen und überprüfen

Die DGL für eine Schwingung ist eine DGL 2-ter Ordnung. Diese DGL wird im ersten Schritt in ein **System von zwei gekoppelten DGLn erster Ordnung** umgeschrieben. Dazu wird eine neue Variable $v(t)$ eingeführt, die **erste Ableitung von $y(t)$** nach der Zeit t (d.h. $v(t)$ ist die Geschwindigkeit). In der umgeschriebenen Form der DGL kommen dann nur noch y und v sowie deren erste Ableitungen vor. Die zweite Ableitung von y wird durch die erste Ableitung von v ersetzt. Die beiden Variablen y und v können auch als Komponenten eines Vektors z mit zwei Komponenten aufgefasst werden, der von der Zeit t abhängt.

$$\frac{d^2 y}{dt^2} = -\frac{k}{m} \cdot y(t)$$

$$\begin{aligned} 1) \quad & \frac{dy}{dt} = v(t) \\ 2) \quad & \frac{dv}{dt} = -\frac{k}{m} \cdot y(t) \end{aligned}$$

$$\begin{bmatrix} y(t) \\ v(t) \end{bmatrix} = z(t) = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} z_1(t) \\ z_2(t) \end{bmatrix}$$

$$\begin{aligned} 1) \quad & \frac{dz_1(t)}{dt} = z_2(t) \\ 2) \quad & \frac{dz_2(t)}{dt} = -\frac{k}{m} \cdot z_1(t) \end{aligned}$$

$$\frac{d}{dt} \begin{bmatrix} y \\ v \end{bmatrix} = \begin{bmatrix} \dot{y} \\ \dot{v} \end{bmatrix} = f(t, y, v) = \begin{bmatrix} v \\ -k/m \cdot y \end{bmatrix}$$

$$\frac{dz}{dt} = \frac{d}{dt} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} \dot{z}_1 \\ \dot{z}_2 \end{bmatrix} = f(t, z_1, z_2) = f(t, z) = \begin{bmatrix} z_2 \\ -k/m \cdot z_1 \end{bmatrix}$$

Zwei gekoppelte DGLn erster Ordnung

Schreiben Sie die DGL für eine **gedämpfte Schwingung mit äußerer Anregung** in ein System von DGLn erster Ordnung um, ebenso die Anfangsbedingungen.

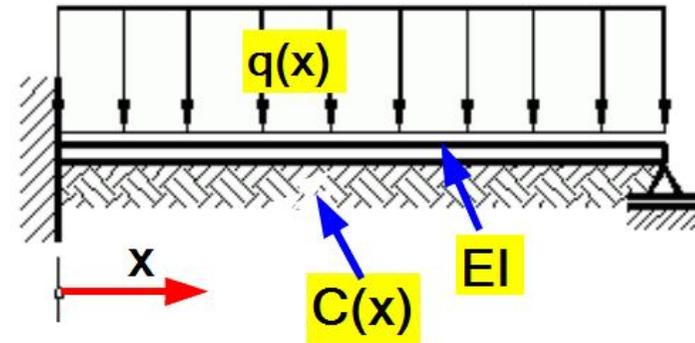
$$\ddot{y}(t) + 2 \cdot \delta \cdot \dot{y}(t) + \omega_0^2 \cdot y(t) = F \cdot \cos(\omega \cdot t)$$

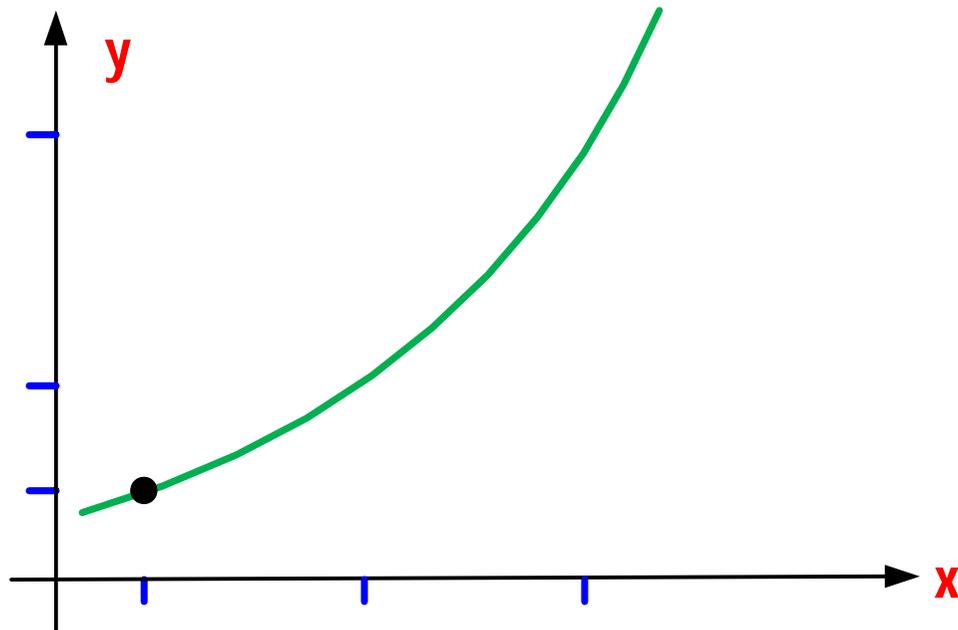
$$y(t = 0) = y_0 \quad \dot{y}(t = 0) = v_0$$

Schreiben Sie folgende DGL 4-ter Ordnung (elastisch gebetteter Träger) in ein System von vier DGLn erster Ordnung um.

- EI Biegesteifigkeit
- $C(x)$ Steifigkeit der elastischen Bettung
- $q(x)$ Streckenlast

$$EI \cdot \frac{d^{(4)}y}{dx^4} + C(x) \cdot y(x) = q(x)$$





Numerische Lösung des
Anfangswertproblems :

$$\text{DGL: } \frac{dy}{dx} = f(x, y) \quad y(x_0) = y_0$$

Konkret :

$$\frac{dy}{dx} = x \cdot y^2 + 2 \quad , y(x_0 = 1) = 3$$

Der Funktionswert y_1 an der Stelle x_1 berechnet sich näherungsweise mit :

x_1 und y_1 werden danach zur Berechnung von $y_2 = y(x_2)$ verwendet, usw.

**Eulersches
Polygonzugverfahren**

Eulersches Polygonzugverfahren für eine DGL erster Ordnung :

$$y_{n+1} = y_n + f(x_n, y_n) \cdot (x_{n+1} - x_n) \quad n = 0, 1, 2 \dots$$

Eulersches Polygonzugverfahren für zwei DGLn erster Ordnung :

Aufgabe :

Stellen Sie die Gleichungen für das Eulersche Polygonzugverfahren zur Lösung der DGLn für eine Schwingung auf.

Ersetze die Differentialquotienten (Ableitungen) durch Differenzenquotienten.

DGL einer Schwingung :

$$\frac{d^2y}{dt^2} + \frac{k}{m} \cdot y(t) = 0$$

AW:

$$y(t = 0) = 5 \quad \dot{y}(t = 0) = 0$$

DGLn erster Ordnung :

$$1) \quad \frac{dy}{dt} = v(t)$$

$$2) \quad \frac{dv}{dt} = -\frac{k}{m} \cdot y(t)$$

$$y(t = 0) = 5$$

$$v(t = 0) = 0$$

Schreibe ein Skript, das das Anfangswertproblem für die DGL einer Schwingung nach dem Eulerverfahren im Zeitintervall $[0, 10]$ löst. Lösung graph. darstellen.

$$k = 1, \quad m = 2, \quad y(t=0) = 5, \quad v(t=0) = 0$$



Das Euler-Verfahren wird in der Praxis nicht verwendet, weil es bei vielen DGLn nicht funktioniert und weil es bessere Verfahren gibt. Standardverfahren zur Lösung von DGLn sind sogenannte **Runge-Kutta-Verfahren**.

Anfangswertproblem :
$$\frac{dy}{dx} = f(x, y) \quad y_0 = y(x_0)$$

Eulerverfahren :

Die **Steigung** $f(x, y)$ im Bereich $[x_0, x_1]$ wird durch $y_1 = y_0 + h \cdot f(x_0, y_0) \quad h = x_1 - x_0$ eine **Konstante ersetzt**, nämlich die Steigung im Punkt (x_0, y_0) , d.h. durch den Wert $f(x_0, y_0)$. Die Steigung ändert sich aber im Bereich $[x_0, x_1]$.

Runge-Kutta-Verfahren :

Die Steigung $f(x, y)$ im Bereich $[x_0, x_1]$ wird durch eine „**gemittelte Steigung**“ ersetzt. Beim klassischen Runge-Kutta-Verfahren berechnet man diesen Mittelwert aus den Steigungen an vier verschiedenen Stellen im Intervall wie folgt ($h = x_1 - x_0$):

$$y_1 = y_0 + (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) / 6 = y_0 + h \cdot \bar{f}$$

Berechne zuerst k_1 , daraus k_2 , daraus k_3 ...
Aus den 4 Werten k_i wird der neue Funktionswert y_1 berechnet.

$$k_1 = h \cdot f(x_0, y_0)$$

$$k_2 = h \cdot f(x_0 + h/2, y_0 + k_1/2)$$

$$k_3 = h \cdot f(x_0 + h/2, y_0 + k_2/2)$$

$$k_4 = h \cdot f(x_0 + h, y_0 + k_3)$$

Runge-Kutta-Verfahren :

$$y_1 = y_0 + (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) / 6$$

$$k_1 = h \cdot f(x_0, y_0) = h \cdot m_1$$

$$k_2 = h \cdot f(x_0 + h/2, y_0 + h \cdot m_1/2) = h \cdot f(x_0 + h/2, y_A) = h \cdot m_2$$

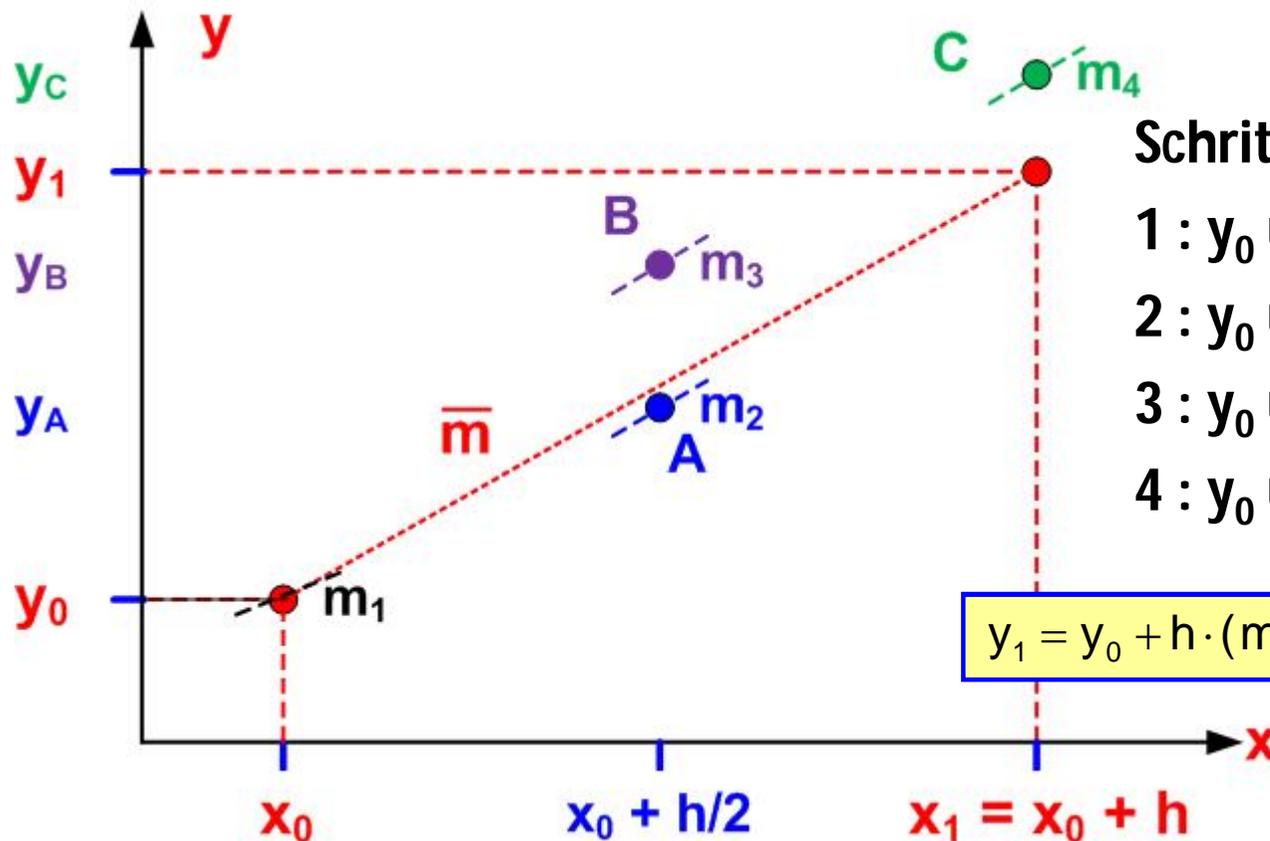
$$k_3 = h \cdot f(x_0 + h/2, y_0 + h \cdot m_2/2) = h \cdot f(x_0 + h/2, y_B) = h \cdot m_3$$

$$k_4 = h \cdot f(x_0 + h, y_0 + h \cdot m_3) = h \cdot f(x_0 + h, y_C) = h \cdot m_4$$

$$y_A = y_0 + h \cdot m_1 / 2$$

$$y_B = y_0 + h \cdot m_2 / 2$$

$$y_C = y_0 + h \cdot m_3$$

Schritte um y_1 zu berechnen1 : y_0 und $m_1 \rightarrow y_A$ und m_2 2 : y_0 und $m_2 \rightarrow y_B$ und m_3 3 : y_0 und $m_3 \rightarrow y_C$ und m_4 4 : y_0 und $m_1, m_2, m_3, m_4 \rightarrow y_1$

$$y_1 = y_0 + h \cdot (m_1 + 2 \cdot m_2 + 2 \cdot m_3 + m_4) / 6$$

Löse das Anfangswertproblem für die DGL einer Schwingung mit `ode45`

- schreibe eine Funktion `schwingung`, die die beiden ersten Ableitungen als Funktion von y und v berechnet und als Spaltenvektor zurückgibt; die Werte von y und v werden in einem Vektor mit zwei Elementen an die Funktion übergeben
- rufe im Skript `schwingung_ode45.m` die Funktion `ode45` auf, wobei die Funktion `schwingung` als Function-Handle übergeben wird

```
% globale Parameter der DGL
global k m;

k = 1;
m = 2;
% Anfangsbedingungen
y0 = 5.0;
v0 = 0;

% DGL lösen
[t, erg] = ode45(@schwingung, [0, 100], [y0, v0]);

plot(t, erg); % Lösung zeichnen
```

`schwingung_ode45.m`

```
function dz_dt = schwingung(t, z)
global k m
    dz_dt(1,1) = z(2);
    dz_dt(2,1) = -k/m*z(1);
end
```

$$\frac{dy}{dt} = v(t)$$
$$\frac{dv}{dt} = -\frac{k}{m} \cdot y(t)$$

Beachte : `erg` ist eine Matrix mit zwei Spalten; `t` und `erg` haben gleich viel Zeilen

MATLAB stellt eine ganze Reihe von Verfahren zur Lösung von DGLn zur Verfügung. In vielen Fällen verwendet man die MATLAB-Funktion `ode45` (ein explizites Runge-Kutta-Verfahren mit **automatischer Schrittweitensteuerung**). Der Aufruf von `ode45` erfolgt typischerweise so :

```
> [ t , y ] = ode45 ( @fdg1 , [ ta , te ] , ya ) ;
```

`@fdg1` :

ein function handle – Funktion `fdg1` berechnet die ersten Ableitungen
 die Funktion `fdg1` ist von der Form $dy_dx = f(x, y)$ oder $dy_dt = f(t, y)$
 Die Größe x bzw. t ist ein Skalar, y ein Vektor mit n Elementen; dy_dx
 bzw. dy_dt ist ein Spaltenvektor mit n Elementen

`[ta , te]` :

der Bereich (Ort oder Zeit), in dem die Lösung berechnet wird

`ya` :

der Anfangswert der Lösung, d.h. der Wert von y an der Stelle $t=ta$
 y_0 ist ein Vektor mit n Elementen – bei einer Schwingung 2 Elemente

`t` und `y` :

die Lösung der DGL : t ist ein Vektor mit Werten im Bereich $[ta , te]$
 y eine Matrix mit n Spalten und genauso vielen Zeilen wie t Elemente besitzt
 - Schwingung Spalte 1 ist die Auslenkung und Spalte 2 die Geschwindigkeit

Die Funktion `ode45` gibt einen Vektor t und eine Matrix y zurück. Der Vektor t besitzt k Zeilen und ebenso die Matrix y . Der Wert von k (also die Anzahl der Zeilen) hängt von vielen Faktoren ab.

Der Vektor t enthält die Zeiten, für die die Werte von y berechnet worden sind. Für das erste Element gilt $t(1) = t_a$ und für das **letzte** Element $t(k) = t_e$. Wie viele verschiedene t -Werte im Intervall $[t_a, t_e]$ verwendet werden, um die Lösung zu berechnen, hängt von vielen Faktoren ab. `ode45` versucht einerseits, mit Hilfe einer **Schrittweitensteuerung (Steuerung des Abstands aufeinanderfolgender Werte von t)** die Anzahl der t -Werte k klein zu halten. Andererseits darf der Abstand aufeinanderfolgender t -Werte nicht zu groß sein, damit der **Diskretisierungsfehler** (Ersetzung der Ableitung durch Differenzenquotienten) nicht zu groß wird.

Sollen die Lösungen zu fest vorgegebenen Zeiten berechnet werden, z.B. im Intervall zwischen 0 und 10 mit einer Schrittweite von 0.1, dann muss `ode45` folgendermaßen aufgerufen werden :

```
[t, y] = ode45(@fdg1, 0:0.1:10, y0);
```

t ist dann ein Vektor mit 101 Zeilen und y besitzt ebenfalls 101 Zeilen.

```
> help ode45
```

```
ode45 Solve non-stiff differential equations, medium order
method. [TOUT,YOUT] = ode45(ODEFUN,TSPAN,Y0) with TSPAN =
[T0 TFINAL] integrates the system of differential equations
 $y' = f(t,y)$  from time T0 to TFINAL with initial conditions
Y0. ODEFUN is a function handle. For a scalar T and a vector
Y, ODEFUN(T,Y) must return a column vector corresponding to
 $f(t,y)$ . Each row in the solution array YOUT corresponds to a
time returned in the column vector TOUT. To obtain
solutions at specific times T0,T1,...,TFINAL (all increasing
or all decreasing), use TSPAN = [T0 T1 ... TFINAL].
```

```
See also ode23, ode113, ode15s, ode23s, ode23t, ode23tb,
ode15i, odeset, odeplot, odephas2, odephas3, odeprint,
deval, odeexamples, rigidode, ballode, orbitode,
function_handle
```

```
> which ode45
```

```
C:\Program Files\MATLAB\R2017b\toolbox\matlab\funfun\ode45.m
```

Aufgabe 1

$$\frac{dy}{dx} = k \cdot x \cdot y(x)$$

Schreiben Sie ein Skript, das die nebenstehende DGL mit Hilfe des Eulerverfahrens löst. Die Anfangsbedingung lautet $y(x=1) = 3$.

Beim Start des Skripts gibt der Anwender einem Wert für k ein, einen Wert für die Schrittweite dx und einen Wert für den Endpunkt von x . Geben Sie die numerisch berechnete Lösung und die exakte Lösung graphisch aus. Geben Sie den relativen Fehler am rechten Intervallende aus.

Aufgabe 2

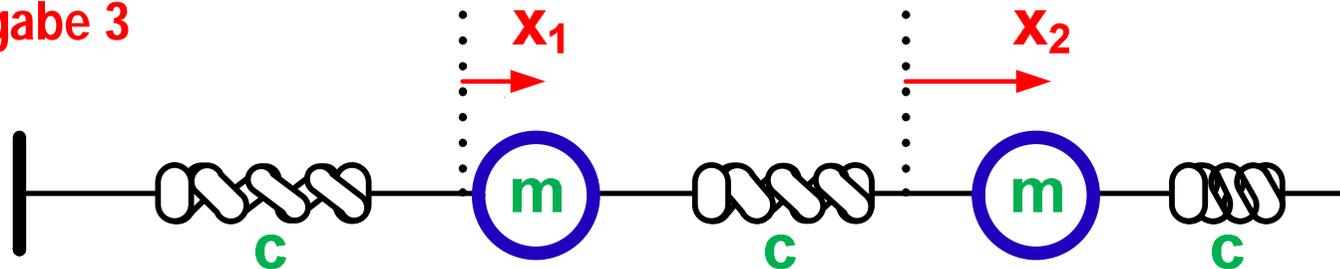
$$\ddot{y}(t) + 2 \cdot \delta \cdot \dot{y}(t) + \omega_0^2 \cdot y(t) = F \cdot \cos(\omega \cdot t)$$

$$y(t = 0) = x_0 \quad \dot{y}(t = 0) = v_0$$

Lösen Sie die DGL für eine gedämpfte Schwingung mit äußerer Anregung unter Verwendung von ode45. Erstellen Sie ein Skript, das beim Aufruf den Anwender auffordert, die Parameter für der DGL einzugeben, ebenso die Anfangswerte. Dann wird die Lösung für das Anfangswertproblem berechnet.

Geben Sie die Lösung graphisch aus. Zeichnen Sie die Auslenkung und die Geschwindigkeit als Funktion der Zeit.

Aufgabe 3



Bewegungsgleichungen :

$$m \cdot \ddot{x}_1(t) = -c \cdot x_1(t) - c \cdot [x_1(t) - x_2(t)]$$

$$m \cdot \ddot{x}_2(t) = -c \cdot [x_2(t) - x_1(t)] - c \cdot x_2(t)$$

$$x_1(t=0) = x_{10} \quad \dot{x}_1(t=0) = v_{10}$$

$$x_2(t=0) = x_{20} \quad \dot{x}_2(t=0) = v_{20}$$

Das obige Modell wird durch ein System von zwei gekoppelten DGLn 2-ter Ordnung beschrieben. Schreiben Sie die DGLn 2-ter Ordnung in ein System von 4 DGLn erster Ordnung um. Hierzu gibt es verschiedene Möglichkeiten (siehe rechts). Wie lauten die DGLn für y ?

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ v_1 \\ x_2 \\ v_2 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ v_1 \\ v_2 \end{bmatrix}$$

- a) Lösen Sie das Anfangswertproblem mit ode45 und geben Sie die Lösungen graphisch aus. Schreiben Sie hierzu eine Funktion, die die Ableitungen berechnet und ein entsprechendes Skript zum Aufruf von ode45.
- b) Versuchen Sie die Bewegung der zwei Massen in einer Animation darzustellen.

Aufgabe 4

Ein Flugzeug verwendet einen Bremsfallschirm und andere Mittel um nach der Landung zu bremsen. Die Änderung der Geschwindigkeit v beim Landen ist gegeben durch:

$$\frac{dv}{dt} = -0.0035 \cdot v^2 - 3$$

Zur Zeit $t=0$ befindet sich das Flugzeug an der Position $s=0$. Es öffnet den Bremsfallschirm und beginnt den Bremsvorgang. Das Flugzeug besitzt eine Geschwindigkeit von 300km/h.

- Berechnet man die Geschwindigkeit als Funktion der Zeit mit Hilfe von **ode45**, dann muss man ein M-File bereitstellen, das die erste Ableitung berechnet. Schreiben Sie das entsprechende M-File für die obige Differentialgleichung.
- Wie lautet der Aufruf der Funktion **ode45** zur Berechnung der Geschwindigkeit als Funktion der Zeit um das Anfangswertproblem im Zeitraum $[0,12]$ zu lösen? Die Geschwindigkeit soll in einer Variablen v gespeichert werden. Zeichnen Sie die Lösung.

Kapitel 9 Einführung in Simulink

9.1 Was ist Simulink ?

9.2 Erstes Simulink-Modell erstellen

9.3 Simulink und Schwingungen

9.4 Vereinfachung von Simulink-Modellen

9.5 Schnittstelle MATLAB – Simulink

9.6 Hausaufgaben

VDI-Richtlinie 3633: „Simulation ist das Nachbilden eines Systems mit seinen **dynamischen Prozessen** in einem **experimentierfähigem Modell**, um zu **Erkenntnissen zu gelangen**, die auf die **Wirklichkeit** übertragbar sind.“

How Simulink Works (Simulink User's Guide)

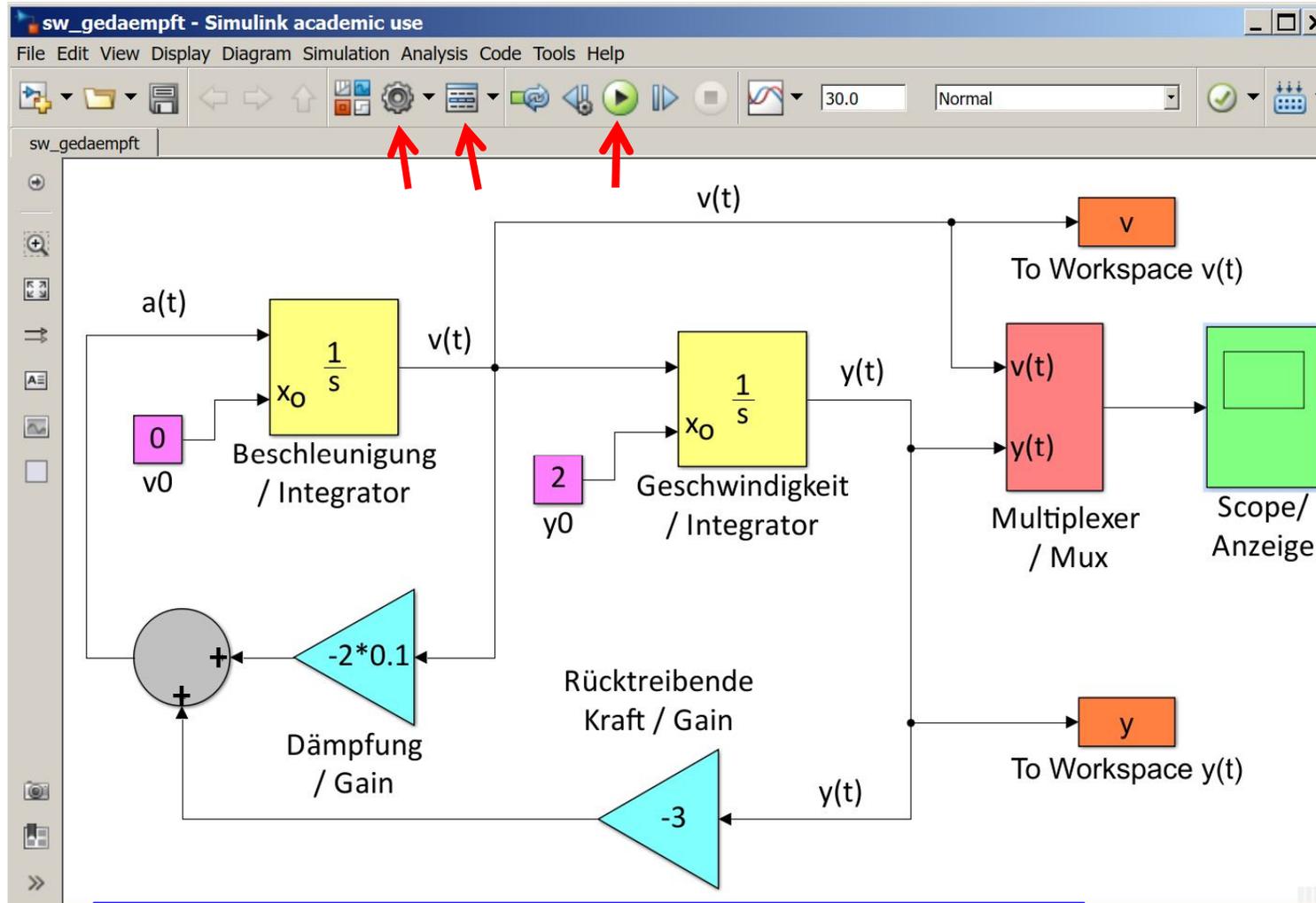
Simulink is a software package that enables you to model, simulate, and analyze systems whose outputs change over time. Such systems are often referred to as **dynamic systems**. The Simulink software can be used to **explore the behavior** of a wide range of **real-world** dynamic systems, including electrical circuits, shock absorbers, braking systems, and many other electrical, mechanical, and thermodynamic systems.

Simulating a dynamic system is a two-step process. First, a user creates a block diagram, using the Simulink model editor, that graphically depicts **time-dependent mathematical relationships** among the system's **inputs, states, and outputs**. The user then commands the Simulink software to simulate the system represented by the model from a specified start time to a specified stop time.

Mathematisches Modell eines einfachen dynamischen Systems :

$$\frac{d^2 y}{dt^2} + 2 * 0.1 * \frac{dy}{dt} + 3 * y(t) = 0$$

$$v_0 = \dot{y}(t = 0) = 0 \quad , \quad y_0 = y(t = 0) = 2$$



Blöcke :

- Integrator
- Addition
- Gain
- Constant
- Mux
- ToWorkspace
- Scope

Signale :

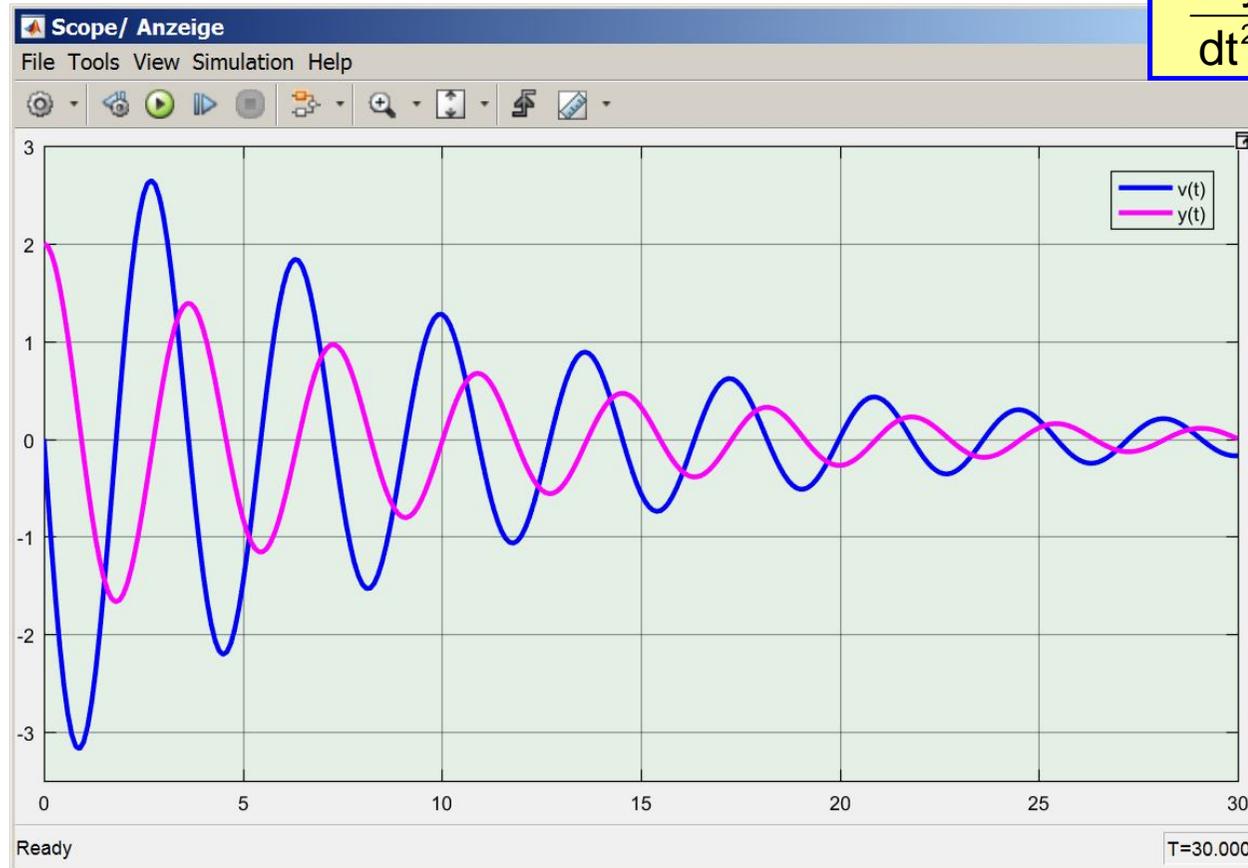
a(t) , v(t) , y(t)

Systemzeit :

t - [0,30]

Simulink-Modell für eine gedämpfte Schwingung

Ergebnis der Simulation



$$\frac{d^2 y}{dt^2} + 2 * 0.1 * \frac{dy}{dt} + 3 * y(t) = 0$$

$$\dot{y}(t = 0) = 0, y(t = 0) = 2$$

Die Blöcke **ToWorkspace** schreiben die Werte der Signale $y(t)$ und $v(t)$ zu bestimmten Zeiten in zwei Vektoren des MATLAB - Workspace. Die Zeitpunkte werden im Vektor t_{out} gespeichert.

$$t_{out}(i) = t_i$$

$$y(i) \Leftrightarrow y(t_i)$$

$$v(i) \Leftrightarrow v(t_i)$$

Für die Simulation kann ein Runge-Kutta-Verfahren mit fester Schrittweite verwendet (**ode4**, Schrittweite $\Delta t = 0.1$) werden. Die Simulation erstreckt sich über ein Zeitintervall von $[0, 30]$.

Workspace			
Name	Value	Min	Max
tout	301x1 double	0	30
v	301x1 double	-3.1709	2.6463
y	301x1 double	-1.6670	2

$$\frac{d^2y}{dt^2} + 2 * 0.1 * \frac{dy}{dt} + 3 * y(t) = 0$$

$$v0 = \dot{y}(t = 0) = 0, \quad y0 = y(t = 0) = 2$$

Die DGL für eine gedämpfte Schwingung kann mit **Simulink** oder mit **MATLAB** gelöst werden.

Simulink :

- DGL (System) und Anfangsbedingungen über eine **graphische(!)** Oberfläche **definieren**
- **Simulink berechnet das zeitliche Verhalten des Systems selbstständig** mit Hilfe geeigneter Verfahren (z.B. **ode45** oder **ode4**)
- Ergebnisse mit geeigneten Blöcken darstellen
- Ergebnisse in Variablen des MATLAB-Workspace speichern

MATLAB :

- DGL in ein System von DGLn erster Ordnung umschreiben und in einer **MATLAB-Funktion codieren** (siehe `schwingung.m`).
- DGL mit Hilfe eines geeigneten Verfahrens lösen (expliziter Aufruf der Funktion **ode45** oder **ode4** oder ...)
Die Ergebnisse werden über Rückgabewerte in einem Array gespeichert.
- Ergebnisse mit Hilfe des `plot`-Befehls graphisch darstellen oder weiterverarbeiten

Kapitel 9 Einführung in Simulink

9.1 Was ist Simulink ?

9.2 Erstes Simulink-Modell erstellen und ausführen

9.2.1 Simulink starten

9.2.2 Simulink-Modell erstellen

9.2.3 Simulation durchführen

9.2.4 Sine Wave-Block

9.2.5 Integrator-Block

9.2.6 Mux-Block

9.2.7 Scope-Block

9.3 Simulink und Schwingungen

9.4 Vereinfachung von Simulink-Modellen

9.5 Schnittstelle MATLAB – Simulink

9.6 Hausaufgaben

Aufgabe :

Das Signal $a(t) = A_0 * \sin(\omega * t)$ und das Integral dieses Signals über die Zeit sollen graphisch dargestellt werden. Lösen Sie die Aufgabe mit Simulink für das Zeitintervall [0, 10].

Welche Blöcke sind für diese Aufgabe notwendig ?

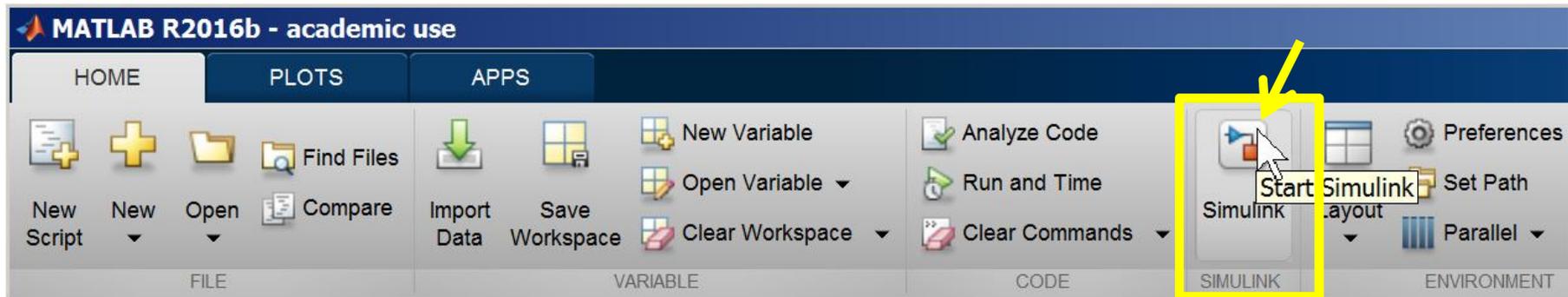
- Block, der ein Sinussignal erzeugt – Parameter : Amplitude, Frequenz
- Block, der ein Signal von 0 bis t integriert
- Block, der Signale über der Zeit darstellt
- eventuell weitere Hilfsblöcke

Teilaufgaben :

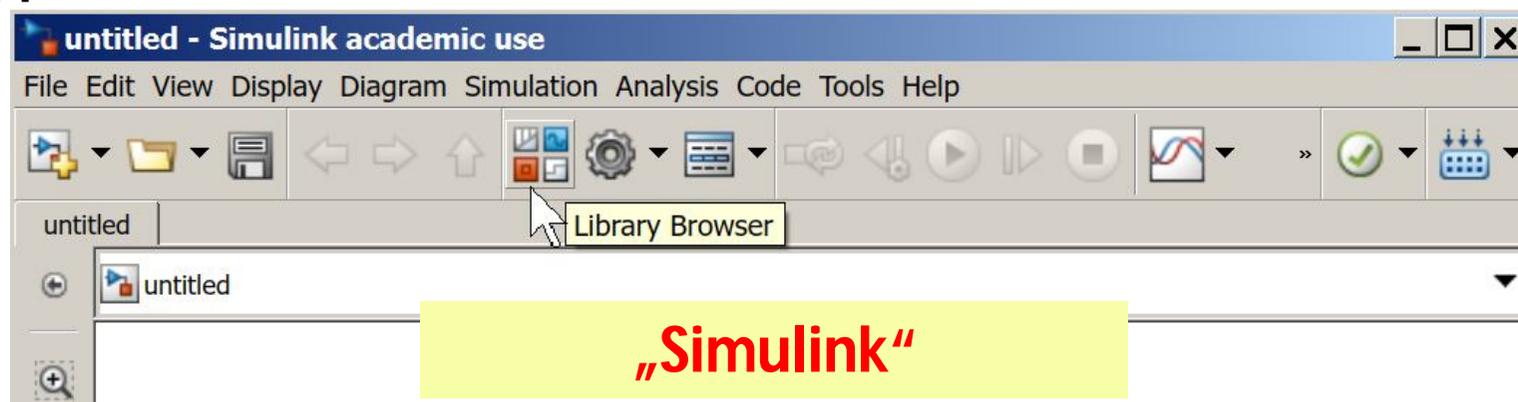
- 1) Simulink starten
- 2) Ein neues Simulink-Modell erstellen
- 3) Simulation starten
- 4) Das Ergebnis der Simulation analysieren

Um mit Simulink zu arbeiten muss zuerst MATLAB gestartet werden. Danach kann Simulink auf verschiedene Arten geöffnet werden.

- Über das Icon „**Simulink**“ in der MATLAB Toolbar. Es öffnet sich die „Simulink Start Page“. Über „**Blank Model**“ wird ein neues Simulink-Modell erzeugt oder man öffnet ein bestehendes Simulink-Modell.



- Im MATLAB Command-Window den Befehl `simulink` eingeben – es wird wieder die „Simulink Start Page“ geöffnet.
- Doppelklick auf ein existierendes Simulink-Modell im Fenster „Current Folder“



9.2.1 Simulink starten - Simulink Library Browser

Suche nach bestimmten Blöcken

Bibliothek auswählen

Simulink Library Browser

Sinus

Suche nach bestimmten Blöcken

Simulink

- Simulink
- Aerospace Blockset
- Communications System Toolbox
- Communications System Toolbox HDL Support
- Computer Vision System Toolbox
- Control System Toolbox
- Data Acquisition Toolbox
- DSP System Toolbox
- DSP System Toolbox HDL Support
- Embedded Coder
- Fuzzy Logic Toolbox
- HDL Coder
- Image Acquisition Toolbox
- Instrument Control Toolbox
- Neural Network Toolbox
- Robotics System Toolbox
- Robust Control Toolbox
- Simscape
- Simulink 3D Animation
- Simulink Coder
- Simulink Control Design
- Simulink Design Optimization
- Simulink Desktop Real-Time
- Simulink Extras
- Simulink Real-Time
- Simulink Verification and Validation
- Stateflow
- System Identification Toolbox
- Vehicle Network Toolbox
- Recently Used

Commonly Used Blocks

Continuous

Dashboard

Discontinuities

Discrete

Logic and Bit Operations

Lookup Tables

Math Operations

Misc

Model-Wide Utilities

Model Verification

Ports & Subsystems

Signal Attributes

Signal Routing

Sinks

Sources

User-Defined Functions

Additional Math & Discrete

Simulink

- Commonly Used Blocks
- Continuous
- Dashboard
- Discontinuities
- Discrete
- Logic and Bit Operations
- Lookup Tables
- Math Operations
- Model Verification
- Model-Wide Utilities
- Ports & Subsystems
- Signal Attributes
- Signal Routing
- Sinks
- Sources
- User-Defined Functions
- Additional Math & Discrete
 - Additional Discrete

Liste von Bibliotheken oder Blöcken

Simulink Library Browser

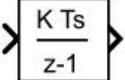
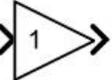
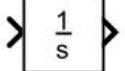
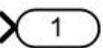
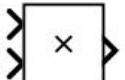
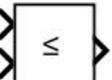
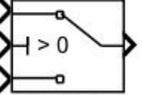
Sinus

Blöcke der Bibliothek „Commonly Used Blocks“

Simulink/Commonly Used Blocks

Simulink

- Commonly Used Blocks
- Continuous
- Dashboard
- Discontinuities
- Discrete
- Logic and Bit Operations
- Lookup Tables
- Math Operations
- Model Verification
- Model-Wide Utilities
- Ports & Subsystems
- Signal Attributes
- Signal Routing
- Sinks
- Sources
- User-Defined Functions
- Additional Math & Discrete
- Aerospace Blockset
- Communications System Toolbox
- Communications System Toolbox H
- Computer Vision System Toolbox
- Control System Toolbox
- Data Acquisition Toolbox
- DSP System Toolbox
- DSP System Toolbox HDL Support
- Embedded Coder
- Fuzzy Logic Toolbox
- HDL Coder
- Image Acquisition Toolbox

			
Bus Creator	Bus Selector	Constant	Data Type Conversion
			
Delay	Demux	Discrete-Time Integrator	Gain
			
Ground	In1	Integrator	Logical Operator
			
Mux	Out1	Product	Relational Operator
			
Saturation	Scope	Subsystem	Sum
			
Switch	Terminator	Vector Concatenate	

Nach dem Start von Simulink wird ein neues Simulink-Modell erzeugt oder ein bereits vorhandenes Simulink-Modell geöffnet.

Simulink-Modelle werden in Dateien mit der Endung **.slx** gespeichert (veraltet **.mdl** abgeleitet von **model**). Diese Dateien sind Binärdateien und nur mit Simulink lesbar.

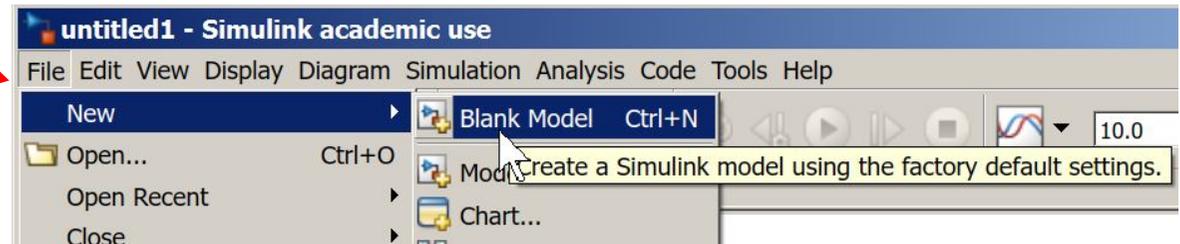
- **Neues (leeres) Simulink-Modell erstellen**

1. Simulink

File->New ->Blank Model

2. File->Save

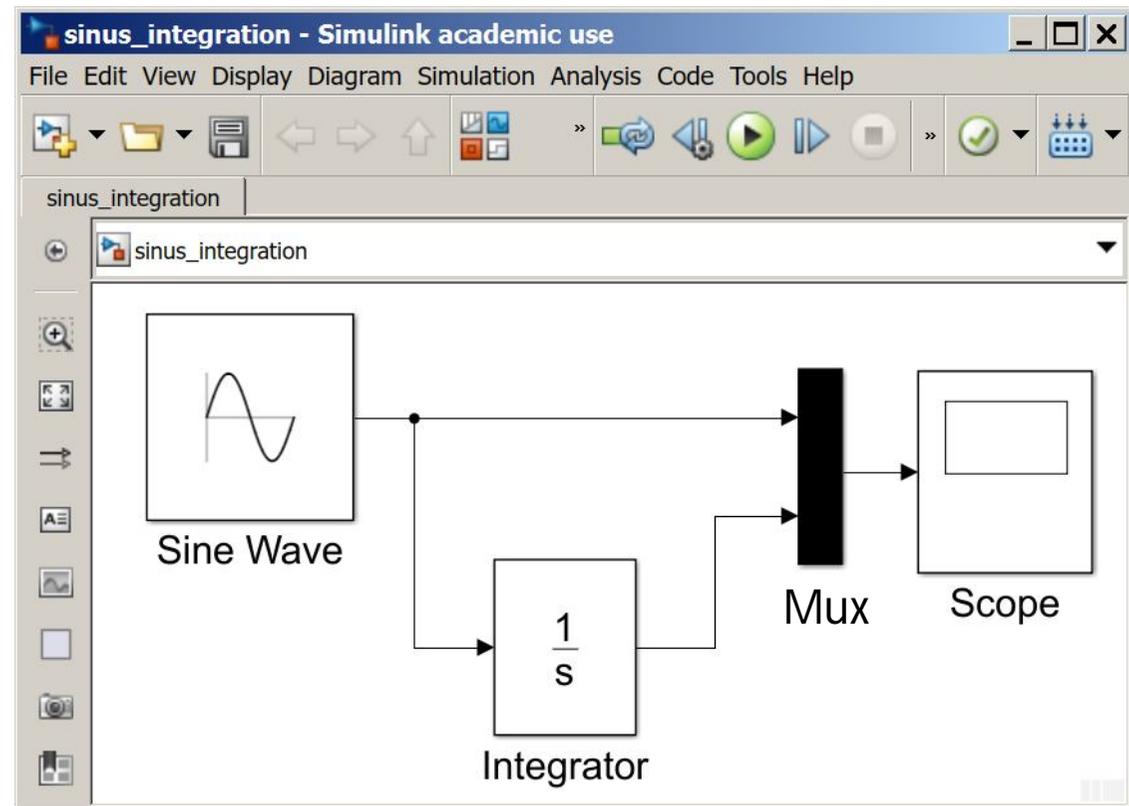
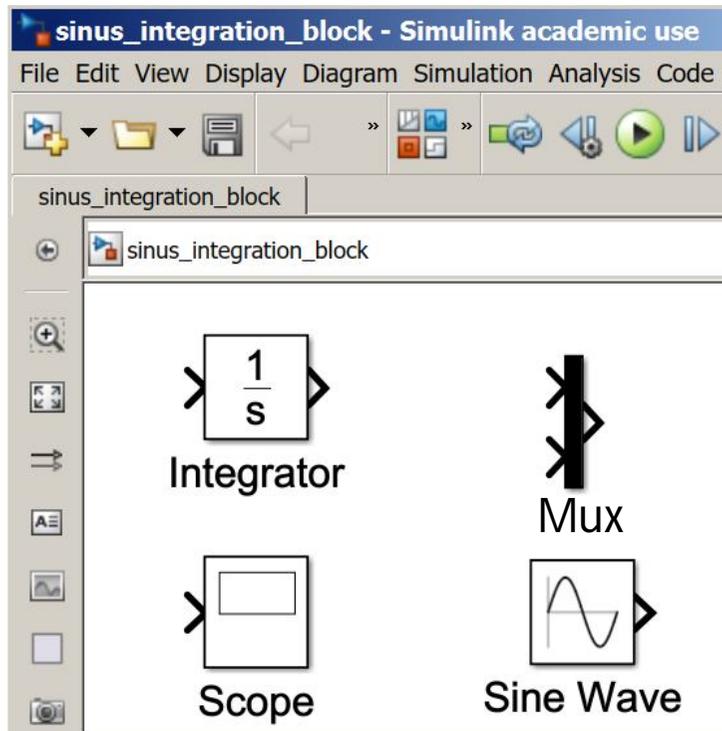
3. Dateinamen für das Modell eingeben



- **Vorhandenes Simulink-Modell öffnen – auf mehrere Arten möglich**

- Wähle in Simulink „File -> Open“ und selektiere die Datei, in der das Modell gespeichert ist.
- Doppelklick auf die Simulink-Datei (Datei mit der Endung **.slx**)
- Dateinamen im MATLAB Command-Window eingeben.
- Wähle „File -> Open“ im Menü von MATLAB

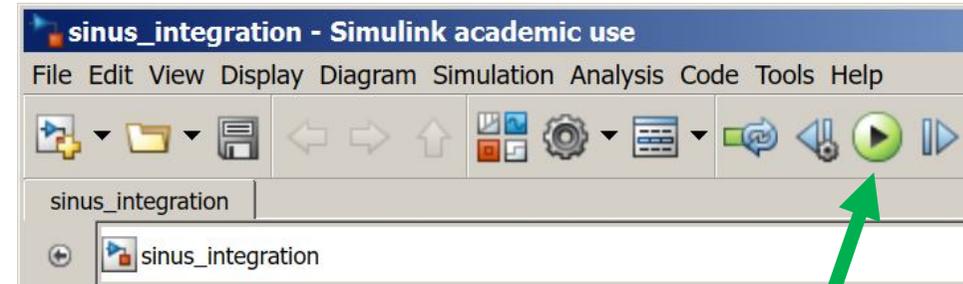
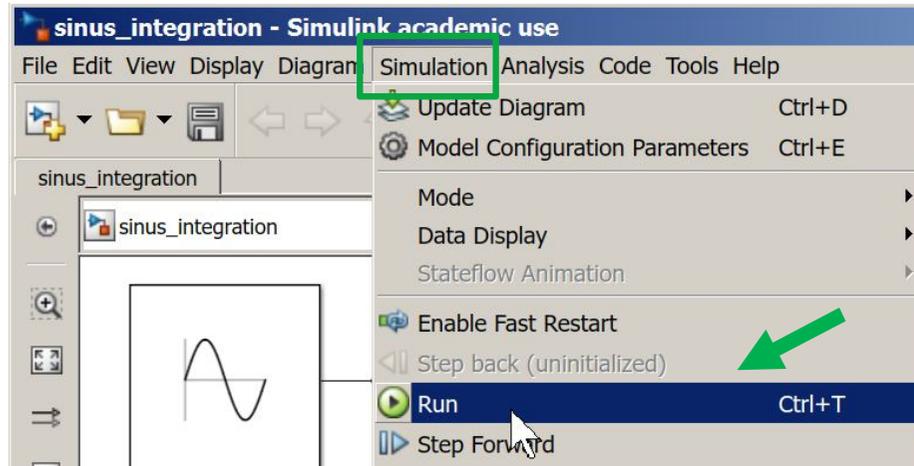
Im Simulink Library Browser werden die notwendigen Blöcke für das Modell gewählt und in ein leeres Modell gezogen (links). Anschließend werden die Blöcke sinnvoll angeordnet, miteinander verbunden und konfiguriert (rechts).



Das Modell enthält die Blöcke :

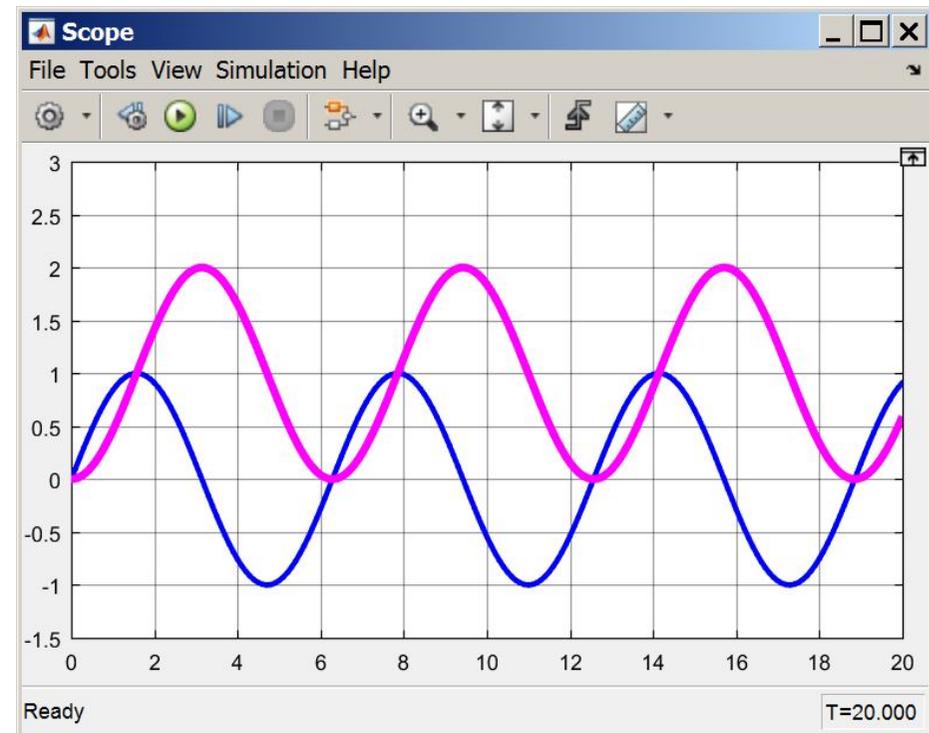
- Integrator
- Multiplexer (Mux)
- Scope – für die Anzeige der Ergebnisse
- Sinusgenerator (Sine Wave) aus der Bibliothek Sources

Ein Simulink-Modell wird über den Button **Run** der Toolbar gestartet oder über das Menü **Simulation -> Run** oder mit **Ctrl+T**.

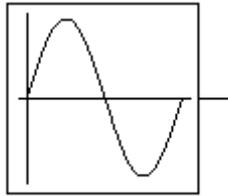


Das Ergebnis der Simulation wird angezeigt, wenn man den Scope-Block per Doppelklick öffnet. Im vorliegenden Simulink-Modell hat der Scope-Block ein Eingangssignal, das aus zwei Teilsignalen besteht. Der Mux-Block erzeugt aus zwei skalaren Signalen ein vektorwertiges Signal, das in den Scope-Block geführt wird.

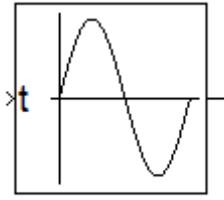
Damit in der Zeichnung eine glatte Kurve dargestellt wird, muss die Schrittweite bei der Simulation begrenzt werden, z.B. auf $\Delta t = 0.05$.



Sine Wave - Sinusgenerator



Sine Wave



Sine Wave

Erzeugt ein sinusförmiges Signal

Parameters

Sine type:

Time (t):

Amplitude:

Bias:

Frequency (rad/sec):

Phase (rad):

Sine type :

- Time based

$$y(t) = \text{Amp} * \sin(\text{Freq} * t + \text{phase}) + \text{bias}$$

Freq = $2 * \pi$ bedeutet eine Periode pro Sek

Die Konstante π kann verwendet werden

$$1 \text{ rad} == 360^\circ / (2 * \pi)$$

- Sample based

Siehe Simulink-Hilfe

Time (t) :

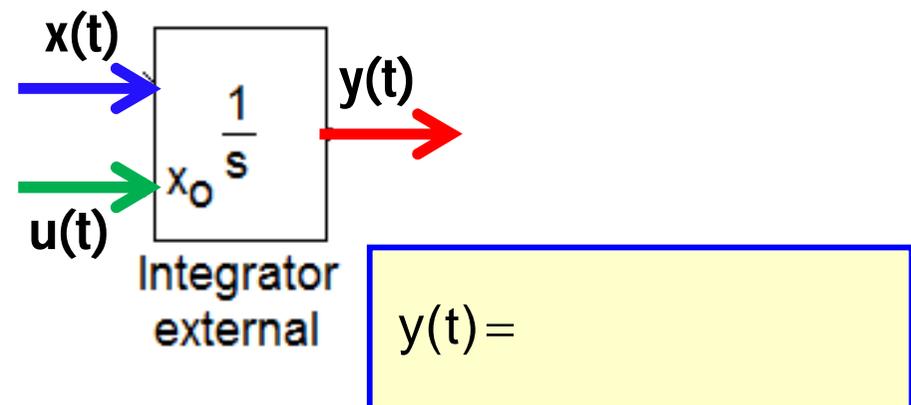
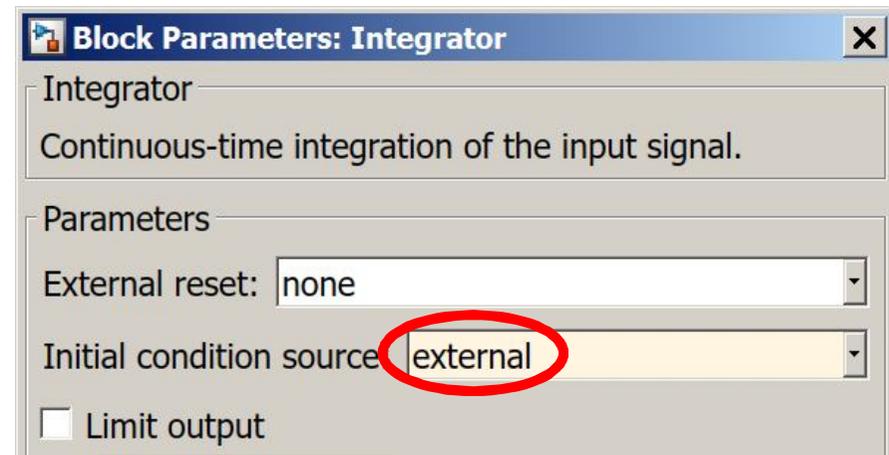
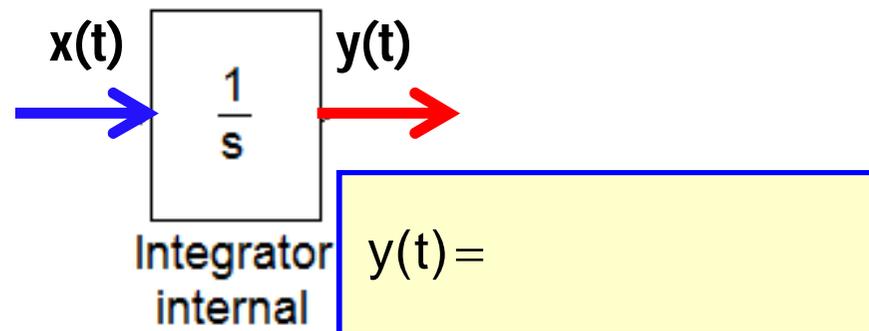
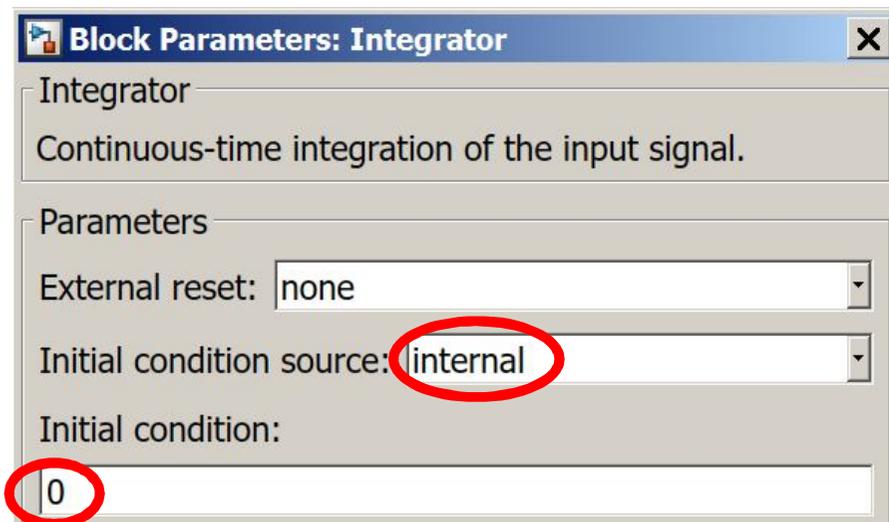
- Use Simulation time – kein Eingang
- Use external signal – mit Eingang

Aufgaben :

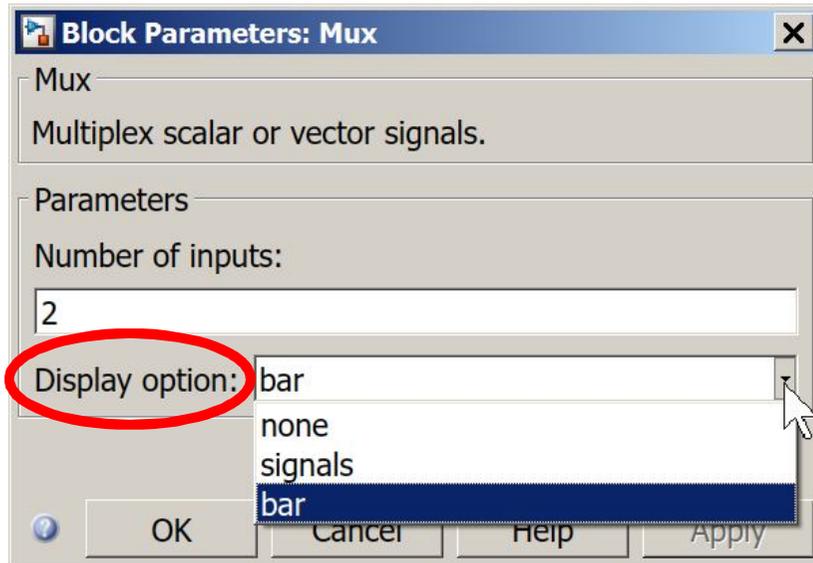
- 5 Schwingungen pro Sekunde, Amplitude 2
- 3.7 Schwingungen pro Sekunde, Anfangsphase 35°
- Welche Wirkung hat bias = 2 ?

9.2.5 Integrator-Block

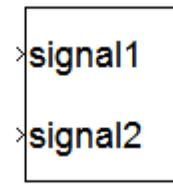
Der Integratorblock integriert das Eingangssignal von der Startzeit (Default 0) bis zur aktuellen Zeit t . Zusätzlich muss noch eine Anfangsbedingung festgelegt (Initial condition) werden. Das Symbol für den Integrator-Block $1/s$ stammt aus der Theorie der **Laplace-Transformationen** (**Integration** im Originalraum entspricht die **Division durch s** im Bildraum).



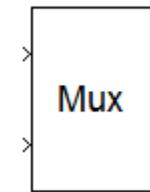
Mux : Combine several input signals into **vector**



bar

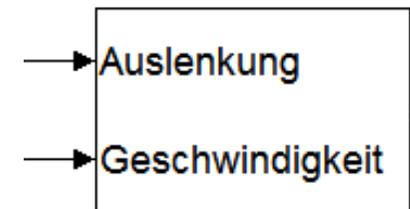


signals

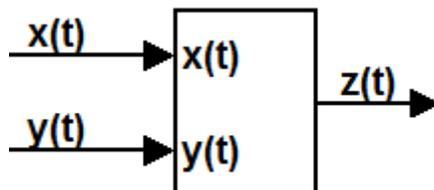


none

Mit Hilfe der „Display option“ wird die Darstellung des Mux-Blocks festgelegt. Wählt man die Option „signals“, werden an den Eingängen die **Namen der Eingangssignale** gezeigt. Dadurch wird ein Modell besser lesbar.

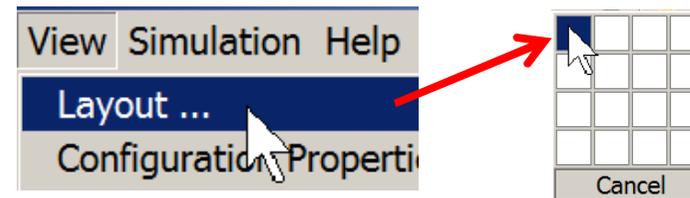
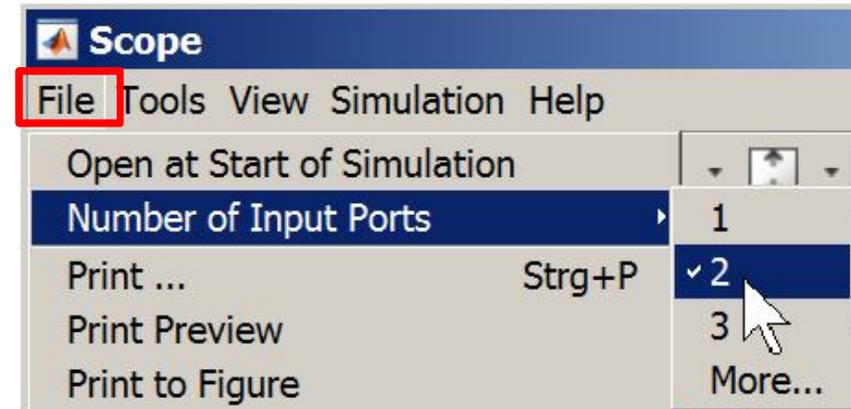
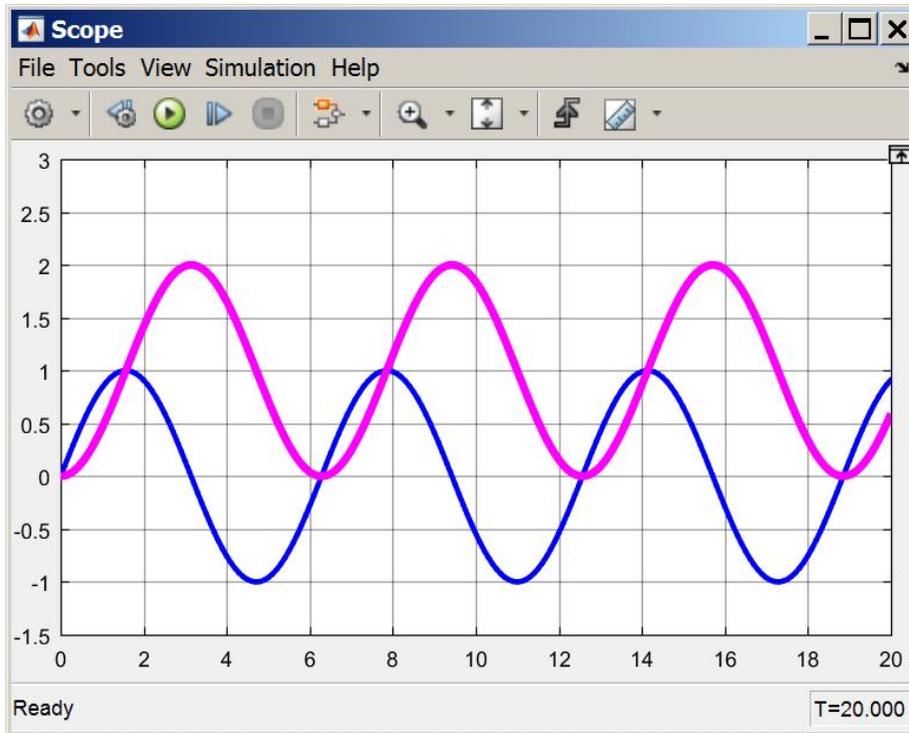


Ein Mux-Block erzeugt aus mehreren Eingangssignalen ein **vektorwertiges Signal** – mehrere Einzelsignale werden zu einem einzigen Signal zusammengefasst.



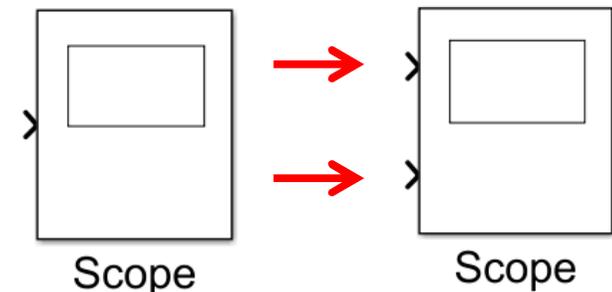
$$z(t) = \begin{bmatrix} z_1(t) \\ z_2(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} \quad \begin{array}{l} z_1(t) = x(t) \\ z_2(t) = y(t) \end{array}$$

Der Scope-Block dient zur graphischen Darstellung von Signalen. Es werden die Eingangssignale als Funktion der Simulationszeit angezeigt.



Number of Input Ports :

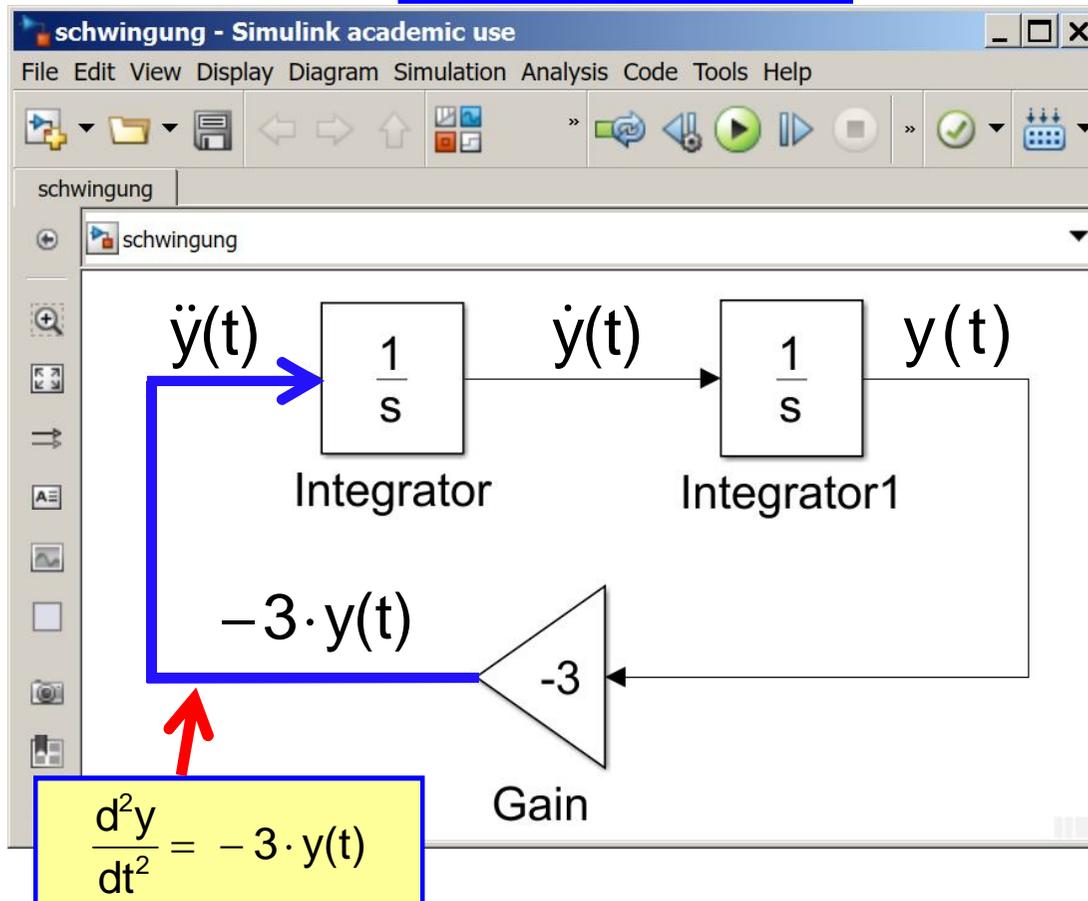
Legt die **Anzahl der Eingänge** des Scope-Blocks fest. Die Signale an jedem der Eingänge werden in separaten Teilfenstern dargestellt; welche Teilfenster angezeigt werden, wird über das Menü **View->Layout** bestimmt. Jedes Teilfenster kann ein oder mehrere Kurven gleichzeitig darstellen (-> MUX).



DGL für eine Schwingung

$$\frac{d^2y}{dt^2} + 3 \cdot y(t) = 0$$

$$y(t=0) = 3 \quad , \quad \dot{y}(t=0) = 0$$



schwingung.slx

Block Parameters: Integrator1

Integrator
Continuous-time integration of the input signal.

Parameters

External reset: none

Initial condition source: internal

Initial condition:

3

Die Anfangsbedingungen für die Lösung der DGL werden in den beiden Integratorblöcken (Initial condition) gesetzt. Man kann die Anfangsbedingungen auch extern setzen, über einen Constant-Block. **Beachte** : Das obige Simulink-Modell **beschreibt** nur die DGL für eine Schwingung. Es ist keine Lösung der DGL.

DGL für eine Schwingung

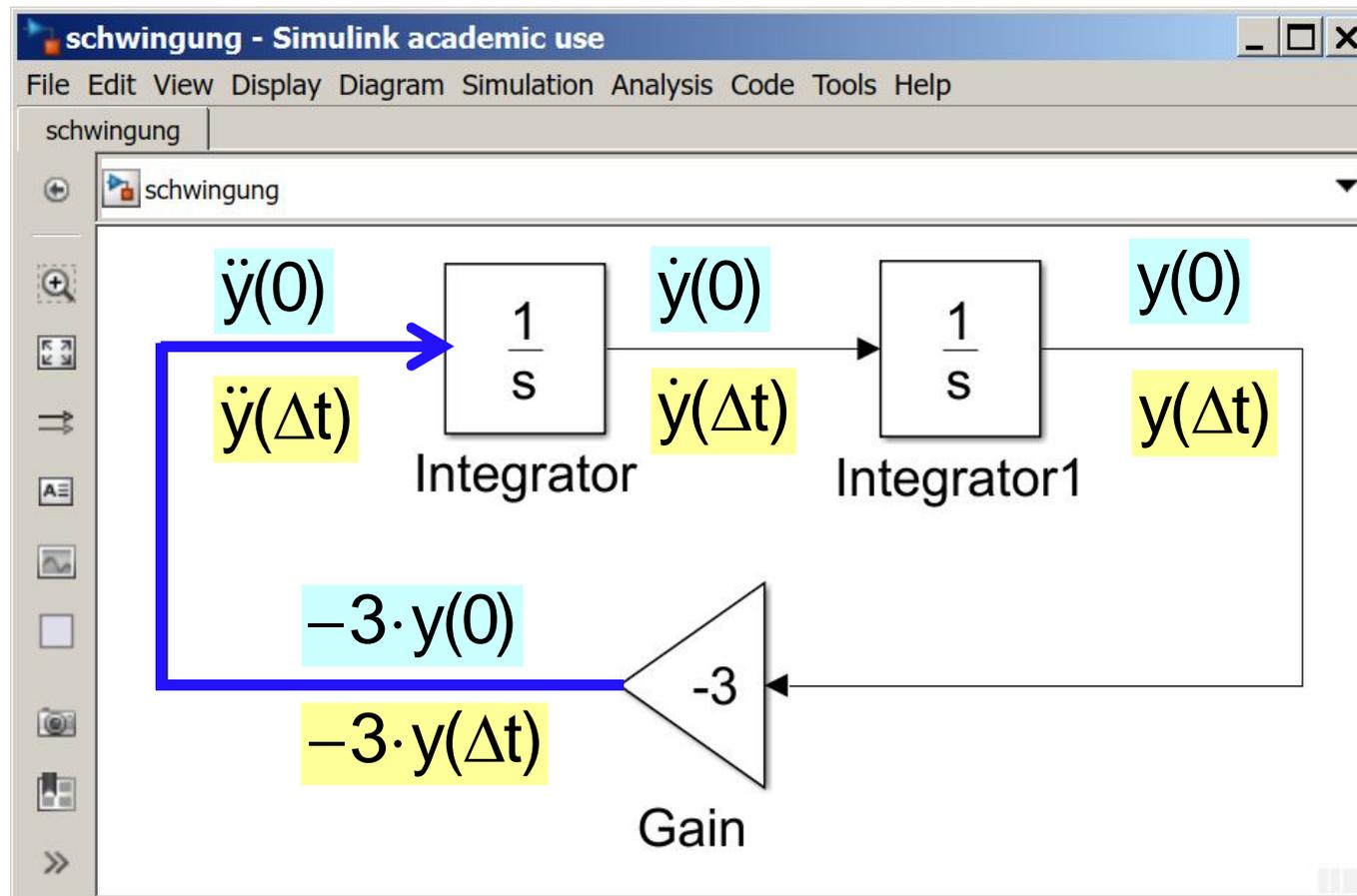
$$\frac{d^2y}{dt^2} + 3 \cdot y(t) = 0$$

$$y(t=0) = 3, \quad \dot{y}(t=0) = 0$$

Wie kann man mit Hilfe der Information, die im Simulink-Modell enthalten ist, eine Lösung berechnen ?

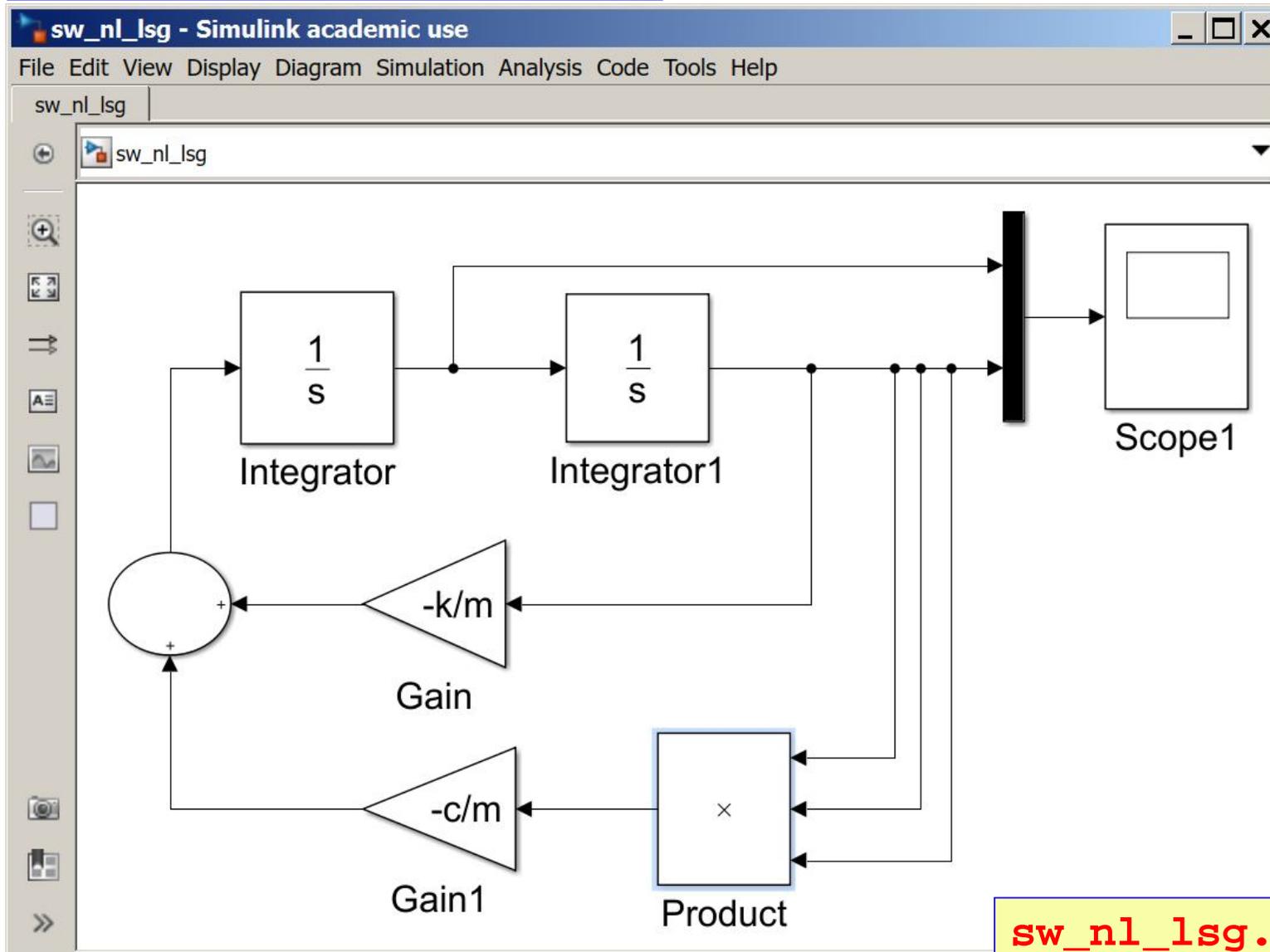
Annahme : Zur Zeit $t=0$ sind Auslenkung, Geschwindigkeit und Beschleunigung bekannt.

Wie erhält man daraus die Größen zur Zeit $t=\Delta t$?



$$\frac{d^2y}{dt^2} + \frac{k}{m} \cdot y(t) + \frac{c}{m} \cdot y^3(t) = 0$$

Der Term $\sim y^3$ ist ein sogenannter nichtlinearer Term.



Block Parameters: Product

Product

Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:

a) * or / for each input port. For example, **/* performs the operation 'u1*u2/u3*u4'.
 b) scalar specifies the number of input ports to be multiplied.

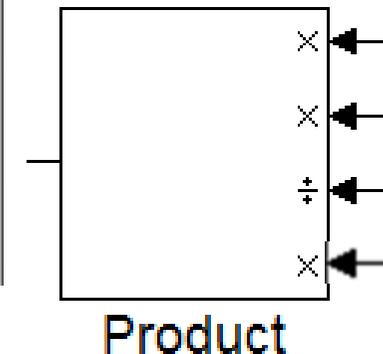
If there is only one input port and the Multiplication parameter is set to Element-wise(.*), a single * or / collapses the input signal using the specified operation. However, if the Multiplication parameter is set to Matrix(*), a single * causes the block to output the matrix unchanged, and a single / causes the block to output the matrix inverse.

Main | Signal Attributes

Number of inputs:

**/*

Multiplication: Element-wise(.*)

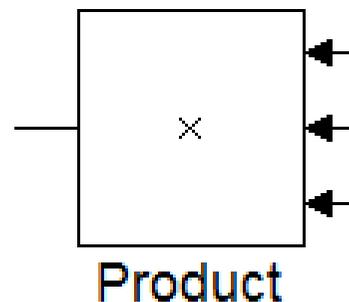


Main | Signal Attributes

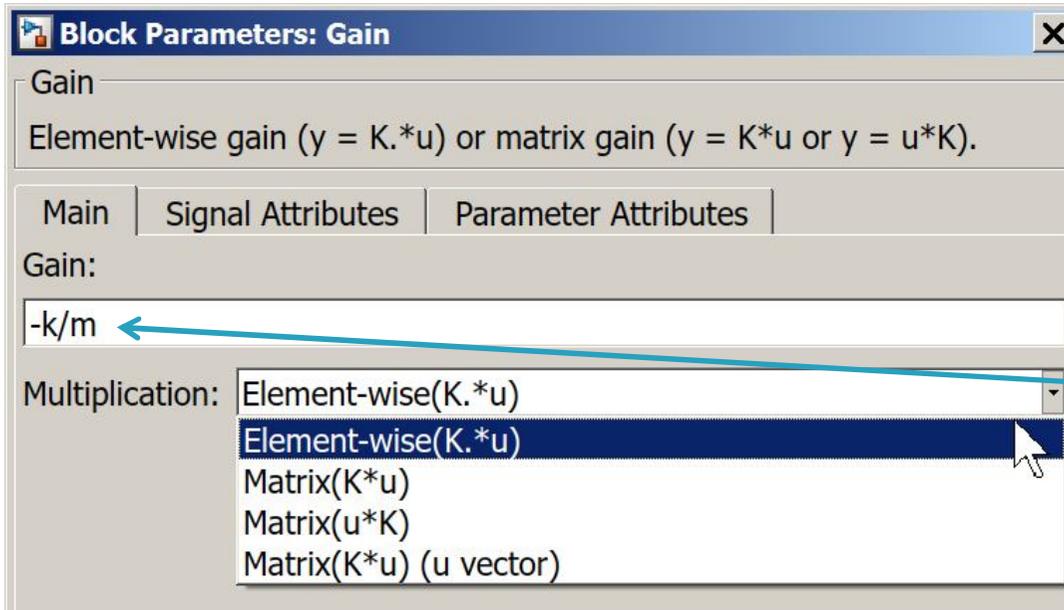
Number of inputs:

3

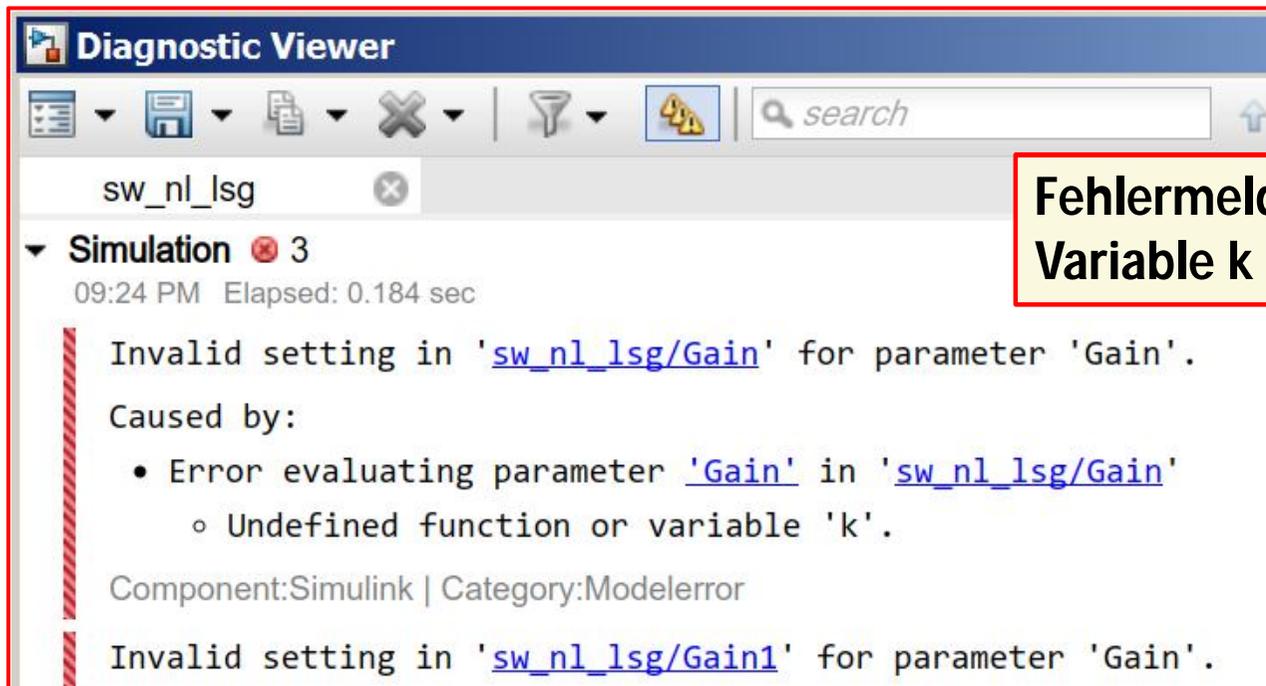
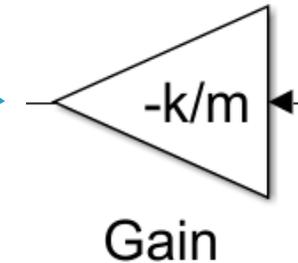
Multiplication: Element-wise(.*)



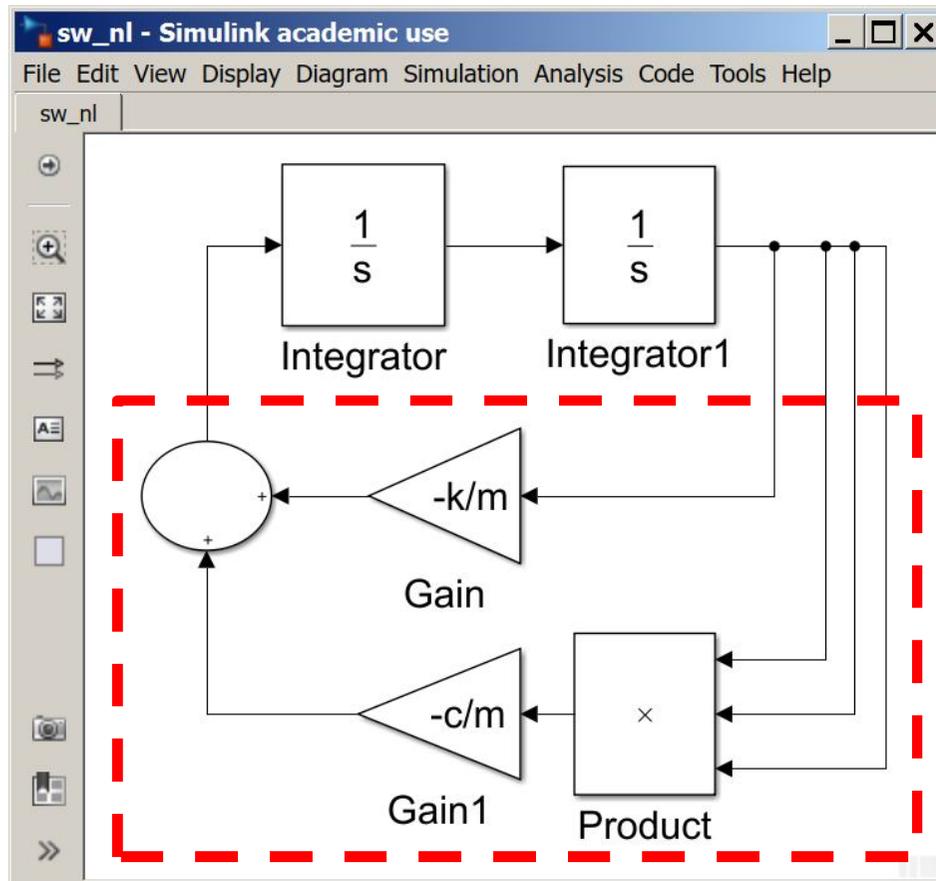
K9_22 9.3.4 Parameter in Simulink-Modellen



Die beiden Variablen **k** und **m**, die im Gain-Block verwendet werden, müssen im MATLAB Workspace gesetzt werden.



Fehlermeldung bei der Simulation.
Variable k ist nicht definiert!



Der Teil des Modells, der die rücktreibende Kraft beschreibt (rot umrandet), soll "vereinfacht" werden.

Es wird eine kompakte Darstellung gesucht, die leichter lesbar und einfacher zu ändern ist. Hierzu werden folgende Blöcke verwendet :

- **Fcn-Block**
- **Subsystem-Block**
- **Skalare Signale zu vektorwertigen Signalen zusammenfassen**

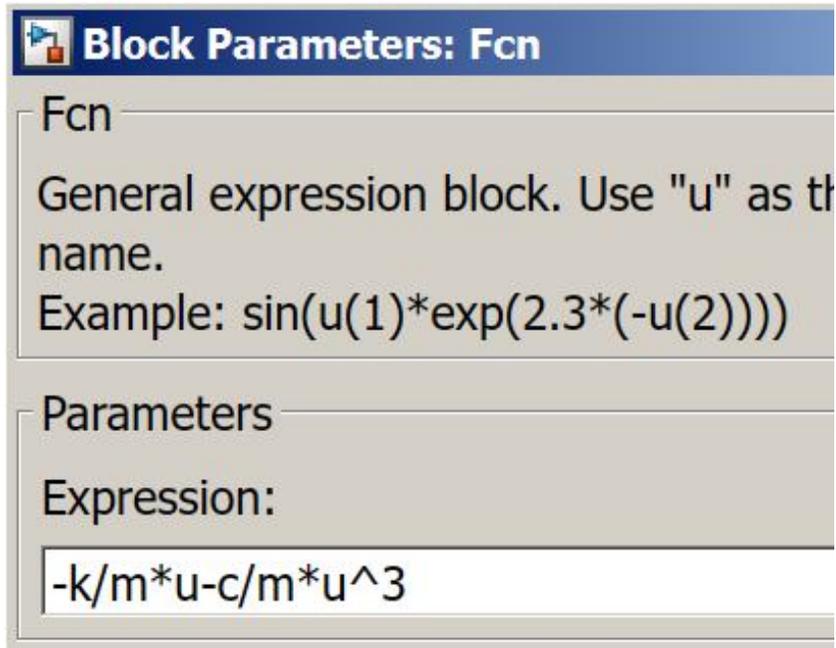
Der Subsystem-Block wird oft auch dazu verwendet, um eine **Hierarchie von Simulink-Modellen** zu erzeugen. Auf oberster (abstrakter) Ebene des Simulink-Modells sieht man nur wenige einfache Blöcke und eine Reihe von Subsystem-Blöcken. In der nächsten Ebene werden die Subsystem-Blöcke genauer definiert. Die Subsystem-Blöcke können wiederum Subsystem-Blöcke enthalten, usw. .

The image displays three windows from the Simulink environment:

- sw_nl_fcn - Simulink academic use:** A Simulink model window showing a block diagram. It features two integrator blocks (labeled $\frac{1}{s}$ and Integrator1) and a function block labeled f(u) and Fcn. The function block is connected to the output of the first integrator and the input of the second.
- Block Parameters: Fcn:** A dialog box for configuring the Fcn block. It shows the name "Fcn", a description: "General expression block. Use 'u' as the input variable name. Example: $\sin(u(1)*\exp(2.3*(-u(2))))$ ", and a field for the "Expression" containing the formula $-k/m*u-c/m*u^3$.
- Simulink Library Browser:** A window showing the Simulink library structure. The "User-Defined Functions" category is selected, and the "Fcn" block is highlighted with a red dashed border. Other blocks visible include "Function Caller", "Initialize Function", "Interpreted MATLAB Function", "MATLAB Function", "MATLAB System", and "S-Function".

Ein **Function-Block** erzeugt ein Ausgangssignal als Funktion des Eingangssignals. Das Feld Expression enthält einen math. Ausdruck, der definiert, wie sich das Ausgangssignal als Funktion des Eingangssignals berechnet. Für das Eingangssignal muss immer der Name **u** verwendet werden.

Das Feld Expression enthält einen math. Ausdruck, der definiert, wie sich das Ausgangssignal als Funktion des Eingangssignals berechnet. Für das Eingangssignal muss immer der Name **u** verwendet werden, egal wie das Eingangssignal extern bezeichnet wird.



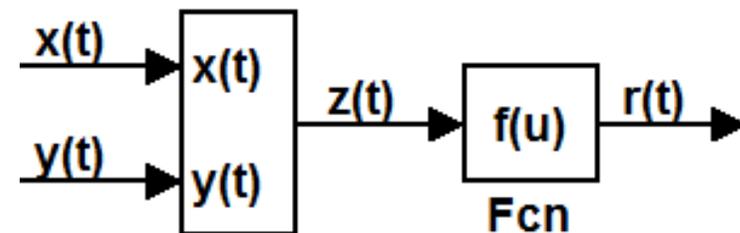
Die **Eingangsgröße** kann ein **skalares** oder ein **vektorielles Signal** sein. Ist die Eingangsgröße ein Vektor, dann wird die i-te Komponente des Signals mit $u(i)$ bezeichnet.

Die Ausgangssignal muss stets ein skalares Signal sein!

Die Regeln, wie der Ausdruck im Feld Expression gebildet wird und was dabei zu beachten ist, sind in der Hilfe ausführlich beschrieben.

Aufgabe:

Ein Function-Block soll den Radius $r(t)$ berechnen, der sich aus den Signalen $x(t)$ und $y(t)$ ergibt. Wie lautet der Ausdruck im Feld Expression des Function-Blocks?



Anstelle eines **Fcn-Blocks** soll ein **Subsystem-Block** verwendet werden.

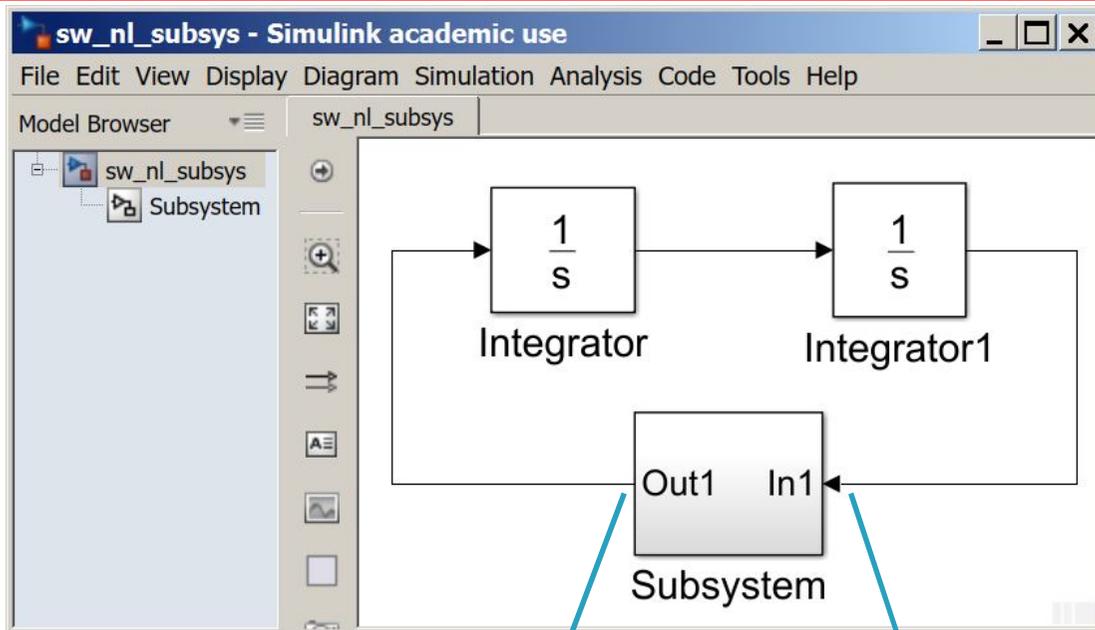
The screenshot shows the Simulink environment with a control loop diagram. The diagram includes two integrator blocks (1/s), a summing junction, two gain blocks (-k/m and -c/m), and a product block (x). A blue selection box highlights the product block and the two gain blocks. A context menu is open over the selection, showing options like 'Cut', 'Copy', 'Paste', and 'Create Subsystem from Selection' (Ctrl+G). The menu path 'Diagram -> Subsystem & Model Reference -> Create Subsystem from Selection' is also visible in the top menu bar.

Selektiere alle Blöcke, die in den Subsystem-Block verlagert werden, ebenso die Eingangs- und die Ausgangssignale. Die Selektion kann auf zwei Arten erfolgen :

- einen rechteckigen Bereich mit Hilfe der Maus wählen
- Drücken der Shift-Taste und Selektion der Blöcke und Signale

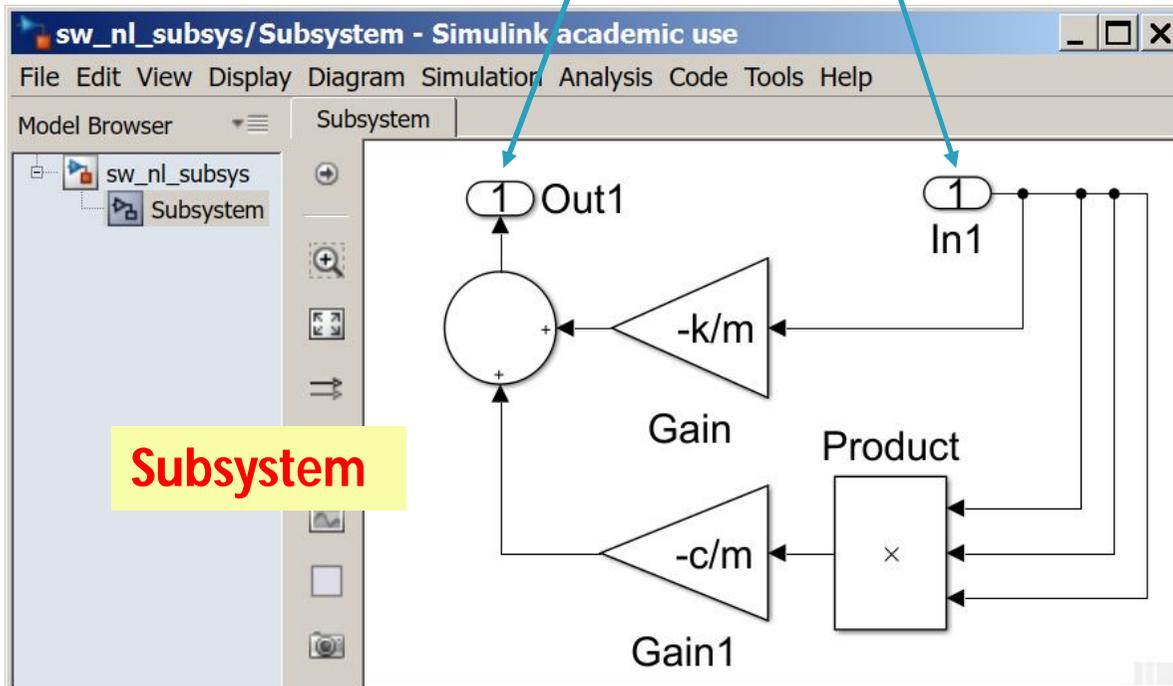
Danach **Menü Diagram-> Subsystem & Model Reference -> Create Subsystem from Selection** oder **Kontextmenü -> Create Subsystem from Selection** wählen.

9.4.2 Subsystem-Block



Nach dem Erzeugen des **Subsystem-Blocks** zeigt sich das Modell wie abgebildet. Mit dem Model-Browser (Menü View->Model Browser) kann man einfach im gesamten Simulink-Modell navigieren.

In diesem Beispiel gibt es jeweils nur ein Ein- und ein Ausgangssignal. Es können aber auch mehrere Signale in den Subsystem-Block geführt und mehrere Signale aus dem Subsystem herausgeführt werden. Beide Modelle werden in einer einzigen Datei gespeichert!

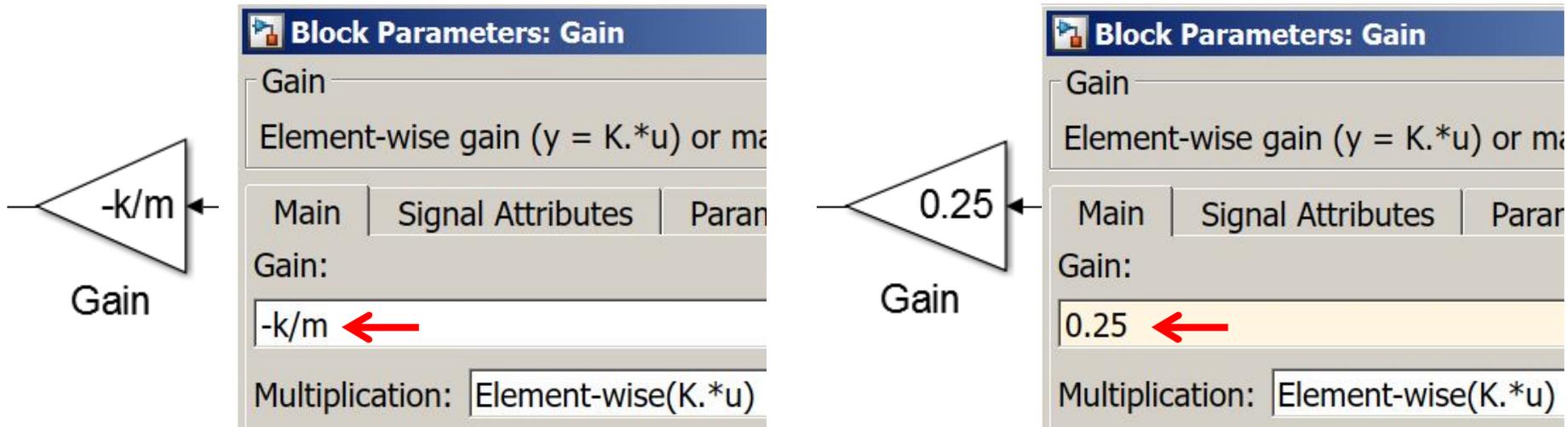


Ein Subsystem ist ein „Simulink-Modell“ mit Ein- und Ausgängen

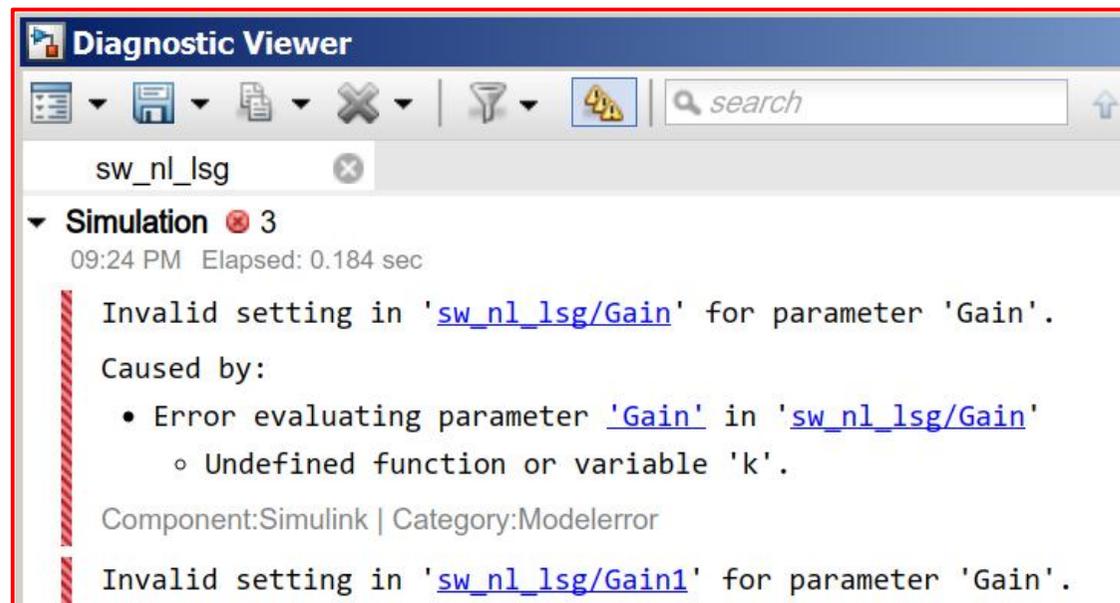
`sw_nl_subsys.slx`

Variablen, die im **MATLAB-Workspace** existieren, dürfen als **Parameter** in Simulink-Blöcken verwendet werden.

Beispiel : Gain-Block



Wird die Simulation gestartet und die Variablen **k** oder **m** sind nicht definiert, dann bricht die Simulation ab und es erscheint eine Fehlermeldung.



Simulink hat Zugriff auf **zwei verschiedene Workspaces** :

- **MATLAB-Workspace** (= Base Workspace)
- **Model-Workspace**

Jedem Simulink-Modell ist ein eigener Workspace zugeordnet, der sogenannte **Model-Workspace**. Variablen aus dem Model-Workspace werden innerhalb eines Simulink-Modells definiert und können nur innerhalb des Modells verwendet werden. Die Definition dieser Variablen erfolgt über den Model-Explorer.

Vorteil: Variablen aus dem Model-Workspace gehören zum Modell und werden im Simulink-Modell gespeichert (in der Datei mit der Endung slx).

Variablen im Model-Workspace haben Vorrang vor Variablen im MATLAB-Workspace.

The screenshot shows the Simulink Model Explorer window. The title bar reads "Model Explorer". The menu bar includes "File", "Edit", "View", "Tools", "Add", and "Help". The toolbar contains various icons for file operations and workspace management. A red arrow points to the "Workspace" icon in the toolbar. A yellow box highlights the text "modelworkspace.slx" in the top right corner. The "Search" field is set to "by Name". The "Model Hierarchy" pane on the left shows a tree structure with "Simulink Root" at the top, followed by "Base Workspace", "modelworkspace", "Model Workspace", "Configuration (Active)", "Code for modelworkspace", and "Advice for modelworkspace". A red arrow points to "Base Workspace" and a blue arrow points to "Model Workspace". The main pane shows the "Contents of: Model Workspace (only)" with a "Filter Contents" field. The "Column View" is set to "Data Objects", and there are "Show Details" and "2 object(s)" options. A table displays the contents of the Model Workspace:

Name	Value	DataType	Min	Max
k	3	double (...)		
m	7	double (...)		

The right pane shows "Model Workspace" settings, including "Workspace data" and "Data source: Model File". There are "Revert" and "Help" buttons at the bottom right.

Configuration Parameters: modelworkspace/Configuration (Active)

★ Commonly Used Parameters | ≡ All Parameters

Select: Solver

Simulation time
Start time: 0.0 Stop time: 10.0

Solver options
Type: Variable-step Solver: ode45 (Dormand-Prince)

Additional options
Max step size: 0.01 Relative
Min step size: auto Absolute
Initial step size: auto Shape p
Number of consecutive min steps: 1

Configuration Parameters: modelworkspace/Configuration (Active)

★ Commonly Used Parameters | ≡ All Parameters

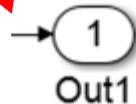
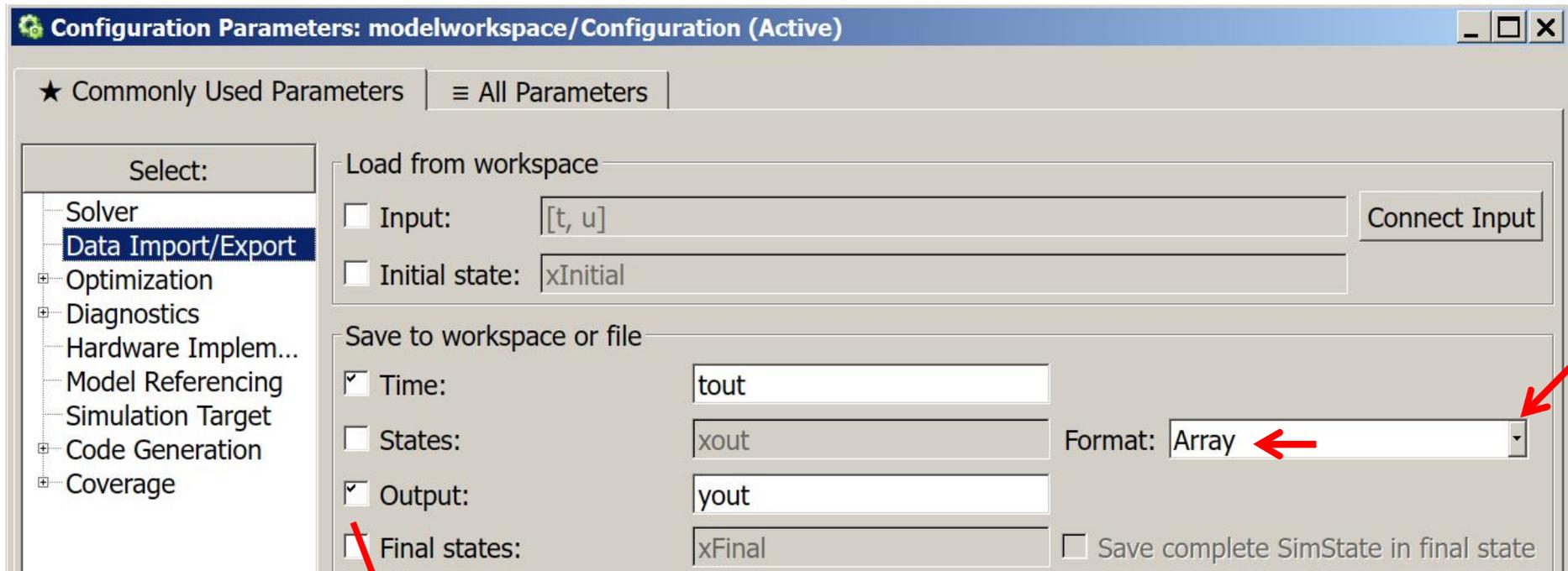
Select: Solver

Simulation time
Start time: 0.0 Stop time: 10.0

Solver options
Type: Fixed-step Solver: auto (Automatic solver selection)

Additional options
Fixed-step size (fundamental sample time): auto

Tasking and sample time options
Periodic sample time constraint:
 Treat each discrete rate as a separate task



Die erste Spalte der Matrix `yout` enthält die Werte des **Block Out 1**. Dieser Block muss im Modell vorhanden sein, ansonsten werden keine Werte gespeichert. Die zweite Spalte der Matrix enthält die Werte des Blocks **Out 2**, falls dieser vorhanden ist, ...

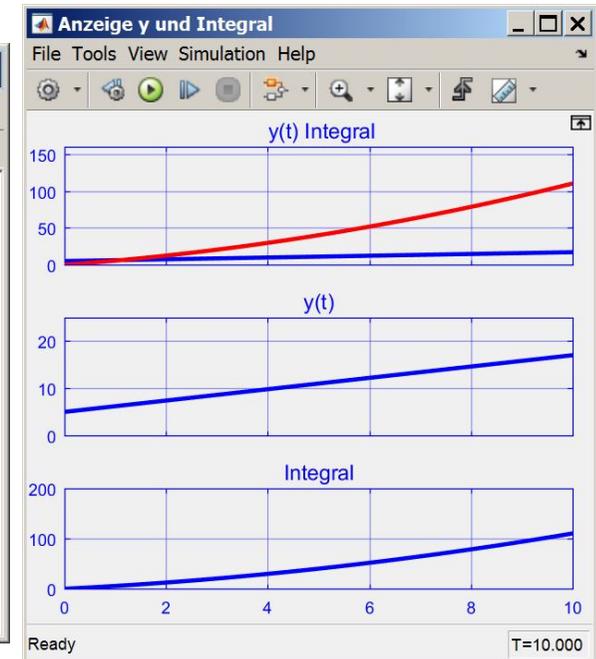
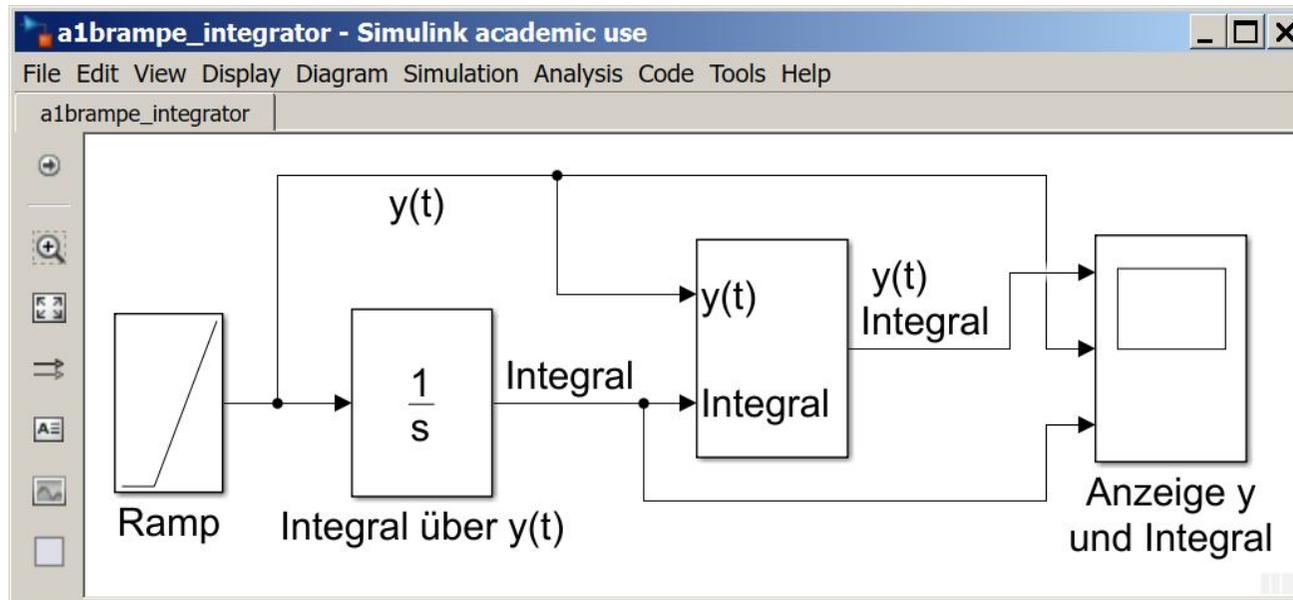
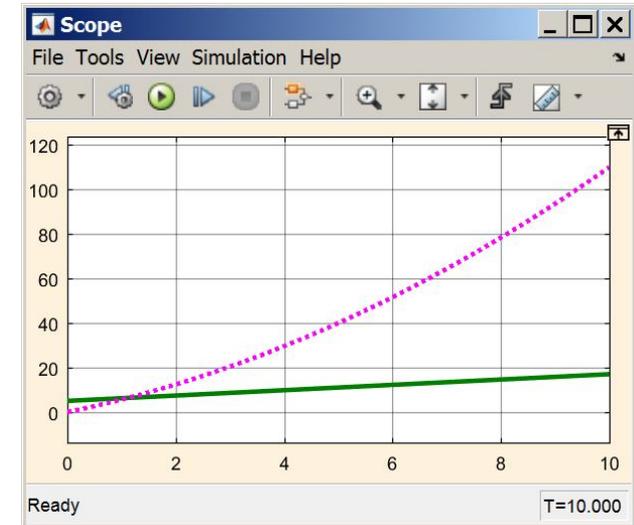
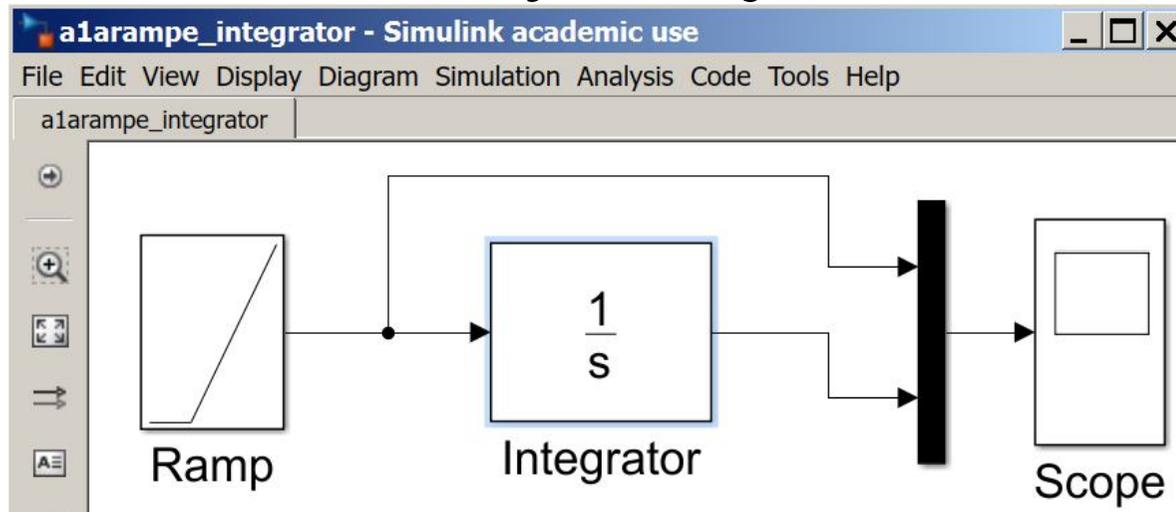
Die Zeilen der Matrix beziehen sich auf die Zeitpunkte, die in `tout` gespeichert werden.



The screenshot shows the 'Block Parameters: To Workspace' dialog box. The dialog has a title bar with a close button. The main content is organized into sections:

- To Workspace:** A text area containing the description: "Write input to specified timeseries, array, or structure in a workspace. For menu-based simulation, data is written in the MATLAB base workspace. Data is not available until the simulation is stopped or paused." Below this is a note: "To log a bus signal, use 'Timeseries' save format."
- Parameters:** A section with several input fields:
 - Variable name:** A text field containing 'erg', with a red arrow pointing to it from the left.
 - Limit data points to last:** A text field containing 'inf'.
 - Decimation:** A text field containing '1', with a red arrow pointing to it from the left.
 - Save format:** A dropdown menu currently showing 'Timeseries'. A red arrow points to the dropdown arrow on the right. The dropdown list is open, showing the following options: 'Timeseries' (highlighted in blue), 'Structure With Time', 'Structure', 'Array', and 'Timeseries'.
- Log fixed-p:** A checkbox that is checked.
- Sample time (:** A text field containing '-1'.

Aufgabe 1 : Erstellen Sie zwei Simulink-Modelle, die die Funktion $y(t) = 1.2 \cdot t + 5$ im Intervall von $[0, 10]$ integrieren und die Ergebnisse wie unten gezeigt darstellen. Wie lautet das analytische Ergebnis?



Aufgabe 2 :

$$\frac{d^2 y}{dt^2} + 2 \cdot \delta \cdot \frac{dy}{dt} + \frac{k}{m} \cdot y(t) + \frac{c}{m} \cdot y(t)^3 = 0$$

$$v_0 = \dot{y}(t=0) = 0, \quad y_0 = y(t=0) = 2$$

Lösen Sie die obige DGL mit Hilfe von Simulink. Der Ausdruck für die zweite Ableitung

$$-2 \cdot \delta \cdot \frac{dy}{dt} - \frac{k}{m} \cdot y(t) - \frac{c}{m} \cdot y(t)^3$$

wird auf verschiedene Arten berechnet:

- Mit Hilfe eines Function-Blocks (hierzu ist auch ein Mux-Block notwendig). Was muss im Feld "Expression" des Function-Blocks eingetragen werden?
- Mit einem Subsystem-Block, der zwei Eingänge besitzt.
- Mit einem Subsystem-Block, der nur einen Eingang besitzt. Verwenden Sie innerhalb des Subsystem-Blocks einen Demux-Block (Gegenstück zum Mux-Block).

Aufgabe 3 :

Ein Flugzeug verwendet einen Bremsfallschirm und andere Mittel um nach der Landung zu bremsen. Die Änderung der Geschwindigkeit v beim Landen ist gegeben durch:

$$\frac{dv}{dt} = -0.0035 \cdot v^2 - 3$$

Zur Zeit $t=0$ befindet sich das Flugzeug an der Position $s=0$. Es öffnet den Bremsfallschirm und beginnt den Bremsvorgang. Das Flugzeug besitzt eine Geschwindigkeit von 300km/h.

Erstellen Sie ein Simulink-Modell, das die Geschwindigkeit und den zurückgelegten Weg als Funktion der Zeit im Intervall $[0,12]$ berechnet und beide Größen in einem Scope-Block mit zwei Eingängen darstellt.

• Lösung 1:

Das Simulink-Modell darf nur die Blöcke **Sum**, **Product**, **Constant**, **Gain**, **Integrator** (external Initial Condition) und **Scope** enthalten. Welche Werte müssen in den Constant-Blöcken und im Gain-Block eingetragen werden?

• Lösung 2:

Das Simulink-Modell darf nur die Blöcke **Function-Block**, **Constant**, **Integrator** (external Initial Condition) und **Scope** enthalten. Welcher Wert muss im Constant-Block eingetragen werden? Geben Sie den Ausdruck an, der im Function-Block eingetragen werden muss. Der zurückgelegte Weg muss nicht berechnet werden.

Praktikum

Ingenieurinformatik

Teilmodul II

Numerik für Ingenieure

(Einführung in MATLAB/Simulink)

Sicherheitsunterweisung:

- Es gilt die Laborordnung
- In den Laboren nicht Rauchen, Essen und Trinken
- Fluchtwege aus dem Labor auf den Flur ins Treppenhaus
- Grüne Fluchtwegemarkierungen an der Flurdecke
- Feuerlöscher auf dem Flur, Feuermelder in beiden Treppenhäusern
- Im Brandfall keinen Aufzug benutzen (möglicher Stromausfall)
- Im Brandfall die Fenster geschlossen halten
- Informationen an den Türen:
Verhalten im Brandfall, Rufnummern für den Notfall, erste Hilfe

Ein Unfall – was ist zu tun?

- Verbandskästen in den Räumen B362 , B372 , B0055 (Sekretariat)
- Notausschalter sind in allen KCA-Laboren vorhanden

Bei Hard- und Softwareproblemen (Login nicht möglich, Fragen zu Netzwerklaufwerken/Backups usw.) helfen die Herren Schneider (Raum B350), Wagner und Tonch (B372).

- 1 MATLAB Entwicklungsumgebung**
- 2 Ein- und Ausgabe – Funktionen zeichnen**
- 3 Debugger und Kontrollstrukturen**
- 4 Gleichungssysteme, Eigenwerte und Eigenvektoren**
- 5 Numerische Lösung von Differentialgleichungen**
- 6 Simulink – Übungsbeispiele**

- 1 MATLAB starten**
- 2 MATLAB - Entwicklungsumgebung**
- 3 Arbeiten mit Skalaren**
- 4 Arbeiten mit Vektoren und Matrizen**
- 5 MATLAB - Skripte**
- 6 MATLAB - Funktionen**
- 7 Matrizen – Drehung in der Ebene**
- 8 MATLAB – Hilfe – Nützliche Befehle**
- 9 Hausaufgaben**

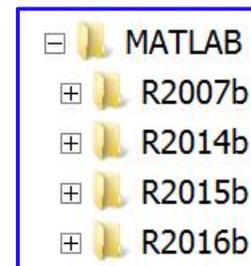
Im Praktikum wird MATLAB in der Version R2016b verwendet.
MATLAB ist auf allen Rechnern im 3-ten Stockwerk installiert

MATLAB starten :

- MATLAB – Icon doppelklicken
- Eintrag im Startmenü wählen
- Start -> Alle Programme -> MATLAB -> R2016b



Auf einem Rechner können verschiedene
Versionen von MATLAB installiert sein.



The screenshot shows the MATLAB R2016b - classroom use interface. The top menu bar includes HOME, PLOTS, and APPS. The toolstrip below contains various icons for file operations, workspace management, code execution, and environment settings. A red box highlights the 'U: \ Matlab' path in the Current Folder browser. The Command Window shows the prompt 'fx >>'. The Workspace browser is empty. The Command History shows a timestamp '%-- 03.10.2016 10:5...'. Red arrows point to the HOME tab, the search bar, and the Current Folder path. Yellow callout boxes provide descriptions for the Current Folder, Command Window, Workspace, and Command History.

Toolstrip

Current Folder

- Arbeitsverzeichnis
- Dateiverwaltung
- Zugriff auf M-Files
 - Skripte
 - Funktionen

Command Window

- Befehle eingeben
- Ergebnisse ausgeben
- Fehlermeldungen anzeigen
- Skripte ausführen
- Funktionen aufrufen

Workspace

- Liste der aktuellen Variablen und ihrer Eigenschaften

Command History

- Liste der bisher eingegebenen Befehle
- Befehle wählen und erneut ausführen



Command Window

```
>> r = 5
r =
    5

>> format compact
>> r
r =
    5

>> 2*pi*r
ans =
    31.4159

>> u = 2*pi*r
u =
    31.4159

>> u = 2*pi*10;
>> u
u =
    62.8319

>> whos
Name      Size      Bytes      Class
ans       1x1         8      double
r         1x1         8      double
u         1x1         8      double
```

Workspace

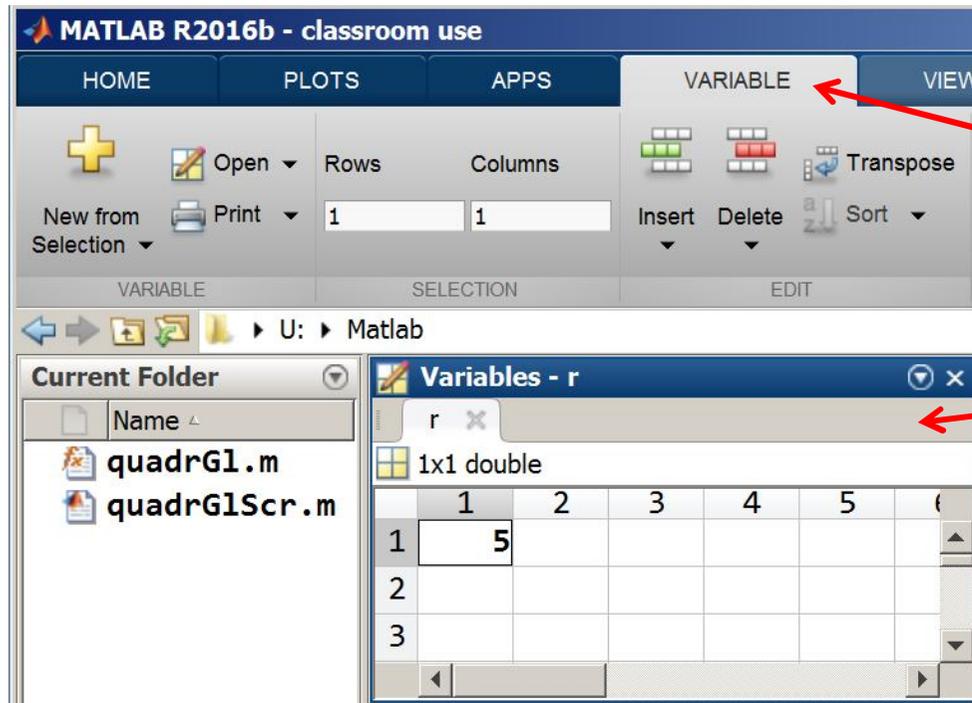
Name	Value	Min	Max
ans	31.4159	31.4159	31.4159
r	5	5	5
u	62.8319	62.8319	62.8319

Die Variable **ans** speichert das Ergebnis einer Anweisung, wenn dieses nicht explizit einer Variablen zugewiesen wird.

pi ist eine vordefinierte Variable und besitzt den Wert 3.1415 ...

Ein **;** (Semikolon) am Ende einer Anweisung unterdrückt die Ausgabe des Ergebnisses im Command Window. Die Anweisung wird aber ausgeführt!

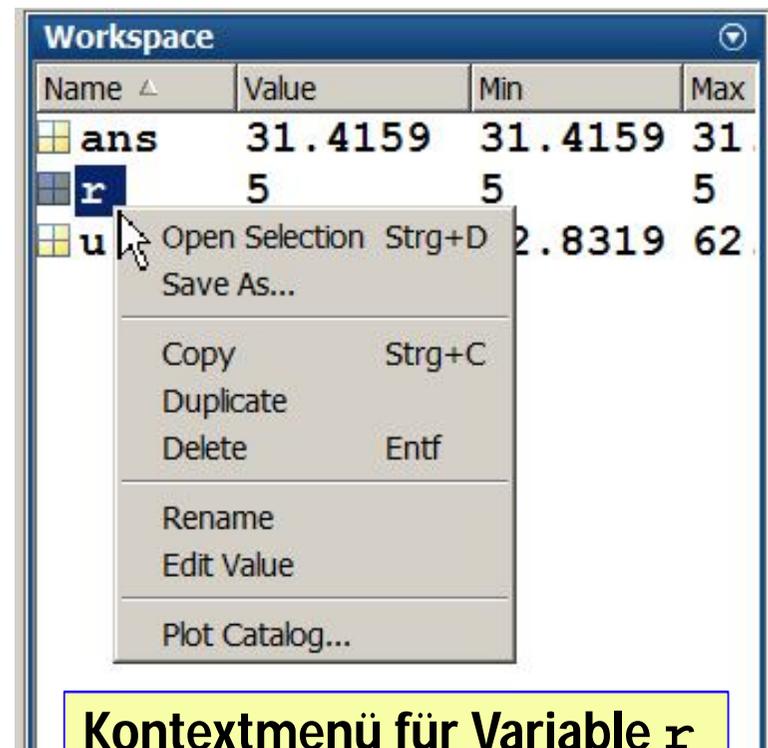
Der Befehl **whos** listet alle Variablen auf, die aktuell Workspace vorhanden sind.



Tab Variable

Variable-Editor

Ein Doppelklick auf den Namen einer Variablen im Workspace-Fenster öffnet den **Variable-Editor** und stellt diese Variable dar – hier die Variable **r**. Dieser Editor ist für die Bearbeitung von Matrizen sehr nützlich – Zeilen und Spalten einfügen oder löschen, Werte ändern. Beachte : Im Toolstrip erscheint ein neuer Tab, wenn der Variable-Editor geöffnet ist.



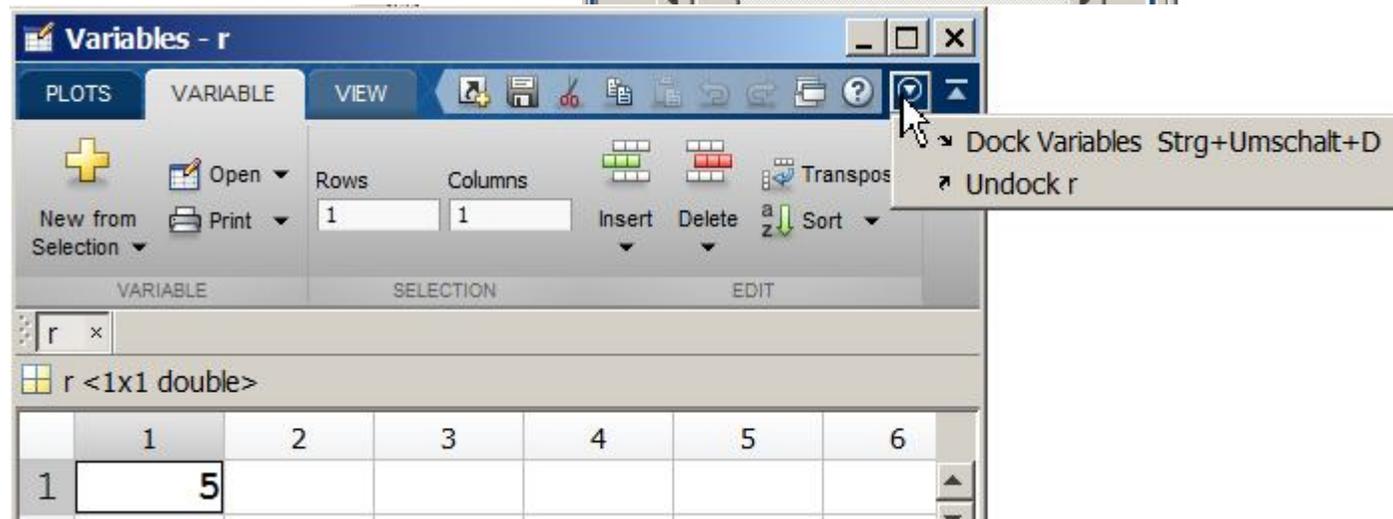
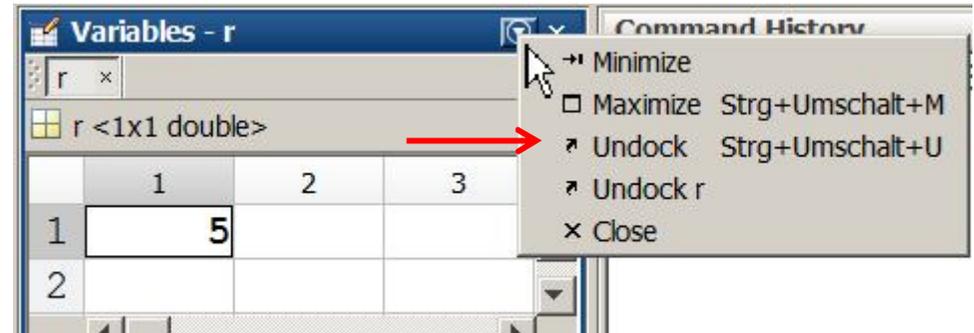
Kontextmenü für Variable r

Anmerkungen :

- Bei MATLAB müssen Variablen nicht explizit definiert werden. Der Befehl `r = 5` erzeugt automatisch eine Variable vom Typ `double` mit dem Namen `r` und weist dieser den Wert 5 zu. Der Typ `double` ist der Standardtyp (Default-Typ) für Variablen.
- Der Befehl `'format compact'` bewirkt eine kompakte Darstellung der Ausgabe im Command-Window. Der Abstand zwischen den Zeilen ist klein.
- Die Variable `pi` ist eine vordefinierte Variable mit dem Wert `3.1415...`
- Die Variable `ans` ist ebenfalls eine vordefinierte Variable. Wird das Ergebnis eines Ausdrucks nicht explizit einer Variablen zugewiesen, so wird das Ergebnis der Variablen `ans` zugewiesen. Der alte Wert von `ans` wird überschrieben.
- Ein **Semikolon ;** am Ende einer Anweisung unterdrückt die Ausgabe des Ergebnisses im Command-Window - der Ausdruck wird aber ausgeführt !
- Der Befehl `whos` gibt Informationen über die aktuell im Workspace vorhandenen Variablen aus. `'Size 1x1'` bedeutet, dass die entsprechende Variable eine 1*1-Matrix (d.h. ein Skalar) ist; in der Spalte `Bytes` wird der Speicherplatzbedarf der Variablen angezeigt; die Spalte `Class` gibt den Datentyp an.

Aufgaben

1. Geben Sie die Befehle $r=5, \dots$ nacheinander im Command- Fenster ein.
2. Ändern Sie den Wert von r über das Kontextmenü der Variablen auf 7.
3. Ändern Sie den Wert von r mit Hilfe des Variable-Editors auf den Wert 10.
4. Ändern Sie den Namen der Variablen in $r1$.
5. Geben Sie dann r und $r1$ im Command-Window ein.
6. Löschen Sie die Variable $r1$.
7. Berechnen Sie $\sqrt{-2}$
8. Wählen Sie „Undock“ für den Variable-Editor. Machen Sie den Vorgang wieder rückgängig, d.h. docken Sie das Fenster wieder an.



```
Command Window
>> A = [1, 2; 3, 4]
A =
     1     2
     3     4
>> x = [2; 3]
x =
     2
     3
>> y = A*x
y =
     8
    18
>> A(2,2)
ans =
     4
>> x(2)
ans =
     3
>> x(2,1)
ans =
     3

>> whos
  Name      Size      Bytes   Class
  A         2x2         32    double
  y         2x1         16    double
  x         2x1         16    double
>> A(3,3) = 9
A =
     1     2     0
     3     4     0
     0     0     9
>> whos
  Name      Size      Bytes   Class
  A         3x3         72    double
  y         2x1         16    double
  x         2x1         16    double
>> A*x
Error using *
Inner matrix dimensions must agree.
>> A(3,4)
Index exceeds matrix dimensions.
```

Aufgaben

1. Geben Sie die Befehle $A = [1, 2; 3, 4], \dots$ (siehe vorherige Folie) der Reihe nach im Command-Window ein. Rechnen Sie die Matrix-Vektor-Multiplikation auf einem Blatt Papier nach! **Im SS2013 konnte die Hälfte der Teilnehmer/innen der Informatik-Klausur diese Aufgabe nicht korrekt lösen!**
2. Nach der Eingabe aller Befehle ist die Matrix A eine 3*3-Matrix. Löschen Sie mit Hilfe des Variable-Editors die 3-te Spalte und dann die 3-te Zeile der Matrix. Sie erhalten dann wieder eine 2*2-Matrix. Geben Sie diese aus.
3. Erzeugen Sie aus der 2*2-Matrix A eine 5*5-Matrix mit Hilfe des Variable-Editors.

Anmerkungen :

- Bei der **Definition** von Vektoren und Matrizen werden **[]-Klammern** verwendet. **Zeilen** werden bei der Definition durch einen Semikolon **;** voneinander getrennt, die **Spalten** durch **Leerzeichen (Space)** oder **Komma ,**.
- Beim **Zugriff auf die Elemente** einer Matrix werden **()-Klammern** verwendet.
- Für einen Zeilenvektor gilt : $x(k) == x(1, k)$
- Für einen Spaltenvektor gilt : $x(k) == x(k, 1)$
- **Konvention** : Namen von Matrizen werden in Großbuchstaben geschrieben.
- MATLAB unterscheidet Groß- und Kleinschreibung

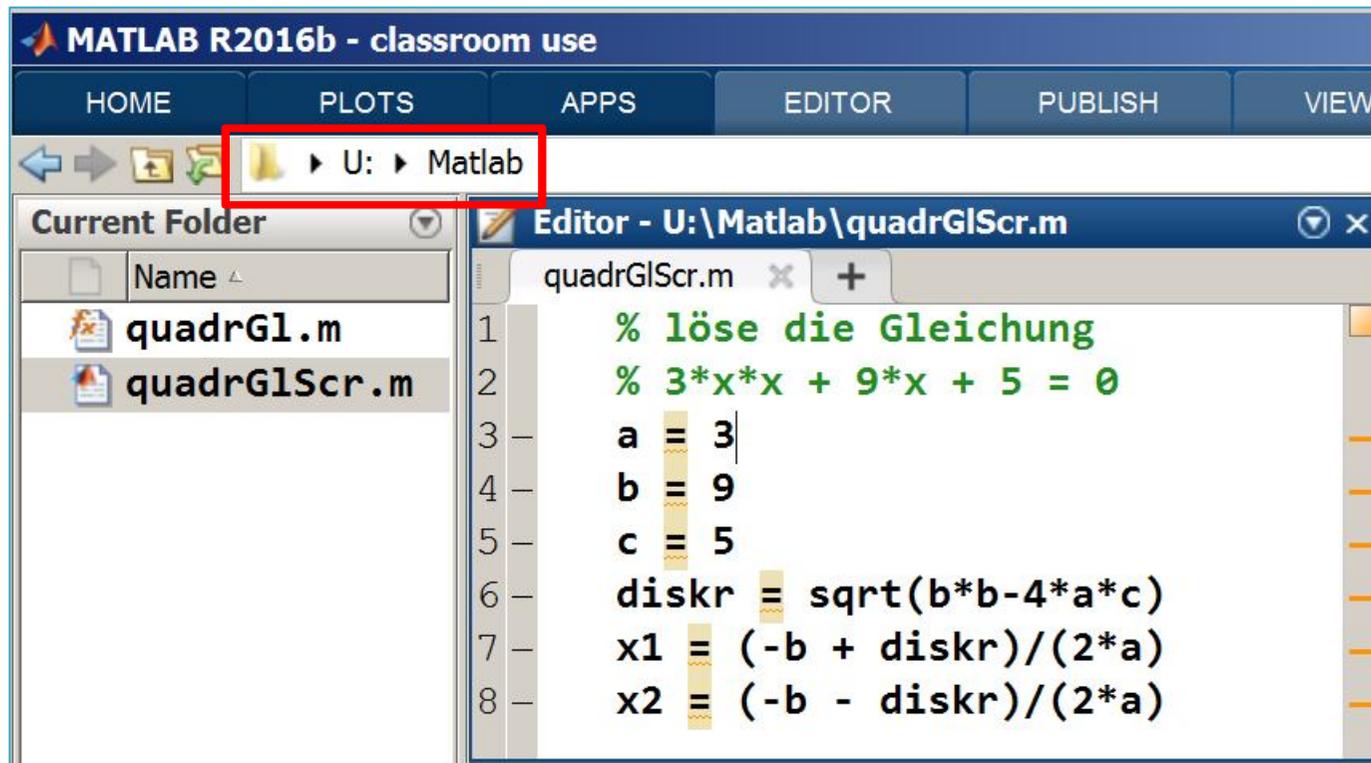
linear indexing !

Ein MATLAB-Skript ist eine ASCII-Datei, die MATLAB-Befehle enthält.

Wird das Skript ausgeführt, dann werden die Befehle, die in der Datei gespeichert sind, der Reihe nach abgearbeitet, so als würde man die Befehle hintereinander im Command-Window eingeben.

Aufgabe :

Erstellen Sie ein Skript `quadrG1Scr.m` zur Lösung einer quadratischen Gleichung. Speichern Sie das Skript auf dem U-Laufwerk im Unterverzeichnis `matlab` (`U:\matlab\quadrG1Scr.m`).

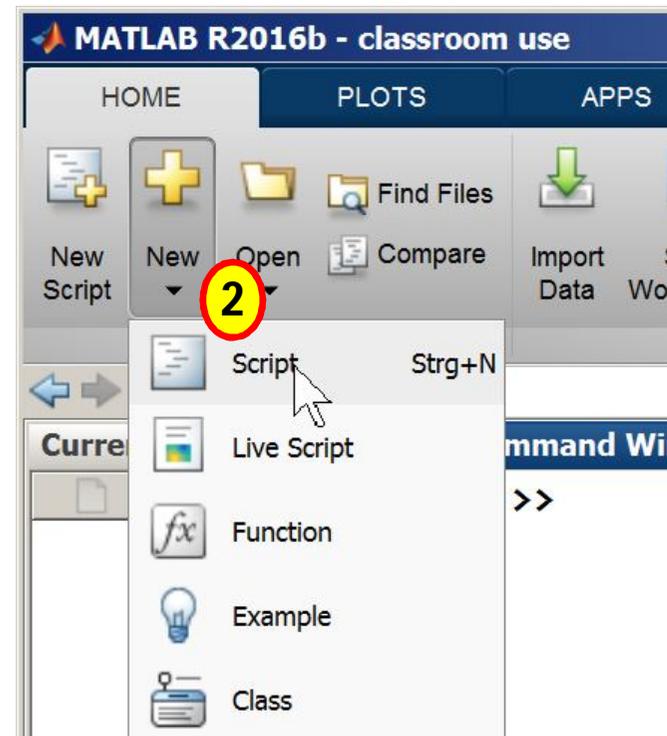
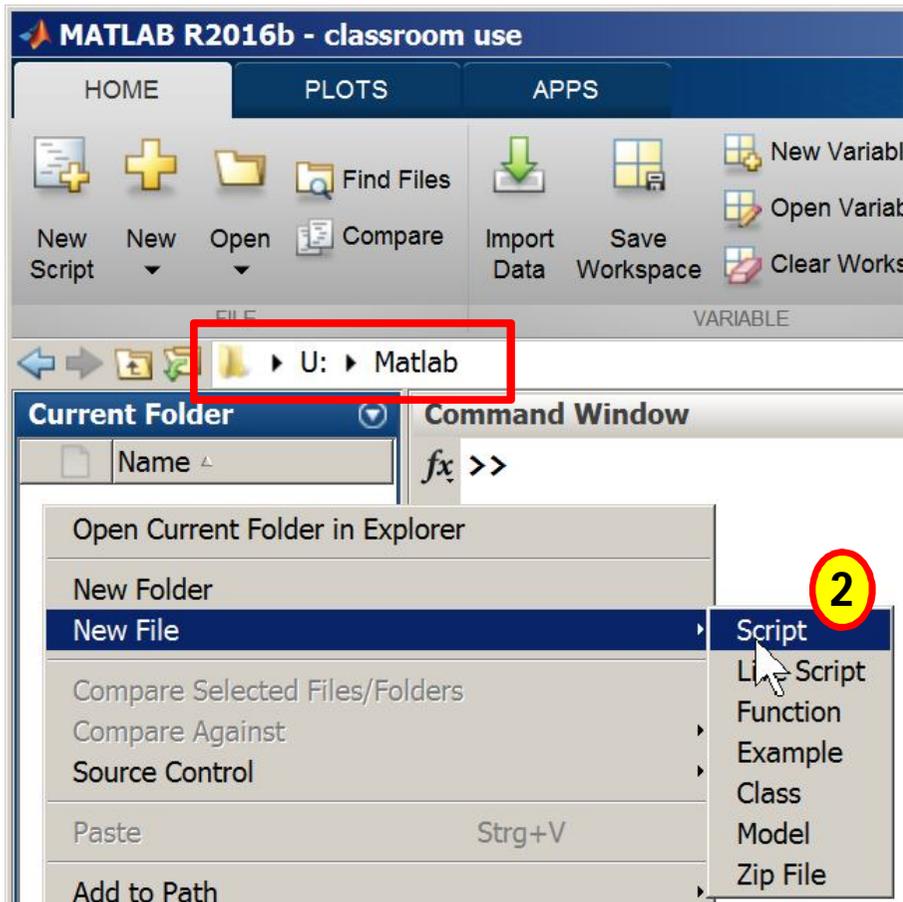


The screenshot shows the MATLAB R2016b interface. The top menu bar includes HOME, PLOTS, APPS, EDITOR, PUBLISH, and VIEW. The current folder is U:\Matlab, highlighted with a red box. The editor window displays the following MATLAB code:

```
1 % löse die Gleichung
2 % 3*x*x + 9*x + 5 = 0
3 a = 3
4 b = 9
5 c = 5
6 disk = sqrt(b*b-4*a*c)
7 x1 = (-b + disk)/(2*a)
8 x2 = (-b - disk)/(2*a)
```

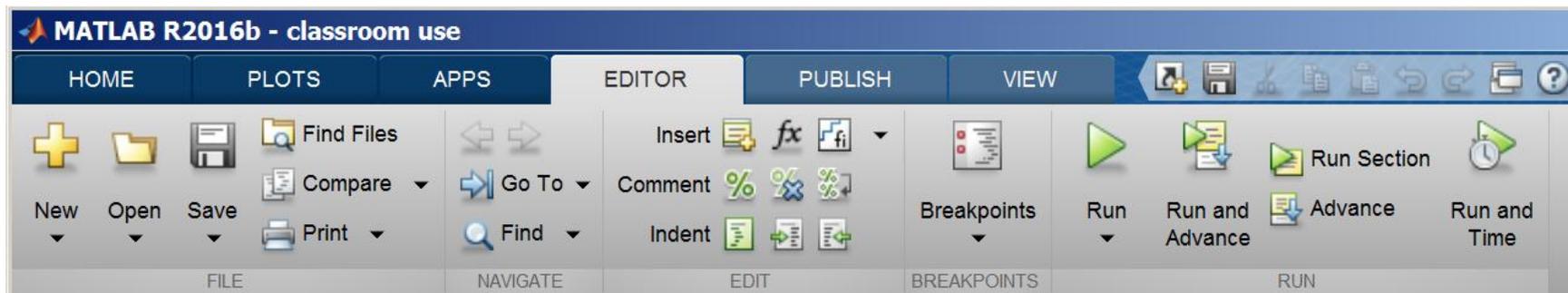
Teilaufgaben :

- 1) Legen Sie auf dem U-Laufwerk ein Verzeichnis mit Namen `matlab` an und setzen Sie dann den „Current Folder“ in MATLAB auf `U:\matlab`.
- 2) Erzeugen Sie ein MATLAB-Skript (neue Datei) mit Namen `quadrGlScr.m`. Hierzu wird entweder das Kontextmenü im Fenster Current Folder aktiviert oder es wird im Tab Home „New Script“ oder „New“ gewählt.



- 3) Schreiben Sie das Skript `quadrG1Scr.m` und speichern sie es.
- 4) Führen Sie das Skript aus. Hierzu gibt es verschiedene Möglichkeiten :
 - a) Kontextmenü der Datei `quadrG1Scr.m` öffnen und den Befehl `Run` wählen.
 - b) Die Datei `quadrG1Scr.m` im Current Folder mit der Maus selektieren und dann in das Command-Window ziehen.
 - c) Im Command-Window `quadrG1Scr` eingeben.
 - d) Den Run-Button im Editor-Tab des Toolstrips anklicken. Hierzu muss das Skript im Editor geladen sein.

Starten Sie das Skript mit allen vier Varianten.



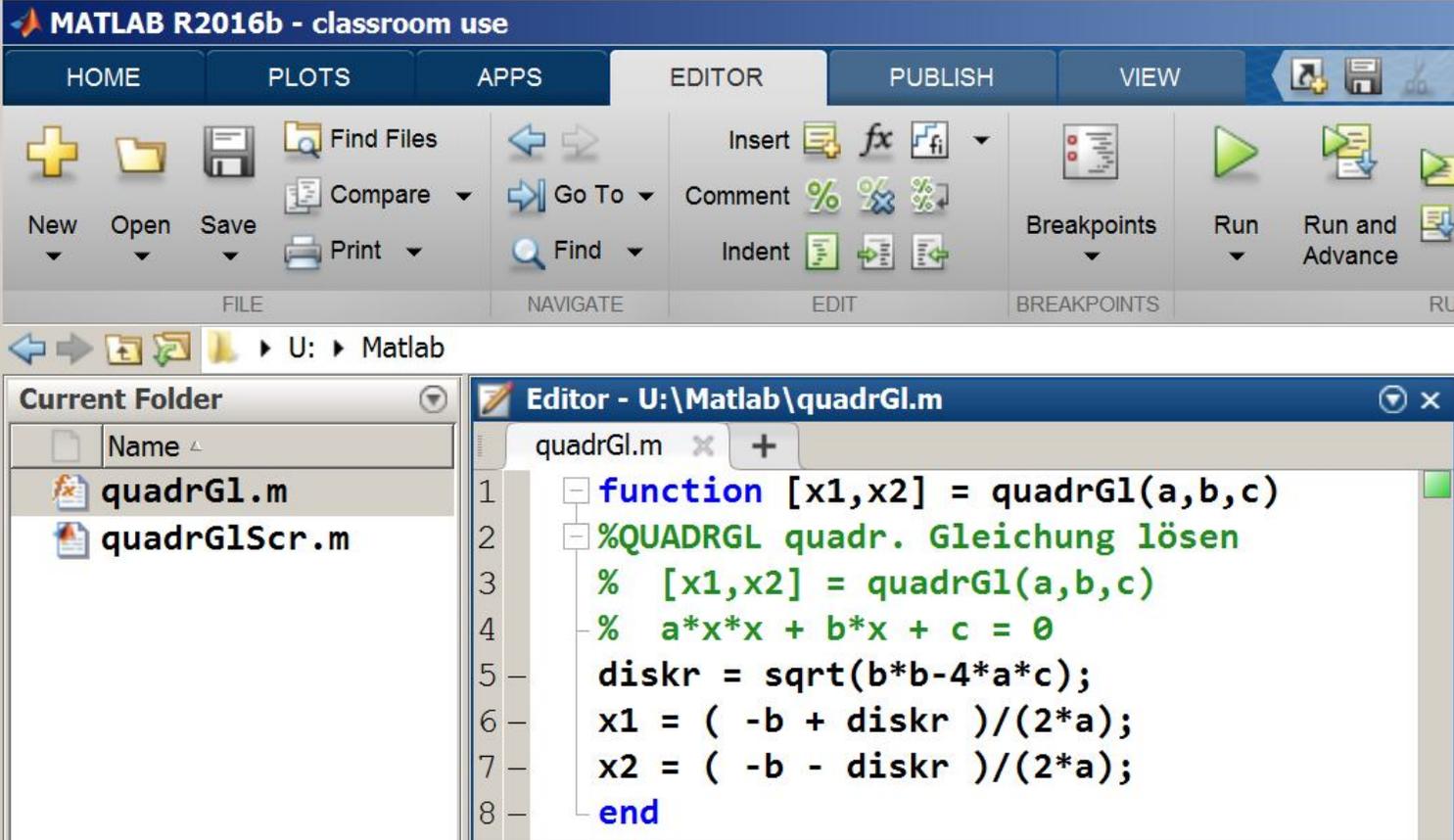
Beachten Sie, dass ein **Skript Zugriff auf die Variablen im Workspace** hat. Es können neue Variablen erzeugt oder existierende Variablen überschrieben werden. Verwenden Sie den `clear`-Befehl um sich davon zu überzeugen. Der Befehl `clear` löscht alle vorhandenen Variablen aus dem MATLAB-Workspace.

Aufgabe :

Erstellen Sie eine **Funktion quadrG1** zur Lösung einer quadratischen Gleichung der Form

$$a * x^2 + b * x + c = 0$$

An die Funktion werden die drei Koeffizienten der quadratischen Gleichung als Parameter übergeben. Als Ergebnis werden die beiden Lösungen der Gleichung zurückgegeben. Was passiert bei komplexen Lösungen ?

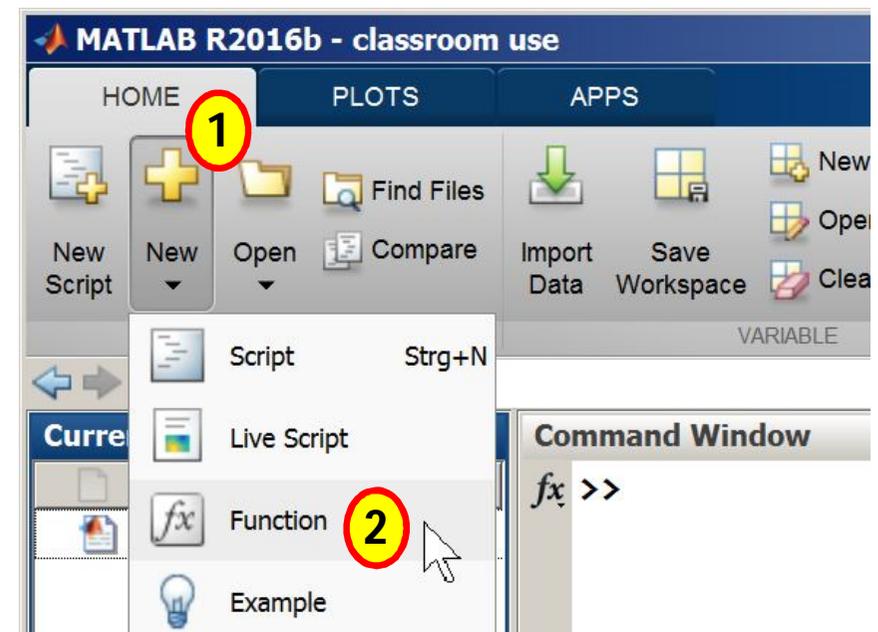
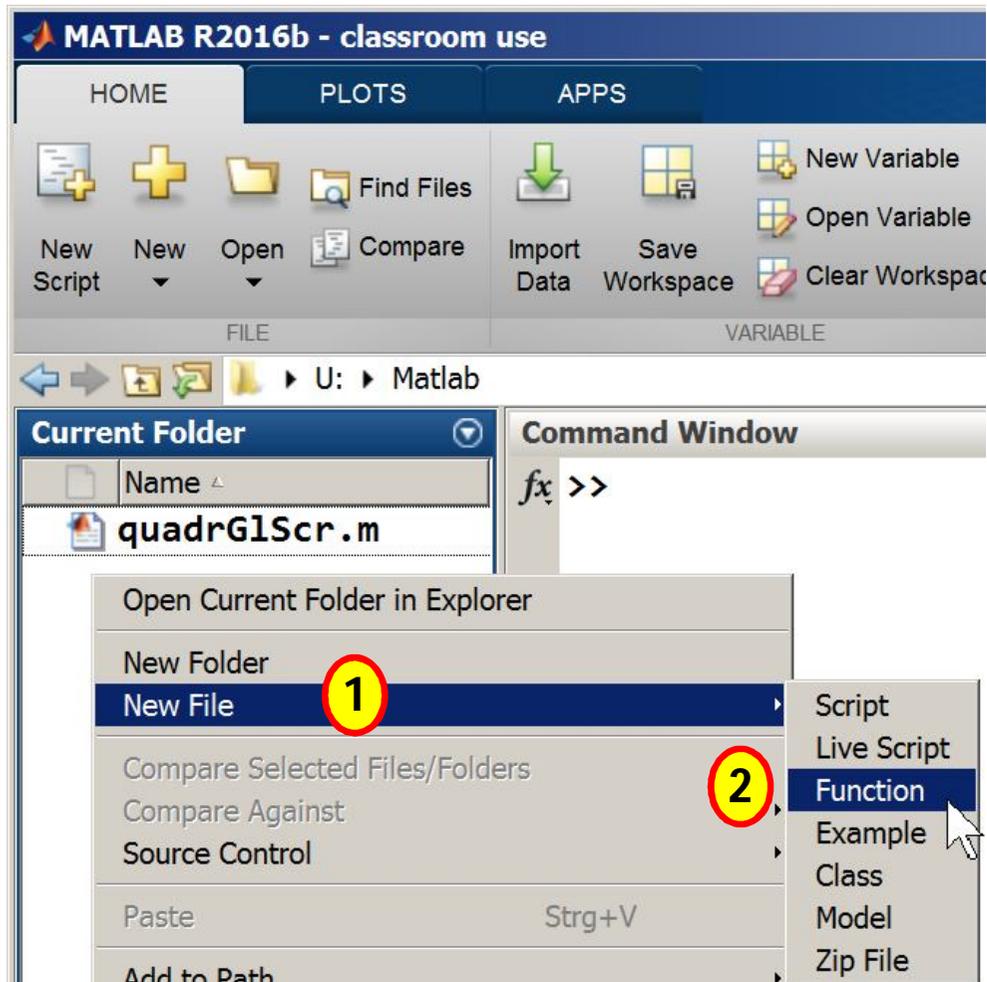


The screenshot shows the MATLAB R2016b interface. The Editor window displays the following code for the function `quadrG1.m`:

```
1 function [x1,x2] = quadrG1(a,b,c)
2 %QUADRGL quadr. Gleichung lösen
3 % [x1,x2] = quadrG1(a,b,c)
4 % a*x*x + b*x + c = 0
5 diskr = sqrt(b*b-4*a*c);
6 x1 = ( -b + diskr )/(2*a);
7 x2 = ( -b - diskr )/(2*a);
8 end
```

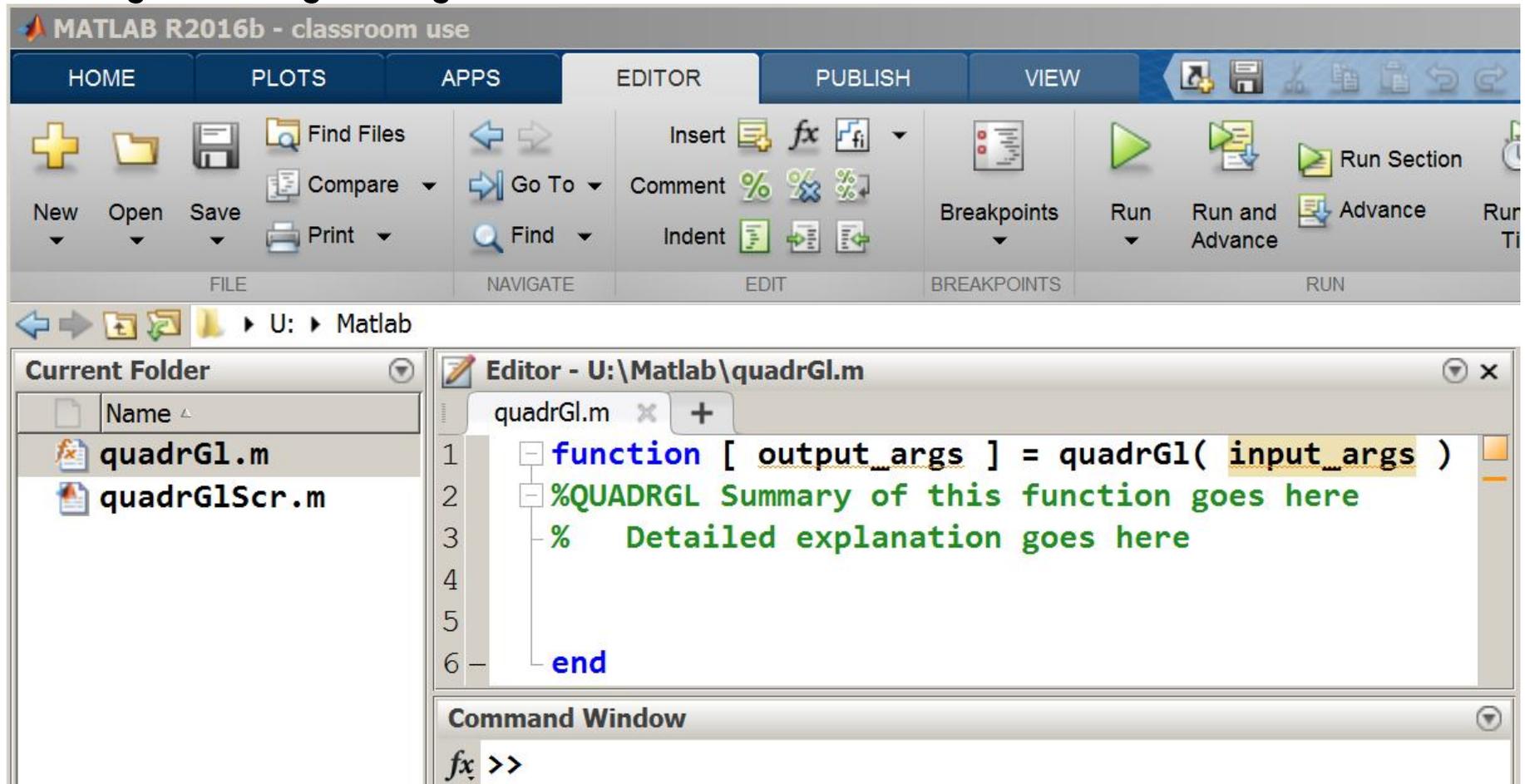
Teilaufgaben :

- 1) Erzeugen Sie eine neue Datei mit Namen `quadrG1.m` (Typ Function) - entweder das Kontextmenü im Fenster „Current Folder“ aktivieren und **Function** auswählen (links) oder über den Toolstrip **New -> Function** (rechts).



Teilaufgaben :

2) Ändern Sie den Namen der Datei von `Untitled.m` auf `quadrG1.m`, wenn Sie die Funktion über das Fenster „Current Folder“ erzeugt haben. MATLAB erzeugt ein Programmgerüst für die neue Funktion.



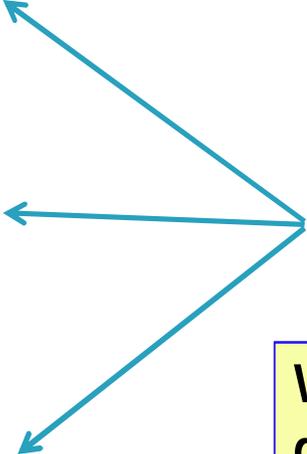
3) Programmieren Sie die Funktion `quadrG1.m`.

Teilaufgaben :

- 4) Testen Sie ihre Funktion für verschiedene Werte von a, b und c. Die Funktion kann auch komplexe Lösungen berechnen! Geben Sie den Hilfe-Text für die Funktion aus.
- 5) Was passiert, wenn Sie die Funktion mit zu wenig oder zu vielen Parametern aufrufen?

```
Command Window
>> [x1,x2]=quadrG1(2,6,3)
x1 =
    -0.6340
x2 =
    -2.3660
>> [y1,y2]=quadrG1(2,1,3)
y1 =
    -0.2500 + 1.1990i
y2 =
    -0.2500 - 1.1990i
>> z1=quadrG1(2,7,3)
z1 =
    -0.5000
>> help quadrG1
quadrG1 quad. Gleichung lösen
[x1,x2] = quadrG1(a,b,c)
a*x*x + b*x + c = 0
```

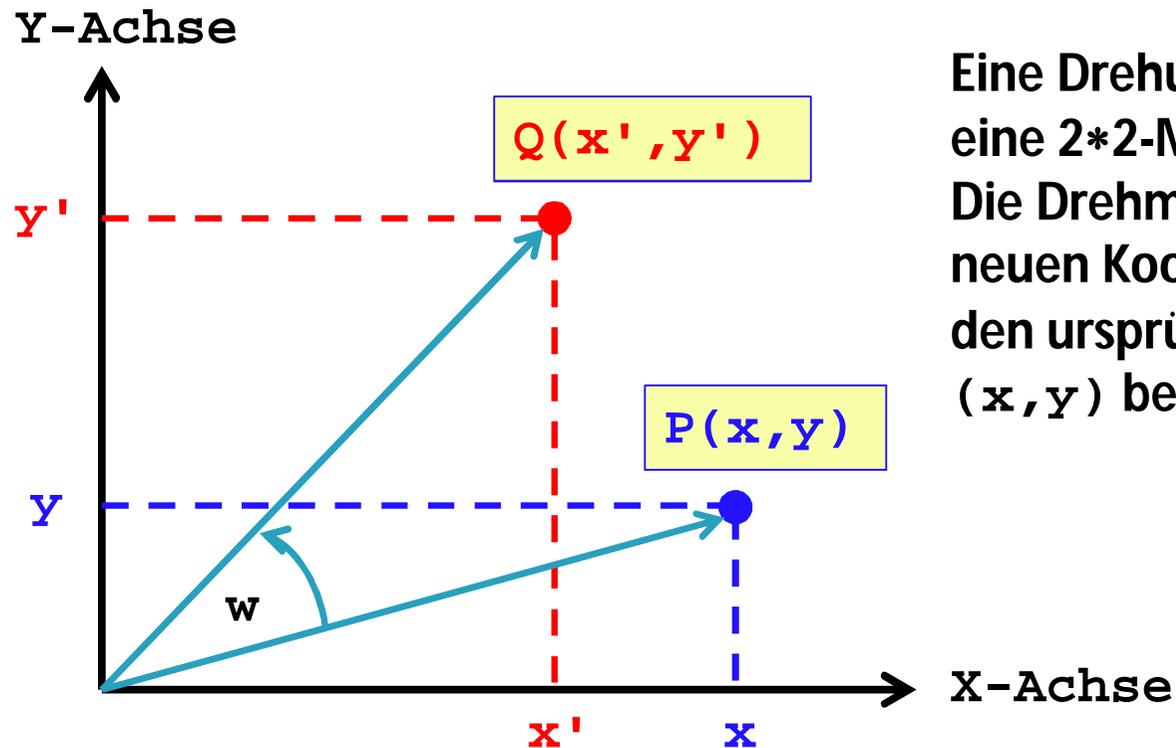
Aufruf der Funktion



Was ist der Unterschied zwischen diesen beiden Aufrufen ?

`[x1 ,x2] = quadrG1 (2 , 6 , 3)`

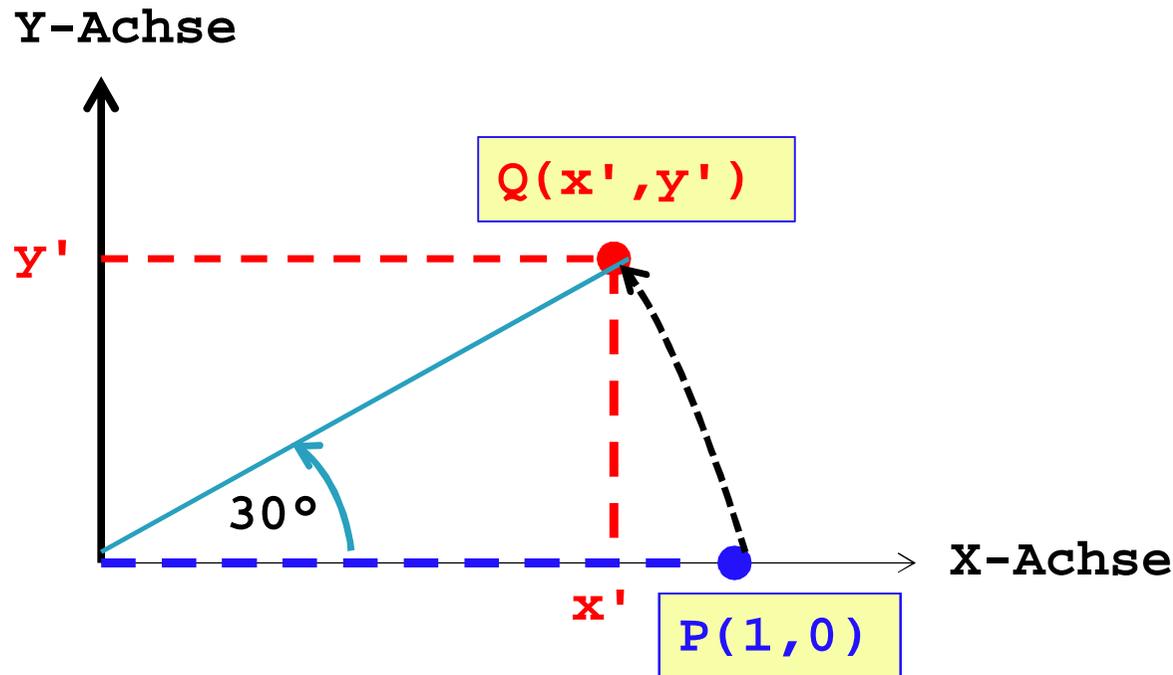
`[x1 ,x2] = quadrG1 (2 , 6 , 3) ;`



Eine Drehung in der Ebene kann durch eine 2*2-Matrix beschrieben werden. Die Drehmatrix gibt an, wie sich die neuen Koordinaten (x', y') aus den ursprünglichen Koordinaten (x, y) berechnen.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(w) & -\sin(w) \\ \sin(w) & \cos(w) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos(w) \cdot x - \sin(w) \cdot y \\ \sin(w) \cdot x + \cos(w) \cdot y \end{pmatrix}$$

Das Koordinatensystem ist fest. Der Punkt P mit den Koordinaten (x, y) wird um den Winkel w um den Ursprung gedreht. Die Koordinaten (x', y') des gedrehten Punktes berechnen sich nach der obigen Formel.



Spezialfall :

P liegt auf der x-Achse und wird um 30° gedreht : $w = 30^\circ$ $P(x=1, y=0)$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(30) & -\sin(30) \\ \sin(30) & \cos(30) \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \sqrt{3}/2 & -0.5 \\ 0.5 & \sqrt{3}/2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \sqrt{3}/2 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 0.866 \\ 0.5 \end{pmatrix}$$

Wird der Punkt P dreimal um jeweils 30° gedreht, dann liegt der gedrehte Punkt auf der y-Achse, d.h. er besitzt die Koordinaten $(0, 1)$.

P1_21 1.7 Matrizen – Drehung in der Ebene

Command Window

```
>> w = 30
w =
    30

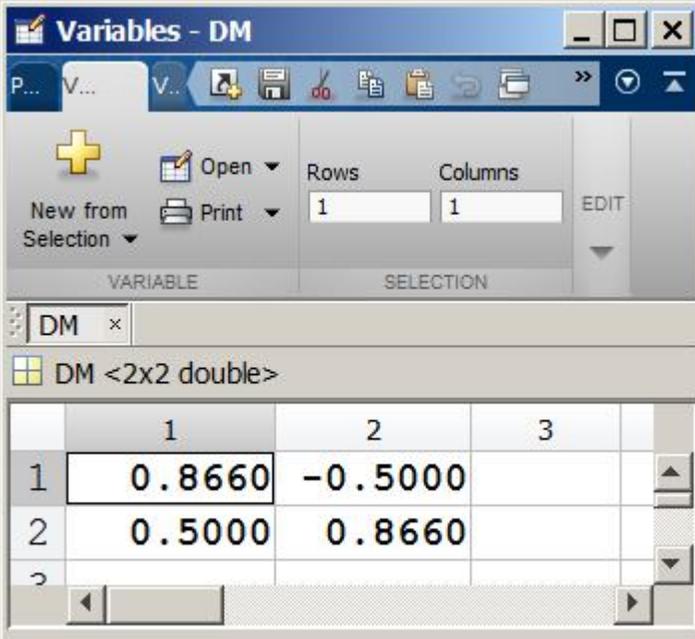
>> DM = [cosd(w), -sind(w); sind(w), cosd(w)]
DM =
    0.8660    -0.5000
    0.5000     0.8660

>> p = [1; 0]
p =
     1
     0

>> q = DM * p
q =
    0.8660
    0.5000

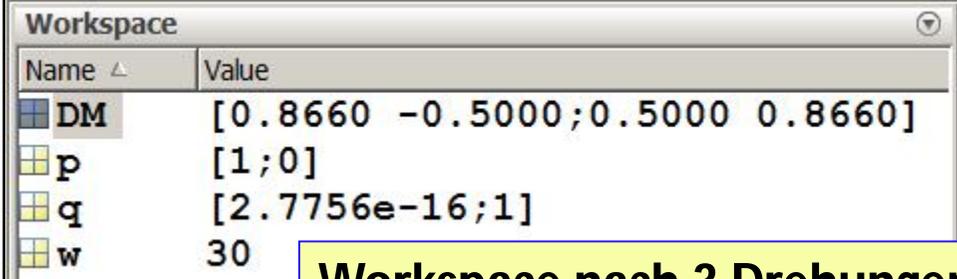
>> q = DM^3 * p
q =
    0.0000
    1.0000
```

sin : Winkel im Bogenmaß angeben
sind : Winkel im Gradmaß angeben



The screenshot shows a window titled 'Variables - DM' with a toolbar and a table. The table displays the values for the matrix DM:

	1	2	3
1	0.8660	-0.5000	
2	0.5000	0.8660	



The screenshot shows a 'Workspace' window with a table listing variables and their values:

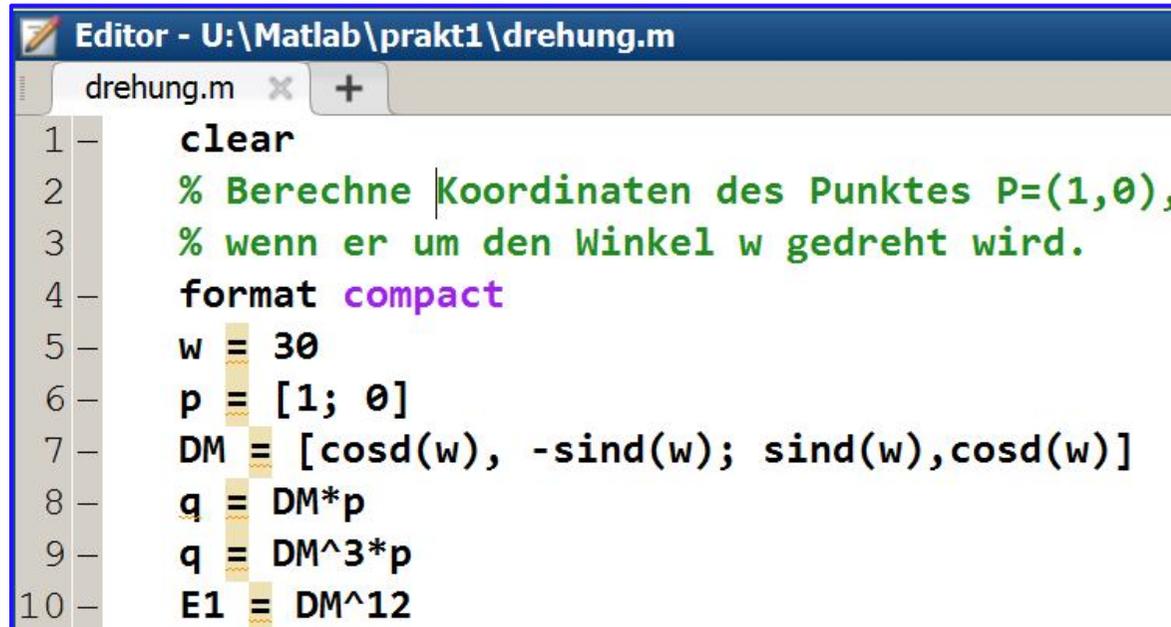
Name	Value
DM	[0.8660 -0.5000; 0.5000 0.8660]
p	[1; 0]
q	[2.7756e-16; 1]
w	30

Workspace nach 3 Drehungen

Die Befehle zur Berechnung der Koordinaten eines Punktes nach einer Drehung werden in einem Skript gespeichert werden. Dann müssen die Befehle nicht immer per Hand eingegeben werden.

Aufgabe :

- 1) Erstellen Sie im Verzeichnis `U:\matlab` das Skript `drehung.m`.
- 2) Schreiben Sie das Skript und führen Sie es aus.



```
Editor - U:\Matlab\prakt1\drehung.m
drehung.m x +
1 clear
2 % Berechne Koordinaten des Punktes P=(1,0),
3 % wenn er um den Winkel w gedreht wird.
4 format compact
5 w = 30
6 p = [1; 0]
7 DM = [cosd(w), -sind(w); sind(w), cosd(w)]
8 q = DM*p
9 q = DM^3*p
10 E1 = DM^12
```

- 3) Die Variablen, die im Skript verwendet werden, sind auch im Workspace verfügbar. Deshalb können Sie die Matrix DM und die Vektoren p und q einfach im Command-Fenster weiter verwenden.

P1_23 1.7 Matrizen – Drehung in der Ebene

- 4) Führen Sie die Drehung dreimal durch. Hierzu können Sie auch den Operator \wedge verwenden.
- 5) Wie oft muss man die Matrix DM mit sich selbst multiplizieren, um die Einheitsmatrix zu erhalten?
- 6) Ändern Sie den Drehwinkel auf 60° und 90° . Führen Sie dann ein paar typische Drehungen durch.
- 7) Ändern Sie das Skript aus 2 wie folgt :

```
Editor - U:\Matlab\prakt1\drehung1.m
drehung1.m x +
1 - clear
2 - % Berechne Koordinaten des Punktes p=(1,0),
3 - % der um den Winkel w gedreht wird.
4 - format compact
5 - w = 30
6 - p = [1; 0]
7 - DM1 = drehmatrix(w)
8 - q = DM1*p
9 - q1 = drehmatrix(60)*drehmatrix(30)*p
10 - q2 = drehmatrix(-60)*drehmatrix(60)*p
```

Schreiben Sie eine Funktion **drehmatrix**, an die ein Winkel w übergeben wird und die dann die zugehörige Drehmatrix zurückgibt.

```
>> drehung.m
w =
    30
p =
     1
     0
DM =
    0.8660    -0.5000
    0.5000     0.8660
q =
    0.8660
    0.5000
q =
    0.0000
    1.0000
ans =
    1.0000     0.0000
   -0.0000     1.0000
>> q1 = DM*DM*p
q1 =
    0.5000
    0.8660
>> q2 = DM * q1
q2 =
    0.0000
    1.0000
>> DM^6
ans =
   -1.0000    -0.0000
```

help	Help : kurzen Hilfetext im Command-Window anzeigen
help <i>sin</i>	Informationen zum Befehl <i>sin</i> anzeigen
help <i>pi</i>	Informationen zur Variablen <i>pi</i> anzeigen
help <i>matlab\elfun</i>	
help <i>datatypes</i>	
help <i>help</i>	
help <i>doc</i>	
help <i>Operator</i>	
doc	ausführliche Informationen im Help-Browser anzeigen
doc <i>sin</i>	
doc <i>pi</i>	
helpbrowser	Help-Browser öffnen
lookfor	Suche in allen M-Files nach einem vorgegebenen Stichwort
lookfor <i>Bessel</i>	
docsearch	Help-Browser Search öffnen
which	Suche nach Funktionen und Dateien
which <i>ode45</i>	Wo ist die Funktion <i>ode45</i> definiert ?

<code>who</code>	Variablen im aktuellen Workspace anzeigen
<code>who a b x</code> <code>who a*</code>	
<code>whos</code>	wie <code>who</code> , aber ausführliche Informationen (size, type)
<code>clear</code>	Variablen und Funktionen aus dem Workspace löschen
<code>clear</code> <code>clear variables</code> <code>clear a b</code> <code>clear all</code>	alle Variablen im Workspace löschen gleiche Wirkung wie <code>clear</code> Variablen <code>a</code> und <code>b</code> aus dem Workspace löschen Alle Variablen, Funktionen, globale Variablen löschen
<code>clc</code>	Command-Window löschen
<code>pwd</code>	Current Folder anzeigen (<code>print working directory</code>)
<code>cd</code>	Current Folder anzeigen, ändern
<code>cd</code> <code>cd u:\matlab</code> <code>cd ..</code>	Gleiche Wirkung wie <code>pwd</code> <code>U:\matlab</code> als Current Folder setzen Elternverzeichnis als Current Folder setzen
<code>ls</code>	Inhalt des Current Folder anzeigen
<code>dir</code>	Inhalt des Current Folder anzeigen

1. Installieren Sie MATLAB auf ihrem Rechner. Wiederholen Sie alle Übungen der ersten Praktikumsstunde noch einmal.
2. Berechnen Sie die Ergebnisse der folgenden Multiplikationen auf einem Blatt Papier und mit Hilfe von MATLAB. Erstellen Sie jeweils ein geeignetes Skript.

a)
$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 6 & -4 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

b)
$$A1 = \begin{pmatrix} 2 & 3 \\ 4 & -2 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

$$A2 = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \cdot \begin{pmatrix} 2 & 3 \\ 4 & -2 \end{pmatrix}$$

Berechnen Sie die Differenz $A1 - A2$ mit Hilfe vom MATLAB. Verwenden Sie geeignete Hilfsvariablen, damit die Lösung möglichst einfach wird. Warum ist die Differenz nicht 0 ?

c)
$$B = \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

3. Berechnen Sie das Produkt der beiden Drehmatrizen A und B und vergleichen Sie das Ergebnis mit der Drehmatrix C. Führen Sie die Rechnungen zuerst auf einem Blatt Papier durch. Versuchen Sie zu beweisen, dass stets gilt $C=A*B$. Was bedeutet das anschaulich?

$$A = \begin{pmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{pmatrix}$$

$$B = \begin{pmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{pmatrix}$$

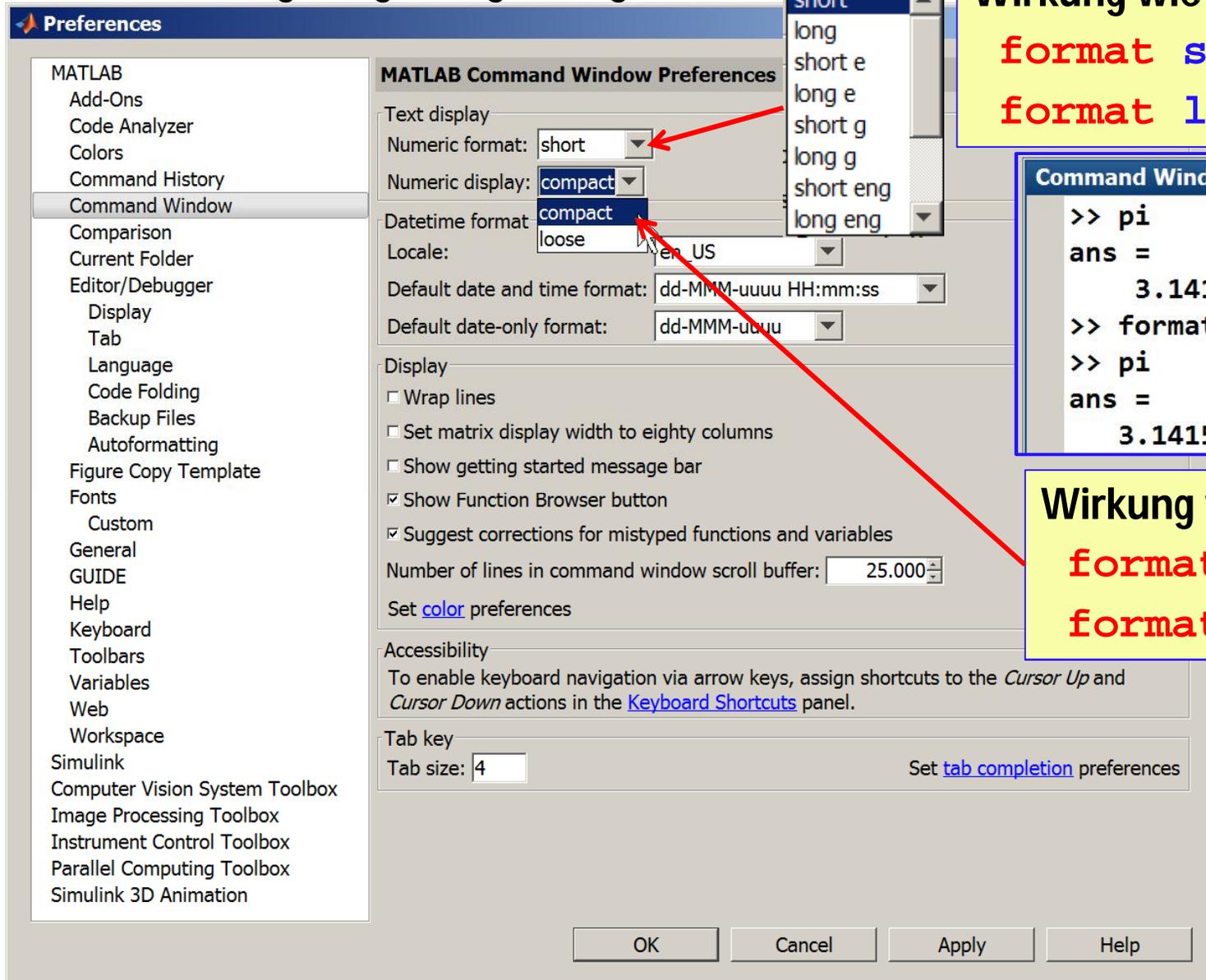
$$C = \begin{pmatrix} \cos(\varphi + \psi) & -\sin(\varphi + \psi) \\ \sin(\varphi + \psi) & \cos(\varphi + \psi) \end{pmatrix}$$

Führen Sie dann die Rechnungen mit MATLAB durch, wobei folgende Werte verwendet werden sollen : $\varphi=30^\circ$ und $\psi=60^\circ$. Verwenden Sie hierzu wieder die Hilfsfunktion **drehmatrix** aus Kapitel 1.7.

- 1 Einstellungen**
- 2 MATLAB-Beispiele kopieren**
- 3 Ein- und Ausgabe**
- 4 Funktionen graphisch darstellen**

1 Einstellungen

Über den Tab **Home (Environment)**->**Preferences** werden Aussehen und Verhalten der Entwicklungsumgebung konfiguriert.



Wirkung wie die Befehle
format short
format long

```

>> pi
ans =
    3.1416
>> format long
>> pi
ans =
    3.141592653589793
    
```

Wirkung wie die Befehle
format compact
format loose

1 Einstellungen

The image shows the MATLAB Preferences dialog box with the 'Display' category selected. A yellow callout box points to the 'MATLAB Colors Preferences' dialog, which is overlaid on the main window. Red arrows point to the 'Colors' and 'Display' categories in the main dialog, and a blue arrow points from the callout box to the 'MATLAB Colors Preferences' dialog.

Fonts
Schrifttyp,- grÖÙe einstellen

Colors
Syntaxhighlighting

MATLAB Editor/Debugger Display Preferences

General display options

- Highlight current line
- Show line numbers
- Enable datatips in edit mode

Right-hand text limit

- Show line

Placement: 75 columns

Width: []

Color: []

MATLAB Colors Preferences

Desktop tool colors

- Use system colors

Text [] Background []

MATLAB syntax highlighting colors

- Keywords [Blue] Comments [Green]
- Strings [Purple] Unterminated strings [Red]
- System commands [Olive] Syntax errors [Red]

MATLAB Command Window colors

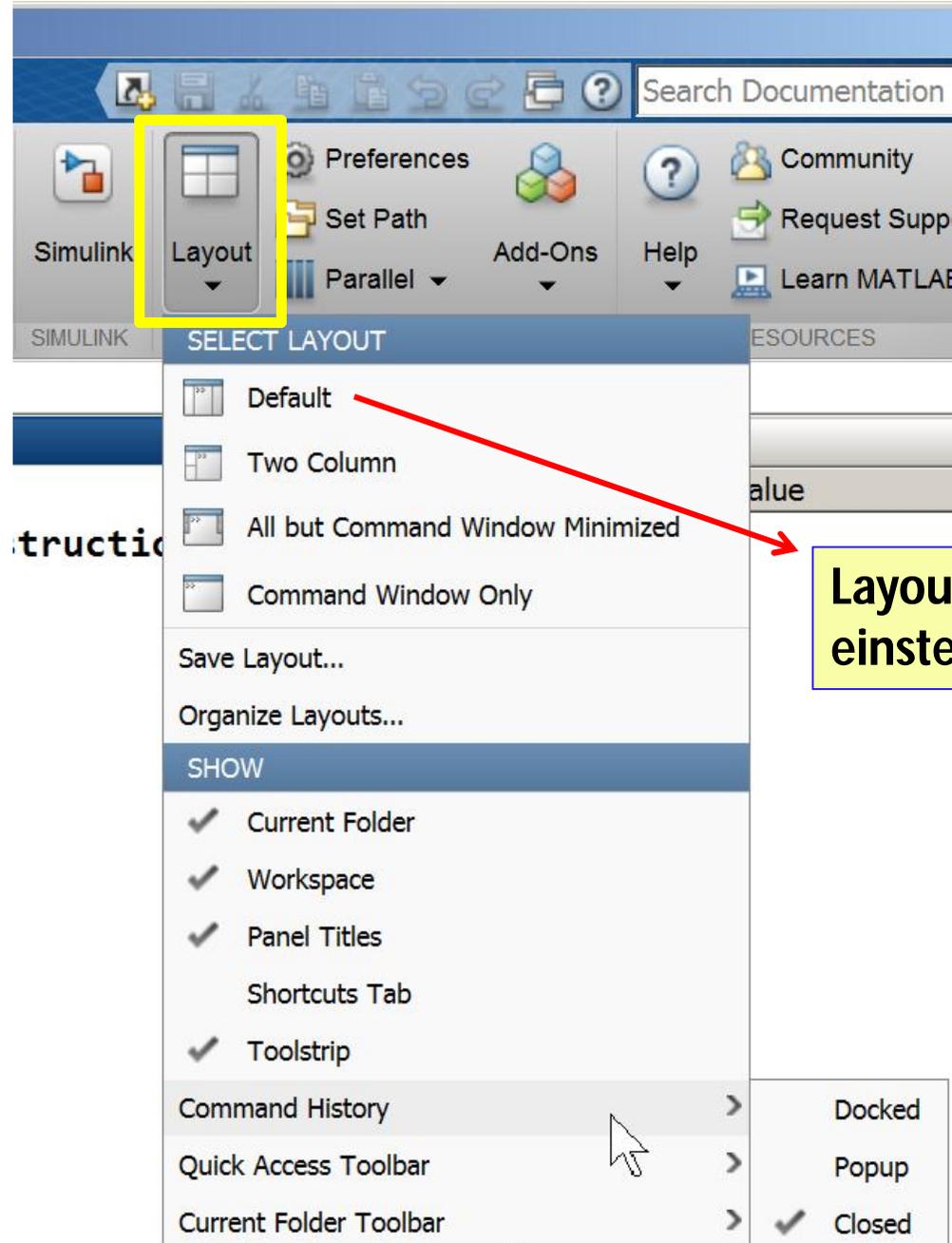
- Error text [Red] Warning text [Orange]
- Hyperlinks [Blue]

Syntax highlighting sample

```
% create a file for output
!touch testFile.txt
fid = fopen('testFile.txt', 'w')
for i=1:10
    fprintf(fid,'%6.2f \n, i);
```

Command Window sample

```
>> samplefunction
Link to sample: link
Warning: Min value set to 0
> In samplefunction at 4
Error using samplefunction
```



Tab Home -> Layout :

- Auswahl der Teilfenster, die in der MATLAB-IDE angezeigt werden
- Anordnung der Teilfenster wählen
- Toolstrip konfigurieren

Layout auf die Standard-einstellung zurücksetzen

Kopieren Sie von I:\Dozenten\Ingenieurinformatik\MatLabKurs\Beispiele die Verzeichnisse **kap1-9**, **prakt1-6** in das Verzeichnis U:\matlab.

U:\Matlab

Computer U-LW (U:) Matlab

Datei Bearbeiten Ansicht Extras ?

Organisieren Freigeben für Brennen Neuer Ordner

Name	Änderungsdatum	Typ
einleitung	10.07.2013 23:26	Dateiordner
kap1	16.08.2013 08:57	Dateiordner
kap2	16.08.2013 08:57	Dateiordner
kap3	16.08.2013 08:57	Dateiordner
kap4	16.08.2013 08:57	Dateiordner
kap6	16.08.2013 08:57	Dateiordner
kap7	16.08.2013 08:57	Dateiordner
kap8dgl	16.08.2013 08:57	Dateiordner
kap9simulink	16.08.2013 08:57	Dateiordner
prakt1	16.08.2013 08:57	Dateiordner
prakt2	16.08.2013 08:57	Dateiordner
prakt3	16.08.2013 08:57	Dateiordner
prakt4	16.08.2013 08:57	Dateiordner
prakt5	16.08.2013 08:57	Dateiordner
prakt6	16.08.2013 08:57	Dateiordner

Erstellen Sie eine Verknüpfung mit **MatlabSkript.pdf**

Beispiele zur Vorlesung

Beispiele für das Praktikum

- 1 Einstellungen
- 2 MATLAB-Beispiele kopieren
- 3 Ein- und Ausgabe
 - 3.1 Ausgabe mit `disp` und `format`
 - 3.2 Ausgabe mit dem Befehl `fprintf`
 - 3.3 Einlesen mit dem Befehl `input`
 - 3.4 Aufgabe
- 4 Funktionen graphisch darstellen

In der Programmiersprache C erfolgt die Ein- und Ausgabe mit Hilfe der Funktionen `scanf` und `printf`.

```
scanf( "%d" , &k ) ;           printf( "X-Wert = %10.4f" , x ) ;
```

Bei MATLAB gibt es ähnliche Funktionen um Werte einzulesen und um Informationen (Text, Werte von Variablen) auszugeben. Standardmäßig werden Werte über das Command-Window eingelesen und Ausgaben in das Command-Window geschrieben. Die wichtigsten Befehle zur Ein- und Ausgabe :

disp Text oder Variablen ausgeben – Formatierung begrenzt möglich

format globale Ausgabeformate festlegen

fprintf Text und Variablen in das Command-Window ausgeben oder in eine Datei schreiben – Formatierung erfolgt über Formatelemente analog zur Programmiersprache C

fscanf Daten aus dem Command-Window oder einer Datei einlesen – Formatierung erfolgt wieder über Formatelemente

input Werte aus dem Command-Window einlesen

Beachte :

Weil bei MATLAB alle Variablen Matrizen sind, gibt es einige wichtige Unterschiede zwischen den Funktionen `printf/scanf` in der Programmiersprache C und den Funktionen `fscanf/fprintf` bei MATLAB.

3.1 Ausgabe mit disp und format

Mit dem Befehl **disp** (display) wird der Wert einer Variablen (allgemeiner eines Ausdrucks) in das Command-Window geschrieben. Der Name der Variablen wird dabei nicht ausgegeben (Unterschied zur Eingabe des Variablennamens).

```

Command Window
>> kreis
Radius:
    5.2000
Umfang:
    32.6726
>> format long
>> kreis
Radius:
    5.2000000000000000
Umfang:
    32.672563597333848
>> format shorte
>> kreis
Radius:
    5.2000e+00
Umfang:
    3.2673e+01
  
```

Den Befehl **disp** verwendet man meist in Skripten oder Funktionen, nur selten im Command-Window. Mit dem Befehl **format** wird die Darstellung der Ausgabe grob gesteuert.

```

Editor - U:\Matlab\kreis.m
kreis.m x +
1 - r=5.2;
2 - disp('Radius:')
3 - disp(r)
4 - u=2*pi*r;
5 - disp('Umfang:')
6 - disp(u)
  
```

String **'Umfang:'** ausgeben
Einfache Hochkommas!

```

format compact
format loose
format short
format long
format shorte
format longe
format shortg
format longg
format hex
help format
  
```

Beachte : **format** ändert nur das Ausgabeformat.
Intern wird der Wert stets gleich gespeichert.

Die formatierte Ausgabe von Variablen erfolgt mit der Funktion `fprintf`. Diese Funktion arbeitet ähnlich wie die Funktion `printf` in der Programmiersprache C.

`fprintf(Formatstring , Variablen)`

Datentyp	Formatelement	Bemerkung
Integer, signed	<code>%d</code> or <code>%i</code>	Base 10
Integer, unsigned	<code>%u</code> <code>%x</code> <code>%X</code>	Base 10 Base 16 (hexadecimal), lowercase letters a-f Same as <code>%x</code> , uppercase letters A-F
Floating-point number	<code>%f</code> <code>%e</code> <code>%E</code>	Fixed-point notation Exponential notation, such as 3.141593e+00 Same as <code>%e</code> , but uppercase, such as 3.141593E+00
Characters	<code>%c</code> <code>%s</code>	Single character String of characters, z.B. <code>'Hallo'</code>

```
> x=13.67;
> fprintf('x=%f\n', x)
x=13.670000
> fprintf('x=%d\n', x)
x=1.367000e+01
> fprintf('x=%e\n', x)
x=1.367000e+01
```

```
> x=101;
> fprintf('x=%f\n', x)
x=101.000000
> fprintf('x=%d\n', x)
x=101
> fprintf('x=%e\n', x)
x=1.010000e+02
```

`%d` oder `%i` ist auch bei Gleitkommazahlen möglich!

Feldweite und Genauigkeit :

Bei Formatelementen für Gleitkommavariablen kann man die Feldweite (d.h. die Anzahl der Spalten, die für die Ausgabe reserviert werden) festlegen und die Anzahl der Nachkommastellen.

Beispiele :

```
fprintf( 'Wert=%10.3f' ,x)
```

```
Wert=___789.123
```

```
fprintf( 'Wert=%8d' ,k)
```

```
Wert=_ _123123
```

```
>> x=13.67;
>> fprintf( 'x=%.0f\n' , x)
x=14
>> fprintf( 'x=%.1f\n' , x)
x=13.7
>> fprintf( 'x=%5.1f\n' , x)
x= 13.7
>> fprintf( 'x=%6.1f\n' , x)
x=  13.7
>> fprintf( 'x=%6.2f\n' , x)
x= 13.67
```

Aufgabe 1 : Vektoren und Matrizen als Variablen in `fprintf`

Untersuchen Sie das Verhalten von `fprintf`, wenn ein Vektor oder eine Matrix als Variable verwendet wird. Wie lauten die Regeln für die Ausgabe?

Führen Sie die folgenden Befehle aus :

```
>x = [ 3 5 7 9 ];
>fprintf('x=%f\n',x)
>fprintf('%f %f\n',x)
>fprintf('%f %f %f %f\n',x)
>fprintf('%f %f\n%f %f\n',x)
>x = [3; 5; 7; 9];
>fprintf('x=%f %f\n',x)
>fprintf('%f %f\n',x(2),x(4))
>fprintf('%f %f',x(2),x(4))
```

①

```
>A = [ 1 2 ; 3 4 ]
>fprintf('%f\n',A)
>fprintf('%f %f\n',A)
>fprintf('%f %f\n',A')
>fprintf('%f %f\n',A(1,:))
>fprintf('%f %f\n',A(:,1))
>fprintf('%f\n%f\n',A(:,1))
>fprintf('%f\n',A(:,1))
>fprintf('%f %f\n',A(:))
```

②

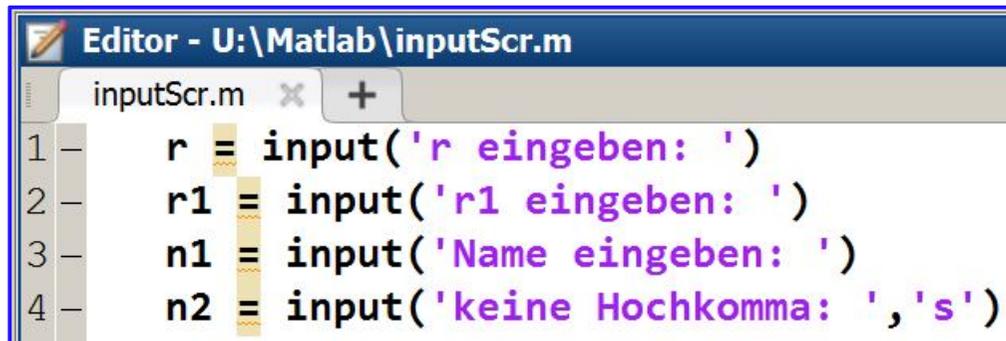
```
>fprintf('%f\n%f\n',A(1,:))
>fprintf('%f %f\n',A(1,1),A(2,2))
>fprintf('%f',A)
```

③

Mit der Funktion `input` lässt sich sehr einfach eine Eingabeaufforderung programmieren. `input` gibt einen Text aus und wartet dann auf die Eingabe. Diese wird als Rückgabewert der Funktion `input` der linken Seite zugewiesen. Die Funktion `input` kann Zahlenwerte und Text (Strings) einlesen.

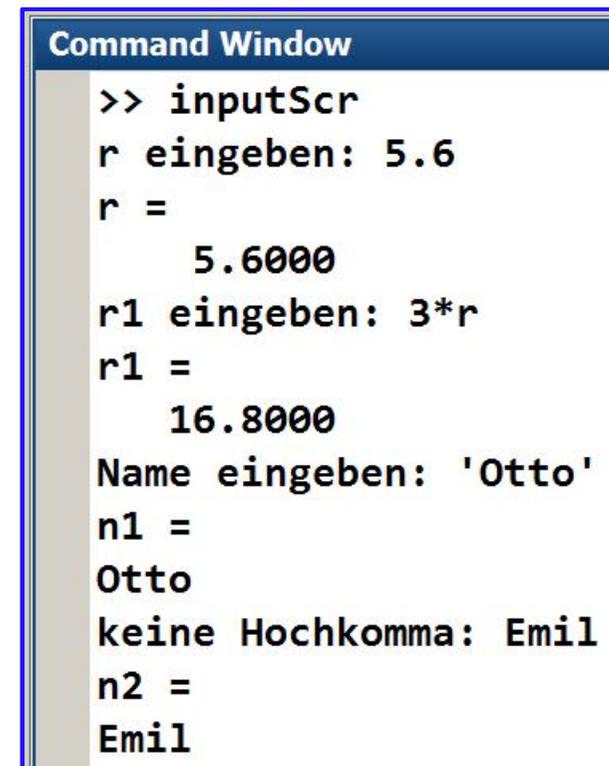
```
variable = input('Text für Eingabeaufforderung')
```

Beispiel :



```
Editor - U:\Matlab\inputScr.m
inputScr.m x +
1 - r = input('r eingeben: ')
2 - r1 = input('r1 eingeben: ')
3 - n1 = input('Name eingeben: ')
4 - n2 = input('keine Hochkomma: ','s')
```

Der zweite Parameter beim Aufruf von `input` ('s') bewirkt, dass die Funktion `input` die Eingabe als String interpretiert. Wird dieser Parameter weggelassen, dann muss alternativ die Eingabe des Strings mit einfachen Hochkommas erfolgen.



```
Command Window
>> inputScr
r eingeben: 5.6
r =
    5.6000
r1 eingeben: 3*r
r1 =
   16.8000
Name eingeben: 'Otto'
n1 =
    Otto
keine Hochkomma: Emil
n2 =
    Emil
```

Aufgabe 2 :

Das Skript `drehung` im Verzeichnis `prakt2` berechnet die neuen Koordinaten eines bestimmten Punktes, der um 30° gedreht wird. Öffnen Sie das Skript und ändern Sie es so ab, dass die X-Y-Koordinaten des Punktes eingelesen werden, ebenso der Drehwinkel. Die Koordinaten des Ausgangspunktes, die Drehmatrix und die Koordinaten des gedrehten Punktes werden mit 3 Nachkommastellen ausgegeben (siehe Beispiel).

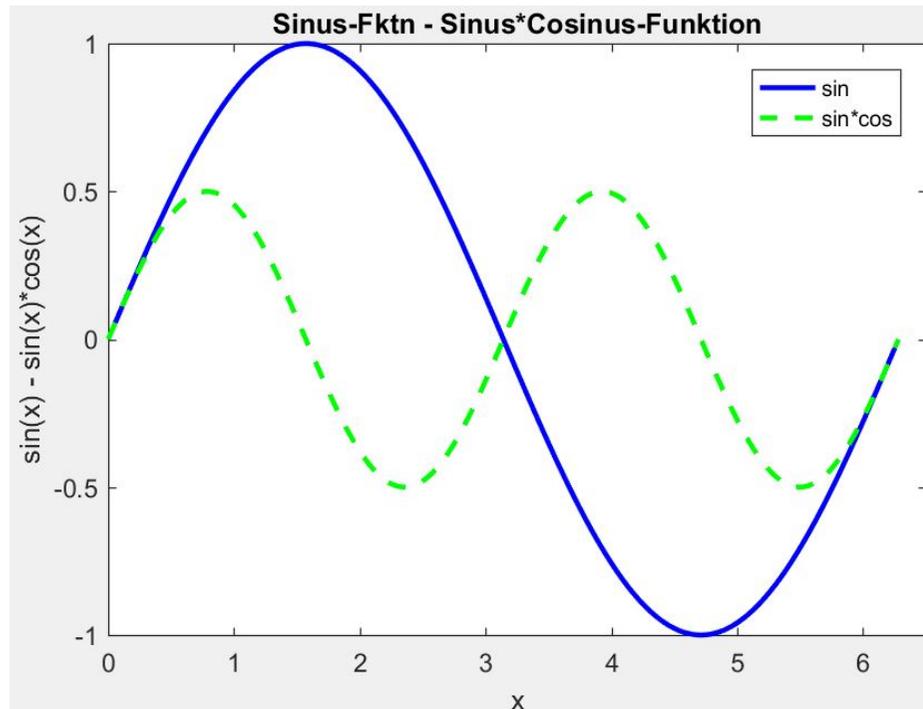
```
Command Window
>> drehung_lsg
Drehung in der Ebene
Drehwinkel in Grad: 35
X-Koordinate eingeben: 1
Y-Koordinate eingeben: 1.5
Drehwinkel w=35.000
Drehmatrix :
    0.819  -0.574
    0.574   0.819
Ausgangspunkt   P(1.000,1.500)
Gedrehter Punkt Q(-0.041,1.802)
Abstand Ausgangspunkt   : 1.802776
Abstand gedrehter Punkt : 1.802776
```



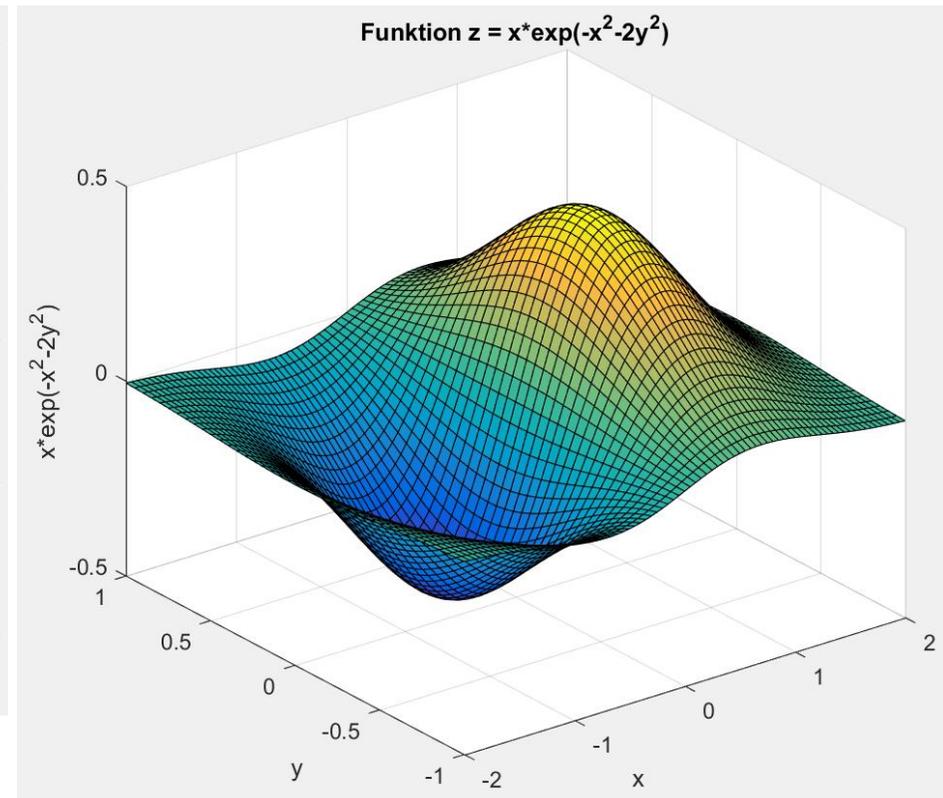
Berechnen Sie den Abstand des Ausgangspunktes vom Ursprung und den Abstand des gedrehten Punktes vom Ursprung. Geben Sie den Abstand jeweils auf 6 Nachkommastellen aus.

Wie kann man aus den Koordinaten des gedrehten Punktes und des Ausgangspunktes den **Drehwinkel berechnen?**

Mit MATLAB lassen sich einfach 2- und 3-dimensionale Graphiken erstellen



2D-Graphik



3D-Graphik

Teilaufgaben

- **Mathematischer Anteil** – Funktion, Daten
- **Darstellung** – Linientyp, Farbe, Beschriftung der Achsen, Überschrift, Strichstärke, Legende
- **Ausgabe der Graphik** – Graphikformate (.jpg, .tiff, .bmp), Speichern und Laden von Graphiken mit Hilfe von MATLAB-Figure-Dateien

Die Funktion `plot(x,y)` zeichnet die Werte von y über x , d.h. die Punkte $(x(k), y(k))$ für alle Werte von k . Die beiden Vektoren x und y müssen die gleiche Länge besitzen. Wird der Vektor x weggelassen (d.h. `plot(y)`) werden die Werte von y über dem Index gezeichnet, d.h. die Punkte $(k, y(k))$. Die Punkte werden standardmäßig mit geraden Linien verbunden.

Der `plot`-Befehl stellt die Zeichnung in einem eigenen Fenster mit dem Titel „Figure 1“ dar (Graphik-Fenster). Die Zeichnung kann anschließend mit weiteren Befehlen bearbeitet werden, z.B. Beschriftung der Achsen. Durch die erneute Eingabe eines `plot`-Befehls wird die Graphik im aktuellen Graphikfenster gelöscht und es wird eine neue Graphik gezeichnet.

Aufgabe 3 :

Erstelle ein Skript `plotbefehl.m`, das die Kurve $y = \sin(x)$ im Intervall $[0, 2\pi]$ zeichnet

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

Beachte : der Vektor x ist ein Zeilenvektor!

Ersetzen Sie den Befehl `plot(x,y)` durch `plot(y)`. Was ist der Unterschied zwischen den beiden Befehlen? Zeichnen Sie danach die Funktion $y = x^2 \cdot \sin(x)$. Wie lautet der Befehl um die y -Werte zu berechnen?

P2_16 4.1 plot-Befehl

Das Skript `plotbefehl.m` wird so abgeändert, dass zwei Kurven in einer Zeichnung dargestellt werden. Hierzu gibt es mehrere Möglichkeiten :

- Die Funktion `plot(x1,y1,x2,y2)` zeichnet die Werte von `y1` über `x1` und die Werte von `y2` über `x2`.
- Erzeuge eine **Matrix `Y`** mit zwei Zeilen. `plot(x,Y)` zeichnet zuerst Zeile 1 der Matrix `Y` über `x` und dann Zeile 2 von `Y` über `x`. Der Vektor `x` und die Matrix `Y` müssen gleich viele Spalten besitzen.
- Zeichne mit Hilfe des `plot`-Befehls die erste Kurve. Danach wird der Befehl **`hold on`** eingegeben, der bewirkt, dass die nachfolgenden `plot`-Befehle die bereits **bestehende Zeichnung** verwenden und keine Neuzeichnung erfolgt. Dadurch werden mehrere Kurven in ein Graphik-Fenster gezeichnet. Mit dem Befehl **`hold off`** wird das Standardverhalten wiederhergestellt, d.h. `plot` initialisiert die Graphik jeweils wieder neu.

```
...  
plot(x1,y1)  
hold on  
...  
plot(x2,y2)
```

Aufgabe 4 :

Zeichnen Sie die Kurven `sin(x)` und `sin(x)*cos(x)` in ein Graphik-Fenster. Ändern Sie das Skript `plotbefehl` entsprechend ab. Probieren Sie alle drei Varianten aus.

Die Darstellung einer Kurve kann durch **optionale Parameter** des `plot`-Befehls geändert werden. Wichtigster Parameter ist die **Line-Specification**, ein String der die Darstellungsform einer Kurve mit Hilfe von 3 Werten (**color**, **marker** und **line style**) spezifiziert (siehe `help plot` und `doc plot`).

Color		Marker		Line Style	
b	blue	.	point	-	solid
g	green	o	circle	:	dotted
r	red	x	x-mark	-.	dashdot
c	cyan	+	plus	--	dashed
m	magenta	*	star	(none)	no line
y	yellow	s	square		
k	black	d	diamond		
w	white	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

kein Marker

```
plot(x,y, 'g*--')
```

```
plot(x,y1, 'g*--', x,y2, 'r+')
```

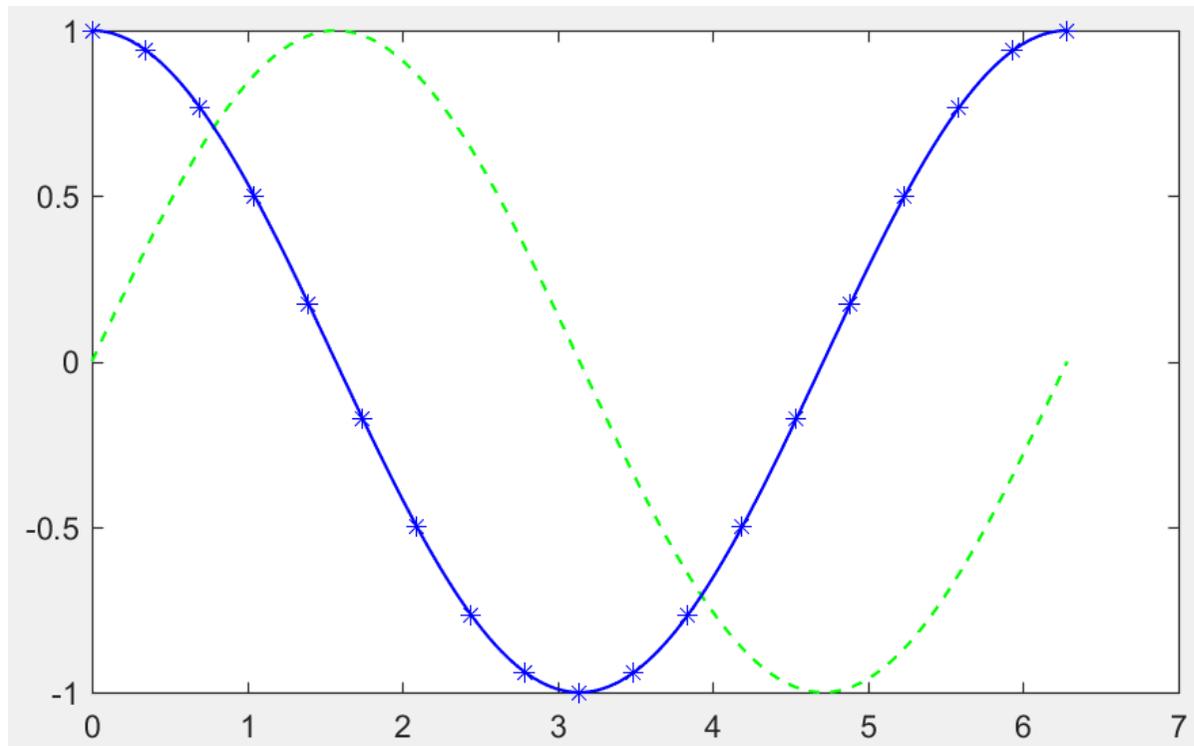
```
plot(x,y, 'r+')
```

```
plot(x,y, 'b-.')
```

Aufgabe 5 :

Zeichnen Sie die Kurven $\sin(x)$ und $\cos(x)$ in eine Graphik. Die **Sinus-Kurve** soll **grün** und **gestrichelt** gezeichnet werden. Die **Cosinus-Funktion** wird **blau** dargestellt mit ' **Line Style solid** '. Einzelne Punkte auf der blauen Kurve werden durch einen ***** (**star**) gekennzeichnet. Der Abstand dieser Punkte auf der x-Achse soll jeweils 20° betragen.

Schreiben Sie die entsprechenden Befehle in das Skript `plotbefehl.m`.



P2_19 4.3 Konfiguration – PropertyName - PropertyValue

Die Darstellung von Graphiken kann auch über **PropertyName-PropertyValue-Paare** modifiziert werden. **PropertyName** ist ein String, der eine bestimmte Eigenschaft einer Graphik bezeichnet. Für die Strichstärke einer Kurve lautet der PropertyName **'LineWidth'**. Durch die Angabe eines **PropertyValues** wird für die entsprechende Property ein bestimmter Wert gesetzt, der dann bei der Darstellung verwendet wird. Der Datentyp des PropertyValues hängt von der Property ab. Für die Property **'LineWidth'** ist der Datentyp des PropertyValues `double`. Die PropertyName-PropertyValue-Paare folgen im `plot`-Befehl auf die Daten, wenn keine Line-Specification verwendet wird oder Sie werden nach der Line-Specification angegeben. Es dürfen mehrere Property-Name-PropertyValue-Paare verwendet werden.

Beispiele :

```
plot(x,y, 'LineWidth', 2)
```

Property `LineWidth` (Strichstärke) bekommt den Wert 2

```
plot(x,y, 'c-', 'LineWidth', 2)
```

Setze eine Line-Specification und ein PropertyName/Value-Element

```
plot(x,y, 'Color', [0.9, 0, 0], 'LineWidth', 2)
```

Property `Color` bekommt den RGB-Wert `[0.9, 0, 0]`,

Property `LineWidth` den Wert 2, siehe doc `ColorSpec`

Nach der Eingabe des plot-Befehls muss eine Graphik beschriftet werden.

`title('Titel der Zeichnung')`

`xlabel('x-Achse')`

Beschriftung der x-Achse, analog y und z

`legend('sin', 'cos')`

Welche Kurve ist welche?

`xlim([0 8])`

X-Achse von 0 bis 8, analog y und z

`axis([0 8 -1 2])`

X-Achse von 0 bis 8, Y-Achse von -1 bis 2

`axis equal`

X- und Y-Achse mit gleichem Maßstab

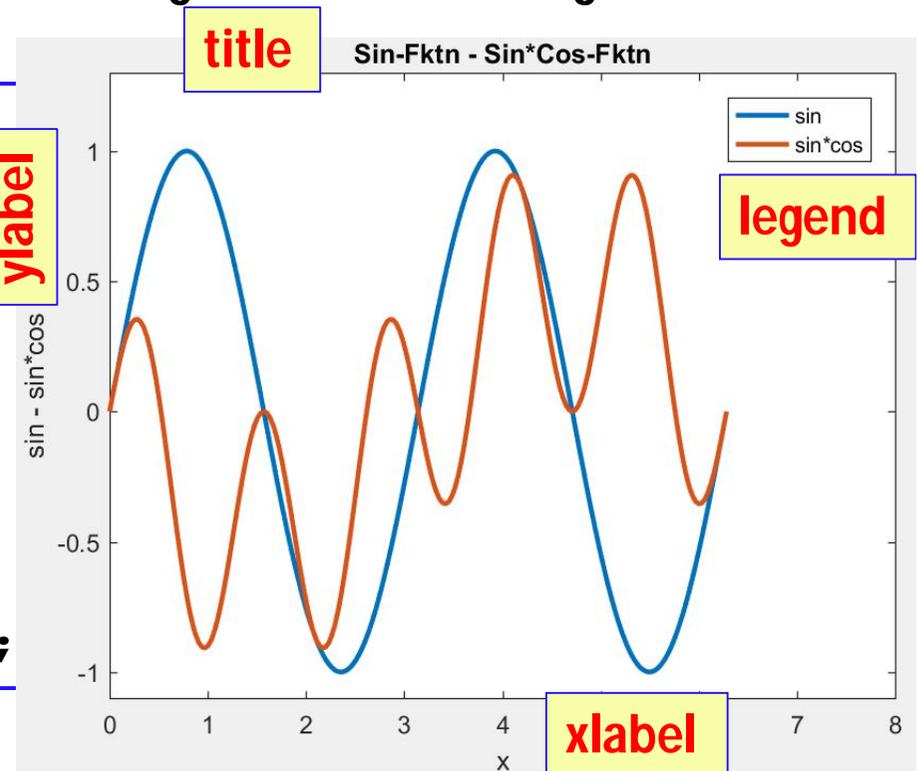
`axis on/off`

Achsen werden gezeichnet / nicht gezeichnet

`grid on/off` Gitterlinien an/aus

```
x = 0:pi/100:2*pi;
w1 = 2.0; w2=3.0;
y1 = sin(w1*x);
y2 = cos(w2*x).*sin(w1*x);
plot(x,y1,x,y2,'LineWidth',2)
xlabel('x')
ylabel('sin(x) - sin(x)*cos(x)')
axis([0 8 -1.1 1.3])
legend('sin', 'sin*cos')
title('Sin-Fktn - Sin*Cos-Fktn');
```

`plotsincos.m`



Mit dem Befehl

```
subplot( m, n, p )
```

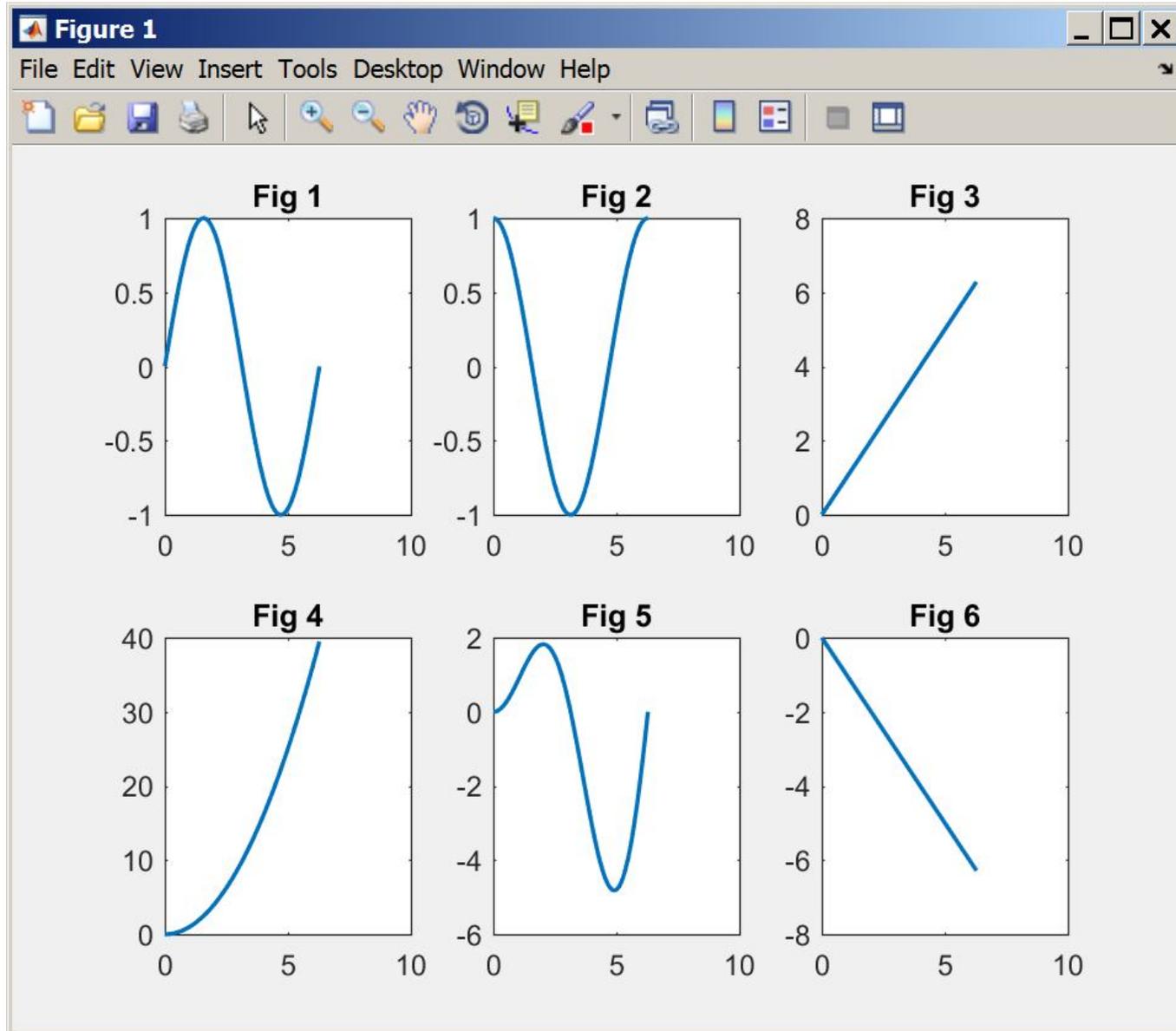
wird ein Fenster in $m \cdot n$ kleinere Teilfenster zerlegt. Damit lassen sich mehrere Plots (Subplots) neben- und untereinander in einem Graphik-Fenster darstellen. Die Subplots werden **zeilenweise durchnummeriert**, beginnend bei 1. Der dritte Parameter des Befehls `subplot` (hier `p`) gibt an, auf welchen Subplot die nachfolgenden Befehle angewendet werden, d.h. `p` wählt den „aktuellen“ Subplot aus. Häufige Anwendung : 3D-Plots mit verschiedenen Ansichten.

Beispiel :

```
x = 0:pi/100:2*pi;
subplot(2,3,1); plot(x, sin(x)); title('Fig 1');
subplot(2,3,2); plot(x, cos(x)); title('Fig 2');
subplot(2,3,3); plot(x, x); title('Fig 3');
subplot(2,3,4); plot(x, x.*x); title('Fig 4');
subplot(2,3,6); plot(x, -x); title('Fig 6');
subplot(2,3,5); plot(x, x.*sin(x)); title('Fig 5');
```

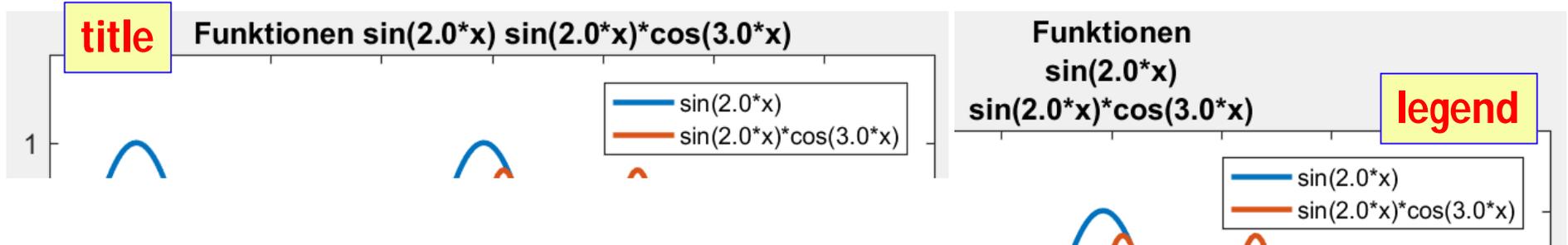
`plotsubplot.m`

Aufgabe 6 : Versuchen Sie das Programm `plotsubplot.m` zu verstehen.



Hausaufgabe :

Ändern Sie das Skript `plotsincos` so, dass in der Legende und im Titel automatisch die aktuellen Werte von `w1` und `w2` angezeigt werden, z.B.



Dieses Problem wird mit dem Befehl `sprintf` gelöst. Der Befehl arbeitet wie `fprintf`. Das Ergebnis wird aber in eine **Stringvariable geschrieben**. Beispiel :

```
f1 = sprintf('sin(%.1f*x)', w1)
```

Besitzt `w1` den Wert `2.3`, dann wird folgender String erzeugt : `'sin(2.3*x)'`

Der String `f1` kann z.B. im Befehl `legend` als Parameter verwendet werden :

```
legend(f1, f2) statt legend('sin', 'sin*cos')
```

Das Formatelement `%s` wird verwendet, um eine Stringvariable mit `sprintf` zu verarbeiten. In einfachen Fällen kann auch der Concatenation-Operator `[]` verwendet werden.

```
titlestr = sprintf('Funktionen %s %s', f1, f2)
```

```
titlestr = [ 'Funktionen ', f1, f2 ]
```

P2_24 4 Hausaufgaben

Versuchen Sie zunächst mit einfachen Beispielen den Befehl `sprintf` zu verstehen. Danach können Sie dann die Befehle für `title` und `legend` programmieren.

The screenshot displays the MATLAB Editor and Command Window. The Editor shows a script named `sprintfScr.m` with 18 lines of code. The Command Window shows the execution of these lines, with blue arrows pointing from the code in the Editor to the corresponding output in the Command Window. The Workspace window at the bottom shows the values of variables `f1`, `f2`, and `f3`.

```
Editor - U:\Matlab\sprintfScr.m
sprintfScr.m x +
1 f1 = sprintf('pi= %.4f',pi)
2
3 h1 = 'Hal'
4 h2 = 'lo'
5 h3 = sprintf('%s%s',h1,h2)
6
7 f2 = sprintf('pi= %.8f',pi)
8 f3 = [ f1 , ' und ', f2]
9 f4 = sprintf('%s und %s',f1,f2)
10
11 x=35;
12 f5 = sprintf('sind(%5.2f)=%.4f',x,sind(x))
13 w=2.7;
14 f6= sprintf('sin(%.1f*x)',w)
15 f7= sprintf('%s(%.1f*x)', 'sin',w)
16
17 n1 = [ f1 , '\n', f2]
18 n2 = sprintf('%s\n %s',f1,f2)

Command Window
>> sprintfScr
f1 =
pi= 3.1416
h1 =
Hal
h2 =
lo
h3 =
Hallo
f2 =
pi= 3.14159265
f3 =
pi= 3.1416 und pi= 3.14159265
f4 =
pi= 3.1416 und pi= 3.14159265
f5 =
sind(35.00)=0.5736
f6 =
sin(2.7*x)
f7 =
sin(2.7*x)
n1 =
pi= 3.1416\npi= 3.14159265
n2 =
pi= 3.1416
pi= 3.14159265

Workspace
Name Value
f1 'pi= 3.1416'
f2 'pi= 3.14159265'
f3 'pi= 3.1416 und pi= 3.14159265'
```

- 1 Debugger**
- 2 Numerische Integration**
- 3 Kurvendiskussion**

Die MATLAB IDE besitzt einen integrierten Debugger um Funktionen und Skripte während der Laufzeit zu analysieren. Ein Debugger dient dazu **Fehler zu finden**, das korrekte Verhalten von Programmen zu **überprüfen** und fremde **Programme besser zu verstehen**.

Wichtige Elemente eines Debuggers :

- **Breakpoint (Haltepunkt)**: eine speziell markierte Zeile im Programm, an der das Programm angehalten wird, wenn es zur Laufzeit diese Zeile erreicht
- **Einzel-schrittverarbeitung** mit **Step**, **Step in**, **Step out** und **Continue**
- Aktuelle **Werte von Variablen anzeigen** und **ändern**

Schritte bei der Anwendung des Debuggers :

1. Setze einen oder mehrere Breakpoints an wichtigen Stellen einer Funktion oder eines Skripts.
2. Starte dann das Programm. Wenn das Programm auf einen Breakpoint läuft, wird die Programmausführung angehalten. Im Command-Window erscheint der **Debug-Prompt** `K>>`
3. Anschließend gibt man MATLAB-Befehle im Command-Window ein, die z.B. Werte von Variablen anzeigen oder verändern. Häufiger verwendet man hierzu aber das Workspace-Fenster oder den Variable-Editor .

1 Debugger

The screenshot displays the MATLAB environment with the Editor window showing a script named `dbkreis.m`. The script contains a function `dbkreis` with a `for` loop and a `while` loop. Two breakpoints are set: one at line 6 (indicated by a red circle and a right-pointing arrow) and another at line 14 (indicated by a red circle and a left-pointing arrow). A yellow box labeled "Break point" is positioned over these two lines. The Debugger toolbar is visible, with a yellow box highlighting the "Breakpoints" button and the "Continue" button. The Command Window shows the execution of the `dbkreis` function, displaying the output of the `disp` and `fprintf` statements. A yellow box labeled "Debug-Prompt K>>" is positioned over the `K>>` prompt in the Command Window.

```
function [ ] = dbkreis( )
disp(' Zahl Quadrat')
% Schleife von 2 bis 4, Schrittweite 1
for k = 2:1:4
    quadrat = k*k;
    fprintf('%5d%8d\n',k,quadrat);
end

% 3 mal Zufallszahl zw. 1 und 100 ziehen
n=1;
while n <= 3
    k = randi(100,1);
    fprintf('Zufallszahl k=%d\n',k);
    if ( k < 10 )
        disp(' k kleiner als 10');
    elseif (k <50)
        disp(' k >= 10 und k < 50');
    else
        disp(' k groesser gleich 50');
    end
    n = n+1;
end
```

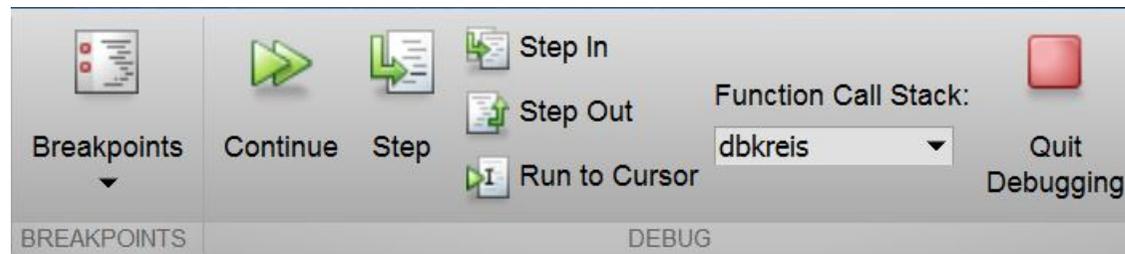
Command Window Output:

```
>> dbkreis
Zahl Quadrat
    2         4
    3         9
K>> k
k =
     3
K>> whos
Name      Size
k         1x1
quadrat   1x1
K>>
```

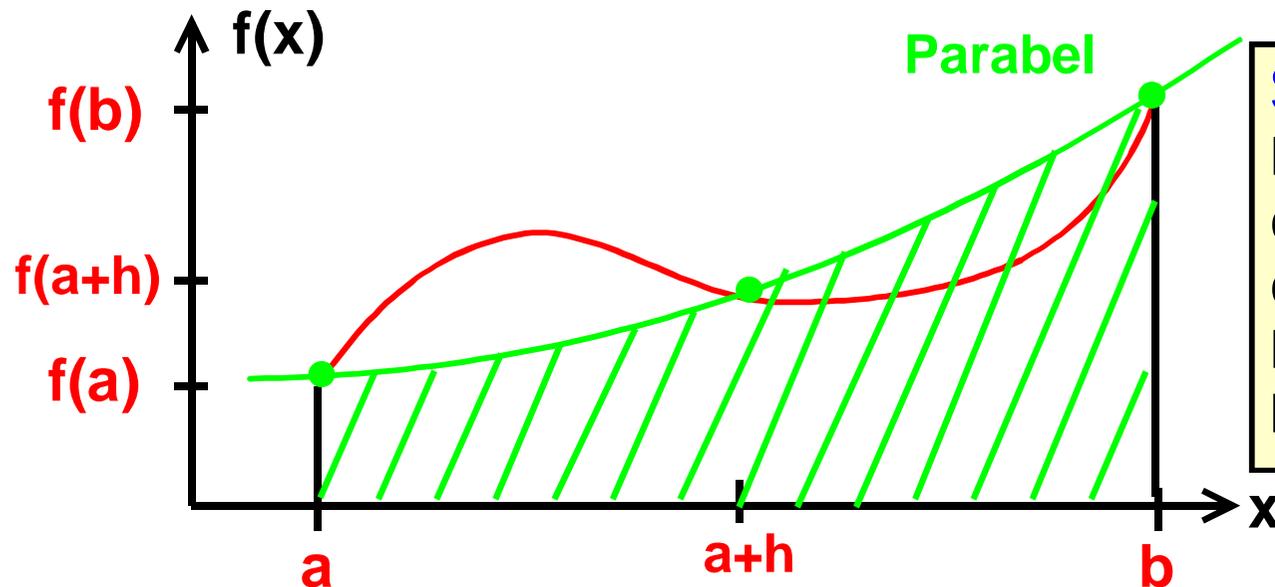
Aufgabe 1

Um den Umgang mit dem Debugger zu üben werden die beiden Funktionen `dbkreis.m` und `dbflaeche.m` im Verzeichnis `\prakt3` verwendet.

1. Starten Sie die Funktion `dbkreis` und versuchen Sie das Programm zu verstehen. Es werden Schleifen (`for`, `while`) und `if-else`-Anweisungen verwendet. Versuchen Sie diese Kontrollstrukturen zu verstehen.
2. Setzen Sie in den Zeilen mit der `for`- und der `while`-Schleife jeweils einen **Breakpoint** und starten Sie das Programm erneut. Welche Variablen sind im Workspace vorhanden? Welche Werte besitzen diese Variablen? Gehen Sie dann Zeile für Zeile (**Step**) durch das Programm.



3. Probieren Sie die Befehle **Step** und **Step in** beim Aufruf der Funktionen `dbumfang` und `dbflaeche` aus. Was ist der Unterschied?
4. Deaktivieren Sie alle Breakpoints und starten Sie das Programm erneut.



Simpsonverfahren

Ersetze den Integranden durch eine Parabel, die durch 3 Punkte auf der Kurve geht. Eine Parabel besitzt 3 freie Parameter.

Fläche unter der Parabel :

$$F = \frac{h}{3} \cdot (f(a) + 4 \cdot f(a+h) + f(b))$$

$$h = (b - a) / 2$$

Teilt man das Intervall $[a, b]$ in n (n gerade) Teilintervalle auf und wendet auf je zwei benachbarte Intervalle die obige Formel an, dann ergibt sich folgende Näherungsformel für das Integral (siehe z.B. Wikipedia) :

$$F = \frac{h}{3} (f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + 2f(a+4h) + \dots + 4f(a+(n-1)h) + f(b))$$

Fehlerabschätzung :

$$-\frac{1}{180} \cdot h^4 \cdot f^{(4)}(x') \cdot (b - a) \quad , \quad a \leq x' \leq b$$

$$h = (b - a) / n$$

```
double simpson(double a, double b, int n)
{
    double x, F, h;
    int i;
    h = (b-a)/n;
    F = exp(-a*a);
    x = a;
    for (i=1; i<n; i++)
    {
        x = x + h;
        if ( (i%2) == 1 )
            F = F + 4.0 * exp(-x*x);
        else
            F = F + 2.0 * exp(-x*x);
    }
    F = F + exp(-b*b);
    F = F*h/3.0;
    return F;
}
```

$$F = \frac{h}{3} (f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + 2f(a+4h) + \dots + 4f(a+(n-1)h) + f(b))$$

Ein C-Programm, das den Wert des Integrals

$$F = \int_a^b e^{-x \cdot x} dx$$

mit Hilfe des Simpsonverfahrens berechnet.

Aufgabe 2

Erstellen Sie eine MATLAB-Funktion `simpson.m`, die das bestimmte Integral

$$F = \int_a^b e^{-x \cdot x} dx$$

zwischen den Grenzen a und b nach dem Simpsonverfahren berechnet. Als Vorlage verwenden Sie das entsprechende C-Programm.

Wie lautet das Ergebnis für das bestimmte Integral

$$\int_{0.5}^2 e^{-x \cdot x} dx$$

wenn Sie für n die Werte 10, 100 und 1000 verwenden? Erstellen Sie dazu ein Skript. Testen Sie Ihr Programm auch mit dem Debugger.

Command Window			
n=	10	y=	0.42078712
n=	100	y=	0.42080038
n=	1000	y=	0.42080038

Aufgabe 3

Das bisherige Integrationsprogramm kann nur den Wert des Integrals für eine ganz spezielle Funktion berechnen.

Ändern Sie die Funktion `simpson.m` so ab, dass man beliebige Funktionen integrieren kann. An die Funktion `simpson.m` muss zusätzlich ein Function-Handle übergeben werden.

```
function [y] = simpson( fun, a, b, n )
```

Testen Sie Ihr Programm, indem Sie die Funktionen e^{-x*x} , $\sin(x)$ und $\sin(x)*\cos(x)$ im Intervall $[1, 2]$ integrieren.

Setzen Sie einen Breakpoint in der Funktion, die $y(x) = e^{-x*x}$ berechnet und berechnen Sie dann das Integral.

```
Command Window
exp(-x*x)
n=      10   y=  0.135256035
n=     100   y=  0.135257258
n=    1000   y=  0.135257258
n=   10000   y=  0.135257258
n=  100000   y=  0.135257258
n= 1000000   y=  0.135257258
n=10000000   y=  0.135257258
sin
n=     1000   y=  0.956449142
sin*cos
n=     1000   y=  0.059374196
```

Aufgabe 4

Schreiben Sie ein Skript `gaussScr`, welches die Fläche unter der Glockenkurve im Bereich $[a, b]$ berechnet:

$$F = \int_a^b e^{-k \cdot x \cdot x} dx$$

Beispiele:

$$k = 1.0 \quad a = -1.0 \quad b = +1.0 \rightarrow F = 1.4936$$

$$k = 1.0 \quad a = -0.1 \quad b = +0.1 \rightarrow F = 0.1993$$

$$k = 2.0 \quad a = -0.5 \quad b = +0.5 \rightarrow F = 0.8556$$

$$k = 1.0 \quad a = -\infty \quad b = +\infty \rightarrow F = 1.7725$$

1. Der Anwender wird zunächst nach den Werten a , b , und k gefragt.
2. Dann erfolgt die numerische Berechnung des Integrals mittels `integral`.

integral Numerically evaluate integral.

`Q = integral(FUN,A,B)` approximates the integral of function `FUN` from `A` to `B` using global adaptive quadrature and default error tolerances.

`FUN` must be a function handle. `A` and `B` can be `-Inf` or `Inf`. If both are finite, they can be complex. If at least one is complex, `integral` approximates the path integral from `A` to `B` over a straight line path.

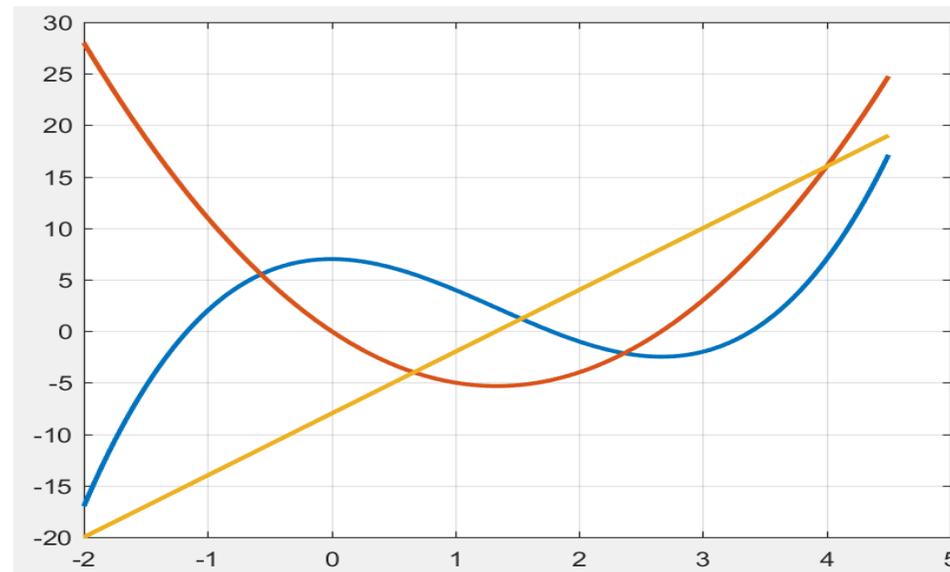
For scalar-valued problems the function `Y = FUN(X)` must accept a vector argument `X` and return a vector result `Y`, the integrand function evaluated at each element of `X`.

3. Zur Berechnung von $y(x) = e^{-k \cdot x \cdot x}$ muss zunächst eine MATLAB-Funktion geschrieben werden, zum Beispiel `function [y] = gauss(x)`, die dann an `integral` übergeben wird: `F = integral(@gauss, a, b)`
4. Tipp: Der Wert des Parameters k kann im Skript `gaussScr` in einer **globalen Variable** gespeichert werden, die dann von der Funktion `gauss` verwendet wird.

Aufgabe 5 :

Schreiben Sie in ein Skript `analysis.m` die Anweisungen zur Lösung der folgenden Aufgaben :

1. Stellen Sie die Funktion $y(x) = x^3 - 4 \cdot x^2 + 7$ sowie deren erste und zweite Ableitung graphisch dar. Tipp: Zunächst einen Vektor mit x-Werten erstellen, anschließend den Vektor mit den y-Werten mittels `polyval` ausrechnen lassen.
2. Geben Sie eine Wertetabelle für das x-Intervall $[0, 3]$ aus, Schrittweite 0.1
3. Bestimmen Sie die x-Koordinaten, an denen die Funktion Nullstellen oder Extrema besitzt. Tipp: Funktion `polyder` und `roots` verwenden.
4. Geben Sie die Koeffizienten der Stammfunktion aus.
5. Berechnen Sie das Integral der Funktion zwischen 1 und 3.
Lösen Sie die Aufgabe auf zwei verschiedene Arten :
 - mit der Funktion `polyint`
 - mit der Funktion `integral`.



Aufgabe 6 :

Schreiben Sie eine Funktion **getPolynomString**, an die ein Polynom übergeben wird (ein Vektor mit den Koeffizienten des Polynoms) und die das Polynom in einer Stringdarstellung zurückgibt. Die Koeffizienten des Polynoms werden jeweils mit einer Nachkommastelle ausgegeben. Anwendung : Beschriftung

Beispiele :

`a=[4.1 3.7 -2.1 0.7]` -> `'y = 4.1*x^3+3.7*x^2-2.1*x+0.7'`

`a=[-4.1 3.7 -2.1 0.7]` -> `'y = -4.1*x^3+3.7*x^2-2.1*x+0.7'`

`a=[-4.1 0.0 0.0 0.7]` -> `'y = -4.1*x^3+0.0*x^2+0.0*x+0.7'`

`a=[-2.1 0.7]` -> `'y = -2.1*x+0.7'` ~~`'y = -2.1*x^1+0.7'`~~

`a=[0.7]` -> `'y = 0.7'` ~~`'y = +0.7'`~~

Hinweise :

Verwenden Sie die Funktion `sprintf` um zwei Strings aneinanderzufügen. Beispiel :

```
neu = sprintf('%s%s', 'abc', 'efg');
```

Hier wird aus den beiden Strings 'abc' und 'efg' der neue String 'abcefg' erzeugt.

Analog kann mit folgendem Ausdruck

```
str = sprintf('%.1f*x^%.0f', 2.3, 4);
```

der String '2.3*x^4' erzeugt werden.

Erzeugen Sie die Stringdarstellung für das Polynom mit einer geeigneten Schleife.

- 1 Lineare Gleichungssysteme**
- 2 Eigenwerte und Eigenvektoren**
- 3 Kette mit zwei Massen**
- 4 Kette mit drei Massen**
- 5 Kette mit zwei unterschiedlichen Massen**

Aufgabe 1

Gegeben ist die Matrix A und der Vektor b . Das lineare Gleichungssystem $A \cdot x = b$ soll gelöst werden.

Ergänzen Sie das Skript `prakt4/allinGln.m` mit den Lösungen der folgenden Aufgaben .

a) Definieren Sie die Matrix A und den Vektor b .

Geben Sie die Matrix A und den Vektor b mit **einer Nachkommastelle** aus.

b) Berechnen Sie die Determinante von A und geben Sie den Wert aus. Welche Aussage lässt sich daraus über die Lösung des Gleichungssystems machen? Besitzt das Gleichungssystem für jeden Vektor b eine eindeutige Lösung? Besitzt die Gleichung $A \cdot x = 0$ neben $x = 0$ noch eine weitere Lösung?

c) Berechnen Sie die Lösung des linearen Gleichungssystems mit Hilfe der Linksdivision und mit der Funktion `mldivide`. Gibt es einen Unterschied? Multiplizieren Sie die Lösung x mit der Matrix A und geben Sie das Ergebnis aus.

d) Berechnen Sie die inverse Matrix von A und geben Sie diese aus.

e) Zeigen Sie, dass MATLAB die inverse Matrix korrekt berechnet hat. Hierzu muss gelten $A^{-1} \cdot A = E$. Geben Sie das Produkt $A^{-1} \cdot A$ aus.

f) Lösen Sie dann das Gleichungssystem mit Hilfe der inversen Matrix. Geben Sie das Ergebnis aus.

$$\begin{array}{c} A \\ \left[\begin{array}{cccc} 4 & 2 & 8 & 2 \\ 7 & 5 & 4 & 1 \\ 8 & 4 & 6 & 1 \\ 2 & 1 & 3 & 1 \end{array} \right] \end{array} * \begin{array}{c} x \\ \left[\begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \right] \end{array} = \begin{array}{c} b \\ \left[\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} \right] \end{array}$$

Aufgabe 2:

Gegeben ist die Matrix A. Besitzt A vier linear unabhängige Eigenvektoren mit reellen Eigenwerten?

Schreiben Sie die Lösungen für folgende Aufgaben in das Skript `prakt4/a2Eigenwert.m`.

$$A = \begin{bmatrix} 4 & 2 & 8 & 2 \\ 2 & 5 & 4 & 1 \\ 8 & 4 & 6 & 1 \\ 2 & 1 & 1 & 1 \end{bmatrix}$$

- Berechnen Sie mit Hilfe von MATLAB die Determinante, die Eigenvektoren und die Eigenwerte. Geben Sie die Ergebnisse aus.
- Berechnen Sie das Produkt der 4 Eigenwerte mit einer for-Schleife. Welche Beziehung gilt ganz allgemein zwischen dem Produkt der Eigenwerte und der Determinante?

- Zeigen Sie, dass der zweite Eigenvektor, der von MATLAB berechnet wird, tatsächlich ein Eigenvektor ist.

D.h. überprüfen Sie, ob folgende Beziehung gilt :

$$A \cdot \vec{v}_2 = \lambda_2 \cdot \vec{v}_2$$

Hierbei ist v_2 der zweite Eigenvektor der Matrix A und λ_2 der zugehörige Eigenwert. Der Vektor v_2 besitzt 4 Komponenten – ein Spaltenvektor mit 4 Zeilen.

- Die Eigenvektoren, die MATLAB berechnet sind **orthogonal** (stehen senkrecht aufeinander) und haben die **Länge 1**. Zeigen Sie, dass das Skalarprodukt von v_1 und v_2 Null ergibt, d.h. die beiden Vektoren sind orthogonal. Überprüfen Sie, dass der Vektor v_1 die Länge 1 besitzt. Wie kann die kartesische Länge eines Vektors berechnet werden? Geben Sie mindestens zwei verschiedene Lösungen an.

Aufgabe 2:

- e) Die Funktion **eig** berechnet die Eigenvektoren einer Matrix **A** und gibt diese in Form einer Matrix zurück – meist **V** genannt. Berechnen Sie die Ausdrücke

$$\mathbf{V}' * \mathbf{V} \quad \mathbf{V}' * \mathbf{A} * \mathbf{V}$$

und geben Sie diese aus. Können Sie die Ergebnisse vorhersagen? Erklären Sie die Ergebnisse (Hinweis Aufgabe d).

- f) Zerlegen Sie den Vektor $\mathbf{b} = [1; 2; 3; 4]$ nach Eigenvektoren von **A**. Welchen Anteil hat der erste, der zweite, der dritte und der vierte Eigenvektor?

D. h. bestimmen Sie die Koeffizienten a_1, a_2, a_3 und a_4 (Werte von Typ double) so, dass gilt :

$$a_1 \cdot \vec{v}_1 + a_2 \cdot \vec{v}_2 + a_3 \cdot \vec{v}_3 + a_4 \cdot \vec{v}_4 = \vec{b}$$

a_1 ist der Anteil des ersten Eigenvektors an \mathbf{b} , a_2 der Anteil des zweiten Eigenvektors, ...

- g) Wie lauten die Koeffizienten a_1, a_2, a_3 und a_4 , wenn \mathbf{b} ein Eigenvektor ist, z.B. $\mathbf{b} = \mathbf{v}_1$? Versuchen Sie das Ergebnis durch Nachdenken zu finden. Berechnen Sie dann das Ergebnis mit Hilfe von MATLAB.

$$\mathbf{A} = \begin{bmatrix} 4 & 2 & 8 & 2 \\ 2 & 5 & 4 & 1 \\ 8 & 4 & 6 & 1 \\ 2 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{pmatrix} v_{11} & v_{21} & v_{31} & v_{41} \\ v_{12} & v_{22} & v_{32} & v_{42} \\ v_{13} & v_{23} & v_{33} & v_{43} \\ v_{14} & v_{24} & v_{34} & v_{44} \end{pmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

4.2 Eigenwerte und Eigenvektoren

```

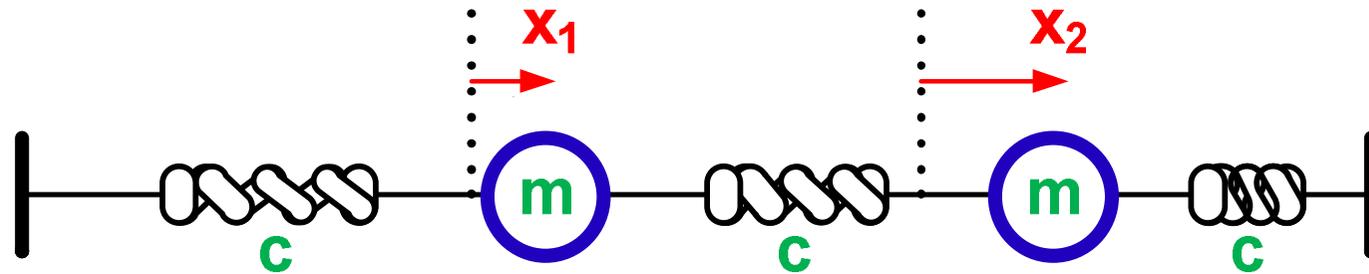
Command Window
Determinante von A:
a  -144
Eigenwerte von A:
    -3.44
     0.83
     3.30
    15.32
b  Eigenvektoren von A:
    0.711  0.050  0.386  0.585
    0.164 -0.007 -0.903  0.396
   -0.650 -0.259  0.186  0.690
   -0.211  0.964  0.024  0.158
c  Produkt der Eigenwerte:
    -144.0000
A*v2 - lambda2*v2
    0.0000000000
    0.0000000000
d  0.0000000000
   -0.0000000000
Skalarprodukt von v1 und v2
    0.0000000000

Länge von v1
    1.0000
    1.0000
    1.0000
Vtransponiert*A*V
e  -3.4416  0.0000  -0.0000  0.0000
    0.0000  0.8288  0.0000  0.0000
   -0.0000  0.0000  3.2961  -0.0000
    0.0000  0.0000  -0.0000  15.3168
Zerlegung des Vektors b=[1;2;3;4]
nach Eigenvektoren von A
Koeffizienten der Eigenvektoren
f  a(1) = -1.753666
    a(2) =  3.116553
    a(3) = -0.768242
    a(4) =  4.076954
Verifikation der Zerlegung von b
    1.000000000000000000
    2.000000000000000000
    2.99999999999999996
    3.99999999999999996
Zerlegung von v1
g  1.0000
    0.0000
   -0.0000

```

$$A \cdot \vec{v}_2 = \lambda_2 \cdot \vec{v}_2$$

Vorlesung : Schwingerkette mit 2 gleichen Massen und gleichen Federkonstanten



Bewegungsgleichungen :

$$m \cdot \ddot{x}_1(t) = -c \cdot x_1(t) - c \cdot [x_1(t) - x_2(t)]$$

$$m \cdot \ddot{x}_2(t) = -c \cdot [x_2(t) - x_1(t)] - c \cdot x_2(t)$$

Anfangsbedingungen :

$$x_1(t=0) = x_{10} \quad \dot{x}_1(t=0) = v_{10}$$

$$x_2(t=0) = x_{20} \quad \dot{x}_2(t=0) = v_{20}$$

Eigenwert-Eigenvektorproblem :

$$\begin{bmatrix} 2 \cdot c/m & -c/m \\ -c/m & 2 \cdot c/m \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \omega^2 \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\vec{x}(t) = a_1 \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} \cdot \cos(\omega_1 \cdot t) + a_2 \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} \cdot \cos(\omega_2 \cdot t) + b_1 \cdot \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} \cdot \sin(\omega_1 \cdot t) + b_2 \cdot \begin{bmatrix} x_1^{(2)} \\ x_2^{(2)} \end{bmatrix} \cdot \sin(\omega_2 \cdot t)$$

$$\begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

$$\begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix} \cdot \begin{bmatrix} b_1 \omega_1 \\ b_2 \omega_2 \end{bmatrix}$$

Lösungsschritte mit MATLAB

1. 'Steifigkeitsmatrix' definieren

$$SM = \begin{bmatrix} 2 \cdot c/m & -c/m \\ -c/m & 2 \cdot c/m \end{bmatrix}$$

2. Eigenwert- Eigenvektorproblem lösen $[V, D] = \text{eig}(SM)$

$$V = \begin{bmatrix} v_{1,1} & v_{2,1} \\ v_{2,1} & v_{2,2} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \end{bmatrix}$$

$$\begin{bmatrix} \omega_1 \\ \omega_2 \end{bmatrix} = \text{sqrt}(\text{diag}(D))$$

3. Anteile der Eigenschwingungen aus den Anfangsbedingungen berechnen
Anfangsauslenkung und Anfangsgeschwindigkeit nach Eigenvektoren zerlegen

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = V \setminus \begin{bmatrix} x_{10} \\ x_{20} \end{bmatrix}$$

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1/\omega_1 \\ 1/\omega_2 \end{bmatrix} .* \text{mldivide}(V, \begin{bmatrix} v_{10} \\ v_{20} \end{bmatrix})$$

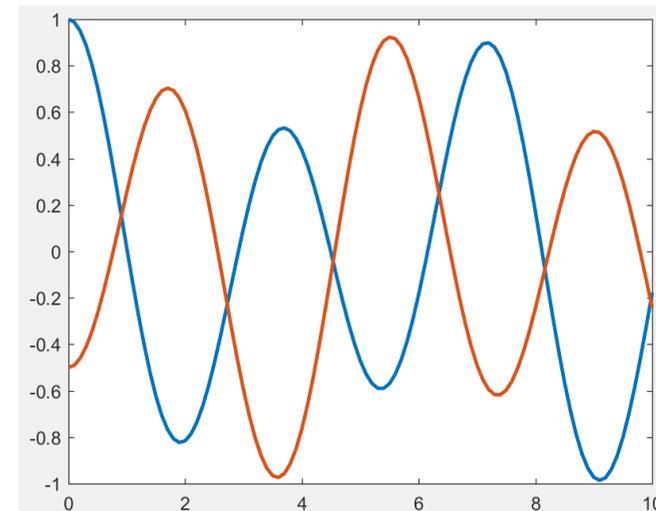
4. Lösung für das Anfangswertproblem :

$$t = 0:0.1:10$$

$$x = V(:,1) * (a_1 * \cos(\omega_1 * t) + b_1 * \sin(\omega_1 * t)) + \\ V(:,2) * (a_2 * \cos(\omega_2 * t) + b_2 * \sin(\omega_2 * t))$$

$$\text{plot}(t, x)$$

Beachte : x ist eine 2*101-Matrix

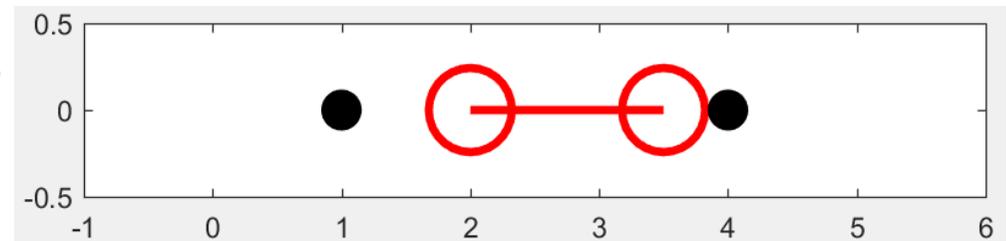


Aufgabe 3 :

Das MATLAB-Skript `prakt4/a3kette2mScr.m` berechnet die Eigenfrequenzen und Eigenvektoren einer linearen Kette mit 2 Massen und 3 Federn. Dann wird die Lösung für das Anfangswertproblem bestimmt und über der Zeit gezeichnet.

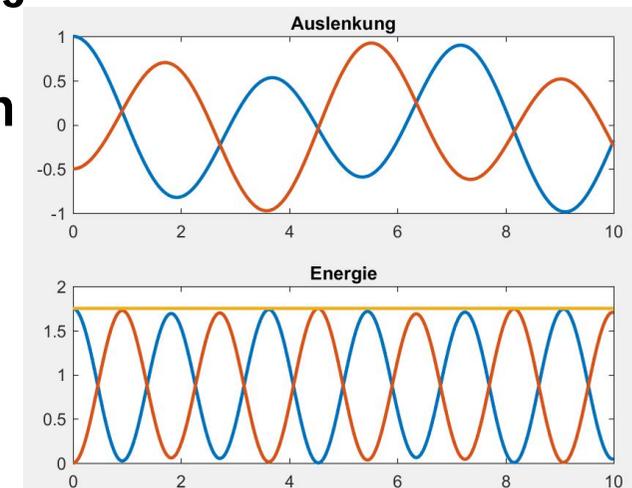
Anschließend wird eine animierte Lösung ausgegeben.

- a) Versuchen Sie das Programm zu verstehen. Wie entsteht die Animation? Was wird genau in den beiden Plot-Anweisungen gezeichnet?



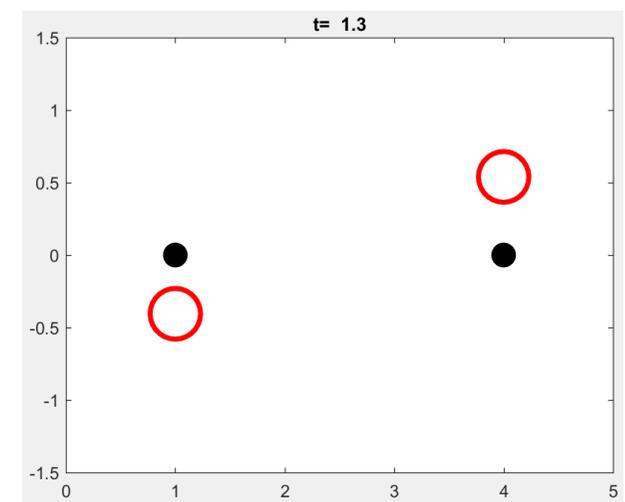
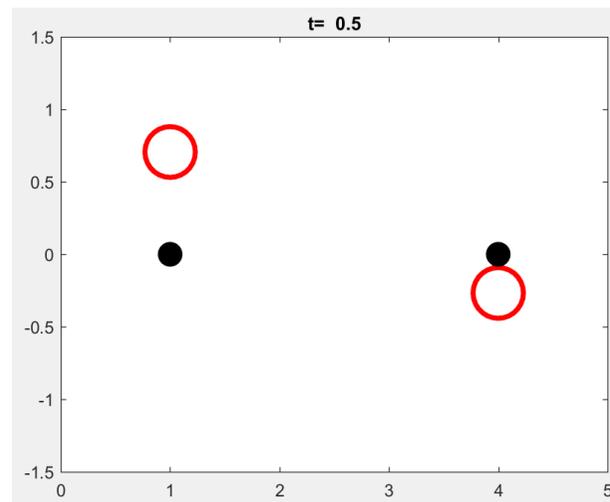
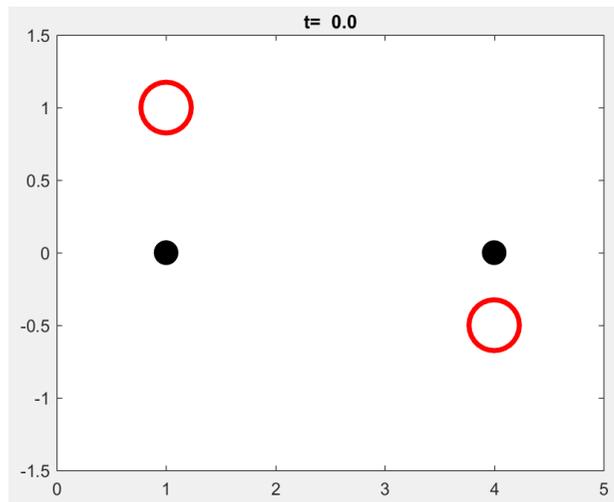
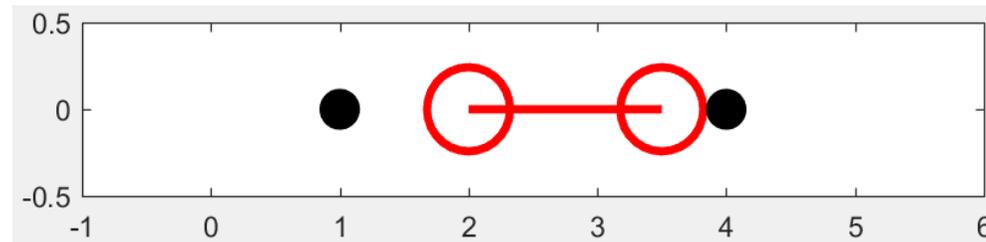
- b) Ändern Sie die Anfangsbedingungen ab.
 c) Versuchen Sie in der Animation die aktuelle Zeit anzuzeigen.
 d) Ändern Sie die Anfangsbedingungen so, dass sich Eigenschwingungen ergeben, d.h. setze $\mathbf{x}_0 = [1 \ 1]$ oder $\mathbf{x}_0 = [1 \ -1]$. Was ergibt sich in diesen Fällen für den Vektor \mathbf{a} ?

- e) Zeichnen Sie in Abhängigkeit der Zeit folgende Größen
- die kinetische Energie, die in beiden Massen insgesamt enthalten ist
 - die potentielle Energie, die in den drei Federn insgesamt steckt
 - die Gesamtenergie im System



Aufgabe 3 :

- f) Was muss am Programm geändert werden, damit die Auslenkungen aus der Ruhelage bei der Animation nicht horizontal sondern vertikal gezeichnet werden?



- g) Was passiert, wenn man den folgenden Abschnitt in der while-Schleife löscht? Welchen Effekt hat dieser Abschnitt?

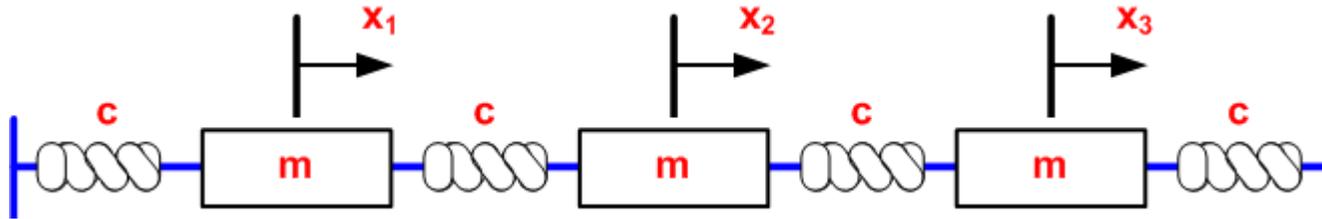
```

if t == 0
    pause(5.0);
else
    pause(delta_t);
end

```

Aufgabe 4 :

Ändern Sie das Skript `prakt4/a4kette3mScr.m` so ab, dass ein System mit 3 Massen gelöst wird. Wie ist die Form der Eigenschwingungen?



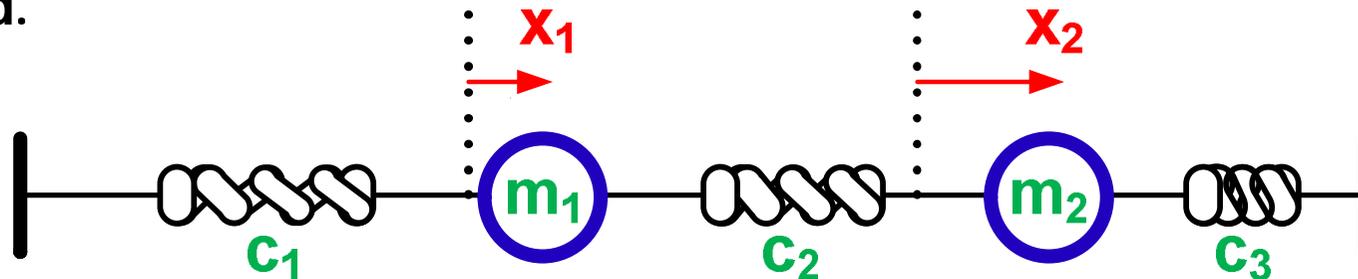
$$\begin{bmatrix} \ddot{x}_1(t) \\ \ddot{x}_2(t) \\ \ddot{x}_3(t) \end{bmatrix} = - \begin{bmatrix} 2c/m & -c/m & 0 \\ -c/m & 2c/m & -c/m \\ 0 & -c/m & 2c/m \end{bmatrix} \cdot \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix}$$

$$\begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = \begin{bmatrix} x_1 \cdot \cos(\omega \cdot t) \\ x_2 \cdot \cos(\omega \cdot t) \\ x_3 \cdot \cos(\omega \cdot t) \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cdot \cos(\omega \cdot t) = \vec{x} \cdot \cos(\omega \cdot t) \quad \begin{bmatrix} \ddot{x}_1(t) \\ \ddot{x}_2(t) \\ \ddot{x}_3(t) \end{bmatrix} = -\omega^2 \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \cdot \cos(\omega \cdot t)$$

$$\begin{bmatrix} 2c/m & -c/m & 0 \\ -c/m & 2c/m & -c/m \\ 0 & -c/m & 2c/m \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \omega^2 \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Aufgabe 5 :

Ändern Sie das Skript `prakt4/a5kette2mScr.m` so ab, dass ein System mit zwei unterschiedlichen Massen und drei unterschiedlichen Federkonstanten gelöst wird.



Bewegungsgleichungen :

$$m_1 \cdot \ddot{x}_1(t) = -c_1 \cdot x_1(t) - c_2 \cdot [x_1(t) - x_2(t)]$$

$$m_2 \cdot \ddot{x}_2(t) = -c_2 \cdot [x_2(t) - x_1(t)] - c_3 \cdot x_2(t)$$

- Stellen Sie zuerst die Gleichungen für das Eigenwert-Eigenvektorproblem auf.
- Ändern Sie dann das Skript entsprechend ab.
- Testen Sie ihr Programm. Wenn die Massen und die Federkonstanten gleich sind, sollte sich wieder die Lösung aus Aufgabe 3 ergeben.

1 Eulerverfahren

2 Eulerverfahren und Function-Handle

3 Lösung von Differentialgleichungen mit ode45

4 Kette mit zwei Massen

Aufgabe 1

In der Vorlesung ist das Eulerverfahren zur Lösung der DGL für eine ungedämpfte Schwingung behandelt worden. Programmieren Sie jetzt analog dazu das Eulerverfahren für eine **gedämpfte Schwingung**. DGL und Anfangsbedingungen lauten allgemein wie folgt :

$$\ddot{y}(t) + 2 \cdot \delta \cdot \dot{y}(t) + \omega^2 \cdot y(t) = 0$$

$$y(t = 0) = y_0 \quad \dot{y}(t = 0) = v_0$$

Die Größe δ beschreibt die Stärke der Dämpfung. Es wird hier angenommen, dass die Dämpfung proportional zur Geschwindigkeit ist. Für $\delta=0$ ergibt sich die ungedämpfte Schwingung. Die Anfangsbedingung lautet : $y(t = 0) = 2, \quad \dot{y}(t = 0) = 0$ d.h. zur Zeit $t=0$ wird die Masse ausgelenkt und dann losgelassen.

a) Schreiben Sie die DGL 2. Ordnung in ein System von zwei DGLn 1. Ordnung um.

b) Wie lautet die Iteration mit der man y_{n+1} und v_{n+1} aus y_n und v_n berechnen kann? y_n ist die Auslenkung zur Zeit t_n und v_n die Geschwindigkeit.

$$t_{n+1} = t_n + \Delta t$$

$$y_{n+1} = y_n +$$

$$v_{n+1} = v_n +$$

$$\text{AW: } t_1 = 0$$

$$y_1 = \text{Anfangsauslenkung}$$

$$v_1 = \text{Anfangsgeschwindigkeit}$$

Aufgabe 1

- c) Öffnen Sie das Skript `a1EulerGedsw.m` und ergänzen Sie es so, dass die DGL im Zeitintervall $[0, 30]$ gelöst wird. Geben Sie die Auslenkung und die Geschwindigkeit in Abhängigkeit von der Zeit graphisch aus. Variieren Sie die Schrittweite Δt .
- d) Wie sieht eine Lösung bei sehr kleiner Dämpfung aus? Wie sieht eine Lösung für sehr große Dämpfung aus? Was bedeutet eigentlich klein oder groß? Wie könnte man groß oder klein definieren?
- e) Variieren Sie bei $\delta=0$ die Schrittweite und das Zeitintervall.
- f) Wie muss die Schrittweite Δt gewählt werden, damit die Lösung auf 1% genau berechnet wird?

Aufgabe 2

Es wird zusätzlich eine äußere Kraft berücksichtigt, die auf das gedämpfte, schwingende System einwirkt – eine sinusförmige Anregung mit der Frequenz ω_1 . Die DGL lautet dann :

$$\ddot{y}(t) + 2 \cdot \delta \cdot \dot{y}(t) + \omega^2 \cdot y(t) = F \cdot \sin(\omega_1 \cdot t)$$

$$y(t=0) = y_0 \quad \dot{y}(t=0) = v_0$$

Die Größe F beschreibt die Stärke der Anregung. F=0 bedeutet, dass keine äußere Kraft einwirkt. Beachten Sie, dass es zwei verschiedene Frequenzen gibt, die Frequenz der **äußeren Anregung** ω_1 und die „**innere Frequenz**“ ω .

a) Wie lautet jetzt das System von DGLn 1. Ordnung?

b) Wie lautet die Iteration, mit der man y_{n+1} und v_{n+1} aus y_n und v_n berechnen kann? y_n ist die Auslenkung zur Zeit t_n und v_n die Geschwindigkeit.

$$t_{n+1} = t_n + \Delta t$$

$$y_{n+1} = y_n +$$

$$v_{n+1} = v_n +$$

$$\text{AW: } t_1 = 0$$

$$y_1 = \text{Anfangsauslenkung}$$

$$v_1 = \text{Anfangsgeschwindigkeit}$$

Aufgabe 2

c) Öffnen Sie das Skript `a2EulerAngeregt.m`. Es enthält die Lösung von Aufgabe 1. Ergänzen Sie das Skript so, dass die gedämpfte Schwingung mit äußerer Anregung im Intervall $[0, 30]$ gelöst wird.

In diesem Skript werden nicht mehr zwei Vektoren y und v verwendet.

y wird zu einer Matrix mit zwei Spalten. Spalte 1 enthält die Auslenkung und Spalte 2 die Geschwindigkeit.

$$y(n) \rightarrow y(n,1)$$

$$v(n) \rightarrow y(n,2)$$

$$y(n+1,1) = y(n,1) + y(n,2) \cdot \Delta t$$

$$y(n+1,2) = y(n,2) - (2 \cdot \delta \cdot y(n,2) + \omega^2 \cdot y(n,1)) \cdot \Delta t$$

Bei einem System von k DGLn erster Ordnung wird eine Matrix mit k Spalten verwendet. Spalte j enthält die Funktionswerte Lösungsfunktion j ($1 \leq j \leq k$). `plot(t(:), y(:,j))` zeichnet die Lösungsfunktion j .

Die Zeile `y(n,:)` enthält die Funktionswerte zur Zeit $t(n)$.

d) Was passiert, wenn die beiden Frequenzen ω und ω_1 annähernd gleich sind?

Wie nennt man dieses Phänomen?

Wie sieht die Lösung für $\omega \gg \omega_1$ typischerweise aus? Was bedeutet das physikalisch?

Wie sieht die Lösung für $\omega \ll \omega_1$ typischerweise aus?

Aufgabe 3

Erstellen Sie ein Skript `a3Euler.m`, das eine beliebige DGL 2. Ordnung lösen kann. Welche DGL vom Eulerverfahren konkret gelöst wird, wird über ein Function-Handle festgelegt. Über dieses Function-Handle werden die beiden ersten Ableitungen berechnet.

- Ergänzen Sie die Funktion `a3gedswdgl.m`. Diese Funktion besitzt zwei Parameter, nämlich die Zeit t und den Vektor y (y enthält zwei Elemente). Das erste Element von y ist die Auslenkung zur Zeit t und das zweite die Geschwindigkeit zur Zeit t . Die Funktion gibt als Rückgabewert die beiden ersten Ableitungen der zugehörigen DGL zurück (siehe DGL 1. Ordnung aus Aufgabe 1).
- Im Skript `a3Euler.m` wird in Zeile 12 ein Function-Handle gesetzt. Damit wird die DGL festgelegt, die im Folgenden gelöst wird. In Zeile 20 werden dann über das Function-Handle die beiden ersten Ableitungen zur Zeit t berechnet. Damit wird die neue Auslenkung und die neue Geschwindigkeit berechnet. Siehe nächste Folie. Testen Sie, ob ihr Programm die richtige Lösung liefert.
- Wie muss die Funktion `a3gedswdgl.m` geändert werden, um eine zusätzliche äußere Kraft der Form $F(t) = 2 \cdot \sin(3 \cdot t)$ zu berücksichtigen.

Bei diesem Beispiel wird dann auch der erste Aufrufparameter (die Zeit t) der Funktion `a3gedswdgl.m` verwendet.

Aufgabe 3

```
Editor - U:\Matlab\prakt5\a3Euler.m
a3Euler.m x +
6 % Anfangswerte setzen
7 n=1;
8 t(n)=0;
9 y(n,1)=2.0;
10 y(n,2)=0.0;
11 % setze Funktion-Handle
12
13
14 % Iteration
15 dt = 0.01; % Schrittweite
16 while ( t(n) < 30 )
17     t(n+1) = t(n) + dt;
18     % berechne die erste Ableitung
19     % mit Hilfe des Function-Handles
20
21     y(n+1,1)= y(n,1) ; % ergänzen
22     y(n+1,2)= y(n,2) ; % ergänzen
23     n = n+1;
24 end
```

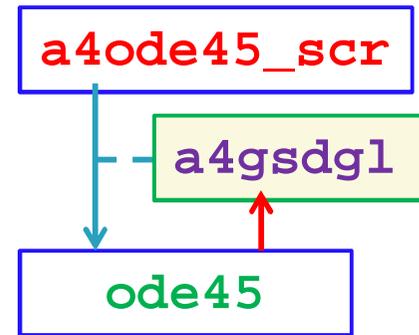
```
Editor - U:\Matlab\prakt5\a3gedswdgl.m
a3gedswdgl.m x +
1 function [dy_dt] = a3gedswdgl(t,y)
2
3     global w d;
4
5     %erste Ableitungen berechnen
6     dy_dt(1,1) =
7
8     dy_dt(2,1) =
9
10    end
```

Aufgabe 4:

Lösen Sie die DGL für eine gedämpfte Schwingung mit Hilfe von `ode45`.

$$\frac{d^2 y}{dt^2} + 2 \cdot \delta \cdot \frac{dy}{dt} + \omega^2 \cdot y(t) = 0$$

$$y(t=0) = 2 \quad \dot{y}(t=0) = 0$$

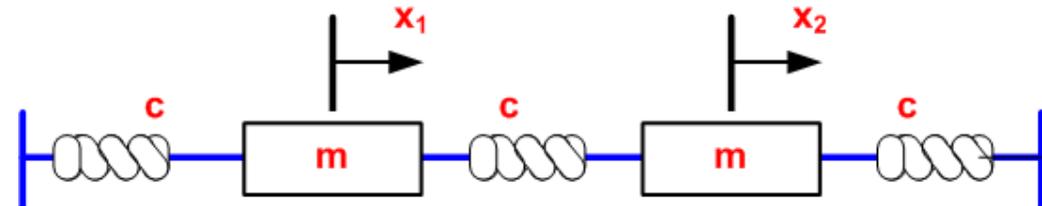


- Verwenden Sie die beiden Dateien `a4ode45_scr` und `a4gsdgl`. Ergänzen Sie das Skript um den Aufruf der MATLAB-Funktion `ode45`. Stellen Sie die Auslenkung und Geschwindigkeit in Abhängigkeit von der Zeit graphisch dar.
- `ode45` verwendet keine feste Schrittweite Δt für die Zeitintervalle. Wie groß ist der Abstand der Zeitpunkte bei $t=0$ und bei $t=20$? Wie viele verschiedene Zeitpunkte werden für die Lösung verwendet? Ändern Sie das Skript so, dass die entsprechenden Antworten ausgegeben werden.
- Ändern Sie den Funktionsaufruf von `ode45` so, dass die Lösungen in Zeitabständen von jeweils 0.02 berechnet werden (siehe `help ode45`).
- Ändern Sie die beiden Programme so ab, dass auch eine äußere Anregung mit berücksichtigt werden kann. Die Amplitude der Anregung und die äußere Frequenz werden im Skript gesetzt.

$$\frac{d^2 y}{dt^2} + 2 \cdot \delta \cdot \frac{dy}{dt} + \omega^2 \cdot y(t) = A \cdot \sin(\omega_1 \cdot t)$$

Aufgabe 5:

Gegeben ist eine Kette mit zwei schwingenden Massen.



Dieses System wird durch folgende DGL beschrieben :

$$m \cdot \ddot{x}_1(t) = -c \cdot x_1(t) - c \cdot [x_1(t) - x_2(t)]$$

$$m \cdot \ddot{x}_2(t) = -c \cdot [x_2(t) - x_1(t)] - c \cdot x_2(t)$$

$$\ddot{x}_1(t) = -2 \cdot x_1(t) + x_2(t)$$

$$\ddot{x}_2(t) = -2 \cdot x_2(t) + x_1(t)$$

Die Massen m und die Federkonstanten werden 1 gesetzt. Damit ergibt sich das System auf der rechten Seite.

a) Schreiben Sie die beiden DGLn 2. Ordnung in ein System von vier DGLn erster Ordnung um. Hierzu müssen Sie vier Hilfsgrößen einführen :

$$y_1(t) = x_1(t) \quad y_2(t) = \dot{x}_1(t) \quad y_3(t) = x_2(t) \quad y_4(t) = \dot{x}_2(t)$$

Das System wird dann durch einen Vektor y mit vier Komponenten beschrieben. Die Anfangsbedingungen legen die 4 Komponenten von y zur Zeit $t=0$ fest.

$$\vec{y}(t) = \begin{bmatrix} x_1(t) \\ \dot{x}_1(t) \\ x_2(t) \\ \dot{x}_2(t) \end{bmatrix}$$

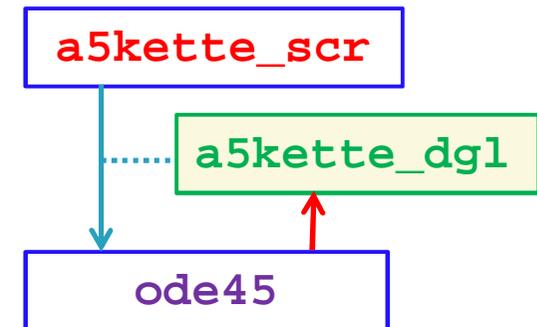
$$\vec{\dot{y}}(t) = \begin{bmatrix} \dot{y}_1(t) = \\ \dot{y}_2(t) = \\ \dot{y}_3(t) = \\ \dot{y}_4(t) = \end{bmatrix}$$

$$\vec{y}(t=0) = \begin{bmatrix} x_1(t=0) = x10 \\ \dot{x}_1(t=0) = 0 \\ x_2(t=0) = x20 \\ \dot{x}_2(t=0) = 0 \end{bmatrix}$$

Aufgabe 5:

Verwenden Sie die beiden Programme `a5kette_scr` und `a5kette_dgl` um das Anfangswertproblem zu lösen.

- b) In der Funktion `a5kette_dgl` wird die erste Ableitung von y berechnet. Die Funktion gibt einen Spaltenvektor mit vier Zeilen zurück.
- c) Im Skript `a5kette_scr.m` wird die Anfangsbedingung gesetzt und die Funktion `ode45` aufgerufen. Diese liefert eine Matrix mit vier Spalten zurück. Welche Spalte enthält was? In welchen Spalten befinden sich die Auslenkungen? Wie wird das festgelegt? Dann werden die Auslenkungen mit Hilfe von `subplot` in zwei Fenstern dargestellt.
- d) Wie müssen die Anfangsbedingungen gewählt werden, damit sich Eigenschwingungen ergeben? Ändern Sie die Anfangsbedingungen so, dass sich Eigenschwingungen ergeben.



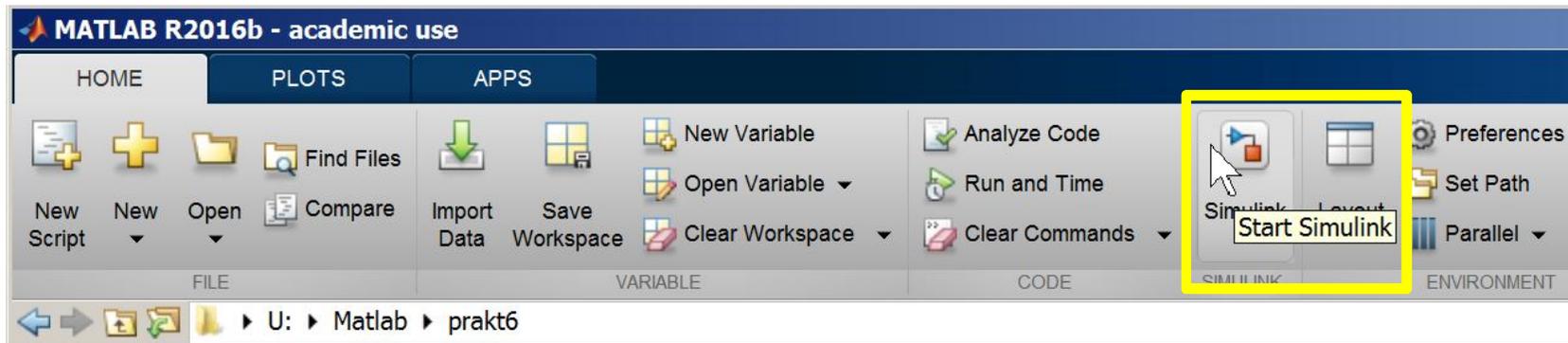
- 1 Sinusgenerator und Integration**
- 2 Nichtlineare Schwingung**
- 3 Nichtlineare Schwingung und MATLAB-Variablen**
- 4 Nichtlineare Schwingung und Function-Block**
- 5 Nichtlineare Schwingung und Subsystem-Block**
- 6 Model-Workspace**
- 7 Nichtlineare gedämpfte Schwingung mit äußerer Kraft, die exponentiell abnimmt**
- 8 Angeregte Schwingung mit einer Dämpfung proportional zum Quadrat der Geschwindigkeit**

Aufgabe 1:

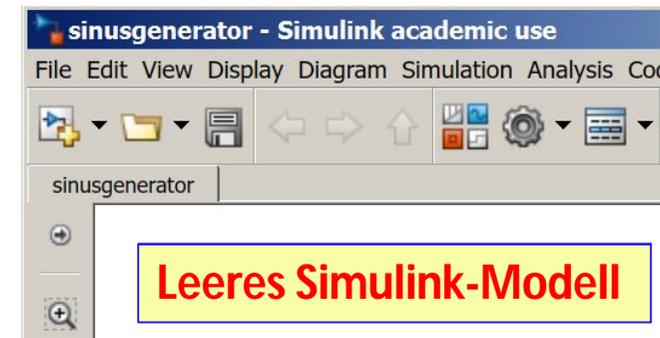
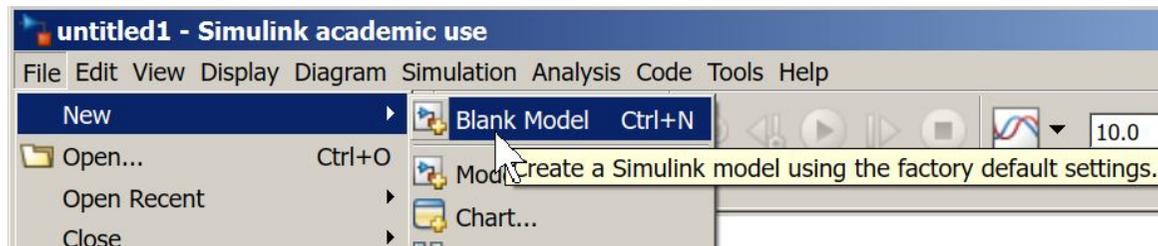
Erstellen Sie ein Simulink-Modell `sinusgenerator.slx`, das ein sinusförmiges Signal $a(t) = 1.2 * \sin(2 * t + 30^\circ)$ im Zeitintervall $[0, 20]$ integriert. Das Ergebnis und das Ausgangssignal sollen graphisch dargestellt werden.

Arbeitsschritte :

1. Starten Sie Simulink über das Simulink-Icon der MATLAB-Toolbar.



2. Erstellen Sie ein Simulink-Modell über File->New->Blank Model und speichern Sie das Modell in der Datei `sinusgenerator.slx` auf dem U-Laufwerk.



3. Block im Simulink Library Browser selektieren und in das Simulink Modell ziehen

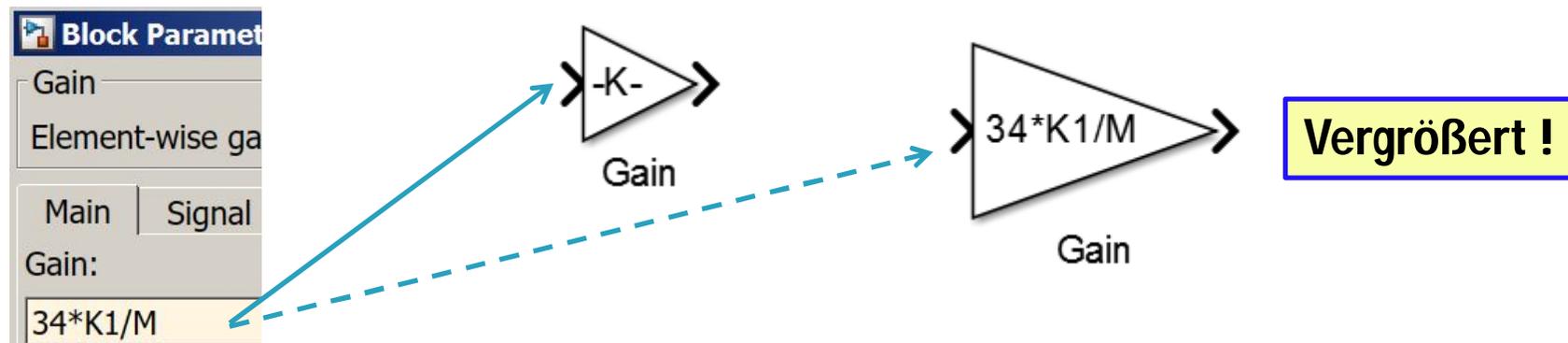
Blöcke verbinden :

bei gedrückter linker Maustaste vom Ausgangsport zum Zielport ziehen

Größe eines Blocks verändern :

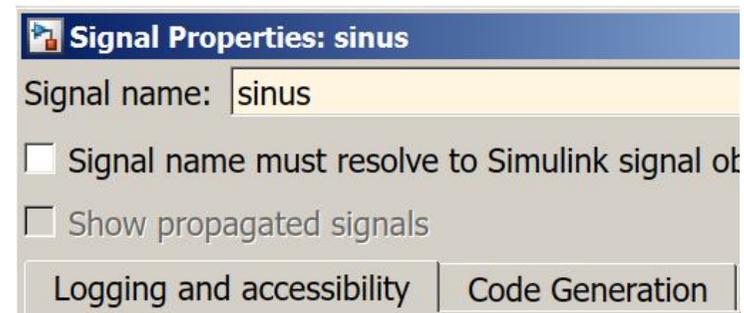
Block selektieren und dann an den Ecken ziehen

Wird die Größe eines Blocks verändert, kann sich die Darstellung von Informationen über den Block ändern. Beispiel : der Faktor, der in einem Gain-Block eingetragen ist



Signal : Namen zuweisen

Signal selektieren und das Kontextmenü öffnen, dann den Menüpunkt „Properties“ wählen und bei „Signal name“ den Namen eintragen



sinusgenerator * - Simulink academic use

File Edit View Display Diagram Simulation Analysis Code Tools Help

sinusgenerator

Explore

- Cut Ctrl+X
- Copy Ctrl+C
- Paste Ctrl+V
- Comment Through Ctrl+Shift+Y
- Comment Out Ctrl+Shift+X
- Delete Del
- Find Referenced Variables
- Create Subsystem from Selection Ctrl+G
- Format
- Rotate & Flip**
 - Clockwise Ctrl+R
 - Counterclockwise Ctrl+Shift+R
 - Flip Block** Ctrl+I
 - Flip Block Name
- Arrange
- Mask
- Library Link
- Signals & Ports
- Requirements Traceability

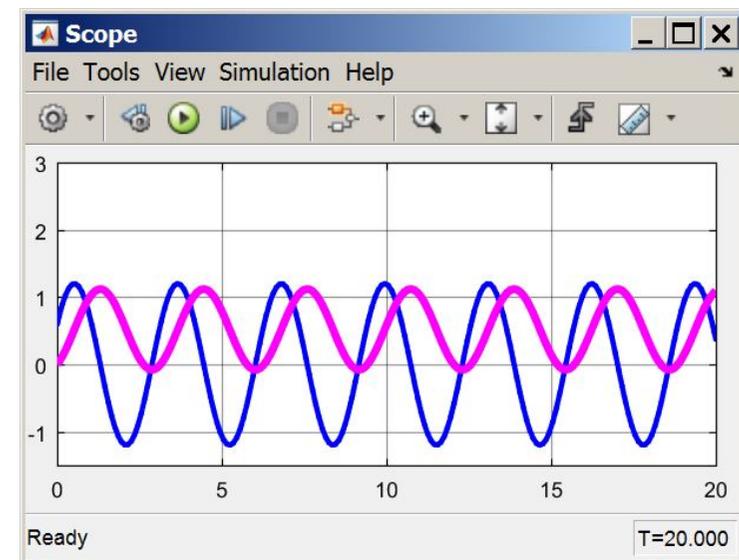
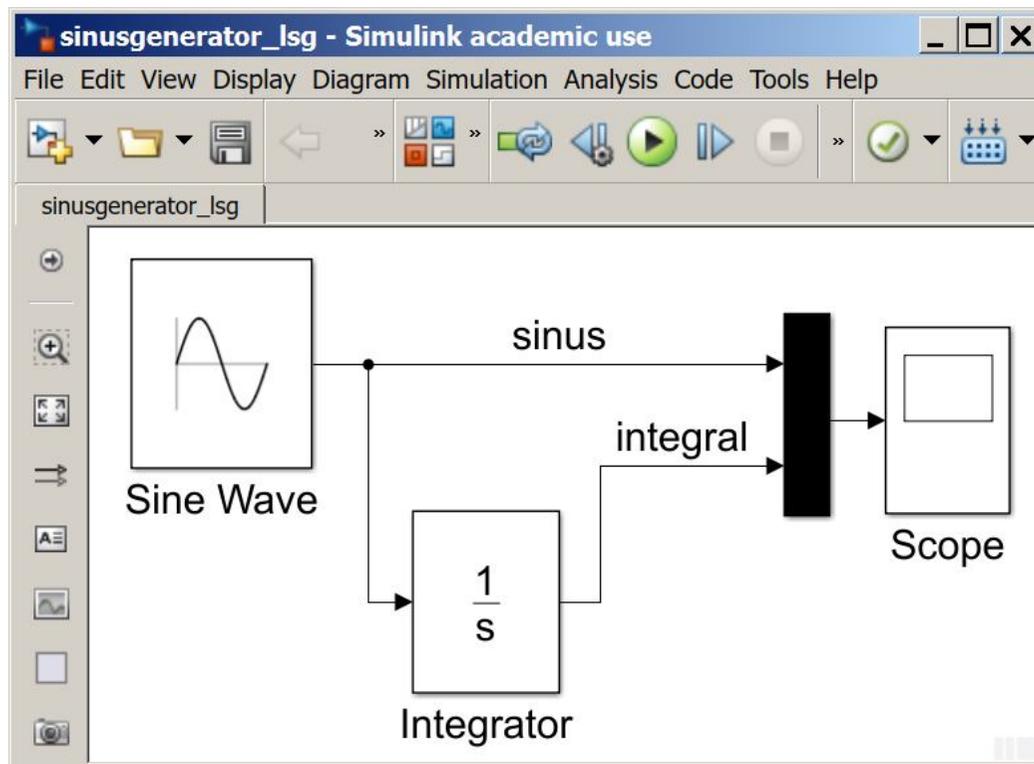
Über das Kontext-Menü lassen sich Eigenschaften eines Blocks konfigurieren und verändern :

- Block drehen
- Block spiegeln (Flip)
- Schriftart und Schriftgröße ändern
- Farben (Fore- Background)

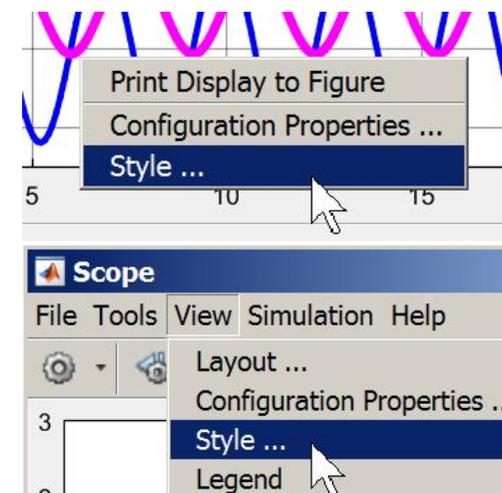
Gain -K- Flip Gain

6.1 Sinusgenerator und Integration

- Setzen Sie die Parameter des Blocks „Sine Wave“ um das gewünschte Signal zu erzeugen. Setzen Sie die Simulationsdauer und starten Sie das Modell. Prüfen Sie das Ergebnis im Block Scope.

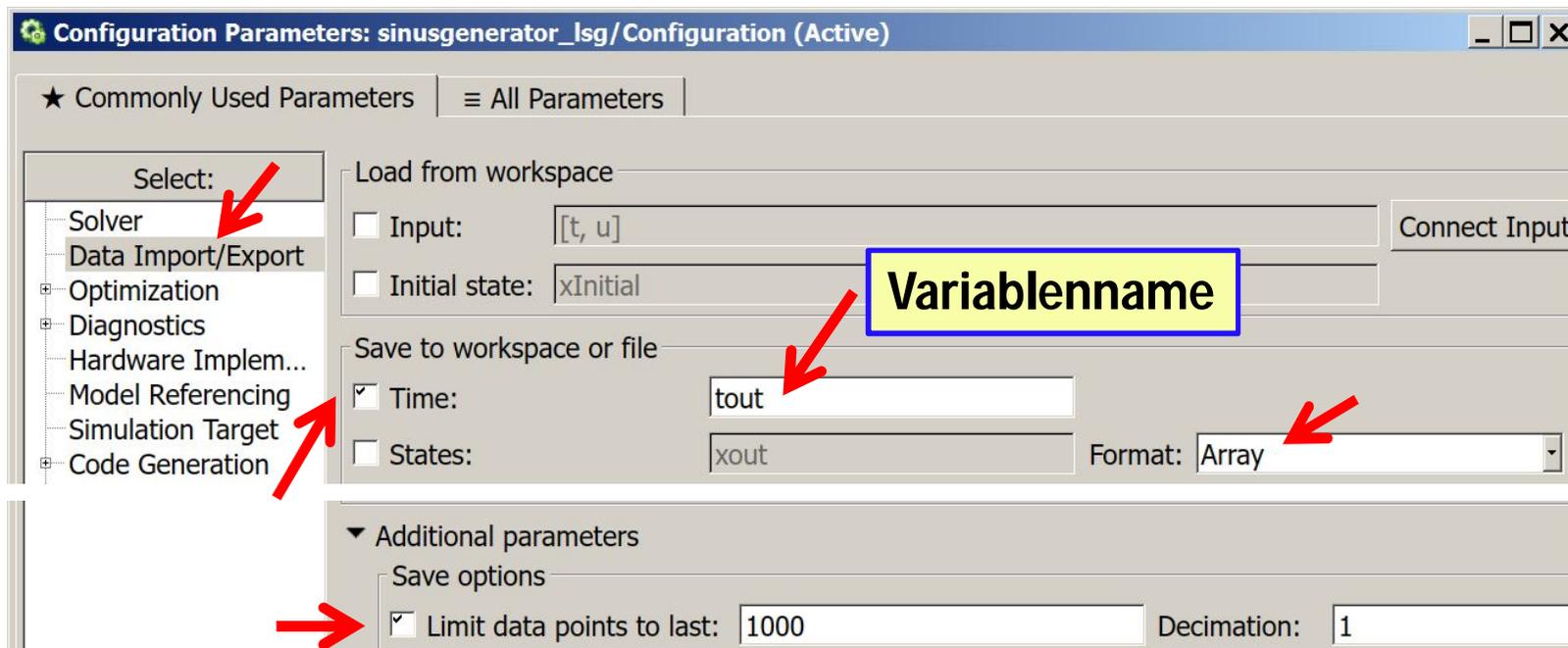
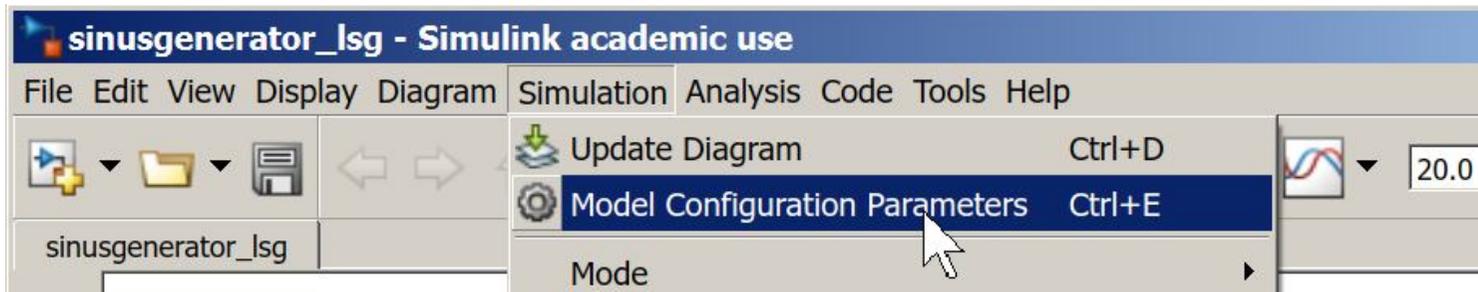


- Ändern Sie die Linienstärke und die Farben der Linien. Öffnen Sie hierzu das Kontext-Menü des Scope-Fensters oder wählen Sie Menü->View



5. Wie viele Zeitpunkte werden verwendet, um das Integral im Zeitraum [0, 20] zu berechnen ? Wie können Sie das beeinflussen?

Öffnen Sie das Menü **Simulation -> Model Configuration Parameters -> Data Import /Export** und aktivieren Sie die Ausgabe der Zeitpunkte (Zeitpunkte, die bei der Simulation verwendet werden) in den Workspace. Starten Sie dann die Simulation. Wie viele Elemente besitzt der Vektor tout? Sind die Zeitabstände äquidistant?



6.1 Sinusgenerator und Integration

6. Die Kurven, die im Scope-Block dargestellt werden, sind nicht glatt, weil die Zeitabstände, an denen die Signalwerte berechnet werden, zu groß sind. Begrenzen Sie die Zeitabstände auf 0.02. Verwenden Sie einem Solver vom Typ **Variable-step** und dann einen Solver vom Typ **Fixed-step**. Wie viele Zeitpunkte werden jeweils verwendet?

The screenshot shows the 'Configuration Parameters' dialog for 'sinusgenerator_lsg/Configuration (Active)'. The 'Variable-step' solver is selected, highlighted by a blue box. The 'Type' dropdown is set to 'Variable-step', also indicated by a red arrow. The 'Max step size' is set to 0.1, with a red arrow pointing to it. The 'Solver' dropdown is set to 'ode45 (Dormand-Prince)'. The 'Simulation time' is set from 0.0 to 20.0. The 'Additional options' section is expanded, showing 'Relative tolerance' at 1e-3, 'Absolute tolerance' at auto, and 'Shape preservation' at Disable All. The left sidebar shows 'Solver' selected under the 'Select:' category, with a red arrow pointing to it.

The screenshot shows the 'Configuration Parameters' dialog for 'sinusgenerator_lsg/Configuration (Active)'. The 'Fixed-step' solver is selected, highlighted by a blue box. The 'Type' dropdown is set to 'Fixed-step', also indicated by a red arrow. The 'Solver' dropdown is set to 'ode4 (Runge-Kutta)'. The 'Fixed-step size (fundamental sample time)' is set to 0.02, with a red arrow pointing to it. The 'Simulation time' is set from 0.0 to 20.0. The left sidebar shows 'Solver' selected under the 'Select:' category, with a red arrow pointing to it.

Zusatzaufgaben :

- a) Setzen Sie den Anfangswert für das Integral auf 0.3.
- b) Weisen Sie den beiden Signalen Namen zu (jeweils ein Signal selektieren und dann das Kontextmenü öffnen).
- c) Ändern Sie die Darstellung des Mux-Blockes, so dass die Signalnamen an den Eingangsports angezeigt werden. Ändern Sie die Hintergrundfarbe des Integratorblocks. Ändern Sie die Foreground Color des Sinus-Blocks.
- d) Ändern Sie die Darstellung des Ergebnisses. Der Scope-Block soll die beiden Signale in zwei verschiedenen Fenstern darstellen.
- e) Begrenzen Sie den y-Bereich, der im Scope-Block angezeigt wird, auf [-2, +2].
- f) Begrenzen Sie die gespeicherte Datenmenge auf den letzten 30 Punkte. Was wird dann im Scope-Block angezeigt ?
- g) Speichern Sie die Werte der beiden Signale in Variablen des Workspace ab. Verwenden Sie verschiedene Lösungsvarianten
 - Scope-Block
 - ToWorkspace-Block
- h) Setzen Sie die Amplitude und die Frequenz der Sinus-Schwingung mit Hilfe der Variablen **A** und **omega** , die im MATLAB-Workspace definiert werden. Überzeugen Sie sich, dass die Werte aus dem Workspace tatsächlich von Simulink übernommen werden.

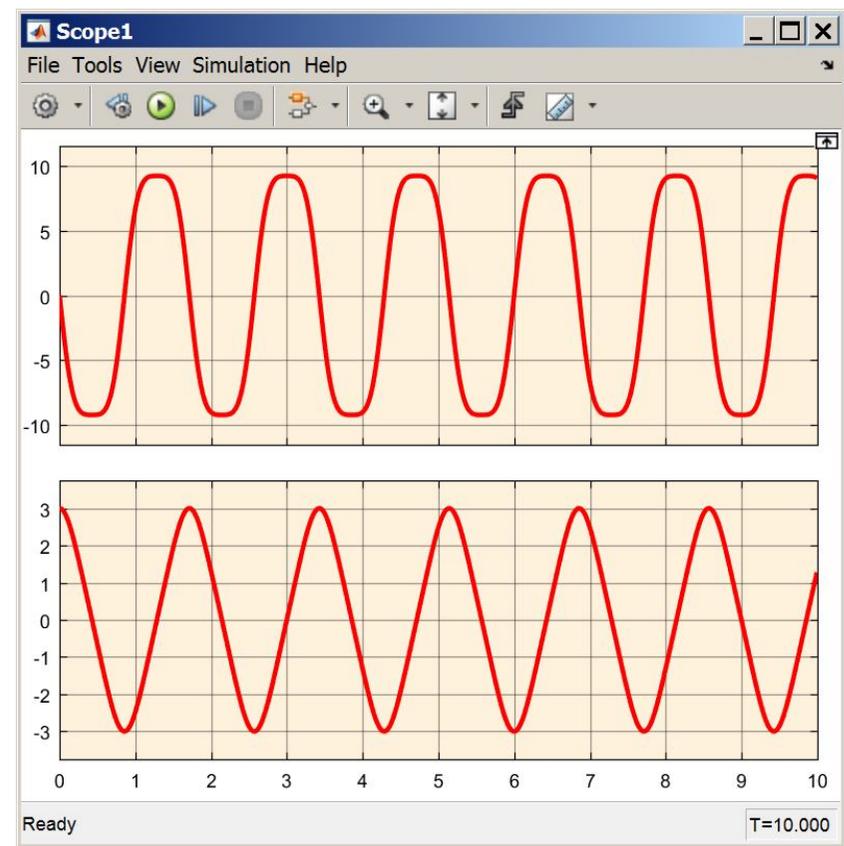
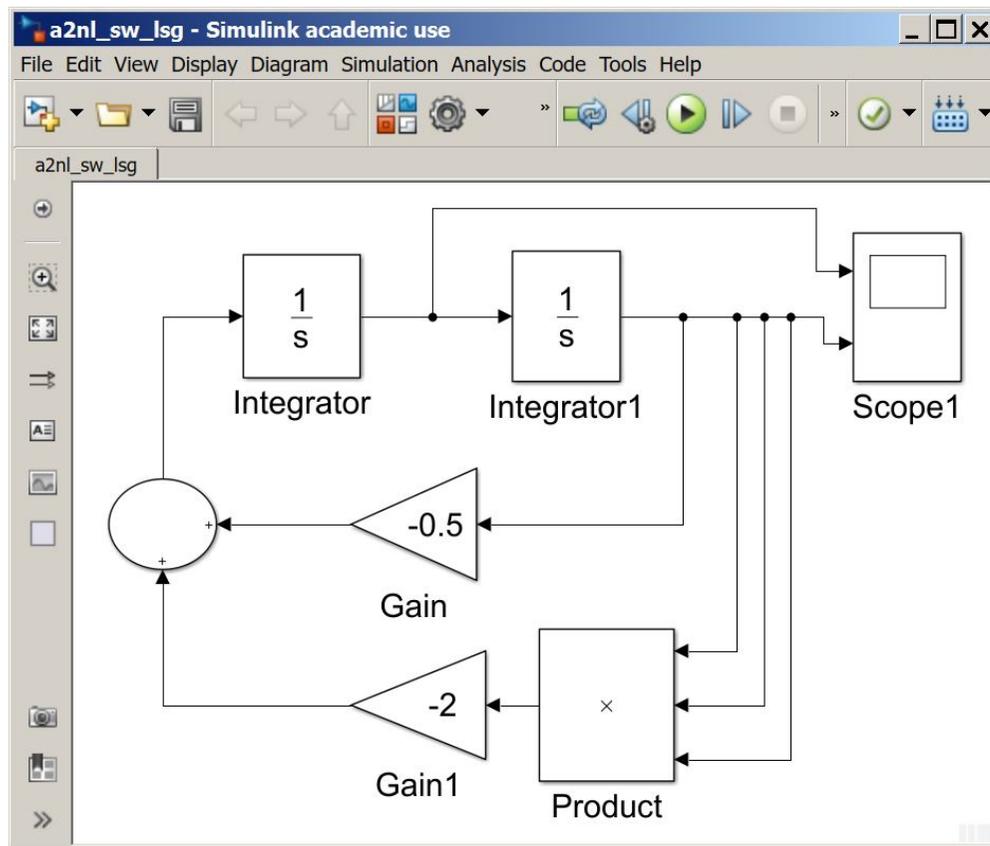
P6_9 6.2 Nichtlineare Schwingung

Aufgabe 2: Erstelle `a2nl_sw.slx`

Lösen Sie die folgende nichtlineare DGL zur Beschreibung einer Schwingung mit Simulink. Auslenkung und Geschwindigkeit sollen in zwei unterschiedlichen Fenstern eines Scope-Blocks dargestellt werden. Begrenzen Sie die Schrittweite.

$$\frac{d^2y}{dt^2} + 0.5 * y(t) + 2 * y(t)^3 = 0$$

$$v_0 = \dot{y}(t = 0) = 0, \quad y_0 = y(t = 0) = 3$$

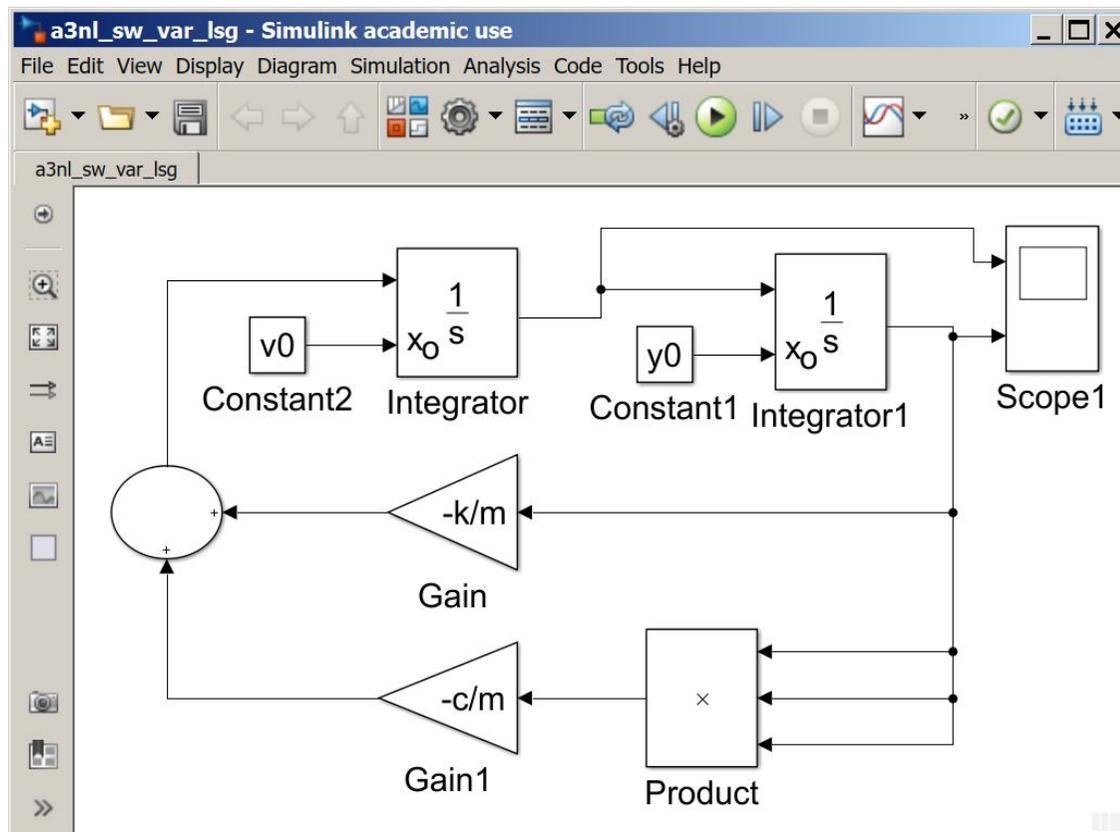


Aufgabe 3:

Verwenden Sie für die Parameter der nichtlinearen DGL drei **Variablen** k , m und c , die im MATLAB-Workspace definiert werden. Die Anfangsbedingungen werden über zwei MATLAB-Variablen $v0$ und $y0$ definiert. Öffnen Sie `a3nl_sw_var.slx` und ändern Sie das Modell geeignet ab. Starten Sie dann die Simulation.

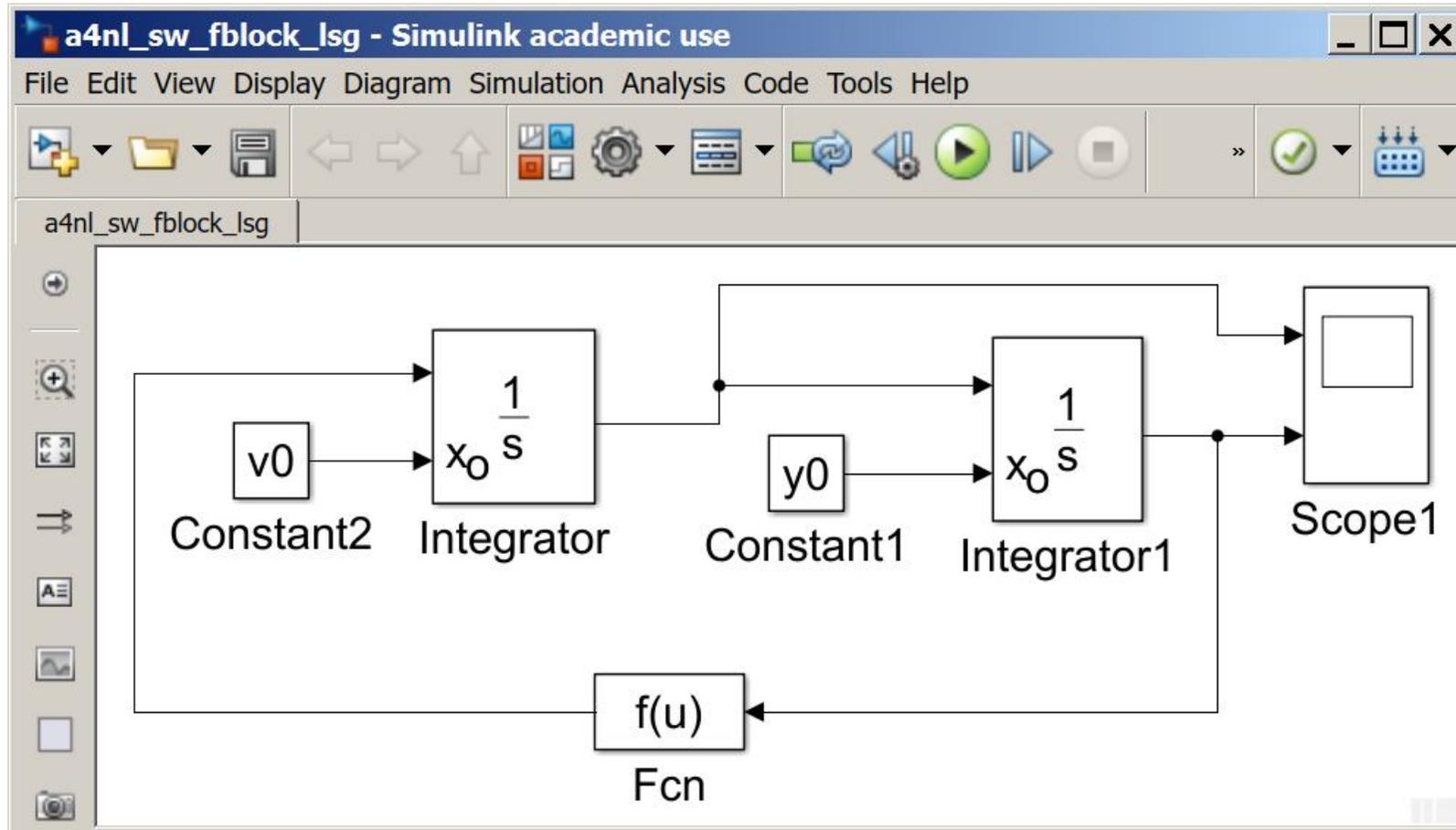
$$\frac{d^2 y}{dt^2} + \frac{k}{m} * y(t) + \frac{c}{m} * y(t)^3 = 0$$

$$\dot{y}(t = 0) = v0 \quad , \quad y(t = 0) = y0$$



Aufgabe 4:

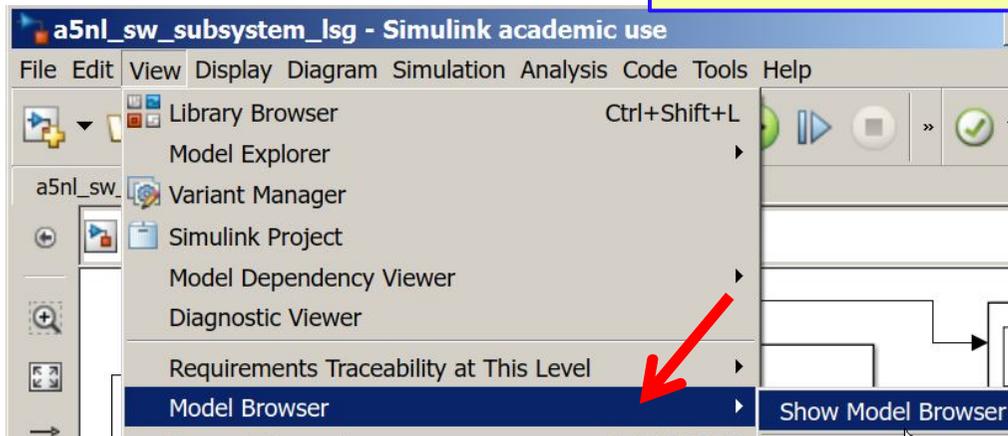
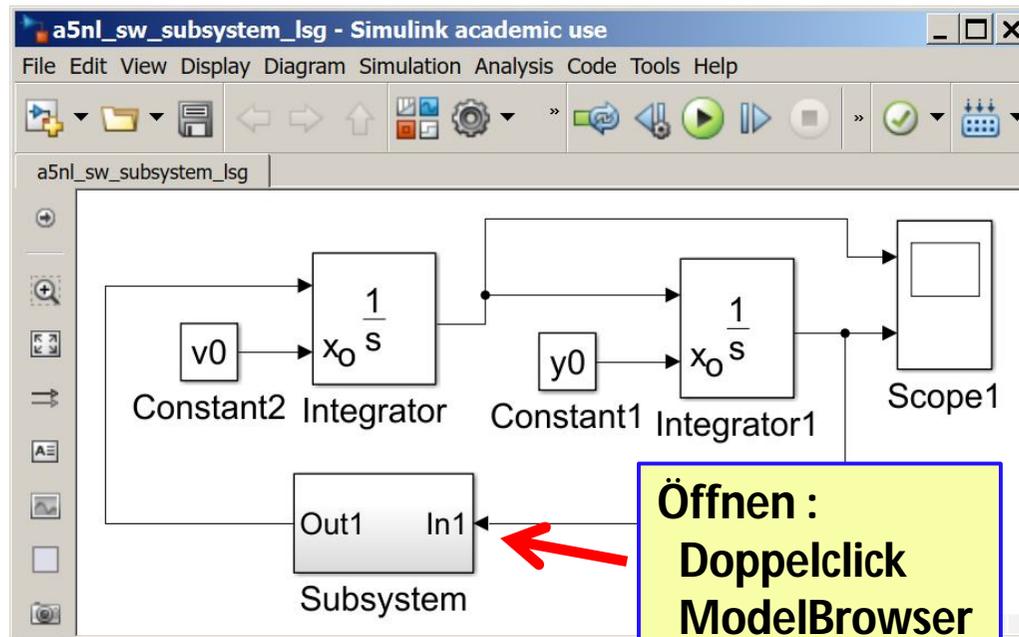
Ersetzen Sie die Blöcke zur Berechnung der Beschleunigung im Modell `a4nl_sw_fblock.slx` durch einen Function-Block.



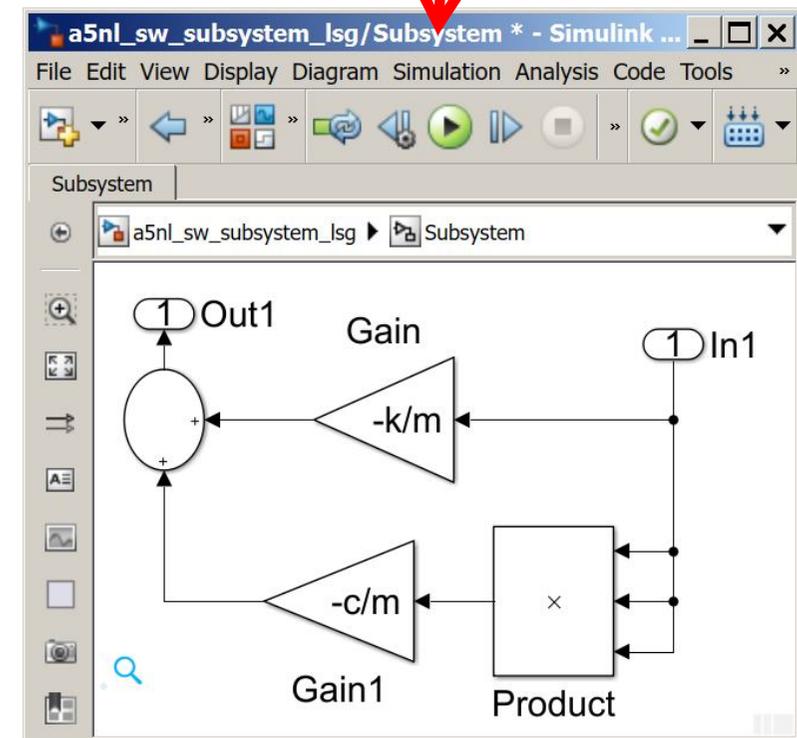
P6_12 6.5 Nichtlineare Schwingung und Subsystem-Block

Aufgabe 5:

Ersetzen Sie die Blöcke zur Berechnung der Beschleunigung im Modell `a5n1_sw_subsystemVar1.slx` durch einen Subsystem-Block.

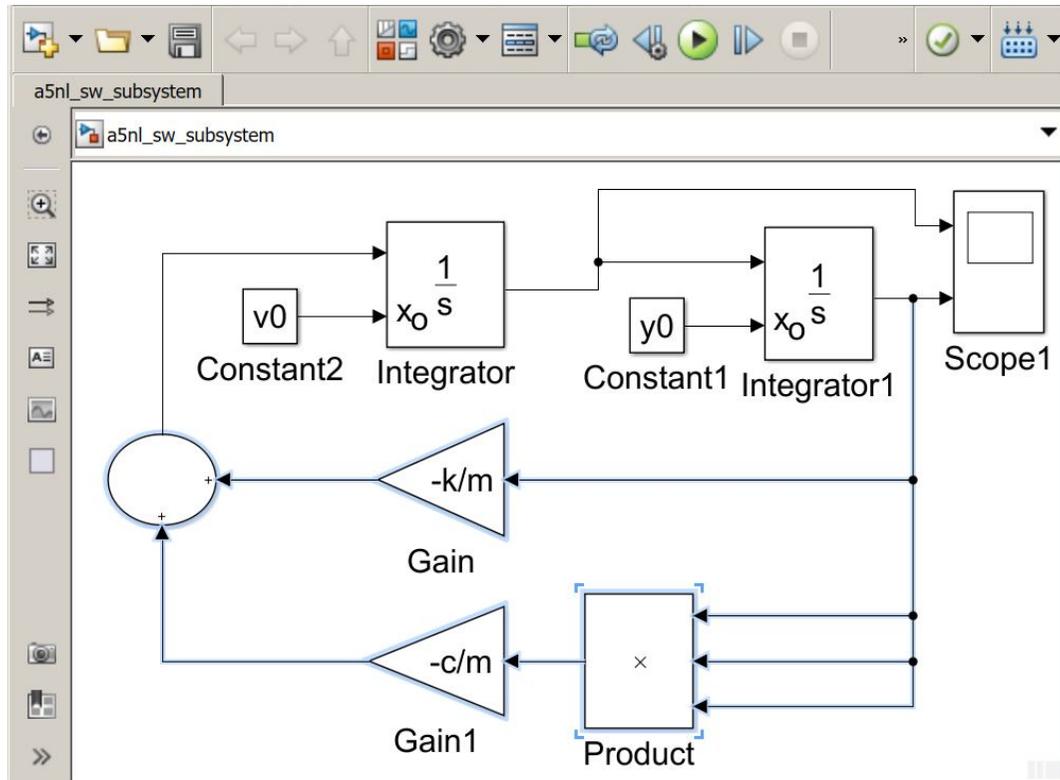


Variante 1 : Subsystem-Block einfügen und danach das Subsystem aufbauen



Variante 2 : Bestehende Blöcke in ein Subsystem verlagern

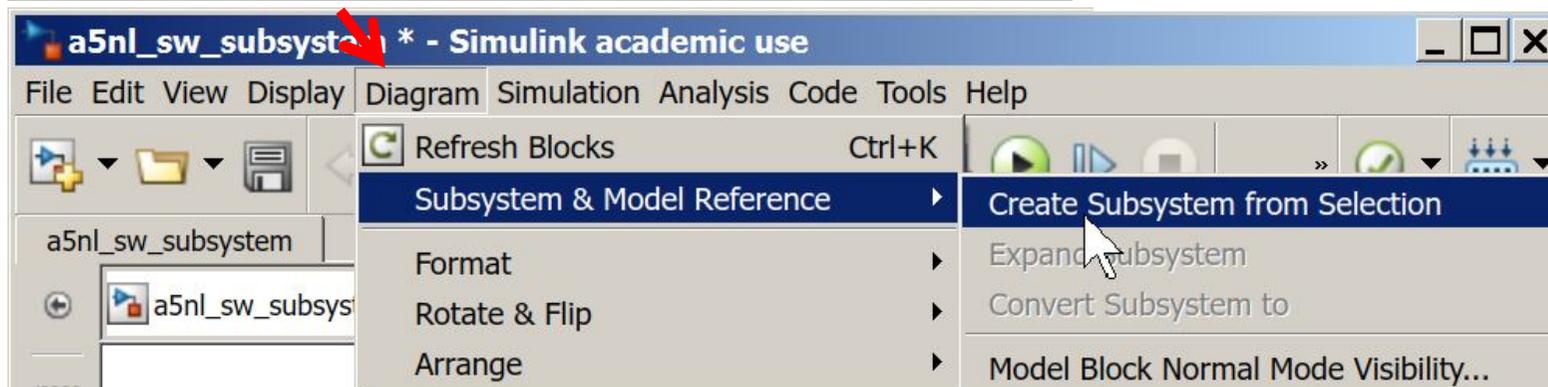
a5nl_sw_subsystemVar2.slx



Alle Blöcke, die in das Subsystem verlagert werden sollen, werden selektiert, ebenso die Eingangs- und die Ausgangssignale. Die Selektion erfolgt auf zwei Arten

- einen rechteckigen Bereich wählen
- Drücken der Shift-Taste und Selektion

Danach Menü **Diagram-> Subsystem&Model Reference -> Create Subsystem from Selection** oder **Kontextmenü -> Create Sub-system from Selection** wählen.



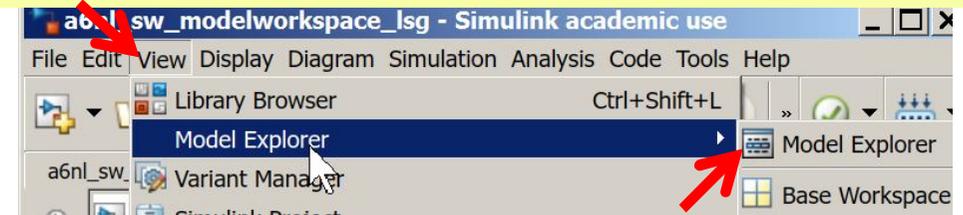
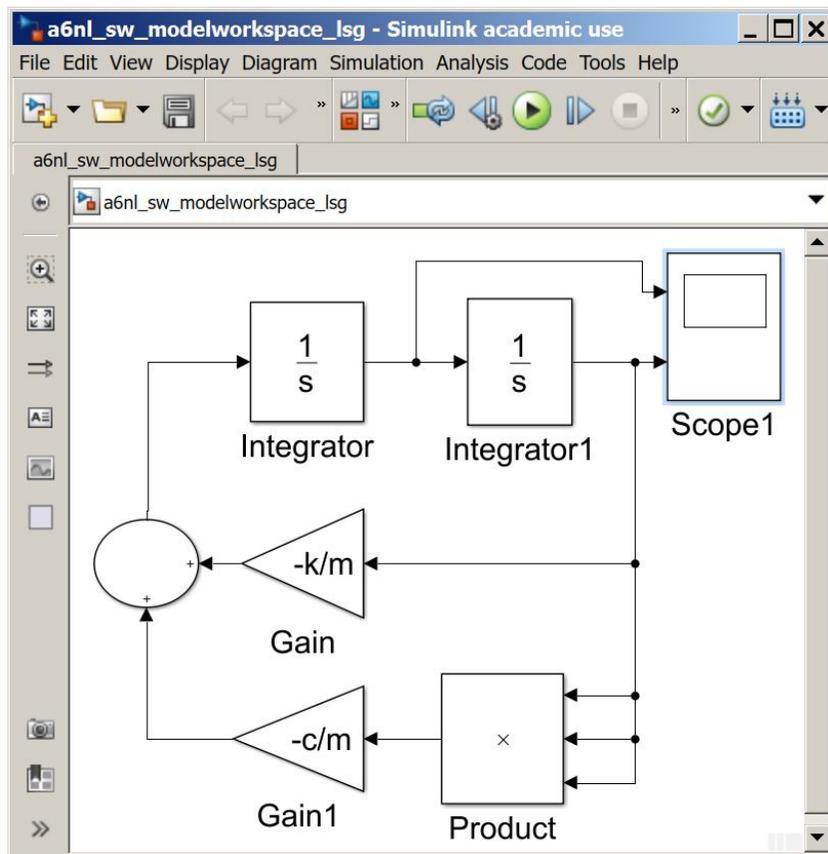
P6_14 6.6 Model-Workspace

Aufgabe 6:

Verwenden Sie für die Parameter der nichtlinearen DGL drei **Variablen** k , m und c , die im **Model-Workspace** (nicht MATLAB-Workspace) definiert werden. Die Anfangsbedingungen werden über zwei Variablen $v0$ und $y0$ definiert. Öffnen Sie das Modell **a6nl_sw_modelworkspace.slx** und ändern Sie dieses geeignet ab. Starten Sie dann die Simulation.

$$\frac{d^2y}{dt^2} + \frac{k}{m} * y(t) + \frac{c}{m} * y(t)^3 = 0$$

$$\dot{y}(t = 0) = v0 \quad , \quad y(t = 0) = y0$$



Name	Value	Data Type
c	4	double (...)
k	1	double (...)
m	2	double (...)
v0	0	double (...)
y0	3	double (...)

P6_15 6.7 Nichtlineare gedämpfte Schwingung und äußerer Kraft

Aufgabe 7 :

Lösen Sie mit Hilfe von Simulink das Anfangswertproblem für folgende DGL:

$$\frac{d^2y}{dt^2} + 2 \cdot \delta \cdot \frac{dy}{dt} + \frac{k}{m} \cdot y(t) + \frac{c}{m} \cdot y(t)^3 = A \cdot e^{-\lambda \cdot t}$$

$$v_0 = \dot{y}(t=0) = 0, \quad y_0 = y(t=0) = 2$$

d.h. für eine nichtlineare gedämpfte Schwingung mit einer exponentiell abnehmenden äußeren Kraft. Ergänzen Sie das Modell `a7n1_sw_ged_ext.slx` um die notwendigen Blöcke. Definieren Sie die fehlenden Parameter im Modell-Workspace.

Hinweis : Verwenden Sie einen MUX-Block mit drei Eingängen für

- die Geschwindigkeit
- die Auslenkung
- die Zeit t

Den Ausgang des MUX-Blockes führen Sie in einen Function-Block der folgenden Ausdruck berechnet :

$$-2 \cdot \delta \cdot \frac{dy}{dt} - \frac{k}{m} \cdot y(t) - \frac{c}{m} \cdot y(t)^3 + A \cdot e^{-\lambda \cdot t}$$

Die Zeit t kann einfach mit Hilfe des Blocks **Clock** (siehe Simulink -> Sources) geholt werden.

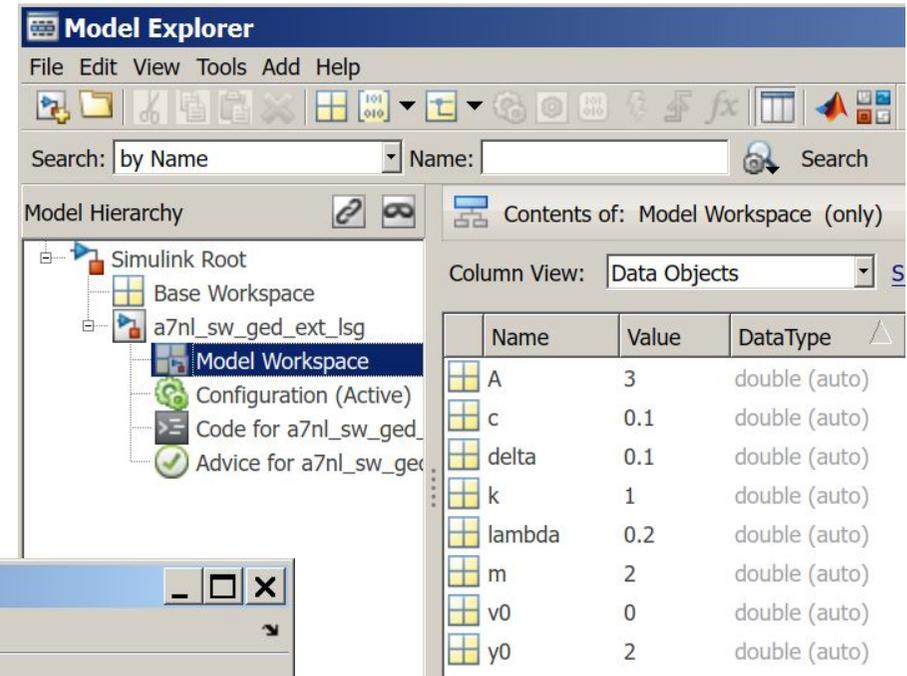


P6_16 6.7 Nichtlineare gedämpfte Schwingung und äußerer Kraft

Wählen Sie die Parameter wie rechts gezeigt. Für die Auslenkung ergibt sich dann das unten gezeigte Bild. Versuchen Sie die Kurve zu erklären.

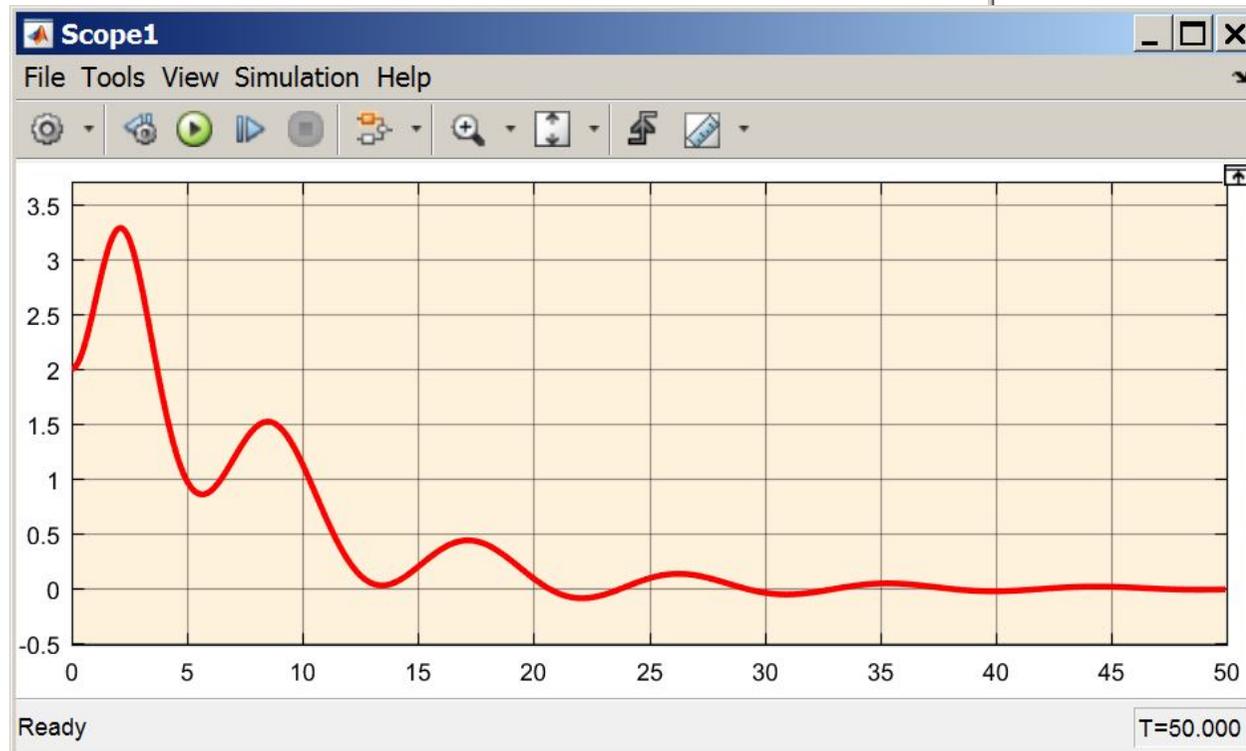
Was passiert, wenn man den Wert von λ verkleinert oder vergrößert?

Was passiert, wenn man den Wert von A verkleinert oder vergrößert?



The Model Explorer window displays the following parameters in the Model Workspace:

Name	Value	Data Type
A	3	double (auto)
c	0.1	double (auto)
delta	0.1	double (auto)
k	1	double (auto)
lambda	0.2	double (auto)
m	2	double (auto)
v0	0	double (auto)
y0	2	double (auto)



Aufgabe 8:

Erstellen Sie ein Simulink-Modell für folgende DGL :

$$\ddot{x} + d \cdot \operatorname{sgn}(\dot{x}) \cdot \dot{x}^2 + \omega_0^2 \cdot x = \sin(\omega \cdot t)$$

$$\dot{x}(t = 0) = 0 \quad , \quad x(t = 0) = 0$$

Stellen Sie die Auslenkung und die Anregung als Funktion der Zeit dar.

Hierbei handelt es sich um eine angeregte Schwingung, bei der die **Dämpfung proportional zum Quadrat der Geschwindigkeit ist**.

Wählen Sie : $\omega_0^2 = 5$, $\omega = 1$ und $d = 0.4$. Definieren Sie alle Parameter im Modell-Workspace.

Die Signum-Funktion (Vorzeichenfunktion) ist erforderlich, damit die Dämpfungskraft stets der Bewegung entgegen wirkt. Für die Signum-Funktion gibt es bereits einen fertigen Simulink-Block. Für die Signum-Funktion gilt :

$$\operatorname{sgn}(x) = \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{falls } x = 0 \\ -1 & \text{falls } x < 0 \end{cases}$$

Siehe auch :

Scherf Modellbildung und Simulation dynamischer Systeme
Oldenbourg Verlag, 4. Auflage, 2010