Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

AT+
RPTU

Construction and implementation of various control designs for inverted pendulum

RCS Project Lab

Yagut Badalova

Fransiska

Angel Iram Ochoa Diaz

Peter Tibenský

2023

Contents

List of Figures

Code listing

Introduction

In the field of control systems, the inverted pendulum stands as a classic and challenging problem, serving as a benchmark for testing the capabilities of control algorithms [1]. This project delves into the domain of inverted pendulum dynamics, focusing on the design, simulation, and implementation of control algorithms to stabilize and control the system dynamics.

The inverted pendulum, with its inherent instability, demands precise control strategies to maintain equilibrium. Our objective is to investigate and implement four known plus one new control methodologies that not only stabilize the pendulum in its upright position but also exhibit robust performance in the face of disturbances and uncertainties.

The report encompasses the theoretical foundations of control systems for inverted pendulum. Subsequently, it delves into the specific challenges posed by inverted pendulum dynamics and the various control strategies that have been proposed in the literature such as Jezierski, Mozaryn and Suski [2], book "The Inverted Pendulum in Control Theory and Robotics" [3] or the book "Advanced Control of Wheeled Inverted Pendulum Systems" [4].

(a) Rotational inverted pendulum from the company quanser [5].

(b) Linear inverted pendulum from the company quanser [6].

Figure 1: Configurations of inverted pendulum.

Within the domain of inverted pendulum systems, there are two main types of

configurations. It is rotational Fig. 1.a and linear configuration Fig. 1.b. In the case of a rotational inverted pendulum, the focus lies on managing the rotational motion of an object around a fixed pivot point. A classic example is a rigid rod attached to a pivot, and other fixed rod attached to the end of it, where the objective is to stabilize the system in an inverted position despite its inherent instability.

On the other hand, a linear inverted pendulum involves translational or linear motion along a vertical axis. This configuration is often represented by a cart on a track, with a pendulum attached to the cart. The task is to control the linear motion of the cart to maintain equilibrium with the pendulum inverted. The setup we are using for simulation and real world implementation is the linear model of inverted pendulum.

Our focus extends beyond theoretical considerations, as the project involves practical implementation on a physical inverted pendulum setup which is located at the chair of Automation and Control of RPTU. This particular setup can be seen on the picture 2. The hardware experimentation provides valuable insights into the real-world applicability of the developed algorithms, offering a bridge between theory and application.

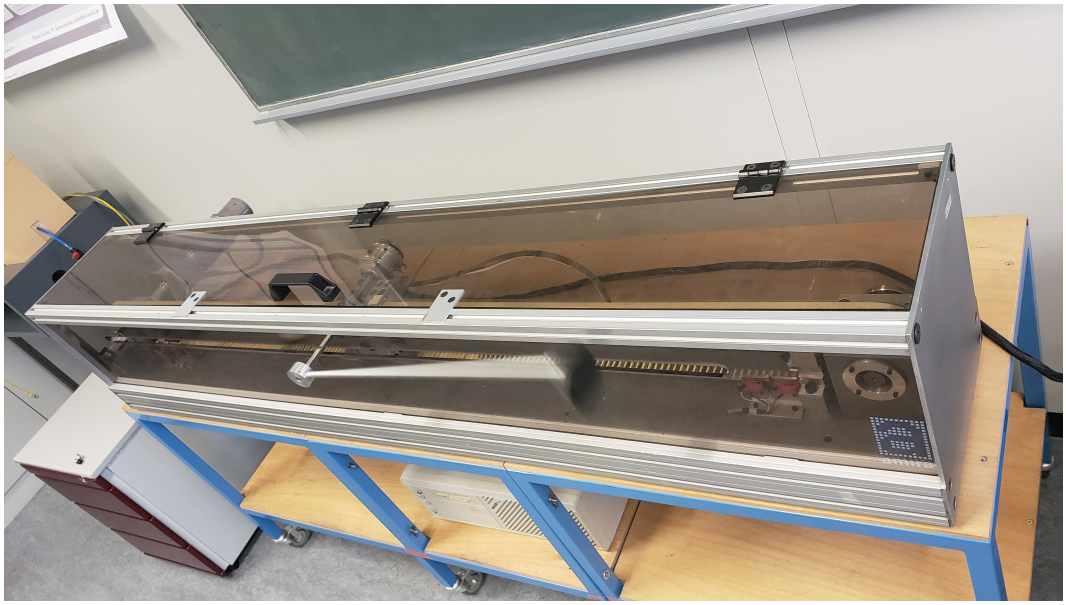

Figure 2: Inverted pendulum setup on which implementation of designed controllers was done.

This project aims to contribute to the field of control systems by presenting a comprehensive exploration of control algorithms for inverted pendulum systems. Through rigorous analysis and experimentation, this report endeavors to showcase the effectiveness and adaptability of the implemented control algorithms, opening avenues for further advancements in the control of dynamic systems.

1 Hardware

1.1 Inverted pendulum

1.2 Controller

2 Software

3 Controllers

3.1 Linear quadratic regulator (LQR)

3.1.1 Theory

The Linear Quadratic Regulator (LQR) has been presented by Rudolf E. Kalmanin in 1960 [7]. This optimal control algorithm is used for stabilizing linear dynamic systems by determining a control input that minimizes a quadratic cost function. LQR is an unified systematic control method for multiple-input multiple-output (MIMO) systems [7]. It is a very popular control algorithm because of its inherent robustness, where the gain and phase margin are guaranteed [8].

Central to the LQR framework is the concept of linear system dynamics. LQR is tailored for linear time-invariant systems, typically characterized by matrices A, B, C, and D, encapsulating the system's behavior [9].

At its core, LQR revolves around the minimization of a quadratic cost function over a specified time horizon. LQR reduces the amount of labor that needs to be put into the design of the controller, although the formulation of the cost function plays a crucial role in the controller performance.

The solution to the LQR problem involves online and offline calculations that can be separated to three distinct parts [7]:

1. **Solving the Riccati differential equation** [1]
2. **Computation of the feedback matrix K*t**
3. **Evaluation of the feedback control law**

Linear quadratic control problem can be formulated as Eq. 3.1

$$\min_{x(t),u(t).t_e} J(x(t),u(t),t_e) with J = \frac{1}{2}x(t_e)^T S x(t_e) + \frac{1}{2}\int_0^{t_e} x(t)^T Q(t) x(t) + u(t)^T R(t) u(t) dt \tag{3.1}$$

where $x(t_e)$ is the end state vector, $Q(t)$ is the state weighting matrix, $R(t)$ is the input weighting matrix and S is the end weighting matrix. Weighting matrices Q, R and S are design parameters and with the help of them, we can change the behavior of the controller. The optimal feedback control law is give by Eq. 3.2 [7]

[1] Riccati differential equation is a type of first-order ordinary differential equation that has a quadratic term in one of its variables

$$u^*(t) = K^*(t)x(t) \tag{3.2}$$

where K^* is the feedback matrix. given by Eq. 3.3 [7]

$$K^*(t) = R(t)^{-1}B(t)^T P(t) \tag{3.3}$$

3.1.2 Simulation

In the context of simulation using MATLAB for control system design, the provided code, obtained from the lab coordinator, incorporates the design of a Linear Quadratic Regulator (LQR) controller used for the control of nonlinear inverted pendulum. The primary objective is to evaluate the functionality of the code under different conditions. To assess the system's robustness and performance, disturbances have been introduced. Additionally, the LQR controller's parameters have been fine-tuned for optimal performance.

The simulation relies on MATLAB's dlqr command Eq. 3.4, where "dlqr" stands for "Linear-quadratic (LQ) state-feedback regulator" specifically designed for discrete-time state-space systems. This MATLAB command is instrumental in computing the state-feedback gain matrix for an LQR controller, considering both state and input weighting matrices. The resulting gain matrix is then utilized in the feedback control law to regulate the system and optimize its performance.

$$[K, P] = dlqr(A, B, Q, R) \tag{3.4}$$

Where K is the feedback matrix and P infinite horizon solution of the associated discrete-time Riccati equation, where P is in the form of Eq. 3.5

$$A^T SA - S - (A^T SB + N)(B^T SB + R)^{-1}(B^T SA + N^T) + Q = 0 \tag{3.5}$$

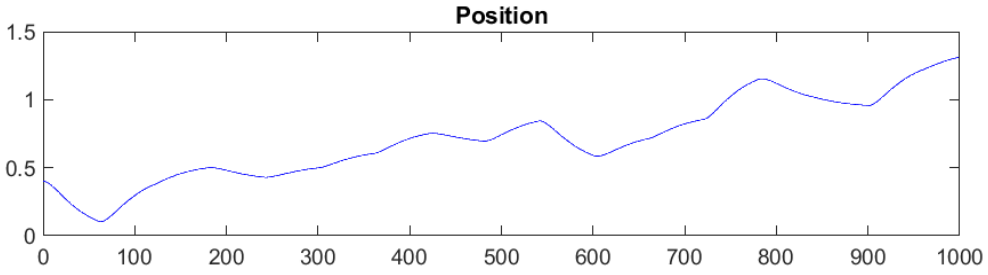

Figure 3.1: Position of the pendulum (LQR simulation).

Simulation outputs

Following the MATLAB simulation, the obtained results reveal distinctive patterns Fig. 3.1, 3.2, 3.3. Notably, each spike observed across all plots corresponds to the introduced disturbance. These disturbances were deliberately added to assess the robustness of the controller under varying conditions.

Figure 3.1 shows the changing position of the pendulum carriage. Figure 3.2 shows the angle of the pendulum in radians and Fig. 3.3 shows the force i.e. torque applied, to move the pendulum. The full code can be found in appendix on the page v.

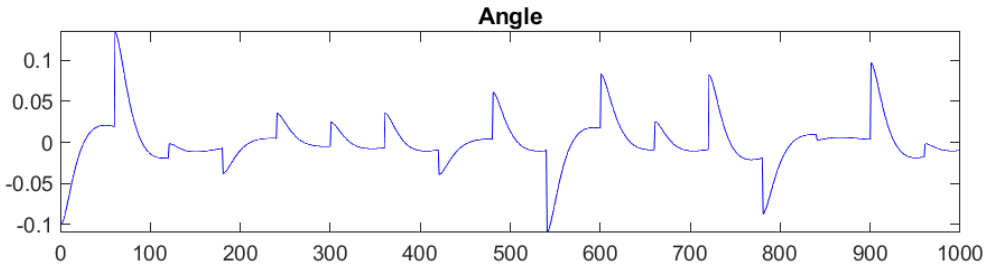

Figure 3.2: Angle of the pendulum (LQR simulation).

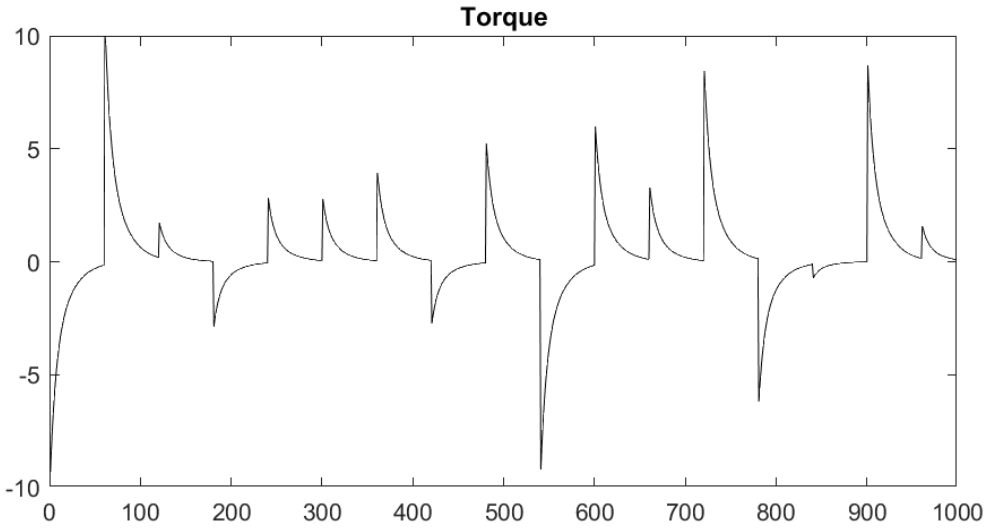

Figure 3.3: Torque of the pendulum (LQR simulation).

Examining the plots, it becomes apparent that the angle of the pendulum Fig. 3.2 achieves stabilization in the unstable upright position within a maximum of 20 samples, depending on the level of disturbance introduced. However, it's important to note that the position of the pendulum Fig. 3.1 does not attain stability; instead, it undergoes movement away from its initial 0 position (when trying to control the pendulum angle).

3.1.3 Implementation

3.2 Model predictive control (MPC)

3.2.1 Theory

Model Predictive Control (MPC) represents an optimal control methodology where the computed control actions are designed to minimize a cost function for a constrained dynamical system over a finite, receding horizon. Its primary advantage over LQR control lies in its superior performance when the process encounters limitations [10]. However, it is acknowledged that MPC require a steeper learning curve and a more intricate implementation process. Often compared to playing chess, MPC depends on a deep understanding of the plant model and subsequent predictions of its behavior, recalculating the best possible output at each step [10][11].

In the MPC framework, the controller, at each time step, receives or estimates the current state of the plant. Based on this information, it computes a sequence of control actions that minimizes the cost over the specified horizon (horizon can be sometimes infinite) [12][10]. This involves solving a constrained optimization problem, heavily relying on an internal plant model and dependent on the current system state. The controller then applies the first computed control action to the plant, disregarding the subsequent ones [12]. This iterative process repeats in each subsequent time step.

In practical applications, despite the finite horizon, MPC inherits several beneficial characteristics from traditional optimal control methodologies. It naturally accommodates multi-input multi-output (MIMO) plants, time delays, and possesses inherent robustness against model errors [11]. Furthermore, nominal stability can be assured by incorporating specific constraints. Overall, while MPC demands a more intricate knowledge of the system and involves a complex implementation, it gives big advantages in handling constraints and offering superior performance [10].

In a mathematical way we can express MPC problem as: finding the best control sequence over a future horizon of N steps Eq. 3.6

$$\min_{u_o,\ldots,u_{N-1}} \sum_{k=0}^{N-1} \|y_k - r(t)\|_2^2 + \rho \|u_k - u_r(t)\|_2^2 \tag{3.6}$$

$$\begin{aligned} s.t. x_{k+1} &= f(x_k, u_k) \\ y_k &= g(x_k) \end{aligned} \tag{3.7}$$

$$\begin{aligned} u_{min} \leq u_k \leq u_{max} \\ y_{min} \leq y_k \leq y_{max} \end{aligned} \tag{3.8}$$

$$x_o = x(t) \tag{3.9}$$

Where Eq. 3.7 stands as the prediction model, Eq. 3.8 as constraints and Eq. 3.9 as state feedback [11].

3.2.2 Simulation

HOW WAS THE SIMULATION DONE

Following the MATLAB simulation, the obtained results reveal distinctive patterns Fig. 3.4, 3.5, 3.6. Notably, each spike observed across all plots corresponds to the introduced disturbance. These disturbances were deliberately added to assess the robustness of the controller under varying conditions.

Figure 3.4 shows the changing position of the pendulum carriage. Figure 3.5 shows the angle of the pendulum in radians and Fig. 3.3 shows the force i.e. torque applied, to move the pendulum. The full code can be found in appendix on the page x.

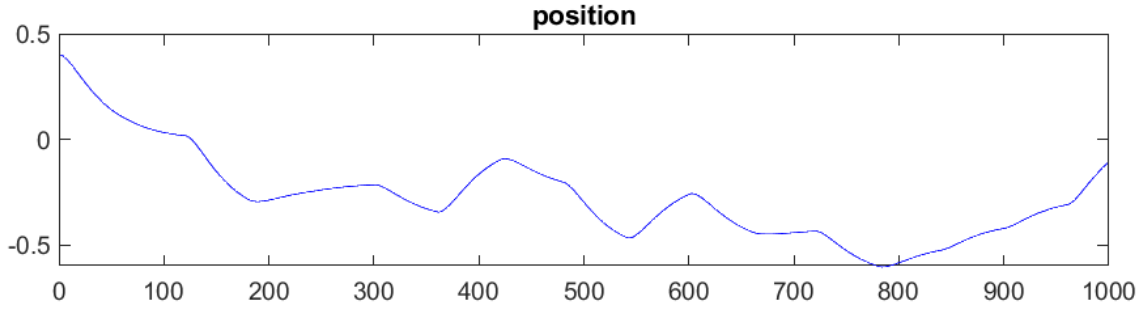

Figure 3.4: Position of the pendulum (MPC simulation).

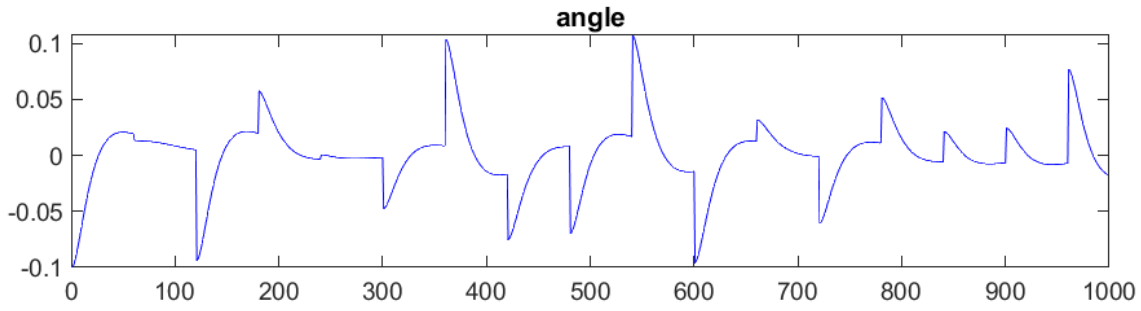

Figure 3.5: Angle of the pendulum (MPC simulation).

Examining the plots, it becomes apparent that the angle of the pendulum Fig. 3.5 achieves stabilization in the unstable upright position within a maximum of 20 samples, depending on the level of disturbance introduced. However, it's important to note that the position of the pendulum Fig. 3.4 does not attain stability; instead, it undergoes movement away from its initial 0 position (when trying to control the pendulum angle).

3.2.3 Implementation

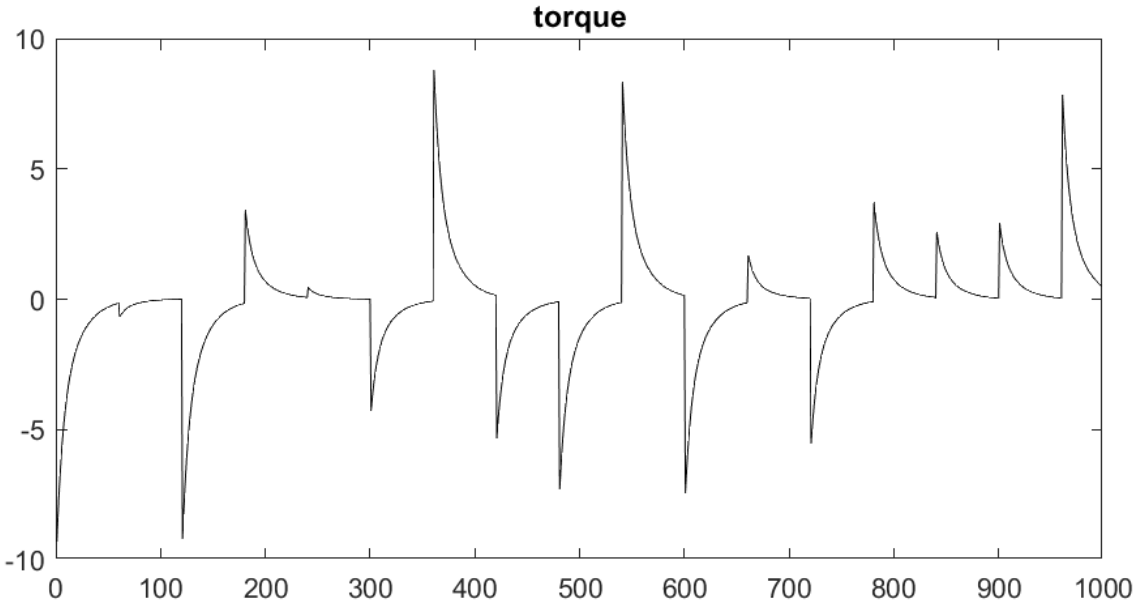

Figure 3.6: Torque of the pendulum (MPC simulation).

3.3 Sliding mode control (SMC)

3.3.1 Theory

Sliding mode control (SMC) is a nonlinear control methodology known for its impressive attributes such as precision, resilience, and straightforward tuning and application. It is applicable to both Single-Input Single-Output (SISO) and Multiple-Input Multiple-Output (MIMO) systems [13][14]. The SMC consists of a two-part controller design. In the initial phase, an N-dimensional sliding surface must be designed where, by definition, the error asymptotically approaches zero. The subsequent phase is concerned with the selection of a control law, that will make the system states be attracted to the sliding surface.

The state-feedback control law is not a continuous function of time, rather, it can transition from one continuous structure to another based on the current position in the state space. The movement of the system as it glides along the boundaries of the control structures is referred to as a sliding mode, and the geometric locus comprising these boundaries is called the sliding surface [15]. SMC exhibits robustness against model uncertainties and disturbances. However, a significant drawback is the occurrence of chattering effects around the surface, which arises as a result of the switching of the control law. This is a common phenomenon in sliding mode control and is highly undesirable in practical applications due to increased control activity and the potential excitation of unmodeled high-frequency dynamics. Nevertheless, these issues can be mitigated or even avoided through the application of appropriate strategies [13].

3.3.2 Simulation

The initial stage in simulating the sliding mode controller in MATLAB involved the creation of the sliding surface Eq. 3.10. Afterwards, the calculation of the output estimate τ Eq. 3.14 needs to be done.

$$s = (\frac{d}{dt} + e)\tilde{\alpha} + (\frac{d}{dt} + \beta)\tilde{\rho} = \dot{\tilde{\alpha}} + e\tilde{\alpha} + \dot{\tilde{\rho}} + \beta\tilde{\rho} = -\dot{\alpha} - e\alpha - \dot{\rho} - \beta\rho = 0 \quad (3.10)$$

$$\dot{s} = -\dot{\omega} - e\omega - \dot{v} - \beta v = 0 \quad (3.11)$$

$$\dot{s} = -\varphi - \alpha + \tau + (- e)\omega + v - \tau - \beta v \quad (3.12)$$

$$\dot{s} = \quad (3.13)$$

$$\hat{\tau} = \frac{j_a(m_\omega + m_e)}{m_e l_{sp} - j_a}\Big(+ \ - (\ - e) - (\ - \beta)v\Big) \quad (3.14)$$

$$\hat{\tau} =$$

$$\hat{\tau} = 12,335239v + 68,877358\alpha - 3,492138(0,125 - e)\omega - 3,492138(1,7568 - \beta)v$$

where q_2 stands for the pendulum angle, $\dot{q_2}$ is the angular velocity of the pendulum, q_1 is position of the base of the pendulum and $\dot{q_1}$ is the linear velocity of the base of the pendulum.

Following the MATLAB simulation, the obtained results reveal distinctive patterns Fig. 3.7, 3.8, 3.9. Notably, each spike observed across all plots corresponds to the introduced disturbance. These disturbances were deliberately added to assess the robustness of the controller under varying conditions.

Figure 3.7 shows the changing position of the pendulum carriage. Figure 3.8 shows the angle of the pendulum in radians and Fig. 3.9 shows the force i.e. torque applied, to move the pendulum. The full code can be found in appendix on the page xvi.

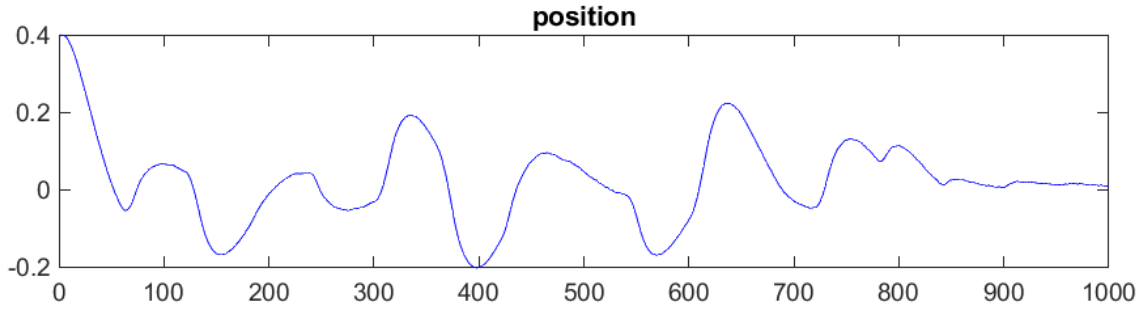

Figure 3.7: Position of the pendulum (SMC simulation).

Examining the plots, it becomes apparent that the angle of the pendulum Fig. 3.8 achieves stabilization in the unstable upright position within a maximum of 40 samples, depending on the level of disturbance introduced. The position of the pendulum Fig. 3.7 attain stability within a maximum of 40 samples. We can clearly see the rapid switching of the control law on the Fig. 3.9.

3.3.3 Implementation

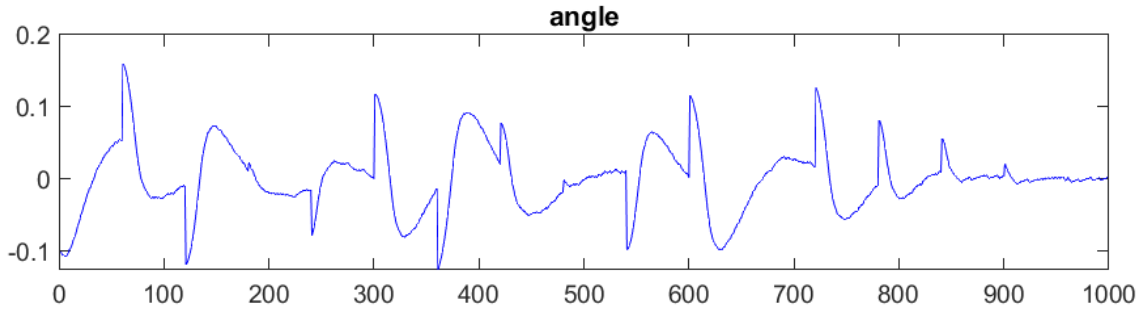

Figure 3.8: Angle of the pendulum (SMC simulation).

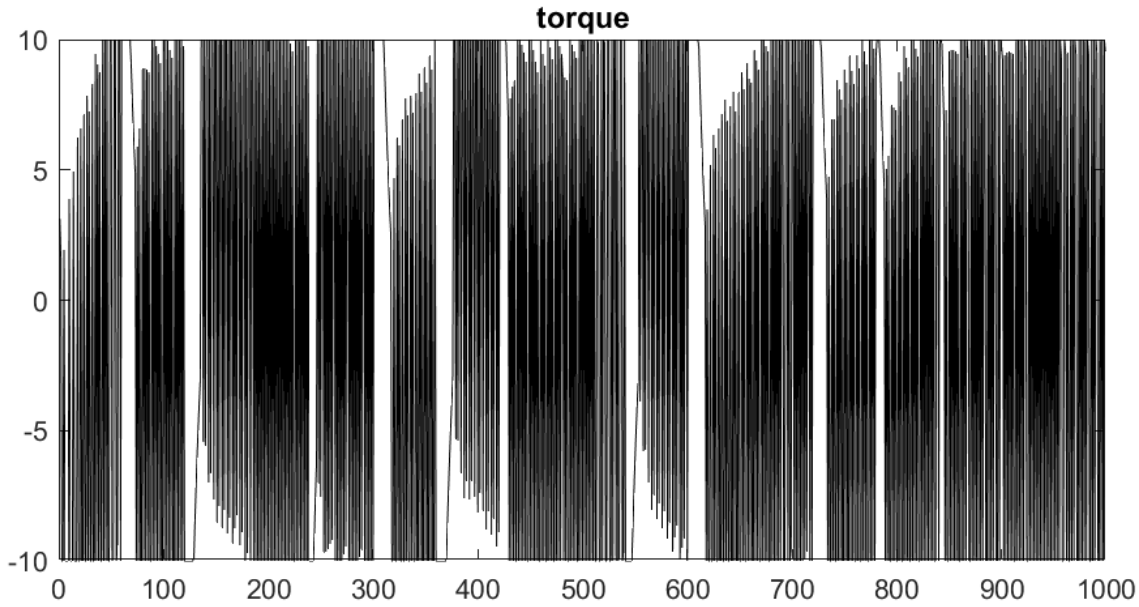

Figure 3.9: Torque of the pendulum (SMC simulation).

3.4 Fuzzy controller

3.4.1 Theory

3.4.2 Simulation

3.4.3 Implementation

3.5 Impedance control

3.5.1 Theory

3.5.2 Simulation

3.5.3 Implementation

Bibliography

[1] Mukhtar Fatihu Hamza, Hwa Jen Yap, Imtiaz Ahmed Choudhury, Abdulbasid Ismail Isa, Aminu Yahaya Zimit, and Tufan Kumbasar. Current development on using rotary inverted pendulum as a benchmark for testing linear and nonlinear control algorithms. *Mechanical Systems and Signal Processing*, 116:347–369, 2019.

[2] Andrzej Jezierski, Jakub Mozaryn, and Damian Suski. A comparison of lqr and mpc control algorithms of an inverted pendulum. In Wojciech Mitkowski, Janusz Kacprzyk, Krzysztof Oprzędkiewicz, and Paweł Skruch, editors, *Trends in Advanced Intelligent Control, Optimization and Automation*, pages 65–76, Cham, 2017. Springer International Publishing.

[3] Olfa Boubaker and Rafael Iriarte Vivar Balderrama. *The Inverted Pendulum in Control Theory and Robotics : From Theory to New Innovations*. 10 2017.

[4] Liping Fan Zhijun Li, Chenguang Yang. *Advanced Control of Wheeled Inverted Pendulum Systems*. 7 2012.

[5] qube servo 3, Dec 2023.

[6] linear flexible inverted pendulum, Dec 2023.

[7] Prof. Dr. Ing. Daniel Goerges. Optimal control 5. linear-quadratic optimal control. University Lecture, 2023.

[8] N. Lehtomaki, N. Sandell, and M. Athans. Robustness results in linear-quadratic gaussian based multivariable control designs. *IEEE Transactions on Automatic Control*, 26(1):75–93, 1981.

[9] In *Wikipedia*, Januar 2024.

[10] Martin Gulan Gergely Takacs. *Zaklady prediktivneho riadenia*. 2018.

[11] Alberto Bemporad. Model predictive control. University Lecture, 2023.

[12] What is model predictive control?

[13] Prof. Dr.-Ing. Steven Liu. Chapter 5. sliding mode control and integrator backstepping. University Lecture, 2023.

[14] Raymond A. Decarlo and Stanisław H. Żak. A quick introduction to sliding mode control and its applications 1. 2008.

[15] Sliding mode control. In *Wikipedia*, Januar 2024.

Source code of linear quadratic regulator (Simulation)

```
clear all;
clc
import casadi.*

%% Parameters and Initialization
Ts = 0.02; % Sampling Time
time = 20; % Total simulation time

% Preallocate arrays for system variables
q   = zeros(2, time/Ts); % Position and angle
qd  = zeros(2, time/Ts); % Linear and angular velocity
qdd = zeros(2, time/Ts); % Linear and angular acceleration
q(:, 1) = [0.4; -0.1]; % Initial values for position and angle
tau   = zeros(1, time/Ts); % Voltage
q_r(1) = 0;  % Reference value of position
q_r(2) = 0;  % Reference value of angle

qd_r = zeros(2, 1); % Reference value of linear and angular velocity

%% State-space
m_p     = 0.329; m_w     = 3.2; l_sp    = 0.44; f_w     = 6.2;
f_p     = 0.009; gra       = 9.81; j_a     = 0.072; Ts = 0.02;

% Continuous-time state-space matrices
A_c = [0    1                                    0                    0
0   -f_w/(m_w+m_p)                        0                    0
0   0                                    0                    1
0   (f_w*m_p*l_sp)/(j_a*(m_w+m_p)) (m_p*l_sp*gra)/j_a     -f_p/j_a];
B_c = [0   ;    1/(m_w+m_p) ;    0    ;    -m_p*l_sp/((m_w+m_p)*j_a)];
C_c = [1    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1];
D_c = [0;0;0;0];

% Convert to discrete-time
sys_cont = ss(A_c, B_c, C_c, D_c);
sys_d = c2d(sys_cont, Ts);

A = sys_d.A; B = sys_d.B; C = sys_d.C; D = sys_d.D;

%% Pendulum parameters
KF=2.6; M0=3.2; M1=0.329; M=M0+M1; ls=0.44; inert=0.072; N_val=0.1446;
N01_sq=0.23315; Fr=6.2; C=0.009; gra=9.81;

a32 = -N_val^2/N01_sq*gra ; a33 = -inert*Fr/N01_sq; a34 = N_val*C/N01_sq;
a35 = inert*N_val/N01_sq; a42 = M*N_val*gra/N01_sq; a43 = N_val*Fr/N01_sq;
    a44 = -M*C/N01_sq;
a45 = -N_val^2/N01_sq; b3=inert/N01_sq; b4=-N_val/N01_sq;
b3_hat = inert/N01_sq+0.1; b4_hat = -N_val/N01_sq+0.1;

%% Animation parameters
xmin = -1;
xmax = +1;

figure;
h     = [];
h(1) = subplot(4,2,1);
h(2) = subplot(4,2,3);
h(3) = subplot(4,2,5);
h(4) = subplot(4,2,7);
h(5) = subplot(2,2,2);
h(6) = subplot(2,2,4);

Disturbance=1;
```

```
65
66  %% Controller Parameters
67  Q = diag([1, 1, 1e7, 1e2]);      % State penalization
68  R = 1e7;                         % Input penalization
69  Np = 10;                         % Prediction horizon
70  [K, P] = dlqr(A, B, Q, R);
71
72  for k = 1:time/Ts-1
73
74  % Control Algorithm
75  tau(1,k) = -K*[q(1,k); qd(1,k);q(2,k); qd(2,k)]; % LQR
76
77  % Voltage Limitation
78  if abs(tau(:,k)) > 10
79  tau(:,k) = sign(tau(:,k))*10;
80  end
81
82  % Inverted Pendulum Math. Model
83  beta_x2 = (1 + N_val^2/N01_sq*(sin(q(2,k)))^2)^(-1);
84  qdd(:,k+1) = [beta_x2*(a32*sin(q(2,k))*cos(q(2,k))+a33*qd(1,k)+...
85  a34*cos(q(2,k))*(qd(2,k))+a35*sin(q(2,k))*qd(2,k)^2+b3*tau(:,k));
86  beta_x2*(a42*sin(q(2,k))+a43*cos(q(2,k))*qd(1,k)+...
87  a44*(qd(2,k))+a45*cos(q(2,k))*sin(q(2,k))*(qd(2,k))^2+b4*cos(q(2,k))*tau(:,
        k))];
88
89  qd(:,k+1) = qd(:,k) + qdd(:,k+1)*Ts;
90  q(:,k+1) = q(:,k) + qd(:,k+1)*Ts;
91  q(2,k+1) = mod(q(2,k+1)+pi,2*pi)-pi;
92
93  % Disturbance
94  if mod(k,60)==0 && Disturbance==1
95  x = rand();
96
97  if x > 0.5 && k ~= 200
98  q(2,k+1) = q(2,k+1) + rand() * 0.12;
99  else
100 q(2,k+1) = q(2,k+1) - rand() * 0.12;
101 end
102
103 if k == 200
104 q(2,k+1) = q(2,k+1) - 0.1;
105 end
106 end
107
108 % Plot Animation
109 plot(0, 'Parent', h(6));
110 hold on;
111 p1 = -q(1,k);
112 p2 = -q(1,k) + ls * exp(1i*(q(2,k)+pi/2));
113 line(real([p1,p2]), imag([p1,p2]));
114 plot(real(p2), imag(p2), '.', 'markersize', 40);
115 hold off;
116
117 % Center plot w.r.t. object
118 if q(1) > xmax
119 xmin = xmin + 0.1;
120 xmax = xmax + 0.1;
121 elseif q(1) < xmin
122 xmin = xmin - 0.1;
123 xmax = xmax - 0.1;
124 end
125
126 % Update animation
127 grid on;
128 axis([h(1)],[0 time/Ts-1 -1 1]); title('position')
129 axis([h(2)],[0 time/Ts-1 -0.5 0.5]);
130 axis([h(3)],[0 time/Ts-1 -1 1]);
131 axis([h(4)],[0 time/Ts-1 -5 5]);
132 axis([h(5)],[0 time/Ts-1 -10 10]);
133 axis([h(6)],[xmin-.2 xmax+.2 -.5 .5]);
```

```
134
135 % Update subplots
136 drawnow;
137
138 plot(q(1,1:k),'b','Parent',h(1)); %Position
139 plot(q(2,1:k),'b','Parent',h(2)); % Angle
140 plot(qd(1,1:k),'b','Parent',h(3)); % Velocity
141 plot(qdd(1,1:k),'b','Parent',h(4)); % Angular velocity
142 plot(tau(1:k),'k','Parent',h(5)); % Torque
143
144 title(h(1), 'Position');
145 title(h(2), 'Angle');
146 title(h(3), 'Velocity');
147 title(h(4), 'Angular Velocity');
148 title(h(5), 'Torque');
149 end
```

Code 3.1: Source code of linear quadratic regulator (Simulation).

Source code of linear quadratic regulator (Real implementation)

```
% Pause for 3 seconds before starting the code execution
pause(3);

% Clear workspace, close all figures, and clear command window
clear all;
close all;
clc;

% Add necessary paths for libraries
addpath(genpath('CLSS Praxis'), genpath('hudaqlib'))

% Hudaq device initialization
dev = HudaqDevice('MF634');

% Total experiment samples
s = 2000;
% Sampling period
Ts = 0.02;

% States during the experiment
x = zeros(s,4);
% Sample states
z2 = zeros(4,1);
% Initial values
x(1,:) = [(AIRead(dev,1)/0.15/100) 0.01*round(-13.1*AIRead(dev,3)) (-AIRead
    (dev,2)/0.96*pi/180) 0];

% Force array to store calculated force values
Force = zeros(s,1);

% Pendulum parameters
m_p     = 0.329; m_w     = 3.2; l_sp    = 0.44; f_w     = 6.2;
f_p     = 0.009; gra       = 9.81; j_a     = 0.072; Ts = 0.02;

% Continuous-time state-space matrices
A_c = [ 0   1                                    0                    0
0   -f_w/(m_w+m_p)                    0                    0
0   0                                0                    1
0   (f_w*m_p*l_sp)/(j_a*(m_w+m_p)) (m_p*l_sp*gra)/j_a     -f_p/j_a];
B_c = [0   ;   1/(m_w+m_p) ;   0   ;   -m_p*l_sp/((m_w+m_p)*j_a)];
C_c = [   1   0   0   0
0   1   0   0
0   0   1   0
0   0   0   1];
D_c = [0;0;0;0];

% Convert to discrete-time
sys_cont = ss(A_c,B_c,C_c,D_c);
sys_d = c2d(sys_cont,Ts);

A = sys_d.A; B = sys_d.B; C = sys_d.C; D = sys_d.D;

% LQR controller parameters
Q = diag([100, .1, 100, .1]);     % State penalization
R = 1e-2;                         % Input penalization
Np = 60;                          % Prediction horizon
[K, P] = dlqr(A, B, Q, R);

% Loop through each sample
for i = 1:s-1

% LQR CONTROL ALGORITHM
Force(i) = ctranspose(-K*x(i,:));

% Save the data
```

```
65 z2(1) = AIRead(dev,1)/0.15/100;      % Position of cart (meter)
66 z2(2) = 0.01*round(-13.1*AIRead(dev,3));     % Speed of cart (m/s)
67 z2(3) = -AIRead(dev,2)/0.96*pi/180;  % Angle of pendulum (radian)
68
69 % Voltage limitation
70 if abs(Force(i)) > 10
71 Force(i) = sign(Force(i))*10;
72 end
73
74 % Apply calculated voltage
75 tic
76 while toc < Ts
77 DOWriteBit(dev, 1, 2, 1);    % Activation pendulum
78 DOWriteBit(dev, 1, 2, 0);    % Channel 1 consists of DO0..DO7
79 DOWriteBit(dev, 1, 2, 1);    % DO2 Requires continuous pulse
80 AOWrite(dev, 2, Force(i));  % Apply calculated voltage
81 end
82
83 % Angular speed calculation (derivative)
84 z2_winkel = -AIRead(dev,2)/0.96*pi/180;
85 z2(4) = (z2_winkel - z2(3))/Ts;    % Angular speed of pendulum
86
87 % Update states
88 x(i+1,:) = ctranspose(z2);
89
90 % Check if the pendulum is out of range
91 if abs(z2(1)) > 0.3 || abs(z2(3)*180/pi) > 10
92 disp('Please bring me back !');
93 pause(3);    % Wait for 3 seconds
94 end
```

Code 3.2: Source code of linear quadratic regulator (Real implementation).

Source code of model predictive controller (Simulation)

```
% Clear workspace, command window, and import CasADi library
clear all;
clc
import casadi.*

%% Parameters and Initialization
Ts = 0.02; % Sampling Time
time = 20; % Total simulation time

% Preallocate arrays for system variables
q   = zeros(2, time/Ts); % Position and angle
qd  = zeros(2, time/Ts); % Linear and angular velocity
qdd = zeros(2, time/Ts); % Linear and angular acceleration
q(:, 1) = [0.4; -0.1]; % Initial values for position and angle
tau   = zeros(1, time/Ts); % Voltage

q_r(1) = 0;  % Reference value of position
q_r(2) = 0;  % Reference value of angle
Q = diag([1, 1, 1e7, 1e2]);      % State penalisation
R = 1e7;                         % Input penalisation
Np = 10;      % Prediction horizon

qd_r = zeros(2, 1); % Reference value of linear and angular velocity

%% State-space
m_p     = 0.329; m_w     = 3.2; l_sp    = 0.44; f_w     = 6.2;
f_p     = 0.009; gra       = 9.81; j_a     = 0.072; Ts = 0.02;

% Continuous-time state-space matrices
A_c = [0    1                                    0                    0
0    -f_w/(m_w+m_p)                    0                    0
0    0                                 0                    1
0    (f_w*m_p*l_sp)/(j_a*(m_w+m_p)) (m_p*l_sp*gra)/j_a     -f_p/j_a];
B_c = [0   ;    1/(m_w+m_p) ;    0    ;    -m_p*l_sp/((m_w+m_p)*j_a)];
C_c = [    1    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1];
D_c = [0;0;0;0];

% Convert to discrete-time
sys_cont = ss(A_c,B_c,C_c,D_c);
sys_d = c2d(sys_cont,Ts);

A = sys_d.A; B = sys_d.B; C = sys_d.C; D = sys_d.D;
[K, P] = dlqr(A, B, Q, R);       % Linear quadratic regulator for
    callculating the P matrix

%% Pendulum parameters
KF=2.6; M0=3.2; M1=0.329; M=M0+M1; ls=0.44; inert=0.072; N_val=0.1446;
N01_sq=0.23315; Fr=6.2; C=0.009; gra=9.81;

a32 = -N_val^2/N01_sq*gra ; a33 = -inert*Fr/N01_sq; a34 = N_val*C/N01_sq;
a35 = inert*N_val/N01_sq; a42 = M*N_val*gra/N01_sq; a43 = N_val*Fr/N01_sq;
    a44 = -M*C/N01_sq;
a45 = -N_val^2/N01_sq; b3=inert/N01_sq; b4=-N_val/N01_sq;
b3_hat = inert/N01_sq+0.1; b4_hat = -N_val/N01_sq+0.1;

%% Animation parameters
xmin = -1;
xmax = +1;

figure;
h    = [];
h(1) = subplot(4,2,1);
```

```
h(2) = subplot(4,2,3);
h(3) = subplot(4,2,5);
h(4) = subplot(4,2,7);
h(5) = subplot(2,2,2);
h(6) = subplot(2,2,4);

Disturbance = 1;

% Hessian, F matrices of cost function
[H, F] = Controller_MPC_CostFunction(A, B, Np, Q, R, P);
s = zeros(1, Np);
s(1, 1) = 1;

for k = 1:time/Ts-1

% CONTROL ALGORITHM - Explicit MPC
tau(1, k) = -s*(H^(-1))*F*[q(1, k); qd(1, k); q(2, k); qd(2, k)];

% Voltage limitation
if abs(tau(:, k)) > 10
tau(:, k) = sign(tau(:, k)) * 10;
end

% Inverted pendulum math model
beta_x2 = (1 + N_val^2/N01_sq*(sin(q(2, k)))^2)^(-1);
qdd(:, k+1) = [beta_x2*(a32*sin(q(2, k))*cos(q(2, k))+a33*qd(1, k)+...
a34*cos(q(2, k))*(qd(2, k))+a35*sin(q(2, k))*qd(2, k)^2+b3*tau(:, k));
beta_x2*(a42*sin(q(2, k))+a43*cos(q(2, k))*qd(1, k)+...
a44*(qd(2, k))+a45*cos(q(2, k))*sin(q(2, k))*(qd(2, k))^2+b4*cos(q(2, k))*
    tau(:, k))];

qd(:, k+1) = qd(:, k) + qdd(:, k+1)*Ts;
q(:, k+1) = q(:, k) + qd(:, k+1)*Ts;
q(2, k+1) = mod(q(2, k+1) + pi, 2*pi) - pi;

% Disturbance
if mod(k, 60) == 0 && Disturbance == 1
x = rand();
if x > 0.5 && k ~= 200
q(2, k+1) = q(2, k+1) + rand()*0.12;
else
q(2, k+1) = q(2, k+1) - rand()*0.12;
end
if k == 200
q(2, k+1) = q(2, k+1) - 0.1;
end
end

% Plotting animation
plot(0, 'Parent', h(6));
hold on;
p1 = -q(1, k);
p2 = -q(1, k) + ls*exp(1i*(q(2, k)+pi/2));
line(real([p1, p2]), imag([p1, p2]));
plot(real(p2), imag(p2), '.', 'markersize', 40);
hold off;

% Center plot w.r.t. object
if q(1) > xmax
xmin = xmin + 0.1;
xmax = xmax + 0.1;
elseif q(1) < xmin
xmin = xmin - 0.1;
xmax = xmax - 0.1;
end

grid on;
axis([h(1)], [0 time/Ts-1 -1 1]); title('position')
axis([h(2)], [0 time/Ts-1 -0.5 0.5]);
axis([h(3)], [0 time/Ts-1 -1 1]);
```

```
133 axis([h(4)], [0 time/Ts-1 -5 5]);
134 axis([h(5)], [0 time/Ts-1 -10 10]);
135 axis([h(6)], [xmin-.2 xmax+.2 -.5 .5]);
136
137 % Update animation
138 drawnow;
139
140 % Plotting system variables
141 plot(q(1, 1:k), 'b', 'Parent', h(1)); % Position
142 plot(q(2, 1:k), 'b', 'Parent', h(2)); % Angle
143 plot (qd(1, 1:k), 'b', 'Parent', h(3)); % Velocity
144 plot(qdd(1, 1:k), 'b', 'Parent', h(4)); % Angular velocity
145 plot(tau(1:k), 'k', 'Parent', h(5)); % Torque
146
147 title(h(1), 'position');
148 title(h(2), 'angle');
149 title(h(3), 'velocity');
150 title(h(4), 'angular velocity');
151 title(h(5), 'torque');
152 end
```

Code 3.3: Source code of model predictive controller (Simulation).

Source code of model predictive controller (Real implementation)

```
% Pause for 3 seconds to allow initialization
pause(3);

% Clear workspace, close all figures, and clear command window
clear all;
close all;
clc;

% Add necessary paths for CLSS Praxis and hudaqlib
addpath(genpath('CLSS Praxis'), genpath('hudaqlib'))

% Create a HudaqDevice object for MF634
dev = HudaqDevice('MF634');

% Number of samples and sampling period
s = 2000;    % Total experiment samples
Ts = 0.02;   % Sampling period

% Initialize arrays for storing system states and control input
x = zeros(s, 4);     % States during the experiment
z2 = zeros(4, 1);    % Sample states
x(1, :) = [(AIRead(dev, 1)/0.15/100) 0.01*round(-13.1*AIRead(dev,
    3)) (-AIRead(dev, 2)/0.96*pi/180) 0]; % Initial values
Force = zeros(s, 1);  % Control input (force applied)

%% State-space
m_p = 0.329; m_w = 3.2; l_sp = 0.44; f_w = 6.2;
f_p = 0.009; gra = 9.81; j_a = 0.072; Ts = 0.02;

% Define continuous-time state-space matrices
A_c = [ 0   1                                   0                   0
0   -f_w/(m_w+m_p)                  0                   0
0   0                               0                   1
0   (f_w*m_p*l_sp)/(j_a*(m_w+m_p)) (m_p*l_sp*gra)/j_a     -f_p/j_a
    ];
B_c = [0   ;   1/(m_w+m_p) ;   0   ;   -m_p*l_sp/((m_w+m_p)*j_a)];
C_c = [   1   0   0   0
0   1   0   0
0   0   1   0
0   0   0   1];
D_c = [0; 0; 0; 0];

% Convert to discrete-time
sys_cont = ss(A_c, B_c, C_c, D_c);
sys_d = c2d(sys_cont, Ts);

A = sys_d.A; B = sys_d.B; C = sys_d.C; D = sys_d.D;

% Hessian, F matrices of cost function
[H, F] = Controller_MPC_CostFunction(A, B, Np, Q, R, P);
a = zeros(1, Np);
a(1, 1) = 1;

for i = 1:s-1

% MPC CONTROL ALGORITHM
Force(i) = ctranspose(-a*(H^(-1))*F*x(i, :));

% Save the data
z2(1) = AIRead(dev, 1)/0.15/100;          % Position of the cart (
    meter)
z2(2) = 0.01*round(-13.1*AIRead(dev, 3)); % Speed of the cart (m/s)
z2(3) = -AIRead(dev, 2)/0.96*pi/180;      % Angle of the pendulum (
    radian)

```

```
62          % Voltage limitation
63          if abs(Force(i)) > 10
64          Force(i) = sign(Force(i))*10;
65          end
66
67          % Apply calculated voltage
68          tic
69          while toc < Ts
70          DOWriteBit(dev, 1, 2, 1);  % Freischaltung Pendel
71          DOWriteBit(dev, 1, 2, 0);  % Channel 1 consists of DO0..DO7
72          DOWriteBit(dev, 1, 2, 1);  % DO2 Requires continuous pulse
73          AOWrite(dev, 2, Force(i)); % Apply calculated voltage
74          end
75
76          % Angular speed calculation (derivative)
77          z2_winkel = -AIRead(dev, 2)/0.96*pi/180;
78          z2(4) = (z2_winkel - z2(3))/Ts; % Angular speed of the pendulum
79
80          % Update system states
81          x(i+1, :) = ctranspose(z2);
82
83          % Check if the pendulum is out of range
84          if abs(z2(1)) > 0.3 || abs(z2(3)*180/pi) > 10    % Der Pendel ist
                ausser Bereich
85          disp('Please bring me back !');
86          pause(3);  % Wait 3 seconds
87          end
```

Code 3.4: Source code of model predictive controller (Real implementation).

Source code of cost function callculation for MPC

```
function [H, F] = CostFunctionMPC(A, B, np, Q, R, P)

[nx, nu] = size(B);          % Get system dimensions

phi = zeros(np*nx, nx);        % Preallocate matrix Phi
for i = 1:np                   % Create matrix M using powers of matrix A
phi(i*nx-nx+1:i*nx, :) = A^i;
end

% Get prediction matrix N
gamma = zeros(nx*np, nu*np);     % Preallocate matrix Gamma
NN = zeros(nx*np, nu);           % Preallocate auxiliary matrix NN with
    zeros
for i = 1:np                     % Calculate auxiliary matrix NN
NN(i*nx-nx+1:i*nx, :) = A^(i - 1) * B;
end
for i = 1:np                   % By shifting NN matrix create matrix Gamma
gamma(i*nx-nx+1:end, i*nu-nu+1:i*nu) = NN(1:np*nx-(i - 1)*nx, :);
end

% Get cost function matrices H,F
psi = kron(eye(np), R);
omega = zeros(np*nx, np*nx);  % Preallocate matrix Omega
for i = 1:np-1                % Create matrix Omega
omega(i*nx-nx+1:i*nx, i*nx-nx+1:i*nx) = Q;
end
omega(np*nx-nx+1:np*nx, np*nx-nx+1:np*nx) = P; % Add last diagonal element
H=2*(psi+gamma'*omega*gamma);
F=2*gamma'*omega*phi;

end
```

Code 3.5: Source code of cost function callculation for MPC.

Source code of sliding mode controller (Simulation)

```
clear all;
clc;

%% Parameters and Initialization
Ts = 0.02; % Sampling Time
time = 20; % Total simulation time
alpha = 6;
beta = 1.8; % Variable that changes the speed of switching or duty cycle
var_k = 10;

% Initial values
q = zeros(2, time / Ts); % Position and angle
qd = zeros(2, time / Ts); % Linear and angular velocity
qdd = zeros(2, time / Ts); % Linear and angular acceleration
q(:, 1) = [0.4; -0.1]; % Initial values for position and angle
tau = zeros(1, time / Ts); % Voltage

q_r(1) = 0; % Reference value of position
q_r(2) = 0; % Reference value of angle

qd_r = zeros(2, 1); % Reference value of linear and angular velocity

%% State-space
% Define system matrices for continuous-time and discrete-time systems
m_p = 0.329; m_w = 3.2; l_sp = 0.44; f_w = 6.2;
f_p = 0.009; gra = 9.81; j_a = 0.072; Ts = 0.02;

A_c = [0 1 0 0;
0 -f_w / (m_w + m_p) 0 0;
0 0 0 1;
0 (f_w * m_p * l_sp) / (j_a * (m_w + m_p)) (m_p * l_sp * gra) / j_a -f_p / j_a];
B_c = [0; 1 / (m_w + m_p); 0; -m_p * l_sp / ((m_w + m_p) * j_a)];
C_c = [1 0 0 0;
0 1 0 0;
0 0 1 0;
0 0 0 1];
D_c = [0; 0; 0; 0];

% Convert to discrete-time system
sys_cont = ss(A_c, B_c, C_c, D_c);
sys_d = c2d(sys_cont, Ts);

A = sys_d.A; B = sys_d.B; C = sys_d.C; D = sys_d.D;

%% Pendulum parameters
KF = 2.6; M0 = 3.2; M1 = 0.329; M = M0 + M1; ls = 0.44; inert = 0.072; N_val = 0.1446;
N01_sq = 0.23315; Fr = 6.2; C = 0.009; gra = 9.81;

% Define coefficients for inverted pendulum math model
a32 = -N_val^2 / N01_sq * gra; a33 = -inert * Fr / N01_sq; a34 = N_val * C / N01_sq;
a35 = inert * N_val / N01_sq; a42 = M * N_val * gra / N01_sq; a43 = N_val * Fr / N01_sq;
a44 = -M * C / N01_sq; a45 = -N_val^2 / N01_sq; b3 = inert / N01_sq; b4 = -N_val / N01_sq;
b3_hat = inert / N01_sq + 0.1; b4_hat = -N_val / N01_sq + 0.1;

%% Animation parameters
xmin = -1;
xmax = +1;

% Create subplots for animation
figure;
```

```
61  h = [];
62  h(1) = subplot(4, 2, 1);
63  h(2) = subplot(4, 2, 3);
64  h(3) = subplot(4, 2, 5);
65  h(4) = subplot(4, 2, 7);
66  h(5) = subplot(2, 2, 2);
67  h(6) = subplot(2, 2, 4);
68
69  Disturbance = 1;
70
71  %% Control
72  for k = 1:time / Ts - 1
73  % Control algorithm
74  s = -qd(2, k) - alpha * q(2, k) - qd(1, k) - beta * q(1, k);
75  tau(1, k) = 12.335239 * qd(1, k) + 68.877358 * q(2, k) - 3.492138 * (0.125
        - alpha) * qd(2, k)...
76  - 3.492138 * (1.7568 - beta) * qd(1, k) - (var_k * sign(s));
77
78  % Voltage limitation
79  if abs(tau(:, k)) > 10
80  tau(:, k) = sign(tau(:, k)) * 10;
81  end
82
83  % Inverted pendulum math model
84  beta_x2 = (1 + N_val^2 / N01_sq * (sin(q(2, k)))^2)^(-1);
85  qdd(:, k + 1) = [beta_x2 * (a32 * sin(q(2, k)) * cos(q(2, k)) + a33 * qd(1,
         k) + ...
86  a34 * cos(q(2, k)) * (qd(2, k)) + a35 * sin(q(2, k)) * qd(2, k)^2 + b3 *
        tau(:, k));
87  beta_x2 * (a42 * sin(q(2, k)) + a43 * cos(q(2, k)) * qd(1, k) + ...
88  a44 * (qd(2, k)) + a45 * cos(q(2, k)) * sin(q(2, k)) * (qd(2, k))^2 + b4 *
        cos(q(2, k)) * tau(:, k))];
89
90  % Update position and velocity
91  qd(:, k + 1) = qd(:, k) + qdd(:, k + 1) * Ts;
92  q(:, k + 1) = q(:, k) + qd(:, k + 1) * Ts;
93  q(2, k + 1) = mod(q(2, k + 1) + pi, 2 * pi) - pi;
94
95  % Disturbance
96  if mod(k, 60) == 0 && Disturbance == 1
97  x = rand();
98
99  if x > 0.5 && k ~= 200
100 q(2, k + 1) = q(2, k + 1) + rand() * 0.12;
101 else
102 q(2, k + 1) = q(2, k + 1) - rand() * 0.12;
103 end
104
105 if k == 200
106 q(2, k + 1) = q(2, k + 1) - 0.1;
107 end
108 end
109
110 % Animation
111 plot(0, 'Parent', h(6));
112 hold on;
113 p1 = -q(1, k);
114 p2 = -q(1, k) + ls * exp(1i * (q(2, k) + pi / 2));
115 line(real([p1, p2]), imag([p1, p2]));
116 plot(real(p2), imag(p2), '.', 'markersize', 40);
117 hold off;
118
119 % Center plot w.r.t. object
120 if q(1) > xmax
121 xmin = xmin + 0.1;
122 xmax = xmax + 0.1;
123 elseif q(1) < xmin
124 xmin = xmin - 0.1;
125 xmax = xmax - 0.1;
126 end
```

```
127 grid on;
128 axis([h(1)], [0 time / Ts - 1 -1 1]);
129 title('position');
130 axis([h(2)], [0 time / Ts - 1 -0.5 0.5]);
131 title('angle');
132 axis([h(3)], [0 time / Ts - 1 -1 1]);
133 title('velocity');
134 axis([h(4)], [0 time / Ts - 1 -5 5]);
135 title('angular velocity');
136 axis([h(5)], [0 time / Ts - 1 -10 10]);
137 title('torque');
138
139 % Update animation
140 drawnow;
141
142 % Plot results
143 plot(q(1, 1:k), 'b', 'Parent', h(1)); % Position
144 plot(q(2, 1:k), 'b', 'Parent', h(2)); % Angle
145 plot(qd(1, 1:k), 'b', 'Parent', h(3)); % Velocity
146 plot(qdd(1, 1:k), 'b', 'Parent', h(4)); % Angular velocity
147 plot(tau(1:k), 'k', 'Parent', h(5)); % Torque
148 title(h(1), 'position');
149 title(h(2), 'angle');
150 title(h(3), 'velocity');
151 title(h(4), 'angular velocity');
152 title(h(5), 'torque');
153 end
```

Code 3.6: Source code of sliding mode controller (Simulation).

Source code of sliding mode controller (Real implementation)

```
pause(3);
clear all;
close all;
clc;

addpath(genpath('CLSS Praxis'),genpath('hudaqlib'))
dev = HudaqDevice('MF634');
mn = 25;
mkdir Messung_25;

s = 2000;  %total experiment sample
Ts = 0.02; % sampling period

x = zeros(s,4); %states during experiment
z2 = zeros(4,1);% sample states
x(1,:) = [(AIRead(dev,1)/0.15/100) 0.01*round(-13.1*AIRead(dev,3)) (-AIRead
    (dev,2)/0.96*pi/180) 0]; % inital values
Force = zeros(s,1);

%% State-space
m_p      = 0.329;m_w      = 3.2;l_sp     = 0.44;f_w      = 6.2;
f_p      = 0.009;gra      = 9.81;j_a     = 0.072;Ts = 0.02;

A_c = [ 0    1                                    0                    0
0    -f_w/(m_w+m_p)                    0                    0
0    0                                 0                    1
0    (f_w*m_p*l_sp)/(j_a*(m_w+m_p)) (m_p*l_sp*gra)/j_a      -f_p/j_a];
B_c = [0   ;    1/(m_w+m_p) ;    0    ;    -m_p*l_sp/((m_w+m_p)*j_a)];
C_c = [    1    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1];
D_c = [0;0;0;0];

sys_cont = ss(A_c,B_c,C_c,D_c);
sys_d = c2d(sys_cont,Ts);

A = sys_d.A;B = sys_d.B;C = sys_d.C;D = sys_d.D;

%% Implementation
alpha = 6;
beta = 1.8; % variable that changes the speed of switching resp the duty
    cycle
var_k = 10;

for i = 1:s-1

%  CONTROL ALGORITHM

s1 = - x(i,4) - alpha * x(i,3)- x(i,2)- beta * x(i,1);
Force(i) = 12.335239*x(i,2)+68.877358*x(i,3) -3.492138*(0.125-alpha)*x(i,4)
    -3.492138*(1.7568-beta)*x(i,2)-(var_k * sign(s1));

% save the data
z2(1) = AIRead(dev,1)/0.15/100; % position of cart (meter)
z2(2) = 0.01*round(-13.1*AIRead(dev,3)); % speed of cart (m/s)
z2(3) = -AIRead(dev,2)/0.96*pi/180; % angle of pendulum (radian)

% voltage limitation
if abs(Force(i))>10
Force(i) = sign(Force(i))*10;
end

% apply calculated voltage
tic
```

```
63  while toc<Ts
64  DOWriteBit(dev,1,2,1)            % Freischaltung Pendel
65  DOWriteBit(dev,1,2,0)            % channel 1 besteht aus DO0..DO7
66  DOWriteBit(dev,1,2,1)            % DO2 requires continuous impuls
67  AOWrite(dev, 2, Force(i));       % apply calculated voltage
68  end
69
70  % angular speed calculation (derivative)
71  z2_winkel = -AIRead(dev,2)/0.96*pi/180;
72  z2(4) = (z2_winkel-z2(3))/Ts; % angular speed of pendulum
73
74  x(i+1,:) = z2';
75
76  if abs(z2(1)) > 0.3 || abs(z2(3)*180/pi) > 10    % Der Pendel ist ausser
        Bereich
77  disp('Please bring me back !');
78  pause(3);                  % wait 3 second
79  end
80
81  end
```

Code 3.7: Source code of sliding mode controller (Real implementation).