



Technische **Hochschule**
Ingolstadt

Fakultät für Elektrotechnik
und Informatik

Skriptum zur Vorlesung

Modellbildung und Simulation in MATLAB-SIMULINK

Prof. Dr.-Ing. Steffen Lehner

Fakultät Elektrotechnik und Informatik

Sommersemester 2018

Vorwort

Das vorliegende Skriptum wird ergänzend zu den Vorlesungen *Angewandte Methematik* sowie *Modellbildung und Simulation technischer und dynamischer Systeme* an der Technischen Hochschule Ingolstadt angeboten.

Es ersetzt weder die persönliche Teilnahme an Vorlesungen und Übungen, noch eine adäquate Mitschrift dieser.

Hauptschwerpunkte sind:

Matlab/Simulink

- Erstellen von Skripten zum Lösen physikalischer Probleme
- Erstellen von Modellen in Simulink und deren Steuerung mittels Skripten
- Numerische Verfahren (Runge-Kutta, Euler etc.)
- Möglichkeiten der Validierung und Verifizierung
- Strukturierung

Angestrebte Ziele:

- Kenntnis von Matlab/Simulink
- Die Studierenden wissen, wie man ein Differentialgleichungssystem (eventuell auch ein nichtlineares) in einem Simulationsmodell formuliert
- Verständnis des numerischen Lösungsverfahrens und der damit verbundenen Einstellmöglichkeiten (Schrittweite, Genauigkeit, Integrationsverfahren)
- Befähigung, geeignete Einstellungen für ein entworfenes Simulationsmodell zu wählen
- Die Studenten sind in der Lage, die Simulationsmodelle sinnvoll zu strukturieren
- Sie sind in der Lage, die Modelle zu verifizieren und zu validieren

Empfehlung:

Sie sollten auch zur Vorlesung stets ein Notebook mit MatLab mitbringen.

THI besitzt über einen Total Academic Headcount (TAH) Lizzenzen für MATLAB, Simulink und ergänzende Toolboxen, die Ihnen kostenlos zur Verfügung stehen.

<https://www.thi.de/service/it-service/angebote-fuer-studenten/matlab/>

Quellen:

Rill/Borchsenius: Skript Ingenieur-Informatik II – Basis für dieses Skriptum

Wolfgang Schweizer: Matlab-Kompakt ISBN 978-3-486-59193-4

Angermann/Beuschel/Rau/Wohlfarth: Matlab-Simulink.Stateflow ISBN 978-3-486-58985-6

Merziger, Wirth: Repetitorium der Höheren Mathematik; ISBN 3-923-923-33-3

Holzmann/Meyer/Schumpich: Technische Mechanik; ISBN 3-519-16520-1

Lothar Papula: Mathematik für Ingenieure Band II; ISBN 978-3-8348-1589-7

<http://www.wikipedia.org>

Inhaltsverzeichnis

1	Einführung in MATLAB.....	5
1.1	Allgemeines	5
1.2	MATLAB Oberfläche und Hilfe.....	6
1.3	Allgemeine Befehle.....	7
1.4	Variablenzuweisungen	8
1.4.1	Zahlen	8
1.4.2	Vektoren und Matrizen	8
1.4.3	Zeichenketten.....	12
1.4.4	Zusammengesetzte Variablen	13
1.5	Operatoren	15
1.5.1	Addition und Subtraktion	17
1.5.2	Multiplikation.....	17
1.5.3	Division.....	17
1.6	Konstruktionen zur Programmsteuerung.....	18
1.6.1	if-Anweisung	19
1.6.2	for-Schleife.....	19
1.6.3	while-Schleifen.....	19
1.6.4	break-Anweisung	21
1.7	Skripte und Funktionen	21
1.7.1	Skripte	21
1.7.2	Funktionen.....	21
1.8	Elementare Plot-Befehle.....	22
1.8.1	3D-Plots	24
2	Lineare Gleichungssysteme.....	27
2.1	Allgemeines	27
2.2	Vorüberlegungen	27
2.3	Quadratische Koeffizientenmatrizen	29
2.3.1	Beispiel Fachwerk.....	29
2.3.2	Wärmeleitung in einer Platte.....	32
2.4	Rechteckige Koeffizientenmatrizen	37
2.4.1	Beispiel Ausgleichsgerade.....	37
2.4.2	Ausgleichspolynome	38
2.4.3	Gebrochen Rationale Funktionen.....	41
2.4.4	Übertragung auf beliebige Ansatzfunktionen	44

2.4.5	Mehrdimensionale Funktionen	44
3	Nichtlineare Probleme	47
3.1	Optimierung	47
3.1.1	Allgemeines.....	47
3.1.2	Optimales Fachwerk	47
3.1.3	Gleichgewichtslage.....	51
3.2	Nichtlineare Gleichungen.....	54
3.2.1	Beispiel: Kreuzprofil	54
3.2.2	Indirekte Lösung.....	56
4	Dynamische Probleme.....	59
4.1	Allgemeines	59
4.1.1	Typen	59
4.2	Numerische Integrationsverfahren.....	60
4.2.1	Integrationsverfahren: Treppenfunktion	60
4.2.2	Integrationsverfahren: Trapez-Verfahren	61
4.2.3	Integrationsverfahren nach Simpson	62
4.2.4	Integrationsverfahren nach Euler (explizit).....	63
4.2.5	Lösungsverfahren in MATLAB	65
4.2.6	Klassisches Runge-Kutta-Verfahren	65
4.2.7	Autonome Systeme.....	69
4.3	Beispiele	70
4.3.1	Räuber-Beute-Modell	70
4.3.2	Federpendel	72
4.3.3	Druckregelung	76
5	Simulink	80
5.1	Simulink-Bibliothek	80
5.2	Simulink-Modelle via Script-Steuerung.....	82
5.2.1	Gleichstrommotor in Simulink.....	82
5.3	Subsysteme.....	87
5.3.1	Virtuelle und Nichtvirtuelle Subsysteme	87
5.4	Scheduling in Simulink	88
5.4.1	Scheduling Multiple Subsystems in a Single Time Step	88
6	Praxisanwendungen.....	92
6.1.1	Lesen von Textdateien	92

1 Einführung in MATLAB

1.1 Allgemeines

MATLAB ist eine Hoch-Leistungs-Sprache für numerische Berechnungen. MATLAB integriert Berechnung, Visualisierung und Programmierung in einer leicht zu benützenden Umgebung (graphische Benutzer-Oberfläche, GUI). Ende der siebziger Jahre wurde MATLAB speziell für Matrix-/Vektoroperationen entwickelt, woraus sich auch der Name **MAT**rix**L**ABoratory ableitet.

In der heutigen Zeit hat sich MATLAB in vielen Bereichen der Industrie als Standardwerkzeug etabliert. Beispiele sind:

- Simulationen, z.B. hydraulischer, mechanischer, elektrotechnischer Komponenten
- Entwicklung von Algorithmen zur Signalanalyse
- Verarbeitung von Messdaten mittels Skripten, insbesondere bei großer Anzahl von Messdateien oder auch deren komplexem Aufbau
- Entwicklungsgrundlage für Regelungstechnische Modelle (meist in Verbindung mit der MATLAB-Toolbox SIMULINK)
- Prototyping, Hardware-In-The-Loop Entwicklungsumgebung
- Visualisierung von Daten, z.B. Simulationsergebnissen oder Messergebnissen
- Entwicklung von Anwendungen, einschließlich graphischer Benutzeroberflächen

In MATLAB können fast alle Programmieraufgaben direkt umgesetzt werden. Viele Hilfsfunktionen erleichtern das Erlernen im Vergleich zu Compilersprachen (z.B. C, C++, FORTRAN etc.) erheblich und man erhält einfachsten Zugriff auf komplexe Funktionen.

MATLAB ist nicht nur auf Basis-Algorithmen der numerischen linearen Algebra beschränkt. Mittels sogenannter *Toolboxen* lässt sich MATLAB um eine Vielzahl anwendungsspezifischer Lösungsverfahren erweitern. *Toolboxen* sind Sammlungen von MATLAB-Funktionen (M-Files). Gebiete für die es Toolboxen gibt sind z.B. Signalverarbeitung, Regelungstechnik, Neuronale Netzwerke, 'fuzzy logic', Wavelets, Simulation und viele andere.

Eine weitere Stärke ist die hervorragend ausgearbeitete Hilfe-Funktion.

MATLAB enthält keine direkten Befehle zur Dimensionierung. Vektoren und Matrizen können somit jederzeit in einem MATLAB-Programm verwendet werden. Die entsprechenden Speicherplätze werden von MATLAB im Rahmen der Rechnerkapazität automatisch belegt.

Achtung: MATLAB unterscheidet Groß- und Kleinschreibung!

Die meisten Beispiele in diesem Skriptum wurden in der MATLAB-Version 7.13 erstellt, funktionieren jedoch auch in neueren Versionen. Einige Bilder sind bereits aus Version 8.1.0.604 (R2013a).

1.2 MATLAB Oberfläche und Hilfe

Nach dem Starten befindet sich mittig in der MATLAB Oberfläche das *Command-Fenster* (Befehlseingabe), links oben das *Current Folder* Fenster, links unten Details zum angewählten Objekt, rechts oben das *Workspace* Fenster (vorhandene Größen) und rechts unten die *Command History*.

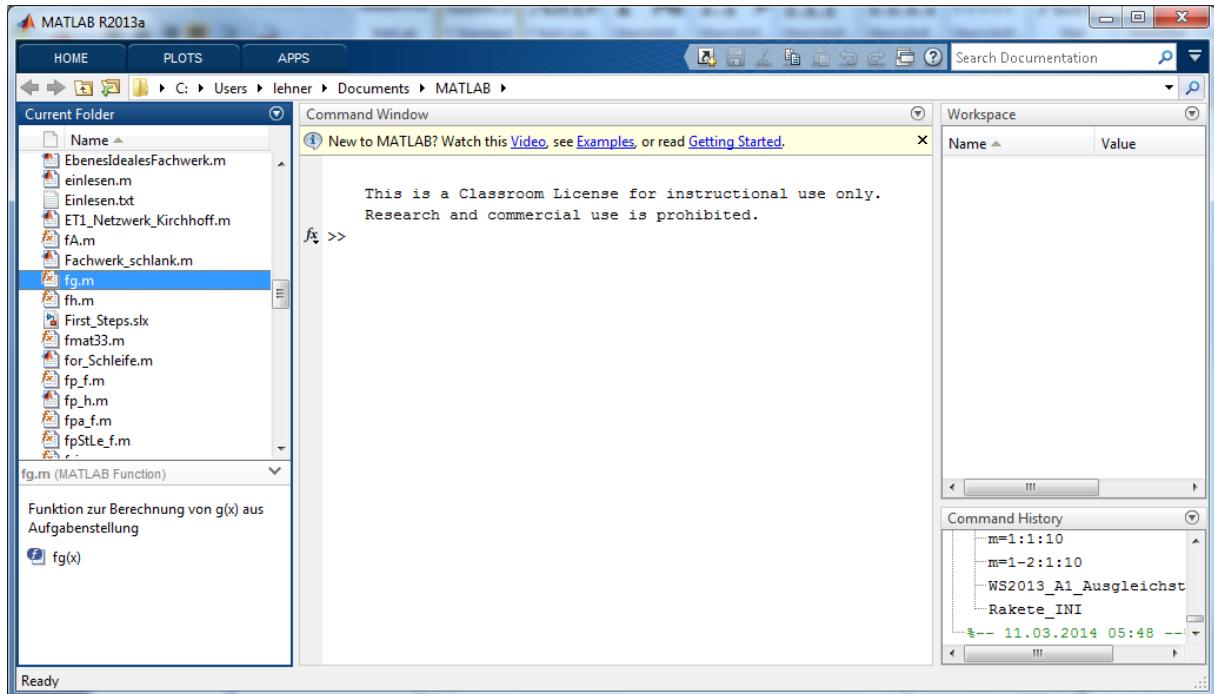


Abbildung 1-1: MATLAB Desktop

Über das Icon ? lässt sich ein Hilfe-Fenster öffnen. Alternativ erhält man mittels dem – durch die Return-Taste übergebenen – Befehl

>> help

alle primären Topics aufgelistet oder durch

>> help *functionname*

Hilfe zur Funktion ***functionname***

oder

>> help *topicname*

eine Liste der unter der Rubrik ***topicname*** verfügbaren Subtopics und Befehle.

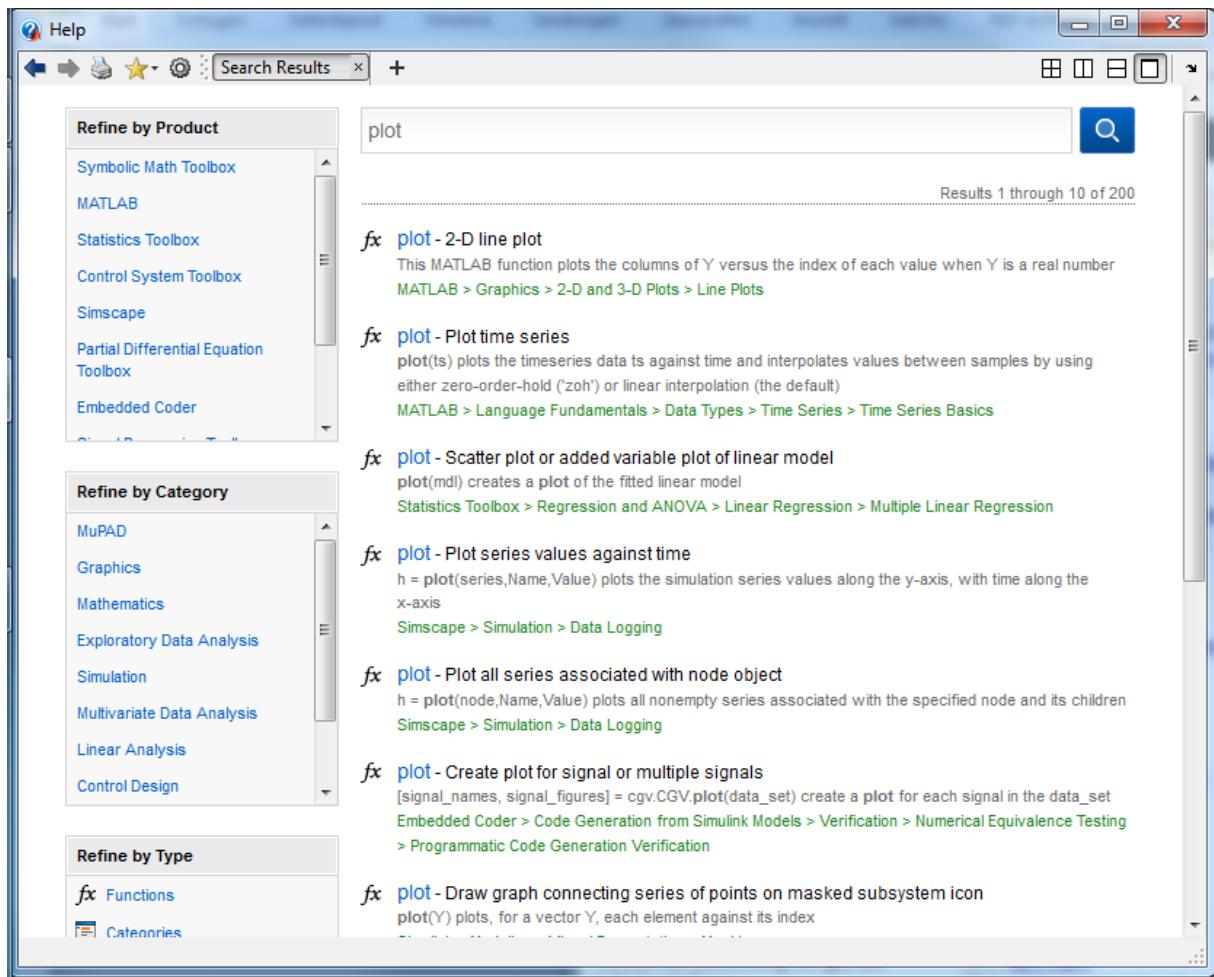


Abbildung 1-2: MATLAB *Help Documentation* bei Hilfesuche nach **plot**

In den Hilfe-Topics finden sich neben Erklärungen teils auch Beispiele zu MATLAB -Befehlen und -Funktionen.

In den folgenden Kapiteln werden die wichtigsten MATLAB -Befehle kurz erläutert. Im Text findet der Leser weitere MATLAB -Befehle anhand von Anwendungsbeispielen erklärt.

1.3 Allgemeine Befehle

Allgemeine MATLAB -Befehle sind im Topic **general** zusammengefasst. Die folgende Tabelle enthält einen Auszug:

Befehl	Bezeichnung	Beispiel
(whos) ,	(ausführliche) Liste, der gerade verwendeten Va-	--
who	riablen	
cd	Änderung des aktuellen Arbeitsverzeichnisses	cd c:\
clear	Löscht den Workspace (Memory)	
save	Speichert alle Workspace-Variablen	save my_ws.mat

1.4 Variablenzuweisungen

In MATLAB können Zahlen, Zeichenketten (Strings), Datenfelder und auch komplexe Datenstrukturen (sog. Structs) einfach über eine Wertzuweisung mit dem Gleichheitszeichen als Operator angelegt werden. Die Daten werden im MATLAB - *Workspace* gespeichert. Sie liegen im RAM und sind somit flüchtig!

1.4.1 Zahlen

Zahlen lassen sich mit direkter Wertzuweisung, wie etwa

```
a = 2.5;
```

in Variablen speichern. Anschließend kann eine solche Variable wie eine Zahl verwendet werden, z.B.

```
b = 3*a;
```

speichert in der neu angelegten Variablen **b** den Wert **b = 7.5**. Ebenso sind Zuweisungen der Form

```
b = b - 1;
```

möglich. Dadurch wird der momentan gespeicherte Wert in **b** um eins verringert.

1.4.1.1 Zahldarstellung

In MATLAB werden Zahlen stets als doppelt genaue reelle (bzw. komplexe) Zahlen gemäß dem *IEEE Standard Binary Floating Point Arithmetic* dargestellt:

Insgesamt stehen 8 Byte (64 Bit) zur Verfügung, wobei 1 Bit für das Vorzeichen verwendet wird, 11 Bit für den Exponenten und 52 Bit für die Mantisse.

Die kleinste darstellbare Zahl ist somit: **realmin** $\approx 2.2251 \cdot 10^{-308}$

und die größte darstellbare Zahl: **realmax** $\approx 1.7977 \cdot 10^{308}$.

Die Genauigkeit der Rundung **rd(x)** einer reellen Zahl **x** ist durch die Konstante

$$\text{eps gegeben: } \left| \frac{x - rd(x)}{x} \right| \leq \text{eps} = 2^{-52} \approx 2.2204 \cdot 10^{-16}$$

Damit ergibt sich als Faustregel eine Genauigkeit von 16 Dezimalstellen. Exponentenunterlauf bzw. -überlauf erfolgt etwa ab 10^{-308} bzw. 10^{308} .

Arithmetische Katastrophen: Eine Division durch 0 führt auf das Ergebnis Inf (für *infinity*), und eine Division von 0 durch 0 auf NaN (für *Not a Number*).

1.4.2 Vektoren und Matrizen

Vektoren und Matrizen sind Datenfelder. Ein Vektor der Dimension **n** enthält **n** Zahlen und eine Matrix der Dimension **n*m** entsprechend **n*m** Zahlen.

Häufig ist die explizite Wertzuweisung der einfachste Weg, die Elemente von Vektoren und/oder Matrizen mit Werten zu belegen.

Die MATLAB -Befehle

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

 (1.4.1)

oder

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

 (1.4.2)

definieren eine 3×3 -Matrix und belegen sie mit Zahlenwerten.

Matrizen werden in eckigen Klammern angegeben. Die Elemente einer Zeile werden durch Leerstellen oder Kommas getrennt. Die Zeilenenden werden durch Strichpunkte (;) angegeben.

Seit der MATLAB-version 11 können Matritzen über mehrere Zeilen fortgesetzt werden ohne die normalerweise notwendigen drei Punkte zur Fortsetzung eines Befehls anzuwenden: Anstelle von (1.4.1) schreibt man:

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

 (1.4.3)

Die Befehle (1.4.1) bis (1.4.3) quittiert MATLAB mit

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

 (1.4.4)

Ist die Ausgabe von Zwischenergebnissen nicht erwünscht, so kann sie durch Setzen eines Strichpunkts am Zeilenende unterdrückt werden. Insbesondere in längeren Skripten kann die ersparte Ausgabe zu einer merklichen Laufzeitverringerung führen.

Die Eingaben

```
A = [ 1 2 ; 3 4 ] ;
B = [ 5 6 ; 7 8 ] ;
C = A + B
```

 (1.4.5)

quittiert MATLAB nur mit dem Endergebnis

```
C =
6      8
10     12
```

 (1.4.6)

Achtung: Die Bedeutung vom Strichpunkt innerhalb und außerhalb von eckigen Klammern ist unterschiedlich!

Mehrere MATLAB -Befehle können in einer Zeile geschrieben werden, indem die einzelnen Befehle durch Kommas oder Strichpunkte voneinander getrennt werden. Wählt man Strichpunkte als Trennzeichen, so werden auch hier die Zwischenergebnisse unterdrückt. Für die Anweisungen aus (2.4.5) schreibt man dazu

```
A = [ 1 2 ; 3 4 ] ; B = [ 5 6 ; 7 8 ] ; C = A + B
```

 (1.4.7)

Wählt man hingegen Kommas als Trennzeichen, also

```
A = [ 1 2 ; 3 4 ] , B = [ 5 6 ; 7 8 ] , C = A + B
```

 (1.4.8)

dann gibt MATLAB auch alle drei Ergebnisse aus.

1.4.2.1 Zeilen- und Spaltenvektoren

Matrix/Vektor und Vektor/Vektor Operationen erfordern die Unterscheidung zwischen Zeilen- und Spaltenvektoren. In MATLAB definiert

$$\mathbf{r} = [1 \ 2 \ 3] \quad \text{oder} \quad \mathbf{r} = [1, 2, 3] \quad (1.4.9)$$

einen Zeilen- und

$$\begin{aligned} \mathbf{c} &= [1; \dots \\ &2; \dots \\ &3] \end{aligned} \quad \text{oder} \quad \mathbf{c} = [1; 2; 3] \quad (1.4.10)$$

einen Spaltenvektor.

Die Zeilen und Spalten einer Matrix lassen sich in MATLAB durch Anfügen eines Apostroph (') vertauschen (transponieren). Mit

$$\mathbf{c} = \mathbf{r}' \quad (1.4.11)$$

wird das Ergebnis der Transposition vom Zeilenvektor \mathbf{r} der Variablen \mathbf{c} zugewiesen.

1.4.2.2 Einzelne Elemente und Teilmatrizen

Matrixelementen lassen sich direkt Werte zuweisen. Die Eingaben

$$\mathbf{A} = [1 \ 2 ; 3 \ 4] ; \mathbf{A}(3, 3) = 7 \quad (1.4.12)$$

quittiert MATLAB mit

$$\begin{aligned} \mathbf{A} = \\ \begin{matrix} 1 & 2 & 0 \\ 3 & 4 & 0 \\ 0 & 0 & 7 \end{matrix} \end{aligned} \quad (1.4.13)$$

Neben der Wertzuweisung $\mathbf{A}(3, 3)=7$ hat MATLAB automatisch die Dimension der Matrix angepasst und noch nicht definierte Elemente mit Null vorbelegt.

Mittels Doppelpunkt können Teile einer Matrix extrahiert werden. Mit dem Befehl

$$\mathbf{r2} = \mathbf{A}(2, :) \quad (1.4.14)$$

wird die 2. Zeile der Matrix \mathbf{A} (1.4.13) in $\mathbf{r2}$ abgelegt und ausgegeben und mit

$$\mathbf{c1} = \mathbf{A}(:, 1) \quad (1.4.15)$$

die 1. Spalte aus der Matrix \mathbf{A} der Variablen $\mathbf{c1}$ zugewiesen. Der Doppelpunkt steht dabei als Platzhalter für alle Zeilen, bzw. alle Spalten einer Matrix.

Die Befehle (1.4.14) und (1.4.15) dienen dazu, Werte von Zeilen oder Spalten einer Matrize zu extrahieren. Leicht abgewandelt lassen sie auch das Beschreiben dieser zu. Beispielsweise beschreiben die Befehle

$$\mathbf{A}(2, :) = [1 \ 2 \ 3] \quad \text{und} \quad \mathbf{A}(:, 2) = [1 \ 2 \ 3] \quad (1.4.16)$$

die 2. Zeile der Matrix **A** mit dem Vektor **[1 2 3]** bzw. **[1 ; 2 ; 3]**. Dabei spielt es keine Rolle, ob es sich bei dem Quellvektor um einen Zeilen oder Spaltenvektor handelt. Nur die Anzahl der Quellelemente muss mit der Anzahl der Zielelemente übereinstimmen.

Aus der Matrix **A** können mit dem Befehl

T = A (1 : 2 , 2 : 3) (1.4.17)

Teilmatrizen gewonnen werden. Dabei wird der Bereich in dem sich die Zeilen 1 bis 2 mit den Spalten 2 bis 3 überdecken der Variablen **T** zugewiesen.

1.4.2.3 Löschen und Vorbelegen

Der MATLAB -Befehl

clear all (1.4.18)

löscht sämtliche Variablen, Skalare, Vektoren, Matrizen und Funktionen aus dem Speicher, womit praktisch ohne Neustart eine neue MATLAB -Sitzung beginnt.

Möchte man selektiv einzelne Variablen oder Funktionen löschen, so ist dem **clear**-Befehl eine entsprechende Liste anzuhängen, beispielsweise

clear v1 v2 v3 f1 f2 (1.4.19)

Normalerweise ist eine Vorbelegung von Variablen in MATLAB nicht erforderlich. In einigen Fällen ist es dennoch zweckmäßig, die Elemente einer **n×m**-Matrix durch

M = zeros (n , m) (1.4.20)

mit Nullen zu belegen. Neben der Vorbelegung wird dabei nämlich die Matrix in der gewünschten Dimension (n Zeilen und m Spalten) erzeugt.

Ein großer Vorteil der Vorbelegung bzw. Vordimensionierung ist die teils deutliche Erhöhung der Rechengeschwindigkeit, gegenüber einer häufigen Umdimensionierung bei Schleifendurchläufen.

Analog zu (1.4.20) werden durch den Befehl

M = ones (n , m) (1.4.21)

alle Elemente einer **m×n**-Matrix mit **1** vorbelegt. Mit dem Befehl

E = eye (n) (1.4.22)

wird eine **n×n**-Einheitsmatrix erzeugt.

Der MATLAB -Befehl

A = diag([1 2 3]) (1.4.23)

erzeugt eine quadratische Matrix **A** (überschreibt **A**), belegt deren Hauptdiagonale mit den Elementen des Vektors **v = [1 2 3]** und die Nebendiagonalelemente mit Null.

In umgekehrter Richtung lassen sich mittels **diag**-Befehl einer Variablen **d** die extrahierten Elemente der Hauptdiagonalen der Matrix **M** zuweisen.

d = diag(M) (1.4.24)

Ist die Matrix **M** nicht quadratisch, so enthält **d** die Elemente der Diagonalen, welche mit dem oberen linken Element, also **M(1)** beginnt.

1.4.3 Zeichenketten

Das Handling von Messdateien erfordert i.d.R. die Auswertung von Zeichenketten in Dateinamen, was die Auswertung umfangreicher Messkampagnen erst ermöglicht. Zudem speichern viele Messgeräte neben Messdaten nützliche Informationen wie Einheiten oder Kommentare. Um diese Informationen auswerten zu können, stehen in MATLAB mächtige Funktionen zur Verfügung.

Eine Zeichenkette *String* ist eine Kette von Zeichen *Characters*, angegeben innerhalb sog. *String Quotes*. Jedes Zeichen wird intern durch eine Zahl dargestellt. Beispiel:

```
>> txt = 'Ich lerne MATLAB'
txt =
Ich lerne MATLAB

>> numbers = double(txt)
numbers =
Columns 1 through 10
 73    99    104    32    108    101    114    110    101    32
Columns 11 through 16
 77    65    84    76    65    66

>> characters = char(numbers)
characters =
Ich lerne MATLAB

>> whos txt numbers characters
  Name      Size            Bytes  Class       Attributes
  characters    1x16             32  char
  numbers        1x16            128  double
  txt            1x16             32  char
```

Mit Hilfe einer Vielzahl von String-Funktionen können Strings beispielsweise verglichen, Teile gelöscht oder verschiedene Strings miteinander verkettet werden. Strings lassen sich ebenso wie Skalare und Vektoren direkt Variablen zuweisen:

```
name = 'Bert';
```

Auf einzelne Elemente von Strings, kann im Speicher mit denselben Zugriffsoperationen wie bei Vektoren zugegriffen bzw. diese verändert werden.

```
name(1)
```

greift auf das erste Element von **name** zu und gibt **B** zurück. Dem in der Klammer indizierten Element kann auch direkt ein Wert zugewiesen werden, z.B.

```
name(1) = 'W';
```

überschreibt den ersten Buchstaben, so dass in **name** jetzt der Wert **Wert** gespeichert ist.

Mit der Funktion **strcmp** kann man prüfen, ob zwei Strings identisch sind.

Die Funktion **strfind** kann verwendet werden, um zu prüfen, ob ein String (Pattern) in einem andern String (Text) enthalten ist. Der Rückgabewert ist die Position des Pattern-Strings im gesamten Text-String.

```
name = 'Rolf Muster';
if strfind(name,'Muster')
disp('Ciao Herr Muster');
end
```

Hier liefert die **strfind**-Funktion als Rückgabewert die Zahl **6**, da **Muster** sich an dieser Stelle des Strings **Rolf Muster** befindet. Kommt das Pattern mehrfach vor, liefert die **strfind**-Funktion einen Vektor. Die Länge des Vektors entspricht der Häufigkeit, mit der das Pattern im Text vorkommt. Die einzelnen Werte des Vektors enthalten die jeweiligen Positionen.

Wenn ein String gesucht wird, der nicht enthalten ist, so gibt die Funktion **strfind** ein leeres Feld **[]** zurück.

Weil Strings genau wie Vektoren eindimensionale Arrays sind, nur dass ihre Elemente als Buchstaben interpretiert werden, lassen sie sich wie Vektoren zusammensetzen. Die Vektoren **e1** und **e2**

```
e1 = [1 0 0];
e2 = [0 1 0];
e = [e1 e2];
```

liefern den zusammengesetzten Vektor **e = [1 0 0 0 1 0]**. In gleicher Weise lassen sich mehrere Strings aneinanderhängen, z.B.

```
vorname = 'Rolf';
nachname = 'Muster';
name = [vorname ' ' nachname];
```

Die beiden mittigen Apostrophe sorgen für ein Leerzeichen zwischen dem Vor- und dem Nachnamen. Die String-Variablen **name** speichert nun den Wert **Rolf Muster**.

Mit dem Befehl **length** ermittelt man die Länge eines Strings.

1.4.4 Zusammengesetzte Variablen

In MATLAB lassen sich Variablen unterschiedlichen Typs in **struct**-Arrays und **cell**-Arrays strukturieren.

1.4.4.1 Struct - Arrays

Daten unterschiedlichen Typs (z.B. *char*, *double*) lassen sich in sog. **struct**-Arrays zusammenfassen. Der Zugriff auf Elemente, sog. Felder *fields* solcher **struct**-Variablen erfolgt mit dem **.**-Operator und Feldnamen.

```
>> person.first = 'Fritz';
>> person.name = 'Schulz';
>> person.numb = 21;
```

Hier wird die *struct*-Variable **person** angelegt, die weitere Datenobjekte (*fields*) enthält, nämlich **first**, **name** und **numb**. Während **first** und **name** Zeichenketten sind, enthält **numb** eine Zahl.

Das Lesen von *fields* erfolgt analog. Beispielsweise liest

```
>> nr = person.numb;
```

numb aus **person** und speichert den Wert in einer gewöhnlichen Variablen **nr**, die in diesem Beispiel nun die Zahl **21** enthält.

Obige Struktur lässt sich erweitern, indem beispielsweise ein zweites Mitglied durch

```
>> person(2)=struct('first','Max','name',...
    'Meier','numb',12);
```

angefügt wird. Man erhält mit der Abfrage

```
>> person
person =
1x2 struct array of fields:
    first
    name
    numb
```

Informationen zur Struktur und mit

```
>> person(2)
ans =
    first:      'Max'
    name:      'Meier'
    numb:      12
```

Informationen zum 2. Mitglied.

Sollen die Mitglieder eine zusätzliche Eigenschaft erhalten, beispielsweise die Straße, so adressiert man einfach ein neues Feld.

```
>> person(2).street = 'Tal-'
```

Die angefügten Felder der anderen Mitglieder sind dann noch leer.

```
>> person(1).street
ans =
[]
```

Ein gelegentlicher Nachteil von *struct*-Arrays ist, dass die Namen der Felder (Elemente) grundsätzlich Strings sind und man zu ihrer Adressierung keine Zahlen nutzen kann. Dieses Problem wird mittels *cell arrays* gelöst.

1.4.4.2 Cell Array -Strukturen

Ebenso wie *struct*-Arrays, können *cell*-Arrays Daten unterschiedlichen Typs in einem ein- oder mehrdimensionalen Array beinhalten.

```
>> c = {'First';'Name';'Ort'}
c =
    'First'
```

```
'Name'
'Ort'
```

Adressiert werden die *Cells* in gleicher Weise wie die Elemente in Matrizen und die Arrays lassen sich auch in der bekannten Weise erweitern:

```
>> c(1:4,2) = {'Karin';'Schmidt';'Ingolstadt';[1 2 3]}
c =
    'First'      'Karin'
    'Name'       'Schmidt'
    'Ort'        'Ingolstadt'
    []           [1x3 double]
```

Auf das Objekt einer *Cell* wird mittels runder Klammern zugegriffen

```
>> c(4,2)
ans =
    [1x3 double]
```

und auf den Inhalt des Objektes mittels geschweifter Klammern.

```
>> c{4,2}
ans =
    1     2     3
```

1.5 Operatoren

Mit dem MATLAB-Befehl **help ops** wird Hilfe zu allen verfügbaren Operatoren aufgelistet. Die gebräuchlichsten sind:

Operator	Bezeichnung	Beispiel
=	Wertzuweisung	x = -5
+	Plus	x = a + b
-	Minus	x = a - b
*	gewöhnliche Multiplikation	x = a*b
.*	elementweise Multiplikation	A = B.*C
^	gewöhnliche Exponentiation	x = a^3
.^	elementweise Exponentiation	A = B.^2
/	gewöhnliche Division	x = a/b
./	elementweise Division	A = B./C
\	Linksdivision	x = A\B
.\ .\	elementweise Linksdivision	x = A.\B

Bedingte Anweisungen werden mit Vergleichsoperatoren

Operator	Bezeichnung	Beispiel
<code>==</code>	gleich	<code>if x==1; y=2; end</code>
<code>~=</code>	ungleich	<code>if x~=1; y=2; end</code>
<code><</code>	kleiner	<code>if x<1; y=2; end</code>
<code><=</code>	kleiner gleich	<code>if x<=1; y=2; end</code>
<code>></code>	größer	<code>if x>1; y=2; end</code>
<code>>=</code>	größer gleich	<code>if x>=1; y=2; end</code>

durchgeführt und können mit den Operatoren

Operator	Bezeichnung	Beispiel
<code>&</code>	und	<code>if x>1 & x<4; y=2; end</code>
<code> </code>	oder	<code>if x==1 x==2; y=3; end</code>

verknüpft werden. Vergleichsoperatoren sind höherwertig als Verknüpfungsoperatoren. Diese wiederum sind höherwertig als Punkt- und dann Strich- Operatoren

Beispielsweise liefert

```
>> a = 1 & 2 > 1                                (1.5.1)
a =
    1 ,
```

während

```
>> a = 1 & 2 == 1                                (1.5.2)
a =
    0
```

als Ergebnis ausgibt. Das jeweilige Vergleichsergebnis wird erst anschließend logisch verknüpft.

Folgende Zeichen haben eine besondere Bedeutung:

Operator	Bezeichnung	Beispiel
<code>()</code>	Funktionsklammern oder Vektor- oder Matrixelemente	<code>x=sqrt(y)</code> <code>x(2)=A(3,4)</code>
<code>:</code>	Platzhalter in Vektoren und Matrizen oder Trennzeichen in Laufanweisungen	<code>A(:,1)=x</code> <code>for i=1:5;x(i)=i;end</code>
<code>[]</code>	Vektor- oder Matrixklammern	<code>A = [1 2 ; 3 4]</code>
<code>.</code>	Dezimalpunkt	<code>x=3.10</code>
<code>...</code>	mehrzeilige Anweisungen	<code>A = [1 2 ; ...</code> <code>3 4]</code>
<code>,</code>	Trennzeichen (Anweisungen und Spalten)	<code>x=1, y=2</code>
<code>;</code>	Trennzeichen (Anweisungen und Zeilen)	<code>x=1; y=2</code>

%	Kennzeichnung eines Kommentars	a=1 % mm
!	Öffnen eines DOS-Fensters	
'	Vertausche Zeilen und Spalten (transponiert)	x=y'

1.5.1 Addition und Subtraktion

Sind die Variablen **A** und **B** beides Skalare oder **A** ein Skalar und **B** eine Matrix (oder umgekehrt), so lassen sich

$$\mathbf{S} = \mathbf{A} + \mathbf{B} \text{ und } \mathbf{D} = \mathbf{A} - \mathbf{B} \quad (1.5.3)$$

uneingeschränkt ausführen. Gleiches gilt für die Kombination aus Skalar und Vektor. Der Skalar wird hierbei zu *jedem* Element der Matrix bzw. des Vektors addiert oder von *jedem* subtrahiert. Die Anwendung auf Matrizen ist – entsprechend der Algebra-Regeln – nur bei gleicher Dimension von **A** und **B** möglich. Dazu werden die Elemente gleicher Position in **A** und **B** jeweils addiert bzw. subtrahiert.

1.5.2 Multiplikation

Matrizenmultiplikationen werden mit

$$\mathbf{A} = \mathbf{A} * \mathbf{B} \quad (1.5.4)$$

durchgeführt, wobei die Spaltenanzahl von **A** gleich der Zeilenzahl von **B** sein muss.

Dimension: $I \times m * m \times n = I \times n$ (1.5.5)

Entsprechend den Rechenregeln mit Matrizen sind für **A** und **B** auch beliebige Kombinationen von Skalar und Matrix bzw. Skalar und Vektor zulässig. Der Skalar wird dabei mit jedem Element der Matrix oder des Vektors multipliziert.

Mit

$$\mathbf{A} = \mathbf{A} . * \mathbf{B} \quad (1.5.6)$$

lassen sich Matrizen gleicher Dimension elementweise multiplizieren, d. h. jedes Element von **A** wird mit dem entsprechenden Element von **B** multipliziert. Sind **A** und/oder **B** Skalare, liefern (1.5.4) und (1.5.6) gleiche Ergebnisse.

1.5.3 Division

Die Matrizendivision wird vor allem bei der Lösung linearer Gleichungssysteme verwendet. Ist durch

$$\mathbf{A} * \mathbf{x} = \mathbf{b} \quad (1.5.6)$$

ein lineares Gleichungssystem gegeben, dann berechnet der *Backslash-Operator*

$$\mathbf{x} = \mathbf{A} \setminus \mathbf{b} \quad (1.5.7)$$

dessen Lösung. In der Mathematik entspricht das der Schreibweise $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, wobei \mathbf{A}^{-1} die Inverse der Matrix **A** darstellt. Bei der numerischen Lösung wird \mathbf{A}^{-1} nicht explizit

berechnet, sondern das Gleichungssystem durch ein Eliminationsverfahren gelöst (LU-Faktorisierung mit Spaltenpivotierung). Die Linksdivision in MATLAB lässt sich stets auf zwei Matrizen passender Dimension anwenden.

$$Q\mathbf{L} = \mathbf{A} \setminus \mathbf{B} \quad (1.29)$$

Die Rechtsdivision

$$Q\mathbf{R} = \mathbf{A} / \mathbf{B} \quad (1.30)$$

entspricht der Multiplikation der Matrix **A** mit der Inversen von **B**.

Die elementweise Linksdivision zweier Matrizen wird mit

$$Q\mathbf{EL} = \mathbf{A} .\setminus \mathbf{B} \quad (1.31)$$

und die elementweise Rechtsdivision

$$Q\mathbf{ER} = \mathbf{A} ./ \mathbf{B} \quad (1.32)$$

berechnet. Die elementweise Linksdivision kann durch Tauschen der Matrizen durch die elementweise Rechtsdivision ersetzt werden.

1.6 Konstruktionen zur Programmsteuerung

In diesem Abschnitt werden einige wichtige MATLAB -Befehle behandelt, mit denen Programmabläufe kontrolliert werden können.

- **if** (Verzweigung)
- **switch** (Mehrfachverzweigung)
- **while** (Schleifen)
- **for** (Zähler-Schleifen)
- **break** (Abbruch)

Programmabläufe werden oft durch Bedingungen, ausgedrückt durch Vergleichsoperationen wie

- | | |
|----|----------------|
| < | kleiner |
| <= | kleiner/gleich |
| > | größer |
| >= | größer/gleich |
| == | gleich |
| ~= | ungleich |

oder auch durch logische Operationen wie

- | | |
|---|----------------------|
| & | logisches and |
| | logisches or |

gesteuert.

1.6.1 if-Anweisung

Die **if**-Anweisung führt einen Programmblock sog. *statements* aus, wenn der **if**-Ausdruck "wahr" ist, also einen Wert ungleich 0 ergibt. Ist der Ausdruck jedoch falsch, kann die optionale Anweisung **else** einen alternativen Programmblock anbieten. Muss im **else**-Pfad eine zusätzliche Bedingung erfüllt sein, kann anstelle von **else**, **elseif** benutzt werden. Mit **end** wird stets der letzte Block einer **if**-Anweisung geschlossen. Alle Anweisungsblöcke werden jeweils durch eines der vier Schlüsselwörter getrennt.

Folgendes triviales Beispiel zeigt die Anwendung mit dem Ungleich-Operator $\sim=$

```
a = 2;
if(a~=0)
    disp('a ist ungleich 0');
else
    disp('a hat den Wert 0');
end
```

Die **else**-Anweisung realisiert dabei eine Fallunterscheidung.

1.6.2 for-Schleife

Die **for**-Schleife durchläuft einen Anweisungsblock, bis eine vorgegebene Anzahl an Durchläufen erreicht ist. **end** beendet den Anweisungsblock.

```
t=linspace(0,1,101); % Vektor t mit 101 Elementen von 0 bis 1
for i=1:length(t)
    x(i) = t(i)^2;
end
```

Im Beispiel legt die Funktion **linspace** einen Vektor **t** mit 101 Stützstellen von 0 bis 1 an. **length(t)** ermittelt die Länge des Vektors **t** (hier 101). Die **for** - Anweisung wird nun 101-mal ausgeführt, wobei die Variable **i** jedes Mal um eins erhöht wird. Der **()** -Operator holt nun bei jedem Durchlauf das **i**-te Element vom Vektor **t** und weist dessen Quadrat dem neu angehängten **i**-ten Element in **x** zu.

1.6.3 while-Schleifen

Die **while** -Schleife wiederholt einen Anweisungsblock beliebig oft, solange bis eine logische Abbruchbestimmung erfüllt ist. **end** legt das Ende des Anweisungsblocks fest.

Das folgende Programm veranschaulicht die Nutzung von **while**, **if**, **else** und **end**. Es sucht eine Nullstelle des Polynoms

$$f(x) = x^3 - 2x - 5$$

mittels Intervallschachtelungsmethode.

```
close all
a=0; % untere Grenze < Erwartungswert
b=3; % obere Grenze > Erwartungswert
i=1; % Zähler für plot
x=0; Fx=0; A=0; B=0; % Ergebnisvektoren für plot
```

```

while b-a > eps*b    % Abbruchbedingung wenn b-a < b*LSB
x=(a+b)/2;             % Mitte zwischen Grenzen
fx=x^3-2*x-5;
X(i)= x; Fx(i)= fx; A(i)= a; B(i)= b;  % für plot
if fx < 0
    a=x;                 % Untere Grenze wird
else
    b=x;
end

i= i+1;      % Laufindex für plot
end
% Ausgabe der Nullstelle:
string = ['Die Nullstelle lautet x = ', num2str(x),]
disp(string)

plot((1:i-1),Fx,'LineWidth',2 )
hold all
plot((1:i-1),A,'LineWidth',2 )
plot((1:i-1),B,'LineWidth',2 )
plot((1:i-1),X,'LineWidth',2 )
legend ('fx','a','b','x')

```

Abbildung 1-3 zeigt das Einschwingverhalten des Verfahrens für die Funktionsvariable **fx**, die gefundene Nullstelle **x** und für die Hilfsvariablen **a** und **b**. Man erkennt, dass sich schon nach 11 Iterationsschritten alle Werte im Rahmen der Darstellungsgenauigkeit nicht mehr ändern.

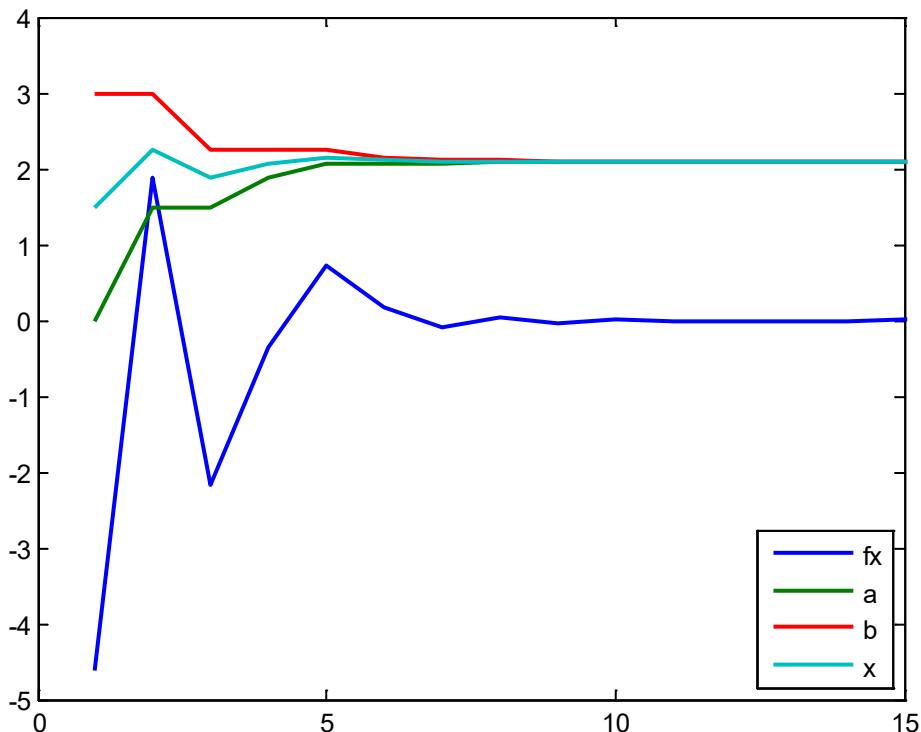


Abbildung 1-3: Einschwingverhalten der Intervallschachtelung

1.6.4 break-Anweisung

Mit der **break**-Anweisung können **for**- oder **while** – Schleifen vorzeitig verlassen werden. Alle Anweisungen in einer Schleife, die nach **break** stehen, werden nicht mehr ausgeführt. In geschachtelten Schleifen, verlässt **break** nur die innerste Schleife.

1.7 Skripte und Funktionen

Viele kleine Aufgaben lassen sich bequem direkt von der MATLAB -Befehlszeile aus erledigen. Für umfangreichere Aufgaben besteht in Matlab die Möglichkeit, eigenen Code zu programmieren und in m-Files – ebenso wie jeder vorhandene Quellcode – abzuspeichern. M-Files lassen sich mit jedem beliebigen Texteditor erstellen und dann wie jede andere MATLAB -Funktion oder wie einen MATLAB -Befehl aufrufen.

Es gibt zwei Arten von M-Files:

- Skripte akzeptieren keine Eingabeparameter und geben keine Ergebnisse zurück. Sie arbeiten mit den Daten im Workspace (und entsprechen den MATLAB -Befehlen).
- Funktionen akzeptieren Eingabeparameter und liefern Ergebnisse zurück. Interne Variablen sind lokal zur Funktion gehörig, d.h. außerhalb nicht definiert und gehen nach Rücksprung verloren.

M-Files lassen sich in beliebigen Verzeichnissen anlegen. Liegen sie nicht im aktuellen Arbeitsverzeichnis (Abfrage mit **pwd**), müssen die Verzeichnisse mit beispielsweise **addpath** bekannt gemacht werden.

Sollten Funktionsnamen im Pfad mehrfach vorhanden sein, führt MATLAB nur die zuerst gefundene Funktion aus. I.d.R. stehen selbstgeschriebene Funktionen vor den MATLAB-Funktionen, weil **addpath** eigene Verzeichnisse im MATLAB-Pfad voranstellt.

Der Inhalt eines m-Files, etwa **myfunction.m**, kann mit **type myfunction** direkt im *Command Window* angezeigt werden und alternativ im Editor.

1.7.1 Skripte

In Skript-Dateien, wie *myscript.m*, lassen sich MATLAB -Befehle in gleicher Weise speichern, wie sie im *Command-Window* eingegeben werden können. MATLAB führt beim Aufruf eines Skripts die gefundenen Befehle aus. Skripte können Daten vom Workspace nutzen, verändern oder neue anlegen, welche für weitere Berechnungen zur Verfügung zu stehen. Auch graphische Ausgaben lassen sich mit Skripten erzeugen, beispielsweise mittels **plot**-Befehl.

1.7.2 Funktionen

MATLAB -Funktionen übernehmen Eingabeargumente und geben Ergebnisse zurück. Sie werden wie Skripte in Dateien mit der Endung *.m gespeichert, wobei der Dateiname identisch zum Namen der beschriebenen Funktion gewählt werden sollte. Beim Erstellen einer neuen Funktion hilft MATLAB, indem unter *File/New/Function* ein

neues File erstellt wird, das die übliche Übergabe-Struktur bereits enthält. Funktionen arbeiten mit lokalen Variablen in einem separaten Workspace, der sich von der MATLAB -Kommandozeile ansprechen lässt und bei Rücksprung aus der Funktion verloren geht.

Der Funktion **fsinp** wird die Variable **x** übergeben, anhand derer sie die Variable **f** berechnet und zurück gibt. **fsinp** soll in **f** nur den positiven Anteil vom Sinus von **x** zurückgeben, sonst Null. Die Funktion **fsinp** ist in der Datei **fsinp.m** gespeichert.

```
function f=fsinp(x)
% fsinp(x) gibt sin(x) zurück, falls sin(x)>0, ansonsten 0
%
s=sin(x);
if(s>0)
    f=s;
else
    f=0;
end
```

1.8 Elementare Plot-Befehle

Numerische Berechnungen können durch Diagramme veranschaulicht werden. Dazu stehen in MATLAB eine Reihe von Befehlen zur Verfügung.

Folgende MATLAB -Befehle zeichnen die Kurven $y_1 = 20x + 20x^2$ und $y_2 = -3 - 5x^3$ mit $n_{max} = 101$ Stützpunkten im Bereich von $x_{min} = -3$ bis $x_{max} = 3$ in ein Diagramm.

```
% Beispiel Plotten von Kurven
% und Näherung von Funktionen
clear all
close all
n_max=101;
x_min=-3;
x_max=3;
x=linspace(x_min,x_max,n_max);
y1= 20.*x + 20*x.^2;
y2= -3 - 5*x.^3;
plot(x,y1,'-r','LineWidth',2); hold on
plot(x,y2,'LineWidth',2); grid on
legend('y1','y2')
```

Kommentaren werden in MATLAB Prozentzeichen % vorangestellt. Die Anweisung **clear all** löscht alle Variablen aus Workspace und **close all** schließt alle eventuell offenen Bild-Fenster *figures*.

Der MATLAB -Befehl **x = linspace(xmin,xmax,nmax)** erzeugt einen Vektor **x**, mit **nmax** Elementen, die äquidistant von **x(1) = xmin** bis **x(nmax) = xmax** angeordnet sind.

Mit dem MATLAB -Befehl **hold on** können weitere Kurven in das gleiche Bild eingezeichnet werden. Der MATLAB -Befehl **grid on** zeichnet ein Gitter.

Linieneigenschaften wie Stil und Farbe können mit

$$\text{plot}(x; y; s) \quad (1.8.1)$$

angepasst werden, wobei im *string s* die Eigenschaften gewählt werden, beispielsweise:

'.'	einzelne Punkte	'y'	yellow
'o'	einzelne Kreise	'm'	magenta
'x'	x-Markierung	'c'	cyan
'+'	+-Markierung	'r'	red
'*'	*-Markierung	'g'	green
'-'	durchgezogene Linie	'b'	blue
:	gepunktete Linie	'w'	white
'--'	strich-punktierte Linie	'k'	black
'---	gestrichelte Linie		

Auch Kombinationen sind erlaubt. So zeichnet der Befehl `plot(x,y,':b')` eine gepunktete blaue Linie. Der Befehl `legend` fügt eine Legende ein, die einen angegebenen String zur entsprechenden Linie zuordnet. Das Layout einer Grafik (*figure*) kann auch nachträglich menügesteuert verändert oder ergänzt werden, beispielsweise Linien, Label oder Marker.

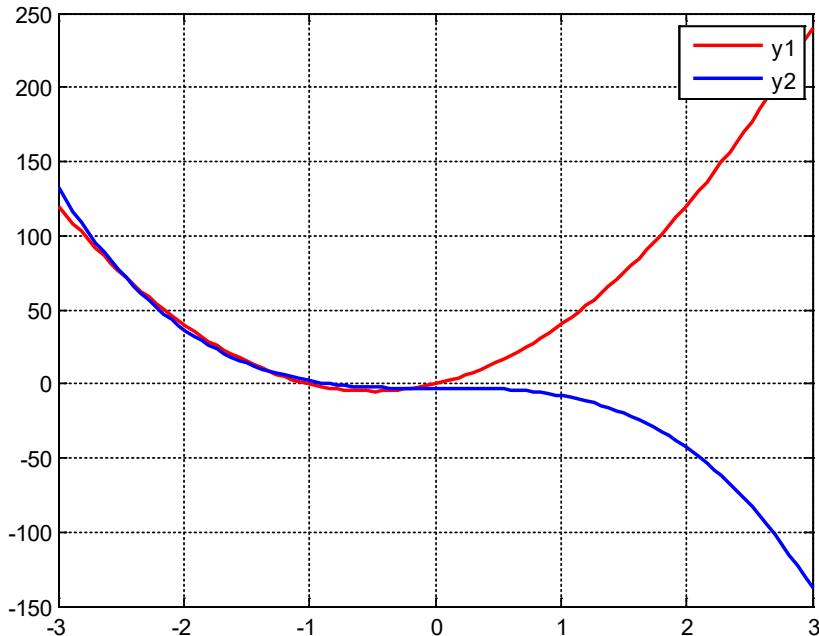


Abbildung 1-4: Die Funktionen $y_1 = 20x + 20x^2$ und $y_2 = -3 - 5x^3$

Der `print`-Befehl druckt Grafiken oder Modelle auf dem Drucker oder in eine Datei. Mittels nachgestellten Optionen lässt sich die Ausgabe spezifizieren. Mit

`print -depsc2 bild.eps` (1.8.2)

wird beispielsweise das aktuelle Bild in der Datei `bild.eps` als farbige Vektorgraphik (Encapsulated Postscript, Level 2) abgespeichert. (Weiteres unter `help print`).

1.8.1 3D-Plots

Bei Feldberechnungen (mehrdimensionale Probleme) stellt man häufig die Ergebnisse in Schnittebenen dar. Skalare Größen (z.B. Druck $p = p(x; y)$) können z.B. farbig dargestellt werden. In MATLAB gibt es eine Vielzahl leistungsstarker 3D-Plotfunktionen, von denen einige exemplarisch anhand eines Beispiels gezeigt werden.

Als Beispiel soll die Funktion

$$z(x, y) = -2e^{-(x-x_1)^2-(y-y_1)^2} + e^{-(x-x_2)^2-(y-y_2)^2} \quad (1.8.3)$$

auf verschiedene Weisen dargestellt werden. Hierzu müssen zunächst wie beim Liniенplot erst Stützstellen auf einem 2D-Gitter berechnet werden. Für x und y werden mit `linspace` äquidistante Punkte erzeugt. Für jede Kombination von x und y wird ein Funktionswert berechnet und in einem 2-dimensionalen Feld $z_{i,j}$ gespeichert.

```
close all
x1=3; % gegeben
y1=1; % gegeben
x2=5; % gegeben
y2=-1; % gegeben
x=linspace(1, 7, 61); % sinnvoll gewählt
y=linspace(-3, 3, 31); % sinnvoll gewählt (möglichst nicht quadratisch)
for i=1:length(y)
    for j=1:length(x)
        r1= (x(j)-x1)^2+(y(i)-y1)^2;
        r2= (x(j)-x2)^2+(y(i)-y2)^2;
        z(i,j)=-2*exp(-r1)+exp(-r2);
    end
end
```

Im Array $z(i,j)$ sind nach Durchlaufende alle z -Ergebnisse für x_j und y_i gespeichert. Zur Darstellung eines 2D-Datenfeldes kann beispielsweise der Befehl `surf` verwendet werden. `surf(z)` stellt eine Fläche aus den Werten in z mit den Indices $i=1, \dots, 30$ und $j=1, \dots, 50$ dar. Die notwendige Zuordnung zu den Achsen x und y wird mit

```
surf(x,y,z) % Oberflächendiagramm
xlabel('x'); ylabel('y'); zlabel('z') % Achsenbeschriftung
```

erreicht, wobei zu beachten ist, dass x die x_j -Werte zu den Spalten der Matrix z sind, y sind die y_i Werte zu den Zeilen von z . Die Zuordnung in `surf` zeigt folgende Tabelle:

$y \setminus x$	1	1.1000	1.2000	1.3000	1.4000	...
-3	2,2634e-06	3,3452e-06	4,8479e-06	6,8897e-06	9,6047e-06	...
-2.8000	7,2169e-06	1,0664e-05	1,5449e-05	2,1945e-05	3,0570e-05	
-2.6000	2,1245e-05	3,1388e-05	4,5462e-05	6,4557e-05	8,9886e-05	
-2.4000	5,7738e-05	8,5295e-05	0,0001	0,0001	0,0002	
-2.2000	0,0001	0,0002	0,0003	0,0004	0,0006	
...						...

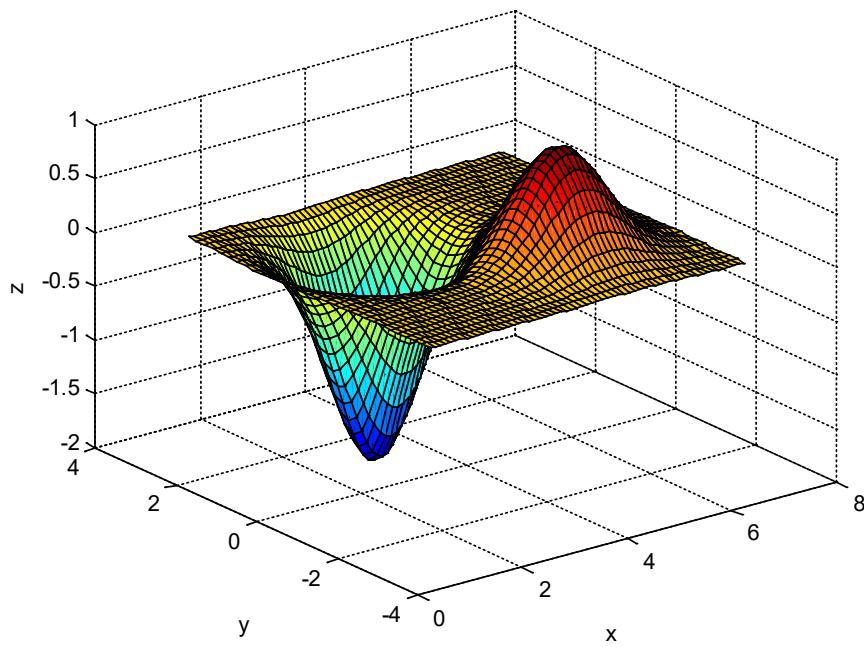


Abbildung 1-5: **surf**-Plot zur Gleichung (2.8.3) für $x_1 = 3; y_1 = 1; x_2 = 5; y_2 = -1$

Mit dem **mesh**-Befehl kann anstelle der gefüllten Flächenelemente ein Netz gezeichnet werden.

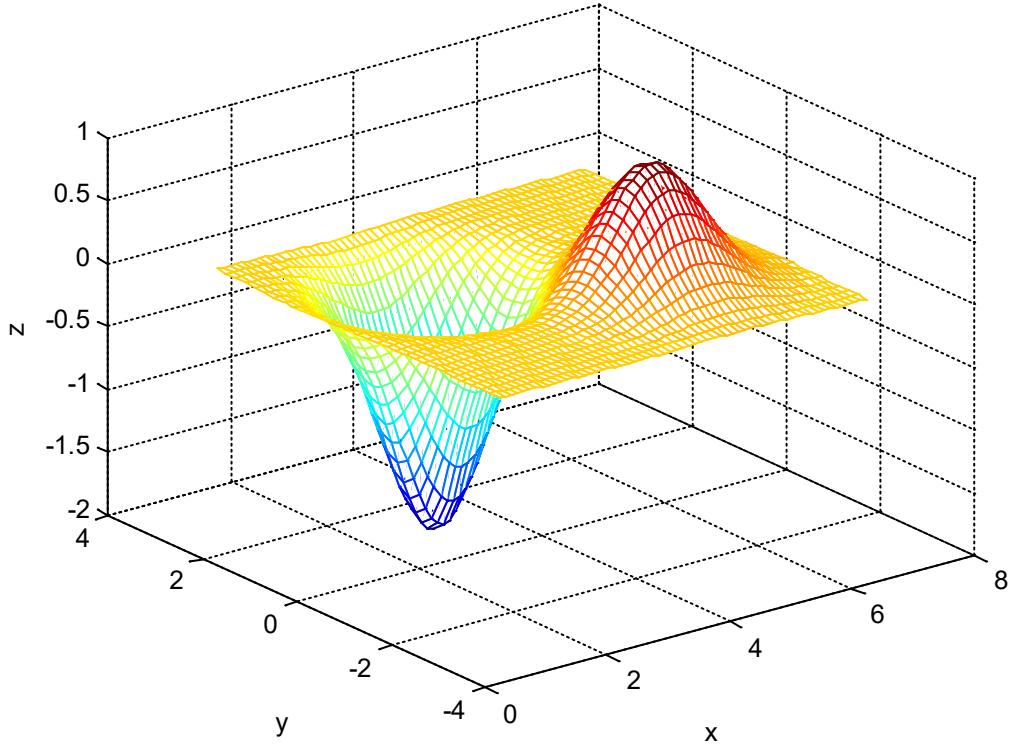


Abbildung 1-6: **mesh**-Plot zur Gleichung (2.8.3) für $x_1 = 3; y_1 = 1; x_2 = 5; y_2 = -1$

Alternativ können mit dem **contour**-Befehl Höhenlinien gezeichnet werden. Die Linien werden von Matlab zwischen z_min und z_max gleichverteilt. Ihre Anzahl wird mit dem 2. Parameter in der Klammer festgelegt.

```
figure          % Öffnet neues Fenster (erhält vorhandenes)
contour(x,y,z,14,'LineWidth',2) %Höhenlinien mit Linienstärke 2
xlabel('x')
ylabel('y')
```

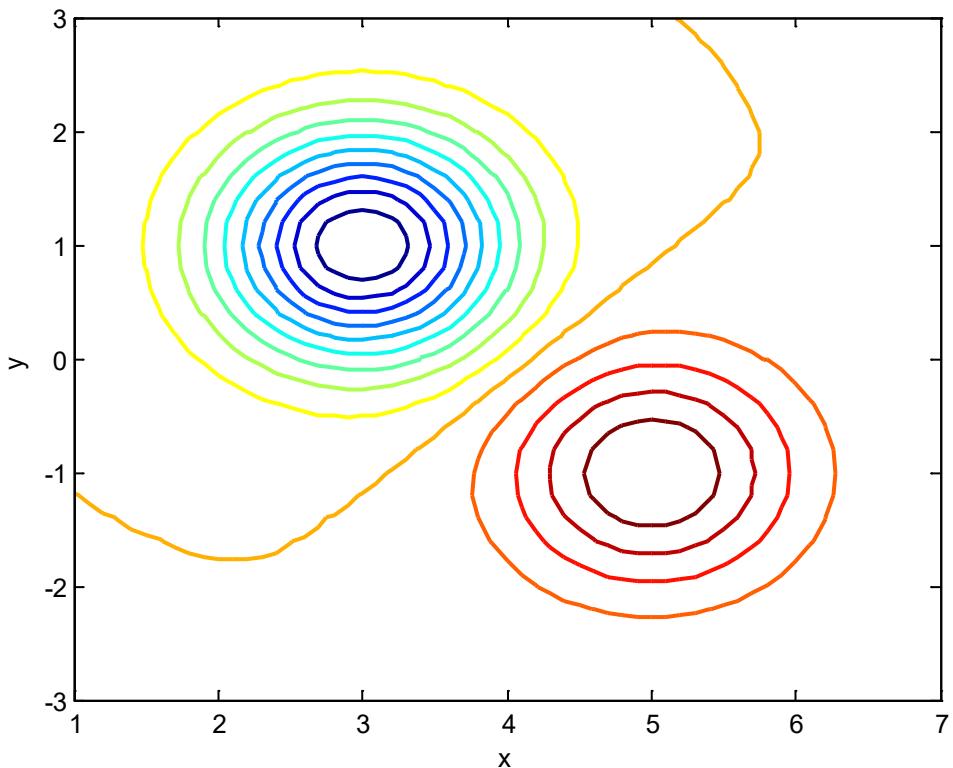


Abbildung 1-7: **contour**-Plot zur Gleichung (1.8.3)

2 Lineare Gleichungssysteme

2.1 Allgemeines

Ein lineares Gleichungssystem ist ein System linearer Gleichungen, die mehrere Unbekannte (*Variable*) enthalten. Für x_n Unbekannte kann es allgemein in der Form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \end{aligned} \tag{2.1.1}$$

angegeben werden, bzw. als

$$\mathbf{A} \mathbf{x} = \mathbf{b} \tag{2.1.2}$$

Die n Unbekannten x_1, x_2, \dots, x_n werden dazu im Vektor \mathbf{x} angeordnet, \mathbf{A} ist die Koeffizientenmatrix und der Vektor \mathbf{b} wird als rechte Seite bezeichnet.

Für $m = n$ Gleichungen ist die Koeffizientenmatrix \mathbf{A} quadratisch und der Vektor \mathbf{b} enthält $m = n$ Elemente. Das Gleichungssystem (2.1.1) ist stets eindeutig lösbar, wenn die Determinante der Koeffizientenmatrix ungleich Null ist

$$\det \mathbf{A} \neq 0. \tag{2.1.3}$$

Bei überbestimmten Gleichungssystemen stehen mehr Gleichungen als Unbekannte zur Verfügung ($m > n$). Hier hat die Koeffizientenmatrix m Zeilen und n Spalten und der Vektor \mathbf{b} enthält m Elemente. Das Gleichungssystem kann dann nur mehr bestmöglich gelöst werden, beispielsweise durch Approximation mittels Ausgleichsgeraden oder Polynomen.

Lineare Gleichungssysteme löst MATLAB mit der Anweisung

$$\mathbf{x} = \mathbf{A} \setminus \mathbf{b} \tag{2.1.4}$$

Vom Anwender sind dazu lediglich die Koeffizientenmatrix \mathbf{A} und die rechte Seite \mathbf{b} zur Verfügung zu stellen. MATLAB untersucht den Aufbau der Koeffizientenmatrix und wählt dann das passende Lösungsverfahren aus.

2.2 Vorüberlegungen

Der einfachste Fall eines linearen Gleichungssystems sind zwei Gleichungen für zwei Unbekannte x und y .

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned} \tag{2.2.1}$$

Stellt man nach y um,

$$\begin{aligned} y &= \frac{b_1}{a_{12}} - \frac{a_{11}}{a_{12}}x \\ y &= \frac{b_2}{a_{22}} - \frac{a_{21}}{a_{22}}x \end{aligned} \quad (2.2.2)$$

so erkennt man, dass hier zwei Geradengleichungen vorliegen, vgl. Abbildung 2-1.

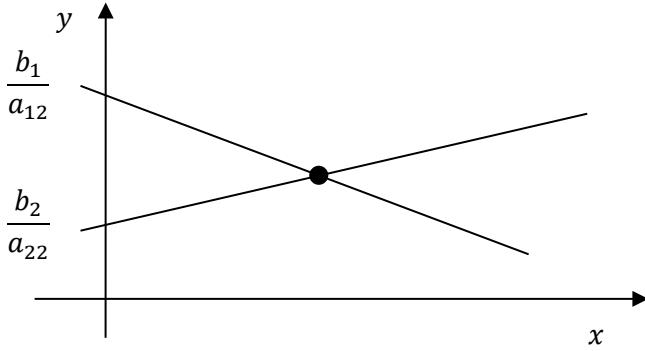


Abbildung 2-1: Beispiel eines Systems von zwei linearen Gleichungen mit genau einer Lösung

Als Lösung können *drei* Fälle auftreten:

- Es existiert *genau eine* Lösung, wenn die Steigungen der Geraden unterschiedlich sind, d.h. $\frac{a_{11}}{a_{12}} \neq \frac{a_{21}}{a_{22}}$
- Es existiert *keine Lösung*, wenn die Steigungen der Geraden identisch sind und die Geraden parallel zueinander liegen, d.h. $\frac{a_{11}}{a_{12}} = \frac{a_{21}}{a_{22}}$ und $\frac{b_1}{a_{12}} \neq \frac{b_2}{a_{22}}$
- Es existieren *unendlich viele* Lösungen, wenn die beiden Geraden identisch sind, also aufeinander liegen, d.h. $\frac{a_{11}}{a_{12}} = \frac{a_{21}}{a_{22}}$ und $\frac{b_1}{a_{12}} = \frac{b_2}{a_{22}}$

Leicht sieht man, die Bedingung $\frac{a_{11}}{a_{12}} = \frac{a_{21}}{a_{22}}$ ist gleichbedeutend mit

$$a_{11}a_{22} - a_{21}a_{12} = 0 \quad (2.2.3)$$

Der linke Teil dieses Ausdrucks berechnet die Determinante der Koeffizientenmatrix

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad (2.2.4)$$

Ein lineares Gleichungssystem ist also nur dann eindeutig lösbar, wenn die Determinante der Koeffizientenmatrix ungleich null ist.

Die Übertragung auf drei lineare Gleichungen mit drei Größen x , y und z kann ebenfalls geometrisch interpretiert werden. Die Gleichungen lauten allgemein

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1 \\ a_{21}x + a_{22}y + a_{23}z &= b_2 \\ a_{31}x + a_{32}y + a_{33}z &= b_3 \end{aligned} \quad (2.2.5)$$

Löst man eine der drei Gleichungen nach $z = z(x, y)$ auf, so sieht man, dass sie eine Ebene im dreidimensionalen Raum beschreibt. Jede der drei Gleichungen beschreibt

nun eine eigene Ebene. Liegen zwei der Ebenen schräg zueinander (nicht parallel), so schneiden sie sich in einer Linie (Schnittlinie). Tritt diese Linie durch die dritte Ebene, dann ist der Durchtrittspunkt die (eindeutige) Lösung des Gleichungssystems.

Auch hier kann es unendlich viele Lösungen geben. Der triviale Fall liegt vor, wenn alle Ebenen aufeinander liegen. Zudem können zwei der Ebenen aufeinander liegen und durch die dritte geschnitten werden. Auf den Schnittlinien finden sich dann unendlich viele Lösungen.

Keine Lösung gibt es hingegen, wenn alle Ebenen zueinander parallel sind und mindestens eine einen Abstand von den anderen Ebenen hat.

Auch hier gibt die Determinante der Koeffizientenmatrix Aufschluss, ob eine eindeutige Lösung existiert. In MATLAB kann die Determinante einer Matrix \mathbf{A} mittels `det(A)` berechnet werden. Ist die Determinante null, so existiert keine eindeutige Lösung. Die Aussagekraft der Determinante ist für beliebig viele Gleichungen (Dimensionen) gültig.

2.3 Quadratische Koeffizientenmatrizen

2.3.1 Beispiel Fachwerk

In Abbildung 2-2 ist ein Fachwerk, bestehend aus 5 Stäben gegeben. In den Knoten I und II sei es statisch bestimmt¹ gelagert und werde durch die vertikale Kraft F belastet. Die äußereren Abmessungen sind durch a und h gegeben und die Parameter p und q bestimmen die Lage von Knoten III.

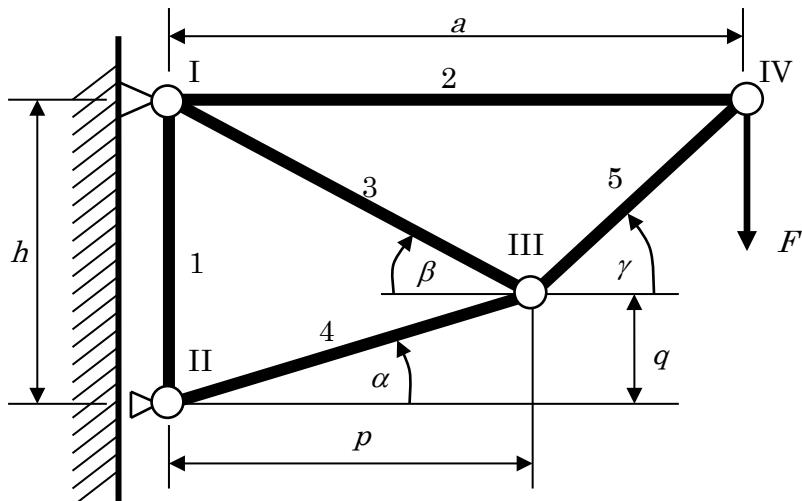


Abbildung 2-2: Ebenes, ideales Fachwerk

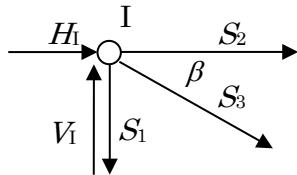
¹ Bei beliebiger Belastung des Systems können die Auflagerreaktionen allein aus den (zeichnerischen und rechnerischen) Gleichgewichtsbedingungen bestimmt werden.

Die Neigung der Stäbe 3, 4 und 5 kann durch die Winkel α , β und γ angegeben werden. Deren Winkelfunktionen können aus

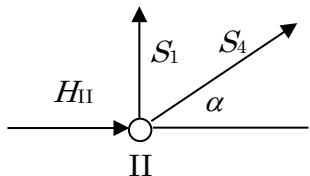
$$\begin{aligned}\sin \alpha &= \frac{q}{\sqrt{p^2+q^2}}, & \cos \alpha &= \frac{p}{\sqrt{p^2+q^2}} \\ \sin \beta &= \frac{h-q}{\sqrt{p^2+(h-q)^2}}, & \cos \beta &= \frac{p}{\sqrt{p^2+(h-q)^2}} \\ \sin \gamma &= \frac{h-q}{\sqrt{(a-p)^2+(h-q)^2}}, & \cos \gamma &= \frac{a-p}{\sqrt{(a-p)^2+(h-q)^2}}\end{aligned}\quad (2.3.1)$$

berechnet werden.

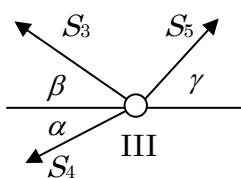
Aus den Kräftegleichgewichten in den Knoten I bis IV erhält man acht Gleichungen für die drei unbekannten Lagerreaktionen H_I , V_I , H_{II} sowie die Stabkräfte S_1 bis S_5 . Dazu ist es zweckmäßig, die Knoten einzeln aufzuzeichnen, und nach horizontalen und vertikalen Komponenten zu zerlegen. Hierbei ist darauf zu achten, dass alle fachwerkinternen Kräfte als Zugkräfte darzustellen sind. Die Richtungsanteile der Pfeile, die dann den Koordinaten x und y parallel gerichtet sind, sind positiv zu werten und jene in entgegengesetzter Richtung negativ.



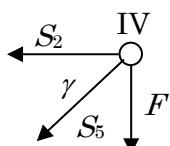
$$\begin{aligned}0_x &= S_2 + S_3 \cdot \cos \beta + H_I \\ 0_y &= -S_1 - S_3 \cdot \sin \beta + V_I\end{aligned}\quad (2.3.2)$$



$$\begin{aligned}0_x &= H_{II} + S_4 \cdot \cos \alpha \\ 0_y &= S_1 + S_4 \cdot \sin \alpha\end{aligned}\quad (2.3.3)$$



$$\begin{aligned}0_x &= -S_3 \cdot \cos \beta + S_5 \cdot \cos \gamma - S_4 \cdot \cos \alpha \\ 0_y &= S_3 \cdot \sin \beta - S_4 \cdot \sin \alpha + S_5 \cdot \cos \gamma\end{aligned}\quad (2.3.4)$$



$$\begin{aligned}0_x &= -S_2 - S_5 \cdot \cos \gamma \\ 0_y &= -F - S_5 \cdot \sin \gamma\end{aligned}\quad (2.3.5)$$

Das folgende MATLAB-Programm belegt die Matrix **A** und die Rechte Seite **b** mit Werten, berechnet die Lösung und gibt die Lagerreaktionen und die Stabkräfte getrennt aus.

```

% ebenes, ideales Fachwerk
%
clear all
%
% gegebene Daten
F=2; % [kN] Belastung
a=3.0; % [m] Feldbreite
h=1.5; % [m] Feldhoehe
p=2.0; % [m] Knoten III x-Position
q=0.5; % [m] Knoten III y-Postion
%
% Hilfs-Groessen
sa = q / sqrt ( p^2 + q^2 ) ;
ca = p / sqrt ( p^2 + q^2 ) ;
sb = (h-q) / sqrt ( p^2 + (h-q)^2 ) ;
cb = p / sqrt ( p^2 + (h-q)^2 ) ;
sg = (h-q) / sqrt ( (a-p)^2 + (h-q)^2 ) ;
cg = (a-p) / sqrt ( (a-p)^2 + (h-q)^2 ) ;
%
% Koeffizientenmatrix
%   H_I V_I H_II S1 S2 S3 S4 S5
A = [ 1 0 0 0 1 cb 0 0 ; ...
      0 1 0 -1 0 -sb 0 0 ; ...
      0 0 1 0 0 0 ca 0 ; ...
      0 0 0 1 0 0 sa 0 ; ...
      0 0 0 0 0 -cb -ca cg ; ...
      0 0 0 0 0 sb -sa sg ; ...
      0 0 0 0 -1 0 0 -cg ; ...
      0 0 0 0 0 0 0 -sg ] ; %
%
% Rechte Seite
b = [ 0; 0; 0; 0; 0; 0; 0; F ];
%
% Loesung:
x = A\b ;
%
% Lagerreaktionen und Stabkraefte
H_I = x(1), V_I = x(2), H_II = x(3), S = x(4:8)

```

Die berechneten Kräfte an den Wandpunkten und den Stäben betragen:

$$\begin{array}{lll}
 H_I = -4 & H_{II} = 4 & S1 = 1 \\
 V_I = 2 & & S2 = 2 \\
 & & S3 = 2.2361 \\
 & & S4 = -4.1231 \\
 & & S5 = -2.8284
 \end{array}$$

2.3.2 Wärmeleitung in einer Platte

Die Ränder einer homogenen Platte seien dauerhaft auf den Temperaturen T_U , T_O , T_L und T_R gehalten. Nach hinreichend langer Zeit, stellt sich ein konstanter Temperaturverlauf über die Platte ein, der durch die "Potential-Gleichung" (Laplace-Gleichung)

$$\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = 0 \quad (2.3.6)$$

beschrieben wird und in dem $T = T(x,y)$ die von den Ortskoordinaten x und y abhängige Temperatur bezeichnet.

Die Potentialgleichung lässt sich für ein eindimensionales Gebilde, beispielsweise einen dünnen Stab, der an seinen beiden Enden auf unterschiedlichen Temperaturen gehalten wird, leicht einsehen. Die Stabseitenwände sollen gegenüber der Umgebung thermisch isoliert sein. Das Material soll eine homogene Wärmeleitfähigkeit besitzen, so dass sich entlang des Stabes ein konstantes Temperaturgefälle einstellt.

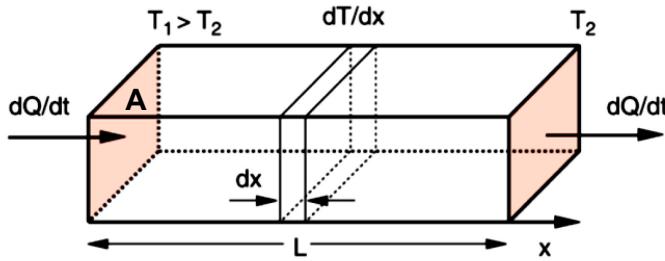


Abbildung 2-3: Wärmeleitung in einem Stab

Wir betrachten nun ein Volumenscheibchen bestehend aus der Stabquerschnittsfläche A und dem Abstand zwischen einer beliebigen Stelle x und $x+dx$.

Die Wärmemenge, die durch den Stabquerschnitt A an der Stelle x in das Volumenscheibchen $A \cdot dx$ mit der Wärmeleitfähigkeit λ eintritt, lässt sich für ein Temperaturgefälle angeben als:

$$\frac{dQ_{ein}}{dt} = -\lambda \cdot A \cdot \frac{\partial T}{\partial x} \quad (2.3.7)$$

(partielle Ableitung wegen der zweiten Größe Zeit t)

Die Temperatur an der Austrittsfläche $x + dx$ beträgt:

$$T(x + dx) = T(x) + \frac{\partial T}{\partial x} dx \quad (2.3.8)$$

Die Wärmemenge, die nun aus dem Volumenscheibchen an der Stelle $x+dx$ wieder austritt ist:

$$\frac{dQ_{aus}}{dt} = -\lambda \cdot A \cdot \frac{\partial}{\partial x} \left(T(x) + \frac{\partial T}{\partial x} dx \right) \quad (2.3.9)$$

Die Differenz der Wärmemengen, die durch die Flächen an x und $x+dx$ strömen, beschreiben die Temperaturänderung im Volumenscheibchen:

$$\begin{aligned}
\frac{dQ}{dt} &= \frac{dQ_{aus}}{dt} - \frac{dQ_{ein}}{dt} \\
&= -\lambda \cdot A \cdot \frac{\partial}{\partial x} \cdot \left(T(x) + \frac{\partial T}{\partial x} dx \right) - \left(-\lambda \cdot A \cdot \frac{\partial T}{\partial x} \right) \\
&= -\lambda \cdot A \cdot \left(\frac{\partial}{\partial x} \cdot \left(T(x) + \frac{\partial T}{\partial x} dx \right) - \frac{\partial T}{\partial x} \right) \\
&= -\lambda \cdot A \cdot \left(\frac{\partial T}{\partial x} + \frac{\partial^2 T}{\partial x^2} dx - \frac{\partial T}{\partial x} \right) \\
&= -\lambda \cdot A \cdot \frac{\partial^2 T}{\partial x^2} dx \\
\frac{dQ}{dt} &= -\lambda \cdot \frac{\partial^2 T}{\partial x^2} dV
\end{aligned} \tag{2.3.10}$$

Sind nun alle Ausgleichsvorgänge abgeschlossen, so bleibt die Temperatur im betrachteten Volumenscheibchen konstant, was bedeutet, dass dieselbe Wärmemenge in das Volumenscheibchen eintritt wie austritt. Somit ist deren Differenz null, also:

$$0 = \frac{dQ}{dt} = -\lambda \cdot \frac{\partial^2 T}{\partial x^2} dV \tag{2.3.11}$$

weshalb

$$0 = \frac{\partial^2 T}{\partial x^2} = \frac{\partial^2 T(x)}{\partial x^2} \tag{2.3.12}$$

sein muss. Im Zweidimensionalen gilt dann Gleichung (2.3.6)

Die partiellen Ableitungen in $a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$ (2.1.1) lassen sich durch Differenzen-Quotienten approximieren. So kann die 2. Ableitung nach x aus der Änderung der 1. Ableitung berechnet werden,

$$\frac{\partial^2 T(x, y)}{\partial x^2} \Big|_{x,y} = \frac{\frac{\partial T(x, y)}{\partial x} \Big|_{\text{vorwärts}} - \frac{\partial T(x, y)}{\partial x} \Big|_{\text{rückwärts}}}{\Delta x}, \tag{2.3.13}$$

in welcher

$$\frac{\partial T(x, y)}{\partial x} \Big|_{\text{vorwärts}} = \frac{T(x + \Delta x, y) - T(x, y)}{\Delta x} \tag{2.3.14}$$

als Vorwärts- und

$$\frac{\partial T(x, y)}{\partial x} \Big|_{\text{rückwärts}} = \frac{T(x, y) - T(x - \Delta x, y)}{\Delta x} \tag{2.3.15}$$

als Rückwärts-Differenzen-Quotienten bezeichnet wird. Allgemein beschreiben die Vorwärts- und Rückwärts-Differenzen-Quotienten folgenden Zusammenhang:

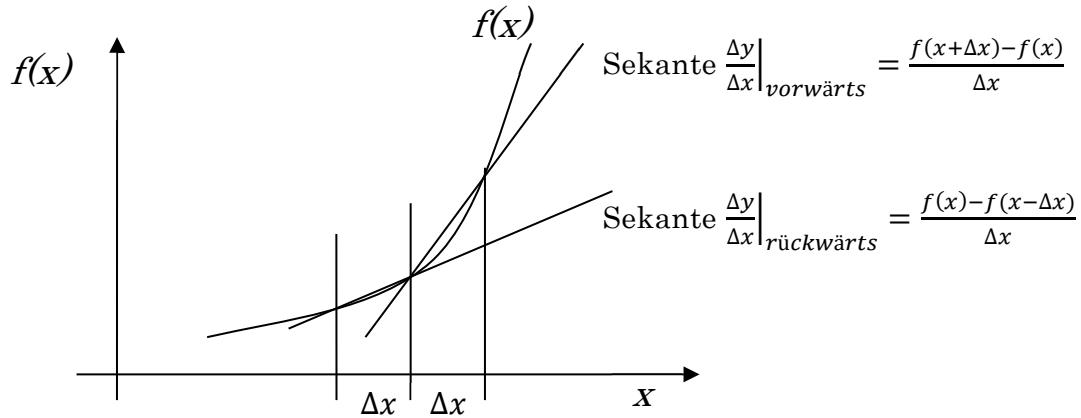


Abbildung 2-4: Vorwärts- und Rückwärts-Differenzen-Quotienten

Werden die Vorwärts- und Rückwärts-Differenzen-Quotienten eingesetzt, so wird

$$\frac{\partial^2 T(x, y)}{\partial x^2} \Big|_{x,y} = \frac{T(x + \Delta x, y) - 2 \cdot T(x, y) + T(x - \Delta x, y)}{\Delta x^2} \quad (2.3.16)$$

Analog dazu erhält man für die 2. Ableitung nach y

$$\frac{\partial^2 T(x, y)}{\partial y^2} \Big|_{x,y} = \frac{T(x, y + \Delta y) - 2 \cdot T(x, y) + T(x, y - \Delta y)}{\Delta y^2} \quad (2.3.17)$$

die "Potential-Gleichung" $\frac{\partial^2 T(x,y)}{\partial x^2} + \frac{\partial^2 T(x,y)}{\partial y^2} = 0$ (2.3.6) lautet dann

$$\frac{T(x + \Delta x, y) - 2 \cdot T(x, y) + T(x - \Delta x, y)}{\Delta x^2} + \frac{T(x, y + \Delta y) - 2 \cdot T(x, y) + T(x, y - \Delta y)}{\Delta y^2} \quad (2.3.18)$$

Für gleiche Gitterabstände $\Delta x = \Delta y$ sowie in jedem Punkt konstante Temperatur (eingeschwungener Zustand) bleibt schließlich

$$T(x + \Delta x, y) - 4 \cdot T(x, y) + T(x - \Delta x, y) + T(x, y + \Delta y) + T(x, y - \Delta y) = 0 \quad (2.3.19)$$

Für eine quadratische Platte mit 4x4 Elementen befinden sich an den 3x3 Knoten die Temperaturen T_1 bis T_9 .

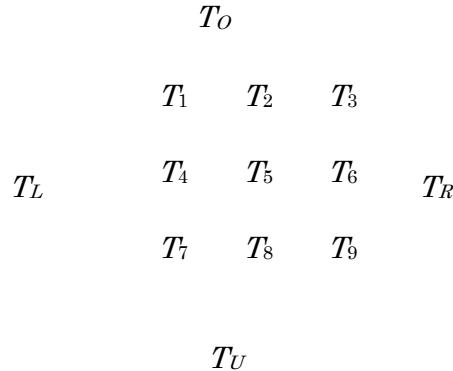


Abbildung 2-5: Segmentierung für Wärmeleitung in einer Platte

Für jeden Knoten lassen sich nun die angrenzenden Differenzen in Gleichung $T(x + \Delta x, y) - 4 \cdot T(x, y) + T(x - \Delta x, y) + T(x, y + \Delta y) + T(x, y - \Delta y) = 0$ (2.3.19) einsetzen:

$$\begin{aligned} 0 &= -4T_1 + T_2 + T_L + T_O + T_4 \\ 0 &= -4T_2 + T_3 + T_1 + T_O + T_5 \\ 0 &= -4T_3 + T_R + T_2 + T_O + T_6 \\ 0 &= -4T_4 + T_5 + T_L + T_1 + T_7 \\ 0 &= -4T_5 + T_6 + T_4 + T_2 + T_8 \\ 0 &= -4T_6 + T_R + T_5 + T_3 + T_9 \\ 0 &= -4T_7 + T_8 + T_L + T_4 + T_U \\ 0 &= -4T_8 + T_9 + T_7 + T_5 + T_U \\ 0 &= -4T_9 + T_R + T_8 + T_6 + T_U \end{aligned}$$

Werden die Unbekannten T_1 bis T_9 als Vektor x zusammengefasst, dann lassen sich die Gleichungen als lineares Gleichungssystem $Ax = b$ schreiben.

Das folgende MATLAB-Programm belegt die Matrix A und die Rechte Seite b mit Werten und berechnet das Temperaturprofil der Platte

```
clear all, close all
%
% Daten (Temperaturen an den Plattenrändern)
T_L = 150; T_R = 100; T_U = 0; T_O = 200;
%
% Koeffizientenmatrix
A = [ 4 -1 0 -1 0 0 0 0 0 ; ...
       -1 4 -1 0 -1 0 0 0 0 ; ...
        0 -1 4 0 0 -1 0 0 0 ; ...
       -1 0 0 4 -1 0 -1 0 0 ; ...
        0 -1 0 -1 4 -1 0 -1 0 ; ...
        0 0 -1 0 -1 4 0 0 -1 ; ...
        0 0 0 -1 0 0 4 -1 0 ; ...
        0 0 0 0 -1 0 -1 4 -1 ; ...
        0 0 0 0 0 -1 0 -1 4 ] ;
%
% Rechte Seite
b = [ T_L+T_O; ...
       T_O; ...
       T_O+T_R; ...
       T_L; ...
       0; ... ]
```

```

T_R; ...
T_L+T_U; ...
T_U; ...
T_R+T_U ];
%
x = A\b;
%
% Temperatur-Verteilung als Matrix
Tv = [ (T_L+T_O)/2    T_O      T_O      T_O      (T_O+T_R)/2 ; ...
        T_L      x(1)    x(2)    x(3)    T_R ; ...
        T_L      x(4)    x(5)    x(6)    T_R ; ...
        T_L      x(7)    x(8)    x(9)    T_R ; ...
        (T_L+T_U)/2  T_U      T_U      T_U      (T_U+T_R)/2 ]
%
% graphische Aufbereitung
surf(Tv), colormap('hot'), shading interp, axis off, view(0,-90)

```

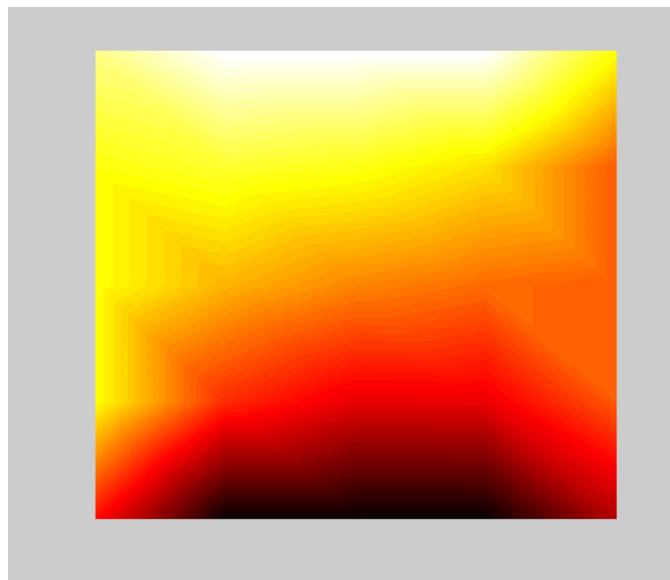


Abbildung 2-6: Wärmeleitung in einer Platte

Die an den Gitterpunkten berechneten Temperaturen werden mit den Randwerten in einer Temperaturverteilungsmatrix angeordnet und in Abbildung 2-5 grafisch veranschaulicht.

Die Befehle **surf(Tv)** und **colormap ('hot')** stellen das Temperaturprofil als Fläche dar (vgl. Abbildung 2-6), deren unterschiedliche Höhen mit dem Farbspektrum '**hot**' eingefärbt werden. Die diskreten Farbwerte an den Gitterpunkten werden interpoliert (**shading interp**) und die Fläche wird dann ohne Achsbeschriftung (**axis off**) von unten (**view(0, -90)**) betrachtet.

2.4 Rechteckige Koeffizientenmatrizen

Für die Auswertung von Messergebnissen sind Ausgleichsfunktionen ein wichtiges Hilfsmittel, um aus vorhandenen Messergebnissen Näherungsmodelle zu erstellen. Weil mehr Messergebnisse und somit mehr Gleichungen als Unbekannte zur Verfügung stehen, treten bei der Berechnung der Ausgleichsfunktionen rechteckige (nicht quadratische) Koeffizientenmatrizen auf. Nachfolgend ist das Vorgehen beschrieben, wie sich Näherungsgleichungen auf Basis vorhandener Messpunkte finden lassen:

- Annahme von bestimmten Funktionen, mit denen die Messdaten angenähert werden sollen. Die Koeffizienten der einzelnen Funktionen sind vorerst noch unbestimmt.
- Anpassen der Koeffizienten, so dass die Näherung im Vergleich zu den Messdaten bestmöglich (optimal) wird. Dies lässt sich mathematisch als Optimierungsproblem (Minimierung der Abweichungen) formulieren.

Näherungsfunktionen sind im einfachsten Falle Geraden, aber auch Polynome oder gebrochen rationale Funktionen. Auch andere geeignete Funktionen können verwendet werden. Die im Folgenden vorgestellte Methodik kann auch ohne Weiteres auf mehrdimensionale Näherungsfunktionen übertragen werden.

2.4.1 Beispiel Ausgleichsgerade

Messwerte unterliegen in der Regel einer gewissen Streuung. Weiß man, dass sie auf einer Geraden liegen sollten, so lassen sie sich durch eine Ausgleichsgerade approximieren. Allerdings würde dies für die Beschreibung mittels üblicher Geradengleichungen

$$\begin{aligned} p + q x_1 &= y_1 \\ p + q x_2 &= y_2 \\ \vdots &\quad \vdots \quad \vdots \\ p + q x_M &= y_M, \end{aligned} \tag{2.4.1}$$

bedeuten, dass alle M Punkte $x_i, y_i, i = 1 \dots M$ auf der Geraden $y = p + q \cdot x$ liegen. Im Allgemeinen ist es unmöglich, mit den zwei Geraden-Parametern p und q gleichzeitig alle M Gleichungen zu erfüllen. Lässt man Abweichungen für y_i zu, dann müssen die Gleichungen in : : :(2.4.1) durch

$$\begin{aligned} p + q x_1 - y_1 &= \epsilon_1 \\ p + q x_2 - y_2 &= \epsilon_2 \\ \vdots &\quad \vdots \quad \vdots \\ p + q x_M - y_M &= \epsilon_M, \end{aligned} \tag{2.4.2}$$

ersetzt werden, wobei hier $\epsilon_1, \epsilon_2, \dots, \epsilon_M$ die y_i -Abweichungen der Messpunkte von der Geraden also die Fehler bezeichnen. Die Gauß'sche "Methode der kleinsten Fehlerquadrate" verlangt nun, dass die Summe der quadratischen Fehler minimal wird (bzw. auch für die halbe Summe, so dass sich die Rechnung ab Gleichung (2.4.4) bzw. (2.4.5) vereinfacht). Die Forderung

$$\epsilon_G^2 = \frac{1}{2} \sum_1^M \epsilon_i^2 = \frac{1}{2} \sum_1^M (p + qx_i - y_i)^2 \rightarrow \text{Minimum} \quad (2.4.3)$$

führt über die partiellen Ableitungen auf die notwendigen Bedingungen

$$\frac{\partial \epsilon_G^2}{\partial p} = pM + q \sum x_i - \sum y_i = 0 \quad (2.4.4)$$

$$\frac{\partial \epsilon_G^2}{\partial q} = p \sum x_i + q \sum x_i^2 - \sum y_i x_i = 0 \quad (2.4.5)$$

die sich nach den Geraden-Parametern auflösen lassen

$$p = \frac{\sum y_i \sum x_i^2 - \sum x_i \sum (x_i y_i)}{M \sum x_i^2 - \sum x_i \sum x_i} \quad (2.4.6)$$

$$q = \frac{M \sum (x_i y_i) - \sum y_i \sum x_i}{M \sum x_i^2 - \sum x_i \sum x_i} \quad (2.4.7)$$

Die Gleichungen : : : (2.4.1) können nun in die Matrix-Schreibweise eines linearen Gleichungssystems überführt werden

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_M \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} \quad (2.4.8)$$

Wegen $M > N$ ist die Matrix A rechteckig. Wenn mehr Gleichungen als Unbekannte vorliegen, ist das Problem überbestimmt.

Erkennt MATLAB beim Ausführen der Anweisung

$$\mathbf{x} = \mathbf{A} \setminus \mathbf{b} \quad (2.4.9)$$

eine Koeffizientenmatrix mit mehr Zeilen als Spalten, so wird automatisch ein Lösungsalgorithmus ausgeführt, der eine Ausgleichsgerade liefert, bei welcher die Summe der Fehlerquadrate der Einzelmessungen minimiert ist.

2.4.2 Ausgleichspolynome

2.4.2.1 Ansatz

Soll beispielsweise das Polynom N -ten Grades

$$y = c_0 + c_1 x + c_2 x^2 + \dots + c_N x^N \quad (2.4.10)$$

durch $M > N + 1$ Messwerte $x_i, y_i, i = 1(1)M$ festgelegt werden, dann erhält man M Gleichungen für die $N + 1$ Unbekannten $c_0, c_1 \dots c_N$

$$y_i = c_0 + c_1 x_i + c_2 x_i^2 + \dots + c_N x_i^N, \quad i=1(1)M. \quad (2.4.11)$$

Hieraus folgt ein lineares Gleichungssystem mit rechteckiger Koeffizientenmatrix

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^N \\ 1 & x_2 & x_2^2 & \cdots & x_2^N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_M & x_M^2 & \cdots & x_M^N \end{bmatrix}}_A \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix}}_b \quad (2.4.12)$$

Die Struktur der Matrix A ist sehr einfach und kann an verschiedene Polynom-Ansätze leicht angepasst werden.

2.4.2.2 Beispiel

Um den ursprünglichen Charakter der Messdaten durch ein Ausgleichspolynom möglichst gut wiederzugeben, muss manchmal auch der Grad des Ausgleichspolynoms angepasst, i.d.R. vergrößert werden, siehe Abbildung 2-7.

Das folgende MATLAB-Skript

```
% Ausgleichspolynome
clear all, close all
% Einlesen der Mess-Daten
load mess.dat;
% sortieren nach x
[xm, idx] = sort(mess(:,1));
ym = mess(idx,2);
% einzelne Punkte plotten
plot(xm,ym,'ok'), hold on
% Parabel = gewählte Approximationsfunktion (Punkteschar hat ähnliche Form)
A = ones(length(xm),1);
A(:,2)=xm;
A(:,3)=xm.^2;
b = ym;
c = A\b;
% Berechnen und Plotten der Parabel
y2 = c(1) ...
+ c(2).*xm ...
+ c(3).*xm.^2 ;
plot(xm,y2,':r','LineWidth',2)
% Erweiterung auf Polynom 4. Grades
A(:,4)=xm.^3;
A(:,5)=xm.^4;
d = A\b;
% Berechnen und Plotten des Polynoms 4. Grades
y4 = d(1) ...
+ d(2).*xm ...
+ d(3).*xm.^2 ...
+ d(4).*xm.^3 ...
+ d(5).*xm.^4 ;
plot(xm,y4,'r','LineWidth',2)
legend('Messwerte','c1+c2*x+c3*x^2','c1+c2*x+c3*x^2+c4*x^3+c5*x^4',...
    'location','north')
xlabel('x','FontWeight','bold', 'FontSize', 14)
ylabel('y','FontWeight','bold', 'FontSize', 14)
```

```
MESS.DAT
8.2480120e-002 4.0480020e+001
8.7333118e-002 3.9114680e+001
9.8859286e-002 3.9207200e+001
1.0202021e-001 4.0335890e+001
1.2624420e-001 3.9612810e+001
```

```

1.3238950e-001 4.0179780e+001
1.2992070e-001 3.9750500e+001
1.5043090e-001 4.0026420e+001
1.7682380e-001 4.1338670e+001
1.7838660e-001 3.9903990e+001
1.6254200e-001 4.0663630e+001
1.8779760e-001 4.1125890e+001
2.2140400e-001 3.9561270e+001
2.3602260e-001 4.1117110e+001
2.4326860e-001 4.1530590e+001
2.7102560e-001 4.2609790e+001
2.5780810e-001 4.0757310e+001
2.7269770e-001 4.0846750e+001
3.0555990e-001 4.0248530e+001
3.0524310e-001 4.1295570e+001
3.4468600e-001 4.1547340e+001
3.5535840e-001 4.1383610e+001
4.0045760e-001 4.0581230e+001
3.9678580e-001 4.1379940e+001
3.9828240e-001 4.1105450e+001
4.1982030e-001 4.2024240e+001
4.0307230e-001 4.3259470e+001
4.7053020e-001 4.2003230e+001
4.8070360e-001 4.3806440e+001
4.5070070e-001 4.6454190e+001
5.2065700e-001 4.5516960e+001
5.5554980e-001 4.5543940e+001
5.8638830e-001 4.7175610e+001
5.4542580e-001 5.1344690e+001
5.7951670e-001 5.2743420e+001
6.7341330e-001 5.1791040e+001
6.4095030e-001 5.9057580e+001
6.2166510e-001 6.2503170e+001
6.5709400e-001 6.9057380e+001
7.0477020e-001 7.5168640e+001

```

liest die Messwerte (x_m, y_m) aus dem File **MESS.DAT** ein und sortiert sie nach aufsteigenden x -Werten. Danach werden Ausgleichspolynome 2. und 4. Ordnung berechnet und grafisch veranschaulicht.

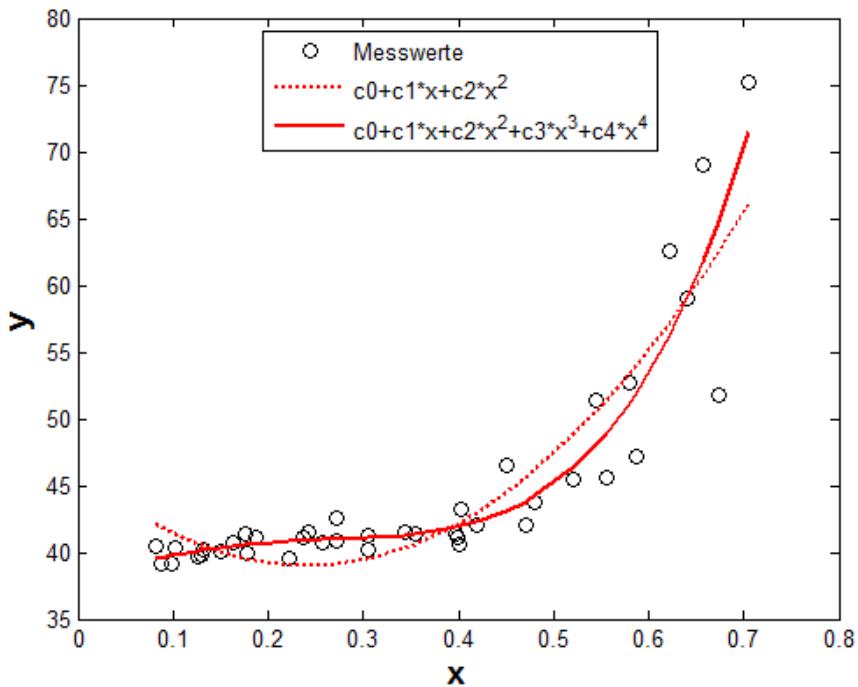


Abbildung 2-7: Ausgleichspolynome zweiten und vierten Grades

2.4.3 Gebrochen Rationale Funktionen

2.4.3.1 Ansatz

Zur Glättung von Messreihen, die ein asymptotisches Verhalten beschreiben, sind Polynome sehr ungünstig, da sie gegen $\pm\infty$ streben. Hier lassen sich mit gebrochen rationalen Ansätzen der Form

$$y = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_P x^P}{1 + b_1 x + b_2 x^2 + \cdots + b_Q x^Q} \quad (2.4.13)$$

vor allem in den Randbereichen wesentlich bessere Ergebnisse erzielen. Mit den Gleichungen

$$y_i = \frac{a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_P x_i^P}{1 + b_1 x_i + b_2 x_i^2 + \cdots + b_Q x_i^Q} \quad (2.4.14)$$

verlangt man wieder, dass die Funktion durch alle M Messpunkte (x_i, y_i) , $i = 1(1)M$ läuft. Die $N = 1 + P + Q$ Kurven-Parameter $a_0, a_1 \dots a_P$ und $b_1 \dots b_Q$ können wieder nur im Sinne eines quadratischen Fehlermittels bestimmt werden, da auch hier in der Regel $M > N$ gilt.

Zur Lösung wird Gleichung (2.4.13) umgeformt. Nach multiplizieren mit dem Nenner und umstellen bleibt

$$a_0 + a_1 x_i + a_2 x_i^2 + \cdots + a_P x_i^P - (b_1 x_i y_i + b_2 x_i^2 y_i + \cdots + b_Q x_i^Q y_i) = y_i \quad (2.4.15)$$

oder in Matrix-Schreibweise

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^P \\ 1 & x_2 & x_2^2 & \cdots & x_2^P \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_M & x_M^2 & \cdots & x_M^P \end{bmatrix}}_A \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_P \\ - \\ b_1 \\ \vdots \\ b_Q \end{bmatrix} = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix}}_x \quad (2.4.16)$$

Die Struktur der Matrix A zerfällt in zwei Anteile, ist aber nach wie vor einfach und kann an verschiedene Polynom-Ansätze leicht angepasst werden.

2.4.3.2 Beispiel

Zeigen Messwerte wie in Abbildung 2-8 ein asymptotisches Verhalten, dann ist die Approximation mittels Ausgleichspolynomen vor allem in den Randbereichen unbefriedigend.

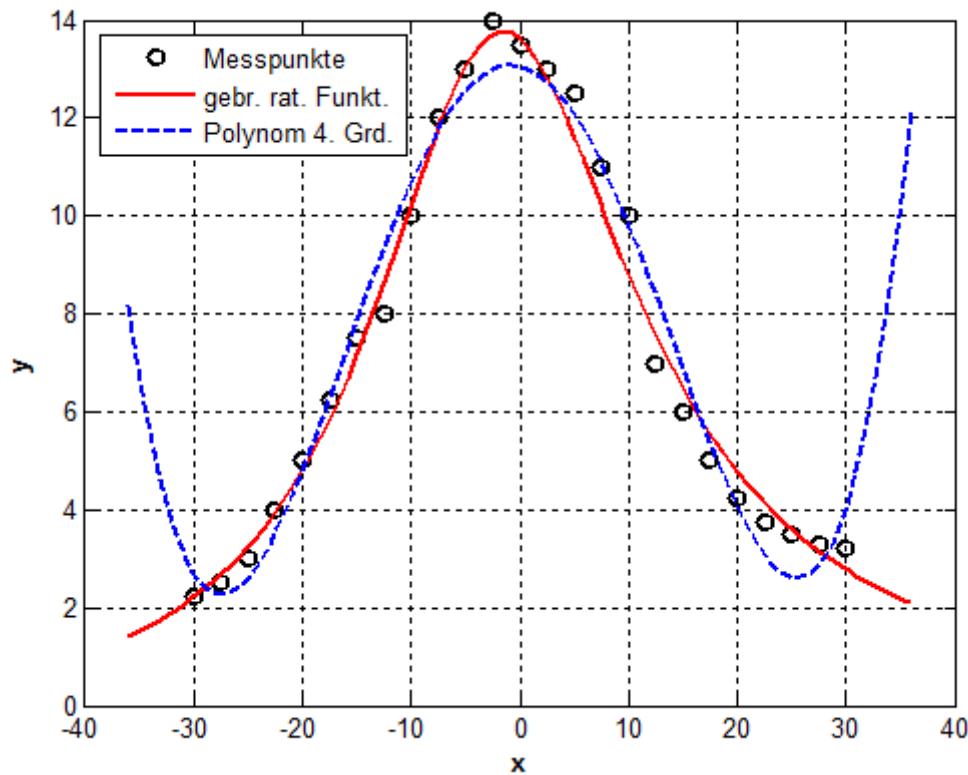


Abbildung 2-8: Ausgleichsrechnung mit gebrochen rationaler Funktion

Die gebrochenrationale Funktion

$$f(x) = \frac{a_0 + a_1 x + a_2 x^2}{1 + b_1 x + b_2 x^2} \quad (2.4.17)$$

erreicht für $x \rightarrow \pm\infty$ den Grenzwert $f(x \rightarrow \pm\infty) = a_2 / b_2$ und ist somit geeignet, die vorliegenden Messwerte zu approximieren.

Folgendes MATLAB-Skript liest Wertepaare aus der Datei `mess_rat.dat` ein und berechnet die Koeffizienten der gebrochen rationalen Ansatzfunktion.

```
clear all, close all
%
% Einlesen der Mess-Daten
load mess_rat.dat;
xm = mess_rat(:,1);
ym = mess_rat(:,2);
%
% einzelne Punkte plotten
plot(xm,ym, 'ok', 'LineWidth', 2)
hold on
%
% Koeffizientenmatrix
A=ones(length(xm),1); A(:,2)=xm; A(:,3)=xm.^2;
A(:,4)=-xm.*ym; A(:,5)=-xm.^2.*ym;
%
% Koeffizienten
c = A\ym;
%
```

```

% Definitionsbereich +-10%
dx = max(xm) - min(xm);
x_min = min(xm)-0.1*dx; x_max = max(xm)+0.1*dx;
xi = linspace(x_min,x_max,101);
%
% Funktion
yr = ( c(1) + c(2).*xi + c(3).*xi.^2 ) ...
./ ( 1 + c(4).*xi + c(5).*xi.^2 ) ;
%
plot(xi,yr,'r','Linewidth',2)
grid on

mess_rat.dat
-30.00 2.25
-27.50 2.50
-25.00 3.00
-22.50 4.00
-20.00 5.00
-17.50 6.25
-15.00 7.50
-12.50 8.00
-10.00 10.00
-7.50 12.00
-5.00 13.00
-2.50 14.00
0.00 13.50
2.50 13.00
5.00 12.50
7.50 11.00
10.00 10.00
12.50 7.00
15.00 6.00
17.50 5.00
20.00 4.25
22.50 3.75
25.00 3.50
27.50 3.30
30.00 3.20

```

Mit den MATLAB-Anweisungen

```

% zum Vergleich Polynom 4. Grades
AP=ones(length(xm),1);
AP(:,2)=xm;
AP(:,3)=xm.^2;
AP(:,4)=xm.^3;
AP(:,5)=xm.^4;
cp = AP\ym;
%
yp = cp(1) ...
+ cp(2).*xi ...
+ cp(3).*xi.^2 ...
+ cp(4).*xi.^3 ...
+ cp(5).*xi.^4 ;
%
hold on
plot(xi,yp,'--b','Linewidth',2)
legend('Messpunkte','gebr. rat. Funkt.','Polynom 4. Grd.', 2)

```

wird zum Vergleich ein Ausgleichspolynom 4. Grades berechnet und gezeichnet.

2.4.4 Übertragung auf beliebige Ansatzfunktionen

Sollen die Daten nicht durch Polynome, sondern durch andere Basisfunktionen angenähert werden, so lautet die Näherungsfunktion allgemein:

$$y(x) = \sum_{i=1}^n c_i \cdot f_i(x) \quad (2.4.18)$$

Hierbei müssen die Funktionen $f_i(x)$ vorgegeben werden. Die gesuchten Unbekannten sind hierin die Koeffizienten c_i . Man beachte dabei, dass dies eine Verallgemeinerung darstellt, d.h. auch Polynome lassen sich in dieser Form darstellen. Für Polynome lauten die Funktionen: $f_1(x) = 1$, $f_2(x) = x$, $f_3(x) = x^2$ etc.

Mit allgemeinen Funktionen erhält man für m Messwerte und n Basisfunktionen die Matrizen

$$\underbrace{\begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & & & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{bmatrix}}_A \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}}_x = \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}}_b \quad (2.4.19)$$

Die Lösung im Sinne der Minimierung der Fehlerquadrate erhält man wiederum mit $\mathbf{x} = \mathbf{A} \setminus \mathbf{b}$.

2.4.5 Mehrdimensionale Funktionen

In der Praxis findet man häufig mehrdimensionale Funktionen, weil selten eine Ausgangsgröße nur von einer Eingangsgröße abhängt. Abbildung 2-9 zeigt als Beispiel das Kennfeld einer Diesel Common Rail Pumpe. Die Hochachse ist das Ansteuersignal vom VCV (Volume Control Valve) der Pumpe als Funktion der Drehzahl n und der geforderten Fördermenge mf (mass fuel).

Man erkennt, dass das Ansteuersignal der Pumpe sowohl von der Drehzahl als auch von der geforderten Fördermenge abhängt, was bedeutet, dass zur korrekten Ansteuerung der Pumpe, neben der gewünschten Fördermenge auch die Drehzahl bekannt sein muss.

In Abbildung 2-9 sind gemessene Werte der Pumpe zusammen mit einer Näherungsfunktion dargestellt. Die gewählte Näherungsfunktion ist ein Paraboloid der beiden Eingangsgrößen.

Sind die Eingangsgrößen durch x und y angegeben, so beschreibt

$$f(x, y) = c_1 + c_2x + c_3y + c_4xy + c_5x^2 + c_6y^2 \quad (2.4.20)$$

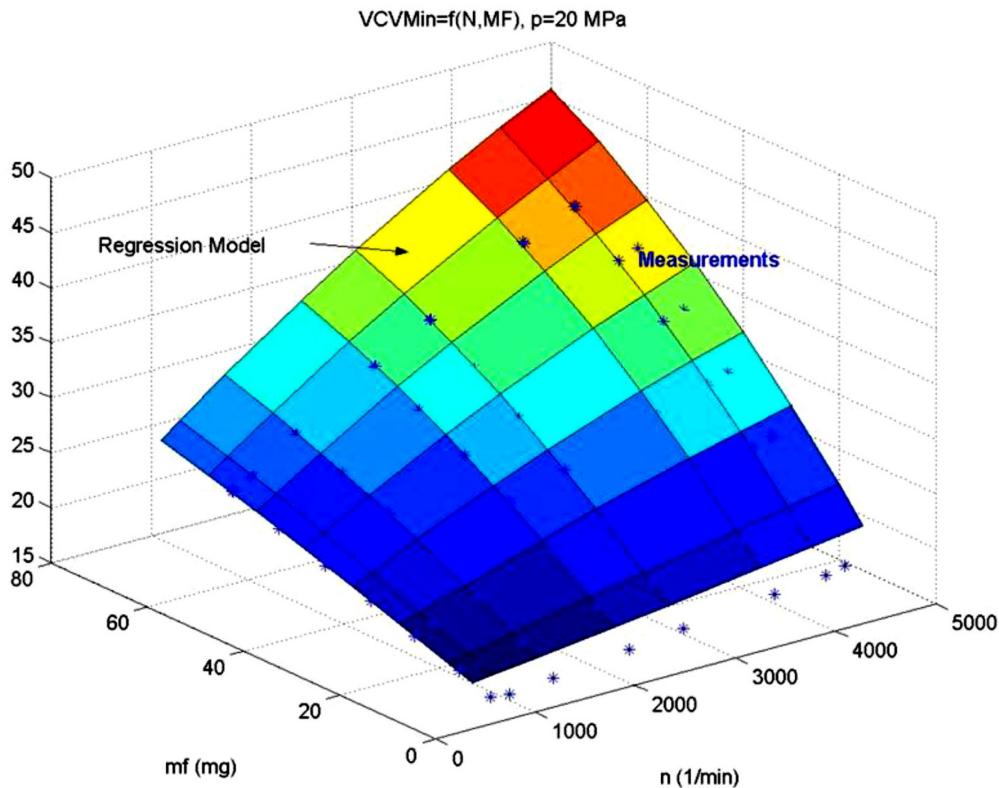


Abbildung 2-9: Kennfeld einer Diesel Pumpe

einen Paraboloid. Hierin werden die Koeffizienten c_i so angepasst, dass eine bestmögliche Übereinstimmung mit den gegebenen Messwerten erreicht wird.

Liegen wiederum m Messwerte für x , y und $z \approx f(x,y)$ vor, so können die Matrizen für alle Messpunkte direkt angegeben werden:

$$\begin{bmatrix} 1 & x_1 & y_1 & x_1y_1 & x_1^2 & y_1^2 \\ 1 & x_2 & y_2 & x_2y_2 & x_2^2 & y_2^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & y_m & x_my_m & x_m^2 & y_m^2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_6 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} \quad (2.4.21)$$

Die Berechnung der optimalen Koeffizienten erfolgt wieder mit $\mathbf{x} = \mathbf{A} \setminus \mathbf{b}$.

Nachstehend ist ein MATLAB-Skript aufgeführt, mit dem sich die Näherung (2.4.20) modellieren lässt. Mangels realen Messwerten, wurden 10 Messpunkte aus Abbildung 2-9 abgelesen und so unterscheidet sich das Regressionsmodell in Abbildung 2-10 doch teils erheblich von Abbildung 2-9

```
% VCV_CRD_Ansteuerung

% f(x,y) = c1 + c2*x + c3*y + c4*x*y+ c5*x^2 + c6*y^2

clear all
close all
% abgelesene Messwerte:
n(1)=800;  mf(1)=21;  %(links vorne)
n(2)=800;  mf(2)=60;  %(links hinten)
```

```

n(3)=1500; mf(3)=10; % (rechts vorne)
n(4)=4000; mf(4)=18; % (rechts hinten)
n(5)=800; mf(5)=8; % (mitte vorne)
n(6)=4700; mf(6)=60; % (mitte hinten)
n(7)=3000; mf(7)=50; % (mitte links)
n(8)=5000; mf(8)=40; % (mitte rechts)
n(9)=800; mf(9)=40; % (links mitte)
n(10)=3000; mf(10)=14; % (rechts mitte)
n(11)=2000; mf(11)=40; % (mitte mitte)

n=n'; % Matrix A braucht Spaltenvektoren
mf=mf';

b = [15 22 15 22 17 38 34 36 15 17 26]';

A = [ones(length(n),1) n mf n.*mf n.^2 mf.^2];
% A=[ones(length(n),1) n' mf' (n.*mf)' (n.^2)' (mf.^2)']; Alternative
% A=[ones(1,length(n)); n; mf; n.*mf; n.^2; mf.^2]'; % Alternative

x = A\b;

N=[800:100:5000];
MF=[0:2:80];

for i=1:length(N)
    for j=1:length(MF)
        VCV(i,j)= x(1) + x(2)*N(i) + x(3)*MF(j) + x(4)*N(i)*MF(j)+...
            x(5)*N(i)^2 + x(6)*MF(j)^2;
    end
end
surf (N, MF, VCV')
xlim([0 5000])
xlabel('n')
ylabel('mf')
zlabel('VCV')

```

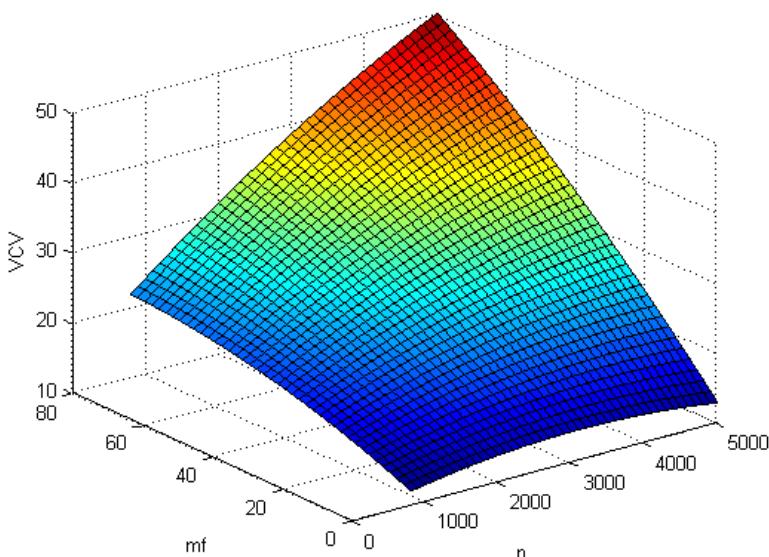


Abbildung 2-10: Approximiertes Kennfeld einer Diesel Common Rail Pumpe mit abgelesenen Messwerten aus Abbildung 2-9

3 Nichtlineare Probleme

3.1 Optimierung

3.1.1 Allgemeines

Viele Bauteile oder Maschinen werden nach vorgegebenen Richtlinien entworfen und gebaut. In der Regel begnügt man sich mit suboptimalen Lösungen, die dann überarbeitet und verbessert werden. Bleiben die Randbedingungen unverändert, dann entsteht so im Laufe der Zeit eine "optimale" Lösung.

Gelingt es, die gewünschten Ziele mathematisch so zu beschreiben, dass die Lösung durch das Minimum eines Gütekriteriums

$$G_K(p_1, p_2, p_3, \dots, p_n) \rightarrow \text{Minimum} \quad (3.1.1)$$

festgelegt ist, dann liefert die Lösung von (3.1.1) einen Satz optimaler Parameter

$$p_{1\text{ opt}}, p_{2\text{ opt}}, p_{3\text{ opt}}, \dots, p_{n\text{ opt}}. \quad (3.1.2)$$

In das Gütekriterium werden nur die n freien Systemparameter $p_1, p_2, p_3, \dots, p_n$, d.h. konstruktiv oder in der Auslegung abänderbare Parameter aufgenommen. In vielen Fällen lassen sich zudem die Parameter nur in gewissen Grenzen verändern

$$p_{i,\text{min}} \leq p_i \leq p_{i,\text{max}}, i = 1(1)n. \quad (3.1.3)$$

Bei der Lösung von (3.1.1) müssen demnach auch Randminima beachtet werden.

Hängt das Gütekriterium nur von einem ($n = 1$) oder zwei ($n = 2$) Parametern ab, dann kann das Gütekriterium grafisch veranschaulicht werden. Für $n \geq 3$ ist dies nicht mehr möglich. Optimale Parameter können hier in der Regel nur mehr durch eine numerische Lösung bestimmt werden. Dazu wird das Problem (3.1.1) in der Form

$$g(x) \rightarrow \text{Minimum} \quad (3.1.4)$$

angeschrieben, wobei g eine skalare Funktion ist und die Variablen im Vektor x zusammengefasst sind.

In MATLAB wird ein direktes Suchverfahren zur Lösung von (3.1.4) verwendet. Der Aufruf erfolgt durch

$$\mathbf{x} = \text{fminsearch}(\text{'PROBLEM'}, \mathbf{x0}) \quad (3.1.5)$$

wobei dem Programm über das M-File **PROBLEM.M** die Funktion $g(x)$ zur Verfügung gestellt werden muss. Die Startwerte werden durch den Vektor **x0** übergeben.

3.1.2 Optimales Fachwerk

3.1.2.1 Freie Parameter

Im Abschnitt 2.3.1 wurde ein einfaches Fachwerk behandelt. Nimmt man an, dass die äußeren Abmessungen a und h festliegen, dann kann noch die Lage von Knoten III

variiert werden. Die freien Parameter p und q sind dabei sinnvollerweise durch $0 < p < a$ und $q < h$ beschränkt.

3.1.2.2 Materialverbrauch

Bei nicht allzu großer Belastung des Fachwerks liegt es nahe, den Materialverbrauch zu minimieren. Verwendet man Stäbe mit gleichem Querschnitt, dann ist der Materialverbrauch durch die Gesamtlänge der Stäbe

$$g_M = \sum_1^5 l_i = h + a + \sqrt{p^2 + (h-q)^2} + \sqrt{p^2 + q^2} + \sqrt{(a-p)^2 + (h-q)^2} \quad (3.1.6)$$

gegeben. Aus der Forderung nach einem minimalen Materialverbrauch

$$g_M = l(p, q) \rightarrow \text{Minimum} \quad (3.1.7)$$

können nun die Parameter p und q berechnet werden.

Die MATLAB-Funktion

```
function g_M = g_M_f(x)
global a h % feste und freie Parameter

p = max(0,min(x(1),a)); q = max(0,min(x(2),h));
%
% Stablaengen
sl = [a, h, sqrt(p^2+(h-q)^2), sqrt(p^2+q^2), sqrt((a-p)^2+(h-q)^2)];
%
g_M = sum(sl); % Kriterium (Gesamtlaenge)
```

berechnet die Gesamtlänge der Stäbe abhängig von den Fachwerkparametern. Die freien Parameter p und q werden dabei dem Vektor \mathbf{x} entnommen. Die fixen Größen a und h werden über "globale" Variable aus dem Hauptprogramm übernommen.

```
% Skript „g_M_h Materialverbrauch“
clear all
close all
global a h
F=1.0; % [N] wirksame Kraft der angehängten Masse
a=3.0; % [m] Feldbreite
h=1.5; % [m] Feldhoehe
%
% Startwerte fuer freie Parameter
p= 2.0; % [m] x-Pos. Knoten III
q= 0.5; % [m] y-Pos. Knoten III
x0=[p, q];
%
% Optimierung
x=fminsearch('g_M_f',x0); % g_M fuer Material, g_B_L fuer Belastung
% Ausgabe der optimalen Parameter
disp(['p_opt = ', num2str(x(1)), ', q_opt = ', num2str(x(2))])
% Darstellung
p_opt=x(1); q_opt=x(2);
plot([0 0], [0 h], 'k', 'LineWidth', 2) % Stab 1
plot([0 p_opt], [0 q_opt], 'k', 'LineWidth', 2) % Stab 4
plot([p_opt 0], [q_opt h], 'k', 'LineWidth', 2) % Stab 3
plot([p_opt a], [q_opt h], 'k', 'LineWidth', 2) % Stab 5
plot([0 a], [h h], 'k', 'LineWidth', 3) % Stab 2

xlabel('Feldbreite in m'), ylabel('Feldhoehe in m'), axis([-5 3.5 -5 2])
```

Die Anfangswerte für p und q werden im Vektor \mathbf{x}_0 zusammengefasst.

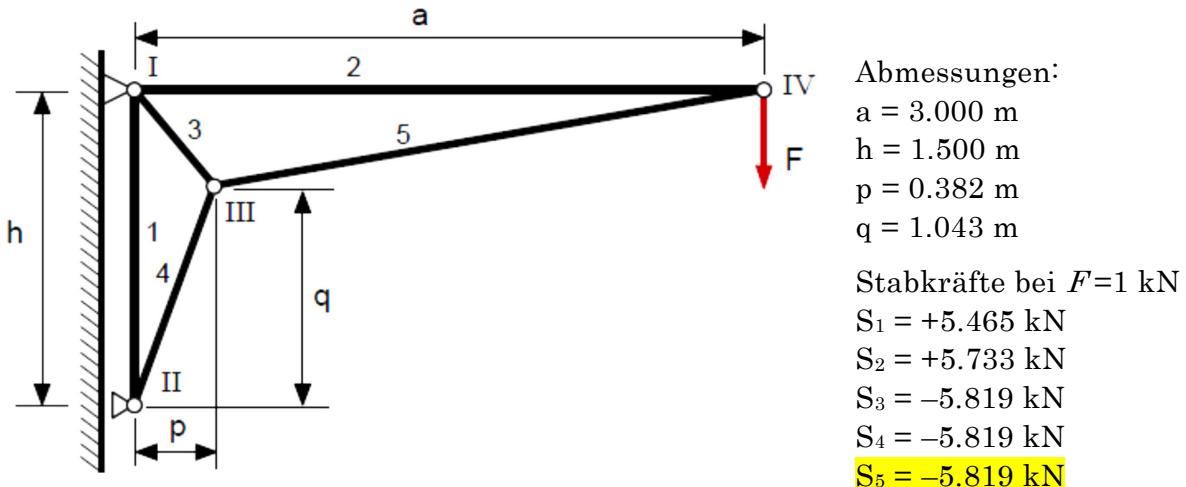


Abbildung 3-1: Fachwerk mit minimalem Materialverbrauch

Die Optimierung nach minimaler Gesamtlänge liefert ein sehr schlankes Fachwerk, das mit Stäben der Gesamtlänge $I_G = 8,864 \text{ m}$ gefertigt werden kann. Ungünstig ist, dass der recht lange **Stab 5 mit einer großen Druckkraft** belastet wird. Im Falle höherer Belastung ist folglich das genutzte Gütfunktional ungeeignet.

3.1.2.3 Belastung

Soll die Belastung des Fachwerks als Kriterium bei seiner Auslegung berücksichtigt werden, dann muss dieses in das Gütfunktional aufgenommen werden. Mit dem auf die äußere Belastung normierten Gütfunktional

$$g_B = \sum_i^5 \left(\frac{S_i - F}{F} \right)^2 \quad \text{und} \quad g_B \rightarrow \min \quad (3.1.8)$$

werden die Stabkräfte S_i bewertet, wobei mit $S_i - F$ Druckstäbe ($S_i < 0$) etwas stärker gewichtet werden. Die entsprechende MATLAB-Funktion lautet dann:

```
function g_B = g_B_f(x)
%
global a h F % feste Parameter
p=x(1); q=x(2); % freie Parameter
%
% Hilfs-Groessen
sa = q / sqrt ( p^2 + q^2 ) ;
ca = p / sqrt ( p^2 + q^2 ) ;
sb = (h-q) / sqrt ( p^2 + (h-q)^2 ) ;
cb = p / sqrt ( p^2 + (h-q)^2 ) ;
sg = (h-q) / sqrt ( (a-p)^2 + (h-q)^2 ) ;
cg = (a-p) / sqrt ( (a-p)^2 + (h-q)^2 ) ;
%
% Koeffizientenmatrix
% H_I V_I H_II S1 S2 S3 S4 S5
A = [ 1 0 0 0 1 cb 0 0 ; ...
0 1 0 -1 0 -sb 0 0 ; ...
0 0 1 0 0 0 ca 0 ; ...
0 0 0 1 0 0 sa 0 ; ...
0 0 0 0 0 -cb -ca cg ; ...]
```

```

0 0 0 0 0 sb -sa sg ; ...
0 0 0 0 -1 0 0 -cg ; ...
0 0 0 0 0 0 -sg ] ;
%
b = [ 0; 0; 0; 0; 0; 0; 0; F ]; % Rechte Seite
%
x = A\b ; % Loesung:
%
S = ( X(4:8) - F ) / F ; % Stabkraefte normiert
%
g_B = S'*S; % Belastungs-Kriterium

```

und das angepasste Hauptprogramm:

```

% Skript „g_B_h Belastung“
clear all
close all
%
global a h % feste Parameter als globale Variable
global F; % für Belastungsoptimum
F=1.0;
a=3.0; % [m] Feldbreite
h=1.5; % [m] Feldhoehe
%
% Startwerte fuer freie Parameter
p= 2.0; % [m] x-Pos. Knoten III
q= 0.5; % [m] y-Pos. Knoten III
x0=[p, q];
%
% Optimierung
x=fminsearch('g_B_f',x0); % g_M fuer Material, g_B fuer Belastung
% Ausgabe der optimalen Parameter
disp(['p_opt = ', num2str(x(1)), ', q_opt = ', num2str(x(2))])
% Darstellung siehe g_M_h

```

Als Ergebnis erhält man ein recht gedrungenes Fachwerk, Abbildung 3-2. Die maximale Stabbelastung beträgt jetzt nur noch etwa das 2-fache der Belastung F . Allerdings wird auch hier ein langer Stab mit einer großen Druckkraft ($S_4 = -2.056 \text{ kN}$) belastet.

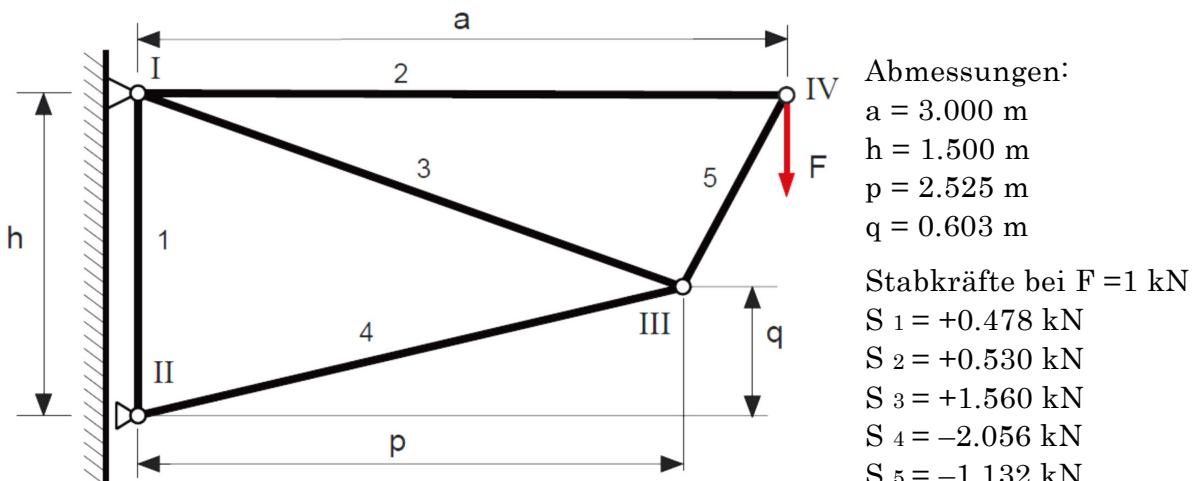


Abbildung 3-2: Fachwerk mit minimaler Stabbelastung

Um die Knickgefahr zu reduzieren, müssen die Stablängen mit berücksichtigt werden. Nach Euler steigt die Knickgefahr proportional zum Quadrat der Stablänge. Gewichtet

man die Stabkräfte mit dem Quadrat der Stablängen, dann erhält man analog zu (4.1.8) mit

$$g_K = \sum_i^5 \left(\frac{S_i - F}{F} l_i^2 \right)^2 \quad \text{und} \quad g_K \rightarrow \min \quad (3.1.9)$$

ein Gütekriterium, das als Lösung ein Fachwerk erzeugt, bei dem die Länge des Stabes mit maximaler Druckbelastung ($S_4 = -2.038 \text{ kN}$) deutlich reduziert ist, siehe Abbildung 3-3.

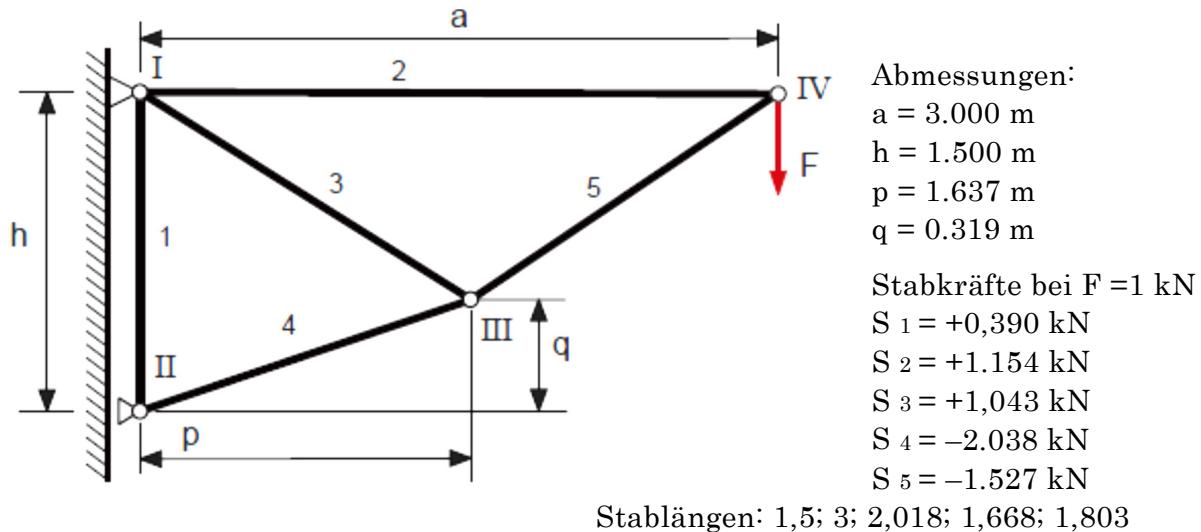


Abbildung 3-3: Fachwerk mit minimaler Knickgefahr

Die Änderungen gegenüber `function g_B_f` lauten:

```

function g_B_L = g_B_L_f(x)
...
x = A\b ;
%
% Stablängen
L=[h
    a
    sqrt(p^2+(h-q)^2)
    sqrt(p^2+q^2)
    sqrt((a-p)^2+(h-q)^2)] ;

%
% Stabkräfte normiert
S = ( X(4:8) - F ) / F ;
%
% Belastungs-Kriterium
g_B_L = sum((S.*L.^2).^2) ;

```

3.1.3 Gleichgewichtslage

Zwei Lampen mit den Gewichten P und Q sind an 5 Drähten aufgehängt, siehe Abbildung 3-4. Die Aufhängungspunkte befinden sich in gleicher Höhe und bilden ein Rechteck mit den Kantenlängen a und b . Unbelastet haben die Drähte die Längen $L_{01} = L_{02}$

$= L_{03} = L_{04} = L_0$ und $L_{05} = L_0/2$. Die Dehnsteifigkeit der Drähte ist durch EA (Elastizitätsmodul*Querschnittsfläche [N]) gegeben. Die Koordinaten x_P, y_P, z_P und x_Q, y_Q, z_Q beschreiben die Positionen der Lampen gegenüber dem Koordinatenursprung 0.

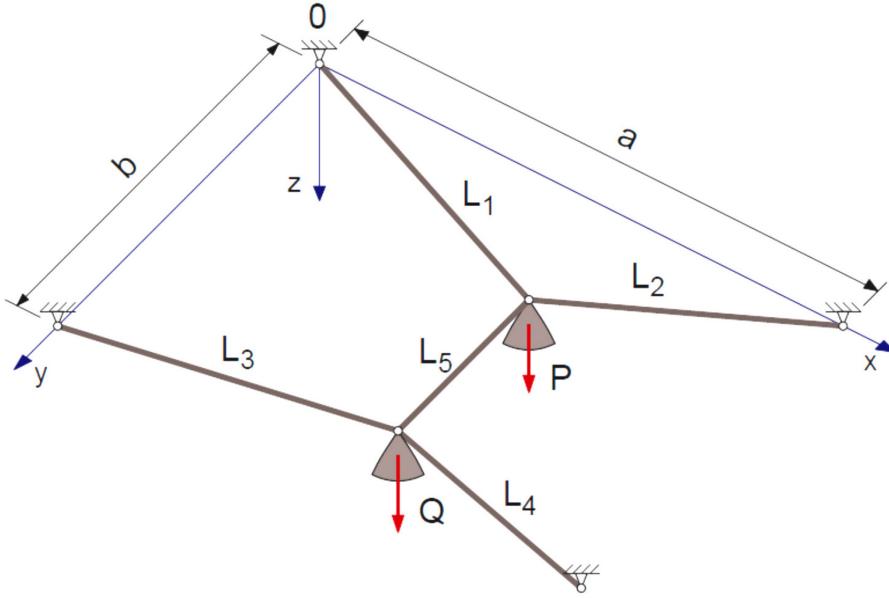


Abbildung 3-4: Gleichgewichtslage

Das System ist im Gleichgewicht, wenn die potentielle Energie minimal wird

$$E_{pot} \rightarrow \text{Min} \quad (3.1.10)$$

Die potentielle Energie des Systems errechnet sich aus der Verformungsenergie der Drähte und dem Potenzial der Gewichtskräfte

$$E_{pot} = \sum_{i=1}^5 \left(\frac{1}{2} \frac{EA}{L_{0,i}} (L_i - L_{0,i})^2 \right) - P \cdot z_P - Q \cdot z_Q, \quad (3.1.11)$$

wobei die Längen der Drähte durch

$$\begin{aligned} L_1 &= \sqrt{x_P^2 + y_P^2 + z_P^2}, & L_2 &= \sqrt{(a - x_P)^2 + y_P^2 + z_P^2} \\ L_3 &= \sqrt{x_Q^2 + (b - y_Q)^2 + z_Q^2}, & L_4 &= \sqrt{(a - x_Q)^2 + (b - y_Q)^2 + z_Q^2} \\ L_5 &= \sqrt{(x_Q - x_P)^2 + (y_Q - y_P)^2 + (z_Q - z_P)^2} \end{aligned} \quad (3.1.12)$$

gegeben sind.

Die MATLAB-Funktion

```
function E_pot=E_pot_f(x)
%
global P Q a b L0 EA
%
% umspeichern
xP=x(1); yP=x(2); zP=x(3);
xQ=x(4); yQ=x(5); zQ=x(6);
%
L1 = sqrt(xP^2+yP^2+zP^2);
```

```

L2 = sqrt((a-xP)^2+yP^2+zP^2);
L3 = sqrt(xQ^2+(b-yQ)^2+zQ^2);
L4 = sqrt((a-xQ)^2+(b-yQ)^2+zQ^2);
L5 = sqrt((xQ-xP)^2+(yQ-yP)^2+(zQ-zP)^2);
%
E_pot = 0.5*EA/L0*...
( (L1-L0)^2 ...
+ (L2-L0)^2 ...
+ (L3-L0)^2 ...
+ (L4-L0)^2 ...
+ 2*(L5-L0/2)^2 ) ...
- ( P*zP + Q*zQ );

```

berechnet die potenzielle Energie in Abhängigkeit der momentanen Lampen-Positionen. Die Konstanten $P\ Q\ a\ b\ L_0\ EA$ werden über globale Variable aus dem Hauptprogramm

```

clear all, close all

global P Q a b L0 EA

P=120; % [N] Gewicht Lampe 1
Q=150; % [N] Gewicht Lampe 1
a=6.0; % [m] Abspannweite
b=4.0; % [m] Abspannbreite
L0=3.25; % [m] unverformte Seillaenge
EA=400000; % [N] Dehnsteifigkeit

%Startwerte (keine Seildehnung)
xP=a/2;
yP=(b-L0/2)/2;
if xP^2+yP^2 < L0^2
zP=sqrt(L0^2-xP^2-yP^2);
else
zP=0;
end
xQ=a/2;
yQ=b-(b-L0/2)/2;
zQ = zP;

% Gleichgewicht (minimale pot. Energie)
x0 = [xP yP zP xQ yQ zQ];
x = fminsearch('E_pot_f',x0);

disp('Positionsaenderung Lampe P von'), disp(x0(1:3)),
disp('nach'), disp(x(1:3))
disp('Positionsaenderung Lampe Q von'), disp(x0(4:6)),
disp('nach'), disp(x(4:6))

```

übernommen. Dort werden auch geeignete Startwerte berechnet.

3.2 Nichtlineare Gleichungen

3.2.1 Beispiel: Kreuzprofil

3.2.1.1 Problem

Die Form eines Kreuzprofils ist in Abbildung 3-5 skizziert. Die Abmessungen b , h und t sollen nun so angepasst werden, dass die Querschnittsfläche A und die Flächenträgheitsmomente I_{yy} , I_{zz} vorgegebene Werte annehmen.

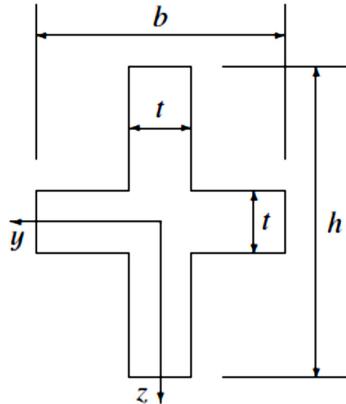


Abbildung 3-5: Kreuzprofil

Die Querschnittsfläche A und die Flächenträgheitsmomente I_{yy} , I_{zz} lassen sich mit folgenden Gleichungen berechnen.

$$A = bt + ht - t^2$$

$$I_{yy} = \frac{1}{12}th^3 + \frac{1}{12}(b-t)\cdot t^3 \quad (3.2.1)$$

$$I_{zz} = \frac{1}{12}tb^3 + \frac{1}{12}(h-t)\cdot t^3$$

Aufgrund der nichtlinearen Terme (quadratisch und kubisch) handelt es sich um ein nichtlineares Gleichungssystem und die Beziehungen können nicht mehr direkt nach den Unbekannten b , h und t aufgelöst werden.

3.2.1.2 Normierung

Da die erste Gleichung in (3.2.1) die Einheit mm^2 besitzt, die beiden anderen aber mm^4 , können extreme Unterschiede in der Größenordnung auftreten. Eine Normierung (einheitenfrei, dimensionslos) der Gleichungen schafft Abhilfe. Dividiert man die erste Gleichung durch A , dann kann sie wie folgt angeschrieben werden

$$1 = \frac{b}{\sqrt{A}} \frac{t}{\sqrt{A}} + \frac{h}{\sqrt{A}} \frac{t}{\sqrt{A}} - \frac{t}{\sqrt{A}} \frac{t}{\sqrt{A}}. \quad (3.2.2)$$

Mit den normierten, das heißt einheitenfreien Abmessungen

$$b_N = \frac{b}{\sqrt{A}}, \quad h_N = \frac{h}{\sqrt{A}}, \quad t_N = \frac{t}{\sqrt{A}} \quad (3.2.3)$$

können dann alle Gleichungen in dimensionsloser Form angeschrieben werden wobei die zweite und dritte Gleichung noch mit dem Faktor 12 durchmultipliziert

$$\begin{aligned} 1 &= b_N t_N + h_N t_N - t_N^2 , \\ \tilde{I}_{yy} &= t_N h_N^3 + (b_N - t_N) \cdot t_N^3 , \\ \tilde{I}_{zz} &= t_N b_N^3 + (h_N - t_N) \cdot t_N^3 , \end{aligned} \quad (3.2.4)$$

und die Abkürzungen

$$\tilde{I}_{yy} = 12 \frac{I_{yy}}{A^2} \quad \text{und} \quad \tilde{I}_{zz} = 12 \frac{I_{zz}}{A^2} , \quad (3.2.5)$$

eingeführt wurden.

3.2.1.3 Vektorschreibweise

Fasst man die Unbekannten, hier die normierten Abmessungen b_N , h_N und t_N , in einem Vektor x zusammen, dann kann das nichtlineare Gleichungssystem (3.2.4) in der Form

$$f(x) = 0 \quad (3.2.6)$$

angeschrieben werden. Man erhält

$$x = \begin{bmatrix} b_N \\ h_N \\ t_N \end{bmatrix} \quad \text{und} \quad f(x) = \begin{bmatrix} b_N t_N + h_N t_N - t_N^2 - 1 \\ t_N h_N^3 + (b_N - t_N) \cdot t_N^3 - \tilde{I}_{yy} \\ t_N b_N^3 + (h_N - t_N) \cdot t_N^3 - \tilde{I}_{zz} \end{bmatrix} \quad (3.2.7)$$

Zur iterativen Lösung von nichtlinearen Gleichungssystemen der Form (3.2.6) gibt es mehrere Lösungsverfahren, die anwendungsbezogen ausgewählt werden müssen.

3.2.1.4 Näherungslösung

Näherungslösungen bieten sich immer dann an, wenn Teile einer Gleichung einen nur geringen Anteil am Ergebnis haben.

Für ein sehr schlankes Profil, also $t \ll b$ und $t \ll h$ bleibt

$$\begin{aligned} 1 &\approx b_N t_N + h_N t_N , \\ \tilde{I}_{yy} &\approx t_N h_N^3 , \\ \tilde{I}_{zz} &\approx t_N b_N^3 . \end{aligned} \quad (3.2.8)$$

Aus der ersten Beziehung erhält man sofort

$$t_N \approx \frac{1}{b_N + h_N} \quad (3.2.9)$$

und die Division der zweiten mit der dritten Beziehung liefert

$$\frac{h_N^3}{b_N^3} \approx \frac{\tilde{I}_{yy}}{\tilde{I}_{zz}} \quad \text{bzw.} \quad \frac{h_N}{b_N} \approx \sqrt[3]{\frac{\tilde{I}_{yy}}{\tilde{I}_{zz}}} \quad (3.2.10)$$

Mit (3.2.9) lautet die dritte Beziehung in (3.2.8)

$$\tilde{I}_{zz} \approx \frac{1}{b_N + h_N} b_N^3 \quad \text{bzw.} \quad b_N^3 \approx (b_N + h_N) \cdot \tilde{I}_{zz} \quad (3.2.11)$$

Dividiert man noch durch b_N , dann bleibt

$$b_N^2 \approx \left(1 + \frac{h_N}{b_N}\right) \cdot \tilde{I}_{zz}, \quad (3.2.12)$$

wobei das Verhältnis h_N/b_N durch (3.2.10) gegeben ist.

$$b_N^2 \approx \left(1 + \sqrt[3]{\frac{\tilde{I}_{yy}}{\tilde{I}_{zz}}}\right) \cdot \tilde{I}_{zz}, \quad (3.2.13)$$

Mit (3.2.12), (3.2.10) und (3.2.9) können dann zumindest Näherungswerte angegeben werden.

Sind für die Querschnittsfläche und die Flächenträgheitsmomente die Zahlenwerte

$$A = 150 \text{ mm}^2; I_{yy} = 3500 \text{ mm}^4; I_{zz} = 1500 \text{ mm}^4 \quad (3.2.14)$$

gegeben, erhält man als Näherungslösung für

$$b^{(0)} = 16.7 \text{ mm}; h^{(0)} = 22.2 \text{ mm}; t^{(0)} = 3.86 \text{ mm}. \quad (3.2.15)$$

Zur Probe eingesetzt in (3.2.1), ergibt das die Profil-Eigenschaften

$$A^{(0)} = 135.25 \text{ mm}^2; I_{yy}^{(0)} = 3581 \text{ mm}^4; I_{yy}^{(0)} = 1586 \text{ mm}^4. \quad (3.2.16)$$

Die Flächenträgheitsmomente werden recht gut angenähert; die Querschnittsfläche ist allerdings um 10% zu klein.

3.2.2 Indirekte Lösung

3.2.2.1 Minimaler Fehler

Setzt man in ein gegebenes nichtlineares Gleichungssystem eine gefundene Näherungslösung $x = x_N$ ein (siehe vorigen Abschnitt), so lässt sich mit dem Spaltenvektor $\epsilon = \epsilon(x)$, der die Abweichungen der einzelnen Gleichungen von der exakten Lösung angibt (= Fehler), eine Funktion

$$f(x) = \epsilon \quad (3.2.17)$$

formulieren. Mit

$$\epsilon(x)^T \epsilon(x) = f(x)^T f(x) = g(x) \rightarrow \text{Minimum} \quad (3.2.18)$$

wird die Summe der Fehlerquadrate minimiert. Als notwendige Bedingung für ein Minimum muss dann

$$\frac{d g(x)}{dx} = 2f(x)^T \frac{d f(x)}{dx} = 0 \quad (3.2.19)$$

gelten, was durch $f(x) = 0$ oder $d f(x)/dx = 0$ erfüllt wird.

Unter der Voraussetzung, dass die gesuchte Lösung x zwar mit $f(x) = 0$ das nichtlineare Gleichungssystem erfüllt, nicht aber gleichzeitig die Funktionalmatrix zum Verschwinden bringt $d f(x)/dx \neq 0$, ist $g(x) \rightarrow \text{Minimum}$ gleichbedeutend mit $f(x) = 0$.

Damit können die Methoden zur Minimierung einer skalaren Funktion zur Lösung nichtlinearer Gleichungssysteme herangezogen werden.

3.2.2.2 Beispiel

Das Beispiel zeigt die erweiterte Anwendung der MATLAB-Routine **fminsearch**. Die Zuweisung nach `[x,fval,exitflag,output]=fminsearch(...)` liefert in **x** die Eingangsparameter, die zum Minimum führen.

Das gefundene Minimum selbst wird in **fval** ausgegeben.

Im **exitflag** ist die Abbruchbedingung kodiert. Z.B. zeigt **exitflag = 1**, dass der Algorithmus erfolgreich konvergiert ist, hierbei kann die maximale Auflösung des Datentyps als Grenze dienen oder ein in der Option **TolX** (zulässige Toleranz) übergebener Wert.

Die Variable **output** liefert eine Struktur mit weiteren Informationen zur Simulation. Beispielsweise wird in **OUTPUT.iterations** die Anzahl durchlaufener Iterations schritte angegeben.

```
% Kreuz-Profil (Hauptprogramm)

clear all
%
% Daten
A=150; I_yy=3500; I_zz=1500;
%
% Normierung
global I_yy_N
I_yy_N = 12*I_yy/A^2;
global I_zz_N
I_zz_N = 12*I_zz/A^2;
%
% Startwerte t<<h, t<<b
b_N = sqrt(I_zz_N*(1+(I_yy_N/I_zz_N)^(1/3))); % (4.2.13)
h_N = b_N * (I_yy_N/I_zz_N)^(1/3); % (4.2.10)
t_N = 1 / ( b_N + h_N ); % (4.2.9)
%
disp(' ')
disp('Direktes-Such-Verfahren')
disp(' ')
[x,fval,exitflag,output] = fminsearch('kreuzp_f',[b_N h_N t_N]);
disp(['iter=',num2str(output.iterations)])
%
disp(' ')
disp('Profil-Abmessungen')
b=x(1)*sqrt(A);h=x(2)*sqrt(A);t=x(3)*sqrt(A);
```

```

disp(['b=',num2str(b)])
disp(['h=',num2str(h)])
disp(['t=',num2str(t)])
disp('')
disp('Profil-Eigenschaften')
disp([' A=',num2str(b*t + h*t - t^2)])
disp(['I_yy=',num2str((t*h^3+(b-t)*t^3)/12)])
disp(['I_zz=',num2str((t*b^3+(h-t)*t^3)/12)])

function kreuzp_f=kreuzp_f(x)
%
global I_yy_N
global I_zz_N
%
% umspeichern
b=x(1); h=x(2); t=x(3);
%
% Gleichungen
f=[( b*t+h*t-t^2 - 1 );...
(t*h^3+(b-t)*t^3-I_yy_N);...
(t*b^3+(h-t)*t^3-I_zz_N)];
%
% Funktional (ohne Faktor 1/2)
kreuzp_f = f'*f ;

```

Nach **iter=69** Iterationen erhält man folgende Ergebnisse:

Direktes-Such-Verfahren

iter=69

Profil-Abmessungen

b=14.7159

h=20.0677

t=5.0436

Profil-Eigenschaften

A=149.9957

I_yy=3500.0247

I_zz=1500.0551

Die geforderten Profil-Eigenschaften werden nun sehr gut erreicht.

4 Dynamische Probleme

Dynamische Probleme sind zeitabhängige Prozesse und werden in mathematische Modellen sog. dynamischen Systemen formuliert. Diese sind homogen bezüglich der Zeit, d. h. ihr Verlauf hängt nur vom Anfangszustand, nicht aber vom Anfangszeitpunkt ab.

4.1 Allgemeines

Zeitabhängiger Prozesse werden durch Differentialgleichungen also durch Ableitungen beschrieben. Differentialgleichungen beschreiben also die zeitliche Änderung von Größen. Die Lösung einer solcher Differentialgleichung liefert folglich eine Gleichung, die den Verlauf der eigentlichen Größe über der Zeit beschreibt.

4.1.1 Typen

Gewöhnliche Differentialgleichungen höherer Ordnung können durch Substitution der höheren Ableitungen stets in ein System gewöhnlicher Differentialgleichungen erster Ordnung umgeschrieben werden.

Partielle Differentialgleichungen lassen sich durch spezielle Ansatzfunktionen (z.B. Methode der Finiten Elemente) in gewöhnliche Differentialgleichungen umformen. Ersetzt man nun noch die Ableitungen durch Differenzenquotienten, dann können partielle Differentialgleichungen über ein lineares Gleichungssystem gelöst werden.

Bei den gewöhnlichen Differentialgleichungen unterscheidet man zwischen Anfangswertproblemen

$$\dot{x} = f(x) \quad \text{für } t_0 \leq t \leq t_E \quad \text{mit} \quad x(t = t_0) = x_0 \quad (4.1.1)$$

und Randwertproblemen

$$\dot{x} = f(x) \quad \text{für } t_0 \leq t \leq t_E \quad \text{mit} \quad x(t = t_E) = x_E \quad (4.1.2)$$

Bei Anfangswertproblemen ist der Anfangszustand $x(t = t_0)$ gegeben. Gesucht ist der Lösungsverlauf $x = x(t)$ im Intervall $t_0 \leq t \leq t_E$. Solche Probleme können numerisch durch schrittweise Berechnung von $x(t)$ gelöst werden.

Beispiel: Welche Bahn durchfliegt ein Stein und wo trifft er auf, wenn er an der Stelle x_0, y_0 mit der Geschwindigkeit \dot{x}_0, \dot{y}_0 abgeworfen wird?

Anders ist es bei Randwertproblemen.

Beispiel: An welcher Stelle x_0, y_0 und mit welchen Geschwindigkeitskomponenten \dot{x}_0, \dot{y}_0 muss ein Stein abgeworfen werden, damit er bei x_E, y_E mit \dot{x}_E, \dot{y}_E auftrifft?

Hier ist eine direkte Berechnung nur dann möglich, wenn die Lösung $x(t)$ explizit angegeben werden kann, wie zum Beispiel beim schießen Wurf ohne Luftwiderstand (Wurfparabel).

In den meisten Fällen können jedoch die Differentialgleichungen nicht mehr analytisch gelöst werden z.B. beim schießen Wurf mit Luftwiderstand. In solchen Fällen können Randwertprobleme nur mehr iterativ gelöst werden.

Ausgehend von beliebigen Startwerten variiert man dabei die Anfangswerte so lange, bis der Lösungsverlauf auf den gegebenen Endzustand führt.

Damit hat man die Lösung des Randwertproblems auf die Lösung von Anfangswertproblemen zurückgeführt.

4.2 Numerische Integrationsverfahren

Während man mittels analytischer (symbolischer) Mathematik für die Lösung einer Differentialgleichung die zugehörige Stammfunktion sucht und anschließend durch Einsetzen der Integrationsgrenzen den Wert der gesuchten Größe für ein Intervall berechnet, bestimmt man mit numerischen Integrationsverfahren den gesuchten Wert der Größe näherungsweise und – bei größeren Zeiträumen – Schritt für Schritt.

Beispiel: Um die Geschwindigkeit $v(t)$ zu erhalten, muss über die Beschleunigung $a(t)$ integriert werden.

$$v(t_1) = v(t_0 = 0) + \int_{t=0}^{t=t_1} a(t) dt \quad 4.2.1$$

Analytisch müsste dazu die Stammfunktion zu $a(t)$ bekannt sein, welche aber, je nach Verlauf von $a(t)$, evtl. sehr schwer zu finden ist oder schlicht nicht existiert. Letzteres ist dann der Fall, wenn die Beschleunigungswerte nur als Messdaten über der Zeit vorliegen.

4.2.1 Integrationsverfahren: Treppenfunktion

Eine numerische Näherung für das Beispielintegral 4.2.1 ist z.B. die Treppenfunktion. Der betrachtete Zeitraum wird dazu in N gleich lange Zeitschritte Δt unterteilt, mit $\Delta t = (t_1 - t_0)/N$ und ersetzt das Integral näherungsweise durch die Summe

$$\int_{t=0}^{t=t_1} a(t) dt \approx \left(\sum_{n=0}^{N-1} a(t_0 + n \cdot \Delta t) \right) \cdot \Delta t \quad 4.2.2$$

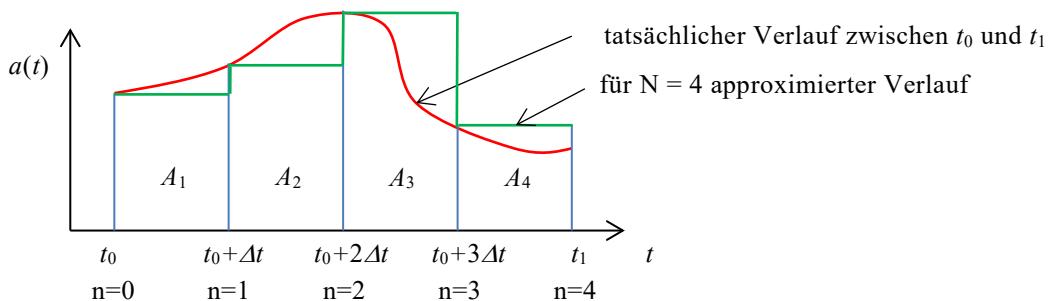


Abbildung 4-1: Treppenfunktion ; Funktionsverlauf $a(t)$; Fläche $v(t)$

Nachteil der Treppenstufenfunktion ist die teils recht große Abweichung zwischen der durch die Treppenstufen approximierten Fläche zur tatsächlichen Fläche unter der Kurve, insbesondere bei großen Gradienten und großen Schrittweiten.

Bei rechnerisch kaum mehr Aufwand als es die Treppenstufenfunktion fordert, liefert das Trapezverfahren wesentlich bessere Approximationsergebnisse.

4.2.2 Integrationsverfahren: Trapez-Verfahren

Für das Trapezverfahren werden die Teilintervalle nicht durch Treppenstufen, sondern durch Trapeze angenähert. Ganz allgemein sind Trapeze Vierecke mit zwei parallelen Seiten.

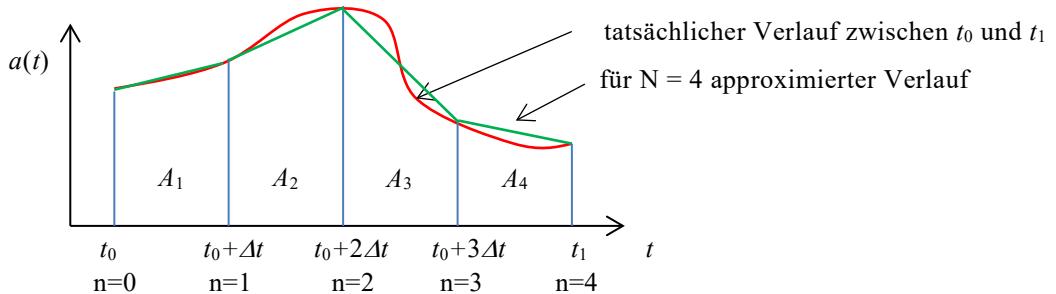


Abbildung 4-2: Trapezverfahren ; Funktionsverlauf $a(t)$; Fläche $v(t)$

Der Gesamtflächeninhalt A unter den Trapezen A_1 bis A_4 berechnet sich zu:

$$A = \frac{a(t_0) + a(t_0 + \Delta t)}{2} \cdot \Delta t + \frac{a(t_0 + \Delta t) + a(t_0 + 2\Delta t)}{2} \cdot \Delta t \dots + \frac{a(t_0 + 2\Delta t) + a(t_0 + 3\Delta t)}{2} \cdot \Delta t + \frac{a(t_0 + 3\Delta t) + a(t_0 + 4\Delta t)}{2} \cdot \Delta t \quad 4.2.3$$

In dieser Summe kommen die Beschleunigungswerte an den Grenzen zwischen den Trapezen jeweils doppelt vor. Somit lässt sich die Gleichung vereinfachend zusammenfassen zu

$$A = \left(\frac{a(t_0)}{2} + a(t_0 + \Delta t) + a(t_0 + 2\Delta t) + a(t_0 + 3\Delta t) + \frac{a(t_0 + 4\Delta t)}{2} \right) \cdot \Delta t \quad 4.2.4$$

und allgemein

$$A = \left(\frac{a(t_0) + a(t_0 + N \cdot \Delta t)}{2} + \sum_{n=1}^{N-1} a(t_0 + n \cdot \Delta t) \right) \cdot \Delta t \approx \int_{t_0}^{t_1} a(t) dt \quad 4.2.5$$

wobei im Beispiel $N = 4$ ist und die Fläche A der Geschwindigkeit v entspricht, die bei positivem $a(t)$ permanent anwächst.

Ein Vergleich der Trapezformel zur Treppenformel zeigt, dass die Summenformel der Trapezformel um einen Term kürzer ist und nur die Durchschnittsbildung der beiden Grenzen als zusätzlicher Rechenaufwand (Programmlaufzeit) hinzugekommen ist. Letztere liefert, insbesondere bei wenigen Stützstellen über den Integrationszeitraum, einen merklichen Genauigkeitsgewinn für das Integrationsergebnis.

4.2.3 Integrationsverfahren nach Simpson

Eine weitere Steigerung der Genauigkeit lässt sich mit der numerischen Integration nach Simpson erreichen. Das zu integrierende Intervall zwischen t_0 und t_1 wird dazu in eine gerade Anzahl von Teilintervallen zerlegt und durch Parabelabschnitte angenähert.

$$\rightarrow \Delta t = (t_1 - t_0)/N \text{ und! } N \text{ ist gerade}$$

Jede der Parabeln wird dabei wie in nachstehender Grafik durch drei Punkte der Funktion gelegt.

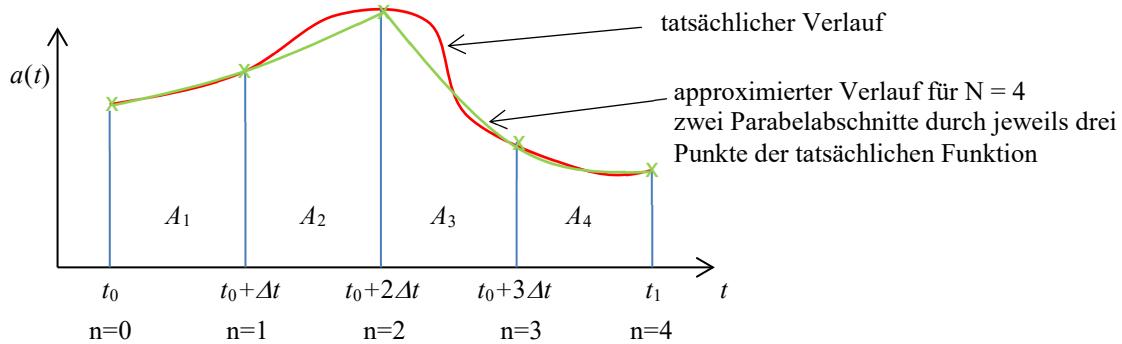


Abbildung 4-3: Simpson-Verfahren ; Funktionsverlauf $a(t)$; Fläche $v(t)$

Die (analytische) Integration des linken Parabelabschnitts t_0 bis $(t_0+2\Delta t)$ liefert

$$A_{l2} = \left[\frac{1}{3}a(t_0) + \frac{4}{3}a(t_0 + \Delta t) + \frac{1}{3}a(t_0 + 2\Delta t) \right] \cdot \Delta t \quad 4.2.6$$

und die Integration des rechten Parabelabschnitts $(t_0+2\Delta t)$ bis $(t_0+4\Delta t)$

$$A_{34} = \left[\frac{1}{3}a(t_0 + 2\Delta t) + \frac{4}{3}a(t_0 + 3\Delta t) + \frac{1}{3}a(t_1) \right] \cdot \Delta t \quad 4.2.7$$

Auch ist wieder zu erkennen, dass der letzte Summand vom vorangehenden Abschnitt und der erste Summand vom darauffolgenden identisch sind. Somit lässt sich für die Gesamtfläche A unter den beiden Parabeln im obigen Beispiel vereinfachend angeben

$$A = \left[a(t_0) + 4a(t_0 + \Delta t) + 2a(t_0 + 2\Delta t) + 4a(t_0 + 3\Delta t) + a(t_1) \right] \cdot \frac{\Delta t}{3} \quad 4.2.8$$

Verallgemeinert für N Abschnitte der Breite Δt folgt

$$A = \left[a(t_0) + 4a(t_0 + \Delta t) + 2a(t_0 + 2\Delta t) + 4a(t_0 + 3\Delta t) + 2a(t_0 + 4\Delta t) + \dots + 2a(t_0 + (N-2)\Delta t) + 4a(t_0 + (N-1)\Delta t) + a(t_1) \right] \cdot \frac{\Delta t}{3} \quad 4.2.9$$

was sich wieder durch eine gut programmierbare Formel anschreiben lässt. Dazu fasst man zuerst die geradzahligen Summanden in einer Summenformel wie folgt zusammen: $4[a(t_0 + \Delta t) + a(t_0 + 3\Delta t) + \dots + a(t_0 + (N-3)\Delta t) + a(t_0 + (N-1)\Delta t)]$

$$= 4 \sum_{n=1}^{\left(\frac{N}{2}\right)} a(t_0 + (N - (2n-1))\Delta t) \quad 4.2.10$$

Trotz der Division durch Zwei bleibt die obere Grenze vom Zählindex ganzzahlig, da die Intervall-Anzahl laut Definition der Simpson-Regel gerade sein muss. Nun fasst man die ungeradzahligen Summanden zusammen, ausgenommen den ersten und den letzten.

$$2[a(t_0 + 2\Delta t) + a(t_0 + 4\Delta t) + \dots + a(t_0 + (N-2)\Delta t)] = 2 \sum_{n=1}^{\left(\frac{N}{2}-1\right)} a(t_0 + (N - 2n)\Delta t) \quad 4.2.11$$

Für die Gesamtfläche A ergibt sich somit

$$A = \left[4 \sum_{n=1}^{\left(\frac{N}{2}\right)} a(t_0 + (N - (2n-1))\Delta t) + 2 \sum_{n=1}^{\left(\frac{N}{2}-1\right)} a(t_0 + (N - 2n)\Delta t) + a(t_0) + a(t_1) \right] \cdot \frac{\Delta t}{3}, \quad 4.2.12$$

wobei die Fläche A der Intergration nach Simpson von den drei oben beschriebenen Verfahren die potentiell besten Approximationsergebnisse für $v(t) = \int_{t_0}^{t_1} a(t) dt$ liefert.

4.2.4 Integrationsverfahren nach Euler (explizit)

Sind die DGL einer Größe $y' = f(y, x)$ sowie die Anfangsbedingung $y(x_0) = y_0$ gegeben, so lässt sich mit dem expliziten Eulerverfahren die Funktion $y(x)$ für $x \geq x_0$ finden.

Beispiel: Der Anfangswert $v(a_0) = v_0$ sei vorgegeben, also die Fallgeschwindigkeit, die ein fallender Körper bei einer Anfangsbeschleunigung a_0 besitzt.

Beim Eulerschen Verfahren geht man nun davon aus, dass sich die Änderungsgeschwindigkeit der gesuchten Größe v , also die Beschleunigung a mit der der Körper fällt, in einem sehr kleinen Zeitintervall nur unwesentlich ändert. Somit lässt sich die Fallgeschwindigkeit nach einem solch kleinen Zeitintervall in guter Näherung angeben als: $v_1 = v_0 + a_0 \cdot \Delta t$.

Nach weiteren Zeitschritten folgen $v_2 = v_1 + a_1 \cdot \Delta t$, $v_3 = v_2 + a_2 \cdot \Delta t$, etc.

Das Fallen eines Körpers unter Berücksichtigung der Luftreibung wird nach dem zweiten Newtonschen Axiom durch die DGL

$$m \cdot a = -m \cdot g + kv^2 \quad 4.2.13$$

beschrieben (mit $a = \frac{dv}{dt}$), wobei $g = 9,81 \text{ ms}^{-1}$ die Erdbeschleunigung ist und

$$k = 0,5 c_w \cdot A \cdot \rho$$

4.2.14

den Luftreibungskoeffizienten angibt, mit dem Widerstandsbeiwert c_w , der Querschnittsfläche A und der Luftdichte ρ .

In expliziter Darstellung lautet die DGL:

$$\frac{dv}{dt} = \frac{k}{m} v^2 - g \quad 4.2.15$$

Schätzt man den c_w -Wert eines Fallschirmspringers mit 0,8 ab, seine Querschnittsfläche mit $0,5 \text{ m}^2$, seine Masse mit 80 kg und die mittlere Luftdichte mit $1,1 \text{ kg/m}^3$, so ergibt sich für $k/m = 0,0275 \text{ m}^{-1}$.

Die DGL lautet somit

$$\frac{dv}{dt} = a = \frac{0,0275}{m} v^2 - 9,81 \frac{\text{m}}{\text{s}^2} \quad 4.2.16$$

Setzen wir für die Fallgeschwindigkeit den Startwert von $v_0 = 0 \text{ m/s}$, so können wir mit einem Tabellenkalkulationsprogramm die Lösungsmethode verdeutlichen.

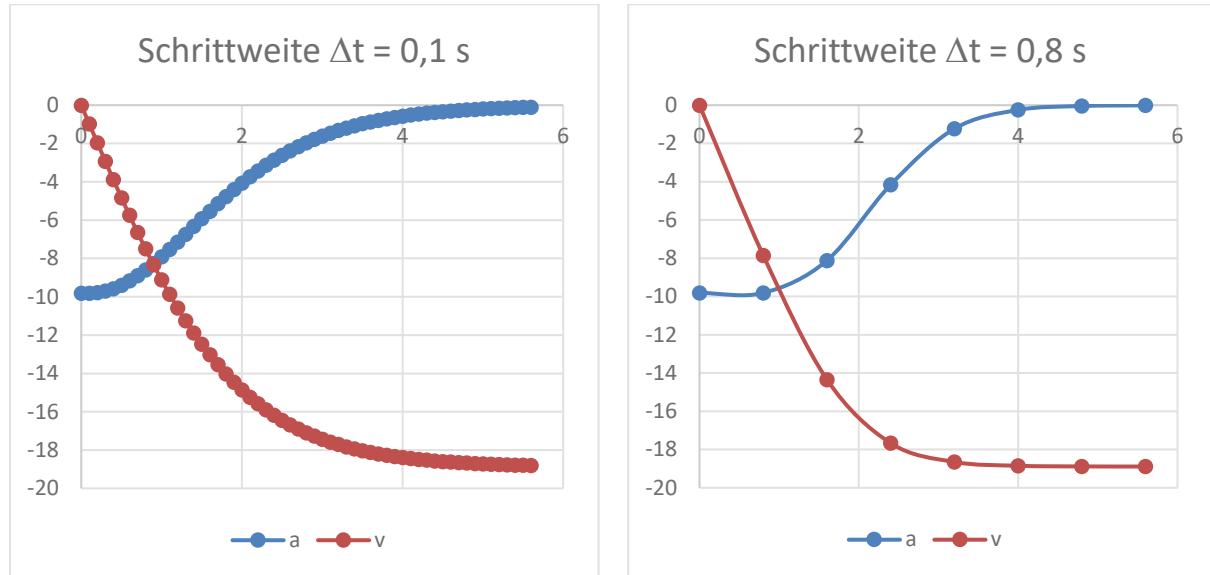


Abbildung 4-4: Simulationsergebnisse; Links für $\Delta t = 0,1 \text{ s}$ und rechts für $\Delta t = 0,8 \text{ s}$.

Auf der x-Achse ist die Zeit t in Sekunden aufgetragen und auf der y-Achse die resultierende Fallbeschleunigung a sowie die Fallgeschwindigkeit v . Beide Diagramme unterscheiden sich vor allem zu Beginn des Fallvorgangs deutlich voneinander, was im jeweiligen Fehler durch die unterschiedlichen Schrittweiten begründet ist. Die größere Schrittweite führt hier zu einem größeren Integrationsfehler.

Aber Vorsicht! Stellt man mit dem Ziel, einen möglichst kleinen Integrationsfehler zu erhalten, die Schrittweite sehr klein ein, kann dies zu zwei weiteren Problemen führen.

Eines ist Zunahme der Simulationsdauer, denn mit der Auflösung skaliert auch die Rechenzeit. Das zweite Problem liegt in der begrenzten Auflösung des verwendeten Zahlentyps, z. B. 10^{16} der Mantisse vom Typ Double. Für die Gleichung $v_{n+1} = v_n + a_n \cdot \Delta t$ mit „normalen“ Werten für v und a (Größenordnung zehn) erzeugt hier ein Zeitschritt von $\Delta t = 10^{-10}$ ein Ergebnis von nur noch einer Genauigkeit von 10^{-5} . Bei Addition sehr

vieler solcher Schritte, kummulieren diese Ungenauigkeiten zu evtl. nicht zu vernachlässigenden Rundungsfehlern. Noch kleiner Werte für Δt können sogar dafür sorgen, dass $v_{n+1} = v_n$ bleibt, was bei vorhandener Beschleunigung offensichtlich falsch ist.

Die Entwicklung moderner Lösungsverfahren hat darum unter anderem stets zum Ziel, mit möglichst wenigen Schritten ein hochgenaues Integrationsergebnis zu erhalten.

4.2.5 Lösungsverfahren in MATLAB

Zur Lösung von Anfangswertproblemen stehen in MATLAB mehrere numerische Lösungsverfahren zur Verfügung, Tabelle 4.1.

Einschritt-Verfahren	
ode23	Runge-Kutta 2./3.-Ordnung
ode45	Runge-Kutta 4./5.-Ordnung
ode23t	Trapez-Regel (implizit)
ode23tb	impliziter Runge-Kutta 2./3. Ordnung
ode23s	impliziter Rosenbrock 2./3. Ordnung

Mehrschritt-Verfahren	
ode113	Adams-Bashforth-Moulton, Ordnung 1–12 mit Prediktor-Korrektor-Formeln
ode15s	Klopfenstein-Shampine (implizit), Ordnung 1–5

Tabelle 4.1: Integrationsverfahren in MATLAB

Die Integrationsverfahren **ode...** werden mit dem MATLAB-Befehl

$$[t, xout] = \text{ode...}('dglsys', [t0, tE], x0) \quad (4.2.17)$$

aufgerufen. Sie lösen Systeme von Differentialgleichungen, die im M-File **dglsys.m** in der Form

$$\dot{x} = f(t, x) \quad (4.2.18)$$

bereitgestellt werden müssen.

Die Berechnung wird mit dem Anfangszustand $x(t_0) = x_0$ im Intervall $t_0 \leq t \leq t_E$ durchgeführt.

Die numerische Lösung erfolgt mit automatischer Schrittweitensteuerung durch Runge-Kutta-Verfahren 2./3. Ordnung, bzw. 4./5. Ordnung.

4.2.6 Klassisches Runge-Kutta-Verfahren

Das klassische Runge-Kutta-Verfahren kann angewendet werden, die Differentialgleichung $\dot{x} = f(t, x)$ bei gegebenen Anfangswerten näherungsweise zu lösen.

Von einer Zustandsgröße Z seien

1.) der aktuelle Wert $Z(t_0)$ zum Zeitpunkt t_0 bekannt, d.h.

anschaulich ein Punkt A(t_0 ; $Z(t_0)$) sowie
2.) eine Differentialgleichung

$$Z(t) = f(t; Z(t)) \quad (4.2.19)$$

für die momentane Änderungsrate $Z(t)$, d.h. für die Steigung $m = m(t) = Z(t)$ zu einem beliebigen Zeitpunkt t .

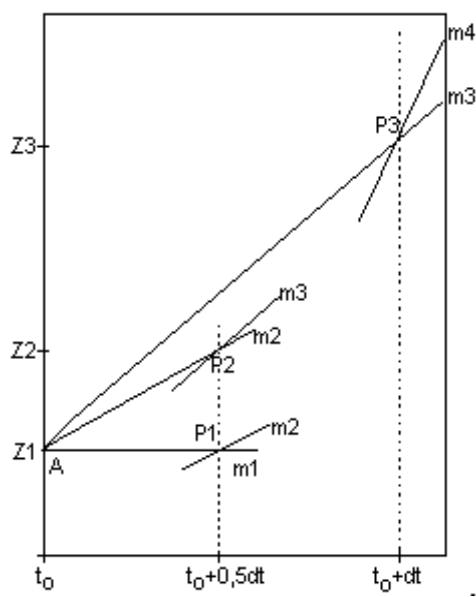
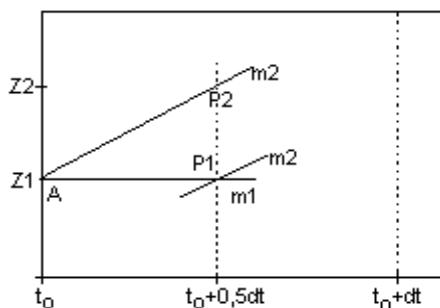
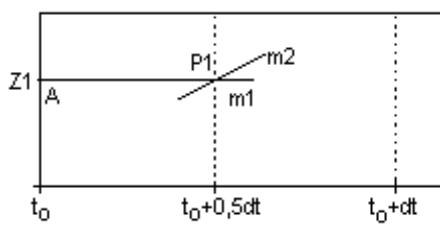
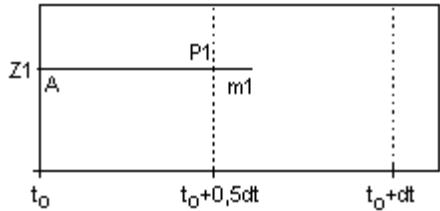
Aus der Steigung $m = m(t_0)$ im Punkt A lässt sich mit Hilfe des Steigungs dreiecks eine lineare Funktion P (Tangente im Punkt A) ermitteln, deren Wert $P(t_0+dt)$ nach dem Euler-Cauchy-Verfahren als Prognosewert für $Z(t_0+dt)$ dient. Im Allgemeinen wird jedoch der Graph von Z im betrachteten Zeitintervall dt mehr oder weniger weit von seiner Tangente abweichen, so dass der Wert $P(t_0+dt)$ nur einen recht ungenauen Prognosewert für den vorherzusagenden Wert $Z(t_0+dt)$ darstellt. Insbesondere bei langen Prognosezeiträumen (vielen Prognoseschritten) wird sich der Verfahrensfehler schnell vergrößern.

Das Runge-Kutta-Verfahren baut auf dem Euler-Cauchy-Verfahren auf. Es verbessert den gesuchten Prognosewert, indem es den für die lineare Fortschreibung $Z(t_0+dt) = Z(t_0) + m(t_0)dt$ benutzten Steigungswert m nicht nur aus der anfänglichen Steigung $Z(t_0)$ bestimmt, sondern von dem (mittels der gegebenen Differentialgleichung bekannten) weiteren Steigungsverhalten der Funktion Z abhängig macht. Dazu werden zunächst mit einer gewissen Teil-Schrittweite "Hilfsprognosen" erstellt und anschließend, über die dabei gewonnenen "Hilfssteigungen" gemittelt. Das hier dargestellte Runge-Kutta-Verfahren 4. Ordnung verwendet dazu "Halbschritte" im Zeitintervall dt und gewinnt dadurch vier "Hilfssteigungen" m_1, m_2, m_3, m_4 , deren gewichtetes arithmetisches Mittel $m = (m_1 + 2m_2 + 2m_3 + m_4)/6$, dann die zur linearen Fortsetzung der Funktion Z benutzte Steigung darstellt. Durch die doppelte Gewichtung der beiden "mittleren Hilfssteigungen" m_2 und m_3 wird dabei i.a. ein durchaus spürbarer zusätzlicher Genauigkeitsgewinn erzielt.

Zur Veranschaulichung des Verfahrens dient als Beispiel die Differentialgleichung

$$Z'(t) = Z(t) + t - 1 \quad (4.2.20)$$

mit dem Anfangswert $Z(0) = 1$ und der (untypisch großen!) Schrittweite $dt = 1$. Die eingezeichneten Geraden sind dabei jeweils mit ihrer Steigung m bezeichnet. Das Beispiel wird unten auch rechnerisch zusammengefasst.



Mit der anfänglichen Steigung $m_1 = Z(t_0)$
 $m_1 = Z(0) = Z(0) + 0 - 1 = 0$ als
1. Hilfssteigung wird – ausgehend vom gegebenen **Startpunkt A** – zunächst ein "Euler-Halbschritt" gemacht. Dadurch erhält man einen
1. Prognosehilfswert $Z_1 = Z(t_0) + m_1 * 0,5dt$.
 $Z_1 = Z(0) + m_1 * 0,5dt = 1 + 0 * 0,5 = 1$

Man unterstellt nun, der zugehörige **Punkt $P_1(t_0+0,5dt; Z_1)$** sei ein Punkt des Graphen der gesuchten Funktion Z , und errechnet durch Einsetzen vom Punkt P_1 in die gegebene Differentialgleichung die zugehörige Steigung als
2. Hilfssteigung $m_2 = f(t_0 + 0,5dt; Z_1)$.
 $m_2 = Z(0+0,5dt) = Z_1 + 0 + 0,5dt - 1 = 0,5$

Mit dieser Hilfssteigung m_2 führt man nun den ersten "Euler-Halbschritt" erneut durch und erhält daraus einen **2. Prognosehilfswert**
 $Z_2 = Z(t_0) + m_2 * 0,5dt$.

$$Z_2 = Z(0) + m_2 * 0,5dt = 1 + 0,5 * 0,5 = 1,25$$

Wieder unterstellt man, der Punkt $P_2(t_0+0,5dt; Z_2)$ sei ein Punkt des Graphen der gesuchten Funktion Z , und errechnet durch Einsetzen des Punktes P_2 in die gegebene Differentialgleichung die zugehörige Steigung als
3. Hilfssteigung $m_3 = f(t_0 + 0,5dt; Z_2)$.
 $m_3 = Z(0+0,5dt) = Z_2 + 0 + 0,5dt - 1 = 0,75$

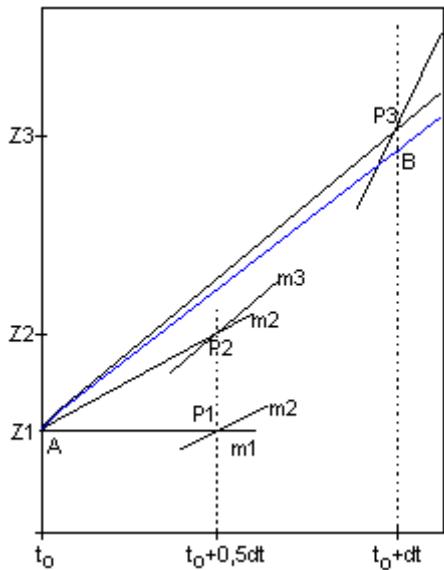
Mit dieser Hilfssteigung m_3 führt man nun einen ganzen "Euler-Schritt" durch und erhält einen

3. Prognosehilfswert $Z_3 = Z(t_0) + m_3 * dt$

$$Z_3 = Z(0) + m_3 * dt = 1 + 0,75 * 1 = 1,75$$

sowie – wieder durch Einsetzen des als Kurvenpunkt interpretierten Punktes $P_3(t_0+dt; Z_3)$ in die gegebene Differentialgleichung – die zugehörige Steigung als
4. Hilfssteigung $m_4 = f(t_0 + dt; Z_3)$.

$$m_4 = Z(0+dt) = Z_3 + 0 + dt - 1 = 1,75$$



Aus den so berechneten vier Hilfssteigungen m_1, m_2, m_3, m_4 wird nun das gewichtete arithmetische Mittel
 $\mathbf{m} = \frac{1}{6} * (m_1 + 2m_2 + 2m_3 + m_4)$
 $m = \frac{1}{6} * (m_1 + 2m_2 + 2m_3 + m_4) = 0,70833\dots$
berechnet, das dann als Prognosesteigung bei der linearen Fortsetzung der Funktion Z benutzt wird. Man erhält so beim Runge-Kutta-Verfahren 4. Ordnung den **Prognosewert $Z(t_0 + dt) = Z(t_0) + m * dt$** . Der zugehörige Punkt $B(t_0 + dt; Z(t_0 + dt))$ wird als der gesuchte "nächste" Kurvenpunkt prognostiziert und dient als Startpunkt für den nächsten Runge-Kutta-Iterationsschritt.

Wie sich die Funktion Z bzw. ihr Graph allerdings zwischen den Punkten A und B genau verhält, bleibt unbekannt und könnte nur durch eine Verfeinerung der Schrittweite näherungsweise ermittelt werden; dadurch würde sich jedoch auch der Prognosepunkt B ändern.

Schrittweite näherungsweise ermittelt werden; dadurch würde sich jedoch auch der Prognosepunkt B ändern.

Zahlenbeispiel für die DGL $Z'(t) = Z(t) + t - 1$ aus (4.1.8) mit einem Anfangswert $Z(0) = 1$ und einen Prognose-Schritt mit der Schrittweite $dt = 1$:

Die Schrittweite $dt = 1$ ist absichtlich untypisch groß gewählt - üblich wäre etwa $dt = 0,1$ mit erheblichem Genauigkeitsgewinn. Sie ist aber gut geeignet, um anhand eines Vergleiches mit dem exakten Wert und dem Prognosewert nach "Euler-Cauchy" die "Genauigkeit" des Runge-Kutta-Verfahrens zu demonstrieren.

Exakte (analytische) Berechnung

Das gegebene Anfangswertproblem hat die Lösungsfunktion $Z(t) = e^t - t$, also ist der exakte Funktionswert $Z(1) = e - 1 = 1,71828\dots$

Prognose nach Euler-Cauchy

$$m = Z(0) = Z(0) + 0 - 1 = 1 - 1 = 0$$

also ist der "Euler-Cauchy-Prognosewert" $Z(1) = Z(0) + m * dt = 1$

Prognose nach Runge-Kutta

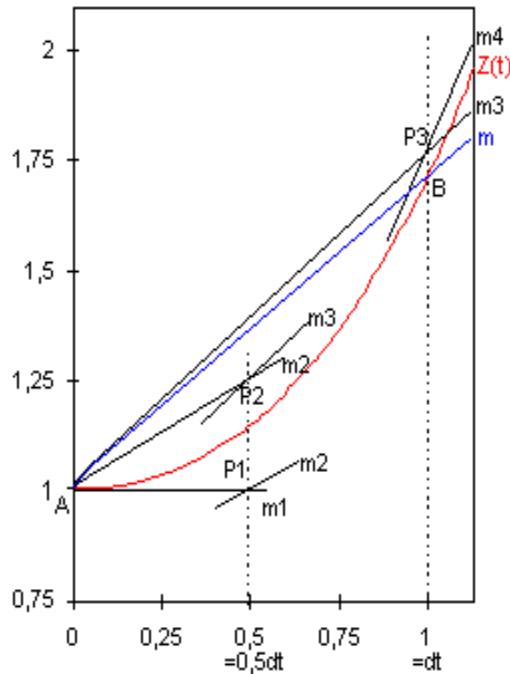
$$\begin{aligned}
m_1 &= Z(0) &= Z(0) + 0 - 1 &= 0 \\
Z_1 &= Z(0) + m_1 * 0,5dt &= 1 + 0 * 0,5 &= 1 \\
m_2 &= Z(0+0,5dt) &= Z_1 + 0 + 0,5dt - 1 &= 0,5 \\
Z_2 &= Z(0) + m_2 * 0,5dt &= 1 + 0,5 * 0,5 &= 1,25 \\
m_3 &= Z(0+0,5dt) &= Z_2 + 0 + 0,5dt - 1 &= 0,75 \\
Z_3 &= Z(0) + m_3 * dt &= 1 + 0,75 * 1 &= 1,75 \\
m_4 &= Z(0+dt) &= Z_3 + 0 + dt - 1 &= 1,75 \\
m &= \frac{1}{6} * (m_1 + 2m_2 + 2m_3 + m_4) &&= 0,70833\dots
\end{aligned}$$

Mit $dt = 1$ ergibt sich für den "Runge-Kutta-Prognosewert"

$$Z(1) = Z(0+dt) = Z(0) + m * dt = 1 + m = 1,70833\dots$$

Trotz der großen Schrittweite ist die Prognose nach Runge-Kutta bei diesem (allerdings "gutmütigen") Beispiel sehr gut.

Der verfahrensbedingte Fehler gegenüber der exakten Lösung beträgt hier nur ca. 0,01 , während er beim Euler-Cauchy-Verfahren mit ca. 0,72 sehr groß ist.



Graphische Darstellung des Runge-Kutta-Verfahrens 4. Ordnung mit eingetragener Lösungsfunktion

Draft:

Integrationsverfahren sind insbesondere dann hilfreich, wenn nichtlineare Komponenten das Bestimmen einer Stammfunktion erschweren oder gar unmöglich machen.

Das schrittweise Bestimmen des Wertes einer Größe an einem bestimmten Zeitpunkt liefert inherent den Funktionsverlauf der Größe bis zu diesem Zeitpunkt. Dies ist sehr vorteilhaft, denn oftmals ist gerade der Verlauf einer Größe von größtem Interesse.

Integrationsverfahren ist gleich durch stetes Wiederholen zweier Verfahrensschritte. Ausgehend von einem Startwert wird im ersten Schritt durch Ableiten der Funktion die Steigung im aktuellen Punkt ermittelt. Im zweiten Schritt wird durch Integration mit der ermittelten Steigung über den Zeitraum Δt der nächste Punkt auf der Funktion approximiert.

4.2.7 Autonome Systeme

Bei der Konstruktion numerischer Lösungsverfahren geht man häufig von dem autonomen System

$$\dot{x} = f(x) \quad \text{mit} \quad x(t_0) = x_0 \quad (4.2.21)$$

aus. Im Unterschied zu (4.1.6) hängt dabei die Rechte Seite nicht mehr explizit von der Zeit t ab. Die Form (4.1.3) stellt aber auch keine Einschränkung dar, da (4.1.6) durch eine einfache Erweiterung

$$\begin{bmatrix} \dot{x} \\ \dot{t} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} f(t, x) \\ 1 \\ f(z) \end{bmatrix} \quad (4.2.22)$$

stets in die Form (4.1.3) umgeschrieben werden kann.

4.3 Beispiele

4.3.1 Räuber-Beute-Modell

Dynamische Vorgänge in der Natur können häufig durch Differentialgleichungen beschrieben werden. Wenn b und r Maße für die aktuellen Populationen von Beute- und Raubtieren sind, dann beschreiben die Differentialgleichungen

$$\begin{aligned} \dot{r} &= (-1 + \alpha b)r \\ \dot{b} &= (1 - \beta r)b \end{aligned} \quad (4.3.1)$$

ein einfaches Räuber-Beute-Modell. Der Punkt kennzeichnet die Ableitungen nach einer dimensionslosen Zeit τ , $\dot{r} = dr/d\tau$ und $\dot{b} = db/d\tau$. Die Parameter α und β beschreiben die Interaktion.

Die Differentialgleichungen sind aufgrund der Terme $\alpha r b$ und $\beta b r$ nichtlinear und können analytisch nicht gelöst werden. Die im File **rb_h.m** zusammengefassten MATLAB-Befehle erzeugen mit dem Funktions-File **rb_f.m** den in Abbildung 4-4 dargestellten Zeitverlauf.

Die Parameter α und β werden im Hauptprogramm als globale Variable definiert und mit Werten belegt.

Mit der MATLAB-Anweisung **global alpha beta** stehen die Variablen dann auch im Funktions-M-File **rb_f.m** zur Verfügung.

```
% Raeuber-Beute-Modell
clear all, close all
%
% Daten
global alpha beta
alpha = 0.02 ; beta = 0.03 ;
%
% Anfangsbedingungen
r0 = 100; b0=15;
%r0 = 1/beta; b0=1/alpha;
x0 = [ r0; b0 ] ;
%
% Integration
[t,xout] = ode23('rb_f',[0,25],x0) ;
%
% graphische Ausgabe
plot(t,xout(:,1),'-r',t,xout(:,2),'linewidth',2)
grid on
legend('r','b')
```

```

function rb = rb_f(t,x)
% Räuber-Beute-Modell
%
% Daten ueber globale Variable
global alpha beta
%
% umspeichern
r=x(1);
b=x(2);
%
% Dynamik
rp = ( -1 + alpha*b ) * r ;
bp = ( 1 - beta*r ) * b ;
%
rb = [ rp; bp ];

```

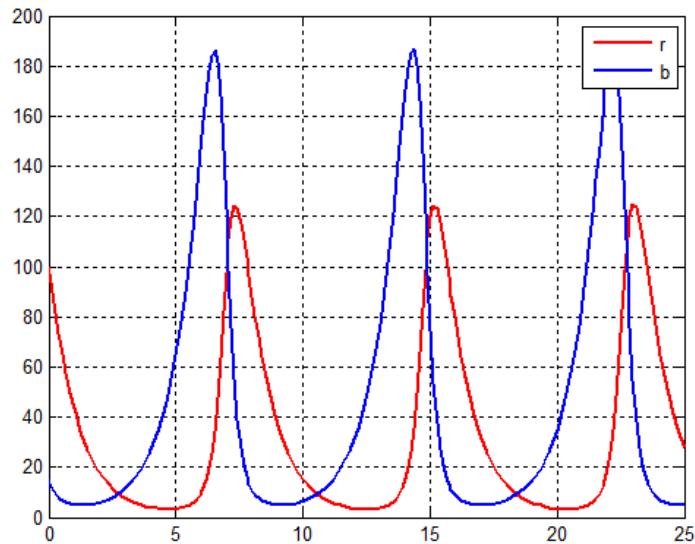


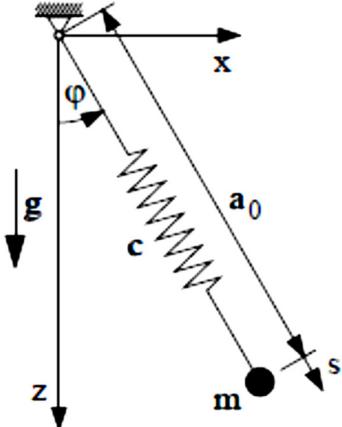
Abbildung 4-5: Räuber-Beute Modell

4.3.2 Federpendel

4.3.2.1 Bewegungsgleichungen

Die Bewegungen einer federnd aufgehängte Punktmasse m können durch die Federlängung s und den Ausschlagwinkel φ beschrieben werden, siehe dazu Abbildung 4-5. Die Bewegungsgleichungen für das Federpendel lauten dann

$$\begin{aligned} m(\ddot{\varphi}(s + a_0) + 2\dot{\varphi}\dot{s}) &= -m g \sin \varphi \\ m(\ddot{s} - \dot{\varphi}^2(s + a_0)) &= -cs + m g \cos \varphi \end{aligned} \quad (4.3.2)$$



wobei

a_0 die ungespannte Länge der Feder,

c die Federkonstante und

g die Erdbeschleunigung bezeichnen.

Abbildung 4-6: Federpendel

Für eine numerische Lösung mit den MATLAB-Solvern müssen die Differentialgleichungen 2. Ordnung in ein System von Differentialgleichungen 1. Ordnung umgewandelt werden.

4.3.2.2 Übergang auf ein System 1. Ordnung

Im ersten Schritt müssen die Differentialgleichungen (4.2.2) nach den 2. Ableitungen $\ddot{\varphi}$ und \ddot{s} aufgelöst werden. Man erhält

$$\begin{aligned} \ddot{\varphi} &= -(2\dot{\varphi}\dot{s} + g \sin \varphi)/(s + a_0) \\ \ddot{s} &= \dot{\varphi}^2(s + a_0) - \frac{c}{m}s + g \cos \varphi \end{aligned} \quad (4.3.3)$$

Mit den Substitutionen $\dot{\varphi} = \omega$ und $\dot{s} = v$ ergibt dies ein System von vier Differentialgleichungen 1. Ordnung

$$\begin{bmatrix} \dot{\varphi} \\ \dot{s} \\ \dot{\omega} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} \omega \\ v \\ -(2\omega v + g \sin \varphi)/(s + a_0) \\ \omega^2(s + a_0) - \frac{c}{m}s + g \cos \varphi \end{bmatrix} \quad (4.3.4)$$

das sehr einfach in eine MATLAB-Funktion umgesetzt werden kann

```

function fp_f = fp_f(t,x)
% Feder-Pendel
%
% daten
global m a_0 c g
%
% umspeichern
phi=x(1);
s =x(2);
om =x(3);
v =x(4);
%
% dgl-sys
phip = om ;
sp = v ;
omp = -(2*om*v+g*sin(phi))/(s+a_0) ;
vp = om^2*(s+a_0)-c/m*s+g*cos(phi) ] ;

fp_f = [phip; sp; omp; vp];

```

Die Daten m , a_0 , c und g werden dabei über globale Variable aus dem Hauptprogramm übernommen.

```

% Feder-Pendel Hauptprogramm
clear all
close all
% daten
global m a_0 c g
m = 5; % Masse [kg]
a_0 = 1; % ungespannte Federlaenge [m]
c = 150; % Federsteifigkeit [N/m]
g = 9.81; % Erdbeschleunigung [m/s^2]

% Anfangsbedingungen
phi_0 = 1*pi/180 ; om_0=0; s_0 = 0; v_0 =0;
x0 = [ phi_0 ; s_0 ; om_0 ; v_0 ] ;

% Integration
[t,xout] = ode45('fp_f',[0,35], x0) ; % 5.2.2.3 alternativer Aufruf 'fpa_f'

% Postprocessing
phi = xout(:,1);
s = xout(:,2);
axes('position',[0.05,0.55,0.4,0.35])
plot(t,s,'LineWidth',1)
grid on
title('s=s(t) [m]', 'FontSize',16)
axes('position',[0.05,0.05,0.4,0.35])
plot(t,phi*180/pi,'LineWidth',1)
grid on
title('\phi=\phi(t) [Grad]', 'FontSize',16)
axes('position',[0.525,0.300,0.45,0.45])
xp = (a_0+s).*sin(phi);
zp = (a_0+s).*cos(phi);
plot( xp, a_0-zp )
axis('equal')
title('Bahnkurve', 'FontSize',16)
text(pi/180,0.02,['\leftarrow start @ \phi_0=' ,num2str(phi_0*180/pi), '°'],...
'HorizontalAlignment','left')

```

Neben den Zeitschrieben $s = s(t)$ und $\varphi = \varphi(t)$ wurde dabei auch die Bahnkurve grafisch dargestellt, Abbildung 4-6.

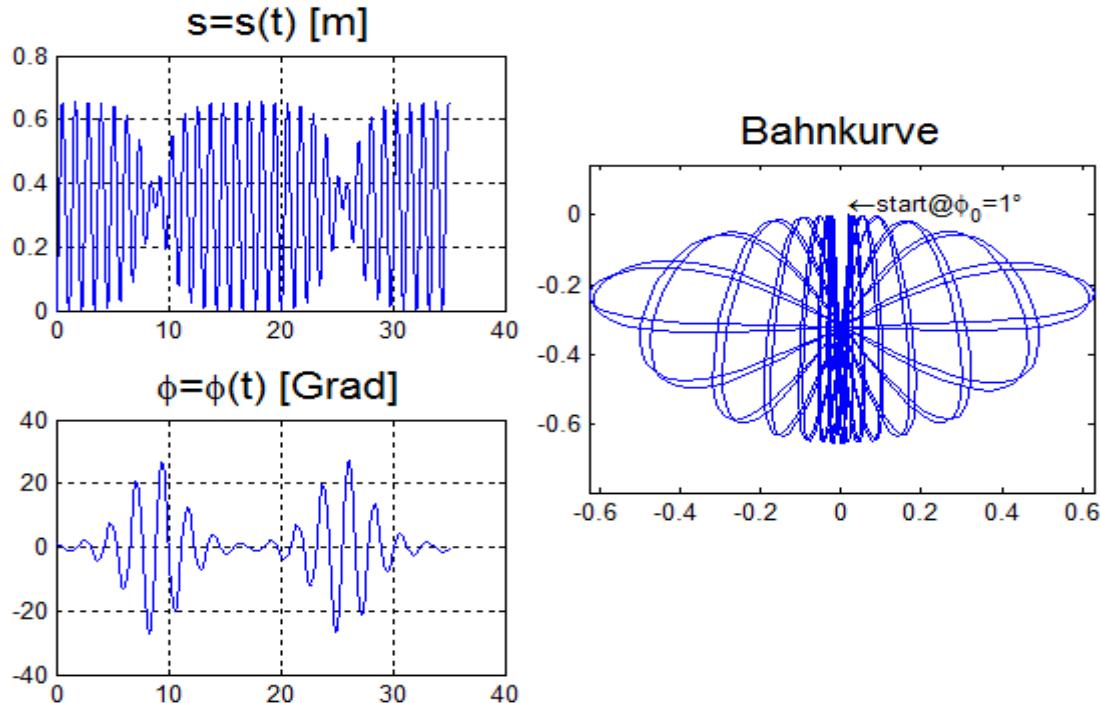


Abbildung 4-7: Zeitschriebe und Bahnkurve für das Federpendel

4.3.2.3 Allgemeine Form

Differentialgleichungen 2. Ordnung aus der Mechanik können allgemein in einer Matrizendifferentialgleichung der Form

$$M(y) \cdot \ddot{y} = q(t, y, \dot{y}) \quad (4.3.5)$$

angeschrieben werden, wobei die verallgemeinerten Lagegrößen im $n \times 1$ Vektor y angeordnet wurden. Die $n \times n$ Matrix M wird dann als Massenmatrix bezeichnet und die verallgemeinerten Kreisel- und Zentrifugalkräfte sowie die eingeprägten Kräfte werden im $n \times 1$ Vektor q zusammengefasst.

Mit der trivialen Gleichung $\dot{y} = \dot{y}$ kann (4.2.5) wieder als Differentialgleichungssystem 1. Ordnung angeschrieben werden

$$\begin{bmatrix} \dot{y} \\ \ddot{y} \\ \dot{x} \end{bmatrix} = \underbrace{\begin{bmatrix} \dot{y} \\ M(y)^{-1} q(t, y, \dot{y}) \\ f(t, x) \end{bmatrix}}_{\dot{x}}. \quad (4.3.6)$$

Die entsprechende MATLAB-Funktion `fpa_f.m` für das Federpendel lautet dann

```
function fpa_f = fpa_f (t,x)
% Feder-Pendel (allgemeine Loesung)
%
% Daten
global m a_0 c g
%
% umspeichern
```

```

phi  = x(1);
s    = x(2);
phip = x(3);
sp   = x(4);
%
% Massenmatrix
M = [ m*(s+a_0) 0 ; ...
0 m ];
%
% Rechte Seite
q = [ -(2*m*phip*sp+m*g*sin(phi)) ; ...
m*phip^2*(s+a_0)-c*s+m*g*cos(phi) ];
%
% Beschleunigungen
ypp = M\q ;
phipp = ypp(1);
spp = ypp(2);
%
% dgl-sys
fpa_f = [ phip ; ...
sp ; ...
phipp ; ...
spp ] ;

```

Mit folgender Vorgehensweise können dann beliebig große und komplizierte Systeme verarbeitet werden:

DGLs in folgende Form bringen:

$$\begin{aligned}
y_{1pp} * a_{1,DGL1} + y_{2pp} * a_{2,DGL1} + \dots + y_{npp} * a_{n,DGL1} &= \text{RechteSeite_1} \\
y_{1pp} * a_{1,DGL2} + y_{2pp} * a_{2,DGL2} + \dots + y_{npp} * a_{n,DGL2} &= \text{RechteSeite_2} \\
\dots \\
y_{1pp} * a_{1,DGLn} + y_{2pp} * a_{2,DGLn} + \dots + y_{npp} * a_{n,DGLn} &= \text{RechteSeite_n}
\end{aligned} \tag{4.3.7}$$

Massenmatrix:

$$M = \begin{bmatrix} a_{1,DGL1} & a_{2,DGL1} & \dots & a_{n,DGL1} \\ a_{1,DGL2} & a_{2,DGL2} & \dots & a_{n,DGL2} \\ \dots & \dots & & \dots \\ a_{1,DGLn} & a_{2,DGLn} & \dots & a_{n,DGLn} \end{bmatrix} \tag{4.3.8}$$

Lagegrößen:

$$q = \begin{bmatrix} \text{RechteSeite_1} \\ \text{RechteSeite_2} \\ \dots \\ \text{RechteSeite_n} \end{bmatrix} \tag{4.3.9}$$

$$ypp = M \setminus q ; \tag{4.3.10}$$

4.3.3 Druckregelung

Bei Diesel Common-Rail-Systemen kommt der Druckregelung hohe Bedeutung zu. Als stark vereinfachtes Modell wird ein Behälter mit einem Ventil (für alle Einspritzventile) und einer Pumpe betrachtet. Die Regelung ist ein PI-Regler.

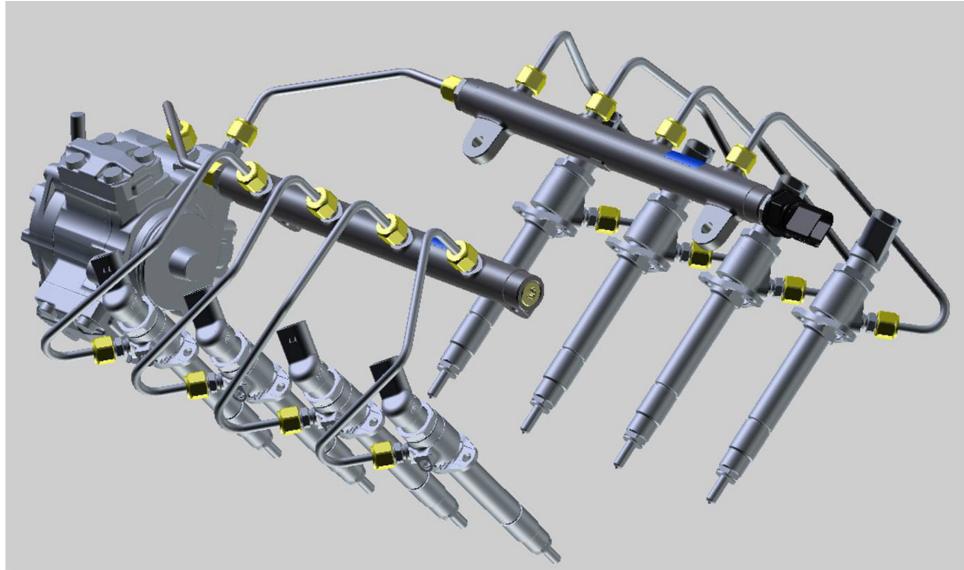


Abbildung 4-8: Diesel Common-Rail System

Abbildung 4-7 zeigt ein Modell eines Achtzylinder-Einspritzsystems mit einer Pumpe, zwei Rails und acht Injektoren.

In Abbildung 4-8 ist ein entsprechendes HSSIM Simulationsmodell dargestellt. Es besteht aus acht Injektoren, einer Pumpe, zwei Rails und entsprechenden Leitungsmo- dellen. Die Druckregelung ist ein PI-Regler, der den Drucksollwert mit dem Raildruck vergleicht und ausregelt.

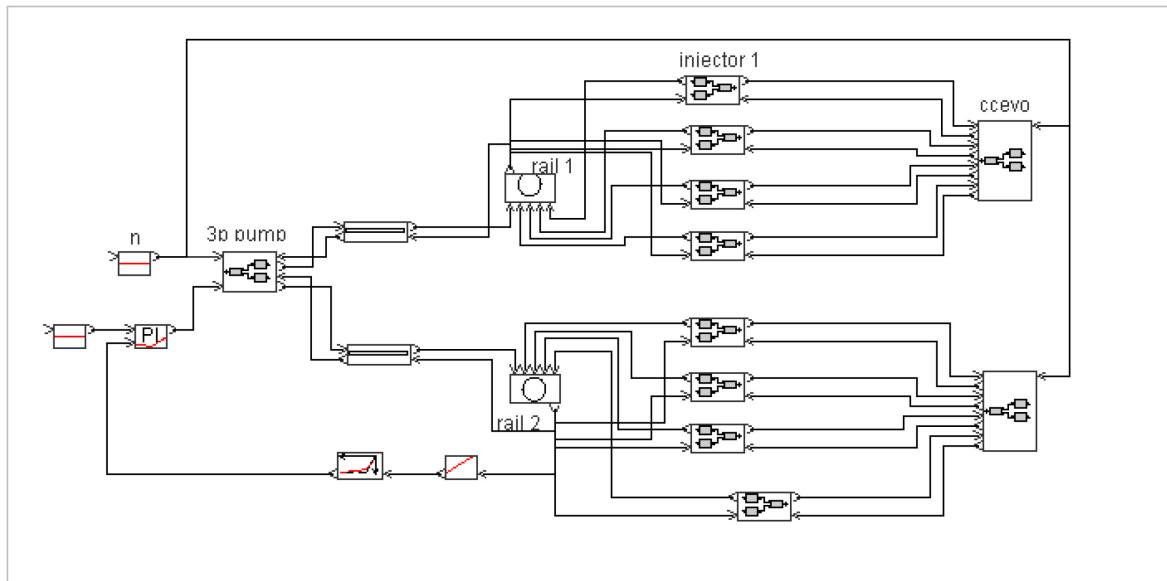


Abbildung 4-9: HSSIM Modell eines Einspritzsystems

Das HSSIM-Modell hat insgesamt ca. 500 Freiheitsgrade, d. h. es wird durch 500 Differentialgleichungen beschrieben.

Zum Verständnis der Druckregelung wird hier anstelle des komplizierten HSSIM-Modells ein stark vereinfachtes Modell untersucht. Abbildung 4-9 zeigt das Ersatzmodell. Es besteht aus einem Behälter, einfachen Blenden als Einspritzung und einer Pumpe, die als Verzögerungsglied erster Ordnung abgebildet ist.

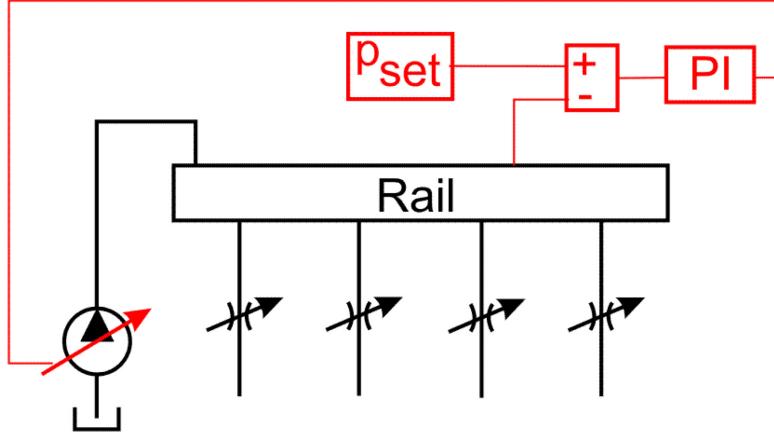


Abbildung 4-10: Vereinfachtes Behältermodell

Für den Raildruck gilt die Gleichung

$$\dot{p} = \frac{E}{V} (Q_p - Q_v) \quad (4.3.11)$$

mit für alle Einspritzventile zusammen dem Pumpenvolumenstrom Q_p und dem Ventilvolumenstrom Q_v .

Das Pumpenmodell als Verzögerungsglied erster Ordnung (Zeitkonstante T) lautet:

$$\dot{Q}_p = \frac{Q_{pi} - Q_p}{T} \quad (4.3.12)$$

Als letzte Gleichung wird der PI-Regler benötigt. Das verwendete Regelgesetz mit der Regelabweichung $e = p_{set} - p$

$$y = K_p (e + K_I \int e dt) \quad (4.3.13)$$

kann durch Differentiation in eine Differentialgleichung für die Regelabweichung umgewandelt werden:

$$\dot{E} = e \quad (4.3.14)$$

Der Ventilvolumenstrom wird nach der Bernoulli-Gleichung berechnet:

$$Q_v = A_v \sqrt{\frac{2p}{\rho}} \quad (4.3.15)$$

und der Pumpenvolumenstrom Q_{pI} wird proportional zum Reglerausgang y angenommen. Nachdem die Pumpe nur positive Volumenströme liefern kann, werden negative Werte abgeschnitten.

Die Matlab-Funktion `crdp_f.m` für die drei Differentialgleichungen lautet:

```

function xp=crdp_f(t,x)
global KP KI AV ptab T E V

% zeitabhängiger Parameter (Druck-Setpoint):
p0=interp1(ptab(:,1),ptab(:,2),t);% erzeugt Zwischenstuetzstellen

% umspeichern:
p = x(1); e = x(2); QP= x(3);

% Druckabweichung vom Setpoint:
dp=(p0-p*1e-6);

% Volumenströme:
QPS=KP/100*(dp + KI*e)*2/60000; % Pumpenvolumenstrom
QPS=max(QPS,0); % nur positive Werte für Pumpenvolumenstrom
QV=AV*sqrt(2*p/850 ); % Ventilvolumenstrom

% Differentialgleichungen:
edot=dp; % Fehlergradient
pdot = E/V*(QP-QV); % Druckgradient
Qdot = (QPS-QP)/T; % Volumenstromgradient

xp=[pdot ; edot; Qdot]; % Werte übergeben

```

Die Reglerparameter sowie die effektive Ventilfläche, die Zeitkonstante der Pumpe und eine Tabelle für den zeitlichen Verlauf des Drucksollwerts sind als globale Parameter definiert. Das Hauptskript lautet:

```

clear all; close all
global KP KI AV ptab T E V
% indizierter Drucksprung (Set-Point)
ptab=[0      100
      1      100
      1.01   200
      2      200
      2.01   100
      3      100];
% Parameter
T = 0.03; % Zeitkonstante Pumpe
KP = 5; % Proportionalanteil vom PI-regler
KI = 10; % Integralanteil vom PI-Regler
AV = pi/4*0.0002^2; % Ventilfläche
E=10000e5; % E-Modul
V=50e-6; % Behältervolumen 50cm^3
%
% Funktionsaufruf Klopfenstein-Shampine-Solver
% normierte Zeit 0...3; Anfangswerte: [pdot, edot, Qdot]
[t,x] = ode15s('crdp_f',[0 3],[1000e5 0 0]);
%
% Ausgabe
plot(t,x(:,1)*1e-5,'linewidth',2);
grid on;
xlabel('t (s)');
ylabel('p (bar)');

```

Abbildung 4-10 zeigt den simulierten Druckverlauf. Man erkennt starke Überschwingen durch den Integralteil des Reglers.

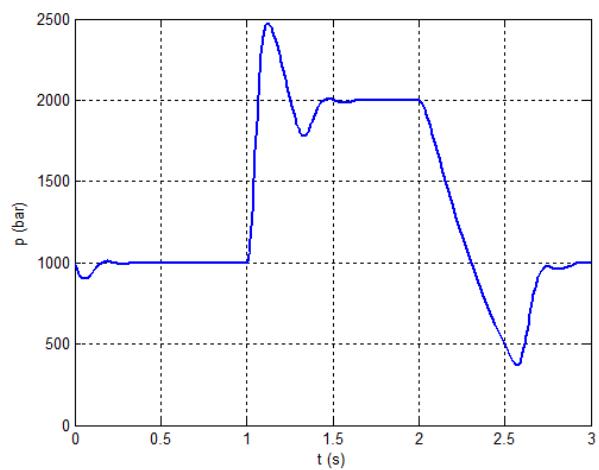


Abbildung 4-11: Simulierter Druckverlauf

5 Simulink

In Modellen lassen sich komplexe Strukturen sehr oft anschaulich darstellen. Für Kollegen sind – bei entsprechendem Modell-Design – die Lösungsstrukturen sowie deren Signalfluss leicht nachvollziehbar. Zudem ist, beim Verwenden mächtiger Blöcke, die Nutzung von Modellen sehr kompakt. Beispielsweise beschränkt sich ein von außen initialisierbarer Integrator-Block auf nur zwei Eingänge: den zur Initialisierung und den zu integrierenden, sowie einen Ausgang. Der Integrationsvorgang, der zudem mit verschiedenen *Solvern* erfolgen kann, wird aus Übersichtlichkeitsgründen nicht abgebildet.

5.1 Simulink-Bibliothek

Die Simulink Bibliothek bietet eine immense Auswahl von Funktionsblöcken, die sich per Drag-and-Drop in die Modelle einbinden lassen. Die mit am häufigsten verwendeten Blöcke sind in der Simulink-Bibliothek "Commonly Used Blocks" zusammengefasst.

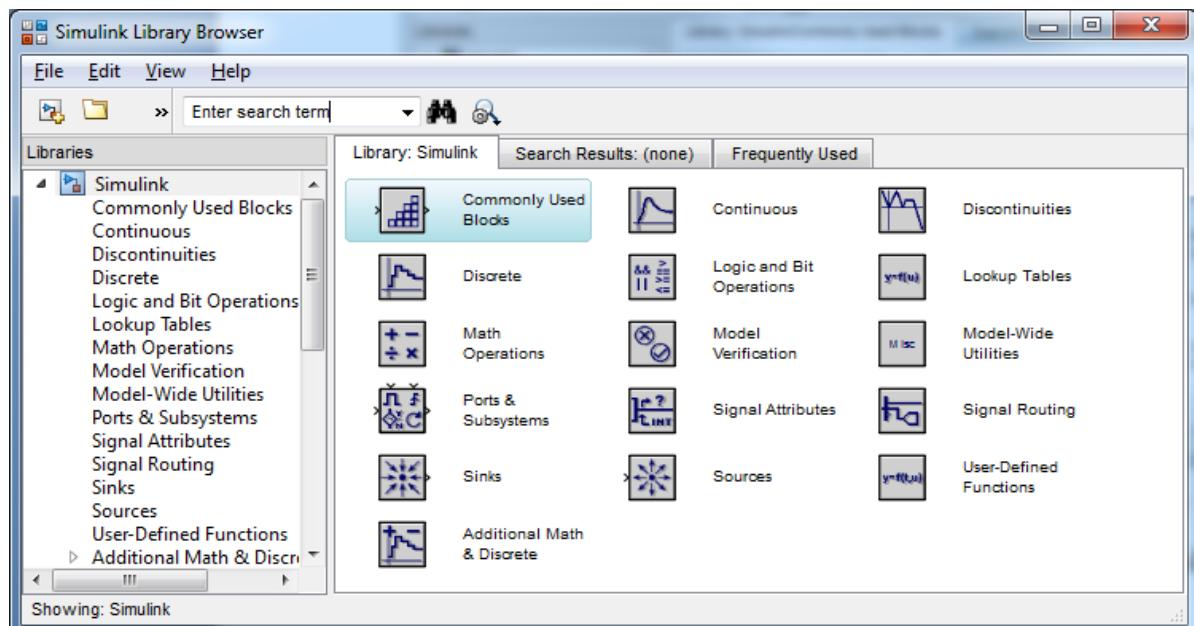
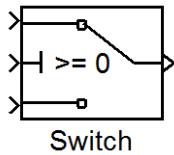


Abbildung 5-1: Simulink "Commonly Used Blocks" Library

Kennt man den Blocknamen (oder antizipiert diesen halbwegs korrekt) kann man den Block mit der Suchfunktion schnell finden.

Ein Doppelklick auf einen Funktionsblock liefert nähere Informationen zu diesem. Ist der Block in einem Modell platziert, so lassen sich seine spezifischen Eigenschaften anpassen. Für den Switch beispielsweise:

Beispiel: Ein Switch schaltet abhängig vom Steuereingang (2) einen der beiden Signaleingänge (1) oder (3) zum Ausgang.



Als Kriterium für das Durchschalten vom ersten Eingang muss im Beispiel der Wert am zweiten Eingang größer oder gleich null sein. Ist das Kriterium nicht erfüllt, wird der dritte Eingang zum Ausgang durchgeschaltet.

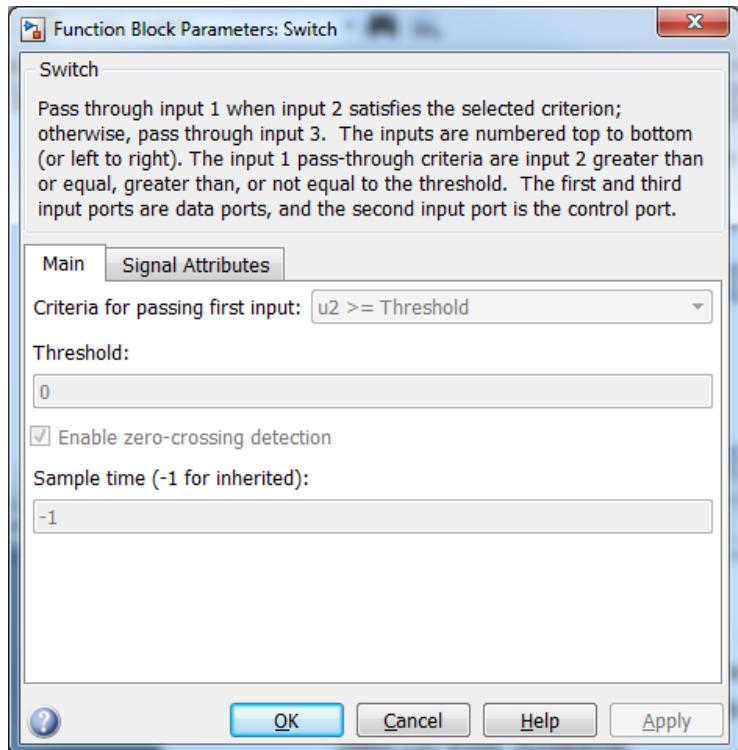


Abbildung 5-2: Switch-Properties

In vielen Blöcken findet man den Parameter *Sample Time*, womit sich die zeitliche Schrittweite, in welcher der Block gerechnet werden soll, spezifizieren lässt.

Als erläuterndes Beispiel soll folgendes Modell dienen. Im Source-Block *Sine Wave* ist ein mit 100 ms zeitlich diskretisiertes Sinussignal gegeben. Dieses dient zwei Vergleichsblöcken als Eingangssignal. Einer der Blöcke wird dann gerechnet, wenn sich zuvor sein Eingangssignal geändert hat, der andere alle 500 ms. Im *Scope* lässt sich die Wirkungsweise erkennen. Das Signal **sampetime500ms** erfährt ein Delay gegenüber dem Zeitpunkt, zu dem die Schwelle durchlaufen wurde, je nachdem wann der aktuelle 500-ms-Zeitschritt vollendet ist. Das Signal **inherited** hingegen, ändert sich zusammen mit dem Sinussignal **sine_wave** im selben 100-ms-Schritt, sobald die Schwelle durchlaufen wurde.

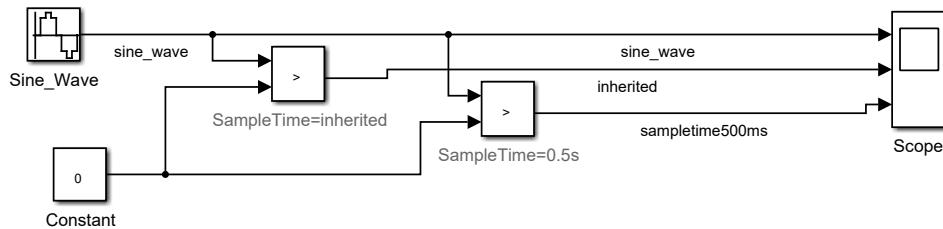


Abbildung 5-3: Modellbeispiel *Sample_Time*

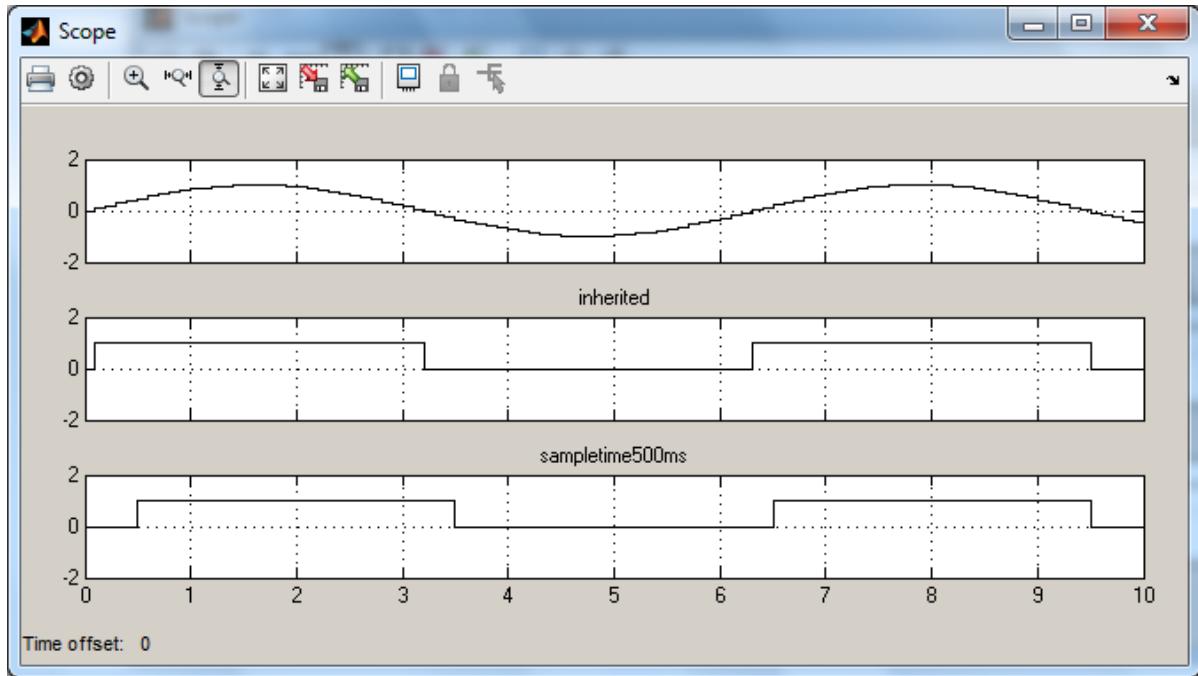


Abbildung 5-4: *Scope* vom Modell Abbildung 5-3

5.2 Simulink-Modelle via Script-Steuerung

5.2.1 Gleichstrommotor in Simulink

Vorab soll die Differentialgleichung eines Gleichstrommotors mit Permanenterregung anhand eines Ersatzschaltbildes hergeleitet werden, welche anschließend in Simulink modelliert und simuliert wird.

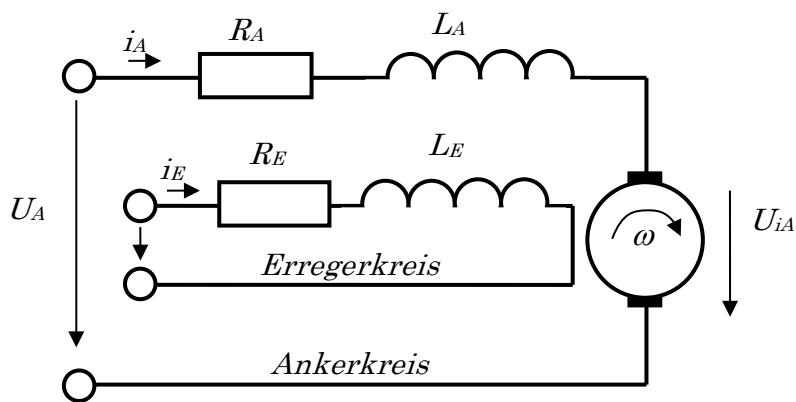


Abbildung 5-5: Ersatzschaltbild eines Gleichstrommotors

In Abbildung 5-5 ist der Erregerkreis elektrisch betrieben. Wird dieser durch einen Permanentmagneten ersetzt, so vereinfacht sich die Anordnung zu

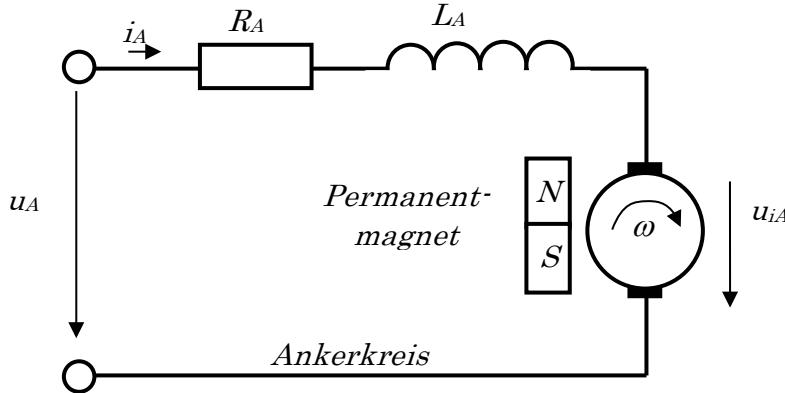


Abbildung 5-6: Ersatzschaltbild eines Gleichstrommotors

An den Klemmen des Ankerkreises liegt die Ankerspannung u_A (= Klemmenspannung). R_A bezeichnet den Ankerwiderstand und L_A die Ankerinduktivität. Die innere Gegen Spannung u_{iA} (auch Elektro-Motorische-Kraft, EMK genannt) wird durch Induktion (Drehzahl) erzeugt. Sie bestimmt – zusammen mit u_A – den fließenden Ankerstrom i_A . Würde sich der Anker reibungsfrei und ohne Last drehen können, wären beide Spannungen gleich groß und der Strom wäre null. Für die Maschengleichung des Ankerkreises gilt folglich

$$u_A = R_A \cdot i_A + L_A \cdot \frac{di_A}{dt} + u_{iA} \quad (5.2.1)$$

Mit einem vom Permanentmagneten konstant erzeugten magnetischen Fluss Φ gilt für die induzierte Gegen Spannung

$$u_{iA} = c_M \cdot \Phi \cdot \omega \quad (5.2.2)$$

und für das Drehmoment

$$M_M = c_M \cdot \Phi \cdot i_A \quad (5.2.3)$$

Die Maschinenkonstante c_M ist dabei ein motorspezifischer Faktor und ω die Winkelgeschwindigkeit in rad/s.

Für das beschleunigende Moment gilt

$$J_A \frac{d\omega}{dt} = M_M - M_L \quad (5.2.4)$$

wobei J_A das Massenträgheitsmoment der Welle und der betriebenen Last bezeichnet, M_M das vom Motor aufgebrachte Moment und M_L das Lastmoment inklusive eventueller Reibungsmomente.

Mit den Gleichungen 5.2.1 bis 5.2.4. lässt sich nun ein Modell erstellen, in dem – abhängig von der Ankerspannung u_A und dem Lastmoment M_L – der Drehzahlverlauf als Ausgangsgröße berechnet wird.

Nach Einsetzen der Gleichungen 5.2.2 in 5.2.1 sowie 5.2.3 in 5.2.4 und Umstellen nach der jeweils höchsten Ableitung folgt

$$\frac{di_A}{dt} = -i_A \cdot \frac{R_A}{L_A} + \frac{u_A}{L_A} - \omega \cdot \frac{c_M \cdot \Phi}{L_A} \quad (5.2.5)$$

und

$$\frac{d\omega}{dt} = i_A \cdot \frac{c_M \cdot \Phi}{J_A} - \frac{M_L}{J_A}. \quad (5.2.6)$$

Angemerkt sei, dass das Umstellen nach der höchsten Ableitung die Modellbildung erheblich vereinfacht. Zudem sollten gemeinsame Faktoren ausgeklammert werden.

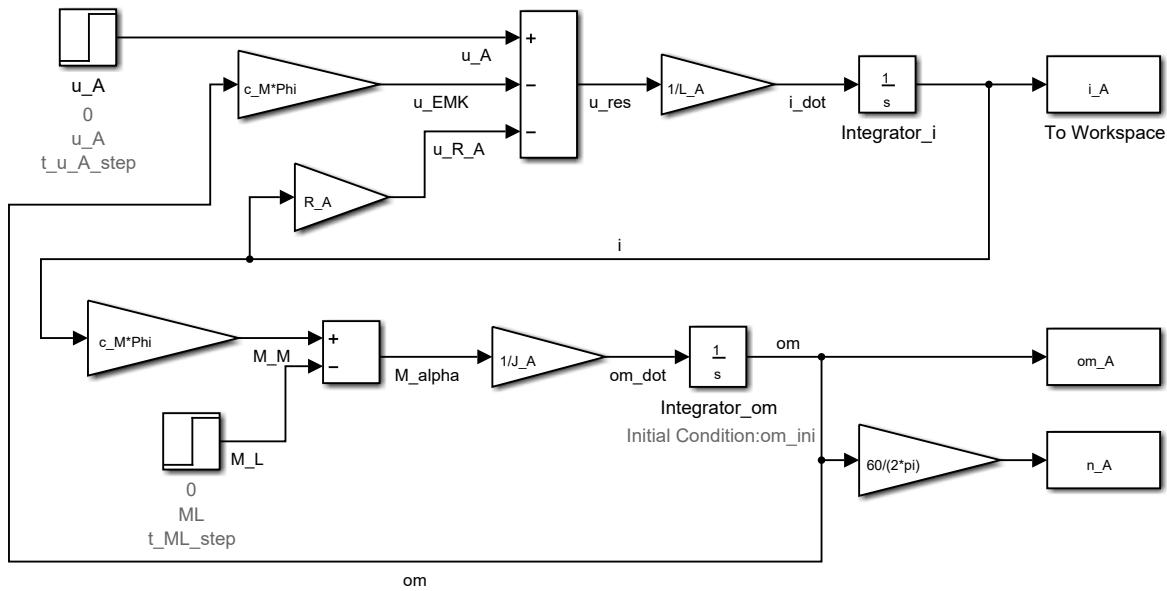


Abbildung 5-7: Modell für permanenterregten Gleichstrommotor

Der "Source" Block "Step" stellt eine Sprungfunktion, hier einen Spannungssprung zum Zeitpunkt $t_{uA,Step}$ vom Initialwert null zum Endwert u_A zur Verfügung.

Die "Sample time" bestimmt die Schrittweite, die der Solver benutzen soll. Somit wird sichergestellt, dass Transienten hinreichend genau simuliert werden. Wird kein oder auch ein zu großer Wert vorgegeben, so wird sie von Matlab automatisch limitiert und eine Warnung ausgegeben.

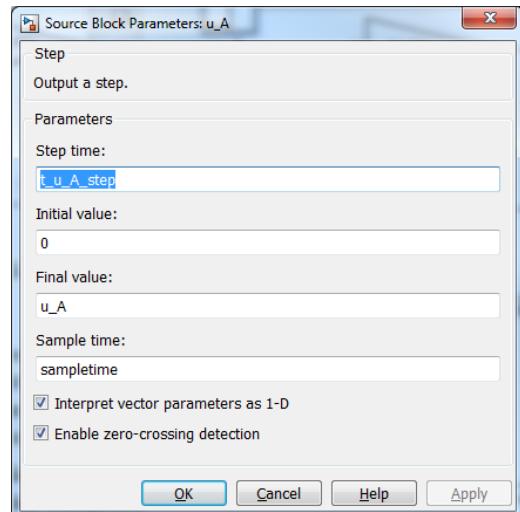


Abbildung 5-8: Parameter für Spannungssprung

Mit den Blöcken "To Workspace" werden die während der Simulation errechneten Werte im Workspace gespeichert und stehen somit beispielsweise zum Post-Processing zur Verfügung.

Mit folgendem Skript lässt sich das Modell initialisieren und starten. Die Simulationsergebnisse werden anschließend mit dem plot-Befehl dargestellt.

```

% DC_Motor_INI
clear all, close all

c_M = 1.0; % Maschinenkonstante
Phi = 1.5; % Vs Magnetischer Fluss vom Dauermagneten
u_A = 20; % V Klemmenspannung
ML = 5; % Nm Lastmoment
L_A = 0.15; % H Ankerinduktivität
R_A = 2.0; % Ohm Ankerwiderstand
J_A = 2.0; % kgm^2 Trägheitsmoment von Anker und Last

om_ini = 0; %13; % Anfangs-Kreisfrequenz
t_u_A_step = 2; % 15; % s
t_ML_step = 12; % 15; % s

sampletime = 0.05;
tend = 30; % 30; % zu simulierende Zeit SimulationStopTime (Im Menü muss
            % dazu der Zahlenwert durch tend ersetzt werden.
sim ('DC_Motor')

subplot(3,1,1), plot(i_A.time,i_A.signals.values)
ylabel('i\Anker in A')
subplot(3,1,2), plot(om_A.time,om_A.signals.values)
ylabel('Omega in rad/s')
subplot(3,1,3), plot(n_A.time,n_A.signals.values)
ylabel('Drehzahl'), xlabel('t in s')

```

Die Dimensionierung des Motors ist im Skript gegeben, zwei Sekunden nach Simulationsbeginn erfolgt ein Spannungssprung von null auf 20 V.

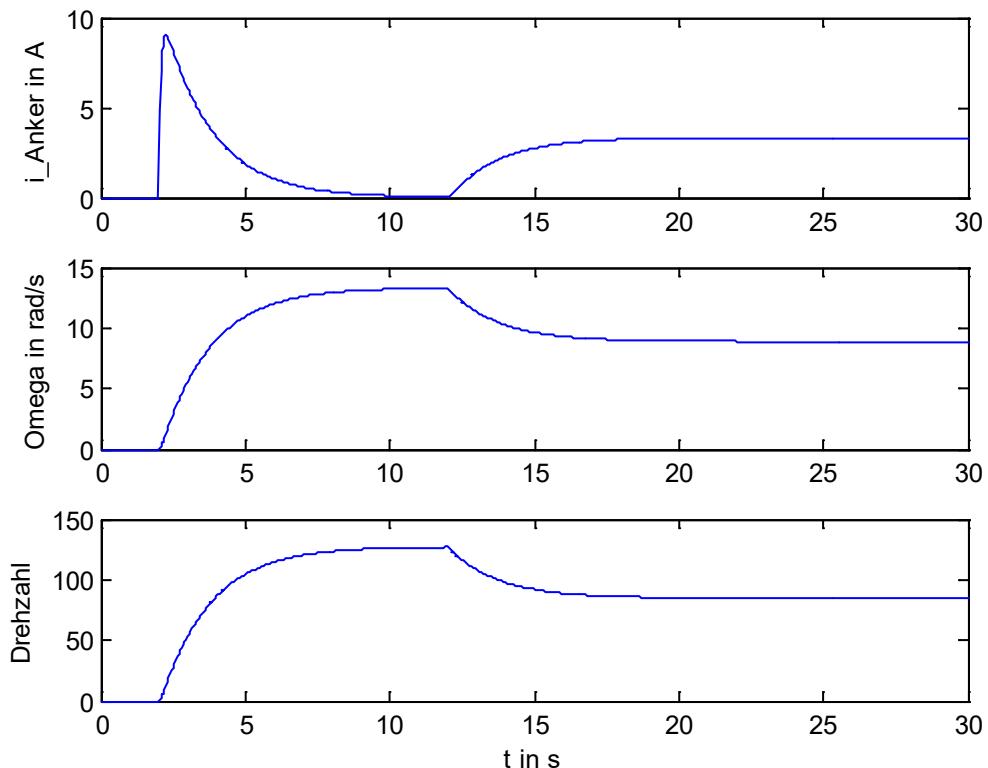


Abbildung 5-9: Modellergebnisse für permanenterregten Gleichstrommotor

Soll der Gleichstrommotor zu Anfang der Simulation bereits drehen, so muss $\omega = \omega_0$ initialisiert werden.

Dies geschieht, indem der Ausgang vom entsprechenden Integrator mit dem gewünschten Initialwert – hier `om_ini` – belegt wird.

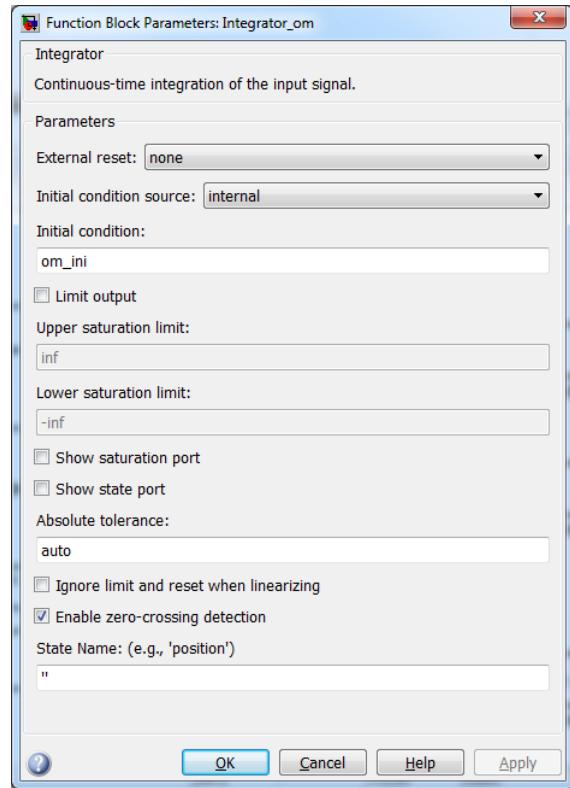


Abbildung 5-10: Parameter für Integratorblock ω

Mit der entsprechenden Skriptänderung

```
om_ini = 13; % 0; % s^-1 Anfangs-Kreisfrequenz
t_u_A_step = 15; % 0; % s
ML = 0; %5; % Nm Lastmoment
```

lässt sich nun beispielsweise das Anhalten des Motors mit einer Ankerspannung $u_A = 0$ V (Anker kurzgeschlossen) und ein Wiederanlauf nach 15 s simulieren.

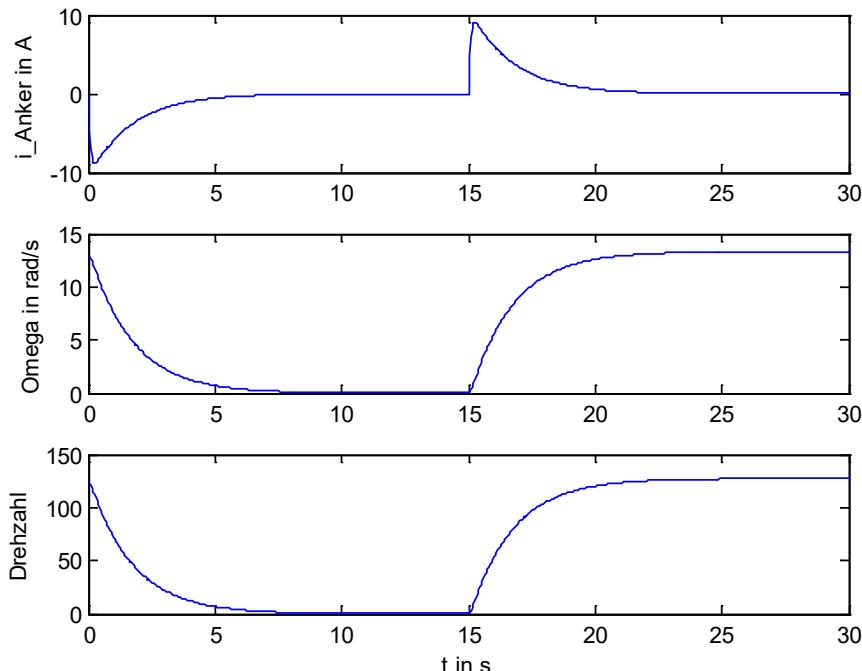


Abbildung 5-11: Modellergebnisse für permanenterregten Gleichstrommotor

Im State-Flow *Chart1* ist nebenstehende Funktion hinterlegt

Vom ersten Verzeigungspunkt beinhaltet die Transition 1 die Bedingung, ob das Signal1 größer null ist. Ist dies der Fall, so wird Signal1 am Ausgang1 zur Verfügung gestellt. Falls nicht, so greift die unbedingte Transition 2. Beim Durchlaufen dieser, wird der Ausgang1 mit null belegt.

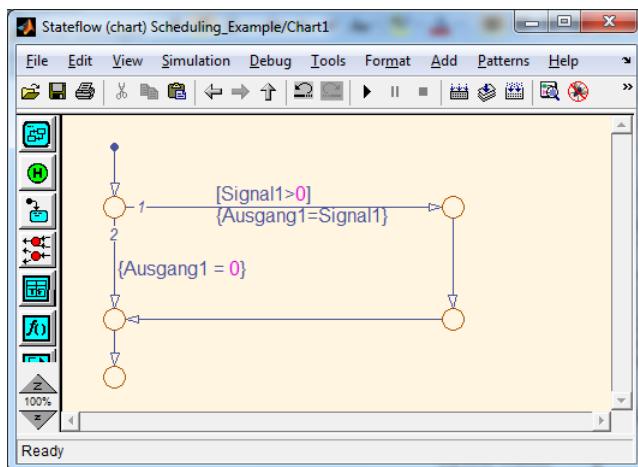


Abbildung 5-17: Inhalt von State-Flow *Chart1*

Das State-Flow *Chart2* ist identisch zu *Chart1*

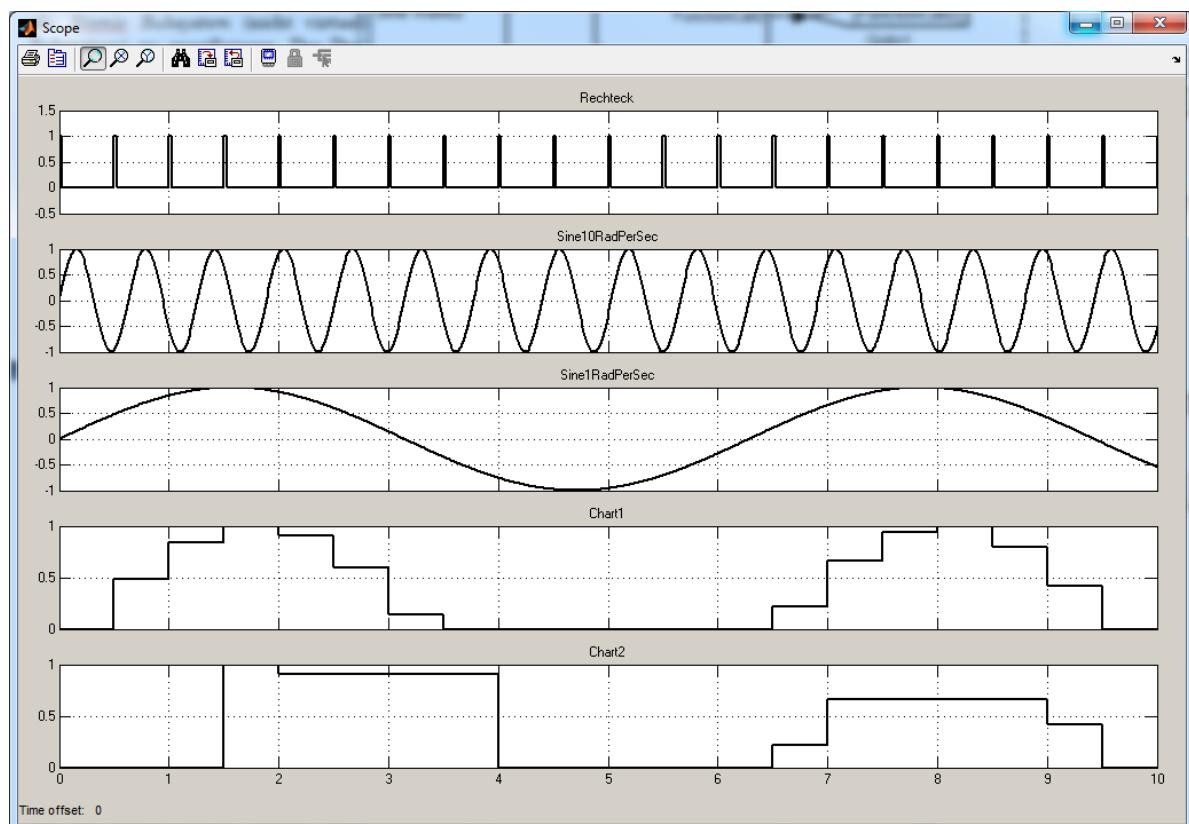


Abbildung 5-18: Simulationsergebnis vom Modell in Abbildung 5-15

Die Darstellung lässt nicht erkennen, ob *Chart1* oder *Chart2* zuerst berechnet wurde, was sich jedoch aus der Anordnung der Aufrufe in *Ladder* ergibt. Vielmehr lassen sich die Bedingungen nachvollziehen, unter denen welches Subsystem berechnet wurde. Es lässt sich erkennen, dass *Chart1* den Momentanwert an seinem Eingang Signal1 zum Zeitpunkt einer steigenden Flanke bewertet und diesen – wenn er denn größer null ist – an seinem Ausgang zur Verfügung stellt. Der Ausgang ändert sich dann erst wieder

zum nächsten positiven Flankenwechsel. Beim *Chart2* kommt eine zusätzliche Bedingung ins Spiel. Hier maskiert *SineWave2* zusätzlich den *FunktionCall2*. Somit müssen beide Sinusquellen positive Werte besitzen, damit der Ausgang von *Chart2* dessen Eingangswert *Signal1* annimmt.

6 Praxisanwendungen

6.1.1 Lesen von Textdateien

Eine häufige Problemstellung in der Ingenieurspraxis ist die Darstellung von Messdaten.

MATLAB unterstützt die Verarbeitung von Dateien ähnlich zur Programmiersprache C, die Anwendung ist allerdings deutlich einfacher. Anhand eines Beispiels sollen die notwendigen Schritte gezeigt werden. Sind die Daten in Form einer Matrix ohne Text in einer Datei gespeichert, so kann der Inhalt der Datei mit dem Befehl `load` direkt eingelesen werden. Beispiele hierzu finden sich im Kapitel zur Ausgleichsrechnung. Hier soll aber ein anspruchsvollerer Fall betrachtet werden:

Die Datei enthält in der ersten Zeile die Namen der Messsignale. Ab der zweiten Zeile sind die Zahlen gespeichert. Die Namen und die Zahlen seien jeweils durch ein Semikolon getrennt, wie im folgenden Beispiel gezeigt.

```
t(s); x(m); v(m/s)
0.000000;0.000000;1.000000
0.040080;0.040077;0.999799
0.080160;0.080139;0.999197
0.120240;0.120168;0.998193
0.160321;0.160149;0.996789
0.200401;0.200066;0.994984
0.240481;0.239902;0.992780
0.280561;0.279642;0.990177
0.320641;0.319269;0.987176
0.360721;0.358769;0.983779
```

Aufgrund der ersten Zeile kann diese Datei nicht mit `load` geladen werden. Hierfür ist eine eigene Funktion erforderlich.

Die Schritte hierfür sind:

- Lesen der Datei in ein byte-array
- Aufspalten der Daten in Zeilen. Zählen der Zeilen.
- Lesen der ersten Zeile, Aufteilen in ein Feld mit den Namen. Bestimmung der Anzahl Namen. Dies ist zugleich die Anzahl der Datenspalten.
- Lesen der Daten in eine Matrix.

6.1.1.1 Lesen der Datei

Das Lesen in ein byte-Array erfolgt mit der Funktion `fread`. Vorher muss die Datei geöffnet werden. Nach dem Lesen wird sie wieder geschlossen.

```
% oeffnen
fid = fopen(fname, 'r');
% bytes lesen
a = fread(fid);
fclose(fid);
```

`fid` ist ein sog. file-identifier. Hierüber erfolgt bei allen Dateioperationen der Zugriff auf die Datei. Es ist eine starke Ähnlichkeit zur Dateibearbeitung in C erkennbar.

6.1.1.2 Zählen der Zeilen

Eine Zeile ist durch das Byte 13 (CR) abgeschlossen. Die Anzahl der Zeilen ist damit die Anzahl der Vorkommnisse des Bytes 13.

Die `find`-Funktion kann verwendet werden, um alle vorkommenden Zeilenumbrüche (Byte 13) zu finden.

```
ind = find(a==13);
```

Die Länge des Indexfeldes `ind` ist somit die Anzahl der Zeilen.

Dez	Hex	Okt	ASCII												
0	0x00	0	NUL	32	0x20	40	SP	64	0x40	100	€	96	0x60	140	`
1	0x01	1	SOH	33	0x21	41	!	65	0x41	101	À	97	0x61	141	a
2	0x02	2	STX	34	0x22	42	"	66	0x42	102	฿	98	0x62	142	b
3	0x03	3	ETX	35	0x23	43	#	67	0x43	103	฿	99	0x63	143	c
4	0x04	4	EOT	36	0x24	44	\$	68	0x44	104	฿	100	0x64	144	d
5	0x05	5	ENQ	37	0x25	45	%	69	0x45	105	฿	101	0x65	145	e
6	0x06	6	ACK	38	0x26	46	&	70	0x46	106	฿	102	0x66	146	f
7	0x07	7	BEL	39	0x27	47	'	71	0x47	107	฿	103	0x67	147	g
8	0x08	10	BS	40	0x28	50	(72	0x48	110	฿	104	0x68	150	h
9	0x09	11	TAB	41	0x29	51)	73	0x49	111	฿	105	0x69	151	i
10	0x0A	12	LF	42	0x2A	52	*	74	0x4A	112	฿	106	0x6A	152	j
11	0x0B	13	VT	43	0x2B	53	+	75	0x4B	113	฿	107	0x6B	153	k
12	0x0C	14	FF	44	0x2C	54	,	76	0x4C	114	฿	108	0x6C	154	l
13	0x0D	15	CR	45	0x2D	55	-	77	0x4D	115	฿	109	0x6D	155	m
14	0x0E	16	SO	46	0x2E	56	.	78	0x4E	116	฿	110	0x6E	156	n
15	0x0F	17	SI	47	0x2F	57	/	79	0x4F	117	฿	111	0x6F	157	o
16	0x10	20	DLE	48	0x30	60	0	80	0x50	120	฿	112	0x70	160	p
17	0x11	21	DC1	49	0x31	61	1	81	0x51	121	฿	113	0x71	161	q
18	0x12	22	DC2	50	0x32	62	2	82	0x52	122	฿	114	0x72	162	r
19	0x13	23	DC3	51	0x33	63	3	83	0x53	123	฿	115	0x73	163	s
20	0x14	24	DC4	52	0x34	64	4	84	0x54	124	฿	116	0x74	164	t
21	0x15	25	NAK	53	0x35	65	5	85	0x55	125	฿	117	0x75	165	u
22	0x16	26	SYN	54	0x36	66	6	86	0x56	126	฿	118	0x76	166	v
23	0x17	27	ETB	55	0x37	67	7	87	0x57	127	฿	119	0x77	167	w
24	0x18	30	CAN	56	0x38	70	8	88	0x58	130	฿	120	0x78	170	x
25	0x19	31	EM	57	0x39	71	9	89	0x59	131	฿	121	0x79	171	y
26	0x1A	32	SUB	58	0x3A	72	:	90	0x5A	132	฿	122	0x7A	172	z
27	0x1B	33	ESC	59	0x3B	73	;	91	0x5B	133	[123	0x7B	173	{
28	0x1C	34	FS	60	0x3C	74	<	92	0x5C	134	\	124	0x7C	174	
29	0x1D	35	GS	61	0x3D	75	=	93	0x5D	135]	125	0x7D	175	}
30	0x1E	36	RS	62	0x3E	76	>	94	0x5E	136	^	126	0x7E	176	~
31	0x1F	37	US	63	0x3F	77	?	95	0x5F	137	_	127	0x7F	177	DEL

6.1.1.3 Lesen und Aufteilen der ersten Zeile

Der Bereich bis zum ersten Zeilenumbruch wird in einen String umgewandelt. Dieser wird mit der `strread`-Funktion verarbeitet und mit dem Trennzeichen aufgeteilt. Das Ergebnis in `h` ist ein `cell`-array. Die Anzahl der Datenspalten entspricht der Länge des `cell`-arrays.

```

l1 = char(a(1:ind(1)))'; % Lese erste Zeile
% Aufteilen in Datenlabels
h = strread(l1,'%s','delimiter',';');
disp([num2str(length(h)) ' Spalten ']);

```

6.1.1.4 Lesen der Daten

```

d = [];
% Einlesen der Daten
count = 1;
for i=2:length(ind)
line = char(a(ind(i-1)+2:ind(i)))'; % liest eine Zeile
d(count,:) = strread(line,'%f','delimiter',';');
count = count+1;
end

```

Die Daten werden zeilenweise aus dem byte-array a mit strread eingelesen. Die Position jeder Zeile ist durch die Positionen der Zeilenumbrüche im ind-Feld festgelegt. Die Zahlen werden in einer dynamisch wachsenden Matrix d gespeichert. Für große Dateien empfiehlt sich, die Datenmatrix vorher zu dimensionieren.

6.1.1.5 Gesamtfunktion

Die Gesamtfunktion `readsimple.m` ist hier.

```

function [d,h]=readsimple(fname)
% liest eine Textdatei mit Labels aus. Daten sind durch ';' getrennt

fid=fopen(fname,'r'); % oeffnen
a=fread(fid); % Bytes lesen

ind=find(a==13); % finden der Zeilenumbrueche
l1=char(a(1:ind(1)))'; % Lese erste Zeile

h=strread(l1,'%s','delimiter','>'); % Aufteilen in Datenlabels

% Einlesen der Daten
d=[];
count=1;

for i=2:length(ind)
line=char(a(ind(i-1)+2:ind(i)))'; % liest eine Zeile
d(count,:)=strread(line,'%f','delimiter',';');
count=count+1;
end

fclose(fid);

```

Ein Anwendungsbeispiel für eine Datei `data.txt` wäre

```
[d,h] = readsimple('data.txt');
```