Model Predictive Control for Cascaded Electrical Oscillators

Reichart, Henrik
32468
Erath, Sarah
32066
Dorka, Richard
30009

February 21, 2025

*Project for the lecture **Embedded Control** with*
Prof. Dr. Lothar Berger** and **Stephan Scholz
in winter semester 2024/25

Contents

1 Introduction

This project aims to implement a model predictive control on a system with Nth-order RLC circuits connected in cascade. It treats the implementation and simulation in Matlab, provides a brief explanation of the state space model of the simulated system, and discusses the implementation of the MPC in the programming language C/C++.

2 Essentials

2.1 Analysis of the given System

To begin with, it is of utmost importance for our project to form the state-space model of our system. The system consists of RLC circuits that are connected in cascade (s. Fig. (1)). The system is described by the following values. First, we use

$$R_1 = R_2 = R_n = 1\,\Omega \;\;\&\;\; L_1 = L_2 = L_n = 0,5\,H \;\;\&\;\; C_1 = C_2 = C_n = 0,1\,F \tag{1}$$

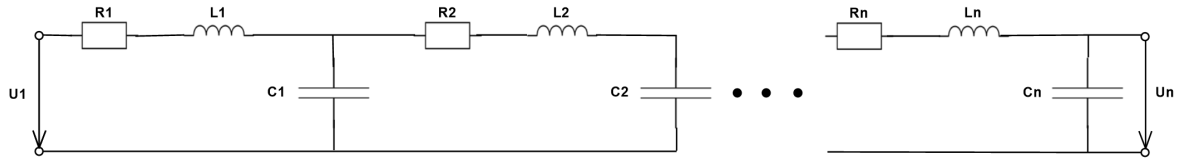

Figure 1: Cascaded RLC oscillator of order n

To derive the state-space equation $\dot{\vec{x}} = A\vec{x}(t) + B(t)$ of our model, we assume the second-order system, which can be described as:

$$M\ddot{\vec{z}}(t) + D\dot{\vec{z}}(t) + Sz(t) = Gu(t) \tag{2}$$

Where the matrix M contains the values of $L \cdot C$ and matrix D contains the values of $R \cdot C$. The introduction of the new variable

$$x_1(t) = z(t), \;\; x_2(t) = \dot{z}(t).$$

and rearrangement of the formula leads to:

$$\begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{pmatrix} = \begin{pmatrix} 0 & I \\ -M^{-1}S & M^{-1}D \end{pmatrix} \begin{pmatrix} \vec{x_1}(t) \\ \vec{x_2}(t) \end{pmatrix} + \begin{pmatrix} 0 \\ M^{-1}G \end{pmatrix} u(t) \tag{3}$$

Therefore

$$A = \begin{pmatrix} 0 & I \\ -M^{-1}S & M^{-1}D \end{pmatrix} \;\;\&\;\; B = \begin{pmatrix} 0 \\ M^{-1}G \end{pmatrix}$$

With

$$M = LC \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}, D = RC \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}, S = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 \\ 0 & -1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix},$$

$$G = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

2.2 Model Predictive Control

MPC or model predictive control is used to model and predict a system's dynamics (s.(2)). Therefore it needs the following Signals:

1. Reference trajectory
2. Control variables
3. Measured outputs

The reference trajectory gives the MPC a direction in the form of different desired values. The control variables will regulate the system, so the output matches the reference. The last variables are the measured outputs or states of the system. With it, the MPC can evaluate the current state of the system and set its control variables accordingly.

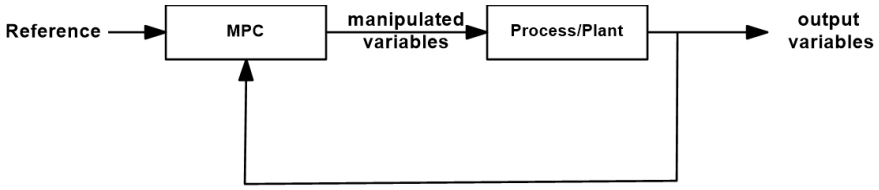

Figure 2: Model Predictive Control

Inside the MPC a plant model is used to predict the system behavior with a mathematical representation of the plant the state space model. For the prediction of the system the MPC needs the prediction horizon. The prediction horizon states how many time steps the MPC

should look in the future. With it the MPC simulates multiple scenarios to find the path closest to the reference. The prediction horizon should be long enough to react to changes depending on the significant dynamics of the plant/system but small enough to not waste computational effort due to unforeseeable events. The MPC's sampling time is as important as the prediction horizon. It mainly defines the rate at which the controller executes the control algorithm, but it also defines the computational effort and the reaction time it can take to react to disturbances. Another design parameter of the MPC is the control horizon. This horizon states the number of control moves until the inputs are held constant. By changing this parameter the MPC can balance computational effort and performance.

The key aspect of the MPC is the Cost Function J (s. Eq. (4)). It represents the weighted square sum of predicted errors and input increments. Basically, it assigns a cost to a trajectory. The minimization of the cost of a dynamically feasible trajectory is the optimization problem of the MPC. The optimization problem is that at each time step the MPC has to select the best control action to drive the predicted output to the reference.

$$J(x_{1:T}, u_{1:t}) = \sum_{t \in T} (e_t^T \lambda_e e_t + u_t^T \lambda_{ci} u_t + \Delta u_t^T \lambda_{ce} \Delta u_t) \tag{4}$$

Another important part of minimization is the constraints that encode domains of allowed states, including the rate of change in inputs and outputs. Constraints are distinguished into soft and hard. Hard constraints must not be exceeded or dropped below, while soft constraints can be exceeded or dropped below but should not.

3 System Model and Control using Matlab

3.1 Introduction

The codes (see code: 17) primary goal is to develop a controller for n RCL filters in series, analyze the system's controllability, and simulate the system under disturbance. The implementation is carried out using MATLAB, with the workflow including parameter initialization, system modeling, and simulation of closed-loop behavior.

3.2 MATLAB Implementation

The MATLAB implementation is structured into several parts. Key aspects are described below:

3.2.1 Initialization of Parameters

Parameters such as resistance R, inductance L, and capacitance C are initialized for n RCL filters in series:

```
n = 3; % Number of RCL filters in series
R = ones(1, n);  % Resistance array (Ohms)
L = 0.5 * ones(1, n); % Inductance array (Henrys)
C = 0.1 * ones(1, n); % Capacitance array (Farads)
```

Listing 1: Initialization of RCL Parameters

3.2.2 State-Space Model

A state-space model is developed for the RCL system. The disturbance matrix and other system matrices are defined as:

```
E1 = [0; 0.1]; % Example disturbance effects
A_new, B_new = RLC_ss_V2(M, N, A1, B1);
E = [inv(M)*[0;0.1;0];zeros(3,1)];
```

Listing 2: Disturbance and State-Space Matrices

3.2.3 Discrete-Time System

To implement MPC, the continuous-time system is discretized with a sampling time T_s:

```
Ts = 0.005; % Sampling time
sys_disc = c2d(sys_cont, Ts, 'zoh');
```

Listing 3: Discretization of the System

3.2.4 Model Predictive Control

An MPC controller is created using a prediction horizon, control horizon, and specified weights for control input and output variables:

```
predictionHorizon = 5;
controlHorizon = 2;
MV = struct('Min', -30, 'Max', 30);
Weights = struct('ManipulatedVariables', 1.0, '
   ManipulatedVariablesRate', 280, 'OutputVariables', 100000);
mpcController = mpc(sys_disc, Ts, predictionHorizon, controlHorizon,
   Weights, MV);
```

Listing 4: MPC Controller Setup

3.2.5 Simulation

The simulation involves generating reference signals and disturbances, and computing control inputs over a specified time horizon. The closed-loop behavior is evaluated with the state-space system dynamics:

```
for k = 1:length(highResTime)
    if mod(k-1, round(controlResolution/Ts)) == 0
        currentOutput = Cd * x; % Measured output
        u(controlIdx) = mpcmove(mpcController, mpcState,
            currentOutput, controlRef(controlIdx), []);
    end
    x = Ad * x + Bd * currentControl + Ed * w(k);
    y(k) = Cd * x;
end
```

Listing 5: Simulation Loop

3.3 Results

The simulation results demonstrate the control performance. The figure 3 below illustrates the control input, disturbances, and system output, with the help of a step response.

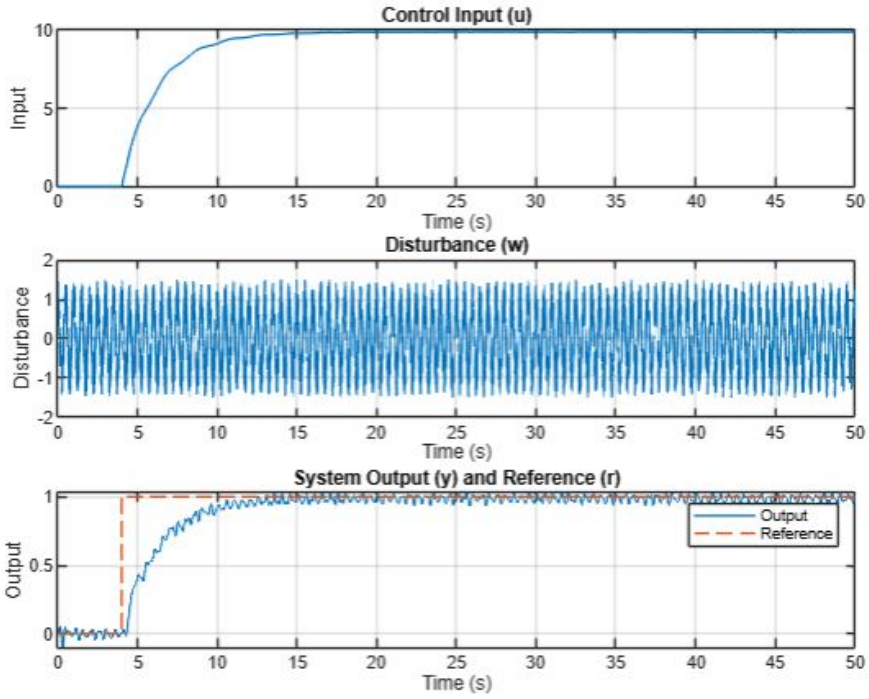

Figure 3: Control Input Noise and output Over Time.

3.4 Weight Optimization

To determine the most suitable weights for the MPC controller, different weight variations for manipulated variables, manipulated variable rates, and output variables are tested. For this, a different code was used (see listing 18), to plot different settings onto one graph. The study explores how different weight configurations affect system performance.

3.4.1 Configuration

These weight settings are used in the Model Predictive Control (MPC) framework to fine-tune control performance by adjusting the relative importance of different control objectives. The defaultWeights array sets baseline values for manipulated variable penalties (control effort), rate of change of the manipulated variables (smoothness of control actions), and output variables (tracking performance). The variations in manipulatedVars, manipulatedVarsRate, and outputVars explore different tuning configurations, affecting how aggressively or smoothly the controller responds to disturbances and reference changes. By iterating through these variations, the study identifies the most suitable weight configuration for optimal system performance.

```
% Default weights
defaultWeights = [0, 250, 100000];
% Variations
```

```
manipulatedVars = [0.001, 0.01, 0.1, 1000.0];
manipulatedVarsRate = [250, 280, 360, 450];
outputVars = [100000, 110000, 115000, 101000];
```

Listing 6: Weight Variations for MPC

3.4.2 Results

The simulation results (see figure 4) demonstrate the control performance, showing the ideal output and system output over time, with different settings. The figure shows that the best settings for the ManipulatedVariables is 0.1, for the ManipulatedVariablesRate is 280, and for the OutputVariables 100000.

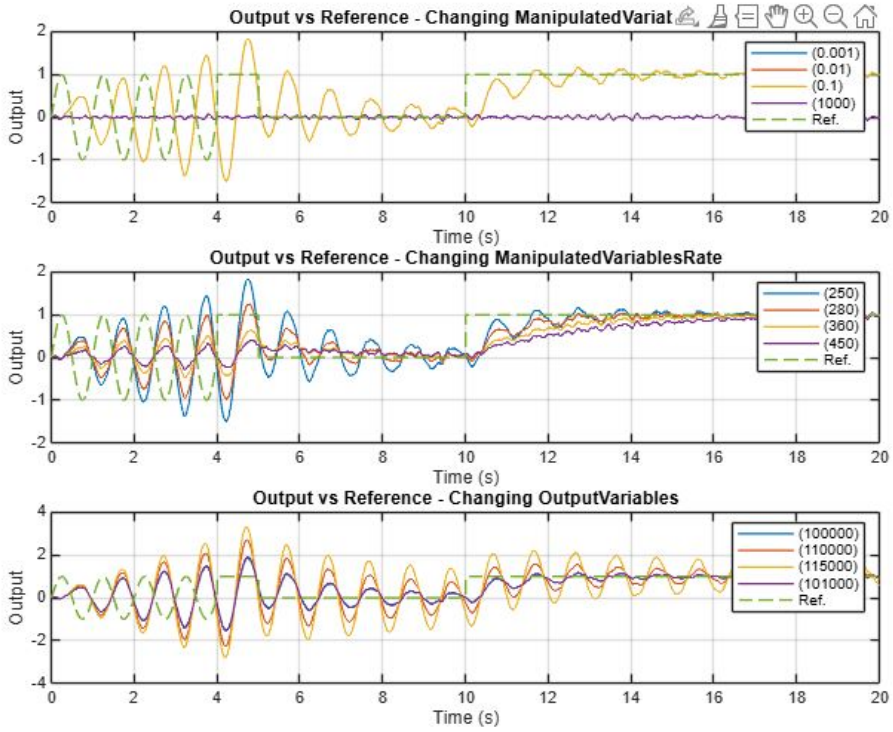

Figure 4: Result showing different weights

3.5 Conclusion

The implemented MPC controller effectively regulates the RCL system under disturbances, ensuring desired output behavior. The weight optimization study provides insight into how different configurations affect system performance. Future work could involve further fine-tuning of MPC parameters for robustness and applying the approach to nonlinear systems.

4 Control using C++ and NLopt

In the following approach a similar Matlab Code as described in Chap. 3 is used for the system simulation. However, the control algorithm of the MPC is implemented in C++ (see code 19) utilizing the NLopt library [3].

4.1 Matlab Integration

As shown in Fig. 5 the C/C++ code is embedded into the Matlab code. The Matlab code handles the simulation part including the simulation of the disturbances and the reactions of the system to the input i.e. the system state and output. The C/C++ code implements the MPC controller itself and is called in the Matlab code as a MEX function. The Matlab command to compute the control input in code 17 `mpcmove()` is replaced by a call to a custom MEX-function `custom_mpc()`. This function takes the previously computed matrices of the state space representation of the cascaded system, a value for the prediction and control horizon of the MPC, the last computed control input, an estimation of the current state of the system including disturbances and the reference trajectory over the upcoming prediction horizon.

In the MEX-function itself i.e. in the C++ code these values are then first parsed to standard C++ data types namely `std::vector<std::vector<double>>` for the matrices, `std::vector<double>` for the one-dimensional vectors and `int` or `double` for the scalar values.

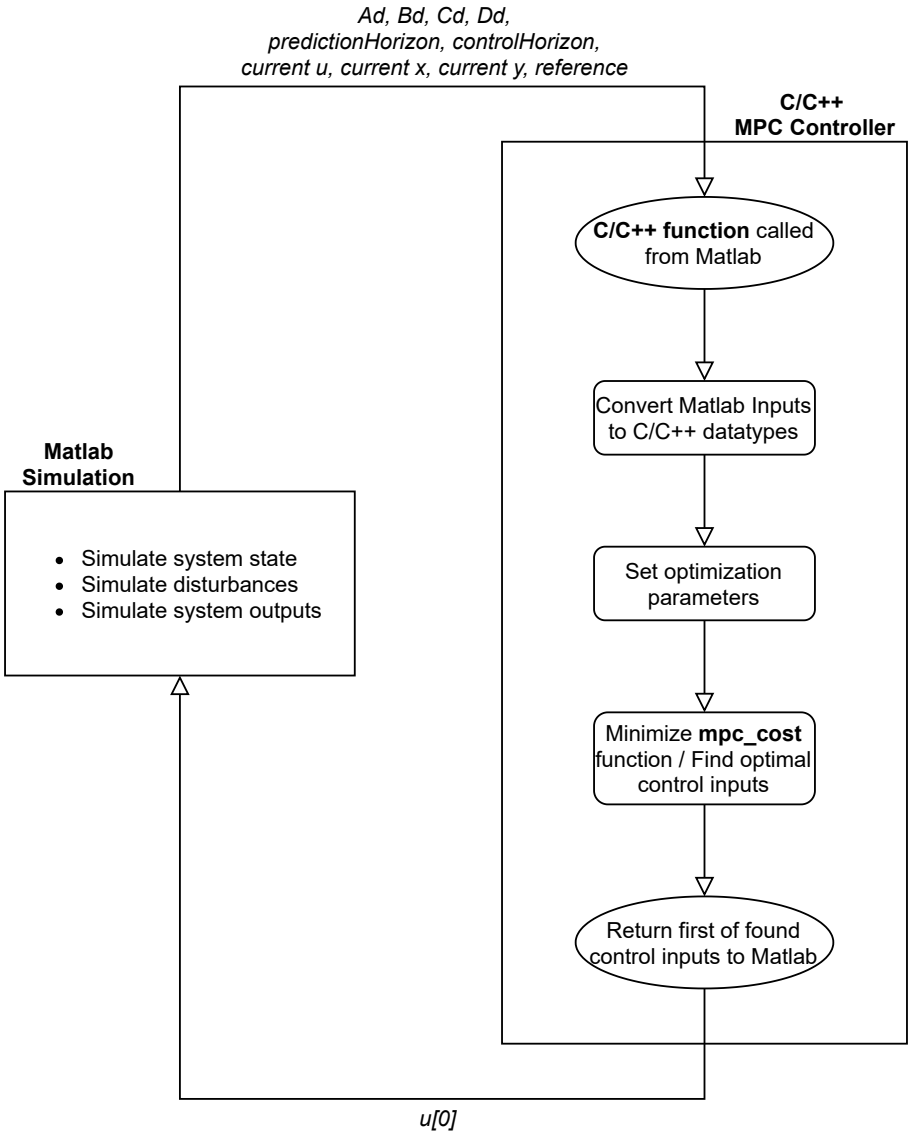

Figure 5: C/C++ Program Overview

4.2 MPC Control and Optimization Problem

To implement the MPC controller an optimization problem needs to be solved. To do this in C++ the low-level optimizer NLopt [3] can be used. It allows for the setting of constraints to the control input and offers the choice of multiple optimization algorithms.

Following the overview in Fig. 5 after converting the Matlab inputs the optimization parameters need to be set:

```
1 nlopt::opt opt(nlopt::LD_MMA, controlHorizon);
2 opt.set_min_objective(mpc_cost, &mpc_data);
3 opt.set_lower_bounds(-2.0);
```

```
4 opt.set_upper_bounds(2.0);
5 opt.set_xtol_rel(1e-10);
6 opt.set_ftol_rel(1e-10);
7 opt.set_maxeval(1000);
```

Listing 7: Setting of the optimization parameters

In the first line, a `nlopt::opt` object is created. With the constructor, the dimensionality of the optimization problem, equal to the control horizon, and an optimization algorithm are specified. Nlopt offers a variety of gradient-based or -free and global or local optimization algorithms. With this code the gradient-based algorithm MMA (Method of Moving Asymptotes) [2] and gradient-free algorithm COBYLA (Constrained Optimization BY Linear Approximations) [1] were tested. In the second line the function to be minimized `mpc_cost` is given as well as additional data that is needed for the function in the form of the pointer `&mpc_data`. This structure contains the current system state, the reference trajectory, the values of the control and prediction horizon as well as the current input and output of the system. Next the upper and lower bound of the optimization i.e. the input to be computed need to be set depending on the reference trajectory and the capabilities of the real system. The last three lines define stopping criteria for the optimization preventing it from taking too long or getting stuck. The optimization stops when an optimization step causes a change in the cost function or the input parameters that is less than the specified values or if the maximum number of evaluations is reached.

The command

```
1 opt.optimize(u_init, min_cost);
```

starts the optimization. The optimizer is soft-started with an initial guess for the input vector `u_init` with its elements being equal to the last control input to the system. The optimal control input vector after the optimization is completed is also stored in `u_init`. Its first element is returned to the Matlab simulation as the next control input.

4.2.1 Cost Function

The cost function to be optimized is formulated in the function `mpc_cost`. An overview of the process can be seen in Fig. 6. First the state variable `x` and the output variable `y` are initialized with the last known system state and output respectively. Also three weighting factors `lambda_e`, `lambda_ce` and `lambda_ci` are defined. `lambda_e` denotes the weight of the tracking error i.e. the deviation of the output from the reference trajectory. For our application, this should be the main factor and is provided with a value of 3. `lambda_ce` is the weight of the control effort which is the difference between the current and the last control input. Penalizing

the control effort smoothens the control input trajectory. However this is less important than following the reference trajectory correctly, so this gets a value of 0.03. Lastly `lambda_ci` can be used to penalize large control inputs. Since the control input is already restricted by the boundary conditions as explained before, this is only weighted by a very small factor of 10^{-4}.

```
State x = mpc_data->x0;
//...//
double y = mpc_data->lastOutput;
//...//
double lambda_e = 3.0;
double lambda_ce = 0.03;
double lambda_ci = 0.0001;
```

Listing 8: Setting of the cost function parameters

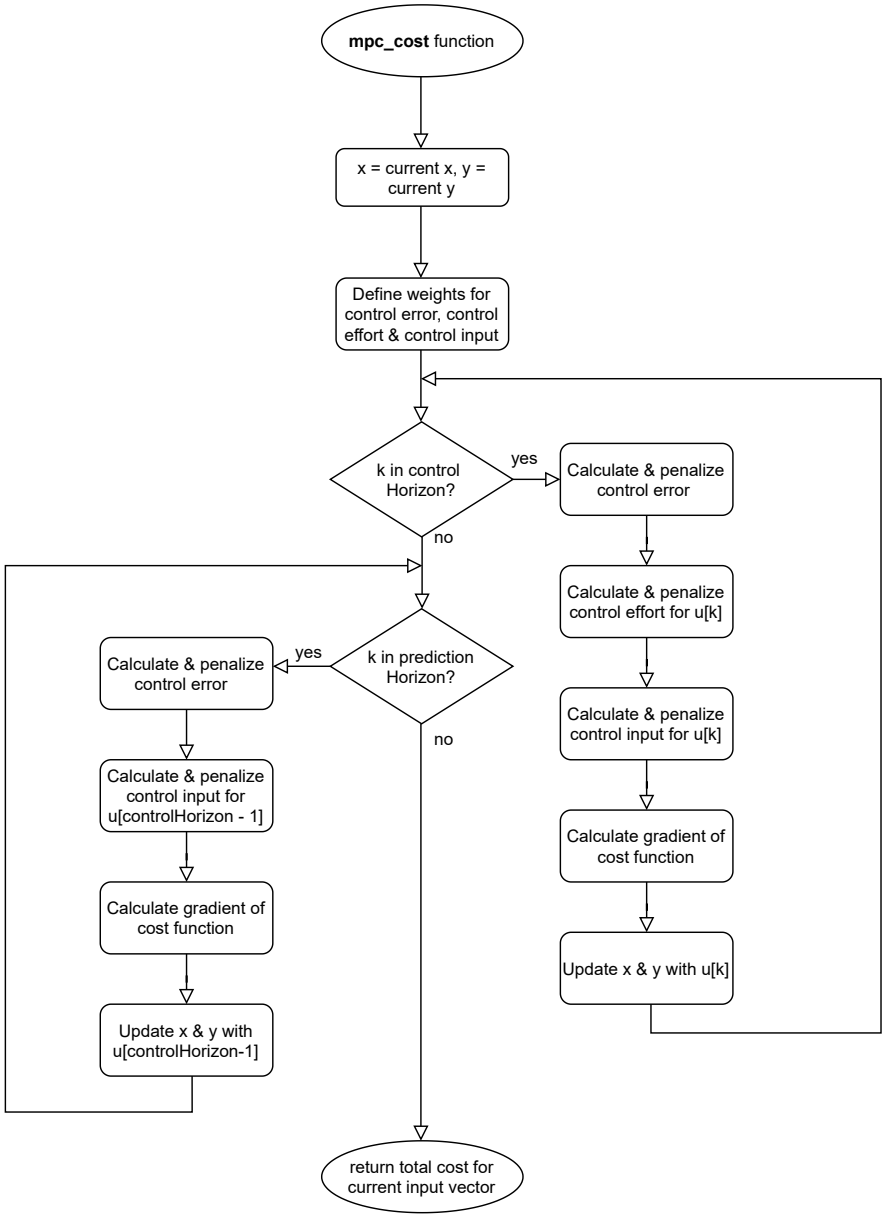

Figure 6: C/C++ Cost Function

For calculating the cost according to the formula in Eq. 4 the program iterates over the elements of the control input vector under test. First, for each time step of the control horizon, the current output is used to penalize the tracking error i.e. the deviation of the calculated output from the trajectory. Subsequently, the control effort i.e. the change in the control effort between the current and the previous time step is penalized. Next, the cost of the control inputs is calculated. Lastly, a state space simulation is carried out to update the state and output of the system. Afterward, the control input is kept constant at the value of the last time step of the control horizon as the program iterates over the remaining prediction horizon. Again the tracking error and control input magnitude are penalized with their respective weights and a

state space simulation is carried out. As the control input is not varied anymore, the control effort can be omitted. If a gradient-based algorithm is used also the gradient of the cost function must be updated. These steps will explained in more detail in the following:

1. **Tracking Error:**
   The tracking error $e[k]$ is calculated using the following formula:

$$e[k] = y[k] - r[k] \tag{5}$$

   with $y[k]$ being the current output and $r[k]$ being the corresponding value of the reference trajectory. This means the cost of the tracking error is updated as follows:

$$cost_e[k] = cost_e[k-1] + \lambda_e \cdot e[k]^2 \tag{6}$$

   And the total cost of the tracking error can be written as:

$$cost_e = \lambda_e \sum_k e[k]^2 \tag{7}$$

   This translates to the following C/C++ code:

```
error = y - mpc_data->r[k];
cost += lambda_e * error * error;
```

Listing 9: Calculation of the tracking error cost

   (a) **Gradient Calculation:**
   Using the state space equations Eq. 7 can be rewritten as:

$$\begin{aligned} cost_e &= \lambda_e \sum_k (y[k] - r[k])^2 \\ &= \lambda_e \sum_k (Cx[k] + Du[k] - r[k])^2 \\ &= \lambda_e \sum_k (C \cdot (Ax[k-1] + Bu[k-1]) + Du[k] - r[k])^2 \end{aligned}$$

   The partial derivative for each $u[k]$ in this case is:

$$\frac{\partial cost_e}{\partial u[k]} = 2\lambda_e \cdot \left[(D \cdot e[k]) + (CB \cdot e[k+1]) \right] \tag{8}$$

   Which is implemented in the C/C++ code as follows:

```
if (!grad.empty()) {
    if (k >= 0 && k < grad.size()) {
```

```
            grad[k] += 2 * lambda_e * Dd * error;
            if (k > 0){
                grad[k-1] += 2 * lambda_e * Cd_Bd * error;
            }
        }
    }
```

Listing 10: Calculation of the Gradient (tracking error term)

Here `Cd_Bd` is a scalar which equals the product of the matrices C and B and has been calculated before.

2. **Control Effort:**
   The control effort $\Delta u[k]$ is calculated using the following formula:

$$\Delta u[k] = u[k] - u[k-1] \tag{9}$$

With $u[k]$ being the current and $u[k-1]$ being the previous control input. This means the cost of the control effort is updated as follows:

$$cost_{ce}[k] = cost_{ce}[k-1] + \lambda_{ce} \cdot \Delta u[k] \tag{10}$$

And the total cost of the control effort can be written as:

$$cost_{ce} = \lambda_{ce} \sum_{k} \Delta u[k]^2 \tag{11}$$

This translates to the following C/C++ code:

```
if (k > 0) {
    delta_u = u[k] - u[k - 1];
}
if (k == 0) {
    delta_u = u[k] - mpc_data->lastInput;
}
cost += lambda_ce * delta_u * delta_u;
```

Listing 11: Calculation of the control effort cost

Where `lastInput` is the last measured/simulated input that was received from the Matlab simulation.

   (a) **Gradient Calculation:**

Eq. 11 can be rewritten as:

$$cost_{ce} = \lambda_{ce} \sum_{k} (u[k] - u[k-1])^2$$

The partial derivative for each u[k] in this case is:

$$\begin{aligned} \frac{\partial cost_{ce}}{\partial u[k]} &= 2 \cdot (\Delta u[k] - \Delta u[k+1]) \\ &= 2 \cdot (\Delta u[k] - u[k+1] - u[k]) \end{aligned} \tag{12}$$

Which is implemented in the C/C++ code as follows:

```
if (!grad.empty()) {
    grad[k] += lambda_ce * 2 * (k == mpc_data->cHorizon-1 ?
        delta_u : (delta_u - u[k+1] + u[k]));
}
```

Listing 12: Calculation of the Gradient (control effort term)

3. **Control Input:** The cost of the manipulated variables is updated as follows:

$$cost\_ci[k] = cost_{ci}[k-1] + \lambda_{ci} \cdot u[k]^2 \tag{13}$$

And the total cost of the control input can be written as:

$$cost_{ci} = \lambda_{ci} \sum_{k} u[k]^2 \tag{14}$$

This translates to the following C/C++ code when iterating over the control horizon:

```
cost += lambda_ci * u[k]*u[k];
```

Listing 13: Calculation of the manipulated variables cost (control horizon)

And the following code when iterating over the prediction horizon i.e., keeping the control input constant:

```
cost += lambda_ci * u[mpc_data->cHorizon-1]*u[mpc_data->cHorizon
    -1];
```

Listing 14: Calculation of the manipulated variables cost (prediction horizon)

(a) **Gradient Calculation:** The partial derivative of Eq. 14 for each u[k] is:

$$\frac{\partial cost_{ci}}{\partial u[k]} = 2 \cdot \lambda_{ci} \cdot u[k] \tag{15}$$

Which is implemented for the control horizon in the C/C++ code as follows:

```
if (!grad.empty()) {
    grad[k] += 2 * lambda_ci * u[k];
}
```

Listing 15: Calculation of the Gradient (control input term)

4. **State Space Simulation** At the end of each iteration, a simulation step must be executed to update the output of the system `y` according to the current element of the control input vector. For this a function `simulate_step` which takes the current system state, the current control input, and the state space matrices is implemented:

```
double simulate_step(State &x, double u, const std::vector<std::
    vector<double>> &A, const std::vector<std::vector<double>> &
    Bd, const std::vector<double> &Cd, double Dd) {
     State x_next(x.size(), 0.0);

     // Compute x_next = Ad * x + Bd * u
     for (size_t i = 0; i < A.size(); ++i) {
         for (size_t j = 0; j < A[i].size(); ++j) {
             x_next[i] += A[i][j] * x[j];
         }
         x_next[i] += Bd[i][0] * u;
     }

     // Compute output y = Cd * x_next + Dd * u
     double y = 0.0;
     for (size_t i = 0; i < Cd.size(); ++i) {
         y += Cd[i] * x_next[i];
     }
     y += Dd * u;

     x = x_next;

     return y;
}
```

Listing 16: Simulation Step function

4.3 Simulation Results

In Fig. 7 an exemplary result of the simulation can be seen. The parameters in this example are the following:

- Prediction horizon: 5
- Control horizon: 2
- Nlopt algorithm: MMA
- Sampling time: 2s
- Weight tracking error: $\lambda_e = 3$
- Weight control effort: $\lambda_{ce} = 0.03$
- Weight control input: $\lambda_{ci} = 0.0001$
- Number of cascaded filters: $n = 4$
- $R = L = C = 1$

This shows a comparison between the Matlab MPC and our custom C/C++ MPC control input and system output. Both controllers manage to track the reference trajectory well even with added disturbances. The custom MPC reacts quicker but also produces more overshoot than the Matlab MPC which tends to lag behind the reference trajectory. Also given the same weights, our custom MPC produces a smoother control input signal.

However, it must be noted that especially the custom MPC controller is very sensitive to parameter changes i.e., changes in sampling time, prediction horizon, control horizon, or the weighting factors. It would be useful to develop a strategy to better match these parameters to the system and the reference trajectory to improve the robustness of the controller.

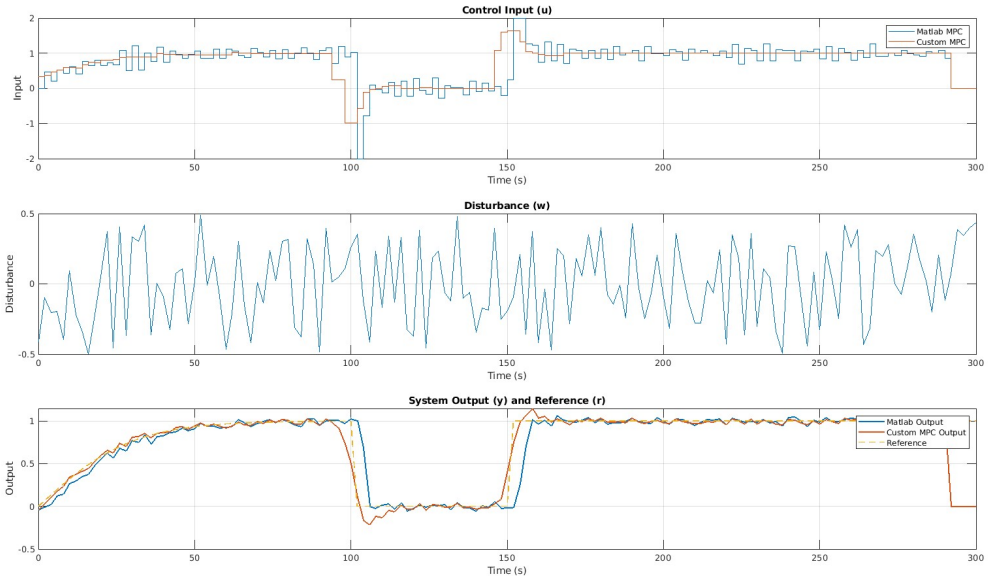

Figure 7: Simulation results: Control input, disturbance, and system output.

5 Appendix/Code

Matlab-Code

```
%%
% Clear workspace
clear; clc; close all;

% Define RCL parameters
n = 4; % Number of RCL filters in series
R = ones(1, n);  % Resistance array (Ohms)
L = 0.5 * ones(1, n); % Inductance array (Henrys)
C = 0.1 * ones(1, n); % Capacitance array (Farads)

% Continuous-time state-space model for a single RCL filter
[M, N, A1, B1] = RLC_osc(R, L, C);
C1 = [1 0];
D1 = 0;

% Disturbance matrix for a single RCL filter
E1 = [0; 0.1]; % Example disturbance effects

% Build the state-space model for n filters in series using the
   RLC_ss function
A = A1;
B = B1;
C = C1;
E = E1;

[A_new, B_new] = RLC_ss_V2(M, N, A1, B1);

% Dynamically calculate matrix sizes
rows_A = size(A, 1);
cols_A = size(A, 2);
rows_A_new = size(A_new, 1);
cols_A_new = size(A_new, 2);

% Update A with consistent dimensions
%A = [A, zeros(rows_A, cols_A_new); zeros(rows_A_new, cols_A), A_new
   ];
```

```
35 A = A_new
36
37 % Update B with zero padding to match dimensions
38 %B = [B; zeros(rows_A_new, size(B, 2))];
39 B = B_new
40
41 % Expand E with consistent zero padding
42 % E = [E; zeros(size(A, 1) - size(E, 1), size(E, 2))];
43
44 %E = [inv(M)*[0;0.1;0];zeros(3,1)]
45 E = [inv(M) * [zeros(size(M, 1) - 1, 1); 0.1]; zeros(size(A, 1) -
       size(M, 1), 1)];
46
47 % Update C with consistent dimensions
48 %C = [C, zeros(size(C, 1), size(A, 2) - size(C, 2))];
49 C = [zeros(1,n-1),1,zeros(1,n)]
50
51 D = 0; % Direct feedthrough remains zero
52
53 % Continuous-time disturbance system
54 sys_cont_dist = ss(A, E, C, D);
55
56 % Continuous-time state-space system
57 sys_cont = ss(A, B, C, D);
58
59 % Discretize the system for MPC (sampling time Ts)
60 Ts = 0.005; % Sampling time
61 sys_disc = c2d(sys_cont, Ts, 'zoh');
62 sys_disc_dist = c2d(sys_cont_dist, Ts, 'zoh');
63
64 % Extract discrete-time matrices
65 [Ad, Bd, Cd, Dd] = ssdata(sys_disc);
66
67 % Compute Kalman controllability matrix
68 Cm = [Bd, Ad * Bd, Ad^2 * Bd, Ad^3 * Bd]; % Extend as needed for
       system order
69 rank_Cm = rank(Cm);
70
71 % Display controllability information
72 disp('Rank of the controllability matrix:');
```

```
disp(rank_Cm);
if rank_Cm == size(Ad, 1)
    disp('The system is controllable.');
else
    disp('The system is not controllable.');
end
Ed = sys_disc_dist.B;

% Define prediction and control horizons
predictionHorizon = 5; % Reduced prediction horizon
controlHorizon = 2;

% Set up MPC variables
MV = struct('Min', -30, 'Max', 30);  % Control input constraints
Weights = struct('ManipulatedVariables', 1.0, '
   ManipulatedVariablesRate', 280, 'OutputVariables', 1000000);

% Create MPC controller using variables directly
mpcController = mpc(sys_disc, Ts, predictionHorizon, controlHorizon,
   Weights, MV);

% Initial conditions
x0 = zeros(size(A, 1), 1); % Initial state (all filters start at rest
   )
Tsim = 50;          % Simulation time in seconds

% Generate time-varying reference signal
highResTime = 0:Ts:Tsim;
highResRef = zeros(size(highResTime));
for k = 1:length(highResTime)
    if highResTime(k) <= 4
        %highResRef(k) = sin(2*pi*1*highResTime(k)); % Sinusoidal
           signal
        highResRef(k)=0;
    elseif highResTime(k) > 5 && highResTime(k) <= 10
        highResRef(k) = 1; % Zero between 5 and 10 seconds
    else
        highResRef(k) = 1; % Set to 1 after 10 seconds
    end
end
```

```
109
110 % Simulate disturbance
111 w = 1 * (rand(length(highResTime), 1) - 0.5); % Random disturbance
112 for k = 1:length(highResTime)
113     w(k) = w(k) + cos(2 * pi * 2 * highResTime(k)); % Cosine with
        amplitude 1 and frequency of 2 Hz
114 end
115
116 % Define control input computation resolution
117 controlResolution = 0.01; % Coarser resolution for control
    computation
118 controlTime = 0:controlResolution:Tsim;
119 controlRef = interp1(highResTime, highResRef, controlTime, 'linear');
120 controlDisturbance = interp1(highResTime, w, controlTime, 'linear');
121
122 % Initialize the MPC state object
123 mpcState = mpcstate(mpcController); % Track the controller's internal
     state
124
125 % Closed-loop simulation
126 y = zeros(length(highResTime), 1); % Pre-allocate output array
127 u = zeros(length(controlTime), 1); % Pre-allocate input array
128 x = x0;                     % Initial state
129
130 controlIdx = 1; % Initialize control index
131
132 for k = 1:length(highResTime)
133     % Compute control only at controlResolution intervals
134     if mod(k-1, round(controlResolution/Ts)) == 0
135         % Compute current output based on system state
136         currentOutput = Cd * x; % Measured output
137
138         % Compute optimal control action
139         u(controlIdx) = mpcmove(mpcController, mpcState,
            currentOutput, controlRef(controlIdx), []);
140         controlIdx = controlIdx + 1;
141     end
142
143     % Update system states (plant dynamics with disturbance)
144     currentControl = u(max(controlIdx-1, 1)); % Use last computed
```

```
        control input
145     x = Ad * x + Bd * currentControl + Ed * w(k); % Update state with
            disturbance
146
147     y(k) = Cd * x; % Compute output for plotting
148 end
149
150 % Plot results
151 figure;
152
153 subplot(3,1,1);
154 stairs(controlTime, u, 'LineWidth', 1);
155 grid on;
156 title('Control Input (u)');
157 xlabel('Time (s)');
158 ylabel('Input');
159
160 subplot(3,1,2);
161 plot(highResTime, w, 'LineWidth', 1);
162 grid on;
163 title('Disturbance (w)');
164 xlabel('Time (s)');
165 ylabel('Disturbance');
166
167 subplot(3,1,3);
168 plot(highResTime, y, 'LineWidth', 1); hold on;
169 plot(highResTime, highResRef, '--', 'LineWidth', 0.5);
170 grid on;
171 title('System Output (y) and Reference (r)');
172 xlabel('Time (s)');
173 ylabel('Output');
174 legend('Output', 'Reference');
175
176 % Save the figure as PNG
177 exportgraphics(gcf, 'control_results.png', 'Resolution', 800);
```

Listing 17: MATLAB Code for MPC of a Linear System

```
1 %% Clear workspace
2 clear; clc; close all;
3
```

```
% Define RCL parameters
n = 3; % Number of RCL filters in series
R = ones(1, n);  % Resistance array (Ohms)
L = 0.5 * ones(1, n); % Inductance array (Henrys)
C = 0.1 * ones(1, n); % Capacitance array (Farads)

%% Default weights and variations
% Default weights
defaultWeights = [0, 250, 100000];
% Variations
manipulatedVars = [0.001, 0.01, 0.1, 1000.0];
manipulatedVarsRate = [250, 280, 360, 450];
outputVars = [100000, 110000, 115000, 101000];

% Combine variations for display
displayMatrix = [manipulatedVars; manipulatedVarsRate; outputVars];

%% Continuous-time state-space model
[M, N, A1, B1] = RLC_osc(R, L, C);
C1 = [1 0];
D1 = 0;

% Disturbance matrix
E1 = [0; 0.1];

% Build the state-space model
A = A1;
B = B1;
C = C1;
E = E1;
[A_new, B_new] = RLC_ss_V2(M, N, A1, B1);

rows_A = size(A, 1);
cols_A = size(A, 2);
rows_A_new = size(A_new, 1);
cols_A_new = size(A_new, 2);

% Update A with consistent dimensions
A = A_new

```

```
% Update B with zero padding to match dimensions
B = B_new

% Expand E with consistent zero padding
E = [inv(M) * [zeros(size(M, 1) - 1, 1); 0.1]; zeros(size(A, 1) -
    size(M, 1), 1)];

% Update C with consistent dimensions
C = [zeros(1,n-1),1,zeros(1,n)]

D = 0; % Direct feedthrough remains zero


%% Discretize system
Ts = 0.005;
sys_cont = ss(A, B, C, D);
sys_cont_dist = ss(A, E, C, D);
sys_disc = c2d(sys_cont, Ts, 'zoh');
sys_disc_dist = c2d(sys_cont_dist, Ts, 'zoh');
[Ad, Bd, Cd, Dd] = ssdata(sys_disc);
Ed = sys_disc_dist.B;

%% Simulation setup
Tsim = 20;
x0 = zeros(size(A, 1), 1);
highResTime = 0:Ts:Tsim;
highResRef = zeros(size(highResTime));
for k = 1:length(highResTime)
    if highResTime(k) <= 4
        highResRef(k) = sin(2*pi*1*highResTime(k));
    elseif highResTime(k) > 5 && highResTime(k) <= 10
        highResRef(k) = 0;
    else
        highResRef(k) = 1;
    end
end
w = 0.5 * randn(length(highResTime), 1);

%% MPC setup
predictionHorizon = 5;
```

```
controlHorizon = 2;
variables = {'ManipulatedVariables', 'ManipulatedVariablesRate', '
    OutputVariables'};
y_all = zeros(length(highResTime), 3, 4);

%% Loop through weights
for varIdx = 1:3
    for wIdx = 1:4
        weights = defaultWeights;
        if varIdx == 1
            weights(1) = manipulatedVars(wIdx);
        elseif varIdx == 2
            weights(2) = manipulatedVarsRate(wIdx);
        elseif varIdx == 3
            weights(3) = outputVars(wIdx);
        end

        MV = struct('Min', -30, 'Max', 30);
        Weights = struct('ManipulatedVariables', weights(1), '
            ManipulatedVariablesRate', weights(2), 'OutputVariables',
            weights(3));
        mpcController = mpc(sys_disc, Ts, predictionHorizon,
            controlHorizon, Weights, MV);

        y = zeros(length(highResTime), 1);
        x = x0;
        mpcState = mpcstate(mpcController);

        for k = 1:length(highResTime)
            u = mpcmove(mpcController, mpcState, Cd * x, highResRef(k
                ), []);
            x = Ad * x + Bd * u + Ed * w(k);
            y(k) = Cd * x;
        end
        y_all(:, varIdx, wIdx) = y;
    end
end

%% Plot results
figure;
```

```
118 for varIdx = 1:3
119     subplot(3,1,varIdx);
120     for wIdx = 1:4
121         plot(highResTime, y_all(:, varIdx, wIdx), 'LineWidth', 1);
                hold on;
122     end
123     plot(highResTime, highResRef, '--', 'LineWidth', 0.5);
124     grid on;
125     title(['Output vs Reference - Changing ', variables{varIdx}]);
126     xlabel('Time (s)'); ylabel('Output');
127     legend({['(' num2str(displayMatrix(varIdx,1)) ')'], ['(' num2str(
            displayMatrix(varIdx,2)) ')'], ['(' num2str(displayMatrix(
            varIdx,3)) ')'], ['(' num2str(displayMatrix(varIdx,4)) ')'], '
            Ref.'});
128 end
```

Listing 18: MATLAB Code finding the correct weights of the MPC

C++-Code

```cpp
#include "mex.h"
#include <vector>
#include <iostream>
#include <cmath>
#include <limits>
#include <nlopt.hpp>

typedef std::vector<double> State; // typedef for State (x) datatype

// Define global variables for system matrices
std::vector<std::vector<double>> A;
std::vector<std::vector<double>> Bd;
std::vector<double> Cd;
double Dd;

// Simulate one step of the system
double simulate_step(State &x, double u, const std::vector<std::
    vector<double>> &A, const std::vector<std::vector<double>> &Bd,
    const std::vector<double> &Cd, double Dd) {
     State x_next(x.size(), 0.0);  // Initialize the next state vector

     // Compute x_next = Ad * x + Bd * u
     for (size_t i = 0; i < A.size(); ++i) {

         for (size_t j = 0; j < A[i].size(); ++j) {
             x_next[i] += A[i][j] * x[j];
         }
         x_next[i] += Bd[i][0] * u;  // Apply control input
     }

     // Compute output y = Cd * x_next + Dd * u
     double y = 0.0;
     for (size_t i = 0; i < Cd.size(); ++i) {
         y += Cd[i] * x_next[i];
     }
     y += Dd * u;

     // Update the state
```

```
37      x = x_next;
38
39          return y;
40  }
41
42  // Cost function for MPC
43  double mpc_cost(const std::vector<double> &u, std::vector<double> &
        grad, void *data) {
44      struct MPCData {
45          State x0;            // Initial state
46          std::vector<double> r;  // Reference trajectory
47          int cHorizon;       // Control horizon
48          int pHorizon;       // Prediction horizon
49          double lastInput;    // Last control input
50          double lastOutput;  // Last system output
51      } *mpc_data = (MPCData *)data;
52
53      State x = mpc_data->x0; // Initialize state with last given
            system state
54      double cost = 0.0;  // Initialize cost variable
55      double y = mpc_data->lastOutput;    // Initialize system output
            with last measured system output
56      double delta_u = 0;                 // Initialize control effort
            variable
57      double Cd_Bd = 0.0;                 // Used for C*B
            multiplication
58      double lambda_e = 5.0; // Error weight
59      double lambda_ce = 0.01; // Control effort weight
60      double lambda_ci = 0.0001; // Control input weight
61      double error = 0.0;          // Initialize Error variable
62
63      //Initialize gradient vector with zeros
64      if (!grad.empty()) {
65          std::fill(grad.begin(), grad.end(), 0.0);
66      }
67
68      //Define gradient size
69      if (grad.size() < mpc_data->cHorizon) {
70      grad.resize(mpc_data->cHorizon, 0.0);
71      }
```

```
72
73      //Print Input Vector under test
74      std::cout << "Testing Input Vector: ";
75      for (int k=0; k <= mpc_data->cHorizon-1; k++){
76          std::cout << u[k] << " ";
77      }
78      std::cout << "\n";
79
80      // Perform multiplication C*B --> results in scalar
81      for (size_t i = 0; i < Bd.size(); ++i) {
82
83                      Cd_Bd += Cd[i]*Bd[i][0];
84                  }
85
86
87      // Iterate over the control horizon (for each control input)
88      for (int k = 0; k < mpc_data->cHorizon; ++k) {
89
90              error = y - mpc_data->r[k];  // Reference trajectory is
                    offset by k
91              cost += lambda_e * error * error;  // Penalize tracking
                    error
92
93              //Gradient calculation for tracking error term
94              if (!grad.empty()) {
95                  if (k >= 0 && k < grad.size()) {
96                      grad[k] += 2 * lambda_e * Dd * error;
97                      if (k > 0){
98                          grad[k-1] += 2 * lambda_e * Cd_Bd * error;
99                      }
100                 }
101             }
102
103         // Penalize control effort (smooth control actions)
104         if (k > 0) {
105             delta_u = u[k] - u[k - 1];  // Control input change
106         }
107         if (k == 0) {
108             delta_u = u[k] - mpc_data->lastInput; // Use last given
                    input for k = 0
```

```
109         }
110         cost += lambda_ce * delta_u * delta_u;  // Penalize large
               changes in control input
111
112         // Gradient calculation for control effort term
113         if (!grad.empty()) {
114             grad[k] += lambda_ce * 2 * (k == mpc_data->cHorizon-1 ?
                   delta_u : (delta_u - u[k+1] + u[k]));
115         }
116
117         cost += lambda_ci * u[k]*u[k];  // Penalize large changes in
               control input
118
119         // Gradient calculation for control input term
120         if (!grad.empty()) {
121             grad[k] += 2 * lambda_ci * u[k];
122         }
123
124         // Simulate system step update
125         y = simulate_step(x, u[k], A, Bd, Cd, Dd);
126
127     }
128
129
130     // After the control horizon, hold the last control input
           constant and simulate the rest of the prediction horizon
131     for (int k = mpc_data->cHorizon; k < mpc_data->pHorizon; ++k) {
132
133         error = y - mpc_data->r[k];  // Reference trajectory is
               offset by k
134         cost += lambda_e * error * error;  // Penalize tracking error
135
136         if (!grad.empty()) {
137
138                 grad[mpc_data->cHorizon-1] += 2 * lambda_e * Dd *
                       error;
139                 if (mpc_data->cHorizon-2 >= 2) {
140                   grad[mpc_data->cHorizon-2] += 2 * lambda_e * Cd_Bd
                         * error;
141                 }
```

```
142             }
143
144         cost += lambda_ci * u[mpc_data->cHorizon-1]*u[mpc_data->
                cHorizon-1];  // use last element of control input vector
                --> held constant
145         if (!grad.empty()) {
146             grad[mpc_data->cHorizon-1] += 2 * lambda_ci * u[mpc_data
                    ->cHorizon-1];
147         }
148
149         y = simulate_step(x, u[mpc_data->cHorizon - 1], A, Bd, Cd, Dd
                );
150
151     }
152
153     return cost;
154 }
155
156 double customMPC(const std::vector<std::vector<double>> &Ad,
157                  const std::vector<std::vector<double>> &Bd_in,
158                  const std::vector<double> &Cd_in,
159                  double Dd_in,
160                  int predictionHorizon,
161                  int controlHorizon,
162                  double currentInput,
163                  const State &currentState,
164                  const std::vector<double> &reference,
165                  double currentOutput) {
166
167     A = Ad;
168     Bd = Bd_in;
169     Cd = Cd_in;
170     Dd = Dd_in;
171
172     State x0 = currentState;    // initial system state is current
            system state
173     std::vector<double> u_init(controlHorizon, currentInput);   //
            soft-start optimizer with current control input
174
175     struct MPCData {
```

```
176         State x0;
177         std::vector<double> r;
178         int cHorizon;
179         int pHorizon;
180         double cInput;
181         double cOutput;
182     } mpc_data = {x0, reference, controlHorizon, predictionHorizon,
        currentInput, currentOutput};
183
184     // Setup the optimization problem
185     nlopt::opt opt(nlopt::LD_MMA, controlHorizon); //algorithm &
        dimensionality
186     opt.set_min_objective(mpc_cost, &mpc_data); // cost function &
        additional data
187     opt.set_lower_bounds(-2.0); // lower constraints on control input
188     opt.set_upper_bounds(2.0);  // upper constraints on control input
189     opt.set_xtol_rel(1e-10);    // stopping criterion for changes in
        control vector per optimization step
190     opt.set_ftol_rel(1e-10);    // stopping criterion for changes in
        result per optimization step
191     opt.set_maxeval(1000);      // stopping criterion for maximum
        number of evaluations
192
193     double min_cost;
194     try {
195
196         opt.optimize(u_init, min_cost); // start optimization
197
198         std::cout << "Optimization completed. Final input: "; //
            Print found optimal control input vector
199         for (double u : u_init) {
200             std::cout << u << " ";
201         }
202         std::cout << "\nFinal cost: " << min_cost << std::endl; //
            Print final cost
203
204     } catch (std::exception &e) {
205         std::cerr << "Optimization failed: " << e.what() << std::endl
            ;
206     }
```

```
207     return u_init[0]; // Return the first control input
208 }
209
210 // Helper function to convert MATLAB matrix to C++ vector
211 std::vector<std::vector<double>> matlabMatrixToVector(const mxArray *
        mat) {
212     size_t rows = mxGetM(mat);
213     size_t cols = mxGetN(mat);
214     double *data = mxGetPr(mat);
215
216     std::vector<std::vector<double>> result(rows, std::vector<double
            >(cols));
217     for (size_t i = 0; i < rows; ++i) {
218         for (size_t j = 0; j < cols; ++j) {
219             result[i][j] = data[i + rows * j];
220         }
221     }
222
223     return result;
224 }
225
226 // Entry point for the MEX function
227 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *
        prhs[]) {
228     if (nrhs != 10) {
229         mexErrMsgIdAndTxt("customMPC:invalidNumInputs", "Ten inputs
                required.");
230     }
231
232     // Parse inputs
233     std::vector<std::vector<double>> Ad = matlabMatrixToVector(prhs
            [0]);
234     std::vector<std::vector<double>> Bd = matlabMatrixToVector(prhs
            [1]);
235     std::vector<double> Cd(mxGetN(prhs[2]));
236     double *CdData = mxGetPr(prhs[2]);
237     for (size_t i = 0; i < Cd.size(); ++i) {
238         Cd[i] = CdData[i];
239     }
240
```

```
241     double Dd = mxGetScalar(prhs[3]);
242     int predictionHorizon = static_cast<int>(mxGetScalar(prhs[4]));
243     int controlHorizon = static_cast<int>(mxGetScalar(prhs[5]));
244     double currentInput = mxGetScalar(prhs[6]);
245 
246     State currentState(mxGetM(prhs[7]));
247     double *cStateData = mxGetPr(prhs[7]);
248     for (size_t i = 0; i < currentState.size(); ++i) {
249         currentState[i] = cStateData[i];
250     }
251 
252     std::vector<double> reference(mxGetM(prhs[8]));
253     double *rData = mxGetPr(prhs[8]);
254     for (size_t i = 0; i < reference.size(); ++i) {
255         reference[i] = rData[i];
256     }
257 
258     double currentOutput = mxGetScalar(prhs[9]);
259 
260     std::cout <<  "Referenz: ";
261     for (size_t i = 0; i < reference.size(); ++i) {
262         std::cout <<  reference[i];
263     }
264 
265     // Call the custom MPC function
266     double controlInput = customMPC(Ad, Bd, Cd, Dd, predictionHorizon
        , controlHorizon, currentInput, currentState, reference,
        currentOutput);
267 
268 
269     // Return the result to MATLAB
270     plhs[0] = mxCreateDoubleScalar(controlInput);
271 
272     std::cout << "Control input returned to MATLAB: " << controlInput
         << std::endl;
273     return;
274 }
```

Listing 19: C++ Code for MPC of a Linear System

References

[1] M. J. D. Powell, “A direct search optimization method that models the objective and constraint functions by linear interpolation,” *Advances in Optimization and Numerical Analysis*, pp. 51–67, 1994.

[2] K. Svanberg, “A class of globally convergent optimization methods based on conservative convex separable approximations,” *SIAM J. Optim*, pp. 555–573, 2002.

[3] Steven G. Johnson. “The NLopt nonlinear-optimization package.” besucht am 12.12.2024. (), [Online]. Available: `http://github.com/stevengj/nlopt`.

List of Figures

List of Tables

Listings