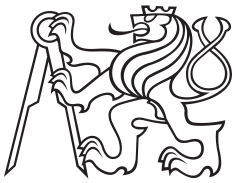
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Using Julia language for automatic control of an educational laboratory model

Štěpán Ošlejšek

Supervisor: doc. Ing. Zdeněk Hurák, PhD.
Study program: Cybernetics and Robotics
May 2023

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Ošlejšek Štěpán** Personal ID number: **499212**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Using Julia language for automatic control of an educational laboratory model

Bachelor’s thesis title in Czech:

Využití jazyka Julia pro automatické řízení výukového laboratorního modelu

Guidelines:

1) By solving a specific problem (and documenting the solution), demonstrate a complete automatic control system design workflow using the Julia programming language (and its packages).
2) In particular, use Julia for:
- assembling nonlinear differential equations modelling the system,
- identification of physical parameters of the model from experimental data,
- linearization of the nonlinear state equations,
- control-related analysis of the linear(ized) model both in time and frequency domains,
- design of some basic linear controllers and their verification through simulation with the nonlinear model, and – possibly – through experiments with a real physical "gadget".
3) Use existing specialized packages such as ModelingToolkit.jl, DifferentialEquations.jl, Symbolics.jl, ControlSystems.jl, ControlSystemIdentification.jl. In case of missing (or broken) functionality, develop your own solution and share with with the international Julia community as open-source software.
4) Implementation/realization of the algorithms for the purpose of laboratory experimentation is nontrivial. Explore the state of the art in this area of rapid prototyping of control algorithms using Julia, and possibly demonstrate the currently attainable functionality using a laboratory model.

Bibliography / sources:

[1] „Julia Course". Graduate Course, Department of Automatic Control, Lund University. 2019. https://www.control.lth.se/education/doctorate-program/julia-course/julia-course-2019/.
[2] Deits, Robin. “Making Robots Walk with Julia.” 7th JuliaCon, 2018. http://blog.robindeits.com/juliacon-2018.
[3] Carlson, Fredrik Bagge. “Control-Systems Analysis and Design with JuliaControl,” 9th JuliaCon, 2022. https://youtu.be/favQKOyyx4o.

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Zdeněk Hurák, Ph.D. Department of Control Engineering FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **03.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

doc. Ing. Zdeněk Hurák, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

I would like to thank doc. Ing. Zdeněk Hurák, Ph.D. for being such a great supportive and patient supervisor. Furthermore, I would like to express my thanks to my family and friends and especially to one extremely close person that has been always by my side no matter what.

Declaration

I declare that I completed the presented thesis independently and that all used sources are quoted in accordance with the Methodological instructions that cover the ethical principles for writing an academic thesis.

Prague, 26th of May, 2023

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o držování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 26. května 2023

Abstract

This thesis explores the use of Julia programming language for the automatic control of the laboratory model, with the ball and hoop model to be more specific. The demonstration will include not only the design process but also an implementation on the real laboratory model all using Julia language with the packages dedicated to an individual task. The first part is going to describe the real laboratory model. In the second part, the modeling of the system is discussed followed by linearization and analysis of the linearized model. The next part deals with a controller design followed by an implementation of the designed controller on a real model. Finally, the measured results are discussed in the last part.

Keywords: Julia, control algorithms, controller design, modeling, simulations

Supervisor: doc. Ing. Zdeněk Hurák, PhD.

Abstrakt

Tato práce se zabývá využitím jazyku Julia pro automatické řízení laboratorního modelu, konkrétně modelu kuličky v obruči. V této práci bude představen nejen proces návrhu řízení, ale také následná implementace na reálném laboratorním modelu vše za použití Julie a jednotlivých balíčků. V první části je popsán reálný laboratorní model. Ve druhé části je diskutováno modelování laboratorního modelu s následnou linearizací a analýzou tohoto lineárního modelu. Další část se zabývá návrhem regulátorů s následnou implementací navrženého regulátoru na reálném modelu. Naměřené výsledky jsou poté diskutovány.

Klíčová slova: Julia, řídicí algoritmy, návrh regulátoru, modelování, simulace

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Objectives

The goal of this thesis is to demonstrate the whole process of designing a control algorithm using the programming language Julia on a ball and hoop system. Mainly the process is going to consist of the following:

- construction of a mathematical model
- estimation of the parameters of the mathematical model based on measured experimental data
- linearization and linear analysis
- designing a proper controller
- implementation of the designed controller

Another objective of this thesis is to develop a Julia package for interaction with the ball and hoop system, providing a simple interface for the end user to simulate the system and also to design controllers and simulate them afterward.

There are various tasks to be done with this particular system (for example, looping a loop, swinging the ball on an outer hoop, etc.). The main task demonstrated in this thesis will be stabilizing the ball in a stable position.

1.2 Motivation

Julia is a fast, dynamically typed, open-source programming language developed in 2012. The primary usage purpose for Julia is scientific computations. However, a handful of people are trying to use Julia for control mainly because it is open source, unlike Matlab, which is considered a *standard* for control and control designing.

These properties mentioned above make Julia an excellent rival for Matlab in a control community and also suitable for exploring its capabilities for controlling a real physical system.

1.3 State of the art

Most of the control algorithms are designed in Matlab or Simulink followed by compilation into a binary executable file that can be run on a certain machine. Despite Julia being a fast and compiled language, almost no one tried running a Julia code instead of the binary executable file, except for one group at Lund University at the Department of Automatic Control. This group is working on a project that can be found at GitLab[1].

[1] `https://gitlab.control.lth.se/labdev/LabConnections.jl`

Chapter 2

Description of the system

The system was developed by Ing. Jiří Zemánek, PhD. and Ing. Martin Gurtner as a part of the article [5], which demonstrates numerical optimal control of the system. They also allowed building the same system for anyone because all the source files can be found at github[1].

In Figure 2.2 is depicted the block diagram of the system which consists of a Raspberry Pi 4B module acting as a central processing unit, a motor driver with a BLCD motor that communicates with the Raspberry Pi via UART, a hoop attached to the motor, and a PiCamera module with additional LED lights.

The Raspberry Pi requests the desired torque of the motor, and the motor driver responds with information about the position (angle), angular velocity, and motor current. The PiCamera on the other hand sends a detected position of the ball with a sampling frequency of 50 Hz. Therefore, the torque will act as an input to the system and the position of the ball will be an output of the system regarding the controlling task. It needs to be mentioned that setting the torque and receiving the detected position of the ball are both independent. More about communication with the motor driver and ball detection is discussed at Section 6.3 and Section 6.2.

The torque itself has boundary values of -0.7 Nm and 0.7 Nm, therefore there is an input saturation as a static nonlinearity.

[1]Source files for building `https://github.com/aa4cc/flying-ball-in-hoop/tree/3a610f4ad3c1a0d1401ce15c19b1147673f82cc7`

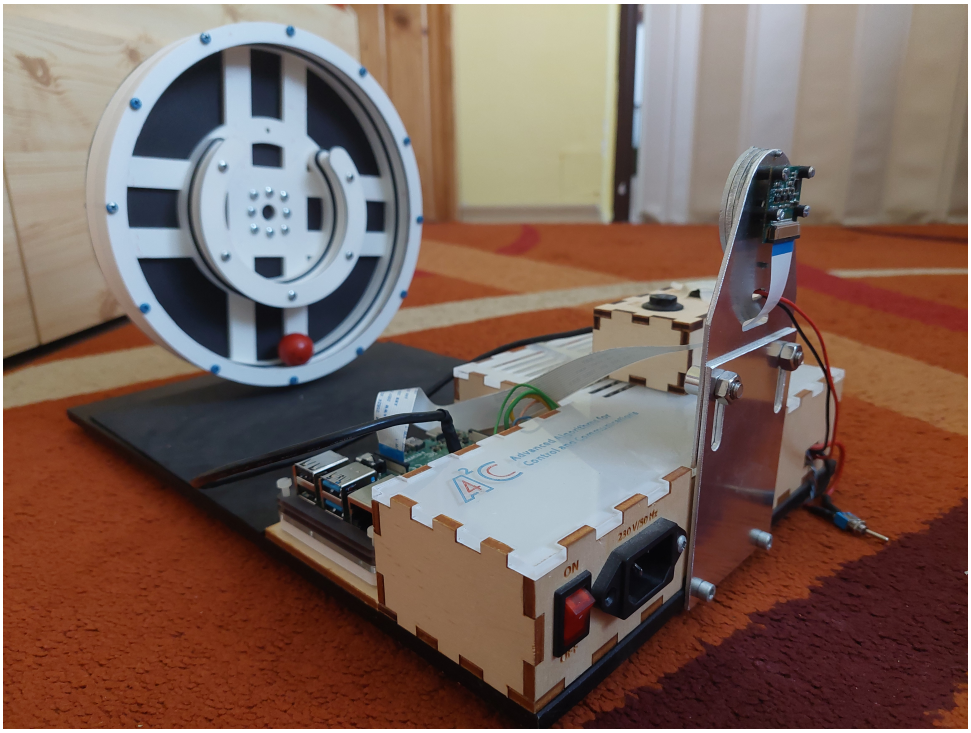


Figure 2.1: Real laboratory model.

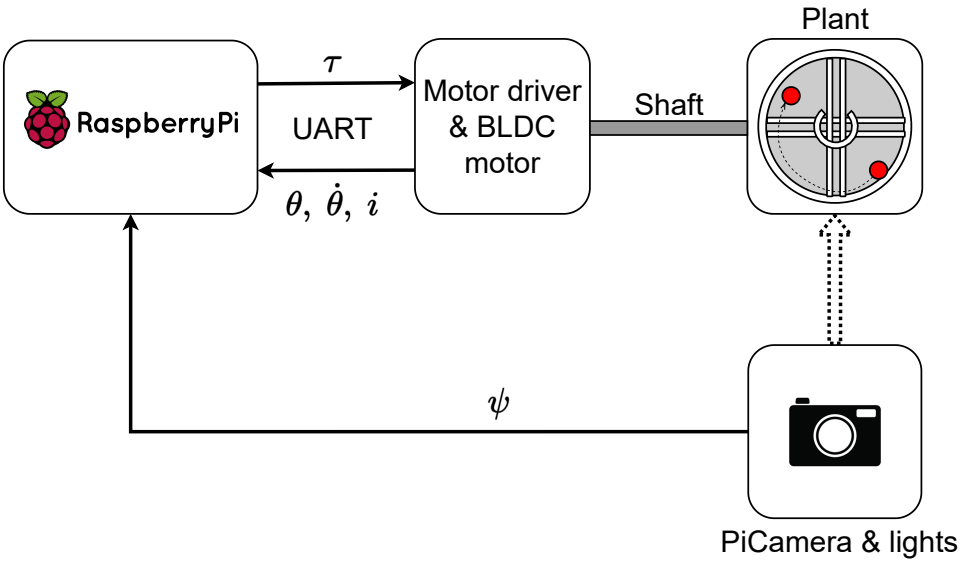


Figure 2.2: Block diagram of the laboratory model.

Chapter 3

Modeling of the system

There exists plenty of methods for modeling dynamical systems such as bond graphs, Euler-Lagrange equations and so on. The best method suited for modeling this particular system is going to be a modeling technique based on Euler-Lagrange equations since expressing an exchange of energy of this system is hardly describable for utilizing the bond graphs modeling technique.

It is crucial to mention that this system is hybrid, meaning that a system can be in a configuration, where a dynamic behavior can be expressed by ordinary differential equations, and can be transitioned to another configuration, where the dynamic behavior is expressed by other ordinary differential equations. These transitions, called guards, are in a form of a certain condition. To change the configuration, the system must meet the guards.

Nevertheless, the main task is to control the ball in a stable position, therefore, a configuration where the ball is rolling inside the bigger hoop will be considered and other configurations are going to be omitted.

There exists a certain package for modeling systems in Julia, the ModelingToolkit.jl package [8]. This package is an equation-based package for modeling a vast majority of systems. Moreover, the ModelingToolkit.jl package has a standard library package called ModelingToolkitStandardLibrary.jl[1] which allows a user to construct components and blocks for mechanical, electrical, magnetic and thermal systems. The ModelingToolkit.jl package has

[1]Official documentation `https://docs.sciml.ai/ModelingToolkitStandardLibrary/stable/`

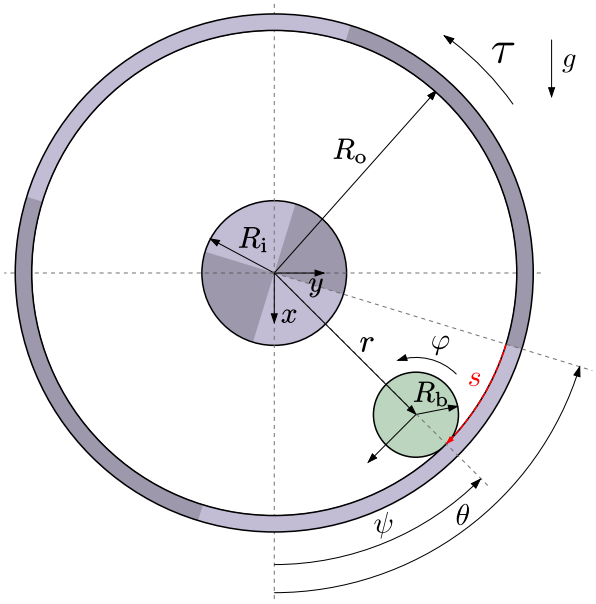


Figure 3.1: Ball and hoop sketch [9].

also embedded the Symbolics.jl package [4] which will be used to derive the Euler-Lagrange equations.

3.1 Euler-Lagrange equations

The sketch of the system is in the Figure 3.1. Firstly, a vector of generalized coordinates $\boldsymbol{q}$ needs to be determined to describe the dynamics of the system using the Euler-Lagrange formalism. A number of elements of the vector $\boldsymbol{q}$ is corresponding to a minimum number of variables required to fully describe the dynamics of the system, therefore, the generalized coordinates need to be chosen appropriately. Let $\boldsymbol{q} = \begin{bmatrix} \theta & \psi \end{bmatrix}^T$.

Secondly, the Euler-Lagrange equations

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}}\right) - \frac{\partial \mathcal{L}}{\partial \boldsymbol{q}} + \frac{\partial D}{\partial \dot{\boldsymbol{q}}} = \boldsymbol{Q} \tag{3.1}$$

need to be derived, where D denotes a dissipative force, $\boldsymbol{Q}$ denotes vector of external forces (in this case $\boldsymbol{Q} = \begin{bmatrix} \tau & 0 \end{bmatrix}^T$), because the torque τ only acts on θ, and $\mathcal{L}$ denotes a Lagrangian defined as

$$\mathcal{L}(\dot{\boldsymbol{q}}, \boldsymbol{q}, t) = \mathcal{T}^*(\dot{\boldsymbol{q}}, \boldsymbol{q}, t) - \mathcal{V}(\boldsymbol{q}, t), \tag{3.2}$$

where $\mathcal{T}^*$ is a kinetic co-energy and $\mathcal{V}$ is a potencial energy. The kinetic co-energy of the ball can be expressed as follows

$$\mathcal{T}^* = \frac{1}{2} m_b v^2 + \frac{1}{2} I_b (\dot{\varphi} + \dot{\theta})^2 + \frac{1}{2} I_h \dot{\theta}^2, \tag{3.3}$$

the potential energy can be expressed as follows

$$\mathcal{V} = -m_b g (R_o - R_b) \cos \psi \tag{3.4}$$

and the dissipative force as

$$D = \frac{1}{2} b_b \dot{\varphi}^2 + \frac{1}{2} b_h \dot{\theta}^2, \tag{3.5}$$

where m_b is the weight of the ball, v is the translational velocity of the ball, $\dot{\varphi}$ is an angular velocity of the ball, I_b is a moment of inertia of the ball, I_h is a moment of inertia of the hoop, b_b and b_h are coefficients of friction of the ball and the hoop respectively. The only remaining task is to express v and $\dot{\varphi}$ in terms of generalized coordinates θ and ψ. Since $\varphi = \frac{s}{R_b} = \frac{R_o}{R_b}(\theta - \psi)$, the angular velocity of the ball and the translational velocity of the ball are defined as:

$$\dot{\varphi} = \frac{R_o}{R_b}(\dot{\theta} - \dot{\psi}) \tag{3.6}$$

$$v = -(R_o - R_b)\dot{\psi} \tag{3.7}$$

In the following few lines of code, Julia will be used to evaluate the Equation 3.1 instead of a manual differentiating using the ModelingToolkit.jl package.

```
using ModelingToolkit
@variables t theta(t) psi(t) tau(t)
@parameters Ro Rb mb Ib Ih g bb bh

# Performs derivative of a term with respect to a wrt_var
d(term, wrt_var) = Symbolics.derivative(term, wrt_var)

# Variable definition
d_theta = d(theta, t)
d_psi = d(psi, t)
phi = (Ro / Rb) * (theta - psi)
d_phi = d(phi, t)

v = -(Ro - Rb) * d_psi
T = 0.5 * (mb * v^2 + Ib * (d_phi + d_theta)^2 + Ih * d_theta^2)
V = -mb * g * (Ro - Rb) * cos(psi)
D = 0.5 * bb * d_phi^2 + 0.5 * bh * d_theta^2

# Lagrangian and Euler-Lagrange equations
L = T - V
```

```
eq1 = d(d(L, d_theta), t) - d(L, theta) + d(D, d_theta) ~ tau
eq2 = d(d(L, d_psi), t) - d(L, psi) + d(D, d_psi) ~ 0
eq1 = simplify(eq1)
eq2 = simplify(eq2)
```

The resulting Euler-Lagrange equations can be written in the following form:

$$\boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{Q}\dot{\boldsymbol{q}} + \boldsymbol{C}\sin\psi + \boldsymbol{D}\tau = 0, \tag{3.8}$$

where

$$\boldsymbol{M} = \begin{bmatrix} I_b\left(\frac{R_o}{R_b}+1\right)^2 + I_h & -\frac{I_b R_o}{R_b}\left(\frac{R_o}{R_b}+1\right) \\ -\frac{I_b R_o}{R_b}\left(\frac{R_o}{R_b}+1\right) & m_b(R_o-R_b)^2 + \frac{I_b R_o^2}{R_b^2} \end{bmatrix}, \tag{3.9}$$

$$\boldsymbol{Q} = \begin{bmatrix} \frac{b_b R_o^2}{R_b^2} + b_h & -\frac{b_b R_o^2}{R_b^2} \\ -\frac{b_b R_o^2}{R_b^2} & \frac{b_b R_o^2}{R_b^2}, \end{bmatrix} \tag{3.10}$$

$$\boldsymbol{C} = \begin{bmatrix} 0 \\ g m_b (R_o - R_b) \end{bmatrix}, \tag{3.11}$$

$$\boldsymbol{D} = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \tag{3.12}$$

3.2 State equations

For a further procedure, it is useful to convert the Equation 3.1 to a state space representation in a general form of

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t) \tag{3.13}$$

$$\boldsymbol{y} = \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{u}, t) \tag{3.14}$$

Julia does not yet offer a conversion from Euler-Lagrange equations to the state space representation or the ability to express a variable from a certain expression. Therefore, the conversion must be done manually with the small help of the Symbolics.jl package. Since the dynamics of the model is described by two second-order differential equations, the $\boldsymbol{x}$ is going to consist of 4 elements. The natural choice of $\boldsymbol{x}$ and y will be $\boldsymbol{x} = \begin{bmatrix} \theta & \dot{\theta} & \psi & \dot{\psi} \end{bmatrix}^T$ and $y = \psi$.

The state space representation can be obtained from Equation 3.8 by multiplying the whole equation by the inverse of the matrix $\boldsymbol{M}$. Thus the state space representation can be written like this:

$$\ddot{\boldsymbol{q}} = \underbrace{-\boldsymbol{M}^{-1}\boldsymbol{Q}}_{\boldsymbol{J}}\dot{\boldsymbol{q}}\underbrace{-\boldsymbol{M}^{-1}\boldsymbol{C}}_{\boldsymbol{K}}\sin\psi\underbrace{-\boldsymbol{M}^{-1}\boldsymbol{D}}_{\boldsymbol{L}}\tau \tag{3.15}$$

The obtained state space representation is described by the following set of ordinary differential equations.

$$\frac{\mathrm{d}}{\mathrm{d}t}\theta = \dot{\theta} \tag{3.16}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\dot{\theta} = J_{11}\dot{\theta} + J_{12}\dot{\psi} + K_1\sin\psi + L_1\tau \tag{3.17}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\psi = \dot{\psi} \tag{3.18}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}\dot{\psi} = J_{21}\dot{\theta} + J_{22}\dot{\psi} + K_2\sin\psi + L_2\tau \tag{3.19}$$

$$y = \psi \tag{3.20}$$

In order to create the ball and hoop system in Julia, the user can use a function called *BallAndHoop* in the BallAndHoopSystem.jl package which returns an ODESystem. The user can additionally specify an operating point, in which a linearization will be performed, as well as a vector of physical parameters of the system.

The following code shows the function for creating the ball and hoop system in Julia. It is important to note that the parameters are initialized to specific values that were obtained as a result of an optimization process. The optimization process is going to be discussed in the next section.

```
function ball_and_hoop(; name, OP=zeros(4))
  @variables t
  # Create input and output structures
  @named input = RealInput()
  @named output = RealOutput()

  # Saturation block
  @named saturation = Limiter(y_max=0.7)

  # Define state variables and initialize them to the OP
  @variables theta(t) = OP[1]
  @variables d_theta(t) = OP[2]
  @variables psi(t) = OP[3]
  @variables d_psi(t) = OP[4]
```

```
  # Define parameters of the ODEs and initialize them
  @parameters L1 = 586.3694971455833 [tunable = true]
  @parameters J12 = 0.06769766279210704 [tunable = true]
  @parameters J11 = -0.4818896328354544 [tunable = true]
  @parameters K1 = -5.119508012862879 [tunable = true]
  @parameters J22 = -0.33509889971201284 [tunable = true]
  @parameters J21 = -0.0032686776857956444 [tunable = true]
  @parameters L2 = 210.1200001949957 [tunable = true]
  @parameters K2 = -73.63359905056758 [tunable = true]
  dt = Differential(t)

  # Vector of states and parameters
  states = [theta, d_theta, psi, d_psi]
  parameters = [L1, J12, J11, K1, J22, J21, L2, K2]

  # State space equations
  eqs = [
    dt(theta) ~ d_theta
    dt(d_theta) ~ J11 * d_theta + K1 * sin(psi) + J12 * d_psi +
                  L1 * saturation.y
    dt(psi) ~ d_psi
    dt(d_psi) ~ J21 * d_theta + K2 * sin(psi) + J22 * d_psi +
                L2 * saturation.y
    output.u ~ psi
    saturation.u ~ input.u
  ]

  # Creating ODESystem from equations, independent variables,
  # states and parameters
  sys = ODESystem(eqs, t, states, parameters; name=name)

  # Composing created ODESystem with other ODESystem blocks
  compose(sys, [input, output, saturation])
end
```

3.3 Parameters estimation

Parameter estimation or parameter identification inherently belongs to a modeling procedure. Having a general description of a system is not enough, therefore, it is desired to have a particular description of a system for further analysis and control design.

The presented approach of estimation of parameters is an optimization-based approach. The overall goal is to identify the physical parameters of the system based on measured data from the experiments. As Figure 2.2 depicts, the only measured variables are $\theta, \dot{\theta}, \psi$ with a known input τ. These data were provided in a .mat file and converted into a HDF5 file.

Firstly, using the packages DiffEqParamEstim.jl [2] with support from Optimisation.jl [3] and OptimisationOptimJL.jl[3], the entries of the matrices $\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}$ are estimated and then the physical parameters are calculated from these entries by solving a set of nonlinear equations. Since all the physical parameters are positive, the problem is formulated as a nonlinear program (NLP). The popular JuMP.jl [7] package is used here to formulate the NLP, with the NLopt.jl [6] package providing a variety of solvers.

3.3.1 Estimation of parameters of the equations

The following optimization task can be formulated as follows:

$$\min_{\boldsymbol{J},\boldsymbol{K},\boldsymbol{L}} \quad \|\boldsymbol{g}(\boldsymbol{J},\boldsymbol{K},\boldsymbol{L})\|^2, \tag{3.21}$$

where

$$\boldsymbol{g}(\boldsymbol{J},\boldsymbol{K},\boldsymbol{L}) = \underbrace{\boldsymbol{x}(\boldsymbol{J},\boldsymbol{K},\boldsymbol{L})}_{\text{solution of ODEs}} - \underbrace{\hat{\boldsymbol{x}}}_{\text{measured data}}. \tag{3.22}$$

The equations 3.16 - 3.19 show that only the equations 3.17 and 3.19 depends on the matricies $\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}$ hence the measured data will be considered $\dot{\theta}$ and $\dot{\psi}$. Since $\dot{\psi}$ is not directly measurable, the ψ is going to be considered instead of $\dot{\psi}$ because ψ and $\dot{\psi}$ are tied together and ψ is also measurable.

To evaluate the objective function which is going to be minimized, a solution of ODEs is required. In other words, the simulation of the system is mandatory. The user can utilize the function *load_hdf5_data* to load the data from experiments.

The structure of the data is the following: the file has groups called 'ExpN', where N is a number of an experiment. In this case, the file has two groups - 'Exp1' and 'Exp2'. Each group has fields that store a sequence of input and states and also timestamps when these variables were sampled.

[2] Official documentation `https://docs.sciml.ai/DiffEqParamEstim/stable/`

[3] Official documentation `https://docs.sciml.ai/Optimization/stable/optimization_packages/optim/`

The function below shows how loading a HDF5 file is implemented and it is available in the BallAndHoop.jl package.

```
function load_hdf5_data(file::String, experiment::String)
  fid = h5open(file, "r")
  group = fid[experiment]
  timestamps = read(group["Timestamps"])
  states = read(group["States"])
  input = read(group["Input"])
  return timestamps, states, input
end
```

As stated before, the packages DiffEqParamEstim.jl, Optimization.jl and OptimizationOptimJL.jl are going to be used for the optimization.

```
using BallAndHoop, ModelingToolkit, DifferentialEquations
using ModelingToolkitStandardLibrary.Blocks
using DiffEqParamEstim, Optimization, OptimizationOptimJL
```

The next step is to load the measured data from the experiments and modify them for future usage.

```
path = "./Experiments"
experiment = "Exp1" # Choosing experiment no. 1

# Loading measured data from experiment 1
timestamps, states, input_data = load_hdf5_data(path, experiment)

# Choosing appropriate variables
data = states[:,2:3]
```

After loading the data, an ODESystem representing the ball and hoop system is initialized. It is required to provide an input to the system from loaded data via a user-defined function. For this case, a block called TimeVaryingFunction from ModelingToolkitStandardLibrary is used. The function will simply index the loaded input vector based on a current time of a simulation. Since conversion to an integer is used, it is necessary to tell a solver that it should treat that function as symbolic. The reason is that a conversion of a symbolic variable to an integer is not a valid operation, hence the macro *@register_symbolic* is used.

```
sampling_freq = 50 # Frequency in Hz

# Providing input values based on a current time of a simulation
u(t) = input_data[Int(floor(sampling_freq*t))+1]
@register_symbolic u(t)

# Creating an instance of the ball and hoop system
@named nonlinear_sys = ball_and_hoop()

# Creating a function block
@named source = TimeVaryingFunction(u)
```

After doing so, a connection between the system and the source block needs to be established. The connection of two or more ODESystems is described in the following lines of code.

```
# Vector of equations representing certain connections
connections = [
  nonlinear_sys.input.u ~ source.output.u
]

# Creating an ODESystem from nonlinear_sys and source
# ODESystems
@named sim_loop = ODESystem(connections, t,
                  systems=[nonlinear_sys, source])

# Using structural simplify for simplification of the system
sim_loop = structural_simplify(sim_loop)
```

The penultimate step is creating an ODEProblem which is passed into an ODE solver and defining an objective function.

```
# Initial condition from the measured data
init_cond = zeros(4)
init_cond[1:3] = states[1,:]

# Timespan from the measured data
tspan = (timestamps[1], timestamps[end])

# Creating ODEProblem from ODESystem, providing
# an initial condition and a timespan
prob = ODEProblem(sim_loop, init_cond, tspan)
```

```
loss = L2Loss(timestamps, data')

objective_fun = build_loss_objective(prob, Rosenbrock23(), loss,
                Optimization.AutoForwardDiff(), maxiters=1e7,
                save_idxs=[2,3])
```

The *build_loss_objective* function calls an ODESolver (in this case, Rosenbrock23, which is a solver for stiff ODEs) and computes an L2 loss given by an Equation 3.22. The *Optimization.AutoForwardDiff* tells how gradients are going to be calculated, and the *save_idxs* means that only solution at indices 2 and 3 ($\dot{\theta}$ and ψ) are going to be used in the loss function evaluation.

The last step is formulating and solving an optimization problem using a particular numerical algorithm. Since the parameters of the matrices $\boldsymbol{J}$, $\boldsymbol{K}$, $\boldsymbol{L}$ were given, these are going to be an initial guess for the numerical algorithm.

```
init_params = [586.3695, 0.0677, -0.48, -5.1195, -0.3351,
               -0.0034, 210.12, -73.6336]

# Creating an optimization problem
optprob = Optimization.OptimizationProblem(objective_fun,
          init_params)

# Solving the optimization problem
optsol = solve(optprob, LBFGS())
```

The *optsol* is a minimizer to the optimization problem. All these parameters are already included in the *ball_and_hoop* function. During the optimization, a few things need to be mentioned.

1. Using a *standard* ODE solver, *Tsit5*, well known as an *ode45* in Matlab, led to an instability of the optimization. Using a stiff solver solved the problem.

2. Many numerical optimization algorithms were tested, and the LBFGS algorithm was the best suited for this task (it converged relatively quickly and was stable).

3.3.2 Estimation of the physical parameters

Since all the matrix entries were estimated, it is time to extract physical parameters from them by solving a set of nonlinear equations. Let $\boldsymbol{\lambda} = \begin{bmatrix} m_b & R_o & R_b & I_b & I_h & b_b & b_h \end{bmatrix}^T$. The problem that needs to be solved can be written in the form of $\boldsymbol{h}(\boldsymbol{\lambda}) = \mathbf{0}$.

Since the problem consists of eight equations with seven variables (g is a well-known constant), the set of equations is overdetermined. Therefore the solving of a set of nonlinear equations will turn into an optimization problem defined as:

$$\min_{\boldsymbol{\lambda}} \quad \|\boldsymbol{h}(\boldsymbol{\lambda})\|^2 \tag{3.23}$$

$$\text{s.t.} \quad \lambda_i > 0, \qquad \forall i = 1, 2, \ldots, 7, \tag{3.24}$$

where

$$\boldsymbol{h}(\boldsymbol{\lambda}) = \underbrace{\boldsymbol{\gamma}(\boldsymbol{\lambda})}_{\substack{\text{evaluated} \\ \text{matrix} \\ \text{entries}}} - \underbrace{\hat{\boldsymbol{\gamma}}}_{\substack{\text{estimated} \\ \text{matrix} \\ \text{entries}}} . \tag{3.25}$$

As mentioned, the packages JuMP.jl and NLopt.jl will be used for the optimization. The NLopt.jl package is a wrapper to NLopt library.

```
using JuMP, NLopt, LinearAlgebra
```

The JuMP.jl package uses a type called *Model* to store all necessary information about an optimization problem. The following few lines of code show how such a Model can be initialized.

```
# Initializing the JuMP model with the NLopt optimizer
model = Model(NLopt.Optimizer)

# Setting a numerical algorithm for the optimization
set_optimizer_attribute(model, "algorithm", :LD_LBFGS)

# Defining all the variables of the optimization problem
# with a constraint and initial guess
@variable(model, mb >= 0, start = 60.8e-3)
@variable(model, bh >= 0, start = 0.0075)
@variable(model, bb >= 0, start = 2.5737e-6)
@variable(model, Ro >= 0, start = 0.0957)
```

```
@variable(model, Rb >= 0, start = 0.0104)
@variable(model, Ih >= 0, start = 0.0015)
@variable(model, Ib >= 0, start = 3.679e-6)
```

The next step is to define an objective function that is going to be minimized. In this case, the custom user-defined function will be defined.

```
function objective(lambda::T...) where {T<:Real}
  # Splitting lambda vector
  mb = lambda[1]
  Ro = lambda[2]
  Rb = lambda[3]
  Ib = lambda[4]
  Ih = lambda[5]
  bb = lambda[6]
  bh = lambda[7]

  # Creating matrices
  M = [Ib*(Ro/Rb+1)^2+Ih -Ib*Ro/Rb*(Ro/Rb+1);
      -Ib*Ro/Rb*(Ro/Rb+1) mb*(Ro-Rb)^2+Ib*Ro^2/Rb^2]
  Q = [bb*Ro^2/Rb^2+bh -bb*Ro^2/Rb^2;
      -bb*Ro^2/Rb^2 bb*Ro^2/Rb^2]
  C = [0; 9.81*mb*(Ro-Rb)]
  D = [-1; 0]

  # Computing inverses and loss
  try
    J = -M \ Q
    K = -M \ C
    L = -M \ D

    gamma = [J[1, 1]; J[1, 2]; J[2, 1]; J[2, 2];
             K[1]; K[2]; L[1]; L[2]]

    gamma_hat = [-0.48; 0.0677; -0.0034; -0.3351;
                 -5.1195; -73.6336; 586.3695; 210.12]

    loss = norm((gamma .- gamma_hat))^2
    return loss

  # Returning large number when inverses do not exist
  catch e
    return Float64(99999999999.0)
  end
end
```

Physical parameter	Value	Units
m_b	60.80000008575297	g
R_o	9.570000006211415	cm
R_b	1.0400000535067755	cm
I_b	$2.4963218878738077 \cdot 10^{-6}$	$\mathrm{kg} \cdot \mathrm{m}^2$
I_h	0.0015000089046344925	$\mathrm{kg} \cdot \mathrm{m}^2$
b_b	$2.5737738382893808 \cdot 10^{-6}$	$\mathrm{N} \cdot \mathrm{s} \cdot \mathrm{m}^{-1}$
b_h	0.0074999999947648515	$\mathrm{N} \cdot \mathrm{s} \cdot \mathrm{m}^{-1}$

Table 3.1: Estimated physical parameters

The objective function evaluates matrices $\boldsymbol{J}$, $\boldsymbol{K}$, $\boldsymbol{L}$ based on current value of $\boldsymbol{\lambda}$ and then computes squared an L2 norm of function $\boldsymbol{h}(\boldsymbol{\lambda})$. The whole computation is in a try-catch block to prevent singular matrices. If some of the matrices are singular, the penalization will be performed by returning a large value of the objective function.

The last step is to pass the objective function to the model and call the *JuMP.optimize!* function to start the optimization. Since the objective function is user-defined, it is required to tell to register it with the *register* function in this case. The passed arguments to this function are a model, a symbol (under which the objective function will be represented, denoted with a colon symbol), a number of input arguments and the function itself.

```
# Register a user-defined objective function
register(model, :objective, 7, objective; autodiff=true)

# Defining a objective -> minimizing registered function
@NLobjective(model, Min, objective(mb, Ro, Rb, Ib, Ih, bb, bh))

# Performing an optimization
JuMP.optimize!(model)
```

The exclamation mark in the *JuMP.optimize!* function tells that this function modifies the input argument. For checking the value of a certain variable, the user can call the function *value* to display the value of the variable.

```
print("Mass of the ball is " * string(value(mb)) * " kg.")
```

```
Mass of the ball is 0.06080000008575297 kg.
```

The Table 3.1 shows all physical parameters and their values obtained by the optimization.

3.4 Simulation of the nonlinear system

The simulation was already performed in the Section 3.3.1, where the simulation was required to optimize the parameters of the state equations. However, the solution of the state equation to the given input was not visualized.

In this section, the simulation process will be discussed in terms of how to simulate an ODESystem in Julia, solver choice and comparison and comparison of the simulation with the measured experiments.

The simulation of an ODESystem in the Julia process is as follows:

1. Create a TimeVaryingFunction block.
2. Define connections (in this case connect an output of the TimeVaryingFunction block with an input of an ODESystem).
3. From these connections create a new ODESystem.
4. Structural simplification of the ODESystem.
5. Define an initial condition and a timespan of the simulation.
6. Create an ODEProblem from the simplified ODESystem, initial condition and timespan.
7. Call the *solve* function with a particular solver.

This process can be for the end user a bit impractical, therefore the BallAndHoop package comes with the function *simulate_system*. This function does all mentioned above. The user passes the following arguments to the function: an ODESystem that is going to be simulated, a function acting as an input to the given system, an initial condition, a timespan of the simulation and a solver. It is important to note that if the input is a vector of sampled input, the *@register_symbolic* macro must be used. The code below demonstrates all steps mentioned above encapsulated into one function that the user can call.

```
function simulate_system(sys::ODESystem, f::Any,
                         init_cond::Vector, tspan::Tuple,
                         solver::Any)
```

```
  @named src = TimeVaryingFunction(f)
  connections = [
    sys.input.u ~ src.output.u
  ]
  @named sim_loop = ODESystem(connections, Blocks.t,
                              systems=[src, sys])
  sim_loop = structural_simplify(sim_loop)
  sim_problem = ODEProblem(sim_loop, init_cond, tspan)
  result = solve(sim_problem, solver)
  return result
end
```

For the visualization, the Plots.jl [2] package is going to be used. This package is by far the most used package for plotting. It provides many backends such as GR, PlotlyJS, PythonPlot and so on. The default backend is set to GR and it will be used for plotting. The LaTeXStrings.jl [4] package is included for passing LaTeX-like syntax as labels and axis titles.

The default plot looks a bit empty and not so great to the eye, therefore a variable *plot_settings* is defined and passed as an argument to all *plot* calls. This variable is then passed to all plotting functions followed by a splatting operator denoted as ... which unpacks values from a tuple and passes them as regular arguments.

```
using Plots, LaTeXStrings

# Setting a font and making a minor grid visible
plot_settings = (
    fontfamily="Computer Modern",
    minorgrid=true
)
```

The function *simulate_system* returns an ODESolution that encapsulates a vector of timestamps and a vector of vectors of the individual solutions. For plotting an individual solution, the key argument *idxs* is passed to indicate which variables are going to be plotted.

The DifferentialEquations.jl [10] offers plenty of solvers, the most used are *Tsit5* and *Rosenbrock23*. Both of them have a Matlab equivalent (*ode45* and *ode23s* respectively). The *Tsit5* solver is not meant to be used for stiff equations whereas *Rosenbrock23* is meant to be used for stiff equations.

[4] GitHub repository at `https://github.com/JuliaStrings/LaTeXStrings.jl`

The following few lines demonstrate how to simulate an ODESystem and visualize the result of the simulation. In the resulting plots, an input signal to the system will be plotted as well as the measured data from the first experiment with the simulated results obtained by the two mentioned solvers.

```
plot(timestamps,input_data,label="",
     xguide=L"t \ [\mathrm{s}]",
     yguide=L"\tau \ [\mathrm{Nm}]";
     plot_settings...)
```

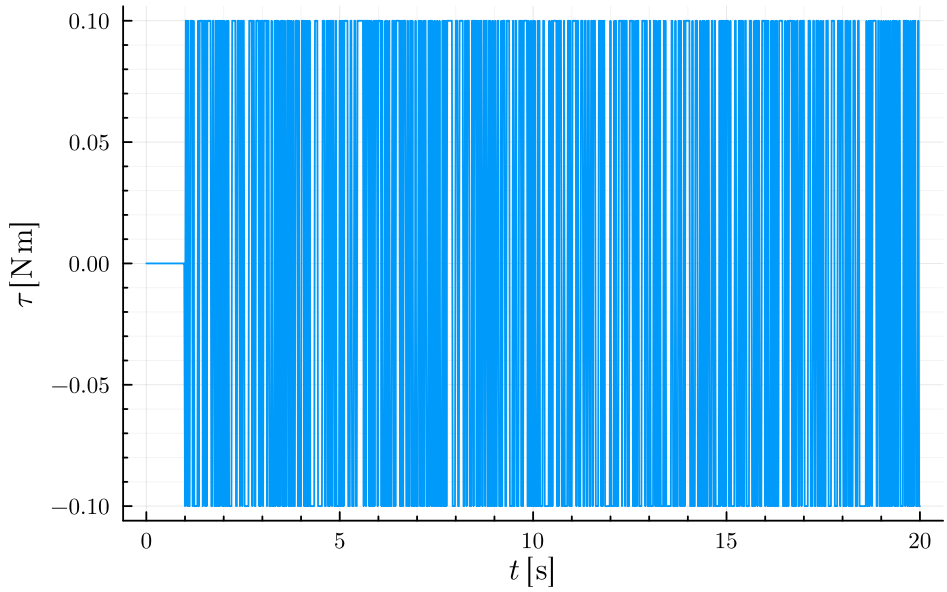


Figure 3.2: Input from the first experiment.

```
# Simulation with the Rosenbrock23 solver
sim_res_rb = simulate_system(nonlinear_sys, u, init_cond,
                             tspan,Rosenbrock23())

# Simulation with the Tsit5 solver
sim_res_tsit = simulate_system(nonlinear_sys, u, init_cond,
                               tspan, Tsit5())

# Plotting both simulations with the measured data
plot(sim_res_rb, idxs=[1], label="Rosenbrock23",
     yguide=L"\theta \ [\mathrm{rad}]",
     legend=:bottomleft;
     plot_settings...)

plot!(sim_res_tsit, idxs=[1], label="Tsit5",
      xguide=L"t \ [\mathrm{s}]")
```

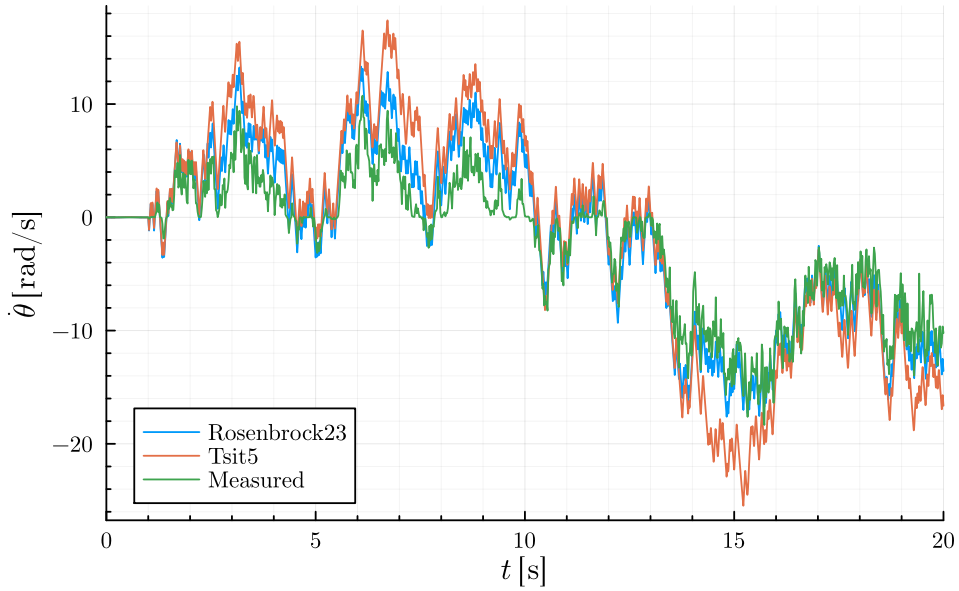


Figure 3.4: Angular velocity of the hoop from the first experiment.

```
plot!(timestamps,states[:,1],label="Measured")
```

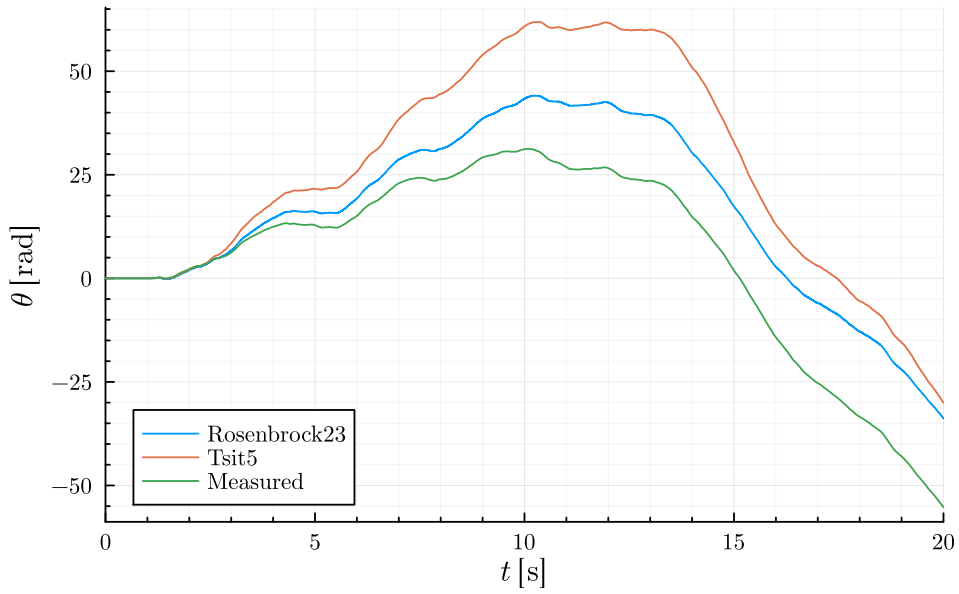


Figure 3.3: Angle of the hoop from the first experiment.

The Figure 3.3 and Figure 3.4 illustrate a significant difference between using the *Tsit5* and *Rosenbrock23* solver. This difference will be reflected

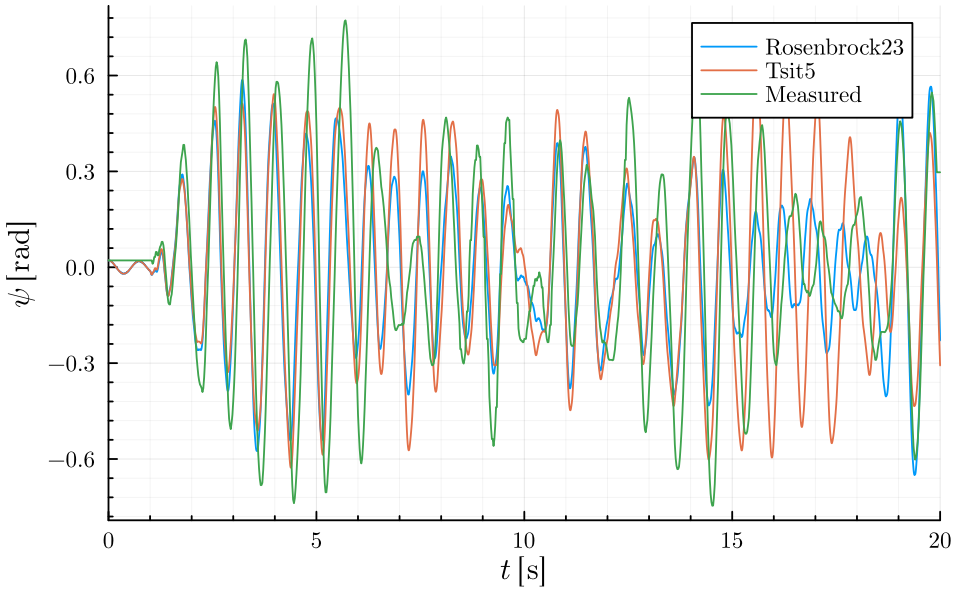


Figure 3.5: Deflection angle of the ball from the first experiment.

when the simulation is going to be conducted using the input from the second experiment. Therefore it is not caused by the particular input to the system.

The Figure 3.6 - Figure 3.9 show the performance of both solvers with the input function provided from the data from the second experiment.

All these simulations show that a proper choice of a solver is mandatory not only for the optimization in this case but also for the simulation of a closed-loop nonlinear system with a controller.

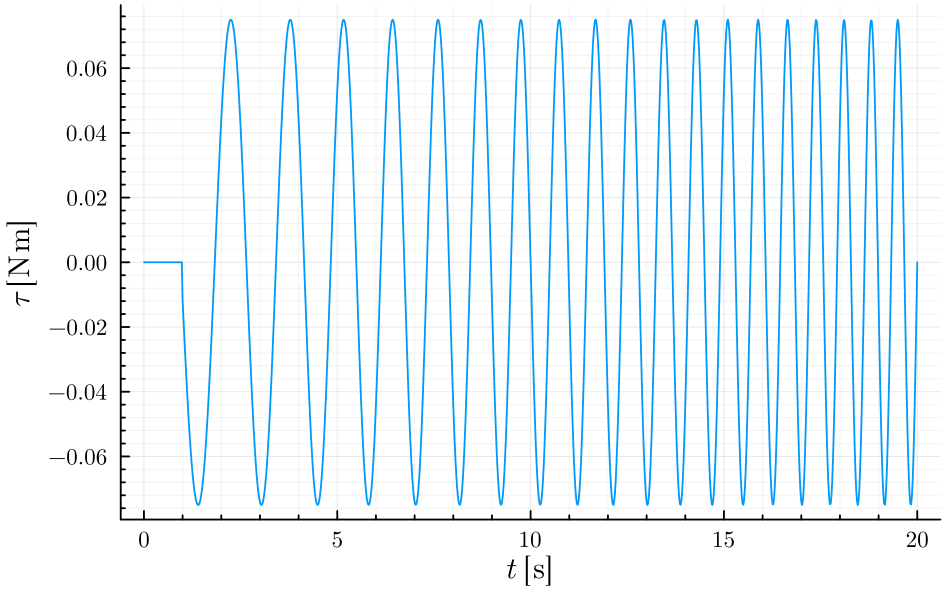


Figure 3.6: Input from the second experiment.

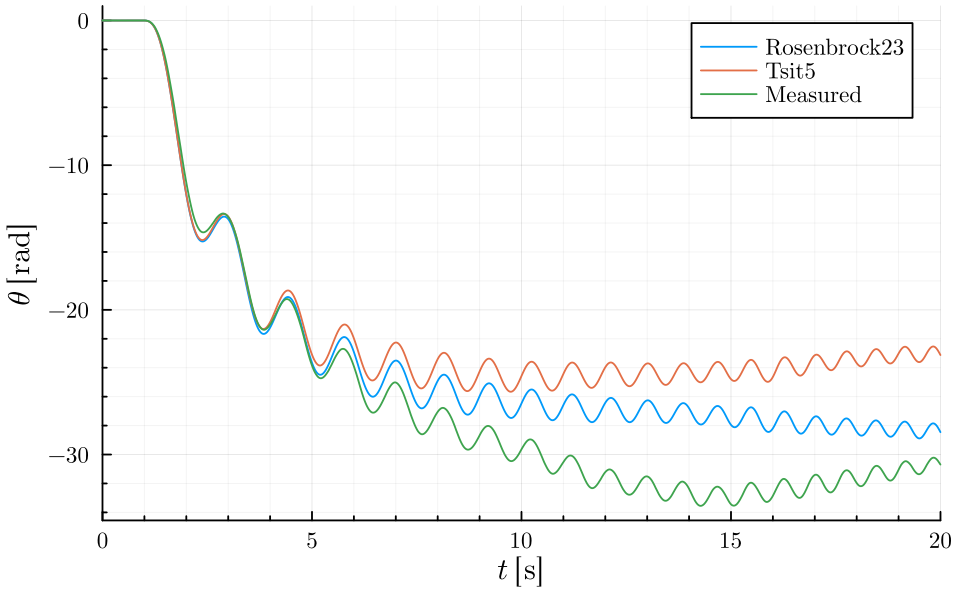


Figure 3.7: Angle of the hoop from the second experiment.

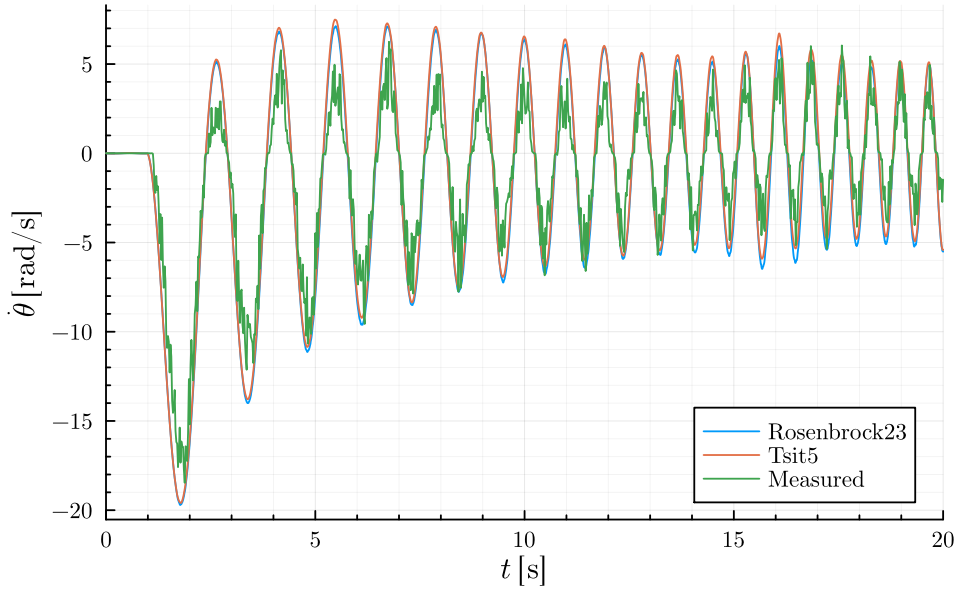


Figure 3.8: Angular velocity of the hoop from the second experiment.

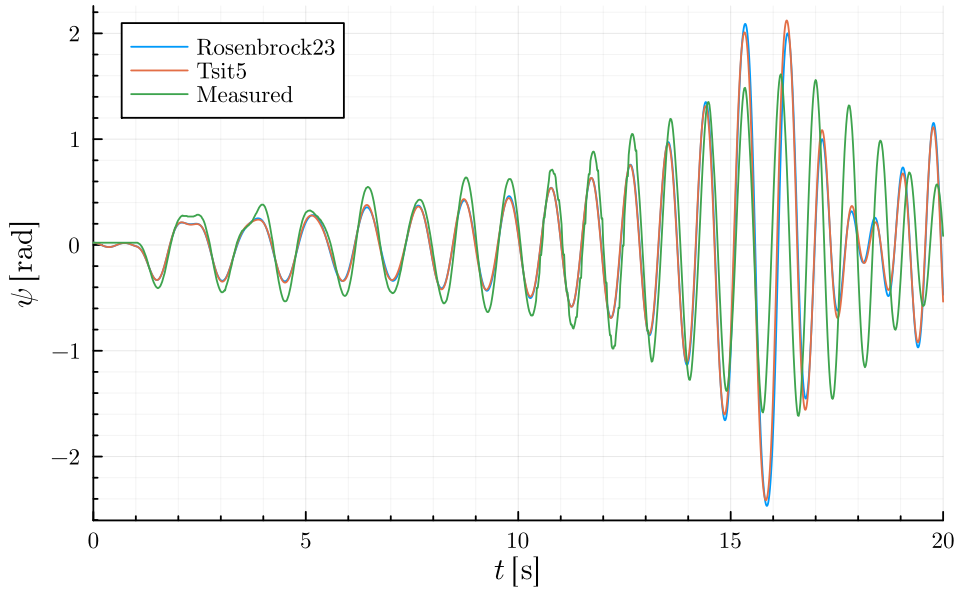


Figure 3.9: Deflection angle of the ball from the second experiment.

Chapter 4

Linearization and linear analysis

Before the design of a controller in Julia will be discussed, it is useful to linearize a given system in a so-called operating point. This will allow using a linear state space representation of a system, a transfer function representation of a system and analysis tools such as frequency response, location of poles and zeros, and so on.

The package ControlSystems.jl [1] is going to be utilized for linear analysis and controller design. For convenience, there exists a package called ControlSystemsMTK [1] which provides an interface between ControlSystems.jl and ModelingToolkit.jl

4.1 Linearization

ModelingToolkit.jl package is shipped with a function called *linearize* which linearizes an ODESystem. However, a better way to linearize a system is to call the function *named_ss* with an ODESystem, input and output as arguments.

This function will perform linearization at the operating point (which can be set by passing an argument to the function *ball_and_hoop*) and return a

[1] Official documentation `https://juliacontrol.github.io/ControlSystemsMTK.jl/dev/`

NamedStateSpace type.

The *named_ss* function is going to give certain inputs and outputs strange names, this can be resolved by calling the *ss* function on the linearized system to get rid of the names.

```
using ControlSystems, ControlSystemsMTK

# Unpacking the input and output from the system
@unpack input, output = nonlinear_sys

# Linearization
lsys = ss(named_ss(nonlinear_sys, [input.u], [output.u]))
```

```
ControlSystemsBase.StateSpace{Continuous, Float64}
A =
  0.0   1.0                      0.0                  0.0
  0.0  -0.4818896328354544      -5.119508012862879    0.06769766279210704
  0.0   0.0                      0.0                  1.0
 -0.0  -0.0032686776857956444  -73.63359905056758    -0.33509889971201284
B =
   0.0
 586.3694971455833
   0.0
 210.1200001949957
C =
 0.0  0.0  1.0  0.0
D =
 0.0

Continuous-time state-space model
```

4.2 Linear analysis

The list of all analysis tools offered by ControlSystems.jl can be found in the official documentation with an example of use.

The state space representation is already obtained by the linearization process. Another way to represent a linear system is by a transfer function

or zero-pole-gain representation. The transfer function can be acquired by calling the *tf* function.

```
tf(lsys)
```

```
TransferFunction{Continuous, ControlSystemsBase.SisoRational{Float64}}
  1.4210854715202004e-14s^3 + 210.12000019499573s^2 + 99.3379968544011s
-------------------------------------------------------------------------
1.0s^4 + 0.8169885325474627s^3 + 73.79530101815311s^2 + 35.46653398922718s

Continuous-time transfer function model
```

Another useful piece of information is about the location of poles and zeros. Poles and zeros can be obtained by calling the functions *poles* and *tzeros* respectively. The user can plot the location as well by calling *pzmap* function.

```
poles(lsys)
```

```
4-element Vector{ComplexF64}:
                  0.0 + 0.0im
 -0.4816611437370447 + 0.0im
  -0.167663694405211 + 8.579375019784992im
  -0.167663694405211 - 8.579375019784992im
```

```
tzeros(lsys)
```

```
2-element Vector{Float64}:
 -0.4727679267190814
  0.0
```

```
pzmap(lsys;xguide="Real axis", yguide="Imaginary axis",
      markersize=5,markerstrokewidth=3, markeralpha=1,
      title="", plot_settings...)
```

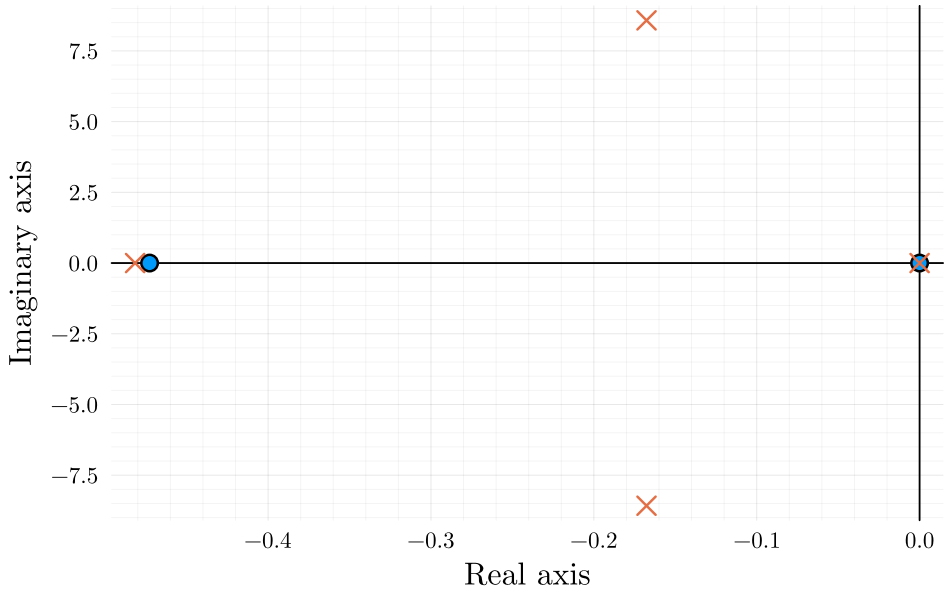


Figure 4.1: Location of poles and zeros in the complex plane.

As Figure 4.1 depicts, the system has four poles, two of them are real and the imaginary ones have a bigger imaginary part compared to the real part. It can be seen that one pole has the same value as one zero of the system and they are going to cancel out. This will affect the controllability or observability of the system which can be computed by taking a full rank of the controllability and observability matrices obtained by calling *ctrb* and *obsv* respectively.

```
# Checking controllability and observability
println(rank(ctrb(lsys)) == 4)
println(rank(obsv(lsys)) == 4)
```

```
true
false
```

The last functionality that is going to be demonstrated in the time domain is plotting a step response and extracting values like rise time, settling time and so on. The step response is obtained by the *step* function and the result is passed to the function *stepinfo*.

```
step_response = step(lsys)
plot(stepinfo(step_response);
     yguide=L"\psi \ [\mathrm{rad}]",
     xguide=L"t \ [\mathrm{s}]",
     plot_settings...)
```

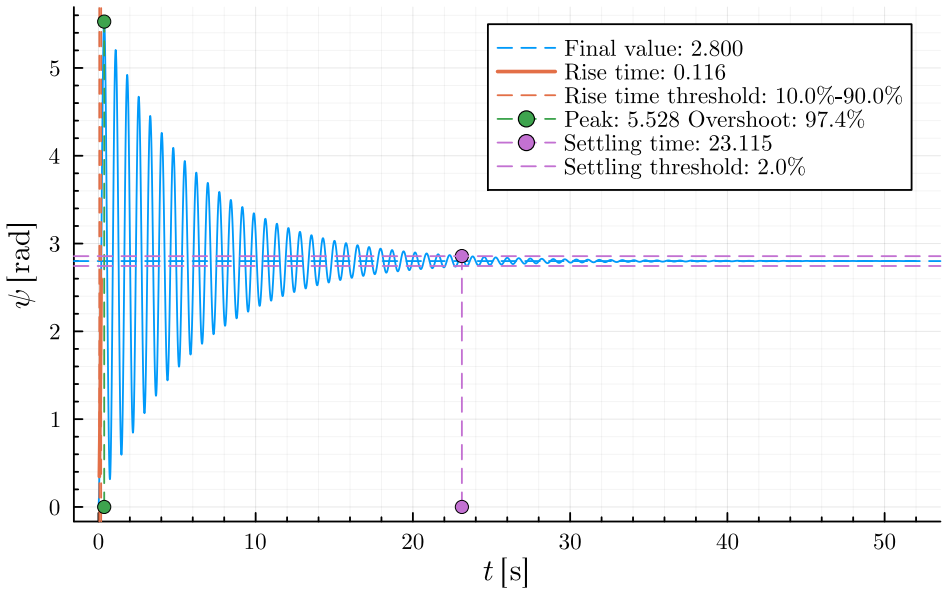


Figure 4.2: Step response of the system.

It is also important to know, how the system behaves in the frequency domain. For analysis in the frequency domain, the *bodeplot*, *nyquistplot* and *margin* can be utilized.

```
# Setting y axis units to dB
setPlotScale("dB")
bodeplot(lsys;xguide=L"\omega \ [\mathrm{rad/s}]", label="",
         yguide=[L"A \ [\mathrm{dB}]" L"\varphi \ [°]"],
         plot_settings...)

# Plotting line at -3 dB and bandwidth frequency
plot!([-3],color=:red, width=0.1,
      label="", seriestype="hline")
plot!([19.25],color=:red, width=0.1,
      label="", seriestype="vline")
```

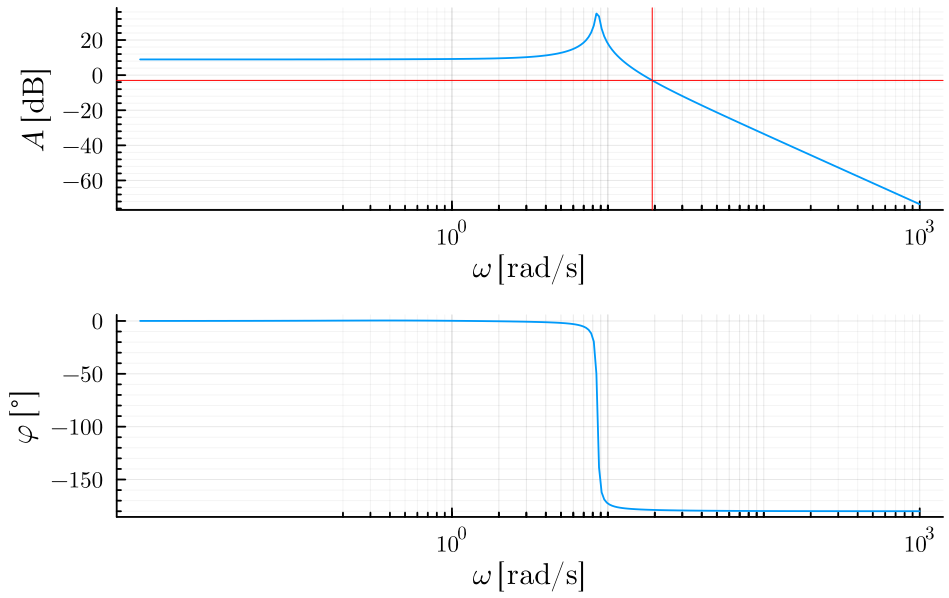


Figure 4.3: Bode plot of the system.

```
nyquistplot(lsys, unit_circle=true; xguide="Real axis",
            yguide="Imaginary axis", title="", label="",
            xlims=[-3,3], plot_settings...)
```

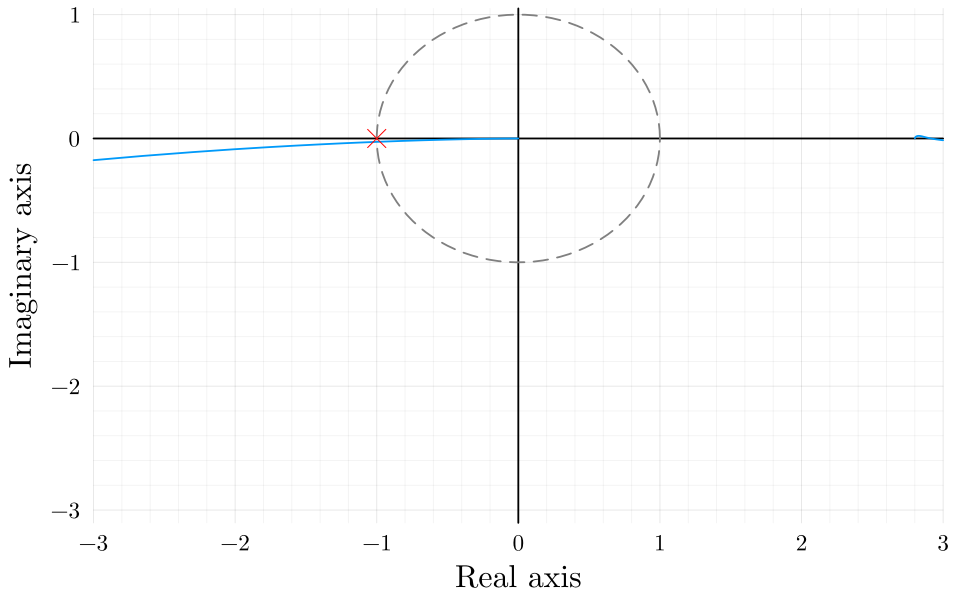


Figure 4.4: Nyquist diagram of the system.

From the Figure 4.3 can be analyzed the bandwidth of the system, in this scenario the bandwidth $\omega_b \approx 19.5$ rad/s.

The last thing to analyze is the gain and phase margins. The *margin* function returns a tuple of matrices of the frequency at which is gain margin occurs, the value of the gain margin and these values for the phase margins as well.

```
wgm, gm, wpm, pm = margin(lsys)
```

```
Gain margin is Inf at NaN rad/s.
Phase margin is 1.570004618148971° at 16.845711769073052 rad/s.
```

4.3 Simulation of the linearized model

For the simulation of the linear model, the *simulate_system* is going to be used. However, the function accepts an ODESystem type, not a StateSpace type. The ControlSystemsMTK offers not only a conversion from an ODESystem to StateSpace but also a conversion from a StateSpace type to an ODESystem type by simply passing a variable of type StateSpace as an argument to the *ODESystem* function.

```
@named linear_sys = ODESystem(lsys)
sol1 = simulate_system(linear_sys, u, init_cond, tspan,
                       Rosenbrock23())
sol2 = simulate_system(nonlinear_sys, u, init_cond, tspan,
                       Rosenbrock23())
plot(sol1, idxs=[3],label="Linear system",
     yguide = L"\psi \ [\mathrm{rad}]";plot_settings...)
plot!(sol2, idxs=[3],label="Nonlinear system",
      xguide = L"t \ [\mathrm{s}]")
plot!(timestamps,states[:,3],label="Measured")
```

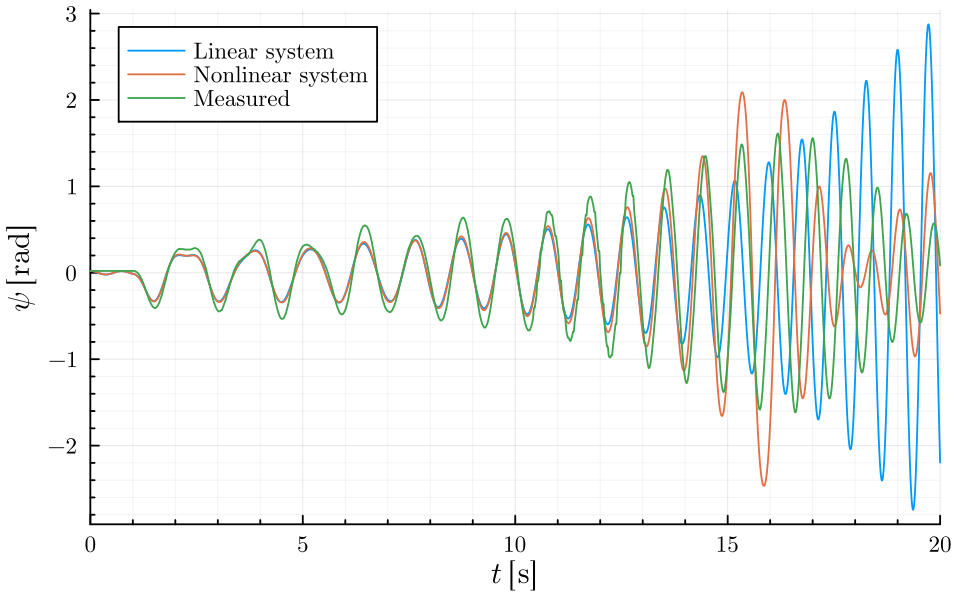


Figure 4.6: Comparison of the deflection angle of the ball for nonlinear and linear systems with the measured data from the second experiment.

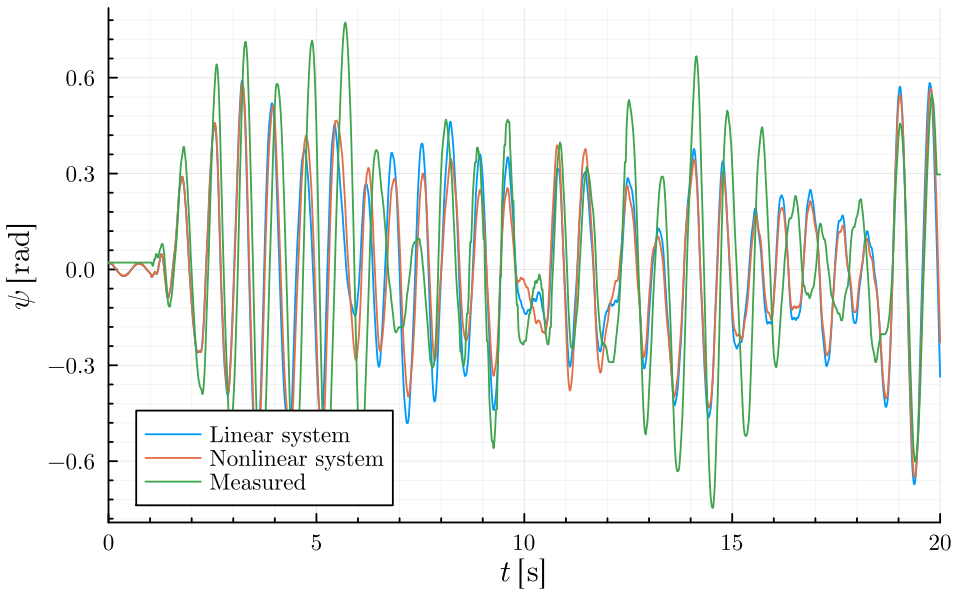


Figure 4.5: Comparison of the deflection angle of the ball for nonlinear and linear systems with the measured data from the first experiment.

Chapter 5

Controller design

This chapter will discuss and demonstrate the design of controllers using the ControlSystems.jl package. While ControlSystems.jl is not the sole package dedicated to control design, there is another package called RobustAndOptimalControl.jl [1] that specializes in designing advanced controllers.

The Figure 5.1 illustrates a general scheme of a controller and a system in a negative feedback loop with input and output disturbances. Since the reference is set to zero and the input disturbance is not going to affect the system, the general scheme can be reduced as it is depicted in the Figure 5.2.

The output disturbance d_o is going to act as an impulse change of the deflection angle of the ball inside the bigger hoop. When designing a controller, a few conditions need to be met. Mainly that the bandwidth in Hz of the negative feedback loop should be at least 10-30 times bigger than the sampling frequency of the camera (50 Hz).

The ControlSystems.jl package offers plenty of methods for designing a controller. The most useful ones are listed below.

- *laglink*(a, M) – creates a phase lagging transfer function in the form of $C(s) = \frac{s+a}{s+a/M}$

[1] Official documentation `https://juliacontrol.github.io/RobustAndOptimalControl.jl/dev/`

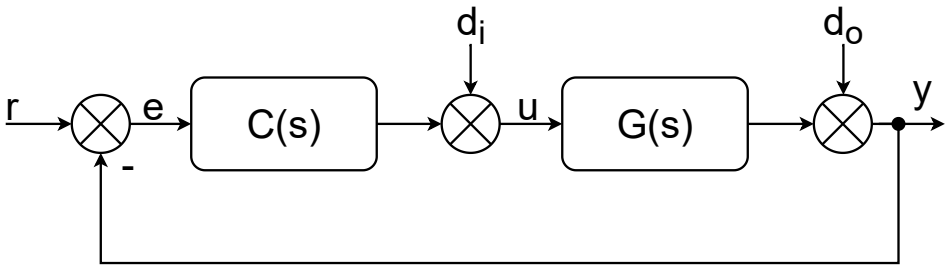


Figure 5.1: General negative feedback loop.

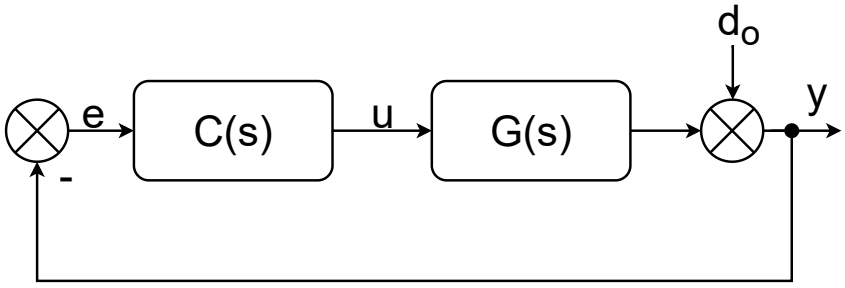


Figure 5.2: Reduced negative feedback loop.

- *leadlink*(b, N, K) – creates a phase leading transfer function in the form of $C(s) = KN\frac{s+b}{s+bN}$
- *loopshapingPI*(G, ω, ϕ_l, r_l) – creates a PI controller based on the loop shaping design method
- *loopshapingPID*(G, ω, M_t, ϕ_t) – creates a PID controller based on the loop shaping design method
- *lqr*(G, Q, R) – creates an optimal gain matrix $\boldsymbol{K}$ for a state feedback, that minimizes the cost function $\int_0^\infty (\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{u}^T\boldsymbol{R}\boldsymbol{u})\mathrm{d}t$
- *pid*(k_p, T_i, T_d) – creates a PID controller in the form of $C(s) = k_p\,(1 + 1/(T_i s) + T_d s)$
- *place*(A, B, p) – creates a gain matrix $\boldsymbol{K}$ such that $\boldsymbol{A} - \boldsymbol{B}\boldsymbol{K}$ has an eigenvalues defined by the vector $\boldsymbol{p}$
- *placePI*(G, ω_0, ζ) – creates a PI controller such that the negative feedback loop has the characteristic polynomial in the form of $s^2 + 2\omega_0\zeta s + \omega_0^2$

5.1 Design via loop shaping method

The presented design method is going to be via loop shaping using the *loopshapingPID* method since frequency domain design is in this case far easier than for example computing a desired location of the poles of the

negative feedback loop.

The designing process was iterative, meaning that the parameters ω, M_t and ϕ_t were changed based on the time and frequency response of the negative feedback loop. To be more precise, the time response is going to be an impulse response regarding the Figure 5.2.

The *loopshapingPID* returns not only a PID controller but also a PID controller with a filtered derivative to prevent an improper transfer function for converting it into a state space representation.

After a few iterations of the controller design, the resulting PID controller with a filtered derivative is in the form of the following transfer function:

```
omega = 10
Mt = 1.25
phi_t = 50
C,kp,ki,kd,_,Cf_pid = loopshapingPID(lsys,omega,Mt,phi_t)
```

```
TransferFunction{Continuous, ControlSystemsBase.SisoRational{Float64}}
0.0026901298082487772s^2 + 0.010720638072209578s + 0.16425557639727875
-----------------------------------------------------------------------
           0.0010236067274141814s^2 + 0.03906614150888865s

Continuous-time transfer function model
```

The filtered derivative part of the controller was slightly modified for better filtering of the higher frequencies.

5.2 Design of the lead compensator

In this section, the lead compensator will be designed. This type of compensator was chosen because it will add up a phase lead to increase the phase margin.

For this, the *leadlinkat* function is going to be used since it is more user-friendly for lead compensator design than *leadlink* function.

```
C_lead = leadlinkat(80, 5)
```

```
TransferFunction{Continuous, ControlSystemsBase.SisoRational{Float64}}
0.02795084971874737s + 1.0
--------------------------
0.005590169943749474s + 1.0

Continuous-time transfer function model
```

5.3 Analysis and simulation of a negative feedback loop

Before an analysis of the negative feedback loop, it is required to determine the transfer function of the negative feedback loop because functions mentioned in the Section 4.2 cannot accept an ODESystem type as an argument unlike the function *simulate_feedback_loop* in the BallAndHoop.jl package.

From the Figure 5.2 can be seen that the transfer function from the input d_o to the output y can be written as follows:

$$G_{\text{cl}}(s) = \frac{1}{1 + C(s)G(s)} \tag{5.1}$$

In the frequency domain, the control engineer is often interested not only in the frequency response of a certain transfer function but in the 'main four' transfer functions, namely sensitivity and complementary sensitivity transfer function, the transfer function of the measurement noise to control signal and the transfer function of the load disturbance to the measurement signal. These can be plotted using the *gangoffourplot* function.

```
setPlotScale("log10")
gangoffourplot(lsys,Cf_pid;label="",plot_settings...)
```

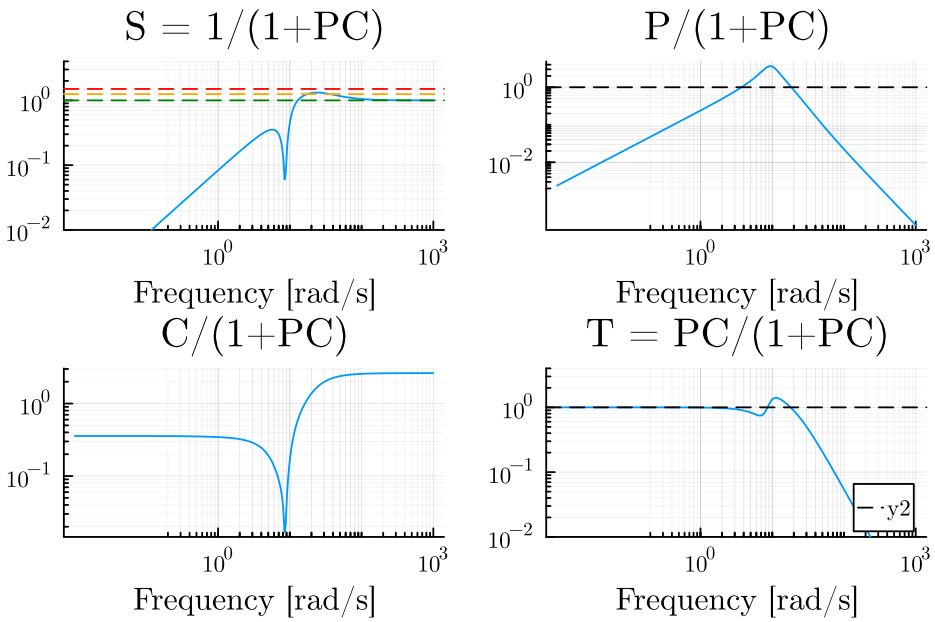


Figure 5.3: Main four transfer functions of the negative feedback loop for the PID controller.

```
gangoffourplot(lsys,C_lead;label="",plot_settings...)
```

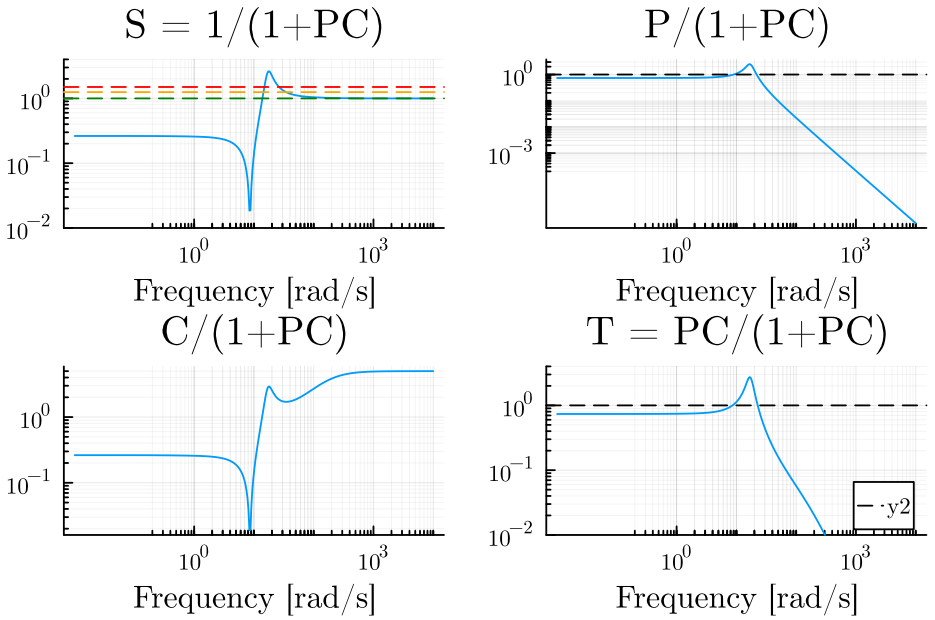


Figure 5.4: Main four transfer functions of the negative feedback loop for the lead compensator.

As mentioned above, for the time domain simulation, the function *simu-*

late_feedback_loop is going to be utilized. The arguments to this function are pretty much the same as the arguments for the *simulate_system* function except that *simulate_feedback_loop* requires also a controller converted to the ODESystem type.

Firstly, the output disturbance function is going to be defined. That function will act as an impulse of a certain amplitude which is going to displace the deflection angle of the ball inside the bigger hoop.

```
# Amplitude of the impulse in radians
amp = 0.4

# Output disturbance function definition
d_o = (t) -> amp*(t<=0.02)
```

The output disturbance function only checks if the current time of the simulation is less than 0.02 s (this is the sampling period of the camera) and multiplies that value by the amplitude. The comparison, when the time of the simulation is at zero, is not appropriate because the solver will most likely skip the exact timestamp. Therefore the sampling period of the camera has been taken into the output disturbance function.

```
@named PIDcontroller = ODESystem(Cf_pid)
@named leadcontroller = ODESystem(C_lead)
res_n_pid = simulate_feedback_loop(PIDcontroller,
                nonlinear_sys, d_o, [], (0,1.5), Rosenbrock23())
res_l_pid = simulate_feedback_loop(PIDcontroller,
             linear_sys, d_o, [], (0,1.5), Rosenbrock23())
res_n_lead = simulate_feedback_loop(leadcontroller,
                nonlinear_sys, d_o, [], (0,1.5), Rosenbrock23())
res_l_lead = simulate_feedback_loop(leadcontroller,
             linear_sys, d_o, [], (0,1.5), Rosenbrock23())
plot(res_n_pid, vars=[-PIDcontroller.input.u],
     yguide=L"\psi \ [\mathrm{rad}]",
     label="Nonlinear system with PID"; plot_settings...)
plot!(res_l_pid, vars=[-PIDcontroller.input.u],
     yguide=L"\psi \ [\mathrm{rad}]",xguide=L"t \ [\mathrm{s}]",
     label="Linear system with PID";plot_settings...)
plot!(res_n_lead, vars=[-leadcontroller.input.u],
     yguide=L"\psi \ [\mathrm{rad}]",xguide=L"t \ [\mathrm{s}]",
     label="Nonlinear system with lead";plot_settings...)
plot!(res_l_lead, vars=[-leadcontroller.input.u],
     yguide=L"\psi \ [\mathrm{rad}]",xguide=L"t \ [\mathrm{s}]",
     label="Linear system with lead";plot_settings...)
```

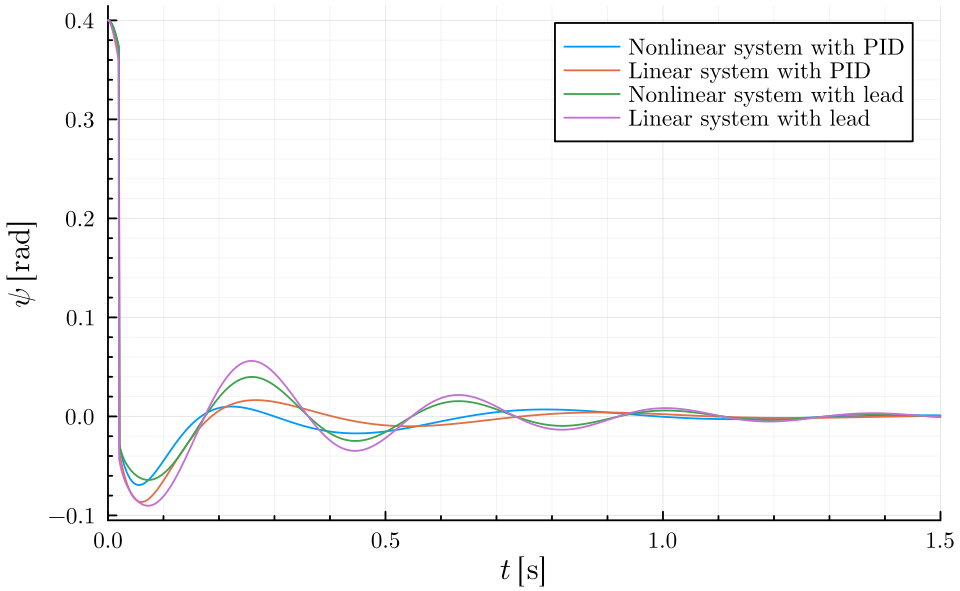


Figure 5.5: Deflection angle of the ball in the negative feedback loop.

The reason why the key argument *idxs* is not used here is that the output y is not a part of the solution of the ODEs. But since $e = -y$, the output can be accessed by passing a key argument *vars* and specifying which variable is going to be plotted. In this case the variable of the controller *-controller.input.u* is passed into the plotting function.

Chapter 6

Rapid prototyping of the control algorithms on the laboratory model

In this chapter, the rapid prototyping in Julia will be demonstrated as well as demonstrating multiprocessing computations in Julia. It needs to be mentioned that even though Julia is just-in-time-compiled, there is a possibility to compile code written in Julia into a binary file or convert a controller designed in the ControlSystems.jl package to a C code for a microcontroller. These possibilities are in the development and testing stage and are not yet fully reliable.

6.1 Structure of the control algorithm

The main goal was to get up and running multiple Julia processes that will communicate with each other. The idea behind this was that the main process will spawn three other processes, the first one will be for the communication with the BLDC motor driver over the UART, the second one will act as a controller and the third one will be for the ball detection.

For this task the package Distributed.jl needs to be loaded. This package is a part of the standard library in Julia and provides a special type called RemoteChannel which is a media for communicating between individual processes. In this case, the third process will send a detected position of the ball to the second process, the second process will compute a desired torque

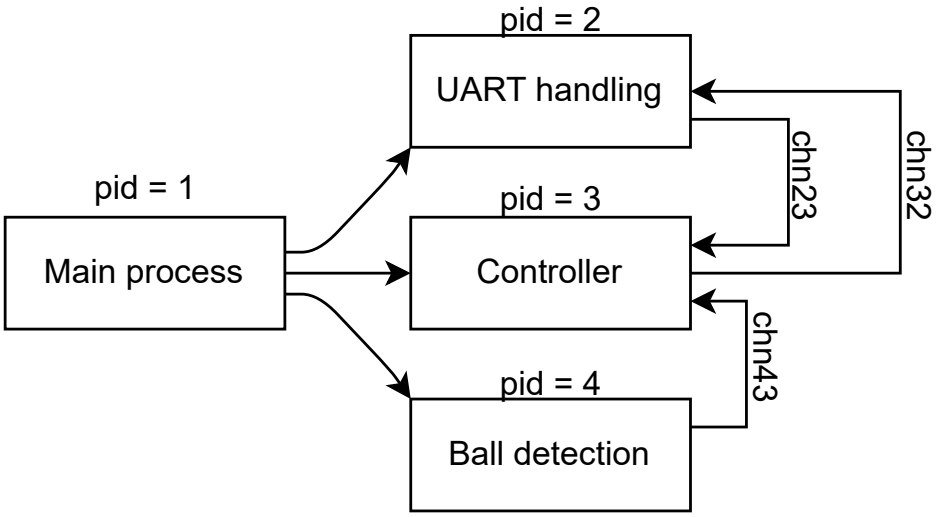


Figure 6.1: Multiprocessing structure of the control algorithm.

and send it to the first process and the first process will respond to the second one with the information about motor's position, speed and current.

6.2 Ball detection

The ball detection was already implemented in Python using OpenCV, Numpy and PiCamera modules. This could be possibly done in Julia with the package Images.jl but there would be a problem in obtaining a frame from the camera without the PiCamera Python module. Therefore the ball detection was not implemented in Julia and loaded and used using the PyCall.jl package that allows running any Python code in Julia.

The process of detection is rather straightforward. The user will choose the color of the ball in the config.json_sample file (only red, green and blue colors are allowed). Then the image is captured by the camera and converted to grayscale by choosing an appropriate channel (R, G, B) and clipping the values to a range from 0 to 255, then the additional mask is applied and finally, thresholding is applied after all of this.

Obtaining an image processed like described above allows to compute the center of the ball using image moments. The deflection angle can be computed from the position by the following formula:

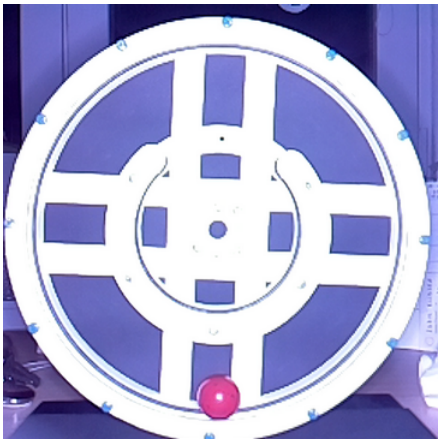


Figure 6.2: Image from the PiCamera.

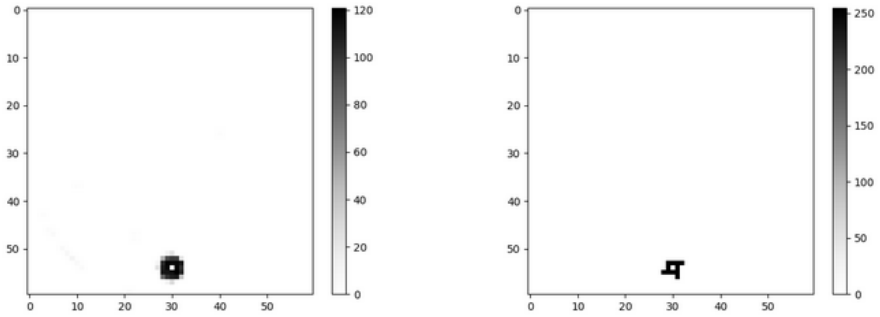


Figure 6.3: Result of the detection of the ball.

$$\psi = \arctan\left(c_x - x, y - c_y\right), \tag{6.1}$$

where c_x and c_y are the coordinates of the center of the picture.

6.3 Motor driver

The Raspberry Pi communicates over the UART with the BLDC motor driver which is implemented on the STM32 chip as well as a BLDC motor controller. The transmitted and received frames have a specific structure. The transmitted frame consists of 8 bytes and the received frame consists of 20 bytes.

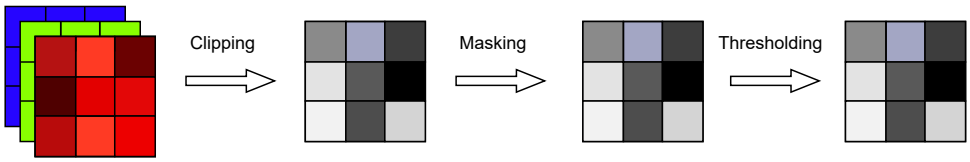


Figure 6.4: Process of the ball detection.

Transmitted frame structure:

- 1st byte = Sync sequence 0xFF
- 2nd byte = Length of the data part + 4 (in bytes)
- 3rd byte = 0x21
- 4th - 7th byte = Torque in Nm (float format)
- 8th byte = End sequence 0x00

Received frame structure:

- 1st byte = Sync sequence 0xFF
- 2nd byte = Length of the first data part + 4 (in bytes)
- 3rd byte = 0x70
- 4th - 7th byte = Angular velocity of the motor (float format)
- 8th - 11th byte = Position of the motor (float format)
- 12th byte = End sequence 0x00
- 13th byte = Sync sequence 0xFF
- 14th byte = Length of the second data part + 4 (in bytes)
- 15th byte = 0x71
- 16th - 19th byte = Current of the motor (float format)
- 20th byte = End sequence 0x00

As a part of the BallAndHoop.jl package, the UART wrapper is implemented using the LibSerialPort.jl[1] package. The implementation can be found at gitlab[2].

[1] Official documentation `https://juliaio.github.io/LibSerialPort.jl/stable/`

[2] Source code for the BallAndHoop.jl package `https://gitlab.fel.cvut.cz/aa4cc/j4c/ball-and-hoop`

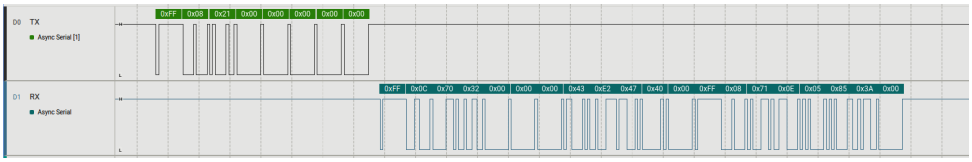


Figure 6.5: Communication protocol of the motor driver

6.4 Implementation of the controller

The last part discussed in this chapter is the implementation of the designed controller. Since the controller will run on the Raspberry Pi, the continuous-time controller needs to be converted to a discrete-time controller. This can be accomplished by using a *c2d* function in the ControlSystems.jl package.

```
# Discretizing the controller with the sampling period
# 0.02 s using the Tustin transformation
C_pid_discrete = c2d(Cf_pid, 0.02, :tustin)
```

```
TransferFunction{Discrete{Float64}, ControlSystemsBase.SisoRational{Float64}}
1.9895532268941687z^2 - 3.781042887492248z + 1.837946360416423
--------------------------------------------------------------
      1.0z^2 - 1.4475426500133597z + 0.4475426500133597

Sample Time: 0.02 (seconds)
Discrete-time transfer function model
```

```
C_lead_discrete = c2d(C_lead, 0.02, :tustin)
```

```
TransferFunction{Discrete{Float64}, ControlSystemsBase.SisoRational{Float64}}
2.4342806945451483z - 1.1514210418177229
----------------------------------------
       1.0z + 0.28285965272742575

Sample Time: 0.02 (seconds)
Discrete-time transfer function model
```

The resulting transfer functions can be converted into recurrent computational algorithms using the inverse Z transform. The formulas for the

computational algorithms after taking the inverse Z transform for the PID controller and lead compensator are

$$\begin{aligned} u[n] =& 1.9895532268941687e[n] - 3.781042887492248e[n-1] \\ &+ 1.837946360416423e[n-2] + 1.4475426500133597u[n-1] \\ &- 0.4475426500133597u[n-2], \end{aligned} \tag{6.2}$$

for the PID controller and

$$\begin{aligned} u[n] =& 2.4342806945451483e[n] - 1.1514210418177229e[n-1] \\ &- 0.28285965272742575u[n-1], \end{aligned} \tag{6.3}$$

for the lead compensator.

The recurrent formulas can be easily implemented in Julia as in other programming languages.

Chapter 7

Experimental results

During the testing of the designed controllers, a few limitations were reached. The first limitation was that running 4 processes on the Raspberry Pi with the Python script for ball detection was computationally intensive. The maximum number of processes that were able to run was three.

The distribution of the processes with the communication depicted in the Figure 6.1 was executed on a personal computer without a problem. The processes were able to send data according to the Figure 6.1. However, when the same code was executed on the Raspberry Pi, the transmission of the data was not executed. It was most likely due to a lack of computational power to execute all the processes.

This fact led to a change in the whole structure of the control algorithm to only one process which then executes the following procedure.

1. Detect the position of the ball from the camera.
2. Compute the difference between the reference and the measured position.
3. Based on that difference compute the appropriate torque acting on the system.
4. Send the computed torque via UART and read the response of the BLDC motor driver.
5. Sleep for 0.02 s.

After rewriting the control algorithm and running it again, the BLDC

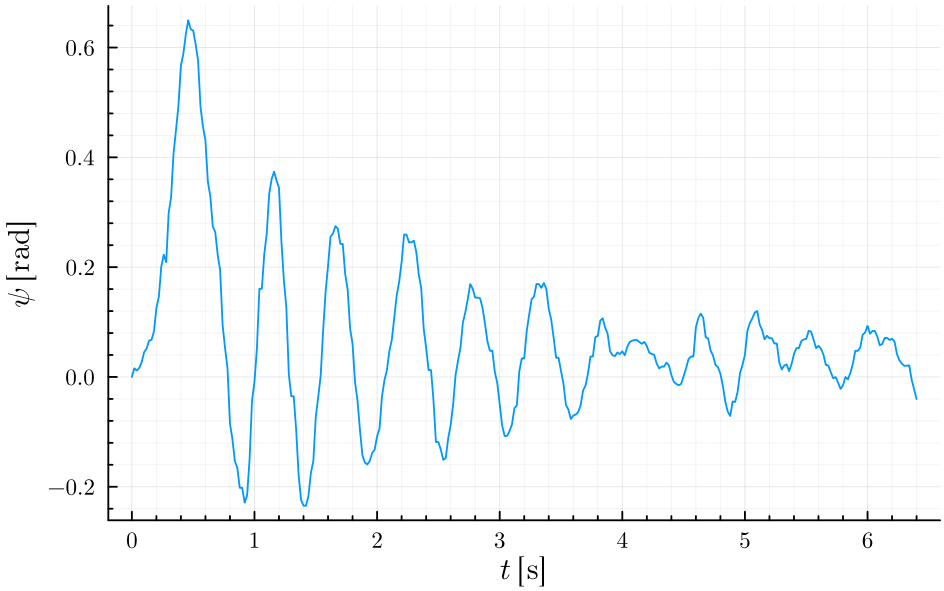


Figure 7.1: Measured deflection angle of the ball with the PID controller.

motor started to increase its speed up to a certain value. This behavior was not supposed to happen. But when the speed of the motor settles at some value, the deflection angle was approximately set to zero. This happened due to the fact, that the control algorithm does not regulate the speed as well as the deflection angle of the ball.

The solve this problem, full-state feedback needs to be designed. However, since not all the states are directly measurable, some kind of estimator needs to be designed as well. One possible solution is mentioned in the paper from Ing. Jiří Zemánek, PhD. and Ing. Martin Gurtner [5], simply using a Kalman filter to estimate the last state and then designing a full-state feedback should solve this problem.

The figures below show the measured deflection angle of the ball, the angular velocity of the BLDC motor and the torque acting on the system.

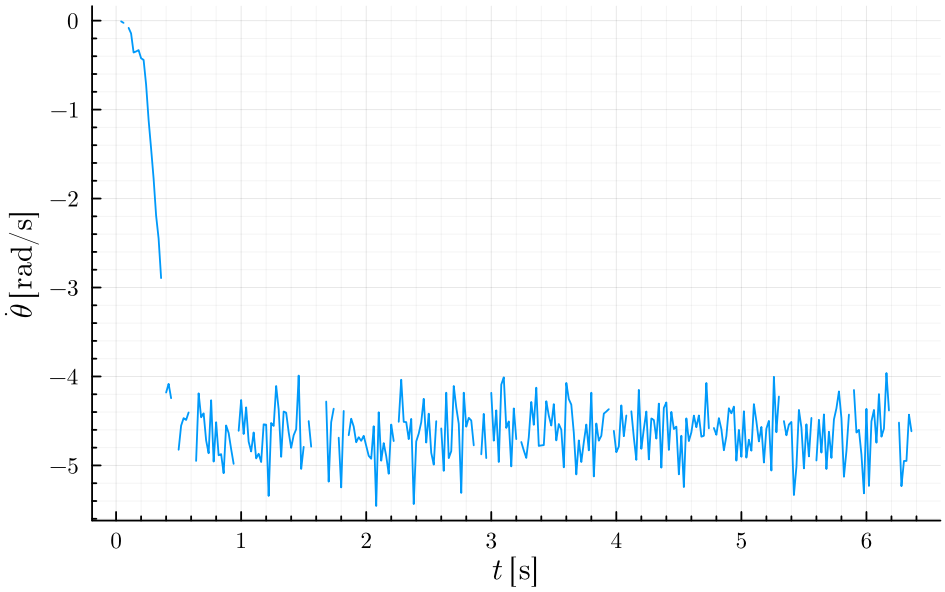


Figure 7.2: Measured angular velocity of the BLDC motor with the PID controller.

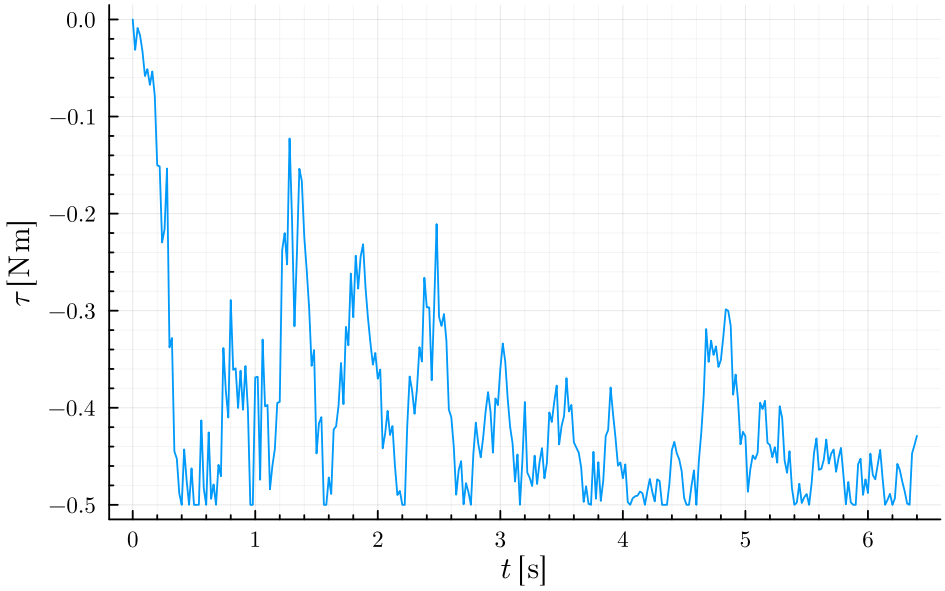


Figure 7.3: Measured torque with the PID controller.

Chapter 8

Conclusions and future work

This thesis fully demonstrates the process of developing control algorithms in Julia, starting from the modeling of the system, identification of the physical parameters of the model, followed by the linearization and controller design, and ending with the implementation of the control algorithm and demonstration of the achieved results.

The main objective has been achieved, and the package for interacting with the ball and hoop system that provides a simple interface for the end user for simulation of the system and also for the controller design was developed.

The realization of the control algorithm was successful, however, the results did not turn out how they were supposed to turn out. For more precise control, a more advanced controller needs to be developed, for example, using a Kalman filter with full-state feedback obtained by the LQR.

One of the future improvements is to include a hybrid description of the model in the BallAndHoop.jl package. During the work on this thesis, a pull request has been made for the ControlSystems.jl package and it was successfully merged into the main branch[1].

Julia on its own is feasible for controlling a real device, however, it depends on the complexity of the system and designed algorithm. In this case, the multiple-processes approach was not suitable at all, unlike the single-process

[1]The merged PR can be found at `https://github.com/JuliaControl/ControlSystems.jl/pull/804`

approach. During testing of the lead compensator, the BLDC motor started to transmit invalid frames and the motor stopped its rotation after a couple of seconds. Therefore the measured data from that experiment is not presented. The controller design in Julia on the other hand was a pleasant experience, however, sometimes the precompilation of the packages lasted way too long.

Appendix A

Bibliography

1. Fredrik Bagge Carlson, Mattias Fält, Albin Heimerson, and Olof Troeng. 2021. ControlSystems.jl: A control toolbox in julia. In *2021 60th IEEE conference on decision and control (CDC)*, 4847–4853. https://doi.org/10.1109/CDC45484.2021.9683403
2. Tom Breloff. 2023. *Plots.jl*. Zenodo. https://doi.org/10.5281/zenodo.7937756
3. Vaibhav Kumar Dixit and Christopher Rackauckas. 2023. *Optimization.jl: A unified optimization package*. Zenodo. https://doi.org/10.5281/zenodo.7738525
4. Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwozdz, Viral B Shah, Alan Edelman, and Christopher Rackauckas. 2021. High-performance symbolic-numerics via multiple dispatch. *arXiv preprint arXiv:2105.03949*.
5. Martin Gurtner and Jiří Zemánek. 2017. Ball in double hoop: Demonstration model for numerical optimal control. *50*, 2379–2384. https://doi.org/10.1016/j.ifacol.2017.08.429
6. Steven G. Johnson. 2007. *The NLopt nonlinear-optimization package*. Retrieved from http://github.com/stevengj/nlopt
7. Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. 2023. JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. *Mathematical Programming Computation*.
8. Yingbo Ma, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral Shah, and Chris Rackauckas. 2021. ModelingToolkit: A composable graph transformation system for equation-based modeling. Retrieved from https://arxiv.org/abs/2103.05244

9. Jiří Zemánek Martin Gurtner. 2017. Retrieved from https://github.com/aa4cc/flying-ball-in-hoop/blob/master/model/text/figs/sketchOfBallInAHoop3.pdf

10. Christopher Rackauckas and Qing Nie. 2017. DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software* 5, 1.