

1 Systemanalysis with Matlab/Simulink

This exercise unit covers two main topics. First, the computer numerics program MATLAB and its associated simulation environment SIMULINK will be introduced. This software package is used for system analysis and simulation of dynamic systems. A free MATLAB license is provided by TU Wien. In the institute's computer lab, MATLAB/SIMULINK version R2025a is available.

Furthermore, controller design using loop shaping will be carried out. For preparation, study the lecture notes for course Quantum Technology and Devices: Control Basics.



All MATLAB/SIMULINK files needed to complete this exercise can be found in `uebung1.zip`. These files will be referred to as “exercise materials” in the following.



For questions or suggestions regarding this exercise, please contact

- Michael Girsch <girsch@acin.tuwien.ac.at>

1.1 Working with Matlab

MATLAB is a computer numerics program. The name is an abbreviation for MATRIX LABORatory, indicating that the program was originally designed for calculations with vectors and matrices. The MATLAB desktop (the actual program window) contains the following windows in the default setting. (However, this can be customized.)

- **Command Window**

This is the input area, which must be displayed in any case. All commands can be entered after the prompt `>>` and executed directly. If a command sequence is terminated with a semicolon, the MATLAB output is suppressed. Pressing the Enter key executes the command line immediately. For multi-line command entries, the Shift and Enter keys can be used to jump to a new line.

- **Editor**

The editor is used to edit programs written in MATLAB (or other text files). Matlab programs have the file extension `.m` if they are created as *Matlab Script* or *Matlab Function*, or the file extension `.mlx` if they are created as *Live Script*. The editor also allows step-by-step execution and debugging of Matlab Scripts.

- **Workspace Browser**

The current variables are displayed in MATLAB in the so-called Workspace. They can be accessed and modified by clicking.

- **Current Folder Browser**

The Current Folder Browser displays the current working directory. Files and folders can be opened, created, edited, etc. It is recommended that all files accessed during calculation or simulation are located in a common, local (current) directory.

Familiarize yourself with MATLAB.



Complete the online course *MATLAB Onramp* provided by MATHWORKS. This explains how to use MATLAB and the basic constructs of the MATLAB programming language. To do this, you must create a MATHWORKS account. Use your TU email address (`...@student.tuwien.ac.at`). This can be linked to the license provided by TU Wien.



In addition to the *Live Scripts* presented in *MATLAB Onramp*, MATLAB programs can also be created as *Matlab Scripts* and *Matlab Functions*. *Matlab Scripts* are characterized by faster execution times than *Live Scripts*, but plots must be displayed in separate windows. *Matlab Functions* have input and output values. Furthermore, the variables within *Matlab Functions* are local, meaning that they do not appear in the Workspace.

Exercise 1.1. In the exercise materials you will find the *Matlab Scripts* `cds_matlab_intro_part1.m` and `cds_matlab_intro_part2.m`. Open these and execute the commands contained therein step by step. Note that the function `mittelwert.m` is called, which must therefore be in the current working directory. To execute individual command sequences, select them in the editor and press F9. To execute an entire M-file, either enter its name (without file extension) in the Command Window or open the file in the editor and press F5. Try to understand all commands and use the help function if necessary.

Exercise 1.2. Given is the linear system of equations

$$\begin{aligned}x_1 + 2x_2 + 4x_3 &= 2 \\2x_1 + 2x_2 + x_3 &= 1 \\3x_1 + 2x_2 &= 3.\end{aligned}\tag{1.1}$$

Write this system of equations in matrix form and determine the solution vector $\mathbf{x} = [x_1, x_2, x_3]^T$. Perform the calculation once with the command `inv()` and once with the command `mldivide()` (or its short form `\`). Consider the differences between the two commands and when each should be used.

Exercise 1.3. Given is the truncated Fourier series expansion of the rectangular function at the n -th summand

$$\text{rect}(x) \approx A \sum_{k=1}^n \frac{4}{\pi(2k-1)} \sin((2k-1)x),\tag{1.2}$$

where A denotes the amplitude. Using a **for** loop, represent this rectangular function for $n = 1, 2, \dots, 100$ in the interval $x \in [0, 10]$. Use the **pause** command to display the function in the **for** loop.

Remark: Choose a suitable step size Δx for the representation such that the sampling theorem is satisfied.

Exercise 1.4. Given is the polynomial

$$p_1(s) = s^3 + 3s^2 + 4s + 9, \quad (1.3)$$

and $p_2(s)$ is the characteristic polynomial of the matrix

$$\mathbf{A} = \begin{bmatrix} -1 & 2 \\ -3 & 10 \end{bmatrix}. \quad (1.4)$$

1. Calculate the polynomial $p_3(s) = p_1(s)p_2(s)$ in MATLAB.
2. Write a function with the interface **pd = polydiff(p)**, which receives as an argument **p** the coefficients of an arbitrary polynomial $p(s)$ and returns as a return value **pd** the coefficients of the derived polynomial $dp(s)/ds$.

Remark: Do not use the command **polyder()**, but implement your own algorithm.

3. Write a function with the interface **s0 = findzero(p,sstart)**, which receives as arguments **p** the coefficients of an arbitrary polynomial $p(s)$ and a starting value **sstart**. Using the Newton method, the function should search for and return a zero **s0** of the polynomial $p(s)$ starting from the starting value **sstart**. Consider a suitable termination criterion. You can use the function **polydiff()** in your algorithm if necessary.
4. Test the function **findzero()** using the polynomial $p_3(s)$. You can, of course, first graphically represent $p_3(s)$.
5. Determine the zeros of $p_3(s)$ using the command **roots()**.
6. Is the characteristic polynomial of the matrix **A** a Hurwitz polynomial?

Remark: Polynomials can be represented in MATLAB as vectors of descending ordered polynomial coefficients, i.e., $p(s) = a_n s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0$ is entered as the vector $\mathbf{p} = [a_n, a_{n-1}, \dots, a_1, a_0]^T$. To evaluate a polynomial at a specific point s , the command `polyval()` can be used. To determine the characteristic polynomial of a square matrix, the command `poly()` can be used. The multiplication of two polynomials corresponds to the convolution of their coefficient vectors. The (discrete) convolution operation can be performed with the command `conv()`.

1.1.1 Control System Toolbox

Toolboxes are collections of functions that extend the functionality of MATLAB. After the initial installation, toolboxes are automatically loaded when MATLAB starts. An overview of the installed toolboxes can be obtained with the command line command `ver`. The *Control System Toolbox* is often useful for control engineering tasks. It supports the analysis and controller design of linear dynamic systems.

Exercise 1.5. Open the file `cds_matlab_intro_part3.m` in the MATLAB editor and work through all the commands step by step. Try to understand all commands and use the help function if necessary.

Determining the behavior of a system to an input (e.g., the step response) is mathematically formulated as an initial value problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t), \quad \mathbf{x}(t_0) = \mathbf{x}_0 \quad (1.5a)$$

$$\mathbf{y} = \mathbf{h}(\mathbf{x}, \mathbf{u}, t) \quad (1.5b)$$

Initial value problems can be solved with numerical integration algorithms. The function of such an integration algorithm will be briefly illustrated using the explicit Euler method (Euler forward method). In the Euler forward method, the time derivative is approximated using the difference quotient

$$\dot{\mathbf{x}}(kT_a) \approx \frac{\mathbf{x}((k+1)T_a) - \mathbf{x}(kT_a)}{T_a} \quad (1.6)$$

with the step size T_a . This results in an approximation of the solution of the initial value problem (1.5b)

$$\mathbf{x}((k+1)T_a) = \mathbf{x}_{k+1} \approx \mathbf{x}_k + T_a \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k, kT_a), \quad \mathbf{x}_0 = \mathbf{x}(t_0). \quad (1.7)$$

Exercise 1.6. In this exercise, the step response of the system

$$\dot{x}_1 = x_2, \quad x_1(0) = 0 \quad (1.8a)$$

$$\dot{x}_2 = -x_1 - 2x_2 + u, \quad x_2(0) = 0 \quad (1.8b)$$

$$y = x_1 \quad (1.8c)$$

for the time period $t = 0, \dots, 10$ s is to be determined by simulation.

1. Solve the initial value problem (1.8) using the Euler forward method with the step size $T_a = 0.5$ s and graphically represent the calculated step response.
2. Solve the initial value problem (1.8) using the MATLAB command `ode45` and graphically represent the calculated step response. Note that the differential equation (ODE) must be passed as a *Function Handle*.
3. Using the `step()` command of the Control System Toolbox, calculate the step response of the continuous system (1.8) and graphically represent the calculated step response.
4. Discretize the system (1.8) using the command `c2d()` (zero-order hold) and a sampling time of $T_a = 0.5$ s and graphically represent the calculated step response. The command `stairs` is suitable for representing time-sampled signals.
5. How do the results differ?

1.1.2 Simulink

SIMULINK is an extension of MATLAB for the simulation and analysis of dynamic systems. Solving initial value problems is simplified using SIMULINK. The graphical user interface allows the description of dynamic systems using block diagrams. SIMULINK provides a library of pre-fabricated function blocks, which can be extended by user-defined blocks. Familiarize yourself with Simulink.



Complete the Simulink Onramp course provided in SIMULINK. To do this, start SIMULINK using the MATLAB command `simulink`. In the window that opens, click on *Simulink Onramp* in the bottom left corner to start the Onramp course. Further information can be found at the link behind the QR code.



The following procedure is recommended for efficient work with SIMULINK:

1. Parameter values are not entered directly in SIMULINK, but are summarized and defined in a Matlab script, which is executed before the simulation. This allows the parameters to be changed easily and centrally. To update the parameters, the Matlab script must be executed again.

Remark: All variables that are in the Matlab workspace are also available in SIMULINK.

2. If the mathematical expressions become more extensive, it is recommended to avoid the interconnection of many individual blocks by using user-defined (programmed) blocks. The corresponding blocks are located in the **User-Defined Functions** group. In the **Fcn** block, both MATLAB functions and user-defined functions can be used.

3. Another way to improve the clarity of models is to use subsystems (**Ports & Subsystems** → **Subsystem**).
4. To reduce the number of signal flow lines displayed on the screen, the **From** and **Goto** blocks from the **Signal Routing** group can be used.
5. LTI systems, which have been created using the Control System Toolbox as a transfer function or in state-space representation, can be directly integrated into Simulink (**Control System Toolbox** → **LTI System**).

After creating a model (block diagram), the simulation settings must be made, which can be done in the *Model Settings* menu item under the *Modeling* tab. Essential here is the choice of simulation duration and integration algorithm (solver). The currently set solver is displayed in the footer of the Simulink model on the right. Clicking on this also opens the simulation settings. Furthermore, limits on the time step and accuracy requirements can be set for the integration algorithm. Within the scope of this introduction, a detailed discussion of the algorithms used will be omitted. An overview of some solvers for initial value problems available in MATLAB, *ode23*, *ode45*, *ode113*, *ode15s*, *ode23s*, *ode23t* or *ode23tb*, can be found in the help (callable e.g. with `doc ode45`) under the heading *Algorithms*. These algorithms are also used by SIMULINK, among others. Furthermore, reference is made to the specialist literature, e.g. [1, 2]. Further familiarize yourself with Simulink.

Exercise 1.7. In the exercise materials you will find the Simulink model `cds_simulink_intro_part1.slx`. Open this model and start the simulation. Make sure to execute the *Matlab script* `init_cds_simulink_intro_part1.m` beforehand. This *Matlab script* creates all variables necessary for the simulation in the MATLAB workspace. Try to understand all blocks and use the help function if necessary.

Implementation of continuous-time dynamic systems using an integrator chain

To be able to solve a higher-order differential equation with Simulink, it must be represented as a system of first-order differential equations. This procedure is shown in Example 1.1.

Example 1.1. The initial value problem

$$\ddot{y} + \cos^2(y)\ddot{y} + a\dot{y} + u = 0, \quad \ddot{y}(0) = \dot{y}(0) = y(0) = 0, \quad u = \sin(t) \quad (1.9)$$

with a parameter $a = 0.3$ is to be represented using an integrator chain. First, the differential equation is converted into state-space representation $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, u)$. Since (1.9) is a third-order differential equation, a three-dimensional state

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y \\ \dot{y} \\ \ddot{y} \end{bmatrix} \quad (1.10)$$

is used. This results in a system of three first-order differential equations

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ -\cos^2(x_1)x_3 - ax_2 - u \end{bmatrix}, \quad \mathbf{x}(0) = \begin{bmatrix} x_1(0) \\ x_2(0) \\ x_3(0) \end{bmatrix}. \quad (1.11)$$

Since we are interested in the solution $\mathbf{x}(t)$, (1.11) is integrated once over time. This results in

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} = \begin{bmatrix} x_1(0) + \int_0^t x_2(\tau) d\tau \\ x_2(0) + \int_0^t x_3(\tau) d\tau \\ x_3(0) + \int_0^t (-\cos^2(x_1(\tau))x_3(\tau) - ax_2(\tau) - u(\tau)) d\tau \end{bmatrix} \quad (1.12)$$

In (1.12) it is evident that three integrators are needed. Each of these integrators can be represented in Simulink using an integrator block. Thus, the system of differential equations can be represented in Simulink as shown in Figure 1.1.

Remark: The initial conditions $(x_{0,1}, x_{0,2}, x_{0,3})$ must be defined in the settings of the individual integrator blocks.

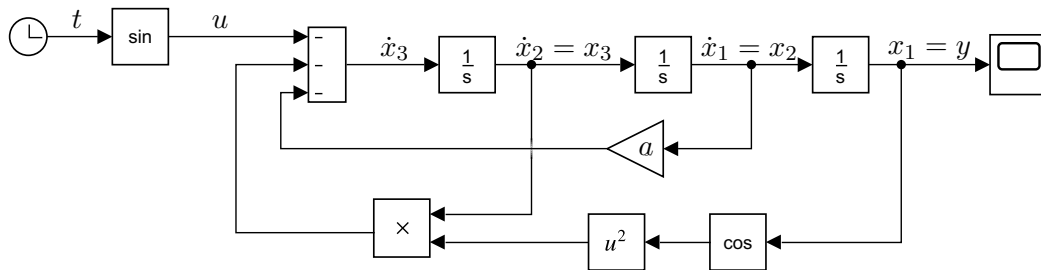


Figure 1.1: Implementation of (1.12) using an integrator chain.

Implementation of continuous-time dynamic systems using *Matlab Function* blocks.

The implementation of a differential equation using an integrator chain becomes unclear for high-dimensional problems and increasing mathematical complexity. This can be prevented by using *Matlab Function* blocks. The use of *Matlab Function* blocks is explained in Example 1.2.

Example 1.2. The initial value problem (1.9) is to be solved using a *Matlab Function* block. The system is converted into state-space representation $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, u)$, cf. (1.11).

Integration then yields

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_0^t \mathbf{f}(\mathbf{x}(\tau), u(\tau)) d\tau \quad (1.13a)$$

$$y(t) = h(\mathbf{x}(t), u(t)). \quad (1.13b)$$

This representation can be implemented directly in Simulink using *Matlab Functions*. To do this, two *Matlab Function* blocks (**User Defined Functions** → **Matlab Function**) are inserted in Simulink. A double click opens a window in the Matlab editor where programming can be done.

Listing 1.1: Implementation of (1.13).

```
function xpunkt = g(u,x,param)
    % Splitting the parameters (for better readability)
    a = param.a;

    % Splitting the state vector (for better readability)
    x1 = x(1);
    x2 = x(2);
    x3 = x(3);

    % Implementation of f(x,u)
    f1 = x2;
    f2 = x3;
    f3 = -cos(x1)^2 * x3 - a * x2 - u;

    % Assignment according to xpunkt = f(x,u)
    xpunkt = [f1; f2; f3];

end

function y = h(u,x,param)
    % Splitting the parameters (for better readability)
    a = param.a;

    % Splitting the state vector (for better readability)
    x1 = x(1);
    x2 = x(2);
    x3 = x(3);

    % Implementation of h(x,u)
    y = x1;

end
```

The representation in (1.13) can then be implemented, for example, using the program code in Listing 1.1. Here, the functions **g** and **h** each have three input parameters (**x**, **u**, **param**) and one return value (**xpunkt** or **y**). If a *Matlab Function* block is open in the Matlab editor, the **Ports and Data Manager** can be opened in the **Editor** tab using the **Edit Data** button. In this, it can be set whether an input parameter should be treated as an input or as a parameter. Parameters must be defined in the Matlab

workspace. It is advisable to organize all required parameters in a *struct array*. It is important here that the *Ports and Data Manager* is set so that the parameter is not *tunable*. The integration of `xpunkt` is now carried out using an integrator block in Simulink. This block can also handle vector-valued signals. It should be noted that the dimension of the initial state of the integrator block must correspond to the dimension of the system state. Finally, Figure 1.2 shows the complete implementation of (1.13) using *Matlab Function* blocks.

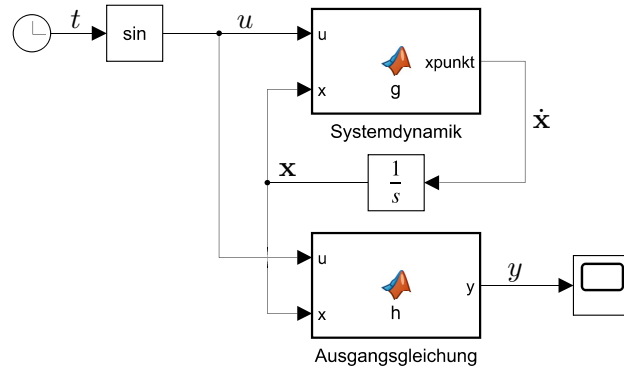


Figure 1.2: Implementation of (1.12) using a Matlab Function.

Implementation of discrete-time dynamic systems using *Matlab Function* blocks

Discrete-time systems can also be implemented using *Matlab Function* blocks. There are several ways to do this. The method explained below has the advantage that the resulting program code can be implemented directly in a digital computer. This allows designed controllers or observers to be ported to industrial plants or test setups with little time expenditure. In particular, it is easy to use the Matlab function directly in the Rapid Prototyping System dSPACE. The following example explains how a discrete-time system can be implemented in Simulink using *Matlab Function* blocks.

Example 1.3. The initial value problem (1.9) is to be simulated using the Euler forward method as a discrete-time system with a sampling time $T_a = 100$ ms. First, analogously to Example 1.2, the system is converted into state-space representation and integrated over time (cf. (1.13)). The integral (1.13) is approximated using the Euler forward method (1.7). This results in

$$\mathbf{x}_{k+1} = \mathbf{x}_k + T_a \mathbf{f}(\mathbf{x}_k, u_k), \quad \mathbf{x}_0 = \mathbf{x}(0) \quad (1.14a)$$

$$y_{k+1} = h(\mathbf{x}_k, u_k). \quad (1.14b)$$

This representation can be implemented directly in a *Matlab Function* in Simulink. The corresponding program code is given in Listing 1.2. Here, the state `xx` is defined as a *persistent* variable, which remains over several function calls. When this variable is created, it is empty (this means the same as the assignment `xx=[]`). Note that

in the first call of the *Matlab Function* block, the variable **xx** is initialized with the initial value **xx0**.

Listing 1.2: Implementation of (1.14).

```
function y = fcn(u,param)
    % Splitting the parameters
    a = param.a;
    Ta = param.Ta;

    % Definition of the discrete state as a persistent variable
    persistent xx;

    % Initialization of the discrete state
    if isempty(xx); xx=param.xx0; end
    % Splitting the state vector
    x1 = xx(1);
    x2 = xx(2);
    x3 = xx(3);

    % Implementation of f(xx,u)
    f1 = x2;
    f2 = x3;
    f3 = -cos(x1)^2*x3 -a*x2 -u;

    % Combining into a vector
    ff = [f1; f2; f3];

    % Approximation using Euler method
    xx = xx + Ta*ff;

    % Implementation of h(xx,u)
    y = x1;
end
```

For the implementation of a discrete-time system in SIMULINK, it is necessary to specify the sampling time. For this purpose, the *Update Method* is set to *discrete* in the *Ports and Data Manager*, see Figure 1.3.

Familiarize yourself with the implementation of dynamic systems.

Exercise 1.8. In the exercise materials you will find the Simulink model `simulink_intro_part2.slx`. This Simulink model implements Example 1.1, Example 1.2, and Example 1.3. Open this model and start the simulation. Make sure to execute the Matlab script `init_cds_simulink_intro_part2.m` beforehand. This *Matlab Script* creates all variables necessary for the simulation in the *Matlab Workspace*. Try to understand all blocks and use the help function if necessary. Pay particular attention to the settings made in the *Ports and Data Manager*.

Exercise 1.9.

1. Implement the system discretized with the command `c2d()` (sampling time

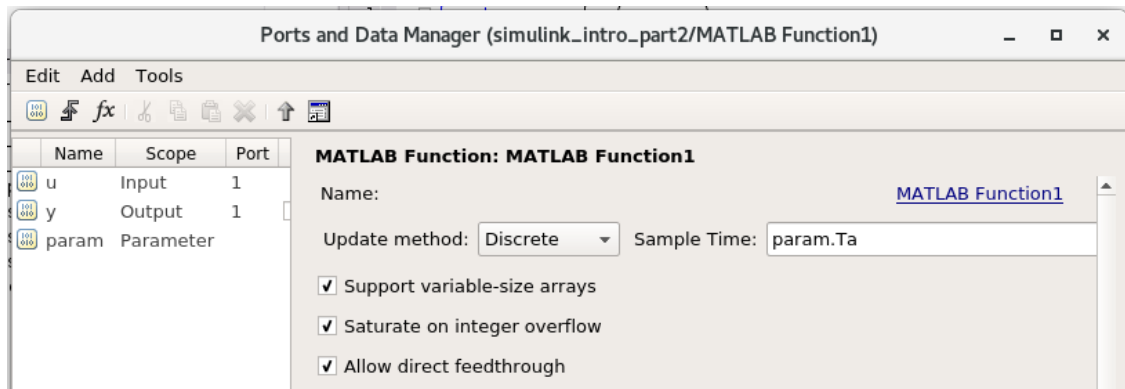


Figure 1.3: Settings in the *Ports and Data Manager* for discrete-time systems.

$T_a = 0.1s$) from Exercise 1.6 in the form of a *Matlab Function* block.

2. Also implement the system using an *LTI Block* and compare the results for different input functions.

1.2 Rotary Flexible Joint

1.2.1 System Description

The laboratory experiment *Rotary Flexible Joint*, shown in Figure 1.4, consists of a base and a carrier that is rotatably mounted to the base and driven by a synchronous motor. The boom is in turn rotatably mounted to the carrier and is elastically connected to the carrier by means of two springs. The present experimental setup can be considered as a simple model of an elastic joint of a robot. Figure 1.5 shows the mechanical equivalent

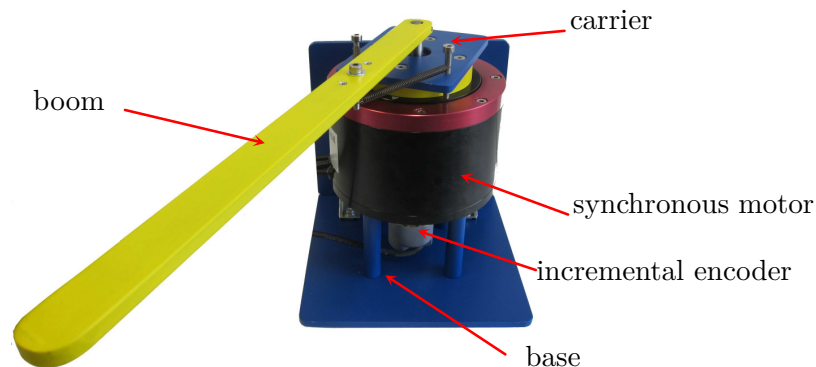


Figure 1.4: The Rotary Flexible Joint Laboratory Experiment.

circuit diagram of the Rotary Flexible Joint. The synchronous motor directly drives the carrier (rotation angle φ_t , rotation angle velocity $\dot{\varphi}_t = \omega_t$). The motor rotor and the carrier are rigidly connected and can therefore be described by a common moment of inertia $I_{t,zz}$. There is also viscous friction (proportional to the rotational speed) with the

friction coefficient d_t and Coulomb friction with the friction coefficient $d_{t,c}$ between the stator and rotor. The boom is rotatably mounted to the carrier (absolute rotation angle φ_a , absolute rotation angle velocity $\dot{\varphi}_a = \omega_a$, see Figure 1.6), with the coupling being achieved via two linear springs (stiffness c_f). Due to the special arrangement of the springs, a nonlinear relationship of the form $\tau_{c,at}(\varphi_a, \varphi_t)$ results for the moment applied by the springs. It is further assumed that viscous friction with the friction coefficient d_a occurs in the boom bearing, which is proportional to the absolute rotational speed $\dot{\varphi}_a = \omega_a$. This is due to the fact that the friction in the incremental encoder used to measure the angle φ_a , see Fig. 1.4, dominates the friction in the bearing between the carrier and the boom. For a precise description of the system's kinematics and the geometric arrangement of the

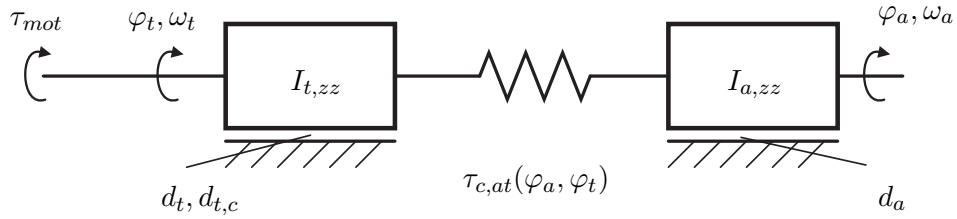


Figure 1.5: Mechanical Equivalent Circuit Diagram of the Rotary Flexible Joint Laboratory Experiment.

springs, a detailed view of the carrier and the boom is shown in Figure 1.6. The carrier and the boom rotate around a common pivot point (origin of the inertial coordinate system $(0_0x_0y_0z_0)$, of the body-fixed coordinate system $(0_tx_ty_tz_t)$ of the carrier, and of the body-fixed coordinate system $(0_ax_ay_az_a)$ of the boom). The center of gravity of the carrier is at the pivot point; the center of gravity SP_a of the boom is offset from the pivot point. As shown in Figure 1.6, two (linear) springs of stiffness c_f serve for the elastic coupling of the boom with the carrier. These are attached between the mounting points MP_{1t} and MP_{2t} on the carrier and the mounting point MP_a on the boom. The location of the mounting points can be seen in Figure 1.6, where it is assumed that MP_{2t} and MP_{1t} are symmetrically positioned with respect to the y_t -axis of the carrier.

To calculate the equations of motion we first describe the kinematics of the system. We use the degrees of freedom $\mathbf{q} = [\varphi_a, \varphi_t]^T$ or their time derivatives $\dot{\mathbf{q}} = [\omega_a, \omega_t]^T$. The rotation matrices for the carrier and boom subsystems are calculated as

$$\mathbf{R}_0^t = \begin{bmatrix} c_{\varphi_t} & -s_{\varphi_t} & 0 \\ s_{\varphi_t} & c_{\varphi_t} & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{R}_0^a = \begin{bmatrix} c_{\varphi_a} & -s_{\varphi_a} & 0 \\ s_{\varphi_a} & c_{\varphi_a} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

The location of the mounting points and the center of gravity from the pivot point of the respective coordinate system is described by the vectors

$$\mathbf{p}_t^{MP_1} = \begin{bmatrix} -r_2 \\ r_3 \\ 0 \end{bmatrix}, \quad \mathbf{p}_t^{MP_2} = \begin{bmatrix} r_2 \\ r_3 \\ 0 \end{bmatrix}, \quad \mathbf{p}_a^{MP_a} = \begin{bmatrix} 0 \\ r_1 \\ 0 \end{bmatrix}, \quad \mathbf{p}_a^{SP_a} = \begin{bmatrix} 0 \\ r_{sa} \\ 0 \end{bmatrix}.$$

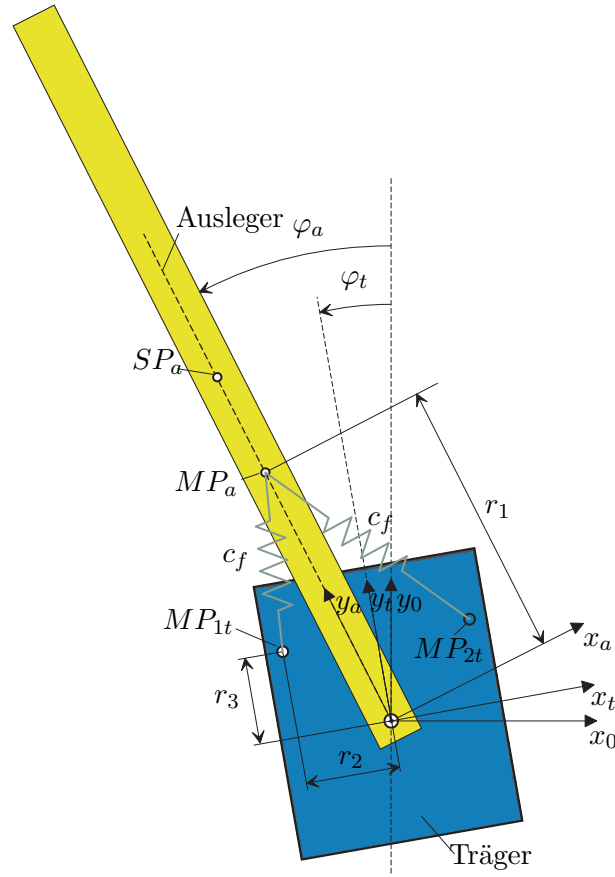


Figure 1.6: Detailed View of the Geometry of the Carrier and the Boom.

In the inertial coordinate system, we have: $\mathbf{p}_0^{MP_i} = \mathbf{R}_0^t \mathbf{p}_t^{MP_i}$, $i = 1, 2$ and $\mathbf{p}_0^{XP_a} = \mathbf{R}_0^a \mathbf{p}_a^{XP_a}$, $X = \{M, S\}$. The manipulator Jacobian matrices are

$$\mathbf{J}_{\mathbf{v},0}^a = \frac{\partial}{\partial \mathbf{q}} \mathbf{p}_0^{SP_a} = \begin{bmatrix} -c_{\varphi_a} r_{sa} & 0 \\ -s_{\varphi_a} r_{sa} & 0 \\ 0 & 0 \end{bmatrix}, \quad \mathbf{J}_{\omega,0}^x = \frac{\partial}{\partial \dot{\mathbf{q}}} \omega_0^x,$$

where the rotational velocity vectors are determined according to the rule $\omega_0^x = [\mathbf{S}(\omega_0^x)_{3,2} \ \mathbf{S}(\omega_0^x)_{1,3} \ \mathbf{S}(\omega_0^x)_{2,1}]^T$ with $\mathbf{S}(\omega_0^x) = \dot{\mathbf{R}}_0^x (\mathbf{R}_0^x)^T$ and $x = \{a, t\}$. The translational velocity of the boom follows from $\mathbf{v}_0^a = \mathbf{J}_{\mathbf{v},0}^a \dot{\mathbf{q}}$.

To calculate the equations of motion using the Lagrange formalism, the kinetic and potential energy of the system are required. The kinetic energy can be represented in the form

$$T = \frac{1}{2} \dot{\mathbf{q}}^T \mathbf{M}(\mathbf{q}) \dot{\mathbf{q}} \quad (1.15)$$

with the symmetric positive definite mass matrix $\mathbf{M}(\mathbf{q})$, where the entries of the mass matrix are determined by the masses and moments of inertia of the bodies involved. The center of gravity of the subsystem consisting of the rotor and carrier lies on the axis of

rotation, which is why only the total moment of inertia of the motor and the carrier about the axis of rotation (z -axis) $I_{t,zz}$ contributes to the kinetic energy. In contrast, the center of gravity SP_a of the boom is offset from the axis of rotation, which is why both translational and rotational components of the kinetic energy occur here.

This boom is essentially a cuboid with length l_a , width b_a , height h_a , and mass density ρ_a . The mass of the boom is given by $m_a = \rho_a l_a b_a h_a$. The location of the center of gravity is evident but is $\mathbf{p}_a^{SP_a} = [0 \ l_a/2 \ 0]^T$. The inertia matrix about the center of gravity SP_a with respect to the body-fixed coordinate system, whose coordinate axes coincide with the principal axes of inertia and whose origin is the same as the center of mass, is $\mathbf{I}_a = \text{diag}(I_{a,xx}, I_{a,yy}, I_{a,zz})$ with

$$I_{a,xx} = \frac{m_a}{12} (h_a^2 + l_a^2), \quad I_{a,yy} = \frac{m_a}{12} (h_a^2 + b_a^2), \quad I_{a,zz} = \frac{m_a}{12} (b_a^2 + l_a^2).$$

The mass matrix is calculated using the relationship

$$\mathbf{M}(\mathbf{q}) = m_a (\mathbf{J}_{\mathbf{v},0}^a)^T \mathbf{J}_{\mathbf{v},0}^a + \sum_{x=\{a,t\}} (\mathbf{J}_{\omega,0}^x)^T \mathbf{R}_0^x \mathbf{I}_x (\mathbf{R}_0^x)^T \mathbf{J}_{\omega,0}^x$$

to

$$\mathbf{M}(\mathbf{q}) = \begin{bmatrix} \frac{1}{4} m_a l_a^2 + I_{a,zz} & 0 \\ 0 & I_{t,zz} \end{bmatrix},$$

where $\mathbf{I}_x$, $x = \{a, t\}$ are the inertia matrices of the boom, carrier, and motor, respectively, in the body-fixed coordinate systems parallel to the principal axes of inertia. The springs are relaxed for $l_f = l_{f0}$, with the length of the spring l_f , i.e., the spring force is $f_c(l_{f0}) = 0$. The components of the potential energy $V = V_1 + V_2$ are

$$V_i = \frac{1}{2} c_f (\Delta l_i)^2 = \frac{1}{2} c_f (l_i - l_{0,i})^2 = \frac{1}{2} c_f \left(\sqrt{\mathbf{l}_i^T \mathbf{l}_i} - l_{0,i} \right)^2,$$

where $\mathbf{l}_i = \mathbf{p}_t^{MP_a} - \mathbf{p}_t^{MP_i}$ represents the vector length of spring $i = \{1, 2\}$ with the relaxed length $l_{0,i}$. The moments due to viscous friction are calculated as

$$\begin{aligned} \boldsymbol{\tau}_{t,d} &= \begin{bmatrix} 0 & 0 & -d_t \omega_t \end{bmatrix}^T \\ \boldsymbol{\tau}_{a,d} &= \begin{bmatrix} 0 & 0 & -d_a \omega_a \end{bmatrix}^T. \end{aligned}$$

The mechanical system of the Rotary Flexible Joint is driven by a synchronous motor (see Figure 1.4), which applies the input torque τ_{mot} to the carrier. The contribution of Coulomb friction at the motor is also taken into account in the form of a generalized moment, which is approximated by the friction torque

$$\tau_c = -d_{t,c} \tanh \left(\frac{\omega_t}{\omega_{t,0}} \right) \quad (1.16)$$

where $\omega_{t,0}$ is a suitably chosen reference speed. It is further assumed that an external disturbance force f_s acts on the center of gravity of the boom, which is oriented in the direction of the (body-fixed) x_a -axis.

The vector of generalized forces and moments is given by

$$\mathbf{f}_q = \left(\mathbf{J}_{\omega,0}^t\right)^T \mathbf{R}_0^t (\boldsymbol{\tau}_m + \boldsymbol{\tau}_{t,d} + \boldsymbol{\tau}_c) + \left(\mathbf{J}_{\omega,0}^a\right)^T \mathbf{R}_0^a \boldsymbol{\tau}_{a,d} + \left(\mathbf{J}_{v,0}^a\right)^T \mathbf{R}_0^a \mathbf{f}_a,$$

$$\text{with } \boldsymbol{\tau}_m = [0 \ 0 \ \tau_{mot}]^T, \quad \boldsymbol{\tau}_{t,d} = [0 \ 0 \ -d_t \omega_t]^T, \quad \boldsymbol{\tau}_c = \left[0 \ 0 \ -d_{t,c} \tanh\left(\frac{\omega_t}{\omega_{t,0}}\right)\right]^T,$$

$$\boldsymbol{\tau}_{a,d} = [0 \ 0 \ -d_a \omega_a]^T \quad \text{and} \quad \mathbf{f}_a = [f_s \ 0 \ 0]^T.$$

The rotation matrices are necessary so that the respective components are expressed in the correct coordinate system. The vector of potential forces is calculated using the relationship

$$\mathbf{g}(\mathbf{q}) = [g_1(\mathbf{q}) \ g_2(\mathbf{q})]^T \quad \text{with} \quad g_1(\mathbf{q}) = \frac{\partial}{\partial q_1} V(\mathbf{q}), \quad g_2(\mathbf{q}) = \frac{\partial}{\partial q_2} V(\mathbf{q}).$$

The equations of motion can be represented in the following form

$$\ddot{\mathbf{q}} = \mathbf{M}(\mathbf{q})^{-1} (-\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}}) \dot{\mathbf{q}} - \mathbf{g}(\mathbf{q}) + \mathbf{f}_q) \quad (1.17)$$

The entries of the Coriolis matrix $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})[k, j]$ can be calculated according to the definition but are all identically zero. This is due to the constant mass matrix $\mathbf{M}$, independent of $\mathbf{q}$. Finally, the mathematical model of the system is given in the form

$$\frac{d}{dt} \mathbf{x} = \mathbf{f}(\mathbf{x}, u, d) \quad (1.18a)$$

$$y = h(\mathbf{x}) = \varphi_a, \quad (1.18b)$$

with the state $\mathbf{x} = [\varphi_a, \omega_a, \varphi_t, \omega_t]^T$, the input $u = \tau_{mot}$, and the disturbance $d = f_s$.

1.2.2 Linearization

For the system (1.18), an analytical calculation of the equilibrium points is not possible. However, these can be determined for $\tau_{motR} = f_{sR} = 0$ by force equilibrium considerations on the boom. For the system (1.18), there are infinitely many equilibrium points $\varphi_{a,R} = \varphi_{t,R}$. We linearize (1.18) around the equilibrium point $\mathbf{x}_R = [0, 0, 0, 0]^T$, while neglecting the Coulomb friction component. Using the input ω_t , f_s and the values from Table 1.1, we obtain the linearized system

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -356.88 & -0.23 & 356.88 & 0 \\ 0 & 0 & 0 & 1 \\ 773.22 & 0 & -773.22 & -38.88 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 \\ 0 & -54.09 \\ 0 & 0 \\ 781.25 & 0 \end{bmatrix}.$$

The eigenvalues of the dynamic matrix of the linearized system at the given equilibrium point with the parameters given in Table 1.1 are $\lambda = [0, -18.55, -10.28 + 25.53j, -10.28 - 25.53j]1/s$.

Designation		Value	Unit
Boom Length	l_a	300	mm
Boom Width	b_a	20	mm
Boom Height	h_a	5.7	mm
Boom Mass Density	ρ_a	2700	$\frac{\text{kg}}{\text{m}^3}$
Total Moment of Inertia of the Carrier and Rotor about the z -axis	$I_{t,zz}$	$12.8 \cdot 10^{-4}$	kgm^2
Geometric Parameter of Mounting Points	r_1	95	mm
Geometric Parameter of Mounting Points	r_2	30	mm
Geometric Parameter of Mounting Points	r_3	30	mm
Spring Stiffness	c_f	221.24	$\frac{\text{N}}{\text{m}}$
Unstressed Spring Length	l_{f0}	34.7	mm
Viscous Friction Constant Carrier	d_t	$4.977 \cdot 10^{-2}$	Nms
Viscous Friction Constant Boom	d_a	$6.42 \cdot 10^{-4}$	Nms
Coulomb Friction Constant Carrier	$d_{t,c}$	$1.43 \cdot 10^{-2}$	Nms
Reference Speed for tanh Approximation	$\omega_{t,0}$	$3 \cdot 10^{-2}$	$\frac{\text{rad}}{\text{s}}$

Table 1.1: Parameters of the Rotary Flexible Joint Laboratory Experiment.

Exercise 1.10. Implement the linearized model of the Rotary Flexible Joint in the form of its state-space representation in MATLAB. Calculate the corresponding open-loop transfer function $G_{u,y} = \frac{y(s)}{u(s)}$ from the input $u = \tau_{mot}$ to the output $y = \varphi_a$ in MATLAB and implement it using an LTI block in Simulink. Check the implementation of the linearized model by comparison with the provided **p-function**. Then compare the step response of the nonlinear and linearized models. Use the Simulink file `Simulation_RFJ_Student.slx`

High friction in the bearings of the synchronous motor was observed in the Rotary Flexible Joint laboratory setup, which is mainly due to the nonlinear Coulomb friction component. For the application of a linear controller the following procedure has proven useful. Since the servo drive for controlling the synchronous motor has a speed controller, a cascaded control loop can be set up, whereby the subordinate controller for controlling the angular velocity ω_t is designed with a significantly larger bandwidth than that of the mechanical system. By specifying the angular velocity ω_t as the input variable, the last equation of the system from (1.18) becomes redundant. For the reduced system, the equilibrium point $\mathbf{x}_R = [0, 0, 0]^T$ clearly applies, with $\omega_t = 0$ and $f_{sR} = 0$. We linearize the reduced system around this equilibrium point and neglect the Coulomb friction component

here as well and obtain

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 \\ -356.88 & -0.23 & 356.88 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 \\ 0 & -54.09 \\ 1 & 0 \end{bmatrix}.$$

Exercise 1.11. Implement reduced linear model in the form of its state-space representation analogously to Exercise 1.10 in MATLAB. Calculate the corresponding open-loop transfer function $G_{u,y} = \frac{y(s)}{u(s)}$ from the input $u = \omega_t$ to the output $y = \varphi_a$ in MATLAB. Then compare the reduced models by simulation in SIMULINK with the complete nonlinear model.

Remark: For the model comparison, use the state ω_t of the complete model as the input $u = \omega_t$ of the reduced models.

1.2.3 Loop Shaping

Exercise 1.12 (Controller Design). Using the loop shaping, design a continuous-time compensation controller for the reduced linearized model of the Rotary Flexible Joint from Exercise 1.11. The step response of the closed loop should meet the following specifications:

Steady-state error	$e_\infty _{r(t)=\sigma(t)} = 0$
Rise time	$t_r = 0.1 \text{ s}$
Overshoot	$\ddot{u} \leq 10 \%$

The controller structure is chosen as

$$R(s) = V \frac{1 + 2\xi(sT) + (sT)^2}{(1 + sT_R)^2} \quad (1.19)$$

where $1 + 2\xi(sT) + (sT)^2$ denotes the compensation term, T_R the time constant for the desired double real pole, and V the gain.

1. Explain why the above controller structure is chosen.
2. Determine the parameters T and ξ of the compensation term such that its zeros correspond to the conjugate complex pole pair of the plant transfer function.
3. Calculate T_r and V using the frequency response method according to the above specifications.
4. Assess the stability of the closed-loop control system.

Remark: Familiarize yourself with the commands `tf()`, `bode()` (with and without return parameters), `feedback()`, and `minreal()` before starting the frequency response design.

Exercise 1.13 (Verification). Implement and test the designed continuous-time controller in the SIMULINK file `RFJ_Regelung.slx`. The courses of the setpoint and disturbance variable are already given. Add the reduced linearized model and the continuous-time controller at the designated locations using LTI blocks.

1. Test by simulation whether the designed controller actually meets the requirements for the reduced linearized system.
2. For the complete nonlinear system, a lower-level speed controller is implemented. Thus, you can also test your controller for this system. What differences do you observe compared to the linear reduced system? What causes these differences?
3. What behavior do you observe with step or ramp-shaped changes in the setpoint and disturbance variable?
4. The setpoint is smoothed via a setpoint filter. Why is the use of such a filter useful? (Note: For the motor, a control variable limit of $|\omega_t| \leq 15 \text{ rad/s}$ applies)

Remark: The visualization of the RFJ in the file `RFJ_Regelung.slx` requires the Simscape Multibody Add-on, which is not installed by default in MATLAB. If you do not have this add-on installed, or if you want to shorten the simulation time, you can also comment out the corresponding block.

Exercise 1.14 (Disturbance Transfer Function). With a step disturbance, the boom oscillates despite control. Calculate the disturbance transfer function $T_{d,y}(z) = \frac{\hat{y}(z)}{\hat{d}(z)}$ of the closed loop from the disturbance $d = f_s$ to the output $y = \varphi_a$ and explain this behavior using the corresponding Bode diagram.

1.3 Influence of the Integration Algorithm

A linear, stable system of the form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ is stiff if the eigenvalues of the system matrix $\mathbf{A}$ vary greatly in magnitude. The system of the form $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$ is therefore stiff (ill-conditioned) if the ratio

$$S = \frac{\max |\sigma(\mathbf{A})|}{\min |\sigma(\mathbf{A})|} \quad (1.20)$$

is very large, where $\sigma(\mathbf{A}) = \{\lambda \in \mathbb{C} : \det(\lambda \mathbf{E} - \mathbf{A}) = 0\}$ denotes the spectrum of $\mathbf{A}$. This is the case if either the time constants or the eigenfrequencies of the individual states vary

significantly. *Implicit* integration algorithms are better suited for solving stiff systems than *explicit* ones. The computation time, the accuracy achieved, and the numerical stability depend, among other things, on the order and time step of the selected integration algorithm.

Exercise 1.15. Given is an RLC circuit, shown in Figure 1.7, which is dynamically described via the states $\mathbf{x} = [i_1, i_2, u_1]^T$ according to

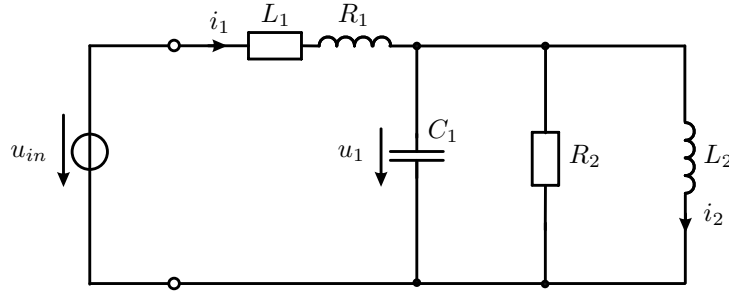


Figure 1.7: RLC circuit.

$$\dot{\mathbf{x}} = \frac{d}{dt} \begin{bmatrix} i_1 \\ i_2 \\ u_1 \end{bmatrix} = \underbrace{\begin{bmatrix} -\frac{R_1}{L_1} & 0 & -\frac{1}{L_1} \\ 0 & 0 & \frac{1}{L_2} \\ \frac{1}{C_1} & -\frac{1}{C_1} & -\frac{1}{R_2 C_1} \end{bmatrix}}_{\mathbf{A}} \mathbf{x} + \begin{bmatrix} \frac{1}{L_1} \\ 0 \\ 0 \end{bmatrix} u_{in}$$

$$\mathbf{y} = \mathbf{x}$$

Use the parameter set $L_1 = 1 \text{ mH}$, $L_2 = 1 \text{ H}$, $R_1 = 100 \Omega$, $R_2 = 1 \Omega$, and $C_1 = 1 \text{ F}$ and the initial state $\mathbf{x}(0) = \mathbf{0}$ for the following investigations. Since the stiffness quotient S of the dynamic matrix $\mathbf{A}$ is relatively large, specifically $S(\mathbf{A}) = 10^5$, this is a stiff system. Create a simulation model for this system in SIMULINK, use the unit step $u_{in} = \sigma(t)$ as input, and investigate the influence of different integration algorithms on the step sizes used, the number of steps, and the computation time with a simulation duration of 100s. Specifically, the explicit Dormand-Prince method, implemented in `ode45`, and the implicit BDF method (Backward Differentiation Formula), implemented in `ode15s` with maximum order 5. Compare the state trajectories and pay particular attention to the course of the state variables in the steady state.

1.4 References

- [1] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth, *Matlab - Simulink - Stateflow, Grundlagen, Toolboxes, Beispiele*, 9th ed. Berlin: De Gruyter, 2017.
- [2] H. Schwarz, *Numerische Mathematik*. Stuttgart: B.G. Teubner, 1997.