UNIVERSITY OF RIJEKA

FACULTY OF ENGINEERING

Graduate University Study of Electrical Engineering

Project report *Control of mechatronics systems*

DEVELOPMENT AND IMPLEMENTATION OF MODEL PREDICTIVE CONTROL ALGORITHM

Rijeka, February 2025.

Leonard Mikša
0069086808

Contents

1. Linear-quadratic regulator

Before introducing the concept of Model Predictive Control (MPC), we first examine the linear-quadratic problem, which is one of the most fundamental problems in control theory. A key solution to this problem is the linear-quadratic regulator (LQR), where the system is described by a set of linear differential equations and the cost function[1] is represented by a quadratic function.

1.1. Cost function

In order to get an intuitive perspective of a cost function, and the concept of 'optimal', let's take a look at one of the real-life examples and decisions we all face on a regular basis. For example, let's say we are planning a meal, and we have four options: cooking at home, ordering takeout, going to a restaurant, or buying a ready-made meal from the grocery store.

	Time	Money	Experience (1-5)
Cooking at home	60min	**10€**	2
Ordering a takeout	**30min**	20€	3
Going to a restaurant	45min	30€	**1**

Table 1.1. 'Cost' weighting when having a meal

From Table 1.1, we see that each option has its pros and cons. Cooking at home is time-consuming but inexpensive and offers a very good experience. Ordering takeout saves time, though it is more expensive, with moderate ratings for experience. Finally, dining at a restaurant requires the most time and money but provides the best experience.

In each column, the optimal choice is highlighted. We can see that the optimal choice depends on the resources we have: time, money, or desire on experience. If we are wealthy, eating at a restaurant would be the best solution for us. If we are a student, the restaurant is not an option, at least not on a regular basis, so we would mostly choose eating at home or at the college.

We want to mathematically assess which choice is optimal, so we set up a function that adds together **'our personal importances'** of time, money, and experience. We will denote them as A, B, and C, respectively. The cost function follows:

$$J = A \cdot \text{Time} + B \cdot \text{Money} + C \cdot \text{Experience}, \tag{1.1}$$

[1] A cost/loss function intuitively describes the 'cost' or 'loss' associated with a certain event occurring within a system.

where J represents the cost. Obviously, the cost function is heavily influenced by weighting parameters A, B, and C, which provide us with a degree of freedom we desire - for each individual they are different.

If time is the most important parameter for us, we set the weighting factor A higher than B and C. This way, we penalize options that take more time to realize. Let's say we are a student, thus money is the most important parameter for us. We will set A to be equal to 1, B to be equal to 5, and C to be equal to 2.

Student	**A**	Time	**B**	Money	**C**	Experience (1-5)	**J**
Cooking at home	**1**	60min	**5**	10€	**2**	2	**114**
Ordering a takeout	**1**	30min	**5**	20€	**2**	3	**136**
Going to a restaurant	**1**	45min	**5**	30€	**2**	1	**197**

Table 1.2. 'Cost' weighting when having a meal for a student

Of course, the optimal solution is the one that has the lowest J. In our case, that is cooking at home. If we were a wealthy individual, we would set A to be moderate, B to be the lowest, and C to be the highest, for example. That way, we would 'penalize' the outcomes with low experience rating, regardless of their expensiveness.

CEO	**A**	Time	**B**	Money	**C**	Experience (1-5)	**J**
Cooking at home	**1**	60min	**0.1**	10€	**10**	2	**81**
Ordering a takeout	**1**	30min	**0.1**	20€	**10**	3	**62**
Going to a restaurant	**1**	45min	**0.1**	30€	**10**	1	**58**

Table 1.3. 'Cost' weighting when having a meal for a wealthy individual

From these two cases, we see that the optimal outcome differs based on the personal preferences. The student is most likely to eat at home, while the CEO is most likely to go to a restaurant. The important point is - **there is no universally optimal solution.**

This analogy served us well to intuitively understand the concept of a cost function. When designing a control system, we do a similar reasoning when balancing between performance and energy consumption, for example. We set up a cost function as follows

$$J = \int_0^\infty \left(x^T \mathbf{Q} x + u^T \mathbf{R} u\right) dt. \tag{1.2}$$

Here, we are essentially looking at the area under the curve of a system response - the smaller the area, the better the performance. Since J represents that area, we are actually dealing with a minimization problem, which is why we are multiplying matrices by their transposed selves. If the

system response has an overshoot, those areas would otherwise cancel out, which is why we made the cost function quadratic. Every quadratic function has relatively easily obtainable minimum, which is illustrated on Figure 1.1.

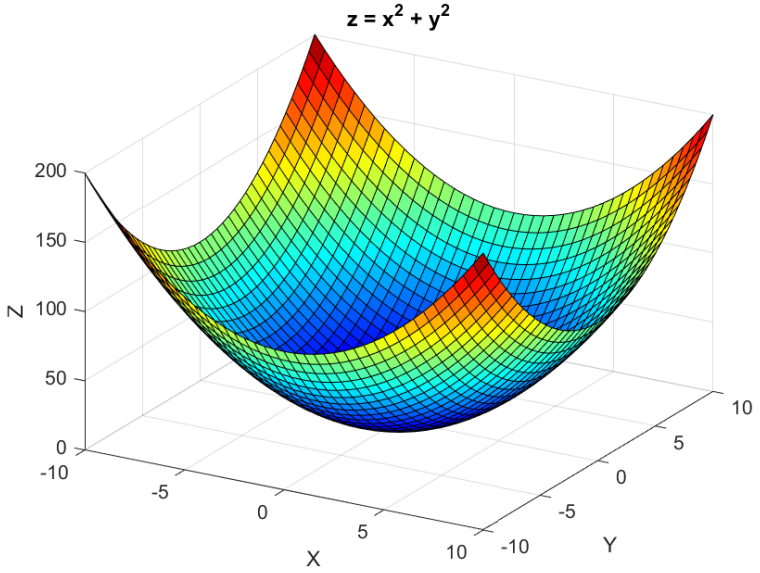

Figure 1.1. 3D plot of a basic quadratic function

By adjusting the matrices $\mathbf{Q}$ and $\mathbf{R}$, we penalize either performance or energy consumption according to our system's requirements, just like we had done in our earlier real-life example with parameters A, B, and C.

1.2. State-space representation of a system

A continuous, linear, and time-invariant system can be described in the state-space form, which requires two equations: state equation (1.3) and output equation (1.4). [1]

$$\dot{x} = Ax + Bu, \tag{1.3}$$

$$y = Cx + Du, \tag{1.4}$$

where x represents a state vector, $\dot{x}$ represents a time derivative of the state vector, y is an output vector, u is an input or control vector. A, B, C, and D are system matrix, input matrix, output matrix, and feedforward matrix, respectively.

Equation (1.3) says that the change of the state vector over time depends on a linear combination of the current state vector and input vector. With state-space representation, we avoid describing a system with a set of differential equations (possibly many of them, and possibly higher order). Instead, we represent a system with two relatively simple matrix equations which give us better insight into dynamics of each state variable.

For example, let's take a look at a separately excited DC machine, and try to describe it in state-space. We assume that the excitation is already formed and time-invariant.

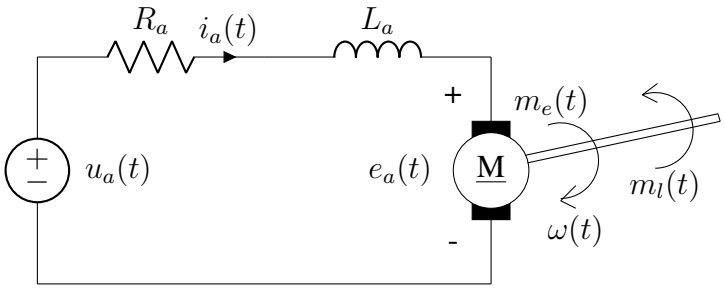

Figure 1.2. Electromechanical model of a separately excited DC electric machine

According to Kirchhoff's voltage law and Newton's second law of motion, we write following equations

$$u_a(t) = R_a i_a(t) + L_a \frac{di_a(t)}{dt} + e_a(t), \tag{1.5}$$

$$m_e(t) = J\frac{d\omega(t)}{dt} + b\omega(t) + m_l(t), \tag{1.6}$$

where $u_a(t)$ stands for armature voltage, $i_a(t)$ for armature current, $e_a(t)$ for back-electromotive force generated due to Faraday's law of electromagnetic induction, $\omega(t)$ for rotational speed of the machine, $m_e(t)$ for electromagnetic torque, and $m_l(t)$ for load torque. Parameters of the machine are R_a, L_a, J, b and c (armature resistance, armature inductance, inertia, friction coefficient, and machine constant, respectively).

Back EMF is proportionally dependent on rotational speed $e_a(t) = c\omega(t)$, while electromagnetic torque is proportionally dependent on the armature current $m_e(t) = ci_a(t)$.

We consider that the DC machine is running idle, so $m_l(t)$ equals zero. Equations (1.5) and (1.6) can be rewritten as

$$\frac{di_a(t)}{dt} = \frac{1}{L_a}u_a(t) - \frac{R_a}{L_a}i_a(t) - \frac{c}{L_a}\omega(t), \tag{1.7}$$

$$\frac{d\omega(t)}{dt} = \frac{c}{J}i_a(t) - \frac{b}{J}\omega(t). \tag{1.8}$$

Now, comparing (1.7) and (1.8) with (1.3) and (1.4), instead of a set of two differential equations we can describe this system in its state-space form. Having in mind that the input of this system is armature voltage, and its outputs are armature current and rotational speed, we can write down matrices A, B, C, and D. First, we can conclude that the state vector comprises of two state variables - armature current $i_a(t)$ and rotational speed $\omega(t)$. According to that, the state equation of a separately excited DC machine follows

$$\begin{bmatrix} \frac{di_a(t)}{dt} \\ \frac{d\omega(t)}{dt} \end{bmatrix} = \begin{bmatrix} -\frac{R_a}{L_a} & -\frac{c}{L_a} \\ \frac{c}{J} & -\frac{b}{J} \end{bmatrix} \begin{bmatrix} i_a(t) \\ \omega(t) \end{bmatrix} + \begin{bmatrix} \frac{1}{L_a} \\ 0 \end{bmatrix} u_a(t). \tag{1.9}$$

Since the feedforward matrix D is a null-matrix, the only thing left is to choose the output. Mostly, the output variable is arbitrary, but we choose such that we can easily observe and measure its value over time. The most convenient choice here would be rotational speed, for it can be measured by an encoder for example. The output equation follows

$$y = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i_a(t) \\ \omega(t) \end{bmatrix} \tag{1.10}$$

Comparing equations (1.10) and (1.9) with (1.3) and (1.4), we can write down state-space matrices denoted as A, B, C, and D as

$$A = \begin{bmatrix} -\frac{R_a}{L_a} & -\frac{c}{L_a} \\ \frac{c}{J} & -\frac{b}{J} \end{bmatrix}, \; B = \begin{bmatrix} \frac{1}{L_a} \\ 0 \end{bmatrix}, \; C = \begin{bmatrix} 0 & 1 \end{bmatrix}, \; D = 0$$

It is relatively simple to implement the LQR algorithm in MATLAB. First, we define the state-spaces matrices with predefined parameters of the system - a separately excited DC motor in this case.

Since we will be working on a linear pneumatic muscle later on, let's try and describe it in state-space as well. We will simplify its mathematical model for the sake of simplicity, since we are still introducing state-space as a concept.

Figure 1.3 shows an operating principle of a linear pneumatic muscle. Upon applying air pressure on the muscle, it contracts and thickens. One could intuitively assume that the muscle would expand under applied air pressure, but the contraction effect is achieved by its membrane, which is most often realized in a braided (e.g. McKibben muscle, sleeved bladder muscle), netted (e.g. Yarlott muscle), or embedded pattern (e.g. Morin muscle, Baldwin muscle), alongside many others [2].

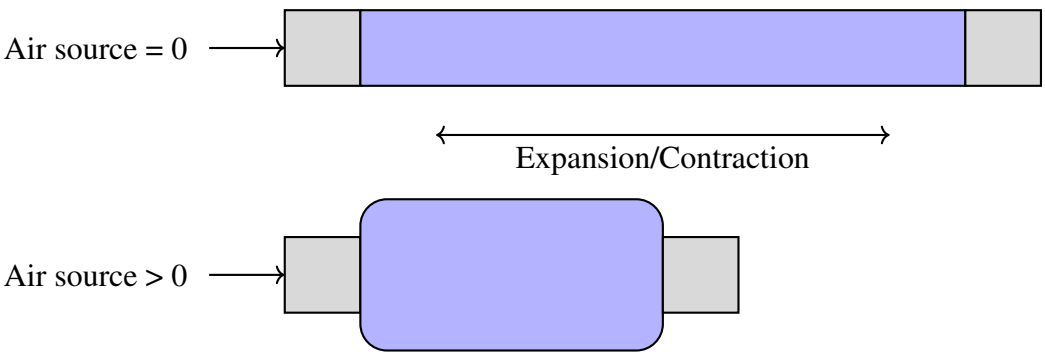

Figure 1.3. An illustration of a pneumatic muscle operating principle

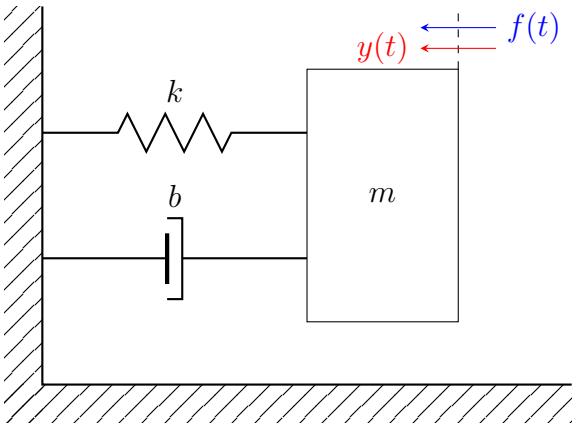

Figure 1.4. The mass-spring-damper model; a second-order system

When the air pressure is released, the muscle elongates, returning back to its original position. This operating principle can be described by a typical mass-spring-damper mechanical system, given some simplifications and omissions.

On Figure 1.4, $f(t)$ represents the force applied to the system, and we consider it as an input. Since our pneumatic muscle receives air pressure and contracts in response, we place the input $f(t)$ and the linear displacement $y(t)$ in the same direction. The more pressure that is applied, the greater the force, so we can link the force with the air pressure.

Following Newton's second law of motion, we write an equation

$$f(t) - m\ddot{y}(t) - b\dot{y}(t) - ky(t) = 0, \tag{1.11}$$

where b represents friction coefficient, and k stands for spring constant. Since this is a second-order differential equation with a single state $y(t)$, we must write it down as two first-order differential equations with two states instead. Therefore we must introduce two new variables which will serve as our state variables: $x_1(t)$ and $x_2(t)$. They're not exactly new, but rather redefined old ones.

$$x(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} y(t) \\ \dot{y}(t) \end{bmatrix} \tag{1.12}$$

The derivatives of our new state variables are

$$\dot{x}_1(t) = x_2(t) \tag{1.13}$$

$$\dot{x}_2(t) = \frac{1}{m} f(t) - \frac{b}{m} x_2(t) - \frac{k}{m} x_1(t) \tag{1.14}$$

Essentially, we split the initial variable in two dimensions, so we can write two first-order differential equations necessary for the state-space representation of the system.

Also, we need to describe valve dynamics which we approximate with a first-order differential equation

$$\tau\dot{q}(t) + q(t) = Ku(t), \tag{1.15}$$

where $q(t)$ represents the air flow rate, τ represents the time constant, K represents the gain, and $u(t)$ represents the voltage applied to the valve.

We can now write the final differential equation used to describe the system as follows

$$\dot{q}(t) = -\frac{1}{\tau}q(t) + \frac{K}{\tau}u(t) \tag{1.16}$$

We can now see that we have three state variables ($x_1(t)$, $x_2(t)$, and $q(t)$), and input vector comprised of two elements ($f(t)$ and $u(t)$). According to the equations (1.13), (1.14), and (1.16), the simplified state-space representation of a linear pneumatic muscle follows

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{q}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{k}{m} & -\frac{b}{m} & 0 \\ 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ q(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \\ \frac{K}{\tau} \end{bmatrix} \begin{bmatrix} 0 \\ f(t) \\ u(t) \end{bmatrix} \tag{1.17}$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ q(t) \end{bmatrix} \tag{1.18}$$

1.3. LQR design

The goal of the LQR design is essentially to solve the optimization problem defined by the equation (1.2), and rewritten here

$$J = \int_0^\infty \left(x^T Q x + u^T R u\right) dt. \tag{1.19}$$

Unlike the PID controller, LQR does not deal with derivative and integral components. Instead, it is focused solely on finding an optimal gain matrix $\boldsymbol{K}$ by minimizing the cost function.

Once $\boldsymbol{K}$ is obtained by solving the Ricatti equation[2][3], [4], [5]

$$A^T P + PA - PBR^{-1}B^T P + Q = 0, \tag{1.20}$$

and using

$$K = R^{-1}B^T P \tag{1.21}$$

the control law follows

$$u = -\boldsymbol{K}x \tag{1.22}$$

[2]Roughly said, Ricatti equation is a matrix equation with a quadratic term (for further details see [3])

Once we have a system defined in state-space, the LQR solution is easily obtainable by using the `lqr` command in MATLAB, whose inputs are matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{Q}$, and $\boldsymbol{R}$. The command returns the optimal gain matrix $\boldsymbol{K}$, matrix $\boldsymbol{P}$ which is a solution of the associated Ricatti equation, and the closed-loop eigenvalues (i.e. poles).

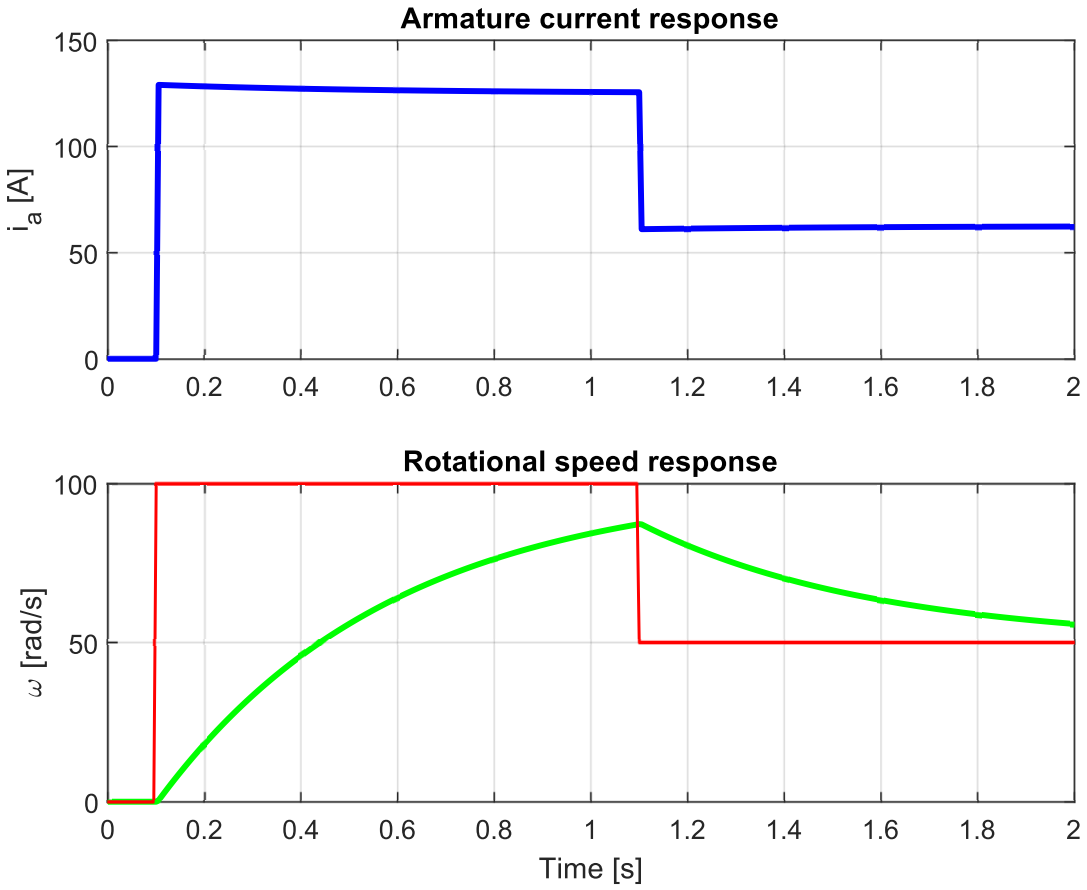

Figure 1.5. State variables responses with low Q_{22}

Figure 1.5 shows a case when we are forcing the aggressive control of the armature current of a previously described DC motor. The $\boldsymbol{Q}$ matrix has a form

$$Q = \begin{bmatrix} Q_{11} & 0 \\ 0 & Q_{22} \end{bmatrix}, \tag{1.23}$$

where we manually adjust elements Q_{11} and Q_{22} to achieve desired performances. Thus, we can achieve an aggressive control of the armature current by setting Q_{11} to be higher than Q_{22}. Figure 1.5 shows a case for a following $\boldsymbol{Q}$ matrix

$$Q = \begin{bmatrix} 10000 & 0 \\ 0 & 1000 \end{bmatrix} \tag{1.24}$$

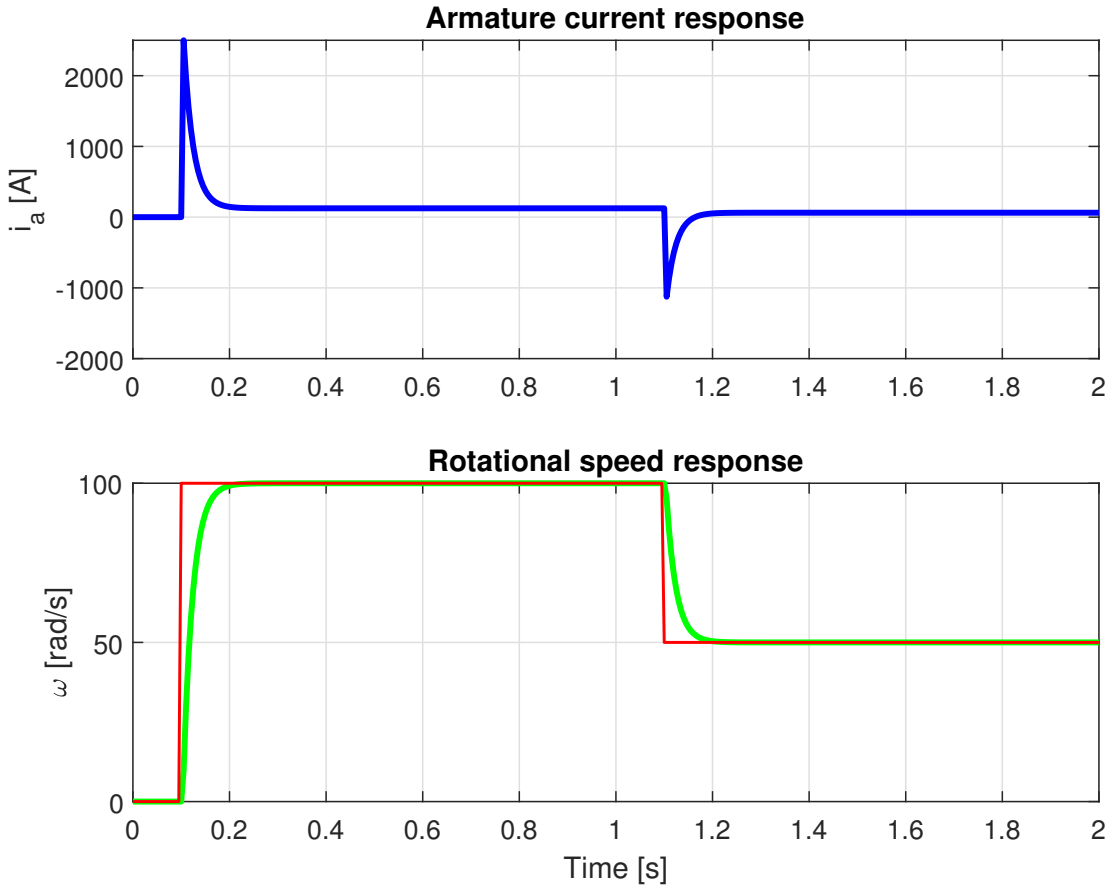

Figure 1.6. State variables responses with high Q_{22}

Figure 1.5 shows a case when we are forcing the aggressive control of the rotational speed. We can achieve that by setting Q_{22} to be higher than Q_{11}. Figure 1.5 shows a case for a following $\boldsymbol{Q}$ matrix

$$Q = \begin{bmatrix} 10 & 0 \\ 0 & 10000 \end{bmatrix} \tag{1.25}$$

We can see that with LQR we can go from one extreme to another - from an extremely precise current control at the expense of rotational speed to an extremely precise control of the rotational speed at the expense of armature current.

Of course, such extreme inrush currents are not acceptable in reality. Therefore, it is a task for an engineer to balance between the desired performance and either the energy consumption or in extension physical limitations of a system.

2. Model Predictive Control

Model Predictive Control (onward referred to as MPC) is an advanced control technique especially suitable for *Multiple-Input-Multiple-Output* (MIMO) systems. While some conventional control algorithms such as PID and previously described LQR are the most popular and widespread, MPC has significant advantages in applications that require real-time optimization and interaction with the environment. Additionally, in comparison to the LQR, MPC is able to handle constraints which will be described in detail in the following sections. This chapter is based on [4] and [5].

2.1. Principles and characteristics of MPC

Since MPC is in practice implemented on digital signal processors, the inevitable first parameter is **sampling time** T_s. There are many techniques and methods to choose the optimal sample time - one neither too big since it wouldn't be able to capture system dynamics and even would reduce stability margin, nor too small since it would require excessive computational power in that case.

We will not deal with details for choosing the optimal sampling time since the topic is described and studied in detail, and taught on every course on digital signal processing. The two perhaps most widespread methods are Nyquist criterion and the open-loop system response where we choose the sampling time such that between 10 and 20 samples fit into the rise time of the open-loop response.

The most intuitive and easiest example to comprehend the properties and features of MPC algorithm is an autonomous driving system. In this section, we will describe many of them on this example.

2.1.1. MPC Controller

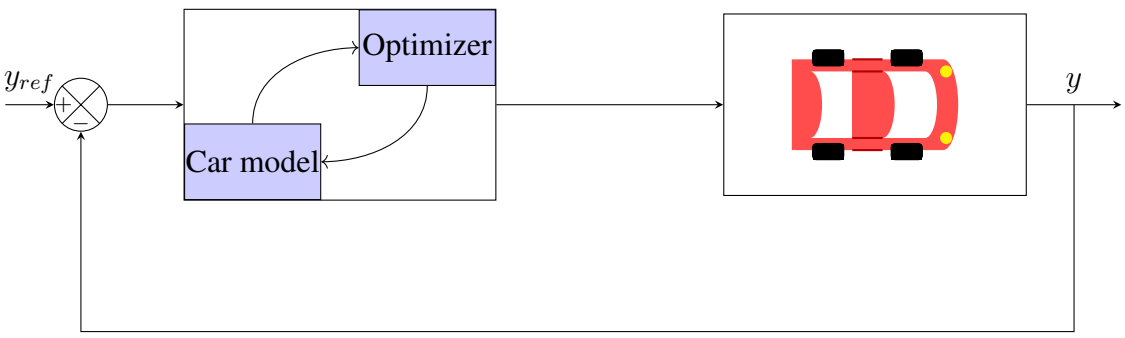

Figure 2.1. A principle scheme of a MPC-driven car

As seen in Figure 2.1, the MPC controller uses the plant model - in this case the car model - to make predictions about the behavior of the future plant output y; that is y_{k+1}, y_{k+2}, ..., y_{k+p}. This tells us that MPC is a predictive control algorithm. The optimizer takes care of the performance at the given moment.

Note that both the output y and the reference y_{ref} can be multiple signals (input/output vector). Most often when using MPC, that is the case - we are dealing with MIMO systems. If we used typical PID control, or even the previously described LQR [3], we would have hard time analyzing and modeling how each of the inputs affects each of the outputs[1].

As said, the reason why MPC controller uses the plant model is to predict its future behavior. A measure of how far MPC predicts the future behavior is called **prediction horizon**, denoted as p.

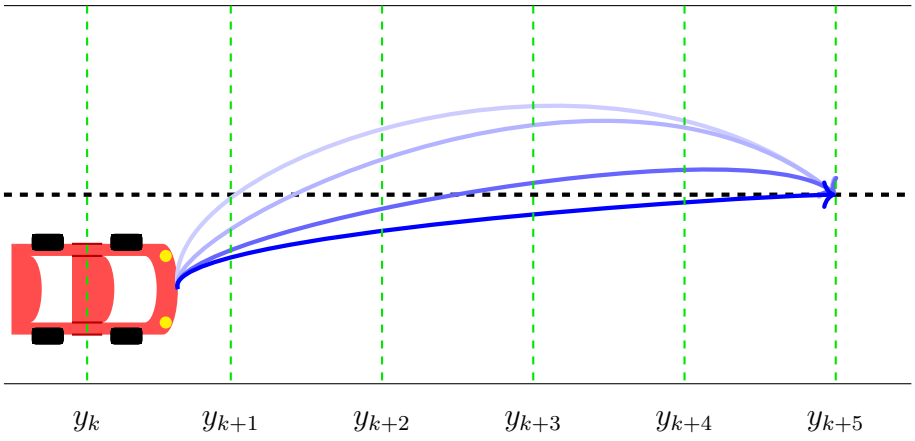

Figure 2.2. Illustration of the prediction horizon on a road lane, case $p = 5$

Figure 2.2 shows an illustration of a single road lane and a car driving on it. We assume that all possible state variables, such as for example speed, are constant and only the steering wheel angle is being adjusted to control the position of the car. The dashed horizontal line is a reference set in the middle of the lane.

At the current time step k, the MPC controller simulates the car's trajectory for the next p - in this case five - time steps if the steering wheel is turned at, let's say, 5°. Then, it computes multiple future scenarios with different steering angles in order to find the predicted path closest to the reference. This is not being done randomly, but in a systematic way. The optimizer within the MPC controller takes care of solving the optimization problem by minimizing the error between the current position of the car and the reference position. This optimization problem is similar to the one we had dealt with in LQR, which means that it takes into account some other factors too, such as smooth transitions of the steering wheel in order to avoid causing lurches. Clearly, a part

[1]Decoupling signals in mathematical model of an induction machine, for example

of the solution to the MPC design will also include a cost function similar to the one defined with (1.2).

In the next time step $k + 1$, the MPC controller repeats the whole process, virtually discarding the previous optimizations in order to find the new one, possibly slightly different from the previous one. This happens because of the external disturbances on the car such as gusting wind or potholes, or maybe even constraints detected such as approaching a pedestrian crossing, a semaphore, or entering an area with a different speed limit.

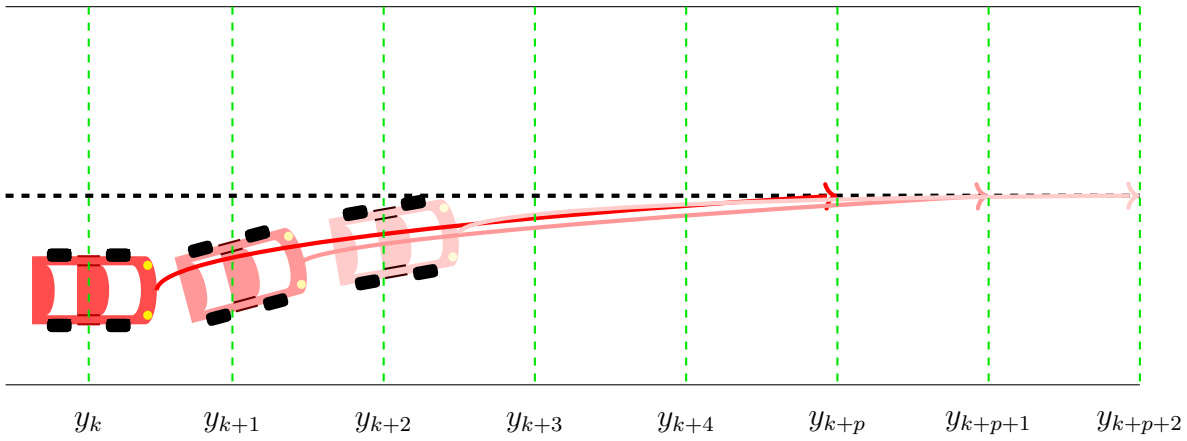

Figure 2.3. Illustration of the iterative nature of the prediction horizon on a road lane

Figure 2.3 shows an illustration of MPC's dynamic nature, which is why MPC is sometimes referred to as *receding horizon control*.

It is a task for an engineer to find an appropriate balance between the system requirements and appliance on one side, and the computational load on the microcontroller on the other side. Of course, if the MCU is yet to be bought, its price is another factor in this consideration. There are some recommendations on how to choose the prediction horizon, such as the one saying that the chosen p shall fit 20 to 30 samples in the transient response of the open-loop system [5].

In summary, the prediction horizon must cover the system dynamics much like the sample time while being conservative in spending too much computational power. Back to our autonomous car example, too small prediction horizon might not be able to stop the car on time if it approaches a red traffic light. On contrary, too big prediction horizon would make the MPC controller throw away much planning made if some unexpected disturbance emerges such as road reconstruction, an accident in front of the car, a sudden rainstorm, etc.

Another design parameter is the **control horizon**. As we can assume by its name, it holds a similar meaning to the prediction horizon, but it deals with control actions rather than output prediction.

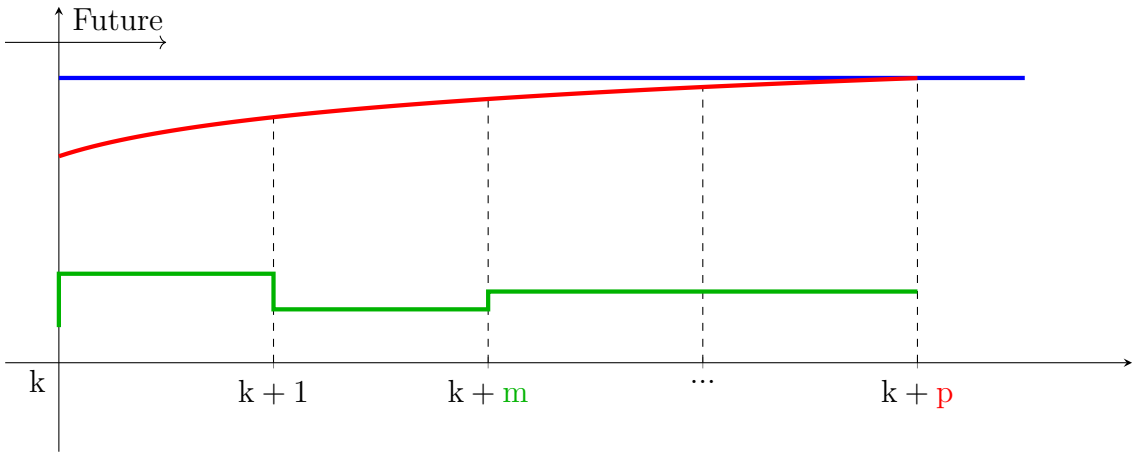

Figure 2.4. Illustration of the control horizon compared to the prediction horizon

For the computed predictions of system outputs represented by a red line on Figure 2.4, the next m future control signals tend to bring the system into the desired optimal state. If $m \neq p$, the rest of the control moves are held constant. The trade-off is similar to the prediction horizon case - too small control horizon causes a lack of precision and maneuvering while too big control horizon increases the complexity, and leads to an excessive waste of computations.

Besides, if the reference represented by a blue line on Figure 2.4 changes, only the first few control steps hold a significant effect on the actuation.

The recommendation for choosing m is to be 10% to 20% of the prediction horizon, without going below $m = 2$.

Another important feature of MPC are **constraints**. They can be incorporated on the inputs, the rate of change of inputs, and on the outputs. Constraints can be either soft or hard. Hard constraints cannot be violated, whereas soft constraints can be violated.

An example of a hard constraint would be a physical limit to the gas pedal and the constraint on lateral position of the car, since the car always must stay within the road lane. An example of a soft constraint could be a speed limit or lane centering.

Choosing appropriate constraints shall be taken cautiously because some of them might conflict with each other, leading to an unfeasible solution of the optimization problem. For example, if we set a hard constraint on both minimum speed limit on a highway and obstacle avoidance, in situations like heavy thunderstorm or a snowstorm the car will have to slow down below the minimum speed limit in order to ensure safety. Such conflicts can be avoided by setting the minimum

speed limit as a soft constraint - in that case, the car would slow down below the limit, but only until it is by any means safe to return into the margins of speed limits.

2.2. Adaptive MPC and Gain-scheduled MPC

All of our considerations so far are applicable on linear time-invariant systems with linear constraints and a quadratic cost function. These properties ensure convexity of the optimization problem since the cost function has a single global minimum, as shown on Figure 1.1.

In most practical cases, the plant is nonlinear, at least not in the whole operating range. In those cases, we can either linearize the plant around one or more operating points, or linearize the plant on the whole operating range using abstract and advanced methods like Koopman operators.

Adaptive MPC provides us capability to linearize the plant model 'on the fly' depending on the current operating point.

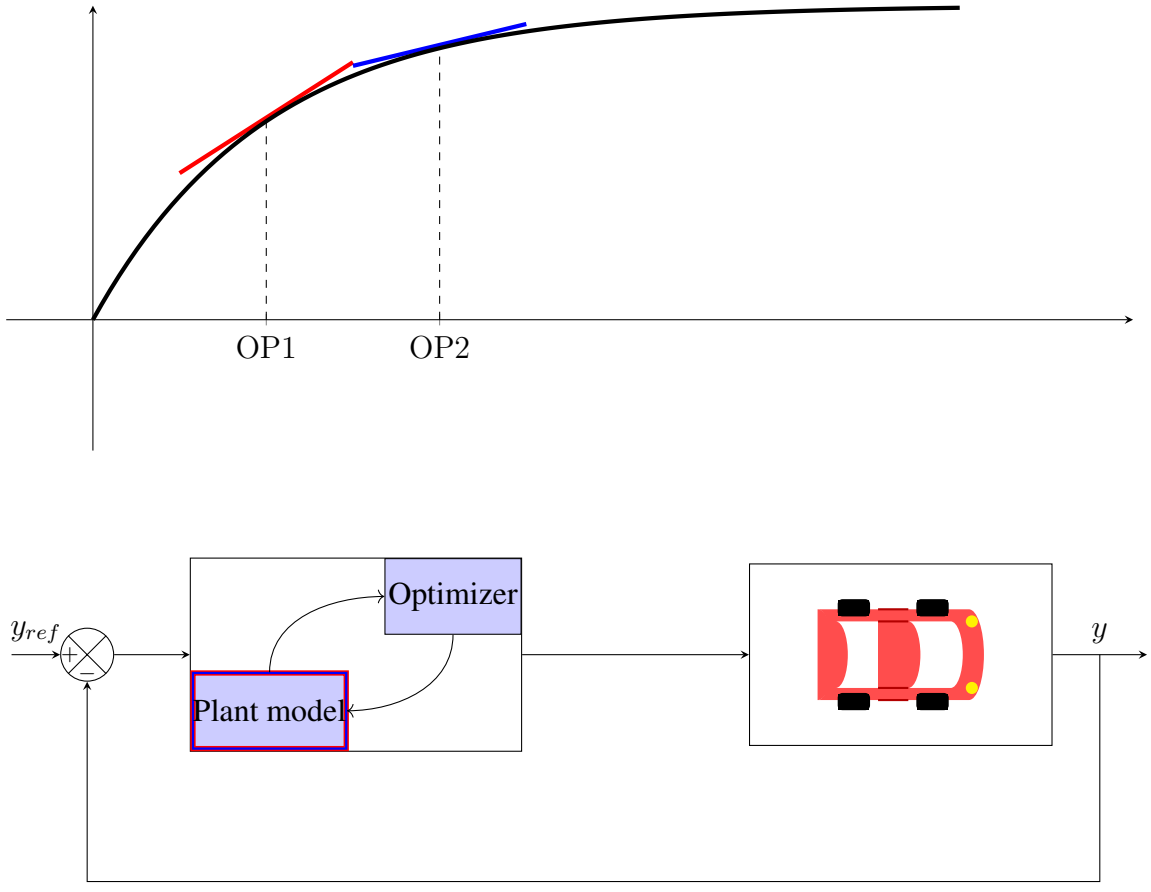

Figure 2.5. A principle scheme of adaptive MPC-driven car

The internal plant model is updated in each time step with its linearized form while the optimization problem remains the same across all operating points. In other words, the number of states, and number of constraints do not change for different operating points.

If they do change, we use **gain-scheduled MPC**. In this variation of MPC algorithm, the linearization is done offline for each operating point. Then, an independent MPC controller is designed for each of the linearized plant models which means that each of the controllers can have different number of states and constraints.

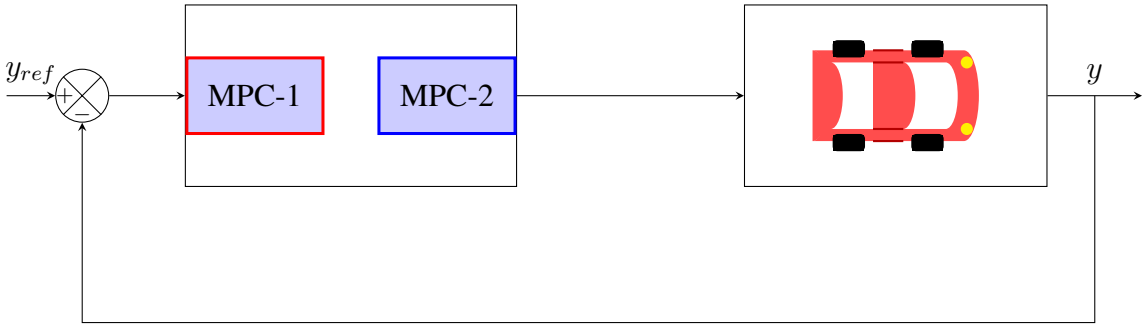

Figure 2.6. A principle scheme of gain-scheduled MPC-driven car

In this case, we also need to have an algorithm that will switch between the predefined MPC controllers for different operating points. The obvious disadvantage of this approach is doubled (in this case) memory consumption in comparison to the adaptive MPC.

2.3. Nonlinear MPC

In cases where we cannot linearize the nonlinear plant model with satisfying precision, and we additionally have nonlinear constraints and nonlinear cost function, we face a non-convex optimization problem.

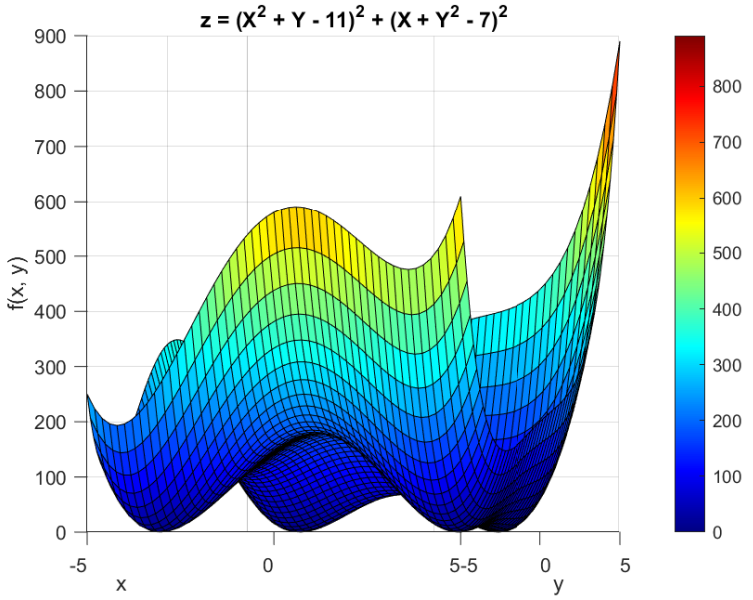

Figure 2.7. 3D plot of a function with multiple local optima

In this case, we may face some challenges regarding the efficiency of the algorithm execution. As shown on Figure 2.7, the cost function may have multiple local optima, and finding the global one might be extremely challenging, and in some cases even impossible.

Nonlinear Model Predictive Control (NMPC) is an advanced control strategy that extends MPC to systems with nonlinear dynamics. Unlike linear MPC, which assumes linear system models, NMPC is capable of handling the complexities associated with nonlinear behavior in system model, constraints, and/or the cost function.

2.3.1. Koopman operator

The Koopman operator is a representation of a dynamical system in terms of the evolution of observables on a function space [6], [7]. Basically, the idea is to lift/embed the nonlinear dynamics into a higher dimensional space where its evolution is approximately linear.

This is done by using a class of linear predictors for nonlinear dynamical systems. Essentially, linear predictor is sort of an artificial dynamical system that is able to predict the future state based on the measurement of the current state and future inputs.

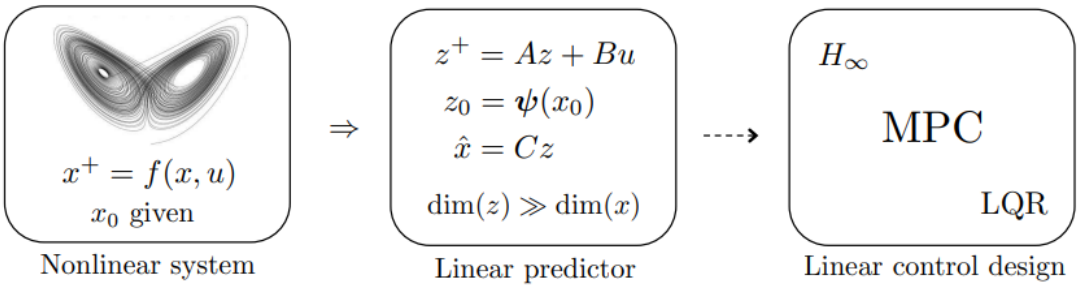

Figure 2.8. Linear predictor for a nonlinear controlled dynamical system [7]

As seen in Figure 2.8, the idea is to use a class of linear predictors as a mid-step towards the linear control design; z represents the lifted state evolving on a higher-dimensional state space, $\hat{x}$ is the prediction of the true state x and ψ is a nonlinear lifting mapping.

Figure 2.9 shows the results of the implemented NMPC obtained by using Koopman operator [7]. Subfigures on the left side show the results regarding the output y, while the subfigures on the right side show the results regarding the control input u.

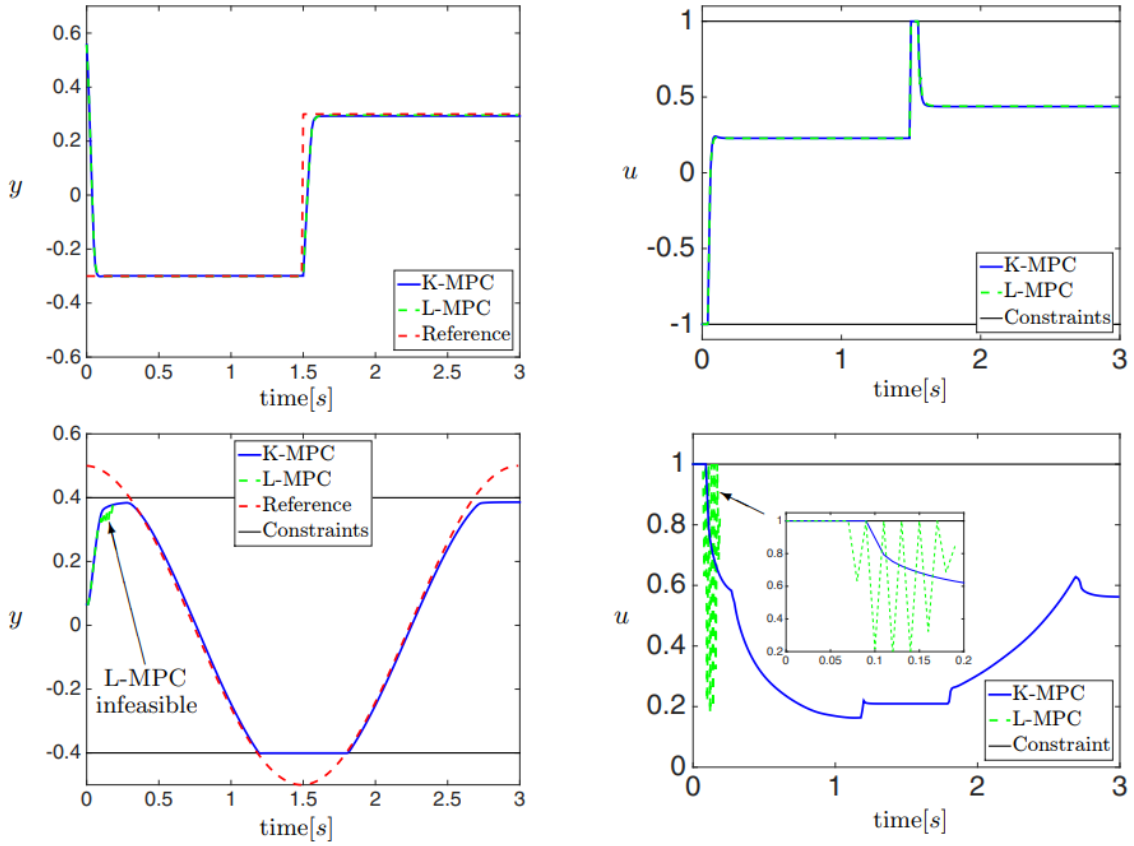

Figure 2.9. Linear predictor for a nonlinear controlled dynamical system [7]

2.4. Applications, advantages and limitations

The superiority of MPC over classical PID control is most visible on its capability to handle MIMO systems, constraints included, relatively easily. With MPC, the constraints can be incorporated on both input and outputs, whereas with the PID controller there would be much more complexity with implementation of the constraints, especially while simultaneously ensuring optimal control.

Furthermore, the main advantage of MPC over LQR are constraints. This makes MPC most suitable for multivariable systems where constraints on inputs or states are critical. Historically, among the first applications of MPC was in process/chemical industry, in the 1980's. Further examples include oil refineries, autonomous vehicles, robotics, aerospace engineering, and other complex control systems.

On the other side, disadvantages of MPC are high computational demands, excessive complexity in solving online optimization problem in real-time, eventually leading to low efficiency and high prices (more expensive MCUs). As seen earlier, there are also many different parameters such as prediction horizon, control horizon, soft and hard constraints. Finding the right and optimal balance between performance and energy consumption and/or stability might be complex and

time-consuming. Formulating optimal constraints can also be very challenging.

In the following chapters, we will try to form a MATLAB/Python-driven script for implementation of MPC on a DC motor and linear pneumatic muscle. It is worth emphasizing that those systems are very simple and using MPC on them is, in most cases (except for some extremely critical appliances), an example of overengineering, since classical control techniques such as PID or LQR are more than satisfactory for those systems.

Nevertheless, we will try to implement MPC on them, as they will hopefully serve as a simple, yet to some degree appropriate example of basic MPC implementation.

3. Implementation of LQR and MPC in MATLAB and Python

The algorithms theoretically described in previous chapters are implemented in both MATLAB and Python environment. The code, step-by-step, will be explained and thoroughly overviewed in this chapter, followed by obtained results and comparison between Linear-Quadratic Regulation and Model Predictive Control. It is worth noting that this work serves only as a base introduction to MPC and its basic implementation rather than for achieving optimal control on a suitable complex system. Both DC motor and linear pneumatic muscle are relatively simple systems where PID control (or LQR at most) is satisfactory enough while the usage of Model Predictive Control is pretty much overengineering in those cases.

3.1. LQR in MATLAB

In the MATLAB simulation, the setpoint is turned off, and the natural response is observed with the initial condition of the rotational speed being 50 $\frac{\text{rad}}{\text{s}}$. Since the DC motor is asymptotically stable, this will cause the DC motor rotational speed converge towards zero state. Furthermore, the parameters of the motor are set arbitrary, but as some conventional values. Based on (1.9), the state-space model is formed in MATLAB.

```
clc;
clear all;
%Leonard Mikša - LQR controlled DC motor

%Initial conditions:
x0 = [0;                    %Armature current [A]
      50];                  %Rotational speed [rad/s]

%Parameters of the system:
Ra = 0.705;                     %Armature resistance [ohm]
La = 0.00905;                   %Armature inductance [H]
c = 3.9;                        %Motor constant [Vs]
J = 2;                          %Inertia [kgm^2]
b = 0.0963;                     %Friction coefficient [Nms/rad]


%State-spaces matrices:
A = [-Ra/La      -c/La;
      c/J         -b/J]

B = [1/La;
     -1/J]

```

```
C = [0      1]

D = [0]

%Regulation matrices:
Q = [10     0;
     0     10000]

R = 1
```

The LQR problem could have been solved with MATLAB's built-in solver for quadratic programming `quadprog`, but since the LQR is not the main focus of this work, we will use MATLAB's built in LQR solver which returns the optimal gain matrix **K**. Since the LQR does not eliminate the steady-state error, we shall also compute *pregain* in order to eliminate it.

```
%Built-in LQR solver:
Ksystem = lqr(A, B, Q, R)
system = ss((A-B*Ksystem), B, C, D);          %Closed-loop system
Kdc = dcgain(system)
Kr = 1/Kdc
system = ss((A-B*Ksystem), B*Kr, C, D);

%Simulation:
t = 0: 0.001: 0.2;                            %Time vector
[y, t, xlqr] = initial(system, x0, t);        %Natural response
```

The simulation is ran with the command `initial`, whose input arguments are the system model, initial conditions, and the time vector. It's solution is stored into arbitrary variables, in our case `y`, `t`, and `xlqr`. The results are already shown with Figure 1.5 and Figure 1.6.

3.2. MPC in MATLAB

3.2.1. Algorithm and input constraints

Since we are implementing discrete MPC, some additional pre-steps are needed, such as choosing the sampling time and discretizing the state-space model. This is done with MATLAB `c2d` command whose input arguments are matrices of the continuous state-space model, and the sample time. Additionally, we can choose which discretization method we will use (Tustin method, *Zero-Order Hold*, *First-Order Hold*, etc.).

Furthermore, we define the parameters of the MPC controller listed and described in the previous chapter. We set prediction horizon to 10 and control horizon to 3.

```
Ts = 0.004;              %Sampling time [s]

% Discretize the state-space model
```

```
4   [Ad, Bd] = c2d(A, B, Ts);
5
6   %%MPC Parameters
7   prediction_horizon = 10;
8   control_horizon = 3;
9   Q = [10  0;
10         0  10000];
11  R = 1;
12  p = prediction_horizon;
13  m = control_horizon;
14
15  %%Initial Conditions
16  x0 = [0; 50];           %Initial state of the system
17  x = x0;                 %Current state
18  u_prev = 0;             %Previous input
19
20  %% Simulation setup
21  sim_time = 0.2;                 %Total simulation time (seconds)
22  steps = sim_time / Ts;          %Number of steps
23  x_trajectory = zeros(2, steps); %Placeholder init. for plotting
24  u_trajectory = zeros(1, steps); %Placeholder init. for plotting
```

The optimization problem solved by discrete MPC minimizes a quadratic cost function

$$J = \sum_{i=1}^{p} \left(x^T Q x + u^T R u \right). \tag{3.1}$$

For the arbitrary **Q** and **R**, in each execution cycle MPC finds optimal control input u for the next p steps. In order to apply the state weighting matrix **Q** across all predicted steps p, we use MATLAB function `kron`, which returns the Kronecker tensor product[1] of the input arguments. In order to apply the state weighting **R** across all predicted steps m, we use the same function to obtain the **Rbar** matrix. Matrix **Qbar** is a 20-by-20 matrix, while **Rbar** is 3-by-3 identity matrix.

$$Q_{\text{bar}} = \begin{bmatrix} 10 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 10000 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 10 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 10000 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 10 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 10000 \end{bmatrix}$$

Furthermore, we diagonally concatenate **Qbar** and **Rbar** in order to obtain the quadratic cost matrix **H**, later on used in the quadratic optimization. Since we are optimizing control inputs u, **H**

[1] Not to be confused with concatenation; explanation in appendix at the end of the document.

will later on be reduced to 3-by-3 instead of 23-by-23, so this step isn't mandatory. However, if we wanted to optimize state trajectories explicitly, then it would be.

$$\mathbf{H} = \begin{bmatrix} \mathbf{Q}_{\text{bar}} & 0 \\ 0 & \mathbf{R}_{\text{bar}} \end{bmatrix}$$

```
%% Optimization setup
%Quadratic Cost Matrices
Q_bar = kron(eye(p), Q);                    %Block diagonal Q
R_bar = kron(eye(m), R);                    %Block diagonal R
H = blkdiag(Q_bar, R_bar);                  %Quadratic cost matrix

%Prediction matrices
Phi = zeros(2*p, 2);
Gamma = zeros(2*p, m);

%Filling the prediction matrices
for i = 1:p
  Phi(2*i - 1:2*i, :) = Ad^i;
  for j = 1:min(i, m)
    Gamma(2*i - 1:2*i, j) = Ad^(i - j) * Bd;
  end
end
```

The prediction matrices $\mathbf{\Phi}$ and $\mathbf{\Gamma}$ represent how the system's dynamics and control inputs affect the predicted states over prediction and control horizons, respectively ($\mathbf{\Phi}$ for system's dynamics and $\mathbf{\Gamma}$ for control inputs; could be intuitively linked with natural/forced responses).

For the next p steps, the future states can be expressed recursively

$$x_{k+1} = Ax_k + Bu_k$$

$$x_{k+2} = A^2x_k + ABu_k + Bu_{k+1}$$

$$x_{k+3} = A^3x_k + A^2Bu_k + ABu_{k+1} + Bu_{k+2}$$

$$\dots$$

From here, we can separate the component affected by control input u, and the other one linked with system dynamics. Thus, we can define the state prediction matrix $\mathbf{\Phi}$, essentially a vertical concatenation of the state matrix $\mathbf{A}$, and the control influence matrix $\mathbf{\Gamma}$.

$$\mathbf{\Phi} = \begin{bmatrix} A \\ A^2 \\ A^3 \\ \vdots \\ A^p \end{bmatrix}, \qquad \mathbf{\Gamma} = \begin{bmatrix} B & 0 & 0 & \cdots & 0 \\ AB & B & 0 & \cdots & 0 \\ A^2B & AB & B & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A^{p-1}B & A^{p-2}B & A^{p-3}B & \cdots & B \end{bmatrix}$$

Generally, we can see that the matrix $\mathbf{\Phi}$ has dimensions $(p \cdot n_x) \times n_x$, while the matrix $\mathbf{\Gamma}$ has dimensions $(p \cdot n_x) \times (m \cdot n_u)$. In our case, prediction horizon is set to 10, while the number of states is two and the number of inputs is one. Thus, matrix $\mathbf{\Phi}$ has dimensions 20×2 while $\mathbf{\Gamma}$ has dimensions 20×3.

The `for` loop is used to fill the empty placeholders for $\mathbf{\Phi}$ and $\mathbf{\Gamma}$ as explained above.

We can now move on to constraints and their incorporation. We define the upper and lower limit for the control input, and its maximum rate of change; for now, we will apply constraint only on the control input u.

```
%Constraints
U_max = 10000;                          %Maximum input
U_min = -10000;                         %Minimum input
delta_U_max = 2;                        %Maximum change in input
```

By setting the constraints so high, we have essentially turned them off. We first want to obtain the system responses similar to the LQR so we can validate the algorithm. Once we do, we will set the limits of the control input to lower values.

Since we will use MATLAB's built-in `quadprog` function for solving the optimization problem, we shall define its input arguments, and constraints are among them. The constraints are encoded as $b_l \leq u \leq b_u$, where b_u and b_l are upper and lower bounds for the inequality constraints (vectors of doubles).

Into each bound, we incorporate the maximum value and the maximum rate of change. We concatenate those two components into a vector, as following.

```
b_u = [U_max*ones(p, 1); delta_U_max*ones(m, 1)];
b_l = [U_min*ones(p, 1); -delta_U_max*ones(m, 1)];

predicted_states = zeros(2, p, steps);    %3D array to store predicted
  states
control_inputs = zeros(m, steps);         %Store control horizon inputs
```

The first part (regarding U_{min}/U_{max}) is linked to the prediction horizon while the second part (regarding $\pm\Delta U_{max}$) is linked to the control horizon. With this, we have obtained two 13×1 vectors with equal values but opposite signs (first 10 values are ± 10000, while the last three values are ± 2).

We initialize the placeholders for storage of the predicted states and control inputs - `predicted_states` with dimensions $n_x \times p \times n_{\text{steps}}$, and `control_inputs` with dimensions $m \times n_{\text{steps}}$. The 3D array stores p predicted values for both state variables (armature current and rotational speed) in each time step. The 2D array stores m predicted values for the control input.

We can now finally move on to the main MPC loop. In each time step, we update the predicted states by multiplying the current state vector with the prediction matrix $\mathbf{\Phi}$. The obtained result is a 20×1 matrix which we then reshape into a 3D array. The 3D array is now storing the predicted states over the whole prediction horizon in each time step. Thus, its dimensions are $2 \times p \times n_s$, where n_s is number of simulation steps; in our particular case, the dimensions are $2 \times 10 \times 50$.

We define the quadratic cost matrix $\mathbf{H}$, mentioned earlier, on a way similar to the one at LQR. We multiply the transposed $\mathbf{\Gamma}$ matrix with the matrix **Qbar** and then with the $\mathbf{\Gamma}$, and add **Rbar** to it. We use the control influence matrix $\mathbf{\Gamma}$ since we are optimizing the control inputs, and not the states directly.

This way, we have essentially *'transformed'* the inherent $\mathbf{Q}$ matrix's influence on the states, to the influence on the control inputs which we are optimizing. Since **Rbar** is inherently tied to the control inputs we don't multiply it with anything. The quadratic cost matrix is now

$$\mathbf{H} = \mathbf{\Gamma}^T \mathbf{Q}_{\text{bar}} \mathbf{\Gamma} + \mathbf{R}_{\text{bar}}$$

Since the `quadprog` function in this case solves a cost function formulated as

$$J = \frac{1}{2} u^T H u + f^T u,$$

originated from the so-called *Lagrange expression* [8], we shall now define the linear cost vector f. In theory, f is derived from

$$f = \mathbf{\Gamma}^T \mathbf{Q}_{\text{bar}} \left(\mathbf{\Phi} \mathbf{x_k} - \mathbf{x_{ref}} \right)$$

Since our setpoint is zero-state, we can omit $\mathbf{X_{ref}}$.

```
%% MPC Loop
for k = 1:steps

%Update state prediction
x_pred = Phi * x;                                   %Predicted states (20x1)
predicted_states(:, :, k) = reshape(x_pred, [2, p]);%Store predicted states
 (2x10x50)

%Define quadratic cost matrix H
H = Gamma' * Q_bar * Gamma + R_bar;                 %[m, m]

%Define linear cost vector f
f = (Gamma' * Q_bar * (Phi * x))';                  %[m, 1]

%Solve the quadratic programming problem
options = optimset('Display', 'off');
u_opt = quadprog(H, f, [], [], [], [], b_l, b_u, [], options);

control_inputs(:, k) = u_opt(1:m);          %Store control inputs over horizon
u = u_opt(1);                               %Extract the first control input
```

```
20  x = Ad * x + Bd * u;                    %Apply the control input
21
22  %Save trajectories
23  x_trajectory(:, k) = x;
24  u_trajectory(k) = u;
25
26  %Update previous input
27  u_prev = u;
28  end
```

Finally, we can call the `quadprog` function to minimize the defined cost function and obtain the optimal control inputs. We store them over the control horizon m and apply the first control input on the state equation. In the end, we save the trajectories into vector variables for plotting.

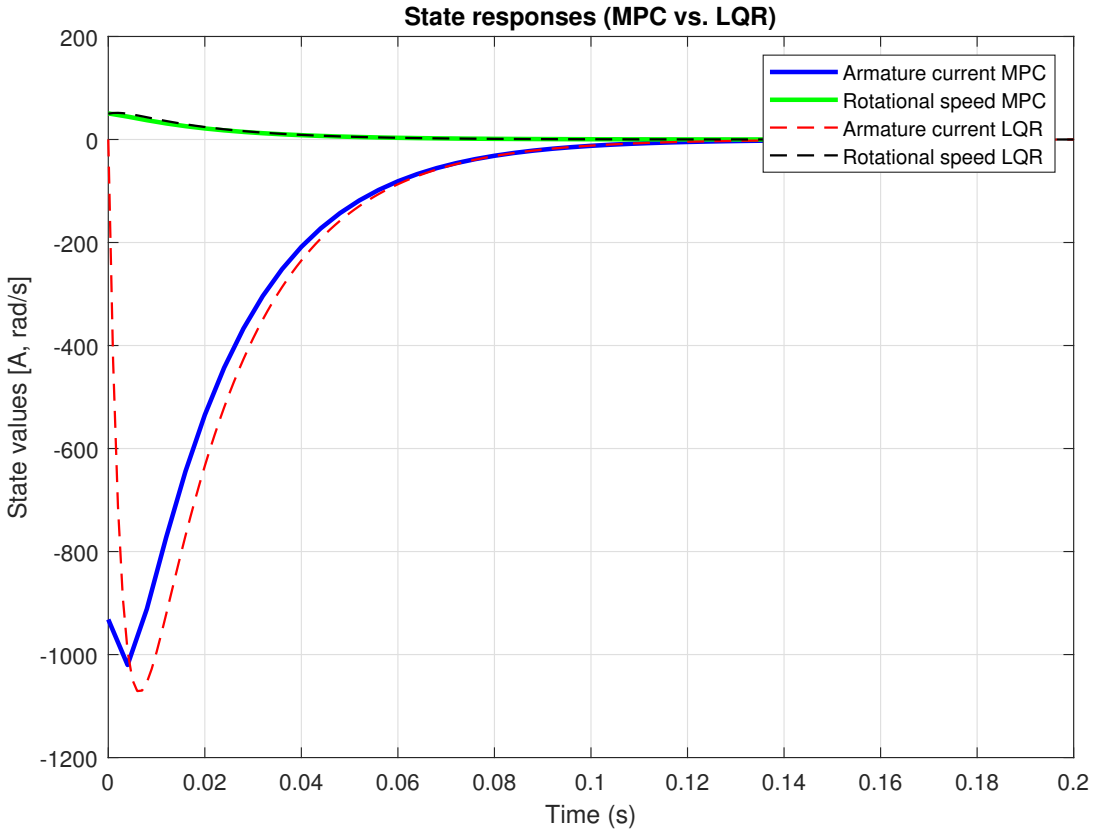

Figure 3.1. State responses comparison between LQR and MPC without constraints applied

Figure 3.1 shows that the MPC algorithm is successfully implemented. The small deviation in the transient is due to discretization effects.

3.2.2. State constraints

First, we have changed the way we approach sampling time. Instead of fixing it ourselves, we calculate it based on the simulation time and desired number of steps.

```
steps = 50;                    %Simulation steps
sim_time = 0.2;                %Simulation time (seconds)
Ts = sim_time / steps;         %Number of steps
```

From now on, we will observe forced response rather than natural response. Thus, we set x_0 to $[0\ 0]^T$. Furthermore, we add the desired minimal and maximal values of the states through a 2×1 vector, since we defined two state variables.

```
x_max = [10000; 500];      %Maximum state values (current [A], speed [rad/s])
x_min = [-10000; -500];    %Minimum state values (current [A], speed [rad/s])
```

Again, by initially setting them to such high values, we essentially turned them off in order to validate the algorithm first.

Inside the main MPC loop, we define the reference signal as a step at 0.02 seconds from 0 to 91 $\frac{\text{rad}}{\text{s}}$, which is the rated speed value for the particular DC motor.

```
%% MPC Loop
for k = 1:steps
  current_time = k * Ts;

  if current_time >= 0.02
    ref = 91 * ones(2 * p, 1); % Set reference to 100 for all states
  else
    ref = zeros(2 * p, 1); % Zero reference before 0.1 seconds
  end

  %Update state prediction
  x_pred = Phi * x + Gamma*u_opt;                    %(20x1 + 20x1)
  predicted_states(:, :, k) = reshape(x_pred, [2, p]);    %n_x x p x steps

  H = Gamma' * Q_bar * Gamma + R_bar;          %Quadratic cost matrix H

  f = (Gamma' * Q_bar * (Phi * x - ref))';      %Linear cost vector f


  %Adjust bounds for states
  b_x_upper = repmat(x_max, p, 1) - Phi * x; % Adjusted upper state bounds
  b_x_lower = repmat(x_min, p, 1) - Phi * x; % Adjusted lower state bounds

  %Combine constraints
  F_total = [Gamma; -Gamma; eye(m); -eye(m)]; % Input constraints
  F_x = eye(2 * p); % State constraints (identity for simplicity)
  F_total = [F_x * Gamma; -F_x * Gamma; eye(m); -eye(m)];
```

```
28
29      b_total = [b_x_upper; -b_x_lower; U_max*ones(m, 1); -U_min*ones(m, 1)];
30
31      %Solve the quadratic programming problem
32      u_opt = quadprog(H, f, F_total, b_total, [], [], [], [], []);
33
34      control_inputs(:, k) = u_opt(1:m); %Store control inputs for the horizon
35      u = u_opt(1);                      %Extract the first control input
36      x = Ad * x + Bd * u;               %Apply the control input
37
38      %Save trajectories
39      x_trajectory(:, k) = x;
40      u_trajectory(k) = u;
41
42      %Update previous input
43      u_prev = u;
44  end
```

Predicted states are now also affected by the calculated optimal control input through $\mathbf{\Gamma}$ matrix. Furthermore, the state constraints are now defined within the MPC loop, since the current state vector affects them in each time step. Using `repmat` MATLAB function, we expand the state constraints vectors to match the product of $\mathbf{\Gamma}$ and state vector x. This is done so that the upper and lower bounds for both state variables are adjusted within the loop.

When we were defining input constraints, we used the simplest, **fixed constraints**

$$b_u \leq u \leq b_l$$

Now, since we are defining additional constraints, and also those that dynamically affect input constraints, and vice versa, we use **linear inequality constraints**

$$\mathbf{F}_{\text{total}} \cdot x \leq b_{\text{total}}$$

Finally, b_{total} is a vector vertically concatenated with both input and state constraint vectors.

We define input constraints as F_{total}, and then expand F_{total} with the identity state constraint matrix $\mathbf{F_x}$. By applying the `quadprog` function, we get the optimal control input **with both input constraints and state constraints taken into account.**

3.2.3. Results

In the following pages, the results obtained in MATLAB simulation are shown for various cases. Initially, the constraints are set to extremely high values so we can check the validity of the algorithm by comparing it with the LQR.

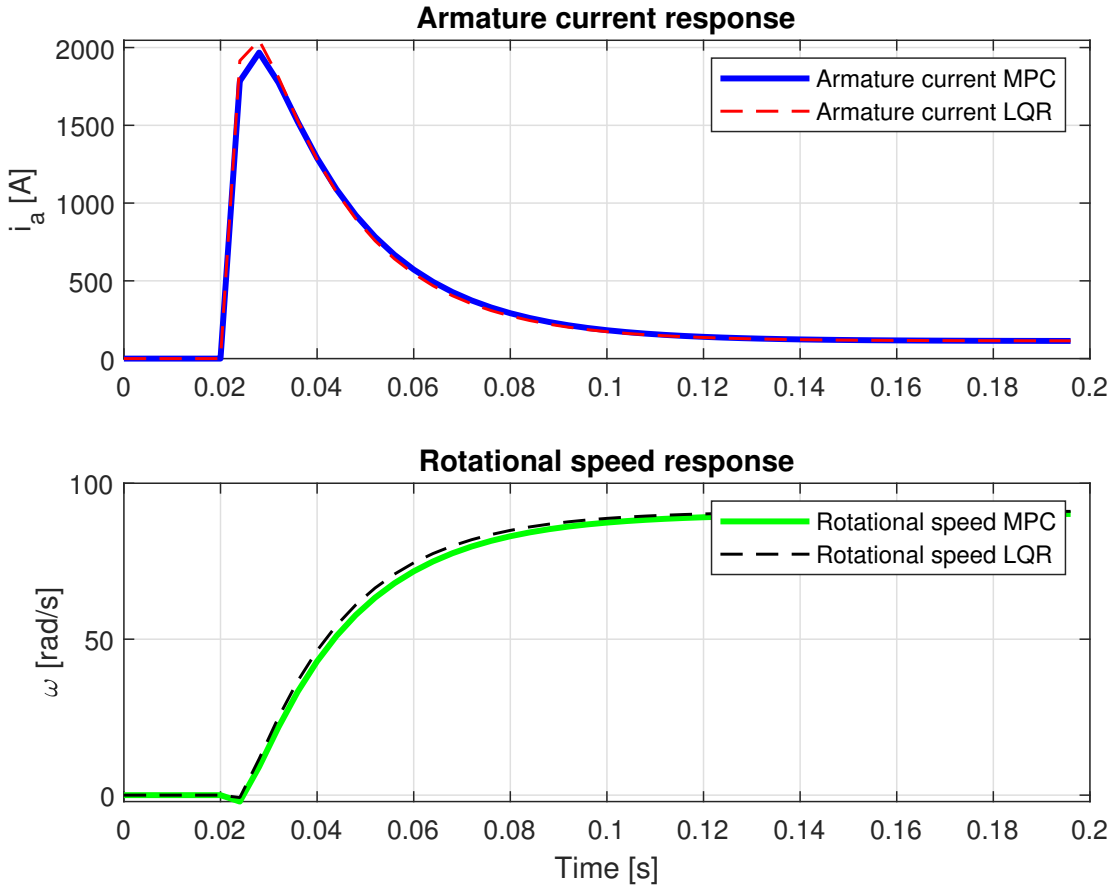

Figure 3.2. State responses comparison for LQR and MPC with turned OFF constraints

Figure 3.2 shows the results of both LQR and MPC when MPC constraints (both input and state) are turned off. As expected, the responses match, save for the little deviations due to discretization error, and LQR's inherent *infinite horizon*.

The deviation caused by horizon difference can be compensated with more aggressive control, that is, the higher values of matrix $\mathbf{Q}$.

Figure 3.3 shows the LQR response compared with the corresponding MPC response when state constraint of the armature current is set to 1000 amps. This, of course, makes the rotational speed response a bit slower, but ensures that the armature current stays within the desired limits.

Figure 3.4 shows the corresponding predictions made in each step. The predictions are based on assumption that the current control input remains applied for the rest of the horizon. Since the horizon is finite and relatively short, we see that in each step the predictions forecast motor's convergence towards zero-state. Only in the next step, the predictions update and take into account the updated control input sequence. We can also see that in the very first step after step reference, the prediction is made without constraint taken into account, that is, before the optimal control input was applied.

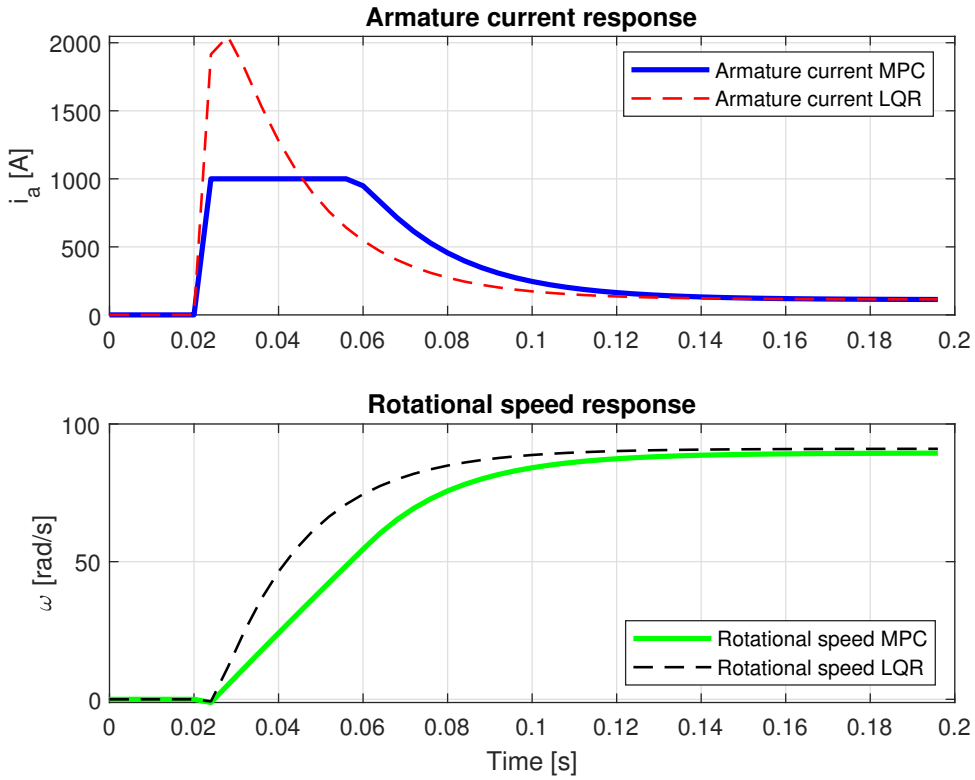

Figure 3.3. State responses comparison for LQR and MPC with turned ON constraints

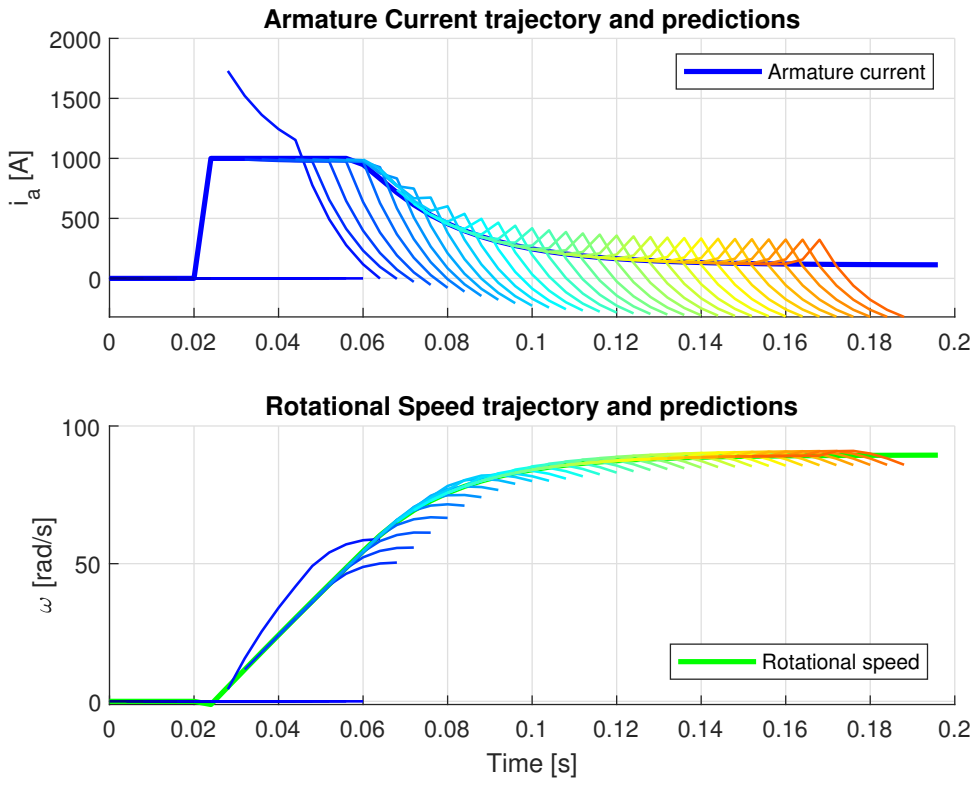

Figure 3.4. State predictions with turned ON constraints

To show that both states can be constrained, figure 3.5 shows MPC responses for the armature current being limited to 1000 amps, and rotational speed to 50 radians per second.

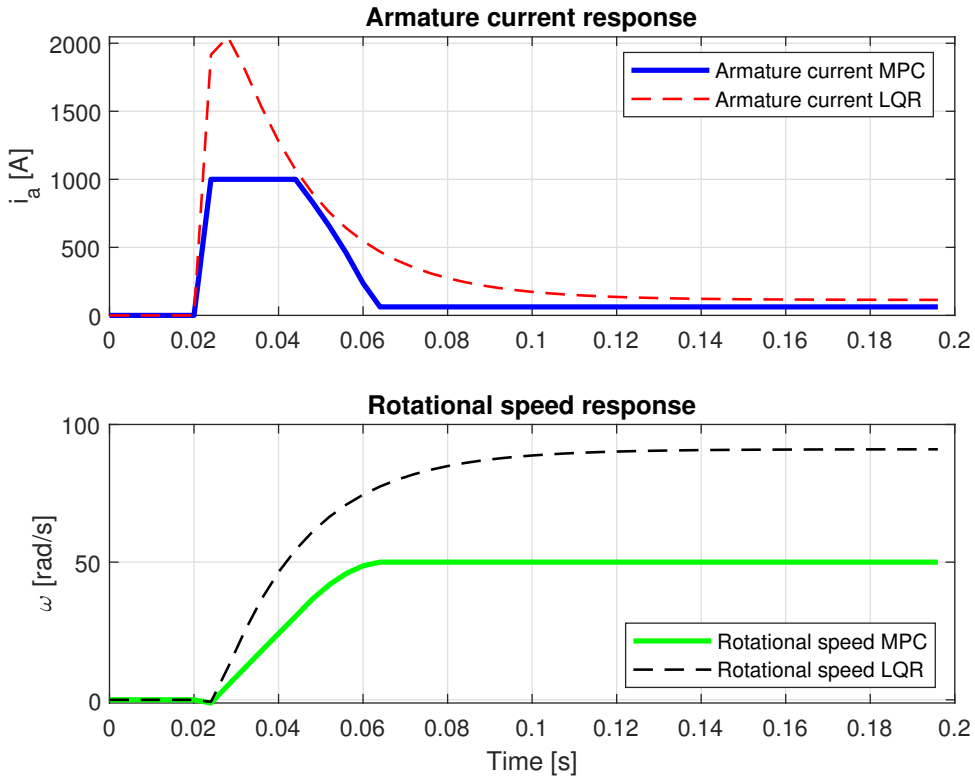

Figure 3.5. State responses comparison for LQR and MPC with constraints applied on both states

This concludes the validation of the MPC algorithm. As we have seen, the responses for LQR and MPC are matching when MPC constraints are turned off; in simulation, they are either set to extremely high values, or entirely removed from the `quadprog` command argument.

Furthermore, MPC states and control inputs are both successfully constrained which proves the algorithm's validity.

4. Implementation and testing of MPC on a linear pneumatic muscle

The motivation of implementing the algorithm on a pneumatic artificial muscle (PAM) is to design a robotic system which could potentially be used in rehabilitation of patients who suffered a stroke. According to [9], about one million people annually suffer a stroke, with many of them being left with permanent immobility of upper extremities. While the standard rehabilitation procedures include physiotherapists and classical robotic assistance systems, they also have some downsides such as rigid structure, inflexibility, and being unable to adjust various anatomies of various patients.

Soft robotics is an emerging field that focuses on creating flexible and compliant robotic devices which are particularly beneficial in physical rehabilitation. Unlike classical robots, soft robots provide enhanced safety and adaptability since they can be tailored to fit individual patients' anatomies, thus enabling personalized therapy solutions. Soft materials also provide greater comfort while using such device, and easier portability of a whole robotic exoskeleton.

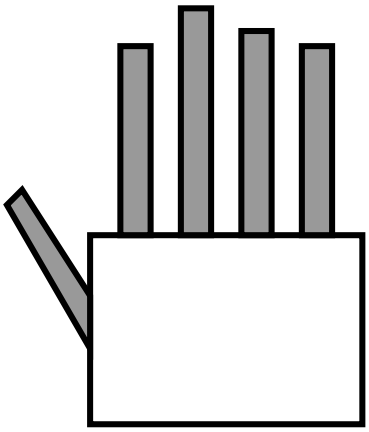

Figure 4.1. An illustration of a robotic arm with pneumatic artificial muscles as actuators

This part of the work focuses on the control algorithm and processing of signals acquired from sensors, such as EMG sensors which measure muscle activity and IMU sensors *(Inertial Measurement Units)* which are consisted of an accelerometer and a gyroscope. Mechanical construction of the robotic exoskeleton will not be discussed.

Modeling of a specific pneumatic muscle is beyond the scope of this work. Developing a detailed model would take much time, and in this simulation, arbitrary values are used for variables like friction coefficient, cross-section, mass, and so on. The goal is to prove that the algorithm is valid for this particular application, and to prove that it is able to track frequent changes in reference signal.

Another aspect beyond the scope of this work is signal processing from EMG and IMU sensors. The reference signal will be acquired based on the sensor readings in real time, but processing of those signals to obtain the appropriate reference signals is very detailed and complex since it takes into account many environmental variables, such as muscle activity, position, velocity, and acceleration.

4.1. Algorithm modifications and corrections

4.1.1. State-space model

A basic model and working principles of PAM were introduced in section 1.2.. Here, we will only modify the state-space model of it by paying attention to pneumatic behavior of the pneumatic muscle. Unlike in section 1.2., force $\mathbf{f}$ must not be treated as an independent input, since the only control input applied is voltage. Thus, we must modify the state-space model by eliminating force as an independent variable, and expressing it with the airflow state $x_3(t)$ as follows

$$\dot{x}_2(t) = \frac{\gamma A}{m} x_3(t) - \frac{b}{m} x_2(t) - \frac{k}{m} x_1(t), \tag{4.1}$$

where γ is proportionality constant linking airflow with pressure, and A is effective cross-sectional area of the pneumatic muscle. Since force is equal to a product of pressure and area, we replace $f(t)$ with $\gamma A x_3(t)$. The modified state-space model is now

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{q}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{k}{m} & -\frac{b}{m} & \frac{\gamma A}{m} \\ 0 & 0 & -\frac{1}{\tau} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ q(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{K}{\tau} \end{bmatrix} u(t) \tag{4.2}$$

4.1.2. Number of states

Since this system has three state variables unlike DC motor model which had two, the modification in the algorithm is that we define number of states as `n_x` with command `n_x = length(A)`. This should have been done in the first place, but now the algorithm automatically determines the number of states which are essential for the complex matrix multiplication. Otherwise, we would have needed to manually change the number of states in many places within the code.

4.1.3. Discrete LQR

Another modification is that instead of continuous LQR we use discrete LQR. This way we ensure that the discretization error is completely eliminated while comparing the responses. For the LQR, state matrices are discretized using MATLAB `c2d` command, with the discretization

method being *Zero-Order Hold*[1]. For extremely high precision and lower discretization error in general, *First-Order Hold*[2], *Tustin method*, and *Matching zeros and poles* can be used.

Number of simulation steps and simulation time are manually defined, while the sample time is defined as a ratio of simulation time and number of simulation steps.

4.1.4. Simulation parameters

Since the pneumatic muscle isn't modeled, the simulation parameters are also arbitrary. Prediction horizon and control horizon shall be determined as discussed in 2.1.1., which requires a real system model. This doesn't affect the algorithm in general, but must be adjusted for a specific application.

4.2. Results

In this section, the obtained results are shown. Figure 4.2 shows the results for a single step

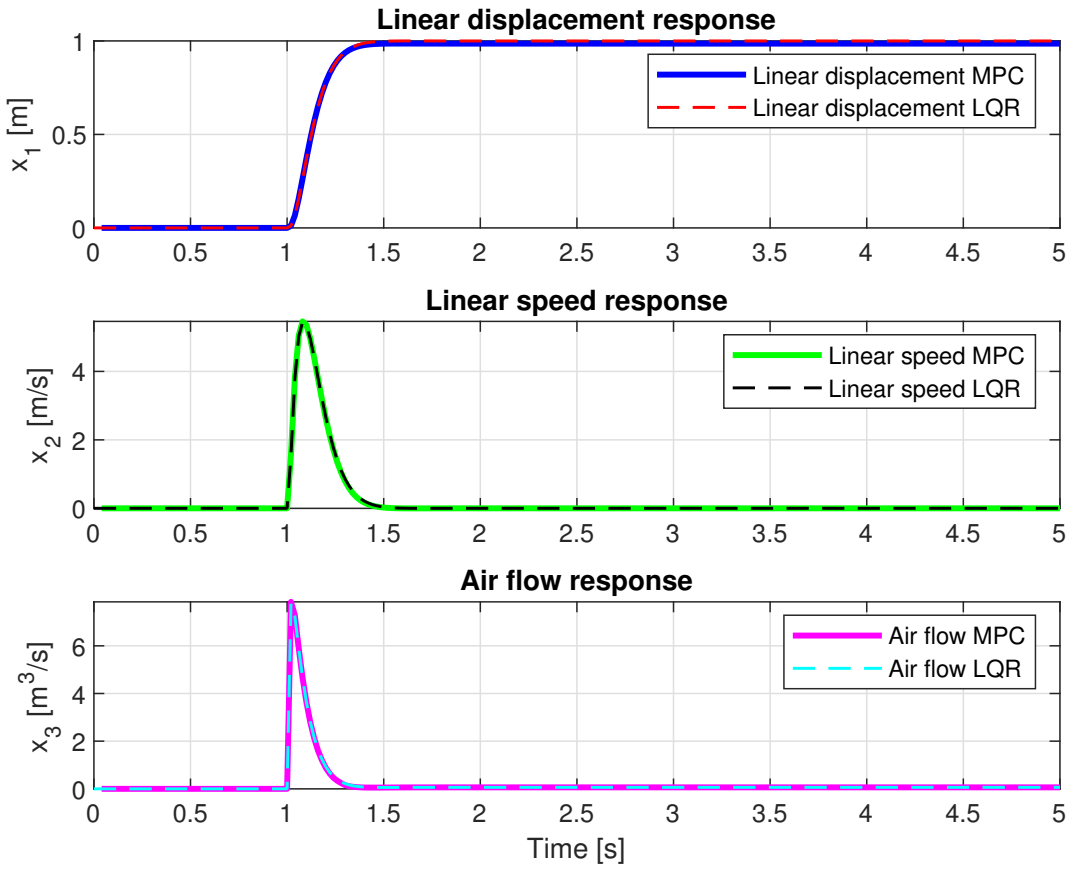

Figure 4.2. State responses comparison for LQR and MPC with a single step reference

[1]Interpolation between samples with constant values
[2]Interpolation between samples with lines

reference change at 1 [s]. Obviously, PAM's contraction/expansion range is way below that, but as mentioned before, parameters of the system are arbitrary. The point is that the reference tracking works properly.

This case could represent an example where a patient wants to simply move their finger in either direction. This is the simplest movement one can do, and according to these responses, it would be successfully realized.

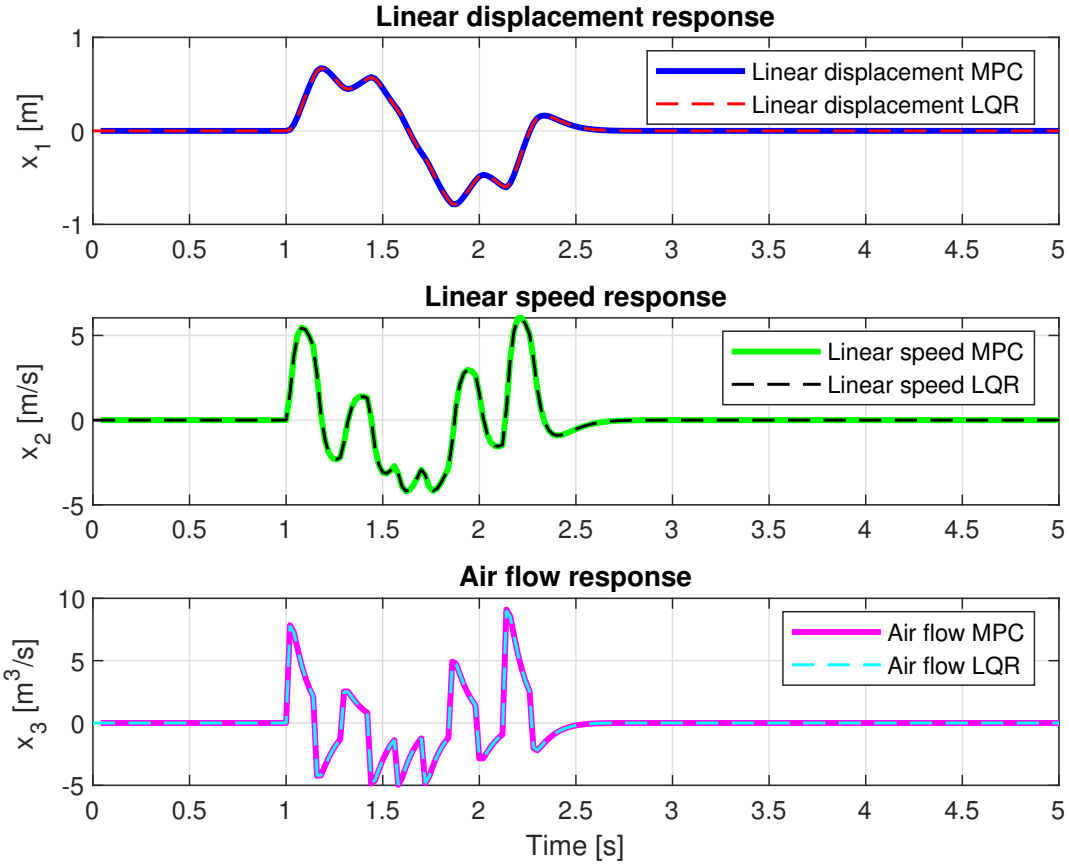

Figure 4.3. State responses comparison for LQR and MPC with multiple step references

As shown in figure 4.3, the algorithm successfully tracks frequent reference changes. These could represent more complex movements the patient might want to accomplish. In that case, reference signals obtained from EMG and IMU sensors would vary in a short time frame.

4.3. Future work

In this section, some propositions for future work on this projects are given. It is important to note that the MPC reference tracking depends on the relationship between prediction horizon, control horizon, and regulation matrices $\mathbf{Q}$ and $\mathbf{R}$. If regulation matrices are set to low values, or

if neither of the state variables is forced to be regulated, the prediction horizon must be extremely high.

This, of course, poses a significant load on the microcontroller in real time, and it would make it unable to execute the algorithm properly. Thus, it is extremely important to take care of the $\mathbf{Q}$ and $\mathbf{R}$ tuning within the simulation.

Regarding possible nonlinearities, it is important to address them, and potentially upgrade this algorithm to a NMPC described in 2.3., either using Koopman operator, or some other technique.

Both input and state constraints are implemented in the algorithm, but distinction between soft constraints and hard constraints isn't. This is a potential subject to future work.

4.3.1. Signal processing

As mentioned before, reference signals for the control algorithm are obtained from a signal processing subsystem which takes various sensor readings into account. For example, signals from EMG sensors are extremely noisy, sometimes even to the point where it is unable to distinguish the original signal. Increasing Signal-to-Noise ratio (SNR) and eliminating the noise is the biggest challenge in this part, and it requires various signal processing techniques, such as filters and more advanced techniques.

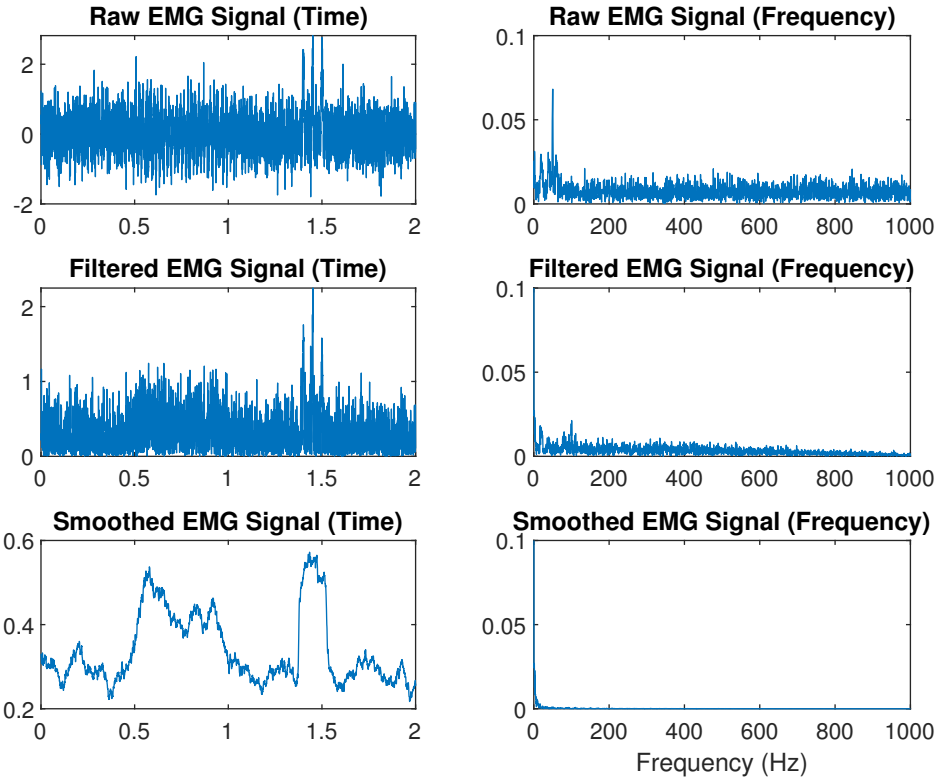

Figure 4.4. A simulation of a noisy EMG signal

A short example simulation in MATLAB is done as following. The signal acquired from EMG sensor is simulated with three cardinal sine functions $\left(\frac{\sin x}{x}\right)$ which represent muscle contractions, and EMG noise is generated with MATLAB `randn` function.

This is a very crude simulation since we will save the smoothed EMG signal to a `.mat` file and load it into our reference variable in the MPC algorithm, but the point is to prove that the algorithm is able to track rapid fluctuations in reference signal.

Once again, it is important to note that this isn't realistic and applicable at all, since the EMG signal, alongside IMU signals, must be processed and analyzed further to obtain corresponding reference signals. Since this is beyond the scope of this work, this is intended only as a starting point for some continuation on this work.

Here, a raw EMG signal is filtered with a band-pass filter in order to reduce low-frequency powerline interference, and high frequency noise. Instead of using one band-pass filter, two filters can be used - one notch filter for 50 Hz noise elimination, and one band-pass filter or low-pass filter for elimination of the high-frequency noise.

Figure 4.5 shows the MPC response when the reference is loaded from the signal generation file. With the proper tuning of regulation matrices and algorithm parameters, the algorithm is able to track extremely rapid changes in the reference signal.

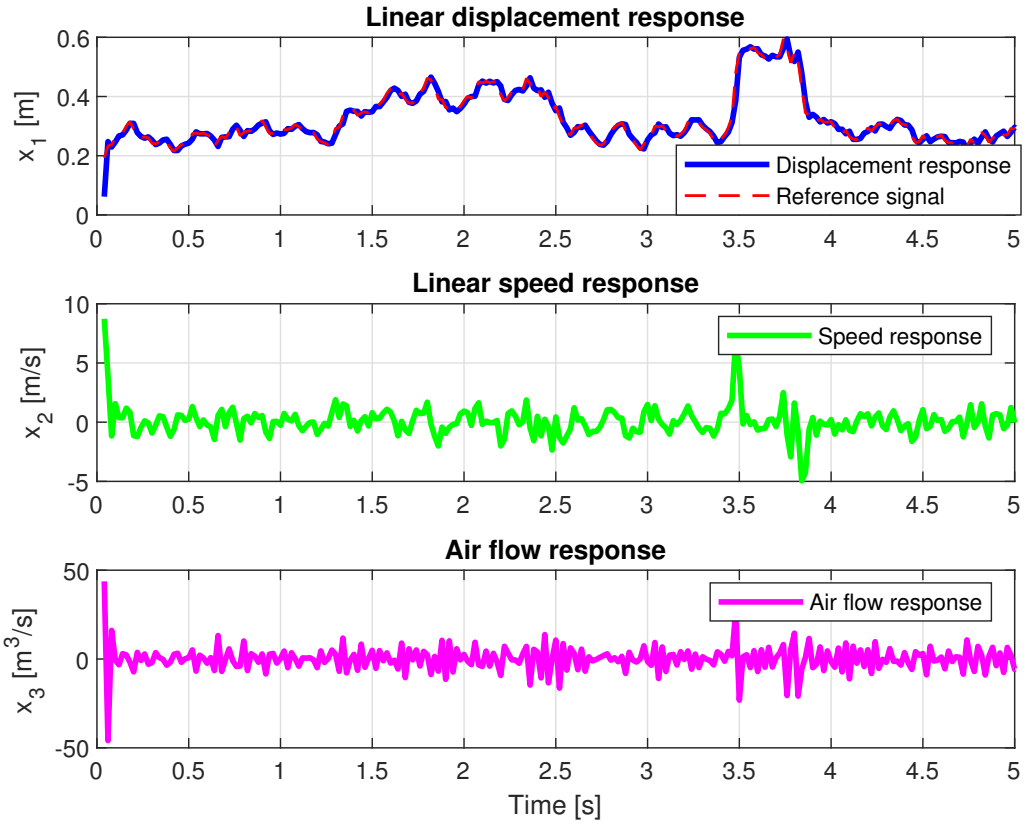

Figure 4.5. A simulation of a noisy EMG signal

Of course, such rapid oscillations in the linear displacement response will not be desirable in the practical implementation, since that implies unpredictable jolting of a pneumatic muscle. That is to be dealt with by signal processing part, since EMG signals will not be directly fed into reference signal anyways. This proves that the control algorithm will not slow down the actuation in practical implementation in the future.

Some notable work on signal processing is done in [10].

4.3.2. Mechanical construction

Mechanical construction is the final crucial part of the soft-robot exoskeleton development. Some work on this matter is already done by other students. See [11], [12], and [13].

Bibliography

[1] N. S. Nise: *"Control Systems Engineering"*, 7th Edition, Wiley, 2014.

[2] M. Tóthová, J. Pitel, A. Hošovský, J. Sárosi: *"Numerical Approximation of Static Characteristics of McKibben Pneumatic Artificial Muscle"*, International Journal of Mathematics and Computers in Simulation, 2015.

[3] João P. Hespanha: *"Linear Systems Theory"*, Princeton Univeristy Press, 2018.

[4] William S. Levine (editor): *"Control System Advanced Methods - The Control Handbook"*, CRC Press, Taylor & Francis Group, 2011.

[5] MATLAB Documentation

[6] David A. Haggerty, Michael J. Banks, E. Kamenar, et. al.: *"Control of soft robots with inertial dynamics"*, Science Robotics, 2023.

[7] M. Korda, I. Mezić: *"Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control"*, Science Direct, University of California, Santa Barbara, 2018.

[8] L. Wang: *"Model Predictive Control System Design and Implementation Using MATLAB®"*, Springer, 2009.

[9] E. Kamenar: Lectures from *"Control of mechatronics systems"*, Faculty of Engineering, University of Rijeka, 2024.

[10] T. Bazina, E. Kamenar, et. al.: *"Koopman-driven grip force prediction through EMG sensing"*, arXiv pre-print, submitted on IEEE, 2024.

[11] K. Jurić: *"Konstrukcija mekog robota za rehabilitaciju šake"*, University of Rijeka, Faculty of Engineering, 2023.

[12] L. Gašparić: *"Konstrukcija pneumatske meke robotske hvataljke"*, University of Rijeka, Faculty of Engineering, 2024.

[13] M. Kladarić: *"Razvoj rukavice za rehabilitaciju koristeći princip meke robotike"*, University of Rijeka, Faculty of Engineering, 2024.