Data-Driven Adaptive Control of Unmanned Surface Vehicles Using Learning-Based Model Predictive Control

Markus Svedberg

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6205
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2023

Abstract

In this thesis, the subject of data-driven control of Unmanned Surface Vehicles (USVs) is explored. The control task is formulated through Nonlinear Model Predictive Path Following Control (NMPFC). System identification (SYSID) and Reinforcement Learning (RL) are employed to improve performance in a data-driven manner. The objectives were to assess the resulting controller's path-following ability, as well as its adaptability to new environments, enabling the use of the controller on different USVs and under different conditions. The evaluation was done in simulation and in experiments with the Saab Kockums AB's Piraya vessel. Based on the results presented, NMPFC gives low-error solutions in both simulation and experiments but seems non-robust against disturbances and model mismatch. The simulation results of the learning-based methods showed that enabling SYSID on a USV with an incorrect initial model would identify the correct model in one update. Moreover, applying SYSID in experiments roughly halved the USV tracking error, compared to the usage of a model identified offline. Lastly, the RL implementation was found to increase performance in offline simulation, though less than SYSID. Moreover, the RL method computational times prohibited real-time control of the Piraya. This led to the method not being deployed in experiments.

Acknowledgments

First of all, I would like to thank Saab Kockums AB and The Department of Automatic Control at LTH for this opportunity.

I would also like to thank Joel Andersson of CasADi (UW), for answering many of my questions about the tool. Thank you also to Professor Sébastien Gros (NTNU) for discussing his work on learning-based control with me.

Moreover, thank you to Jonas Rosquist (Saab Kockums), Jens-Olof Lindh (Saab Kockums), and Mattias Nilsson (Saab Kockums) for their interest in the project, for making me feel at home while testing in Karlskrona. In addition, I would like to extend a special thanks to all of my colleagues at Saab Kockums Malmö for the time I got to spend with them.

Finally, I am grateful to my supervisors Birgitta Wingqvist (Saab Kockums/LTH) and Björn Olofsson (LTH) for encouraging me to explore the methods I found interesting for solving the problem at hand. I am also grateful for their insightful feedback, and clarifying discussions.

Contents

List of Abbreviations

DOF	Degrees of Freedom
GN	Gauss-Newton (method)
GPS	Global Positioning System
INS	Inertial Navigation System
IPOPT	Interior Point Optimizer
MPC	Model Predictive Control
MPFC	Model Predictive Path Following Control
MSE	Mean Square Error
NED	North East Down
NMPC	Nonlinear Model Predictive Control
NMPFC	Nonlinear Model Predictive Path Following Control
PEM	Prediction Error Method
QRQP	QR Quadratic Programming
RL	Reinforcement Learning
ROS	Robot Operating System
SGD	Stochastic Gradient Descent
SYSID	System Identification
SeGD	Semi-Gradient Descent
USV	Unmanned Surface Vehicle
XTE	Cross Track Error

1

Introduction

In recent years, the development of autonomous vehicles has been progressing at a rapid pace. Of special interest are small autonomous boats, usually referred to as *Unmanned Surface Vehicles* (USVs). The interest in these developments stems from uses in the defense industry, coast guard, mapping, ocean research, and transport. However, steering a ship with precision is no easy feat, due to the highly complex nature of the ship-water interactions. Previous solutions to USV control include PID control, LQR controllers, Stochastic Control, as well as LOS path-following control, sliding mode control, and cross-tracking control [Fossen, 2021]. In some applications, like docking at a wharf, or moving through an archipelago with small margins of error, precision control of USVs is necessary. In this thesis, we investigate model-based control, for path-following control of an USV. Any model-based control method requires mathematical modeling and parameter estimation. The first of these tasks might be very difficult due to the complex dynamics involved in the system. With regards to USVs, this includes accounting hydrostatics, hydrodynamics, the geometry of the vessel, wind, etc [Fossen, 2021]. In the second task, it is often necessary to estimate the parameters which are part of the model through a gray-box experiment. This includes running controlled experiments which might be time-consuming and costly, in order to record enough data to fit to the model. Furthermore, if the model is nonlinear, such as an USV model, an estimated model will only be valid in situations similar to those encountered during estimation. Often, the limiting factor is the velocity the USV is traveling at when the estimation is done. Thus, obtaining such a model through offline system identification comes with many drawbacks. Therefore, it is

relevant to ask if the performance and number of operative situations of the USV can be increased by employing these kinds of methods online instead.

1.1 Prior Work

Previously, in [Kockum, 2022] a *Model Predictive Control* (MPC) controller was developed for the USV to be studied in this thesis, the *Piraya*. It handled traveling to way-points in an archipelago successfully, at low speeds. That work used, and we will use, a gray-box model of the Piraya, obtained using offline system identification. Ship modeling is detailed in [Fossen, 2021], where various models are presented for ships operating in different conditions, at different velocities, and more. Like the Piraya model, these are gray-box models and require parameter estimation from real-world data. A gray-box model of the Piraya based on models from [Fossen, 2021] was obtained in [Ljungberg, 2021] for use at 2 m/s.

In [Faulwasser et al., 2017], *Model Predictive Path Following Control* (MPFC) was used for control of a three-joint robot, which thus gained writing capabilities. No instance of the MPFC method applied to USV path-following control could be found. Furthermore, in [Martinsen et al., 2022], the authors successfully implemented *System Identification* (SYSID) and *Reinforcement Learning* (RL) using Q-learning method on the *ReVolt* and *milliAmpere* fully actuated vessels, improving control performance in both simulation and experiments.

1.2 Problem Formulation

This thesis is an investigation into how online learning techniques such as system identification and reinforcement learning can improve the path-following capabilities of the Piraya USV. It will be explored how this can be implemented in the context of model predictive control. Furthermore, the thesis aims to investigate whether these kinds of methods enable the development of control algorithms which are vessel-independent, and thus can adapt themselves to the control of new vessels. The purpose of this thesis is to further the development in the maneuvering of USVs, and online learning control methods.

The following topics are part of the problem formulation of this thesis:

- How can path-following Model Predictive Control be implemented to control the Piraya?
- What methods, from the theory of control systems and machine learning, can be used to increase tracking performance in the context of Model Predictive Control?
- How do the learning methods adapt to the dynamics of the vessel, and external disturbances that are present?
- Can the learning methods extend the operative range of the vehicle, by learning dynamics online? For example, increase the range of velocities at which the model-based controller can be used?

1.3 Limitations

External obstacle avoidance is not included in the problem formulation, and the path is thus assumed to be obstacle-free. The actuator dynamics of the Piraya are not modeled, introducing unknown dynamics. Only learning-based control methods which include MPC are investigated. Waves are not to be compensated for as a disturbance, only winds and currents, which are relatively stationary, are.

2

Background

2.1 Piraya USV

The Piraya is an Unmanned Surface Vehicle owned and developed by Saab Kockums AB [WARA-PS, 2023]. The vessel, which measures 4 meters from bow to stern, 1.4 meters from port to starboard, and weighs approximately 300 kg, can be seen in figure 2.1. Mounted at the stern is a 20 Horsepower outboard engine. Rudder action is achieved by rotating the motor, with a maximum angle of 30° in either direction. The Piraya is equipped with a video camera and LiDAR, as well as RADAR sensors for navigation. Further, it is equipped with a GPS sensor measuring position, heading, and speed. The sensor suite is adaptable and could also include an *Inertial Navigation System* (INS) containing an accelerometer, gyro, and magnetometer, which supplies measurements of the current vessel rotation (in 3D space) as well as velocities in 3 dimensions. Communication with the USV from a base station is handled using a long-range WiFi-antenna. Any received commands in terms of throttle, rudder, and gear are processed by a low level controller before being mechanically actuated on the craft. Furthermore, it is possible to remotely control the USV with an RC remote, which acts as a fail-safe during autonomous testing, and the vessel will shut down if the signal connection is lost. Onboard the Piraya, a *Robot Operating System* (ROS) runs on a Ubuntu 20.04 Intel NUC i7-7567@3.5GHz with 16GB RAM, and allows for remote code execution and logging over WiFi. A 9-parameter model of the Piraya has previously been estimated through offline system identification in [Ljungberg, 2021]. This model is presented in Section 3.1.

(a) Piraya USV in stationary state

(b) Piraya USV banking starboard

Figure 2.1 The Piraya USV during initial testing on 2023-04-05. Photos: Saab Kockums / Glenn Pettersson.

2.2 Model Predictive Control

Model Predictive Control is a technique for control of dynamical systems [Rawlings et al., 2017]. As in other control applications, a *controller* is responsible for delivering inputs, or *control signals* to the system, with the aim of driving the system *state* (e.g., position, or velocity) to some desired, or *reference state*. Unlike traditional control techniques like the widely-used PID controller, which computes the control signal from various manipulations of the *state error*, i.e., the difference between the current state and the reference state, the MPC controller makes use of a dynamic model of the system when calculating the control signal [Rawlings et al., 2017]. A dynamic model can be used to predict how the system state will evolve over time, given the current system state and the control signal applied. Intuitively, in MPC this process is reversed, and the controller tries to find what sequence of control signals applied from now on, will achieve the desired behavior. Furthermore, MPC controllers are able to explicitly account for *constraints* on both the control signals and states. That is, it is possible to specify a limit on the control signal, like a maximum throttle, or in the state, like a maximum velocity [Rawlings et al., 2017]. The controller will then find (if possible) a way to drive the system to the reference state, while respecting those constraints, which might look very different from the unconstrained case.

In order to find those control signals, numerical optimization is used. A *cost function* is specified, which assigns a numerical value to the behavior of the system [Rawlings et al., 2017]. A good cost function should give a small value when the system is performing as desired, e.g., when it is close to or at the reference state, and when excess control signal is not applied. When the MPC controller is ran, the numerical optimization will find the sequence of control signals which minimize the cost function, over some *prediction horizon*, i.e., a number of time-steps into the future. Since the cost function summarizes the desired behavior of the system into one numerical value, the solution is necessarily a compromise between driving the system to the reference state, not applying too much control signal, and staying within the constraints. The behavior of the real system, controlled by MPC, can then be adjusted by *tuning* the cost function, i.e., making it "cost more" to deviate from the reference state, or by making it "cost more" to use a high control signal [Rawlings et al., 2017]. Here, the former would

lead to faster control behavior, and the latter would lead to slower control, while reducing control input, like gas for an engine.

When deployed *online*, i.e., controlling a system in real-time, the MPC controller is ran at some frequency, and calculates the optimal control signal sequence at that time, given the newest measurements. The first calculated control signal is applied to the actuators of the real system, while the rest of the control sequence is discarded. In the next iteration, this is repeated, and the first signal in the calculated sequence is applied, while the rest are discarded [Rawlings et al., 2017]. This is done since the model used in MPC is never a perfect description of the real world, and the optimal controls found at the next time step will tend to vary from the ones found at the previous. Having to redo the optimization every iteration places an inherent limitation on the frequency at which the controller can be run, since the numerical optimization is a time-intensive process, and must finish before the next iteration can start.

Path Following MPC

In [Faulwasser, 2013], the author outlines the theory for a method of *Model Predictive Path Following Control* (MPFC), where a system is controlled along a predefined path but chooses the velocity along the path dynamically. This is then implemented in a writing robot in [Faulwasser et al., 2017]. For USVs, trajectory tracking through way-point-following is a more common method [Fossen, 2021].

2.3 Learning-Based Control

Recently, machine learning and statistical techniques have been extensively applied in the context of control systems [Moe et al., 2018]. These techniques are often model-free, and instead learn the behavior of the systems from data. One broad category of machine learning algorithms used in control in recent years is *Reinforcement learning* (RL) and in particular, Q-learning [Gros and Zanon, 2020].

Reinforcement learning is a subset of machine learning in which an agent, meaning any learning system, learns an optimal *policy* (controller) by interactions with its environment. The objective of the agent is to maximize

a numerical reward, or minimize a numerical cost, cf. MPC. The reward to be maximized is not just related to the current or next state but to all subsequent states. Thus, an RL agent will learn to maximize the total reward during the span of any task [Sutton and Barto, 2018]. Any RL algorithm contains trial and error, whereby the agent learns of the reward for any action by trying that action, and how that will impact the subsequent rewards it will receive. Through enough trial and error, the value of any action in any state will be known, and the policy can easily be chosen as the action with the lowest cost in any state. In order to achieve this, *exploration* is necessary, i.e., trying different actions and learning of the result. However, an optimal, final, policy, will not explore, as it should always maximize the total reward received. This is called *exploitation*, but it cannot be done without first exploring [Sutton and Barto, 2018]. A good RL agent should be able to balance these two aspects of learning. Too much exploration will reduce the reward gained during any task, while too much exploitation might prevent the agent from learning better ways of accomplishing the task, yielding higher rewards in the long term.

Learning-Based MPC

In [Gros and Zanon, 2020], a method for improving the performance of an MPC controller using *Reinforcement Learning* (RL) was first proposed. The method was then improved in [Zanon et al., 2019] where implementation details were discussed. This method was then expanded to use online *System Identification* (SYSID) through the *Prediction Error Method* (PEM) along with RL in [Martinsen et al., 2020], where various approaches to combining the SYSID and RL updates were investigated. Finally, these methods were applied for *Nonlinear Model Predictive Control* (NMPC) control of an USV in [Martinsen et al., 2022], where control performance was successfully increased using these methods.

3

Model-Based Ship Control

In this chapter, the framework for controlling an USV along a path with MPC is established. Firstly, general ship modeling in six *degrees of freedom* (DOF) is briefly introduced. Then, relevant coordinate frames are discussed. After that, a 3-DOF model for USVs is shown, which is suitable for navigation and will be used for the remainder of the thesis. Then, a simple model for external disturbances is shown. In the final section on ship modeling, the model for the Piraya found in [Ljungberg, 2021] is shown, which later is used for simulation. In the following section, NMPC is introduced in general. Finally, the method of *Nonlinear Model Predictive Path Following Control* (NMPFC) is discussed.

3.1 Ship Modeling in 6-DOF

Marine craft motion can in general be described in six degrees of freedom. These are surge u: motion in the forward direction, sway v: motion in the sideways direction, yaw r: rotation about the vertical axis, roll p: rotation about the forward axis parallel with u, pitch q: rotation about the sideways axis parallel with v and heave w: motion in the vertical direction [Fossen, 2021], see Figure 3.1. These velocities are collected in the generalized velocity vector $\boldsymbol{\nu} = [u, v, w, p, q, r]^T$. Furthermore, the pose vector $\boldsymbol{\eta} = [x, y, z, \phi, \theta, \psi]^T$ is introduced. Here (x, y, z) is the position of the ship in Cartesian space, and (ϕ, θ, ψ) are Euler angles about the x, y, and z axes, respectively. These angles correspond to the angular velocities $p, q,$

and r. The generalized velocity vector of any ship abides by the following differential equation [Fossen, 2021]

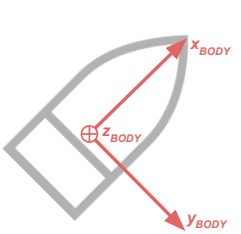
$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{\eta}) + \boldsymbol{g}_0 = \boldsymbol{\tau}_{\text{act}} + \boldsymbol{\tau}_{\text{w,c}} + \boldsymbol{\tau}_{\text{wave}} \tag{3.1}$$

Here $\boldsymbol{M} \in \mathbb{S}^6_{++}$ is the system inertia matrix, $\boldsymbol{C}(\boldsymbol{\nu}) \in \mathbb{R}^{6\times6}$ is the velocity-dependent Coriolis matrix and $\boldsymbol{D}(\boldsymbol{\nu}) \in \mathbb{R}^{6\times6}_{++}$ is a velocity-dependent damping matrix. The $\mathbb{R}^{6\times6}_{++}$ notation refers to a 6×6 positive definite matrix. Futhermore, $\boldsymbol{g}(\boldsymbol{\eta})$ is a pose dependent force vector of gravitational and buoyancy forces, while $\boldsymbol{g}_0$ is a vector containing static forces. Further, $\boldsymbol{\tau}_{\text{w,c}}$ is the torque applied to the ship by wind and currents, lumped together, $\boldsymbol{\tau}_{\text{wave}}$ is torque applied to the ship by surrounding waves, and $\boldsymbol{\tau}_{\text{act}}$ is a vector control of inputs. The total torque is commonly denoted, component-wise as $\boldsymbol{\tau} = [X, Y, Z, K, M, N]^T$. This should be interpreted as X being a force acting in the direction of the surge, Y being a force acting in the direction of sway, and Z being a force acting in the heave direction. Furthermore, the torques K, M, and N should be interpreted as torques around these same axes [Fossen, 2021]. This can also be seen in Figure 3.1.

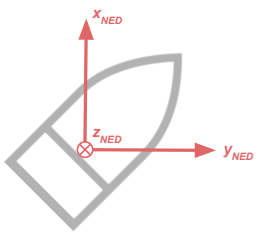
Figure 3.1 Figure showing the 6 DOF of a marine ship; surge u, sway v and heave w and angular velocities p, q, and r. Photo: Saab Kockums / Glenn Pettersson.

Coordinate Frames

It should be noted that the matrices and vectors in Equation (3.1) depend on the frame of reference used. In this thesis, two primary frames are used, as shown in Figure 3.2. The first frame is the *BODY* frame. This frame is aligned with the geometry of the craft, with the first axis spanning from the aft to the fore of the craft. The second axis is aligned from port to starboard, and the third axis is pointed ”downwards”, into the water, see Figure 3.2a. This is the system in which the generalized velocity vector $\boldsymbol{\nu}$ will be represented. The second frame is in this thesis used for expressing the position of the craft in the pose vector $\boldsymbol{\eta}$, the *NED* (North-East-Down) frame. This frame has the x-axis pointed north, the y-axis pointed east and shares the z-axis with the BODY system, see Figure 3.2b [Fossen, 2021]. The different degrees of freedom and the coordinate systems used are summarized in Table 3.1.



(a) BODY frame. The x-axis points from aft to fore, and the y-axis points from port to starboard. The z-axis points ”downwards”.



(b) NED frame. The x-axis points northwards, and the y-axis eastwards. The z-axis points ”downwards”.

Figure 3.2 Illustration of the BODY and NED (North-East-Down) frames, looking at a vessel from above.

Table 3.1 Table summarizing the 6 degrees of freedom of a marine craft and which coordinate systems they are measured in. Table adapted from [Fossen, 2021].

NED	BODY		
Position Angle	Velocity Angular Velocity	Forces Moments	Motion
x	u	X	surge
y	v	Y	sway
z	w	Z	heave
ϕ	p	K	roll
θ	q	M	pitch
ψ	r	N	yaw

The NED pose vector $\boldsymbol{\eta}$ and the BODY velocity vector $\boldsymbol{\nu}$ are coupled by the following differential equation

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}(\boldsymbol{\eta})\boldsymbol{\nu} \tag{3.2}$$

where $\boldsymbol{J}(\boldsymbol{\eta})$ is a transformation matrix defined according to

$$\boldsymbol{J}(\boldsymbol{\eta}) = \begin{bmatrix} \boldsymbol{R}(\boldsymbol{\eta}) & \boldsymbol{0}^{3\times3} \\ \boldsymbol{0}^{3\times3} & \boldsymbol{T}(\boldsymbol{\eta}) \end{bmatrix}, \tag{3.3}$$

where $\boldsymbol{R}(\boldsymbol{\eta}) \in \mathrm{SO}(3)$ is a rotational matrix and $\boldsymbol{T}(\boldsymbol{\eta}) \in \mathbb{R}^{3\times3}$ is a transformation matrix of the Euler angles [Fossen, 2021].

3-DOF Maneuvering Models

For maneuvering purposes, [Fossen, 2021] proposes to simplify the 6-DOF model into a 3-DOF model in terms of the surge u, sway v and yaw r. This restricts the motion of the ship to the surface, ignoring wave height, pitch and roll. When navigating in calm waters like in an archipelago, this is likely to be a good simplification, since the other degrees of freedom are mostly excited in the presence of waves. Thus the pose vector is redefined to $\boldsymbol{\eta} = [x, y, \psi]^T$ and the generalized velocity vector is redefined to $\boldsymbol{\nu} = [u, v, r]^T$. Furthermore, it is common to assume that all currents are constant and *irrotational*, meaning they are of the form $\boldsymbol{\nu}_c = [u_c, v_c, 0]^T$. Under that

assumption, the relative velocity $\boldsymbol{\nu}_r = \boldsymbol{\nu} - \boldsymbol{\nu}_c$ is introduced in order to simplify further. Under these conditions, the equations (3.1), (3.2) and (3.3) are reduced to the following equation [Fossen, 2021].

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R}(\psi)\boldsymbol{\nu}_r \tag{3.4a}$$

$$\boldsymbol{M}\dot{\boldsymbol{\nu}}_r + \boldsymbol{C}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r + \boldsymbol{D}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r + \boldsymbol{g}(\boldsymbol{\eta}) + \boldsymbol{g}_0 = \boldsymbol{\tau}_{\text{act}} + \boldsymbol{\tau}_{\text{w,c}} + \boldsymbol{\tau}_{\text{wave}} \tag{3.4b}$$

where $\boldsymbol{R}(\psi) \in \text{SO}(3)$ is a rotational matrix about the z axis

$$\boldsymbol{R}(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the inertia matrix $\boldsymbol{M} \in \mathbb{S}^3_{++}$ is constant and composed of two parts, rigid-body components as well as a hydrodynamic component, $\boldsymbol{M} = \boldsymbol{M}_{\text{RB}} + \boldsymbol{M}_{\text{A}}$. Similarly, the Coriolis matrix $\boldsymbol{C} \in \mathbb{R}^{3\times3}$ is composed of a rigid-body and a hydrodynamic component $\boldsymbol{C}(\boldsymbol{\nu}_r) = \boldsymbol{C}_{\text{RB}}(\boldsymbol{\nu}_r) + \boldsymbol{C}_{\text{A}}(\boldsymbol{\nu}_r)$, however it is skew-symmetric. Finally, the damping matrix $\boldsymbol{D}(\boldsymbol{\nu}) \in \mathbb{R}^{3\times3}_{++}$ is composed of a linear and nonlinear component, $\boldsymbol{D}(\boldsymbol{\nu}) = \boldsymbol{D}_{\text{l}} + \boldsymbol{D}_{\text{n}}(\boldsymbol{\nu}_r)$. The matrices are detailed in [Fossen, 2021], and contain linearization coefficients of the nonlinear mappings to the general velocities and their squares, from the forces X, Y, and N acting upon the ship. The model contains up to 22 parameters [Fossen, 2021].

Currents, Wind, and Waves. The external forces of currents and winds are such that they vary slowly with time. Thus, it is reasonable to try to estimate them as constant offsets in the model, which could be slowly adapted over the course of an experiment

$$\boldsymbol{\tau}_{\text{w,c}} = \begin{bmatrix} X_{\text{w,c}} \\ Y_{\text{w,c}} \\ N_{\text{w,c}} \end{bmatrix} \tag{3.5}$$

where $X_{\text{w,c}}$ and $Y_{\text{w,c}}$ are forces in the x and y-directions. Further, $N_{\text{w,c}}$ is a constant torque on the ship, which can be interpreted as the wind catching the vessel at an angle, and inducing a rotational force. When navigating along straight sections of path, this torque can be expected to be constant, as long as the wind is constant.

The behavior of waves is too complex to be modeled as any constant bias or offset, especially since they induce short-term oscillations. Furthermore, they affect the heave, pitch and roll of the vessel, which are ignored in the 3-DOF maneuvering model. Therefore, the force from waves was not accounted for, setting $\boldsymbol{\tau}_{\text{waves}} = \mathbf{0}$.

Offline Identified Model of the Piraya

A fewer-parameter model than the one discussed so far has been created and identified for the Piraya in [Ljungberg, 2021]. This is a 9-parameter model, with the quadratic damping terms mentioned previously assumed to be 0, containing only linear terms and quadratic cross terms. Since the model was intended for use at low velocities, those terms could be assumed to be negligible [Ljungberg, 2021]. Furthermore, the parameters are aggregates of multiple of the parameters found in [Fossen, 2021]. It can be written

$$\dot{\boldsymbol{\nu}} = \boldsymbol{\Psi}\boldsymbol{\nu} + \boldsymbol{\tau}_{\text{act}} = \begin{bmatrix} X_u & X_{vr}r & 0 \\ Y_{ur}r & Y_v & 0 \\ N_{uv}v & 0 & N_r \end{bmatrix} \boldsymbol{\nu} + \begin{bmatrix} X_\tau & 0 \\ 0 & Y_\tau \\ 0 & N_\tau \end{bmatrix} \begin{bmatrix} F_x \\ F_y \end{bmatrix} \tag{3.6}$$

where X_u, Y_v and N_r are linear damping terms on u, v and r, respectively. Furthermore, X_{vr}, Y_{ur} and N_{uv} couple the quadratic velocities vr, ur and uv to the generalized velocities u, v and r. Finally, X_τ, Y_τ and N_τ couple the actuator forces F_x and F_y, described in the next section, with the generalized velocity vector $\boldsymbol{\nu}$. The model was sampled at 5 Hz and will be used in simulation, since the offline system identification efforts have yielded numerical values for these parameters [Ljungberg, 2021].

Control Actuation. The Piraya is driven by a single outboard motor, see Figure 2.1b. The force generated by the motor is modeled as a single force vector applied at the motor mounting point. The force direction is determined by the angle α at which the motor is rotated, which creates a *virtual rudder* [Ljungberg, 2021]. An illustration can be found in Figure 3.3.

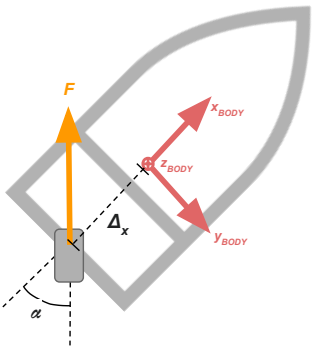


Figure 3.3 The force F from the engine is applied at the aft of the ship, generating a force along the x and y directions, as well as a torque, dependent on the virtual rudder angle α.

Commands sent to the engine and rudder are labeled $0 \leq b \leq 100$, which is the engine throttle in percent, and $-100 \leq c \leq 100$, which is the rudder angle in percent, where $c = -100$ signifies maximum rudder in the opposite direction of $c = 100$. This corresponds to the physical rudder maximum of $\pm 30°$. Thus the virtual rudder angle can be calculated as $\alpha = \frac{\pi}{6} \cdot \frac{c(t)}{100}$. The model for the forces is

$$F_x(t) = b(t)\cos\left(\frac{\pi}{6} \cdot \frac{c(t)}{100}\right) \tag{3.7}$$

$$F_y(t) = b(t)\sin\left(\frac{\pi}{6} \cdot \frac{c(t)}{100}\right) \tag{3.8}$$

Furthermore, since the engine is not placed at the vessel's center of rotation, a torque will also be generated, which is modeled as

$$\tau_\psi(t) = \Delta_x F_y(t) - \Delta_y F_x(t) \tag{3.9}$$

where Δ_x and Δ_y are the offsets of the engine with respect to the center of rotation, along x and y respectively. In this case, the engine is assumed to be placed in the center of the aft of the boat, yielding $\Delta_y = 0$, again, see Figure 3.3. The total actuating force on the craft is assumed to be proportional to this force and torque, yielding

$$\boldsymbol{\tau}_{\text{act}} = \begin{bmatrix} X_\tau F_x(t) \\ Y_\tau F_y(t) \\ N_\tau F_y(t) \end{bmatrix} \tag{3.10}$$

where X_τ, Y_τ and N_τ are parameters to be found. In this model, the *wake-effect* is ignored. The wake effect says that the output power of a propeller engine is lowered at higher velocities. Thus, here the engine is assumed to output the same force at any velocity [Ljungberg, 2021].

Discrete Piraya Model. The model from [Ljungberg, 2021] was created for the estimation of these parameters through offline SYSID. In order to aid the parameter estimation of model (3.6), the model was discretized in [Ljungberg, 2021], in the following way

$$\begin{bmatrix} \boldsymbol{\eta}_{k+1} \\ \boldsymbol{\nu}_{k+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\eta}_k \\ \boldsymbol{\nu}_k \end{bmatrix} + T_s \cdot \begin{bmatrix} \boldsymbol{R}(\psi) \\ \boldsymbol{\Psi} \end{bmatrix} \boldsymbol{\nu}_k + \begin{bmatrix} 0 \\ \boldsymbol{\tau}_{\text{act}} + \boldsymbol{\tau}_{\text{w,c}} \end{bmatrix} \tag{3.11}$$

with $\boldsymbol{\Psi}, \boldsymbol{R}(\psi), \tau_{\text{act}}$ and $\tau_{\text{w,c}}$ defined previously. In [Ljungberg, 2021], *Euler's Method* was used for discretization when obtaining the simulation model. The advantage of Euler's Method is the computational simplicity compared to other methods, and the fact that if a continuous model is linear in its parameters, the discrete model will be as well, which will come in useful later [Martinsen et al., 2022]. The disadvantage of this method is in the linear rate of convergence, $O(T_s)$, and thus the prediction accuracy for simulation of multiple time-steps will be relatively low.

For using this model with a time-step size $T_s \neq 0.2$, not corresponding to 5 Hz, the parameters in $\boldsymbol{\Psi}$ and τ_{act} have to be scaled to compensate. In [Kockum, 2022] it was found that simply scaling the coefficients linearly by the ratio between the desired and actual model frequency, $T_s/0.2$, worked well. This is later used when simulating the model in real-time.

3.2 Model Predictive Control

Model predictive control begins with taking advantage of a known model of the system we want to control. So, consider a system, with a state vector $\boldsymbol{x} \in \mathbb{R}^{n_x}$, an input vector $\boldsymbol{u} \in \mathbb{R}^{n_u}$ and an output vector $\boldsymbol{y} \in \mathbb{R}^{n_y}$, where n_x, n_u and n_y are the number of states, controls and outputs, respectively. Further assume it follows the model

$$\begin{aligned} \dot{\boldsymbol{x}} &= f(\boldsymbol{x}, \boldsymbol{u}) \\ \boldsymbol{y} &= h(\boldsymbol{x}, \boldsymbol{u}) \end{aligned} \tag{3.12}$$

where f describes the dynamics of the system and h is an output map of the system. The system is subject to constraints on the states and controls, $\boldsymbol{x}(t) \in \mathcal{X}, \boldsymbol{u}(t) \in \mathcal{U}, \forall t$ and $\boldsymbol{x}(0) = \boldsymbol{x}_0$, where $\mathcal{X}$ is the set of allowed states, $\mathcal{U}$ is the set of allowed control inputs, and $\boldsymbol{x}_0$ is the initial condition of the system [Rawlings et al., 2017]. The goal, in this case, is to construct a control signal $\boldsymbol{u}(t)$ such that the output state function $\boldsymbol{y}(t) \longrightarrow \boldsymbol{y}_{\text{ref}}(t)$ where $\boldsymbol{y}_{\text{ref}}(t)$ is some desired output of the system.

Discrete-Time Control

When MPC is implemented as an online method, the system needs to be discretized:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \hat{f}(\boldsymbol{x}_k, \boldsymbol{u}_k), \\ \boldsymbol{y}_k &= \hat{h}(\boldsymbol{x}_k, \boldsymbol{u}_k), \\ \boldsymbol{x}_0 &= \boldsymbol{x}(0), \\ \boldsymbol{x}_k &\in \mathcal{X}, \boldsymbol{u}_k \in \mathcal{U}, k = 1 \ldots N_p - 1 \end{aligned} \tag{3.13}$$

where $\boldsymbol{x}_k = \boldsymbol{x}(T_s k)$, T_s is the sampling period and $k \in \mathbb{N}$ is the iteration number. Here $\hat{f}$ and $\hat{h}$ are discretizations of the model f and output map h, respectively. Using this discretized model, MPC should generate a control sequence $\{\boldsymbol{u}_k\}_{k=0}^{N_c-1}$, giving the system the desired behavior. Here $N_p \in \mathbb{N}$ is the *prediction horizon*, the number of time steps of look-ahead in the optimization [Rawlings et al., 2017]. The principle is illustrated in Figure 3.4

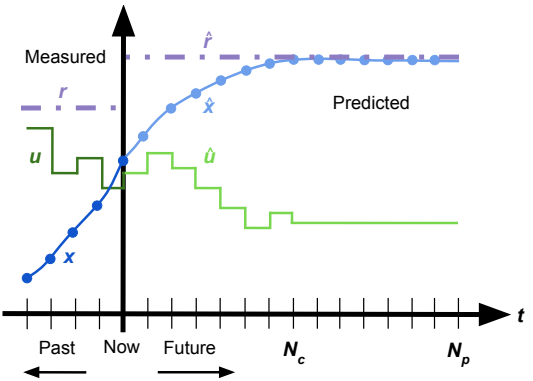


Figure 3.4 Illustration of the MPC principle, applied in every time-step. In the past, at every time-step the state x_t was measured, under the influence of the control signal u_t, driving the system toward the reference r_t. At the time "now", a sequence of control signals $\{\hat{u}_t\}_{t=0}^{N_c}$ is found to be optimal in order to drive the predicted states $\{\hat{x}_t\}_{t=0}^{N_p}$ to the reference $\hat{r}$. It can be seen that up to the control horizon N_c, one control signal is computed per time-step, and after that and up until the prediction horizon N_p, the final computed control signal is held constant. At the time "now" the reference can be seen to change.

This control sequence should achieve the control objective by minimizing the cost function, which should quantify the difference between the current output $\boldsymbol{y}_k$ and the desired output over the prediction horizon. The cost function is defined as the sum of the *stage cost function* $\ell(\boldsymbol{y}, \boldsymbol{u}, \boldsymbol{y}_{\text{ref}})$, for every predicted state. Explicitly, this is usually formulated as the following minimization problem [Rawlings et al., 2017]

$$\underset{\{\boldsymbol{u}_k\}_{k=0}^{N_c-1}}{\text{minimize}} \sum_{k=0}^{N_c-1} \ell(\boldsymbol{y}_k, \boldsymbol{u}_k, \boldsymbol{y}_{\text{ref}}) + \sum_{k=N_c}^{N_p-1} \ell(\boldsymbol{y}_k, \boldsymbol{u}_{N_c-1}, \boldsymbol{y}_{\text{ref}}) \tag{3.14a}$$

$$\text{subject to} \quad \boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{3.14b}$$

$$\boldsymbol{y}_k = h(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{3.14c}$$

$$\boldsymbol{x}_0 = \boldsymbol{x}(0), \tag{3.14d}$$

$$\boldsymbol{x}_k \in \mathcal{X}, \boldsymbol{u}_k \in \mathcal{U}, k = 1 \ldots N_p - 1 \tag{3.14e}$$

here $N_c \in \mathbb{N}$ is the *control horizon* and determines the number of control signals to calculate. The first sum in Equation (3.14a) corresponds to steps in which the control actions are to be determined by the MPC controller, while in the second sum, the cost for the remaining steps is calculated assuming that the final control action u_{N_c-1} is kept for all of the remaining iterations

Stage Cost Function

Introducing the error $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{y}_{\text{ref}}$, the stage cost function is written $\ell(\boldsymbol{e}, \boldsymbol{u})$, and is commonly picked to be a quadratic function. The cost function provides the ability to penalize the error in every state individually, as well as the magnitude of the individual control signals. This can be written

$$\ell(\boldsymbol{e}, \boldsymbol{u}) = \boldsymbol{e}^T \boldsymbol{Q} \boldsymbol{e} + \boldsymbol{u}^T \boldsymbol{R} \boldsymbol{u} \tag{3.15}$$

where $\boldsymbol{Q} \in \mathbb{R}_{++}^{n_y \times n_y}$ and $\boldsymbol{R} \in \mathbb{R}_{++}^{n_u \times n_u}$ are often picked to be diagonal matrices. Furthermore, it is common to penalize large changes in control signal between iterations, to avoid excessive tear on actuators. This is done similarly to $\boldsymbol{e}$ and $\boldsymbol{u}$, adding a quadratic cost to $\Delta\boldsymbol{u} = \boldsymbol{u} - \boldsymbol{u}_{\text{prev}}$ as follows:

$$\ell(\boldsymbol{e}, \boldsymbol{u}, \Delta\boldsymbol{u}) = \boldsymbol{e}^T \boldsymbol{Q} \boldsymbol{e} + \boldsymbol{u}^T \boldsymbol{R} \boldsymbol{u} + \Delta\boldsymbol{u}^T \boldsymbol{P} \Delta\boldsymbol{u} \tag{3.16}$$

where $\boldsymbol{P} \in \mathbb{R}_{++}^{n_u \times n_u}$ is often a diagonal matrix [Rawlings et al., 2017]. The interpretation of the cost function should be that increasing the size of the elements in $\boldsymbol{Q}$ will encourage the controller to find solutions with lower state error $\boldsymbol{e}$. This will be accomplished with larger control signal magnitudes and changes. Conversely, if keeping the control signal small is important, increasing the size of the terms in $\boldsymbol{R}$ will accomplish this. Similar changes to performance can be achieved by augmenting the matrix $\boldsymbol{P}$, which will affect the rate of change in the control signal. It should be noted that only the relative size of the terms in the matrices matter. The MPC problem in Equation (3.14a) can now be rewritten in the following form, with $\boldsymbol{u}_f = \boldsymbol{u}_{N_c-1}$

$$\underset{\{\boldsymbol{u}_k\}_{k=0}^{N_c-1}}{\text{minimize}} \sum_{k=0}^{N_c-1} \left[\boldsymbol{e}_k^T \boldsymbol{Q} \boldsymbol{e}_k + \boldsymbol{u}_k^T \boldsymbol{R} \boldsymbol{u}_k + \Delta \boldsymbol{u}_k^T \boldsymbol{P} \Delta \boldsymbol{u}_k\right] + \sum_{k=N_c}^{N_p-1} \left[\boldsymbol{e}_k^T \boldsymbol{Q} \boldsymbol{e}_k + \boldsymbol{u}_f^T \boldsymbol{R} \boldsymbol{u}_f\right] \tag{3.17a}$$

$$\text{subject to} \quad \boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{3.17b}$$

$$\boldsymbol{y}_k = h(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{3.17c}$$

$$\boldsymbol{x}_0 = \boldsymbol{x}(0), \tag{3.17d}$$

$$\boldsymbol{x}_k \in \mathcal{X}, \boldsymbol{u}_k \in \mathcal{U}, k = 1 \ldots N_p - 1 \tag{3.17e}$$

This is solved to compute a control signal at every iteration. The initial condition $\boldsymbol{x}_0$ at every iteration is taken to be the current position, and the minimization generates a control sequence $\{\boldsymbol{u}_k^*\}_{k=0}^{N_c-1}$ which is optimal with respect to the cost function (3.17a). Of this sequence, the first control signal $\boldsymbol{u}_0^*$ is applied to the real system, while the rest is discarded. In the next control iteration, a new optimal $\boldsymbol{u}_0^*$ is calculated.

3.3 Model Predictive Path Following Control

A MPC controller like the one formulated in Equation (3.17) can be used for steering a system to way-points, or trajectory tracking. In order to make the system follow *paths* in space, the *Model Predictive Path Following Control* (MPFC) approach introduced by [Faulwasser, 2013] can be used. Here, a path is a geometric curve in the output space: $\mathcal{P} = \{\boldsymbol{y} \in \mathbb{R}^{n_y} | z \in \mathbb{R} \mapsto \boldsymbol{y} = p(z)\}$, where $z \in [0, z_{\max}]$ is a scalar *path parameter*, and p is a parameterization of the curve. The controller is tasked with advancing z, thus advancing the system along the path. This is similar to following a logged trajectory, but in contrast to trajectory following, while z evolves with time, it is not specified at what time the system should be at a specific state [Faulwasser et al., 2017]. Thus, we use $\dot{z}$, the *path velocity*, which is the derivative of the path parameter z with respect to time. The path velocity represents the speed of the vessel along the path. The controller is designed to advance the system along $\mathcal{P}$ at a variable pace, and the controller chooses $\dot{z}$ as a trade-off between accurate curve tracking and advancing along the path at a desired rate. An illustration can be seen in Figure 3.5.

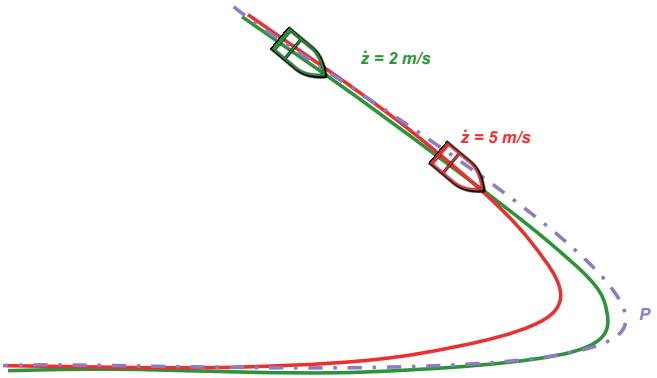


Figure 3.5 An illustration of MPFC, which relates to the trade-off of traveling at the desired speed, and making sure to stay on the path $\mathcal{P}$ (purple). The vessel in red keeps a higher speed but does not stay on the path as well as the slower vessel in green. By varying the speed dynamically, based on the path ahead, one can travel quickly when possible, like along straight path sections, while still sticking to corners by lowering the speed.

MPFC objective

There are two mathematical formulations of MPFC given by [Faulwasser, 2013]. The first one emphasizes reaching the goal, and the path velocity is only limited by the ability of the controller to keep the system on track. That is formulated as follows

$$\lim_{t\to\infty} \quad p(t) - p(z(t)) = 0, \tag{3.18a}$$

$$\lim_{t\to\infty} \quad z(t) - z_{\max} = 0, \qquad \dot{z}(t) > 0, \tag{3.18b}$$

where $\dot{z}(t) > 0$ ensures forward movement along the path. Further, the second variation of MPFC replaces Equation (3.18b) with

$$\lim_{t\to\infty} \quad \dot{z}(t) - \dot{z}_{\text{ref}} = 0, \qquad \dot{z}(t) > 0 \tag{3.19}$$

where $\dot{z}_{\text{ref}}$ is a specified speed the system should aim to keep along the path. This version of MPFC will aim to keep the speed at $\dot{z}_{\text{ref}}$, but make adjustments in order to fulfill the condition of staying on the path.

Virtual States and Virtual Control

In order to use MPC for path-following, the system in Equation (3.12) is extended with one *virtual* input $\tilde{u}$ and multiple virtual states, $\tilde{\boldsymbol{x}} = \left[z, \dot{z}, \ldots, z^{(\hat{r})}\right]^T$. Here, $\hat{r}$ is the *relative vector degree* of the system [Faulwasser, 2013]. The relative vector degree essentially specifies the *order* of the system, the number of integral steps between the control input, $\boldsymbol{u}$, and the output state $\boldsymbol{y}$. The virtual input is connected to the virtual states by $z^{(\hat{r}+1)} = \tilde{u}$. This should be interpreted as an integrator chain of length $\hat{r}$ from the virtual input $\tilde{u}$ to the virtual state z. In MPFC, the virtual states $\tilde{\boldsymbol{x}}$ represent the path parameter and its derivatives. From the discussion in Section 3.3 we have the constraints $z \in [0, z_{\max}]$ and $\dot{z} > 0$. The other virtual states and the virtual input $\tilde{u}$ are unconstrained. Using this, the continuous time model Equation (3.12) is extended as follows,

$$\dot{\boldsymbol{x}}^e = f^e(\boldsymbol{x}^e) = \begin{pmatrix} \dot{\boldsymbol{x}} \\ \dot{\tilde{\boldsymbol{x}}} \end{pmatrix} = \begin{pmatrix} f(\boldsymbol{x}, \boldsymbol{u}) \\ \tilde{f}(\tilde{\boldsymbol{x}}, \tilde{u}) \end{pmatrix} \tag{3.20a}$$

$$\dot{\boldsymbol{y}}^e = h^e(\boldsymbol{x}^e) = \begin{pmatrix} \boldsymbol{y} \\ \tilde{\boldsymbol{y}} \end{pmatrix} = \begin{pmatrix} h(\boldsymbol{x}, \boldsymbol{u}) \\ \tilde{\boldsymbol{x}} \end{pmatrix} \tag{3.20b}$$

where $\tilde{f}(\tilde{\boldsymbol{x}}, \tilde{u}) = \left[\dot{z}, \ddot{z}, \ldots, z^{(\hat{r})}, \tilde{u}\right]^T$. This extended model can then be discretized with Euler's method for discrete control. In MPFC, the virtual output z, generated from these dynamics, maps to a position $p(z)$ on the path. This position is then used as a reference for the stage cost function in Equation (3.16) where we substitute $\boldsymbol{e} = \boldsymbol{y} - p(z)$ for the error. In the following then, the error signifies the difference between the vessel position and the position on the path which the controller "wants" to be at. Further, a virtual state error is introduced, based on equations (3.18b) and (3.19), as

$$\tilde{\boldsymbol{e}} = \begin{bmatrix} z - z_{\max} \\ \dot{z} - \dot{z}_{\text{ref}} \end{bmatrix} \tag{3.21}$$

which we want to drive to zero [Faulwasser, 2013]. The first component of $\tilde{\boldsymbol{e}}$ going to zero implies the vessel is at the end of the path, while the second component being zero implies the vessel is keeping the desired speed along the path. This further allows us to introduce the extended error vector and extended input vector,

$$\boldsymbol{e}^e = \begin{bmatrix} \boldsymbol{e} \\ \tilde{\boldsymbol{e}} \end{bmatrix}, \quad \boldsymbol{u}^e = \begin{bmatrix} \boldsymbol{u} \\ \tilde{u} \end{bmatrix} \tag{3.22}$$

It should be noted that $\Delta \boldsymbol{u}$ is not given an extension, since the virtual control signal $\tilde{u}$ just exists in computer memory and thus there is no actuator wear to worry about. Furthermore, extended cost matrices $\boldsymbol{Q}^e \in \mathbb{R}_{++}^{n_y+2 \times n_y+2}$ and $\boldsymbol{R}^e \in \mathbb{R}_{++}^{n_u+1 \times n_u+1}$ are introduced, with penalties as appropriate on the path parameter and path velocity, and on the control input, respectively. Using this new extended system, a new cost function containing the new virtual states can be written as follows

$$\ell(\boldsymbol{e}^e, \boldsymbol{u}^e, \Delta \boldsymbol{u}) = (e^e)^T \boldsymbol{Q}^e \boldsymbol{e}^e + (u^e)^T \boldsymbol{R}^e \boldsymbol{u}^e + \Delta \boldsymbol{u}^T \boldsymbol{P} \Delta \boldsymbol{u}$$

Finally, the new constraint sets are $\mathcal{X}^e = \mathcal{X} \times [z_0, z_{\max}] \times [0, \infty] \times \mathbb{R}^{\hat{r}-2}$ and $\mathcal{U}^e = \mathcal{U} \times \mathbb{R}$, as discussed previously. The full MPFC formulation can be written similarly to Equation (3.17)

$$\underset{\{u_k\}_{k=0}^{N_c-1}}{\text{minimize}} \sum_{k=0}^{N_p-1} \left[(\boldsymbol{e}^e)_k^T \boldsymbol{Q}^e \boldsymbol{e}_k^e + (\boldsymbol{u}^e)_k^T \boldsymbol{R}^e \boldsymbol{u}_k^e + (\Delta \boldsymbol{u}^e)_k^T \boldsymbol{P}^e \Delta \boldsymbol{u}_k^e \right] \tag{3.23a}$$

$$\text{subject to} \quad \boldsymbol{x}_{k+1}^e = f^e(\boldsymbol{x}_k^e, \boldsymbol{u}_k^e), \tag{3.23b}$$

$$\boldsymbol{y}_k^e = h^e(\boldsymbol{x}_k^e, \boldsymbol{u}_k^e), \tag{3.23c}$$

$$\boldsymbol{x}_0^e = \left[\boldsymbol{x}(0)^T, z_0, 0, \ldots, 0 \right]^T, \tag{3.23d}$$

$$\boldsymbol{x}_k^e \in \mathcal{X}^e, \boldsymbol{u}_k^e \in \mathcal{U}^e, k = 1 \ldots N_p - 1 \tag{3.23e}$$

where z_0 at every iteration is chosen to be the path parameter which is the minimizer of $||\boldsymbol{x}_0 - p(z_0)||_2$, i.e. the position on the path which is closest to the current position of the system [Faulwasser et al., 2017]. As before, $\boldsymbol{x}_0$ is chosen as the current position of the system every iteration. In the rest of the thesis, this is the MPFC formulation used, and the superscript e of the extended system is dropped for the sake of notational brevity. In Figure 3.6 the solution to one iteration of an example of prediction along a predicted path can be seen.

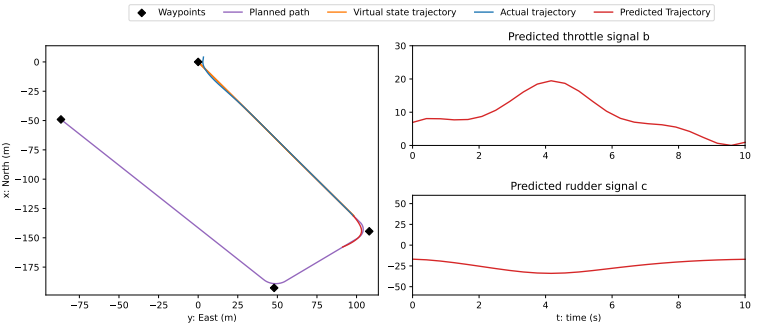


Figure 3.6 Illustration of the solution of an iteration of MPFC. To the left the planned path is shown in purple, along with the trajectory already traveled in blue. Part of the virtual state trajectory can be seen in the top of the left picture, where the orange indicates what the virtual state position $p(z)$ was while the vessel was off the path. In red, the predicted trajectory of the MPFC solution can be seen sticking close to the planned path. Finally, in the two right plots, the throttle (upper right) and rudder (lower right) signals needed to achieve this trajectory are shown.

3.4 Nonlinear Model Predictive (Path Following) Control

In *Nonlinear Model Predictive Control* (NMPC), the principle of MPC detailed above is applied to a non-linear system, i.e., a system where the model function f is non-linear. Similarly, when applying the MPFC formulation to the control of a non-linear system, it becomes a *Nonlinear Model Predictive Path Following Control* (NMPFC) problem. In a nonlinear setting, the same method discussed in Section 3.3 is used, but without guarantees of global optimality of the computed solution to the minimization problem, $\{\boldsymbol{u}_k^*\}_{k=0}^{N_c-1}$. This is due to the fact that a non-linear f will give rise to a non-convex optimization problem in most instances. In a non-convex optimization problem, there are multiple points of local minima, as compared to a convex optimization problem, where the only minimum is the global minimum of the function [Böiers, 2010]. In practice, this requires the use of more robust optimization methods, which might increase computation time.

4

Learning-Based Ship Control

While Model Predictive Control is a powerful tool for control, it relies heavily on the model being accurate to real-world behavior. If some internal system dynamics are left out of the model, or if external disturbances are not modeled, performance is quickly degraded. Unlike a classic PID controller, there is no integral action compensating for steady-state errors in the standard MPC implementation [Rawlings et al., 2017]. In the case of USV control, our model is approximate, and there are multiple disturbances like winds, currents, and waves which interact with the system in a manner that is complex to model [Fossen, 2021]. One method for improving MPC performance is system identification, which updates the prediction model parameters in the MPC, ensuring the model fits as close to reality as possible. Further, there are multiple model-free approaches for adapting the MPC controller to account for unmodeled dynamics. One such method is Q-learning, which tries to maximize the reward received by the agent, without consideration of the real-world meaning of parameters.

4.1 System Identification

In system identification, the parameters of a model are calculated from the real-world behavior of the system. This necessitates the creation of a mathematical model $f_{\boldsymbol{\theta}}$, which is expected to predict the behavior of the system given the correct set of parameters $\boldsymbol{\theta}$. Furthermore, data on

real-world behavior has to be collected. In the *Prediction Error Method* (PEM) [Åström, 1981], the model is used to predict the next state of the system, $\boldsymbol{x}^+ = f_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{u})$ given the current state and control signal $\boldsymbol{x}$ and $\boldsymbol{u}$. Here the subscript $\boldsymbol{\theta}$ indicates that the prediction depends on the model parameters. The difference between the predicted position and the next measured position can then be used to calculate a new parameter vector $\boldsymbol{\theta}^+$, which more accurately describes the behavior of the system.

Online Prediction Error Method

Once a model has been determined and an initial guess of the parameters $\boldsymbol{\theta}$ is acquired, an online system identification algorithm can update these parameters in order to reduce the mismatch between the behavior predicted by the model, and the observed behavior of the system. Given a measured state $\boldsymbol{x}_t$ with control signal $\boldsymbol{u}_t$, as well as the next measured state $\boldsymbol{x}_{t+1}$ and the predicted state $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)$, the prediction error is

$$\boldsymbol{\varepsilon}_t = \boldsymbol{x}_{t+1} - f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t) \tag{4.1}$$

Identifying the parameters $\boldsymbol{\theta}$ that minimize the prediction error is an example of a *non-linear least squares* problem (NLS), where the goal is to minimize the squared error [Martinsen et al., 2020]

$$\phi(\boldsymbol{\theta}) = \boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon} \tag{4.2}$$

which for a batch of prediction errors, $\mathcal{E} = \{\boldsymbol{\varepsilon}_t\}_{t=B-W}^{B}$, is a NLS problem

$$\min_{\boldsymbol{\theta}} \ \phi(\boldsymbol{\theta}) = \min_{\boldsymbol{\theta}} \ \sum_{t=B-W}^{B} \boldsymbol{\varepsilon}_t^T \boldsymbol{\varepsilon}_t = \min_{\boldsymbol{\theta}} \ \sum_{t=B-W}^{B} ||\boldsymbol{x}_{t+1} - f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)||^2 \tag{4.3}$$

Here, B is the batch size, and W the window size. The batch size determines the number of new data points to be gathered before each update, while the window size determines the total number of data points to be kept and used for the update. After a parameter update, the prediction errors corresponding to these old data points are then recomputed using Equation (4.1), but using the updated parameters $\boldsymbol{\theta}^+$ for prediction. Thus, when $W > B$, the method ensures that not only does the model fit the new data, but also the old data. NLS problems are commonly solved using the Gauss-Newton method [Böiers, 2010]

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta_f (\boldsymbol{J}_f^T \boldsymbol{J}_f + \lambda_f \boldsymbol{I})^{-1} \boldsymbol{J}_Q^T \mathcal{E} \tag{4.4}$$

where $\lambda_f \boldsymbol{I}$ is a regularization term ensuring the positive definiteness of the (modified) matrix hessian $\boldsymbol{H}_f = \boldsymbol{J}_f^T \boldsymbol{J}_f + \lambda_f \boldsymbol{I}$. Moreover, β_f is the *learning rate* and the Jacobian J_f and batch of errors $\mathcal{E}$ are defined as

$$\boldsymbol{J}_f = \begin{bmatrix} \nabla_{\boldsymbol{\theta}} \boldsymbol{\varepsilon}_1(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{x}_1) \\ \nabla_{\boldsymbol{\theta}} \boldsymbol{\varepsilon}_2(\boldsymbol{x}_1, \boldsymbol{u}_1, \boldsymbol{x}_2) \\ \vdots \\ \nabla_{\boldsymbol{\theta}} \boldsymbol{\varepsilon}_T(\boldsymbol{x}_{T-1}, \boldsymbol{u}_{T-1}, \boldsymbol{x}_T) \end{bmatrix}, \quad \mathcal{E} = \begin{bmatrix} \boldsymbol{\varepsilon}_1 \\ \boldsymbol{\varepsilon}_2 \\ \vdots \\ \boldsymbol{\varepsilon}_T \end{bmatrix} \tag{4.5}$$

Note that if the model is linear in the parameters, $f_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t) = \boldsymbol{Y}(\boldsymbol{x}_t, \boldsymbol{u}_t)\boldsymbol{\theta}$, the gradient is $\nabla_{\boldsymbol{\theta}} \boldsymbol{\varepsilon}_t(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1}) = -\boldsymbol{Y}(\boldsymbol{x}_t, \boldsymbol{u}_t)$. So the update step reduces to the modified Newton Method for such systems. With $\lambda_f = 0$, and $\beta_f = 1$ this method should find the optimal $\boldsymbol{\theta}$ in just one step [Böiers, 2010]. However, in practice, it is necessary to set $\lambda_f > 0$, but small, to avoid trying to invert a singular expression [Martinsen et al., 2020]. This, in turn, implies the optimal $\boldsymbol{\theta}$ will not be found at every batch update, but a good approximation.

Piraya Online PEM

When running SYSID online on the Piraya, the model proposed in [Ljungberg, 2021] is used as a basis. Thus, the parameterized model $f_{\boldsymbol{\theta}}$ is chosen to be the model in Equation (3.11), with parameters $\boldsymbol{\theta}_{\text{model}}$

$$\boldsymbol{\theta}_{\text{model}} = \begin{bmatrix} X_u & X_{vr} & X_\tau & Y_{ur} & Y_v & Y_\tau & N_{uv} & N_r & N_\tau \end{bmatrix}^T \tag{4.6}$$

Environmental bias vector. In addition to identifying the model parameters, a vector of environmental biases is also estimated in order to help with the prediction of the model state in the case where disturbances such as wind and currents are present, corresponding to Equation (3.5). This is done by introducing the parameterized bias vector

$$\boldsymbol{\nu}_{\text{bias}} = \begin{bmatrix} u_{\text{b}} \\ v_{\text{b}} \\ r_{\text{b}} \end{bmatrix} \tag{4.7}$$

It is important to note that in the model Equation (3.5) the bias is modeled as two forces and a torque, while here a generalized velocity vector is used to model the disturbance. Thus the bias is introduced into the model as

$$\dot{\boldsymbol{\eta}} = R(\psi)\boldsymbol{\nu} + \boldsymbol{\nu}_{\text{bias}} \tag{4.8}$$

The reason for this is the rotational invariance gained by modeling the bias this way in NED. Since no rotational matrix is applied to $\boldsymbol{\nu}_{\text{bias}}$, any identified disturbance will still be valid after the USV makes a turn. The biases are model parameters $\boldsymbol{\theta}_{\text{bias}} = \{u_b, v_b, r_b\}$, and are to be learned on-line using system identification and reinforcement learning. Thus, the set of parameters to be identified using SYSID online is

$$\boldsymbol{\theta}_{\text{SYSID}} = \begin{bmatrix} \boldsymbol{\theta}_{\text{model}}^T & \boldsymbol{\theta}_{\text{bias}}^T \end{bmatrix}^T \tag{4.9}$$

4.2 Reinforcement Learning Based Model Predictive Control

As previously mentioned, the performance of any MPC controller is highly dependent on the accuracy of the model used for prediction. In the previous section, it was outlined how to adjust model parameters based on real-world data in order to more closely match the real system. However, when the mathematical model is known to be approximate, the performance improvements possible by this method is limited by which dynamics are modeled. In order to increase performance beyond the limits of the modeled dynamics, Reinforcement Learning can be used to improve the performance of the controller in a model-free manner [Gros and Zanon, 2020].

This section is outlined as follows. First, the idea of Reinforcement Learning and the Q-learning algorithm are introduced in a discrete-space context. This is not strictly necessary for the thesis, but is shown in Appendix A. Then, Function Approximation is described, which allows the usage of Reinforcement Learning concepts in continuous space, and thus approximate Q-learning is outlined. After that, the manner in which this can be combined with MPC, as shown in [Gros and Zanon, 2020], is described.

Discrete Reinforcement Learning

Reinforcement Learning is most often introduced in a discrete-space setting. While discrete RL is not used in this thesis, the notation and concepts introduced in that setting carry over to continuous-space RL. The first important concept is the *state* s, which encompasses all information about an agent at a given time. Furthermore, an *action* a is any decision an agent can make which changes the state. In RL, a *reward* $R(s)$ is introduced, which is the *ground truth*, and the RL agent wishes to maximize the reward received over an entire run, called *value*. The value is measured by a *value function* $V_\pi(s)$ and signifies the total reward received when starting from state s and taking actions according to the *policy* $\pi(s)$ afterwards. Finally, the *action-value* function or Q-function, $Q_\pi(s, a)$ similarly measures the value of the state s, if the action a is taken, and the policy $\pi(s)$ is followed afterwards [Sutton and Barto, 2018]. Details on how this can be turned into a learning-method can be found in Appendix A.

Function Approximators in Continuous Space

Parameterized Q-Function. The methods discussed in Appendix A can be shown to converge to the optimal value- or Q-function for discrete sets of states $\mathcal{S}$ and of actions $\mathcal{A}$. The value- or Q-function is usually kept track of in tables and is updated by the principles of dynamic programming [Sutton and Barto, 2018]. For a continuous space of possible states and actions, that kind of method becomes unfeasible, requiring infinite memory. Instead, the method employed in continuous space RL is called *Function Approximation*, where the true value- or Q-function is approximated by a continuous function parameterized by a vector $\boldsymbol{\theta}$. In this case, since we will be applying Q-learning, the parameterized Q-function can be written as [Sutton and Barto, 2018].

$$Q(s, a) \approx \hat{Q}(s, a, \boldsymbol{\theta}) \tag{4.10}$$

Realistically, the number of state-action combinations visited in any task typically outnumbers the number of parameters in the Q-function. This implies that the approximate function will not be able to perfectly fit to and equal the true function. However, this turns out to be a strength of the function approximation method. Since the Q-function is parameterized to be continuous, the value of any state and action in a region close to a visited state s, should be similar to the value of $\hat{Q}(s, a, \boldsymbol{\theta})$. This enables the method

to *generalize* [Sutton and Barto, 2018], making informed decisions about states it has never visited before, thanks to experience of the value of other, similar states. A common method for function approximation is the usage of *artificial neural networks* (ANN) [Sutton and Barto, 2018]. That is not the method applied in this thesis however, due to ANN approximations leaving few guarantees on the resulting control scheme [Gros and Zanon, 2020]. The approximation used here is outlined in Section 4.2.

Fitting Optimal Functions. In order to approximate the true Q-function, the *Mean Square Error* (MSE) of a batch of observed transitions $\{(s_t, a_t, s_t^+)\}_{t=0}^{B}$ is measured, and is to be minimized

$$\text{MSE}_Q(\boldsymbol{\theta}) = \frac{1}{B}\sum_{t=0}^{B}\left[Q_\pi(s_t, a_t) - \hat{Q}(s_t, a_t, \boldsymbol{\theta})\right]^2 \tag{4.11}$$

where $\hat{Q}(s_t, a_t, \boldsymbol{\theta})$ is the parameterized function approximation of the Q-function, and $Q_\pi(s_t, a_t)$ is the *true* Q-function corresponding to the policy $\pi(s)$, which at this point is unknown [Sutton and Barto, 2018].

A common method of optimization in machine learning is *Stochastic gradient descent* (SGD) [Sutton and Barto, 2018]. In SGD, the gradient descent method is used on a single transition, with a randomly chosen subset of the data. The gradient of the function to be minimized, i.e., the MSE, will point in the direction for which the function increases the most. This method works by taking steps in the opposite direction of the gradient, where it decreases the most, with respect to this single transition. The parameter update with respect to the transition (s_t, a_t, s_t^+) is [Sutton and Barto, 2018]

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \beta\nabla_{\boldsymbol{\theta}}\text{MSE}_Q(\boldsymbol{\theta}) \\ &= \boldsymbol{\theta}_t + \beta\left[Q_\pi(s_t, a_t) - \hat{Q}(s_t, a_t, \boldsymbol{\theta})\right]\nabla_{\boldsymbol{\theta}}\hat{Q}(s_t, a_t, \boldsymbol{\theta})\end{aligned} \tag{4.12}$$

where β once again is the learning rate of the method. Now, just as in the tabular case, the actual Q-function is unknown, and the method needs to be modified into a *bootstrap* [Sutton and Barto, 2018] method in order to work with solely the estimate $\hat{Q}$. This gives rise to the name of *Semi-Gradient Methods* (SeGD), as they do not properly approximate the gradient of the MSE. In the function approximation variant of Q-learning, with *average reward* setting, SeGD uses the difference between $\hat{Q}(s_t, a_t, \boldsymbol{\theta})$ and

$R_{t+1} - \bar{R} + \hat{Q}(s_{t+1}, a_{t+1}, \boldsymbol{\theta})$ for its update step, where $\bar{R}$ is the average reward received thus far [Sutton and Barto, 2018]

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \beta\hat{\delta}\nabla_{\boldsymbol{\theta}}\hat{Q}(s_t, a_t, \boldsymbol{\theta}) \tag{4.13}$$

where

$$\hat{\delta} = \left[R_{t+1} - \bar{R} + \hat{Q}(s_{t+1}, a_{t+1}, \boldsymbol{\theta}) - \hat{Q}(s_t, a_t, \boldsymbol{\theta})\right] \nabla_{\boldsymbol{\theta}}\hat{Q}(s_t, a_t, \boldsymbol{\theta}) \tag{4.14}$$

It should be noted the resemblance that this update bears to the discrete update Equation (A.7), with $\hat{\delta}$ as in Equation (A.6), substituting $\hat{Q}$ for Q. The function approximation version of the Q-learning algorithm from [Sutton and Barto, 2018] is shown in Algorithm 1, and should be able to find the optimal Q-function, and thus the optimal policy.

Algorithm 1 Function Approximation Q-learning

Initial state s
Learning rate $\beta > 0$, Greedy parameter $\epsilon > 0$
Parameter vector $\boldsymbol{\theta}$ arbitrary
while *learning* **do**
  $p \leftarrow$ sample from $U(0, 1)$
  $a \leftarrow \begin{cases} \underset{a}{\text{argmax}}\ \hat{Q}(s, a, \boldsymbol{\theta}), & p > 1 - \epsilon \\ \text{random action}, & p < \epsilon \end{cases}$
  $R, s^+ \leftarrow \text{system}(s, a)$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta\hat{\delta}\nabla_{\boldsymbol{\theta}}\hat{Q}(s, a, \boldsymbol{\theta})$
  $s \leftarrow s^+$
end while

MPC as a Function Approximator

The previous sections have left out the important part of how to parameterize the Q-function approximation. Before doing so, we exchange the total value and reward scheme for a *total cost* and stage cost scheme. In this scheme, the stage cost is received as truth, and the total cost received over the entire run is to be minimized. The theory presented thus far still holds, as maximization and minimization are equivalent up to a sign change. Now, as part of the method presented in [Gros and Zanon, 2020], the Q-function is chosen as the MPC prediction cost, Equation (3.14a).

Furthermore, a few terms are added to the MPC total cost in order to extend the parameterization capabilities. An initial cost λ and a final cost matrix, $\boldsymbol{V}^f = \text{diag}\left(V_1^f, \ldots, V_6^f\right)$, which is used as a quadratic cost on the final predicted state error, are added. These parameters, the values in cost function, along with the SYSID parameters discussed in Section 4.1, constitute the parameter vector $\boldsymbol{\theta}$ which the RL algorithm may alter. That is,

$$\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{\theta}_{\text{SYSID}}^T & \boldsymbol{\theta}_{\text{MPC}}^T \end{bmatrix}^T \tag{4.15}$$

where $\boldsymbol{\theta}_{\text{MPC}}$ includes the initial cost λ and the diagonal elements of the cost matrices $\boldsymbol{Q}$, $\boldsymbol{R}$, $\boldsymbol{P}$ and $\boldsymbol{V}^f$. Thus, $\boldsymbol{\theta}$ contains 32 parameters. The entire Q-function parameterization is as below [Gros and Zanon, 2020] [Zanon et al., 2019]

$$\begin{aligned} Q_\theta(s,a) = \underset{\{u_k\}_{k=1}^{N_p-1}}{\text{minimize}} \; & \lambda_\theta + \sum_{k=0}^{N_p-1} \left[\boldsymbol{e}_k^T \boldsymbol{Q} \boldsymbol{e}_k + \boldsymbol{u}_k^T \boldsymbol{R} \boldsymbol{u}_k + \Delta \boldsymbol{u}_k^T \boldsymbol{P} \Delta \boldsymbol{u}_k \right] \\ & + \boldsymbol{e}_{N_p}^T \boldsymbol{V}^f \boldsymbol{e}_{N_p} \end{aligned} \tag{4.16a}$$

$$\text{subject to} \quad \boldsymbol{x}_{k+1} = f_\theta(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{4.16b}$$

$$\boldsymbol{y}_k = h(\boldsymbol{x}_k, \boldsymbol{u}_k), \tag{4.16c}$$

$$\boldsymbol{x}_0 = \left[\boldsymbol{s}^T, z_0, 0, \ldots, 0\right]^T, \tag{4.16d}$$

$$\boldsymbol{u}_0 = a, \tag{4.16e}$$

$$\boldsymbol{x}_k \in \mathcal{X}, \boldsymbol{u}_k \in \mathcal{U}, k = 1 \ldots N_p - 1 \tag{4.16f}$$

An argument for why the MPC total cost might be a good candidate for the Q-function parameterization is, that by construction, the MPC optimization objective is to minimize the total cost over the prediction horizon N_p, which could be considered as truncated version of the RL-problem of minimizing the cost over the entire run.

Second Order Q-learning based Parameter Updates

Given the Q-function parameterized by the MPFC cost function in Equation (4.16), we employ a similar method as in the system identification algorithm in Section 4.1. We use a second-order method outlined in [Zanon et al., 2019]. Instead of semi-gradient steps, we use hessian steps.

This is done with the Modified Gauss-Newton method to minimize the squared TD error as a function of the parameters $\boldsymbol{\theta}$. In practice, in order to calculate the temporal difference, $\hat{\delta}$, both $\hat{Q}_{\boldsymbol{\theta}}(s_{t+1}, a_{t+1}, \boldsymbol{\theta})$ and $\hat{Q}_{\boldsymbol{\theta}}(s_t, a_t)$ are taken as the MPC solution cost at time t and $t+1$, respectively. Given a batch of transitions $\{(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{x}_{t+1}, \boldsymbol{u}_{t+1})\}_{t=0}^{B}$, the TDs $\boldsymbol{\delta} = \{\hat{\delta}_t\}_{t=0}^{T}$ can be calculated. Then, the function to minimize is

$$\psi(\boldsymbol{\theta}) = \boldsymbol{\delta}^T \boldsymbol{\delta} \tag{4.17}$$

The NLS problem arising from minimizing the squared TD error is

$$\begin{aligned} \min_{\boldsymbol{\theta}} \ \psi(\boldsymbol{\theta}) &= \min_{\boldsymbol{\theta}} \ \sum_{t=0}^{B} \hat{\delta}_t^T \hat{\delta}_t \\ &= \min_{\boldsymbol{\theta}} \ \sum_{t=0}^{B} ||R_{t+1} - \bar{R} + \hat{Q}_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, \boldsymbol{u}_{t+1}) - \hat{Q}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, \boldsymbol{u}_t)||^2 \end{aligned} \tag{4.18}$$

The modified Gauss-Newton method applied in this case looks like

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \beta_Q (\boldsymbol{J}_Q^T \boldsymbol{J}_Q + \lambda_Q \boldsymbol{I})^{-1} \boldsymbol{J}_Q^T \boldsymbol{\delta} \tag{4.19}$$

where $\lambda_Q \boldsymbol{I}$ is a regularization term ensuring the positive definiteness of the (modified) hessian matrix matrix $\boldsymbol{H}_Q = \boldsymbol{J}_Q^T \boldsymbol{J}_Q + \lambda_Q \boldsymbol{I}$, and

$$\boldsymbol{J}_Q = \begin{bmatrix} \nabla_{\boldsymbol{\theta}} \hat{\delta}_1(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{x}_1, \boldsymbol{u}_1) \\ \nabla_{\boldsymbol{\theta}} \hat{\delta}_2(\boldsymbol{x}_1, \boldsymbol{u}_1, \boldsymbol{x}_2, \boldsymbol{u}_2) \\ \vdots \\ \nabla_{\boldsymbol{\theta}} \hat{\delta}_B(\boldsymbol{x}_{B-1}, \boldsymbol{u}_{B-1}, \boldsymbol{x}_B, \boldsymbol{u}_B) \end{bmatrix}, \quad \boldsymbol{\delta} = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_T \end{bmatrix} \tag{4.20}$$

here, the target value at time $t+1$, $R_{t+1} - \bar{R} + \hat{Q}_{\boldsymbol{\theta}}(\boldsymbol{x}_{t+1}, \boldsymbol{u}_{t+1})$ is taken to be independent of $\boldsymbol{\theta}$ [Martinsen et al., 2020]. Thus $\boldsymbol{J}_Q$ reduces to

$$\boldsymbol{J}_Q = \begin{bmatrix} \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{x}_1, \boldsymbol{u}_1) \\ \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_1, \boldsymbol{u}_1, \boldsymbol{x}_2, \boldsymbol{u}_2) \\ \vdots \\ \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(\boldsymbol{x}_{B-1}, \boldsymbol{u}_{B-1}, \boldsymbol{x}_B, \boldsymbol{u}_B) \end{bmatrix} \tag{4.21}$$

Just as in the system identification case, the matrix $\boldsymbol{J}_Q^T \boldsymbol{J}_Q$ may be singular. Thus $\lambda_Q > 0$ small is required, and the solution is not optimal.

4.3 Combining System Identification and Q-learning

Since both the SYSID and RL updates in the previous section modify the parameters $\boldsymbol{\theta}$, it is plausible that these updates will conflict. This is natural as the two algorithms have different goals. The SYSID algorithm tries to predict the next position as accurately as possible, while the RL algorithm tries to minimize the total cost of the run. While one could expect that a more accurate model would help in achieving the RL objective, this does not necessarily have to be the case. In [Zanon et al., 2019] it is shown that for a perfect action-value-space parameterization, the updates of the model parameters by system identification should not make the Q-function parameterization violate the Bellman equation. However, since we do not assume the parameterization to be perfect, the conflict must be handled.

In [Martinsen et al., 2020] the *Smallest Singular Value Projection* is proposed as a projection where the SYSID update does not impact RL performance. It projects the SYSID update into a basis consisting of the p smallest singular values of the RL update step. Intuitively, this makes sense, as the RL objective will not be sensitive to changes in this direction in the parameter space, and shows good performance in practice in [Martinsen et al., 2020]. Given the singular value decomposition of the hessian generated by the Q-learning step,

$$\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V} = \boldsymbol{H}_Q \tag{4.22}$$

a basis of the p smallest singular values is formed by the last p rows of $\boldsymbol{V}$, and we call it $\underline{\boldsymbol{V}}$. The smallest singular value projection is then given by

$$\Delta\boldsymbol{\theta}_f^S = \underline{\boldsymbol{V}}^T\underline{\boldsymbol{V}}\Delta\boldsymbol{\theta}_f \tag{4.23}$$

where $\Delta\boldsymbol{\theta}_f$ is the unmodified parameter change computed by the SYSID algorithm [Martinsen et al., 2020]. Finally, the total update step using both SYSID and RL is

$$\boldsymbol{\theta} \leftarrow \beta_Q\Delta\boldsymbol{\theta}_Q + \beta_f\underline{\boldsymbol{V}}^T\underline{\boldsymbol{V}}\Delta\boldsymbol{\theta}_f \tag{4.24}$$

where β_Q and β_f are the learning rates of the respective methods. The batch size of the respective methods B is set to be the same, in order to simplify the operation.

5

Implementation

In this chapter, an overview of the entire program flow is first shown. Then, the *Cross Track Error* (XTE) is described and used as a performance measure. After that, the symbolic and numeric framework that was used is described. This includes details on setting up the problem, how to compute the Q-function gradient, and what solvers were used. Following, there is a brief explanation of how the paths to be followed were generated. Finally, the ROS interface is outlined, and the communication between the controller and USV is described.

5.1 Program Overview

An overview of the implementation can be seen in Figure 5.1. The program is initialized with a set of way-points (WPs) that the USV is supposed to travel through. These are turned into a smooth path $\mathcal{P}$. The paths were created as part-linear, part-circular paths which approximately follow through the way-points specified. This was necessary since the path followed by MPFC needs to be at least continuously differentiable [Faulwasser et al., 2017], in order for the optimization problem to be well-defined and solvable in reasonable time. So given a set of way-points and a turning radius, a path was created according to the method outlined in Appendix B. This path is then sent to the `interface`. The interface is the center of the program, and is responsible for taking the latest sensor data $\boldsymbol{x}$ at fixed intervals T_s, and calling the `controller` with the sensor data, as well as the time elapsed since last update Δt, the current parameter vector $\boldsymbol{\theta}$ and the path to be traveled. The controller will run the NMPFC

solver, with implementation details described in Section 5.3. The solver will return the optimal control signal $\boldsymbol{u}^*$ as the solution to Equation (3.23), as well as the solution information, including the entire predicted path and control signals, $\hat{\boldsymbol{x}}, \hat{\boldsymbol{u}}$, along with the total cost and the Lagrange multipliers μ, ζ, also described in Section 5.3. The control signal is actuated on the USV, and the solution information along with the elapsed time, current parameter vector and cross-track-error χ is sent to the parameter updater. The cross-track-error is described in Section 5.2. Further, the `parameter updater` calls on the `SYSID` and `RL` objects with this information, and receives the parameter updates $\Delta\boldsymbol{\theta}_f$ and $\Delta\boldsymbol{\theta}_Q$ according to equations (4.4) and (4.19), respectively. The parameter updater then combines these using the projection in Equation (4.24), yielding a final parameter update $\Delta\boldsymbol{\theta}$, which is updated in the interface. It should be noted that in case this is not a batch update iteration cycle, the parameter change will be 0. The interface then logs all this information, sending it to be printed and plotted. While this is happening, the actuated signals lead to movement of the USV, which are reflected in the sensors, whose outputs are used in the next cycle.

5.2 Cross Track Error

The performance of the controller will be measured using the cross-track-error [Fossen, 2021], denoted χ. The XTE is defined as the distance from the vessel to the nearest point on the path, and is a measure of how well the vessel follows the path, see Figure 5.2. It can be calculated as

$$\chi = ||\boldsymbol{\eta} - p(z)||_2 \tag{5.1}$$

where $\boldsymbol{\eta}$ is the USV position, $p(z)$ is the position on the path given the path parameter z, which is chosen such that it minimizes the equation above. Furthermore, the XTE is used as the true RL cost, and is substituted for R in the RL method, Equation (4.18).

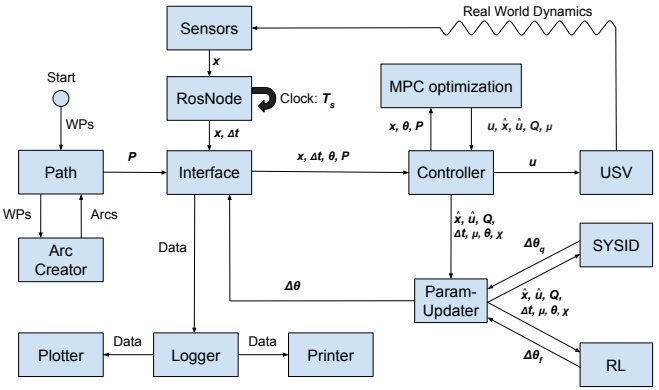


Figure 5.1 Overview of the program, and essential signals. Here `WPs` are the way-points, $\mathcal{P}$ is the path to be followed. T_s is the tick-rate of the program, Δt is the (measured) time between iterations, $\boldsymbol{x}$ is the measured state and $\boldsymbol{\theta}$ is the parameter vector. Furthermore, $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{u}}$ are the predicted states and controls respectively, Q is the MPC cost and μ are the Lagrange multipliers of the optimization problem. Moreover, $\boldsymbol{u}$ is the control signal sent to the USV and χ is the cross-track error. Finally, $\Delta\boldsymbol{\theta}$, $\Delta\boldsymbol{\theta}_f$ and $\Delta\boldsymbol{\theta}_q$ are the combined parameter update, parameter update due to SYSID and parameter update due to RL, respectively. `Data` contains all of these signals.

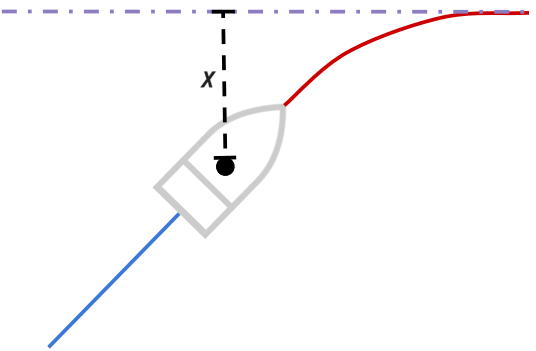


Figure 5.2 Illustration of the XTE, χ. It is defined as the closest distance between that path and vessel.

5.3 CasADi Optimization

The tool used for the algebraic formulation of, and numerical solution of the optimization problems is CasADi, an open source tool for numerical optimization and algebraic differentiation [Andersson et al., 2019]. CasADi uses computational graphs in a symbolic framework, with algebraic differentiation to be able to calculate arbitrary derivatives, Jacobians and Hessians. This information can then be used to interface with various numerical solvers that can take advantage of this information, such as QRQP [Andersson and Rawlings, 2018] or the external IPOPT [Wächter and Biegler, 2006]. CasADi is written in C++, and has interfaces to Python and Matlab. Thus, all code was written in Python, with the performance-critical optimization done through native CasADi C++ code.

NMPFC in CasADi

The problem is formulated through the low-level CasADi interface `nlpsol`, based on the lecture [Mehrez, 2019]. Symbolic `SX` variables are created for all of the states, virtual states and control signals, in all time-steps. Furthermore, symbolic states were created for the extended parameter vector $\boldsymbol{\theta}$, which allowed differentiation of the model $f_{\boldsymbol{\theta}}$ and the Q-function $Q_{\boldsymbol{\theta}}$ with respect to the parameters. Then, model constraints were expressed as equality constraints of the form $f_{\boldsymbol{\theta}}(\boldsymbol{x}_k, \boldsymbol{u}_k) - \boldsymbol{x}_{k+1}$, set to zero, for $k = 0, \dots, N_p$. Similarly, for the virtual states, the integrator dynamics $\tilde{f}(\tilde{\boldsymbol{x}}_k, \tilde{u}_k) - \tilde{\boldsymbol{x}}_{k+1}$, were set as equality constraints for $k = 0, \dots, N_p$. As a side effect, this turned the extended states $\{\boldsymbol{x}_k^e\}_{k=0}^{N_c-1}$ into decision variables along with the extended control signals $\{\boldsymbol{u}_k^e\}_{k=0}^{N_c-1}$. This makes the problem sparser, hopefully improving solution times. The reference path is then initialized as a function of the path parameter z, introduced as another `SX` variable. Using these symbolic variables, the MPC cost according to Equation (3.23) was computed as a symbolic function of these variables and the path, and set as optimization objective. Finally, the constraints are set by creating numeric vectors corresponding to the lower and upper bounds of the relevant expressions, with the lower bound set equal to the upper bound in case of equality constraints. The constraint vectors are built separately with respect to constraints on decision variables, $\boldsymbol{x}, \boldsymbol{u}$, and constraints on expressions, such as the model equations. The order of the variables in the constraint vector should be the same as the order in which the variables are input to the solver. With this setup, the NMPFC solution can be calculated by inputting

the current state and model parameters to the solver, yielding numerical values for all symbolic variables and Lagrange multipliers μ^* and ζ^*.

Q-function Gradient

The Q-function is as mentioned chosen to be the total MPC cost. In order to make use of the SeGD methods shown in Section 4.2, it is necessary to compute the gradient of the Q-function with respect to the parameter vector, $\nabla_\theta Q_\theta(\boldsymbol{x}, \boldsymbol{u})$. When doing SYSID, the gradient with respect to the model, $\nabla_\theta f_\theta$ was easily computed by differentiating the model equations. This is not the case here, since Q_θ is defined as the solution to a minimization problem (Equation (4.16)). The methodology is given by [Zanon et al., 2019], where the authors showed that the relevant gradient can be computed using the *Lagrangian* of the optimization problem. The Lagrangian is given as

$$\begin{aligned} \mathcal{L}_\theta(\boldsymbol{x}, \boldsymbol{u}, \mu, \zeta) = V_\theta^f(\boldsymbol{x}_{N_p}) + \sum_{k=0}^{N_p-1} \ell_\theta(\boldsymbol{x}_k, \boldsymbol{u}_k, \Delta\boldsymbol{u}_k) + \\ + \mu_0^T(\boldsymbol{x}_0 - s) + \sum_{k=0}^{N_p-1} \left[\mu_{k+1}^T(f_\theta(\boldsymbol{x}_k, \boldsymbol{u}_k) - \boldsymbol{x}_{k+1}) + \zeta_k^T g_\theta(\boldsymbol{x}_k, \boldsymbol{u}_k)\right] \end{aligned} \tag{5.2}$$

where the first row of the definition is the objective function, the MPC cost to be minimized, and the second row is the cost of violating the optimization problem constraints, such as the initial condition, model update equation, and state constraints (g_θ) [Böiers, 2010]. The variables $\{\mu_k, \zeta_k\}_{k=0}^{N_p-1}$ are the *Lagrange Multipliers* of the problem. It can be shown that [Zanon et al., 2019], given this formulation, the gradient of the Q-function can be computed as

$$\nabla_\theta Q_\theta(\boldsymbol{x}, \boldsymbol{u}) = \nabla_\theta \mathcal{L}_\theta(\boldsymbol{x}, \boldsymbol{u}, \mu^*, \zeta^*) \tag{5.3}$$

where μ^*, ζ^* is the collection of optimal Lagrange multipliers in the dual-primal solution of the minimization problem. It should be noted that while $\mathcal{L}_\theta$ is a complicated function, it is straightforward to differentiate through CasADi, and can be formulated as shown in [Andersson, 2023b] based on [Andersson and Rawlings, 2018].

Solver Settings

As mentioned above, CasADi supports multiple numerical solvers, including IPOPT and QRQP, among others. IPOPT is a very robust solver for nonlinear optimization, and delivers good solutions even with a lower number of iterations: Thus, the maximum number of iterations was set to 25. For those reasons, IPOPT is used in most cases for the sake of robustness and speed. QRQP is a less robust solver when solving these kinds of problems, as it is designed for solving *Quadratic Programs* [Andersson and Rawlings, 2018], but delivers more accurate sensitivity information when a solution is found. That is, the gradients computed by CasADi using the Lagrange multipliers of the QRQP solution will be more accurate than when using those provided by IPOPT [Andersson, 2023a]. Therefore, since the Q-function gradient needs to be computed in order to do RL updates, QRQP is chosen as the solver for the entire run when using RL. Since QRQP often delivers solutions with constraint violations, the number of iterations is not capped in this case, in order to allow for the solution to converge. Furthermore, if QRQP delivers a solution with constraint violations, another valid is found with IPOPT and actuated, and is not used in the RL calculations. In general, the optimization problems are solved with a *warm-start*, using the previous solution to the optimization problem as an initial guess for the next solve. This significantly decreases computation times. However, warm-starting makes the solver more likely to get stuck in local minima. Depending on the minima, this might be a good or a bad thing, which can not be predicted beforehand.

5.4 ROS Interface

The communication between the Piraya and the Python code was done through the `Robot Operating System` (ROS). ”ROS is an open-source software development kit for robotics applications” [Stanford Artificial Intelligence Laboratory et al., 2018]. While ROS offers various capabilities, the ones used for this thesis is communication through `topics` and `services`. Both of these are used for communication, transfer of data, but handle it in different ways. Communication through topics work on a *publish-subscribe* messaging pattern. In this pattern, a `publisher`, for example a sensor, will publish data to a topic, without awareness of any recipient. A `subscriber` which wants to receive that

data then, will subscribe to that same topic, and will receive data when it is published. The subscription is also without knowing who is publishing to that topic. Furthermore, the regime is asynchronous, with publishers publishing data when available, and subscribers receiving data when they actively retrieve it. When invoking a service request to for example, set the throttle, the desired code is executed remotely, and the request will wait on the code to finish executing, and a reply to be received, before continuing.

Controller Node

The controller node contains the Python code described in the earlier sections. The controller is executed at a regular rate through ROS, specified by a `tickrate` specified in the main program. The controller is invoked through the `ROSinterface` class, where the `do_iteration` method is called. Here, the latest position is processed through a subscriber on the `odo_sim_ned` or `odo_combo_ned` topics, for running a simulation vs. against the real boat, respectively. The sensor data from the Piraya are sent with 10 Hz, limited by the GPS frequency. Thus, the controller node contains a subscriber which solely saves the latest message received. When the controller is executed, the latest message is processed and the latest position is supplied as a parameter to the CasADi optimizer. When the optimization is complete, the first control pair $\boldsymbol{u}_0^*$ is logged by publishing on the topic `control_signal`, and then sent to the services `throttle_service` and `rudder_service` for execution on the Piraya. The time between publishing control signals is calculated for the system identification algorithm. After this, the program flow is the same as when running in Python simulation. It should be noted that when running on the Piraya, both the Python code and ROS node were run onboard, on the same machine. So ROS was used as a messaging layer between the Python code, and the hardware, where sensor data and control actuation was handled outside the scope of this thesis.

Piraya Node

There were two versions of the Piraya node, one for running real-time simulations and testing the communication interface, and one for communicating with the actual USV. The pose $\boldsymbol{\eta}$ and velocity $\boldsymbol{\nu}$ are sent as *Odometry* messages, native to ROS. It should be noted that ROS usually operates in the *East-North-Up* (ENU) frame, rather than NED. This meant the messages had to be converted to NED before being used.

ROS Simulation. In addition to the Python simulation, another simulation was set up as a ROS node. This node executed a model simulation step at regular intervals specified by `gps_message_rate`, 100 Hz. In that case, the model parameters are scaled by a factor of 20, as suggested in [Kockum, 2022]. The ROS simulation published Odometry messages and behaved identically to the messages sent by the Piraya. Unlike the Piraya, however, the commands were received by subscribing to `control_signal`, rather than by executing service code.

ROS-Piraya. The node running locally on the Piraya is responsible for running the `throttle_service` and `rudder_service` calls, setting the underlying actuator set-points to the desired values. Furthermore, it collects sensor data and publishes the data to various topics. This included the data from the INS, published to `ins`, and from the GPS, published to `pir/gps`. These messages containing the raw sensor data were also processed locally, into an Odometry, containing the position and heading as measured by the GPS, as well as the velocities measured by the INS.

6

Simulation

Two sets of simulation tools were developed in order to evaluate the ideal-case performance of the controller and learning algorithms. The first ”offline” simulation tool was running the USV model as part of the same program as the controller. This is the best-case scenario in terms of algorithm performance since the real-time aspect of control is removed. In the second ”ROS” simulation tool, the USV model was run as a separate ROS node, publishing its position and subscribing to control signals, while the controller was run on another ROS node, subscribing to the USV position, and publishing the corresponding control signals. Simulating in this way allows analysis of the effect of the time delay on the control performance, while the model is still known to be correct. In this chapter, first, the simulation scenarios are shown, then all of the simulation settings are presented. Finally, the simulation results are presented, first running NMPFC, then NMPFC+SYSID, and finally, NMPFC+SYSID+RL. The Offline simulation is shown first, as the ideal case performance, and then the ROS simulation is shown.

6.1 Simulation Setup

Evaluation Missions

The missions that were used to evaluate the performance of the controller are given using a set of way-points, through which a path is created by the method described in Appendix B. The first way-point of each mission is set as (0,0). The other way-points are then given as offsets in meters from

this position. In simulation, only one mission was deemed necessary to demonstrate the capabilities of the methods. It can be seen in Figure 6.1.

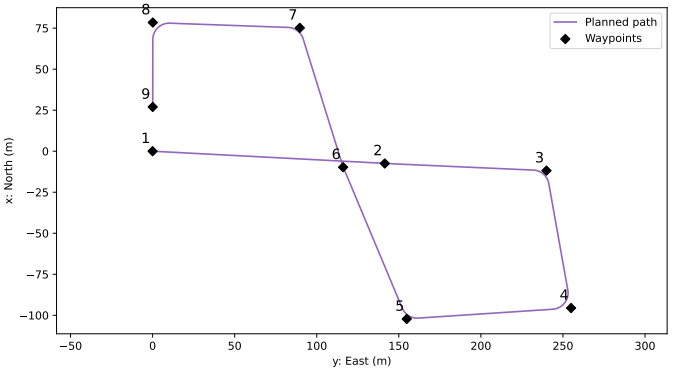


Figure 6.1 Simulation mission. Way-points: {(0, 0), (41, -116), (126, -124), (119, -15), (-51, 37), (-167, 71), (-176, 165), (-87, 177), (-9, 35)}. The path is 740 meters long.

Simulation Settings

Initial Condition and Environment. In simulation, the vessel was placed on the first way-point of the path to be followed, facing along the path, but with a random offset. The random offset was ±5 meters in both the north and east direction. Furthermore, the initial angle was offset by ±30°. Moreover, the initial velocity was 3 m/s in the forward direction. The path was generated with a turning radius of 10 m. Finally, it was possible to add a constant disturbance as a simplified model of wind or current, which in this case was set to 0.4 m/s north, 0 m/s east.

Model Parameters. The model parameters used for the actual movement of the vessel in simulation were the ones found by [Ljungberg, 2021]. The movement was simulated using Equation (3.11). When testing SYSID though, other parameter values were made up in order to investigate if the algorithm would revert them back to the real parameters. These are dubbed ”slow”, as they tend to produce lower speed solutions. The real and ”slow”

set of parameter values can be seen in Table 6.1. The parameter evolution results are shown alongside the tracking results in the later sections.

Table 6.1 Table showing the parameters used in simulation of the USV, and the ones used as an initial guess for the SYSID algorithm.

Parameter	Simulation	SYSID "slow"
X_u	-0.0478	-1.0000
X_{vr}	0.1259	0.5000
X_τ	0.0166	0.8000
Y_v	-0.2375	-0.5370
Y_{ur}	0.1526	-0.1526
Y_τ	0.0737	0.1740
N_r	-0.3191	-0.7190
N_{uv}	0.0031	0.0700
N_τ	-0.0345	-1.0000

Algorithm Parameters

The NMPFC controller was tuned to use a prediction horizon of $N_p = 50$, and $N_c = 25$, yielding a look-ahead of 10 seconds. The controls signal were evenly spread out over the prediction horizon so that from the controller's perspective, each control signal would be actuated for $2T_s$, i.e., 0.4 seconds. This was done because of considerations of unmodeled actuator dynamics, possibly making more tightly spaced actuator signals unnecessary. Furthermore, increasing the control horizon to $N_c = 50$ was found to have little benefit in simulation. In reality, however, the controller was run at 5 Hz as mentioned, and thus control signals were actually actuated every 0.2 seconds. Finally, the reference path speed was set as $\dot{z}_{\text{ref}} = 3.5$ m/s. This speed ensures maneuverability, and challenges the SYSID method, as the offline SYSID was done at 2 m/s.

Controller Tuning. All of the weight matrices used with the controller can be seen in Table 6.2. There were two tuning profiles created, an "aggressive" profile, which leads to strong control signals and low error, and a "robust" profile, which leads to larger errors but slower changes in the control signal. As mentioned in Section 3.2, the $\boldsymbol{Q}$-matrix penalizes the

states x, y, ψ, u, v and r. The first three elements correspond to penalties on $\boldsymbol{\eta} - p(z)$, i.e., the tracking error, while the latter three penalize $\boldsymbol{\nu}$, discouraging high velocities. It can be seen that there is no penalty on the heading since the calculation of the heading of the path was not implemented. Furthermore, it should be noted that there are large penalties on the magnitude of v and r. This is done to mitigate looping behavior, as the controller would often decide on a path including 180° or larger turns when the path did not call for it. Another reason for these penalties is the fact that the measurements of v were not used in experiments, discussed in Section 7.1, and thus we want to avoid states with a large amount of sway.

The matrix $\tilde{\boldsymbol{Q}}$ penalizes the virtual state error, $(z - z_f, \dot{z} - \dot{z}_{\text{ref}})$. Again, it can be seen in Table 6.2 that there is a large penalty placed on the virtual state speed error, guiding the USV to maintain the speed $\dot{z}_{\text{ref}}$, though the controller can deviate when deemed favorable, as outlined in Section 3.3. The penalty on $\dot{z} - \dot{z}_{\text{ref}}$ is larger in the robust tuning case, ensuring the USV will keep going forward even in the face of disturbances. Furthermore, there was only a very small penalty placed on the control signals in $\boldsymbol{R}$, and none on the virtual control in $\tilde{R}$. Finally, in the robust profile, a relatively large penalty was placed on the change of control signal, seen in the matrix $\boldsymbol{P}$. These penalties are introduced in order to slow down the change in the control signal, since because of the unmodeled actuator dynamics, the control signals sent will not be actuated instantaneously.

Table 6.2 Table showing two controller tuning profiles, one for aggressive control tuned for minimum cross-track-error, only used in simulation, and one for robust control, used in both simulation and experiments. "diag" signifies a square matrix with the input as diagonal elements. A dash '-' indicates no difference.

Weight Matrix	Aggressive Tuning	Robust Tuning
$\boldsymbol{Q}$	$\text{diag}\left(\begin{bmatrix} 10 & 10 & 0 & 0 & 10 & 10 \end{bmatrix}\right)$	-
$\tilde{\boldsymbol{Q}}$	$\text{diag}\left(\begin{bmatrix} 0 & 10 \end{bmatrix}\right)$	$\text{diag}\left(\begin{bmatrix} 0 & 100 \end{bmatrix}\right)$
$\boldsymbol{R}$	$\text{diag}\left(\begin{bmatrix} 0.001 & 0.01 \end{bmatrix}\right)$	-
$\tilde{R}$	0	-
$\boldsymbol{P}$	$\text{diag}\left(\begin{bmatrix} 0 & 0 \end{bmatrix}\right)$	$\text{diag}\left(\begin{bmatrix} 0.1 & 1 \end{bmatrix}\right)$

Constraints. In Table 6.3 the constraints $\mathcal{X}^e$ and $\mathcal{U}$ can be seen. The constrained signals are $u, z, \dot{z}, b$ and c. The velocity and throttle are constrained for safety reasons, to avoid losing control of the vessel, while the rudder signal is constrained for performance reasons found during experiments. Since going backward requires a gear change, the forward velocity is also constrained to not be negative. The virtual state is constrained to be on the path, with $z_{\max}$ being the length of the path, and $\dot{z} > 0$ ensures we are advancing along the path.

Table 6.3 Table showing the signal constraints

Signal	Min	Max	Unit
u	0	5	m/s
z	0	$z_{\max}$	m
$\dot{z}$	0.001	-	m/s
b	0	30	%
c	-60	60	%

Learning Tuning. In Table 6.4 the settings for SYSID and RL are shown. The learning rates β_f and β_q can be seen, as well as the number of dimensions for the projection p, the batch size B, and the SYSID window size W. Furthermore, in Table 6.5, the initial guesses for the initial cost and final cost matrix introduced in the cost function when RL is enabled, are shown. These are then adapted by the Q-learning algorithm. The initial cost λ is just a constant additive cost, while $\boldsymbol{V}_f$ should be understood to operate on the final state in the same manner as $\boldsymbol{Q}$ above, which can be seen in Equation (4.16). Another decision made at this point was to fix the rotational velocity bias $r_b = 0$. This parameter was hard to justify from a physical point of view, and also tended to degrade performance by adapting aggressively when turning.

Table 6.4 Table showing the parameters set for the SYSID and RL algorithms.

Parameter	β_f	β_q	p	B	W
Value	1	0.1	32	100	200

Table 6.5 Table of the initial guesses for the extra cost parameters used when RL was enabled.

Cost Parameter	Value
λ	10000
$\boldsymbol{V}_f$	$\text{diag}\left(\begin{bmatrix} 0 & 0 & 0 & 10 & 2000 & 2000 \end{bmatrix}\right)$

6.2 Simulation Results

The offline simulation was run at a frequency of 5 Hz, matching the model sample frequency. Here, the simulation does not proceed before receiving a control signal from the NMPFC optimization. This corresponds to the simplification of having no computation time in the controller.

Offline Simulation

The simulation performance is evaluated using the average and maximum cross-track-error, $\overline{\chi}$ and χ_{max} as well as the average and maximum NMPFC calculation time, $\overline{T}$ and T_{max}. Only the final 50 % of the data is used for the calculation of the average and max XTE, to allow for the method to have some time to learn, without affecting the final performance metric. In Table 6.6 a summary of the performance in simulation of the different controllers in various scenarios can be seen. It can be seen that with aggressive tuning, the NMPFC algorithm in an ideal case can steer the USV along the path with an average error of under 0.1 m, and a maximum error of under one meter. The robust-tuning controller achieves an average error of less than half a meter, and a maximum error under 2 m. Introducing a disturbance decreases the performance of the controller, and using the wrong model parameters decreases performance significantly, as expected. Enabling SYSID at this point, increases the performance of the controller to be the same as in the ideal case. Enabling RL on the other hand, does increase performance over the non-learning case, but substantial performance degradation is still present when compared to the ideal case. Finally, SYSID+RL still increases performance over the non-learning case, but performs worse than either method on its own.

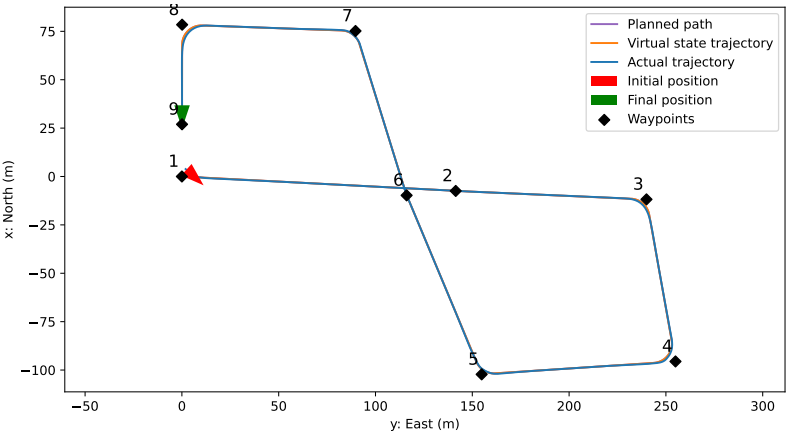
It should be noted that the computation time of bare NMPFC and NMPFC+SYSID are well below the tick-rate of 200 ms, showing that these

methods are suitable for real-time use. However, when enabling RL, since another solver is used as mentioned in Section 5.3, the computation time increases by over 10 times, both in average and worst case. Thus, at this point, it was concluded that the RL method is not ready for use in real-time. It is therefore not applied in the ROS simulation nor on the Piraya.

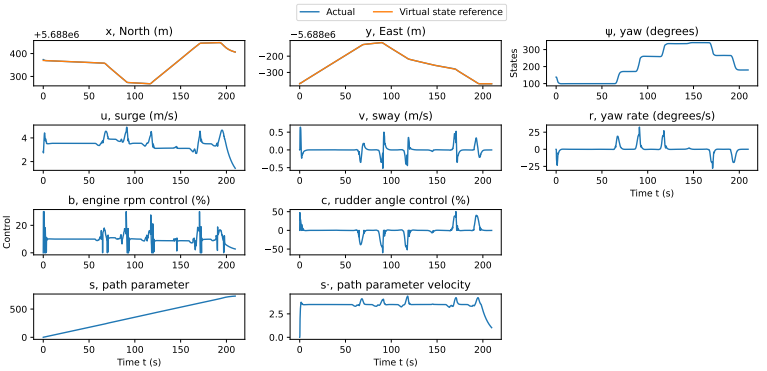
Table 6.6 Summary of the Offline simulation results. The average and maximum cross-track-error, $\overline{\chi}$ and $\chi_{\max}$ as well as the average and maximum MPC calculation time, $\overline{T}$ and $T_{\max}$ are shown. The robust and aggressive tuning parameters are shown in Table 6.2. The robust tuning is used when indicated with 'R'. When the 0.4 m/s north disturbance is present, this is marked with 'D'. Finally, if the slow SYSID parameter guess is used, this is marked with the letter 'S'.

Simulation	$\overline{\chi}$ (m)	$\chi_{\max}$ (m)	$\overline{T}$ (ms)	$T_{\max}$ (ms)
Aggressive	0.07	0.85	29	86
Robust (R)	0.48	1.81	30	111
R Disturbance (RD)	1.17	3.63	30	106
RD slow (RDS)	3.56	11.81	31	100
RDS SYSID	0.47	1.81	29	68
RDS RL	1.63	6.55	320	1210
RDS SYSID+RL	2.32	8.28	460	1260

NMPFC. The following results were obtained when running the controller without learning enabled. In Figures 6.2 and 6.3 the results are shown for the aggressive and robust controller, respectively. Here the correct model parameters were used, and no disturbance was present. Furthermore, in Figures 6.2a and 6.3a, the trajectories followed can be seen, and in Figures 6.2b and 6.3b the state evolution, control signal and cross-track-error over time can be seen. It is evident that the more aggressively tuned controller follows the path more closely, but it can also be seen that the control signal is less smooth. Therefore, as expected, the robustly tuned controller is more suitable for real-world use.

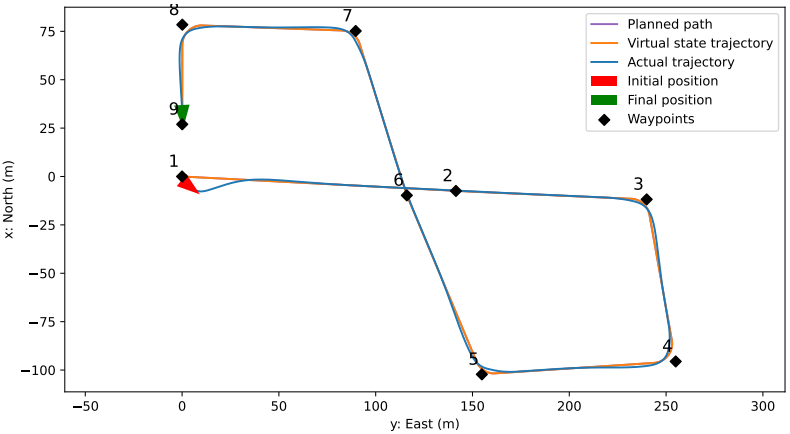


(a) The path followed by the offline simulated aggressive USV.

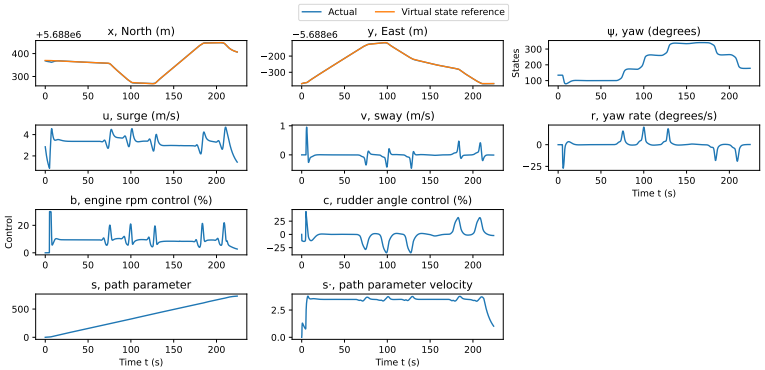


(b) Signals over time from the offline simulated aggressive USV.

Figure 6.2 Results of offline simulation of the USV with an aggressively tuned controller.



(a) The path followed by the offline simulated robust USV.



(b) Signals over time from the offline simulated robust USV.

Figure 6.3 Results of offline simulation of the USV with a robustly tuned controller.

SYSID-NMPFC. Before enabling system identification, we want to establish a sense of the performance when the model is incorrectly identified. To this end, in Figure 6.4 the control performance of the NMPFC controller with the "slow" parameters can be seen. Furthermore, a disturbance has also been introduced. It is evident that the controller does not follow the path well. Not shown is the fact that the average velocity was just 1.8 m/s.

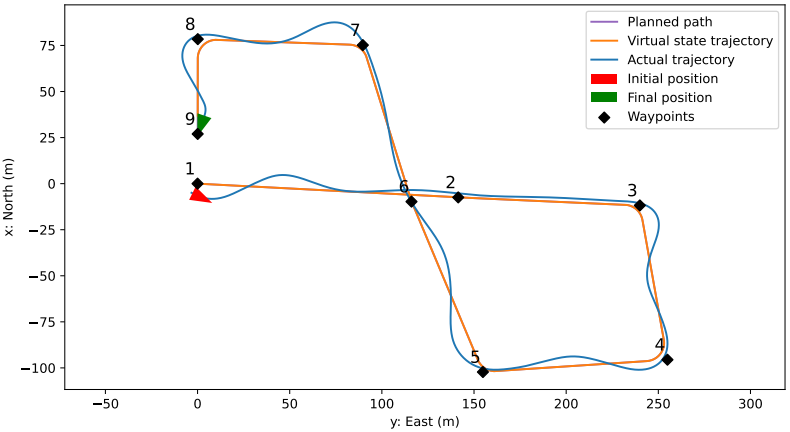
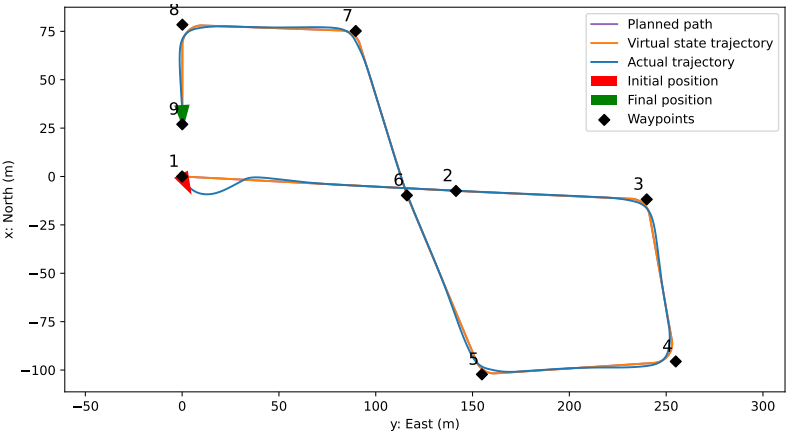
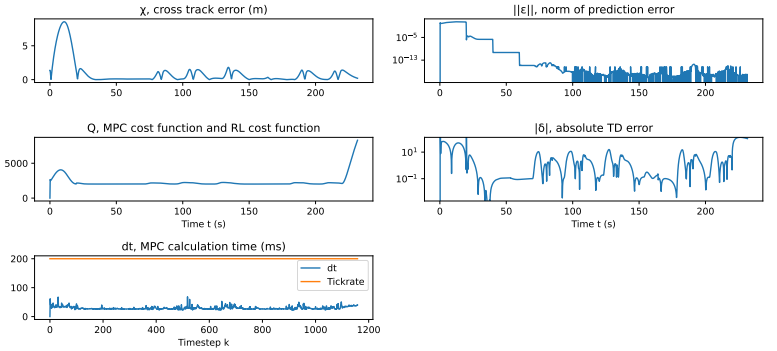


Figure 6.4 Results of offline simulation of the USV with external disturbance present, and the slow parameter guess

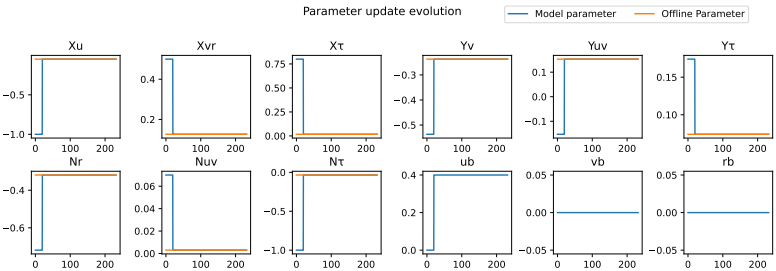
In Figure 6.5 the performance when SYSID is enabled is shown. Moreover, in Figure 6.5a the resulting path can be seen. While the first 5 seconds of the path are similar, the SYSID method quickly adjusts, and the controller afterward performs just as well as in the nominal case. The parameter evolution is shown in Figure 6.5c, and confirms that the optimal model is found within the first update iteration, at 20 s, as the theory predicts. It can also be seen that the identified disturbance does not change as the vessel turns, indicating that we successfully identified the disturbance in the NED system. Finally, in Figure 6.5b, among other things, it can be seen that the magnitude of the prediction error, $||\boldsymbol{\varepsilon}||$ goes to practically zero, and the temporal difference error δ decreases quickly after the parameters are updated. Also evident is that the calculated cost Q is lowered as the model converges.



(a) The path followed by the offline simulated USV when SYSID was enabled.



(b) Performance statistics of the USV controller when SYSID is used on the offline simulated USV.

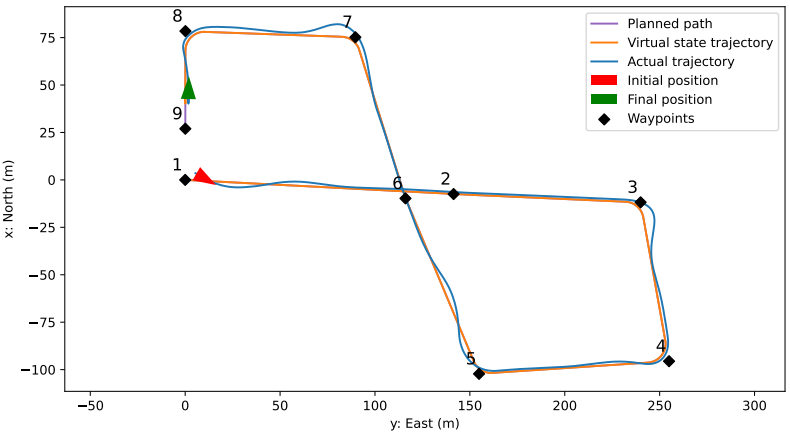


(c) The model parameter evolution using SYSID on the offline simulated USV.

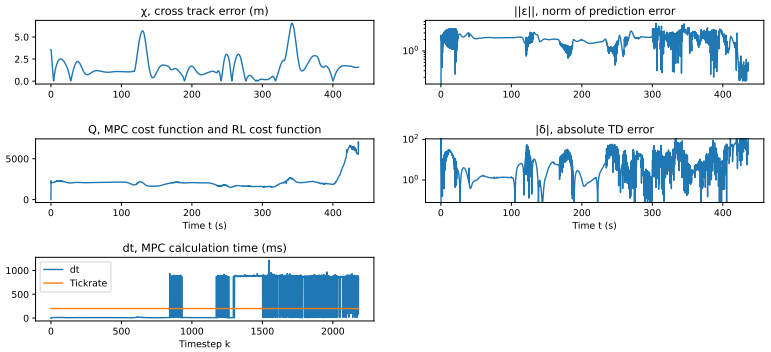
Figure 6.5 Results of offline simulation of the USV with system identification enabled. An external disturbance present, and the slow parameter guess was used.

RL-NMPFC. In Figure 6.6 the results when RL is enabled with the same scenario settings as in the previous simulation are shown. In Figure 6.6a the corresponding path can be seen. It should be noted that the controller seems to perform well when heading straight, but not as well when turning. In Figure 6.6b, it can be seen that the magnitude of the prediction error $||\boldsymbol{\varepsilon}||$ stays constant and large. While this method is not necessarily expected to converge to the correct model parameters, it is a reasonable assumption that doing so would improve performance. Furthermore, temporal difference error δ is relatively constant and low when heading straight, but increases while turning. This agrees with the resulting path, where performance was worse while turning. This might be because of the relatively limited exposure to turning behavior. Also, in the last third of the run, it can be seen that the path velocity no longer hovers around 3.5 m/s, indicating that performance was lost. Finally, it can be seen that the computation time often exceeds the tick-rate. This could be due to the fact that the RL-adaptation of the cost function changes how well the problem is posed. In a more ill-posed problem, it is feasible that the solutions require more iterations to be found. That is especially true since QRQP is not designed to solve general non-linear problems. Furthermore, the longer computation times seem to all be about equal. This is likely a result of the solver hitting the internal cap for maximum number of iterations. Now, the model parameter evolution is shown in Figure 6.6c. The RL cost and cost function parameter evolution are harder to interpret and are thus placed in Appendix C. They

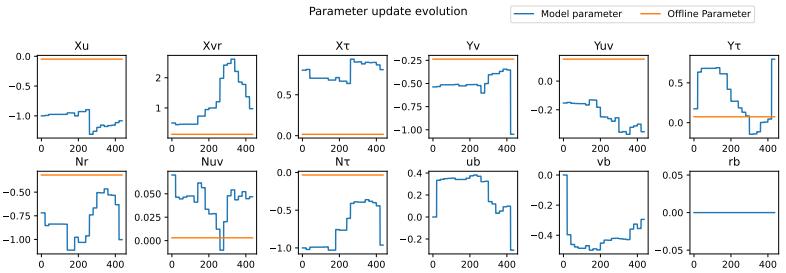
can thus be seen in Figures C.1a and C.1b, respectively. The results are difficult to interpret, but show that the RL method does not converge to the offline model. Furthermore, some cost parameters are changed to being negative, raising issues with the positive definiteness of the cost function.



(a) The path followed by the offline simulated USV when RL was enabled.



(b) Performance statistics of the USV controller when RL is used on the offline simulated USV.

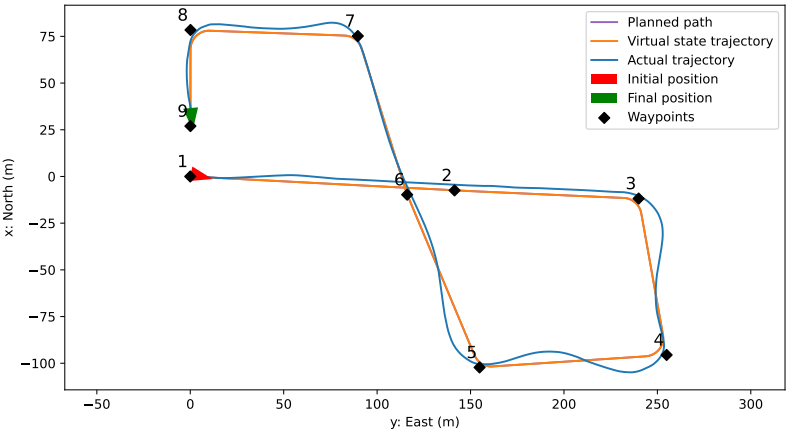


(c) The model parameter evolution using RL on the offline simulated USV.

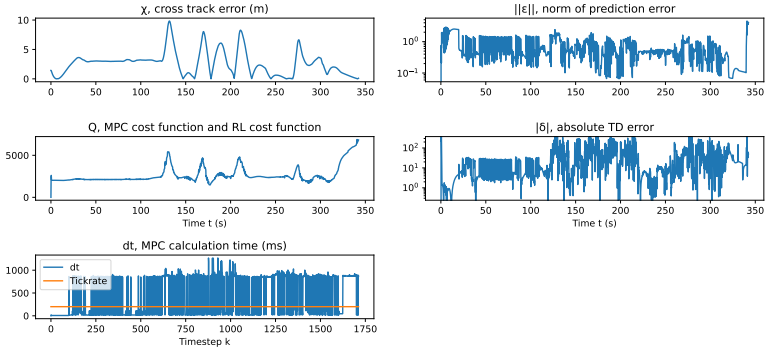
Figure 6.6 Results of simulation of the USV with Reinforcement Learning enabled. An external disturbance present, and the slow parameter guess was used.

SYSID-RL-NMPFC

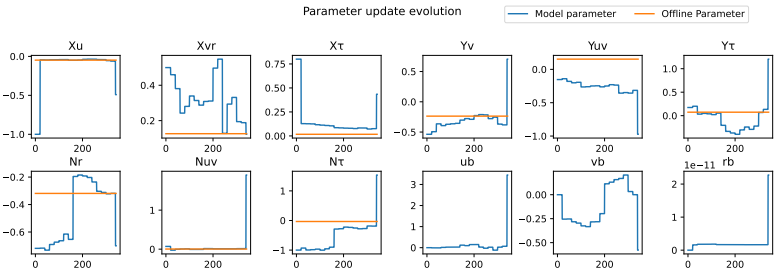
Finally, SYSID and RL were run together, hopefully combining the benefits of both methods. The results can be seen in Figure 6.7. The path shown in Figure 6.7a indicates, however, that the disturbance was not compensated for, and the turns were made later, leading to large deviations from the reference path. In Figure 6.7b the performance statistics are shown. Just as in the previous case, computation times very often exceed the tick-rate, for the same reasons that were given previously. The prediction error magnitude, in this case, is lowered initially, but only by about an order of magnitude. The temporal difference error, similarly to the last case, seems to be large in turns and low on straight sections of the path. In Figure 6.7c, it can be seen that some of the model parameters converge toward the offline parameters, for parts of the run. After that, it seems that the system identification was overruled by the reinforcement learning method. Just as before, the RL cost and cost function parameter's evolution are hard to interpret and are put in Appendix C. The evolution can be seen in Figure C.2a and C.2b, respectively.



(a) The path followed by the offline simulated USV when SYSID and RL was enabled.



(b) Performance statistics of the USV controller when SYSID and RL is used on the offline simulated USV.



(c) The model parameter evolution using SYSID and RL on the offline simulated USV.

Figure 6.7 Results of offline simulation of the USV with System Identification and Reinforcement Learning enabled. An external disturbance present, and the slow parameter guess was used.

ROS Real-Time Simulation

The settings in Section 6.1 were used also in the ROS simulation. The crucial difference is in the simulation update rate and the fact that this simulation emulates real-time communication with the real USV. This simulation is ran at 100 Hz, corresponding to the update rate of the fastest sensor on the USV, the INS. Thus, the model publishes its current position on the `odo_sim_ned` topic once every 0.01 s. The model subscribes to `control_signal` and always uses the most recent one. The controller was still run at 5 Hz, using the latest position published as input.

A summary of the results of the simulations in ROS is given in Table 6.7, where we can see that the ideal-case performance with robust tuning is an average cross-track-error of half a meter and a maximum error of less than 2 m. Introducing a disturbance worsens the performance as expected, and in a similar way to in the Offline simulation. When the slow parameters were used, the error is of the same size as when simulating offline, but the USV lost the path and could not finish the run. Finally, just as in the Offline simulation, the SYSID algorithm increases the performance to the ideal-case levels.

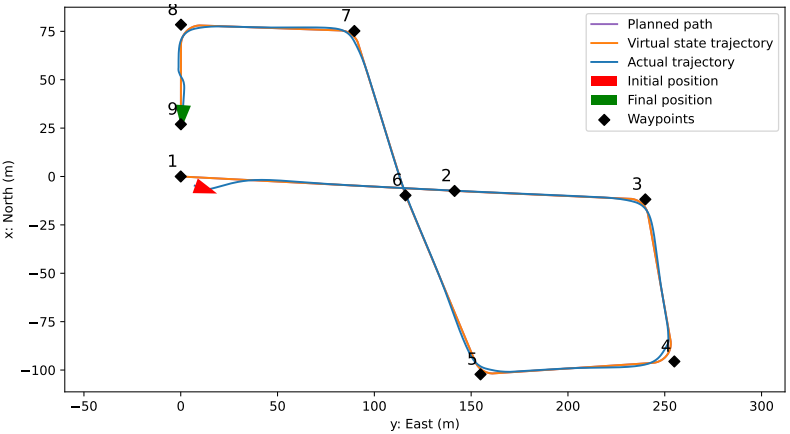
The computational time roughly doubled when compared to Offline simulation. This might be because of the fact that ROS was run in a VM.

Furthermore, since the prediction performance is worse when running in ROS, because of the real-time aspect, warm-starting the solvers with the previous solution will be less advantageous. This could also contribute to worsened computation times.

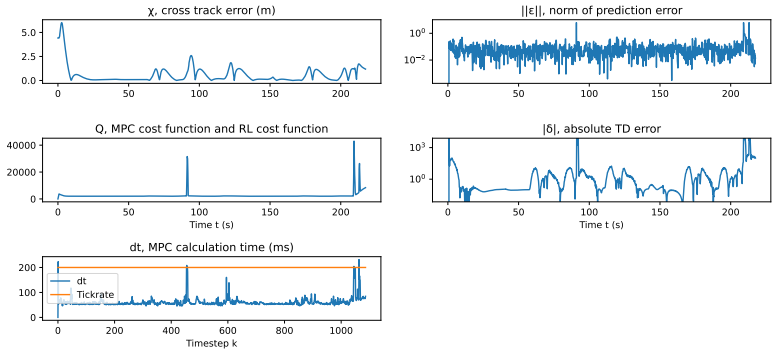
Table 6.7 Summary of the ROS simulation results. The average and maximum cross-track-error, $\overline{\chi}$ and $\chi_{\max}$ as well as the average and maximum MPC calculation time, $\overline{T}$ and $T_{\max}$ are shown. The robust tuning parameters are shown in Section 6.1. The robust tuning is used when indicated with 'R'. When the 0.4 m/s north disturbance is present, this is marked with 'D'. Finally, if the slow SYSID parameter guess is used, this is marked with the letter 'B'.

Simulation	$\overline{\chi}$ (m)	$\chi_{\max}$ (m)	$\overline{T}$ (ms)	$T_{\max}$ (ms)
Robust (R)	0.50	1.85	61	231
R Disturbance (RD)	1.23	3.66	62	231
RD slow (RDS) **DNF**	3.29	9.92	62	217
RDS SYSID	0.51	1.98	65	266

NMPFC. When running in the ideal case, the real-time aspect of the control did not seem to matter much. Performance was roughly the same as in offline simulation, which can be seen in Figure 6.8. The path looks mostly the same, shown in Figure 6.8a. It is however notable, that in addition to the computation time increasing, the prediction error is quite large throughout the simulation. This can be seen in Figure 6.8b. The magnitude of the prediction error is rarely less than 10^{-2}, likely because of the delay introduced by the NMPFC computation time. This error also increases significantly during the iterations with long computation times.



(a) The path followed by the USV controller simulated in ROS.



(b) Performance statistics of the USV controller simulated in ROS.

Figure 6.8 Results of simulation of the USV in ROS. The robust tuning, and correct model parameters were used and no disturbance was present.

SYSID-NMPFC. As in the Offline simulation case, in order to evaluate the SYSID performance, a base-case performance measurement was made with disturbance and slow parameters. It seems that using the correct pa-

rameters is more important when running in real-time. That can be seen in Figure 6.9, where the run did not finish, as the USV turned off the track and did not recover. The controller not recovering when going off-track has been observed to be a common occurrence with the NMPFC implementation. In this case, the solver outputs $\boldsymbol{u}^* = (0, 0)$, deciding that stopping is the optimal solution.

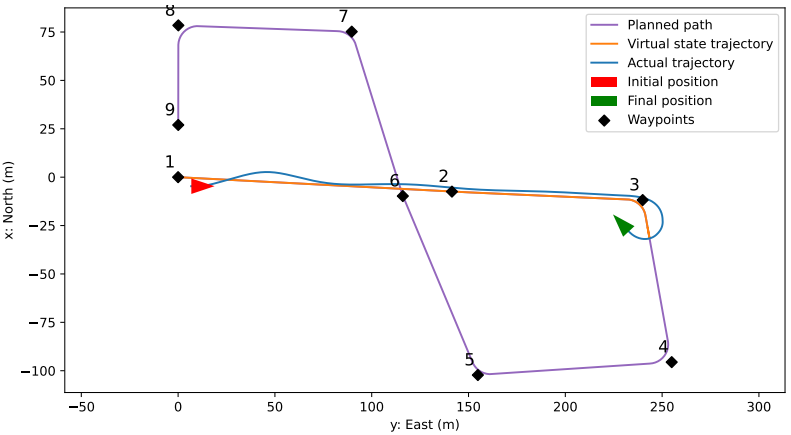
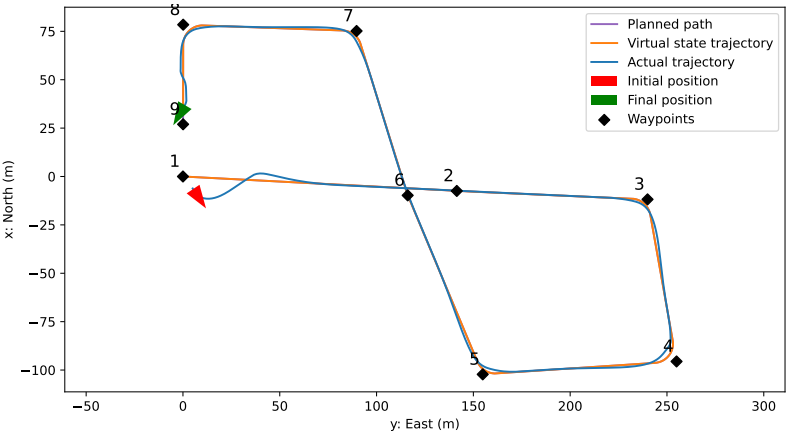
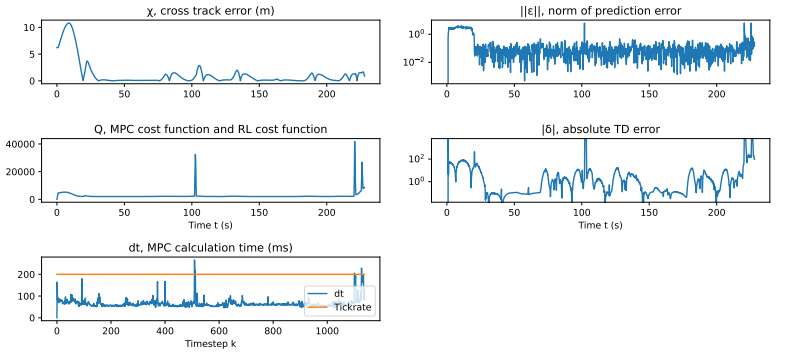


Figure 6.9 Results of simulation of the USV in ROS. Shown is the path. The robust tuning, and slow model parameters were used and the disturbance was present.

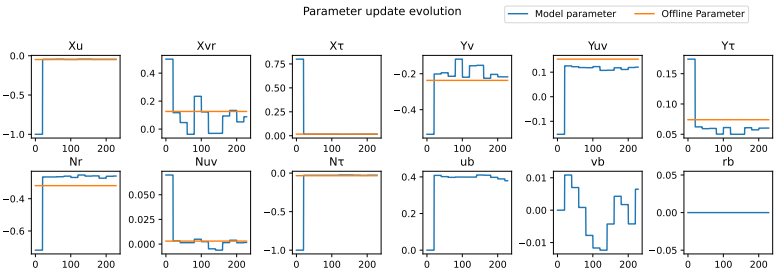
In the last simulation, SYSID was enabled, and the results are shown in Figure 6.10. The path is largely the same as the ideal-case path, except for an initial data-collecting part, see Figure 6.10a. The performance statistics were largely the same as in the ideal case. However, the prediction error magnitude seems to settle slightly lower than in the ideal case. This indicates that the SYSID method is adjusting the model to compensate for the computational delay. The model parameter convergence is shown in Figure 6.10c. It can be seen that some parameters converge to their offline values, while other parameters do not settle. Furthermore, some converge to different values than the offline estimates. This might be a case where some of the delay effects can be compensated for by adjusting these parameters to be different than the offline estimates.



(a) The path followed by the USV controller simulated in ROS when SYSID was enabled.



(b) Performance statistics of the USV controller simulated in ROS when SYSID was enabled.



(c) The model parameter evolution using SYSID on the USV simulated in ROS

Figure 6.10 Results of simulation of the USV in ROS with system identification enabled. An external disturbance was present, and the slow parameter guess was used.

7

Experiments

The experiments were run on 2023-06-02 in the archipelago outside the Saab Kockums Shipyard in Karlskrona. A map of the area can be seen in Figure 7.1. During the day, light wind from the North-west of about 3 m/s was present. The settings were the same as during simulation, in order to facilitate comparison. Before the start of each mission, the USV was manually steered to point toward the beginning of the path, and then the controller was enabled. Furthermore, since the simulated results showed that the RL method was unsuitable for real-time use, the experiments compared running NMPFC with and without SYSID.

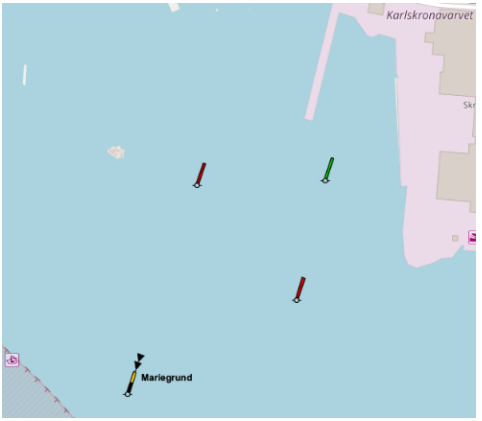


Figure 7.1 Map showing the testing area outside the Saab Kockums AB Wharf in Karlskrona [Open Sea Map, 2023].

7.1 Experimental Setup

Evaluation Missions

The missions used for the evaluation of performance were selected in two ways. The first mission ”Eight” is similar to the simulation mission, and can be seen in Figure 7.2, in order to facilitate comparison. The second mission ”Criss-Cross” is intended to evaluate turning performance, and can be seen in Figure 7.3. The vessel was started in the neighborhood of the first way-point and had to find its way onto the path and onwards on its own.

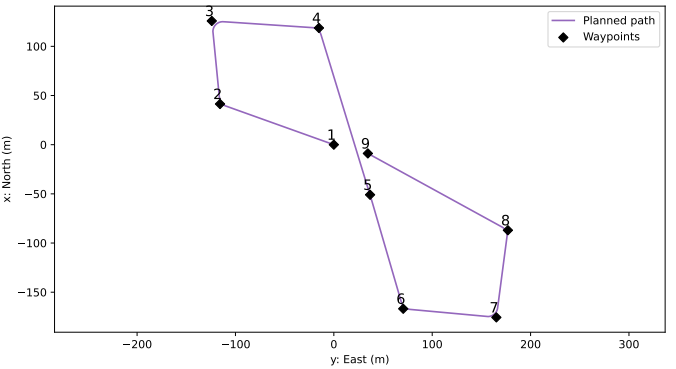


Figure 7.2 Mission ’Eight’. Way-points: {(0, 0), (41, -116), (126, -124), (119, -15), (-51, 37), (-167, 71), (-176, 165), (-87, 177), (-9, 35)}. The path is approximately 945 meters long.

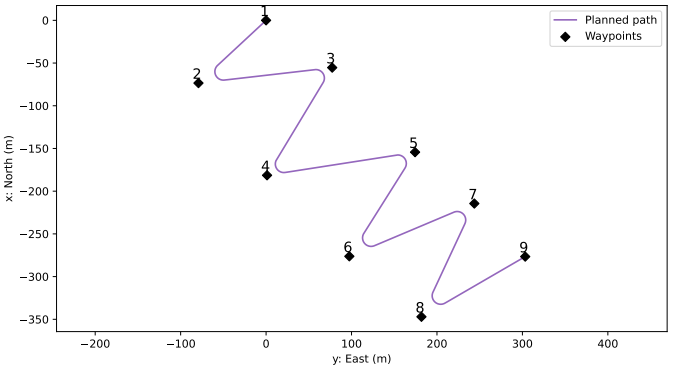


Figure 7.3 Mission 'Criss-Cross', in Karlskrona. Way-points: {(0, 0), (-73, -79), (-55, 77), (-181, 1), (-154, 174), (-276, 97), (-214, 244), (-347, 182), (-277, 303)}. The path is approximately 980 meters long.

Sensor Configuration

The INS was supposed to be used for the measurement velocity, but the measurements of u and v were found to be unreliable. This led to the GPS total velocity being used as a measurement of u, v being set to zero, while the rotational velocity measurement of the INS was used for the measurement of r. Thus the measurements sent to the controller were of the form $(x_{\text{GPS}}, y_{\text{GPS}}, \psi_{\text{GPS}}, u_{\text{GPS}}, 0, r_{\text{INS}})$. It should be noted that not being able to measure v, reduces the ship model specified in Equation (3.4) from a 9-parameter model to a 4-parameter model, potentially reducing performance significantly.

7.2 Experimental Results

The following results were obtained during experiments in Karlskrona. It can be seen that in the baseline case, without learning, the average cross-track error is below two meters in both missions, and the maximum error is about six meters. The controller performed better on mission "Eight",

which is expected, as the mission contains more straight-line tracks. When enabling system identification during mission "Eight" the error roughly halved. The average cross-track-error was reduced to below one meter, and the maximum error is below 3 m. However, when evaluating SYSID on mission "Criss-Cross", the run did not finish, and tracking performance was worse than with learning disabled. This was likely because of the wind picking up, increasing the disturbance to about 5 m/s with 7 m/s gusts. The controller has proved fragile when pushed off the course, discussed in Section 8.1. In this case, the USV during multiple attempts was pushed outside of the path by the wind, during a turn where the wind was at its back.

Finally, the calculation time of the controller stayed below the tick-rate of 200 ms at all times, indicating that an increase in controller frequency might be feasible in the future.

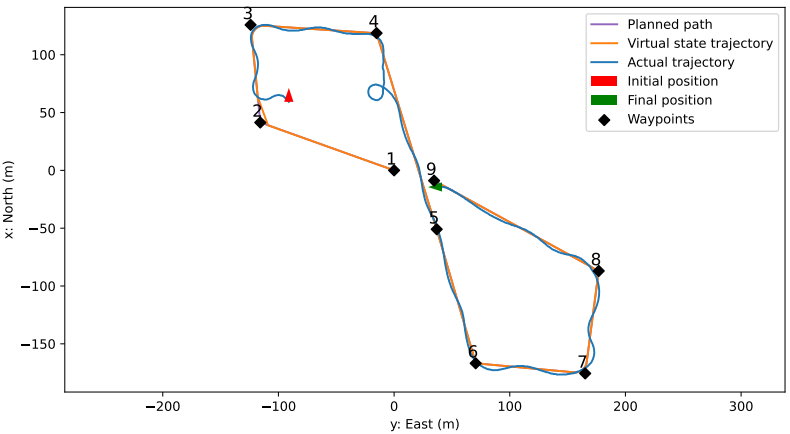
Table 7.1 Summary of the Experimental results. The average and maximum cross-track-error, $\overline{\chi}$ and $\chi_{\max}$ as well as the average and maximum MPC calculation time, $\overline{T}$ and $T_{\max}$ are shown. The robust tuning parameters are shown in Section 6.1.

Experiment	Mission	$\overline{\chi}$ (m)	$\chi_{\max}$ (m)	$\overline{T}$ (ms)	$T_{\max}$ (ms)
Robust (R)	Eight	1.53	5.50	37	126
R	Criss-Cross	1.7	6.23	36	119
R SYSID	Eight	0.90	2.85	32	85
R SYSID **DNF**	Criss-Cross	2.75	5.32	38	106

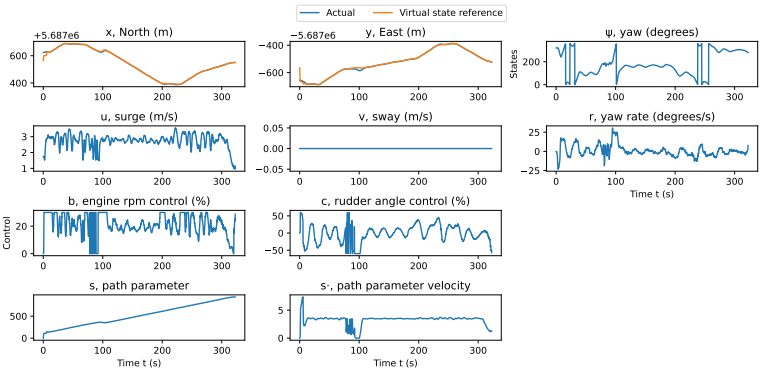
NMPFC

Mission "Eight". The following plots show the performance of the controller when no learning was enabled. In Figure 7.4a, the mission "Eight" path can be seen. The USV manages to join the path, and follows it through each way-point to the end. However, it is immediately obvious that the path-following performance is worse than in simulation. The USV "wobbles" around the path, and at one point does a "loop", a behavior which was explicitly punished. This is likely both because of unmodeled dynamics, especially in the actuators, and because of disturbances. In Figure 7.4b the control signals are shown. Here it is evident that the measured velocities, as well as the applied control signals, are more jagged than in simulation.

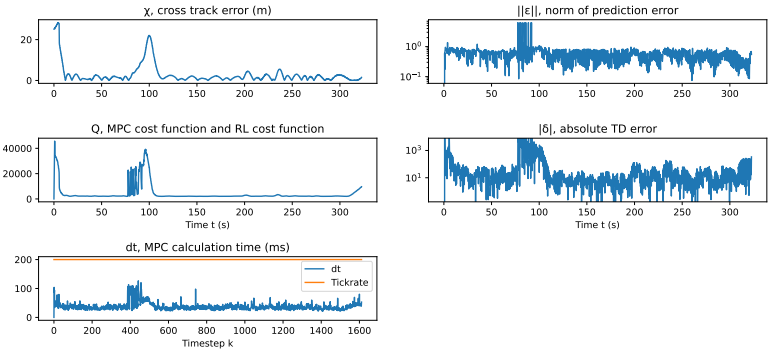
That behavior might a result of noise in the measurements, and the controller aggressively trying to compensate. This is supported by the fact that the rudder angle signal is less jagged than the throttle, as rudder changes are punished more than throttle changes. Furthermore, the average throttle signal is set significantly higher in the real experiment when compared to simulation, $b = 20$ vs. $b = 10$, see figure (6.3b). This indicates that there is a significant model mismatch in this parameter. In Figure 7.4c then, it can be seen that the the prediction error of the offline model is in the order of magnitude of 1. An increase in prediction error, as well as the calculation time can be seen at around 70 s. This corresponds to the USV looping, and the prediction error increases because of the fact that the sensors report $v = 0$, which it definitely is not when turning. It is plausible that this also is the reason that the calculation time increases at this time, since the predicted trajectory used for warm-starting the solver is inaccurate when turning.



(a) The path that was followed by the Piraya USV.



(b) The measured and output signals from and to the USV.

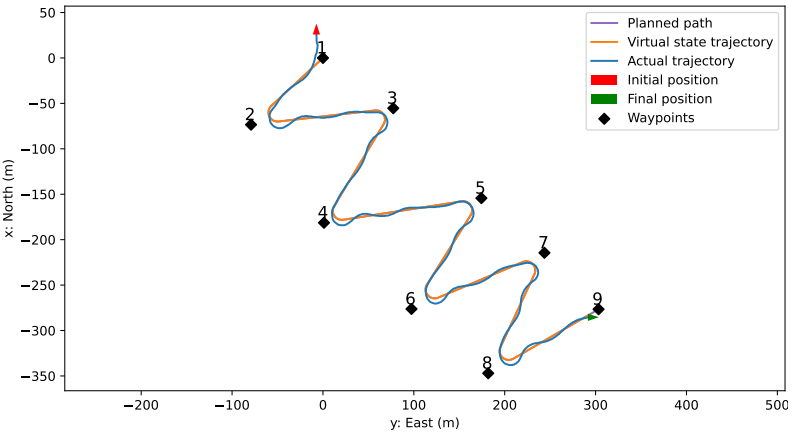


(c) Performance statistics

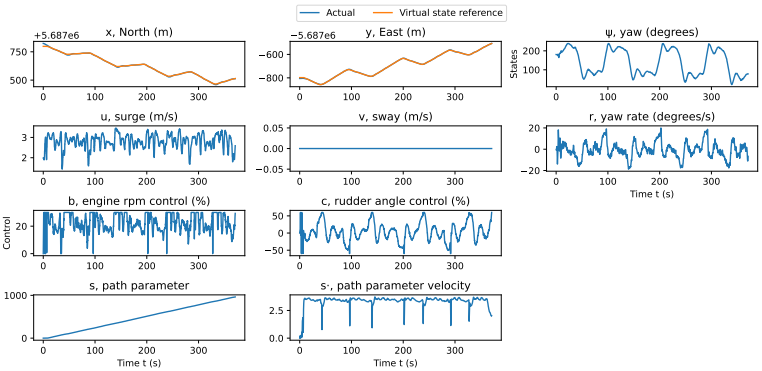
Figure 7.4 Experimental results when running the controller on the Piraya USV along the Mission "Eight".

Mission ”Criss-Cross”. During the evaluation of turning performance with the mission ”Criss-Cross”, which can be seen in Figure 7.5, similar results are found. The path can be seen in Figure 7.5a, which similarly to the previous mission shows wobbling behavior. The USV does successfully complete the mission, however, with satisfactory performance. As expected with this sensor configuration, the performance is worse when turning com-

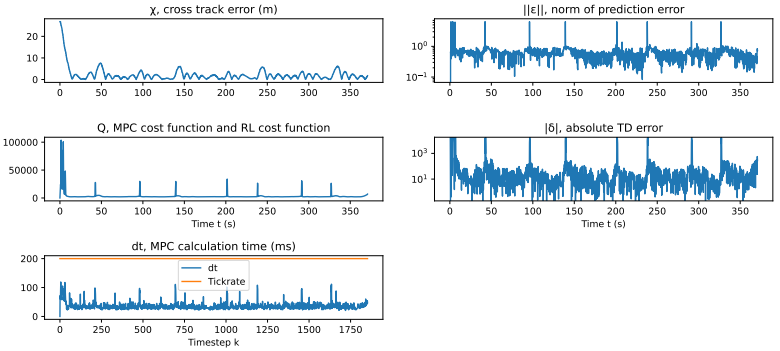
pared to during straight sections. In Figure 7.5b, all controller signals are shown. Once again the velocities and control signals vary a lot, especially the forward velocity and throttle. Interestingly, when looking at the path parameter velocity, the velocity drops at even intervals, corresponding to the turns made by the USV. This clearly demonstrates the NMPFC principle in action, with path velocity being lowered in order to improve path-following performance. Finally, the performance statistics in Figure 7.5c are similar to the ones in mission ”Eight”, with spikes in prediction error, and calculation time during turning sections.



(a) The path that was followed by the Piraya USV.



(b) The measured and output signals from and to the USV



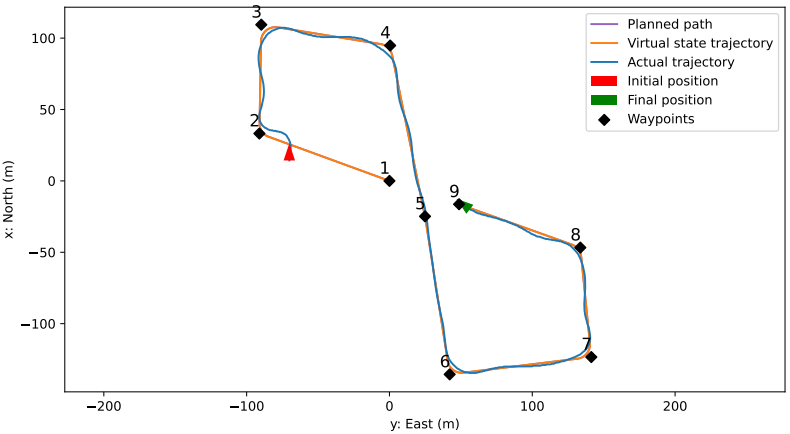
(c) Performance statistics

Figure 7.5 Experimental results when running the controller on the Piraya USV along the Mission "Criss-Cross".

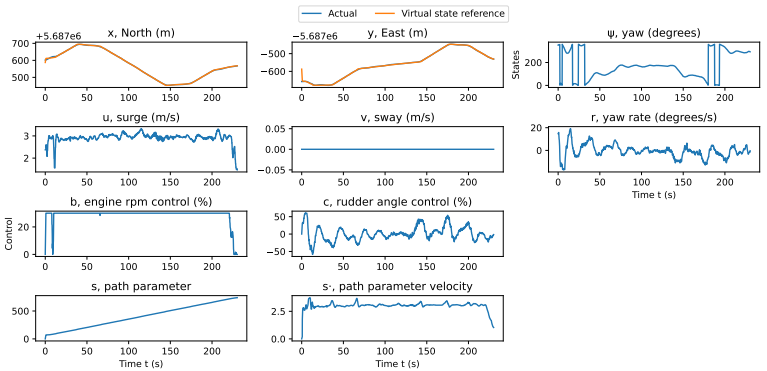
SYSID-NMPFC

Finally, the results when enabling system identification are shown, mission "eight" in Figure 7.6, and "criss-cross" in Figure 7.7. It can be seen in Figures 7.6a and 7.7a that the tracking performance quickly improves when compared to the non-learning case. The wobbling during straight sections is reduced, and the overshoot while turning is replaced with an undershoot. The undershoot while turning is similar to what is seen with this tuning in simulations, like in Figure 6.3a. What can also be seen in Figure 7.7a however, is the fact that the mission was not finished, as previously discussed.

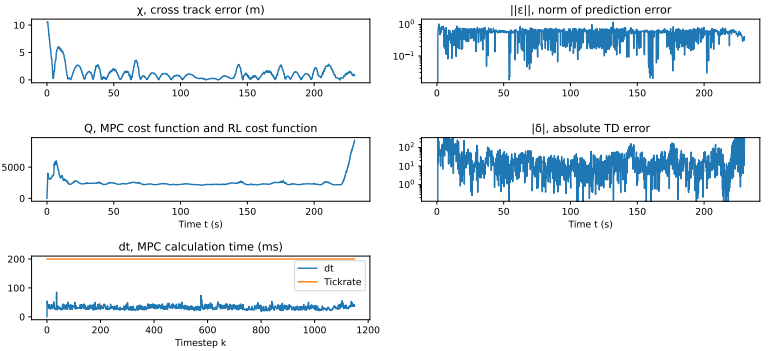
Mission "Eight". The signals during mission "Eight" are shown in Figure 7.6b, where it can be seen that the velocities and control signals are smoother than in the non-learning case. It should also be noted that the throttle is set to max $b = 30$, for most of the run, while not achieving surge u or path velocity $\dot{z}$ of 3.5 m/s, indicating that the strength of disturbances had increased at this point. It is interesting that the cross-track-error performance increased in spite of this. Furthermore, the performance statistics shown in Figure 7.6c indicate that the prediction error decreased compared to the non-learning case in Figure 7.4c. Finally, in Figure 7.6d the model parameter evolution is shown. First of all, it should be noted that the parameters relating to the sway v are either not affected by the system identification, or set to 0, as a consequence of the constant zero measurement. When it comes to the parameters X_u, X_τ, N_r and N_τ, it can be seen that they have periods of relative stationary, intertwined with periods of rapid change. This could be a consequence of the under-parameterized model, and the limited window of data used for identification. Thus, certain parts of the data-set might correspond to different optimal parameters for estimation. It can also be seen that neither of the offset biases u_b and v_b are stationary. This does not necessarily have to be incorrect, as part of the unmodeled dynamics, i.e., the disturbances, are expected to change with time. However, it is also very likely that the residual dynamics reflected in these parameters varies over the dataset, just as with the other parameters.



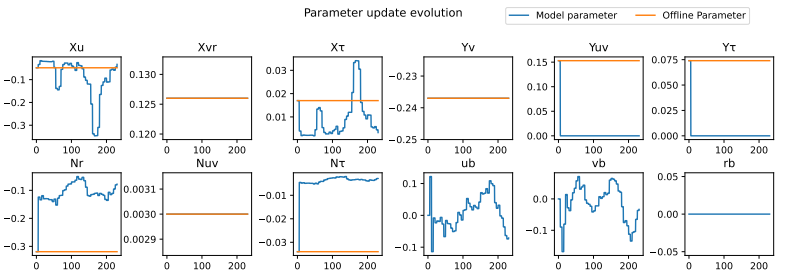
(a) The path that was followed by the Piraya USV with SYSID enabled.



(b) The measured and output signals from and to the USV with SYSID enabled.



(c) Performance statistics with SYSID enabled.

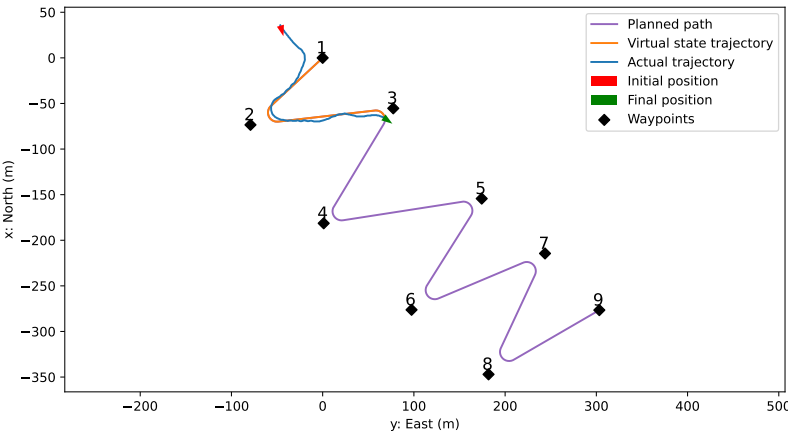


(d) Model parameter evolution

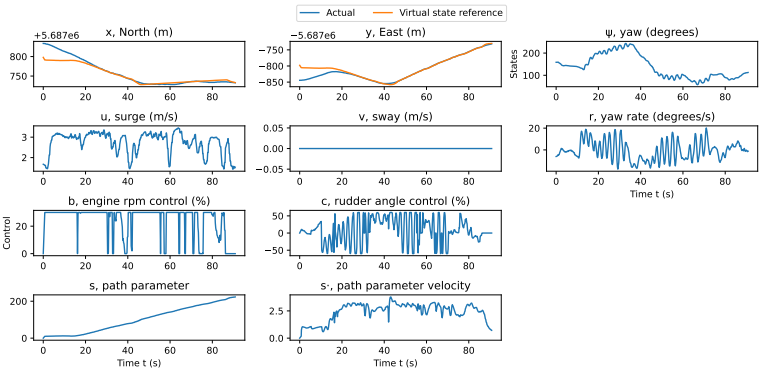
Figure 7.6 Experimental results when running the controller on the Piraya USV along the Mission "Eight" with SYSID enabled.

Mission "Criss-Cross". The signals of mission "Criss-Cross" are shown in Figure 7.7b. As previously mentioned, this run did not finish. At the end of the run, it can be seen that the USV is facing 90° away from the path. This lead to the solver outputting $\boldsymbol{u}^* = (0, 0)$, effectively deciding that staying in place is the most optimal solution, as previously discussed. The Piraya turning this far off-path and not recovering indicates that there were large unmodeled disturbances present during this experiment, which pushed the USV off the path at an important moment. Moreover, in the signal plot, it should be noted that the rudder signal is very active, as well as the yaw rate changing very quickly. Just as before, it is difficult to tell if this is

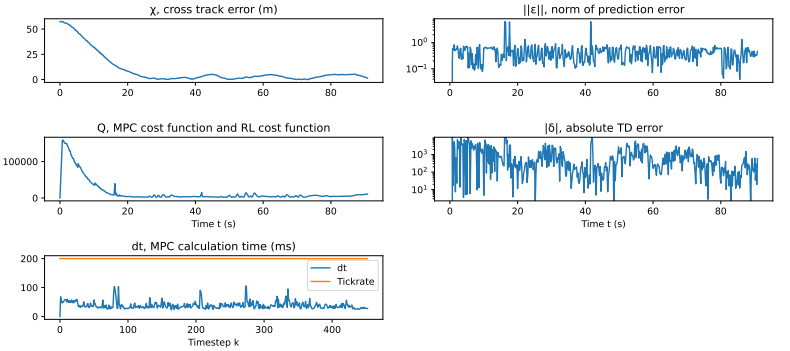
a result of increased disturbances or the controller being over-active. The latter does seem less likely though, as overactive rudder signals have been effectively punished in the other scenarios. Also, the throttle is often set to max, $b = 30$, further indicating the presence of a significant disturbance. The performance statistics during the mission are shown in Figure 7.7c, where the behavior is similar to the previous mission. Finally, the model parameters are shown in Figure 7.7d. The live parameters, X_u, X_τ, N_r and N_τ seem to converge to stationarity in this case, while the disturbances u_b and v_b do not converge. In this case it is more likely that this is because of the disturbances changing with time, since the same behavior is not seen in the model parameters. It should, however, be noted that this data-set is smaller, and might thus not be representative for the method performance over time.



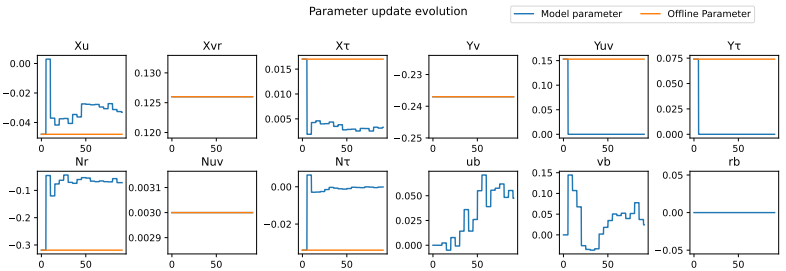
(a) The path that was followed by the Piraya USV with SYSID enabled.



(b) The measured and output signals from and to the USV with SYSID enabled.



(c) Performance statistics with SYSID enabled.



(d) Model parameter evolution

Figure 7.7 Experimental results when running the controller on the Piraya USV along the Mission "Criss-Cross" with SYSID enabled.

8

Discussion

In this section, the employed methods are discussed in terms of strengths, weaknesses, and possible improvements. First NMPFC is evaluated as a path-following formulation, then SYSID is discussed for learning model parameters, and finally, SYSID+RL is spoken about in terms of improving overall controller performance.

8.1 Model Predictive Path Following Control

The simulation results seem to indicate that in the ideal case, MPFC is a good method for path-following control. This can be seen when studying the results of the controller 'A' in offline simulation, see Table 6.6, where the average error was less than 1 dm. Furthermore, it is also relatively easy to implement, as it is done the same way the system model is in a regular MPC controller. The method also lends itself well to producing different behavior through tuning, as noted when comparing Figures 6.2a and 6.3a. Thus the method can be tailored to fit the specific needs of the application. Moreover, when compared to trajectory control, the operator does not have to consider at which time the USV should be at any specific position. This removes the need for generation of a trajectory that is feasible for the vessel at hand. Rather, the method can be tuned to produce a certain behavior, which will be applied to any path the vessel is set to travel along. Thus, allowing the controller to dynamically adjust the USV velocity in a compromise between speed and tracking accuracy increases the autonomy of the vessel, since human input on the timing along the path is not necessary. Finally, the method translates well into real-time control, see Figure 6.8.

However, from the results presented the method does not seem to be very robust against going off-path. This leads to low robustness to incorrect model parameters, as seen in Figure 6.9, and low robustness against actual disturbances, as shown in Figure 7.7a. An explanation of this behavior is that re-joining the path would necessitate moving farther from it, before coming closer. Staying still, on the other hand, is penalized since the controller is punished for not proceeding along the path at the desired rate. However, the solution does stay close to the path, which is desirable. Intuitively then, it can be reasoned that there are multiple local minima in the NMPFC solution. This is also to be expected since the problem is non-convex.

In terms of solutions to this problem, calculating the heading to be followed might be a way to increase the robustness in these cases, as there are more weights to ensure the vessel does not go off-path. That would also increase the likelihood that the USV is pointing along or towards the path, which does seem to be the most crucial factor in terms of getting the desired solutions from the solver. In practice, a way to decrease the prevalence of this issue is to increase the cost on $\dot{z} - \dot{z}_{\text{ref}}$. This leads to stationary solutions being more costly, encouraging the solver to head forwards. However, this leads to less adaption of the velocity when turning, which was supposed to be one of the advantages of this method. Furthermore, one could imagine also punishing velocities close to zero in the cost function. That does however increase the non-convexity of the MPC problem, which was the root of the issue in the first place.

Another possible solution could be to ignore the cost of the first few predicted states. That would allow for solutions that temporarily travel away from the path, as that cost would not be counted. This solution, however, does mean that short-term performance is degraded also in turns and along straight sections. Early in the thesis, this option was explored but was not found to solve the problem. Another possible solution is to intelligently warm-start the solver, i.e., with guesses that have the velocity and throttle set to some fixed above zero value. This was also explored but turned out to be difficult in practice, with the solver often returning constraint-violating output.

8.2 SYSID-NMPFC

The System Identification method used here, the Prediction Error Method with a rolling window, shows great performance uplifts when used online. In offline simulation, the method finds the correct parameters in just one iteration. When comparing the results of using slow parameters and slow parameters with SYSID, the average error decreases by a factor of about 8, see Table 6.6, for that choice of parameters, see Table 6.1. Moreover, in the ROS simulation, the model parameters mostly converged within one iteration and even managed to slightly adjust for the delay introduced by the computation time. This led to a small decrease in prediction error when compared to the ideal case. It also made the difference between finishing the run and the controller quitting when running with the slow parameters, as discussed above. Finally, when running against the real USV, both the average and maximum error decreased by a factor of two, see Table 7.1, compared to a model which had been sampled specifically for the Piraya. Since the model from [Ljungberg, 2021] was sampled at 2 m/s and the tests were run at 3.5 m/s the offline model was not expected to be accurate in these conditions. This confirms the usefulness of online SYSID in the case of nonlinear systems, or in cases where the model used is known to be inaccurate.

Furthermore, implementation of the method is very simple, and is also computationally cheap, with computation times that are negligible compared to the MPC computation time. Thus, the method should be considered for any MPC control application where offline system identification may be prohibited. Since offline system identification might be a burdensome and expensive operation, another application of the method is the ability to improve a model quickly obtained through offline SYSID. Looking ahead, this also allows for the development of a single control algorithm that could be deployed to different vessels. The control algorithm could in that case be developed on any chosen vessel, and then be applied directly to another, with the SYSID method compensating for the parameter differences between the vessels.

As expected, however, the method is only as good as the mathematical model provided, and the sensor data that are collected. Even in the ROS simulation, the prediction performance was well below the offline simulation case,

likely because of unmodeled dynamics, such as the real-time computational delay, and the fact that the model updates are compounded non-linearly between control iterations. Then, when running against a real system, most unmodeled behavior will not be possible to capture, and thus lead to non-stationary parameter estimates. Therefore, a large model should be preferred when using this method. However, when a larger model is used, more and higher-fidelity sensor data are necessary to excite the method. Therefore, as expected, the method is limited by the kind of sensors equipped on the system.

8.3 SYSID-RL-NMPFC

From the results in Table 6.6, it is indicated that this method of Reinforcement Learning has the potential to improve the tracking performance of an USV. However, when compared to SYSID in simulation, the method does not perform as well. This is to be expected, as the offline simulation is what might be considered the ideal case for the SYSID method. The RL method's potential is in adjusting a model known to be approximate, in order to increase performance in the face of unmodeled dynamics. It is therefore unfortunate that this implementation of the RL method was not suitable for real-time use, as that is where it intuitively should perform best. This seems to be an issue with the choice of numerical solver and automatic differentiation framework. While using CasADi with the QRQP method often resulted in quick calculations and good control performance, but also constraint violations and random spikes in calculation times made it not practical for real-time use, which can be seen in Figure 6.6b. Using IPOPT when constraint violations were generated was a solution that worked as a remedy to the first problem. The second problem is harder to tackle, however, as these periods with long calculation times appeared at what seemed like random intervals.

Furthermore, another issue with the RL framework is how to generate exploration through the policy. Since the Q-learning method only guarantees the learning of the optimal Q-function in the case that every state-action pair is visited, it is necessary to take actions that would be detrimental to performance in most situations. This might be an indication that offline Q-learning would be better to apply in this case, where a lot of data

could be collected offline, and where maneuvers that are not often applied could be sampled. Furthermore, this allows for batch updating of the data [Sutton and Barto, 2018], just as in the SYSID method applied here. Then the Q-values, which in this case have been the MPC costs, can be recomputed after every parameter update iteration, allowing for more learning with a smaller dataset. This has to be done offline, however, since re-calculating the best control signals for every past trajectory simply takes too much time to be done online.

9

Conclusion

In this thesis, we have shown that Nonlinear Model Predictive Path-Following control can be implemented on the Piraya USV. It was shown to result in good solutions in simulation and is easy to implement when an NMPC formulation is already present. However, the results show that the method is not very robust against model error and external disturbances, leading to halting, $\boldsymbol{u}^* = (0, 0)$ solutions.

The first method implemented to improve tracking performance was System Identification through PEM. In offline simulation, the method quickly identified the exact model parameters and the external disturbances. It also showed robustness when running in real-time simulation, though the unmodeled time delay increases the prediction error. Finally, when running on the real Piraya USV, it did roughly double the tracking performance of the controller. Here the method showed adaptability to model the changing disturbances in the environment. Furthermore, the experiments showed that the method increased the range of velocities the model-based controller can be used at. Moreover, results show that this online method has the potential to save time and effort when compared to offline SYSID, and could therefore be deployed for control of other vessels.

The second method used was Reinforcement Learning through Q-learning. It was implemented and evaluated in offline simulation. While it was found to increase the performance of the controller, it did not increase performance as much as the System Identification. Interestingly, performance increased even though the parameters did not converge towards the offline-identified parameters, nor correctly identify the external disturbance. Furthermore,

the particular implementation of Q-learning in this thesis was found to be too computationally demanding for real-time use, which is where the method in theory should be able to increase performance beyond SYSID.

9.1 Future Work

The most obvious thing to try as future work is the control of other vessels, using this implementation, unmodified, and see how well the methods can adapt. Moreover, it would be interesting to see the performance of this algorithm with the full Piraya sensor array working. Then a larger model could also be tried to identified online, which should improve performance. Furthermore, this would increase the richness of the Q-function parameterization, possibly allowing for a better fit of the Q-function, and even larger performance improvements could be made through RL. In addition, more advanced RL methods such as *policy-gradient methods*, which directly modify the controller policy, would be interesting to implement. Finally, an investigation into a dedicated symbolic framework and numerical solver combination is probably necessary for the deployment of the RL methods online.

Bibliography

Alexiou, J. (2015). *How to find arc that connects two segments - Answer*. URL: https://stackoverflow.com/a/28030228 (visited on 2023-02-28).

Andersson, J. A. E., J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl (2019). "CasADi – A software framework for nonlinear optimization and optimal control". *Mathematical Programming Computation* **11**:1, pp. 1–36. DOI: 10.1007/s12532-018-0139-4.

Andersson, J. A. (2023a). *Computing KKT conditions of IPOPT, and sensitivity with respect to parameters for asNMPC*. https://github.com/casadi/casadi/discussions/3156. [Online; accessed 29-May-2023].

Andersson, J. A. (2023b). *Lagrangian as symbolic object*. URL: https://groups.google.com/g/casadi-users/c/h05oCge2vkk (visited on 2023-05-22).

Andersson, J. A. and J. B. Rawlings (2018). "Sensitivity analysis for nonlinear programming in CasADi". *IFAC-PapersOnLine* **51**:20. 6th IFAC Conference on Nonlinear Model Predictive Control NMPC 2018, pp. 331–336. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2018.11.055.

Åström, K. (1981). "Chapter 5 - maximum likelihood and prediction error methods". In: EYKHOFF, P. (Ed.). *Trends and Progress in System Identification*. Pergamon, pp. 145–168. ISBN: 978-0-08-025683-2. DOI: https://doi.org/10.1016/B978-0-08-025683-2.50010-7.

Böiers, L.-C. (2010). *Mathematical Methods of Optimization*. English. Lärobok i matematik. Studentlitteratur AB. ISBN: 978-91-44-07075-9.

Faulwasser, T. (2013). *Optimization-based solutions to constrained trajectory-tracking and path-following problems*. Contributions in Systems Theory and Automatic Control. 3. Shaker Verlag, Aachen. DOI: https://doi.org/10.2370/9783844015942.

Faulwasser, T., T. Weber, P. Zometa, and R. Findeisen (2017). "Implementation of nonlinear model predictive path-following control for an industrial robot". *IEEE Transactions on Control Systems Technology* **25**:4, pp. 1505–1511. DOI: 10.1109/TCST.2016.2601624.

Fossen, T. (2021). *Handbook of Marine Craft Hydrodynamics and Motion Control*. Wiley. ISBN: 9781119575016. DOI: 10.1002/9781119575016.

Gros, S. and M. Zanon (2020). "Data-driven economic nmpc using reinforcement learning". *IEEE Transactions on Automatic Control* **65**:2, pp. 636–648. DOI: 10.1109/TAC.2019.2913768.

Kockum, S. (2022). "Autonomous Docking of an Unmanned Surface Vehicle using Model Predictive Control". eng. *Department Of Automatic Control, Lund University*. Master's Thesis. ISSN: 0280-5316. URL: http://lup.lub.lu.se/student-papers/record/9096721.

Ljungberg, F. (2021). "Systemidentifiering för piraya". Technical Report, SAAB Kockums AB.

Martinsen, A. B., A. M. Lekkas, and S. Gros (2020). "Combining system identification with reinforcement learning-based MPC". *IFAC-PapersOnLine* **53**:2. 21st IFAC World Congress, pp. 8130–8135. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.2294.

Martinsen, A. B., A. M. Lekkas, and S. Gros (2022). "Reinforcement learning-based NMPC for tracking control of ASVs: theory and experiments". *Control Engineering Practice* **120**, p. 105024. ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2021.105024.

Mehrez, M. (2019). "Optimization based solutions for control and state estimation in dynamical systems (implementation to mobile robots) a workshop". *University of Waterloo*. DOI: 10.13140/RG.2.2.21613.23521.

Moe, S., A. M. Rustad, and K. G. Hanssen (2018). "Machine learning in control systems: an overview of the state of the art". In: Bramer, M. et al.

(Eds.). *Artificial Intelligence XXXV*. Springer International Publishing, Cham, pp. 250–265.

Open Sea Map (2023). *OpenSeaMap - The free nautical chart*. https://map.openseamap.org/. [Online; accessed 30-May-2023].

Rawlings, J., D. Mayne, and M. Diehl (2017). *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing. ISBN: 9780975937730. URL: https://books.google.co.uk/books?id=MrJctAEACAAJ.

Stanford Artificial Intelligence Laboratory et al. (2018). *Robotic operating system*. Version ROS Melodic Morenia. URL: https://www.ros.org.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Second Edition. The MIT Press. URL: http://incompleteideas.net/book/the-book-2nd.html.

Wächter, A. and L. T. Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". *Mathematical Programming* **106**:1, pp. 25–57. ISSN: 1436-4646. DOI: 10.1007/s10107-004-0559-y.

WARA-PS (2023). *WARA-PS Portal - PIRAYA*. URL: https://portal.waraps.org/page/piraya (visited on 2023-05-16).

Zanon, M., S. Gros, and A. Bemporad (2019). "Practical reinforcement learning of stabilizing economic mpc". In: *2019 18th European Control Conference (ECC)*, pp. 2258–2263. DOI: 10.23919/ECC.2019.8795816.

A

Discrete-Space Reinforcement-Learning

This chapter introduces the notation, concepts, and algorithms necessary to perform discrete-space Reinforcement Learning, in particular, Q-learning [Sutton and Barto, 2018]. In discrete-space RL, the world is often thought of as a grid, and actions a take the agent from one square (state s) on the grid to another. Furthermore, often these states come with rewards $R(s)$, which either encourage or discourage the agent to come back to that state. In Figure (A.1) an illustration of states, actions, and rewards is shown. In this example, there are five states $s \in \{-2, -1, 0, 1, 2\}$, and two actions in each state, go left or go right, $a \in \{-1, 1\}$. Furthermore, there are rewards placed in states $s = -2$ and $s = 2$, with $R(-2) = -1$ and $R(2) = 1$. These have been placed in order to encourage the agent to head to the rightmost state, $s = 2$.

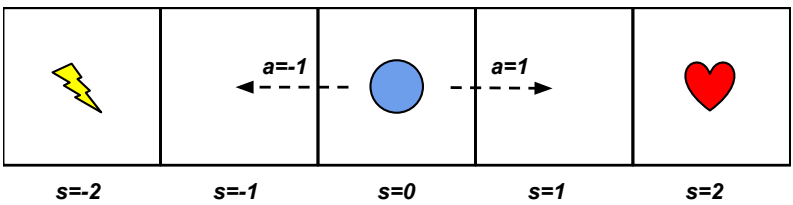


Figure A.1 A simple discrete RL problem with five states, two actions, and two rewards. The heart gives reward 1, and the lightning bolt gives reward -1. The agent should, through RL, learn to always walk right.

RL methods in general are concerned with, through trial and error, learning of the rewards of each state, and from that information, knowing what actions to take in any situation. This is formulated as the agent wanting to maximize the reward received not just in the next state, but during the entire task.

A.1 Value-, Action-Value-Function, and Policies

Since an RL agent will aim to maximize the total reward over the rest of the task, it is natural to introduce a *Value Function* $V_\pi(s)$, which gives the expected total (including future) reward in the state s, taking actions following the policy $\pi(s)$ [Sutton and Barto, 2018]. Further, the *Action-Value Function* $Q_\pi(s,a)$ or the *Q-function*, measures the expected total reward in the state s, and taking the action a, while subsequently following the policy $\pi(s)$. This also implies the need for a reward function $R(s)$, which measures the *true* reward of any state [Sutton and Barto, 2018]. A visualization of the connection between the value, Q-function, and rewards, as well as the state s, future state s^+ and action a, can be seen in Figure A.2.

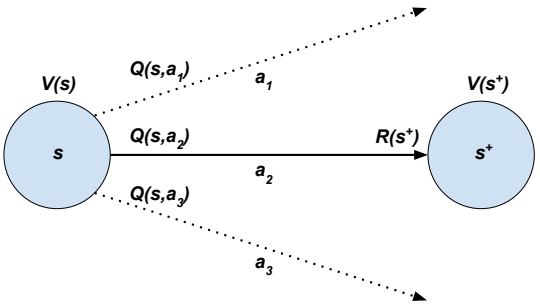


Figure A.2 Figure showing a current state s, with state value $V(s)$. An action, a_2 is chosen by some policy $\pi(s)$ out of the possible actions a_1, a_2 and a_3. The actions taken in state s have the state-action values $Q(s,a_1), Q(s,a_2)$ and $Q(s,a_3)$. Applying action a_2 leads to the state $s^+ = f(s,a_2)$, where the reward $R(s^+)$ is received. Finally, the future state s^+ has the state value $V(s^+)$.

Given any reward function, there exists at least one *optimal policy* π^*, for which $V_{\pi^*}(s) \geq V_\pi(s)$ for all states s and policies $\pi(s)$

[Sutton and Barto, 2018]. The optimal policies share a *optimal value function* $V_*(s)$. and *Optimal Action-Value Function* $Q_*(s, a)$, which measure the maximum total reward possible from any state, and from any state given an action in that state, respectively. Naturally, these are connected by the simple equation

$$V_*(s) = \max_a Q_*(s, a) \tag{A.1}$$

Intuitively, the action with the biggest total reward as given by the Q-function must be equal to the value of the optimal policy in that state. Further, the optimal policy is obtained given the optimal Q-function, as

$$\pi_*(s) = \operatorname*{argmax}_a Q_*(s, a) \tag{A.2}$$

where the optimal policy takes the action which maximizes the optimal Q-function and thus achieves the largest value in the state s. In the context of a system that evolves with time, through a dynamics function like $s^+ = f(s, a)$, where s^+ is the next state, the optimal value function and Q-function satisfy the following recursive equation, called the *Bellman Equation* [Sutton and Barto, 2018]

$$V_*(s) = R(s^+) + V_*(s^+) \tag{A.3a}$$

$$Q_*(s, a) = R(s^+) + \max_{a^+} Q_*(s^+, a^+) \tag{A.3b}$$

which states that the optimal value of any state, must be equal to the reward acquired in the transition to the next state $R(s^+)$, plus the maximum total reward achievable in the next state $V(s^+)$ [Sutton and Barto, 2018].

A.2 Temporal Difference Methods

In Reinforcement Learning Methods, the objective is to learn either the optimal value function, Q-function, the policy, or a combination of the three [Sutton and Barto, 2018]. Since the Bellman Equation can easily be used to verify if any approximation of these have achieved optimality, it is a useful tool in the learning process. In *Temporal Difference methods*, (TD methods), the *temporal difference error* (TD error) is formed as the mismatch in the Bellman Equation for a specific transition (s, s^+), with an associated reward R. Starting from an initial guess of value function $V(s)$, the TD error is formed as [Sutton and Barto, 2018]

$$\delta = R(s^+) + V(s^+) - V(s) \tag{A.4}$$

where $\delta = 0$ means the Bellman equation is satisfied and would imply the approximate V is equal to the optimal value function in that state.

In the TD(0) method, the TD error is used to learn the value function V_π corresponding to a given policy π [Sutton and Barto, 2018]. Intuitively, if $\delta \neq 0$, updating the value function in that state $V(s) \leftarrow V(s) + \delta$ would ensure no TD error is incurred next time this transition is made. However, as is common in machine learning, a *learning rate* $\beta > 0$ is introduced instead, and the update made is

$$V(s) \leftarrow V(s) + \beta\delta \tag{A.5}$$

which still reduces the temporal difference in this state. The purpose of introducing the learning rate is to increase the robustness of the method in the face of, e.g., noisy measurements, giving lesser importance to individual transitions. It should be noted that since the unknown value function is used twice in the equation, the value function iterates are not expected to converge exactly to the value function V_π. This can still be useful, since the reward, R, received in any transition (s, s^+) is considered the truth, and thus the value $R(s^+) + V(s^+)$ contains more information than $V(s)$. Intuitively then, the value function estimate is updated to match the rewards R received during training [Sutton and Barto, 2018].

A.3 Q-Learning

Q-learning is a method for learning the optimal policy π_* associated with the reward $R(s)$ [Sutton and Barto, 2018]. An advantage of Q-learning is that this can be done while allowing the system to follow any other policy during learning, as long as that policy visits all state-action pairs. This is called *off-policy* learning since we are not *on* (using) the policy we are trying to learn. As the name implies, in this case, the TD error is formed with Q-functions, and the algorithm can be shown to have an initial guess Q converge to the optimal Q_*. The optimal policy is then found by Equation (A.2). Given a transition (s, a, s^+) and reward R, the TD error is [Sutton and Barto, 2018]

$$\delta = R(s^+) + \max_{a^+} Q(s^+, a^+) - Q(s, a) \tag{A.6}$$

which captures the error in the Bellman Equation (A.3b). Similar to TD(0), the Q-function estimate is updated using the TD error.

$$Q(s, a) \leftarrow Q(s, a) + \beta\delta \tag{A.7}$$

where β once again is the learning rate. As previously stated, in Q-learning, any policy can be followed during the learning phase as long as it visits all state-action combinations. A common choice for the policy to be followed during Q-learning is called an ϵ-*greedy* policy [Sutton and Barto, 2018], π_ϵ. For some small probability $\epsilon > 0$, and a random number $p \sim U(0, 1)$, where $U(0, 1)$ is the uniform distribution between 0 and 1, the policy is

$$\pi_\epsilon(s) = \begin{cases} \underset{a}{\text{argmax}}\ Q(s, a), & p > 1 - \epsilon \\ \text{random action}, & p < \epsilon \end{cases} \tag{A.8}$$

as can be seen above, a ϵ-greedy policy is greedy with probability $1 - \epsilon$, which is exploitative behavior. On the other hand, the policy engages in exploration with probability ϵ, where any other sub-optimal action is taken at random. The full Q-learning procedure is shown in Algorithm (2) [Sutton and Barto, 2018].

Algorithm 2 Tabular Q-learning

Initial state s
Learning Rate $\beta > 0$, Greedy parameter $\epsilon > 0$
$Q(s, a)$ arbitrary for all (s, a)
while *learning* **do**
  $p \leftarrow$ sample from $U(0, 1)$
  $a \leftarrow \begin{cases} \underset{a}{\text{argmax}}\ Q(s, a), & p > 1 - \epsilon \\ \text{random action}, & p < \epsilon \end{cases}$
  $R, s^+ \leftarrow \text{system}(s, a)$
  $Q(s, a) \leftarrow Q(s, a) + \beta \left(R + \max_{a^+} Q(s^+, a^+) - Q(s, a) \right)$
  $s \leftarrow s^+$
end while

B

Path Parameterization

In this chapter, the creation of a part-linear, part-circular path for the purpose of model predictive path following control is detailed. The necessary input is a set of way-points $\mathcal{W}$ to follow, as well as a turning radius R, which is used as the path circle radius. Crucially, the path is by this construction ensured to be contiously differentiable in order to facilitate the optimization in MPFC.

B.1 Piece-Wise Linear Paths

Given a set of way-points $\mathcal{W} = \{(x_i, y_i)\}_{i=1}^{M}$ a piece-wise linear path can be established by creating the set of directional vectors

$$\mathcal{D} = \left\{ \frac{(x_i - x_{i-1}, y_i - y_{i-1})}{\sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}} \right\}_{i=1}^{M} \tag{B.1}$$

The linear paths are then given by $(x(z), y(z))_i = \boldsymbol{d}_i z + \boldsymbol{w}_i$, where $\boldsymbol{d}_i \in \mathcal{D}$, $\boldsymbol{w}_i \in \mathcal{W}$ and $z \in [0, z_{\max,k}]$. Here $z_{\max,i} = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$ is the length of the path to be traveled.

B.2 Circular Arc Path Corners

In order to facilitate the optimization of the path parameter z in the NMPFC solution, the path needs to be continuously differentiable, $\mathcal{P} \in C^1$ [Faulwasser, 2013]. This is achieved by generating circular arcs connecting the linear segments defined by the way-points specified. Given two linear

paths $\ell_i : \boldsymbol{d}_i z + \boldsymbol{w}_i$, $\ell_{i+1} : \boldsymbol{d}_{i+1} z + \boldsymbol{w}_{i+1}$ intersecting in the point $\boldsymbol{p} = (x_i, y_i)$, a circular arc of specified radius R is constructed by the following algorithm, visualized in Figure B.1. The method is based on [Alexiou, 2015] with modification.

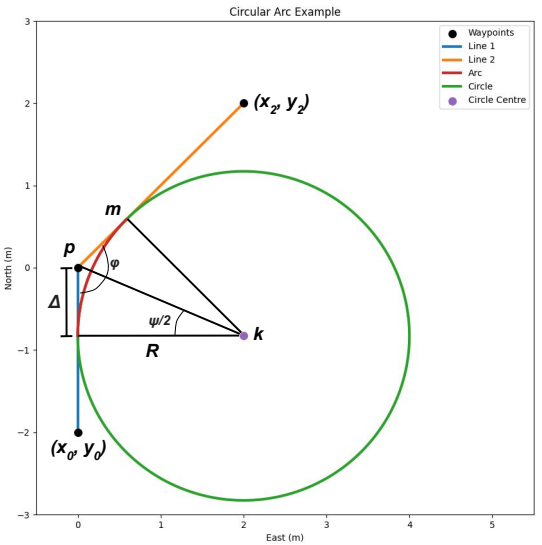


Figure B.1 Illustration that motivates the formulas behind circular arc creation. The way-points are $\{(-2,0),(0,0),(2,2)\}$, and the circle radius $R = 2$. Here, Δ is the overlap distance between the arc and the lines. The angles ψ and φ correspond to the arc interior angle and the angle between the lines, respectively. Furthermore, $\boldsymbol{k}$ is the center of the circle, $\boldsymbol{m}$ is the endpoint of the arc, and $\boldsymbol{p}$ is the intersection of the two lines. Based on [Alexiou, 2015].

The angle φ between the two directional vectors $\boldsymbol{d}_i$ and $\boldsymbol{d}_{i+1}$ is calculated based on the dot product as

$$\varphi = \pi - \cos^{-1}\left(\frac{\boldsymbol{d}_i \cdot \boldsymbol{d}_{i+1}}{||\boldsymbol{d}_i|| \cdot ||\boldsymbol{d}_{i+1}||}\right)$$

Thus, the overlap distance from the line intersection point $\boldsymbol{p}$ to the arc start and endpoints is given by the definition of the co-tangent,

$$\Delta = R \cot\left(\frac{\varphi}{2}\right)$$

Now, let $\boldsymbol{m}$ be the endpoint of the arc. It can be expressed as

$$\boldsymbol{m} = \boldsymbol{p} + \Delta \boldsymbol{d}_{i+1}$$

The handedness of the turn, i.e., if it is a left or right turn, can be determined by the sign of the determinant of the matrix $\begin{bmatrix} \boldsymbol{d}_i & \boldsymbol{d}_{i+1} \end{bmatrix}$ consisting of the directional vectors of the bounding lines. If $\det \begin{bmatrix} \boldsymbol{d}_i & \boldsymbol{d}_{i+1} \end{bmatrix} > 0$, the turn is right-handed, otherwise it is left-handed. With this information, the circle arc center $\boldsymbol{k}$ follows as a point a distance of R away from $\boldsymbol{m}$, in the perpendicular direction of $\boldsymbol{d}_{i+1}$, amounting to a 90° rotation. Here, the direction of the rotation depends on the left- or right-handedness of the arc:

$$\boldsymbol{k} = \boldsymbol{m} - r \cdot \boldsymbol{R}\left(\frac{\pi}{2}\right) \boldsymbol{d}_{i+1} \cdot \operatorname{sgn}\left(\det \begin{bmatrix} \boldsymbol{d}_i & \boldsymbol{d}_{i+1} \end{bmatrix}\right)$$

where $\operatorname{sgn}(\cdot)$ is the sign function, and $\boldsymbol{R}(\hat{\psi})$ is a 2×2 rotational matrix,

$$\boldsymbol{R}(\hat{\psi}) = \begin{bmatrix} \cos\hat{\psi} & -\sin\hat{\psi} \\ \sin\hat{\psi} & \cos\hat{\psi} \end{bmatrix}$$

This in turn enables the calculation of the coordinates of a point on the arc, $\boldsymbol{c}(\hat{\psi})$, given the angle relative to the circle arc start, $\hat{\psi} \in [0, \psi]$

$$\boldsymbol{c}(\hat{\psi}) = \boldsymbol{k} + (\boldsymbol{k} - \boldsymbol{m})\boldsymbol{R}(\hat{\psi})$$

where $\boldsymbol{R}(\hat{\psi})$ once again is the 2×2 rotational matrix, and ψ is the size of the interior angle of the arc determined as

$$\psi = \pi - \varphi$$

which can be seen in Figure B.1, since $\varphi/2$ and $\psi/2$ form a right-angle triangle, so $\frac{\varphi}{2} + \frac{\psi}{2} = \frac{\pi}{2}$. Finally, the length of the arc is $L = r\psi$.

B.3 Path Creation

The continuously differentiable path $\mathcal{P}$ is given on the following form, inspired by [Faulwasser et al., 2017]

$$p(z) = \sum_{i=0}^{N_{\mathcal{P}}-1} H(z - \tilde{z}_i)H(\tilde{z}_{i+1} - z)\boldsymbol{q}_i(z - \tilde{z}_i) \tag{B.2}$$

where H is the Heaviside step-function, $\{\tilde{z}_i\}_{i=0}^{N_{\mathcal{P}}}$ are distances along the path at which a new segment starts, and

$$\boldsymbol{q}_i(z) = \begin{cases} \boldsymbol{d}_i z + \boldsymbol{w}_i, & i \text{ even} \\ \boldsymbol{k}_i + (\boldsymbol{k}_i - \boldsymbol{m})_i \boldsymbol{R}(\hat{\psi}(z)), & i \text{ odd} \end{cases} \tag{B.3}$$

where $\hat{\psi}(z) = \frac{z\psi}{L} \cdot \text{sgn}\left(\det\begin{bmatrix}\boldsymbol{a}_0 & \boldsymbol{a}_1\end{bmatrix}\right)$ depends on the handedness of the turn,. This ensures that the circle is traversed in the right direction, and scales the mapping of the interval endpoints $\hat{\psi}(0) \to 0$ and $|\hat{\psi}(L)| \to \psi$ correctly. Furthermore, $\{(\boldsymbol{k}_i, \boldsymbol{m}_i)\}_{i=1}^{N_{\mathcal{P}}}$ are the central and final points describing the circular arcs. The number of segments, $N_{\mathcal{P}}$ is always odd, meaning that the path always starts and ends with a line segment to and from the final and starting point, respectively. A complete path "through" the points $\{(-2, 0), (0, 0), (2, 2), (1, 4)\}$ can be seen in Figure B.2.

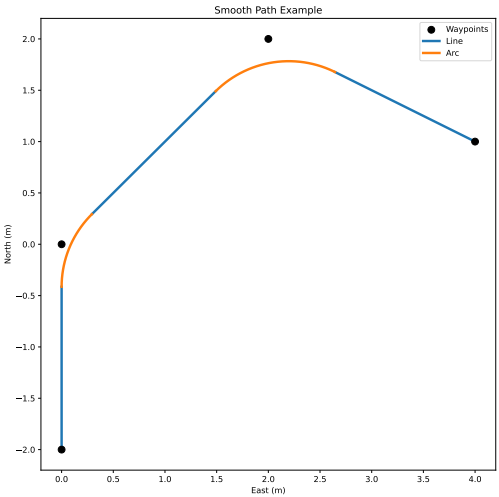
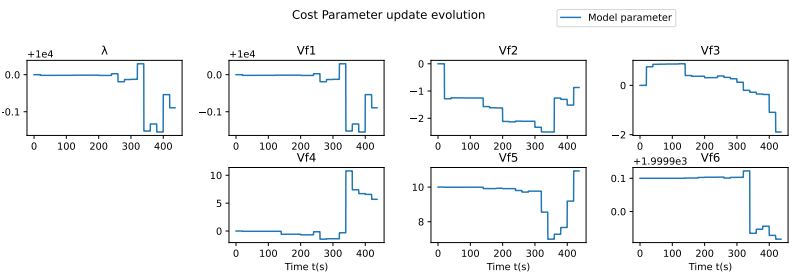


Figure B.2 Smooth path created through the points {(-2, 0), (0, 0), (2, 2), (1, 4)}, with a turning radius of 1 m.

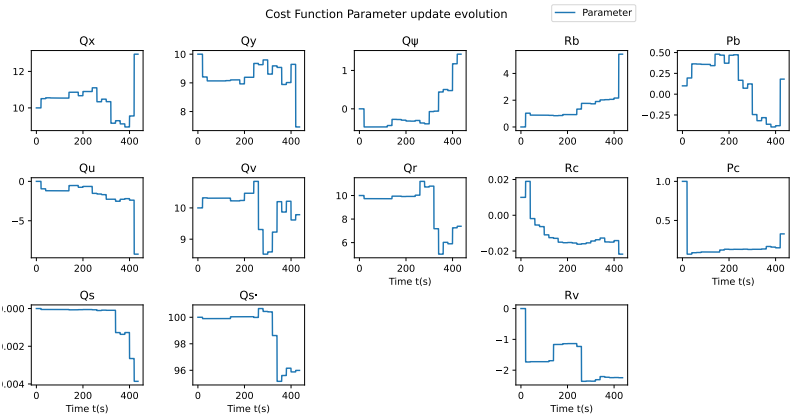
C

Reinforcement Learning Parameter Evolution

In this appendix, the evolution of all the parameters tuned by reinforcement learning are shown for the sake of completeness. First, the offline simulated NMPFC+RL run is shown in Figure C.1, and after that NMPFC+SYSID+RL is shown in Figure C.2. In Figures C.1a and C.2a the cost parameters λ and $\boldsymbol{V}^f$ are shown, while the cost function parameters $\boldsymbol{Q}, \boldsymbol{R}, \tilde{\boldsymbol{Q}}, \tilde{R}$ and $\boldsymbol{P}$ are shown in Figures C.1b and C.2b.

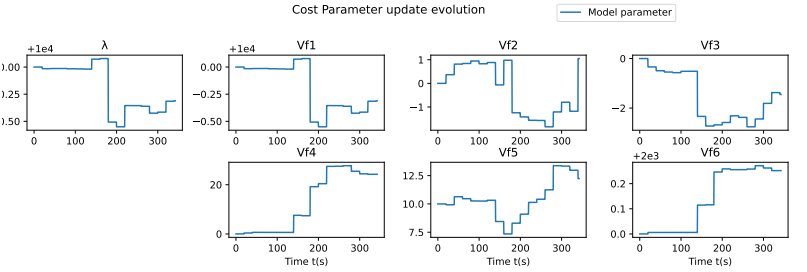


(a) The cost parameter evolution using RL on the simulated USV.

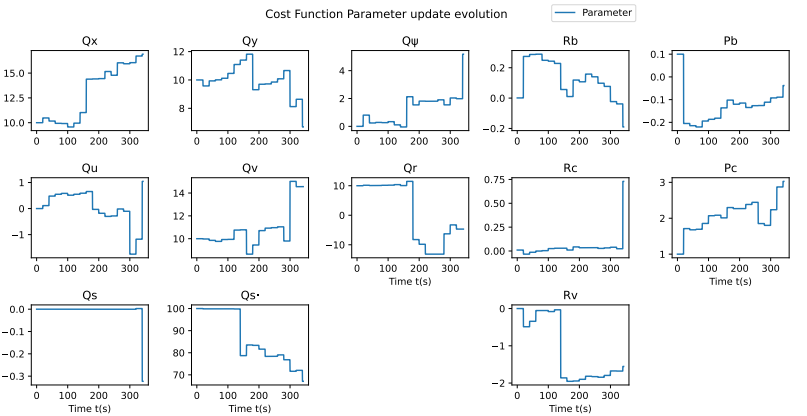


(b) The cost function evolution using RL on the simulated USV.

Figure C.1 Figure showing the RL cost parameters and cost function parameters' evolution during the USV simulation with RL enabled.



(a) The cost parameter evolution using SYSID and RL on the simulated USV.



(**b**) The cost function evolution using SYSID and RL on the simulated USV.

Figure C.2 Figure showing the RL cost parameters and cost function parameters' evolution during the USV simulation with SYSID and RL enabled.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2023

Document Number
TFRT-6205

Author(s)
Markus Svedberg

Supervisor
Birgitta Wingqvist, Saab Kockums, Sweden
Björn Olofsson, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

Data-Driven Adaptive Control of Unmanned Surface Vehicles Using Learning-Based Model Predictive Control

Abstract

In this thesis, the subject of data-driven control of Unmanned Surface Vehicles (USVs) is explored. The control task is formulated through Nonlinear Model Predictive Path Following Control (NMPFC). System identification (SYSID) and Reinforcement Learning (RL) are employed to improve performance in a data-driven manner. The objectives were to assess the resulting controller's path-following ability, as well as its adaptability to new environments, enabling the use of the controller on different USVs and under different conditions. The evaluationwas done in simulation and in experiments with the Saab Kockums AB's Piraya vessel. Based on the results presented, NMPFC gives low-error solutions in both simulation and experiments but seems non-robust against disturbances and model mismatch. The simulation results of the learning-based methods showed that enabling SYSID on a USV with an incorrect initial model would identify the correct model in one update. Moreover, applying SYSID in experiments roughly halved the USV tracking error, compared to the usage of a model identified offline. Lastly, the RL implementation was found to increase performance in offline simulation, though less than SYSID. Moreover, the RL method computational times prohibited real-time control of the Piraya. This led to the method not being deployed in experiments.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-109

Recipient's notes

Security classification

http://www.control.lth.se/publications/