

Praktikum MATLAB®/Simulink® II

Prof. Dr.-Ing. U. Konigorski
Skript



TECHNISCHE
UNIVERSITÄT
DARMSTADT

REGELUNGSTECHNIK **rtm**
UND MECHATRONIK

Praktikum MATLAB®/Simulink® II

Prof. Dr.-Ing. U. Konigorski

Skript



Technische Universität Darmstadt
Institut für Automatisierungstechnik und Mechatronik
Fachgebiet Regelungstechnik und Mechatronik
Prof. Dr.-Ing. U. Konigorski

Landgraf-Georg-Straße 4
64283 Darmstadt
Telefon 06151/16-25200
www.rtm.tu-darmstadt.de

Das Gesamtdokument ist unter CC BY-ND veröffentlicht:



<https://creativecommons.org/licenses/by-nd/4.0/>

Der Inhalt dieses Dokuments ausschließlich der Logos, des Layouts und der Schriftarten ist unter CC BY-SA veröffentlicht:



<https://creativecommons.org/licenses/by-sa/4.0/>



Inhalt des Praktikums und Voraussetzungen

Das Praktikum MATLAB/Simulink II wird an der Technischen Universität Darmstadt im Masterstudium angeboten und richtet sich im Wesentlichen an Studenten der Automatisierungstechnik und Mechatronik.

Wie auch schon im Praktikum MATLAB/Simulink I werden wieder zwei Ziele verfolgt: Zum einen die Anwendung regelungstechnischer Methoden, zum anderen die Vermittlung von Kenntnissen in MATLAB und Simulink. Dazu wird als Beispielsystem ein Schlitten-Pendel-System betrachtet (siehe Abbildung 4.1 auf Seite 26), für das im Rahmen dieses Praktikums eine Regelung um den unteren und oberen Arbeitspunkt sowie eine Steuerung des Aufschwungs entworfen werden wird.

Dieses Praktikum MATLAB/Simulink II baut auf dem Praktikum MATLAB/Simulink I auf und setzt entsprechende *Grundkenntnisse in MATLAB und Simulink* voraus. Darüber hinaus wird davon ausgegangen, dass die Teilnehmer schon eine Vorlesung gehört haben, in denen die *Grundlagen zum Reglerentwurf im Zustandsraum* vermittelt werden.

Die Unterlagen zu diesem Praktikum bestehen aus

- dem vorliegenden Skript,
- den Versuchsunterlagen sowie
- zu manchen Versuchen aus vorbereiteten MATLAB-Funktionen.

Vor jedem Versuch sollte das entsprechende Kapitel im Skript durchgearbeitet werden. In diesem werden zum einen die in dem jeweiligen Versuch verwendeten regelungstechnischen Methoden erläutert. Dabei sind diese Erläuterung bei Themen, die zu den Grundlagen der Regelung im Zustandsraum gehören, sehr knapp gehalten und sollen bewusst kein Skript zu diesem Thema ersetzen. Zum anderen werden die für den jeweiligen Versuch benötigten MATLAB-Funktionen und -Funktionalitäten erklärt.

Die Versuchsunterlagen enthalten die an den einzelnen Versuchsnachmittagen zu lösenden Aufgaben.



Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Modellierung und Simulation eines Schlitten-Pendel-Systems | 7 |
| 2 | Steuerbarkeit und Beobachtbarkeit | 31 |
| 3 | LQ-Regelung und Animation | 41 |
| 4 | Beobachterentwurf – Benutzeroberflächen | 53 |
| 5 | Aufschwingsteuerung für das Pendel | 75 |
| 6 | Trajektorienfolgeregelung | 99 |



Versuch 1

Modellierung und Simulation eines Schlitten-Pendel-Systems

In den Vorlesungen Systemdynamik und Regelungstechnik I und II (siehe [10] und [2]) werden theoretische Grundlagen in Bezug auf Modellbildung und Reglerentwurf vermittelt. Das Praktikum MATLAB/Simulink I vermittelt Grundlagen in MATLAB/Simulink, welche in diesem und den folgenden Versuchen ausgebaut werden. Eine gute Einführung in MATLAB/Simulink bietet [16].

| | | |
|----------|---|-----------|
| 1 | Das Schlitten-Pendel-System | 8 |
| 2 | Physikalische Modellbildung | 9 |
| 2.1 | Strukturbild | 9 |
| 2.2 | Nichtlineare Modellbildung | 10 |
| 2.3 | Lineare Modellbildung | 18 |
| 2.4 | Simulation | 21 |
| 3 | Vorbereitungsaufgaben | 25 |
| 4 | Anhang | 26 |
| 4.1 | Variable und Parameter des Schlitten-Pendel-Systems | 26 |
| 4.2 | S-Function-Template | 26 |

1 Das Schlitten-Pendel-System

An einem Schlitten-Pendel-System, das dem Institut für Automatisierungstechnik zur Verfügung steht, sollen weiterführende Konzepte der Regelungstechnik eingeführt werden. Die zur Modellierung notwendigen Größen sind in Abbildung 1.1 dargestellt. Dabei sind sämtliche Reibungskräfte zu vernachlässigen. Der Schlitten wird durch eine äußere Kraft $F(t)$ angeregt. Über diese Kraft soll im Laufe des Praktikums die Position des Schlittens $x_s(t)$ und der Winkel $\varphi(t)$ geregelt werden.

Sämtliche zur Modellbildung notwendigen Größen sind Tabelle 4.1 im Anhang aufgelistet.

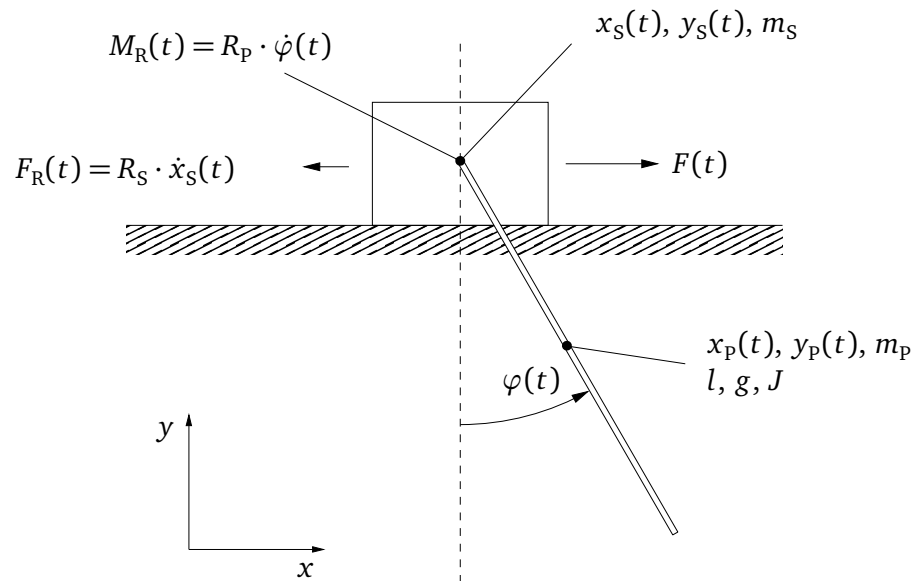


Abbildung 1.1: Darstellung des Schlitten-Pendel-Systems mit den zur Modellbildung zu verwendenden Größen.

2 Physikalische Modellbildung

Damit eine physikalische Modellbildung durchgeführt werden kann, muss das System durch Gleichungen beschrieben werden. Meistens handelt es sich dabei um gewöhnliche, gelegentlich auch um partielle Differentialgleichungen. Es existieren mehrere Verfahren zur Aufstellung der Bewegungsgleichungen, z. B. NEWTON-EULER-Methode oder die LAGRANGESchen Gleichungen.

In diesem Kapitel wird zunächst die lineare und nichtlineare Modellbildung anhand eines Heizlüfters vorgestellt.

2.1 Strukturbild

Die verschiedenen Modellierungsmöglichkeiten sollen anhand eines einfachen nichtlinearen Beispielsystems, welches hier ein Heizlüfter gemäß der Vorlesung SDRT I [10] darstellt, verdeutlicht werden. Als Eingangsgröße wird die Spannung $u(t)$ und als Regelgröße die elektrische Leistung $P(t)$ angesetzt. Die systembeschreibenden Gleichungen lauten:

$$u(t) = L \frac{di(t)}{dt} + R \cdot i(t) \quad (2.1)$$

$$P(t) = u(t) \cdot i(t) . \quad (2.2)$$

Bezeichnungen und Einheiten sämtlicher Größen sind in Tabelle 2.1 aufgelistet.

Tabelle 2.1: Größen und Einheiten, die zur physikalischen Modellbildung eines Heizlüfters benötigt werden.

| Größe | Bezeichnung | Einheit |
|--------|--------------|----------|
| $u(t)$ | Spannung | V |
| $i(t)$ | Strom | A |
| $P(t)$ | Leistung | W |
| L | Induktivität | H |
| R | Widerstand | Ω |

Wird (2.1) nach $\frac{di(t)}{dt}$ aufgelöst, so kann das in Abbildung 2.1 dargestellte Strukturbild erstellt werden.

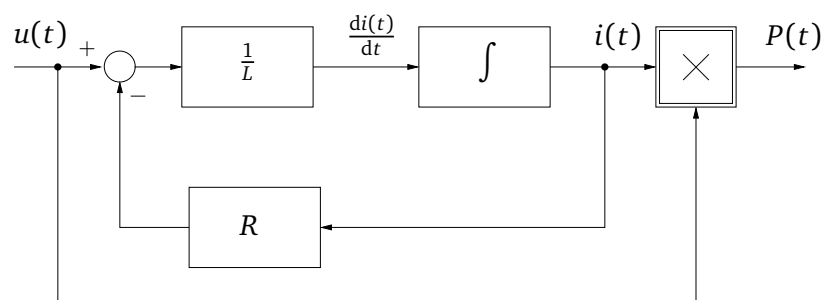


Abbildung 2.1: Strukturbild des Heizlüfters.

2.2 Nichtlineare Modellbildung

Simulink stellt zahlreiche Bibliotheken zum Aufbau von Modellen zur Verfügung. In diesem Versuch werden die im Abbildung 2.2 aufgeführten Simulink-Bibliotheken und Blöcke zur Modellbildung verwendet. Wie ersichtlich, enthält die Bibliothek *User-Defined Functions* unter anderem die Blöcke *Fcn*, *MATLAB Function* und *S-Function*, die im Weiteren getrennt behandelt werden. Die Lösung der Differentialgleichungen erfordert die Benutzung der Integratoren aus der *Continuous*-Bibliothek. Zur Durchführung der

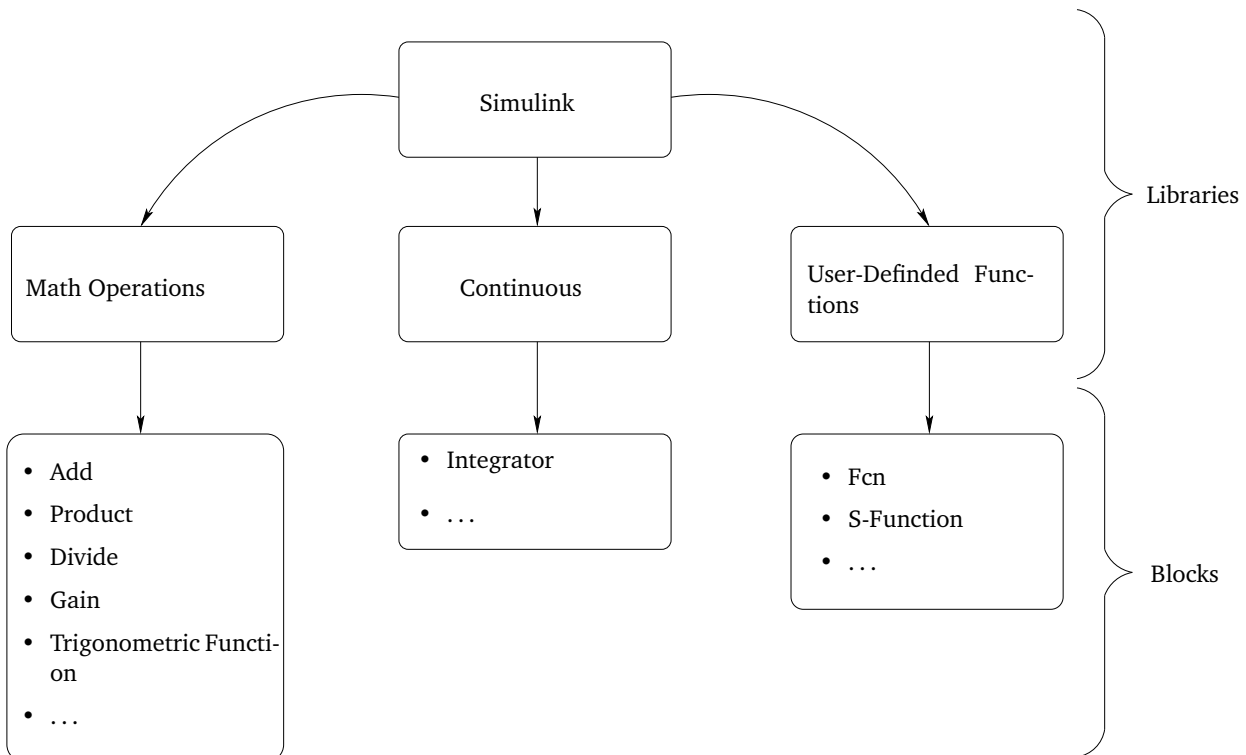


Abbildung 2.2: Simulink-Bibliotheken, die in diesem Versuch zum Aufbau des Modells genutzt werden. Zur Simulation werden außerdem die Bibliotheken *Sources*, *Sinks* und ggf. *Signal Routing* benötigt.

Simulation sind außerdem die Blöcke aus den Bibliotheken *Sources* und *Sinks* notwendig. Oft lässt sich mittels der Bibliothek *Signal Routing* die Übersichtlichkeit der erstellten Modelle steigern.

2.2.1 Modellierung mit Hilfe der Math Operations-Bibliothek

Der Aufbau eines Simulink-Modells mit Hilfe der Math Operations-Bibliothek entspricht der Darstellung des Systems in Form eines Blockschaltbildes, wie es in der Vorlesung SDRT I [10] behandelt wird. Mit Hilfe dieser Blöcke und Integratoren können die Differentialgleichungen nachgebildet und anschließend simuliert werden. In Abbildung 2.3 ist das in Simulink erstellte Modell des Heizlüfters dargestellt.

Für weitere Informationen sei auf das Praktikum MATLAB/Simulink I und die MATLAB-Hilfe verwiesen.

2.2.2 Modellierung mit Hilfe von Fcn-Blöcken

Fcn-Blöcke können lineare und nichtlineare Beziehungen enthalten. Es besteht somit die Möglichkeit, eine nichtlineare Funktion mit Hilfe eines solchen Fcn-Blockes zu realisieren. Fcn-Blöcke haben einen

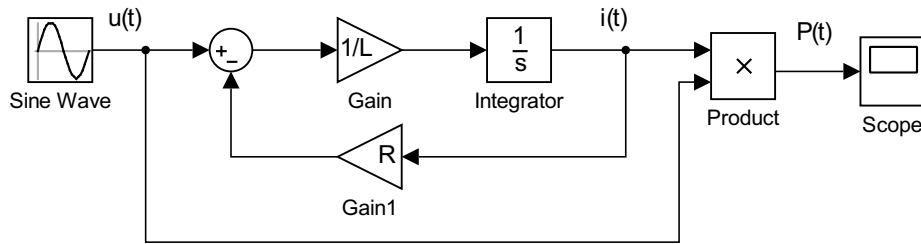


Abbildung 2.3: Heizlüfter als Blockschaltbild in Simulink.

Ein- und einen Ausgang, wobei der Eingang ein Vektor sein kann, der Ausgang dagegen ein Skalar ist. Ein Fcn-Block ist in der Lage

- Numerische Konstanten
- Arithmetische Operationen (+, -, *, /, ^)
- Vergleichende und logische Operationen (==, !=, >, <, >=, <=, &&, ||, !)
- etc.

zu verarbeiten bzw. auszuführen. Für Details sei auf die MATLAB-Hilfe verwiesen. Ein Nachteil besteht darin, dass mit Fcn-Blöcken keine Matrixoperationen durchführbar sind.

In der Struktur gemäß Abbildung 2.4 sind zwei verwendete Fcn-Blöcke und deren Rechenvorschrift erkennbar.

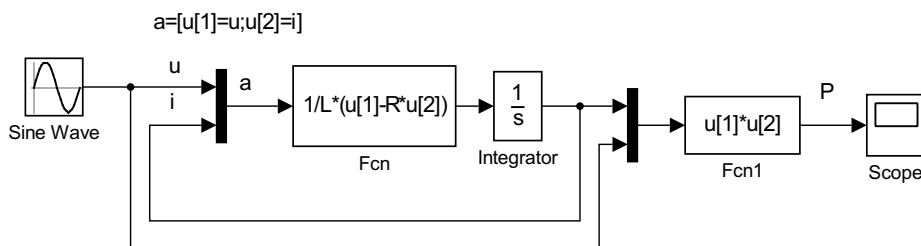


Abbildung 2.4: Aufbau des Systems mit Fcn-Blöcken.

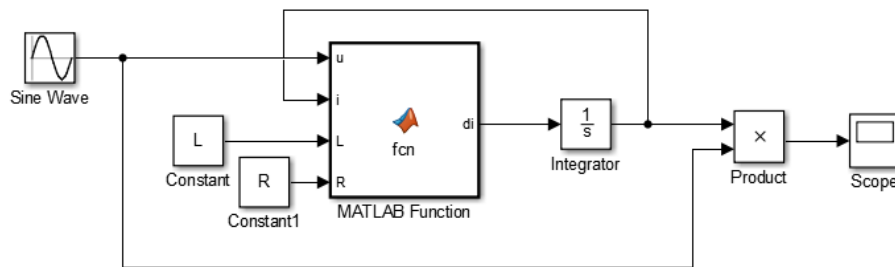
2.2.3 Modellierung mit Hilfe von MATLAB-Funktion-Blöcken

Etwas flexibler ist man mit dem MATLAB-Funktion-Block (*MATLAB Function*). Mit einem Doppelklick auf diesen Block öffnet sich der normale Editor mit einem Funktionsprototypen. Es können dann beliebig Ein- und Ausgänge hinzugefügt werden. D. h. die Argumente der Funktion erscheinen automatisch als Eingangsgrößen des Blocks, die Rückgabewerte als Ausgangsgrößen.¹

In Abbildung 2.5b sind beispielsweise die Eingangsgrößen u , i , L und R als Argumente und di als Rückgabewert definiert. Abbildung 2.5a zeigt die sich damit ergebenden Ein- und Ausgänge im Simulink-Modell. Ein- wie Ausgangswerte können dabei Skalare, Vektoren oder Matrizen sein. (Auch Strukturen/Busse wären möglich.)

Innerhalb der Funktion kann dann im Grunde normaler Matlabcode geschrieben werden. Dabei können beliebig weitere Variablen definiert werden und es können auch Ablaufsteuerungen wie Schleifen oder if-Abfragen verwendet werden.

¹ Darüber hinaus ist es auch möglich, Funktionsargumente als „Parameter“ zu definieren. Diese erscheinen dann nicht als Eingangsgrößen des Blocks, sondern diese werden direkt aus dem Workspace geladen.



(a) Aufbau des Systems mit einem MATLAB-Function-Block

(b) Inhalt des MATLAB-Function-Blocks

Abbildung 2.5: Aufbau des Systems mit einem MATLAB-Function-Block

Zur Durchführung der Simulation wird aus dieser Funktion zunächst C-Code erzeugt. Dadurch ergeben sich folgende Einschränkungen gegenüber normalen MATLAB-Funktionen:

- Es können innerhalb eines MATLAB-Function-Blocks nur Funktionen aufgerufen werden, die die Code-Generierung unterstützen. (Diese Einschränkung kann mit `coder.extrinsic` umgangen werden.)
- Der Coder muss die Größe von Vektoren und Matrizen erkennen können. D.h. alle verwendeten Vektoren und Matrizen sollten zunächst initialisiert werden (z.B. `A = zeros(2, 3)`) bevor auf einzelne Elemente zugegriffen wird.
- Sind Funktionsargumente als Parameter definiert, dann können diese ohne weiteres auch Strukturen sein. Jedoch dürfen diese keine Strings oder Cell-Arrays beinhalten.

Es ist auch möglich, persistent-Variablen innerhalb eines MATLAB-Function-Blocks zu verwenden. Auf diese wird später noch eingegangen. Diese Variablen erlauben es, Werte zwischen den einzelnen Auswertungen eines MATLAB-Function-Blocks zu speichern.

Allerdings ist dabei zu beachten, dass dies zu unerwarteten Ergebnissen führen kann! Lediglich bei Simulationen mit fester Schrittweite und diskreten Systemen bzw. dem ode1-Verfahren wird jeder Block pro Zeitschritt auch nur einmal aufgerufen. Die Solver ode2 bis ode8 besitzen zwar eine feste Schrittweite, führen jedoch mehrere Berechnungen pro Zeitschritt aus. Und bei den Solvern mit variabler Schrittweite kann keine feste Anzahl der Funktionsauswertungen pro Zeitschritt angegeben werden. In diesen Fällen ist es anzuraten, auf persistent-Variablen zu verzichten und die benötigten Werte über einen externen Memory-Block zurückzuführen.

2.2.4 Modellierung mit Hilfe von S-Functions

Eine weitere Methode, die Modelle zu simulieren, ist die Verwendung von sog. *S-Functions*. Ein S-Function-Block mit eingetragenen Parametern greift auf einen M-File mit einer speziellen Struktur zu.

Die übergebenen Parameter werden in der S-Function weiterverwendet. Diese Schnittstelle zwischen Simulink und MATLAB wird mittels API-Befehlen (The Stateflow[®] Application Programming Interface) realisiert. Generell gibt es zwei Arten solcher S-functions, nämlich

- Level-1 S-function: geschrieben zur Verwendung mit der Simulink-Version 2.1. Die Kompatibilität mit allen Simulink-Versionen ist gewährleistet.
- Level-2 S-function: mit der Version 2.2 eingeführt. Im Allgemeinen wird empfohlen, die verwendeten Level-1 M-Files in die neueren Level-2 M-Files umzuschreiben und diese zu verwenden.

Diese Files können direkt in MATLAB-Programmiersprache M, oder auch in C, C++, Fortran und Ada geschrieben werden. Bei den letzteren bedarf es jedoch einer Kompilierung als MEX-Dateien (MATLAB Executable File), wodurch die entsprechenden Bezeichnungen entstehen:

- C MEX-file S-Function
- C++ MEX-file S-Function
- Fortran MEX-file S-Function
- Ada MEX-file S-Function
- M-file S-Function (direkt in M geschrieben)

In diesem Praktikum wird eine Level-2 M-File S-Function verwendet. Im Anhang ist ein Template abgelegt. Dieses gibt die Struktur vor, so dass die Funktionsweise einer S-Function deutlich wird. In der Funktion `setup` werden die Eigenschaften der S-Function definiert, sowie Teilfunktionen registriert, die das Verhalten des Systems charakterisieren und während der Simulation von Simulink aufgerufen werden. Innerhalb der Teilfunktionen können sämtliche MATLAB-Befehle verwendet werden, auch Matrizen können verarbeitet werden. Somit ist es notwendig, das M-File in Bezug auf das zu untersuchende System selbstständig anzupassen bzw. zu ergänzen.

Programmierung des Heizlüftermodells

Der genauere Aufbau und die Funktionsweise einer S-Function soll anhand des Heizlüfter-Modells aufgezeigt werden. Das entsprechende Simulink-Modell ist in Abbildung 2.6a dargestellt.

Das Fenster *Function Block Parameters* in Abbildung 2.6b enthält den Namen der S-Function `luefterSFcn`, sowie die übergebenen Parameter `L` und `R`. Im entsprechenden M-File lautet die Deklaration der Funktion:

```
function luefterSFcn(block)
```

Die Programmierung von Level-2 M-File S-Functions erfolgt objektorientiert. Die Definition von Eigenschaften des Modells erfolgt durch Methodenaufrufe auf dem Objekt `block`; die Definition der Funktionalität erfolgt durch Implementation (und Registrierung mittels `block.RegBlockMethod()`) von Methoden.

Nachfolgend sind einige wichtige Variablen (Attribute) genannt:

- `block.ContStates.Data`
Zustandsvektor x des Systems
- `block.InputPort(i).Data`
Eingang u_i des Systems
- `block.OutputPort(j).Data`
Ausgang y_j des Systems
- `block.DialogPrm(k).Data`
repräsentiert den Parameter k , in der Reihenfolge, wie er im Dialogfeld eingegeben wurde.

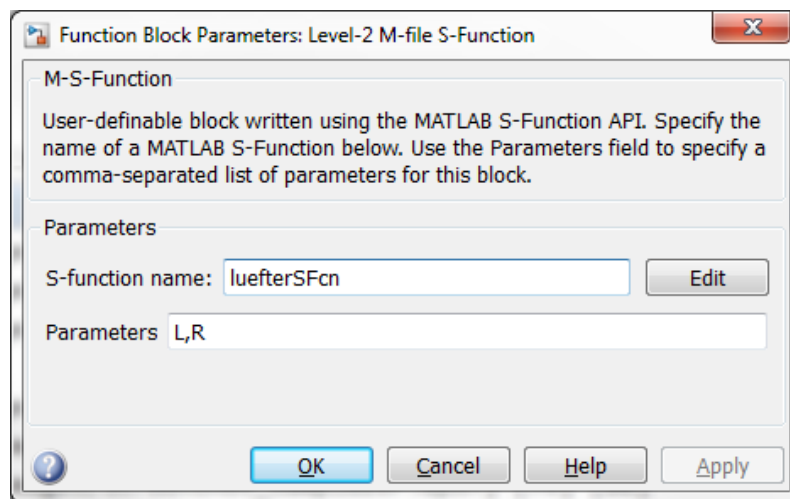
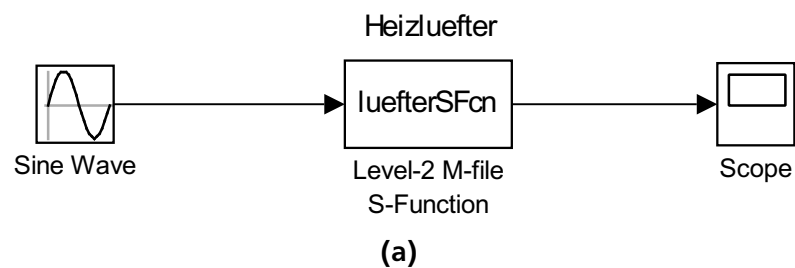


Abbildung 2.6: Modellierung des Heizlüfters mit Hilfe von S-Function: (a) Strukturbild mit dem S-Function-Block; (b) Eingabe der Parameter für den S-Function-Block.

Wichtige Attribute, die zur Konfiguration des Modells benötigt werden, sind:

- `block.NumContStates`
Definiert die Ordnung des Systems (zeitkontinuierliche Simulation).
- `block.NumInputPorts`
Bestimmt die Anzahl der Systemeingänge.
- `block.InputPort(1).DirectFeedthrough`
Gibt an, dass ein Durchgriff vom Eingang u_1 besteht.
- `block.NumOutputPorts`
Bestimmt die Anzahl der Systemausgänge.
- `block.NumDialogPrms`
Bestimmt die Anzahl der im Dialog vorzugebenden Parameter.
- `block.SampleTimes`
Hier können verschiedene Abtaststraten eingestellt werden. Soll eine zeitkontinuierliche Simulation durchgeführt werden, ist `[0 0]` anzugeben.

In der Unterfunktion `setup` werden diese Eigenschaften zugewiesen. Diese Funktion wird einmalig zu Beginn der Simulation ausgeführt, um Schnittstellen und Simulationseigenschaften des Blocks festzulegen.

Ein Beispiel zeigt Listing 2.1.

Listing 2.1: Anpassung der Unterfunktion `setup` an das Heizlüfter-Modell. Quelle: S-function template.

```
1 function setup(block)

    % Anzahl der Ein-/Ausgänge
    block.NumInputPorts = 1;
    block.NumOutputPorts = 1;

6
    % Eigenschaften des 1. Eingangs
    block.InputPort(1).Dimensions = 1;
    block.InputPort(1).DatatypeID = 0; % double
    block.InputPort(1).Complexity = 'Real';
11 block.InputPort(1).DirectFeedthrough = true;
    block.InputPort(1).SamplingMode = 'Sample';

    % Eigenschaften des 1. Ausgangs
    block.OutputPort(1).Dimensions = 1;
16 block.OutputPort(1).DatatypeID = 0; % double
    block.OutputPort(1).Complexity = 'Real';
    block.OutputPort(1).SamplingMode = 'Sample';

21
    % Anzahl der Zustände
    block.NumContStates = 1;

    % Anzahl der Parameter
    block.NumDialogPrms = 2;

26
```

```

% Abtastzeit definieren -> zeitkontinuierlich
block.SampleTimes = [0 0];

```

31

```

% weitere Methoden registrieren
block.RegBlockMethod('InitializeConditions', @InitializeConditions);
block.RegBlockMethod('Outputs', @Outputs);
36 block.RegBlockMethod('Derivatives', @Derivatives);
block.RegBlockMethod('Terminate', @Terminate);

```

end

In der Unterfunktion InitializeConditions (Listing 2.2) werden die Anfangsbedingungen festgelegt. Sie wird bei jedem Start der Simulation ausgeführt.

In diesem Beispiel wird davon ausgegangen, dass zu Beginn kein Strom fließt ($i(t_0) = 0$). Sollen Anfangsbedingungen frei definiert werden können, so muss dies über weitere Parameter geschehen. Die Parameterliste des Konfigurationsdialogs könnte z. B. zu L, R, I_0 erweitert werden.

Listing 2.2: Anpassung der Unterfunktion Derivatives an das Heizlüfter-Modell. Quelle: S-function template.

```

1 function InitializeConditions(block)

    % Strom i(t=0) = 0
    block.ContStates.Data = 0;

6 end

```

In der Unterfunktion Derivatives sind die Ableitungen der Zustände zu berechnen. Hier gilt

$$\frac{di(t)}{dt} = \frac{u(t)}{L} - \frac{R}{L} \cdot i(t), \quad (2.3)$$

was zu dem im Listing 2.3 dargestellten Quellcode führt.

Listing 2.3: Anpassung der Unterfunktion Derivatives an das Heizlüfter-Modell. Quelle: S-function template.

```

function Derivatives(block)

    % Parameter auslesen
4   L = block.DialogPrm(1).Data;
   R = block.DialogPrm(2).Data;

    % Zustand auslesen
   i = block.ContStates.Data;

9   % Eingang auslesen
   u = block.InputPort(1).Data;

```

```

14      % Ableitung berechnen
      i_d = (u - i*R)/L;

19

      % Ableitung zuweisen
      block.Derivatives.Data = i_d;

end

```

Als Ausgang soll die nichtlineare Beziehung $i(t) \cdot u(t)$ berechnet werden. Auf $u(t)$ kann somit nur zugegriffen werden, sofern ein Durchgriff existiert (was im Listing 2.2 bereits festgelegt ist). Somit nimmt die Unterfunktion Outputs die im Listing 2.4 dargestellte Struktur an.

Listing 2.4: Anpassung der Unterfunktion Outputs an das Heizlüfter-Modell. Quelle: S-function template.

```

function Outputs(block)
2
    % Zustand auslesen
    i = block.ContStates.Data;

    % Eingang auslesen
7    u = block.InputPort(1).Data;

    % Ausgang berechnen
    p = u * i;

12

    % Ausgang zuweisen
    block.OutputPort(1).Data = p;

17 end

```

Die Funktionen Derivatives und Outputs werden während der Simulation fortlaufend ausgeführt. Für weitere Informationen sei auf die MATLAB-Hilfe **Simulink** → **Overview of S-Functions** → **How S-Functions Work** → **Simulation Stages** verwiesen.

Soll ein diskretes Modell verwendet werden, muss anstatt der Funktion Derivatives die Funktion Update verwendet werden, die zuvor mit dem Aufruf `block.RegBlockMethod` zu registrieren ist.

Kontrolle: Anhand der systembeschreibender Gleichungen (2.3) und (2.2) sollte schließlich überprüft werden, ob alle darin enthaltenen Informationen an das M-File übergeben wurden.

S-Functions bieten einen großen Funktionsumfang, benötigen jedoch eine vergleichsweise umfangreiche Initialisierung, die eine Verwendung bei derart kleinen Simulationsmodellen kaum rechtfertigt. Umfangreichere Modelle mit steigender Komplexität können jedoch relativ komfortabel beschrieben und simuliert werden können. Desweiteren können Änderungen an den Modellen recht schnell vorgenommen werden.

2.3 Lineare Modellbildung

In vielen Fällen liegen Modellgleichungen in nichtlinearer Form vor. Für die Analyse oder auch für einen Reglerentwurf ist allerdings häufig ein in Bezug auf einen bestimmten Arbeitspunkt linearisiertes Modell ausreichend. Dabei muss beachtet werden, dass das linearisierte Modell das Verhalten des nichtlinearen Modells nur in der Nähe des Arbeitspunktes ausreichend genau beschreibt. Als Vorteil können einfachere Zusammenhänge und Entwurfsmethoden der linearen Theorie angegeben werden.

Das allgemeine Vorgehen bei einer Linearisierung von nichtlinearen Differentialgleichungen kann [10, 2] entnommen werden.

Das linearisierte Beispielsystem

Die systembeschreibende DGL gemäß (2.3) soll um den Arbeitspunkt (U_0, I_0) linearisiert werden. Die Notation wird wie folgt gewählt (siehe DIN EN 60027-6:2008-02):

- $u(t) = U_0 + \Delta u(t)$ ist die Eingangsgröße,
- U_0 stellt den Wert der Eingangsgröße am Arbeitspunkt (AP) dar,
- $\Delta u(t)$ repräsentiert die Abweichung vom Arbeitspunkt.

Arbeitspunktbestimmung: Die Werte von Ein- und Ausgangsgröße am Arbeitspunkt ergeben sich über die Bedingung

$$\frac{di(t)}{dt} = 0$$

woraus

$$U_0 = R \cdot I_0$$

folgt.

Linearisierung der Gleichung (2.1): Entwickelt man die rechte Seite von

$$0 = \underbrace{L \frac{di(t)}{dt} + R \cdot i(t) - u(t)}_{f\left(\frac{di(t)}{dt}, i(t), u(t)\right)}$$

in eine Taylorreihe um den Arbeitspunkt, erhält man

$$\begin{aligned} 0 &= \left. \frac{\partial f\left(\frac{di(t)}{dt}, i(t), u(t)\right)}{\partial \frac{di(t)}{dt}} \right|_{\substack{i(t)=I_0 \\ u(t)=U_0}} \Delta \frac{di(t)}{dt} + \left. \frac{\partial f\left(\frac{di(t)}{dt}, i(t), u(t)\right)}{\partial i(t)} \right|_{\substack{i(t)=I_0 \\ u(t)=U_0}} \Delta i(t) + \\ &\quad + \left. \frac{\partial f\left(\frac{di(t)}{dt}, u(t), i(t)\right)}{\partial u(t)} \right|_{\substack{i(t)=I_0 \\ u(t)=U_0}} \Delta u(t) \\ &= L \cdot \Delta \frac{di(t)}{dt} + R \cdot \Delta i(t) - \Delta u(t). \end{aligned}$$

Somit ergibt sich die Differentialgleichung

$$\frac{di(t)}{dt} = -\frac{R}{L} \Delta i(t) + \frac{1}{L} \Delta u(t).$$

Analog erfolgt die Linearisierung der Gleichung (2.2),

$$0 = \underbrace{u(t)i(t) - P(t)}_{f(P(t), u(t), i(t))},$$

für die sich unter Vernachlässigung der Glieder höherer Ordnung

$$0 = U_0 \cdot \Delta i(t) + I_0 \cdot \Delta u(t) - \Delta P(t)$$

bzw.

$$\Delta P(t) = U_0 \cdot \Delta i(t) + I_0 \cdot \Delta u(t)$$

ergibt.

Beachten Sie: Die im linearisierten Modell auftretenden Variable sind auf den Arbeitspunkt bezogen. Die vom Heizlüfter umgesetzte elektrische Leistung ergibt sich daher (bei Vernachlässigung der Glieder höherer Ordnung) zu

$$P(t) = P_0 + \Delta P(t), \quad (2.4)$$

$P_0 = U_0 I_0$. Aus Gründen der Bequemlichkeit wird jedoch im Allgemeinen in der linearisierten Systemdarstellung $P(t)$ statt $\Delta P(t)$ geschrieben. Beim Vergleich von nichtlinearem und linearisiertem Modell sowie bei der Implementierung einer (linearen) Regelung an der nichtlinearen Strecke muss dies berücksichtigt werden.

2.3.1 Modellbildung im Zustandsraum

Für die Modellbildung im Zustandsraum bietet Simulink zusätzlich zu den Blockschaltbildern undFcn-Blöcken die Möglichkeit, lineare Zustandsraummodelle zu verwenden. Dem *State-space model*-Block müssen die Matrizen des Zustandsraummodells übergeben werden. Dies soll anhand des Beispielsystems gezeigt werden. Dazu werden die Matrizen **A**, **B**, **C** und **D** der Zustandsraumdarstellung

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)\end{aligned}$$

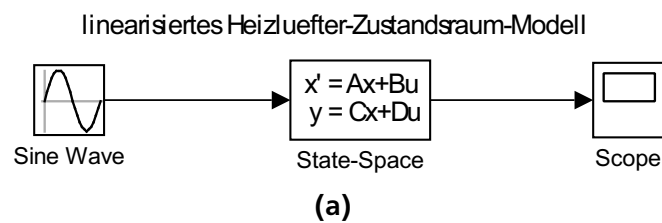
vorher im Command Window oder einem M-File (siehe Listing 2.5) definiert.

Listing 2.5: Zustandsraumdarstellung des linearisierten Heizlüfter-Modells.

```
% Zustandsraummodell des Beispielsystems
% x_d=A*x+B*u
3 % y=C*x+D*u
A=[-R/L]
B=[1/L]
C=[U_0]
D=[I_0]
```

In Abbildung 2.7 ist der Aufbau und die Eigenschaften (Properties) des *State-space model*-Blockes aufgezeigt.

Als Parameter können unter anderem die Anfangsbedingungen (Initial conditions) vorgegeben werden.



Function Block Parameters: State-Space

State Space

State-space model:
 $\frac{dx}{dt} = Ax + Bu$
 $y = Cx + Du$

Parameters

A:

B:

C:

D:

Initial conditions:

Absolute tolerance:

State Name: (e.g., 'position')

OK Cancel Help Apply

(b)

Abbildung 2.7: Modellierung des Heizlüfters in ZRD: (a) Strukturbild mit dem State-Space-Block; (b) Eingabe der Parameter für den State-Space-Block.

2.4 Simulation

Auf die Grundlagen einer Simulation in Simulink soll hier nicht eingegangen werden. Hierzu sei auf das Praktikum MATLAB/Simulink I verwiesen.

2.4.1 Simulationseinstellungen

In der Regel werden dynamische Systeme durch gewöhnliche Differentialgleichungen (ordinary differential equation ODE) beschrieben. Für die Lösung solcher DGLs werden in MATLAB verschiedene Algorithmen verwendet, die z. B. im Praktikum MATLAB/Simulink I vorgestellt werden. In Tabelle 2.2 ist ein kurzer Auszug aus der Tabelle der entsprechenden Versuchsbeschreibung dargestellt. Simulink

Tabelle 2.2: Übersicht der Algorithmen zur Lösung der Differentialgleichungen.

| <i>Solver</i> | <i>Beschreibung</i> | <i>Verfahren</i> |
|---------------|------------------------------------|------------------|
| ode45 | Runge Kutta Verfahren | explizit |
| ode23 | Runge Kutta Verfahren | |
| ode113 | Adams Bashford Moulton Verfahren | |
| ode15s | Gear's Verfahren | implizit |
| ode23s | modifiziertes Rosenbrock-Verfahren | |
| ode23tb | implizites Runge Kutta Verfahren | |

bietet die Möglichkeit den gewünschten Algorithmus einzustellen. Diese Optionseinstellungen befinden sich im Modellfenster unter *Simulation* → *Configuration Parameters*. Im Tab *Solver* können die gewünschten Einstellungen gemacht werden (siehe Abbildung 2.8). Teilweise weisen die erzeugten Kurvenverläufe

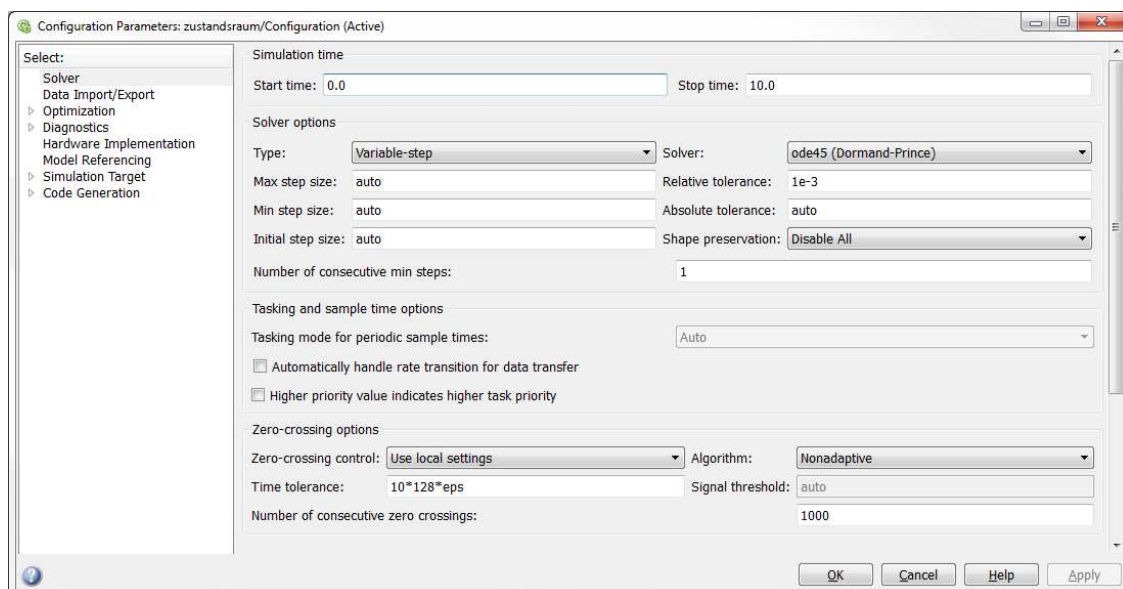


Abbildung 2.8: Configuration Parameters - Fenster.

einen kantigen Verlauf auf. Dies kann behoben werden, indem die maximale Schrittweite (*Max step size*) des Algorithmus verkleinert wird. Dabei nimmt die Simulationsdauer zu.

2.4.2 Darstellung der Ergebnisse

In diesem Abschnitt wird beispielhaft dargestellt, wie Sie Plots für die Dokumentation erstellen und Quellcode in die Dokumentation einbinden können.

Erzeugen von Plots

Bei einer Simulation ist gelegentlich von Interesse, bestimmte Daten zwischen dem Command Window und dem Simulink-Modell zu transferieren. Dies geschieht mit Hilfe von *From Workspace* und *To Workspace* Blöcken. Optional kann diese Einstellung in dem Configuration Parameters-Dialog im Tab *Data Import/Export* vorgenommen werden.

Eine weitere Möglichkeit besteht darin, die in einem Scope dargestellten Daten in den Workspace zu speichern. Dies kann im Scope im Dialog *Parameters* im Tab *Data history* mit *Save data to workspace* aktiviert werden. Es ist im Allgemeinen sinnvoll, die Einstellung *Format: Structure with time* zu wählen.

Für eine erleichterte Auswertung der Simulationsergebnisse sollten diese in sinnvoll gewählten Diagrammen dargestellt werden. Hierzu sind folgende Befehle sehr nützlich:

- `figure`
erstellt ein neues Figure-Objekt.
- `subplot`
erstellt ein Unterdiagramm in einem Figure. Somit können zusammengehörige Diagramme in unterschiedlichen Achsensystemen unter- und nebeneinander gezeichnet werden.
- `plot`
erstellt eine oder mehrere Kurve(n). Durch Verwendung unterschiedlicher Linientypen können mehrere Kurven, die in ein Achsensystem gezeichnet wurden, unterschieden werden. Achten Sie bei der Erstellung von Diagrammen darauf, dass diese auch schwarz/weiß ausgedruckt gut ablesbar sind.
- `title`, `xlabel`, `ylabel`
werden verwendet, um einen Titel sowie Achsenbeschriftungen anzubringen.
- `legend`
fügt eine Legende hinzu.
- `linkaxes`
synchronisiert die Achsen mehrerer Achsensysteme. Im Allgemeinen ist es sinnvoll, die x-Achsen zusammengehöriger Diagramme zu synchronisieren.
- `print`
druckt den Inhalt eines Figure-Objekts. Dieser Befehl kann auch verwendet werden, um Grafikdateien zu erstellen. Hierzu muss der benötigte Treiber (z. B. *depsc* oder *dbmp*) ausgewählt werden.

Um weitere Informationen zu diesen Befehlen zu erhalten, nutzen sie die Onlinehilfe, z. B. `help plot` oder `doc plot`.

Im Listing 2.6 sind die Befehle beispielhaft verwendet, um Abbildung 2.9 zu erstellen.

Listing 2.6: Beispielbild erstellen

```
%% Beispieldaten
vt = 0:0.1:10;
3  vu = sin( vt );
```



```

vy1 = vu .^ 2;
vy2 = vu * 0.5;

%% Grafische Darstellung der Daten
8 hFig = figure;

vhAx(1) = subplot( 3, 1, 1 );
plot( vt, vu, 'k' );
title( 'Ein- / Ausgangsverhalten' );
13 ylabel( 'Eingang u in V' );

vhAx(2) = subplot( 3, 1, 2:3 );
plot( vt, vy1, 'k', vt, vy2, 'k--' );
ylabel( 'Ausgänge y1, y2 in m' );
18 xlabel( 'Zeit t in s' );
legend( 'y1', 'y2', 'Location', 'SouthEast' );

linkaxes( vhAx, 'x' );

23 % hiermit kann ein gewünschtes Bildformat eingestellt werden
set( hFig, 'Position', [500 100 600 400] )

%% Bild speichern
% damit das Bild das mit set( ..., 'Position', .. )
28 % eingestellte Format beibehält
set( gcf, 'PaperPositionMode', 'auto' );
print( '-depsc', 'Beispielbild.eps' )

```

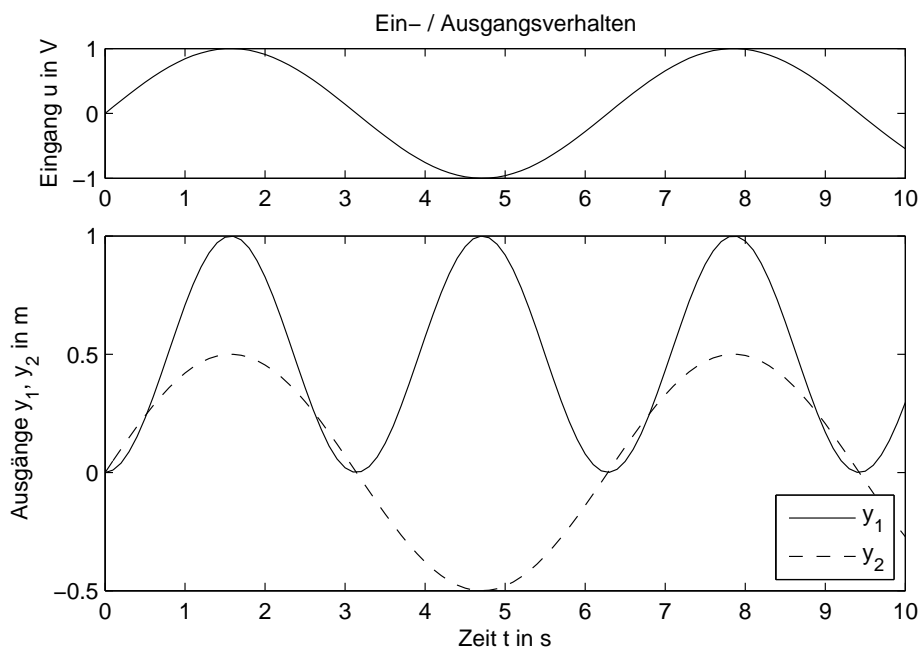


Abbildung 2.9: Beispielbild

Einbinden von Quelltext

Wenn Sie \LaTeX verwenden, um Ihre Versuchsprotokolle zu erstellen, können Sie den Quelltext der erstellten Funktionen mit `\lstinputlisting` aus dem Paket `listings` in Ihren Text einbinden.

Listing 2.7 zeigt einen Ausschnitt des \LaTeX -Quelltexts, um das in Listing 2.6 dargestellte MATLAB-Skript einzubinden. Hierzu muss in der Präambel des Dokuments das Paket `listings` geladen und der Stil (hier genannt: `Matlab_colored`) definiert werden. Dies kann man den in Listing 2.8 angegebenen Befehlen geschehen.

Listing 2.7: \LaTeX -Befehl zur Einbindung von Listing 2.6

Im Listing `\ref{V1:lst:PlotsInMatlab}` sind die Befehle beispielhaft →
← verwendet, um Abbildung `\ref{V1:fig:Beispielbild}` zu erstellen.

```
% Listing des Quellcodes einfügen
\lstinputlisting[style=Matlab_colored,caption={Beispielbild erstellen}, →
  <label={V1:lst:PlotsInMatlab}]
5      {./Versuch1/Vorbereitung/Matlab/Beispielbild.m}
```

Listing 2.8: \LaTeX -Befehle zur Einrichtung von Listings

```
% Paket listings einbinden
\usepackage{listings}

% Stil Matlab_colored definieren
5 \lstdefinestyle{Matlab_colored}
  {
    language = Matlab,
    tabsize = 4,
    framesep = 3mm,
10    frame=tb,
    classoffset = 0,
    basicstyle = \ttfamily,
    keywordstyle = \bfseries\color{rgb}{0,0,1},
    commentstyle = \itshape\color{rgb}{0.133,0.545,0.133},
15    stringstyle = \color{rgb}{0.627,0.126,0.941},
    extendedchars = true,
    breaklines = true,
    prebreak = \text{rightarrow},
    postbreak = \text{leftarrow},
20    numbers = left,
    numberstyle = \tiny,
    stepnumber = 5
  }
```

3 Vorbereitungsaufgaben

Folgende Aufgaben sollen zur Durchführung des Versuches vorbereitet werden.

Aufgabe 1.1 (Vorbereitung):

Stellen Sie in Bezug auf Abbildung 1.1 und Tabelle 4.1 die nichtlinearen Bewegungsgleichungen des Schlitten-Pendel-Modells auf. Dabei bietet sich die Anwendung von LAGRANGESchen Gleichungen zweiter Art an.

Es sollen folgende Punkte beachtet werden:

- Die Schlittenposition x_S und der Pendelstabwinkel φ stellen die generalisierten Koordinaten dar.
- Die Bewegung des Pendelstabes ist von der Fallbeschleunigung $g \approx 9,81 \text{ m/s}^2$ abhängig.
- Der Pendelstab kann als ein dünner Stab mit einem Massenträgheitsmoment (bezüglich des Mittelpunktes bei $\frac{l}{2}$) mit $J = \frac{1}{12} m_P l^2$ charakterisiert werden.
- Reibungen $F_R = R_S \dot{x}_S$ und $M_R = R_P \dot{\varphi}$ sollen berücksichtigt werden.
- Als Zwischenergebnis sei angegeben [1]:

$$(m_S + m_P) \cdot \ddot{x}_S + m_P \ddot{\varphi} \frac{l}{2} \cos \varphi - m_P \dot{\varphi}^2 \frac{l}{2} \sin \varphi = F - R_S \dot{x}_S \quad (3.1)$$

$$\frac{1}{2} l m_P \cdot \left(\ddot{x}_S \cos \varphi + \frac{2}{3} l \ddot{\varphi} + g \sin \varphi \right) = -R_P \dot{\varphi} . \quad (3.2)$$

- In beiden Gleichungen treten $\ddot{\varphi}$ und \ddot{x}_S auf. Eliminieren Sie in einer Gleichung \ddot{x}_S und lösen Sie diese nach $\ddot{\varphi}$, die andere nach \ddot{x}_S auf.

Aufgabe 1.2 (Vorbereitung):

- Linearisieren Sie die gewonnenen Bewegungsgleichungen um die Arbeitspunkte $\varphi_{AP} = 0 \text{ rad}$ und $\varphi_{AP} = \pi \text{ rad}$.
- Schließlich sind die entsprechenden *vollständigen* Zustandsraumdarstellungen mit dem Zustandsvektor

$$\Delta \mathbf{x} = [\Delta x_S \quad \Delta \dot{x}_S \quad \Delta \varphi \quad \Delta \dot{\varphi}]^T$$

und dem Ausgangsvektor

$$\Delta \mathbf{y} = [\Delta x_S \quad \Delta \varphi]^T$$

zu wählen.

Protokoll:

Dokumentieren Sie die wichtigsten Rechenschritte in Ihrem Protokoll.

4 Anhang

4.1 Variable und Parameter des Schlitten-Pendel-Systems

Tabelle 4.1: Größen und Einheiten, die zur physikalischen Modellbildung des Schlitten-Pendel-Systems gemäß Abbildung 1.1 benötigt werden (aus [6]).

| Größe | Bezeichnung | Wert/Einheit |
|----------------------------------|---|---|
| $x_s(t), x_p(t), y_s(t), y_p(t)$ | Position | m |
| $F(t)$ | Kraft | N |
| $F_R(t)$ | viskose Reibung des Schlittens auf der Doppelspurführung, $F_R = R_S \dot{x}_s$ | N |
| $M_R(t)$ | Reibungsdrehmoment der Pendelstablagerung, $M_R = R_p \dot{\varphi}$ | Nm |
| $\varphi(t)$ | Winkel | rad |
| J | Massenträgheitsmoment des Pendels bezüglich der Achse im Mittelpunkt $l/2$ | kg m^2 |
| l | Länge des Pendels | 0,41 m |
| g | Fallbeschleunigung | $9,81 \text{ m/s}^2$ |
| m_s | Masse des Schlittens | 7,055 kg |
| m_p | Masse des Pendels | 0,177 kg |
| R_S | Reibungskoeffizient des Schlittens auf der Doppelspurführung | $1,4 \frac{\text{Ns}}{\text{m}}$ |
| R_p | Reibungskoeffizient der Pendelstablagerung | $5,1 \cdot 10^{-3} \frac{\text{Nm}}{\text{rad s}^{-1}}$ |

4.2 S-Function-Template

Im nachfolgenden Listing ist das S-Function-Template aufgeführt. Dieses ist als Grundkonstrukt zur Beschreibung linearer und nichtlinearer Modelle anhand gewöhnlicher Differentialgleichungen zu verwenden.

Hinweis: Das Template ist für die Versuchsdurchführung bei den Praktikumsunterlagen zur Verfügung gestellt.

Listing 4.1: Auflistung des S-Function-Template

```
function sFunctionTemplate(block)
% sFunctionTemplate
3 % Dies ist eine Vorlage für eine Level-2 M-File S-Function.
```

```

        setup(block);

end

8

% *****
% Initialisierung
% *****
13 function setup(block)

    % Anzahl der Ein-/Ausgänge
    block.NumInputPorts = 1;
    block.NumOutputPorts = 1;

18

    % Eigenschaften des 1. Eingangs
    block.InputPort(1).Dimensions = 1;
    block.InputPort(1).DatatypeID = 0; % double
    block.InputPort(1).Complexity = 'Real';
23 block.InputPort(1).DirectFeedthrough = false;
    block.InputPort(1).SamplingMode = 'Sample';

    % Eigenschaften des 1. Ausgangs
    block.OutputPort(1).Dimensions = 1;
28 block.OutputPort(1).DatatypeID = 0; % double
    block.OutputPort(1).Complexity = 'Real';
    block.OutputPort(1).SamplingMode = 'Sample';

    % Anzahl der Zustände
33 block.NumContStates = 2;

    % Anzahl der Parameter
    block.NumDialogPrms = 3;

38

    % Abtastzeit definieren -> zeitkontinuierlich
    block.SampleTimes = [0 0];

43

    % weitere Methoden registrieren
    block.RegBlockMethod('InitializeConditions', @InitializeConditions);
    block.RegBlockMethod('Outputs', @Outputs);
48 block.RegBlockMethod('Derivatives', @Derivatives);
    block.RegBlockMethod('Terminate', @Terminate);

end

53

% *****

```

```

% Anfangsbedingungen setzen
% *****
function InitializeConditions(block)
58
    % Die Anfangsbedingungen werden als Parameter übergeben
    x1_0 = block.DialogPrm(1).Data;
    x2_0 = block.DialogPrm(2).Data;

63    block.ContStates.Data = [x1_0; x2_0];

end

68 % *****
% Ausgänge berechnen
% *****
function Outputs(block)

73    % Zustände auslesen
    x = block.ContStates.Data;
    x1 = x(1);

    block.OutputPort(1).Data = x1;

78 end

% *****
83 % Ableitungen berechnen
% *****
function Derivatives(block)

    % Parameter auslesen
88    p1 = block.DialogPrm(3).Data;

    % Zustände auslesen
    x = block.ContStates.Data;
    x2 = x(2);

93    % Eingang auslesen
    u = block.InputPort(1).Data(1);

98    % Ableitungen berechnen
    x1_d = x2;
    x2_d = p1 * u;

103

```

```
    % Ableitungen zuweisen
    block.Derivatives.Data = [x1_d, x2_d];

108 end

% *****
% Aufräumen (wenn nötig)
113 % *****
function Terminate(block)
end
```



Versuch 2

Steuerbarkeit und Beobachtbarkeit

Im vorigen Versuch wurde für das Pendel-Schlitten-System ein nichtlineares Modell erstellt. Aus diesem wurden durch Linearisierung zwei lineare Zustandsraummodelle für die Arbeitspunkte $\varphi_{AP} = 0$ und $\varphi_{AP} = \pi$ abgeleitet.

In diesem Versuch soll das System auf verschiedene Eigenschaften, die für den Reglerentwurf von Bedeutung sind, untersucht werden.

| | | |
|----------|--|-----------|
| 5 | Systeme im Zustandsraum | 33 |
| 5.1 | Hinweis zur Notation | 33 |
| 6 | Normalformen des Zustandsraummodells | 35 |
| 6.1 | Diagonalform als kanonische Normalform | 35 |
| 6.2 | Konjugiert komplexe Eigenwerte als reelle Parameter in Blöcken | 35 |
| 7 | Steuerbarkeit | 36 |
| 7.1 | Steuerbarkeitskriterium von KALMAN | 36 |
| 7.2 | Steuerbarkeitskriterium von GILBERT | 36 |
| 7.3 | Steuerbarkeitskriterium von HAUTUS | 37 |
| 8 | Beobachtbarkeit | 38 |
| 8.1 | Beobachtbarkeitskriterium von KALMAN | 38 |
| 8.2 | Beobachtbarkeitskriterium von GILBERT | 38 |
| 8.3 | Beobachtbarkeitskriterium von HAUTUS | 38 |

| | | |
|----------|--------------------|-----------|
| 9 | MATLAB | 39 |
| 9.1 | Matrizen | 39 |

5 Systeme im Zustandsraum

Im Folgenden wird angenommen, dass ein System durch lineare Differentialgleichungen erster Ordnung beschrieben werden kann und folglich in der Zustandsraumdarstellung der Form

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$$

darstellbar ist.

Es existieren einige Kriterien, die anhand der Zustandsraumdarstellung Aussagen über die Stabilität und Steuerbarkeit eines Systems liefern. In der Vorlesung SDRT II von Professor Adamy werden diese Kriterien eingehend behandelt.

In diesem Versuch sollen die Eigenschaften des Schlitten-Pendel-Modells anhand der im Folgenden vorgestellten Kriterien untersucht werden.

5.1 Hinweis zur Notation

Die im Weiteren beschriebene Notation ist auf DIN EN 60027-6:2008-02 und an das SDRT II -Skript [2] angelehnt. Damit soll eine eindeutige Bezeichnung aller in dieser Versuchsbeschreibung aufgeführten Größen sichergestellt werden. Wird von einem MIMO-System in Zustandsraumdarstellung

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t)$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t)$$

ausgegangen, so haben die einzelnen Matrizen und Vektoren folgende Bedeutung:

1. Matrizen mit konstanten Elementen bzw. Komponenten:

- **A**: $(n \times n)$ -Systemmatrix mit konstanten Elementen.
- **B**: $(n \times m)$ -Eingangsmatrix. Das System besitzt m -Eingänge.
- **C**: $(r \times n)$ -Ausgangsmatrix. Das System besitzt r -Ausgänge.
- **D**: $(r \times m)$ -Durchgangsmatrix.

2. Vektoren mit zeitveränderlichen Komponenten:

- **x**(t): $(n \times 1)$ -Zustandsvektor
- **u**(t): $(m \times 1)$ -Eingangsvektor
- **y**(t): $(r \times 1)$ -Ausgangsvektor.

Es ist ersichtlich, dass die Matrizen durch große und die Vektoren durch entsprechende kleine Buchstaben in Fettdruck dargestellt werden. Dabei werden Vektoren ohne weitere Kennzeichnung immer als Spaltenvektoren (stehende Vektoren) aufgefasst.

Liegt z. B. ein SISO-System vor, so wird der n -dimensionale Ausgangs-Zeilenvektor mit \mathbf{c}^T und der n -dimensionale Eingangs-Spaltenvektor mit \mathbf{b} bezeichnet. Die entsprechende Zustandsraumdarstellung lautet somit:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{b}u(t)$$

$$y(t) = \mathbf{c}^T\mathbf{x}(t) + du(t) .$$

Die einzelnen Elemente der Matrizen und Vektoren werden durch entsprechende Kleinbuchstaben (mit zugehörigen Indizes) in Dünndruck dargestellt.

6 Normalformen des Zustandsraummodells

6.1 Diagonalform als kanonische Normalform

Wird die Systemmatrix \mathbf{A} durch die sogenannte Ähnlichkeitstransformation diagonalisiert, so befinden sich alle n -Eigenwerte (auch konjugiert-komplexe) $(\lambda_1, \lambda_2, \dots, \lambda_n)$ des Systems auf der Hauptdiagonalen der so entstandenen Systemmatrix $\tilde{\mathbf{A}}$. Somit gilt [12]:

$$\tilde{\mathbf{A}} = \text{diag } \lambda_i = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}.$$

Die Ähnlichkeitstransformation wird gemäß

$$\tilde{\mathbf{A}} = \mathbf{T}^{-1} \mathbf{A} \mathbf{T} \quad (6.1)$$

durchgeführt, wobei sich die Transformationsmatrix \mathbf{T} aus n (normierten) Eigenvektoren \mathbf{v}_i wie folgt zusammensetzt:

$$\mathbf{T} = [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \dots \quad \mathbf{v}_n]. \quad (6.2)$$

Die Eigenvektoren \mathbf{v}_i erfüllen dabei folgende Bedingung [18]:

$$\mathbf{A} \mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \Leftrightarrow \quad (\mathbf{A} - \lambda_i \mathbf{I}) \cdot \mathbf{v}_i = 0.$$

Die Eingangsmatrix $\tilde{\mathbf{B}}$ und die Ausgangsmatrix $\tilde{\mathbf{C}}$ werden mit

$$\tilde{\mathbf{B}} = \mathbf{T}^{-1} \mathbf{B} \quad (6.3)$$

$$\tilde{\mathbf{C}} = \mathbf{C} \mathbf{T} \quad (6.4)$$

ermittelt.

Eine Matrix \mathbf{A} , die durch Ähnlichkeitstransformation (6.1) diagonalisiert werden kann, heißt diagonal-ähnlich [5]. Voraussetzung hierfür ist also die Existenz einer regulären Matrix \mathbf{T} .

6.2 Konjugiert komplexe Eigenwerte als reelle Parameter in Blöcken

Oft ist es erwünscht (um z. B. nur reelle Parameter und Signale auszuwerten), dass alle Elemente der Systemmatrix $\tilde{\mathbf{A}}$ reell sind. Die entsprechende Darstellungsform wird *Modalform* genannt, und kann mittels Funktion `canon` mit MATLAB ermittelt werden. Besitzt das System z. B. reelle Eigenwerte λ_1, λ_4 und zusätzlich konjugiert-komplexe Eigenwerte $\delta \pm j\omega$, so liegt $\tilde{\mathbf{A}}$ in der Form

$$\tilde{\mathbf{A}} = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \delta & \omega & 0 \\ 0 & -\omega & \delta & 0 \\ 0 & 0 & 0 & \lambda_4 \end{bmatrix}$$

vor. Sind alle Eigenwerte reell, so entspricht die Modalform der Diagonalform $\text{diag } \lambda_i$ einer Matrix.

7 Steuerbarkeit

In [2] wird vollständige Steuerbarkeit wie folgt definiert (Notation geändert):

Ein dynamisches System heißt vollständig steuerbar, wenn jeder Anfangszustand \mathbf{x}_0 durch Wahl eines geeigneten Verlaufs des Eingangsgrößenvektors $\mathbf{u}(t)$ innerhalb eines endlichen Zeitintervalls $[0, T_e]$ in jeden beliebigen Zustand \mathbf{x}_{soll} überführt werden kann.

7.1 Steuerbarkeitskriterium von KALMAN

Liegt die $(n \times n \cdot m)$ -Steuerbarkeitsmatrix \mathbf{S}_S

$$\mathbf{S}_S = [\mathbf{B} \quad \mathbf{AB} \quad \mathbf{A}^2\mathbf{B} \quad \dots \quad \mathbf{A}^{n-1}\mathbf{B}]$$

vor, so kann die vollständige Steuerbarkeit mittels des Steuerbarkeitskriteriums von KALMAN wie folgt festgestellt werden:

Steuerbarkeitskriterium von KALMAN (Zitat aus [13])

Das System (\mathbf{A}, \mathbf{B}) ist genau dann vollständig steuerbar, wenn die Steuerbarkeitsmatrix \mathbf{S}_S den Rang n hat:

$$\text{Rang } \mathbf{S}_S = n.$$

In MATLAB kann die Steuerbarkeitsmatrix \mathbf{S}_S mit der Funktion `ctrb` ermittelt werden.

7.2 Steuerbarkeitskriterium von GILBERT

Steuerbarkeitskriterium von GILBERT (Zitat aus [13])

Das System $(\text{diag } \lambda_i, \tilde{\mathbf{B}})$, dessen Zustandsraummodell in kanonischer Normalform vorliegt, ist genau dann vollständig steuerbar, wenn die Matrix $\tilde{\mathbf{B}}$ keine Nullzeile besitzt und wenn die p Zeilen¹ $\tilde{\mathbf{b}}_i^T$ der Matrix $\tilde{\mathbf{B}}$, die zu den kanonischen Zustandsgrößen eines p -fachen Eigenwertes gehören, linear unabhängig sind.

Voraussetzung für das Kriterium ist also, dass das System zuvor diagonalisiert wurde.

Wie aus Gln. (6.1) und (6.3) ersichtlich ist, setzt diese Transformation voraus, dass die Matrix \mathbf{T} regulär ist. Eine Matrix ist regulär, wenn ihre Zeilenvektoren und ihre Spaltenvektoren linear unabhängig sind [4]. Da sich \mathbf{T} gemäß Gl. (6.2) aus den Eigenvektoren \mathbf{v}_i von \mathbf{A} zusammensetzt, ist \mathbf{T} nur dann regulär, wenn alle Eigenvektoren \mathbf{v}_i linear unabhängig sind. Dies ist immer der Fall, wenn alle Eigenwerte λ_i von \mathbf{A} paarweise verschieden sind [9]. Treten jedoch mehrfache Eigenwerte auf, so ist dies i. Allg. nicht mehr der Fall.

Zur Veranschaulichung ist auf Abbildung 7.1 ein System (dritter Ordnung mit zwei Eingängen und einem Ausgang) in Diagonalform dargestellt. Offensichtlich ist das System nicht steuerbar, wenn die Eingangsmatrix

$$\tilde{\mathbf{B}} = \begin{bmatrix} \tilde{b}_{11} & \tilde{b}_{12} \\ \tilde{b}_{21} & \tilde{b}_{22} \\ \tilde{b}_{31} & \tilde{b}_{32} \end{bmatrix}$$

¹ Dies weicht von der üblichen Notation ab, nach der \mathbf{b}_i für die i -te Spalte der Matrix \mathbf{B} steht.

eine Nullzeile besitzt.

Besitzt die Eingangsmatrix keine Nullzeile und treten im System mehrfache Eigenwerte auf (z. B. $\lambda_1 = \lambda_2$), so muss es möglich sein, die zugehörigen Zustände \tilde{x}_1 und \tilde{x}_2 über die beiden Systemeingänge in unterschiedlicher Weise zu beeinflussen. Dies ist dann der Fall, wenn die zugehörigen Zeilen der Eingangsmatrix linear unabhängig sind, also die Matrix

$$\tilde{\mathbf{B}}_{12} = \begin{bmatrix} \tilde{b}_{11} & \tilde{b}_{12} \\ \tilde{b}_{21} & \tilde{b}_{22} \end{bmatrix}$$

regulär ist.

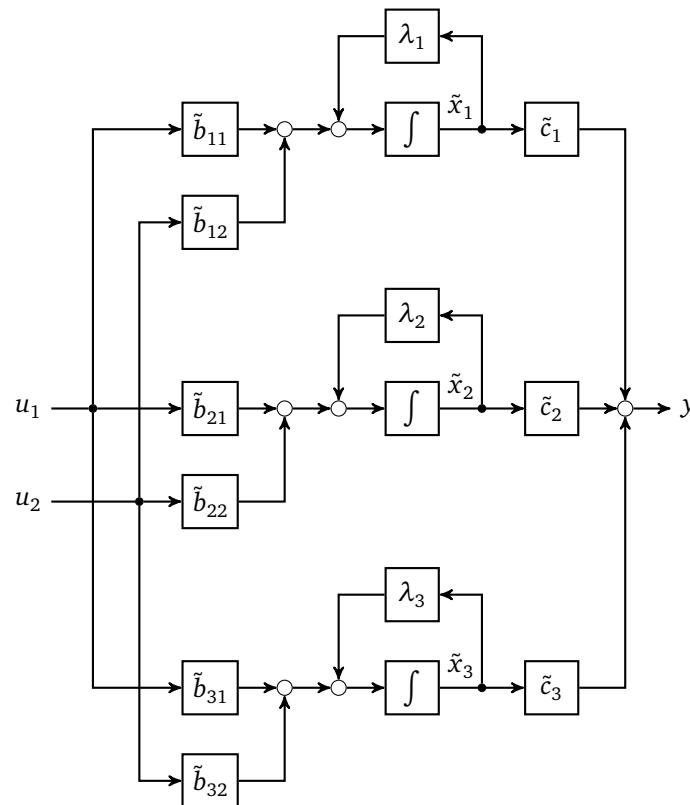


Abbildung 7.1: Blockschaltbild zur Verdeutlichung der Kriterien von GILBERT

7.3 Steuerbarkeitskriterium von HAUTUS

Steuerbarkeitskriterium von HAUTUS (Zitat aus [13])

Das System (\mathbf{A}, \mathbf{B}) ist genau dann vollständig steuerbar, wenn die Bedingung

$$\text{Rang} \begin{bmatrix} \lambda_i \mathbf{I} - \mathbf{A} & \mathbf{B} \end{bmatrix} = n$$

für alle Eigenwerte λ_i ($i = 1, 2, \dots, n$) der Matrix \mathbf{A} erfüllt ist.

8 Beobachtbarkeit

Die entsprechende Definition aus [2] lautet (beim Zitieren wurde die Notation an die im Abschnitt 5.1 festgelegte angepasst):

Ein dynamisches System heißt vollständig beobachtbar, wenn jeder Zustand $\mathbf{x}(t_0)$ exakt bestimmt werden kann aus dem Verlauf des Eingangsgrößenvektors $\mathbf{u}(t)$ und dem Verlauf des Ausgangsgrößenvektors $\mathbf{y}(t)$ innerhalb eines endlichen Zeitintervalls $[t_0, T_e]$.

8.1 Beobachtbarkeitskriterium von KALMAN

Das Beobachtbarkeitskriterium von KALMAN setzt die Kenntnis der $(r \cdot n \times n)$ -Beobachtbarkeitsmatrix \mathbf{S}_B

$$\mathbf{S}_B = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \mathbf{CA}^2 \\ \vdots \\ \mathbf{CA}^{n-1} \end{bmatrix}$$

voraus.

Beobachtbarkeitskriterium von KALMAN (Zitat aus [13])

Das System (\mathbf{A}, \mathbf{C}) ist genau dann vollständig beobachtbar, wenn die Beobachtbarkeitsmatrix \mathbf{S}_B den Rang n hat:

$$\text{Rang } \mathbf{S}_B = n.$$

In MATLAB kann die Beobachtbarkeitsmatrix \mathbf{S}_B mit der Funktion `obsv` ermittelt werden.

8.2 Beobachtbarkeitskriterium von GILBERT

Beobachtbarkeitskriterium von GILBERT (Zitat aus [13])

Das System $(\text{diag } \lambda_i, \tilde{\mathbf{C}})$, dessen Zustandsraummodell in kanonischer Normalform vorliegt, ist genau dann vollständig beobachtbar, wenn die Matrix $\tilde{\mathbf{C}}$ keine Nullspalte besitzt und wenn die p Spalten $\tilde{\mathbf{c}}_i$ der Matrix $\tilde{\mathbf{C}}$, die zu den kanonischen Zustandsvariablen eines p -fachen Eigenwertes gehören, linear unabhängig sind.

8.3 Beobachtbarkeitskriterium von HAUTUS

Beobachtbarkeitskriterium von HAUTUS (Zitat aus [13])

Das System (\mathbf{A}, \mathbf{C}) ist genau dann vollständig beobachtbar, wenn die Bedingung

$$\text{Rang} \begin{bmatrix} \lambda_i \mathbf{I} - \mathbf{A} \\ \mathbf{C} \end{bmatrix} = n$$

für alle Eigenwerte λ_i ($i = 1, 2, \dots, n$) der Matrix \mathbf{A} erfüllt ist.

9 MATLAB

Im folgenden werden einige Funktionen von MATLAB vorgestellt. Die zugrundeliegenden Informationen sind der MATLAB-Hilfe [15] entnommen. Auf diese wird auch für weitere Informationen zu den aufgeführten Befehlen verwiesen.

9.1 Matrizen

- **size(X)**

Die Funktion `size(X)` ermittelt die Größe einer Matrix `X`.

Mit `size(X, dim)` wird die Größe der Dimension `dim` der Matrix `X` bestimmt.

```
>> X = [1 2 3; 4 5 6];
>> sX = size( X )
sX =
     2     3
>> sX = size( X, 2 )
sX =
     3
```

- **length(X)**

Die Funktion `length(X)` ermittelt die Länge eines Vektors `X`.

```
>> X = [1; 2; 3];
>> lX = length( X )
lX =
     3
```

- **any(X)**

Die Funktion `any(X)` ermittelt, ob mindestens ein Element des Vektors `X` ungleich 0 ist und gibt dann `TRUE` zurück. Ist `X` eine Matrix, arbeitet es spaltenweise. Mit `any(X, dim)` kann vorgegeben werden, entlang welcher Dimension `any` arbeiten soll.

```
>> X = [1 0; 0 0; 3 2];
>> any( X, 2 )
ans =
     1
     0
     1
```

- **all(X)**

Die Funktion `all(X)` arbeitet wie `any(X)`, verlangt jedoch, dass *alle* Elemente ungleich 0 sind.

- **unique(X)**

Die Funktion `unique(X)` gibt alle Werte von `X` in sortierter Reihenfolge und ohne Wiederholungen zurück.

```
>> X = [1 2 1; 4 2 6];
>> unique( X )
ans =
     1
     2
     4
     6
```

- `rank(X)`
Die Funktion `rank(X)` gibt den Rang der Matrix `X` zurück.
- `ctrb(A, B)`
Die Funktion `ctrb(A, B)` berechnet die KALMAN'sche Steuerbarkeitsmatrix.
- `obsv(A, C)`
Die Funktion `obsv(A, C)` berechnet die KALMAN'sche Beobachtbarkeitsmatrix.
- `ss(A, B, C, D)`
Die Funktion `sys = ss(A, B, C, D)` erzeugt ein State-Space-Objekt für das gegebene System. Auf die Matrizen kann mit `sys.a`, `sys.b` usw. zugegriffen werden.
- `ssdata(sys)`
Die Funktion `[A, B, C, D] = ssdata(sys)` liest aus dem State-Space-Objekt `sys` die systembeschreibenden Matrizen aus und legt sie in `A, B` usw. ab.

Versuch 3

LQ-Regelung und Animation

In diesem Versuch soll, aufbauend auf der Modellbildung in Versuch 1 und 2, eine LQR-Regelung für das Schlitten-Pendel-System entworfen werden. Dabei wird in diesem Versuch davon ausgegangen, dass alle Zustände messbar sind. Der Beobachterentwurf ist Bestandteil des nächsten Versuchs.

Simulink ist sehr gut für die Simulation dynamischer Systeme und besonders Regelkreisen geeignet. Um Simulationen automatisiert auszuwerten ist es von Nutzen, aus MATLAB-Funktionen heraus Simulationen in Simulink zu starten und dabei Parameter austauschen zu können. Dies ist auch Teil dieses Versuchs. Insbesondere wird auf die verschiedenen „Workspaces“ (Gültigkeitsräume für Variablen) in MATLAB eingegangen.

Um Programme und den Base-Workspace übersichtlicher zu halten, können Struktur-Variablen verwendet werden, die hier vorgestellt und benutzt werden.

Um Modelle mechanischer Systeme auf Plausibilität zu prüfen und mit Ihnen zu arbeiten, kann es nützlich sein, sich das Modellverhalten in einer Animation zu betrachten. In diesem Versuch wird deutlich, wie ohne großen Aufwand solche Animationen in MATLAB erstellt werden können.

| | |
|--|-----------|
| 10 LQ-Reglerentwurf | 42 |
| 11 MATLAB | 43 |
| 11.1 LQ-Reglerentwurf | 43 |
| 11.2 Strukturen in MATLAB | 43 |
| 11.3 Workspaces und deren zugeordneten Variablen | 45 |
| 11.4 Starten einer Simulation | 46 |
| 11.5 Grafikobjekte | 46 |
| 11.6 Animation von Simulationsergebnissen | 49 |
| 11.7 Interpolation mit <code>interp1</code> | 51 |

10 LQ-Reglerentwurf

In diesem Versuch soll der Entwurf des Reglers nicht durch direkte Vorgabe der Pole, sondern mittels eines quadratischen Gütemaßes erfolgen. Es gilt die lineare, zeitinvariante Regelstrecke

$$\begin{aligned}\dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \quad \text{mit } \mathbf{x}(0) = \mathbf{x}_0 \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t)\end{aligned}$$

durch eine Regelung so in die Endlage $\mathbf{x}(t \rightarrow \infty) = \mathbf{0}$ zu führen, dass das Gütemaß

$$J = \int_{t=0}^{\infty} (\mathbf{x}^T(t)\mathbf{Q}\mathbf{x}(t) + \mathbf{u}^T(t)\mathbf{R}\mathbf{u}(t)) dt$$

minimal wird. \mathbf{Q} und \mathbf{R} sind dabei positiv definite, symmetrische Matrizen. (\mathbf{Q} könnte auch positiv semi-definit sein, jedoch müsste dann eine schwieriger zu prüfende Voraussetzung nachgewiesen werden.)

Das Regelgesetz ist von der Form

$$\mathbf{u}(t) = -\mathbf{K}\mathbf{x}(t)$$

mit der Reglermatrix \mathbf{K} , die sich aus der Lösung des Optimierungsproblems zu

$$\mathbf{K} = \mathbf{K}_{\text{opt}} = \mathbf{R}^{-1}\mathbf{B}^T\mathbf{P}$$

ergibt. \mathbf{P} wird dabei über die algebraische Matrix-RICCATIgleichung

$$\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} - \mathbf{P}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\mathbf{P} = -\mathbf{Q}$$

berechnet. Voraussetzung für die Existenz der Lösung ist, dass die betrachtete Strecke vollständig steuerbar ist.

Für eine Herleitung wird auf [9] oder [2] verwiesen.

Dieses Reglerentwurfsverfahren ist, wie zuvor erwähnt, eine Optimierungsaufgabe. Somit werden nicht die Pole des geschlossenen Regelkreises vorgegeben, sondern es wird über die Gewichtsmatrizen \mathbf{Q} und \mathbf{R} festgelegt, wie stark die einzelnen Zustände \mathbf{x} bzw. Eingänge \mathbf{u} im Gütefunktional bestraft werden. Am einfachsten werden die Matrizen dabei als Diagonalmatrizen festgelegt. Mit der Matrix

$$\mathbf{Q} = \begin{bmatrix} q_1 & 0 & \dots & 0 \\ 0 & q_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & q_n \end{bmatrix} \quad q_i > 0, \forall i = 1, \dots, n$$

kann dann bspw. über die Koeffizienten q_i die Gewichtung der Bestrafung für die einzelnen Zustandsgrößen x_i festgelegt werden.

11 MATLAB

Im folgenden sind einige Konzepte und Funktionen von MATLAB vorgestellt. Die zugrundeliegenden Informationen sind der MATLAB-Hilfe [15] entnommen. Auf diese wird auch für weitere Informationen zu den aufgeführten Befehlen verwiesen.

11.1 LQ-Reglerentwurf

In MATLAB steht für den LQ-Reglerentwurf die Funktion `lqr` zur Verfügung. Für die Verwendung bitte die MATLAB-Hilfe beachten.

Zum Erstellen der Gewichtungsmatrizen ist der Befehl `diag` von Nutzen. Mit diesem kann eine Diagonalmatrix aus einem Vektor erzeugt werden.

11.2 Strukturen in MATLAB

Das Verwalten von umfangreichen Datenmengen wird in MATLAB, bei Verwendung der bis jetzt bekannten Variablen, sehr schnell unübersichtlich. Um diesem Problem entgegenzuwirken, können Datenstrukturen verwendet werden. In einer solchen Struktur werden Variablen in einer einzigen Struktur-Variable zusammengefügt. Dabei können die Variablen auch von verschiedenen Typen sein, und auch Strukturen in Strukturen sind möglich. Die Variablen innerhalb der Struktur werden auch als Felder bezeichnet. So kann man beispielsweise eine Struktur `Student` erstellen, in der Informationen wie Name, Matrikelnummer, Fachsemester, o. Ä. enthalten sind.

Strukturen können, wie man es von gewöhnlichen Variablen kennt, an Funktionen übergeben werden oder deren Rückgabewerte bilden.

Mit nachfolgender Syntax ist es möglich, einem Feld in einer Struktur eine Zuweisung zuzuordnen:

```
strukturname.feldname = zuweisung
```

Im folgendem Beispiel wird eine Struktur mit einem String und zwei Zahlenwerten angelegt:

```
>> Student.Name = 'Claudia M.';
>> Student.Matrikelnummer = 1234567;
>> Student.Fachsemester = 9;
```

Lässt man sich den Inhalt dieser Struktur anzeigen, so sieht das so aus:

```
>> Student
Student =
      Name: 'Claudia M.'
Matrikelnummer: 1234567
  Fachsemester: 9
```

Eine weitere Möglichkeit, eine Struktur zu erzeugen, ist der Befehl `struct`:

```
>> clear all;
>> Student = struct('Name', 'Claudia M.', ...
    ... 'Matrikelnummer', 1234567, 'Fachsemester', 9)
Student =
    Name: 'Claudia M.'
Matrikelnummer: 1234567
Fachsemester: 9
```

Es können jederzeit die Daten in der Struktur geändert

```
>> Student.Matrikelnummer = 7654321;
```

weitere Felder ergänzt

```
>> Student.Hochschulsemester = 11;
```

oder Felder gelöscht

```
>> Student = rmfield(Student, 'Fachsemester');
```

werden.

Nach erneutem Aufruf von Student wird nun

```
>> Student
Student =
    Name: 'Claudia M.'
Matrikelnummer: 7654321
Hochschulsemester: 11
```

ausgegeben.

Es ist auch möglich, Vektoren von Strukturen anzulegen, wie folgendes Beispiel zeigt:

```
>> Student(2).Name = 'Claudio M.'
Student =
1x2 struct array with fields:
    Name
Matrikelnummer
Hochschulsemester

>> Student(2)
ans =
    Name: 'Claudio M.'
Matrikelnummer: []
Hochschulsemester: []
```

Weitere nützliche Befehle im Umgang mit Strukturen sind `setfield` und `getfield` zum Setzen und Auslesen von Feldern. In diesem Zusammenhang sind auch die „Dynamic Field Names“ interessant. Mit `isstruct` kann überprüft werden, ob eine Variable eine Struktur ist und mit `isfield` ob eine Struktur bestimmte Felder enthält. Mit `orderfields` können die Felder, die standardmäßig in der Reihenfolge des Hinzufügens sortiert sind, neu geordnet werden.

11.3 Workspaces und deren zugeordneten Variablen

Beim bisherigen Umgang mit MATLAB wurde noch nicht auf den Aspekt der verschiedenen Workspaces eingegangen. So besitzt MATLAB einen globalen Workspace, den Base-Workspace. In diesem werden Variablen gespeichert, die im „Command Window“ oder in Skripten deklariert (erstellt) wurden. Diese Variablen sind von allen Skripten benutzbar. Im Gegensatz hierzu wird jeder Funktion ihr eigener Workspace zugeordnet, und daher sind Variablen die einer bestimmten Funktion definiert wurden für alle anderen Funktionen und Skripte oder für das „Command Window“ nicht zugänglich. Ebenso kann eine Funktion nicht auf den Base-Workspace zugreifen. Ein Workspace kann also als Gültigkeitsraum für Variablen angesehen werden.

MATLAB stellt aber zwei Befehle zur Verfügung, mit denen diese Einschränkung teilweise aufgehoben werden kann. Der eine Befehl lautet `assignin` und besitzt die Syntax

```
assignin(WS, 'Var', Val)
```

Dabei muss `WS` entweder `'base'` oder `'caller'` sein. Diese Funktion weist der Variable mit dem Namen `Var` im Base-Workspace (`WS = 'base'`) oder im Workspace der aufrufenden Funktion (`WS = 'caller'`) den Wert `Val` zu.

Das „Gegenstück“ zu dieser Funktion ist die Funktion `evalin` mit der Syntax

```
ret = evalin(WS, 'Expr')
```

Diese führt den Ausdruck `Expr` im angegebenen Workspace aus. (Wieder ist nur `'base'` oder `'caller'` möglich.) Wenn dieser Ausdruck einen Wert zurückgibt, dann wird dieser in die Variable `ret` geschrieben. In der einfachsten Form ist der Ausdruck nur ein Variablenname, damit kann bspw. aus einer beliebigen Funktion der Wert von Variablen im Base-Workspace ausgelesen werden.

Diese Befehle werden in der folgenden Beispielfunktion `aeTest` verwendet.

Listing 11.1: Beispielfunktion `aeTest`

```
function aeTest()  
  
    evalin( 'base', 'clear' );  
  
5    assignin( 'base', 'a', 5 );  
  
    a = 10;  
  
    a_base = evalin( 'base', 'a;' );  
  
10    disp( 'a_in_Funktion:' );  
    disp( num2str(a) );  
  
    disp( 'a_in_Base-Workspace:' );  
15    disp( num2str( a_base ) );  
  
end
```

Diese löscht zunächst alle Variablen im Base-Workspace, legt dann die Variable `a` einmal im Base-Workspace und einmal im eigenen Function-Workspace ab und erzeugt den Ausruck

```
>> aeTest
a in Funktion:
10
a in Base-Workspace:
5
```

Diese Funktionen sind u. a. dann interessant, wenn es darum geht aus Funktionen heraus Simulink-Modelle zu starten, die in Blöcken Variablen aus dem Base-Workspace benutzen.

11.4 Starten einer Simulation

Eine Simulation wird von MATLAB aus mit `sim` gestartet (siehe auch Praktikum MATLAB/Simulink I).

Simulink liest Parameter standardmäßig aus dem Base-Workspace aus. Allerdings gibt es die Variablen über den Block „To Workspace“ bzw. aus den Scopes an den Workspace der Funktion zurück, die die Simulation gestartet hat.

Soll eine Simulation mit `sim` aus einer Funktion gestartet werden, und liegen die benötigten Simulationsparameter im Workspace der Funktion, so besteht eine Möglichkeit darin, die Daten mit den in Abschnitt 11.3 beschriebenen Methoden in den Base-Workspace zu schreiben.

Eine elegantere Alternative besteht darin, das Verhalten von Simulink anzupassen. Mit Hilfe der Funktion `simset` kann eine Struktur von Parametern erstellt werden, mit der eine Simulation mit Simulink konfiguriert werden kann. Ohne Angabe von Parametern erzeugt `simset` eine Struktur mit Standardwerten. Man erkennt, dass standardmäßig die benötigten Daten dem Base-Workspace entnommen werden.

```
>> simset
[...]
SrcWorkspace: [ {'base'} | 'current' | 'parent' ]
[...]
```

Der Parameter `SrcWorkspace` kann jedoch auch zwei andere Werte annehmen. Durch Angabe von `current` werden die Daten dem aktuell aktiven Workspace, aus dem die Simulation gestartet wird, entnommen. Ein Beispiel ist nachfolgend aufgeführt.

```
>> stOptions = simset( 'SrcWorkspace', 'current' );
>> sim( 'model', Tend, stOptions );
```

11.5 Grafikobjekte

Im folgenden Abschnitt soll die Verwaltung graphischer Objekte durch MATLAB näher gebracht werden. Wie schon aus dem Praktikum MATLAB/Simulink I bekannt, weist MATLAB jedem Figure ein sogenanntes Handle – eine Zahl, die eine Art Adresse darstellt – zu. Dies gilt auch für jedes andere erzeugte Grafikobjekt. Mit Hilfe dieser Handles können die verschiedenen Objekte angesprochen und ihre Eigenschaften bestimmt, geändert oder ausgelesen werden (`set`, `get`). Außerdem können Objekte gezielt gelöscht werden (`delete`).

Die wesentlichen Objekte für diesen Versuch sind Grafik-Fenster (*figures*, das sind wirklich *nur* die Fenster), Achsensysteme (*axes*) und Linienzüge/Kurven (*lines*, „entartet“ auch einfache Punkte).

```
>> handleF1 = figure()
```



```
handleF1 =  
    1  
>> handleF2 = figure()  
handleF2 =  
    2
```

Es wird hier jeweils ein leeres Figure geöffnet und dessen Handle in der entsprechenden Variablen gespeichert. Für MATLAB gilt das zuletzt geöffnete Figure als aktiv. In diesem (Figure 2) wird im Folgenden, mit der Funktion `axes`, ein Achsensystem erzeugt werden.

```
>> handleAxes2 = axes()  
handleAxes2 =  
    318.0018
```

Damit auch in Figure 1 ein Achsensystem erstellt werden kann, muss dieses in den Zustand des *momentan aktiven* Figures versetzt werden. Dies geschieht mittels `figure(handleFigure)`.

```
>> figure(handleF1);  
>> handleAxes1 = axes()  
handleAxes1 =  
    318.0018
```

Entsprechend wird mit `axes(handleAxes)` das durch `handleAxes` bezeichnete Achsensystem aktiviert.

In diese Achsensysteme können dann verschiedene Funktionen gezeichnet werden. Dem schon bekannten Befehl `plot` kann als erstes Argument ein Handle auf ein Achsensystem übergeben werden. Dann werden die übergebenen Daten in dieses Achsensystem gezeichnet. Wird kein Handle angegeben, wird automatisch das aktuelle Achsensystem verwendet. Existiert noch kein Achsensystem oder Figure, dann wird eines erzeugt. (Das bekannte Verhalten aus dem Praktikum MATLAB/Simulink I.)

```
>> x = linspace(-pi,pi);  
>> hPlot2 = plot(handleAxes2, x, cos(x));  
>> hPlot1 = plot(handleAxes1, x, sin(x));
```

Der Rückgabewert des `plot`-Befehls ist ein Handle auf den gezeichneten Linienzug. Wenn mehrere Kurven auf einmal gezeichnet wurden, wird entsprechend ein Vektor mit Handles zurückgegeben.

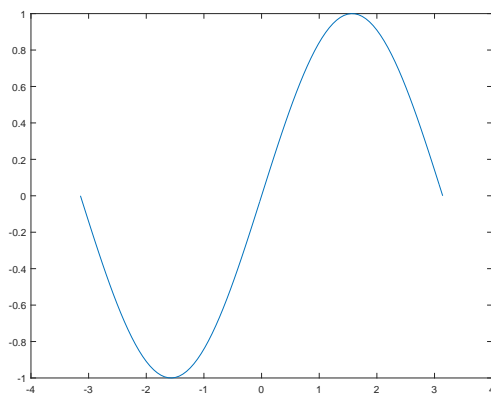
Die Figures des Beispiels sehen nun so aus wie in Abbildung 11.1.

Eine weitere Möglichkeit eine Kurve zu zeichnen, ist der Befehl `line`, der allerdings verglichen mit `plot` ein relativ „niedriger“ Befehl ist. Im Gegensatz zum `plot`-Befehl berücksichtigt der `line`-Befehl nicht die automatische Farbfolge und für die Zuweisung von Farbe und Linienart kann nicht die Abkürzung über die „LineSpec“-Strings (z. B. `'g--*'` für grün, gestrichelt und sternförmige Marker) genommen werden. Dafür werden mit der `line`-Funktion nicht alle vorher gezeichneten Linien gelöscht, was beim `plot`-Befehl passiert. Um dieses Verwerfen der alten Daten beim `plot`-Befehl zu verhindern, muss vor dem zweiten `plot`-Aufruf der `hold`-Befehl (`hold('all')` oder `hold('on')`) verwendet werden.

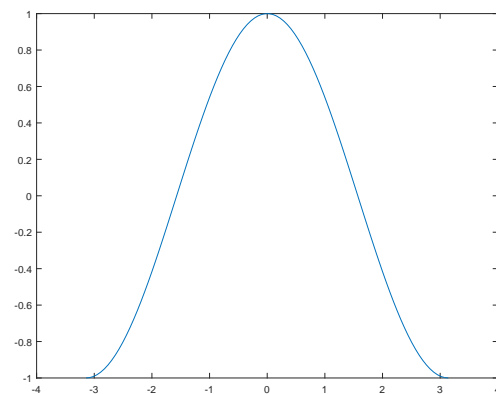
Ändern von Objekt-Eigenschaften:

`set(ObjectHandle, Eigenschaft, Wert)`

Mit der Funktion `set` lassen sich verschiedene Eigenschaften der Objekte (Graphen/Plots, Achsensysteme, Fenster/Figures, Textfelder, etc.) ändern. Im obigen Beispiel wird bspw. mittels



(a) Figure 1



(b) Figure 2

Abbildung 11.1: Figures zum Beispiel (siehe Text)

```
>> set(hPlot1, 'LineStyle', '--');
```

die Linieneigenschaft der Kurve mit dem Handle `hPlot1` (diejenige in Figure 1) geändert. Der Kurvenverlauf wird nun strichliert dargestellt. Die Syntax bleibt dabei für jedes beliebige Objekt identisch. Für eine Liste der einstellbaren Optionen sei auf die Dokumentation der einzelnen Objekte verwiesen.

Auslesen von Objekt-Eigenschaften:

`get(ObjectHandle, Eigenschaft)`

Über `get` lassen sich die aktuellen Werte der einzelnen Eigenschaften eines Objekts auslesen.

```
>> marker = get(hPlot1, 'Marker')
marker =
none
```

Hier wird die Eigenschaft *Marker* der Kurve `hPlot1` ausgelesen. Es wird die Art des verwendeten Markertyps zurückgegeben.

Mit `get(handle)` werden alle Eigenschaften des durch `handle` bezeichneten Objekts zurückgegeben.

Löschen von Objekten:

`delete(ObjectHandle)`

`delete` löscht das zum gegebenen Handle gehörende Objekt. (Und alle von diesem Objekt abhängigen Objekte. D. h., wenn ein Achsensystem gelöscht wird, dann sind alle Kurven, die in diesem Achsensystem gezeichnet wurden, auch gelöscht.)

```
>> delete(hPlot1)
```

Es wird der Kurvenverlauf des $\sin(x)$ in Figure 1 gelöscht. Das Achsensystem verbleibt jedoch in Figure 1.

11.6 Animation von Simulationsergebnissen

Oftmals ist es sinnvoll, die Ergebnisse einer Simulation zu animieren, um sich einen anschaulichen Überblick über das Verhalten des (mechanischen) Systems zu verschaffen. Eine Animation entsteht dadurch, dass die verschiedenen (geometrischen) Systemzustände in schneller Folge nacheinander gezeigt werden.

Das Vorgehen bei einer solchen Animation wird an einem kleinen Beispiel („springender Punkt“) gezeigt. Der Code ist in den beiden folgenden Listings 11.2 und 11.3 gezeigt und wird darunter kommentiert.

Listing 11.2: Funktion SpringenderPunkt

```
function SpringenderPunkt()
    [y x] = werte();
3    animierePunkt(y, x);
end % function SpringenderPunkt

% subfunction werte
function [ outY outX ] = werte()
8    outX = linspace(0,pi);
    outY = abs(sinc(outX));
end % subfunction werte
```

Listing 11.3: Funktion animierePunkt

```
function [] = animierePunkt(inY, inX)

    % Anzahl Einzelbilder
    nBilder = length(inY);
5

    % ein neues Figure mit Achsensystem erzeugen
    figure();
    axes();

10    % mit axis werden die Achsen auf den maximal gebrauchten Bereich →
        ←skaliert
    axis([0 max(inX) 0 1]);

    % das erzeugte Achsensystem mit eingestellter Skalierung festhalten
    hold on;
15

    % -----
    % Eigentlicher Beginn der Animation
    % (Schleife über alle Zustände)
    % -----
20    for i=1:nBilder
        if i>1
            delete(hPunkt);
        end
        hPunkt = plot(inX(i),inY(i),'Marker','o','MarkerSize',10);
25        pause(0.025); % Pause in Sekunden
```

```

end
hold off;
% -----
% Ende der Simulation
% -----

```

30

```

end % function animierePunkt

```

Die Funktion `SpringenderPunkt` erzeugt lediglich Wertepaare, die der Punkt später „entlangspringen“ soll und ruft damit die Funktion `animierePunkt` auf, in der die eigentliche Animation implementiert ist.

In `animierePunkt` wird zunächst mit `figure()` und `axis()` ein neues Figure mit Achsensystem erzeugt und an die übergebenen Datenpunkte angepasst. Durch den Befehl `hold on` wird das erzeugte Achsensystem und alle weiter geplotteten Zustände im Achsensystem festgehalten.

Die Schleife beschreibt die eigentliche Animation. Hier wird beim ersten Schleifendurchlauf ($i=1$) der erste Zustand (`inX(1),inY(1)`) in das erzeugte Achsensystem geplottet. Im zweiten und in den darauffolgenden Schleifendurchläufen wird zunächst der zuletzt gezeichnete Zustand mit `delete` entfernt und mit `plot` der aktuelle Zustand gezeichnet. Mit dem letzten Befehl in der Schleife `pause` wird die Animation verlangsamt, damit für uns ein angenehm flüssiges Bild entsteht. Bei Nichtverwendung des Befehls würde die Animation so schnell ablaufen, dass es einem Betrachter nicht möglich wäre, der Animation visuell zu folgen.

Eine andere Möglichkeit, die Zustände neu zu zeichnen, wäre, direkt die Daten der betreffenden Linienobjekte über `set` zu ändern:

```

set(hLines, 'XData', neueXWerte, 'YData', neueYWerte)

```

Würde der Plot einfach für jedes Einzelbild mit dem bekannten `plot`-Befehl komplett neu erstellt werden, dann würde, insbesondere bei Animationen mit mehreren Objekten, ein Flackern zu erkennen sein.

Speichern von Animationen

Animationen wie die im obigen Beispiel erzeugte, können auch gespeichert und in das `avi`-Format konvertiert werden. Mit

```

frame = getframe(handle)

```

wird der Inhalt des mit `handle` angegebenen Figures oder Achsensystems als Rastergrafik in einer Struktur in `frame` gespeichert. In diesem Zusammenhang sind die Befehle `gcf` und `gca` interessant, die das Handle des aktiven Figures bzw. Achsensystem zurückgeben (`get current figure/axes`).

Mit diesem Befehl kann dann jedes Einzelbild der Animation in einem Vektor (von Strukturen) abgelegt werden:

```

[...]
vFrames(end+1) = getframe(hAnimationAxes);
[...]

```

Diese Struktur kann dann von MATLAB als Film interpretiert und mit `movie` wieder abgespielt werden (siehe MATLAB-Hilfe).

Ein paar Hinweise zu diesen Funktionen:

- **Wichtig:** Da die Einzelbilder als unkomprimierte Rastergrafiken gespeichert werden, wird so ein „Film“ sehr schnell sehr groß. (Ein einziges Einzelbild kommt bei der Standardgröße eines Figures auf etwa 700 kB). Dies kann dazu führen, dass MATLAB stark „ausgebremst“ wird.
- Mit `getframes` wird im Grunde ein Screenshot gemacht. D. h., wenn ein Fenster einer anderen Anwendung das „Filmfenster“ überdeckt, wird dieses aufgenommen. Wenn der Bildschirmschoner angeht, wird ebenfalls dieser aufgenommen.
- Zum Teil wird der Film innerhalb eines neuen Achsensystems abgespielt. Dies kann mit `movie(figure(), ...)` verhindert werden.

Um eine „Film“-Struktur in ein avi-Format zu konvertieren, gibt es den Befehl `movie2avi`. Für weitere Informationen dazu wird auf die MATLAB-Hilfe verwiesen. Dort sind auch die möglichen Codecs angegeben, die ausprobiert werden können, um das bestmögliche Ergebnis zu erhalten.

11.7 Interpolation mit `interp1`

Sind (Funktions-)Werte an diskreten Punkten gegeben, dann können mit der Funktion `interp1` Werte zwischen den gegebenen Punkten interpoliert werden:

$$y_i = \text{interp1}(x, Y, x_i)$$

Hierbei ist Y ein Vektor oder eine Matrix mit genauso vielen *Zeilen* wie der Vektor x lang ist. Dadurch ist ein Zusammenhang $y_l(x_l)$ an diskreten Punkten gegeben. y_i ist der (interpolierte) Wert an der Stelle x_i .

(Wenn der zu interpolierende Wert von mehr als einer Variablen abhängt, also wenn ein Zusammenhang der Form $y_l(x_{l,1}, x_{l,2})$, $y_l(x_{l,1}, x_{l,2}, x_{l,3})$ oder allgemein $y_l(\mathbf{x}_l)$ vorliegt, dann können die Funktionen `interp2`, `interp3` bzw. `interpN` verwendet werden.)

Beispiel:

```
>> x = [0 1 2 3 4].';
>> Y = [0 0; 1 2; 2 4; 6 6; 5 7];
>> yi = interp1(x, Y, 3.25)
yi =
    5.7500    6.2500
```

Standardmäßig erfolgt die Interpolation linear (wie im vorherigen Beispiel), es können aber auch als viertes Argument andere Methoden verwendet werden, z. B. 'nearest' oder 'cubic'.

```
>> yi = interp1(x, Y, 3.25, 'nearest')
yi =
     6     6

>> yi = interp1(x, Y, 3.25, 'cubic')
yi =
    5.9844    6.3203
```

Zudem kann im fünften Argument noch angegeben werden, ob extrapoliert werden soll, wenn x_i außerhalb des Wertebereiches von x liegt. Standardmäßig gibt `interp1` ansonsten NaN zurück.

```
>> yi = interp1(x, Y, 5, 'linear')
```

```
yi =  
    NaN    NaN  
  
>> yi = interp1(x, Y, 5, 'linear', 'extrap')  
yi =  
     4     8
```

Versuch 4

Beobachterentwurf – Benutzeroberflächen

Im letzten Versuch wurde ein Zustandsregler für das Schlitten-Pendel-System entworfen. Da in einer praktischen Umsetzung nicht alle Zustandsgrößen gemessen werden könnten, wird in diesem Versuch ein Beobachter entworfen.

Außerdem soll in diesem Versuch ein Einblick in das Erstellen grafischer Benutzeroberflächen (GUIs) vermittelt werden. Benutzeroberflächen können dazu dienen, selbst geschriebene Programme für andere Personen einfach nutzbar zu machen, ohne dass diese sich mit der genauen Syntax der Funktionen auseinandersetzen bzw. überhaupt MATLAB-Kenntnisse besitzen müssen. Die Entwicklungsumgebung GUIDE von MATLAB ermöglicht es benutzerfreundliche Oberflächen auf sehr einfache Weise zu erstellen, ohne dass dazu besonders fortgeschrittene Programmierkenntnisse nötig sind.

| | |
|---|-----------|
| 12 Beobachterentwurf | 55 |
| 12.1 Vollständiger Luenbergerbeobachter | 55 |
| 13 Benutzeroberflächen (GUIs) | 57 |
| 13.1 Einführung | 57 |
| 13.2 Gestaltung der GUI mittels GUIDE | 57 |
| 13.3 GUI-Funktionalität | 61 |
| 14 Erstellen einer grafischen Benutzeroberfläche | 63 |
| 14.1 Eingabefelder für Q- und R-Matrix | 64 |
| 14.2 Auswahl des Arbeitspunktes | 66 |
| 14.3 Berechnung des Reglers K | 67 |
| 15 Weitere MATLAB-Funktionen | 70 |
| 15.1 Polvorgabe | 70 |

| | |
|---|----|
| 15.2 Cell-Arrays | 70 |
| 15.3 Persistent und globale Variablen | 71 |

12 Beobachterentwurf

Im letzten Versuch wurde eine Zustandsregelung für das Schlitten-Pendel-System entworfen. Dabei wurde davon ausgegangen, dass alle Zustände messbar seien. Bei einer praktischen Umsetzung wäre dies aber nicht der Fall, da (mit vertretbarem Aufwand) zwar die Position des Schlittens und der Winkel des Pendels, aber nicht die dazugehörigen Geschwindigkeiten gemessen werden können.

Eine Möglichkeit, die fehlenden Zustandsgrößen zu bestimmen, wäre es, die Position und den Winkel numerisch abzuleiten. Dabei kann es aber wegen immer auftretendem Messrauschen zu Problemen kommen.

In diesem Versuch wird eine zweite Möglichkeit angewendet, und zwar wird für das System ein Zustandsbeobachter entworfen.

Im Folgenden wird kurz auf die Grundlagen eines solchen Beobachters eingegangen, für weitere Informationen wird auf [9] oder [13] verwiesen. Die hier gewählte Darstellung ist [13] entnommen.

12.1 Vollständiger Luenbergerbeobachter

Der Luenbergerbeobachter für das durch die Zustandsraumdarstellung

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \quad \mathbf{x}(0) = \mathbf{x}_0 \quad (12.1)$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) \quad (12.2)$$

gegebene System hat die in Abbildung 12.1 gezeigte Struktur. Der obere Teil stellt dabei die Strecke selbst dar, der eigentliche Beobachter ist der umrandete untere Teil. Dieser besteht zum einen aus dem Modell der Strecke, zum anderen aus der Rückführung der Differenz zwischen tatsächlichem Ausgangswert \mathbf{y} und dem geschätzten Ausgangswert $\hat{\mathbf{y}}$ über die Beobachterrückführmatrix \mathbf{L} .

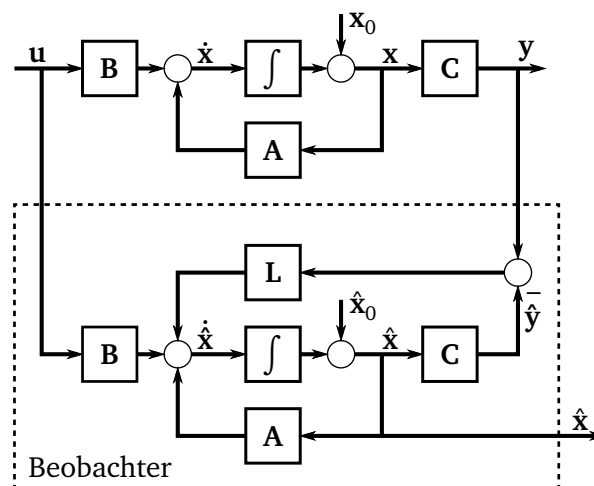


Abbildung 12.1: Luenbergerbeobachter [13]

Aus dem Blockschaltbild lässt sich die Gleichung

$$\frac{d\hat{\mathbf{x}}(t)}{dt} = (\mathbf{A} - \mathbf{L}\mathbf{C}) \cdot \hat{\mathbf{x}}(t) + \mathbf{B}\mathbf{u}(t) + \mathbf{L}\mathbf{y}(t) \quad (12.3)$$

für die Ableitung des geschätzten Zustandes $\hat{\mathbf{x}}$ direkt ablesen. Dieser Schätzwert sollte natürlich möglichst genau dem tatsächlichen Zustand \mathbf{x} entsprechen. Um eine Aussage über die Abweichung von Schätz- zu Istwert treffen zu können, wird der Beobachtungsfehler

$$\mathbf{e}(t) = \mathbf{x}(t) - \hat{\mathbf{x}}(t)$$

betrachtet. Leitet man diesen ab ($\frac{d\mathbf{e}(t)}{dt} = \frac{d\mathbf{x}(t)}{dt} - \frac{d\hat{\mathbf{x}}(t)}{dt}$) und ersetzt in dieser Ableitung $\frac{d\mathbf{x}(t)}{dt}$ und $\frac{d\hat{\mathbf{x}}(t)}{dt}$ durch (12.1) bzw. (12.3), so erhält man

$$\dot{\mathbf{e}}(t) = (\mathbf{A} - \mathbf{LC}) \cdot \mathbf{e}(t), \quad \mathbf{e}(0) = \mathbf{x}_0 - \hat{\mathbf{x}}_0. \quad (12.4)$$

Der Beobachtungsfehler geht also für $t \rightarrow \infty$ gegen 0, wenn die Eigenwerte der Matrix $(\mathbf{A} - \mathbf{LC})$ einen negativen Realteil haben.

Der Beobachtungsfehler sollte aber nicht nur irgendwann abklingen, sondern so schnell, dass dieser die Regelung der Strecke möglichst nicht beeinflusst. D. h., dass die Beobachtereigenwerte links von den (dominanten) Eigenwerten des geschlossenen Regelkreises liegen sollten. Als Anhaltswert kann gelten, dass die Beträge der Realteile der Beobachtereigenwerte zwei- bis sechsmal größer als die des geschlossenen Regelkreises sein sollten. (Bei betragsmäßig zu großen Beobachtereigenwerten hat das Messrauschen einen starken negativen Einfluss, daher können die Beobachtereigenwerte nicht beliebig weit nach links gelegt werden.)

Die Eigenwerte einer Matrix ändern sich nicht, wenn die Matrix transponiert wird, d. h., dass die Eigenwerte von $(\mathbf{A} - \mathbf{LC})$ gleich denen von $(\mathbf{A}^T - \mathbf{C}^T \mathbf{L}^T)$ sind.

Damit entsprechen aber die Eigenwerte von (12.4) genau den Regelungseigenwerten des zu (12.1) und (12.2) dualen Systems

$$\begin{aligned} \dot{\mathbf{x}}_T(t) &= \mathbf{A}^T \mathbf{x}_T(t) + \mathbf{C}^T \mathbf{u}_T(t), & \mathbf{x}_T(0) &= \mathbf{x}_{T0} \\ \mathbf{y}_T(t) &= \mathbf{B}^T \mathbf{x}_T(t) \end{aligned}$$

mit der Rückführung

$$\mathbf{u}_T(t) = -\mathbf{L}^T \mathbf{x}_T(t).$$

Diese Interpretation hat hier den Nutzen, dass damit schon in MATLAB implementierte Verfahren zur Synthese von Zustandsregelungen für den Beobachterentwurf benutzt werden können. Dabei werden die für den Reglerentwurf relevanten Matrizen \mathbf{A} , \mathbf{B} und \mathbf{R} durch \mathbf{A}^T , \mathbf{C}^T bzw. \mathbf{L}^T ersetzt.

Der Polvorgabeentwurf kann dann mit der Formel von Ackermann (für Systeme mit einem Eingang) oder durch die Vollständige Modale Synthese durchgeführt werden. Siehe dazu [9] und [13].

Reduzierter Beobachter

Wenn, wie meist, einige Zustandsgrößen gemessen werden können (oder zumindestens mit einer algebraischen Gleichung aus den Messwerten bestimmt werden können), müssten nicht alle Zustandsgrößen mit einem Beobachter geschätzt werden. Dies wird beim Entwurf eines *reduzierten Beobachters* berücksichtigt, auf den in diesem Versuch aber nicht weiter eingegangen wird.

13 Benutzeroberflächen (GUIs)

13.1 Einführung

Grafische Benutzeroberflächen (GUI - Graphical User Interface) bestehen aus einem Fenster (*figure*) und mehreren weiteren Grafikobjekten, die auf diesem Figure angeordnet sind. (Im letzten Versuch wurden schon die Objekte *figure*, *axes* und *lines* vorgestellt.) Auch wenn die äußere Erscheinung von bspw. einem Textfeld und einem Linienzug in einem Achsensystem nicht viel gemeinsam hat, so werden diese doch intern von MATLAB gleich behandelt. Es handelt sich in beiden Fällen lediglich um ein Objekt, das bestimmte Eigenschaften hat. Die Eigenschaften können jeweils mit `set` gesetzt und `get` ausgelesen werden.

Welche Eigenschaften ein Objekt besitzt, hängt jedoch von der Art des Objektes ab. So besitzt ein Linienzug u. a. eine Linienart, ein Textfeld dagegen eine Position, eine Größe und eine Zeichenkette als Eigenschaft.

Im Grunde kann mit `set` eine komplette GUI aufgebaut werden, dies wäre aber sehr mühsam. So müssten vorher per Hand die genauen Positionen und Abmessungen der Objekte der GUI festgelegt und dann einzeln mit `set` an die entsprechenden Objekte übertragen werden. MATLAB verfügt aber mit GUIDE über eine Entwicklungsumgebung für Oberflächen, die im nächsten Abschnitt vorgestellt wird und die den Entwurf von GUIs sehr einfach macht.

Es ist allerdings nur der erste Schritt, ein paar Objekte in einem Fenster zu verteilen. So gut wie immer ist es gewollt, interagieren zu können, z. B. durch das Betätigen von Schaltflächen. Die Aufgabe festzustellen, wann bspw. auf ein bestimmtes Objekt geklickt wurde, wird einem von MATLAB abgenommen. Dazu muss nur festgelegt werden, auf welche Ereignisse (z. B. Klicken, Tastendruck) ein Objekt reagieren und welche Funktion (Ereignisfunktion) dann aufgerufen werden soll. Auch das ist mit Hilfe von GUIDE einfach einzustellen.

In diesem Abschnitt werden GUIDE und grundsätzliche Funktionen von GUIs vorgestellt. Als Beispiel dient dann im nächsten Abschnitt 14 der Anfang der GUI, die in diesem Versuch entworfen werden soll.

13.2 Gestaltung der GUI mittels GUIDE

Das MATLAB-Modul GUIDE (Graphical User Interface Design Environment) ist eine Entwicklungsumgebung, mit der es sehr einfach ist, eine Benutzeroberfläche zu gestalten.

Zum Öffnen von GUIDE verwendet man den Befehl

```
>> guide
```

Es öffnet sich das Fenster *GUIDE Quick Start* (Abbildung 13.1), in dem man zwischen verschiedenen Vorlagen (Templates) wählen kann. Es ist möglich, über **Blank GUI (Default)** eine neue GUI zu erzeugen oder eine bereits erzeugte GUI über Registerreiter **Open Existing GUI** zu laden.

Bei Auswahl von *Blank GUI (Default)* wird ein leeres Figure „untitled.fig“ (siehe Abbildung 13.2) und ein gleichnamiges M-File „untitled.m“ erzeugt. Im Figure werden alle benötigten weiteren Grafikobjekte

Tabelle 13.1: GUIDE-Werkzeuge

| Werkzeug | Beschreibung |
|-----------------|--|
| Select | Zum Auswählen eines Objekts in der GUI |
| Push Button | Rechteckfeld, welches bei Klick-Ereignis eine Funktion aufruft |
| Slider | Schieberegler |
| Radio Button | rundes Auswahlfeld |
| Check Box | quadratisches Auswahlfeld |
| Edit Text | Textfeld für Ein- und Ausgabe |
| Static Text | statisches Textfeld |
| Pop-up Menu | Auswahlliste (nur ein Element sichtbar) |
| Listbox | Auswahlliste (ggfs. auch mehrere Elemente sichtbar) |
| Toggle Button | An/Aus-Schalter |
| Axes | Achsensystem (siehe Versuch 2) |
| Panel | Rechteckfeld für grafische Gestaltung |
| Button Group | Zum Zusammenfassen mehrerer Radio-Buttons zu einer Auswahlgruppe |
| ActiveX Control | Zum Einbinden von externer Software |

erstellt. Im M-File werden später die Ereignisfunktionen erstellt (automatisch von GUIDE) und können dort mit Funktionalität versehen werden (durch den Ersteller).

In GUIDE befindet sich das Figure im Entwurfsmodus. D. h., dass neue Objekte auf dem Figure platziert und verändert werden können. Um das Figure auszuführen, kann in der horizontalen Symbolleiste von GUIDE auf den grünen Pfeil geklickt werden. Es öffnet sich dann ein neues Fenster mit dem Figure, und in diesem normalen „Ausführungsmodus“ funktionieren dann die Objekte normal, d. h., dass bspw. ein Klicken auf eine Schaltfläche diese betätigt und nicht nur markiert.

Um nun das Figure zu editieren, stehen verschiedene „Werkzeuge“ zur Verfügung (links in Abbildung 13.2, die Erscheinung variiert etwas zwischen den MATLAB-Versionen). In Tabelle 13.1 sind die möglichen Werkzeuge aufgeführt.

Nun wird beispielhaft im Figure eine Befehlsschaltfläche (*Push Button*) erstellt. (Das Erstellen und Editieren anderer Objekte erfolgt analog.) Hierzu wählt man das Werkzeug *Push Button* aus und zieht mit der Maus die Schaltfläche in gewünschter Größe auf dem Figure auf. Klickt man nun doppelt auf die neu erstellte Schaltfläche, öffnet sich der *Property Inspector* (siehe Abbildung 13.3) mit den Eigenschaften des Objekts. In diesem können die Objekteigenschaften geändert werden.

Eine kleine Auswahl an wichtigen Eigenschaften ist in Tabelle 13.2 zusammengestellt. Besonders wichtig ist die Eigenschaft *Tag* (engl. für Etikett), die den Namen festlegt, mit welchem man später auf dieses Objekt zugreifen kann. Es werden auch die Ereignisfunktionen wie z. B. *Callback* und *CreateFcn* angezeigt. Erstellt werden sollten diese Funktionen jedoch lieber mit einem Rechtsklick auf das betreffende Objekt. In dem dann erscheinenden Kontextmenü können die Ereignisfunktionen dann unter „View Callbacks“ aufgerufen bzw. erstellt werden (siehe Abbildung 13.4). Dann ist die Syntax des Ereignisfunktionsaufrufs auf jeden Fall korrekt. (Dennoch kann es nützlich sein, diese Eigenschaften per Hand zu ändern wenn man den Namen (*Tag*) des Objekts nachträglich ändern will. So kann man die Funktionen per Hand anpassen.)

Wichtig: Die GUI sollte immer über das gleichnamige m-File (bzw. den grünen Pfeil in GUIDE) und *nicht* über das fig-File gestartet werden, damit alle Initialisierungen korrekt vorgenommen werden!

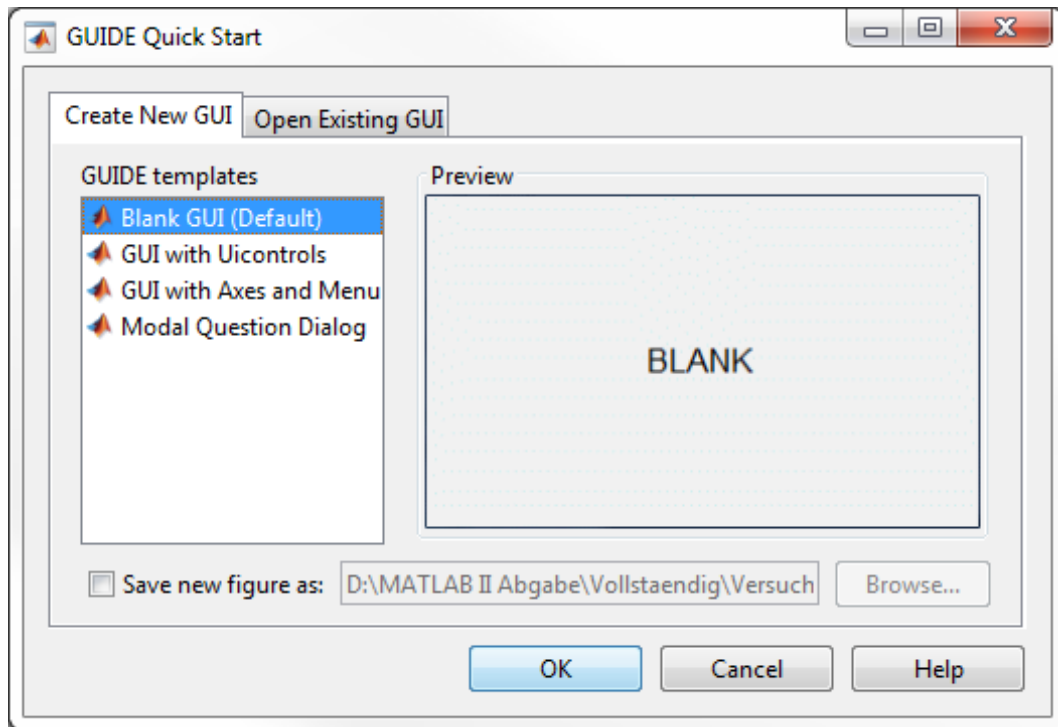


Abbildung 13.1: GUIDE Startfenster

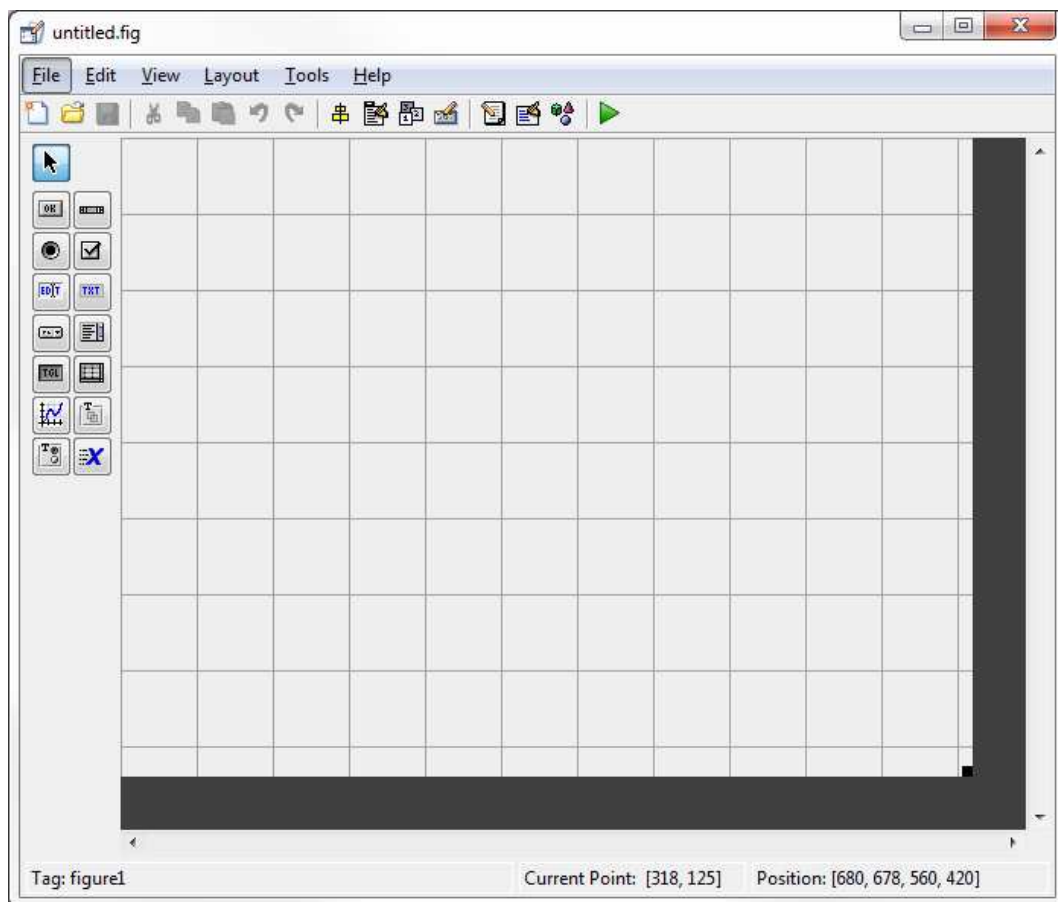


Abbildung 13.2: Leere GUI in GUIDE

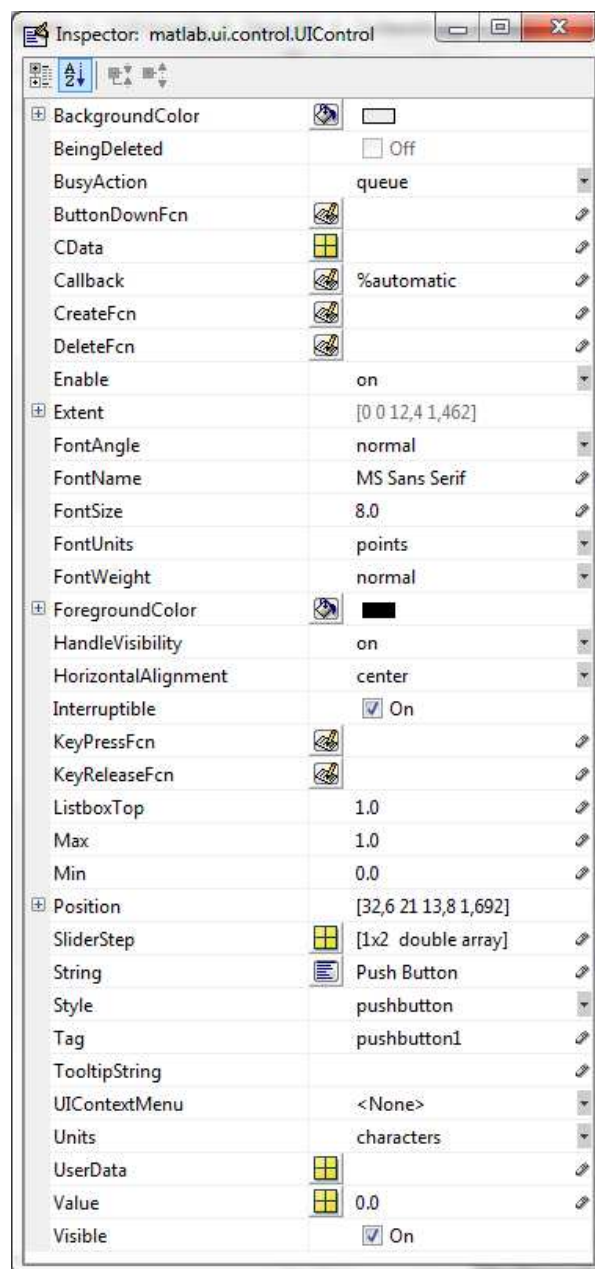


Abbildung 13.3: Property Inspector

Tabelle 13.2: Auswahl wichtiger Objekteigenschaften

| Eigenschaft | Beschreibung |
|-------------|--|
| Tag | Name, unter dem das Objekt im M-File angesprochen werden kann. |
| Callback | Funktion, die beim „Hauptereignis“ auf das Objekt ausgeführt wird. |
| CreateFcn | Funktion, die beim Erzeugen des Objekts ausgeführt wird. |
| String | Beschriftung eines Objekts |
| Max, Min | Grenzen eines Schiebereglers (Slider) |
| SliderStep | Schrittweite eines Schiebereglers in x- und y-Richtung |

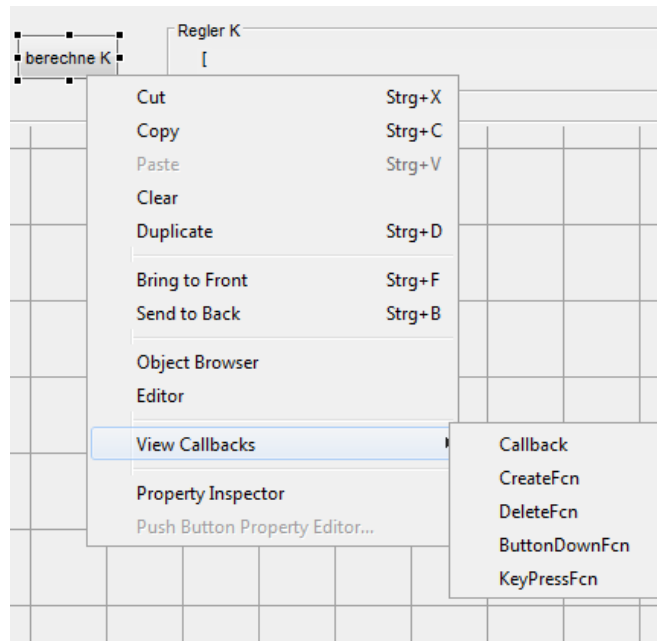


Abbildung 13.4: Auswahl Ereignisfunktionen

13.3 GUI-Funktionalität

Im Folgenden wird kurz auf das „Hauptereignis“ eines Objekts (bei Schaltflächen bspw. das Klicken auf die Schaltfläche) sowie das Ereignis „OpeningFcn“ der GUI eingegangen und nochmal kurz das Ändern und Auslesen von Objekteigenschaften erläutert.

„Hauptereignis“: Callback

Wie schon erwähnt, wird beim Erstellen eines neuen Objektes die m-Datei modifiziert. Diese wird durch eine entsprechende Callback-Funktion

`Tag_Callback(hObject, eventdata, handles)`

automatisch erweitert. Klickt man nun auf ein Objekt (während der Ausführung, nicht der Erstellung) in der GUI (z. B. *Push Button*) oder ändert dies (z. B. *Slider*), wird die zum Objekt passende Callbackfunktion aufgerufen und ausgeführt. Der übergebene Parameter `hObject` ist das Handle des aufrufenden Objekts. Die weiteren Übergabeparameter sind für diesen Versuch nicht relevant. Bei Interesse findet man weitere Erläuterungen in der Hilfe unter „Callback Syntax and Arguments“.

Ereignis „Öffnen“ der GUI: `OpeningFcn`

Die Funktion

`GUI_OpeningFcn(hObject, eventdata, handles, varargin)`

wird direkt vor dem Anzeigen der GUI, nach dem Erzeugen aller Objekte aufgerufen. Diese kann genutzt werden, Startwerte zu generieren und diese den Objekten zuzuweisen.

Handles

Zum Ändern oder Auslesen von Objekt-Eigenschaften (wie in Versuch 3 beschrieben) muss das Handle des betreffenden Objekts bekannt sein. Zunächst sind aber nur die *Tags* der Objekte bekannt.

In der Regel (aber nicht in den Create-Funktionen) sind die Handles im Funktionsargument `handles` verfügbar. Die Feldnamen der Struktur entsprechen dabei den *Tags* der Objekte. Um bspw. das Handle des Objekts mit dem Tag `Q11` zu ermitteln, kann das entsprechende Feld gelesen werden:

```
hQ11 = handles.Q11;
```

Alternativ gibt die Funktion `guihandles` eine Struktur zurück, in der die Handles aller Objekte des aktiven Figures gespeichert sind. Um bspw. das Handle des Objekts mit dem Tag `Q11` zu ermitteln, werden zunächst mit

```
stH = guihandles();
```

alle Handles der GUI in `stH` gespeichert, und dann das gesuchte Handle mit

```
hQ11 = stH.Q11;
```

ermittelt.

Eigenschaften auslesen und setzen: `get` und `set`

Bei bekanntem Handle `objecthandle` eines Objekts können die aus Versuch 3 bekannten Funktionen

```
get(objecthandle, 'property')
```

und

```
set(object, 'property', value)
```

verwendet werden, um Eigenschaften auszulesen und zu ändern.

So wird z. B. der aktuelle Wert eines Schiebereglers mit dem Tag `slider1` über

```
pos = get(stH.slider1, 'value');
```

ermittelt und in `pos` gespeichert.

Die Hintergrundfarbe der Schaltfläche `pushbutton1` würde mit

```
set(stH.pushbutton1, 'BackgroundColor', [0.10 0.30 0.70]);
```

geändert werden, oder mit

```
set(stH.edit1, 'String', erg);
```

würde ein Ergebnis `erg` im Textfeld `edit1` ausgegeben.

Es können an dieser Stelle unmöglich alle Objekte mit deren Eigenschaften vorgestellt werden. Um die Funktionalität der verschiedenen Objekte und deren mögliche Eigenschaften zu erfahren, wird auf die MATLAB-Hilfe verwiesen.

14 Erstellen einer grafischen Benutzeroberfläche

Im vorangegangenen Abschnitt 13 wurde allgemein gezeigt, wie mit Hilfe von GUIDE eine graphische Benutzeroberfläche erzeugt wird.

Im Folgenden sollen diese Grundlagen dazu genutzt werden, das nachfolgende Beispiel besser zu verstehen.

Das Beispiel, dessen Figure und Code auch zur Verfügung gestellt wird, stellt zugleich den Anfang einer GUI zur Simulation des Schlitten-Pendel-Systems dar, die in diesem Versuch vervollständigt werden soll.

Der als Beispiel vorliegende Stand der GUI (Abbildung 14.1) verfügt über

- Eingabefelder für die Q - und R -Matrix,
- eine Auswahlmöglichkeit für den Arbeitspunkt,
- eine Schaltfläche zur Berechnung des Reglervektors K und ein Ausgabefeld für diesen Vektor sowie
- einem Achsensystem, in dem später die Animation des Pendels zu sehen sein soll.

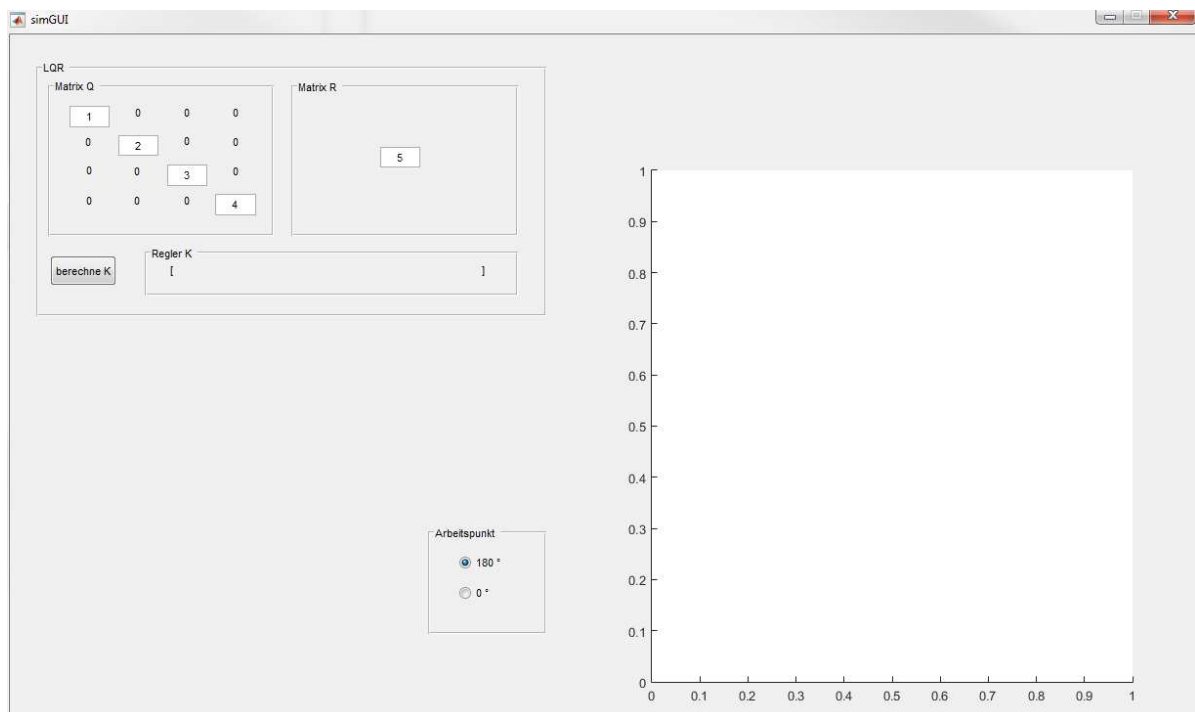


Abbildung 14.1: Gegebener Stand der GUI

Die wesentlichen Schritte, mit denen dieser Stand erreicht wurde, sind im Folgenden aufgeführt.

14.1 Eingabefelder für Q- und R-Matrix

Im ersten Schritt werden Eingabemöglichkeiten für die **Q**- und **R**-Matrix des LQR-Verfahrens erstellt.

Es existieren zwei Arten von Textfeldern. Zum einen gibt es das *statische* Textfeld. Wie der Name schon vermuten lässt, kann der Benutzer der Oberfläche später keinen Einfluss auf den Inhalt des Feldes nehmen. Daher wird es im Wesentlichen zur Ausgabe von Nachrichten, Hinweisen und Ergebnissen des Programms oder für nicht veränderliche Texte (z. B. Überschriften) genutzt.

Im Gegensatz dazu kann der Benutzer über die *dynamischen* Textfelder Eingaben vornehmen. Die Handhabung gleicht der des statischen Textfeldes. Im m-File existiert für ein dynamisches Feld allerdings zusätzlich eine Callback-Funktion, die nach einer Eingabe des Benutzers beim Verlassen des Textfeldes aufgerufen wird.

Für den konkreten Fall der **Q**- bzw. **R**-Matrix bietet sich an, jedes einzelne Element der Matrizen durch ein eigenes Textfeld zu repräsentieren. Wobei lediglich die Diagonalelemente ein dynamisches Textfeld benötigen, da in der Regel nur diese verwendet werden und alle anderen Elemente mit Nullen besetzt sind.

Anmerkung 1: Es wäre auch möglich, beide Matrizen durch jeweils ein Textfeld darzustellen und direkt als Matrix in der bekannten „MATLAB-Syntax“ einzugeben bzw. auszulesen. Allerdings führt diese Methode zum einen zu einer recht unübersichtlichen Darstellung und ist zum anderen sehr anfällig für Schreibfehler, weshalb es sich für ungeübte Nutzer nur schwer handhaben lässt.

Anmerkung 2: Auf die Darstellung der Neben-Diagonalelemente könnte auch verzichtet werden. Diese sorgen lediglich für eine ansprechendere Optik.

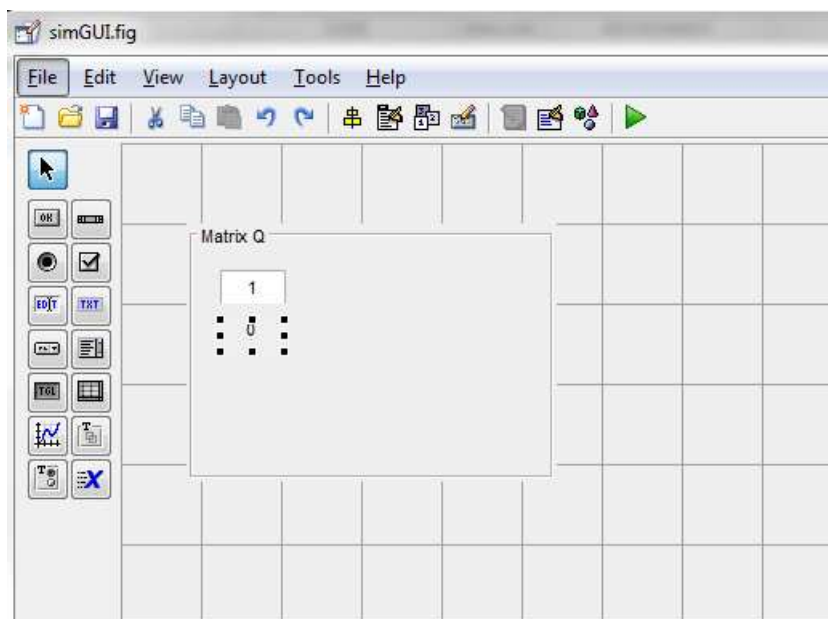


Abbildung 14.2: Statisches (unten) und dynamisches (oben) Textfeld

Nach dem Platzieren der Textobjekte (Abbildung 14.2) sollten die *Tags* mit aussagekräftigen Namen belegt werden. Im vorliegenden Falle der Matrizen bietet sich an jedem Feld die Bezeichnung des jeweiligen Elementes zuzuweisen. Hier werden also die Eingabefelder für die Diagonalelemente der Matrix **Q** mit Q11, Q22, Q33 und Q44 betitelt.

Optional kann man die einzelnen Textobjekte mit Hilfe von einem Rahmen (*Panel*) zu einer Art Einheit zusammenfassen (Abbildung 14.3). Neben dem optischen Aspekt bieten Panels auch den Vorteil, dass

die zusammengefassten Objekte fest mit dem Panel verbunden sind. Daher verschiebt man das Panel immer mit allen verbundenen Objekten.

Initialisierung

Wie im Abschnitt 13.3 beschrieben, wird hier die `OpeningFcn` der GUI zur Initialisierung der gewünschten Startwerte verwendet. Folgendes Beispiel zeigt die Funktion `simGUI_OpeningFcn`, in der festgelegt wird, dass bspw. in das Textfeld Q11 beim Starten der GUI eine „1“, in Q22 eine „2“ usw. eingetragen wird. (Die nicht eingerückten Codezeilen stammen von GUIDE. Das kann in diesem Versuch ignoriert werden.)

Listing 14.1: `simGUI_OpeningFcn`

```
% --- Executes just before simGUI is made visible.
function simGUI_OpeningFcn(hObject, eventdata, handles, varargin)
3 % This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles     structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simGUI (see VARARGIN)

8 % Choose default command line output for simGUI
handles.output = hObject;

% Update handles structure
13 guidata(hObject, handles);

% UIWAIT makes simGUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);

18 set(handles.Q11, 'String', 1);
   set(handles.Q22, 'String', 2);
   set(handles.Q33, 'String', 3);
   set(handles.Q44, 'String', 4);
   set(handles.R, 'String', 5);
23
% end function simGUI_OpeningFcn
```

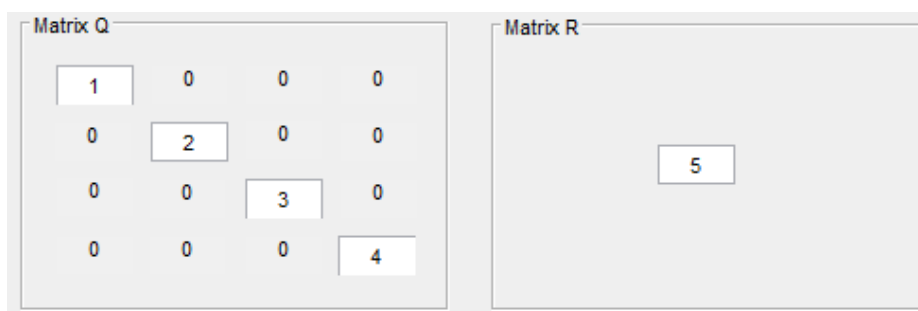


Abbildung 14.3: Eingabefelder für Matrizen Q und R mit Rahmen

14.2 Auswahl des Arbeitspunktes

Im Folgenden ist die Programmierung der in Abbildung 14.4 gezeigten Auswahlfelder (*Radio-Buttons*) ap1 und ap2 zur Auswahl des Arbeitspunktes erläutert. Hierbei ist darauf zu achten, dass sich die beiden Auswahlfelder zur Auswahl von AP 0° und 180° wechselseitig ausschließen und somit niemals beide Auswahlfelder zugleich aktiv (ausgewählt) sein können.

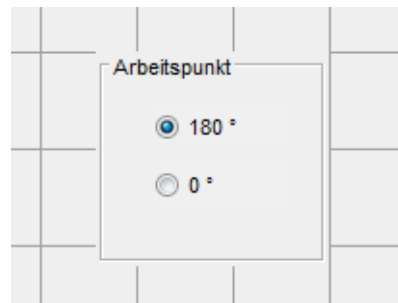


Abbildung 14.4: Auswahlfelder für Arbeitspunkt

Der Wert eines Auswahlfeldes kann über die Eigenschaft *Value* ausgelesen oder gesetzt werden. Dabei ist die Eigenschaft 1, wenn das Feld angewählt ist und 0, wenn nicht.

In diesem Beispiel ist das Umschalten der Auswahlfelder manuell gesteuert. Beim Klicken auf ein Auswahlfeld wird die entsprechende Callback-Funktion ausgeführt. In Listing 14.2 ist die Callback-Funktion des Auswahlfeldes ap1 abgedruckt. In dieser wird zunächst der Wert des zu dieser Funktion gehörenden Auswahlfeldes (also auf das gerade geklickt wurde) ermittelt, und dann wird das andere Auswahlfeld entsprechend auf den anderen Wert gesetzt. Dazu muss über die Funktion *guihandles* wie im vorherigen Kapitel beschrieben das Handle des jeweils anderen Auswahlfelds bestimmt werden.

Die Callback-Funktion des anderen Auswahlfelds ap2 ist entsprechend aufgebaut (Listing 14.3).

Eine andere Möglichkeit zu erreichen, dass immer nur ein Feld ausgewählt ist, wäre das Verwenden einer *Button Group*.

Listing 14.2: ap1_Callback

```
1 % --- Executes on button press in ap1.
function ap1_Callback(hObject, eventdata, handles)

    h = guihandles();

6    value = get(hObject, 'Value');

    if value == 1
        set(h.ap2, 'Value', 0);
    elseif value == 0
11    set(h.ap2, 'Value', 1);
    end

% end function ap1_Callback
```

Listing 14.3: ap2_Callback

```
1 % --- Executes on button press in ap2.
```

```
function ap2_Callback(hObject, eventdata, handles)
```

```
h = guihandles();
```

```
6 value = get(hObject, 'Value');
```

```
if value == 1
```

```
    set(h.ap1, 'Value', 0);
```

```
elseif value == 0
```

```
11     set(h.ap1, 'Value', 1);
```

```
end
```

```
% end function ap2_Callback
```

14.3 Berechnung des Reglers K

Nachdem nun Eingabefelder für die Matrizen Q und R und eine Auswahlmöglichkeit für den Arbeitspunkt erstellt worden sind, besteht der nächste Schritt darin, eine Schaltfläche in die GUI zu integrieren, bei deren Betätigung der Reglervektor K mit den Funktionen aus dem Versuch 3 berechnet und in einem Textfeld ausgegeben wird.

Wiederum analog zur oben beschriebenen Einbindung von Textfeldern werden zunächst ein *Push Button*-Objekt sowie ein statisches Textfeld erzeugt und mit entsprechend eindeutigen Bezeichnungen belegt. Die Schaltfläche wird noch über die Eigenschaft *String* im *Property Inspector* beschriftet. Diese sind hier (siehe Abbildung 14.5) noch durch weitere Rahmen geordnet, was aber keinen Einfluss auf die Funktion hat.

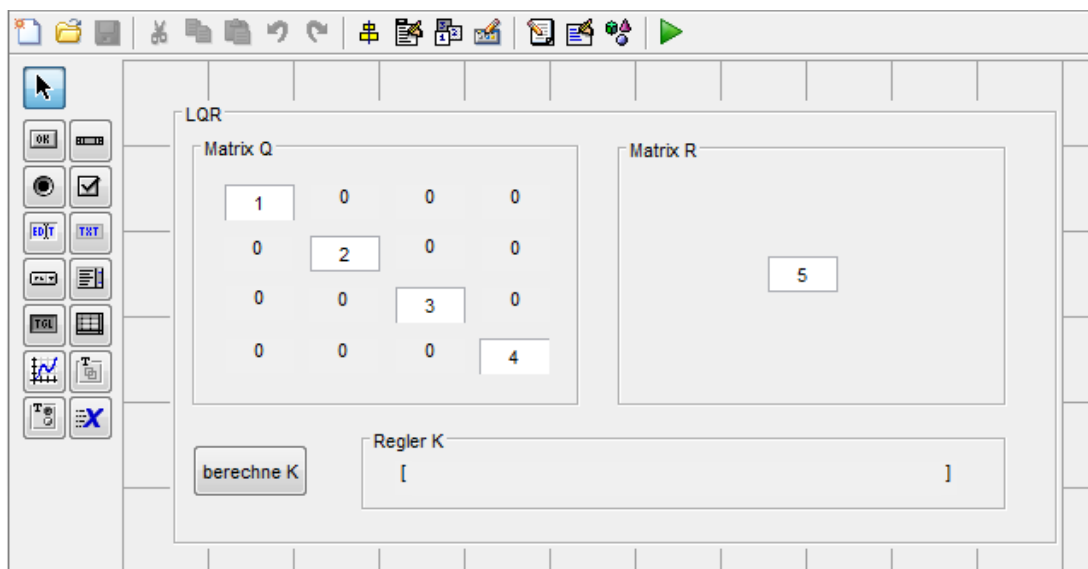


Abbildung 14.5: GUI im Entwurf mit hinzugefügter Schaltfläche und Ausgabefeld für Vektor K

Beim Klicken auf die „berechneK“-Schaltfläche (im normalen „Ausführungsmodus“) wird nun die zugehörige Ereignisfunktion `berechneK_Callback` ausgeführt, welche im Listing 14.4 abgedruckt ist.

Listing 14.4: `berechneK_Callback`

```

1  % --- Executes on button press in berechneK.
function berechneK_Callback(hObject, eventdata, handles)
    % hObject      handle to berechneK (see GCBO)
    % eventdata    reserved - to be defined in a future version of MATLAB
    % handles      structure with handles and user data (see GUIDATA)

6

    % Struktur mit den Handles aller Objekte der GUI erzeugen
    h = guihandles();

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
11  % Auslesen der Matrix Q
    q11 = str2num(get(h.Q11, 'String'));
    q22 = str2num(get(h.Q22, 'String'));
    q33 = str2num(get(h.Q33, 'String'));
    q44 = str2num(get(h.Q44, 'String'));

16

    Q = diag([q11 q22 q33 q44]);
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
21  % Auslesen von R
    R = str2num(get(h.R, 'String'));
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
26  % Auslesen des Arbeitspunkts
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    value = get(h.ap1, 'Value');
    if (value == 1)
31        AP = pi;
    else % (value == 0)
        AP = 0;
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

36

    stPendel = ladePendel();
    [A, B, C, D] = linPendelZR(stPendel, AP);
    K = berechneLQR(A, B, Q, R);

41

    % Anzeigen des Vektors 'K' im Textfeld 'reglerK'
    set(h.reglerK, 'String', num2str(K));

% end function berechneK_Callback

```

Zum Berechnen des Reglers muss man auf den Inhalt der Matrizenfelder von Q und R zugreifen sowie den Wert eines der beiden Auswahlfelder für den Arbeitspunkt ermitteln können. Hierfür benötigt man natürlich die Handles dieser Felder, welche – wie schon beschrieben – über die Funktion `guihandles` ermittelt werden können.

Beim Auslesen der Matrizenelemente ist darauf zu achten, dass die String-Eigenschaft von Textfeldern immer eine Zeichenkette ist. Um diese in einen numerischen Wert umzuwandeln, wird hier die Funktion `str2num` verwendet. Umgekehrt wird am Ende der Funktion `berechneK_Callback` der Wert des Vektors `K`, der im Textfeld `reglerK` angezeigt werden soll, mit der Funktion `num2str` in ein Zeichenkette umgewandelt. (Zahlenwerte würden an dieser Stelle zwar auch automatisch in Zeichenketten konvertiert werden, aber in diesem Fall werden Vektoren immer als Spaltenvektoren betrachtet, was hier optisch nicht passen würde.)

15 Weitere MATLAB-Funktionen

15.1 Polvorgabe

MATLAB stellt die Funktion

place

zur Verfügung, mit der ein Reglerentwurf im Zustandsraum mit Polvorgabe erfolgen kann. Für die Syntax und weitere Informationen wird auf die MATLAB-Hilfe verwiesen. (Es wird auch (noch) die Funktion `acker` angeboten. Aus numerischen Gründen ist jedoch die Funktion `place` zu bevorzugen.)

15.2 Cell-Arrays

Je nachdem welche Objekte für die GUI verwendet werden, kann es sein, dass auch mit Cell-Arrays gearbeitet werden muss (z. B. bei *Pop-up Menus*). Deshalb wird hier kurz auf die wesentlichen Punkte eingegangen.

Ein Cell-Array besteht aus einer oder mehrerer „Zellen“. Eine Zelle ist dabei eine Art Container, die jeden beliebigen MATLAB-Datentyp aufnehmen kann. Daher können in *einem* Cell-Array im Unterschied zu einem Vektor von numerischen Werten oder von Strukturen auch Daten *verschiedener* Typen gespeichert werden. Bei einem numerischen Vektor müssen dagegen alle Einträge Zahlenwerte, bei einem Vektor von Strukturen müssen alle Elemente Strukturen mit gleichen Feldnamen sein.

Ein Cell-Array wird erzeugt, indem die Einträge in geschweiften Klammern geschrieben werden. Folgende Eingabe erzeugt ein Cell-Array mit drei Elementen, davon zwei Zeichenketten und eine Zahl.

```
>> clTest = {'Eintrag 1', 'Eintrag 2', 15}
clTest =
    'Eintrag 1'    'Eintrag 2'    [15]
```

Das Wichtigste beim Verwenden der Cell-Arrays ist, daran zu denken, dass der Zugriff auf die einzelnen Elemente ebenfalls über geschweifte Klammern erfolgen muss.

```
>> val = clTest{3}
val =
    15
```

Werden runde Klammern verwendet, dann wird die entsprechende Zelle, aber nicht deren Inhalt, zurückgegeben.

```
>> val = clTest(3)
val =
    [15]
```

Dies ist ein bedeutender Unterschied, da auf die Zelle nicht die gewohnten Operationen ausgeführt werden können.


```
>> val + 3
??? Undefined function or method 'plus' for
input arguments of type 'cell'.
```

MATLAB verwendet Cell-Arrays hauptsächlich als „Vektor von Zeichenketten“ an Stellen, an denen eine variable Anzahl von Zeichenketten erwartet oder zurückgegeben wird. Ein Beispiel wäre die `legend`-Funktion, die neben der bekannten Syntax

```
legend('Kurve 1', 'Kurve 2')
```

auch mit einem Cell-Array verwendet werden kann.

```
legend({'Kurve 1', 'Kurve 2'})
```

Ein anderes Beispiel tritt mit der Funktion `uigetfile` auf. Diese erlaubt das Auswählen einer vorhandenen Datei mit der gewohnten Oberfläche. Mit der Option `MultiSelect` erlaubt diese Funktion auch die Auswahl mehrerer Dateien. In diesem Fall ist der Rückgabewert vom Typ Cell-Array.

```
clSelectedFiles = uigetfile('MultiSelect', 'on')
clSelectedFiles =
    'systemPendel_V3.m'    'simGUI.fig'    'simGUI.m'
```

(Zur Auswahl einer Datei zum Speichern steht `uiputfile` zur Verfügung. MATLAB stellt auch verschiedene Funktionen zur Verfügung, die dazu dienen, einfache Eingaben oder Ausgaben über ein kleines Fenster zu realisieren. Bei Interesse dazu am besten die MATLAB-Hilfe der Funktion `msgbox` betrachten. Von dort aus sind die anderen Funktionen verlinkt.)

15.3 Persistent und globale Variablen

Normalerweise hat jede MATLAB-Funktion ihren eigenen Workspace, in dem sie Variablen speichert. Nach Beendigung einer Funktion werden alle „normalen“ internen (lokalen) Variablen gelöscht und stehen nicht mehr zur Verfügung. Funktionen können nur über die Parameterliste und die Rückgabewerte Daten untereinander austauschen.

Im vorangegangenen Versuch haben Sie mit `assignin` und `evalin` zwei Möglichkeiten kennengelernt, diese Einschränkungen zu umgehen.

Es gibt weiterhin die Möglichkeit, Variablen als persistent oder global zu deklarieren.

- Variablen, die als persistent (engl. für nachhaltig, bleibend) deklariert sind, werden von MATLAB auch nach Beenden der Funktion gespeichert und stehen wieder zur Verfügung, wenn diese **selbe** Funktion wieder aufgerufen wird. Bei erneutem Aufruf der Funktion kann mit dem alten Wert gearbeitet werden. Andere Funktionen können **nicht** auf die persistent Variable zugreifen.
- Variablen, die als global deklariert sind, sind im Gegensatz zu lokalen Variablen nicht nur im lokalen Gültigkeitsraum (Workspace), sondern global verfügbar. Die Verwendung globaler Variablen gilt im Allgemeinen als schlechter Programmierstil, da durch ihre übermäßige Verwendung Programme sehr schnell unübersichtlich werden können. Dies ist besonders der Fall, wenn Daten nicht über die Parameterliste von Funktionen ausgetauscht werden, sondern schlicht aus Faulheit global definiert werden. In manchen Fällen ist ihre Verwendung jedoch durchaus gerechtfertigt oder sogar notwendig.

Wird eine persistent oder globale Variable zum ersten Mal deklariert, so enthält sie eine leere Matrix.

Persistent oder globale Variablen werden nicht durch den Befehl `clear` gelöscht. Globale Variablen können mit `clear global` oder `clear all` gelöscht werden, persistent Variablen mit `clear functions` oder `clear all`. Außerdem führt eine Änderung an einer Funktion dazu, dass die zu dieser Funktion gehörenden persistent Variablen gelöscht werden.

`who global` zeigt alle globalen Variablen an.

Ein kleines MATLAB-Beispiel zu persistent und globalen Variablen:

Folgendes Listing definiert eine Funktion, die bei jedem Aufruf den Wert des übergebenen Arguments aufaddiert und die aktuelle Summe zurückgibt.

Listing 15.1: Funktion `Summierer`.

```
1 function s = Summierer(a)

    persistent s_p;

    if isempty(s_p) % s_p zum ersten Mal deklariert?
6       s_p = 0;
    end

    s_p = s_p + a;
    s = s_p;

11 end % function Summierer
```

Entsprechend erhält man folgendes Verhalten:

```
>> Summierer(5)
ans =
     5
>> Summierer(5)
ans =
    10
```

Von „außen“ hat man keinen Einfluss auf den Wert der Variable `s_p` der Funktion. Im Vergleich dazu könnte man diese Variable auch als global deklarieren, wie in der Funktion

Listing 15.2: Funktion `SummiererG`.

```
function s = SummiererG(a)

3     global g_p;

    if isempty(g_p) % g_p zum ersten Mal deklariert?
        g_p = 0;
    end

8     g_p = g_p + a;
    s = g_p;

end % function SummiererG
```

Das grundsätzliche Verhalten ist dasselbe wie im Beispiel zuvor

```
>> SummiererG(5)
ans =
     5
>> SummiererG(5)
ans =
    10
```

nur dass jetzt auch von anderen Stellen direkt auf die Variable `g_p` zugegriffen werden kann, z. B. direkt über das Command-Window (Base-Workspace):

```
>> global g_p;
>> g_p = 100;
>> SummiererG(5)
ans =
    105
```

(Es ist notwendig, die Variable zunächst mit `global` explizit im Base-Workspace zu deklarieren.)



Versuch 5

Aufschwingsteuerung für das Pendel

In diesem Versuch wird für das bisher betrachtete Schlitten-Pendel-System eine Steuerung berechnet, die das Pendel von der unteren in die obere Ruhelage überführt. Bei dem hier verwendeten Ansatz zur Berechnung, der ausführlich dargestellt wird, führt dies auf ein Differentialgleichungssystem mit Randbedingungen. Dies kann mit der MATLAB-Funktion `bvp4c` gelöst werden. Die Steuerung wird dann in Simulink aufgebaut und am Modell verifiziert.

Die Funktion `bvp4c` erwartet als Argumente Funktionen, ähnlich dem DGL-Löser `ode...`, mit dem schon im Praktikum MATLAB/Simulink I gearbeitet wurde. Dabei wurden die übergebenen Funktionen direkt über den Namen angegeben. In diesem Versuch wird eine flexiblere Möglichkeit gezeigt, diese Funktionen zu benutzen.

Neben den Lösern für DGLs verfügt MATLAB auch über weitere Funktionen, die eine ähnliche Syntax aber andere Aufgaben haben, z. B. `fzero` (Nullstellenberechnung) und `fminsearch` (Optimierer). So können die in diesem Versuch erworbenen Fähigkeiten leicht bei anderen Problemstellungen angewandt werden.

| | |
|--|-----------|
| 16 Steuerung und Trajektorienberechnung | 77 |
| 16.1 Steuerung | 77 |
| 16.2 Berechnung der Steuerung | 78 |
| 17 MATLAB | 80 |
| 17.1 Function-Handles | 80 |
| 17.2 Der Löser <code>bvp4c</code> | 82 |
| 17.3 Parametrisieren von Funktionen | 84 |
| 17.4 Lookup-Tables | 87 |
| 18 Anhang – Herleitung der Steuerung | 89 |
| 18.1 Systemdarstellung | 89 |

| | |
|---|-----------|
| 18.2 Anfangs- und Endzustand, Randbedingungen | 90 |
| 18.3 Ansatzfunktion | 91 |
| 18.4 Berechnung bei flachem Ausgang | 91 |
| 18.5 Berechnung bei nicht-flachem Ausgang | 92 |
| 18.6 Erweiterung der Ansatzfunktion | 92 |
| 18.7 Berechnung der Parameter | 93 |
| 18.8 Anmerkungen | 93 |
| 18.9 Anwendung auf Beispielsystem | 94 |
| 19 Anhang – Randwertprobleme | 96 |
| 19.1 Lösbarkeit von Randwertproblemen | 96 |
| 19.2 Geringere DGL-Ordnung als Anzahl der Randbedingungen | 97 |

16 Steuerung und Trajektorienberechnung

16.1 Steuerung

In diesem Versuch soll der Aufschwung des Pendels am Simulationsmodell durchgeführt werden. D. h., es soll erreicht werden, dass das Pendel aus der unteren Ruhelage durch eine geeignete Bewegung des Schlittens in die obere Ruhelage überführt wird.

Dazu soll in diesem Versuch eine Steuerung verwendet werden. Dabei wird zu einem gewünschten Ausgangsgrößenverlauf $y^*(t)$ der dafür nötige Eingangsgrößenverlauf $u^*(t)$ berechnet und dieser dann auf das System gegeben.

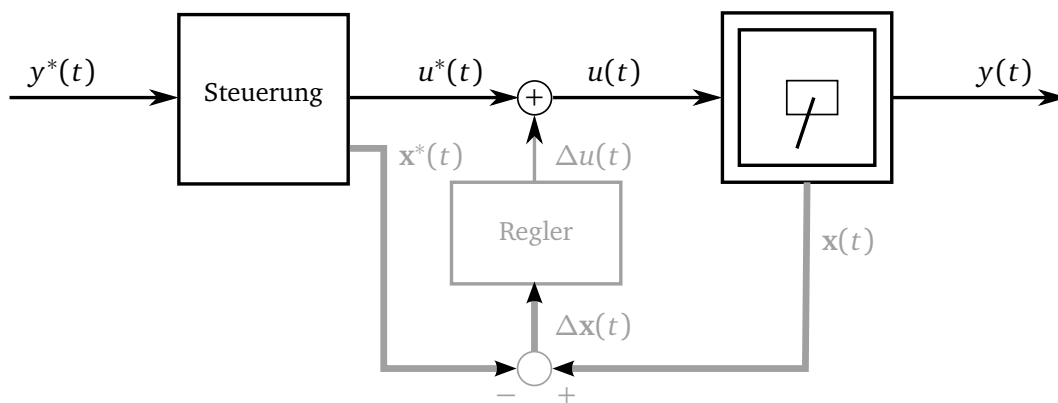


Abbildung 16.1: Steuerung mit Trajektorienfolgeregelung (letzte in grau)

Abbildung 16.1 zeigt in schwarz die Struktur der Steuerung. Im Idealfall, wenn das Modell, welches zur Berechnung von $u^*(t)$ verwendet wurde, genau mit dem realen Modell übereinstimmt und keine Störungen auftreten, wird der reale Modellausgang $y(t)$ genau dem gewünschten Ausgang $y^*(t)$ entsprechen.

Da aber das reale System immer von dem Modell abweichen wird, mit dem die Trajektorie berechnet wurde und zudem noch (unbekannte) Störungen auf das reale System wirken, wird der reale Verlauf vom berechneten Verlauf abweichen. Um diese Abweichung zum Sollverlauf auszuregeln, kann eine Trajektorienfolgeregelung entworfen werden. Diese, hier als Zustandsregler ausgeführt, ist in Abbildung 16.1 in grau eingezeichnet. Es wird ein Regler hinzugefügt, der als Eingangsgröße die Abweichung $\Delta \mathbf{x}(t)$ des Istzustands $\mathbf{x}(t)$ zum Sollzustand $\mathbf{x}^*(t)$ erhält. Aus dieser Differenz $\Delta \mathbf{x}(t)$ berechnet der Regler ein $\Delta u(t)$, welches zu $u^*(t)$ dazuaddiert werden muss. (Würden beide Zustände exakt übereinstimmen wäre die Regelabweichung $\Delta \mathbf{x}(t)$ demnach Null und der Regler hätte keine Funktion.)

Der Vorteil einer solchen Struktur aus Steuerung und Regelung liegt darin, dass Führungs- und Störverhalten des Systems getrennt ausgelegt werden können. Würde nur ein Regler (ohne Steuerung) verwendet werden, so müsste man bei dessen Auslegung einen Kompromiss zwischen gutem Führungsverhalten und gutem Störverhalten eingehen (sofern sich überhaupt ein Regler finden ließe). Zudem benötigen nichtlineare Regler eher große Rechenkapazitäten im Betrieb. Da bei dem Ansatz mit Steuerung die Berechnung des Eingangsgrößenverlaufs $u^*(t)$, der Zustände $\mathbf{x}^*(t)$ und der Reglerparameter (siehe nächsten Versuch) im voraus und nicht in Echtzeit stattfindet, können auch aufwendigere Berech-

nungen durchgeführt werden. [8]

In diesem Versuch soll die Steuerung des Systems berechnet und in Simulink implementiert werden. Die Trajektorienfolgeregelung ist Inhalt des folgenden Versuchs.

16.2 Berechnung der Steuerung

In diesem Versuch wird ein Steuergrößenverlauf $u^*(t)$ berechnet, der das Pendel von der unteren Gleichgewichtslage in die obere überführen soll. Dieser Übergang soll zum Zeitpunkt $t = 0$ beginnen und bei $t = T$ abgeschlossen sein. Diese Zeit T wird im Folgenden als Übergangszeit und der Ausgangs- und Endzustand mit \mathbf{x}_0 bzw. \mathbf{x}_T bezeichnet.

$$\mathbf{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{x}_T = \begin{bmatrix} 0 \\ 0 \\ \pi \\ 0 \end{bmatrix} \quad (16.1)$$

Das Schlitten-Pendel-System wird in einer etwas vereinfachten Form betrachtet, damit die Gleichungen möglichst kompakt bleiben. Als Eingangsgröße u wird nicht mehr die Kraft F , die auf den Schlitten wirkt, sondern die Schlittenbeschleunigung $\ddot{x} = \ddot{x}_1$ verwendet. Die Reibung wird auch weiterhin vernachlässigt. Damit ergibt sich für das System die nichtlineare Zustandsraumdarstellung

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} x_2 \\ 0 \\ x_4 \\ -\frac{m_p \frac{l}{2} g \sin x_3}{m_p \frac{l^2}{4} + J} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ -\frac{m_p \frac{l}{2} \cos x_3}{m_p \frac{l^2}{4} + J} \end{bmatrix} u \quad (16.2)$$

$$y = x_1, \quad (16.3)$$

wobei $J = \frac{1}{12} m_p l^2$ das Massenträgheitsmoment des Pendelstabes bezüglich dessen Schwerpunktes ist.

Für den Systemausgang $y^*(t)$ wird die Gleichung

$$y^*(t, p_1, p_2) = (-p_1 - 3p_2) \cdot \left(\frac{t}{T}\right)^3 + (3p_1 + 8p_2) \cdot \left(\frac{t}{T}\right)^4 + (-3p_1 - 6p_2) \cdot \left(\frac{t}{T}\right)^5 + p_1 \cdot \left(\frac{t}{T}\right)^6 + p_2 \cdot \left(\frac{t}{T}\right)^7 \quad (16.4)$$

vorgegeben, die neben der Zeit t noch von den zwei Parametern p_1 und p_2 abhängt, auf deren Funktion später eingegangen wird.

Diese Gleichung beruht auf dem in [7] vorgestellten Vorgehen, welches im Abschnitt 18 für Interessierte dargestellt ist. (Es ist jedoch nicht notwendig, dieses nachzuvollziehen, die Korrektheit bzw. Sinnhaftigkeit dieser Gleichung zeigt sich im Weiteren.)

Aus den ersten beiden Zeilen der Zustandsgleichung (16.2) folgt $\ddot{x}_1 = u$ und mit der Ausgangsgleichung (16.3) ergibt sich dann $u^* = \ddot{y}^*$, also

$$u^*(t, p_1, p_2) = \ddot{y}^*(t, p_1, p_2) = \left(6 \cdot (-p_1 - 3p_2) \cdot \left(\frac{t}{T}\right) + 12 \cdot (3p_1 + 8p_2) \cdot \left(\frac{t}{T}\right)^2 + 20 \cdot (-3p_1 - 6p_2) \cdot \left(\frac{t}{T}\right)^3 + 30p_1 \cdot \left(\frac{t}{T}\right)^4 + 42p_2 \cdot \left(\frac{t}{T}\right)^5 \right) \cdot \frac{1}{T^2}. \quad (16.5)$$

Setzt man dieses $u^*(t, p_1, p_2)$ wieder in die Zustandsgleichung ein, so stehen in den ersten beiden Zeilen $\dot{y}^* = \dot{y}^*$ und $\ddot{y}^* = \ddot{y}^*$ (was nicht überrascht, da mit diesen Zeilen das $u^*(t)$ bestimmt wurde). Die letzten beiden Zeilen

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} x_4 \\ -\frac{m_p \frac{l}{2} g \sin x_3}{m_p \frac{l^2}{4} + J} \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{m_p \frac{l}{2} \cos x_3}{m_p \frac{l^2}{4} + J} \end{bmatrix} \ddot{y}^*(t, p_1, p_2) \quad (16.6)$$

von (16.2) ergeben ein System zweiter Ordnung von gewöhnlichen Differentialgleichungen. Zusammen mit den Anfangs- und Endbedingungen

$$\begin{bmatrix} x_3 \\ x_4 \end{bmatrix}_{t=0} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}_{t=T} = \begin{bmatrix} \pi \\ 0 \end{bmatrix} \quad (16.7)$$

ist damit ein Randwertproblem formuliert. Da mit den Randwerten vier Bedingungen an die Lösung gestellt werden, müssen die Differentialgleichungen (16.6) noch über zwei freie Parameter verfügen. Dies sind hier p_1 und p_2 . (Womit allerdings die Lösbarkeit noch nicht garantiert ist. In Anhang 19 wird anhand von Beispielen kurz auf die Lösbarkeit von Randwertproblemen eingegangen.)

Bei der Lösung des Randwertproblems werden die Anfangs- und Endbedingungen der Zustände x_1 und x_2 (jeweils die ersten beiden Zeilen in (16.1)) nicht beachtet. Dies ist auch nicht nötig, da diese aufgrund der Konstruktion von $y^*(t, p_1, p_2)$ für beliebige p_1 und p_2 immer erfüllt sind: Dass $y^*(0, p_1, p_2) = \dot{y}^*(0, p_1, p_2) = 0$ ist, ist direkt zu sehen. Dass auch $y^*(T, p_1, p_2) = \dot{y}^*(T, p_1, p_2) = 0$ gilt, ist durch Berechnen der Ableitungen von (16.4) und Einsetzen zu überprüfen. Zudem gilt durch die gewählte Funktion $y^*(t, p_1, p_2)$ auch immer $\ddot{y}^*(0, p_1, p_2) = \ddot{y}^*(T, p_1, p_2) = 0$. Dies ist mathematisch nicht nötig, jedoch wird dadurch sichergestellt, dass die Eingangsgröße $u^*(t, p_1, p_2)$ stetig verläuft, d. h. bei $t = 0$ und $t = T$ keinen Sprung macht.

Wenn es also gelingt, eine Lösung für das durch (16.6) und (16.7) gegebene Randwertproblem zu finden, d. h. Parameter p_1 und p_2 zu finden, für die das Problem lösbar ist, ist damit durch (16.5) der Verlauf der Eingangsgröße $u^*(t)$ festgelegt, mit dem das Pendel von der unteren in die obere Gleichgewichtslage gebracht werden kann. MATLAB stellt mit der Funktion `bvp4c` einen numerischen Löser für genau dieses Problem zur Verfügung, welcher im folgenden Kapitel kurz vorgestellt wird.

Natürlich werden für diesen Versuch einige Vereinfachungen vorgenommen, die eine Realisierung einer solchen Steuerung am realen System deutlich erschweren. So wird hier die Reibung überhaupt nicht betrachtet. Insbesondere die Haftung stellt dabei eine Herausforderung dar. Auch werden Stellgrößen- und Zustandsbeschränkungen durch das beschriebene Entwurfsverfahren nicht direkt berücksichtigt.

17 MATLAB

Die den folgenden Befehlsbeschreibungen zugrundeliegenden Informationen sind, sofern nicht anders angegeben, der MATLAB-Hilfe [15] entnommen, auf die auch für weitere Informationen verwiesen wird. (Es sind zu den einzelnen Befehlen nicht alle Varianten von Argumenten und Rückgabewerten aufgeführt.)

17.1 Function-Handles

Mittels Function-Handles können indirekte Funktionsaufrufe gemacht werden. Beliebige Funktionen wie z. B. `sin`, `cos` oder auch selbstgeschriebene m-Functions können einer Variablen zugeordnet werden. Dazu muss ein „@“ vor den Funktionsnamen geschrieben werden. Die Variable (das Handle) kann an andere Funktionen weitergegeben werden, um dann mit Hilfe von ihr die durch dieses Handle „referenzierte“ Funktion aufzurufen. MATLAB-Funktionen wie `bvp4c` oder `ode...` erwarten als Eingangsparameter Function-Handles. Zur Veranschaulichung ein kleines Beispiel: Die Befehle

```
>> fhandle = @sin;  
>> x = -pi:0.01:pi;  
>> plot(x, fhandle(x))
```

erzeugen das Figure in Abbildung 17.1.

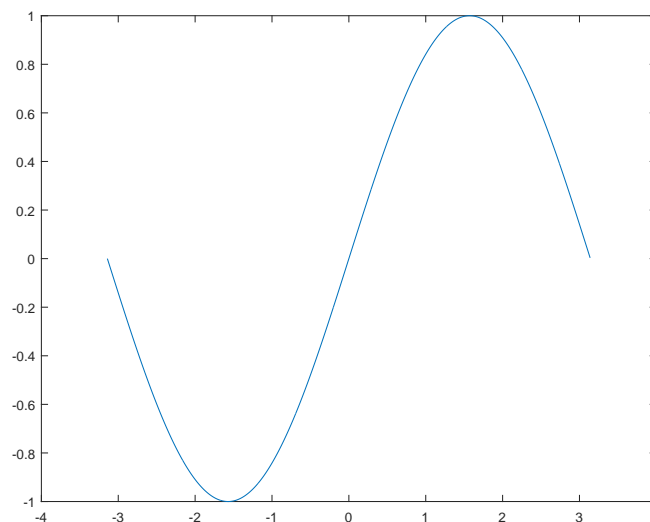


Abbildung 17.1: Figure zum Beispiel für Function-Handles (siehe Text)

Wenn eine MATLAB-Funktion ein Function-Handle erwartet, kann meist auch ein String mit dem Funktionsnamen angegeben werden. Zum Beispiel sind die beiden Aufrufe

```
>> ode(@Funktionsname)  
>> ode('Funktionsname')
```

gleichwertig. Das wirft daher ggfs. die Frage auf, worin der Vorteil der Handles begründet ist.

Zum einen sind Function-Handles wichtig, wenn man selbst Funktionen erstellen will, die Operationen auf beliebigen Funktionen ausführen sollen. Um dies zu verdeutlichen, dient folgendes Beispiel: Die Funktion `Integrator` implementiert einen numerischen Integrierer einfachster Art (Rechteckzerlegung). In diesem wird als erstes Argument ein Handle der zu integrierenden Funktion erwartet.

Listing 17.1: Beispielfunktion `Integrator`.

```
function int = Integrator(hFct, a, b, n)
    % hFct: Handle der zu integrierenden Funktion
    % a    : Linke Grenze
    % b    : Rechte Grenze
    % n    : Anzahl Intervalle für Zerlegung

    h = (b - a) / n;    % Schrittweite

    int = 0;

    for i = a:h:(b-h)
        int = int + hFct(i) * h;
    end % for i

end % function Integrator
```

Diese Funktion kann wie folgt verwendet werden:

```
>> Integrator(@sin, 0, 1, 1000)
ans =
    0.4593
>> Integrator(@exp, 0, 1, 1000)
ans =
    1.7174
```

(Bleibt noch anzumerken, dass MATLAB mit dem Befehl `integral` schon über einen numerischen Integrierer verfügt.)

Längere Skripte können mit Function-Handles auch übersichtlicher gestaltet werden, indem zu Beginn die zu verwendenden Funktionen festgelegt werden. Insbesondere wenn die Funktion an mehreren Stellen benötigt wird, ist diese Methode viel weniger fehleranfällig als das Ändern aller betroffenen Stellen im Programm per Hand.

Listing 17.2: Beispielskript zu Function-Handles.

```
hFct = @testfunktion1;
%hFct = @testfunktion2;
%hFct = @testfunktion3;

[... viel Code ...]

[t x] = ode23(hFct, [0 Tend], X0);
```

```
10 [... mehr Code ...]
[t2 x2] = ode23(hFct, [0 Tend], X02);
```

17.2 Der Löser bvp4c

MATLAB stellt mit der Funktion `bvp4c` einen Löser für Randwertaufgaben (zu Randwertaufgaben siehe auch Anhang 19), die durch ein DGL-System erster Ordnung

$$\mathbf{y}'(x, \mathbf{y}, \mathbf{p}) = \mathbf{f}(x, \mathbf{y}, \mathbf{p}) \quad (17.1)$$

und die Randbedingungen in der Form

$$\mathbf{g}(\mathbf{y}(a), \mathbf{y}(b), \mathbf{p}) = \mathbf{0} \quad (17.2)$$

gegeben sind, zur Verfügung. (Der Vektor der freien Parameter \mathbf{p} ist jeweils optional.) Der Funktionsname steht ungefähr für boundary value problem, 4th order collocation method und beschreibt die zugrundeliegende Lösungsmethode.

Vereinfacht gesagt wird dabei das Intervall $[a, b]$ von x in dem die Lösung von \mathbf{y} gesucht wird in Teilintervalle zerlegt (collocation, engl. für Anordnung, Nebeneinanderstellung). In jedem dieser Teilintervalle wird ein Polynom dritten Grades angenommen, welches am Mittelpunkt und den Randpunkten des Teilintervalls Gl. (17.1) erfüllen soll. Dadurch entsteht ein nichtlineares Gleichungssystem für die Koeffizienten der Polynome [17]. Interessierte finden eine genauere Beschreibung mit mehreren Beispielen in [17] (dieses Dokument ist auch in der MATLAB-Hilfe verlinkt).

Ein Randwertproblem ist durch die Angabe des DGL-Systems (17.1) und den dazugehörigen Randbedingungen (17.2) definiert und diese müssen dem Löser natürlich zur Verfügung gestellt werden. Zusätzlich erwartet der Löser Startwerte für die Intervallaufteilung für x sowie für $\mathbf{y}(x)$ an den Intervallpunkten und ggfs. für \mathbf{p} :

```
sol = bvp4c(odefun, bcfun, solinit)
```

Zusätzlich können im vierten Argument auch noch Optionen vorgegeben werden:

```
sol = bvp4c(odefun, bcfun, solinit, options)
```

- `odefun`: Handle auf eine Funktion, die die Implementation von Gl. (17.1) darstellt. Diese berechnet daher aus den übergebenen Werten für x , \mathbf{y} und ggfs. \mathbf{p} den Wert für \mathbf{y}' :

```
dydx = odefun(x, y)
dydx = odefun(x, y, parameters)
```
- `bcfun`: Handle auf eine Funktion, die dem Löser die Randbedingungen nach Gl. (17.2) „mitteilt“. Diese Funktion muss folgende Syntax besitzen:

```
res = bcfun(ya, yb)
res = bcfun(ya, yb, parameters)
```

Beim Aufruf werden dieser Funktion die kompletten Vektoren $\mathbf{y}(a)$ und $\mathbf{y}(b)$ vom linken und rechten Rand übergeben. Der Rückgabvektor `res` enthält die Residuen der Randbedingungen, d. h. die Differenz der Ist-Randwerte zu den Soll-Randwerten. (Die Reihenfolge spielt dabei keine Rolle.)
- `solinit` ist eine Struktur mit den Elementen `x`, `y` und (optional) `parameters`, die die Startwerte für die numerische Lösungssuche darstellen. `x` ist dabei ein Vektor, `y` und `parameters` sind Vektoren

bzw. Matrizen mit der gleichen Spaltenanzahl wie \mathbf{x} . Die Zeilenanzahl entspricht der Anzahl der Elemente von \mathbf{y} bzw. \mathbf{p} .

Wenn ein Startwert für die Parameter vorgegeben wird, dann **müssen** auch die beiden Funktionen `odefun` und `bcfun` über das dritte Argument verfügen, auch wenn diese Funktionen die Variable `parameters` gar nicht benutzen sollten.

MATLAB stellt die Funktion `bvpinit` zur Verfügung, um die Struktur `solinit` zu erzeugen, jedoch stellt diese keinen großen Vorteil gegenüber der „manuellen“ Erzeugung dar.

- `options`: Hierzu wird auf die MATLAB-Hilfe verwiesen.
- Rückgabewert `sol`: Die Struktur `sol` enthält die gefundene Lösung (sofern tatsächlich eine Lösung gefunden wurde) und ist wie `solinit` aufgebaut.

Folgendes kleines Beispiel ist der MATLAB-Hilfe entnommen und leicht abgewandelt: Es wird die Lösung der Differentialgleichung

$$y'' + |y| = 0$$

unter den Randbedingungen

$$y(0) = 0 \quad \text{und} \quad y(4) = -2$$

gesucht.

Umschreiben in die durch Gl. (17.1) und (17.2) gegebene Form ergibt mit

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} y \\ y' \end{bmatrix}$$

die Gleichungen

$$\mathbf{y}' = \begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} y_2 \\ -|y_1| \end{bmatrix}$$

und

$$\mathbf{g} = \begin{bmatrix} y_1(0) \\ y_1(4) + 2 \end{bmatrix} = \mathbf{0}.$$

Folgendes Listing zeigt die Funktion, die dieses Randwertproblem numerisch löst.

Listing 17.3: Funktion `sampleBVP`.

```
function sol = sampleBVP()

3   solinit.x = linspace(0, 4, 5);
   solinit.y = [linspace(0, -2, 5); zeros(1, 5)];

   sol = bvp4c(@twoode, @twobc, solinit);

8 end % function sampleBVP

% subfunction twoode
function dydx = twoode(x,y)
```

```

13     dydx = [ y(2); ...
              -abs(y(1))];
end % function twoode

% subfunction twobc
function res = twobc(ya,yb)
18     res = [ ya(1); ...
              yb(1) + 2];
end % function twobc

```

Dann kann mit

```
>> sol = sampleBVP();
```

eine Lösung des Randwertproblems gefunden werden.

```
>> plot(sol.x, sol.y(1,:))
```

plottet die Lösung $y(x)$ wie im Screenshot in Abbildung 17.2 gezeigt. (Dieses Randwertproblem hat noch eine zweite Lösung. Um diese zu finden, müssen andere Startwerte vorgegeben werden. Siehe dazu die MATLAB-Hilfe.)

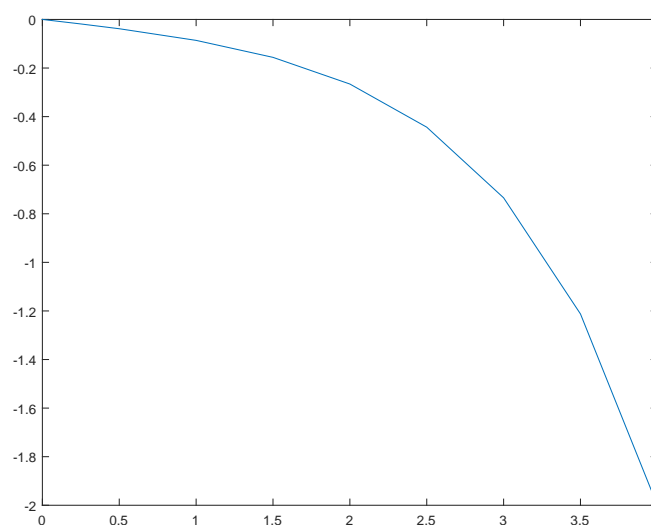


Abbildung 17.2: Figure zum Beispiel für `bvp4c` (siehe Text)

Wenn `bvp4c` keine Lösung findet (es wird in diesem Fall eine Warnung oder ein Fehler ausgegeben), kann das zum einen daran liegen, dass das Randwertproblem keine Lösung besitzt. Es kann aber auch sein, dass die Startwerte zu ungünstig lagen und/oder die maximale Intervallanzahl für die Zerlegung des Intervalls von x zu gering ist. Die maximale Intervallanzahl kann über `options` verändert werden. Es kann auch versucht werden, die „schlechte Lösung“ direkt als Startwert für einen neuen Durchlauf zu verwenden.

17.3 Parametrisieren von Funktionen

Wenn man eine Berechnung wie diese nicht nur für einen Satz von Systemparametern (z. B. Pendelmasse, -länge) durchführen möchte sondern für mehrere, dann müssten jedesmal die Werte dieser Parameter

in der entsprechenden odefun-Funktion (in der Gl. (16.6) ja mit den Systemparametern implementiert sein muss) geändert werden. Dies ist sehr aufwendig und fehleranfällig. In diesem Abschnitt werden zwei Möglichkeiten vorgestellt, Funktionen vor der Übergabe an weitere Funktionen (z. B. hier dem Löser bvp4c) zu parametrisieren.

17.3.1 Anonyme Funktionen

Anonyme Funktionen können verwendet werden, um einfache Funktionen zu definieren, ohne gleich eine m-Function erstellen zu müssen. Sie werden ähnlich definiert wie Function-Handles. Nach dem @-Zeichen stehen in Klammern die Argumente der anonymen Funktion. Direkt danach folgt die Rechenvorschrift. Dazu ein kleines Beispiel:

```
>> hABFct = @(a, b) a^2+a*b+b^2
hABFct =
    @(a,b)a^2+a*b+b^2
>> hABFct(2, 4)
ans =
    28
```

In diesem Beispiel wurden nur elementare Rechenoperationen in der anonymen Funktion ausgeführt. Es können natürlich auch andere Funktionen aufgerufen werden. Für das folgende Beispiel wird zunächst die einfache Funktion pow definiert:

Listing 17.4: Funktion pow.

```
function p = pow(a, b)

    p = a.^b;

5 end % function pow
```

Mit dieser Funktion kann nun bspw. eine anonyme Funktion definiert werden, die nur noch ein Argument erwartet und dieses immer hoch drei nimmt:

```
>> hFctPow = @(x) pow(x, 3);
>> hFctPow(4)
ans =
    64
```

Statt einer festen Zahl (wie hier „3“) kann auch eine Variable, die im aktuellen Workspace vorhanden ist, verwendet werden, wie z. B.

```
>> b = 5;
>> hFctPow = @(x) pow(x, b)
hFctPow =
    @(x)pow(x,b)
>> hFctPow(4)
ans =
    1024
```

Wichtig in diesem Zusammenhang ist, dass MATLAB zwar `b` in der Definition der anonymen Funktion `@(x)pow(x,b)` anzeigt, aber den Wert von `b` zum Zeitpunkt der Definition der anonymen Funktion meint. D. h., dass ein Ändern oder gar Löschen der Variablen `b` **keinen** Einfluss auf `hFctPow` mehr hat:

```
>> b = 10;
>> hFctPow(4)
ans =
    1024
```

Diese anonymen Funktionen stellen damit eine Möglichkeit dar, Funktionen vor der Übergabe an Löser zu parametrisieren.

17.3.2 Persistent Variablen

Persistent und globale Variablen haben Sie bereits im vorhergehenden Versuch kennengelernt.

Im Rahmen dieses Versuches sind vor allem die persistent Variablen interessant, da mit diesen eine Parametrisierung von Funktionen möglich ist. Dies ist im nächsten Beispiel gezeigt. In diesem wird noch der Befehl `nargin` (number of arguments, input) verwendet, der innerhalb einer MATLAB-Funktion die Anzahl der tatsächlich übergebenen Parameter zurückgibt.

Listing 17.5: Funktion offset.

```
function o = offset(v, initoffset)

    persistent s_offset;

5     if (nargin == 2)
        s_offset = initoffset; % Offsetwert speichern
        o = []; % Irgendeinen Ausgangswert setzen
        return; % Funktion verlassen
    end

10    o = v + s_offset;

end % function offset
```

Zunächst muss diese Funktion initialisiert werden, indem sie mit zwei Argumenten aufgerufen wird:

```
>> offset(0, 3)
ans =
     []
```

Im Folgenden kann diese Funktion mit nur einem Argument benutzt werden:

```
>> offset(6)
ans =
     9
```

Es ist zwar im Grunde möglich, die Parameter auch mit globalen Variablen zu „übergeben“, dies führt jedoch zu äußerst unübersichtlichen Programmen und daher ist von solch einem Vorgehen unbedingt abzuraten.

Welche dieser vorgestellten Möglichkeiten zur Parametrisierung verwendet wird, ist Geschmackssache. Der Weg über anonyme Funktionen dürfte in der Regel einfacher anzuwenden sein und darüber hinaus den Vorteil, dass es auch möglich ist, gleichzeitig mehrere unterschiedlich parametrisierte Varianten der selben Funktion zu erzeugen. Bei dem Weg mit persistent Variablen gelten immer die Parameter der letzten Initialisierung.

Das Beispiel `offset`, welches in diesem Abschnitt mit persistent-Variablen vorgestellt wurde, würde mit anonymen Funktionen wie folgt umgesetzt werden. Die Funktion `offsetA` verfügt nun nicht mehr über eine persistent Variable zum Speichern des Offset-Wertes, sondern erwartet grundsätzlich zwei Argumente.

Listing 17.6: Funktion `offsetA`.

```
function o = offsetA(v, offset)
2
    o = v + offset;

end % function offsetA
```

Anstelle der Initialisierung muss in diesem Fall eine anonyme Funktion erzeugt werden, die nur noch ein Argument erwartet und einen festen Offset von 3 vorgibt:

```
>> hOffsetFct = @(x) offsetA(x, 3)
hOffsetFct =
    @(x)offsetA(x,3)
```

Im Folgenden kann dann das Handle der anonymen Funktion verwendet werden:

```
>> hOffsetFct(6)
ans =
    9
```

17.4 Lookup-Tables

Lookup-Tables sind eine Möglichkeit in Simulink Kennfelder zu realisieren. In unserem Fall soll der Zusammenhang $u^*(t)$ durch eine (eindimensionale) Kennlinie beschrieben werden. Dabei sind für eine endliche Anzahl an Stützstellen t_j , die im Vektor `t` gespeichert sind, die Werte $u_j^* = u^*(t_j)$ im Vektor `u` gegeben. Mit dieser Variablenbenennung könnte das Lookup-Table wie in Abbildung 17.3 benutzt werden. (`t` und `u` müssen dabei im Workspace gespeichert sein.)

Besondere Beachtung verdient auch das Auswahlfeld „Look-up method“. In diesem kann die Interpolations- und Extrapolationsmethode vorgegeben werden. Letztere ist für den aktuellen Versuch dann interessant, wenn über einen längeren Zeitraum als die Übergangszeit T simuliert werden soll.

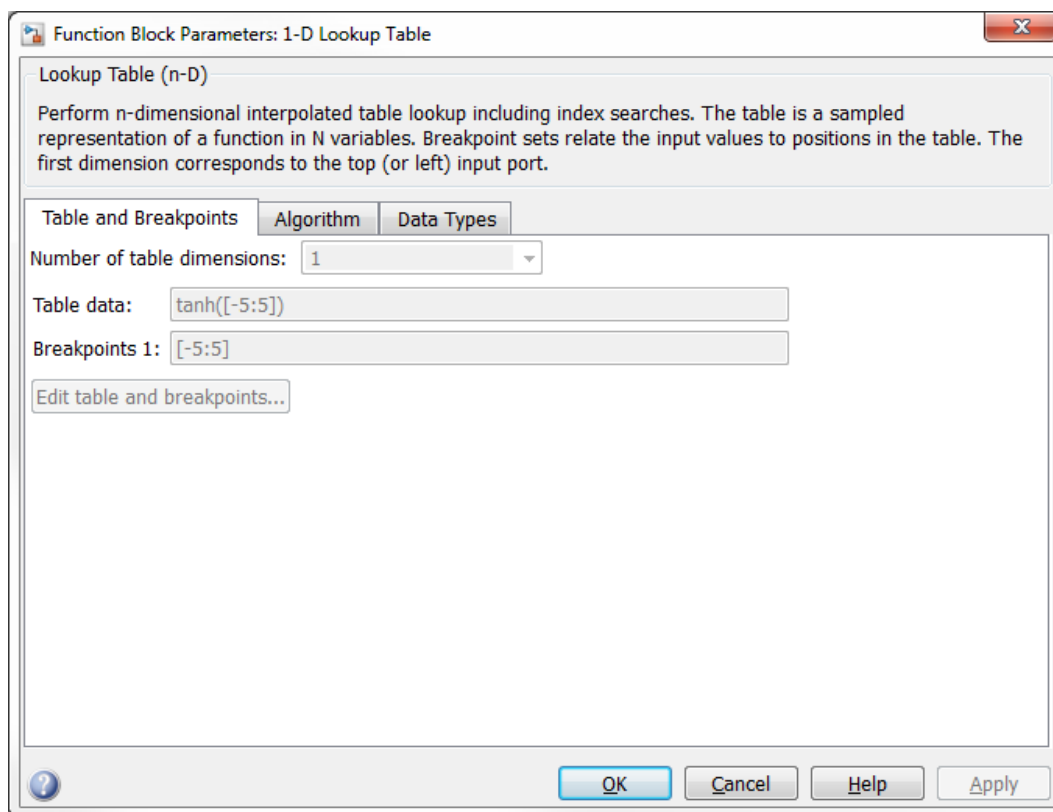


Abbildung 17.3: Parameter des „Lookup Table“-Blocks von Simulink.

18 Anhang – Herleitung der Steuerung

In diesem Kapitel wird das in [7] vorgestellte Verfahren zur Berechnung von Trajektorien vorgestellt. Bei der Darstellung wurden soweit wie möglich die Bezeichnungen nach [3] verwendet.

18.1 Systemdarstellung

Eine Trajektorie wird hier immer zwischen zwei Arbeitspunkten, also Gleichgewichtspunkten des Systems, berechnet. Es wird davon ausgegangen, dass die Trajektorie zum Zeitpunkt $t = 0$ beginnt und bei $t = T$ endet, dabei wird das System vom Zustand \mathbf{z}_0 in den Zustand \mathbf{z}_T überführt.

Der in diesem Versuch verwendete Ansatz zur Berechnung der Steuerung $u^*(t)$ eines nichtlinearen Systems geht von einer Systemdarstellung in der Form

$$\begin{bmatrix} \dot{z}_1 \\ \vdots \\ \dot{z}_{\delta-1} \\ \dot{z}_\delta \\ \dot{z}_{\delta+1} \\ \vdots \\ \dot{z}_n \end{bmatrix} = \begin{bmatrix} z_2 \\ \vdots \\ z_\delta \\ \alpha(\mathbf{z}) + \beta(\mathbf{z}) \cdot u \\ \tilde{q}_{\delta+1}(\mathbf{z}, u) \\ \vdots \\ \tilde{q}_n(\mathbf{z}, u) \end{bmatrix} \quad (18.1)$$
$$y = z_1 \quad (18.2)$$

aus, in der $\mathbf{z} = [z_1, \dots, z_n]^T$ der n -dimensionale Zustandsvektor des Systems ist. Im Folgenden werden wie in [3] die ersten δ Zeilen des DGL-Systems (18.1) als externe und die verbleibenden Zeilen als interne Dynamik bezeichnet. δ selber ist die sogenannte Differenzenordnung des Systems.

Diese Form ist ggfs. schon aus RT3 bekannt. In der RT3-Vorlesung bzw. im Skript zu dieser Vorlesung ist auch beschrieben, wie ein nichtlineares System allgemein in diese Form gebracht werden kann. Darauf wird im Rahmen dieses Praktikums nicht eingegangen, da das hier behandelte System direkt aus der Modellbildung in dieser Form vorliegt.

Bei der Berechnung einer Steuerung geht es darum, zu einem gegebenen Verlauf des Systemausgangs den nötigen Systemeingang zu berechnen.

Ist ein System in der Form (18.1) und (18.2) gegeben, dann entsprechen die ersten δ Zustände dem Systemausgang $y(t)$ und dessen ersten $\delta - 1$ Ableitungen. Der Zustandsvektor kann also auch als

$$\mathbf{z} = [y, \dot{y}, \dots, y^{(\delta-1)}, z_{\delta+1}, \dots, z_n]^T$$

geschrieben werden, wobei die Zustände $z_{\delta+1}, \dots, z_n$ im Weiteren zum Vektor $\boldsymbol{\eta}$

$$\boldsymbol{\eta} = [z_{\delta+1}, \dots, z_n]^T = [\eta_1, \dots, \eta_{n-\delta}]^T$$

zusammengefasst (und entsprechend in $\eta_1, \dots, \eta_{n-\delta}$ umbenannt) werden. Damit kann der Zustandsvektor \mathbf{z} auch als

$$\mathbf{z} = [y, \dot{y}, \dots, y^{(\delta-1)}, \boldsymbol{\eta}^T]^T$$

geschrieben werden.

Somit lässt sich die externe Dynamik mit der δ -ten Zeile von (18.1) durch

$$y^{(\delta)} = \alpha(y, \dot{y}, \dots, y^{(\delta-1)}, \boldsymbol{\eta}) + \beta(y, \dot{y}, \dots, y^{(\delta-1)}, \boldsymbol{\eta}) \cdot u \quad (18.3)$$

beschreiben. Für die interne Dynamik ergibt sich das DGL-System

$$\begin{bmatrix} \dot{\eta}_1 \\ \vdots \\ \dot{\eta}_{n-\delta} \end{bmatrix} = \begin{bmatrix} \tilde{q}_{\delta+1}(y, \dot{y}, \dots, y^{(\delta-1)}, \boldsymbol{\eta}, u) \\ \vdots \\ \tilde{q}_n(y, \dot{y}, \dots, y^{(\delta-1)}, \boldsymbol{\eta}, u) \end{bmatrix}. \quad (18.4)$$

18.2 Anfangs- und Endzustand, Randbedingungen für $y^*(t)$

Allgemein gilt für eine Ruhelage eines dynamischen Systems, dass die Ableitung des Zustandsvektors verschwinden muss:

$$\dot{\mathbf{z}}_{\mathbf{z}=\mathbf{z}_0, u=u_0} = \mathbf{0}.$$

Im Folgenden wird der Ausgangszustand \mathbf{z}_0 betrachtet, also die Ruhelage bei $t = 0$. Liegt ein System in der beschriebenen Form vor, dann folgt aus den ersten $\delta - 1$ Zeilen von (18.1) mit (18.2), dass für den Verlauf der Ausgangsgröße $y(t)$ an der Stelle $t = 0$

$$\begin{aligned} z_{0,1} &= y(0) = y_0 \\ z_{0,2} &= \dot{y}(0) = 0 \\ &\vdots \\ z_{0,\delta} &= y^{(\delta-1)}(0) = 0 \end{aligned}$$

gelten muss. Für $u(t = 0) = u_0$ erhält man aus (18.3)

$$u_0 = -\frac{\alpha(y_0, 0, \dots, 0, \boldsymbol{\eta}_0)}{\beta(y_0, 0, \dots, 0, \boldsymbol{\eta}_0)}$$

und letztlich

$$\begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{q}_{\delta+1}(y_0, 0, \dots, 0, \boldsymbol{\eta}_0, u_0) \\ \vdots \\ \tilde{q}_n(y_0, 0, \dots, 0, \boldsymbol{\eta}_0, u_0) \end{bmatrix}. \quad (18.5)$$

Entsprechend lässt sich die Ruhelage \mathbf{z}_T berechnen, die durch die Größen y_T , u_T und $\boldsymbol{\eta}_T$ bestimmt ist. (Für das vorliegende System siehe Abschnitt 18.9.2.)

Aus der Vorgabe, dass \mathbf{z}_0 und \mathbf{z}_T jeweils eine Ruhelage darstellen sollen, lassen sich Randbedingungen für die vorzugebende Trajektorie $y^*(t)$ ableiten. Diese stellen eine erste Einschränkung an $y^*(t)$ dar:

Bei $t = 0$ muss

$$y^*(0) = y_0, \quad \dot{y}^*(0) = 0, \quad \dots, \quad y^{*(\delta-1)}(0) = 0$$

gelten, und bei $t = T$ entsprechend

$$y^*(T) = y_T, \quad \dot{y}^*(T) = 0, \quad \dots, \quad y^{*(\delta-1)}(T) = 0.$$

Zudem kann noch gefordert werden, dass auch die δ -te Ableitung von $y^*(t)$ verschwindet. Damit wird hier erreicht, dass die Eingangsgröße bei $t = 0$ und $t = T$ stetig ist. Somit ergibt sich letztendlich

$$y^*(0) = y_0, \quad \dot{y}^*(0) = 0, \quad \dots, \quad y^{*(\delta)}(0) = 0 \quad (18.6)$$

und

$$y^*(T) = y_T, \quad \dot{y}^*(T) = 0, \quad \dots, \quad y^{*(\delta)}(T) = 0. \quad (18.7)$$

Aus der internen Dynamik folgen noch weitere Einschränkungen an mögliche Trajektorien $y^*(t)$, auf die weiter unten eingegangen wird.

18.3 Ansatzfunktion $\hat{y}^*(t)$

Zunächst wird nun ein Ausgangsgrößenverlauf $\hat{y}^*(t)$ vorgegeben, die die insgesamt $2\delta + 2$ Randbedingungen (18.6) und (18.7) erfüllt. Ein Ansatz für einen solchen Verlauf wäre beispielsweise ein Polynom vom Grade $2\delta + 1$:

$$\hat{y}^*(t) = a_0 + a_1 \cdot \left(\frac{t}{T}\right) + a_2 \cdot \left(\frac{t}{T}\right)^2 + \dots + a_{2\delta+1} \cdot \left(\frac{t}{T}\right)^{(2\delta+1)} \quad (18.8)$$

Solch ein Polynom verfügt über $2\delta + 2$ Koeffizienten a_i , mit denen die $2\delta + 2$ Randbedingungen erfüllt werden können.

Für das hier vorliegende System ergibt sich mit diesem Ansatz die in Abschnitt 18.9.3 angegebene Lösung.

18.4 Berechnung bei flachem Ausgang

Für den Fall, dass die Differenzenordnung δ des Systems gleich der Systemordnung n ist, wird die Berechnung der Trajektorie einfach. In diesem Fall kann für die Ausgangstrajektorie ein beliebiger Verlauf $y^*(t)$ vorgegeben werden, der lediglich n -mal differenzierbar sein und noch die Randbedingungen (18.6) und (18.7) erfüllen muss, wie z. B. $\hat{y}^*(t)$ nach (18.8).

Der Zustandsvektor \mathbf{z} entspricht für $\delta = n$ gerade

$$\mathbf{z} = (y^*, \dot{y}^*, \dots, y^{*(n-1)})^T$$

und in der letzten Zeile von Gl. (18.1) ($\delta = n!$) steht damit

$$y^{*(n)} = \alpha(y^*, \dot{y}^*, \dots, y^{*(n-1)}) + \beta(y^*, \dot{y}^*, \dots, y^{*(n-1)}) \cdot u^*.$$

Da $y^*(t)$ vorgegeben ist, sind sämtliche Ableitungen von $y^*(t)$ ebenfalls bekannt. Somit ergibt sich für die Steuerung

$$u^*(t) = \frac{y^{*(n)} - \alpha(y^*, \dot{y}^*, \dots, y^{*(n-1)})}{\beta(y^*, \dot{y}^*, \dots, y^{*(n-1)})}.$$

Der Ausgang $y = f(\mathbf{z})$ für den $\delta = n$ gilt wird auch als flacher Ausgang bezeichnet.

Das zu behandelnde System liegt jedoch nicht in dieser flachen Form vor, so dass die Berechnung der Trajektorie etwas komplizierter wird. So kann der Verlauf der Ausgangsgröße nicht beliebig vorgegeben werden, sondern dieser ermittelt sich zum Teil aus der internen Dynamik.

18.5 Berechnung bei nicht-flachem Ausgang

Im Fall eines nicht-flachen Ausgangs (der in diesem Praktikum vorliegt) muss auch die interne Dynamik (18.4) beachtet werden, die zusätzliche Forderungen an $y^*(t)$ stellt.

Diese stellt ein Differentialgleichungssystem der Ordnung $n - \delta$ dar. Da mit den Randwerten

$$\boldsymbol{\eta}_0 = \begin{bmatrix} \eta_{0,1} \\ \vdots \\ \eta_{0,n-\delta} \end{bmatrix} \quad (18.9)$$

und

$$\boldsymbol{\eta}_T = \begin{bmatrix} \eta_{T,1} \\ \vdots \\ \eta_{T,n-\delta} \end{bmatrix}, \quad (18.10)$$

die durch die Gleichgewichtsbedingungen bei $t = 0$ und $t = T$ vorgegeben sind, nicht nur Anfangs- oder Endwerte, sondern Anfangs- und Endwerte gegeben sind, handelt es sich hier um ein Randwertproblem. Damit dieses lösbar ist, müssen noch $n - \delta$ Parameter zur Verfügung gestellt werden (siehe Kapitel 19).

Dies wird hier durch eine Funktion $\tilde{y}^*(t, p_1, \dots, p_{n-\delta})$ erreicht, die zu der schon bestimmten Funktion $\hat{y}^*(t)$ addiert wird und mit $p_1, \dots, p_{n-\delta}$ über die geforderten Parameter verfügt.

Es ergibt sich also als Ansatz für den Verlauf $y^*(t)$ der Ausgangsgröße

$$y^*(t, p_1, \dots, p_{n-\delta}) = \hat{y}^*(t) + \tilde{y}^*(t, p_1, \dots, p_{n-\delta}). \quad (18.11)$$

Bei der Konstruktion von $\tilde{y}^*(t, \mathbf{p})$ – die Parameter $p_1, \dots, p_{n-\delta}$ werden im Folgenden häufig zum Vektor \mathbf{p} zusammengefasst – muss darauf geachtet werden, dass unabhängig der Wahl der Parameter \mathbf{p} die homogenen Randbedingungen

$$\tilde{y}^*(0, \mathbf{p}) = 0, \quad \dot{\tilde{y}}^*(0, \mathbf{p}) = 0, \quad \dots, \quad \tilde{y}^{*(\delta)}(0, \mathbf{p}) = 0 \quad (18.12)$$

und

$$\tilde{y}^*(T, \mathbf{p}) = 0, \quad \dot{\tilde{y}}^*(T, \mathbf{p}) = 0, \quad \dots, \quad \tilde{y}^{*(\delta)}(T, \mathbf{p}) = 0 \quad (18.13)$$

der externen Dynamik eingehalten werden. (Diese unterscheiden sich von (18.6) und (18.7) dadurch, dass auch die „nullte Ableitung“ gleich Null sein muss. Damit erfüllt die Gesamtfunktion $y^*(t)$ nach (18.11) die Randbedingungen (18.6) und (18.7).)

18.6 Erweiterung der Ansatzfunktion $\tilde{y}^*(t)$

Eine solche Funktion lässt sich konstruieren, indem man den Polynomansatz weiterführt. Dabei sind aber nicht nur $n - \delta$ Terme

$$p_1 \cdot \left(\frac{t}{T}\right)^{2\delta+2} + \dots + p_{n-\delta} \cdot \left(\frac{t}{T}\right)^{n+\delta+1}$$

hinzuzunehmen, sondern auch noch $2\delta + 1$ Terme mit denen erreicht werden kann, dass für beliebige \mathbf{p} die Bedingungen (18.12) und (18.13) erfüllt sind:

$$\tilde{y}^*(t, \mathbf{p}) = \tilde{a}_0(\mathbf{p}) + \tilde{a}_1(\mathbf{p}) \cdot \left(\frac{t}{T}\right) + \dots + \tilde{a}_{2\delta+1}(\mathbf{p}) \cdot \left(\frac{t}{T}\right)^{(2\delta+1)} + p_1 \cdot \left(\frac{t}{T}\right)^{2\delta+2} + \dots + p_{n-\delta} \cdot \left(\frac{t}{T}\right)^{n+\delta+1} \quad (18.14)$$

Die Parameter $\tilde{a}_0(\mathbf{p}), \dots, \tilde{a}_{2\delta+1}(\mathbf{p})$ werden dabei über die Randbedingungen (18.12) und (18.13) bestimmt und hängen im Allgemeinen von den Parametern \mathbf{p} ab.

Für das Beispielsystem ist die Berechnung in Abschnitt 18.9.4 gezeigt.

18.7 Berechnung der Parameter \mathbf{p}

Zusammenfassend hat man bis jetzt einen Ansatz für die Trajektorie, der von Parametern abhängt und der garantiert die Randbedingungen der externen Dynamik erfüllt. Damit braucht die externe Dynamik bei der folgenden Lösung des DGL-Systems für die interne Dynamik nicht mehr beachtet werden.

Für die Stellgröße ergibt sich durch Einsetzen von $y^*(t, \mathbf{p})$ und dessen Ableitungen in (18.3) die Gleichung

$$u^*(t, \mathbf{p}) = \frac{y^{*(\delta)}(t, \mathbf{p}) - \alpha(y^*(t, \mathbf{p}), \dot{y}^*(t, \mathbf{p}), \dots, y^{*(\delta-1)}(t, \mathbf{p}))}{\beta(y^*(t, \mathbf{p}), \dot{y}^*(t, \mathbf{p}), \dots, y^{*(\delta-1)}(t, \mathbf{p}))}.$$

Dieses $u^*(t, \mathbf{p})$ sowie $y^*(t, \mathbf{p})$ und dessen Ableitungen können jetzt in das DGL-System (18.4) für die interne Dynamik eingesetzt werden. Man erhält auf diese Weise das DGL-System

$$\begin{aligned} \dot{\eta}_1 &= \tilde{q}_{\delta+1}(\boldsymbol{\eta}, t, \mathbf{p}) \\ &\vdots \\ \dot{\eta}_{n-\delta} &= \tilde{q}_n(\boldsymbol{\eta}, t, \mathbf{p}) \end{aligned}$$

mit den Randbedingungen (18.9) und (18.10).

Das Problem der Bestimmung einer Trajektorie für ein nichtlineares System ist somit auf das Finden einer Lösung eines nichtlinearen DGL-Systems mit Randbedingungen und zusätzlichen Parametern zurückgeführt. Für diese Aufgabe stellt MATLAB die Funktion `bvp4c` zur Verfügung.

Als letzten Schritt werden dann die mit diesen Verfahren bestimmten Parameter \mathbf{p} in die Gleichung (16.5) eingesetzt. Damit ist der Verlauf der Stellgröße $u^*(t)$ bekannt, die auf das System gegeben werden muss, um das Pendel aufschwingen zu lassen.

Die praktische Umsetzung der in diesem Abschnitt vorgestellten Schritte ist Bestandteil des Praktikums.

18.8 Anmerkungen

Die auffindbaren Lösungen hängen natürlich von der mehr oder weniger willkürlich gewählten Ansatzfunktion für $y^*(t)$ ab. (Der vorgestellte Polynomansatz stellt nur eine Möglichkeit dar, eine solche Ansatzfunktion zu wählen.)

Bei ungünstiger Wahl dieser Funktion kann es durchaus sein, dass keine Lösung gefunden wird. Auch wird in diesem Versuch keine optimale Lösung gesucht.

18.9 Anwendung auf Beispielsystem

18.9.1 Systemdarstellung

Aus den gegebenen Systemgleichungen (16.2) und (16.3) lässt sich durch Vergleich mit (18.1) und (18.2) für die System- und Differenzenordnung die Werte

$$n = 4$$

$$\delta = 2$$

ablesen.

18.9.2 Randbedingungen

In dem betrachteten Beispiel soll der Aufschwung des Pendels berechnet werden. Der Schlitten soll sich zum Beginn und zum Ende des Aufschwingvorgangs in der Mitte ($y = 0$) befinden. Damit ergibt sich als Ausgangsruhelage

$$y_0 = 0 \quad (18.15)$$

$$\boldsymbol{\eta}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (18.16)$$

$$u_0 = 0$$

und als Endruhelage

$$y_T = 0 \quad (18.17)$$

$$\boldsymbol{\eta}_T = \begin{bmatrix} \pi \\ 0 \end{bmatrix} \quad (18.18)$$

$$u_T = 0.$$

Die Werte für y_0 , $\boldsymbol{\eta}_0$, y_T und $\boldsymbol{\eta}_T$ sind auch schon in (16.1) angegeben.

18.9.3 Ansatzfunktion $\hat{y}^*(t)$

Nach dem vorgestellten Verfahren ergibt sich in diesem Fall die äußerst einfache Funktion

$$\hat{y}^*(t) = 0.$$

18.9.4 Erweiterung der Ansatzfunktion $\tilde{y}^*(t, \mathbf{p})$

In diesem Fall erhält man mit $n = 4$ und $\delta = 2$ folgenden Ansatz für $\tilde{y}^*(t, \mathbf{p})$:

$$\begin{aligned} \tilde{y}^*(t, p_1, p_2) = & \tilde{a}_0(p_1, p_2) + \tilde{a}_1(p_1, p_2) \cdot \left(\frac{t}{T}\right) + \tilde{a}_2(p_1, p_2) \cdot \left(\frac{t}{T}\right)^2 + \tilde{a}_3(p_1, p_2) \cdot \left(\frac{t}{T}\right)^3 + \\ & \tilde{a}_4(p_1, p_2) \cdot \left(\frac{t}{T}\right)^4 + \tilde{a}_5(p_1, p_2) \cdot \left(\frac{t}{T}\right)^5 + p_1 \cdot \left(\frac{t}{T}\right)^6 + p_2 \cdot \left(\frac{t}{T}\right)^7. \end{aligned}$$

Die zu erfüllenden Randbedingungen lauten

$$\tilde{y}^*(0, p_1, p_2) = 0, \quad \dot{\tilde{y}}^*(0, p_1, p_2) = 0, \quad \ddot{\tilde{y}}^*(0, p_1, p_2) = 0$$

und

$$\tilde{y}^*(T, p_1, p_2) = 0, \quad \dot{\tilde{y}}^*(T, p_1, p_2) = 0, \quad \ddot{\tilde{y}}^*(T, p_1, p_2) = 0.$$

Es müssen also die ersten beiden Ableitungen von $\tilde{y}^*(t, p_1, p_2)$ gebildet werden:

$$\begin{aligned} \dot{\tilde{y}}(t, p_1, p_2) = & \\ & \left(\tilde{a}_1 + 2\tilde{a}_2 \cdot \left(\frac{t}{T}\right) + 3\tilde{a}_3 \cdot \left(\frac{t}{T}\right)^2 + 4\tilde{a}_4 \cdot \left(\frac{t}{T}\right)^3 + 5\tilde{a}_5 \cdot \left(\frac{t}{T}\right)^4 + 6p_1 \cdot \left(\frac{t}{T}\right)^5 + 7p_2 \cdot \left(\frac{t}{T}\right)^6 \right) \cdot \frac{1}{T} \end{aligned}$$

$$\begin{aligned} \ddot{\tilde{y}}(t, p_1, p_2) = & \\ & \left(2\tilde{a}_2 + 6\tilde{a}_3 \cdot \left(\frac{t}{T}\right) + 12\tilde{a}_4 \cdot \left(\frac{t}{T}\right)^2 + 20\tilde{a}_5 \cdot \left(\frac{t}{T}\right)^3 + 30p_1 \cdot \left(\frac{t}{T}\right)^4 + 42p_2 \cdot \left(\frac{t}{T}\right)^5 \right) \cdot \frac{1}{T^2}. \end{aligned}$$

Mit den Randbedingungen bei $t = 0$ folgt sofort, dass

$$\tilde{a}_0 = \tilde{a}_1 = \tilde{a}_2 = 0.$$

Aus den Randbedingungen bei $t = T$ entsteht das lineare Gleichungssystem

$$\begin{array}{rrrrr} \tilde{a}_3 + & \tilde{a}_4 + & \tilde{a}_5 + & p_1 + & p_2 & = & 0 \\ \tilde{3a}_3 + & 4\tilde{a}_4 + & 5\tilde{a}_5 + & 6p_1 + & 7p_2 & = & 0 \\ \tilde{6a}_3 + & 12\tilde{a}_4 + & 20\tilde{a}_5 + & 30p_1 + & 42p_2 & = & 0 \end{array}$$

aus welchem man

$$\begin{aligned} \tilde{a}_3 &= -p_1 - 3p_2 \\ \tilde{a}_4 &= 3p_1 + 8p_2 \\ \tilde{a}_5 &= -3p_1 - 6p_2 \end{aligned}$$

erhält. Werden diese Koeffizienten wieder in die Ansatzfunktion für $\tilde{y}^*(t, \mathbf{p})$ ($= y^*(t, \mathbf{p})$) eingesetzt, ergibt sich Gleichung (16.4).

19 Anhang – Randwertprobleme

19.1 Lösbarkeit von Randwertproblemen

Die allgemeine Lösung einer Differentialgleichung n -ter Ordnung verfügt über n Konstanten, die über weitere Bedingungen bestimmt werden können.

So hat die Differentialgleichung zweiter Ordnung

$$\ddot{y} + y = 0$$

die allgemeine Lösung

$$y(t) = c_1 \cos t + c_2 \sin t$$

mit den Konstanten c_1 und c_2 .

Sind neben der Differentialgleichung zwei Anfangsbedingungen, d.h. zwei Bedingungen zum *selben* (aber sonst beliebigen) $t = t_0$, gegeben, so lassen sich die beiden Konstanten eindeutig bestimmen. Mit den Anfangsbedingungen

$$y(t_0) = y_0, \quad \dot{y}(t_0) = v_0$$

ergäbe sich mit $t_0 = 0$ beispielsweise

$$c_1 = y_0, \quad c_2 = v_0.$$

Sind Bedingungen zu *verschiedenen* t gegeben, so ist die Lösbarkeit des Problems nicht garantiert, wie folgende Beispiele zeigen. So kann es eine

- eindeutige Lösung,

$$y(0) = 0, y(1) = 1 \Rightarrow c_1 = 0, c_2 = \frac{1}{\sin 1}$$

- unendlich viele Lösungen oder

$$y(0) = 1, y(\pi) = -1 \Rightarrow c_1 = 1, c_2 \in \mathbb{R}$$

- keine Lösung geben [5].

$$y(0) = 1, y(\pi) = 0 \Rightarrow \text{keine Lösung}$$

19.2 Geringere DGL-Ordnung als Anzahl der Randbedingungen

Sind mehr Bedingungen vorgegeben, als die Ordnung der Differentialgleichung ist, so erhält man mehr Bestimmungsgleichungen für die Konstanten als Parameter, also ein überbestimmtes Gleichungssystem. Damit sind die Randbedingungen nicht unabhängig voneinander vorgebar.

Gibt man beispielsweise zur Differentialgleichung aus dem obigen Beispiel die Bedingungen

$$y(0) = 0 \quad (19.1)$$

$$\dot{y}(0) = v_0 \quad (19.2)$$

$$y(1) = s_1 \quad (19.3)$$

$$\dot{y}(1) = v_1 \quad (19.4)$$

an, so erhält man die Bestimmungsgleichungen

$$0 = c_1$$

$$v_0 = c_2$$

$$s_1 = c_1 \cos 1 + c_2 \sin 1$$

$$v_1 = -c_1 \sin 1 + c_2 \cos 1 .$$

Daraus lässt sich ablesen, dass eine Lösung nur existiert wenn $s_1 = v_0 \sin 1$ und $v_1 = v_0 \cos 1$ ist. Es sind also effektiv nur zwei Bedingungen vorgebar.

Damit die Bestimmungsgleichungen genauso viele Unbekannte wie Gleichungen besitzen, müsste die Differentialgleichung noch über zwei freie Parameter verfügen. Eine solche Gleichung wäre beispielsweise

$$\frac{1}{p_1^2} \ddot{y} + y = p_2 .$$

Die allgemeine Lösung und deren Ableitungen lauten in diesem Fall

$$y(t) = c_1 \cos p_1 t + c_2 \sin p_1 t + p_2$$

$$\dot{y}(t) = -c_1 p_1 \sin p_1 t + c_2 p_1 \cos p_1 t$$

$$\ddot{y}(t) = -c_1 p_1^2 \cos p_1 t - c_2 p_1^2 \sin p_1 t .$$

Damit erhält man mit den oben angegebenen Randbedingungen (19.1) bis (19.4) ein System von vier (nichtlinearen) Gleichungen mit vier Unbekannten:

$$0 = c_1 + p_2$$

$$v_0 = c_2 p_1$$

$$s_1 = c_1 \cos p_1 + c_2 \sin p_1 + p_2$$

$$v_1 = -c_1 p_1 \sin p_1 + c_2 p_1 \cos p_1 .$$

Über die Anzahl der Lösungen kann keine direkte Aussage gemacht werden, sondern diese müssten in Abhängigkeit der Werte für v_0 , s_0 und v_1 numerisch bestimmt werden.

Für die Randbedingungen $v_0 = 0$, $s_1 = 1$, $v_1 = 0$ ist das gegebene Problem beispielsweise nicht lösbar, für $v_0 = 0$, $s_1 = 1$, $v_1 = 1$ wäre eine Lösung (von unendlich vielen) $c_1 = -p_2$, $c_2 = 0$, $p_1 = 2,3311$ und $p_2 = 0,5920$.

Zusammenfassend lässt sich aus den Beispielen dieses Abschnitts zum einen ableiten, dass bei (auch linearen) Randwertproblemen die Lösbarkeit nicht garantiert ist. Wenn eine Lösung gefunden wurde, so ist deren Eindeutigkeit nicht garantiert.

Wenn mehr Randbedingungen als die Ordnung der DGL vorgegeben werden sollen, dann ist dies nur dann sinnvoll möglich, wenn die DGL selber noch über so viele freie Parameter verfügt, dass die Anzahl der freien Parameter plus der DGL-Ordnung der Anzahl der Randbedingungen entspricht. Auch in diesem Fall ist die Lösbarkeit bzw. Eindeutigkeit der Lösung nicht garantiert.

Versuch 6

Trajektorienfolgeregelung

Im Versuch 5 wurde eine Steuerung entworfen, die das Pendel von der unteren in die obere Ruhelage überführen soll. Da eine Steuerung jedoch über keine Rückführung verfügt, ist sie anfällig gegenüber Modellierungsungenauigkeiten und Störungen. Auch zeigt sich, dass aufgrund der numerischen Fehler das Pendel in der Simulation selbst ohne die erwähnten Ungenauigkeiten nach Erreichen der oberen Lage wieder zurückschwingt.

In diesem (letzten) Versuch des Praktikum MATLAB/Simulink II soll eine Trajektorienfolgeregelung für das System entworfen werden, so dass das Pendel die obere Ruhelage auch bei Parameterunsicherheiten erreicht und beibehält.

| | |
|--|------------|
| 20 Trajektorienfolgeregelung | 100 |
| 20.1 Linearisierung um eine Trajektorie | 100 |
| 20.2 LQ-Reglerentwurf | 101 |
| 21 MATLAB | 103 |
| 21.1 Lösen der algebraischen RICCATIgleichung | 103 |
| 21.2 Lösen der differentiellen RICCATIgleichung | 103 |
| 21.3 Aufbau der Regelung unter Simulink | 105 |
| 22 Anhang – Zur Herleitung des LQ-Reglers für LZV-Systeme | 106 |

20 Trajektorienfolgeregelung

Dieser Abschnitt schließt direkt an die Einführung des letzten Versuches an, in der der grundsätzliche Aufbau einer Steuerung mit Trajektorienfolgeregelung erläutert wurde. Abbildung 20.1 zeigt noch einmal diese Struktur.

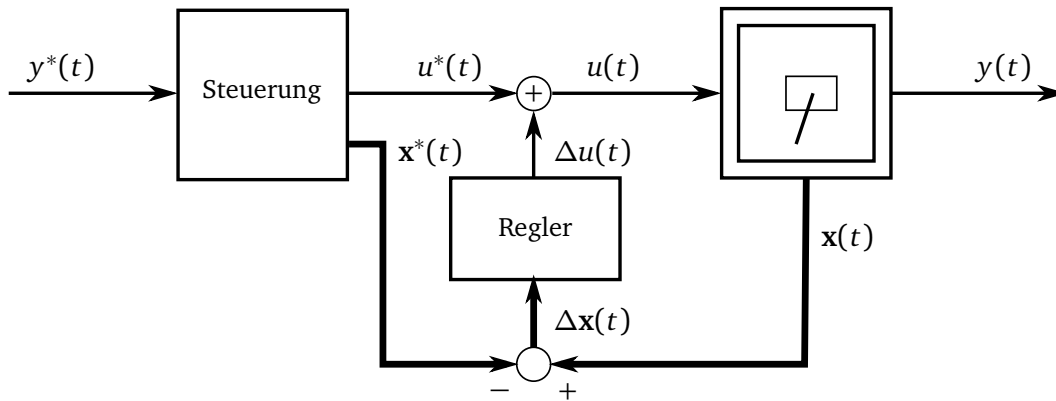


Abbildung 20.1: Steuerung mit Trajektorienfolgeregelung

20.1 Linearisierung um eine Trajektorie

Wenn davon ausgegangen wird, dass zu jedem Zeitpunkt t die Abweichungen des tatsächlichen Zustandes $\mathbf{x}(t)$ zu dem durch die Trajektorie vorgegebenen Sollzustand $\mathbf{x}^*(t)$ gering ist, kann der Ansatz verfolgt werden, das System um die Trajektorie zu linearisieren und dann einen Regler für das lineare System zu entwerfen.

Im Gegensatz zu der aus den Grundvorlesungen bekannten Linearisierung um einen festen Arbeitspunkt, bei der kleine Auslenkungen $\Delta \mathbf{x}(t)$ um den Zustand \mathbf{x}_0 am Arbeitspunkt betrachtet werden, werden bei der Linearisierung um eine Trajektorie kleine Abweichungen um die Trajektorie betrachtet:

$$\begin{aligned}\mathbf{x}(t) &= \mathbf{x}^*(t) + \Delta \mathbf{x}(t) \\ \mathbf{u}(t) &= \mathbf{u}^*(t) + \Delta \mathbf{u}(t) .\end{aligned}$$

Setzt man diese Ausdrücke für $\mathbf{x}(t)$ und $\mathbf{u}(t)$ in die allgemeine Zustandsgleichung $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u})$ ein, so erhält man

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}^*(t) + \Delta \mathbf{x}(t), \mathbf{u}^*(t) + \Delta \mathbf{u}(t)) \\ &= \mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t)) + \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}^*(t), \mathbf{u}^*(t)} \cdot \Delta \mathbf{x}(t) + \left. \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right|_{\mathbf{x}^*(t), \mathbf{u}^*(t)} \cdot \Delta \mathbf{u}(t) + R ,\end{aligned}$$

wobei eine Taylorreihenentwicklung angewandt wurde und R die Terme höherer Ordnung bezeichnet. Vernachlässigt man diese Terme höherer Ordnung und bezeichnet die Differenz zwischen tatsächlicher Beschleunigung $\dot{\mathbf{x}}(t)$ und der Sollbeschleunigung $\dot{\mathbf{x}}^*(t) = \mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t))$ mit

$$\Delta \dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(t) - \mathbf{f}(\mathbf{x}^*(t), \mathbf{u}^*(t)) ,$$

dann ergibt sich

$$\Delta \dot{\mathbf{x}}(t) = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}^*(t), \mathbf{u}^*(t)} \cdot \Delta \mathbf{x}(t) + \left. \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right|_{\mathbf{x}^*(t), \mathbf{u}^*(t)} \cdot \Delta \mathbf{u}(t) .$$

Aus einem Vergleich mit der bekannten linearen Zustandsgleichung kann man die (zeitvarianten) Systemmatrizen zu

$$\mathbf{A}(t) = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}^*(t), \mathbf{u}^*(t)} \quad (20.1)$$

$$\mathbf{B}(t) = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right|_{\mathbf{x}^*(t), \mathbf{u}^*(t)} \quad (20.2)$$

ablesen und damit die linearisierte Zustandsgleichung in die Form

$$\Delta \dot{\mathbf{x}}(t) = \mathbf{A}(t) \Delta \mathbf{x}(t) + \mathbf{B}(t) \Delta \mathbf{u}(t) \quad (20.3)$$

bringen. Die Linearisierung um die Trajektorie führt also auf ein linear-zeitvariantes (LZV) System. Die Ausgangsgleichung lautet für das vorliegende System

$$\Delta \mathbf{y}(t) = [1 \ 0 \ 0 \ 0] \cdot \Delta \mathbf{x}(t) \quad (20.4)$$

und ist zeitinvariant.

Für die Berechnung der Systemmatrizen nach Gl. (20.1) und (20.2) in diesem Versuch werden diese in Abhängigkeit von \mathbf{x} und \mathbf{u} angegeben:

$$\mathbf{A}(\mathbf{x}, \mathbf{u}) = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right|_{\mathbf{x}, \mathbf{u}} \quad (20.5)$$

$$\mathbf{B}(\mathbf{x}, \mathbf{u}) = \left. \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right|_{\mathbf{x}, \mathbf{u}} . \quad (20.6)$$

Um die Systemmatrizen $\mathbf{A}(t)$ und $\mathbf{B}(t)$ für einen bestimmten Zeitpunkt $t = t_a$ zu berechnen, wird dann zunächst aus den Gleichungen der Steuerung/Solltrajektorie (Versuch 5) der Zustand $\mathbf{x}_a = \mathbf{x}^*(t_a)$ und Eingang $\mathbf{u}_a = \mathbf{u}^*(t_a)$ bestimmt. Diese Werte können dann in Gl. (20.5) und (20.6) eingesetzt werden, um $\mathbf{A}(t_a)$ und $\mathbf{B}(t_a)$ zu erhalten.

20.2 LQ-Reglerentwurf

Für solch ein LZV-System lässt sich ein Entwurfsverfahren wie der bekannte LQR-Entwurf für LZI-Systeme durchführen, der das LAGRANGESche Gütemaß

$$J = \frac{1}{2} \int_{t_0}^{t_1} (\mathbf{x}^T(t) \mathbf{Q} \mathbf{x}(t) + \mathbf{u}^T(t) \mathbf{R} \mathbf{u}(t)) dt \quad (20.7)$$

minimiert.

Dieser führt auf das zeitvariante, lineare Regelgesetz

$$\mathbf{u}(t) = -\mathbf{K}(t) \mathbf{x}(t) . \quad (20.8)$$

Die Matrix $\mathbf{K}(t)$ wird über die Gleichung

$$\mathbf{K}(t) = \mathbf{R}^{-1} \mathbf{B}^T(t) \mathbf{P}(t) \quad (20.9)$$

berechnet, wobei $\mathbf{P}(t)$ die Lösung der Matrix-RICCATIdifferentialgleichung

$$\dot{\mathbf{P}}(t) = \mathbf{P}(t) \mathbf{B}(t) \mathbf{R}^{-1} \mathbf{B}^T(t) \mathbf{P}(t) - \mathbf{P}(t) \mathbf{A}(t) - \mathbf{A}^T(t) \mathbf{P}(t) - \mathbf{Q} \quad (20.10)$$

ist. Die Herleitung dieser Gleichungen ist für Interessierte in Kapitel 22 gezeigt.

Um Gl. (20.10) lösen zu können, muss eine Anfangsbedingung vorgegeben werden. Dafür bietet es sich an, einen LQ-Entwurf für das System in der Endlage, also $t \geq T$ durchzuführen, und die dabei aus der algebraischen RICCATIgleichung (mit $\mathbf{A}_T = \mathbf{A}(T)$ und $\mathbf{B}_T = \mathbf{B}(T)$)

$$\mathbf{0} = \mathbf{P}_T \mathbf{B}_T \mathbf{R}^{-1} \mathbf{B}_T^T \mathbf{P}_T - \mathbf{P}_T \mathbf{A}_T - \mathbf{A}_T^T \mathbf{P}_T - \mathbf{Q} \quad (20.11)$$

berechnete Matrix \mathbf{P}_T als Endwert $\mathbf{P}(T) = \mathbf{P}_T$ für (20.10) zu verwenden. (Die Differentialgleichung (20.10) wird dann „rückwärts“ gelöst.)

21 MATLAB

Im Folgenden werden kurz einige MATLAB-Befehle erläutert, die im Rahmen dieses Versuchs nützlich sind. Diese Informationen sind der MATLAB-Hilfe [15] entnommen, auf die auch für weitere Informationen verwiesen wird. (Es sind zu den einzelnen Befehlen nicht alle Varianten von Argumenten und Rückgabewerten aufgeführt.)

21.1 Lösen der algebraischen RICCATIgleichung

21.1.1 care

Für die Lösung der *algebraischen* RICCATIgleichung steht in MATLAB die Funktion `care` (continuous algebraic riccati equation) zur Verfügung:

$$X = \text{care}(A, B, Q, R, S, E)$$

gibt die Lösung der Gleichung

$$A^T X E + E^T X A - (E^T X B + S) \cdot R^{-1} \cdot (B^T X E + S^T) + Q = 0$$

zurück.

21.2 Lösen der differentiellen RICCATIgleichung

21.2.1 ode...

Die DGL-Löser von MATLAB wurden schon im MATLAB/Simulink-Praktikum I vorgestellt und verwendet. Hier noch einmal kurz die wichtigsten Punkte.

Der Aufruf der `ode...`-Funktionen (ordinary differential equation) erfolgt mit der Syntax

$$[t \ y] = \text{ode...}(\text{odefun}, \text{tspan}, y_0)$$

wobei `odefun` ein Handle der Funktion ist, die den zu lösenden Zusammenhang $\dot{y} = f(t, y)$ angibt. Dabei muss `odefun` so wie folgt aufgerufen werden können:

$$ydot = \text{odefun}(t, y)$$

`tspan` gibt an, für welche Zeiten die Gleichung gelöst werden soll. Dafür kann `tspan` entweder ein Vektor mit zwei Elementen

$$t = [t_0 \ t_e]$$

oder ein Vektor mit mehr als zwei Elementen

```
t = [t0 t1 t2 ... te]
```

sein.

In beiden Fällen wird von t_0 nach t_e integriert. (So ist es auch möglich, Differentialgleichungen rückwärts zu integrieren!) Werden nur Anfangs- und Endpunkt angegeben, gibt MATLAB in t und y die Ergebnisse zu allen Zeitpunkten zurück, die MATLAB auch intern berechnet hat. Im zweiten Fall gibt MATLAB den Wert von $y(t)$ zu genau den angegebenen Punkten zurück. (Intern wird aber die Lösung zu mehr Punkten berechnet, wenn dies nötig sein sollte, um die vorgegebene Genauigkeit zu erreichen.)

y_0 ist der Anfangswert der Lösung zum Zeitpunkt t_0 .

Die Funktionen `ode...` geben in y die numerisch bestimmten Werte von $y(t)$ zu den in t ausgegebenen Zeitpunkten zurück. Ist $y(t)$ vektorwertig, dann sind die Lösungsvektoren zeilenweise angeordnet. (D. h. jede Zeile entspricht der Lösung zu dem entsprechenden in t angegebenen Zeitpunkt.)

21.2.2 reshape

Die DGL-Löser können nur mit Gleichungen arbeiten, bei denen y ein Vektor (oder Skalar) ist. In diesem Fall soll jedoch eine DGL berechnet werden, deren gesuchte Funktion $P(t)$ matrixwertig ist. In diesem Zusammenhang ist die Funktion `reshape` nützlich. Der Aufruf

```
B = reshape(A, m, n)
```

überträgt die Einträge von A *spaltenweise* in die $(m \times n)$ -Matrix B (Beispiel beachten). Die Matrix A muss daher genau m mal n Einträge besitzen.

Beispiel:

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

>> B = reshape(A, 1, 9)
B =
     1     2     3     4     5     6     7     8     9

>> C = reshape(B, 3, 3)
C =
     1     4     7
     2     5     8
     3     6     9
```

Wenn eine Matrix in einen Vektor umgewandelt werden soll, dann kann alternativ auch der „colon“-Operator verwendet werden, der alle Elemente spaltenweise in einen Spaltenvektor anordnet:

```
>> B = A(:).'
```

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

21.2.3 flipud/fliplr

Um eine Matrix zu spiegeln (im anschaulichen Sinne) stehen die Befehle `flipud` und `fliplr` zur Verfügung. Erster spiegelt eine Matrix horizontal (up/down), der andere vertikal (left/right).

Beispiel:

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

```
A =
```

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

```
>> B = flipud(A)
```

```
B =
```

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

21.3 Aufbau der Regelung unter Simulink

Leider kann das im letzten Versuch vorgestellte Look-up-Table nur eindimensionale Ausgangswerte verarbeiten. Der Sollzustand $\mathbf{x}^*(t)$ und $\mathbf{K}(t)$ sind jedoch vektorielle Größen ((4x1) bzw. (1x4)-Vektoren).

Um dieses Problem zu lösen, können Sie entweder mehrere Look-Up-Tables verwenden, und die einzelnen Komponenten der Vektoren dann zusammenführen. Eine andere Möglichkeit wäre die Verwendung des Blocks „MATLAB Function“, in der beliebige MATLAB-Funktionen aufgerufen werden können. So auch das schon vorgestellte `interp1`. Machen Sie sich in diesem Fall Gedanken darüber, was passiert wenn t größer als die Übergangszeit T wird, also außerhalb der vorhandenen Daten liegt. Des weiteren wäre es auch möglich, den Block „From Workspace“ zu verwenden.

22 Anhang – Zur Herleitung des LQ-Reglers für LZV-Systeme

Die Herleitung des LQ-Reglers für LZV-Systeme verläuft zu großen Teilen analog der Herleitung des LQ-Reglers für LZI-Systeme, wie sie bspw. in [9] gezeigt ist. Aus diesem Grunde wird hier nicht jeder Schritt gezeigt, sondern nur die wichtigen Zwischenschritte dargestellt und auf die Unterschiede zum LZI-Entwurf eingegangen. Für die ausgelassenen Schritte wird auf [9] verwiesen.

In [9] wird allgemein das Problem der Optimierung eines Gütefunktional

$$J = f_e(\mathbf{x}(t_e), t_e) + \int_{t_0}^{t_e} f_0(\mathbf{x}(t), \mathbf{u}(t), t) dt \quad (22.1)$$

behandelt.

Der Zusammenhang zwischen $\mathbf{x}(t)$ und $\mathbf{u}(t)$ ist dabei über die Nebenbedingung

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t), \quad (22.2)$$

die beim LQR-Entwurf der Zustandsgleichung entspricht, gegeben. Außerdem ist noch die Anfangsbedingung

$$\mathbf{x}(t_0) = \mathbf{x}_0 \quad (22.3)$$

gegeben.

Zur Lösung des durch (22.1), (22.2) und (22.3) gegebenen Optimierungsproblems werden noch die LAGRANGESchen Multiplikatoren

$$\Psi(t)$$

eingeführt und die Hamilton-Funktion

$$H(\mathbf{x}, \mathbf{u}, \Psi, t) = \Psi^T(t) \cdot \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) - f_0(\mathbf{x}(t), \mathbf{u}(t), t) \quad (22.4)$$

definiert.

Dann wird die Lösung des Optimierungsproblems durch die Transversalitätsbedingung

$$\Psi(t_e) = -\frac{\partial f_e}{\partial \mathbf{x}(t_e)}, \quad (22.5)$$

die Steuerungsgleichung

$$\frac{\partial H}{\partial \mathbf{u}} = \mathbf{0} \quad (22.6)$$

und die adjungierte Differentialgleichung

$$\dot{\Psi} = -\frac{\partial H}{\partial \mathbf{x}} \quad (22.7)$$

beschrieben. (Die Herleitung ist in [9] gegeben.)

In unserem Fall ist

$$\mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) = \dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \quad (22.8)$$

und

$$f_0(\mathbf{x}(t), \mathbf{u}(t), t) = \frac{1}{2}\mathbf{x}^T(t)\mathbf{Q}\mathbf{x}(t) + \frac{1}{2}\mathbf{u}^T(t)\mathbf{R}\mathbf{u}(t) . \quad (22.9)$$

Da hier das LAGRANGESCHE Gütemaß (20.7) verwendet wird, fällt die Transversalitätsbedingung (22.5) weg.

In Gl. (22.2) ist die Zeitabhängigkeit der Nebenbedingung \mathbf{f} explizit angegeben (und nicht nur indirekt über ein zeitabhängiges $\mathbf{x}(t)$ und $\mathbf{u}(t)$), so dass Gl. (22.8) den Annahmen, unter denen die Lösung des Optimierungsproblems entwickelt wurde, entspricht.

(Es dürfte auch die Funktion $f_0(\mathbf{x}(t), \mathbf{u}(t), t)$ im Integral des Gütefunktional zeitabhängig sein. Dies bedeutet, dass auch \mathbf{Q} und \mathbf{R} zeitvariante Gewichtungsmatrizen sein könnten.)

Die Hamilton-Funktion (22.4) lautet hier

$$H = \Psi^T \cdot (\mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)) - \frac{1}{2}\mathbf{x}^T(t)\mathbf{Q}\mathbf{x}(t) - \frac{1}{2}\mathbf{u}^T(t)\mathbf{R}\mathbf{u}(t) \quad (22.10)$$

Da in den Gleichungen (22.6) und (22.7) nur partielle Ableitungen nach \mathbf{u} , \mathbf{x} bzw. Ψ vorkommen, aber keine Ableitungen nach der Zeit t , bedeutet das, dass auch die nächsten Schritte vollkommen analog dem in [9] gezeigten Vorgehen für den zeitinvarianten Fall sind.

So ergibt sich aus der Steuerungsgleichung (22.6)

$$\mathbf{u}(t) = \mathbf{R}^{-1}\mathbf{B}^T(t)\Psi(t) \quad (22.11)$$

und mit diesem $\mathbf{u}(t)$ weiter mit (22.8)

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{R}^{-1}\mathbf{B}^T(t)\Psi(t) \quad (22.12)$$

sowie aus der adjungierten Differentialgleichung (22.7)

$$\dot{\Psi}(t) = \mathbf{Q}\mathbf{x}(t) - \mathbf{A}^T(t)\Psi(t) . \quad (22.13)$$

Es wird ein lineares, zeitvariantes Regelgesetz

$$\mathbf{u}(t) = -\mathbf{K}(t)\mathbf{x}(t) \quad (22.14)$$

angenommen. (Das würde sich auch automatisch ergeben (siehe z. B. [11]), nur wäre die Herleitung dann umfangreicher. Daher wird im Weiteren nur gezeigt, dass die in Abschnitt 20.2 angegebenen Gleichungen den optimalen *linearen* Regler nach (22.14) beschreiben.)

Aus Gl. (22.11) folgt, da die Matrizen \mathbf{R} und $\mathbf{B}(t)$ fest vorgegeben sind, dass der Vektor $\Psi(t)$ durch

$$\Psi(t) = -\mathbf{P}(t)\mathbf{x}(t) \quad (22.15)$$

darstellbar sein muss. Ersetzt man damit $\Psi(t)$ und $\dot{\Psi}(t)$ (Produktregel beim Ableiten beachten) in Gl. (22.13), dann erhält man

$$-\dot{\mathbf{P}}(t)\mathbf{x}(t) - \mathbf{P}(t)\dot{\mathbf{x}}(t) = \mathbf{Q}\mathbf{x}(t) + \mathbf{A}^T(t)\mathbf{P}(t)\mathbf{x} .$$

In dieser Gleichung kann $\dot{\mathbf{x}}(t)$ noch durch Gl. (22.12) ersetzt werden, wobei auch in (22.12) zuvor noch $\Psi(t)$ durch (22.15) ersetzt wird. So ergibt sich

$$-\dot{\mathbf{P}}(t)\mathbf{x}(t) - \mathbf{P}(t)(\mathbf{A}(t)\mathbf{x}(t) - \mathbf{B}(t)\mathbf{R}^{-1}\mathbf{B}^T(t)\mathbf{P}(t)\mathbf{x}(t)) = \mathbf{Q}\mathbf{x}(t) + \mathbf{A}^T(t)\mathbf{P}(t)\mathbf{x}(t) .$$

Aus allen Summanden dieser Gleichung kann $\mathbf{x}(t)$ nach rechts ausgeklammert werden, so dass man diese auch

$$(-\dot{\mathbf{P}}(t) - \mathbf{P}(t)\mathbf{A}(t) - \mathbf{A}^T(t)\mathbf{P}(t) + \mathbf{P}(t)\mathbf{B}(t)\mathbf{R}^{-1}\mathbf{B}^T(t)\mathbf{P}(t) - \mathbf{Q})\mathbf{x}(t) = \mathbf{0}$$

schreiben kann. Da diese Gleichung für alle $\mathbf{x}(t)$ erfüllt sein muss, muss der Term in der Klammer alleine schon $\mathbf{0}$ sein, also

$$\dot{\mathbf{P}}(t) + \mathbf{P}(t)\mathbf{A}(t) + \mathbf{A}^T(t)\mathbf{P}(t) - \mathbf{P}(t)\mathbf{B}(t)\mathbf{R}^{-1}\mathbf{B}^T(t)\mathbf{P}(t) + \mathbf{Q} = \mathbf{0} .$$

Dies ist genau die angegebene RICCATIdifferentialgleichung (20.10). Um aus dieser den Verlauf für $\mathbf{P}(t)$ bestimmen zu können, muss noch eine Anfangsbedingung angegeben werden. In [9] wurde diese aus der Transversalitätsbedingung (22.5) abgeleitet. In diesem Fall wird jedoch das LAGRANGESche Gütemaß verwendet, so dass diese nicht existiert. Als Anfangsbedingung wird wie in [8] diejenige Matrix $\mathbf{P}(t_e)$ verwendet, die sich bei einem zeitinvarianten Optimalreglerentwurf für den Systemzustand zum Zeitpunkt $t = t_e$ ergibt. Das hat dann auch den Vorteil, dass sich die Regelparameter nach Erreichen der oberen Ruhelage nicht mehr schlagartig ändern müssen, sondern dass einfach die letzte Reglereinstellung der Trajektorienfolgeregelung für die Regelung um die obere Ruhelage verwendet werden kann.

Literaturverzeichnis

- [1] ABEL, D. und A. BOLLIG: *Rapid Control Prototyping, Methoden und Anwendungen*. Springer, 2006.
- [2] ADAMY, J.: *Systemdynamik und Regelungstechnik II*. Shaker, 2007.
- [3] ADAMY, J.: *Systemdynamik und Regelungstechnik III*. Shaker, 2007.
- [4] FINCKENSTEIN, K. G. F. VON, J. LEHN, H. SCHELLHAAS und H. WEGMANN: *Arbeitsbuch Mathematik für Ingenieure, Band I*. B. G. Teubner, 2002.
- [5] FINCKENSTEIN, K. G. F. VON, J. LEHN, H. SCHELLHAAS und H. WEGMANN: *Arbeitsbuch Mathematik für Ingenieure, Band II*. B. G. Teubner, 2002.
- [6] FRANKE, A.: *Modellierung und Regelung eines mechatronischen Systems. Eine Realisierung des invertierten Pendels mit momentgeregeltem Antriebsmotor*. Diplomarbeit, TU Clausthal, 1997.
- [7] GRAICHEN, K., V. HAGEMEYER und M. ZEITZ: *A new approach to inversion-based feedforward control design for nonlinear systems*. Automatica, 41:2033–2041, 2005.
- [8] HOPP, C.: *Trajektorienfolge­regelung mittels linear-zeitvarianter Ausgangsrückführung*. Lehmanns Media, Berlin, 2005.
- [9] KONIGORSKI, U.: *Mehrgrößenreglerentwurf im Zustandsraum*. Skript, Darmstadt, 2007.
- [10] KONIGORSKI, U.: *Systemdynamik und Regelungstechnik I*. Institut für Automatisierungstechnik, Technische Universität Darmstadt, WS 07/08.
- [11] KWAKERNAAK, H. und R. SIVAN: *Linear Optimal Control Systems*. Wiley Interscience, New York, etc., 1972.
- [12] LUNZE, J.: *Regelungstechnik 1*. Springer, Berlin, 4. Aufl., 2006.
- [13] LUNZE, J.: *Regelungstechnik 2*. Springer, Berlin, 4. Aufl., 2006.
- [14] MARKERT, R.: *Technische Mechanik, Teil B*. Fachbereich Mechanik, Technische Universität Darmstadt, 2002.
- [15] MATHWORKS, T.: *MATLAB R2007b Online Documentation*.
- [16] PIETRUSZKA, W.: *MATLAB und Simulink in der Ingenieurpraxis*. B.G Teubner Verlag, Wiesbaden, 2. Aufl., 2006.
- [17] SHAMPINE, L. F., J. KIERZENKA und M. W. REICHEL: *Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c*. http://www.mathworks.com/bvp_tutorial, 2000.
- [18] VOIGT, C., A. K. ARONSEN und J. ADAMY: *Formelsammlung der Matrizenrechnung*, Juni 2006.