CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering
Department of Control Engineering

Predictive Control for the SK8O Robot

Master's thesis

Petr Brož

Supervisor
Loi Do

2024

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Brož Petr**	Personal ID number:	**466185**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Predictive control for the SK8O robot

Master’s thesis title in Czech:

Prediktivní řízení pro robota SK8O

Guidelines:

The main goal is to design a predictive controller for the motion control of the SK8O robot. The controller should be able to balance and steer the robot. To this end, the following steps will be taken:
1) Select a predictive control scheme suitable for the motion control of the SK8O robot. Specifically, choose a scheme from the variations of Model Predictive Control (MPC) for nonlinear systems.
2) Verify the designed controller through simulations.
3) Deploy the controller on the actual robot hardware.
4) Assess the controller’s performance through hardware experiments. Compare the performance with other methods.

Bibliography / sources:

[1] Korda, Milan, and Igor Mezić. "Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control." Automatica 93 (2018): 149-160.
[2] Brunton, Steven L., et al. "Modern Koopman Theory for Dynamical Systems." SIAM Review 64.2 (2022): 229-340
[3] Rawlings, James Blake, David Q. Mayne, and Moritz Diehl. Model predictive control: theory, computation, and design. Vol. 2. Madison, WI: Nob Hill Publishing, 2017.
[4] Borrelli, Francesco, Alberto Bemporad, and Manfred Morari. Predictive control for linear and hybrid systems. Cambridge University Press, 2017.
[5] Dominik, Hodan. Reinforcement learning-based control system for the SK8O robot. Master’s thesis. České vysoké učení technické v Praze. Výpočetní a informační centrum., 2023.

Name and workplace of master’s thesis supervisor:

Ing. Loi Do Department of Control Engineering FEE

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

Date of master’s thesis assignment: **29.08.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until:
by the end of summer semester 2023/2024

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

III. Assignment receipt

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

.

Date of assignment receipt	Student’s signature

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Petr Brož
January 2024

Acknowledgments

I thank Loi Do for his his guidance and extensive advice, on both control and typography. And for letting me use his LaTeX template!

I thank Martin Gurtner and Krištof Pučejdl, without whom Sk8o wouldn't exist.

And last but not least, I thank David Šibrava and Eliška Čukanová. For everything.

Abstract

In this master's thesis, a controller capable of balancing and steering Sk8o, an unstable, segway-like bipedal wheeled robot, is designed. This is achieved using Model Predictive Control, formulated as a quadratic program. This program is solved using OSQP. The resulting controller is validated in simulations, deployed onto the robot and tested in hardware experiments, comparing it to a preexisting LQR controller. Three distinct experiments are shown: stabilization in the presence of an external disturbance, following a simple trajectory (driving forward and backwards) and following a complex trajectory (slalom). Additional controllers using Koopman Model Predictive Control are found via Extended Dynamic Mode Decomposition and are likewise tested and deployed on the robot. The thesis concludes by discussing issues of those additional controllers.

Keywords: Model Predictive Control, Koopman Model Predictive Control, Receding Horizon, Extended Dynamic Mode Decomposition, Sk8o, segway, unstable system, quadratic program, OSQP

Abstrakt

V této diplomové práci se věnuji návrhu regulátoru schopného balancovat a řídit nestabilního dvounohého kolového robota Sk8o, jehož struktura je podobná vozidlu segway. Tohoto je dosaženo pomocí metody Model Predictive Control, která je řešena jako kvadratický program. K řešení tohoto programu je užit solver OSQP. Výsledný regulátor je ověřen v simulacích a nasazen na robota, kde jsou následně provedeny experimenty. Tři různé experimenty jsou ukázány: stabilizace při zavedení vnější poruchy, následování jednoduché trajektorie (jízda vpřed a vzad) a následování složitější trajektorie (slalom). Regulátor je porovnán s již existujícím regulátorem typu LQR. Následně jsou formulovány další regulátory metodou Koopman Model Predictive Control, s pomocí algoritmu Extended Dynamic Mode Decomposition. Tyto jsou též nasazeny na robota a ověřeny v simulacích a experimentech. Na konci práce jsou probrány problémy těchto dalších regulátorů.

Klíčová slova: Model Predictive Control, Koopman Model Predictive Control, Receding Horizon, Extended Dynamic Mode Decomposition, Sk8o, segway, nestabilní systém, kvadratický program, OSQP

Contents

1 | Introduction

The robot Sk8o, captured in figure 1.1 in all of its dazzling beauty, could not possibly be accused of lacking style. Until recently, it did however lack a predictive controller. In this thesis, I rectify the situation by implementing *model predictive control* (MPC) for Sk8o.

MPC is a control scheme that selects optimal sequences of control actions by harnessing a model to predict the evolution of the controlled system.

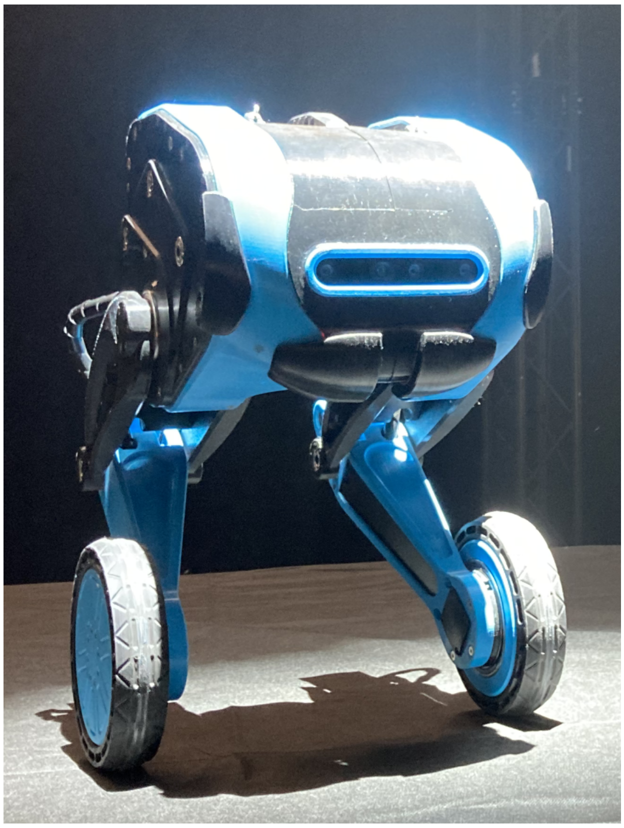

Figure 1.1: Photo of Sk8o

1.1 Problem Statement

My task is twofold. First and foremost, the robot must be balanced, as by itself it is unstable. The second objective is to steer the robot in accordance with references to its velocity and yaw rate, which are continuously provided by a human operator.

1.2 Thesis Outline

I begin by describing the Sk8o robot in chapter 2. Its sensors, actuators and processors are discussed, as well as the overall hardware architecture. Also introduced within the chapter is a nonlinear mathematical model that will be used in further parts of the thesis.

In the somewhat technical chapter 3, I describe changes I made to the robot's programming in order to allow for the deployment of MPC. I also perform preliminary tests of the robot's computational performance.

Chapter 4 details the design of the controller. It defines a *linear predictor*[1] and formulates MPC as a quadratic program. A section detailing the implementation in Sk8o follows. I then verify the controller through simulations and conclude the chapter by performing experiments on the real hardware.

Having achieved the goals set out by my assignment, I decided to explore further. In chapter 5, I proceed to a data-driven variant of MPC, the *Koopman model predictive control* (KMPC), which makes use of a more complex *lifted linear predictor*, in an effort to better forecast the robot's future. The majority of the chapter concerns itself with obtaining the lifted linear predictor. Once that is achieved, the resulting controller is validated in simulations and hardware experiments, just like the MPC controller before it. The performance of the KMPC controller falls short of my expectations, and possible causes are discussed.

The thesis concludes in chapter 6, in which I also provide a list of open problems.

[1]Usually, the entity referred to here as linear predictor would simply be called a *linear model*. I use the name linear predictor for the sake of compatibility with another control scheme – *Koopman* MPC – which I also tackle later in the thesis.

2 | Sk8o Robot

In this chapter I will provide a brief description of the actuators, sensors and processors of the robot I was tasked with controlling. I will touch upon the internal organisation of the robot and the information flow within it. I'll conclude the chapter by introducing a model of the robot.

The Sk8o robot was designed, built and programmed by Krištof Pučejdl and Martin Gurtner of the AA4CC group at CTU FEE, while the dynamical model I make use of was derived by Dominik Hodan in his master's thesis [6]. Thus, this chapter does not describe my contributions, except where explicitly stated.

Sk8o, depicted in figure 1.1, is a bipedal, wheeled robot. Its mode of locomotion is similar to that of the Segway vehicle – the robot drives its wheels to turn and move its body, while also maintaining an upright position. Additionally, the robot is capable of extending and contracting its legs.

2.1 Actuators

Sk8o is endowed with four eX8108 105KV BLDC motors, each of which is controlled by Ben Katz's 3-phase motor controller[1]. Two motor assemblies are placed at the wheels of the robot, while the remaining two reside in the robot's *hips*. Those two hip motors are sufficient for fully controlling the lengths of the legs – this is due to the structure of the limb, which contains a kinematic loop.

Two modes of operation are supported by the motor controllers. Either the position of the motor (measured by a hall effect sensor mounted on the controller) or the motor's torque (calculated from currents applied to it) can be controlled. The first mode is used in the hip motors, which are tasked with maintaining a selected leg extension, while the wheel motors use the torque-control mode. The maximum torque setpoint is software-limited to 0.7 N m.

[1] https://github.com/bgkatz/3phase_integrated

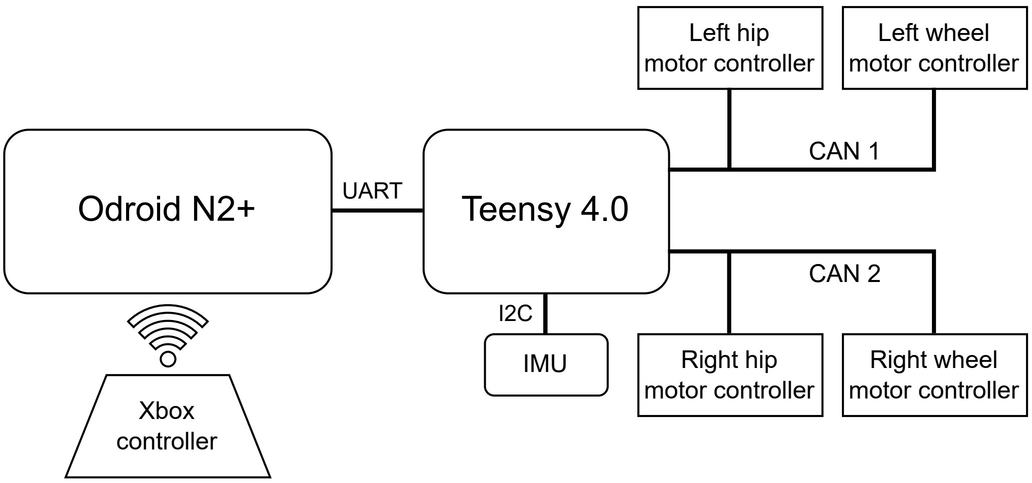
Left hip motor controller
Left wheel motor controller
CAN 1
Odroid N2+
UART
Teensy 4.0
CAN 2
I2C
IMU
Right hip motor controller
Right wheel motor controller
Xbox controller

Figure 2.1: Hardware diagram of Sk8o

2.2 Sensors

Each motor controller measures and reports the position and velocity of its motor, as well as the torque exerted by it. An ICM-42688 IMU attached to the robot body provides measurements of accelerations, angular rates and Euler angles. The IMU has a built-in antialiasing filter, whose bandwidth is set to 1051 Hz. Lastly, a RealSense camera can be mounted on the robot. The camera will be omitted from further discussion, as it is not utilized in this thesis.

2.3 Hardware Architecture

Figure 2.1 shows the connections between individual components of the Sk8o robot's hardware. There are two computational resources available: the Teensy 4.0 board and the Odroid N2+ computer.

The single-cpu Teensy is programmed in C++ and its main loop runs at 1 kHz. Its tasks include periodically gathering measurements from the motor controllers and from the IMU and transmitting them (on demand) to the Odroid computer.

The Odroid computer boasts a total of six CPU cores[2] and runs Ubuntu. It allows the robot to interface with the outside world – most notably, to wirelessly connect to an Xbox controller, which is used by the robot's operator to steer it. Individual functions of the Odroid computer are implemented as services, programmed either in Python or in C++. Communication between the services is facilitated by *Lightweigh Communications and Marshalling* (LCM)[3], using a publisher-subscriber pattern.

Originally, controlling Sk8o was done exclusively from within the Teensy board – i.e., the Odroid computer would only concern itself with passing commands from the Xbox controller to Teensy. As part of the preparatory work for my thesis, I've made some modifications (elaborated in section 3.1) which allow for the roles to be reversed: a controller can now run in the Odroid computer, with Teensy acting merely as a middleman passing the required actions to the motor controllers. This allows for computationally demanding control schemes to be used, such as in [6], or indeed, in this very thesis.

[2]Consisting of a Quad-core Cortex-A73 and a Dual-core Cortex-A53.

[3]See `https://github.com/lcm-proj/lcm` for details.

2.4 System model

I use the Segway model of the robot derived in [6]. It approximates Sk8o as a wheeled inverted pendulum of constant length – the position of the hip motors is therefore fixed. This results in a nonlinear system with two inputs and six states

$$u^T = \begin{bmatrix} u_\mathrm{L} & u_\mathrm{R} \end{bmatrix}, \qquad x^T = \begin{bmatrix} \dot\chi & \dot\varphi & \dot\psi & \chi & \varphi & \psi \end{bmatrix}, \tag{2.1}$$

where u_L and u_R are torques applied to the left and right wheels, respectively, χ is the distance travelled, φ is the angular deviation of the pendulum from the upright position, and ψ is the angle of rotation around the vertical axis. The states are also illustrated in figure 2.2.

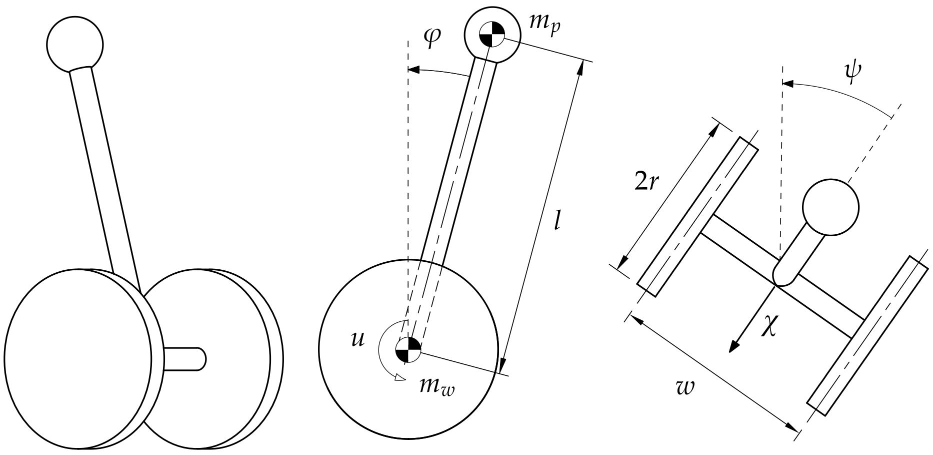

Figure 2.2: The Segway model (figure reproduced with permission from [6])

The equations of the model, as well as values of m_w, m_p, w, l, and r are reproduced in appendix A.

3 | Preparatory Work

When I began work on my thesis, the only control scheme deployed on the Sk8o robot consisted of an LQR controller running on the Teensy board. Model predictive control is significantly more computationally demanding than LQR, as in each step it queries a quadratic program solver. Thus, given a choice between using the Teensy microcontroller or the full-fledged Odroid computer, it is natural to pick the latter. As controlling Sk8o from Odroid was novel at the time, it seemed prudent to first ascertain its feasibility and to remove any obstacles that might arise[1].

Recalling figure 2.1, using Odroid moves the controller a step further away from the actuators. Several questions needed to be answered before proceeding to MPC implementation:

- Will this move introduce significant communication delays?
- Will Odroid be usable for a real-time application?
- Which combinations of prediction horizon length[2] and control frequency are achievable?

To help me answer these questions, I ported a version of the LQR controller (tuned for a control frequency of 200 Hz) from Teensy to Odroid. This experiment was eventually successful in two ways – firstly, I was able to balance and steer Sk8o by LQR running on Odroid (albeit with worse performance compared to running it on Teensy[3]). More importantly, in porting the controller I became aware of a few

[1]Another motivation for exploring the issues of controlling Sk8o from Odroid was the concurrent work of Dominik Hodan, who in [6] controls Sk8o using reinforcement learning – an approach that likewise makes use of the Odroid computer.

[2]The term prediction horizon will be explained in more detail in chapter 4. In a nutshell, the length of the prediction horizon N describes how far into the future the controller must plan. As such, the value of N plays an important role in determining the computational complexity of MPC.

[3]In particular, the controller exhibited a tendency towards oscillations. I suspect this could have been caused by the inevitable increase in communication delays, even after issues discussed

problems in the existing codebase that needed to be addressed.

3.1 Resolving Communication Issues

Simplifying somewhat for the sake of brevity, the program running on the Teensy consisted of a *main loop* that was executing without any timing restrictions and a secondary *control loop* that executed once per millisecond. The main loop executed the following pseudocode:

```
1    read and save a message from CAN 1, if available
2    read and save a message from CAN 2, if available
3    read 1 byte from UART into buffer, if available
4    if buffer contains whole UART message
5        parse and save UART message
6        empty buffer
```

The control loop then undertook all other functions. It could (by a UART message) be switched into a mode in which it ran the LQR controller (with references coming over UART and calculated actions passing to motor controllers via CAN). The other mode, relevant to my use case, was called *motor mode*. In it, the microcontroller awaits a UART message with actions and passes it over to motor controllers directly.

There are two issues with this implementation: firstly, lines 1 and 2 of the main loop's pseudocode together take 30 µs to execute. Since the length of a UART message containing actions to be passed to motor controllers is 35 bytes, and both CAN lines are checked before receiving each byte over UART, a needless delay of 1.05 ms is incurred. This is significant compared to the transmission time of the message over UART, which given the baudrate of 2M, equals 175 µs.

The second issue results from the message being retransmitted to motor controllers from within the control loop. Depending on when the last byte of the message is received by Teensy, up to another millisecond of delay adds up.

further in this chapter were solved. Perhaps this could be fixed by retuning the LQR – though I chose not to spend time on doing so, as running LQR on Odroid wasn't the objective of this thesis.

I fixed the first issue by replacing line 3 of the pseudocode above with logic that continues reading bytes from UART until the message currently being received is complete (or until there are no more bytes to be read, whichever happens first). The second problem is then fixed by retransmitting the motor references instantly upon receiving the message, from the main loop. Thus, the new main loop is equivalent to following pseudocode:

```
1     read and save a message from CAN 1, if available
2     read and save a message from CAN 2, if available
3     WHILE a byte is available to read from UART
4         read 1 byte from UART into buffer
5         IF buffer contains whole UART message
6             BREAK
7     IF buffer contains whole UART message
8         parse UART message
9         IF message contains commands for motors
10            IF motor mode is active
11                send commands to motors over CANs
12        ELSE
13            save UART message
14        empty buffer
```

3.2 Real-time Usability of Odroid

An undisputable advantage that is lost when moving control from Teensy to Odroid is that of simplicity. The program running on Teensy makes use of just one timer-based interrupt. It is therefore easy to understand the microcontroller's timing, and one can be certain that its performance will be real-time.

The same cannot be said for Odroid, as it is much more complex. It's not a reasonable goal to attempt to understand the inner workings of the whole operating system and its scheduling to such depth as to be certain of its performance.

Something I have briefly investigated was whether a real-time operating system could be made to work on Sk8o. This, alas, didn't lead anywhere. There is no official support for the real-time kernel from Odroid's manufacturer Hardkernel, and while there is evidence of someone trying to apply a real-time patch on their own[4], doing so apparently leads to driver issues.

This regrettably means that there are no guarantees when it comes to the real-time performance of Odroid. Still, some steps in the right direction could be taken. I decided to dedicate two cores of the faster Cortex-A73 CPU to the critical services needed for control – one for the controller service itself and the other for an existing service that facilitates UART communications with the Teensy. This is done by adding the cores to the isolcpus parameter of Odroid's boot.ini file and launching the services with affinities for their respective CPU cores via taskset.

I also measured the jitter of the aforementioned LQR running in C++ on Odroid, on its dedicated CPU core. I ran the program for 30 s, during which time its main loop executed a total of $6 \cdot 10^3$ times. I have calculated the jitter j_i in each step as

$$j_i = t_{i-1} - t_i - T \ , \tag{3.1}$$

where t_i is the time of executing the i-th loop and T is the control period of the LQR, i.e. 5 ms. Note the absence of absolute value in the equation. The result is visualized in figure 3.1.

I deemed this result acceptable. The jitter appears fairly consistent and never exceeded 2 % of the control period.

3.3 Predicton Horizon Length and Control Frequency

The goal of this last preliminary test was to obtain an estimate of the time it will take to solve the MPC quadratic program and to pick a feasible combination of horizon length and control frequency. To this end, I first generated 10^4 random initial conditions of the Sk8o system. For each one, I formulated the MPC quadratic program, as will be described in section 4.2. To solve the QP, I used the Python interface[5] for the OSQP solver, described in [11]. I noted the times that elapsed

[4]`https://forum.odroid.com/viewtopic.php?f=55&t=41129`

[5]https://github.com/osqp/osqp-python

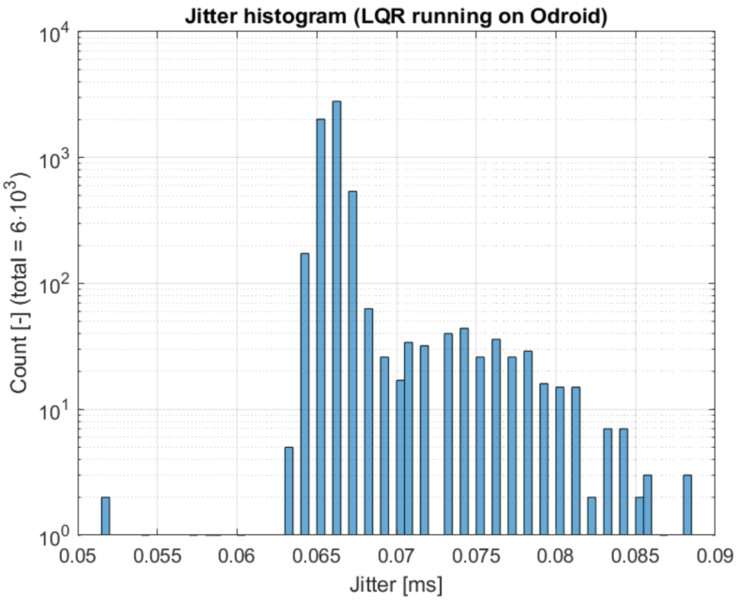

Figure 3.1: Histogram of jitters measured over 30 s of LQR run time

until OSQP returned a solution. This process was repeated for various lengths of prediction horizon N, sweeping 5 to 100. The solver's setup parameters were kept at their default values, with the exception of `warm_start` (set to True) and the couple `eps_abs` and `eps_rel`, which were both set to 10^{-4}.

The testing program was written in Python (which at that point I had decided to use instead of C++) and ran on the Odroid computer on its own dedicated CPU core – mirroring the conditions under which the real controller would eventually run.

The mean values of times taken to find a solution are shown in figure 3.2. The results were encouraging and led me to choose the combination of a control frequency of 50 Hz and control horizon length of 20 steps. Looking at the figure, this choice might appear overly cautious – a frequency of 50 Hz gives the controller 20 ms to come up with a solution. The mean time taken in the test to find a solution at this horizon length was just 1.3 ms. There were however reasons to be careful.

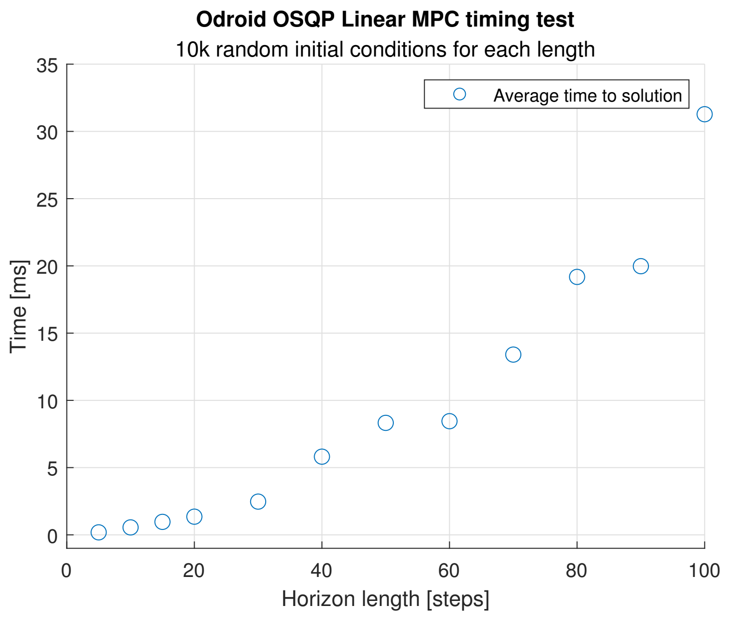

Figure 3.2: Results of Odroid OSQP timing test

Firstly, the MPC used in this test was based on a slightly modified version of the model described in section 2.4 – one that omitted the states x and ψ, reducing the state vector's dimension from 6 to 4. It wasn't clear at the time whether the full model would be necessary or not[6].

Secondly, a linear predictor obtained by linearization of the nonlinear model was used in the test. I planned to eventually also try other, more computationally demanding linear predictors – in particular, the Koopman variant of model predictive control, described in chapter 5, would make use of *lifted linear predictors*. More details will be provided in the dedicated chapter – for now, I'll limit myself to noting that using a lifted linear predictor is akin to artificially increasing the state space dimension. The dense MPC formulation I describe in 4.2 minimizes the impact of this increase on computational complexity, however, it is not eliminated fully.

[6]It would turn out to be necessary, because with the reduced model Sk8o had a tendency of drifting away due to imperfect velocity tracking.

Thirdly, in the real controller, some of the time budget would need to be allocated to tasks other than solving the quadratic problem – to publishing the solution and possibly also debug data as LCM messages, at the very least.

And lastly, one needs also to consider the *duration* of the control horizon as measured in *seconds*, rather than just its length measured in steps. Since the duration of each step of the horizon is equal to the control period, doubling the control frequency while keeping the horizon's length constant will halve its duration. For the combination of 50 Hz and 20 steps, the duration of the horizon is 0.4 s, a time I considered sufficiently long to capture the evolution of the robot's dynamics. If I wished to increase the frequency to 100 Hz, I would need 40 steps to achieve the same duration. And the time to solution for 40 steps averaged at 5.8 ms already. In doubling the frequency, I would go from needing 6.5 % of available time to 58 %.

In light of all those points, I chose to err on the side of caution and select 20 steps at 50 Hz.

4 | Model Predictive Control

Model predictive control (MPC) is a control scheme that seeks to, in each control step, plan out an optimal sequence of actions. This optimality is understood in the sense that the controlled system's trajectory, which would result from such a sequence of actions, minimizes a user-defined cost function. Once such a sequence is found (which is achieved by solving a quadratic program), the first element of the sequence is applied to the system, and the rest of the sequence is discarded. In the next step, the whole process repeats. MPC only predicts (and plans) a certain number of steps into the future – it only considers a finite prediction horizon.

I will begin by describing a *linear predictor*, which facilitates the forecasts of the system's evolution. This will be followed by a concrete definition of MPC and its formulation as a quadratic program. After that, I will describe how I implemented MPC in the Sk8o robot and conclude by verifying the resulting controller – both in simulations and by running experiments on the physical hardware.

4.1 Linear Predictor

Let the system, whose evolution needs to be predicted, be a nonlinear discrete-time system

$$x_{k+1} = \tau(x_k, u_k) \; , \tag{4.1}$$

where $x_k \in \mathbb{R}^6$ is the state vector of the system at time k, $u_k \in \mathbb{R}^2$ is the input applied to the system at time k, and $\tau(x, u)$ is a nonlinear function. I assume that all the states of the system are measured.

The sought linear predictor is in the form of a linear discrete-time system

$$\begin{aligned} z_{k+1} &= Az_k + Bu_k \ , \\ \hat{x}_k &= Cz_k \ , \end{aligned} \tag{4.2}$$

where $\hat{x}_k$ is the predicted state of the nonlinear system at time k and z_k is the predictor's state at time k. Now, let's discuss the dimension of z_k with respect to the order of the nonlinear system.

Firstly, if the dimension of z_k is equal to that of x_k, the option naturally presents itself to set $z_k = x_k$. In that case, C is an identity matrix and matrices A, B may be identified by linear approximation of $\tau(x, u)$ at a suitable operating point. Alternatively, if trajectories of the nonlinear system are available, A, B may be found in a data-driven way, for example, via the Extended Dynamic Mode Decomposition algorithm, which will be described in section 5.3. The linear predictor with $z_k = x_k$ will henceforth be called the *unlifted linear predictor*.

Conversely, a lifted linear predictor is one where the dimension of z_k is greater than that of x_k. Use of lifted linear predictors is the defining characteristic of the *Koopman MPC* (KMPC), and as such, will be dealt with in chapter 5.

For the rest of this chapter, I will only use the unlifted linear predictor.

For completeness' sake, one could also devise a linear predictor whose z_k had a lower dimension than x_k. Such a predictor would then only be capable of forecasting a subset of the original system's states. While such a predictor could be useful in some settings, I don't expect the need for reduction of the state's dimension to arise in Sk8o, which is already fairly low-dimensional. I will thus not discuss this variant further.

4.2 MPC Formulation

Assuming that an unlifted linear predictor was obtained, it is now possible to formulate MPC as a quadratic program to be solved by a QP solver.

The goal is to drive the state of the system x_k towards a reference r_k. This is equivalent to driving the regulation error $e_k = r_k - x_k$ towards zero. A second requirement is to minimize used actions, so as to conserve energy. Those objectives (and their relative weighing) are described by a standard quadratic cost function

$$J = \frac{1}{2} e_N^T S e_N + \frac{1}{2} \sum_{k=0}^{N-1} e_k^T Q e_k + u_k^T R u_k^T \ , \tag{4.3}$$

where N is the length of prediction horizon and e_k, u_k are the regulation error and control action at time k, respectively. Matrix $S \geq 0$ penalizes the error at the end of the prediction horizon, matrix $Q \geq 0$ does the same within the horizon and $R > 0$ penalizes the actions.

The full optimization problem is then

$$\begin{aligned} \min_{u_k, e_k} \quad & J(\{u_k\}_{k=0}^N, \{e_k\}_{k=0}^N) \ , \\ \text{subject to} \quad & z_{k+1} = A z_k + B u_k \ , \\ & e_k = r_k - C z_k \ , \\ & u_{\min} \leq u_k \leq u_{\max} \ , \\ \text{parametrized by} \quad & x_t = \text{given} \ , \\ & z_0 = x_t \ , \\ & r_k = \text{given} \ , \\ & k = 0, 1, ..., N \ , \end{aligned} \tag{4.4}$$

where matrices A, B, C constitute an unlifted linear predictor (4.2), and $u_{\min}$, $u_{\max}$ are limits imposed on the actions. I do not impose any constraints on the system's state.

The optimization problem (4.4) as formulated scales with the dimension of z_k, as it is a decision variable. While this is not a concern in this chapter, where the dimension of z_k is constant[1], it would become an issue in chapter 5, where the

[1] In an unlifted linear predictor, the dimension of z_k is always equal to that of the original system, i.e. 6 in the case of Sk8o.

dimension of z_k is increased[2]. Thus, I chose to convert the problem to the *dense* MPC formulation, where only u_k are decision variables.

Let

$$\begin{aligned} U &= [u_0^T, u_1^T, ...u_{N-1}^T]^T \ , \\ r &= [r_0^T, r_1^T, ..., r_N^T]^T \ . \end{aligned} \tag{4.5}$$

The dense formulation of MPC as a quadratic program can then be written as

$$\begin{aligned} \min_{U} \quad & \frac{1}{2}U^T H U + \begin{bmatrix} z_0^T & r^T \end{bmatrix} F^T U \ , \\ \text{subject to} \quad & u_{\min} \leq u_k \leq u_{\max} \ , \\ \text{parametrized by} \quad & x_t = \text{given} \ , \\ & z_0 = x_t \ , \\ & r_k = \text{given} \ , \\ & k = 0, 1, ..., N \ , \end{aligned} \tag{4.6}$$

Here the predictions of future states are no longer explicitly spelled out as constraints but rather are folded into matrices H and F, whose construction is elaborated in appendix B. I will not replicate the whole derivation of the dense MPC formulation in this thesis – it may be found for example in textbook [2].

4.3 Implementation in Sk8o

I implemented the MPC as a linux service named `MPC control` running on Odroid N2+. Diagram 4.1 shows how this new service interacts with other existing services on Odroid. Thin arrows with hexagonal labels represent `LCM` messages[3].

[2]As a result of using lifted linear predictors.

[3]The diagram shows the messages as point to point connections for sake of clarity – just to avoid possible confusion, I'd like to note that messages in Lightweight Communications and Marshalling are implemented as broadcasts. Individual services then subscribe only to those

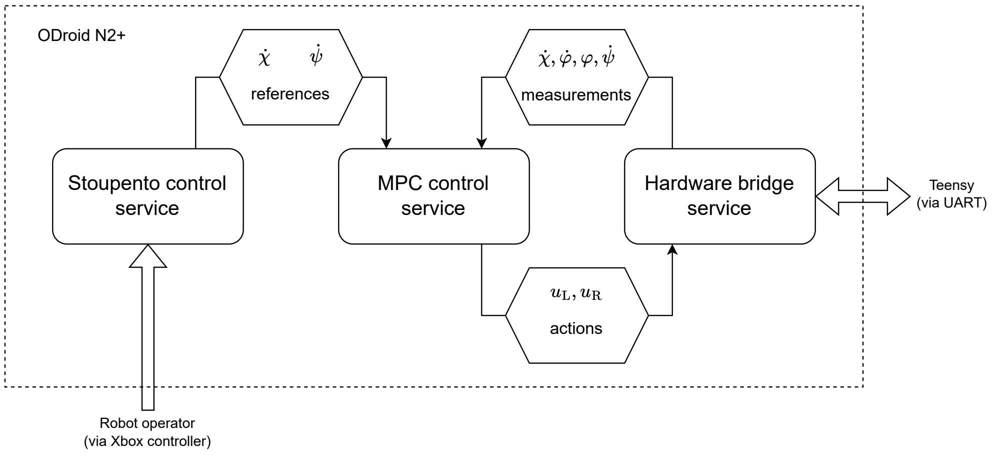
ODroid N2+
$\dot{\chi}$ $\dot{\psi}$
references
$\dot{\chi}, \dot{\varphi}, \varphi, \dot{\psi}$
measurements
Stoupento control service
MPC control service
Hardware bridge service
Teensy (via UART)
u_L, u_R
actions
Robot operator (via Xbox controller)

Figure 4.1: Diagram of information flow between Odroid services

`MPC control` makes use of two threads – one runs at 50 Hz and is responsible for solving the QP (4.6) and publishing the found action u to `LCM`. The second thread concerns itself with incoming `LCM` communications, the most important of which are the measurements of Sk8o's sensors, and velocity references sent by the user. For handling the incoming messages, I reused code originally developed by Dominik Hodan in his master's thesis [6]. The `MPC control` service runs on its own dedicated CPU core, as described in section 3.2.

My `MPC Control` service receives references for forward speed $\dot{\chi}$, and yaw rate $\dot{\psi}$, from an existing `Stoupento control` service[4], which gets them from an Xbox controller held by the robot's operator. In normal operation, the operator inputs their desired values of $\dot{\chi}$, $\dot{\psi}$ by manipulating the joystick on the Xbox controller – I expanded the `Stoupento control` service to also allow the sending of a predefined sequence of references, which I use in further sections to conduct experiments on the robot. The `Stoupento control` service runs in Python.

messages that are of interest to them – represented in the diagram by a thin arrow entering a service block.

[4]The word *Stoupento* in this service's name was an early candidate for the robot's name, before its creators settled on calling it Sk8o. As of the time of writing, the word remains present across the codebase.

The `Hardware bridge` service is written in C++ and handles UART communications with the Teensy board, and by extension, with the actuators and sensors of the robot as described in chapter 2. It too existed prior to my work – I made some minor changes to it, allowing it to pass motor actions (which are published by my `MPC control` service) to Teensy and making it run on a dedicated CPU core. Apart from passing commands to Teensy, `Hardware bridge` also periodically[5] requests measurements from it, which are then sent over to the `MPC Control` service.

There is a slight simplification in the diagram – `Hardware bridge` does not actually publish the states of the Sk8o system; rather, it publishes the measured velocities of the wheels and measurements from the IMU unit. The wheel velocities are then converted inside the `MPC control` service into $\dot{\chi}$, $\dot{\psi}$ (using a known wheel's radius and distance between the wheels). The IMU measurements are converted into $\dot{\varphi}$ and φ. The convertor also handles an offset of four degrees in φ (as the IMU does not read zero when the robot is standing upright). This conversion code is taken from Dominik Hodan's work.

In Hodan's work, the converter did not provide the measurements of χ and ψ. As both of those states are necessary for me, I obtain them by numerically integrating the measured values $\dot{\chi}$ and $\dot{\psi}$, respectively, in `MPC control` service's control loop, as illustrated in figure 4.2.

The references for those states are likewise integrated from $\dot{\chi}_{\text{ref}}$ and $\dot{\psi}_{\text{ref}}$. This leaves me with two undefined references – $\dot{\varphi}_{\text{ref}}$ and φ_{ref}. I chose to always set those references to zero, the rationale being that a perfectly upright position of the robot is best as far as maintaining balance is concerned. A zero φ_{ref} is clearly at odds with nonzero $\dot{\chi}_{\text{ref}}$, but this conflict is present at the very heart of my task, and can be handled by MPC without issue. It simply forces the MPC to make compromises between perfect balance and perfect velocity tracking.

The entity described as `MPC controller` in figure 4.2 is tasked with solving the dense-formulated MPC quadratic program (4.6). It is implemented as a Python object, internally using a *Linear predictor* object. I intentionally separated this

[5]Once per 2 ms.

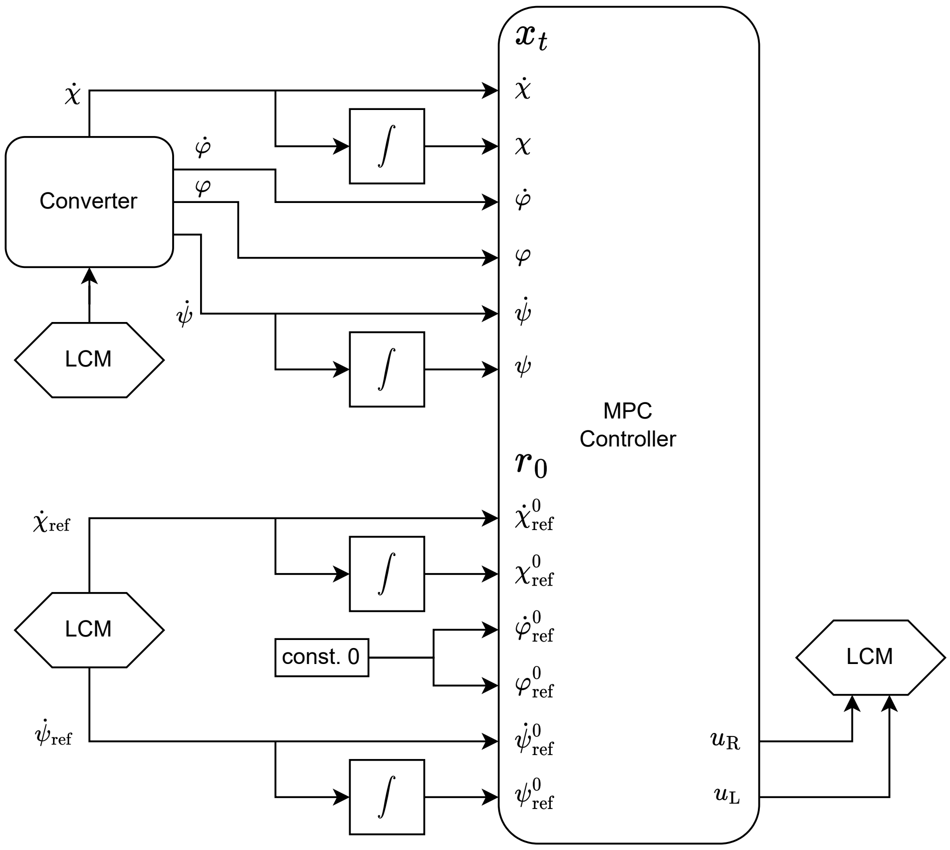

Figure 4.2: Internal diagram of MPC control service

functionality to ease the future progression to lifted linear predictors. For now, it simply implements an unlifted linear predictor as described in 4.1.

4.3.1 MPC Setup

The MPC uses a prediction horizon $N = 20$, as determined in section 3.3, and limits $u_{\min} = -0.5$ N m, $u_{\max} = 0.5$ N m. The limits on actions were determined experimentally. I had started with values of ± 0.7 N m – which are the maximums put in place by the robot's creators – but found that Sk8o's wheels occasionally lost contact with the ground and slipped[6]. Lowering the limits to ± 0.5 N m appeared to solve this issue.

One more detail remains to be addressed – the MPC formulation (4.6) requires a reference r_k for each step of the prediction horizon, and I have so far only shown how the reference for the first step, r_0, is obtained (in figure 4.2). The references for $\dot{\chi}$, $\dot{\psi}$, $\dot{\varphi}$ and φ are kept constant across the whole prediction horizon, while the references for χ and ψ follow a ramp with slope defined by references for their derivatives, i.e.

$$\begin{aligned}
\dot{\chi}^i_{\text{ref}} &= \dot{\chi}^0_{\text{ref}} \,, \\
\dot{\varphi}^i_{\text{ref}} &= 0 \,, \\
\dot{\psi}^i_{\text{ref}} &= \dot{\psi}^0_{\text{ref}} \,, \\
\chi^i_{\text{ref}} &= \chi^0_{\text{ref}} + i \cdot \dot{\chi}^0_{\text{ref}} \,, \\
\varphi^i_{\text{ref}} &= 0 \,, \\
\psi^i_{\text{ref}} &= \psi^0_{\text{ref}} + i \cdot \dot{\psi}^0_{\text{ref}} \,,
\end{aligned} \tag{4.7}$$

where the upper index $i \in 0, 1, \dots N$ marks the step of prediction horizon which the reference belongs to. Note that preview is not used – I assume that no knowledge of the future references is available. This is consistent with my problem statement (section 1.1).

[6] I think this is related to the fact that, not long before my work, Sk8o's tires were replaced with new ones made out of a slightly different material and their design was somewhat altered.

Matrices A, B are obtained by linear approximation of the nonlinear model from section 2.4. The jacobian of the nonlinear state transfer function is calculated, using the equilibrium (all states zero) as operating point. It is then discretized using a sampling time of 20 ms. The resulting matrices are

$$A = \begin{bmatrix} 9.2\cdot 10^{-1} & -1.3\cdot 10^{-3} & 0 & 0 & -7.5\cdot 10^{-1} & 0 \\ 2.8\cdot 10^{-1} & 1.0 & 0 & 0 & 3.2 & 0 \\ 0 & 0 & 9.6\cdot 10^{-1} & 0 & 0 & 0 \\ 1.9\cdot 10^{-2} & 1.2\cdot 10^{-5} & 0 & 1 & -7.6\cdot 10^{-3} & 0 \\ 2.8\cdot 10^{-3} & 2.0\cdot 10^{-2} & 0 & 0 & 1 & 0 \\ 0 & 0 & 1.9\cdot 10^{-2} & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

$$B = \begin{bmatrix} -3.1\cdot 10^{-1} & -3.1\cdot 10^{-1} \\ 1.1 & 1.1 \\ 1.2 & -1.2 \\ -3.2\cdot 10^{-3} & -3.2\cdot 10^{-3} \\ 1.1\cdot 10^{-2} & 1.1\cdot 10^{-2} \\ 1.2\cdot 10^{-2} & -1.2\cdot 10^{-2} \end{bmatrix}. \tag{4.9}$$

And lastly I used the following matrices Q, R:

$$Q = diag([20, 0.237, 0.119, 20, 6000, 1]) \qquad R = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{4.10}$$

where $diag(d)$ denotes a diagonal matrix with elements d on its diagonal. The (experimentally determined) tuning of weighs in Q, particularly the emphasis on state φ, reflects the fact that balancing the robot is the more important goal. Indeed, if balance is lost and the robot falls, the secondary goal of following velocity and yaw rate references cannot be fulfilled in the slightest.

Matrix S is then obtained as the infinite-horizon solution of discrete-time Riccati equation associated with an LQR problem defined by matrices A (4.8), B (4.9), Q, R (4.10). I reproduce it below, replacing elements whose absolute value is $< 10^{-3}$ with zeroes.

$$S = \begin{bmatrix} 5.9 \cdot 10^3 & 1.7 \cdot 10^3 & 0 & 2.5 \cdot 10^3 & 2.3 \cdot 10^3 & 0 \\ 1.7 \cdot 10^3 & 4.9 \cdot 10^2 & 0 & 7.1 \cdot 10^2 & 7.3 \cdot 10^2 & 0 \\ 0 & 0 & 2.8 \cdot 10^{-1} & 0 & 0 & 6.1 \cdot 10^{-1} \\ 2.5 \cdot 10^3 & 7.1 \cdot 10^2 & 0 & 2.5 \cdot 10^3 & 1.2 \cdot 10^3 & 0 \\ 2.3 \cdot 10^3 & 7.3 \cdot 10^2 & 0 & 1.2 \cdot 10^3 & 1.3 \cdot 10^4 & 0 \\ 0 & 0 & 6.1 \cdot 10^{-1} & 0 & 0 & 1.9 \cdot 10^1 \end{bmatrix} \tag{4.11}$$

4.4 Simulations

In this section I reproduce three simulations verifying the capability of MPC to control Sk8o. The simulator uses the model described in section 2.4 and is implemented in Python. The nonlinear system is numerically integrated with an integration step of 2 ms, using the fourth-order Runge-Kutta method. The MPC controller formulated earlier in this chapter controls the system at 50 Hz.

Except for the first simulation, the simulations start with the system in equilibrium, i.e. $x_0 = [0, 0, 0, 0, 0, 0]^T$.

Even though the references used in the experiments are known in advance, I do not use preview in my controllers. This is done to mimick the intended use case of the robot, where references are provided by a human operator.

4.4.1 Stabilization

The first test confirms that the controller is able to bring the system from a nonzero initial condition back to the equilibrium. The initial condition of the simulation is $x_0 = [1, 1, 1, 0, 0, 0]^T$. The simulation is captured in figure 4.3.

4.4.2 There and Back Again

In this simulation, the MPC is given a simple set of references – first, it is tasked with driving the robot forward for two seconds, then it ought to keep it still for three seconds, then it is supposed to drive it backwards for two seconds.

The simulation result is visualized in figure 4.4. I've elected not to plot the states $\dot{\psi}$ and ψ, as they (unsurprisingly) remain zero for the duration of the simulation.

4.4.3 Slalom

This last task consists of a set of references that, if followed perfectly, would see the robot move along a slalom-like trajectory. The slalom consists of eight 180° arcs with common radius of 25 cm. The trajectory also progressively increases in difficulty. The time allotted for passing the first two arcs is 4 s, the second couple must be passed in 3 s, third in 2.25 s and fourth in 1.69 s. The result is visualized in figure 4.5.

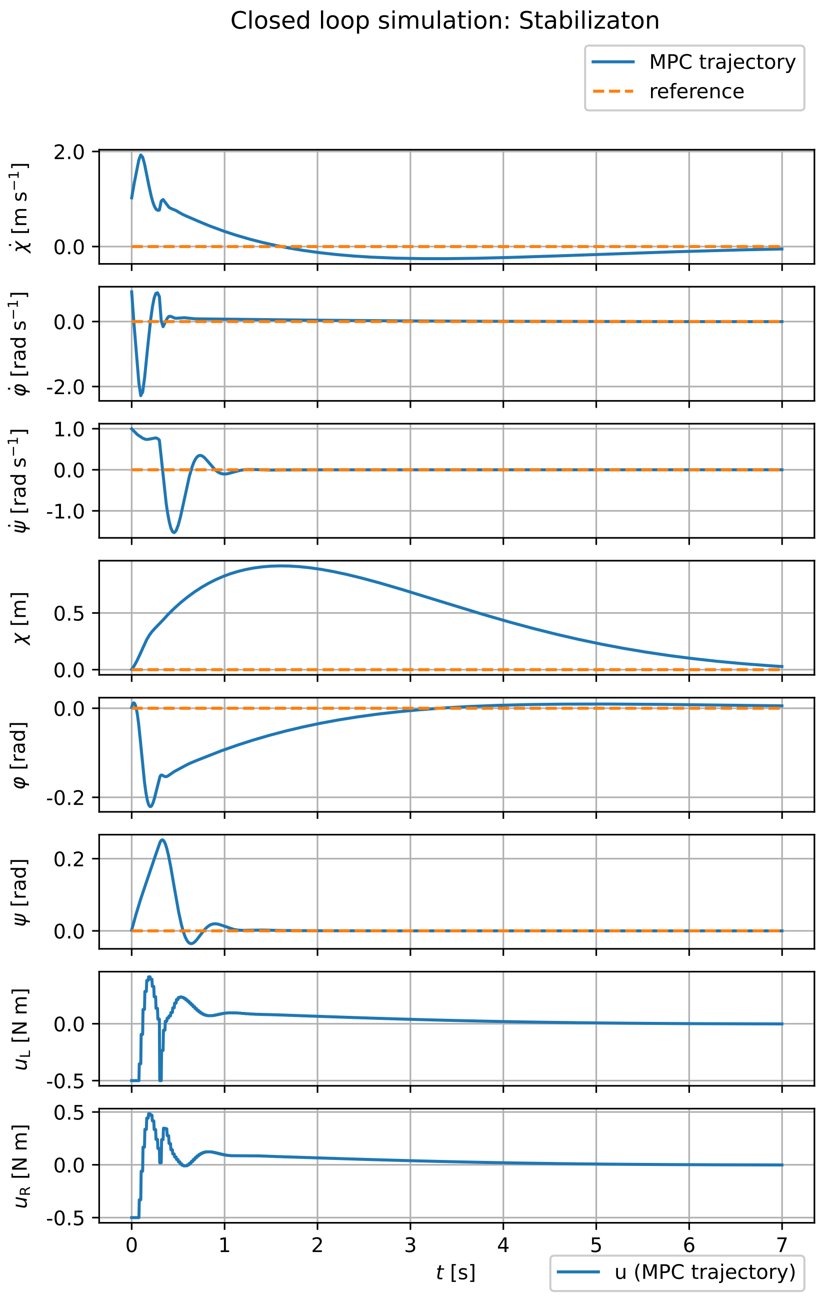

Figure 4.3: MPC simulation: Stabilization

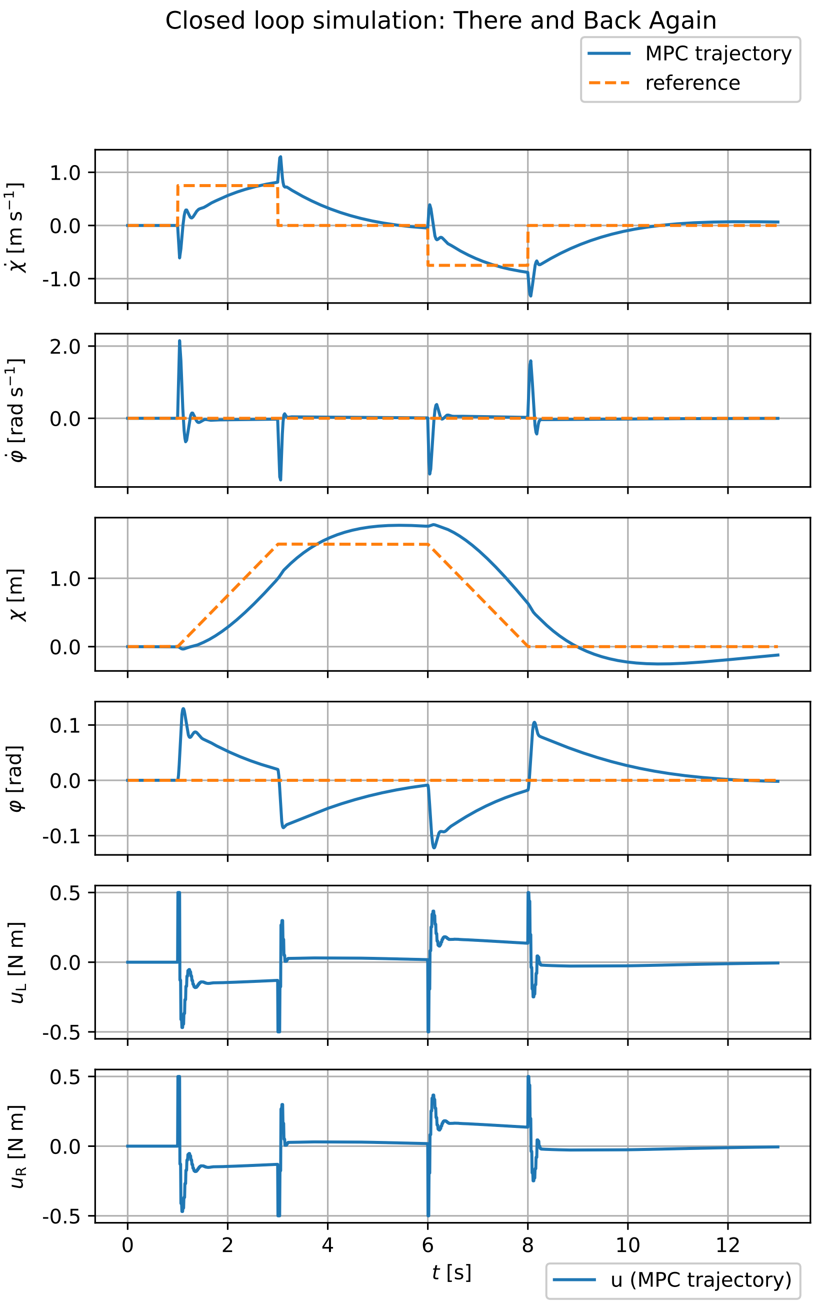

Figure 4.4: MPC simulation: There and Back Again

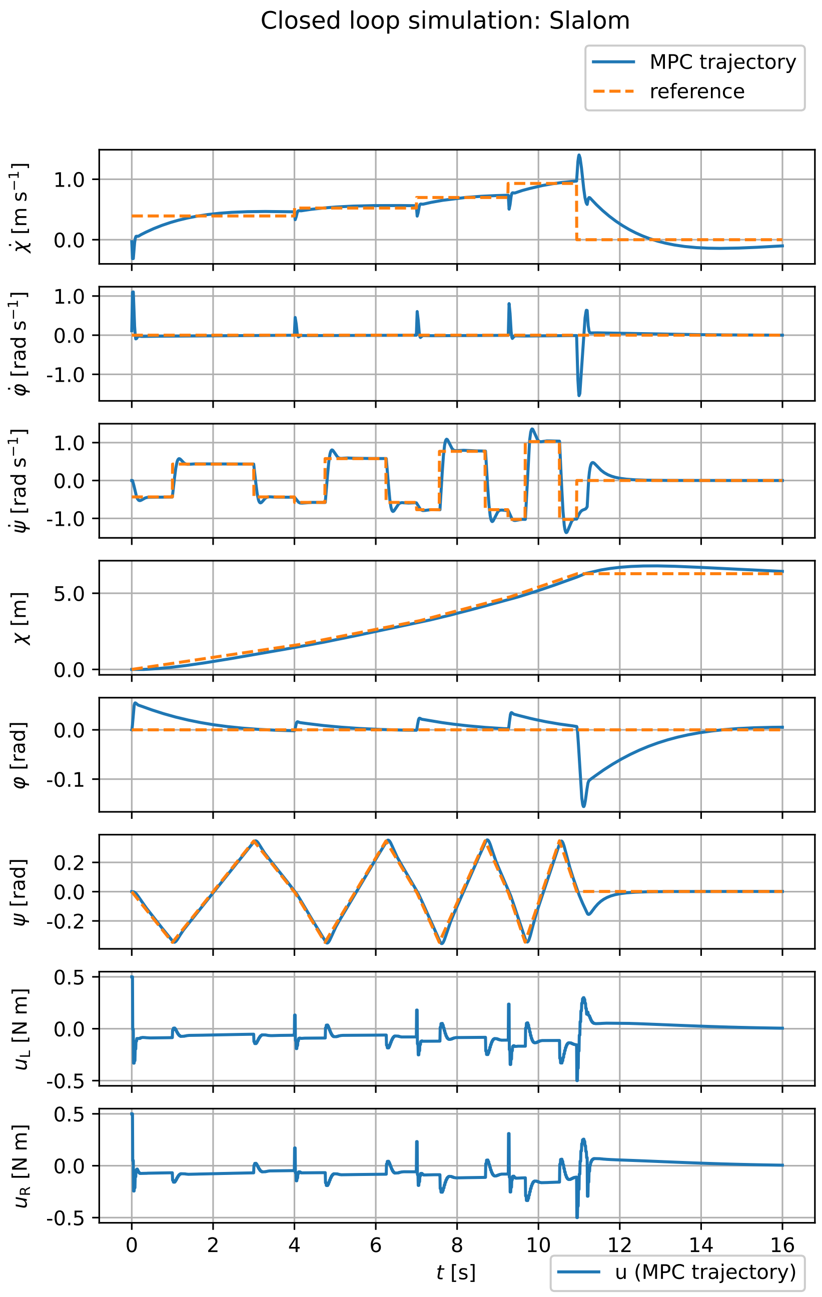

Figure 4.5: MPC simulation: Slalom

4.5 Experiments on Sk8o

This section will show measurements from the physical Sk8o robot. Three experiments were performed – Stabilization, There and Back Again, and Slalom. The last two mentioned experiments are analogous to simulation experiments presented in section 4.4, using the exact same references.

In the Stabilization experiment, the references given to the robot are zero (i.e. it is tasked with balancing in place). An external disturbance is then applied to the robot using a simple contraption which will be described shortly in section 4.5.1.

All experiments were performed once with MPC controlling the robot and then a second time, using the original LQR controller. Both resulting trajectories are overlaid for comparison in figures 4.7, 4.8, and 4.9.

For legibility's sake, I omit the control actions from those figures. Figures with the controls included can be found in appendix D. The appendix version of figure 4.7 is also longer, showing three consecutive disturbances instead of one.

4.5.1 Minimalistic Repeatable Disturbance Applier

To disturb the robot, I devised a simple device, which I'll refer to as the Minimalistic Repeatable Disturbance Applier (MRDA). Its design is captured in figure 4.6. As the name implies, its purpose is to make the disturbances applied to the robot consistent across multiple experiments.

MRDA is essentially a simple pendulum. Specifically, it consists of a PET bottle filled with water (represented by point mass m in the diagram), hung on a plastic twine of length l. Sk8o (drawn in blue) is positioned directly below the pivot point of the pendulum. The bottle is brought to a defined height h_1 and released. It swings down to h_2 and impacts the unsuspecting robot in the back of its body[7]. The values of of parameters used above are listed in table 4.1.

[7]This point of impact is not chosen to make the whole endeavour appear more dastardly. Rather, I picked it to minimize the risk of damaging the robot's electronics, which are positioned closer to the front of its body.

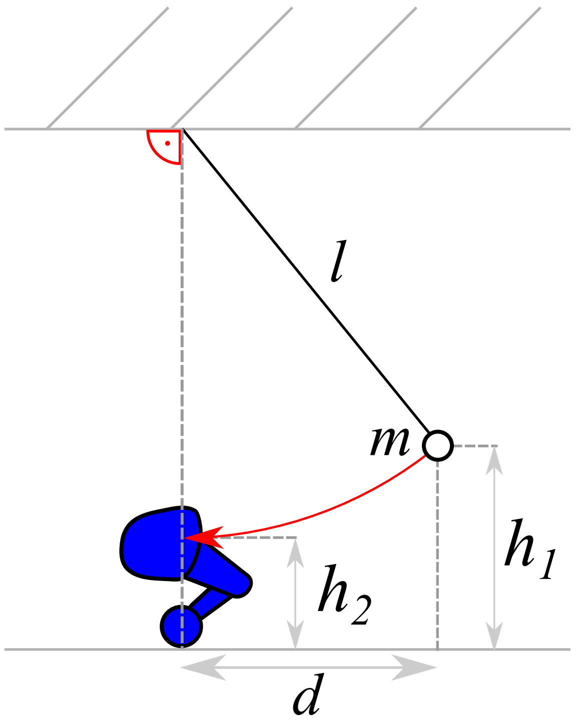

Figure 4.6: Diagram of Minimalistic Repeatable Disturbance Applier

Parameter	Value	
l	460	cm
d	170	cm
h_1	70	cm
h_2	30	cm
m	500	g

Table 4.1: Parameters of Minimalistic Repeatable Disturbance Applier

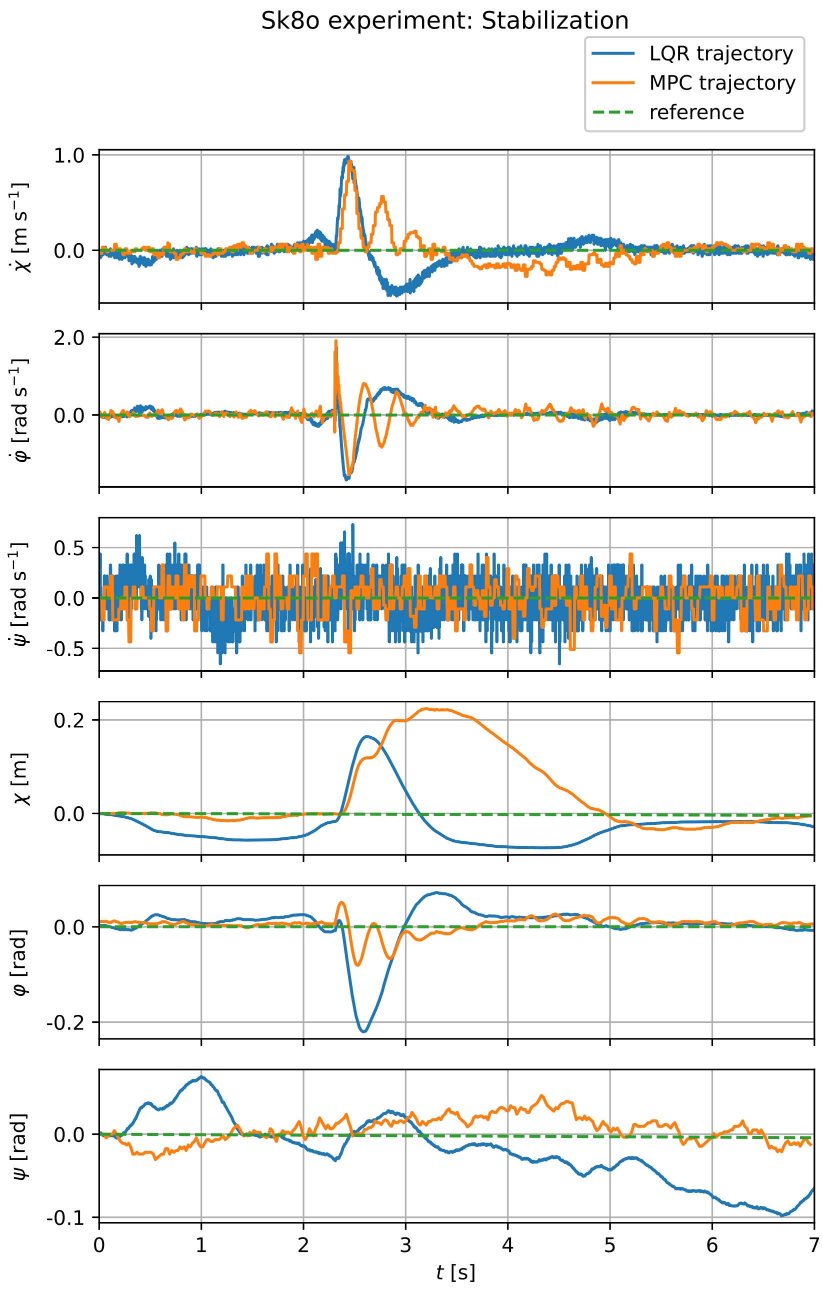

Figure 4.7: Sk8o experiment: Stabilization

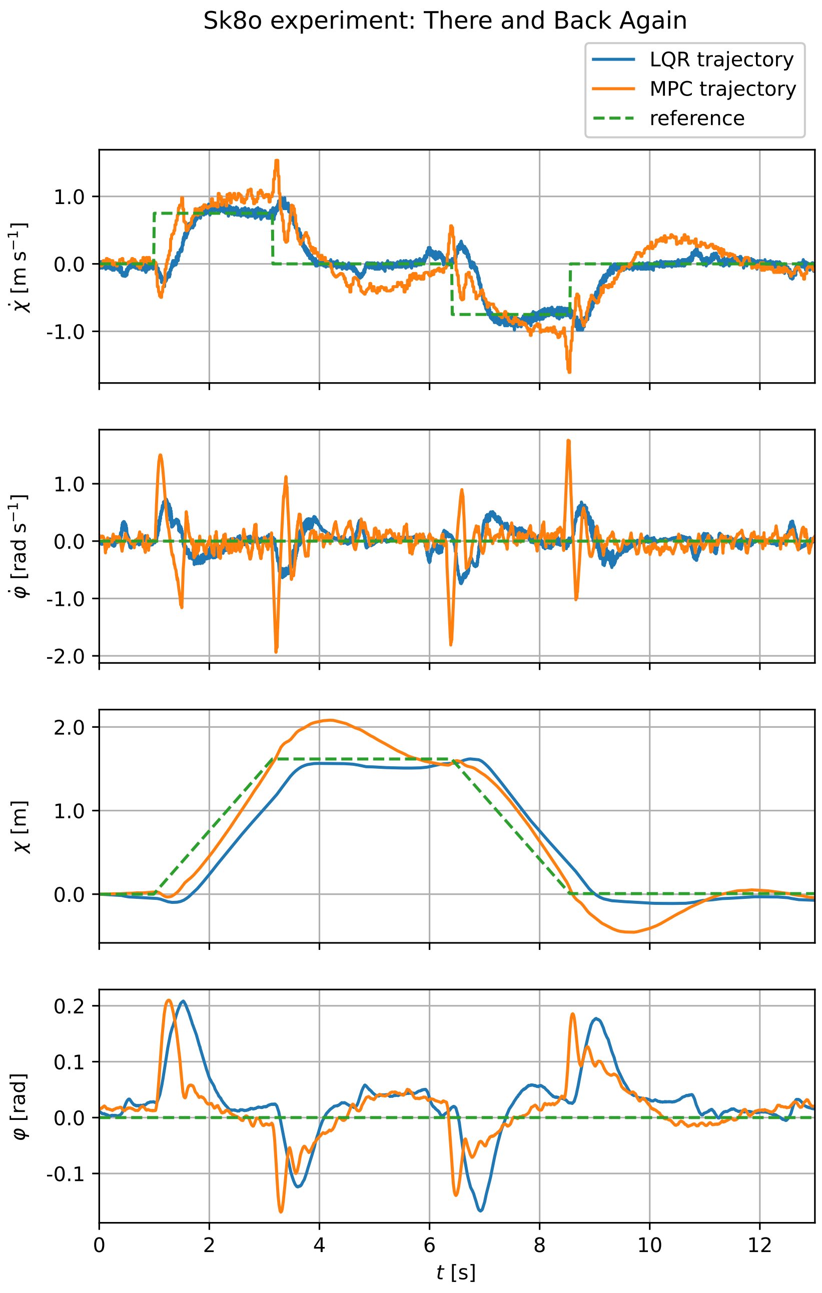

Figure 4.8: Sk8o experiment: There and Back Again

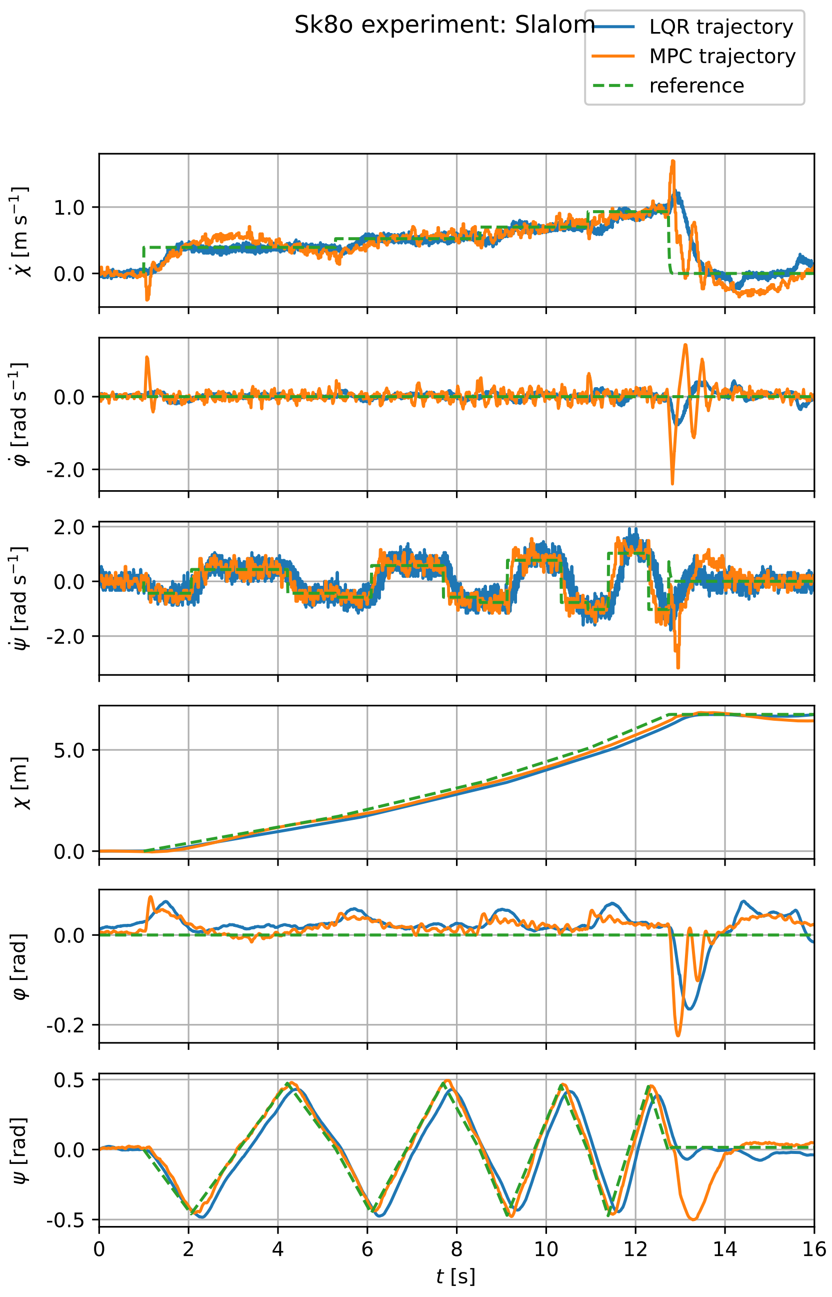

Figure 4.9: Sk8o experiment: Slalom

4.5.2 Discussion of Experimental Results

First and foremost, it is clear from the experiments that the MPC controller I designed and implemented is capable of steering and balancing Sk8o. I've shown that it can track references for both $\dot{\chi}$ and $\dot{\psi}$, and to balance the robot even in the presence of an external disturbance.

I've also compared the trajectories of Sk8o controlled by my MPC controller with trajectories obtained by controlling Sk8o by a preexisting LQR controller. The results show the performance of the two controllers to be comparable. Determining which of the two controllers is *better* is somewhat problematic though, for several reasons.

Firstly, the two controllers are tuned differently. It wasn't clear which matrices Q, R were used to tune the original LQR controller[8]. Even if they were available, it isn't certain that they would perform well in MPC due to differences between the controllers.

Given the different tuning, it isn't at all obvious how to judge the quality of tracking. For example, in figure 4.7, state φ appears to be regulated better by the MPC, while in state χ LQR's trajectory seems superior. This could be caused by the difference in tuning alone.

The two controllers have different action limits – I had to reduce the MPC's action limit to 0.5 N m to avoid slipping, while the LQR's actions are constrained to $0.7\,\mathrm{N\,m}$[9].

The MPC is also disadvantaged by running at a lower frequency (50 Hz compared to the LQR's 1 kHz). A secondary handicap is caused by the fact that it runs on Odroid, unlike the LQR which runs on Teensy. Recalling figure 2.1, Odroid is further away from the robot's actuators and sensors. Thus, the MPC is burdened by greater communications delays than the LQR.

[8]Though some matrices can be found in Adam Kollarčík's master's thesis [7], the controller was apparently subsequently retuned by others. To the best of my knowledge, the matrices used to tune the latest iteration of LQR running on Sk8o are lost to history.

[9]In Teensy, the LQR's actions are clipped to these values, while in my MPC, the limits are formulated as constraints on the optimization problem.

And lastly, to make any meaningful comparison, a statistically significant amount of experiments ought to be conducted. I wasn't able to do this due to time constraints.

To sum up: I deem the MPC regulator fit for use. Its performance is comparable to that of the original LQR.

5 | Koopman Model Predictive Control

Having implemented a working MPC controller in chapter 4, I decided to go beyond the requirements of my assignment and expand the implementation to *Koopman Model Predictive Control* (KMPC).

KMPC works, for the most part, the same as MPC, with one crucial difference: it uses a *lifted* linear predictor. In a nutshell, the lifted linear predictor expands (*lifts*) the state vector into a higher dimension, adding to it nonlinear functions of the states (called *observables*). Operating on this lifted state can allow this predictor to better forecast the evolution of the nonlinear system, which is supported by the *Koopman operator theory* on dynamical systems.

The structure of this chapter reflects the fact that KMPC is basically MPC with a different linear predictor. The majority of the chapter will deal with the problem of identifying a the lifted linear predictor. Section 5.7 will note the few ways in which KMPC's formulation differs from MPC. Just like in chapter 4, I'll conclude by assessing KMPC in simulations and hardware experiments (sections 5.8 and 5.9, respectively).

5.1 Koopman Operator

What follows is a very brief introduction of the theory upon which lifted linear predictors are based. The idea of representing the behaviour of nonlinear systems in a lifted space was first introduced in [8] by Koopman in the 1930s. For a rigorous discussion of Koopman operator theory, see [3]. For the sake of brevity, I will also limit myself to autonomous systems – the extension to controlled systems can be found in [9].

Consider an autonomous, discrete-time dynamical system

$$x_{k+1} = \tau(x_k), \qquad x_k \in \mathcal{M}, \tag{5.1}$$

where $\mathcal{M}$ is a finite-dimensional state space and $\tau : \mathcal{M} \mapsto \mathcal{M}$ is a nonlinear function. The first step is to define new functions in the form $g : \mathcal{M} \mapsto \mathbb{R}$, which are called *observables*. Observables belong to an infinite-dimensional vector space of functions $\mathcal{F}$. Note that observables need not be linear.

The *Koopman operator* $\mathcal{K} : \mathcal{F} \mapsto \mathcal{F}$ is defined as an operator that advances observables one step into the future. In other words, $\mathcal{K}$ is defined by the equation

$$(\mathcal{K}g)(x_k) = g(\tau(x_k)) = g(x_{k+1}). \tag{5.2}$$

Crucially, the Koopman operator itself is linear. Assuming that $\mathcal{F}$ contains functions mapping the state x to its components, $\mathcal{K}$ captures the dynamics of (5.1) in their entirety. This would make the Koopman operator a perfect candidate for a linear predictor. Alas, it is also infinite-dimensional, so it cannot be directly used for control synthesis.

Not all is lost, however – by taking a finite-dimensional subset of $\mathcal{F}$, it is possible to obtain a finite-dimensional approximation of $\mathcal{K}$. And this approximation can be used to build a lifted linear predictor [9].

5.2 Lifted Linear Predictor

Just like in section 4.1, the goal is to predict the evolution of a discrete-time, nonlinear system defined by

$$x_{k+1} = \tau(x_k, u_k)\ . \tag{(4.1) revisited}$$

This will once again be achieved by a linear predictor in the form

$$\begin{aligned} z_{k+1} &= Az_k + Bu_k \ , \\ \hat{x}_k &= Cz_k \ . \end{aligned} \qquad \text{((4.2) revisited)}$$

The difference here is that the dimension of z is no longer equal to that of x. The *lifted state* z of the lifted linear predictor is defined as

$$z = \mathcal{G}(x) = \begin{bmatrix} g_0(x) \\ g_1(x) \\ \vdots \\ g_{L+5}(x) \end{bmatrix} , \tag{5.3}$$

where $g_i(x)$ are observables, as defined in section 5.1, and $\mathcal{G}(x)$ is the *lifting function*. L shall be called the *lifting dimension* and denotes the number of observables in excess of six. I choose to fix the first six observables as elements of x, i.e.

$$\begin{aligned} g_0(x) &= \dot{\chi} \ , \\ g_1(x) &= \dot{\varphi} \ , \\ g_2(x) &= \dot{\psi} \ , \\ g_3(x) &= \chi \ , \\ g_4(x) &= \varphi \ , \\ g_5(x) &= \psi \ . \end{aligned} \tag{5.4}$$

The lifting dimension and the choice of functions $g_i(x)$ for $i > 5$ are design parameters of the lifted linear predictor.

Once the observables have been chosen, matrices A, B of (4.2) can be identified in a data-driven fashion using the *Extended dynamic mode decomposition* algorithm, which I introduce in section 5.3. Matrix C is trivial to obtain due to the choice of the first six observables – it takes the form of a zero-padded identity matrix. The matrices A, B, C, and function $\mathcal{G}(x)$ together constitute a lifted linear predictor.

5.3 Extended Dynamic Mode Decomposition

The Extended Dynamic Mode Decomposition (EDMD) algorithm, introduced in [12], is the tool that facilitates identification of linear predictors from trajectory data. I use a variant of EDMD, described in [9], which is adapted for use in controlled systems. If one desired to apply the algorithm to a non-lifted linear predictor, they could easily do so – all that is needed is to view the predictor as a lifted one, but with a trivial lifting function $\mathcal{G}(x) = x$.

Let's assume that a set of trajectory data of the nonlinear system defined by equation (4.1) is available. It could be obtained as measurements of the physical system, or it could have originated from simulations. The data is organised into triplets of x_i, x_i^+, and u_i so that

$$x_i^+ = \tau(x_i, u_i) \ , \qquad i = 0, 1, ...D \ , \tag{5.5}$$

where D is the number of data points and $\tau(x, u)$ is the state transfer function of the nonlinear system. It is not necessary for all the data points to have originated from a single trajectory, but equation (5.5) must be satisfied.

The data is aranged into matrices and the states x_i, x_i^+ and lifted, obtaining matrices

$$\begin{aligned} X_{\text{lift}} &= [\mathcal{G}(x_0), \mathcal{G}(x_1), ..., \mathcal{G}(x_D)] \ , \\ X_{\text{lift}}^+ &= [\mathcal{G}(x_0^+), \mathcal{G}(y_1^+), ..., \mathcal{G}(x_D^+)] \ , \\ U &= [u_0, u_1, ...u_D] \ . \end{aligned} \tag{5.6}$$

EDMD then solves the optimization problem

$$\min_{A,B} ||X_{\text{lift}}^+ - AX_{\text{lift}} - BU||_F, \tag{5.7}$$

with $||\cdot||_F$ representing the Frobenius norm. Denoting the Moore-Penrose pseudoinverse of a matrix as $(\cdot^\dagger)$, the analytical solution to 5.7 is

$$[A, B] = X^+_{\text{lift}}[X_{\text{lift}}, U]^\dagger \ . \tag{5.8}$$

5.4 Obtaining the Data for EDMD

The EDMD algorithm requires trajectory data of the nonlinear system to be predicted. This section will describe how I obtained said data. At the section's end, I mention some approaches that did not work.

5.4.1 Closed-loop Simulation

I generated the trajectory data by simulating the nonlinear Sk8o system described in section 2.4, with a stabilizing controller added. The system is numerically integrated using the fourth-order *Runge-Kutta* method with integration step $\delta t = 2\,\text{ms}$. It is controlled with control period $T = 20\,\text{ms}$, which corresponds to the control frequency used by MPC running on the real robot (the controllers used will be discussed later in this section).

The resulting simulation is resampled at frequency $f_s = T^{-1}$. Multiple simulated trajectories are obtained in the form of $x_i^{(s)}$, $u_i^{(s)}$, $i = 0, \ldots N^{(s)}$, where (s) denotes the index of the simulated trajectory and where $N^{(s)}$ is the number of samples in simulated trajectory (s). Each trajectory is simulated for up to $0.5\,\text{s}$, which means that the maximum value of $N^{(s)}$ is 25. Some simulations are terminated sooner, as will be described below, resulting in $N^{(s)} < 25$ for some values of (s).

I arrange the obtained samples (from all trajectories) into matrices X, X^+

$$\begin{aligned}
X &= \begin{bmatrix} x_0^{(0)} & x_1^{(0)} & \ldots & x_{N^{(0)}-1}^{(0)} & x_0^{(1)} & \ldots & x_{N^{(1)}-1}^{(1)} & x_0^{(2)} & \ldots \end{bmatrix} , \\
X^+ &= \begin{bmatrix} x_1^{(0)} & x_2^{(0)} & \ldots & x_{N^{(0)}}^{(0)} & x_1^{(1)} & \ldots & x_{N^{(1)}}^{(1)} & x_1^{(2)} & \ldots \end{bmatrix} , \\
U &= \begin{bmatrix} u_0^{(0)} & u_1^{(0)} & \ldots & u_{N^{(0)}-1}^{(0)} & u_0^{(1)} & \ldots & u_{N^{(1)}-1}^{(1)} & u_0^{(2)} & \ldots \end{bmatrix} ,
\end{aligned} \tag{5.9}$$

which are then lifted, obtaining $X_{\text{lift}} = \mathcal{G}(X)$, $X^+_{\text{lift}} = \mathcal{G}(X^+)$. Thus I arrive at the matrices required by EDMD.

To introduce variability into the data sets, I initialize each simulation (s) with a random initial condition $x_0^{(s)}$. The components of $x_0^{(s)}$ are picked by sampling uniform distributions with following (experimentally found) limits:

$$\begin{aligned} |\dot{\chi}_0^{(s)}| &\leq 0.5 \ , \\ |\dot{\varphi}_0^{(s)}| &\leq 0.1 \ , \\ |\dot{\psi}_0^{(s)}| &\leq 0.05 \ , \\ |\chi_0^{(s)}| &\leq 0.5 \ , \\ |\varphi_0^{(s)}| &\leq \frac{\pi}{8} \ , \\ |\psi_0^{(s)}| &\leq 0.05 \ . \end{aligned} \tag{5.10}$$

In addition to constraining the initial condition of the simulation, in each simulation step i the following condition is checked:

$$|\varphi_i^{(s)}| \leq \frac{\pi}{4} \tag{5.11}$$

If condition (5.11) is violated, the simulation of that trajectory is terminated. This is done for two reasons – firstly, to deemphasize irrecoverable falls in the data[1], and secondly to stay within the interval where the nonlinear model is valid. The real Sk8o is a hybrid system in the sense that when $|\varphi| \approx \frac{\pi}{2}$, an impact of the robot's body with the floor occurs. The model does not take this into account.

I simulated two batches of data, each consisting of multiple trajectories. The first set, which I'll refer to as the *identification set*, totalled 3000 s of simulations, while the second, *validation set*, contained trajectories totalling 500 s.

The last thing that remains to be discussed regarding the simulation are the controllers used in it. For each trajectory, I generate a new controller based on

[1]I.e., not to train the predictors too much in forecasting the future in situations that the robot can't handle. If $|\varphi|$ is over 45°, the result is most likely going to be a loss of balance and fall. I'd rather have a predictor that works better when the robot is upright and worse when failure is inevitable, than one that perfectly forecasts its fall towards the floor at the cost of worsening its performance during normal operation.

one of two baseline controllers – either a LQR or an MPC with unlifted predictor, as described in chapter 4. The controllers are given zero references on all states. All the controllers use the same matrix $Q^{(s)}$ to penalize the states:

$$Q^{(s)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 200 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 \end{bmatrix} \forall (s) \; . \tag{5.12}$$

The matrix penalizing actions is randomly generated for each controller to make them distinct. Two variants are used:

$$R^{(s)}_{\alpha} = \begin{bmatrix} 50\rho_0 & 0 \\ 0 & 50\rho_0 \end{bmatrix} \qquad R^{(s)}_{\beta} = \begin{bmatrix} 50\rho_1 & 0 \\ 0 & 50\rho_2 \end{bmatrix} , \tag{5.13}$$

where ρ_0, ρ_1, ρ_2 are obtained by sampling a uniform distribution $(0, 1]$. In this way, controllers with equal penalties on both wheels and controllers with differing penalties are generated.

To further enhance the data, I sabotage the used controllers by adding random pertrubations to their outputs. The actions $u_i^{(s)}$ are obtained as

$$u_i^{(s)} = u_i^{\mathrm{ctrl}(s)} + u_i^{\mathrm{nrand}(s)} , \tag{5.14}$$

where $u_i^{\mathrm{ctrl}(s)}$ is the output of the controller used in simulated trajectory (s) and $u_i^{\mathrm{nrand}(s)}$ is a random vector whose components are obtained by sampling a normal distribution parametrized by mean value $\mu = 0$ and standard deviation $\sigma = 0.1$.

5.4.2 Unsuccessful Approaches

Obtaining usable data proved to be one of the major challenges in harnessing EDMD. Before arriving at using closed-loop data as described in section 5.4.1, I had tried several other approaches, some of which are listed in this section.

I've experimented with generating open-loop data. The method was similar to that described above, except that instead of u_i^{ctrl} being generated by feedback controllers, it would be obtained from a variety of functions – including sines and square waves of varying frequencies, amplitudes and offsets, constants and others. Using open-loop data in the identification and validation sets appeared to worsen the performance[2] of found linear predictors. Mixing it with closed loop data did not seem to help either – the best results were achieved when the proportion of open loop data in the mix reached zero.

Simulating trajectories that were significantly longer than 0.5 s or that used a lower magnitude of initial conditions likewise led to poor results. I think this was caused by the data insufficiently exploring the state-space. In the case of long (5 s or longer) trajectories, I think the issue was that the controllers brought the system close to equilibrium too quickly, and most of the data was thus taken up by the system being almost at rest, disturbed only by the perturbations I added to the controller's output.

I also attempted to use data gathered from the real robot, by driving it around with the MPC controller found in chapter 4 and logging the actions and state measurements. This proved unusable – linear predictors identified on data from the real robot performed horribly, often predicting that the system will remain in its current state even if the robot was about to fall. I suspect the issue was similar to that described above – insufficient exploration of the state space. While this could perhaps be remedied by driving the robot more recklessly[3] , I elected not to go down that path, not wishing to damage Sk8o.

5.5 Evaluation Metric

Before I proceed to finding the lifted linear predictors, let me define how I evaluated the predictors' performance. It was necessary to introduce a metric, since many different lifting functions $\mathcal{G}(x)$ and with them many different linear predictors can

[2]By performance I mean the predictor's ability to forecast the evolution of the nonlinear system – how I evaluated it will be described in selection 5.5.

[3]I.e. supplying faster-varying references of greater magnitude, leading to greater perturbations of the controller's output.

be devised – thus a need for comparing their performance arises. The metric I use is based on *normalized root mean square error* (NRMSE).

To calculate the metric, I use a set of simulated trajectories – the 500 s validation set mentioned in section 5.4.1. Given a linear predictor consisting of matrices A, B, C, and a lifting function $\mathcal{G}(x)$, as described in section 5.2, and given a simulated trajectory $x_i^{(s)}$, $u_i^{(s)}$, $i = 0, 1, ..., N^{(s)}$, the first step is to calculate the *predicted trajectory* in the following fashion:

$$\begin{aligned} z_0 &= \mathcal{G}(x_0^{(s)}) \ , \\ \hat{x}_i &= C z_i \ , \\ z_{i+1} &= A\mathcal{G}(\hat{x}_i) + B u_i^{(s)} \ , \end{aligned} \tag{5.15}$$

where $\hat{x}_i = [\hat{\dot{\chi}}_i, \hat{\dot{\varphi}}_i, \hat{\dot{\psi}}_i, \hat{\chi}_i, \hat{\varphi}_i, \hat{\psi}_i]^T$ is the predicted state of the system in step i of the trajectory, and z is the state of the linear predictor. Then, NRMSE of the predicted trajectory is calculated on a per-state basis as

$$\begin{aligned} \text{NRMSE}(\dot{\chi})^{(s)} &= \frac{||(\dot{\chi}_0^{(s)}, \dot{\chi}_1^{(s)}, ...\dot{\chi}_{N^{(s)}}^{(s)}) - (\hat{\dot{\chi}}_0, \hat{\dot{\chi}}_1, ...\hat{\dot{\chi}}_{N^{(s)}})||}{||(\dot{\chi}_0^{(s)}, \dot{\chi}_1^{(s)}, ...\dot{\chi}_N^{(s)}) - \mu(\dot{\chi}_i^{(s)})||} \ , \\ &\vdots \\ \text{NRMSE}(\psi)^{(s)} &= \frac{||(\psi_0^{(s)}, \psi_1^{(s)}, ...\psi_{N^{(s)}}^{(s)}) - (\hat{\psi}_0, \psi_1, ...\hat{\psi}_{N^{(s)}})||}{||(\psi_0^{(s)}, \psi_1^{(s)}, ...\psi_{N^{(s)}}^{(s)}) - \mu(\psi_i^{(s)})||} \ , \end{aligned} \tag{5.16}$$

where $\mu(\cdot)$ signifies the mean of the argument over $i = 0, ...N^{(s)}$.

The calculation is repeated for all trajectories in the validation set. Then, the final value of the NRMSE metric of the linear predictor is obtained by summing all NRMSE($\cdot$) results across all states and all trajectories.

I've also divided the resulting sum by the number of trajectories in validation set.

This is mentioned only for completeness' sake and serves no purpose in the end[4].

The metric is formulated for a lifted linear predictor but may be equally applied to an unlifted linear predictor by setting $\mathcal{G}(x) = x$.

5.6 Finding a Lifted Linear Predictor

Now, equipped with a definition of the lifted linear predictor (from section 5.2), EDMD (from section 5.3), data to use in EDMD (from section 5.4.1) and a metric to compare the quality of linear predictors (from section 5.5), it is finally time to identify a lifted linear predictor.

First however, a set of observables must be chosen. Picking the right observables (beyond the first six, which I have fixed as components of the state vector of the nonlinear system (4.1)) is quite challenging, as the options are by definition infinite, and their selection can drastically affect the quality of predictions produced. A few examples will follow, showing how this task was tackled in previous works.

One can start by exploiting their knowledge of the nonlinear system's dynamics. For example, when seeking to predict the evolution of a pendulum with angle θ, the observable $\sin(\theta)$ presents itself as a promising candidate. This method was used in [4] to forecast the states of an array of coupled pendulums or in [10] to predict the dynamics of a power grid. Another option is to generate a family of parametric functions, as in [9], where a hundred thin plate spline radial basis functions are employed to anticipate the behaviour of forced Van der Pol oscillator. *Delay embeddings*, used in [1] or in [5], provide yet another possibility – having values from previous time steps act as observables.

I decided to take inspiration from all of the papers referenced above. I experimented with various choices of observables, some of which I describe below. In describing them, I only note the *added* observables – the first six are always assumed to be the original Sk8o states.

[4] I added the division when I anticipated that I might compare metrics calculated on different validation data sets.

- Observables found as terms of analytical solution of the nonlinear model, described in section 2.4. To avoid clutter, those are reproduced in appendix C.
- The humble constant. Here only a single observable which maps the state to a constant $g(x) = 1$ is used.
- Thin spline radial basis functions. Inspired by [9], these observables take the form $g(x) = |\alpha(x) - \alpha_0|^2 \log(|\alpha(x) - \alpha_0|)$, where $\alpha(x)$ is one of the components of x and $\alpha_0 \in \mathbb{R}$ is a parameter. Various amounts of splines with various parameters were tested.
- Using no observables at all. This essentially identifies an unlifted linear predictor, using a data-driven approach instead of the linear approximation used in section 4.1.

I've also experimented with adding delay embeddings to the sets of observables mentioned above, but their addition seemed to consistently worsen the predictors' performance. Therefore, I will not discuss them further.

Selected linear predictors are ranked by their NRMSE metric in table 5.1. The table also lists the lifting dimension L, defined previously as the number of observables in excess of six. For the sake of comparison, I also evaluated the unlifted linear predictor from section 4.1 and included it in the table.

NRMSE	**L**	**Description of observables**
3.806	19	Solution of nonlinear model
6.126	0	No observables (unlifted predictor via EDMD)
6.127	1	Constant observable ($g(x) = 1$)
6.186	250	Splines
6.498	150	Splines
6.991	100	Splines
8.674	50	Splines
26.290	0	No observables (predictor with matrices from eqs. (4.8), (4.9))

Table 5.1: NRMSE values of selected linear predictors

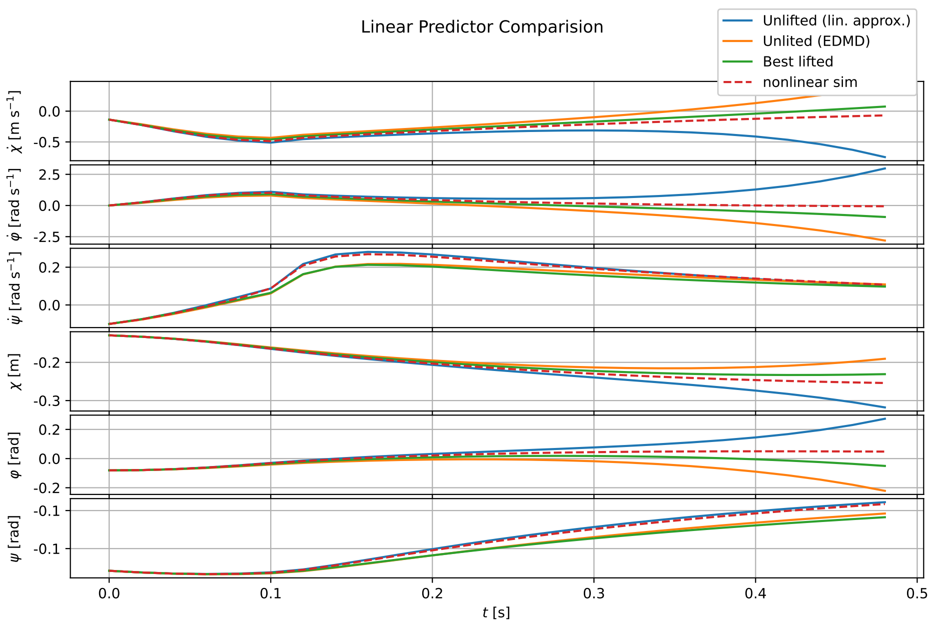

Figure 5.1: Predictions of linear predictors compared to simulated trajectory

Just for illustration, figure 5.1 shows the predictions of 3 linear predictors on a sample simulated trajectory[5]. The predictions shown come from the best predictor according to the evaluation metric, from an unlifted predictor using matrices obtained by linear approximation, and from an unlifted predictor found by EDMD.

[5]Generated without perturbations on controller's output.

5.7 Difference in Formulation and Implementation of KMPC

The KMPC formulation

$$\begin{aligned}
\min_{U} \quad & \frac{1}{2}U^T H U + \begin{bmatrix} z_0^T & r^T \end{bmatrix} F^T U \ , \\
\text{subject to} \quad & u_{\text{min}} \leq u_k \leq u_{\text{max}} \ , \\
\text{parametrized by} \quad & x_0 = \text{given} \ , \\
& z_0 = \mathcal{G}(x_0) \ , \\
& r_k = \text{given} \ , \\
& k = 0, 1, ..., N \ ,
\end{aligned} \tag{5.17}$$

differs from the MPC formulation (4.6) only in the initialization of z_0. This is also the only point where the implementation differs – in KMPC, in each loop, the lifting function $\mathcal{G}(x)$ is evaluated.

Apart from this, the KMPC controller differs from the MPC controller only in the linear predictor it uses. The matrices Q, R are the same as those I used in the MPC controller (see (4.10)).

5.8 Simulations

The simulations used for verifying the KMPC controller are the same as the ones used for verifying MPC in section 4.4.

The meanings of the legends are the following:

- MPC (lin) – MPC using unlifted linear predictor found by linearization, from chapter 4.
- MPC (dd) – MPC using the second best predictor (according to NRMSE metric) found in section 5.2 – the unlifted linear predictor obtained via EDMD.
- KMPC – MPC using the best predictor (according to NRMSE metric) found in section 5.2 – the lifted predictor with observables from analytical solution of the nonlinear model.

The simulation results are visualized in figures 5.2, 5.3 and 5.4.

5.9 Experiments on Sk8o

The experiments too are exactly the same as those performed in section 4.5.

They are reproduced in figures 5.5, 5.6, 5.7,5.8, with actions omitted for readability. I've also decided to split the slalom plot in twain, as it is becoming fairly crowded. The full plots may be found in appendix D.

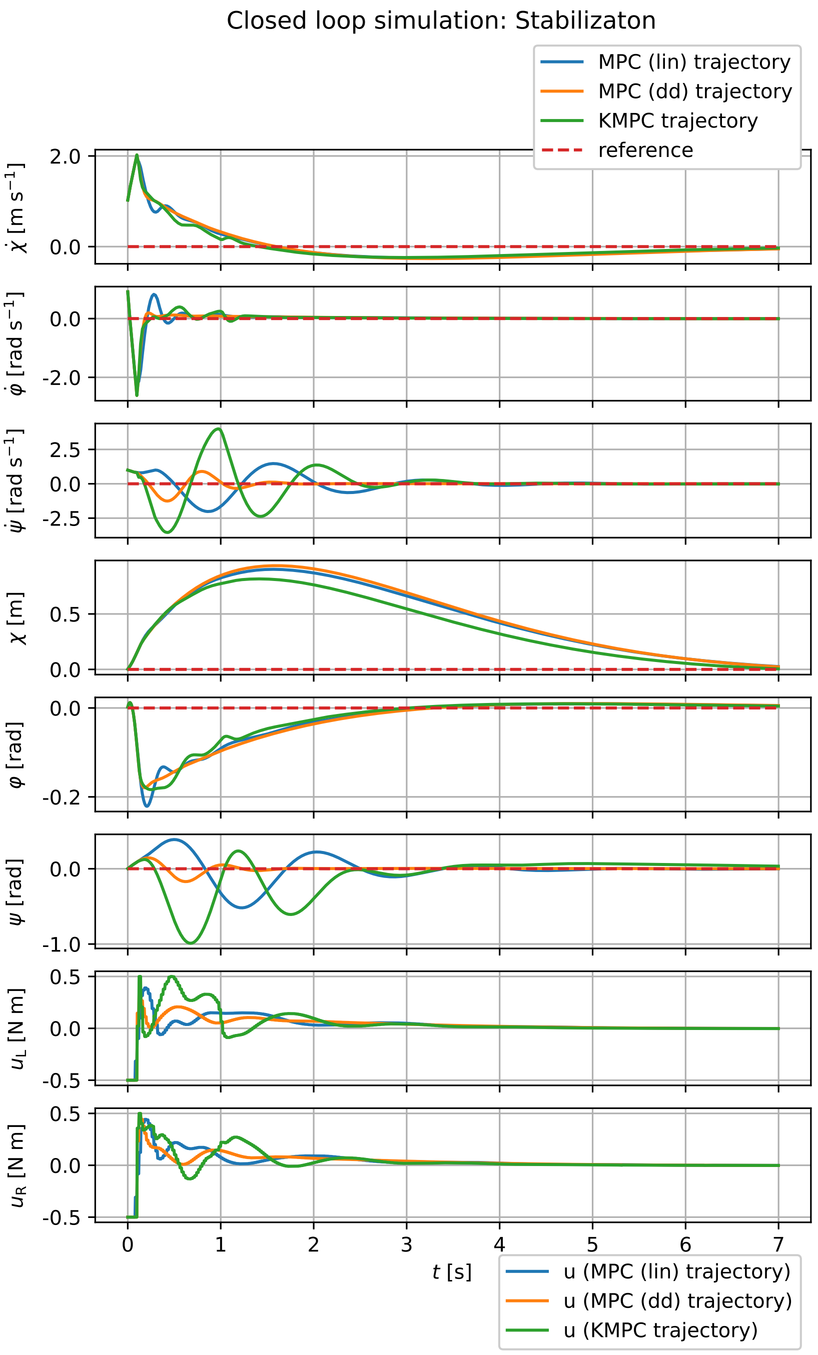

Figure 5.2: Simulation: Stabilization

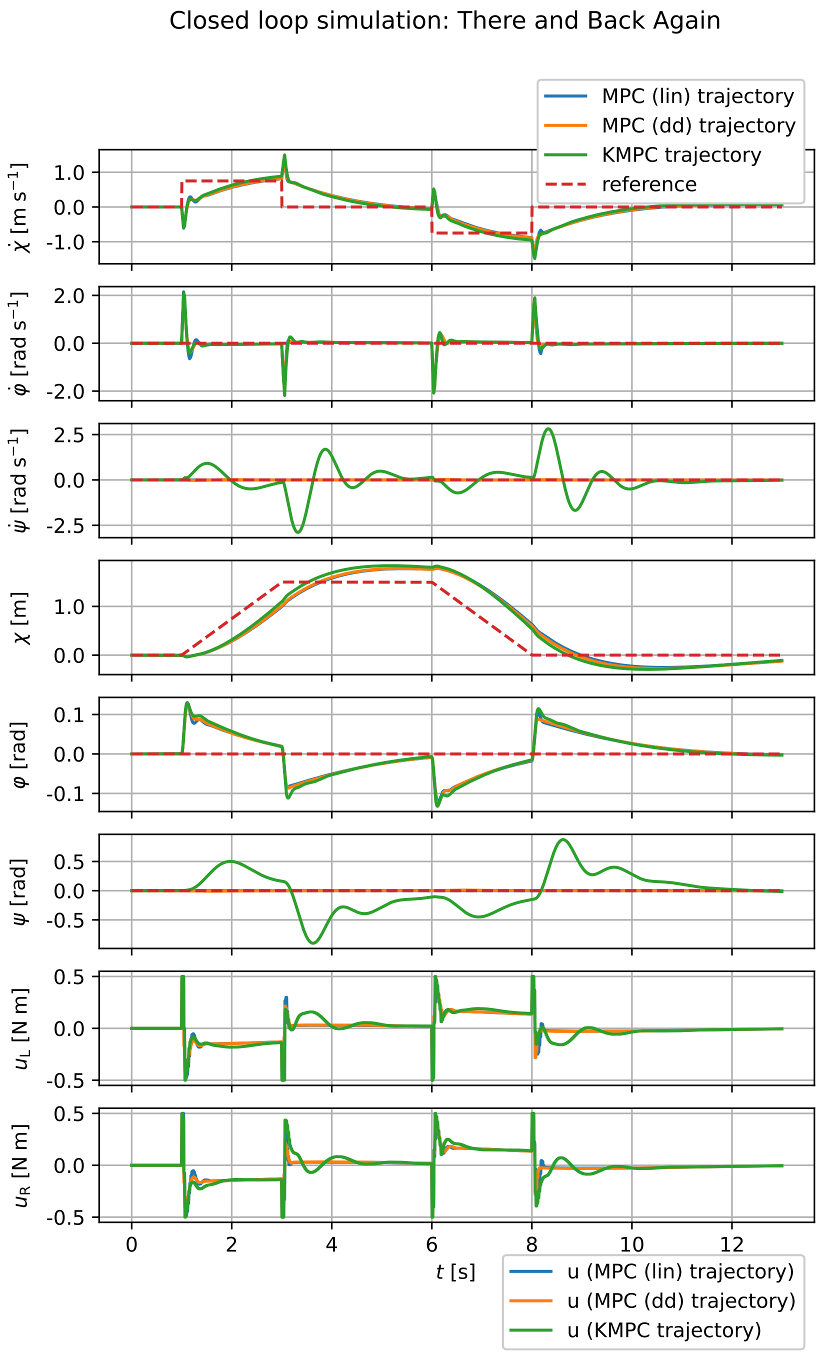

Figure 5.3: Simulation: There and Back Again

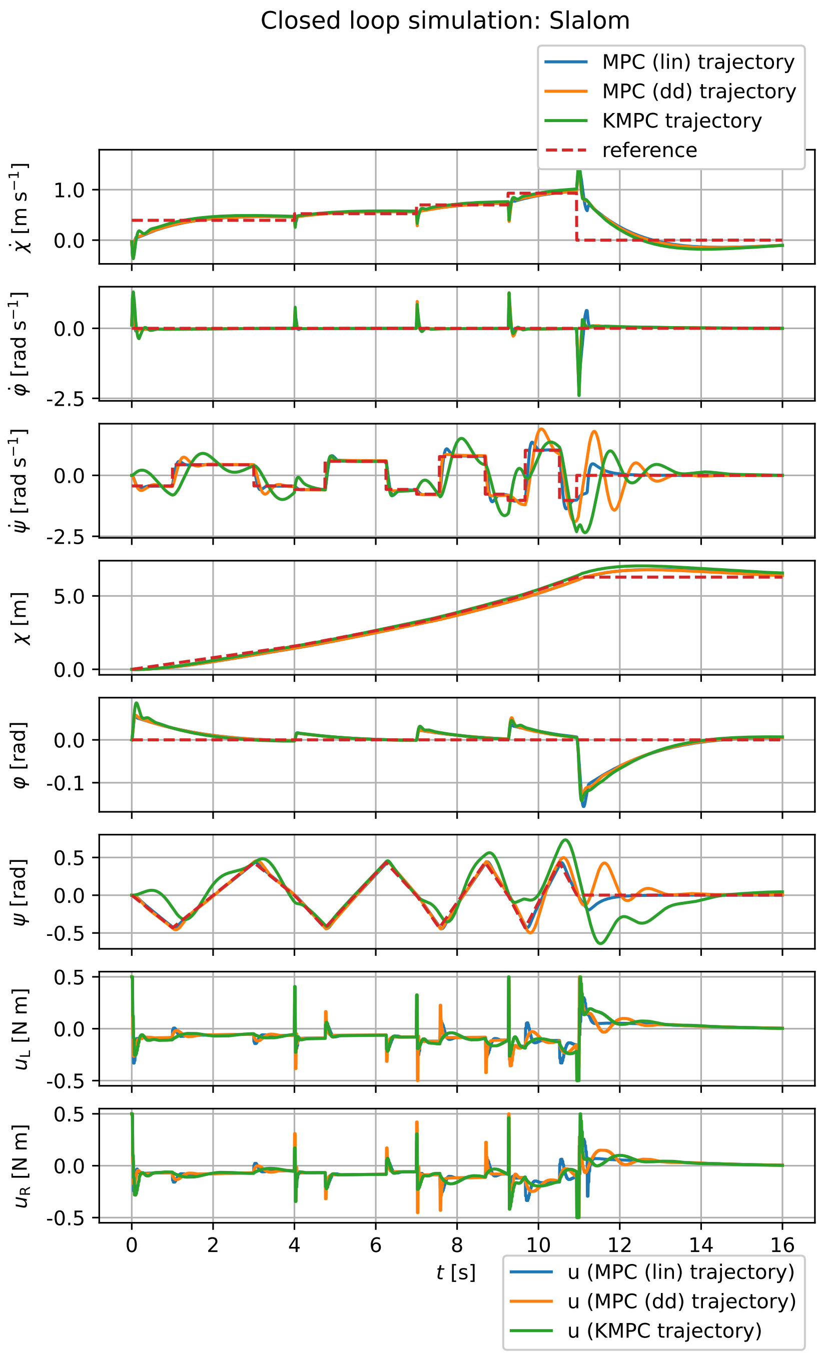

Figure 5.4: Simulation: Slalom

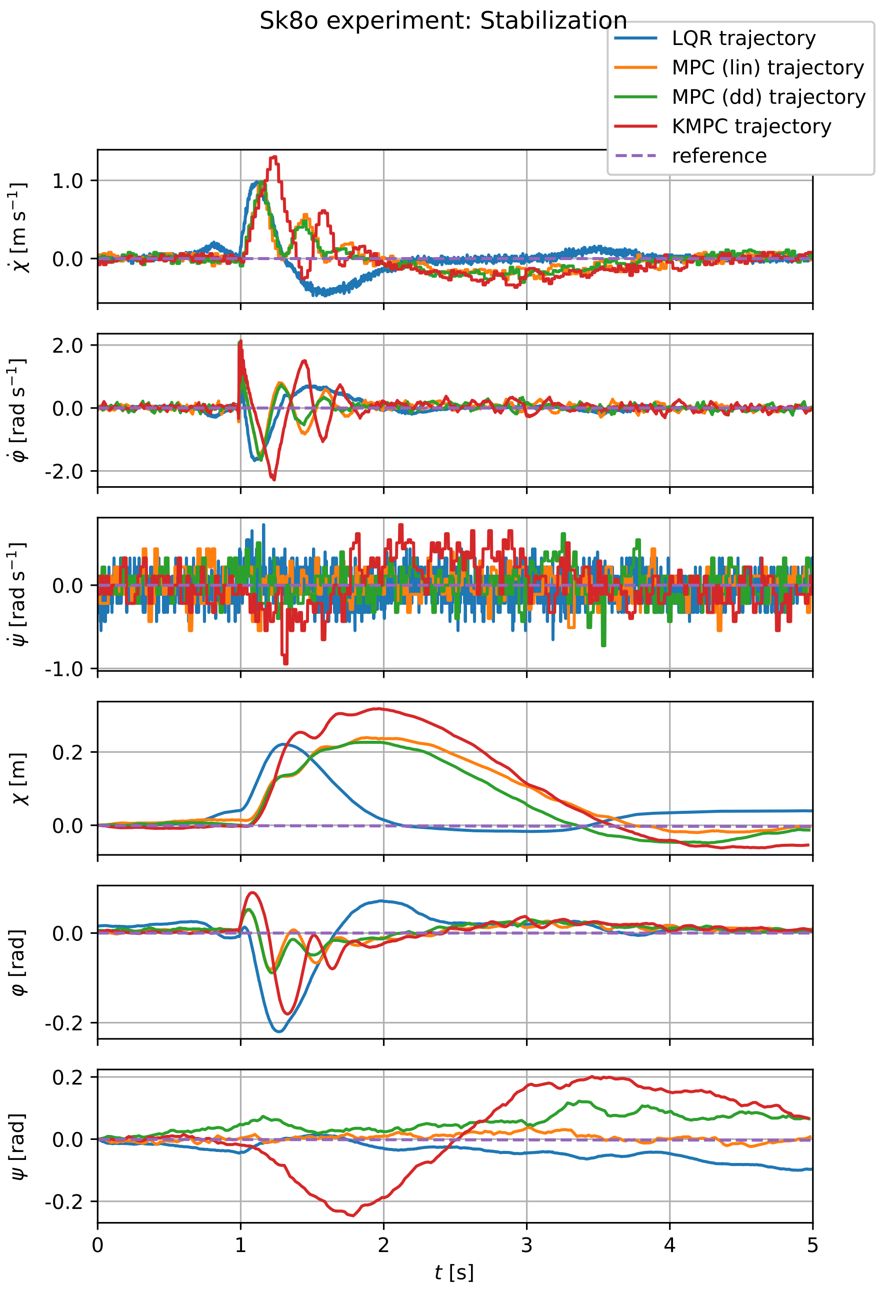

Figure 5.5: Experiment: Stabilization

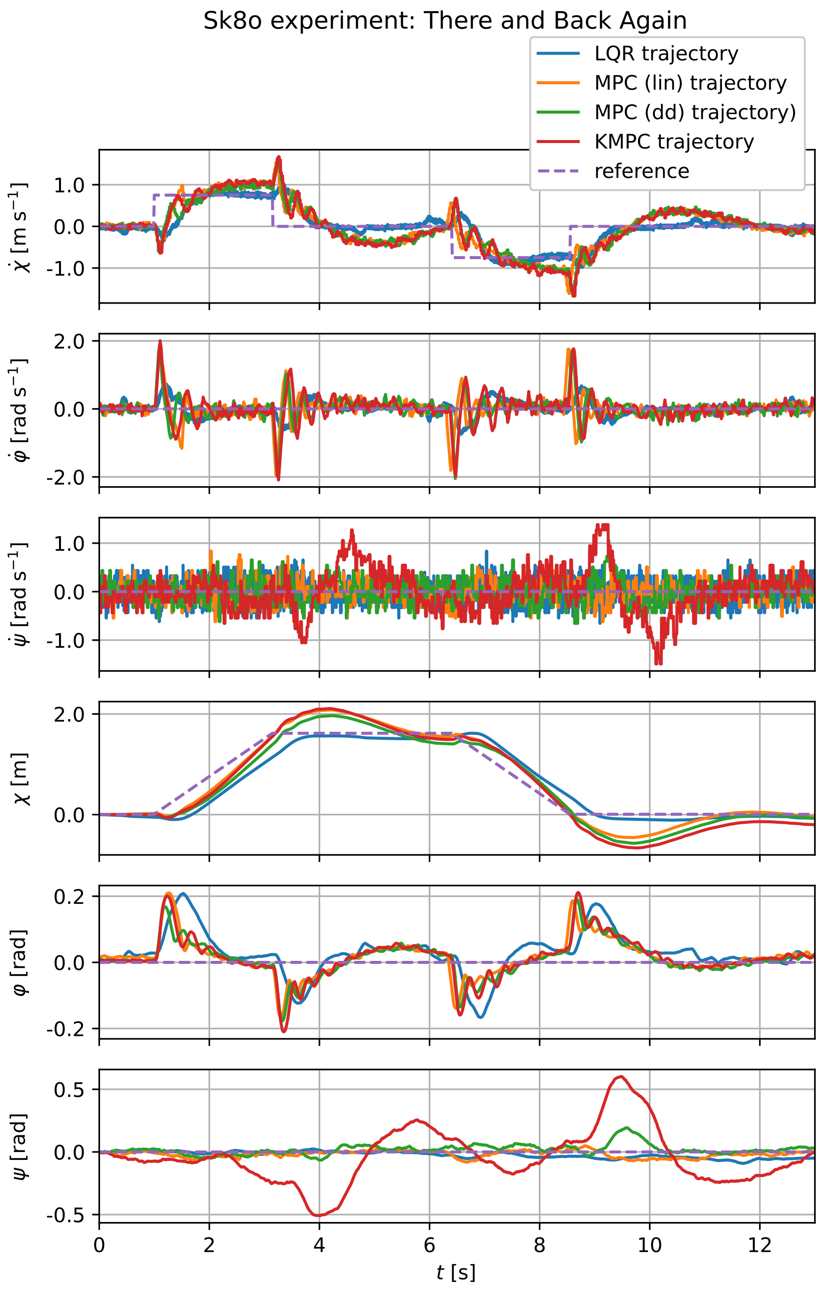

Figure 5.6: Experiment: There and Back Again

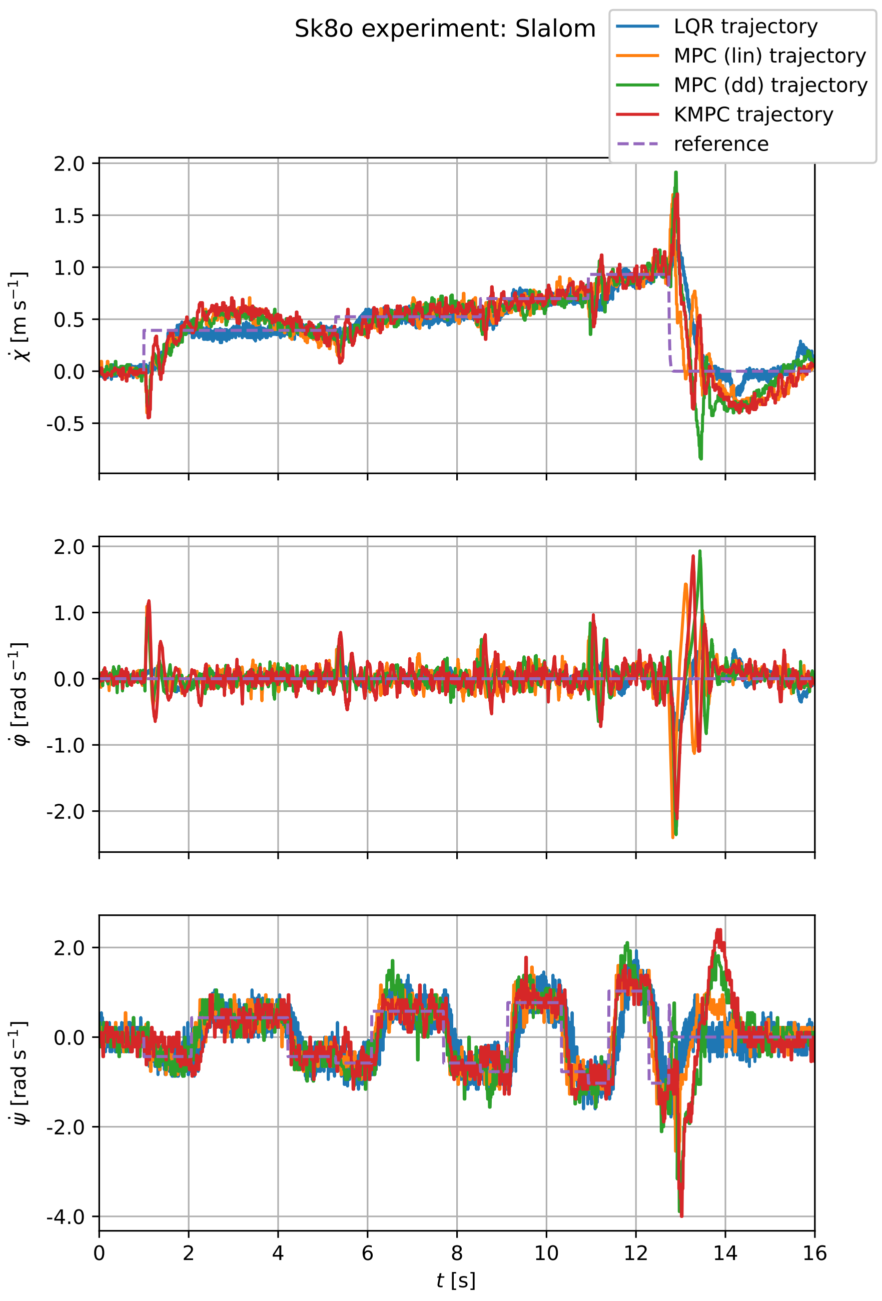

Figure 5.7: Experiment: Slalom A

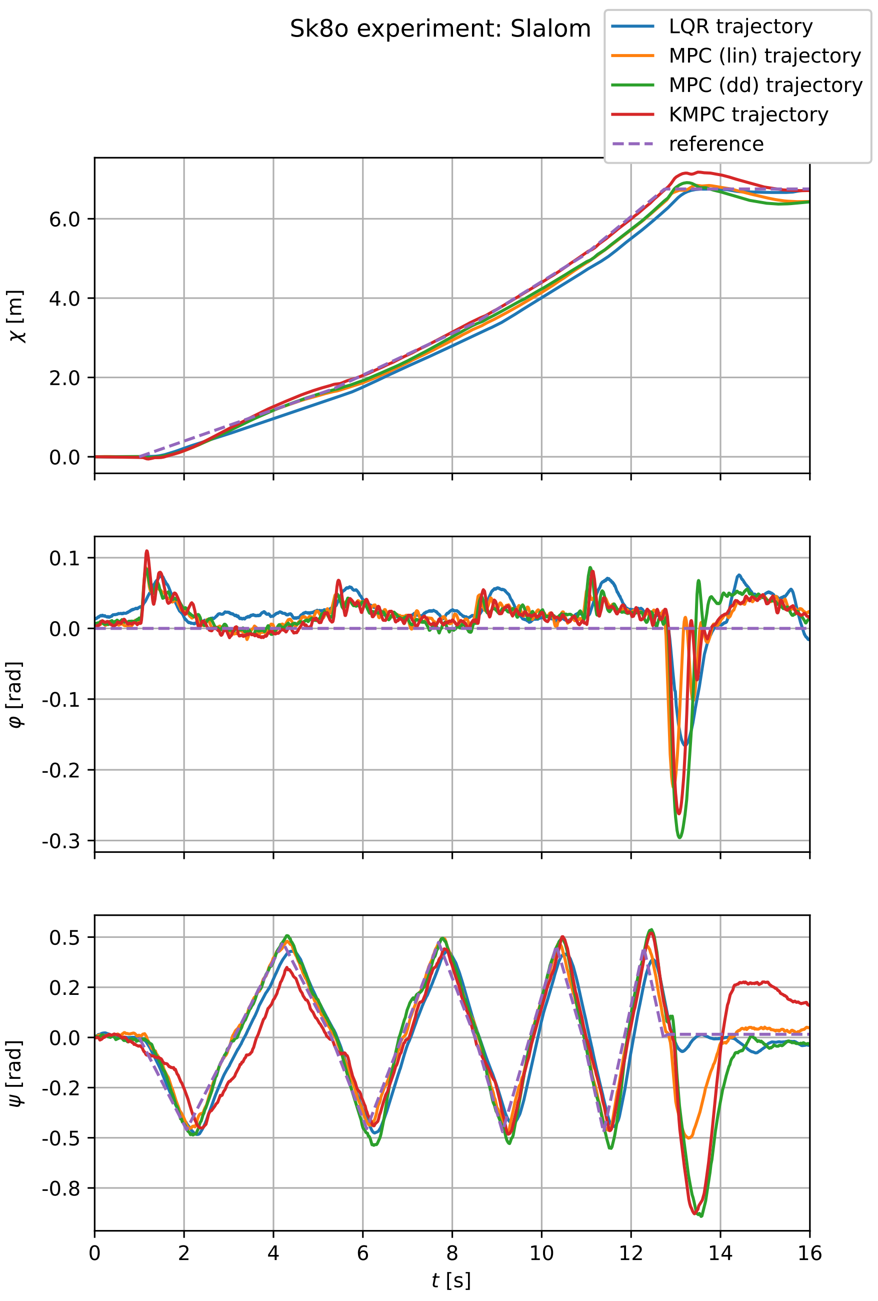

Figure 5.8: Experiment: Slalom B

5.10 Discussion

Although capable of balancing and steering the robot, both controllers found in this chapter (the KMPC controller with lifted linear predictor[6] and the MPC with unlifted predictor obtained via EDMD) failed to deliver better results than the MPC controller from chapter 4, in spite of the fact that their linear predictors scored better in the evaluation metric, as documented in table 5.1. At best, they perform similarly to the original MPC (such as in tracking $\dot{\chi}_{\mathrm{ref}}$ in figure 5.6). At worst, their performance is inferior (for example in state ψ in figure 5.8, where both data-driven controllers exhibit a larger overshoot at $t \approx 13$ s.).

I believe that the simulation data used to identify and evaluate the predictors (see section 5.4.1) are at fault. All of the trajectories are simulated with zero references – I suspect this causes the identified linear predictor to overfit to trajectories corresponding to the regulation task and worsens its ability to predict the states of the robot during reference tracking tasks. The logical next step would be to add references to the datasets, alas, I ran out of time and could not implement this. At least one other issue is present, which I discuss below.

5.10.1 Pitfalls of Using the Full Model with EDMD

When testing controllers which used linear predictors obtained via EDMD, I encountered a curious, seemingly non-sensical behaviour: the performance of the robot appeared to degrade with distance travelled. At first, the robot would track references for velocity $\dot{\chi}_{\mathrm{ref}}$ and yaw rate $\dot{\psi}_{\mathrm{ref}}$ as expected. After driving it around for a while though, it would start to visibly oscillate (in its yaw ψ) when given a $\dot{\chi}_{\mathrm{ref}} > 0$, even as $\dot{\psi}_{\mathrm{ref}}$ was zero. The same behaviour did not occur in the MPC controller found in chapter 4 (where the linear predictor is obtained by linear approximation).

The phenomenon is captured in figure 5.9, where the same experiment is performed twice, both times using the same MPC controller with an unlifted linear predictor obtained via EDMD. The trajectory labeled as MPC (dd) was obtained by running

[6] I only discuss the linear predictor that scored the best in the evaluation metric, but I have also experimented with predictors using other observables. The results were was similar or worse.

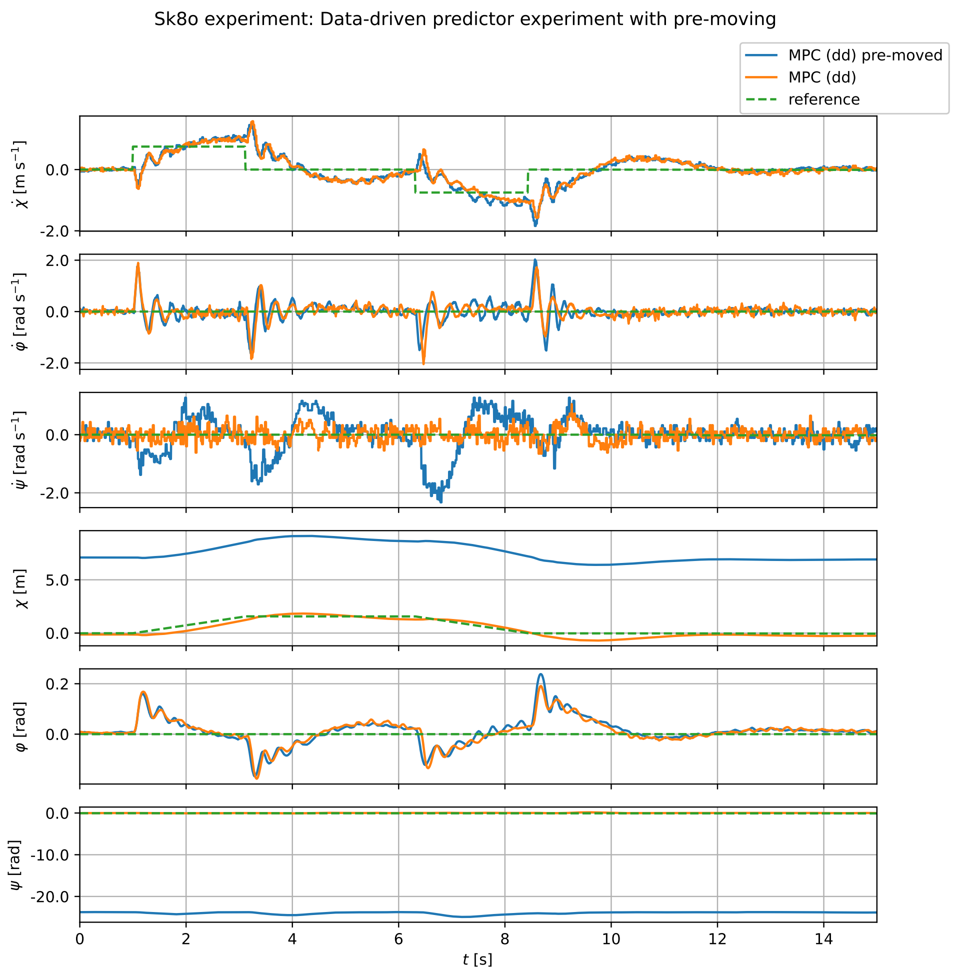

Figure 5.9: Failure of pre-moved data driven controller

the experiment right after starting the controller, while the other trajectory, MPC (dd) pre-moved, was preceded by a period of driving the robot around, which resulted in nonzero initial values of χ and ψ[7]. As show in the figure, merely changing the values of those two states significantly worsened the performance in

[7]Only the reference for the trajectory without pre-moving is plotted in the figure.

regulating $\dot{\psi}$ and, as a result, ψ.

I speculated that, probably due to insufficiencies of the identification data, EDMD has incorrectly identified an effect of one or both of the states χ, ψ on the remaining states of the system. This would prove to be the case.

Consider the unlifted linear predictor found via EDMD. Its matrix A is equal to

$$\begin{bmatrix} 9.5\cdot10^{-1} & 1.8\cdot10^{-2} & 0 & 2.9\cdot10^{-3} & -4.7\cdot10^{-1} & -1.1\cdot10^{-1} \\ 1.4\cdot10^{-1} & 9.4\cdot10^{-1} & 0 & -9.3\cdot10^{-3} & 2.1 & 0 \\ 0 & 0 & 9.5\cdot10^{-1} & 0 & -1.4\cdot10^{-3} & -3.7\cdot10^{-2} \\ 1.9\cdot10^{-2} & 2.1\cdot10^{-4} & 0 & 1.0 & -5.4\cdot10^{-3} & 0 \\ 1.5\cdot10^{-3} & 1.9\cdot10^{-2} & 0 & -1.2\cdot10^{-4} & 1.0 & 0 \\ 0 & 0 & 2.0\cdot10^{-2} & 0 & 0 & 1.0 \end{bmatrix}. \tag{5.18}$$

I've replaced all elements of (5.18) whose absolute value is less than 10^{-4} with zeroes for legibility.

The nonlinear model (see section 2.4) assumes that Sk8o moves over a flat horizontal plane. There is therefore no reason for states χ and ψ to affect the dynamics of the system in any way. In other words, the columns 4 and 6, corresponding to effects of states χ and ψ, respectively, should each contain only one nonzero element (blue), like in the matrix A obtained by linear approximation, which I reproduce in ((4.8), revisited). This is clearly not the case. Considering specifically the entries of (5.18) shown in red, the linear predictor apperently assumes that χ affects the future of the states $\dot{\chi}$, $\dot{\varphi}$, and φ, and that ψ does the same to $\dot{\chi}$ and $\dot{\psi}$.

$$A = \begin{bmatrix} 9.2\cdot10^{-1} & -1.3\cdot10^{-3} & 0 & 0 & -7.5\cdot10^{-1} & 0 \\ 2.8\cdot10^{-1} & 1.0 & 0 & 0 & 3.2 & 0 \\ 0 & 0 & 9.6\cdot10^{-1} & 0 & 0 & 0 \\ 1.9\cdot10^{-2} & 1.2\cdot10^{-5} & 0 & 1 & -7.6\cdot10^{-3} & 0 \\ 2.8\cdot10^{-3} & 2.0\cdot10^{-2} & 0 & 0 & 1 & 0 \\ 0 & 0 & 1.9\cdot10^{-2} & 0 & 0 & 1 \end{bmatrix} \tag{(4.8), revisited}$$

The presence of such, physically unjustifiable, state relationships is presumably also a contributing factor to the inferior performance of controllers found in this chapter.

In hindsight, it would have been better to use the version of the nonlinear model without states χ and ψ, as is done in Hodan's work. The states could then be added back in manually, by augmenting the matrices of linear predictors. Alternatively, it might perhaps be enough to enhance the simulated datasets by adding trajectories with larger magnitudes of the two states.

6 | Conclusion

The goal of this thesis was to design and deploy a predictive controller capable of balancing and steering Sk8o. I fulfilled this objective by implementing an MPC controller in chapter 4. I've tested the controller in simulations and on the physical robot, devising three different experiments which demonstrated the controller's ability to follow both simple and complex references and to withstand an external disturbance. The experiments confirmed that the controller is fit for use. Its performance is comparable to that of the preexisting LQR controller.

In chapter 5, I expanded the scope of this thesis to include Koopman MPC, in the hopes of improving upon my MPC controller. This proved to be a challenging task, with unexpected complexities emerging particularly when it came to obtaining trajectory data for the identification of linear predictors via EDMD. I wasn't able to solve all of the encountered issues in the time I had available. As a result, the KMPC controller I implemented is functional and able to balance the robot but exhibits quirks that make it impractical to use when steering Sk8o for prolonged time.

6.1 Open Problems and Future Work

What follows is a short list of problems that I did not have time to address.

6.1.1 EDMD with Reduced Order Model

As mentioned in section 5.10.1, I encountered a problem where EDMD would learn nonsensical effects of states φ and ψ on the system's dynamics. As mentioned therein, a promising course of action presents itself – if the identification of a linear predictor via EDMD was reformulated to make use of a modified model with states φ, ψ removed, this problem would vanish. The states could then be reintroduced by augmenting the matrices of the resulting linear predictor.

6.1.2 Better Simulation Data Sets

The simulated trajectories used for identification and evaluation of linear predictors, while usable, could be improved upon. The trajectories currently consist only of regulation tasks, where a controller attempts to move the Sk8o system from an initial condition back to the equilibrium. If reference tracking trajectories were added, the datasets would more faithfully recreate the real use case of the robot, and might allow for better-performing linear predictors to be identified.

6.1.3 Soft Constraints

I hadn't imposed any hard constraints on the states of Sk8o to ensure that the MPC optimization problem 4.6 always had a solution. Adding soft constraints on some states (particularly on φ) could perhaps improve the controller's performance. If the requirement that the robot remains upright was formulated via a soft constraint, more freedom would be allowed in tuning the cost matrix Q.

6.1.4 Hip Position Bug

There is an issue that sometimes occurs when Sk8o is being controlled from Odroid. It manifests itself as a sudden movement of one of Sk8o's hip motors, resulting in a rapid change in its height. This in turn results in a quick change in the other states, often followed by a loss of balance and fall of the robot.

I assume this issue is in some way related to communication, as I've only seen it happen while controlling Sk8o from Odroid. The bug is also not limited to controlling the robot via MPC – I had also observed it back when I controlled Sk8o with LQR running on Odroid. The uncalled-for movement of Sk8o's hip *could* be caused by a UART message with motor commands from Odroid becoming corrupted, briefly changing the required hip setpoints. So far however, this is just a hunch – while the behaviour would be consistent with such a cause, there is a checksum in place in the UART communication that should prevent it. I've yet to investigate whether it is sufficient.

Frustratingly, the error appears fairly infrequently and without any apparent regularity, making observation, not to mention replication, difficult. Sometimes it may happen twice in 10 minutes of operation, other times, half an hour passes without the bug rearing its ugly head. I'd especially like to resolve this problem, as its presence means that using my MPC controller may at any time be unexpectedly interrupted by Sk8o's rendezvous with the floor.

Appendices

A | Equations of the Segway model

This appendix is reproduced from Hodan's master's thesis [6], with some minor changes and typo corrections.

The nonlinear model introduced in section 2.4 used x as its state variable. Defining $x = [\dot{q}_g, q_g]$, $q_g = [\chi, \varphi, \psi]$, and $u = [u_\mathrm{L}, u_\mathrm{R}]$, the model is described by the equation

$$\ddot{q}_g = M_g^{-1}(B_g u - (C_g + D_g)\dot{q}_g - G_g) \tag{A.1}$$

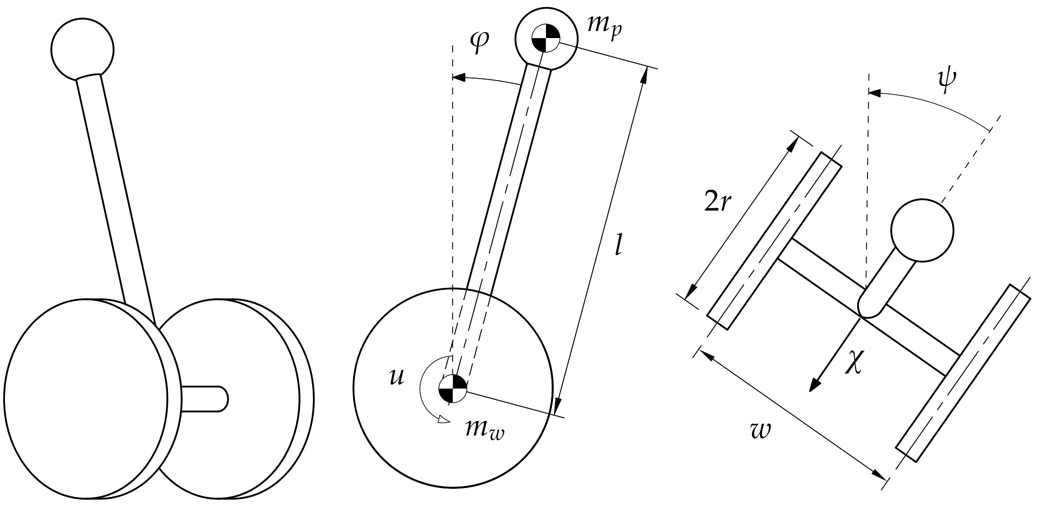

Figure A.1: The Segway model (figure reproduced with permission from [6])

The values of matrices used in equation (A.1) are

$$M_g = \begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ 0 & 0 & m_{33} \end{bmatrix}, \quad Q_g = \begin{bmatrix} 0 & c_{12} & c_{13} \\ 0 & 0 & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix},$$

$$D_g = \begin{bmatrix} d_{11} & d_{12} & 0 \\ d_{21} & d_{22} & 0 \\ 0 & 0 & d_{33} \end{bmatrix}, \quad B_g = \begin{bmatrix} 1/r & 1/r \\ -1 & -1 \\ -w/2r & w/2r \end{bmatrix}, \quad G_g = \begin{bmatrix} 0 \\ -mlg\sin\varphi \\ 0 \end{bmatrix}, \tag{A.2}$$

Symbol	**Parameter**	**Value**	**Unit**
b	damping coefficient	$1 \cdot 10^{-2}$	Nmsrad^{-1}
J	wheel moment of inertia about its turning axis	$7.35 \cdot 10^{-4}$	kgm^2
K	wheel moment of inertia about the vertical axis	$3.9 \cdot 10^{-4}$	kgm^2
m_w	wheel mass	$3.0 \cdot 10^{-1}$	kg
r	wheel radius	$8.0 \cdot 10^{-2}$	m
w	distance between wheels	$2.9 \cdot 10^{-1}$	m
l	height of centre of mass	$2.907 \cdot 10^{-1}$	m
I_{px}	roll moment of inertia of the pendulum	$1.5625 \cdot 10^{-2}$	kgm^2
I_{py}	pitch moment of inertia of the pendulum	$1.18625 \cdot 10^{-2}$	kgm^2
I_{pz}	yaw moment of inertia of the pendulum	$1.18625 \cdot 10^{-2}$	kgm^2
m_p	mass of the segway without wheels	4.0	kg

Table A.1: Parameters of the Segway model. All moments of inertia are shown with respect to the centre of mass of the body

where the elements are

$$
\begin{aligned}
&m_{11} = m_p + 2m_w + 2\frac{J}{r^2}, \qquad m_{12} = m_{21} = m_p l \cos\varphi, \qquad m_{22} = I_{py} + m_p l^2,\\
&m_{33} = I_{pz} + 2K + (m_w + \frac{J}{r^2})\frac{w^2}{2} - (I_{pz} - I_{px} - m_p l^2)\sin^2\varphi,\\
&\quad c_{12} = -m_p l \dot{\varphi}\sin\varphi, \qquad c_{13} = m_p l \dot{\psi}\sin\varphi, \qquad c_{23} = (I_{pz} - I_{px} - m_p l^2)\dot{\psi}\sin\varphi\cos\varphi,\\
&\quad c_{31} = m_p l \dot{\psi}\sin\varphi, \qquad c_{32} = -c_{23}, \qquad c_{33} = -(I_{pz} - I_{px} - m_p l^2)\dot{\varphi}\sin\varphi\cos\varphi,\\
&\quad d_{11} = \frac{2b}{r^2}, \qquad d_{12} = d_{21} = -\frac{2b}{r}, \qquad d_{22} = 2b, \qquad d_{33} = \frac{w^2}{2r^2}b
\end{aligned}
\tag{A.3}
$$

The values of the physical parameters can be found in table A.1.

B | Matrices of the dense MPC formulation

The dense MPC formulation 4.6 uses matrices F and H. Details of their assembly follow below.

Note that matrices A, B, C come from the linear predictor used, while matrices R, Q and S define the MPC's cost function.

$$H = \bar{C}^T \bar{Q} \bar{C} + \bar{R}, \tag{B.1}$$

$$F = \begin{bmatrix} \hat{A}^T \bar{Q} \bar{C} \\ -\bar{T} \bar{C} \end{bmatrix}^T . \tag{B.2}$$

$$\bar{Q} = \begin{bmatrix} C^T Q C & & & & \\ & C^T Q C & & & \\ & & \ddots & & \\ & & & C^T Q C & \\ & & & & C^T S C \end{bmatrix} \tag{B.3}$$

$$\bar{T} = \begin{bmatrix} QC & & & & \\ & QC & & & \\ & & \ddots & & \\ & & & QC & \\ & & & & SC \end{bmatrix} \qquad \bar{R} = \begin{bmatrix} R & & & & \\ & R & & & \\ & & \ddots & & \\ & & & R & \\ & & & & R \end{bmatrix} \tag{B.4}$$

$$\bar{A} = \begin{bmatrix} 0 & & & \\ A & 0 & & \\ & \ddots & \ddots & \\ & & A & 0 \end{bmatrix} \qquad \bar{B} = \begin{bmatrix} B & & \\ & \ddots & \\ & & B \end{bmatrix} \tag{B.5}$$

$$\bar{C} = \begin{bmatrix} B & & & & \\ AB & B & & & \\ A^2B & AB & B & & \\ \vdots & & & \ddots & \\ A^{N-1}B & & \dots & & B \end{bmatrix} \qquad \hat{A} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ \vdots \\ A^N \end{bmatrix} \tag{B.6}$$

C | Observables Derived from Model

The observables I use in the KMPC controller found in chapter 5 originated as the analytical solution of the nonlinear model described in appendix A.

I solved the model in Matlab, substituting in the parameters of the model and splitting the solution for each of Sk8o's six states into individual terms. I then combined all of the terms into a single set of functions, which is reproduced in (C.1).

Not all of the functions f_i are fit for use as observables $g_i(x)$ however, as some f_i are functions of u. I formulated two approaches of tackling this:

- A : Don't use f_i if it is a function of u.
- B : Use all f_i, setting $u_{\mathrm{L}} = u_{\mathrm{R}} = 0.1$.

Additionally, I was concerned that the denominators of some of the functions might become zero. I likewise formulated two approaches:

- 0 : Don't worry about it and use the functions as they are.
- 1 : Replace each denominator den with $limit(den)$. The $limit(den)$ function returns den if $\|den\| > 0.1$, otherwise it returns $0.1 \cdot \mathrm{sgn}(den)$.

By combining those approaches I arrive at 4 sets of observables. I ran EDMD for all of them. They all scored essentially the same in the evaluation metric described in section 5.5 (their results starting to differ in 4th decimal place), so I decided to only discuss the one that scored the best out of them, which was the combination A1. I.e., removing functions of u and limiting denominator values.

The order in which the rest scored was A0, B1, B0.

$$
\begin{aligned}
f_0(x) &= -\frac{\dot{\varphi}}{0.80\cos(\varphi)^2-1.0} & f_{14}(x,u) &= -\frac{u_\mathrm{R}}{0.80\cos(\varphi)^2-1.0} \\
f_1(x,u) &= -\frac{u_\mathrm{L}}{0.80\cos(\varphi)^2-1.0} & f_{15}(x) &= -\frac{\dot{\chi}}{0.80\cos(\varphi)^2-1.0} \\
f_2(x,u) &= -\frac{u_\mathrm{R}}{0.80\cos(\varphi)^2-1.0} & f_{16}(x) &= -\frac{1.0\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_3(x) &= -\frac{\dot{\chi}}{0.80\cos(\varphi)^2-1.0} & f_{17}(x) &= -\frac{\dot{\varphi}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_4(x) &= -\frac{\dot{\varphi}^2\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{18}(x,u) &= -\frac{u_\mathrm{L}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_5(x) &= -\frac{\dot{\psi}^2\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{19}(x,u) &= -\frac{u_\mathrm{R}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_6(x) &= -\frac{\cos(\varphi)\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{20}(x) &= -\frac{\dot{\chi}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_7(x) &= -\frac{\dot{\varphi}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{21}(x) &= -\frac{\dot{\varphi}^2\cos(\varphi)\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_8(x,u) &= -\frac{u_\mathrm{L}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{22}(x) &= -\frac{\dot{\psi}^2\cos(\varphi)\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} \\
f_9(x,u) &= -\frac{u_\mathrm{R}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{23}(x) &= \frac{\dot{\psi}}{11.36\sin(\varphi)^2+1.0} \\
f_{10}(x) &= -\frac{\dot{\chi}\cos(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{24}(x,u) &= \frac{u_\mathrm{L}}{11.36\sin(\varphi)^2+1.0} \\
f_{11}(x) &= -\frac{\dot{\psi}^2\cos(\varphi)^2\sin(\varphi)}{0.80\cos(\varphi)^2-1.0} & f_{25}(x,u) &= \frac{u_\mathrm{R}}{11.36\sin(\varphi)^2+1.0} \\
f_{12}(x) &= -\frac{\dot{\varphi}}{0.80\cos(\varphi)^2-1.0} & f_{26}(x) &= \frac{\dot{\psi}\cdot\dot{\chi}\sin(\varphi)}{11.36\sin(\varphi)^2+1.0} \\
f_{13,u}(x) &= -\frac{u_\mathrm{L}}{0.80\cos(\varphi)^2-1.0} & f_{27}(x) &= \frac{\dot{\varphi}\cdot\dot{\psi}\cos(\varphi)\sin(\varphi)}{11.36\sin(\varphi)^2+1.0}
\end{aligned}
\tag{C.1}
$$

D | Supplemental Experiment Figures

This appendix contains figures of experiments performed on the physical robot in their full form, including control actions. Figures for the stabilization experiment also show the whole experiment, which consisted of applying three consecutive disturbances to the robot via the device described in 4.5.1. This figure was limited to only the first disturbance in preceding chapters to improve legibility.

The meaning of the legends used within the figures follows:

- LQR – refers to the original LQR controller running on Teensy
- MPC (lin) – MPC controller with unlifted linear predictor obtained via linearization, as described in section 4.1.
- MPC (dd) – MPC controller with unlifted linear predictor obtained by EDMD, as described in section 5.3.
- KMPC – Koopman MPC with lifted linear predictor using observables described in appendix C.

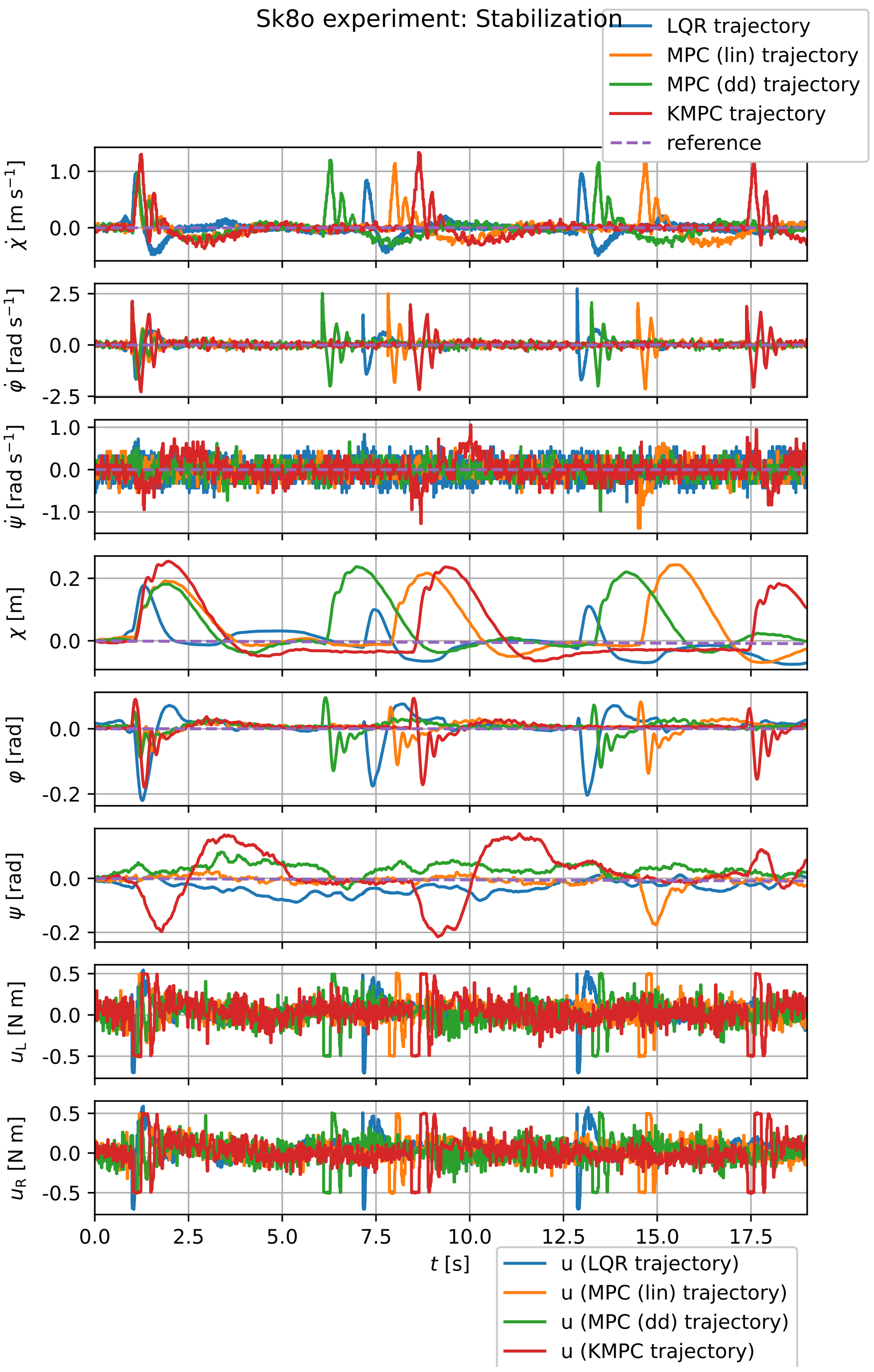

Figure D.1: Full Experiment: Stabilization

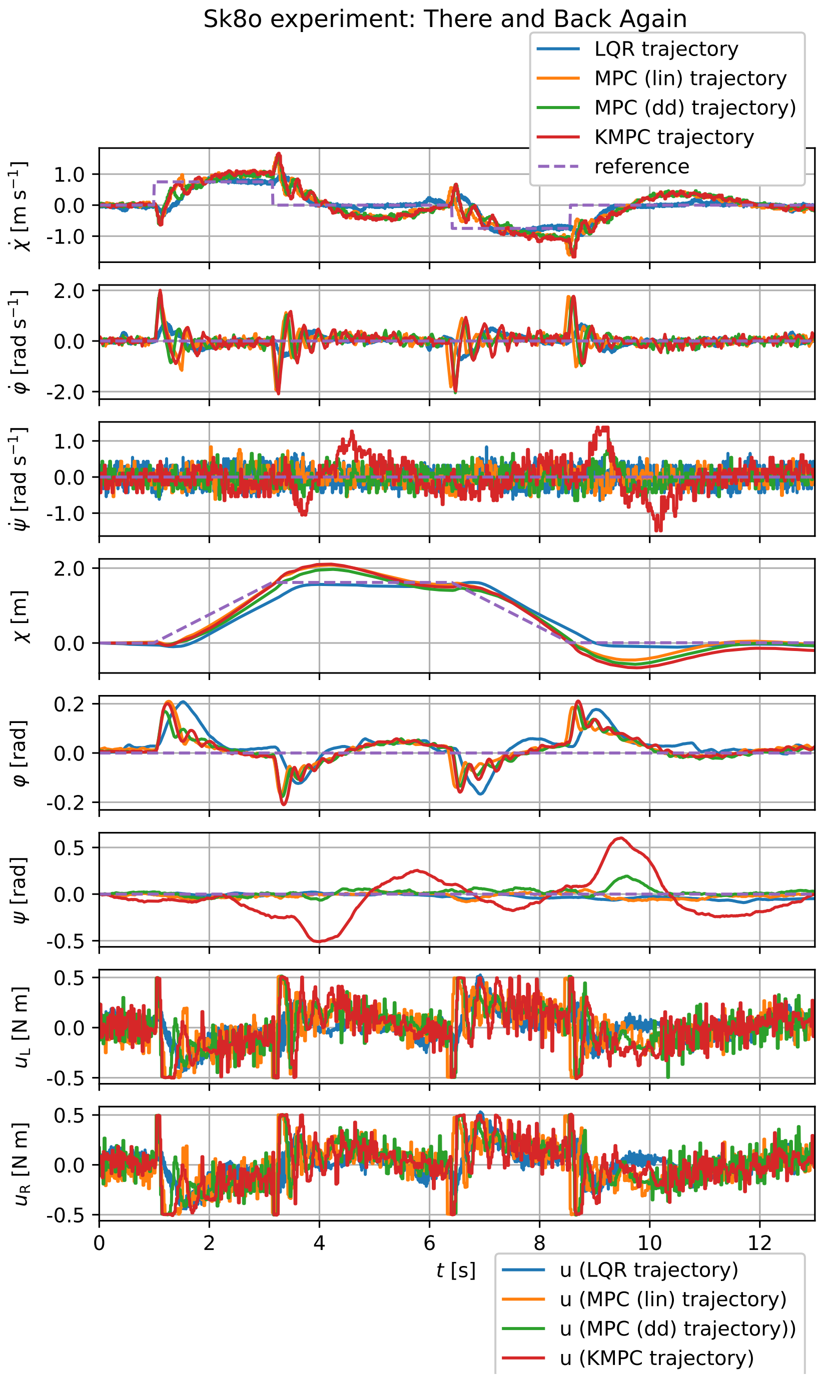

Figure D.2: Full experiment: There and Back Again

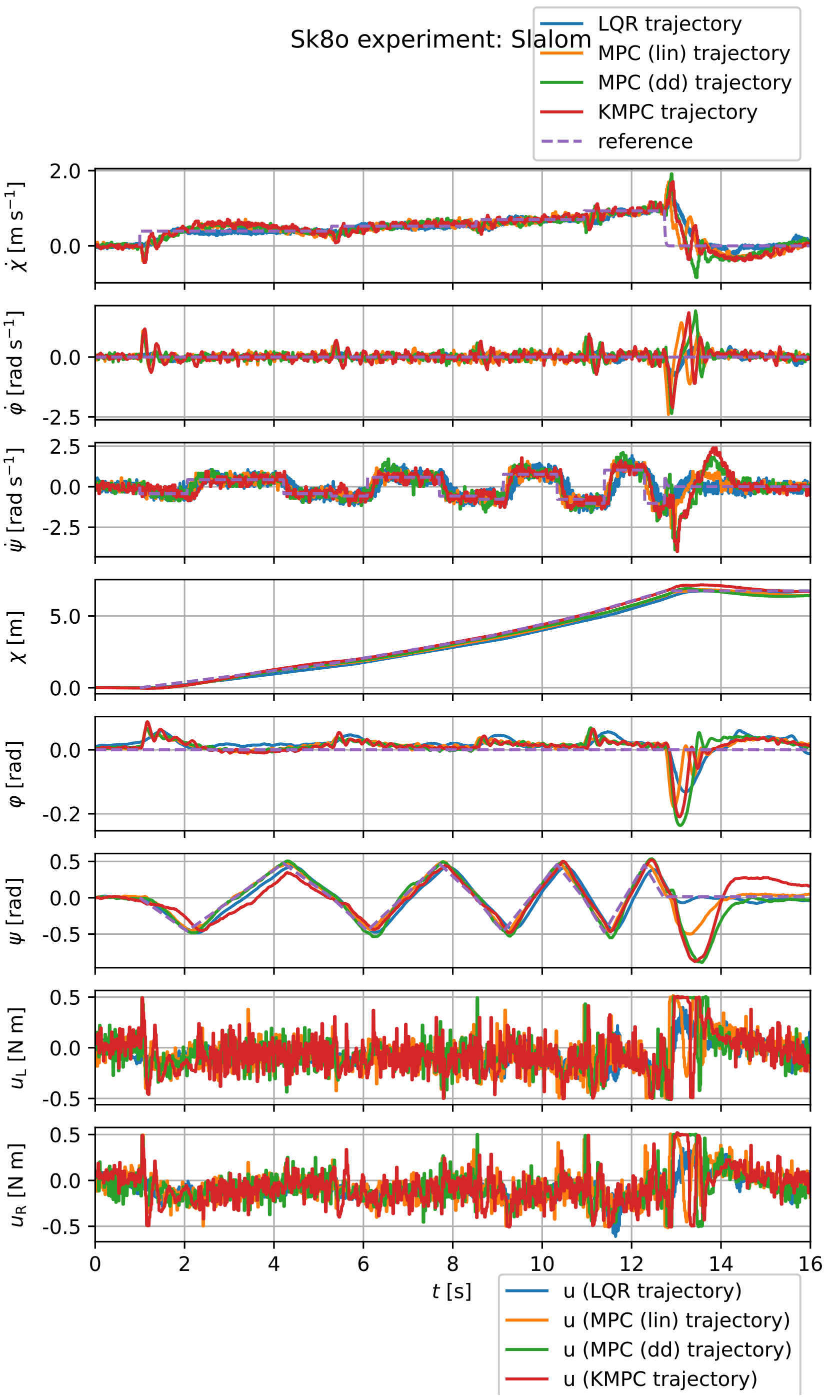

Figure D.3: Full experiment: Slalom

References

[1] Hassan Arbabi, Milan Korda, and Igor Mezić. A data-driven koopman model predictive control framework for nonlinear partial differential equations. In *2018 IEEE Conference on Decision and Control (CDC)*, pages 6409–6414, 2018.

[2] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. Predictive control for linear and hybrid systems. 2017.

[3] Steven L. Brunton, Marko Budišić, Eurika Kaiser, and J. Nathan Kutz. Modern koopman theory for dynamical systems, 2021.

[4] Loi Do, Milan Korda, and Zdeněk Hurák. Controlled synchronization of coupled pendulums by koopman model predictive control, 2022.

[5] Dominik Fischer. Control system for morphing lattice-based structures, 2023.

[6] Dominik Hodan. Reinforcement learning-based control system for the sk8o robot, 2023.

[7] Adam Kollarčík. Modeling and control of two-legged wheeled robot, 2020.

[8] B. O. Koopman and J. V. Neumann. Dynamical systems of continuous spectra. 1932.

[9] Milan Korda and Igor Mezić. Linear predictors for nonlinear dynamical systems: Koopman operator meets model predictive control. *Automatica*, 93:149–160, 2018.

[10] Milan Korda, Yoshihiko Susuki, and Igor Mezić. Power grid transient stabilization using koopman model predictive control. *IFAC-PapersOnLine*, 51(28):297–302, 2018. 10th IFAC Symposium on Control of Power and Energy Systems CPES 2018.

[11] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[12] Matthew O. Williams, Maziar S. Hemati, Scott T.M. Dawson, Ioannis G. Kevrekidis, and Clarence W. Rowley. Extending data-driven koopman analysis to actuated systems. *IFAC-PapersOnLine*, 49(18):704–709, 2016. 10th IFAC Symposium on Nonlinear Control Systems NOLCOS 2016.