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

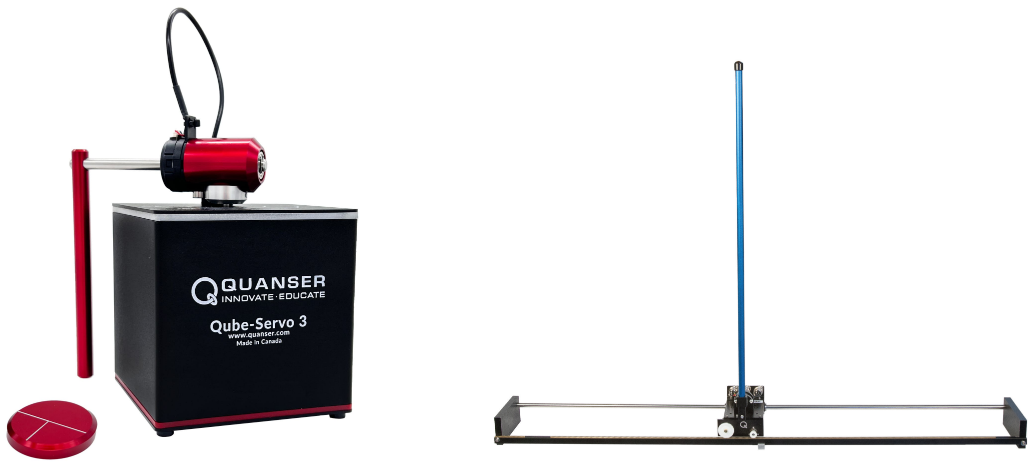
Code listing

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.

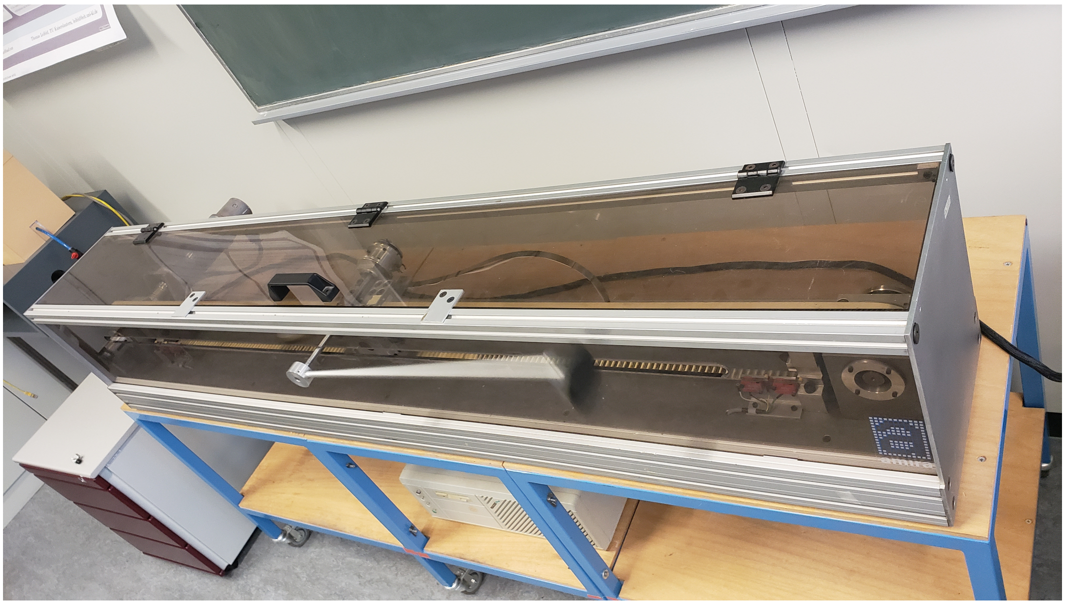


Figure 2: Inverted pendulum setup on which implementation of designed controllers was done.

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

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

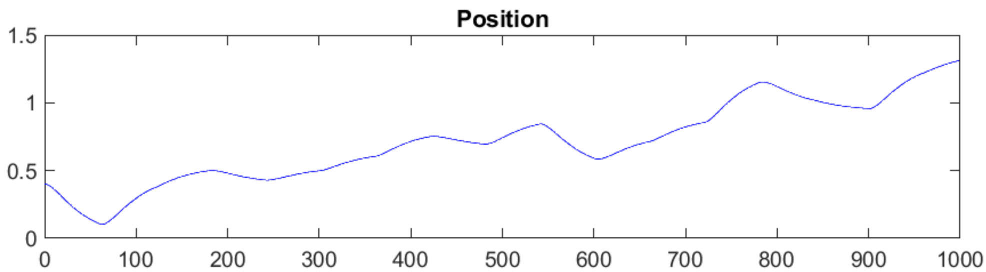


Figure 3.1: Position of the pendulum (LQR simulation).

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 iv.

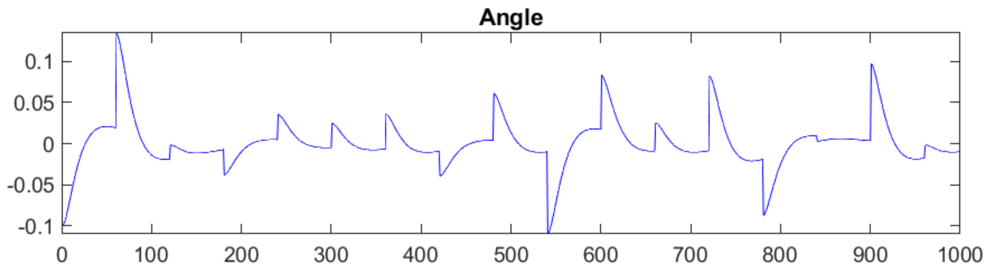


Figure 3.2: Angle of the pendulum (LQR simulation).

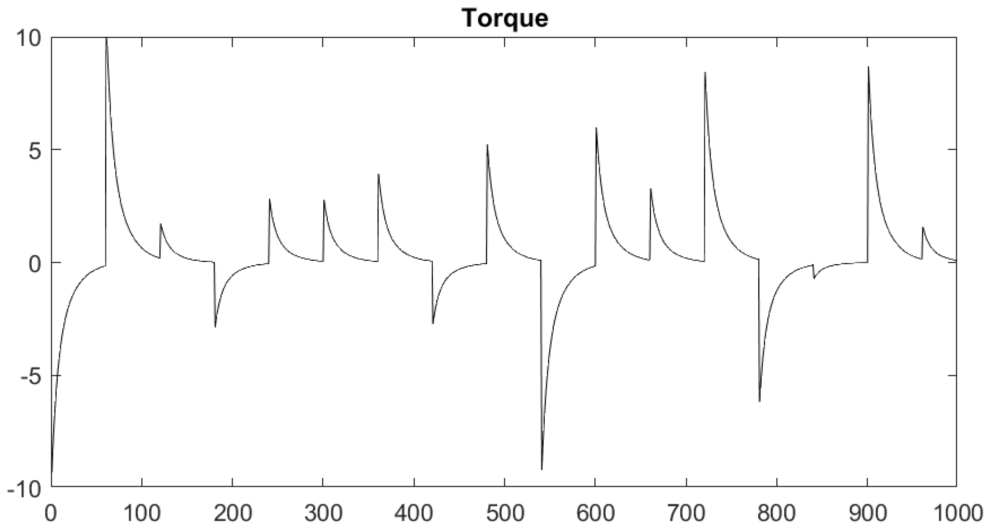


Figure 3.3: Torque of the pendulum (LQR simulation).

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 entails a steeper learning curve and a more intricate implementation process. Often likened to playing chess, MPC hinges on a profound understanding of the plant model and subsequent predictions of its behavior, recallculating 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 traits from traditional optimal control methodologies. It naturally accommodates multi-input multi-output (MIMO) plants, adeptly handles time delays, and possesses inherent robustness against modeling 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.

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

$$s.t. x_{k+1} = f(x_k, u_k) \tag{3.7}$$

$$y_k = g(x_k) \tag{3.8}$$

$$u_{min} \leq u_k \leq u_{max} \tag{3.9}$$

$$y_{min} \leq y_k \leq y_{max} \tag{3.10}$$

$$x_o = x(t) \tag{3.11}$$

Where Eq. 3.7 stands as the prediction model, Eq. 3.9 as constraints and Eq. 3.11 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 3.1 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 ix.

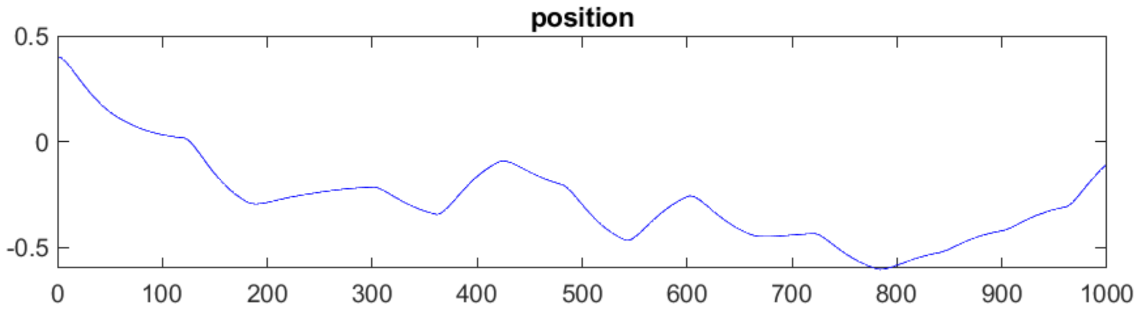


Figure 3.4: Position of the pendulum (MPC simulation).

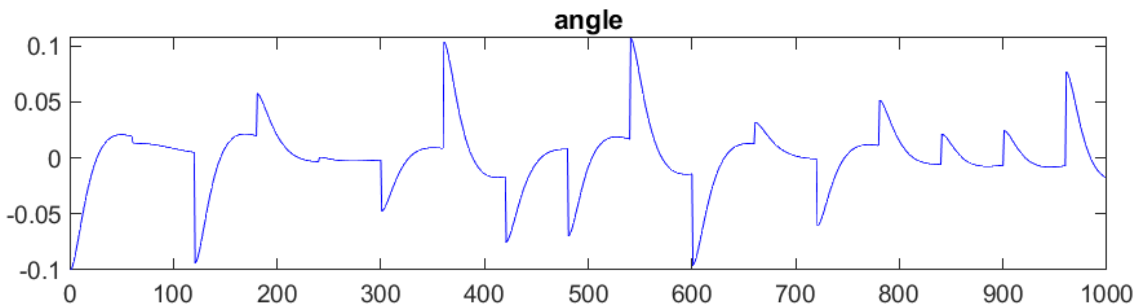


Figure 3.5: Angle of the pendulum (MPC simulation).

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

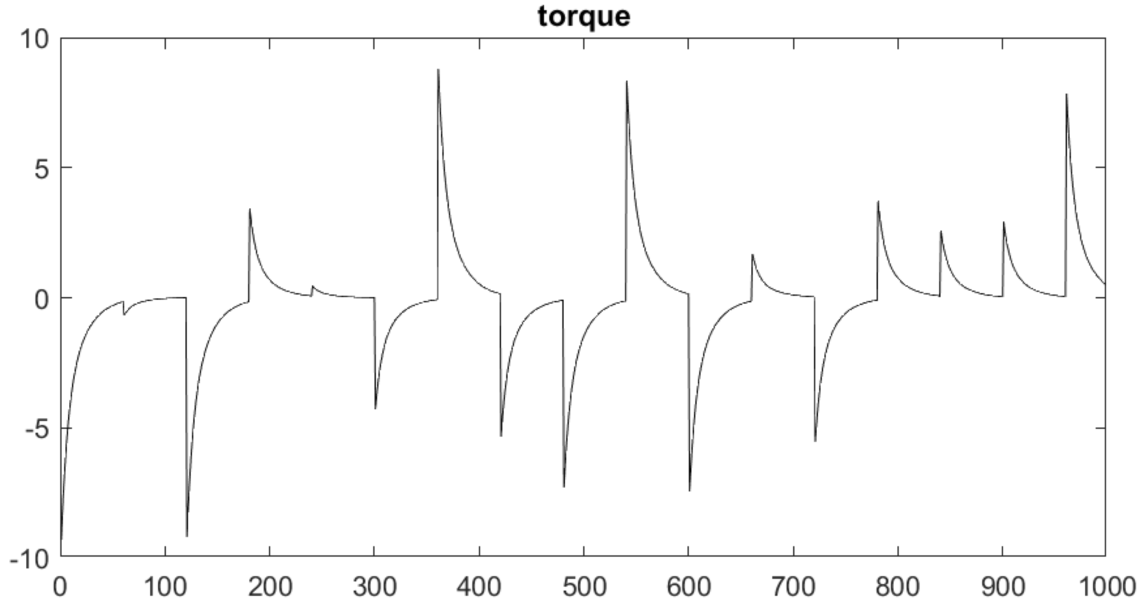


Figure 3.6: Torque of the pendulum (MPC simulation).

3.3 Sliding mode control (SMC)

3.3.1 Theory

3.3.2 Simulation

3.3.3 Implementation

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?

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;
```

```
%% Controller Parameters
Q = diag([1, 1, 1e7, 1e2]);       % State penalization
R = 1e7;                          % Input penalization
Np = 10;                          % Prediction horizon
[K, P] = dlqr(A, B, Q, R);

for k = 1:time/Ts-1

% Control Algorithm
tau(1,k) = -K*[q(1,k); qd(1,k);q(2,k); qd(2,k)]; % LQR

% 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

% Plot 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

% Update animation
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]);
axis([h(4)],[0 time/Ts-1 -5 5]);
axis([h(5)],[0 time/Ts-1 -10 10]);
axis([h(6)],[xmin-.2 xmax+.2 -.5 .5]);
```

```
% Update subplots
drawnow;

plot(q(1,1:k),'b','Parent',h(1)); %Position
plot(q(2,1:k),'b','Parent',h(2)); % Angle
plot(qd(1,1:k),'b','Parent',h(3)); % Velocity
plot(qdd(1,1:k),'b','Parent',h(4)); % Angular velocity
plot(tau(1:k),'k','Parent',h(5)); % Torque

title(h(1), 'Position');
title(h(2), 'Angle');
title(h(3), 'Velocity');
title(h(4), 'Angular Velocity');
title(h(5), 'Torque');
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
```

```
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
while toc < Ts
DOWriteBit(dev, 1, 2, 1);   % Activation pendulum
DOWriteBit(dev, 1, 2, 0);   % Channel 1 consists of DO0..DO7
DOWriteBit(dev, 1, 2, 1);   % DO2 Requires continuous pulse
AOWrite(dev, 2, Force(i));  % Apply calculated voltage
end

% Angular speed calculation (derivative)
z2_winkel = -AIRead(dev,2)/0.96*pi/180;
z2(4) = (z2_winkel - z2(3))/Ts;    % Angular speed of pendulum

% Update states
x(i+1,:) = ctranspose(z2);

% Check if the pendulum is out of range
if abs(z2(1)) > 0.3 || abs(z2(3)*180/pi) > 10
disp('Please bring me back !');
pause(3);    % Wait for 3 seconds
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]);
```

```
axis([h(4)], [0 time/Ts-1 -5 5]);
axis([h(5)], [0 time/Ts-1 -10 10]);
axis([h(6)], [xmin-.2 xmax+.2 -.5 .5]);

% Update animation
drawnow;

% Plotting system variables
plot(q(1, 1:k), 'b', 'Parent', h(1)); % Position
plot(q(2, 1:k), 'b', 'Parent', h(2)); % Angle
plot (qd(1, 1:k), 'b', 'Parent', h(3)); % Velocity
plot(qdd(1, 1:k), 'b', 'Parent', h(4)); % Angular velocity
plot(tau(1:k), 'k', 'Parent', h(5)); % Torque

title(h(1), 'position');
title(h(2), 'angle');
title(h(3), 'velocity');
title(h(4), 'angular velocity');
title(h(5), 'torque');
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)

```

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

% Apply calculated voltage
tic
while toc < Ts
DOWriteBit(dev, 1, 2, 1);  % Freischaltung Pendel
DOWriteBit(dev, 1, 2, 0);  % Channel 1 consists of DO0..DO7
DOWriteBit(dev, 1, 2, 1);  % DO2 Requires continuous pulse
AOWrite(dev, 2, Force(i)); % Apply calculated voltage
end

% Angular speed calculation (derivative)
z2_winkel = -AIRead(dev, 2)/0.96*pi/180;
z2(4) = (z2_winkel - z2(3))/Ts; % Angular speed of the pendulum

% Update system states
x(i+1, :) = ctranspose(z2);

% Check if the pendulum is out of range
if abs(z2(1)) > 0.3 || abs(z2(3)*180/pi) > 10    % Der Pendel ist ausser Bereich
disp('Please bring me back !');
pause(3);  % Wait 3 seconds
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.