Preface

This thesis was written as part of my M.Sc. degree in Industrial Cybernetics at the Department of Engineering Cybernetics, Norwegian Unitversity of Science and Technology (NTNU). The thesis is a continuation of my specialization project. I would like to thank my supervisors Morten Breivik and Emil Thryi, without whom I would have never been able to fully understand and transform the subject matter into something worth writing about.

Pivoting out of electrical engineering into the world of autonomous vessels has not been easy, for the specialization project I took it as an absolute win to simply understand what I was doing. In this thesis I was able to build upon that specialization and really experiment with the functionalities of the developed algorithm.

During the semester my supervisors have helped me with hour long bi-weekly follow up meetings, answered all of my emails, and supplied me with research material and great sources. In the meetings we discussed not only progress, but concepts and pracitcal ideas for improving my work. An extra thanks is extended to Emil Thyri for supplying me with a MATLAB simulator to use for testing the algorithm. The Algorithm was developed in MATLAB v2021b, using a framework by CasADi (v3.5.5) (Andersson et al. 2019) and an IPOPT (Wächter and Biegler 2006) solver. The algorithm also uses the MATLAB mapping toolbox as well as MSS toolbox (Fossen and Perez 2004). CasADi's example pack includes an example on direct multiple shooting which was used as a skeleton during development. Figures for the thesis were drawn using Inkscape and Draw.io. Lastly I would like to thank Olex AS for letting me use their software to store AIS data over the course of a few days to get a look at the ocean traffic along the Norwegian coast. This was used as inspiration for the creation of simulated testing scenarios.

Erlend Hestvik

11.06.2022

Abstract

I really hate writing abstracts...

Sammendrag

Jeg hater virkelig å skrive sammendrag...

Contents

List of Figures

Acronyms

AIS Automatic Identification System

ASV Autonomous Surface Vessel

COLREGs Convention on the International Regulations for Preventing Collisions at Sea

dCPA distance at Closest Point of Approach

DOF Degrees Of Freedom

IPOPT Interior Point OPTimizer

LOS Line of Sight

MPC Model Predictive Control

NED North East Down

NLP NonLinear Programming

OCP Optimal Control Problem

OS Own Ship

RK4 Runge Kutta 4th order

tCPA time to Closest Point of Approach

TS Target Ship

1 Introduction

1.1 Motivation

- Worked on the same subject matter for a ”fordypningsprosjekt” (finn godt ord).
- Autonomous vehicle control is an important milestone on the journey to a fully autonomous life.
- It’s also just fricking cool on a conceptual level.
- Fishing industries and other marine industries are ’behind the curve’ and not given as much attention as land based industries.
- A great learning opportunity for practical implementations of theory learned over the past two years.
- All in all a highly relevant project for the career trajectory I want.
- AI is pretty cool too I guess
- wanted to see if there could be a difference if autonomous vessels had more advanced prediction algorithms.
- just make something up.
- Find picture of some autonomous vessel or ferry

I remember when I took the subject TTK4115 - Linear System Theory; I thought to myself ”I’m never gonna need this MPC stuff”... funny how that worked out.

1.2 Previous Work

- cite Loe 2007 For in-depth look at many different methods.
- cite Vagale et al. 2021 For a review of path planning algorithms.
- cite Zhang et al. 2021 For another big review on navigation systems for ASV
- cite Huang et al. 2020 For another review of Collision avoidance.
- cite Park et al. 2020 For an alternative approach to Trajectory planning with similar-ish results.
- cite Vestad 2019 for a nice study in route planning and sea lanes.
- Cite someone to prove that Autonomous surface vessels are real? :thinking_emoji:
- item I *need* to use Convention on the International Regulations for Preventing Collisions at Sea (COLREGs) at least once so that the formating looks nice later.

1.3 Problem Description

- Many papers on trajectory planning emjoy simple ”cpa” prediction.
- Many algorithms end up creating a very ’active’ vessel, which is different from how most humans navigate.
- Trajectory planning and collision avoidance in one package
- Would it help if we had the tools to more accurately predict other vessels.

1.4 Contributions

- A novel MPC based path following trajectory planner that accounts for both static and dynamic obstacles.
- An evaluation of the fitness of numerical optimization as trajectory planning backbone.
- documented simulations experimenting with the difference 'Prediction Level' makes.
- documented problems that numerical optimization based trajectory planner algorithms might encounter.
- propose mitigation methods for aformentioned problems.

1.5 Outline

- Chapter 2 is background theory.
- Chapter 3 is algorithm development.
- Chapter 4 is simulation results.
- Chapter 5 is conclusion and future work.

2 Background Theory

This chapter will introduce the concepts and theory neccessary to understand the design and intent behind the trajectoy planning algorithm, as well as the discussion on it's functionality. The goal of the chapter is is to give the reader enough intuition of the applied theory that the proposed arguments and solutions should make sense. In addition, the chapter is structured so that it should be easy to quickly navigate and read about specific topics.

2.1 Vessel modelling

A mathematical model is a tool for describing physical systems and expressing how they change over time, respond to external forces, and how stable the system might be. Models are very useful when designing control systems as they translate the physical into equations that computers can understand. When making a model it is often useful to separate the dynamics of the different parts of the system we are interested in, these are the Degrees Of Freedom (DOF) the system has, and is often the directions the system can move, though they can also just be nondescript generalized coordinates. Deciding which DOF to separate out and model the dynamics of is often what separates models from each other, it is pointless to model an aspect of a system that there is no intent to interact with. For example a ship has six DOF, see Figure 1, for modelling a control system for stationkeeping all six are important because stationkeeping involves keeping the whole ship as steady as possible. When modelling for path following on the other hand it is not important what the heave, roll, or pitch of our vessel is and so the dynamics of those DOF can safely be ignored. The model used to describe our vessel in this thesis is based on the theory and notation by (Fossen 2011), and is a 3DOF nonlinear mass-damper system with thruster dynamics and no external disturbances such as wind or currents. The dynamics of the vessel can be described by the differential equations below:

$$\dot{\boldsymbol{\eta}} = \mathbf{R}(\psi)\boldsymbol{\nu} \tag{2.1}$$

$$\mathbf{M}\dot{\boldsymbol{\nu}} + \mathbf{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \mathbf{D}(\boldsymbol{\nu})\boldsymbol{\nu} = \boldsymbol{\tau} \tag{2.2}$$

where $\boldsymbol{\eta}$ is the pose of the vessel parameterized in the tangetnal plane North East Down (NED), $\boldsymbol{\nu}$ are the velocities of the vessel, parameterized in the BODY frame of the vessel. And $\boldsymbol{\tau}$ are forces and torque applied to the system. The NED frame can be said to be inertial for short distance control objectives, and in this frame the x-axis points towards true north, the y-axis points east and the z-axis points down towards the center of the planet. Thus; $\boldsymbol{\eta} = [x\ ,\ y\ ,\ \psi]^T \in \mathbb{R}^3$ are the vessel's North, East and Heading values, which are the three DOF of the system. The velocities $\boldsymbol{\nu} = [u\ ,\ v\ ,\ r]^T \in \mathbb{R}^3$ are the surge, sway, and yaw rate of the vessel. In the BODY frame there are no fixed rules for where the axis are pointing, but the common convention for modelling vehicles is that the x-axis points along the lognitudal axis of the vessel, the y-axis points along the lateral axis and the z-axis points along the vertical axis. This is also seen in Figure 1. The anchor point for the BODY frame is arbitrary but always fixed to the vessel and moves with it. The forces and toruqe $\boldsymbol{\tau} = [F_X\ ,\ F_Y\ ,\ F_N]^T \in \mathbb{R}^3$ are the forces acting along the logitudal axis, lateral axis, and torque about the vertical axis of the vessel's BODY frame. The rotation matrix $\mathbf{R}(\psi)$ rotates the BODY velocities into NED movement about the vessel's heading, and is definied as:

$$\mathbf{R}(\psi) = \begin{bmatrix} cos\psi & -sin\psi & 0 \\ sin\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

In (2.2) the $\mathbf{M}$ matrix is the inertia matrix of the system, which describes how 'heavy' the DOF are to nudge, in addition to the vessel's inherent inertia from being massive

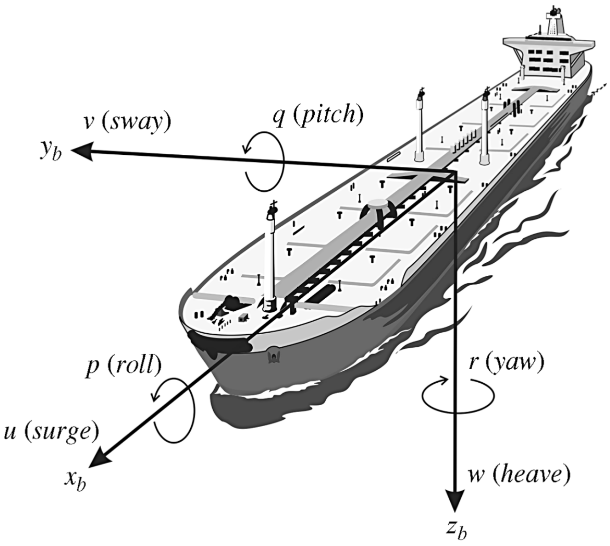


Figure 1: A ships 6 degrees of Freedom, from (Fossen 2011).

the vessel must also push water out of the way when it moves, this is what is known as hydrodynamic added mass and is linearly added to the inertia matrix. The coriolis matrix $\mathbf{C}$ also has to include hydrodynamic added mass, however for the purpose of this thesis it is not important to know the parameters for either of these matrices or for the dampening matrix $\mathbf{D}$. That is because a trajectory planning algorithm needs to work regardless of vessel parameters, (Pedersen 2019) explains more in-depth how system parameters can be found. Continuing on, the dampening matrix is a linear combination of the linear dampening stemming from water viscosity and non-linear dampening from cross-flow, once again the parameters themselves are not strictly relevant to this thesis, but intuition is important. The result are matrices in the following form:

$$\mathbf{M} = \begin{bmatrix} m11 & m12 & m13 \\ m12 & m22 & m23 \\ m31 & m32 & m33 \end{bmatrix} \tag{2.4}$$

$$\mathbf{C}(\boldsymbol{\nu}) = \begin{bmatrix} 0 & 0 & c13(\boldsymbol{\nu}) \\ 0 & 0 & c23(\boldsymbol{\nu}) \\ c31(\boldsymbol{\nu}) & c32(\boldsymbol{\nu}) & 0 \end{bmatrix} \tag{2.5}$$

$$\mathbf{D}(\boldsymbol{\nu}) = \begin{bmatrix} d11(\boldsymbol{\nu}) & 0 & 0 \\ 0 & d22(\boldsymbol{\nu}) & d23(\boldsymbol{\nu}) \\ 0 & d32(\boldsymbol{\nu}) & d33(\boldsymbol{\nu}) \end{bmatrix} \tag{2.6}$$

The dampening matrix can be a bit of a compuational nightmare and can be simplified to a linear and diagonal matrix without too much of a detrimental impact on our simulations. The justification for this simplification is the underlying assumption that the reference output from the trajectory planner algorithm will be parsed through a final control module that will account for dampening. The risk is that the end result from the trajectory planner turns out to be infeasible, but that's a problem for another thesis.

$$\mathbf{D}(\boldsymbol{\nu}) = \begin{bmatrix} d11 & 0 & 0 \\ 0 & d22 & 0 \\ 0 & 0 & d33 \end{bmatrix} \tag{2.7}$$

Finally, a word on heading vs course. Throughout this thesis the terms course and heading might be used interchangeably, but the words are strictly not synonymous. Heading is equivalent with yaw as both denote a rotation about the vessel's third axis, the difference between the two is that yaw is often a relative term describing a change by some degrees from one arbitrary pose to another. Heading is an absolute term and is often based on compass directions, meaning 0°heading equates to the nose of the vessel pointing towards true north. Neither heading nor yaw is equivalent with course, which is strictly the direction of travel relative to true north. In a simplified world void of disturbances heading and course will align during straight line travel, but external forces such as wind or currents will cause the two angles to deviate. Likevise sideslip caused by a non-zero sway velocity when turning will also introduce a deviation between course and heading (Fossen 2011). However this difference is mostly unrelated to the work put forth in this thesis, and so the terms heading and course might be used interchangeably. Althought it often makes sense to deliberately pick one term over the other.

2.2 Trajectory Planning

(TODO: Savner et lite avsnit om Shortest Signed Angle eller WrapTo2Pi.) Because the vessel dynamics are described by a model expressed as a set of time-invariant ordinary differential equations, any desired state can be reached by solving for the sequence of inputs that will take the vessel from a given initial condition to said state. In the context of this thesis "state" refers to the pose, $\boldsymbol{\eta}$, of the vessel. The simplest application of this would be moving in a straight line from point A to point B. The solution is simply to find the input sequence which turns the vessel to the correct course and then maintaining a forward speed until point B is reached. The straight vector line from point A to point B can be thought of as the desired or reference path, while the sequence of states achieved by applying the input sequence is the trajectory. Instead of having just one destination there might be multiple waypoints forming the path, and the optimal input sequence that makes the controlled vessel, from here on called "Own Ship" (OS), travel along the path depends on what criteria are considered important. A trajectory generated with fuel economy in mind might look very different from a trajectory generated with shortest transit time in mind, even if both are following the same path. Other factors such as obstacles or disturbances will also influence the trajectory, combining all the factors and generating the desired input sequence is the act of trajectory planning.

There are many methods for trajectory planning. Some are conceptually simple and fast to compute, but lack robustness and situational adaptability. Other method can be incredibly complex and and computationally expensive, but in return incredibly robust to disturbances and adaptable to any situation. An example of a simple trajectory planner would be a Line of Sight (LOS) guidance law while something extremely advanced would be training a machine learing algorithm. For an overview: in this thesis a LOS guidance law will be applied to generate a reference trajectory, the reference is then used as part of a formulation of an Optimal Control Problem (OCP) with a cost to penalize deviation from the reference in addition to other factors. The OCP is then discretized as a NonLinear Programming (NLP) problem using a method called direct multiple shooting, finally the NLP is solved with an Interior Point OPTimizer (IPOPT) solver. One of the big advantages of OCP is that it allows formulating constraints directly on the states, which is a big deal when it comes to collosion avoidance.

Line of Sight Guidance

This guidance method is perhaps the most intuitive; consider the waypoints $\mathrm{WP_k}$ and $\mathrm{WP_{k+1}}$, the simplest path from one to the other would be straight line. Therefor the most obvious control method would be to maneuver onto the straight line, and follow it along to the end. The distance of the OS to the straight line is called the cross track error y_e and the distance along the line to the end is called the along track error x_e. The along track error is not of any importance to this thesis, it is assumed that the controlled vessel will maintain a steady velocity, and there are no temporal constraints on reaching the goal.

As explained in (Lekkas and Fossen 2013); given the OS's position (x,y), the cross track and along track errors from the straigth line as defined by $\mathrm{WP_k}$ $(\mathrm{x_k},\mathrm{y_k})$ and $\mathrm{WP_{k+1}}$ $(\mathrm{x_{k+1}}, \mathrm{y_{k+1}})$ are:

$$\begin{bmatrix} x_e \\ ye \end{bmatrix} = \mathbf{R}^T(\gamma_p) \begin{bmatrix} x - x_k \\ y - y_k \end{bmatrix} \tag{2.8}$$

where R is the rotation matrix from the inetiral frame to the straight line's frame. Here, γ_p is the horizontal path-tangental angle, or the 'angle' of the straight line path in relation to the inetrial frame if that makes more sense, see Figure 2 for a visual decomposition. The rotation matrix $\mathbf{R}$ is given by:

$$\mathbf{R}(\gamma_p) = \begin{bmatrix} cos(\gamma_p) & -sin(\gamma_p) \\ sin(\gamma_p) & cos(\gamma_p) \end{bmatrix} \tag{2.9}$$

with γ_p:

$$\gamma_p = \operatorname{atan2}(y_{k+1} - y_k, x_{k+1} - x_k) \tag{2.10}$$

The control objective is to drive $y_e(t) \rightarrow 0$. as t trends towards infinity. Assuming a steady velocity this is done by selecting a course that steers the OS in the direction that reduces y_e. How fast the error y_e is suppressed is tuned by a proportional gain factor, $\varDelta$, that is often called look ahead distance. The desired heading is given by:

$$\psi_d = \gamma_p + \arctan(\frac{-y_e}{\varDelta}) \tag{2.11}$$

and consequently desired course:

$$\chi_d = \psi_d + \beta \tag{2.12}$$

where β is the sideslip of the OS. Because real life situations are rarely, if ever, devoid of disturbances that introduce sideslip and crab angles there is one common improvement that can be made: Integrate the cross track error and use both current cross track error and it's integral when calculating desired heading. The equation for $\dot{y}_{int}$ and ψ_d then become:

$$\dot{y}_{int} = y_e \tag{2.13}$$

$$\psi_d = \gamma_p - \arctan(K_p y_e + K_i y_{int}) \tag{2.14}$$

where $\mathrm{K_p}$ and $\mathrm{K_i}$ are gain parameters proportional to the lookahead distance, typically $K_p = (1/\varDelta)$, $K_i = Kp * \kappa$ with $\kappa > 0$ being some design variable.

A reference trajectory is generated by using the LOS law as described to guide the contolled vessel from it's initial position through all the waypoints, and saving the desired positions and velocities after each time step. For a path with more than two waypoints a simple index incrementation can be used when the vessel is within a certain distance from it's current target waypoint. The reference trajectory from t_0 to N (TODO: Dette er vel teknisk sett blanding av diskret og kontinuerlig notasjon) iterations of LOS applications is of the form:

$$\overline{\boldsymbol{\eta}}_{ref} = [\boldsymbol{\eta}_{t0}\ ,\ \boldsymbol{\eta}_{t+1}\ ,\ \ldots\ ,\boldsymbol{\eta}_N] \in \mathbb{R}^{3\times N} \tag{2.15a}$$

$$\overline{\boldsymbol{\nu}}_{ref} = [\boldsymbol{\nu}_{t0}\ ,\ \boldsymbol{\nu}_{t+1}\ ,\ \ldots\ ,\boldsymbol{\nu}_N] \in \mathbb{R}^{3\times N} \tag{2.15b}$$

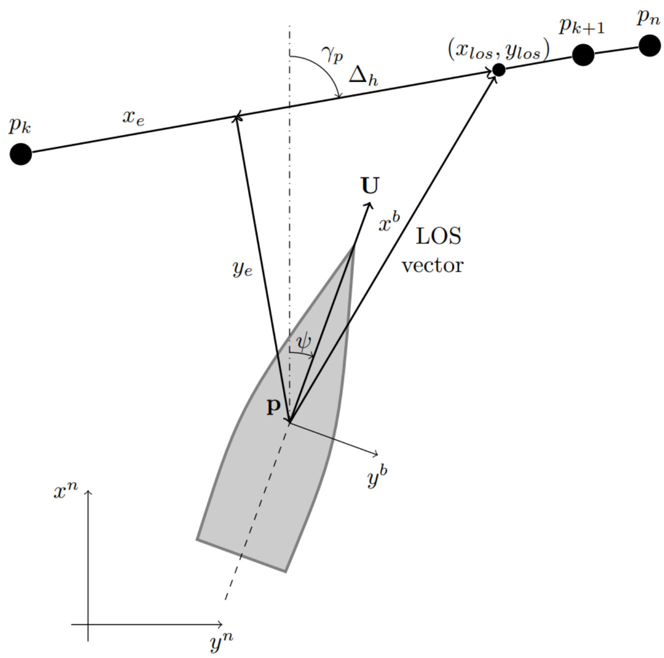


Figure 2: Line of sight guidance geometry for straight lines, here with zero sideslip. Image courtesy of (Lekkas and Fossen 2013)

Optimal Control Problem

Numerical optimization is a vast field within mathematics, (Wright, Nocedal et al. 1999) explains it well: There are no universal optimization algorithm. Instead an algorithm must be tailored to the optimization problem. Within the context of trajectory planning there are different parameters to optimize for, some examples are: maintaining steady velocity, suppressing sway, minimalizing fuel waste, minimizing distance to goal, and there are many more. The general expression for an optimization problem can be written as simple as:

$$\underset{x \in \mathbf{R}^n}{\text{Minimize}} \quad f(x) \tag{2.16a}$$

$$\text{Subject to:} \quad c_i(x) = 0, \quad i \in \mathcal{E} \tag{2.16b}$$

$$c_i(x) \geq 0, \quad i \in \mathcal{I} \tag{2.16c}$$

Where $f(x)$ is the objective function where the optimization objectives are encoded. The functions c_i are constraint on the system which $f(x)$ exists in, and $\mathcal{E}$ and $\mathcal{I}$ are indices pertaining to if the constraint c_i is an equality or inequality constraint. In the context of this thesis the thing to minimize is some nebulous cost function associated with path following, and the constraints are the physical model of the system that guarantees feasibility as well as safety constraints to avoid collisions. The cost function is then some function of the vessel's state, reference trajectory, and control input. The two constraints are the system dynamics from (2.2) and (2.1). And then additional constraints for collision safety and initial conditions. A new general OCP definition is thus given by the following:

$$\begin{aligned}
\text{Minimize} \quad & L(\boldsymbol{\theta}(t), \boldsymbol{\theta}_{ref}(t), \boldsymbol{\tau}(t)) && (2.17a)\\
\text{Subject to:} \quad & \dot{\boldsymbol{\theta}}(t) = \mathbf{J}(\boldsymbol{\theta}, \boldsymbol{\tau}) && (2.17b)\\
& \mathbf{h}(\boldsymbol{\theta}(t), \boldsymbol{\tau}(t)) \leq \mathbf{0} && (2.17c)\\
& \boldsymbol{\theta}(t_0) - \overline{\boldsymbol{\theta}}_0 = \mathbf{0} && (2.17d)
\end{aligned}$$

where L is the cost function, $\boldsymbol{\theta} = [\boldsymbol{\eta}^T \ , \ \boldsymbol{\nu}^T]^T$ and $\boldsymbol{\tau}$ is still the same as in (2.2). $\mathbf{J}(\boldsymbol{\theta}, \boldsymbol{\tau})$ Are the model dynamics (2.1), (2.2). $\overline{\boldsymbol{\theta}}_0$ are the given intial conditions of the system. The solution to the optimization problem is the series of inputs $\boldsymbol{\tau}$ which minimizes the integral of the cost L from t_0 to t_{end}. And L has the form of a quadratic function akin to a weighted least squares: (TODO: Eh, grei formulering?)

$$L(\boldsymbol{\theta}(t), \boldsymbol{\theta}_{ref}(t), \boldsymbol{\tau}(t)) = (\boldsymbol{\theta}(t) - \boldsymbol{\theta}_{ref}(t))^T \mathbf{Q} (\boldsymbol{\theta}(t) - \boldsymbol{\theta}_{ref}(t)) + \mathbf{K}_\tau \boldsymbol{\tau}^2 \tag{2.18}$$

Where the diagonal of the $\mathbf{Q}$ matrix are the weight coefficients of deviating from the reference and $\mathbf{K}_\tau$ is another weighting matrix for applied forces and torques.

Solving the OCP can be done using a multitude of methods, Eriksen and Breivik 2017 suggest discretizing the OCP into a NLP using a method called direct multiple shooting.

NonLinear Programming

The author would like to note that the technique used in this section, direct multiple shooting, is outside the scope of the author's knowledge. Everything the author knows about this technique was learned over the course of the master thesis project, and it's highly recommended to read the full formulation by (Eriksen and Breivik 2017) which is the formulation that the implementation for this thesis heavily builds upon. Another great resource for direct multiple shooting are the video lectures of (Gros 2017). Also note that functions and definitions from the previous section on OCP carry over, for example $\boldsymbol{\theta} = [\boldsymbol{\eta}^T \ , \ \boldsymbol{\nu}^T]^T$ still holds.
Direct multiple shooting is a OCP discretization technique where both the states and control inputs are excplicitly defined as decision variables. The NLP is then a reformulation of (2.17) where L is redefined as a discretized cost function with N_p control intervals steps:

$$\Phi(\boldsymbol{\omega}, \boldsymbol{\omega}_{ref_{1:N_p}}) = \sum_{k=0}^{N_p-1} ((\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_{ref_{k+1}})^T Q (\boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_{ref_{k+1}}) + K_{\boldsymbol{\tau}} \boldsymbol{\tau}_{\boldsymbol{k}}^2) \tag{2.19}$$

where $\boldsymbol{\omega} = [\boldsymbol{\theta}_0^T \ , \ \boldsymbol{\tau}_0^T \ , \ \dots \ , \ \boldsymbol{\theta}_{N_p-1}^T \ , \ \boldsymbol{\tau}_{N_p-1}^T \ , \ \boldsymbol{\theta}_{N_p}^T]^T \in \mathbb{R}^{9N_p+6}$ is a vector containing $9N_p + 6$ decision variables. Because $\boldsymbol{\tau}_k$ is the control input; it is separated out as it's own part of the function. Here, $\mathbf{Q}$ is still a sparse 6x6 matrix where the diagonal contain the tuning parameters, and $\mathbf{K}_{\boldsymbol{\tau}}$ are still tuning parameters on control input. The complete NLP will end up in the form of:

$$\begin{aligned}
\min_{\boldsymbol{\omega}} \quad & \boldsymbol{\Phi}(\boldsymbol{\omega}, \boldsymbol{\omega}_{ref_{1:N_p}}) && (2.20a)\\
\text{Subject to:} \quad & \boldsymbol{\omega}_{lb} \leq \boldsymbol{\omega} \leq \boldsymbol{\omega}_{ub} && (2.20b)\\
& \mathbf{g}(\boldsymbol{\omega})_{lb} \leq \mathbf{g}(\boldsymbol{\omega}) \leq \mathbf{g}(\boldsymbol{\omega})_{ub} && (2.20c)
\end{aligned}$$

where $\boldsymbol{\omega}_{lb}$ and $\boldsymbol{\omega}_{ub}$ are the lower and upper bounds on the permitted values for $\boldsymbol{\omega}$, this is meant to limit the decision variables to physically feasible values when solving the NLP. $\mathbf{g}(\boldsymbol{\omega})$ is a vector of constraint functions that are similarly bound by an lower and upper bounds, where the bounds define if any given function in $\mathbf{g}$ is an equality or inequality constraint. Due to the way direct multiple shooting defines the decision variables the programmed solver that solves the NLP is free to place the states and velocities anywhere within the constraints. It is therefor important to implement equality constraints that

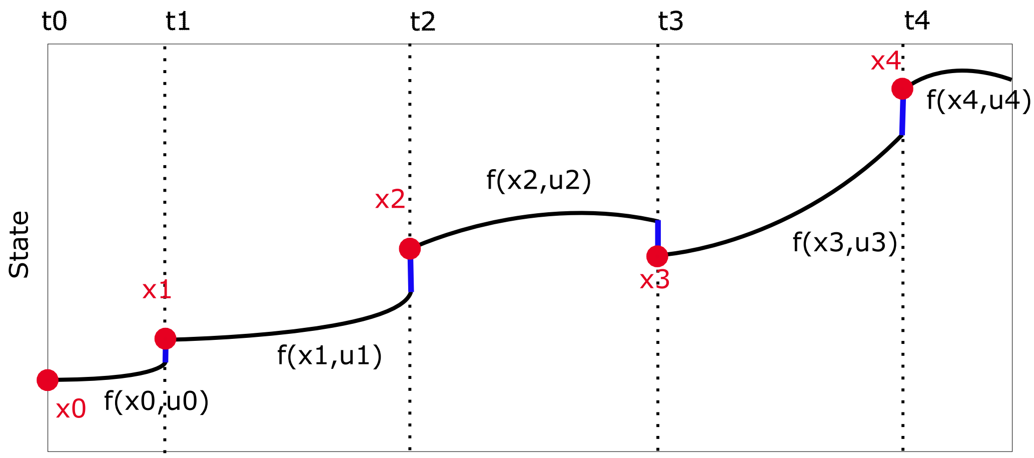


Figure 3: A physically feasible trajectory is formed by "pinching" the shooting gaps close. Reproduction from (Gros 2017).

force the ending of one control interval and the beginning of the next to line up. This is called closing the shooting gaps, an illustration of what shooting gaps are can be seen in Figure 3. These equality constraints are called shooting constraints and to create them begin by defining an integrator function $\mathbf{F}(\boldsymbol{\theta}_k, \boldsymbol{\tau}_k)$ using any technique, in this thesis the following Runge Kutta 4th order (RK4) method will be used:

$$\begin{aligned} k_1 = & \quad \mathbf{f}(\boldsymbol{\theta}_k \ , \ \boldsymbol{\tau}_k) \\ k_2 = & \quad \mathbf{f}(\boldsymbol{\theta}_k + \frac{h}{2}k_1 \ , \ \boldsymbol{\tau}_k) \\ k_3 = & \quad \mathbf{f}(\boldsymbol{\theta}_k + \frac{h}{2}k_2 \ , \ \boldsymbol{\tau}_k) \\ k_4 = & \quad \mathbf{f}(\boldsymbol{\theta}_k + hk_3 \ , \ \boldsymbol{\tau}_k) \\ \mathbf{F}(\boldsymbol{\theta}_k, \boldsymbol{\tau}_k) = & \quad \boldsymbol{\theta}_k + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{2.21}$$

with h being a selected discretized time step size. With $\mathbf{F}$ it is now possible to calculate $\boldsymbol{\theta}_{k+1}$ given $\boldsymbol{\theta}_k$ and $\boldsymbol{\tau}_k$ The shooting constraints are then formed as:

$$\mathbf{g}(\boldsymbol{\omega}) = \begin{bmatrix} \overline{\boldsymbol{\theta}}_0 - \boldsymbol{\theta}_0 \\ \mathbf{F}(\boldsymbol{\theta}_0, \boldsymbol{\tau}_0) - \boldsymbol{\theta}_1 \\ \mathbf{F}(\boldsymbol{\theta}_1, \boldsymbol{\tau}_1) - \boldsymbol{\theta}_2 \\ \vdots \\ \mathbf{F}(\boldsymbol{\theta}_{N_p-1}, \boldsymbol{\tau}_{N_p-1}) - \boldsymbol{\theta}_{N_p} \end{bmatrix} \tag{2.22}$$

Setting the lower and upper bounds for $\mathbf{g}$ equals to zero enforces the equality constraints and pinches the shooting gaps close. The final missing piece for the trajecotry planner is to formulate constraints to ensure a collision free trajectory. Similarly to the shooting constraints the obstacle constraints are also placed in $\mathbf{g}$, their formulation is discussed in Chapter 2.3

The theory behind constructing an NLP in a way that a machine can understand and solve it is a topic for a whole new thesis. In this thesis, CasADi (Andersson et al. 2019), is used as a framework for constructing the NLP, the NLP is solved with an IPOPT solver, (Wächter and Biegler 2006), that comes included with CasADi. A practical explanation of constructing and solving the NLP is the topic of Chapter 3.

Model Predictive Control

With the system dynamics modelled, and a control law formulated as an NLP it is now possible to conduct trajectory planning by selecting a discretized time step size, h, deciding how many control intervals to predict foward in time, and then solving the NLP from any initial condition (which will still be discussed in Chapter 3). Because the IPOPT solver solves for all control intervals simultaneously, its output contains the optimal trajectory as decided by the selected cost function. It also contains optimal velocities and control inputs needed to achieve the desired state, as described by the system dynamics. However, it is unrealistic to assume that the modelled dynamics are able to perfectly represent reality, blindly following the optimal trajectory is therefor a fool's errand. This is where the control technique Model Predictive Control (MPC) comes into play. MPC is a method in which the system is simulated from the present until the end of the control period. The first control inputs from the solution are saved and applied to the system for it's next control interval, the rest of the solution is then discarded and new measurements of the state of the system are taken. Using the new measurements as the new intial conditions, the process starts over; Simulate until the end of control period, apply first control input to next control interval, discard rest of solution, redo measurements, repeat. This introduces feedback to the system, which allows it to react and adjust to unmodelled disturbances, this greatly increases the robustness of the automated system. (Qin and Badgwell 1997) (TODO: usikker på den her, den er delvis relevant for det som er skrevet.)

2.3 Collision Avoidance

- Mangler kanskje et lite avsnitt om state machine for å holde på COLREGs klassifikasjoner frem til situasjonen er klarert.

Having constructed the means of finding an optimal trajectory, the next task at hand is making sure the trajectory is collision free. It would be difficult to claim any sort of optimality without asserting if the trajectory is able to effectively avoid obstacles, collision avoidance is therefor just as important a task as the construction of the trajectory planning algorithm. Collision avoidance is an umbrella term for many different smaller tasks; from risk assessment to escape maneuvers. For the purposes of this thesis it is assumed that information about obstacles in the near vicinity of the OS is readily available and not subject to disturbances or distortion. The task at hand can then be separated out into two pieces: Static obstacle avoidance and COLREGs compliance.

Static Obstacles

(TODO: kan muligens droppe hele dette delkapittelet) A static obstacle is any object or hindernace in the water that does not move on a timescale comparable to the one of the OS, such as skerries or a pier. Static obstacles are tricky, the way to handle them will depend a lot on how information about obstacles are gathered and stored. This aspect of the trajectory planning algorithm is therefor reserved for Chapter 3.3 which is about this thesis's implementations specifically.

COLREGs Compliance

The COLREGs (IMO 1972) are a set of rules developed with the purpose of preventing collisions between two or more vessels at sea. The rules are sectioned into six parts; A - General, B - Steering and Sailing Rules, C - Light and Shapes, D - Sound and Light Signals, E - Exemptions, F - Verification of Compliance. In part A it is written "These Rules shall apply to all vessels upon the high seas and in all waters connected therewith navigable by seagoing vessels.", which means any aspiring Autonomous Surface Vessel (ASV) must be able to comply. It is part B that is the most relevant to the work of this thesis, as it contains the rules for maneuvering in the vicinity of other vessels. The following is a non-exhaustive list of the rules that are most relevant for this thesis, a more comprehensive examination of the rules can be found in (Cockcroft and Lameijer 2012).

Rule 7: Risk of Collision

(d) *In determining if risk of collision exists the following considerations shall be among those taken into account:*

(d)(i) *such risk shall be deemed to exist if the compass bearing of an approaching vessel does not appreciably change;*

(d)(ii) *such risk may sometimes exist even when an appreciable bearing change is evident, particularly when approaching a very large vessel or a tow or when approaching a vessel at close range.*

Rule 8: Action to avoid collision

(a) *Any action taken to avoid collision shall be taken in accordance with the Rules of this Part and shall, if the circumstances of the case admit, be positive, made in ample time and with due regard to the observance of good seamanship.*

(b) *Any alteration of course and/or speed to avoid collision shall, if the circumstances of the case admit, be large enough to be readily apparent to another vessel observing visually or by radar; a succession of small alterations of course and/or speed should be avoided.*

Rule 13: Overtaking

(b) *A vessel shall be deemed to be overtaking when coming up with another vessel from a direction more than 22.5 degrees abaft her beam.*

Rule 14: Head-on situation

(a) *When two power-driven vessels are meeting on reciprocal or nearly reciprocal courses so as to involve risk of collision each shall alter her course to starboard so that each shall pass on the port side of the other.*

Rule 15: Crossing situation

When two power-driven vessels are crossing so as to involve risk of collision, the vessel which has the other on her own starboard side shall keep out of the way and shall, if the circumstances of the case admit, avoid crossing ahead of the other vessel.

Rule 17: Action by stand-on vessel

(a)(i) *Where one of two vessels is to keep out of the way the other shall keep her course and speed.*

(b) *When, from any cause, the vessel required to keep her course and speed finds herself so close that collision cannot be avoided by the action of the give-way vessel alone, she shall take such action as will best aid to avoid collision.*

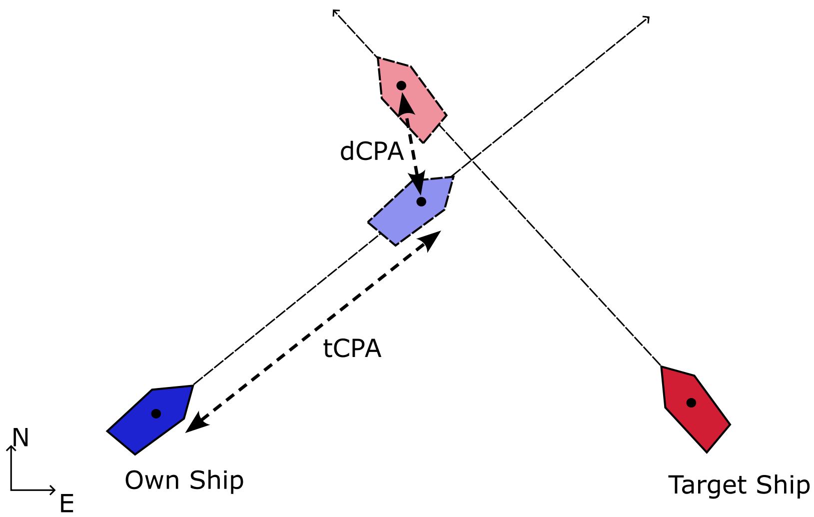


Figure 4: Visualizing dCPA and tCPA.

These rules are not easily explained to a layperson, and even less easily to a computer algorithm. To formulate the constraints that will enforce COLREGs compliance it is sensible to start by considering rule 7; is there any risk of collision between the OS and any other vessel? A common risk assessment tool is to calculate the distance at Closest Point of Approach (dCPA) and time to Closest Point of Approach (tCPA) between two vessels as shown by Kufoalor et al. 2018 and Figure 4:

$$t_{AB}^{CPA} = \begin{cases} \frac{\mathbf{P}_{BA} \cdot \mathbf{V}_{A|B}}{||\mathbf{V}_{A|B}||^2} & \text{if } ||\mathbf{V}_{A|B}|| > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.23a}$$

$$d_{AB}^{CPA} = ||(\mathbf{P}_A + t_{AB}^{CPA}\mathbf{V}_A) - (\mathbf{P}_B + t_{AB}^{CPA}\mathbf{V}_B)|| \tag{2.23b}$$

where $\mathbf{V}_{A|B} = \mathbf{V}_A - \mathbf{V}_B$ with $\mathbf{V}_A$, $\mathbf{V}_B$, $\mathbf{P}_A$ and $\mathbf{P}_B$ being the respective velocities and positions of two given vessels A and B parameterized in NED. To determine if the OS and a given TS should be considered to be in an active situation, the calculated dCPA can be compared to some lower threshold limit. If the dCPA is below the set threshold the next step is to assert which COLREGs rule is currently in effect for the OS, this is what will be called 'active COLREGs situation' for the rest of this thesis, and asserting which COLREGs is currently in effect will be called 'COLREGs classification'.

(TODO: denne delen av masteroppgaven er veldig lik fordypningsprosjekt...) There have been multiple studies on COLREGs classification, (Woerner 2016) lays out an algorithmic approach based on the relative bearings between the OS and a given TS. In this algorithm numerical values from the COLREGs rules are used as the as the criteria for determining which COLREGs sitation the OS finds itself in. The algorithm yields the expected results, but it's a bit opaque and hard to follow. (Tam and Bucknall 2010) suggests a similar approach formulated in a more natural language that is easier and more intuitive to follow. This method first considers the relative bearing from the TS to the OS:

$$\phi = \text{atan2}((E_{TS} - E), (N_{TS} - N)) - \chi \tag{2.24}$$

where (N_{TS}, E_{TS}) and (N, E) are the positions of the TS and OS respectively, and χ is the course of the OS. With the OS as a centerpiece, 4 sectors can be defined by angles offset from the OS's course, where the relative bearing ϕ deciding which sector the TS is in. Similarly the relative bearing from the TS to the OS can be used to determine COLREGs situation. See Figure 5 for a visualization.

With COLREGs situation classified the last step is to determine the constraints so that the OS behaves compliant with the rules. One method, which was written about in the author's fordypningsproject is to add circular regions as consrtaints tied to the position and heading of the TS in which the OS is in an active situation with. An example of what a singular constraint placed like this would look like can be seen in Figure 6. To achieve this the trajectory of the TS must be discretized with the same time step size as the OS. At each control interval in the NLP, using the known values for the heading and position of the TS at that instance ψ_{TS_k}, (N_{TS_k}, E_{TS_k}), calculate an appropriate constraint origin:

$$\mathbf{o_{dc}} = \begin{bmatrix} N_{TS_k} & E_{TS_k} \end{bmatrix} + H * \begin{bmatrix} \cos(\phi_c) \\ sin(\phi_c) \end{bmatrix} \tag{2.25}$$

where H is the desired distance from the center of the TS to the constraint origin, and ϕ_c is the desired relative bearing from the TS to the constraint origin. The constraints are added to $\mathbf{g}(\omega)$ the following way:

$$\mathbf{g}(\omega) = \begin{bmatrix} \vdots \\ ||\mathbf{X}_k - \mathbf{o_{dc}}|| \\ \vdots \end{bmatrix} \tag{2.26}$$

where $\mathbf{X}_k$ are the north and east positions in the decision variables $\boldsymbol{\omega}$. The square root of the lower bounds value for $\mathbf{g}(\omega)$ denotes the radius of the circle constructed by the

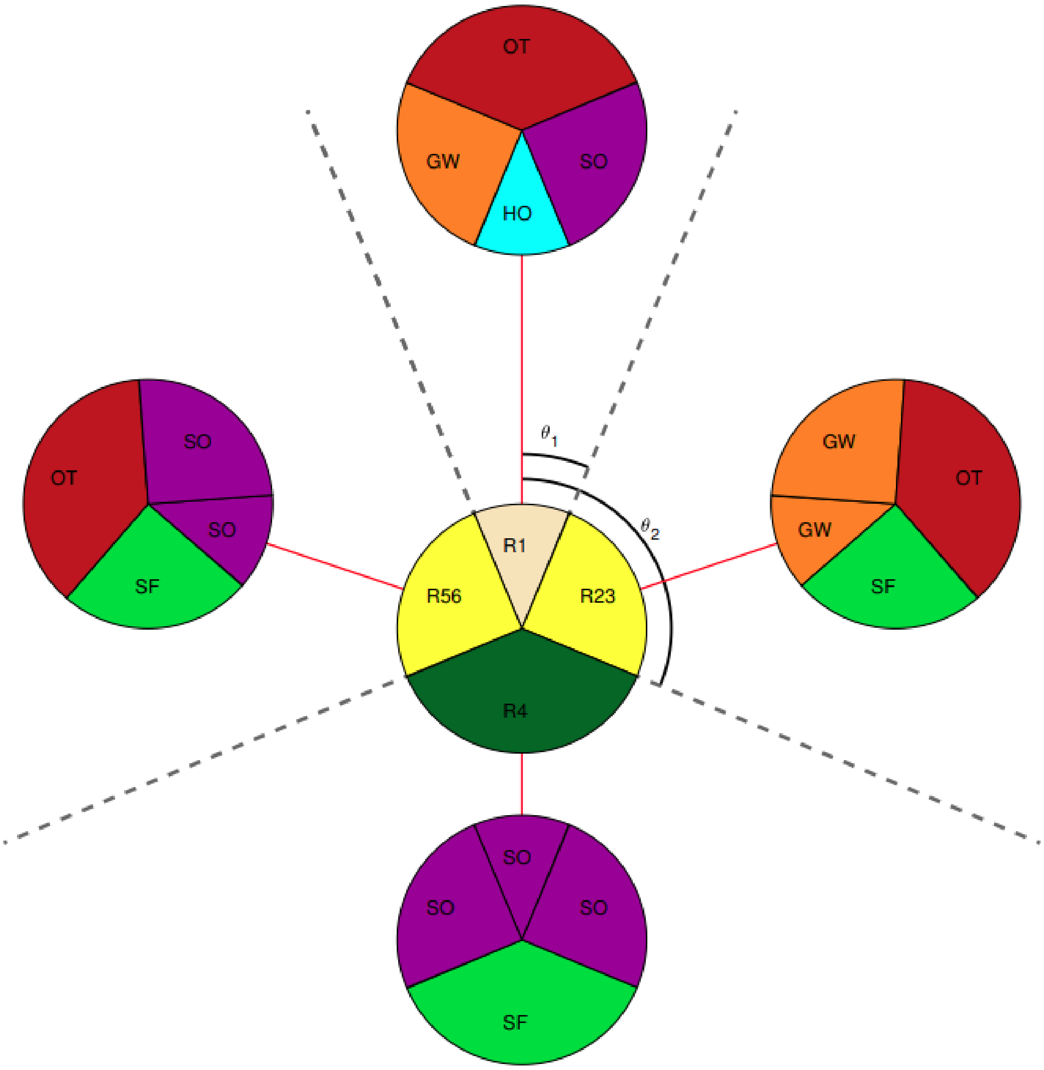


Figure 5: COLREGs classifcation; with OS in the center we can place the TS in one of four regions. Similarly the relative bearing from TS to OS can be assigned regions with region 1 pointed directly at the OS and the rest following in a clockwise rotation. Courtesy of Emil Thyri.

consrtraint function, the upper bound value should be infinite.

2.4 Target Ship Prediction

The apparent COLREGs compliance of the trajectory planner and collision avoidance algorithm can only be as good as it's ability to infer intent and predict the trajectories of other TSs. Infering intent and tracking other TS is an important task for human navigators, and a key part of collision avoidance. An ASV might have the instruments required to achieve full spatial and situational awareness, but it's ability to fully utilize these instruments is often underdeveloped.

(TODO: BRIDGE THIS GAP.)
To adress the issue of intent infering, (Cho et al. 2018) proposes a method that provides a decision-making procedure for safe navigation by predicting the maneuvering intent of TSs. In their work, a graphical model is constructed to infer intent by combining an intent model with a dynamic model. Each action of the TS influence the ship's assigned maneuvering intent probability, which denotes a probability that the ship is

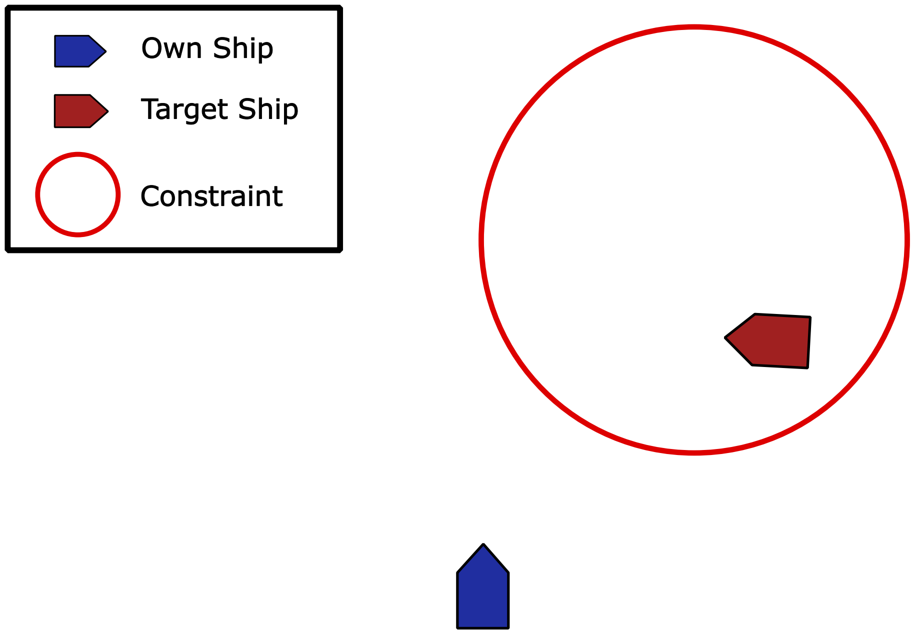


Figure 6: Example of a single placed constraint based on the position, heading, and COLREGs classification. Depicted would be a suggested placement for a Give way situation.

going to be compliant or non-compliant w.r.t COLREGs. In another study, (Schöller et al. 2021) attempts to predict the trajectory of TSs via an estimation scheme consisting of a Long Short-term Memory model in a Generative Adverserial Network configuration. The estimation scheme is backed by historical Automatic Identification System (AIS) data and outputs a probabilistic heat map of the future trajectory of TSs.

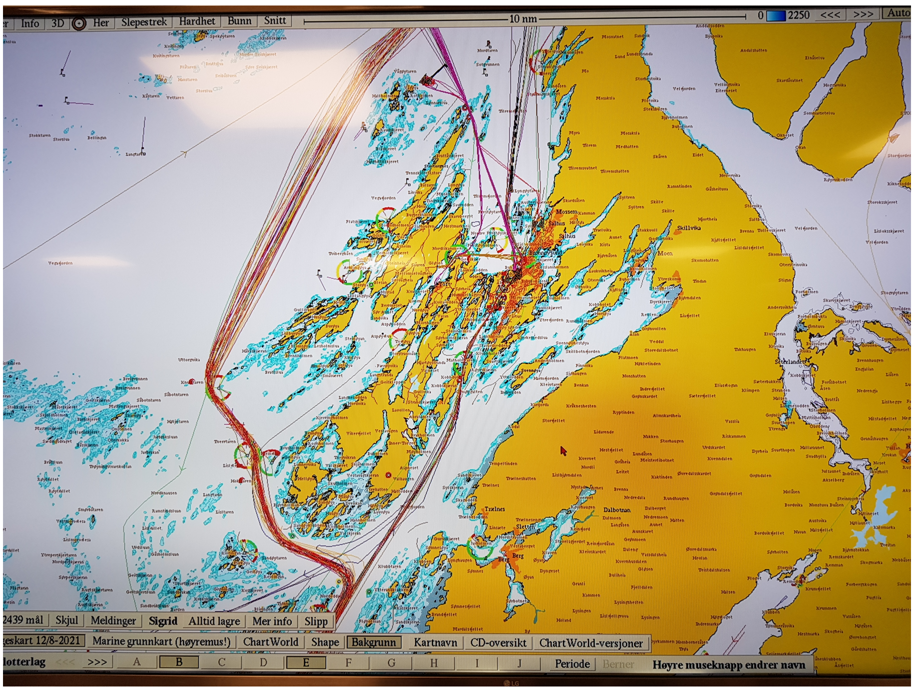


Figure 7: Photo of a typical ENC, here we can see the lines formed by saving AIS positional data over time. Image courtesy of Olex AS.

(TODO: BRIDGE THIS GAP.)
In a similar sense, (Zhang et al. 2021) notes in their survey how massive AIS data can be used for global route optimization by creating set waypoints based on the information

extracted from massive AIS data. This AIS data can easily be compiled by storing the positional data of AIS transpoders over a period of time, as see in Figure 7, which is the result of saving data for about three days.

This thesis will make no attempts at implementing a prediction method for tracking and infering the intent of TSs. However it will operate under the assumption that the technology for more accurately predicting the trajectories of TSs is coming, and one of the key points of study for this thesis is the question of how improved prediction capacity will affect the behaviour of the proposed trajectory planning algorithm. In this thesis, the prediction capabilities afforded to the algorithm will be one of two options.
1. 'simple prediction' which is simply assuming that TS will maintain a steady speed and course over ground, a common method as noted by (Huang et al. 2020) in their review.
2. 'full prediction' which is some nebulous futuristic interaction and intent aware prediction method that is able to accurately predict the future trajectory of TS.

(TODO: siste avsnitt passer kanskje bedre i discussion.)
Even if the 'full prediction' capabilities are a far off possibility, an intermediate step could be extending the information sent via common AIS tde auto-navigation data or other details on the vessels's intended near-future maneuvers, perhaps a list of waypoints and the temporal constraints for reaching them or something similar. Having information is always better than predicting or infering intent, any aiding data that can assist in correctiransponders, as these are becoming more and more common in smaller vessel, and mandatory in bigger ones. The extended AIS data could inclung the predictions made by a computer would massively benefit the development of a 'full prediction' capable algorithm.

3 Trajectory Planner

This chapter presents a step-by-step walkthrough of the developed trajectory planning and collision avoidance algorithm, and explains some of the design decisions that were made during development. Additionally the chapter will include som analyzis of problems that arose during development, and the implementationts that were made to overcome them. First the general dataflow of the algorithm is presented so that an intuition can be gained as for how the individual parts of the algorithm are connected. Secondly each step of the algorithm is presented in the order of execution from top to bottom. Lastly a brief look at how the output from the algorithm is put to use.

The author wants to stress that while the trajectory planner is presented as a finished product, the algorithm was in active development until 12 days before the thesis deadline. The code will have bugs, there will be unreadable spaghetti code, and at times logical errors. The code should have be delivered as an attachment/appendix, or it can be found on github at: (TODO: LINK).

(TODO: enten i dette kapittelet eller i discussion må det skrives om tallverdi valg på funksjoner som dcpa grense etc.) (TODO: bør vel også nevne at koden er skrevet i MATLAB? er ikke det mest brukte kodespråket i verden).

3.1 Dataflow

(TODO: eller Overview?) The core of the design is to construct a path following trajectory that is simultaneously able to comply with COLREGs and avoid getting stuck on terrain or other obstacles. The dataflow of the algorithm is depicted in Figure 8, to avoid clutter the diagram does not include every subfunction and minor detail, it's a representative diagram, not a blueprint. On the left we begin with a higher level system, the algorithm relies on getting information about it's own vessel and information about other ships, in the diagram called "tracks". Additionally static obstacles and miscellaneous other settings for both debugging and behaviour tuning is to be supplied from said higher level system.

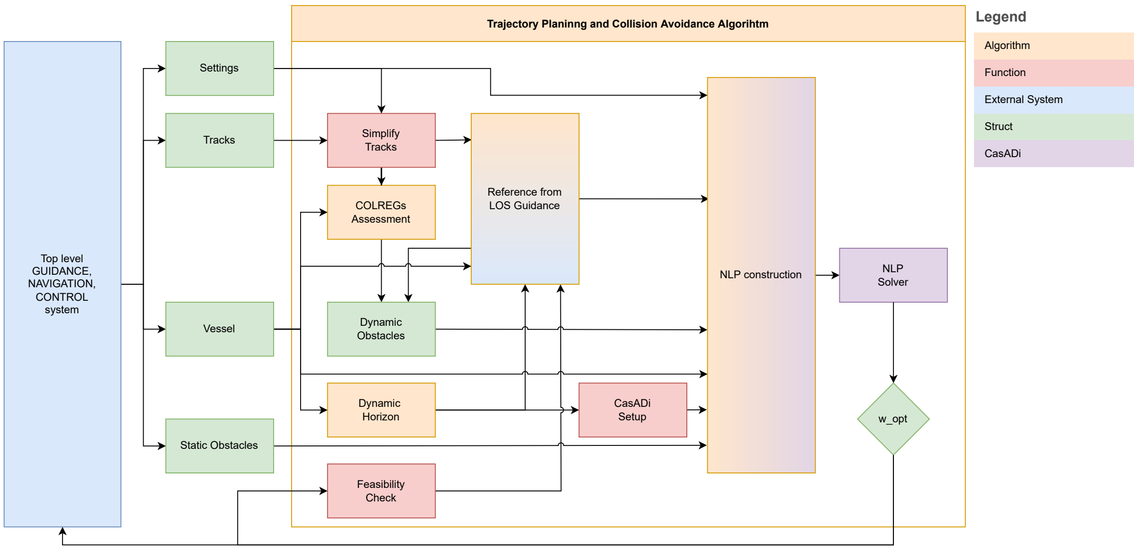


Figure 8: A simplified overview of the developed algorithm.

The algorithm is designed so that full prediction is the assumed standard prediction level. For testing and debugging purposes the tracks structure can be modified to emulate the formfactor of a simple prediction level, in a real implementation the simplification should not be neccessary as the data in the tracks struct should already be in the correct format for either prediction level. The data in the tracks struct is parsed through a COLREGs assessment algorithm to determine if any of the TSs need to be considered an active dynamic obstacle. If a TS is deemed to be active the tCPA and COLREGs sitation is determined and kept as a flag. The flag is stored in a persistent variable that the COLREGs assessment algorithm checks to avoid overwriting the classification of an active situation.

Next, the information stored in the Vessel struct, the struct dedicated to the OS, is used to calculate the desired time horizon for this call's MPC. Horizon distance and discretization step length are then needed by the CasADi setup function to create the RK4 method that discretizes the vessel dynamics. On the very first call of the algorithm the feasibility check is skipped, because there is no previous trajctory to parse, on all subsequent calls of the function the previously calculated optimal trajectory is checked for infeasibility. Feasibility, time horizon and discretization step information is then used to generate a reference trajectory for the OS, a trajectory is also generated for all the TS in the tracks struct, which are used to place dynamic constraints later.

The last step of the setup is to initialize the NLP using the OS's initial conditions, which creates the first six decision variables of $\boldsymbol{\omega_0}$ and the first six elements of the constraint function $\mathbf{g}(\boldsymbol{\omega})$. Afterwars the algorithm iterates through all the control intervals, NP, from $K = 0 : NP - 1$, as decided by discretization step length and time horizon, constructing the NLP piece by piece in the following order:
(TODO: Denne kan kanskje gjøres om til sånn fin algorithm type)
First three new decision variables $\boldsymbol{\tau_k}$ are made, the appropriate reference states are extracted from the reference trajectory, making sure that the heading reference doesn't wrap the wrong way about $[0, 2\pi]$. Then the discretized dynamics are used to integrate one control interval forward using $\boldsymbol{\omega_0}$, $\boldsymbol{\tau_k}$, and the reference values. Six new decision variables, $\boldsymbol{\omega_k + 1}$, and their shooting constraints are made. Lastly dynamic and static obstacles are placed in $\mathbf{g}$ and k is incremented by one.

After the NLP is constructed the initial guess for $\boldsymbol{\omega}$ is replaced by the previous optimal trajectory if it was feasible. When the NLP is solved the time it took is recorded, and if it didn't take too long to solve we save the solution to be used as the previous optimal trajectory for the next time the algorithm runs. Some plots for debugging can then be made if desired, and the resulting optimal states for the next control interval returned as the output of the algorithm.

3.2 Setup

The setup is all of the code that is run from when the trajectory planning algorithm is called, up until the construction of the NLP. This is the modular part of the code, where functionality can easily be added or removed without having to refactor the rest of the algorithm. Anything from new and improved situational awareness models to reference trajectory creation and COLREGs compliance ideas slot right into the setup. Of course these mentioned systems could just as well exist outside of the trajectory planning algorithm, but for this thesis it's designed as an all-in-one package.

In the current version of the trajectory planning algorithm there are four major and four minor tasks to get through in the setup. First the major tasks:

- Conduct COLREGs assessment.
- Calcualte dynamic horizon.
- Run CasADi initialization.
- Generate reference trajectories for OS and TSs

and the four minor tasks:

- Declare and initialize persistent variables.
- Initialize dynamic obstacles, and simplify tracks if needed.
- Conduct feasibility check on the previous optimal trajectory if it exists.
- fetch static obstacles, this is only a task because of how the MATLAB simulator is set up.

Persistent Variables

(TODO: vet ikke helt hvorfor jeg går så grundig til verks...) In MATLAB, persistent variables are stored in memory when a script has terminated, and is loaded back as they were the next time the script runs. This can be used to create rudimentary state machines, or to check the outcome of the previous iteration for anomalies. Persistent variables are declared without an initialization, and the method for initializing them without overwriting every time is shown in Algorithm [1].

Algorithm 1 Function: Initalize persistent variable

```
persistent Var
if isempty(Var) then
    Var ← Initial value
end if
```

Because persistent variables persist it is advised to manually clear the script when starting the MATLAB simulator, otherwise residual perisstent variables from unrelated scenario files can cause debugging problems. In the algorithm there are seven persistent variables. Two are used to store the optimal trajectory between iterations. One to store the discretized function F, so it doesn't have to be remade every time the algorithm runs. one for storing COLREGs flag to act as a state machine. One for storing a variable called firsttime, used to execute code only the first time the algorithm is called. One that enables or disables obstacles, only left intact as a debugging tool. And the last one to store the previous iterations heading reference, which is used to prevent a Wrap-to-2-Pi problem. (TODO: SKriv om hva Wrap-to-2-pi problem er for noe).

The reason there are two persistent variables to store the previous optimal trajectory is poor planning: the original optimal trajectory was dumped if it took too long to solve the NLP. But later when I implemented a feasiblity check this would cause issues since feasiblity and time-to-solve for the NLP are not neccessarily linked. Saving the previous optimal trajectory twice so that one is use for feasiblity check, while the other is the initial guess substitution candidate, was a very quick hack solution that worked well enough that it survived until the final version of the code.

Simplify Prediction

This part of the setup is only neccessary in simulations, the idea is to prepare the tracks struct so that it can be parsed by the COLREGs assessment algorithm regardless of desired prediction level. Since it is much easire to truncate excess waypoints and 'step down' from full prediction to simple, than the other way around, the algorithm was developed with full prediction level as the standard. If it's desired to step down to a simple prediction level the tracks simply need to be parsed though algorithm [2].

COLREGs assessment

(TODO: synes dette kapittelet er elendig beskrevet, vet ikke om jeg orker skrive det bedre).

Algorithm 2 Function: Simplify TS prediction

```
for i = 1 :size(tracks,2) do
    if simple then
        tracks(i).wp(1:2) ← tracks(i).eta(1:2)
        tracks(i).wp(3:4) ← tracks(i).eta(1:2)
                + 1 nmi * [cos(tracks(i).eta(3)) , sin(tracks(i).eta(3))]^T
        tracks(i).wp = [tracks(i).wp(1:2)^T , tracks(i).wp(3:4)^T]
        tracks(i).current_wp ← 1
    end if
end for
```

The COLREGs assessment algorithm solves two problems, the first is figuring out if a TS vessel will be be within close enough proximity that it should be considered an active COLREGs situation. The second problem is deciding which COLREGs situation such a encounter might be. For two vessels with constant speed and course this is a rather simple task; first apply the equations (2.23), and then run a COLREGs assessment function such as one laid out by (Thyri and Breivik 2022). However if we want to take advantage of full prediction level a bit more logic is needed to assure full coverage of the intended path. Another functionality needed to ensure COLREGs compliance is a state machine that holds onto the designated COLREGs situations, the rules state that once a situation has started it persists until both vessels are clear of each other.

The idea for the implementation in this thesis is to extend the dCPA and tCPA check so that it's conducted for every waypoint in the OS and TS reference path. That way it's possible to know roughly where both vessels should be at the dCPA point. The COLREGs assessment algorithm is therefor split into two parts; the first part is an extended CPA check, the second part is the COLREGs situation assessment.

A function getCPAlist is created to take two vessel structs as input, one as the assigned OS, and the other the TS. The function iterates through all the waypoints in the OS assigned struct and does three things:
Figure out the pose of the OS and TS.
Run the CPA check from said pose.
Calculate the time and distance to next OS waypoint so that the pose of the TS at that point can be accurately found.

Finding the pose of the OS is trivial, the first iteration of the function the pose is just the current positon and course of the vessel, for all subsequent waypoints the waypoints themselves are the position and the heading is the direction towards the next waypoint. The last waypoint does not need to be checked because at that point the OS stops moving. While this is a fantastically simple way of interating forward through the waypoints, the tradeoff is less accuracy of the CPA check when turning. Finding the pose of the TS is slightly more involved, since there are no waypoints to lean on the method here is different than for the OS. To begin we know where the TS is when the check is conducted, time and an assumption of constant velocity is then used to calculate the pose. (TODO: hvordan vise dette uten å ta med hele kodesnutten?).

With the pose of both vessels calculated, the next step is to check the CPA, which is the same equation as (2.23), the values for dCPA and tCPA, as well as the pose of both vessels are stored in vectors for later. The tCPA value is also added to a timer which is used to track how much time is passing for the OS to reach the checked waypoints. The full path CPA check is then ran with the vessel inputs flipped, so that the TS waypoints are checked, the lowest values of each CPA list are then compared to select which dCPA is truly the shortest distance. If both dCPA and tCPA are under some set threshold a COLREGs classification is set. (TODO: [...] as described? det er jo referert til tre forskjellige kilder med metoder, tror det skal være greit forstått).

Dynamic Horizon

The purpose of the dynamic horizon is to shorten the amount of control intervals when the OS approaches the final destination. The reason for this is that having many control intervals stationary at the end position can unbalance the cost function and lead to poor performance for the remainder of the path. The dynamic horizon can also be coded so that it's dynamic with respects to COLREGs situations; always having enough control intervals to clear the furthet out COLREGs situation, but less control intervals than the nominal value so the NLP can be solved faster, which would lead to a better reactive performance.

The distance between two waypoints $\mathrm{WP}_1 = (N_1, E_1)$, $\mathrm{WP}_0 = (N_0, E_0)$ is calculated with by the following equation:

$$D_1 = \sqrt{(N_1 - N_0)^2 + (E_1 - E_0)^2} \tag{3.1}$$

which means the distance to goal is the sum of all distances between waypoints:

$$D_{goal} = \sum_{\mathrm{wp}=1}^{N} D_{wp} \tag{3.2}$$

with WP_0 being the position of the OS and WP_N being the goal. Assuming a near constant velocity the time to reach goal is:

$$T_{goal} = \frac{D_{goal}}{u} \tag{3.3}$$

where u is the surge of the OS. It's not neccessary for this check to be 100% accurate, it is expected that the OS will deviate from the path due to physical constraints and obstacles anyway. To prevent accidentally dividing by zero the surge is capped at a lower bound value of $0.001 m/s$.

```
if vessel.nu(1) < 0.001
    vessel.nu(1) = 0.001;
end
```

Now is where there should have been an implementation for comparing the time to reach goal with the greatest active COLREGs situation tCPA, and picked one of them as the time horizon. Sadly that functionality ended up being implemented wrongly and therefor left out of the final version of the code because the tCPA list was not sanitized to only include active COLREGs situations. The correct way of determining time horizon should be:

Algorithm 3 Dynamic Horizon

if Any cflag is set **then**
  tempselect ← min value between T_{goal} and maxseconds
  finaltime ← min value between tempselect and $Active\_tCPA_{max} + 20$
else
  finaltime ← min value between T_{goal} and maxseconds
end if
h ← Desired step length
N ← ceil(finaltime / h)

where h and maxseconds are cosntants for this thesis, N is the number of control intervals. In the current version of the algorithm only the finaltime under the if sentence condition is hardcoded to return false, due to the afformentioned poor implementation.

CasADi setup

The casadi setup is where the vessel model and dynamics from (2.1) are implemented, as well as the cost function and RK4 method from (2.2). As the chapter title suggests, everything is implemented using CasADi's framework. The pose and velocity vectors are combined as one SX.sym vector, while the forces and toruqe is made separately. A vector for the references is also created.

```
    % System matrices.
    x = SX.sym('x',6); % x = [N, E, psi, u, v, r]'
    tau = SX.sym('tau',3); % tau = [Fx, Fy, Fn]';
    xref = SX.sym('xref',6); % xref = [Nref, Eref, Psi_ref, Surge_ref,
        sway_ref, r_ref]'
```

(TODO: Kan skrive inn matrisene på nytt så leser slipper å klikke tilbake til kapittel 2 for å huske hvordan de så ut.) The vessel mass matrix **M**, (2.4), is created with the following parameter values: while the values for **C** are $c13 = -m22 * x(5)$, $c23 = m11 *$

Table 1: Estimated model parameters for Milliampere (Pedersen 2019)

Parameter	Value	Unit
m11	2131.80	Kg
m12	1.00	Kg
m13	141.02	Kgm
m21	-15.87	Kg
m22	2231.89	Kg
m23	-1244.35	Kgm
m31	-423.76	Kgm
m32	-397.64	Kgm
m33	4351.56	Kgm^2

$x(4)$, $c31 = -c13$, $c32 = -c23$. The dampening matrix **D** was originally implemented the way (Pedersen 2019) explains, however turned out to be very computationally expensive for the IPOPT solver. For the sake of brevity, and because full scale tests using the Milliampere ferry fell through due to maintenance, the simplified diagonal matrix (2.7) was instead implemented, using some very generous dampening factors:

$$\mathbf{D} = \begin{bmatrix} 200 & 0 & 0 \\ 0 & 200 & 0 \\ 0 & 0 & 1000 \end{bmatrix} \tag{3.4}$$

The differential equation for $\boldsymbol{\nu}$ is then:

$$\dot{\boldsymbol{\nu}} = \mathbf{M} \backslash (\boldsymbol{\tau} - (\mathbf{C} + \mathbf{D}) * x(4:6)) \tag{3.5}$$

$\boldsymbol{\nu}$ can then be integrated forward one step length with simple euler integration:

$$\boldsymbol{\nu} = x(4:6) + h\dot{\boldsymbol{\nu}} \tag{3.6}$$

The equation for $\dot{\boldsymbol{\eta}}$ is the same as (2.1), with ψ being $x(3)$.

$$\dot{\boldsymbol{\eta}} = \begin{bmatrix} \cos(x(3)) & -\sin(x(3)) & 0 \\ \sin(x(3)) & \cos(x(3)) & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{\nu} \tag{3.7}$$

With everything set up the cost function L can be created and defined as a casadi function as so:

```
    %% Cost function and weights.
```

```
Kp = diag([8*10^-1, 8*10^-1]); % Tuning parameter for positional
    reference deviation.
Ku = 6.7*10^2; % Tuning parameter for surge reference deviation.
Kv = 7.2*10^2; % Tuning parameter for suppressing sway.
Kfy = 1 * 10^-5;
% Experimental cost on heading reference:
K_phi = 6*10^-5;

R2 = [cos(x(3))     -sin(x(3));...
      sin(x(3))      cos(x(3))];
Error = R2'*(x(1:2) - xref(1:2));

L = Error'* Kp * Error + Ku * (x(4)-xref(4))^2...
        + Kv * (x(5)-xref(5))^2 + Kfy * tau(2)^2...
        + K_phi * (ssa(x(3)-xref(3)))^2;

%% Continous time dynamics.
f = Function('f', {x, tau, xref}, {xdot, L});
```

The parameter values on the weights were chosen by trial and error until the trajectory planning algorithm managed to track a reference path reasonably well. cost on deviation from heading reference is actually undesirable, in a real use case for the algorithm disturbances such as wind, waves and currents would knock the heading about and cause undue increases in cost. (TODO: as discussed in the theory...) The correct course should naturally be found when velocities are restricted to only surge, the optimal way to move is in the direction of the goal after all. However one of the quirks of using numerical optimization to guide a vehicle is that the solver has no idea what a boat or a heading is. If the reference trajectory experiences a discontinous jump in course by leaving the bounds of $[0, 2\pi]$ and wrapping around the algorithm might have the great idea to turn the long way around. A miniscule cost associated with turning the wrong way was experimented with to mitigate that possibility.

The last step in the CasADi setup is to discretize the continuous time dynamics by using a RK4 method:

```
% Discrete time dynamics.
M = 4; %RK4 steps per interval
DT = T/N/M;
X0 = MX.sym('X0',6);
Tau = MX.sym('Tau',3);
Xd = MX.sym('Xd',6);
X = X0;
Q = 0;
for j=1:M
    [k1, k1_q] = f(X, Tau, Xd);
    [k2, k2_q] = f(X + DT/2 * k1, Tau, Xd);
    [k3, k3_q] = f(X + DT/2 * k2, Tau, Xd);
    [k4, k4_q] = f(X + DT * k3, Tau, Xd);
   X=X+DT/6*(k1 +2*k2 +2*k3 +k4);
   Q = Q + DT/6*(k1_q + 2*k2_q + 2*k3_q + k4_q);
end
F = Function('F', {X0, Tau, Xd}, {X, Q}, ...
              {'x0', 'tau', 'Xd'}, {'xf', 'qf'});
```

If you are wondering why the system is initialized with SX.sym but the RK4 method uses MX.sym I have no answer; CasADi's example pack includes an example for direct multiple shooting, that example includes an RK4 method on the form shown above. Since my algorithm uses the direct multiple shooting example as a skeleton this RK4 method with MX.sym was carried along until the final version.

Once we have constructed F it can be stored as a presistent variable in MATLAB, then there is no need to rerun CasADi setup potentially saving milliseconds.

Feasibility check

Later it will be shown how the previous optimal trajectory can be substituted in for an initial guess to feed the IPOPT solver. While that will be discussed later, the need for a feasibility check arose after frustration with the optimal trajectory getting stuck in an infeasible state. The first idea to check for feasibility was to somehow read the printout CasADi outputs to the MATLAB command window, but to the author's knowledge hat turned out to be impossible. Luckily there is a very easy way to conduct the check manually.

In this context feasibility means the trajectory is physically possible, a jump that covers more distance than the vessel dynamics allows means the trajectory is infeasible. To check for feasibility simply iterate through each point in the previous optimal trajectory and check the distance to the next one. Distance is calculated with the same general formula as (3.1). If the distance between two points is greater than some set limit then the trajectory is deemed to have been infeasible, which will have ramifications later. In this thesis the limit for feasibility is a very generous five meters, a lot more than the Milliampere ferry's max speed of two meters per second can move, but obviously this limit must be tuned to fit the vessel it's controlling.

(TODO: Jeg kan ta med kodesnutt for hver eneste 'task', da vil leser etter hvert få sett alle de viktigste punktene fra algoritmen.)

Reference from LOS

The theory behind this chapter was thoroughly discussed in (2.2), most of the code for this was also provided by Emil thyri, the MATLAB simulator's developer, as it is a neccessary for simulating TSs. Therefor this chapter will be brief. The logic implemented is no different from the discussed theory, one modification that was made specifically for this thesis was functionality for reducing the speed before creating the reference trajectory. If the feasibility check disucssed above deems the previous optmial trajectory to have been infeasible, something has possibly gone very wrong in the path ahead. The feasibility check says nothing about what went wrong, so in absence of information the best course of action would be to reduce the vessel's speed until the path ahead clears.

This reference trajectory does not have to be made using LOS guidance, any guidance law for path or trajectory planning can be applied as long as it is easily discretized to the same step length as the trajectory planning algorithm uses. One important criteria for picking a reference trajectory method is to consider runtime, it would be ill-adviced to use an algorithm that takes a very long time to calculate a trajectory.

3.3 NLP Construction and Solver

With the setup out of the way it's time to construct and solve the NLP. This part of the algorithm mostly consists of piecing together CasADi's framework and calculating constraints. Consturction of the decision variable vector $\boldsymbol{\omega}$ and the constraint function vector $\mathbf{g}(\boldsymbol{\omega})$ is done piecewise in the following way:

using the general algorithm, the shooting gap constraints look like this as an example:

```
    g = [g {Xk_end - Xk}];
    lbg(g_counter:g_counter+5) = [0; 0; 0; 0; 0; 0];
    ubg(g_counter:g_counter+5) = [0; 0; 0; 0; 0; 0];
    g_counter = g_counter + 6;
```

Algorithm 4 Construction of CasADi sym vectors and their bounds

Xk ← MX.sym(['X_'num2str(k)] , n) ▷ where n is the amount of elements in Xk
$\boldsymbol{\omega} \leftarrow [\boldsymbol{\omega}$, {Xk}]
$\boldsymbol{\omega}_{lb} \leftarrow [\mathrm{Xk}(1)_{\mathrm{lb}}, ..., \mathrm{Xk}(n)_{\mathrm{lb}}]$
$\boldsymbol{\omega}_{ub} \leftarrow [\mathrm{Xk}(1)_{\mathrm{ub}}, ..., \mathrm{Xk}(n)_{\mathrm{ub}}]$
$\boldsymbol{\omega}_{0} \leftarrow [\mathrm{Xk}(1)_{\mathrm{ref}}, ..., \mathrm{Xk}(n)_{\mathrm{ref}}]$ ▷ Only applicable for the decision variables

where Xk_end is the end of the previous integration step and Xk are the newest decision variables, this will be discussed soon. Here, the length of the upper and lower bounds are pre-allocated to make the NLP slighly more computationally efficient, it adds up over the course of a long simulation. MATLAB will complain about memory allocation for g, but testing showed that it was faster to do it this way than pre-allocating a list. (TODO: skulle ønske jeg tok vare på tallene jeg hadde for dette...)

Integration step

To integrate one control interval forward first create three new decision variables for forces using the general form shown in algorithm [4]. The upper and lower bounds for these decision variables are the max force and torque the engine can output. The appropriate velocity and position references are then extracted from the reference trajectory. In the case for this algorithm the reference trajectory calculates velocities in NED, so they will have to be transformed to BODY in order to be useful. The position reference is straight forward, but the heading reference needs a bit of work to make sure things don't get messy.

To begin with the reference trajectory does not output heading, but it can easily be found by considering the velocity references, which are in NED. the direction of the velocity in NED is the desired course. We allow ourselves to use this course as heading reference, even though as discussed in the theory; that's not an ideal situation.

For the first loop of the NLP construction it's important to make sure the heading reference is the same signed angle as the initial position heading. This is done by checking the difference between heading_ref - initial_heading versus wrapTo2Pi(heading_ref) - initial heading. whichever resulting angle is the smallest is the version of the heading reference we want to keep. For every k after 0 The heading reference is kept in the correct sign with the following code:

```
eta_ref(3) = previous_eta_ref(3)...
                  + ssa(eta_ref(3) - previous_eta_ref(3));
previous_eta_ref = eta_ref;
```

where ssa() is the shortest signed angle of the difference, and previous_eta_ref(3) is the heading reference from the previous control interval.

With the references gathered the discretized time dynamics are used to integrate one control interval forward, with the end states of the integration saved to use as shooting constraints, and the cost saved in an integrator variable. New decision variables are then made for the next control interval and new shooting gap constraints are made to ensure consistency between the intervals.

In the end it looks something like this:

```
Tauk = MX.sym(['Tau_' num2str(k)], 3);
     w = [w {Tauk}];
     lbw(7+k*9:9+k*9)= [-800;  -800;   -800];
     ubw(7+k*9:9+k*9) = [800;   800;    800];
     w0(7+k*9:9+k*9) = [0; 0; 0];

```

```
        % fetch reference values
        eta_dot_ref = [reference_trajectory_los(3:4,k+1);...
                  (atan2(reference_trajectory_los(4,k+2), reference_trajectory_los(3,k+2)) - ...
                   atan2(reference_trajectory_los(4,k+1), reference_trajectory_los(3,k+1))) / h];

        surge_ref = sqrt(eta_dot_ref(1)^2 + eta_dot_ref(2)^2);
        nu_ref = [surge_ref;0;eta_dot_ref(3)];
        eta_ref = [reference_trajectory_los(1:2,k+1); ...
                    atan2(eta_dot_ref(2),eta_dot_ref(1))];

        % We want the reference to start close to initial position.
        if k == 0
            unwrap_diff = abs(eta_ref(3) - initial_pos(3));
            wrap_diff = abs(wrapTo2Pi(eta_ref(3)) - initial_pos(3));

            if unwrap_diff > wrap_diff % check if distance between ref
                                       % and init_pos is greater when unwrapped
                eta_ref(3) = wrapTo2Pi(eta_ref(3));
            end
            previous_eta_ref = eta_ref;
        end

        %% Heading control
        if k > 0
            eta_ref(3) = previous_eta_ref(3) + ssa(eta_ref(3) - previous_eta_ref(3));
            previous_eta_ref = eta_ref;
        end

        xref_i = [eta_ref; nu_ref];

        % Integrate until the end of the interval.
        Fk = F('x0', Xk, 'tau', Tauk, 'Xd', xref_i);
        Xk_end = Fk.xf;
        J = J + Fk.qf;

        % New NLP variable for state at the end of interval.
        Xk = MX.sym(['X_' num2str(k+1)], 6);
        w = [w {Xk}];
        lbw(10+k*9:15+k*9) = [-inf; -inf; -inf; -2.3; -2.3; -pi/4];
        ubw(10+k*9:15+k*9) = [inf; inf; inf; 2.3; 2.3; pi/4];
        w0(10+k*9:15+k*9) = [xref_i(1); xref_i(2); xref_i(3); xref_i(4); xref_i(5); xref_i(6)];

        % Add constraints.
        g = [g {Xk_end - Xk}];
        lbg(g_counter:g_counter+5) = [0; 0; 0; 0; 0; 0];
        ubg(g_counter:g_counter+5) = [0; 0; 0; 0; 0; 0];
        g_counter = g_counter + 6;
```

Dynamic Obstacles constraints

The new control interval now needs constraints to ensure a collision free trajectory, starting with the dynamic constraints. First check if there are any constraints, if there are none the whole step can simply be skipped. Similarly we want to skip this step if it's the first time the algorithm is run, the reason why discussed later. (TODO: HUSK DETTE)

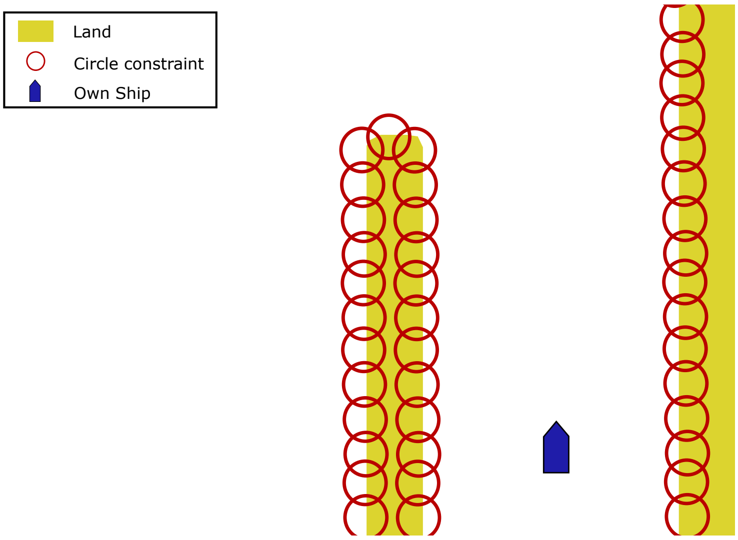


Figure 9: First approach to placing static obstacle constraints, accurate but lead to overload of constraints and poor computational performance.

Next, iterate through all the TSs in the tracks struct and check their assigned COLREGs flags. Each COLREGs situation should have their own constraint locations, however the exact optimality of placement will depend on situation complexity, available space, velocities of the ships involved, and other factors. In this thesis the placement is simplified greatly by having just pattern for each situation.

The general implementation for placing a constraint is the following function:

Algorithm 5 General function for placing dynamic constraint origin point

```
Offsetangle ← atan2(TS.traj(4,k+1) , TS.traj(3,k+1)) + offset
Offsetdir ← [ cos(Offsetangle) , sin(Offsetangle) ]
o_dc ← TS.traj(1:2 , k+1) + Offsetdist * Offsetdir
```

where offset is the angle offset from the TS's heading, and the Offsetdist is the distance from the center of the TS to the placement of the constraint origin point. The origin point is placed in $\mathbf{g}(\boldsymbol{\omega})$ as shown in (2.26) while the square of the desired radius of the constraint is placed in $\mathbf{g}(\boldsymbol{\omega})_{lb}$.

Static Obstacles constraints

A singular static obstacle $\mathcal{O}_{s_i}$ is presumed to be in the form of a polygon with n corners parameterized in NED so that:

$$\mathcal{O}_{s_i} = \begin{bmatrix} N_1 & E_1 \\ N_2 & E_2 \\ \vdots & \vdots \\ N_n & E_n \end{bmatrix}^T \tag{3.8}$$

where the point $(N_1\ ,\ E_1)$ is the first point of the polygon defining the obstacle, and the following points are sequential in either a clockwise or counter-clockwise direction. With

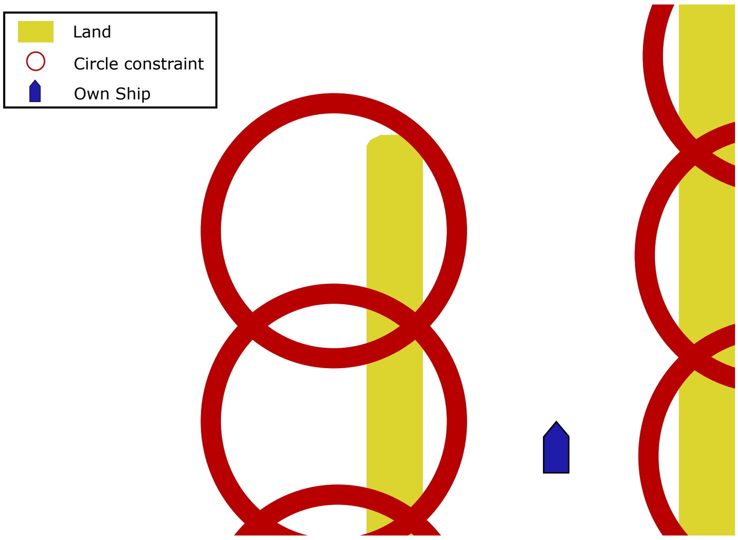


Figure 10: Second approach to placing static obstacle constraints, avoiding the constraint overload at the cost of greatly reducing available space.

N obstacles that can be put together on the form:

$$\mathcal{O}_s = [\mathcal{O}_{s_1} \ , \ \mathbf{NaN} \ , \ \mathcal{O}_{s_2} \ , \ \dots \ , \ \mathbf{NaN} \ , \ \mathcal{O}_N] \in \mathbb{R}^{2\times c} \tag{3.9}$$

where $\mathbf{NaN} = [NaN \ , \ NaN]^T$ is inserted between each obstacle to separate them, and the column dimension c is dependant on the amount and shape of the obstacles.

Properly incorporating static obstacles into the algorithm proved to be a bit of a hassle. The first solution, seen in Figure 9, was to iterate through every polygon in the static obstacle matrix, interpolating every edge to create a saturation of points that would be used as the center for circular constraints, akin to the dynamic obstacles. This was a terrible idea because it meant simulations with lots of static obstacles ended up having hundreds of thounsands of constraints, which made the NLP impossible to solve. The second solution, seen in Figure 10, was to increase the size of the circular constraints significantly so that less were needed. This worked for a while during initial testing and experiments, but eventually proved to obstruct far too much usable space. This was especially noticeable in tighter corridors of water such as a canal or near a pier. Constraints could also end up blocking the waters on the other side of a static obstacle, potentially making it impossible to traverse around oblong static obstacles. Eventually the idea of the scan lines came about. At first the scan lines were going to be used to place circular constraints, but it didn't take long to realize that reusing the logic of cross track error from LOS guidance would be a much better idea.

In the final version of the algorith, static obstacle constraints can be generated the following way:
Consider a fan of scan lines radiating from the OS with fixed length and angle. The intersection point between a scan lines and the line between any two columns in $\mathcal{O}_s$ forms the basis of the constraint $\mathbf{o}_{sc} = (o_n \ , \ o_e)$ in NED.
The constraint function is the cross track error between the position of the vessel and a line orthogonal to the scan line which crosses the point $\mathbf{o}_{sc}$, calculated like in (2.8):

$$\mathbf{o}_{sc_{ye}} = -\sin(\gamma_p) * (N - o_n) + \cos(\gamma_p) * (E - o_e) \tag{3.10}$$

where (N, E) are the north and east position of the OS, and γ_p is the angle of the orthogonal line w.r.t NED. See Figure 11 for a visualization of the geometry. The scan lines are generated at each discretized step of the reference trajectory, and every found intersection between a line the static obstacles has it's own associated cross track error that is added to the function $\mathbf{g}(\omega)$:

$$\mathbf{g}(\omega) = \begin{bmatrix} \vdots \\ \mathbf{o}_{sc_{ye}} \\ \vdots \end{bmatrix} \tag{3.11}$$

with the lower bounds of $\mathbf{g}(\omega)$ defining the allowed distance beteen the vessel and the constraint lines, while the upper bounds should be infinite. This creates a convex free set bound by the static obstacles around the reference trajectory. (TODO: kutt den siste setningen hvis jeg ikke kommer på mer å skrive.)

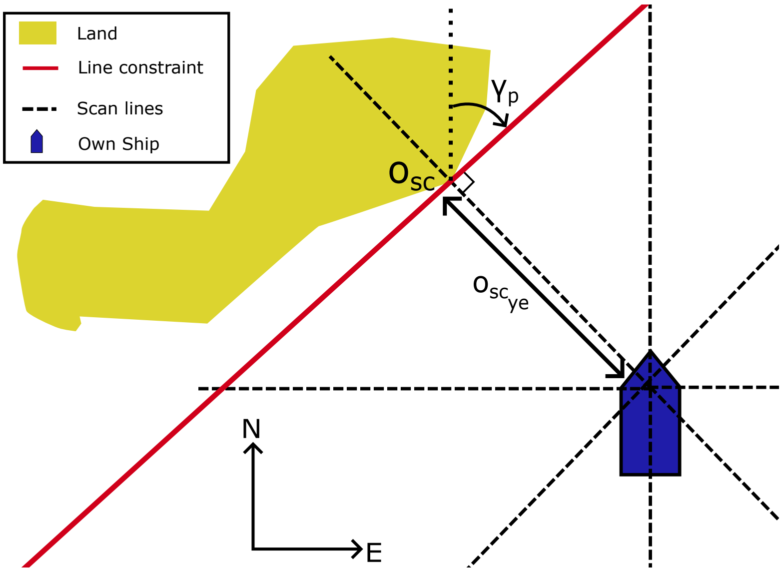


Figure 11: Geometry for straight line constraints used to handle static obstacles.

The algorithm for creating the intersection points and finding the appropriate angle for the constraint line is a bit involved. Since every control interval needs it's own set of constraints we need to iterate forward in time through decision variables that don't exist yet. The solution is to instead use the previous optimal trajectory if it exits, or the reference trajectory if we must. This will create a slight distortion in where the static obstacles end up being placed, but it the distortion is less prominent the closer we are to the current position of the OS, so it's not that big of a deal as the problem corrects itself when it draws near. (TODO: denne distortionen er ikke forklart noe særlig. det kommer jo av forskjell mellom forrige optimale bane og ny optimal bane kan føre til at statiske hindringer som burde eksistere ikke gjør det eller motsatt.)

While iterating through the selected trajectory the scanlines are fairly easy to construct, but conducting the intersection check is actually very complicated. Luckily there exists a MATLAB function for just this purpose, polyxpoly, which comes as part of the MATLAB mapping toolbox. Polyxpoly outputs the x, y, and an index for which polygon edge were part of the intersection. It does this for every intersection it find, when we run the check using all the scanlines and all the static obstacle polygons at the same time this can lead to some duplicate hits. In a similar vein it is possible for a scanline to completely pass through an obstacle polygon, in which case it would intersect twice, with the second intersection being on the backside of the obstacle. Both the backside intersection and duplicate intersections are undesirable, the output from polyxpoly can be "sanitized" with the following code snip:

```
[xi, yi, ii] = polyxpoly(x,y,xbox,ybox);
% Keep first hit:
A = [xi, yi, ii];
[~,uidx] = unique(A(:,3),'stable');
```

```
    A_without_dup = A(uidx,:);
    xi = A_without_dup(:,1);
    yi = A_without_dup(:,2);
    ii = A_without_dup(:,3:4);
```

The static obstacle constraint parameters can then be collected by combining the xi and yi vectors into a single matrix, keeping in mind that the output has the x and y axis opposite from what we have gotten used to by now. The angle of the constraint can then be calculated using some spaghetti code that should have instead been a lookup table if the author were a bit smarter. The implementation for calculating the angle γ_p (here called pi_p) ended up being:

```
    static_obs_constraints = zeros(3,length(xi));
    for i = 1:length(xi)
        intersectionpoint = [yi(i); xi(i)];
        %horrible 2am spaghetti:
        line = pos - intersectionpoint; % The vector that takes us from intersection point current position
        transposedline = [-line(2);line(1)]; % Get Orthogonal of said vector.
        tangent = intersectionpoint + transposedline; % create point along orthogonal vector

        pi_p = atan2(tangent(2) - intersectionpoint(2), tangent(1) - intersectionpoint(1));
        static_obs_constraints(:,i) = [intersectionpoint(1); intersectionpoint(2); pi_p];
    end
```

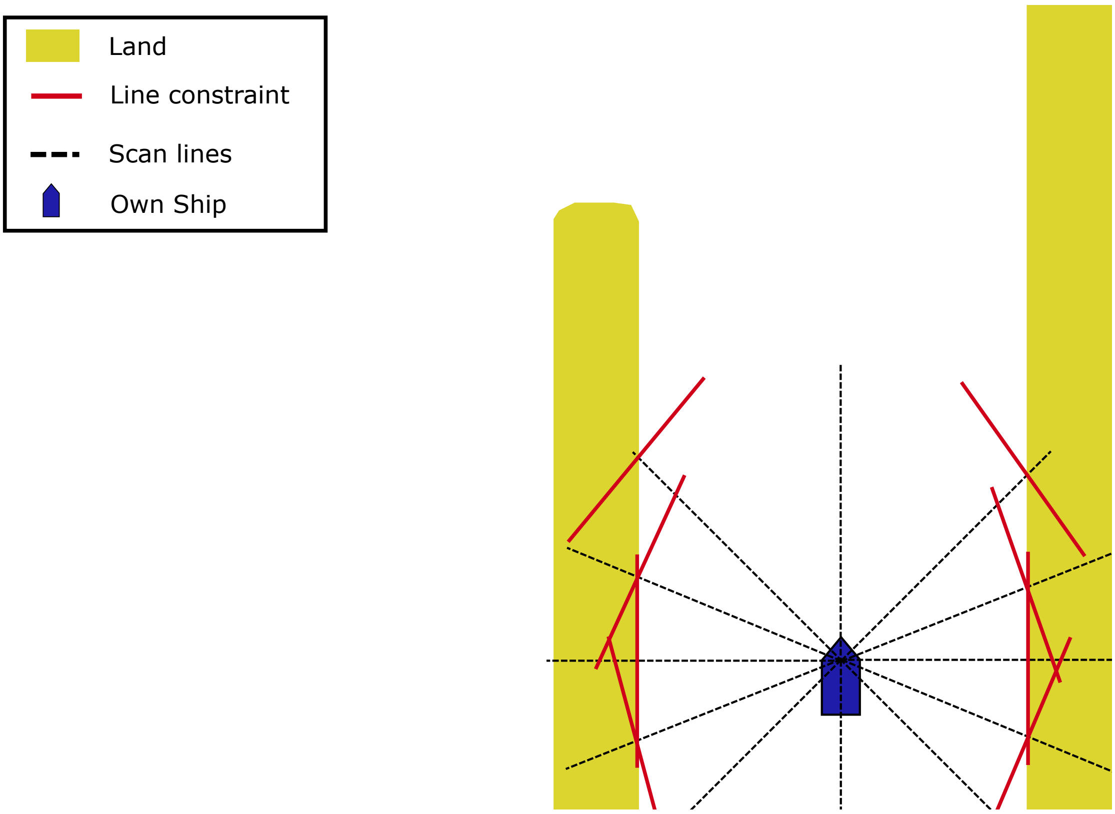


Figure 12: Current approach to placing static obstacle constraints, ditching the circular constraints in favor of straight lines based on proximity. Combines the best of both prior versions.

where pos is the position of the OS. Instead of all this the angle between the intersection point and OS should indicate which scanline resulted in said intersection, each scanline can only have one orthogonal vector and they can be pre-calculated and put in a list. you then only have to grap the correct index from the list to get your angle pi_p. The constraint function placed in $\mathbf{g}(\boldsymbol{\omega})$ is Equation (3.10), with the lower bound value for g deciding how close the OS is allowed the line. In this thesis that lower bound is hardcoded to be 5 meters, but just as with dynamic obstacles this value should really be some function of the complexity of the situation.

Solver

After the construction of the NLP is finished a solver instance is created with CasADi, selecting IPOPT as the desired solver. Additionally the solver instance is able to take in a few options for changing tolerances and tweaking other aspects of the solver. There are three options which are very useful for the algorithm. The first is options.ipopt.max_iter, which lets us set a hardcap on how many iterations the IPOPT solver is allwed to use. Great for reducing runtime. The second option is the options.ipopt.print_level, which controls how much information is printed to the command window, this has no actual effect solver, but printing to command window takes time. lowering the print level is great for running simulations faster.

In the final version of the algorithm I've set options.ipopt.max_iter to 200 for the first time the algorithm runs, and 400 for the rest. The reason is that the IPOPT solver generally gets very close to a solution in just a few iterations, but then takes a really long time to get all the way to the finish line. For the first iteration there are no obstacles enabled, if the solver can't get to the optimal solution in 200 iterations then it's not gonna get to one in 2000 either. With obstacles enabled 400 iterations seems like a nice compromise between wanting a fast solution, and giving the solver enough tries to get reasonably close to an optimum.

After setting the options and creating the solver instance, the very last task left before solving is to substitute the initial guess $\boldsymbol{\omega}_0$ with the previous optimal trajectory if was deemed feasible by the feasibility check. Having an initial guess that is close to a optimal solution makes the NLP a lot easier to solve. This is also why the initial guess sym is filled with the reference values while constructing the NLP, it's a lot better to have some initial guess than to guess 0 for everything (Gros 2017). Because the amount of control intervals will vary between calls some logic is implemented to make sure the new $\boldsymbol{\omega}_0$ is of the same size. Either by grafting on some of the reference values or by trimming the end, depending on if it's too long or short.

The solver instance is then executed and timed, as long as it took less than 30 seconds to solve we save the result for next time otherwise next iteration will have to rely on the reference as a initial guess.

4 Simulation Results

To test the capabilities of the trjectory planning algorithm it is useful to conduct simulations of various scenarios. With a simulator it is possible to cover a wide assortment of scenarios in a timely fashion, this helps explore the full range of the algorithm's behaviour without having to conduct time consuming full scale tests. NTNU also has a full-scale functional prototype of an autonomous ferry that could be used to conduct real life tests. However during the period of working on this thesis the ferry was out of commission due to a thruster failure. The MATLAB simulator emlpoyed for this thesis was developed by Emil Thyri and is used with permission. In this chapter the results are presented with figures to show the developemnt of the scenario over time, in addition to these figures there exists a youtube video compiling all the results in video format, the video can be found as an attachment to the thesis, or by following this link: https://www.youtube.com/watch?v=522OtL2MRGo.

All the simulations are conducted under the assumption that the OS has perfect vision for spotting and tracking dynamic obstacles. Disturbances are also largely ignored, the simulation features no current or wind induced sideslip, crab angle is also not considered.

4.1 Scenario Overview

The scnearios used for this thesis are constructed to test both trajectory planning and collision avoidance capabilities through a combination of both trivial and complex situations. The scenarios are also designed so that behaviour differences between full and simple TS prediction can be observed. Any time we encounter a TS that maintains a steady course and velocity there will not be any observable difference, therefor most of the scenarios are constructed so that encounters occur when ships are turning. The first set of scenarios are simple situations to establish baseline behaviour in the various COLREGs situations. In these scenarios there are only two agents and there are mostly no meaningful differences observed between simle and full prediction of TSs. The second set of scenarios are more complex by featuring more agents and longer paths to follow. These scenarios often feature multiple COLREGs situations that can even overlap, additionally TSs will not be considerate of the OS and will exhibit reckless behaviour in order to test a sort of worst case scenario. The complex scenarios also incorporate static obstacles to show how the algorithm handles both types of obstacles at the same time.

Simple COLREGs Situations

These scenarios feature two agents, the OS and the TS, each entering a fully open space while maintaining a steady course and fixed speed. The agents then cross in manners as described by the COLREGs rules discussed in prior chapters.

Turning COLREGs Situations

Similar to the simple COLREGs situations these scenarios all feature two agents who enter a fully open space. The difference is as the name implies that these scenarios feature a turn by the TS. Shortly after both agents are in motion the TS will alter it's course, changing the COLREGs situation from one apparent situation to another.

Canals

This scenario features a set of canals that form a T-junction as well as a choke point on one of the junction points that restricts the traversable space. There are three agents present and they all meet roughly at the choke point, the scenario is set up so that the dynamic constraints of the TSs completely block the path of the OS if full prediction is used.

Fjord

The fjord is construct as a miniature version of the Trondheimsfjord, this scenario is designed as a stress test of COLREGs situations. With multiple TSs crossing, turning and overtaking the OS simultaneously this scenario will show how the trajctory plannign algorithm differs with prediction level.

Helloya

The situation in this scenario is specifically modelled after a spot near Brønnøysund and is not an entirely uncommon situation when in transit along the coast of Norway. Traffic that wishes to avoid the narrow pass leading in to or out of Brønnøysund's will elect to take a wider path on the outside of the local archapeligo. The result is a path with a very prominent turn that is invisible at a glance, but very obvious to any experienced navigator. The simulation is conducted with the OS arriving from both the north and south direction with both full and simple prediction enabled.

Skjærgård With Traffic

Skjærgård is a Norwegian term for a section of ocean where there are many small islands and skerries, while the term translates to archapeligo a skjærgård is generally small in scale. This scenario puts a lot of stress on the trajectory planner which has to deal with both moving dynamic obstacles as well as the static obstacles that are sometimes blocking the reference path.

Skjærgård Without Traffic

A simpler version of the previous scnario, this time with no traffic but with more skerries near the reference path.

Miscellaneous

These scenarios are not meant to simulate any specific situation, rather these are meant to showcase quirks, features, and bugs encountered while developing and testing the algorithm. While some of the problems shown here were taken care of and are no longer present in the current iteration of the algorithm they are nonetheless important to showcase and discuss.

4.2 Results

Following this chapter there will be a lot of very similar looking figures. The left figure will always feature the already travelled trajectories, as well as the active static obstacles immediately around the OS and the first ten dynamic constraint circles. The figure on the right features the projected optimal trajectory, the reference path and trajectory, and active dynamic obstacle constraint origin points. Both figures feature velocity vectors on all vessels, and for both figures the sizes of the vessels are exaggerated so that they're visibile. Each scenario will be briefly looked at on it's own, pointing out obervations or discrepancies from what one would expect. If a scenario did not show any significant difference between prediction levels the figures for the simple prediction were not included, the accompanying shows every scenario in full, both with simple and full prediction. Afterwards some typical issues, quirks and problems will be looked at before a general discussion caps off the chapter.

While the figures are pretty to look at, it is highly recommended to watch the video results to see the full picture.

One thing to disclose before jumping in: Over the course of the thesis these simulations were ran many times, and while the results are always consistent for consecutive reruns, the simulation seem to be very sensitive to daily cosmic radiation. The simulations could be run one day, then something unrelated in the MATLAB files were changed, and then the results would be different the next day. This is important because the plots had to be remade a few times for the thesis, but not for the youtube video. There are some discrepencies between the two versions, but the overall results are still the same.

rest of page intentioanlly left empty for formating

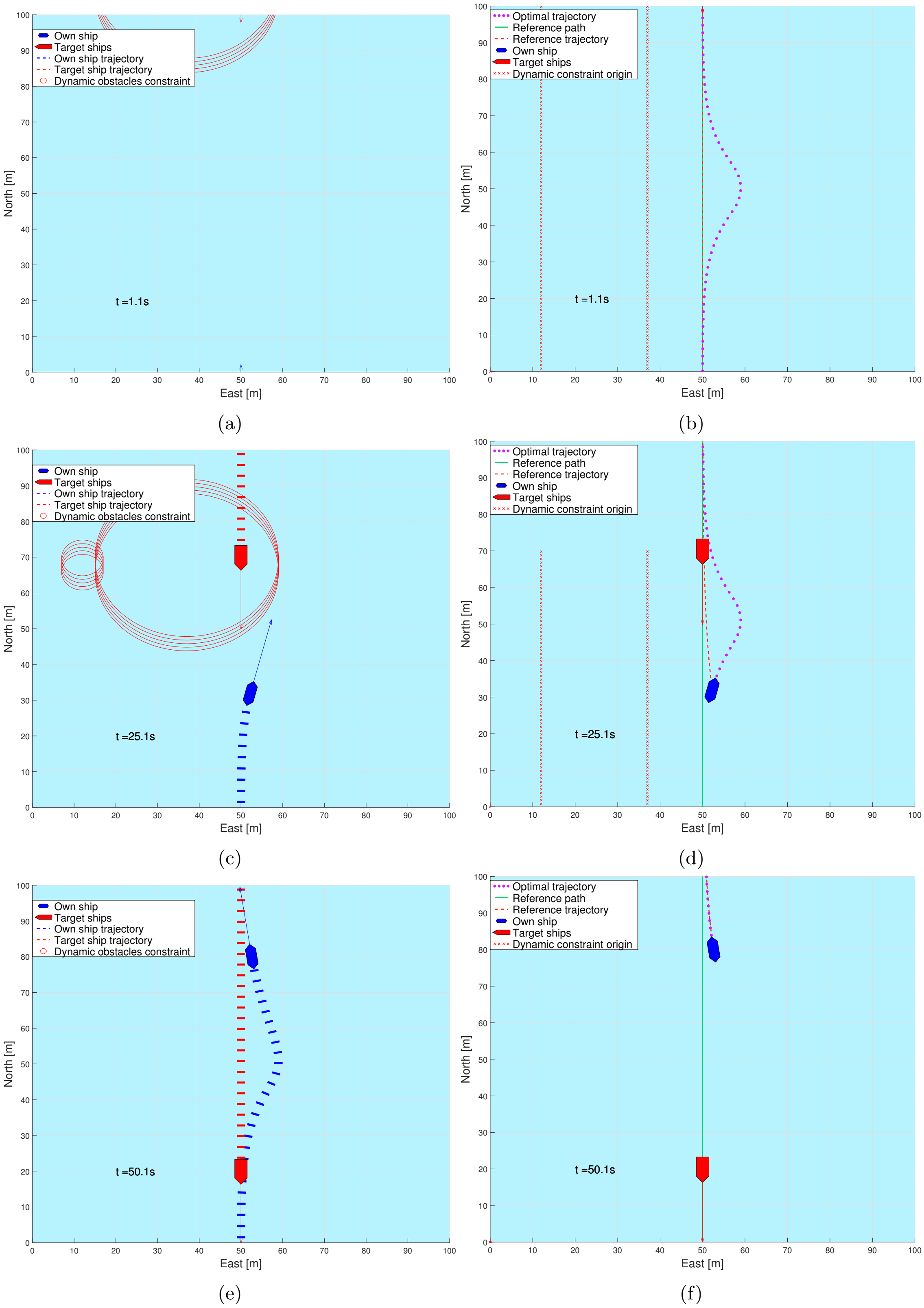


Figure 13: Simple Head on situation. Result independent of prediction level.

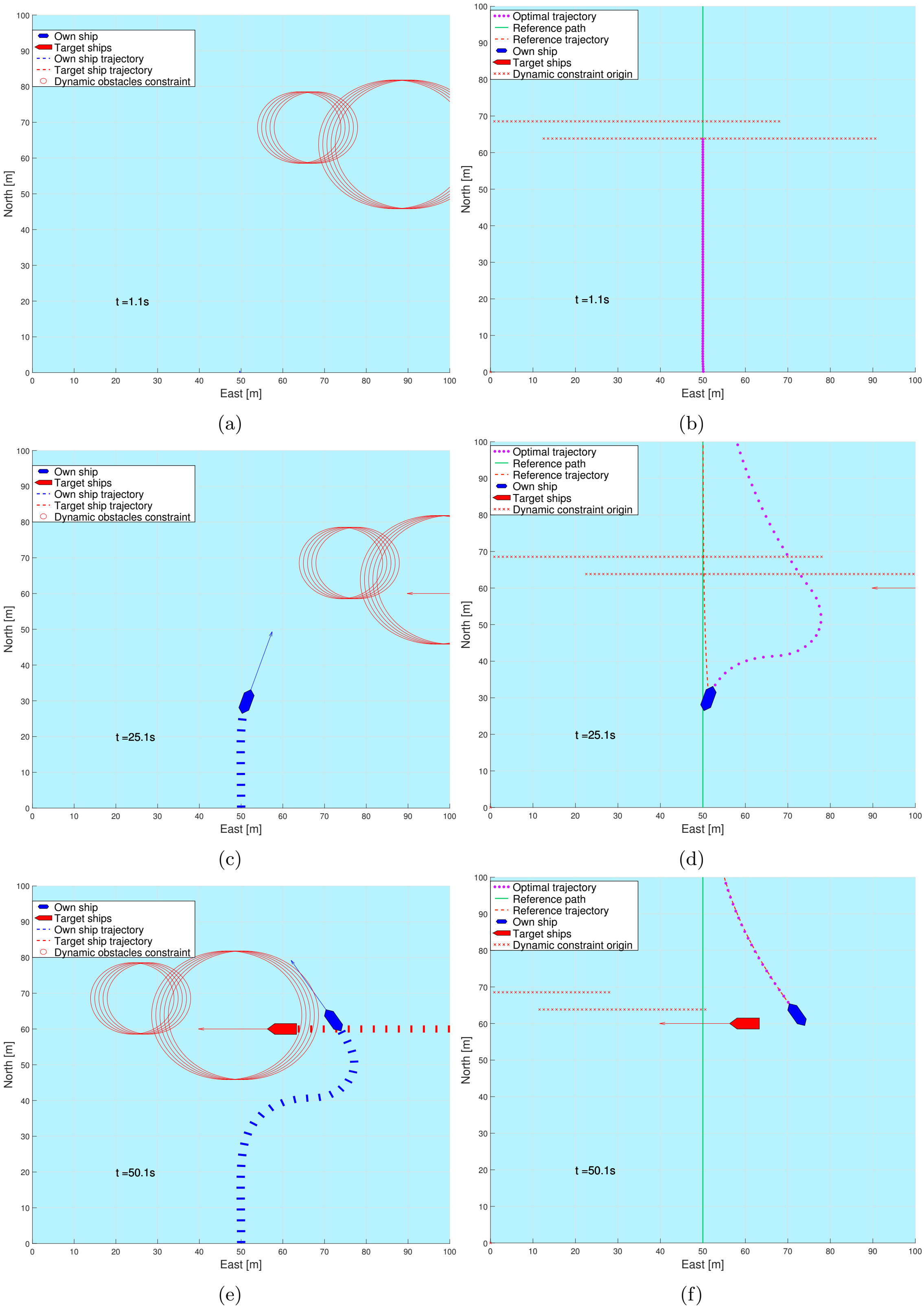


Figure 14: Simple Give Way situation. Result independent of prediction level.

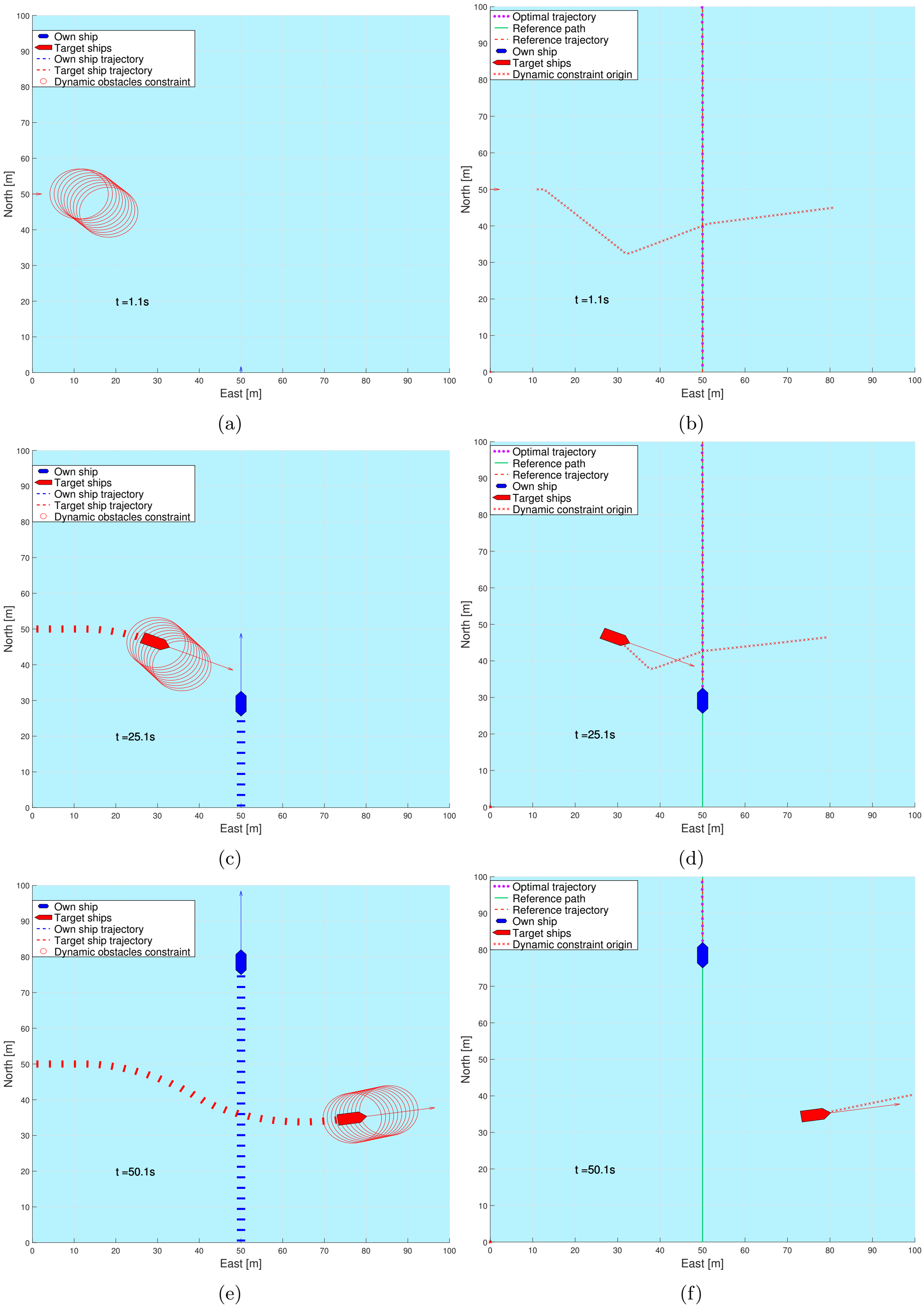


Figure 15: Simple stand on situation. Here shown with full prediction, OS correctly stands on.

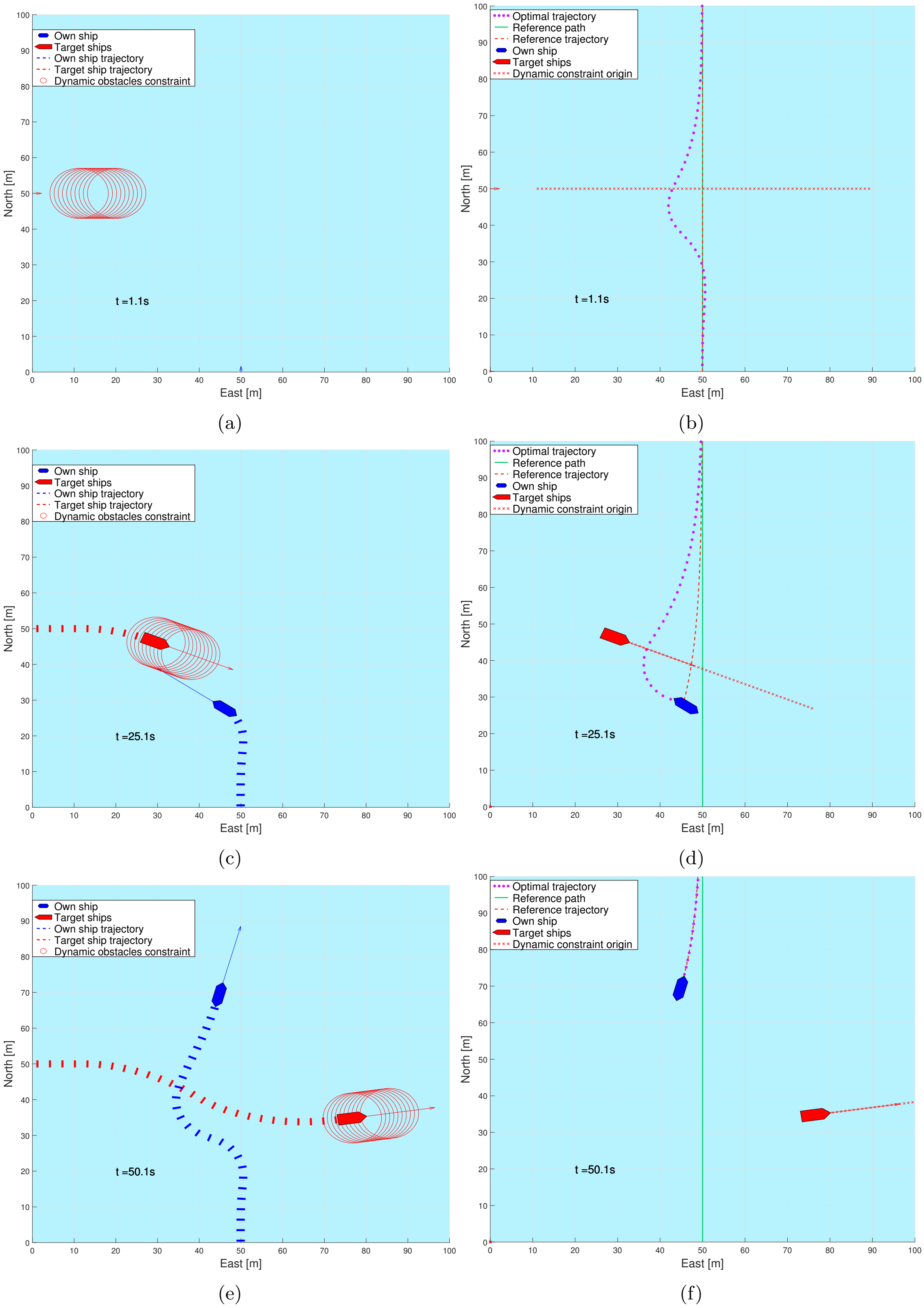


Figure 16: Simple stand on situation. Here shown with simple prediction, the OS can be observed to yield when it shouldn't.

4.2.1 Simple Head On

The simple situations are not really meant to test anything, rather this scenario and the others are used for establishing a baseline of capabilities. For example for this scenario specifically it is established that the OS will always attempt to pass the head-on vessel on the port side when the conditions are ideal. This results holds for both simple and full prediction levels. The result for this scenario is seen in Figure 13

4.2.2 Simple Give Way

This scenario shows how the algorithm behaves when giving way to a crossing vessel. It might actually be a bit overzealous with this constraint size, but that's something that can be tuned with further experimentations. One important observation to make from this simple scenario is when obstacles are enabled on the second iteration of the algorithm, the constraints block the previous optimal trajectory and causes it to be infeasible. This is more clearly observed in the video version. When the infeasibility is detected the next result is a much shorter path as the speed is reduced, and then finally the full trajectory that gives way is found. This very simple simulation highlights one of the potential problems with the algorithm, if it's turned on while the OS is in an active COLREGs situation performance might be poor. The results are seen in Figure 14

4.2.3 Simple Stand On

This scenario is one of the only two scenarios that assumes a cooperative TS. This is because controlling the TSs to be cooperative turned out to be a real time sink, and so it was dropped for all larger simulations. This is also the first simulation where a difference between simple and full prediction can be observed, and it's mostly the fault of poor constraint placement, which will be more thoroughly discussed in Chapter 4.3. The full prediction result can be seen in Figure 15 and shows that the OS does not deviate from it's speed or course, which is exactly the behaviour we would want. The simple prediction version on the other hand, seen in Figure 16 turns to cross behind the incoming TS.

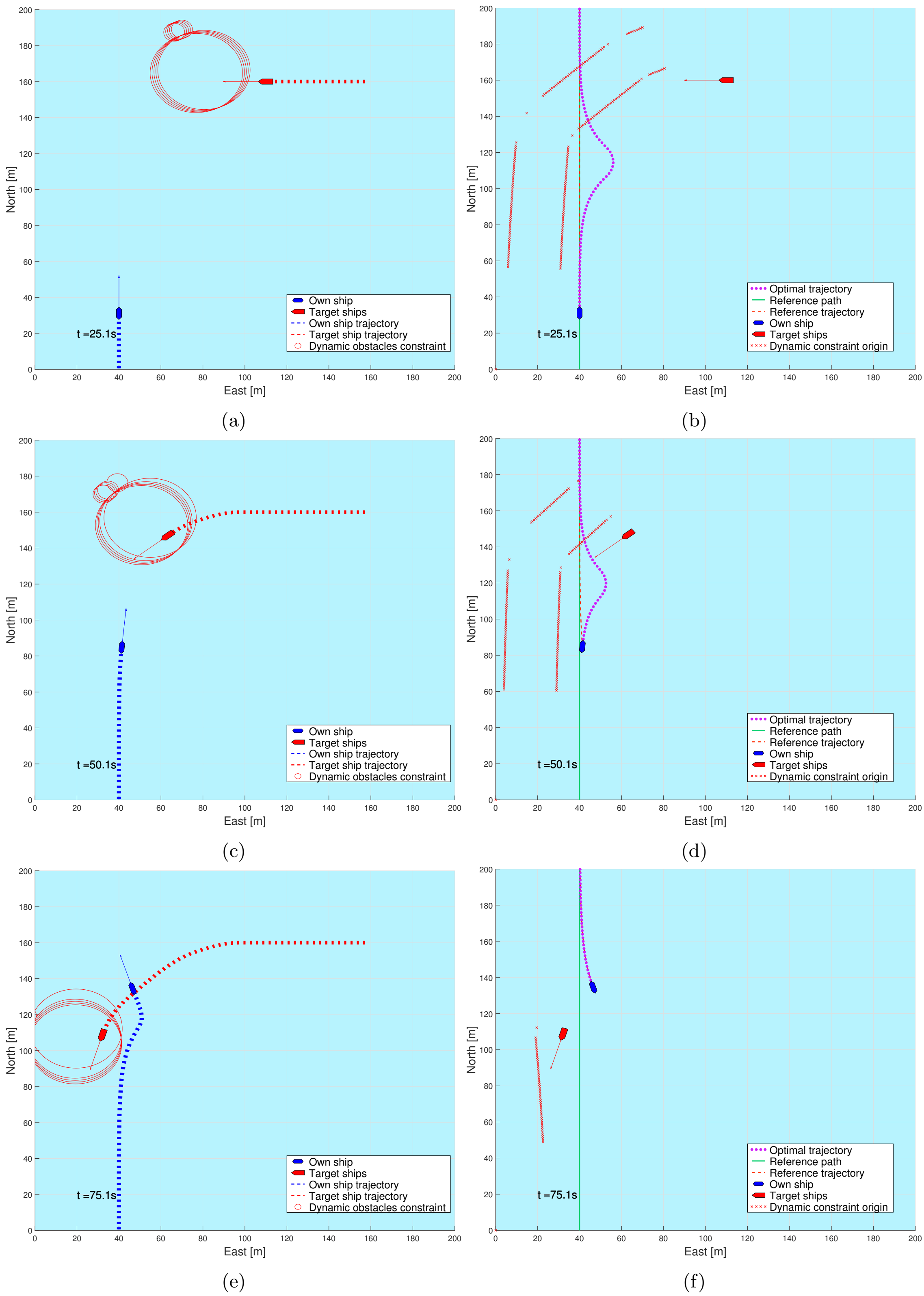


Figure 17: Head on situation with a turn, Result for this were the same regardless of prediction level.

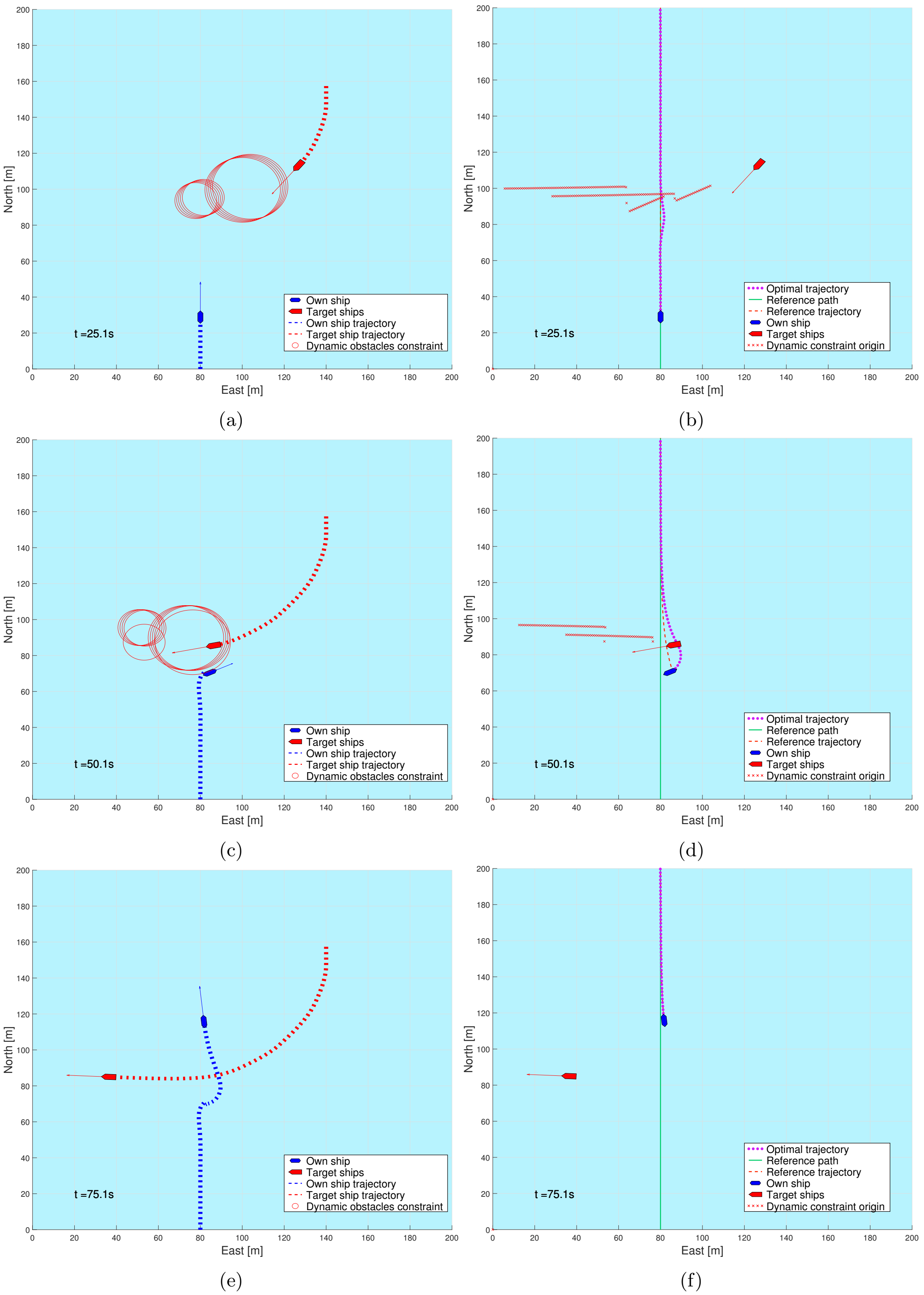


Figure 18: Give way with a turn, here with full prediction. Observe the OS not execting to have to yield until it's almost too late.

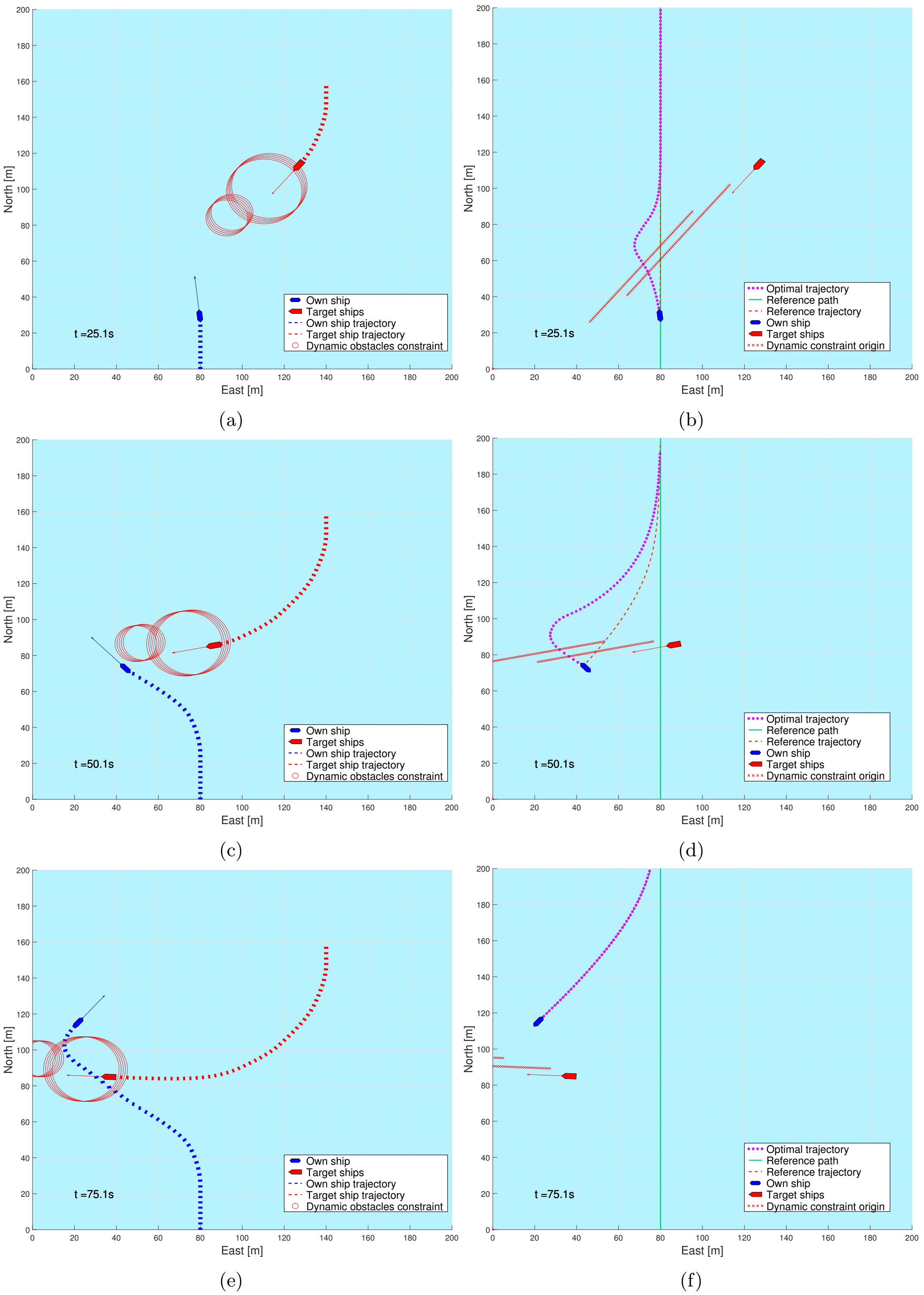
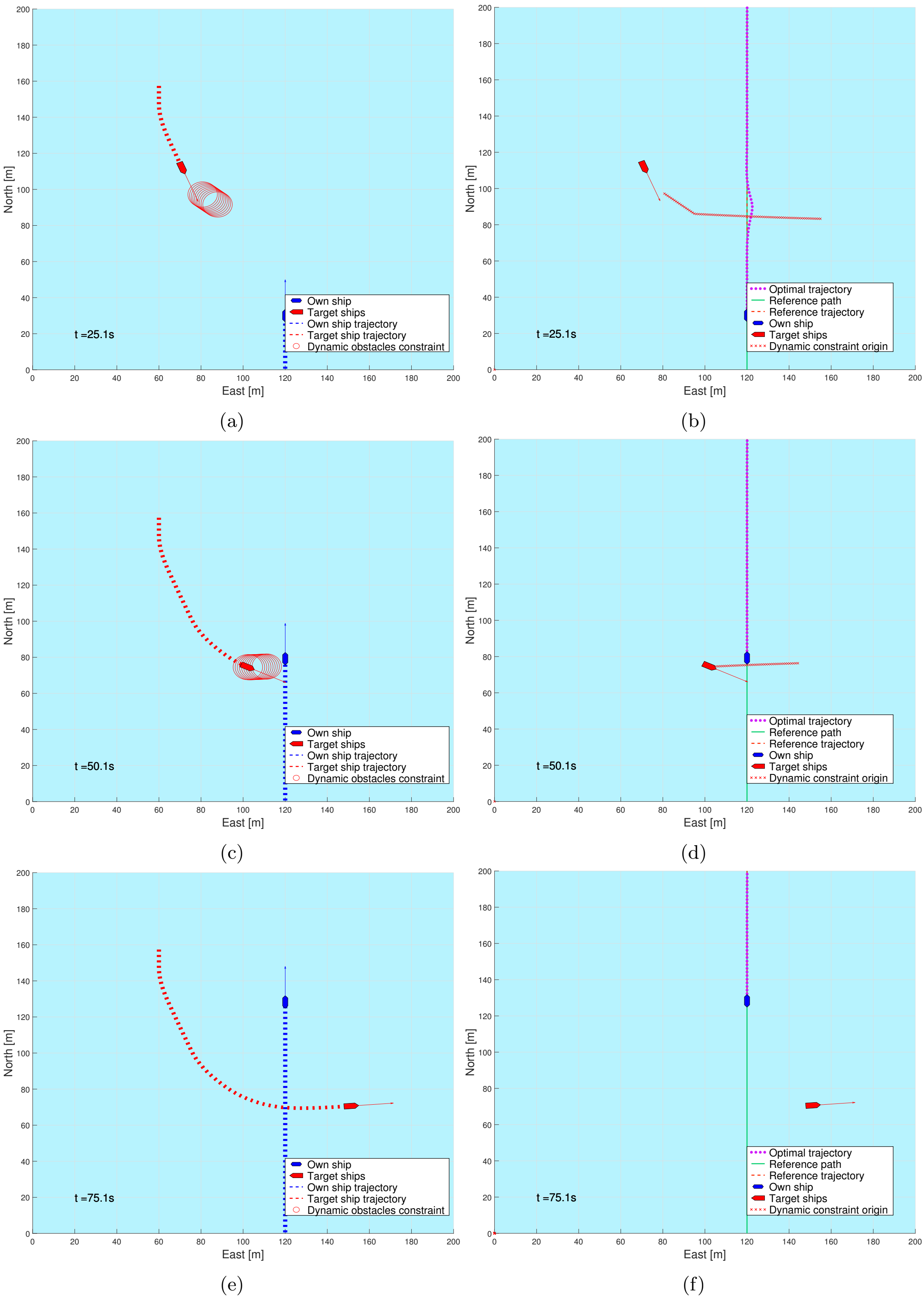


Figure 19: Give way with a turn, here with simple prediction. Observe as the OS gets dragged along by the constraints of the turning TS.



(a) (b) (c) (d) (e) (f)

Figure 20: Stand on situation with turn. Result independent of prediction level.

4.2.4 Turn Head On

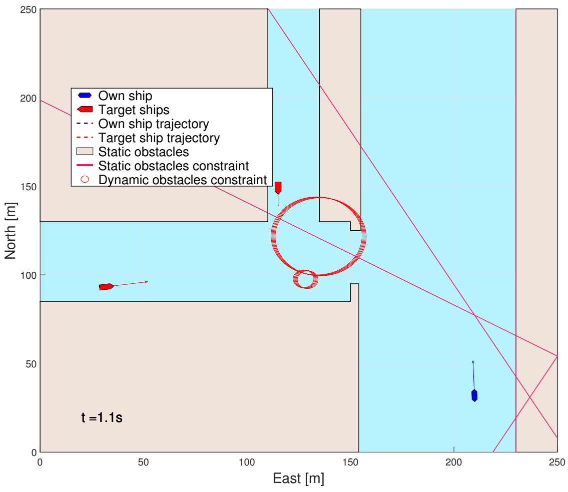
Very similar result to the straight path head on situation, which is not entirely unexpected. One aspect of this scenario that has become very obvious with hindsight is that the incoming TS should have approached from north west instead of north east. That way any a poorly calculated optimal trajectory would have been dragged towards the wrong side for the crossing. Instead as the situation is set up the trajectory will always be pushed towards crossing on the correct side. (TODO: kanskje forklar dette bedre?). The results for this scenario are seen in Figure 17

4.2.5 Turn Give Way

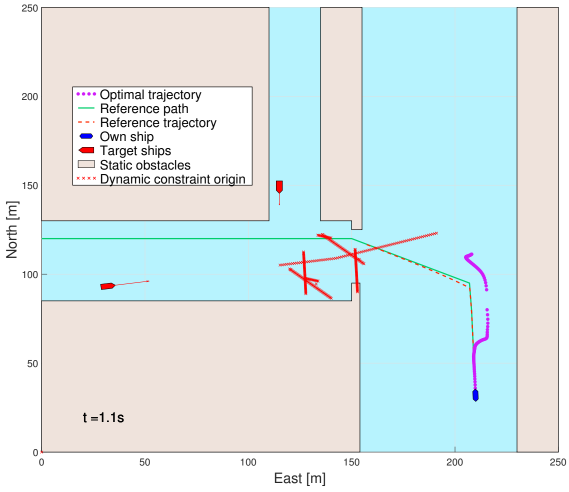
This scenario finally shows a huge difference in behaviour between the prediction levels. With full prediction the OS anticipates the incoming TS's intent to cross. Though the prediction is not perfect and the OS actually gets caught inside the constraints for one iteration. The result with full prediction is seen in Figure 18, and aside from getting caught and pushed out by the constraints the behaviour is pretty good. With simple prediction on the other hand, as seen in Figure 19, the optimal trajectory gets caught by the incoming crossing TS's constraints and dragged along some 100 meters off-course.

4.2.6 Turn Stand On

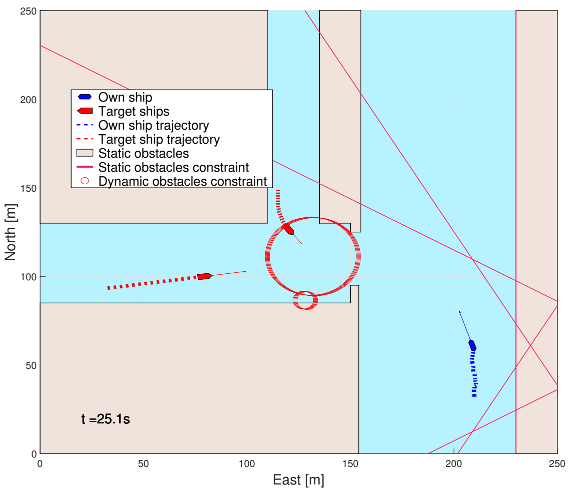
The results for this scenario was not affected by prediction level, in fact nothing at all happens in this scenario. Which is exactly the desired result, but it's not very interesting to read or write about. The results are seen in Figure 20.


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =1.1s
North [m]
East [m]

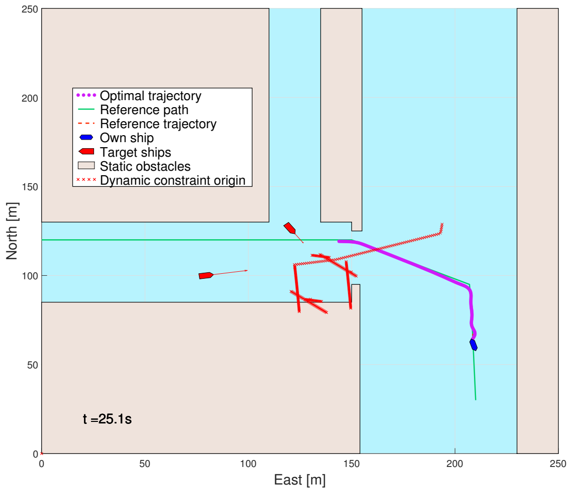
(a)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =1.1s
North [m]
East [m]

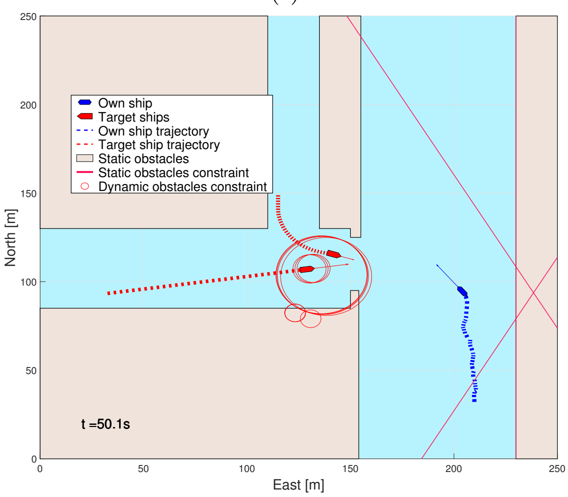
(b)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =25.1s
North [m]
East [m]

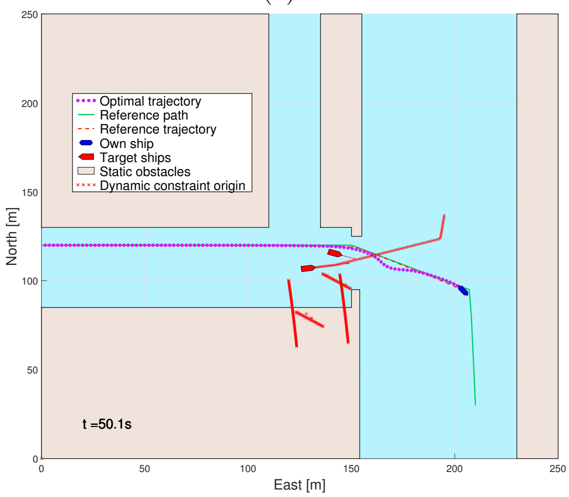
(c)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =25.1s
North [m]
East [m]

(d)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =50.1s
North [m]
East [m]

(e)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =50.1s
North [m]
East [m]

(f)

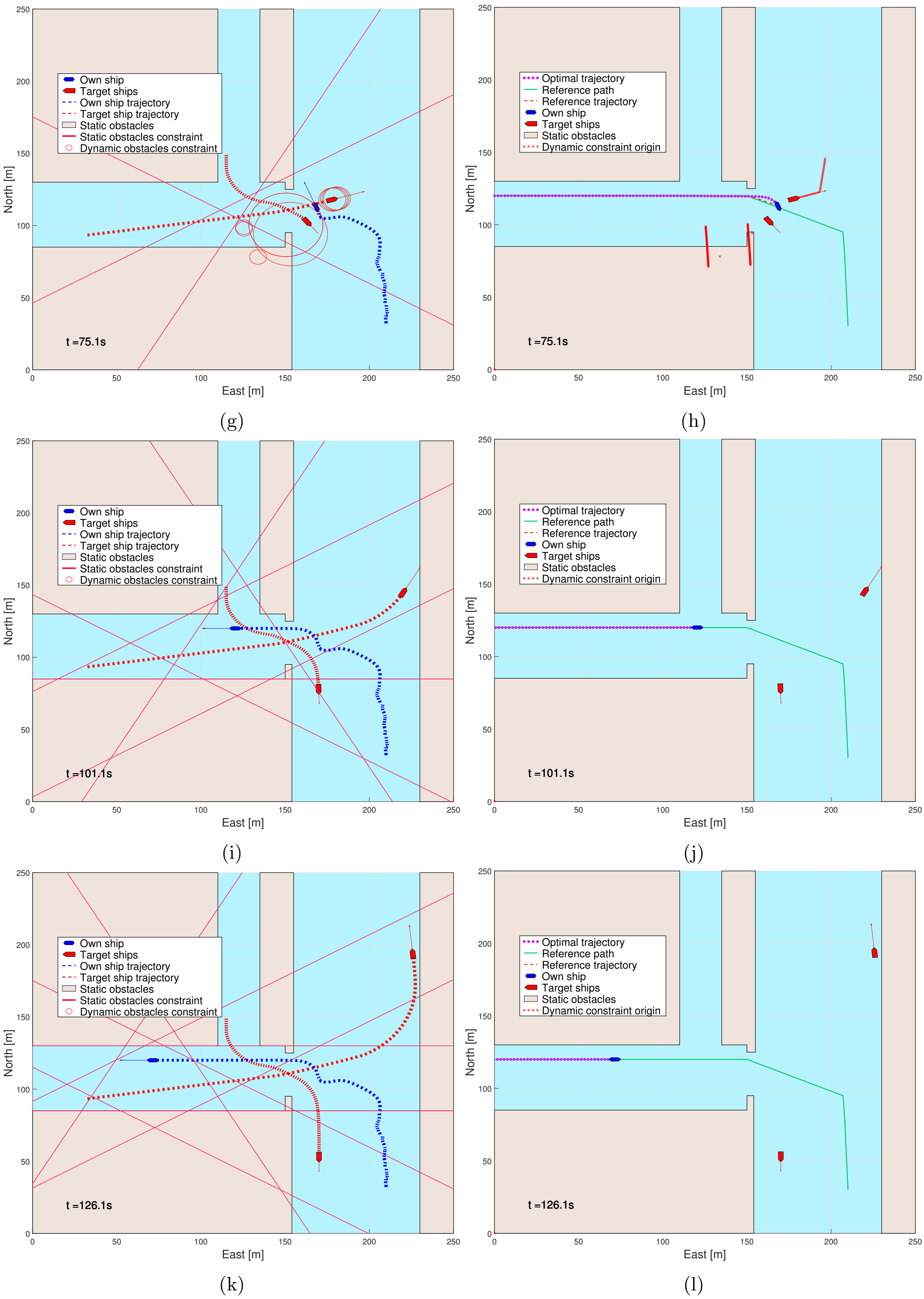
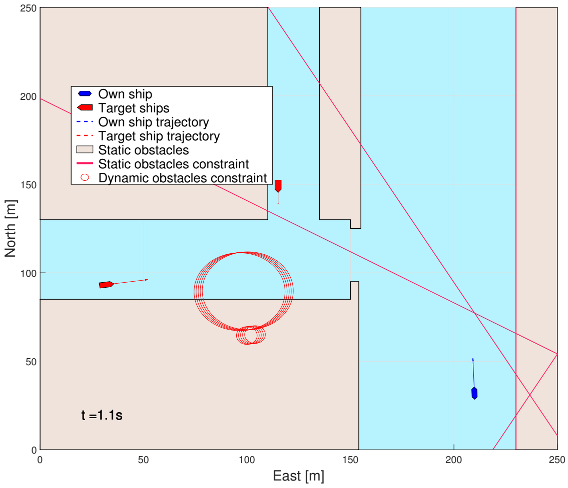
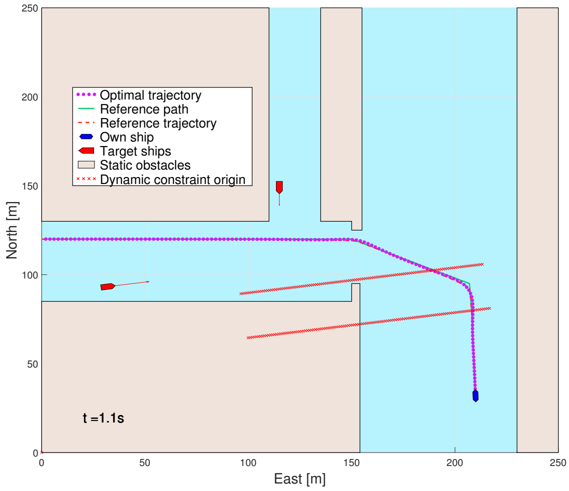


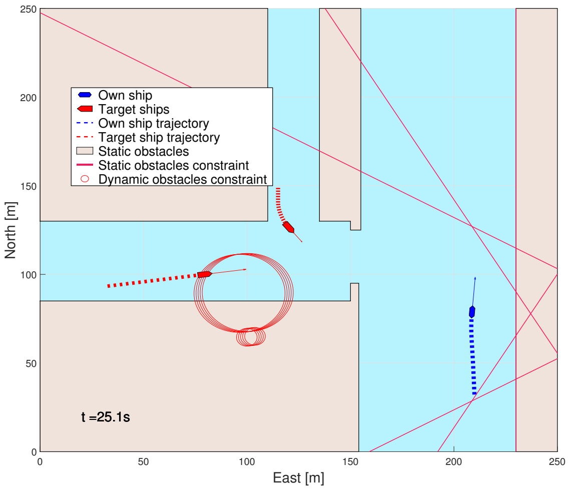
Figure 21: Canals situation. Here shown with full prediction.


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =1.1s
North [m]
East [m]

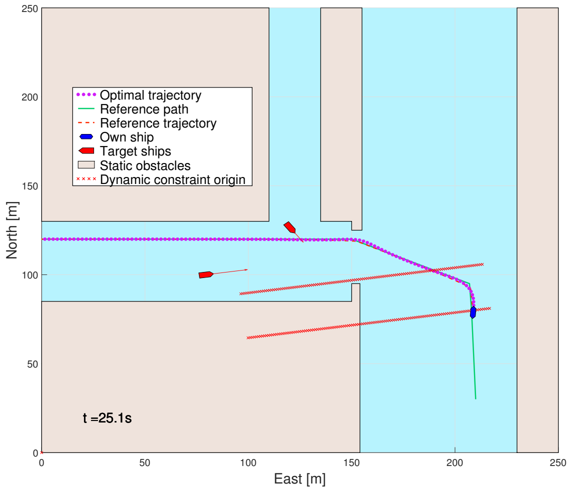
(a)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =1.1s
North [m]
East [m]

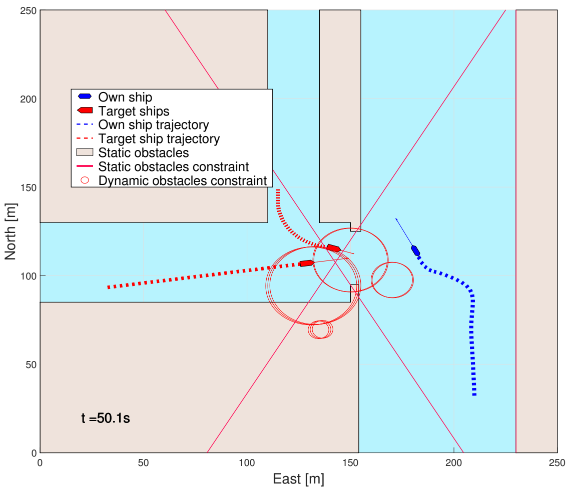
(b)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =25.1s
North [m]
East [m]

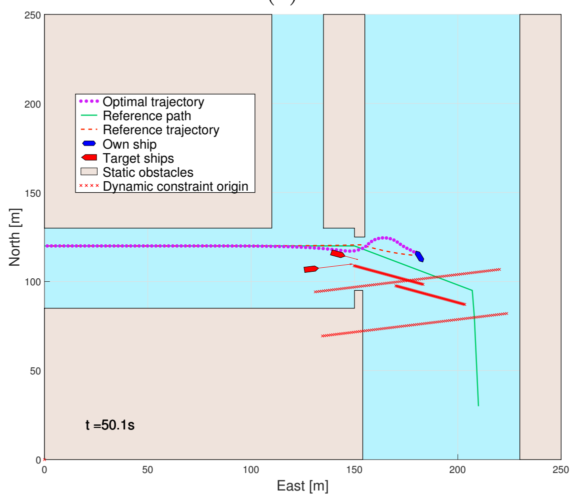
(c)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =25.1s
North [m]
East [m]

(d)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =50.1s
North [m]
East [m]

(e)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =50.1s
North [m]
East [m]

(f)

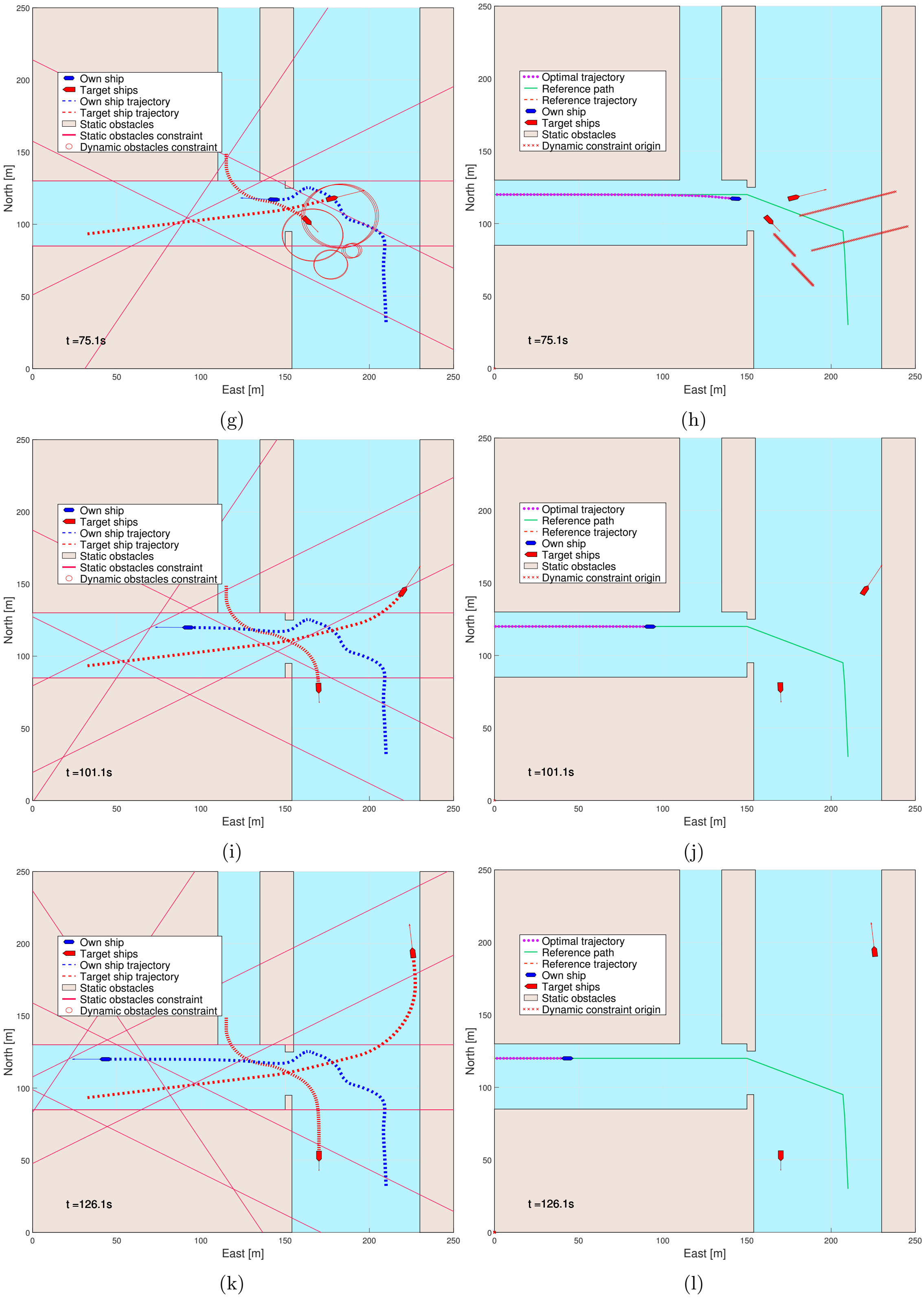
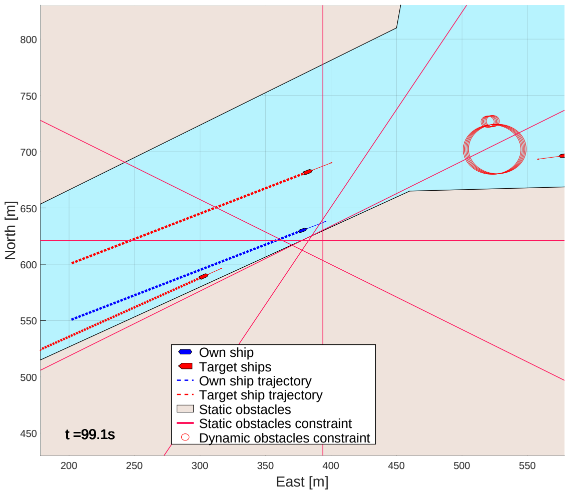
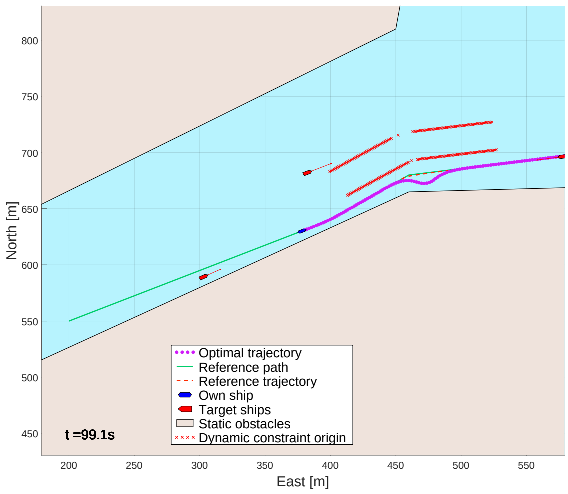


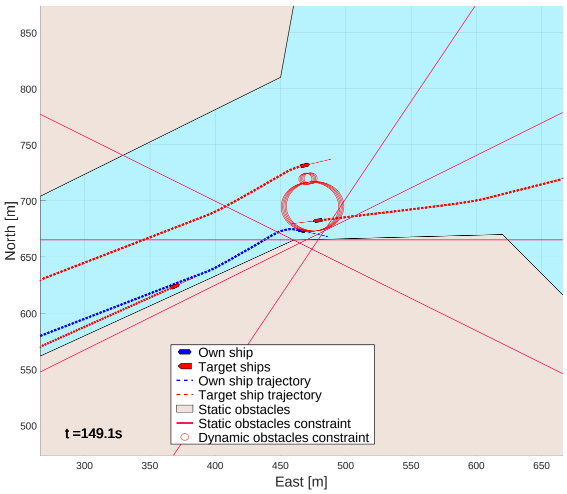
Figure 22: Canals situation. Here shown with simple prediction.


North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =99.1s

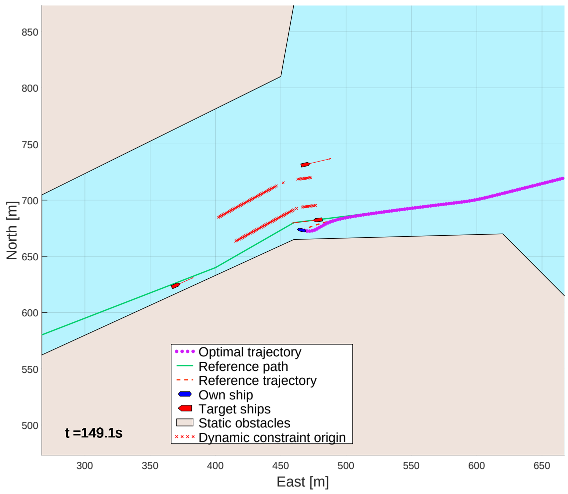
(a)


North [m]
East [m]
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =99.1s

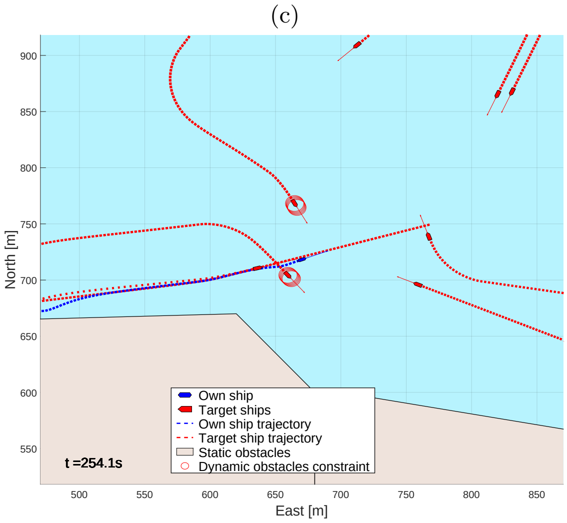
(b)


North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =149.1s

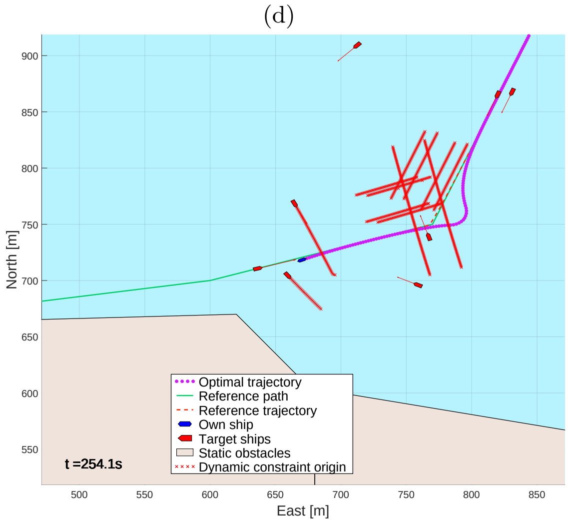
(c)


North [m]
East [m]
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =149.1s

(d)


North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Dynamic obstacles constraint
t =254.1s

(e)


North [m]
East [m]
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =254.1s

(f)

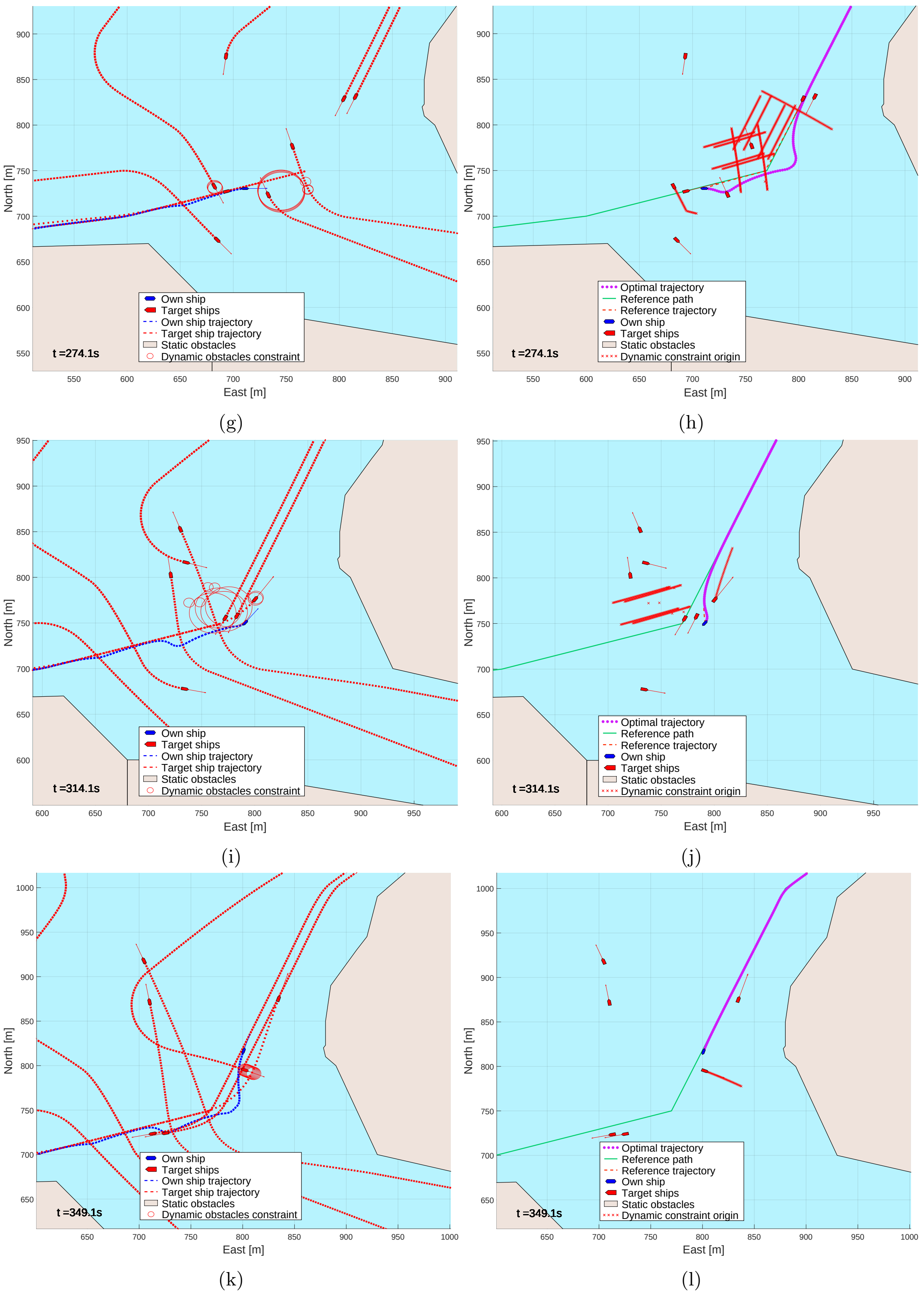
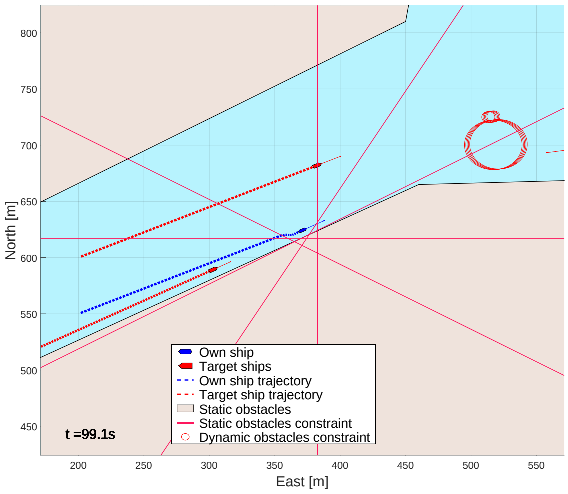
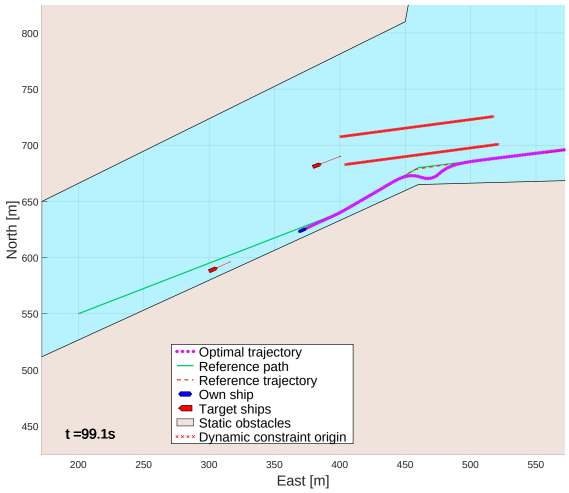


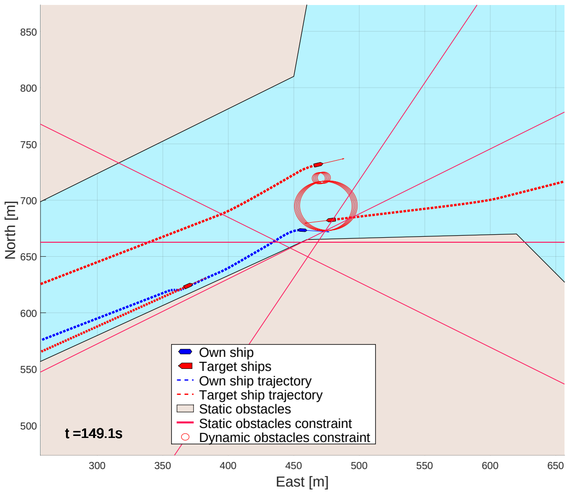
Figure 23: Fjord situation. Here shown with full prediction. Observe the OS handles the stress test pretty well.


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =99.1s
North [m]
East [m]

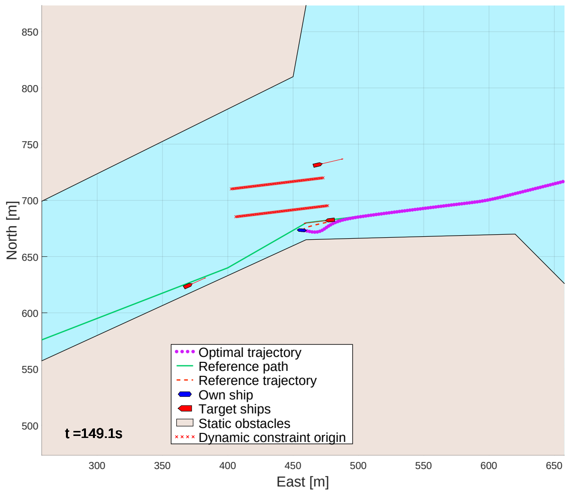
(a)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =99.1s
North [m]
East [m]

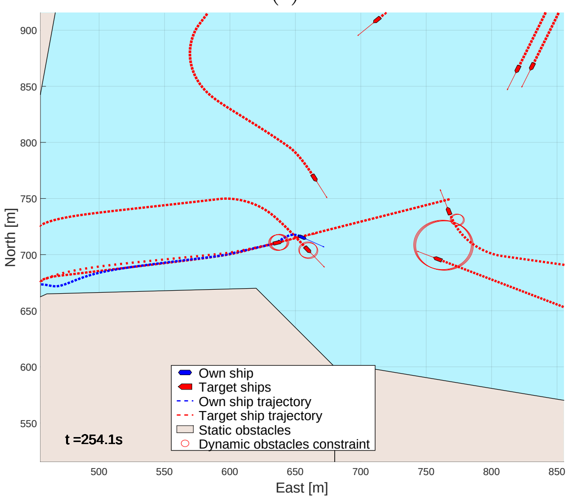
(b)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =149.1s
North [m]
East [m]

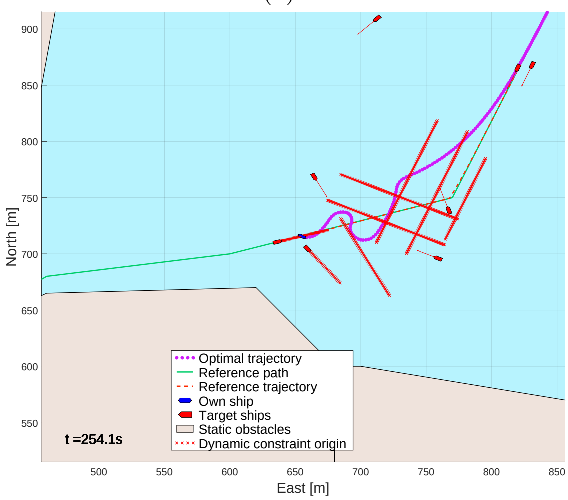
(c)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =149.1s
North [m]
East [m]

(d)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Dynamic obstacles constraint
t =254.1s
North [m]
East [m]

(e)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =254.1s
North [m]
East [m]

(f)

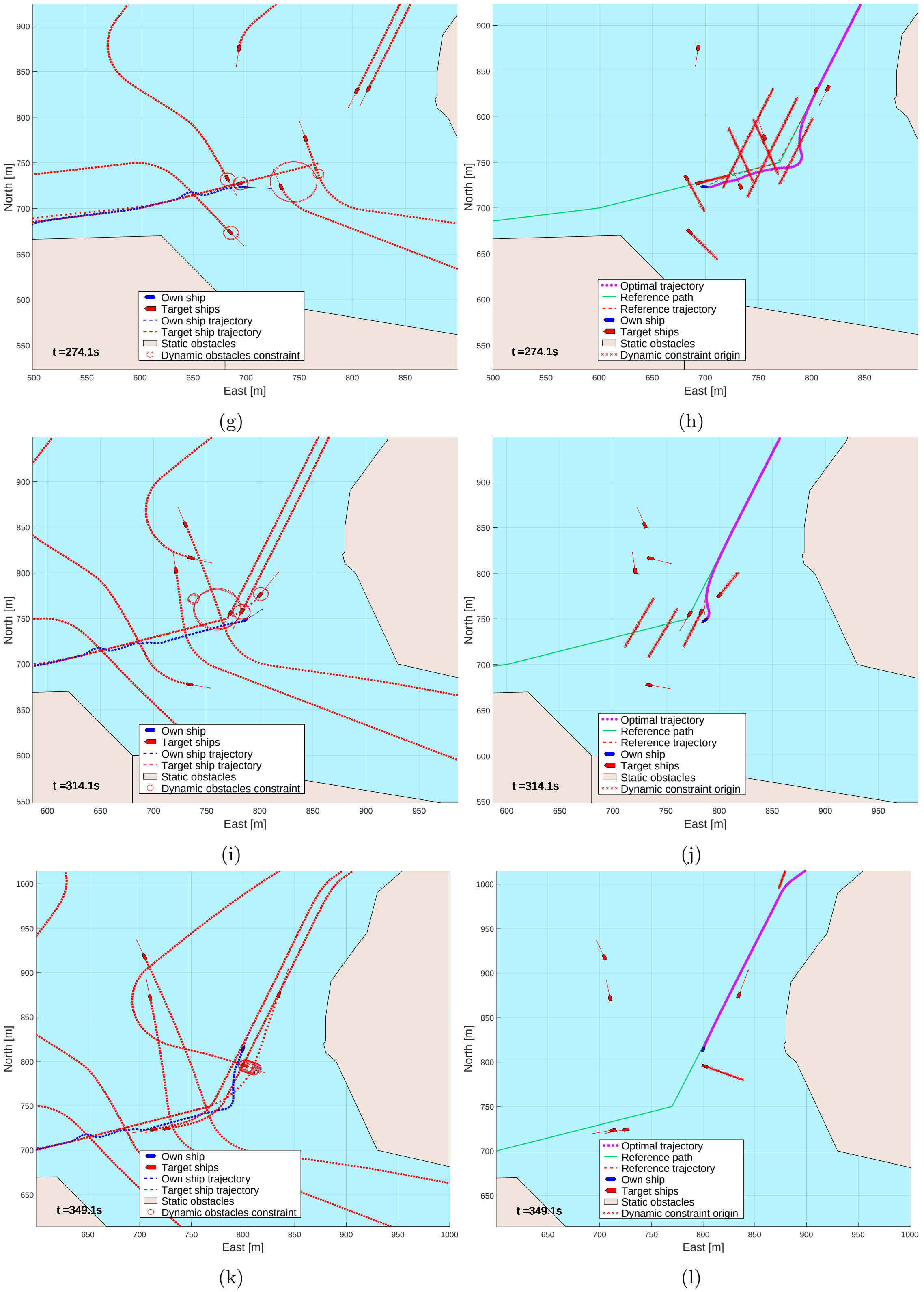


Figure 24: Fjord situation. Here shown with simple prediction. Observe the OS behaves much more erratically compared to the full prediction level.

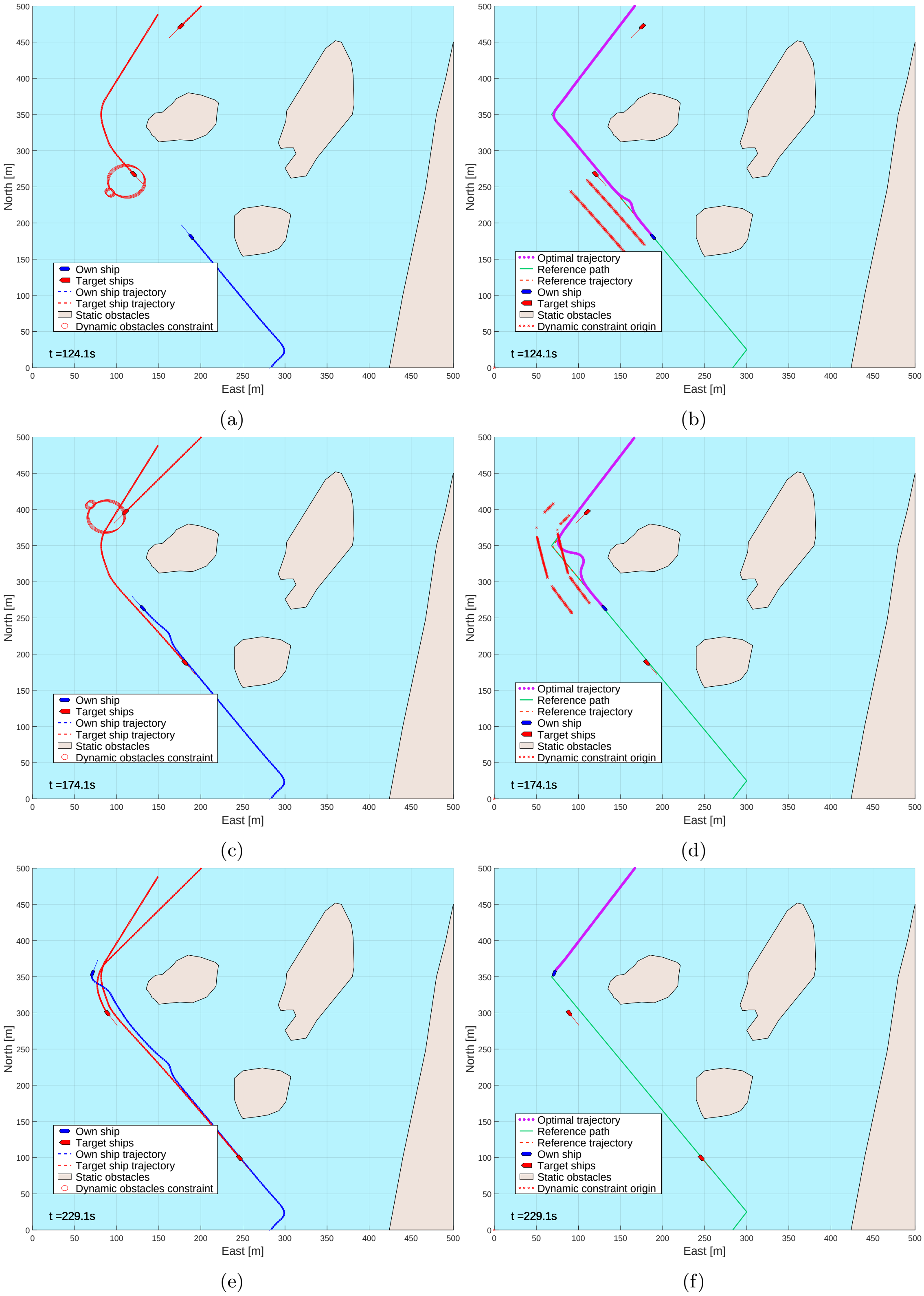
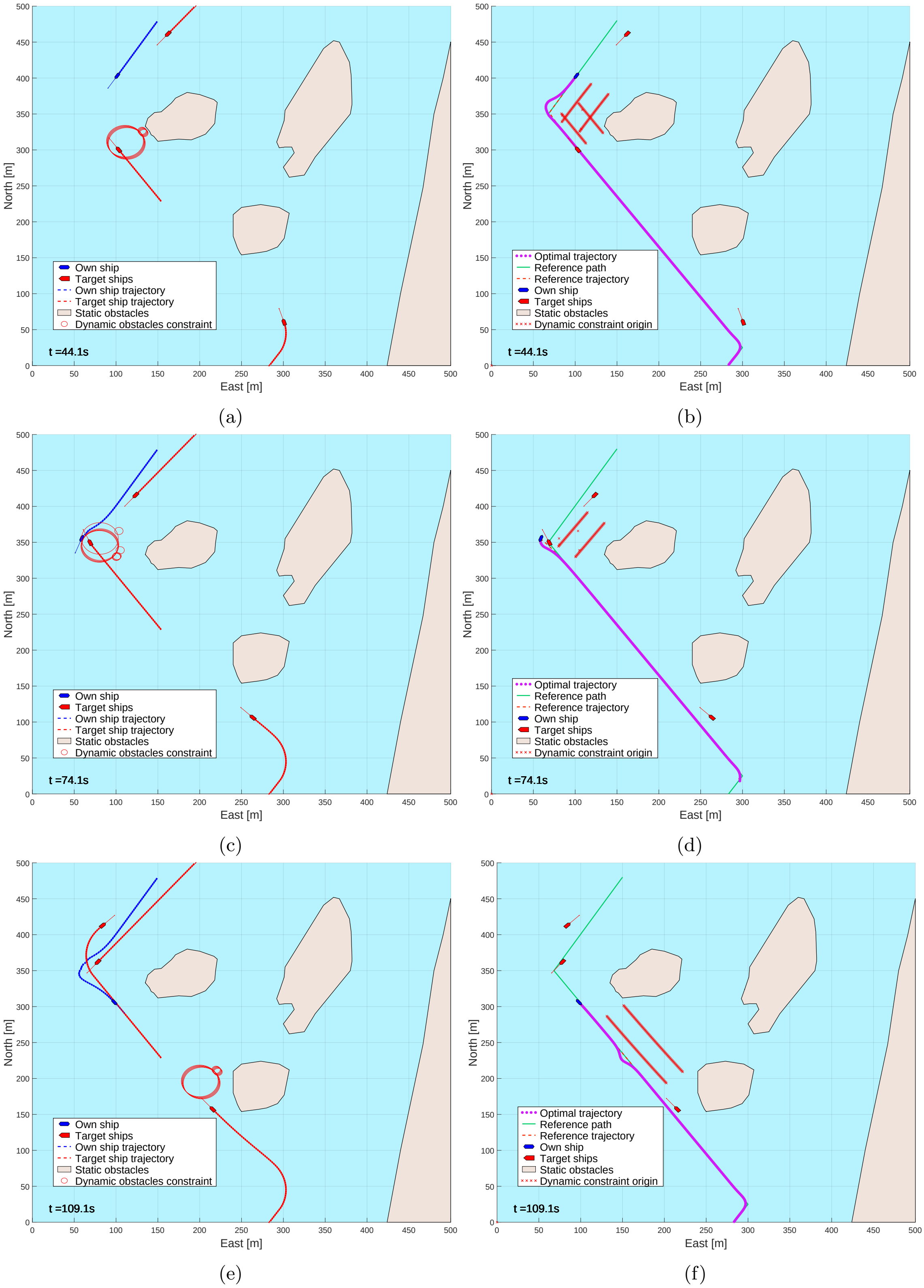


Figure 25: Helloya Situation. Here, OS behaves to expectations independently of prediction level.



t =44.1s
t =74.1s
t =109.1s
North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Dynamic obstacles constraint
Optimal trajectory
Reference path
Reference trajectory
Dynamic constraint origin
(a)
(b)
(c)
(d)
(e)
(f)

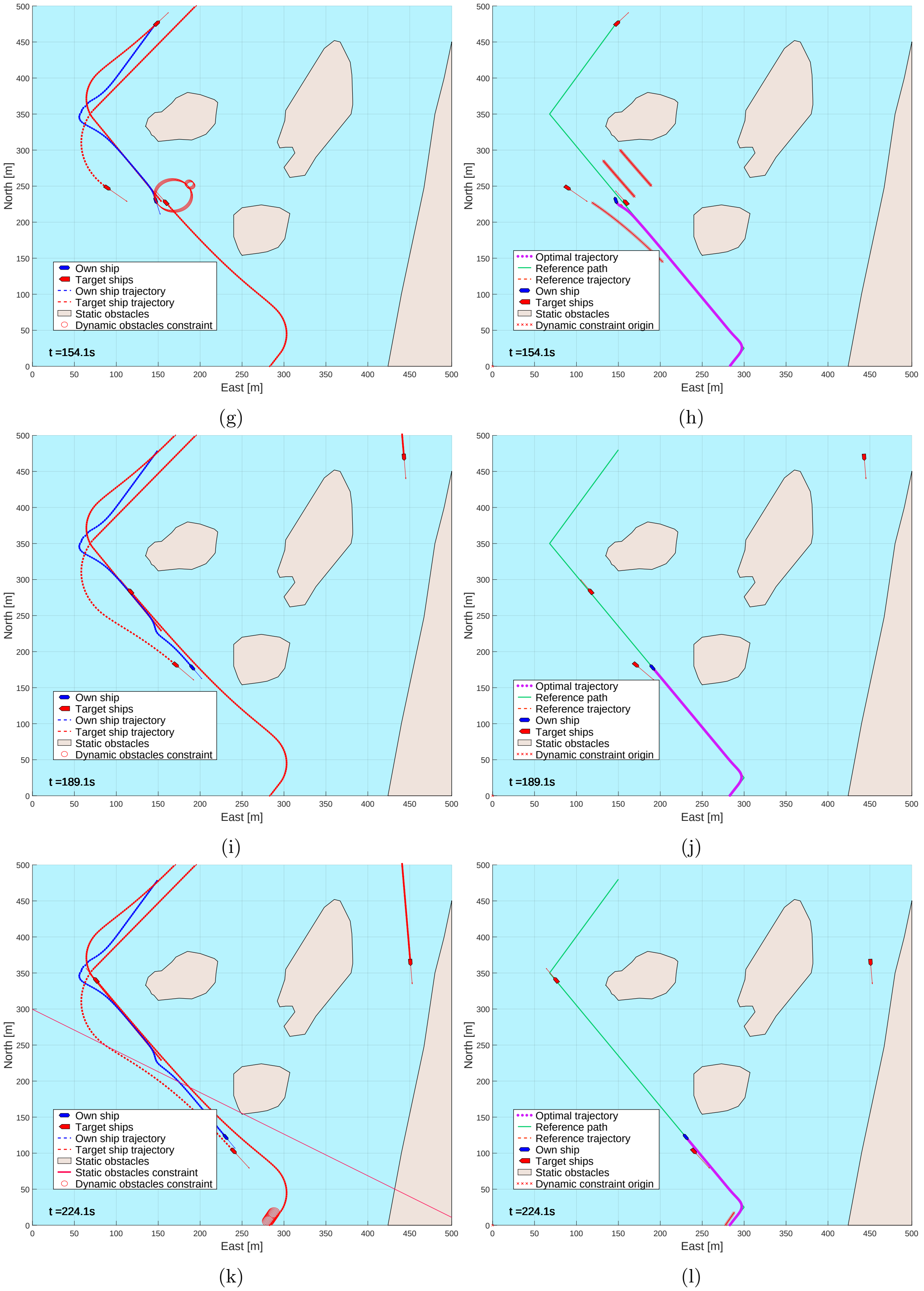
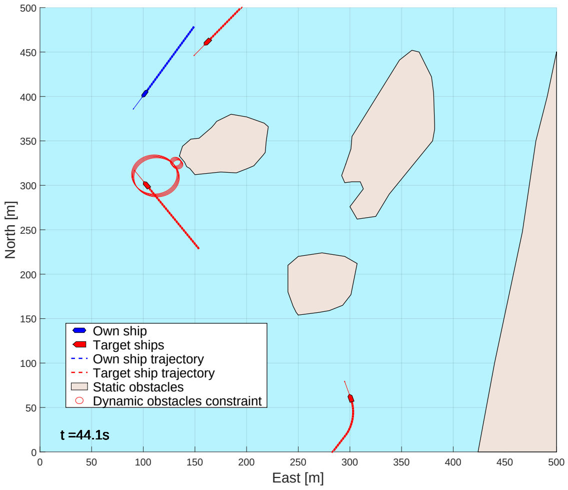
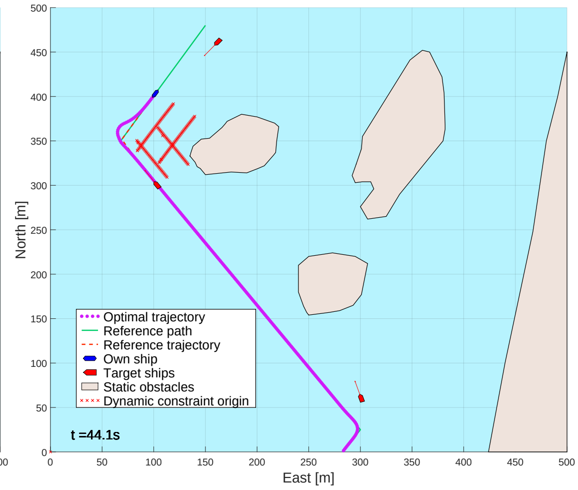


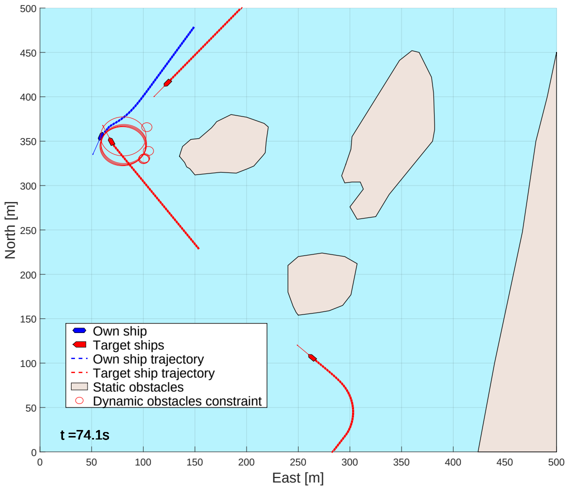
Figure 26: Helloya situation in reverse. Here with full prediction, OS behaves to expectations.


North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Dynamic obstacles constraint
t =44.1s

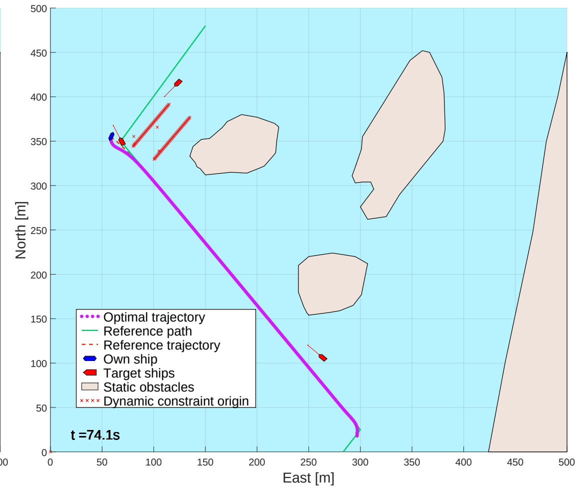
(a)


North [m]
East [m]
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =44.1s

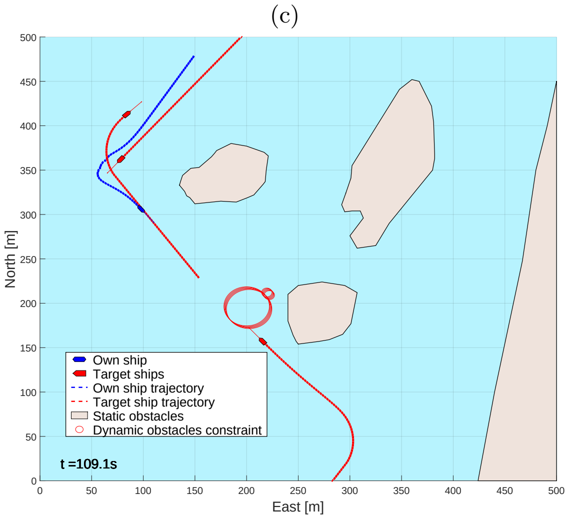
(b)


North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Dynamic obstacles constraint
t =74.1s

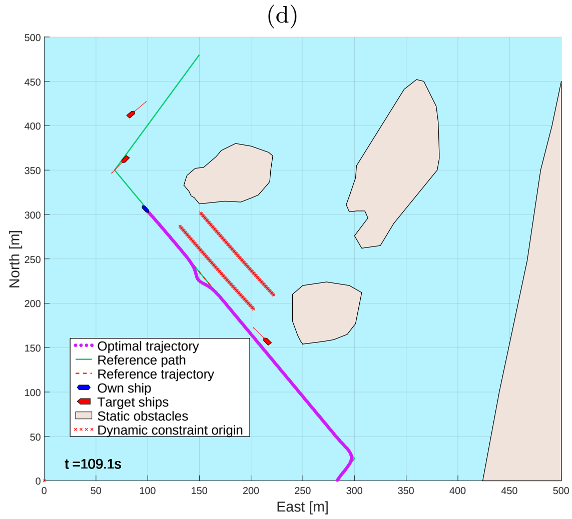
(c)


North [m]
East [m]
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =74.1s

(d)


North [m]
East [m]
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Dynamic obstacles constraint
t =109.1s

(e)


North [m]
East [m]
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =109.1s

(f)

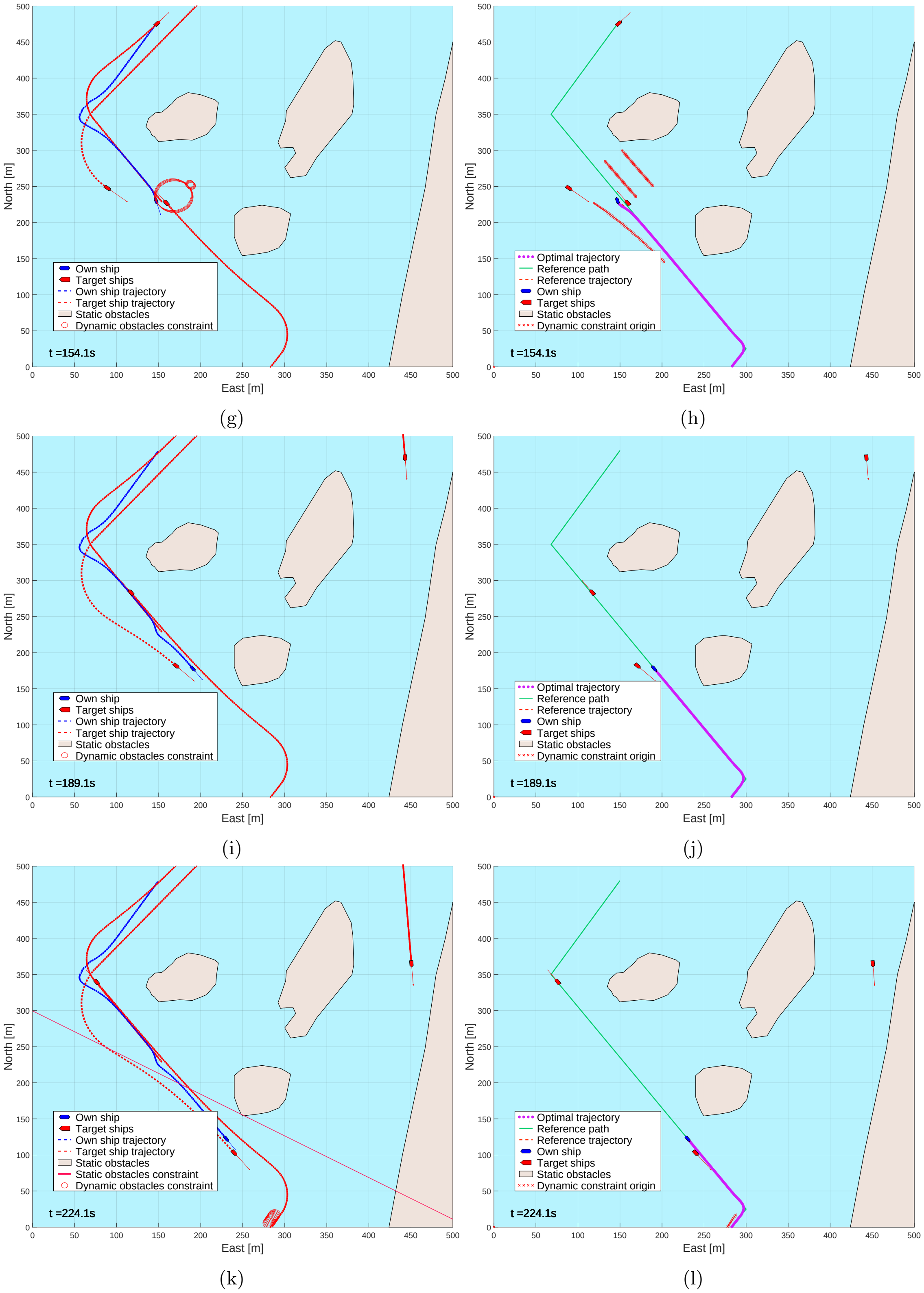


Figure 27: Helloya situation in reverse. Here with simple prediction, OS behaves slightly erratically

4.2.7 Canals

Results are seen with full prediction in Figure 21, and with simple prediction in Figure 22. This was the first scenario designed to be a bit more complicated than just an open ocean encounter. In this scenario there are walls blocking the available space, as well as a bottleneck that gets blocked by the constraints of the incoming TSs. The TS approaching from the north also turns in a way that would be obvious to a navigator, but not easily understood by a simple prediction algorithm.

The immediate difference between the prediction levels is that full prediction forsees the bottleneck closing and slams the breaks. Leading to a jittery trajectory as the algorithm goes through the following process:
1.) First the algorithm is able to find a feasible and optimal solution when obstacles are disabled.
2.) Then the obstacles are enabled which break the continuity of the previous optimal trajectory, making the newest optimal trajectory infeasible.
3.) The speed is then reduced and the algorithm is able to find an optimal solution because the bottleneck is not blocked for long.
4.) Because the previous trajectory was feasible the speed is set back to the nominal value, the bottleneck is once again blocked leading to an infeasible path.
Repeating step 3 and 4 until an opening between all the constraints finally shows itself. Every time the NLP is infeasible the result used in the MPC might not have consistent heading or speed with the previous control interval, which is why the result looks so jittery.

The simple prediction version on the other hand, presuming that one of the northernmost TS will simply phase through the wall, proceeds withtout a care. As the OS gets closer to the bottleneck, so too do the constraints about to block the way. This is why the optimal trajecotry bulges upwards, luckily the gap is not closed for long, and the OS is able pass without having to go through the same song and dance as the full prediction version.

4.2.8 Trondheimsfjord

This scenario was designed as a COLREGs assessment stress test. The result for full prediction is seen in Figure 23, and with simple prediction in Figure 24. This scenario is full of uncooperative TSs, and the simple prediction level algorithm is unable to cope. With all the overlapping crossing TS turning onto the path of the OS, as well as a TS overtaking from behind the algorithm is not able to find a consistent trajectory, jumping between different optimal solutions over the course of the simulation. The full prediction results on the other hand are pretty good with the OS being able to make it through the crucible with very few adjustments to the course, passing the oncoming TS pack on the correct side as well.

While runtime optimization isn't the focus of this thesis, it should be noted that the full prediction level simulation was significantly faster to run, mostly due to the NLP being solved much faster when the constraints don't move around as much.

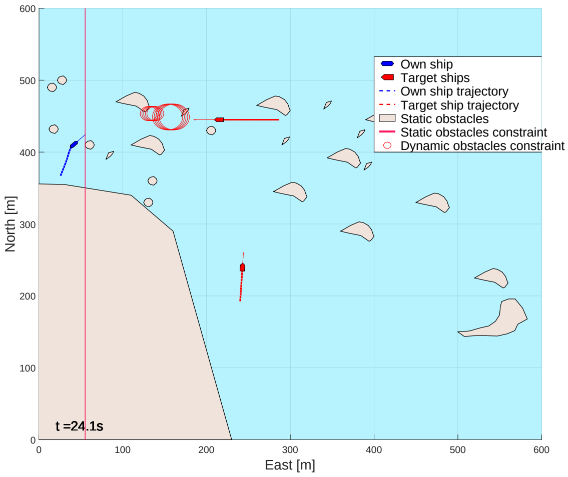
4.2.9 Helløya

The results for this scenario are seen in Figure 25 and are similar enough for both prediction levels that only full prediction is shown. Considering the simple scenarios it should be no surprise that the OS is able to pass by the first TS on the correct side. The inbisible turn is also handled really well by both prediction levels. The boring results here lead to the idea that it's probably the direction of the turn that makes it so that there is no difference between the prediction levels, so a reverse of the scenario was created.

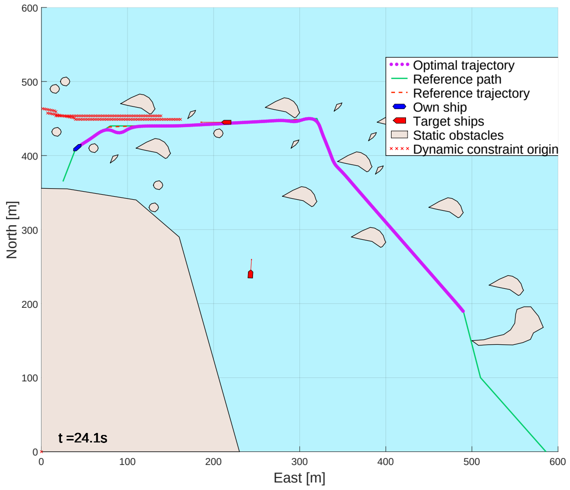
4.2.10 Helløya Reversed

The only scenario that features the OS heading southwards, which lead to a very interesting discovery that has since been patched out: The algorithm could for the longest time not handle turning from some angles to another, the heading reference would pick the wrong turn direction, and so the resulting trajectory would take a loop. This was discussed in Chapter 3.3 and will be discussed a bit more in the miscellaneous results. Back to Helloya in reverse, the results for full prediction are in Figure 26, and simple in Figure 27. This time the 'invisible' turn is handled differently depending on prediction level, with the simple prediction trajectory being pushed towards cutting the corner, while the full prediction version ends up crossing in front. Later the head on TS is easily avoided by both prediction levels.

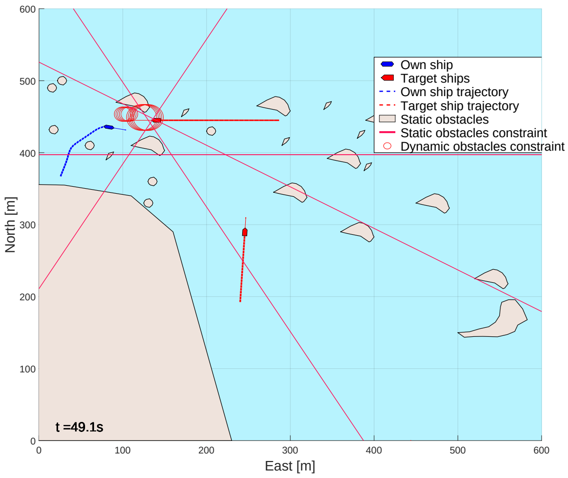
Not captured in these figures, but observable in the video version, is the overtaking going very poorly for the simple prediction level. This is likely due to the fact that the constraints for the overtaking TS ends up being placed on top of the initial guess, making the solver scramble to find a new solution.


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =24.1s
North [m]
East [m]

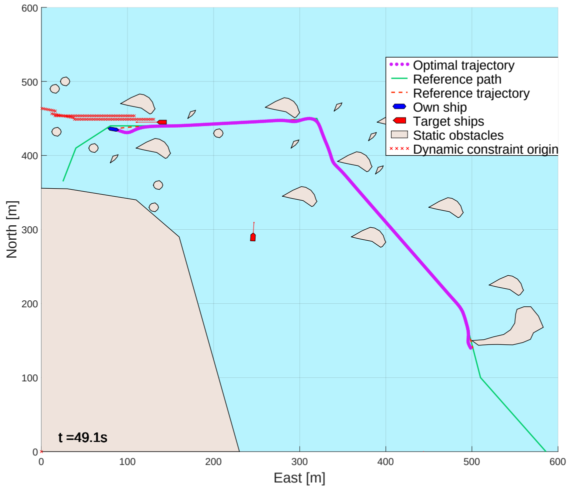
(a)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =24.1s
North [m]
East [m]

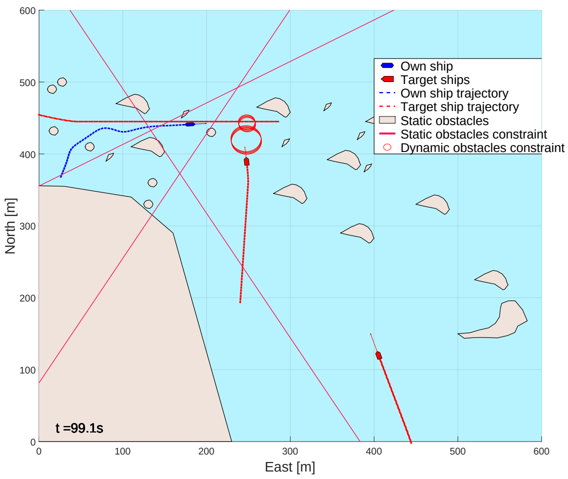
(b)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =49.1s
North [m]
East [m]

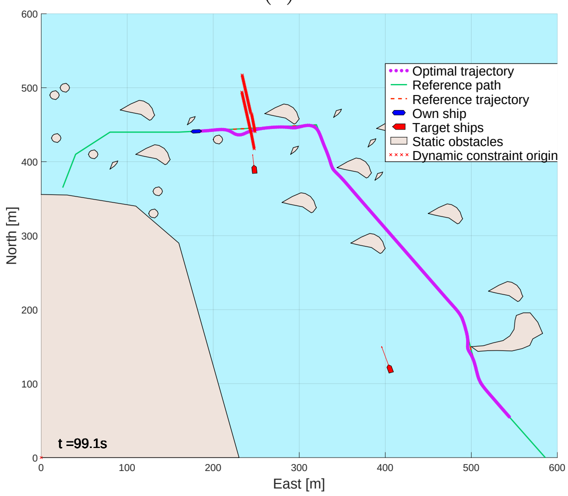
(c)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =49.1s
North [m]
East [m]

(d)


Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =99.1s
North [m]
East [m]

(e)


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =99.1s
North [m]
East [m]

(f)

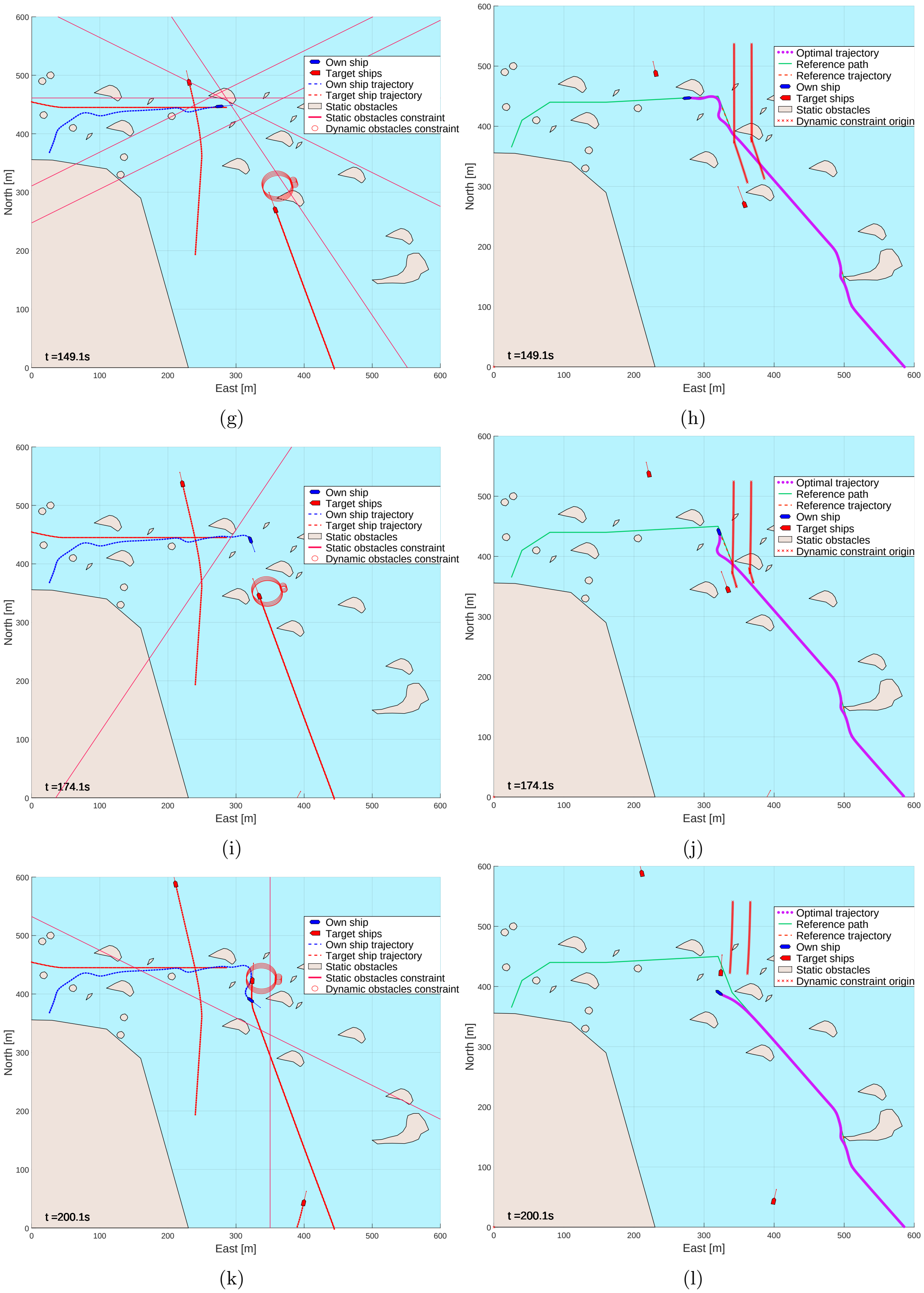
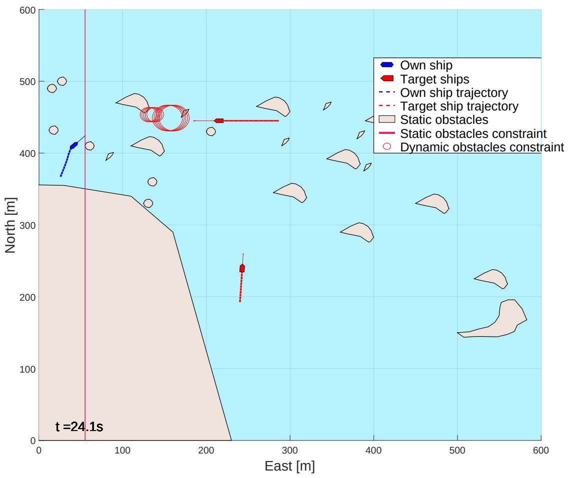
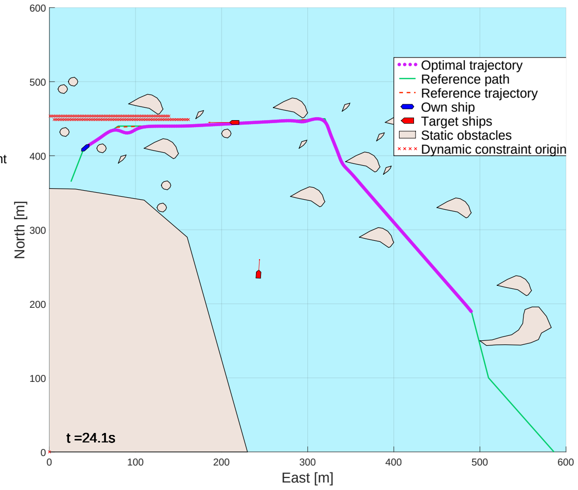


Figure 28: Skjærgård with traffic situation. Here with full prediction.



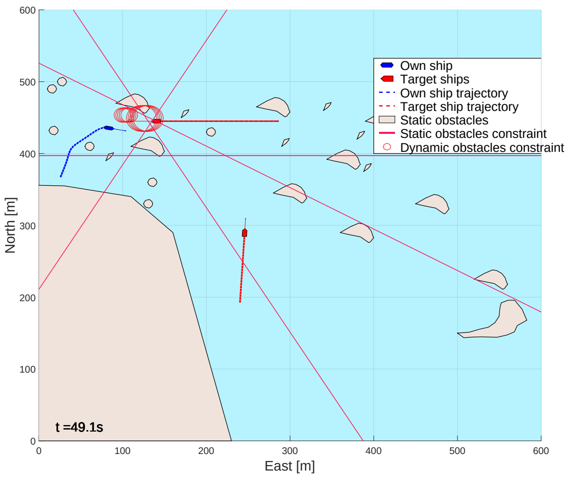
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =24.1s
North [m]
East [m]

(a)



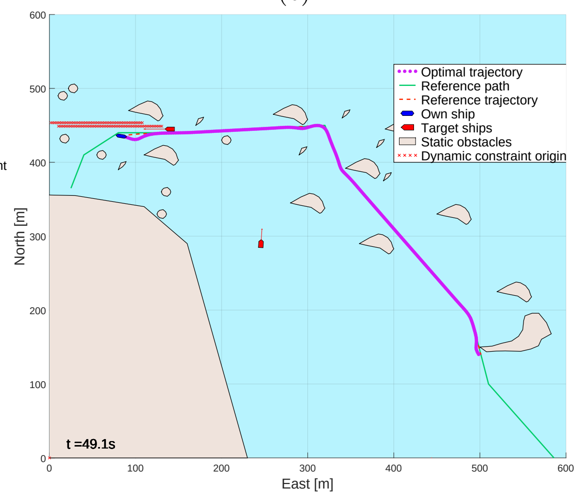
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =24.1s
North [m]
East [m]

(b)



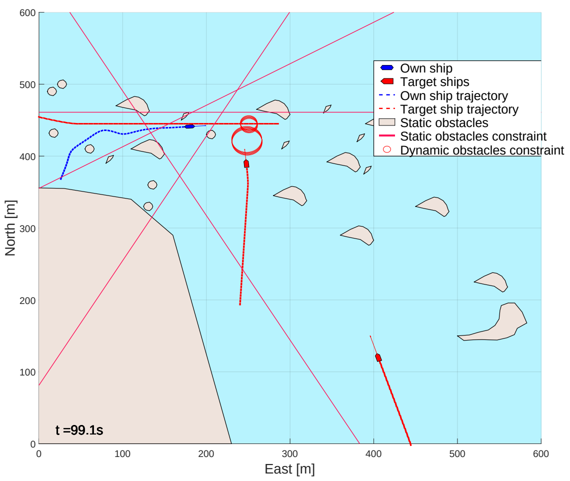
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =49.1s
North [m]
East [m]

(c)



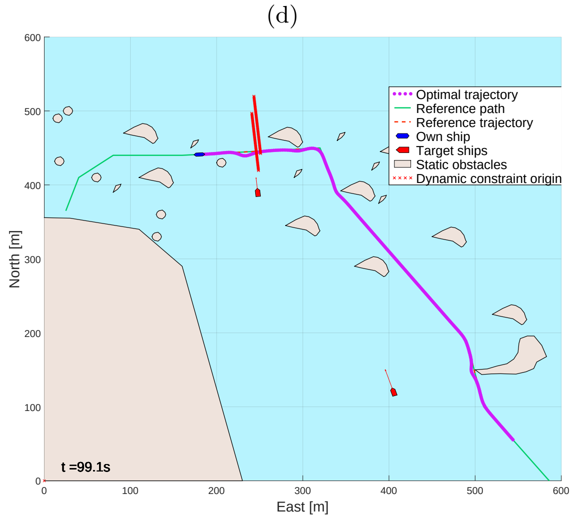
Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =49.1s
North [m]
East [m]

(d)



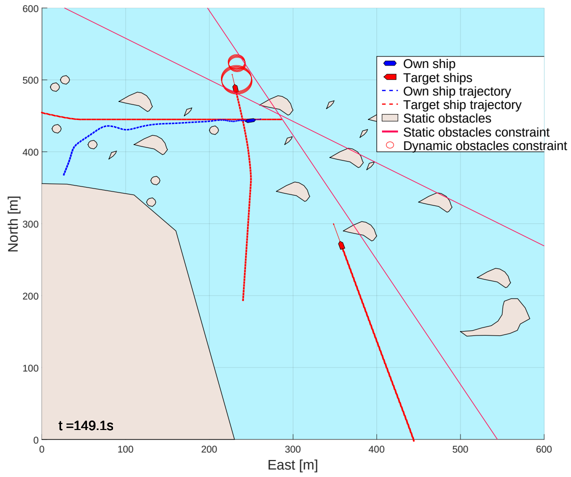
Own ship
Target ships
Own ship trajectory
Target ship trajectory
Static obstacles
Static obstacles constraint
Dynamic obstacles constraint
t =99.1s
North [m]
East [m]

(e)

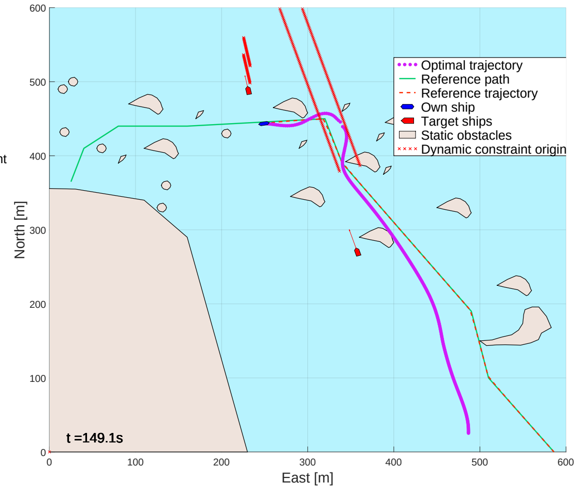


Optimal trajectory
Reference path
Reference trajectory
Own ship
Target ships
Static obstacles
Dynamic constraint origin
t =99.1s
North [m]
East [m]

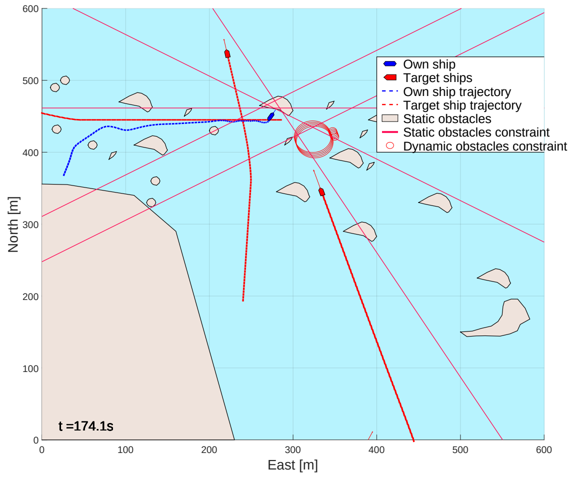
(f)



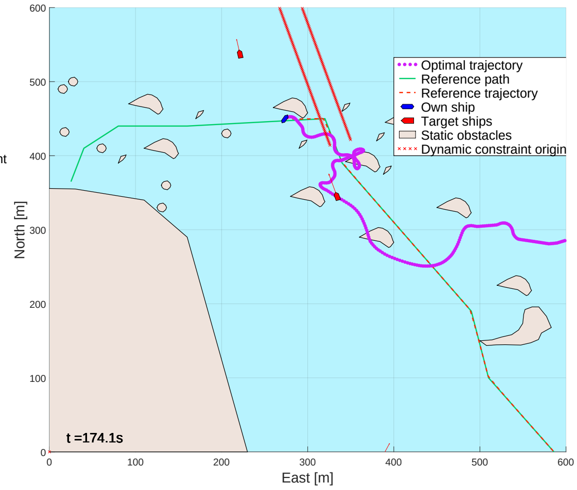
(g)



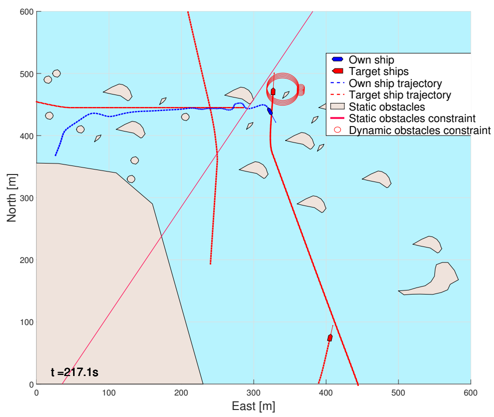
(h)



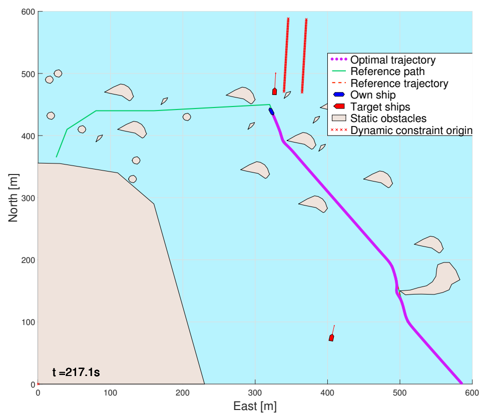
(i)



(j)

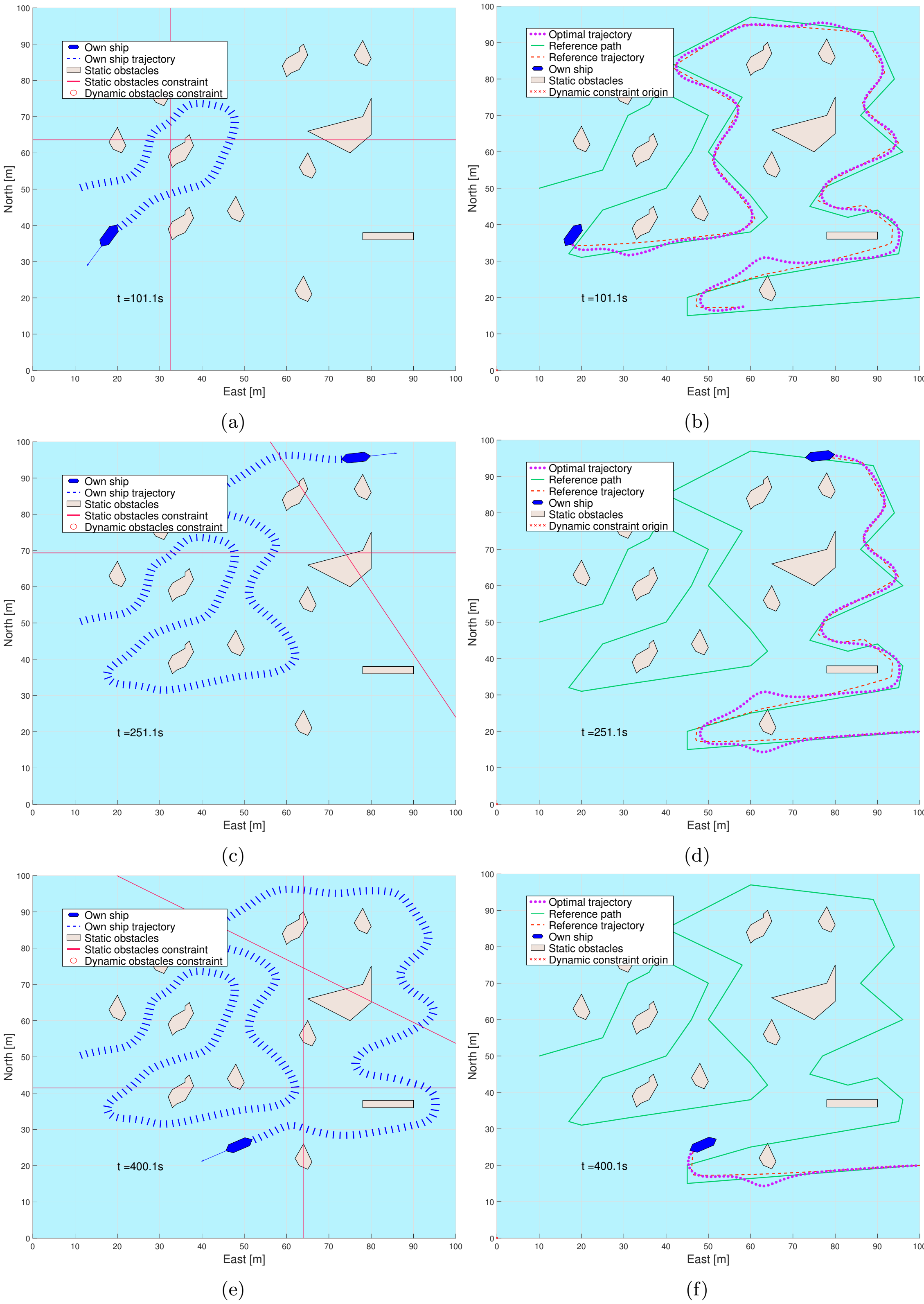


(k)



(l)

Figure 29: Skjærgård with traffic situation. Here with simple prediction.



(a) (b) (c) (d) (e) (f)

Figure 30: Skjærgård without traffic simulation. Result independent of prediction level due to no TSs.

4.2.11 Skjærgård with Traffic

This was the final scenario created and is one of the more complex ones. This scenario combines sporadic static obstacles with different COLREGs situations, the third TS in particular ended up being a lot of trouble for the simple prediction level algorithm. The results for full prediction are seen in Figure 29, and the simple in Figure **??**. This scenario was very easy with the full prediction level, by anticipating the incoming TS's turn the path remains unblocked by constraints and the crossing happens without a hitch. With a simple prediction level on the other hand, the algorithm has an absolute nightmare trying to find a way through, getting stuck in a loop of infeasible results that aren't easily captured in a single frame but very obvious in the video results. When the incoming TS finally starts to turn the algorithm is able to find an opening and get past.

4.2.12 skjærgård without Traffic

This is a simple path following scenario, result seen in Figure 30. This scenario was designed to stress test the static obstacles implementation, and I dare say it passes with flying colors. The reference path in this scenario is intentionally placed too close to some of the obstacles so that the effect of the constraints can be observed. Here, we see that the very last static obstacle is placed slightly on top of the reference path, but the trajectory planner has no problem going around.

4.2.13 Miscellaneous

Over the course of this thesis, these simulations have been ran countless times. Every now and then a quirk is spotted but it's then quickly patched out or fixed by the afformentioned cosmic radiation. However I had the foresight to save some of the more interesting ones, which will now be discussed a bit on their own before moving on to the general discussion.

Bad prediction:
This is a very important result to highlight, but not one that is shown in any of the scenarios presented. If the prediction is wrong about where the TS is going, it's can be really bad for the algorithm. The same of course goes for malicious actors or TSs who are non COLREGs complaint. All three can lead to the OS getting caught inside active dynamic constrants, which the IPOPT solver absolutely can not handle, the results are seen in Figure 31. This is one of the risks of using numerical optimization and placing hard constraints on the position, of course the hard constraints are meant to be safety boundaries, if they are violated you most likely have bigger problems than the trajectory planner spitting out gibberish. If the OS is ever inside a hard constraint like this there needs to be a contingency algorithm ready to step in and make escape maneuvers.

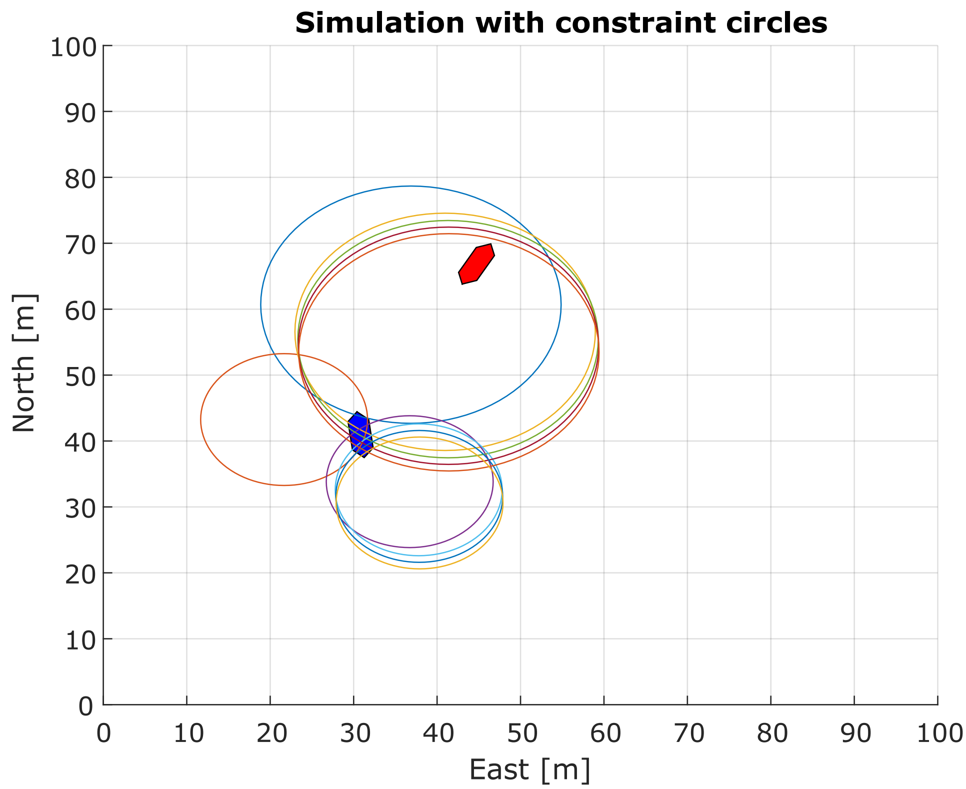
Blocked Path:
This isn't really a problem as much as it is the author wanting to show more closely what happens during the first three iterations of the Canals simulation. This is so that the looping process mentioned when discussing the Canals result are a bit easier to understand. The first frame is the first time the algorithm has been run, in this state there are no obstacles enabled and the optimal trajectory closely hugs the reference. In the next frame the obstacles are enabled and the resulting trajectory becomes split in half and infeasible as one side of the trajectory ends up on the far side of the blockading obstacle. The last frame shows the resulting trajectory after the speed reference was lowered drastically, observe how the beginning of the optimal trajectory wiggles a bit as it isn't actually possible to slow down as fast as the reference demands. These are seen in Figure 32.

"Wrap To 2 PI" problem:
Finally, a closer look at the nebulous wrap2topi problem that has been mentioned a

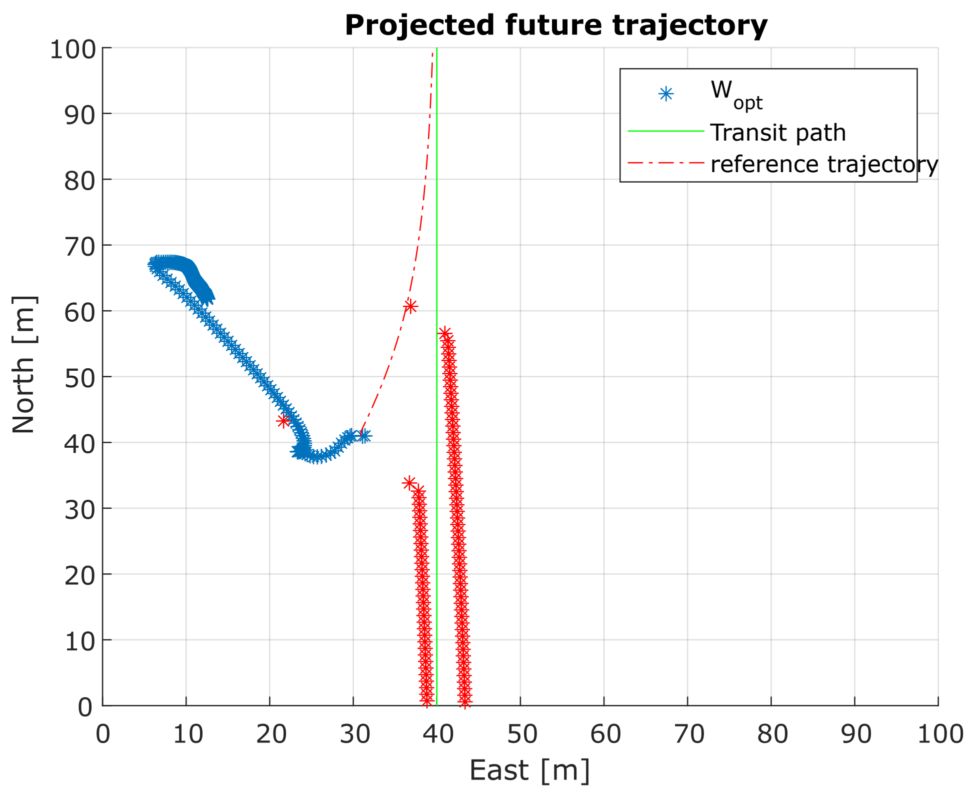
couple of times. This is a special simulation to show the consistenty of the problem, the first plot in Figure 33 shows the heading reference and "optimal" values for each control interval k. The second plot shows the projected optimal trajectory. when trying to turn from a heading due south towards a heading due west the reference for the heading experiences a discontinity as it jumps from π radians to $-\frac{\pi}{2}$ radians. And for some reason the solver decides to follow the reference, when at the time of this simulation the heading and heading reference did not appear in the cost function. I have no idea why the heading was followed like this, to my knowledge the heading should come "by itself" based on the dynamics of the system and the suppresion of any sway. When only surge is allowed the heading has to be pointing in the right direction to keep up with the reference trajectory, it shouldn't matter if it's out of phase by $2n\pi$ radians. The core of this problem was never discovered, but a fix was luckily not too complicated to implement. The fix was discussed when constructing the NLP in Chapter 3.3.

Trajectory stuck:
This quirk is mostly amusing, but can be a killer if left unpatched. In Figure 34 you can see the projected optimal trajectory nicely tucked inside a small island, where it is stuck and can not get out. The problem that caused this is two-fold. The first issue is that the algorithm is unable to settle on a consistent trajectory, which means that the static obs will flicker in and out of being active. Recall that the static obstacles are created using the previous optimal trajectory as an anchor for checking future positions. The second problem is that the static constraint lines are active in both directions; it is proximity to the line that is illegal, not being on a "wrong" side. So if the optimal trajectory jumps around a lot because the solver is unable to find a good consistent optimal solution it might eventually jump inside a static obstacle polygon and get stuck. The fix for this problem turned out to be rather simple luckily, I just needed to check for feasiblity before substituting in the previous optimal path as initial guess.

Leaning into turns:
This problem is simply a quirk of numerical optimization and my cost function. If you look at Figure 35 you will see that the OS turns ever so slightly the wrong way before executing the give way maneuver. This is actually a very big deal with respects to COLREGs, and therefor a highly undersired behaviour. What happens is that due to the way the cost function is set in this thesis smooth curves are "cheaper" than wavery turns, leaning ever so slighly the wrong way before beginning the proper turn makes the turn smoother overall, making it the optimal trajectory. I could not find a solution to this problem short of redesigning the cost function, luckily it's not that big of a problem for the simulations. In a real life scenario one would hope that inertias and dampening would drown this quirk out.

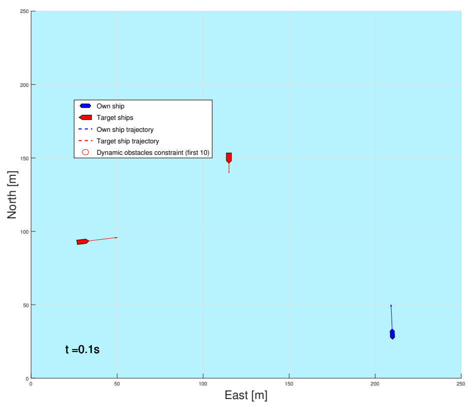


(a) When prediction goes wrong, the OS can get caught by moving constraints. (Old style figure).

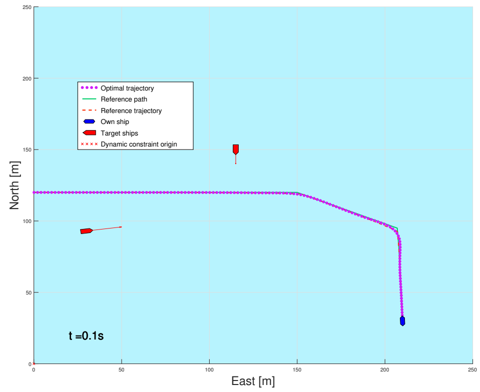


(b) When caught inside an active constraint, the solver is unable to find a feasible solution. (Old style figure).

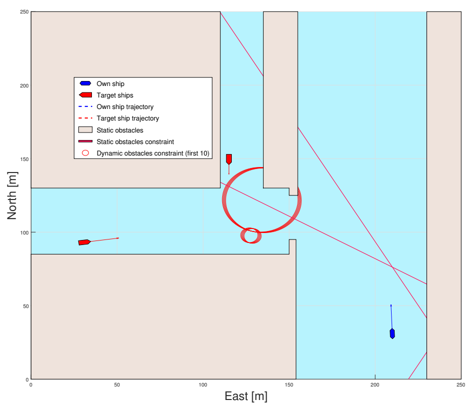
Figure 31: This is what can happen when the prediction does not match the actual trajectory of TSs.



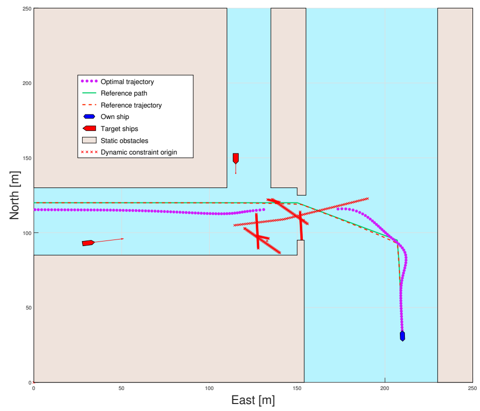
(a) Start of simulation, no active obstacles.



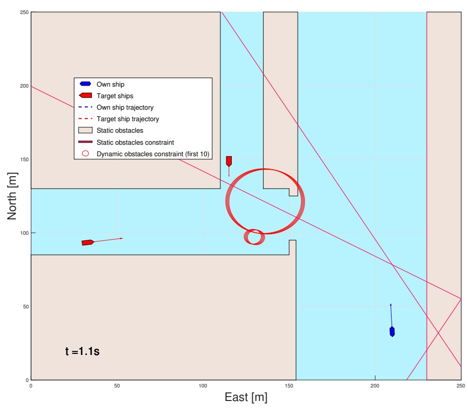
(b) Start of simulation, no active obstacles.



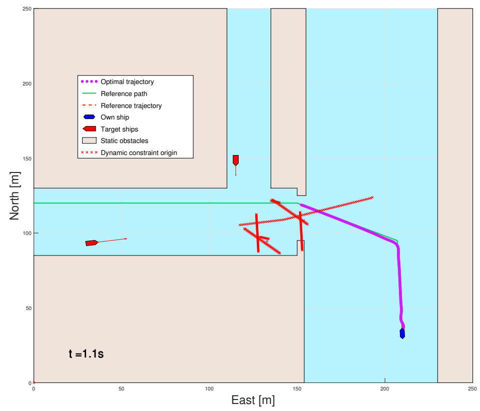
(c) Obstacles activate, breaking the optimal trajectory.



(d) Obstacles activate, breaking the optimal trajectory.

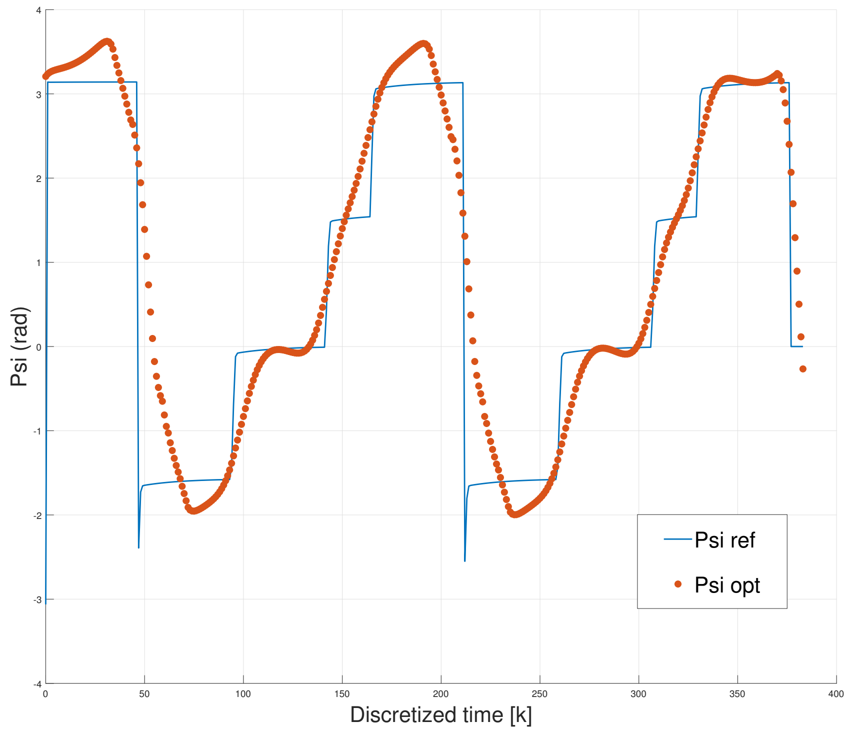


(e) Speed is reduced, resulting in a shorter optimal trajectory.

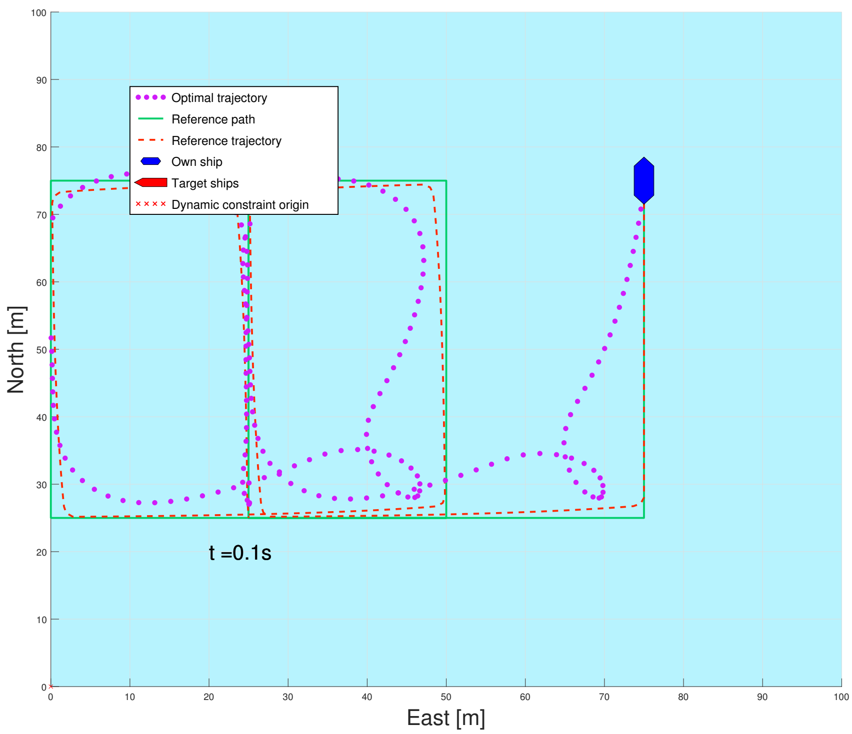


(f) Speed is reduced, resulting in a shorter optimal trajectory.

Figure 32: How optimal path is calculated with lower speed when infeasibility is detected.



(a) ref



(b) wopt

Figure 33: Without proper course reference, this sometimes happens.

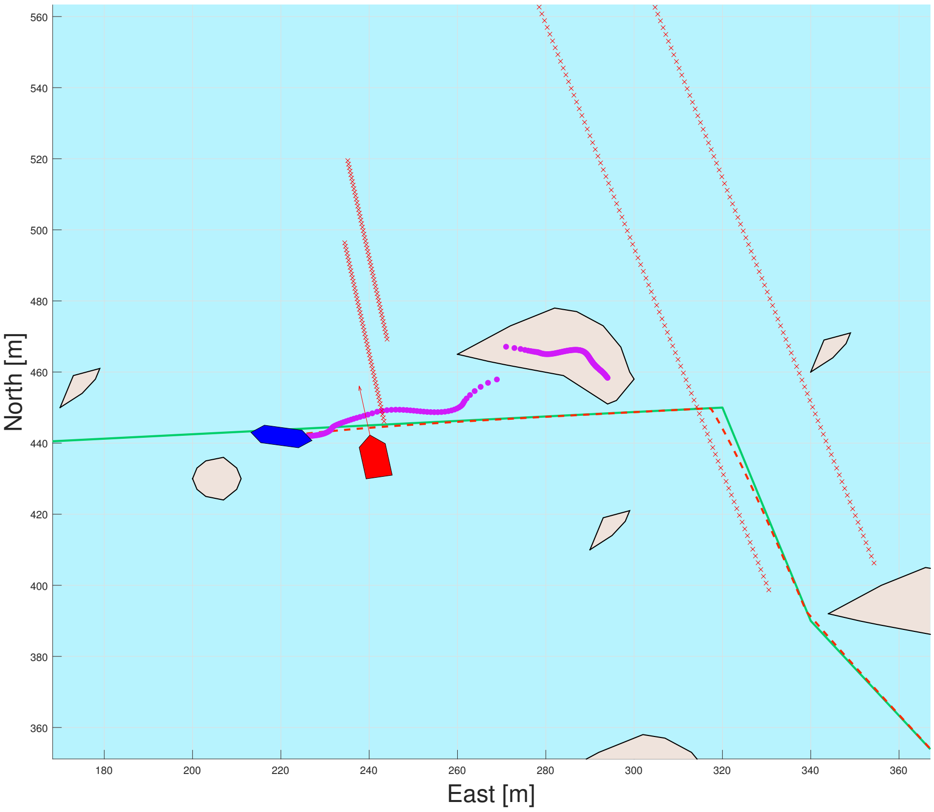
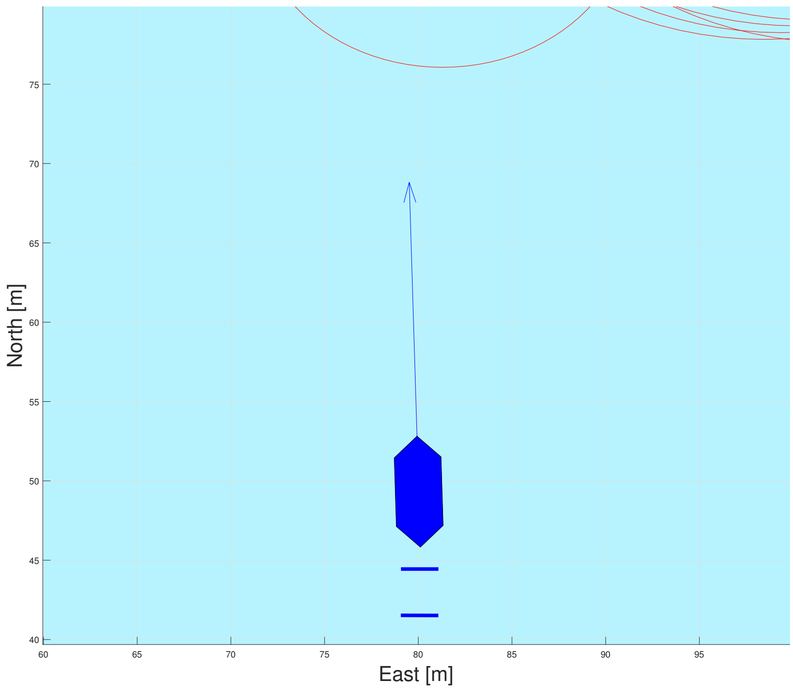
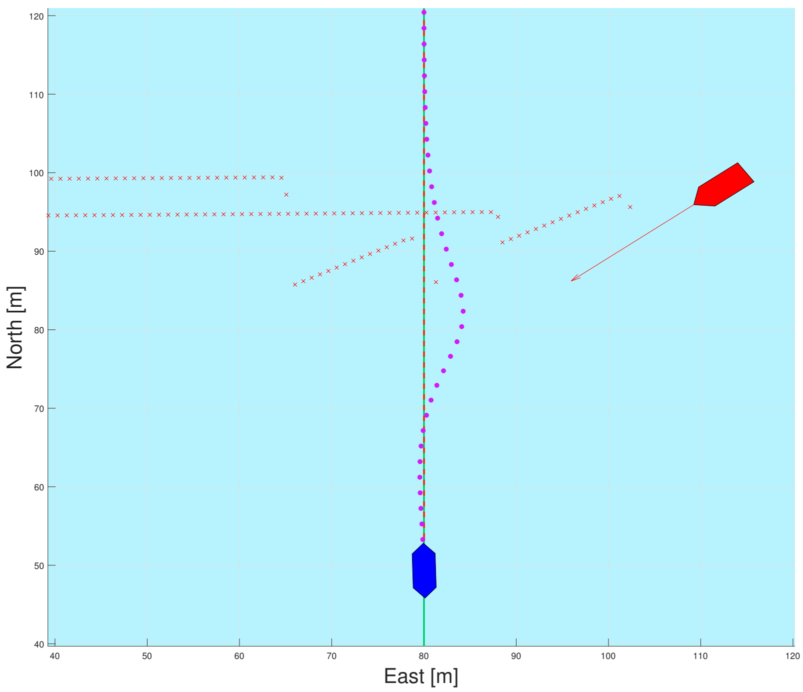


Figure 34: Stuck inside a static obstacle.



(a) By zooming in it is observed that the OS turns slightly to port side.



(b) Meanwhile the optimal trajectory clearly is clearly a turn to starboard.

Figure 35: A quirk of numerical optimization, sometimes turning to the wrong side leads to a 'smoother' curve.

4.3 Discussion

- Hvorfor er viktigere en hva
- ikke overanalyser resultat, ikke dra ville konklusjoner.
- Hvis et resultat er mye værre enn forventet kan det godt være det er bugs.
- i tillegg til det resultatene viser kan jeg også skrive om det jeg kan se med debugging.
- WrapTo2Pi problems (shortest signed angle stuff)
- Turning the wrong way to get a more even turn, Optimization leads to this problem.
- if(~isempty(previous_w_opt)) && feasibility == previous_feasibility && feasibility skaper problemer
- We really don't want to put a cost on heading reference more than neccessary, heading will often not be correct due to disturbances. heading is also just plain wrong any time we deviate from the reference trajecotry.
- With 'full' prediction solving the NLP is often computationally more efficient due to a better previous_w_opt.
- WHEN BEING OVERTAKEN: needs a better method for Standing On, putting constraints on overtaking vessel is not sufficient.
  See helloya rev without pred.
- Standing on in general needs a better way of handling constraints.
- Perhaps an alternative way of achieving COLREGs compliance would be to asess whether the OS is in 'Stand on' or 'Give way' mode, and modifying the cost function as well as dynamically placing constraints based on the situation. instead of just placing constraints that lead to 'immitating' COLREGs compliance.
- I wanted to test the algorithm agasint itself, by having all vessels in a situation be controlled by it's own instance. However the logic rewrite needed to conduct this test proved to be a bit too much
- Performance is actually quite poor when TS cooperate in ways that were not predicted, because that can 'drag' the optimal trajectory along with it.

It is impossible to capture all the intricacies and quirks of the algorithm behaviour in still images, the video results are a much better depiction.

The static obstacles present in the Canals scenario. proved to be a bit of a challenge, and it was this scenario that prompted the creation of the feasibility check, as well as a desire for a better static obstacles implementation. With circular static obstacle constraints this scenario was a nightmare to run due to the exceedingly long time to calculate an optimal trajectory. After the new line constraints were introduced a new problem of getting the optimal trajectory stuck inside static obstacles was observed, leading to the feasibility check. What used to happen is that when the bottleneck was blocked by the dynamic obstacles, the code for finding static obstacles would not catch many of the walls that are behind the bottleneck, leading to an optimal trajectory that would sometimes pass through the walls. The next iteration, when the static obstacles algorithm did catch the walls; half the optimal trajectory points would be pushed out of the static obstacle, while the other half would be pushed deeper in. An unrecoverable situation without the feasibility check to forcefully get rid of bad optimal trajectories as initial guesses.
Canals also inspired another feature of the algorithm, running the first iteration without any active obstacles. The rationale for this is that ideally the ASV or autonav running this algorithm would never be turned when a collision is imminent. By not having any

active obstacles for the first iteration the algorithm is able to find an optimal trajectory a lot faster than normal. This means quickly getting a good initial guess for the next iteration when obstacles are turned on.

Constraint placement is just plain bad sometimes...

being overtaken is very bad...

(Skjaergard with traffic) led to a logic change for keeping the previous optimal trajectory as initial guess, what would sometimes happen before the change was that the optimal trajectory would get stuck inside one of the islands, amusing for sure, but terrible for performance. There were also experiments with placing bugger islands dead center on the reference path, but that experiment wasn't given enough time to yield good results, and was ultimately scrapped. Ultimately the algorithm doesn't work all too well if the reference path or reference trajectory passes through an obstacle. it is able to adjust to small mistakes where the reference strafes close by an obstacle, but it can't find it's way if too much of the reference ends up in illegal positions. Of course, this isn't entirely unexpected, if the planner was able to deftly dodge all sorts of islands in the middle of the reference path it wouldn't need waypoints at all.

4.4 Improvements over Previous Version

- Definite improvements in terms of computational efficiency. This greatly increases the likelyhood of finding an optimal solution
- Because of the better efficiency the algorithm is also able to handle more control intervals, This means it is better at handling both greater time horizon and shorter control interval steps.
- The new method for handling static obstacles is much less prone to misplaced or inefficnet constraints. (her ta gjerne med figuren som viser problemer med sirkel constraints for statiske hindringer).
- The new way of handling dynamic constraints should in theory make the algorithm better suited for more complex situations with more agents, however the placement of dynamic constraints remains largely unchanged. Dynamic Constraint placement is bigger 'bottleneck' than agent culling for how complex situations are handled.
- More robust when an encounter leads to an infeasible solution.
- Improved COLREGs assessment
- But does it behave *noticeably* better? Yes.

5 Conclusion and Future Work

Conclusion

- Summary of results, obviously.
- Compare with the 'problem description', have I succesfully contributed to anything?
- Final thoughts on my own work

Future Work

- More work needed for optimizing placement of constraint, as well as when constraints need to be active.
- More work needed for situational awareness adjustion of parameters
- more work needed to examine scenarios with and without full TS prediction; more velocities, more variation of vessel sizes, more diverse enviornments.
- More work related to a variable cost function, a more adaptable cost function could for example yield better COLREGs compliance.
- More work related to Optimizing runtime of algorithm, tuning the IPOPT tolerances to balance computation speed and desired behaviour.
- More work related to tuning COLREGs compliance, testing more COLREGs situations and quantifying what constitutes good behaviour.
- Extracting the Algorithm from the MATLAB simulator it's built into and making a more stand-alone / generic software or algorithm.
- And more that I will think about as I write.

References

Andersson, Joel AE, Joris Gillis, Greg Horn, James B Rawlings and Moritz Diehl (2019). 'CasADi: a software framework for nonlinear optimization and optimal control'. In: *Mathematical Programming Computation* 11.1, pp. 1–36.

Cho, Yonghoon, Jungwook Han and Jinwhan Kim (2018). 'Intent inference of ship maneuvering for automatic ship collision avoidance'. In: *IFAC-PapersOnLine* 51.29, pp. 384–388.

Cockcroft, AN and JNF Lameijer (2012). *Guide to the Collision Avoidance Rules*. Oxford: Butterworth-Heinemann. Cockcroft, AN and Lameijer, JNF.

Eriksen, H. Bjørn-Olav and Morten Breivik (2017). 'MPC-based mid-level collision avoidance for ASVs using nonlinear programming'. In: *2017 IEEE Conference on Control Technology and Applications (CCTA)* (Mauna Lani Bay Hotel). IEEE. Hawaii, USA, pp. 766–772.

Fossen, Thor I (2011). *Handbook of marine craft hydrodynamics and motion control*. John Wiley & Sons.

Fossen, Thor I and Tristan Perez (2004). *Marine Systems Simulator (MSS)*. URL: https://github.com/cybergalactic/MSS.

Gros, Sebastien (2017). *Numerical optimal control, lecture 4: Shooting methods*. Video lecture. URL: https://www.youtube.com/watch?v=UqWRcbdwPP8.

Huang, Yamin, Linying Chen, Pengfei Chen, Rudy R Negenborn and PHAJM Van Gelder (2020). 'Ship collision avoidance methods: State-of-the-art'. In: *Safety science* 121, pp. 451–473.

IMO (1972). *International Regulations for Preventing Collisions at Sea*. Wikisource Archive. URL: https://en.wikisource.org/wiki/International_Regulations_for_Preventing_Collisions_at_Sea.

Kufoalor, D. K.M., E. F. Brekke and T. A. Johansen (2018). 'Proactive Collision Avoidance for ASVs using A Dynamic Reciprocal Velocity Obstacles Method'. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2402–2409. DOI: 10.1109/IROS.2018.8594382.

Lekkas, Anastasios M and Thor I Fossen (2013). 'Line-of-sight guidance for path following of marine vehicles'. In: *Advanced in marine robotics*, pp. 63–92.

Loe, Øivind (2007). 'Collision avoidance concepts for marine surface craft'. In: *Specialization project report. Norwegian University of Science and Technology (NTNU), Trondheim, Norway.*

Park, Shinkyu, Michal Cap, Javier Alonso-Mora, Carlo Ratti and Daniela Rus (2020). 'Social Trajectory Planning for Urban Autonomous Surface Vessels'. In: *IEEE Transactions on Robotics* 37.2, pp. 452–465.

Pedersen, Anders Aglen (2019). 'Optimization based system identification for the milliAmpere ferry'. MA thesis. NTNU.

Qin, S Joe and Thomas A Badgwell (1997). 'An overview of industrial model predictive control technology'. In: *AIche symposium series*. Vol. 93. 316. New York, NY: American Institute of Chemical Engineers, 1971-c2002., pp. 232–256.

Schöller, Frederik ET, Thomas T Enevoldsen, Jonathan B Becktor and Peter N Hansen (2021). 'Trajectory prediction for marine vessels using historical AIS heatmaps and long short-term memory networks'. In: *Proceedings of 13th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles Elsevier*. Vol. 54. 16. IFAC. Oldenburg, Germany: Elsevier, pp. 83–89.

Tam, CheeKuang and Richard Bucknall (2010). 'Collision risk assessment for ships'. In: *Journal of Marine Science and Technology* 15.3, pp. 257–270.

Thyri, Emil Hjelseth and Morten Breivik (2022). 'A domain-based and reactive COLAV method with a partially COLREGs-compliant domain for ASVs operating in confined waters'. In: *Field Robotics* 2, pp. 632–677. DOI: https://doi.org/10.55417/fr.2022022.

Vagale, Anete, Rachid Oucheikh, Robin T Bye, Ottar L Osen and Thor I Fossen (2021). 'Path planning and collision avoidance for autonomous surface vehicles I: A review'. In: *Journal of Marine Science and Technology* 26, pp. 1292–1306.

Vestad, Vegard Nitter (2019). 'Automatic and practical route planning for ships'. MA thesis. NTNU.

Wächter, Andreas and Lorenz T Biegler (2006). 'On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming'. In: *Mathematical programming* 106.1, pp. 25–57.

Woerner, Kyle (2016). 'Multi-contact protocol-constrained collision avoidance for autonomous marine vehicles'. PhD thesis. Massachusetts Institute of Technology, USA.

Wright, Stephen, Jorge Nocedal et al. (1999). 'Numerical optimization'. In: *Springer Science* 35.67-68, p. 7.

Zhang, Xinyu, Chengbo Wang, Lingling Jiang, Lanxuan An and Rui Yang (2021). 'Collision-avoidance navigation systems for Maritime Autonomous Surface Ships: A state of the art survey'. In: *Ocean Engineering* 235, p. 109380.

Appendix

```
function [vessel, resulting_trajectory] = MPC_with_Assist(vessel, tracks, parameters, settings)
import casadi.*

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% INITIAL CONDITIONS and persistent variables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    persistent previous_w_opt
    persistent previous_w_opt_F
    persistent F
    persistent firsttime
    persistent obstacle_state
    persistent cflags
    persistent previous_eta_ref
%     persistent pimultiplier
%     persistent previous_feasibility

    % Initialize CasADi

    if(isempty(firsttime))
        firsttime = 1;
        obstacle_state = false; % No obstacles on first iteration
        previous_w_opt = [];
        cflags = [];
        previous_w_opt_F = [];
        previous_eta_ref = [];
%         pimultiplier = 0;
%         previous_feasibility = 0;
    end


    %Initialize COLREGs flag.
    if(isempty(cflags)) % THIS CAN BE USED TO HARDCODE FLAGS IF NEEDED:
         cflags = zeros([1,size(tracks,2)]);
%          cflags = [2, 1];
    end

    %% Settings
    simple = settings.simple; % Enable to discard all traffic pattern assistance.
%     chaos = 0; % Do not use
%     pimultiplier = 0;
    %%

    if ~isempty(tracks)
        dynamic_obs(size(tracks,2)) = struct;
    else
        dynamic_obs = []; % Failsafe in case there are no dynamic obstacles present.
    end

    for i = 1:size(tracks,2)

        if simple
            tracks(i).wp(1:2) = [tracks(i).eta(1);tracks(i).eta(2)
```

```
                ];
            tracks(i).wp(3:4) = [tracks(i).eta(1);tracks(i).eta(2)] +...
                1852 * [cos(tracks(i).eta(3)) , sin(tracks(i).eta(3))]';
            tracks(i).wp = [tracks(i).wp(1:2)' tracks(i).wp(3:4)']; % Truncate excess waypoints.
            tracks(i).current_wp = 1;
        end

        [dynamic_obs(i).cflag, dynamic_obs(i).dcpa, dynamic_obs(i).tcpa] = COLREGs_assessment(vessel,tracks(i),cflags(i));
        cflags(i) = dynamic_obs(i).cflag; % Save flag in persistent variable for next iteration.
    end

    [N,h] = DynamicHorizon(vessel, dynamic_obs);
%     T = N * h;

    if(isempty(F))
        F = CasadiSetup(h,N);
    end


    %% Feasibility check
%     if N < 180
%         fixed_feas = 1;
%     else
%         feasibility = 1;
%     end

%   OLD AND OUTDATED STUFF
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % previous feas. | Feasibility | obstacle state %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %       1        |      1      |       1        %
    %       0        |      1      |       0        %
    %       1        |      0      |       0        %
    %       0        |      0      |       0        %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if ~isempty(previous_w_opt_F)
        feasibility = feasibility_check(previous_w_opt_F);
    else
        feasibility = 1;
    end

%   OLD AND OUTDATED STUFF
%     obstacle_state = false;
%     if previous_feasibility && feasibility
%         obstacle_state = true;
%     end
%     previous_feasibility = feasibility;

%     feasibility = 1;

    %%

```

```
    % Initialize position and reference trajectory.
    initial_pos = vessel.eta;
    if wrapTo2Pi(initial_pos(3)) < pi/6
%       initial_pos(3) = wrapTo2Pi(initial_pos(3)); % THIS NEEDS MORE WORK
        if ~isempty(previous_w_opt) && ssa(initial_pos(3)-previous_w_opt(3)) > pi
            if initial_pos(3) > previous_w_opt(3)
                initial_pos(3) = wrapTo2Pi(initial_pos(3));
            end
        elseif ~isempty(previous_w_opt)
            initial_pos(3) = wrapTo2Pi(initial_pos(3));
        end
    end
    initial_vel = vessel.nu;

    % reference LOS for OS and TS
    [reference_trajectory_los, ~] = reference_trajectory_from_dynamic_los_guidance(vessel, parameters, h, N, feasibility);
    for i = 1:size(tracks,2)
    dynamic_obs(i).traj = reference_trajectory_from_dynamic_los_guidance(tracks(i), parameters, 0.5, N, feasibility);
    end

    %% Obstacles
    enable_Static_obs = obstacle_state; %Obstacle state is purely for debugging.
    enable_dynamic_obs = obstacle_state;
    static_obs = get_global_map_data();
%     interpolated_static_obs = Interpolate_static_obs(static_obs);
%     Static_obs_constraints = Static_obstacles_check(static_obs, reference_trajectory_los);
%     THIS CHECK IS HANDELED IN THE MAIN LOOP NOW


    %% NLP initialization.
    % Start with empty NLP.
    w={};
    w0 = zeros(9*N+6,1); % Initial guess.
    lbw = zeros(9*N+6,1);
    ubw = zeros(9*N+6,1);
    J = 0;
    g={};
    lbg = zeros(50*N+6,1);
    ubg = zeros(50*N+6,1);

    % "lift" initial conditions.
    Xk = MX.sym('X0',6);
    w = [w {Xk}];
    lbw(1:6) = [-inf; -inf; -inf; -2.5; -2.5; -pi/4];
    ubw(1:6) = [ inf;  inf;  inf;  2.5;  2.5;  pi/4];
    w0(1:6) = [initial_pos(1); initial_pos(2); initial_pos(3); initial_vel(1); initial_vel(2); initial_vel(3)];


%     Uk = MX.sym('U0',3);
```

```
%       w = {w{:}, Uk};
%       lbw = [lbw; -2.5; -2.5; -pi/4];
%       ubw = [ubw; 2.5; 2.5; pi/4];
%       w0 = [w0; 0; 0; 0];

        g = [g, {[initial_pos; initial_vel] - Xk}];
        lbg(1:6) = [0; 0; 0; 0; 0; 0]';
        ubg(1:6) = [0; 0; 0; 0; 0; 0]';

%       g = [g, {initial_vel - Xk}];
%       lbg = [lbg; 0; 0; 0];
%       ubg = [ubg; 0; 0; 0];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MAIN LOOP
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
loopdata = zeros(N+1,7);
static_obs_collection = [];
NaNs = [NaN; NaN; NaN];
c_origins = zeros(2,50*N+6);
c_radius = zeros(50*N+6,1);
c_counter = 1;
g_counter = 7;
%loopdata = [k xref_i uref_i]
    for k = 0:N-1
        % New NLP variable for control.

        Tauk = MX.sym(['Tau_' num2str(k)], 3);
        w = [w {Tauk}]; %#ok<AGROW>
        lbw(7+k*9:9+k*9)= [-800;   -800;    -800];
        ubw(7+k*9:9+k*9) = [800;    800;     800];
        w0(7+k*9:9+k*9) = [0; 0; 0];

        % Integrate until the end of the interval.
        eta_dot_ref = [reference_trajectory_los(3:4,k+1);...
                      (atan2(reference_trajectory_los(4,k+2),
                         reference_trajectory_los(3,k+2)) - ...
                       atan2(reference_trajectory_los(4,k+1),
                         reference_trajectory_los(3,k+1))) / h];

        surge_ref = sqrt(eta_dot_ref(1)^2 + eta_dot_ref(2)^2);
        nu_ref = [surge_ref;0;eta_dot_ref(3)]; %Burde være vessel.
            speed som referanse.
%           nu_ref = [sqrt(eta_dot_ref(1)^2 + eta_dot_ref(2)^2); 0;
    eta_dot_ref(3)];
%           nu_ref = vessel.eta_dot_ref;

        eta_ref = [reference_trajectory_los(1:2,k+1); atan2(
            eta_dot_ref(2),eta_dot_ref(1))];
%           eta_ref = [reference_trajectory_los(1:2,k+1); wrapTo2Pi(
    atan2(eta_dot_ref(2),eta_dot_ref(1)))];

        % We want the reference to start close to initial position.
        if k == 0
            unwrap_diff = abs(eta_ref(3) - initial_pos(3));
            wrap_diff = abs(wrapTo2Pi(eta_ref(3)) - initial_pos(3))
                ;

```

```
                if unwrap_diff > wrap_diff % check if distance between ref and init_pos is greater when unwrapped
                    eta_ref(3) = wrapTo2Pi(eta_ref(3));
                end
                previous_eta_ref = eta_ref;
            end


            %% Test greier
            if k > 0
                eta_ref(3) = previous_eta_ref(3) + ssa(eta_ref(3) - previous_eta_ref(3));
                previous_eta_ref = eta_ref;
    %             unwrap_diff = abs(eta_ref(3) - previous_eta_ref(3));
    %             wrap_diff = abs(wrapTo2Pi(eta_ref(3)) - previous_eta_ref(3));
    %
    %             if unwrap_diff > wrap_diff % check if distance between ref and init_pos is greater when unwrapped
    %                 eta_ref(3) = wrapTo2Pi(eta_ref(3));
    %             end
    %             previous_eta_ref = eta_ref;
            end
    %         if k > 0
    %             if wrapTo2Pi(previous_eta_ref(3)) > 21*pi/12 && wrapTo2Pi(eta_ref(3)) < 3*pi/12 % Positive wrap
    %                 pimultiplier = pimultiplier + 2*pi;
    %             end
    %             if wrapTo2Pi(previous_eta_ref(3)) < 3*pi/12 && wrapTo2Pi(eta_ref(3)) > 21*pi/12 % Negative wrap
    %                 pimultiplier = pimultiplier - 2*pi;
    %             end
    %         end
    %         eta_ref(3) = eta_ref(3) + pimultiplier;
    %         previous_eta_ref = eta_ref;
            %%
    %         eta_ref = [reference_trajectory_los(1:2,k+1); 0];

            xref_i = [eta_ref; nu_ref];

            Fk = F('x0', Xk, 'tau', Tauk, 'Xd', xref_i);
            Xk_end = Fk.xf;
            J = J + Fk.qf;

            % New NLP variable for state at the end of interval.
            Xk = MX.sym(['X_' num2str(k+1)], 6);
            w = [w {Xk}]; %#ok<AGROW>
            lbw(10+k*9:15+k*9) = [-inf; -inf; -inf; -2.3; -2.3; -pi/4];
            ubw(10+k*9:15+k*9) = [inf; inf; inf; 2.3; 2.3; pi/4];
            w0(10+k*9:15+k*9) = [xref_i(1); xref_i(2); xref_i(3); xref_i(4); xref_i(5); xref_i(6)];

    %         Uk = MX.sym(['U_' num2str(k+1)], 3);
    %         w = {w{:}, Uk};
    %         lbw = [lbw; -2.5; -2.5; -pi/4];
    %         ubw = [ubw; 2.5; 2.5; pi/4];
    %         w0 = [w0; 0; 0; 0];

```

```
        % Add constraints.
        g = [g {Xk_end - Xk}]; %#ok<AGROW>
        lbg(g_counter:g_counter+5) = [0; 0; 0; 0; 0; 0];
        ubg(g_counter:g_counter+5)= [0; 0; 0; 0; 0; 0];
        g_counter = g_counter + 6;


        if ~isempty(dynamic_obs) && ~firsttime && enable_dynamic_obs

        for i = 1:size(dynamic_obs,2)

            if dynamic_obs(i).cflag == 1 % HEAD ON
                if (k > (floor(dynamic_obs(i).tcpa/h) - floor(30/h))) && (k < (floor(dynamic_obs(i).tcpa/h) + floor(30/h)))
                    %% Constraint rundt båten, origo offset til styrbord
                    %Constraint 1:
                    c_orig = place_dyn_constraint(dynamic_obs, k, i, pi/2, 13);
                    c_rad = 22;
                    g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                    lbg(g_counter) = c_rad^2;
                    ubg(g_counter) = inf;
                    g_counter = g_counter + 1;
                    c_origins(:,c_counter) = c_orig;
                    c_radius(c_counter) = c_rad;
                    c_counter = c_counter + 1;

                    %Constraint 2:
                    c_orig = place_dyn_constraint(dynamic_obs, k, i, pi/2, 38);
                    c_rad = 5;
                    g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                    lbg(g_counter) = c_rad^2;
                    ubg(g_counter) = inf;
                    g_counter = g_counter + 1;
                    c_origins(:,c_counter) = c_orig;
                    c_radius(c_counter) = c_rad;
                    c_counter = c_counter + 1;
                end
            elseif dynamic_obs(i).cflag == 2 % GIVE WAY
                if (k > (floor(dynamic_obs(i).tcpa/h) - floor(20/h))) && (k < (floor(dynamic_obs(i).tcpa/h) + floor(20/h)))
                    %% Forbudt å snike seg forbi forran target ship
                    %c_orig = place_dyn_constraint(dynamic_obs, control
                    %                               interval, TS id, angle
                    %                               offset, distance offset)
                    c_orig = place_dyn_constraint(dynamic_obs, k, i, pi/8, 10);
                    c_rad = 18;
```

```
                g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                lbg(g_counter) = c_rad^2;
                ubg(g_counter) = inf;
                g_counter = g_counter + 1;
                c_origins(:,c_counter) = c_orig;
                c_radius(c_counter) = c_rad;
                c_counter = c_counter + 1;

                %Constraint 2:
                c_orig = place_dyn_constraint(dynamic_obs, k, i, pi/12, 33);
                c_rad = 10;
                g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                lbg(g_counter) = c_rad^2;
                ubg(g_counter) = inf;
                g_counter = g_counter + 1;
                c_origins(:,c_counter) = c_orig;
                c_radius(c_counter) = c_rad;
                c_counter = c_counter + 1;
            end
        elseif dynamic_obs(i).cflag == 3 % STAND ON
            if (k > (floor(dynamic_obs(i).tcpa/h) - floor(20/h))) && (k < (floor(dynamic_obs(i).tcpa/h) + floor(20/h)))
                %% Contraint rundt TS som sikkerhetsmargin
                c_orig = place_dyn_constraint(dynamic_obs, k, i, pi, 0);
                c_rad = 7;
                g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                lbg(g_counter) = c_rad^2;
                ubg(g_counter) = inf;
                g_counter = g_counter + 1;
                c_origins(:,c_counter) = c_orig;
                c_radius(c_counter) = c_rad;
                c_counter = c_counter + 1;
            end
        elseif dynamic_obs(i).cflag == 4 % OVERTAKING
            if (k > (floor(dynamic_obs(i).tcpa/h) - floor(20/h))) && (k < (floor(dynamic_obs(i).tcpa/h) + floor(20/h)))
                %% Constraint rundt TS som sikkerhetsmargin
                c_orig = place_dyn_constraint(dynamic_obs, k, i, 0, 0);
                c_rad = 10;
                g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                lbg(g_counter) = c_rad^2;
                ubg(g_counter) = inf;
                g_counter = g_counter + 1;
                c_origins(:,c_counter) = c_orig;
                c_radius(c_counter) = c_rad;
                c_counter = c_counter + 1;
            end
        elseif dynamic_obs(i).cflag == 5 % SAFE
            if dynamic_obs(i).dcpa < 20
```

```
                        if (k > (floor(dynamic_obs(i).tcpa/h) - floor(20/h))) && (k < (floor(dynamic_obs(i).tcpa/h) + floor(20/h)))
                            c_orig = place_dyn_constraint(dynamic_obs, k, i, 0, 0);
                            c_rad = 8;
                            g = [g {(Xk(1:2) - c_orig)'*(Xk(1:2) - c_orig)}]; %#ok<AGROW>
                            lbg(g_counter) = c_rad^2;
                            ubg(g_counter) = inf;
                            g_counter = g_counter + 1;
                            c_origins(:,c_counter) = c_orig;
                            c_radius(c_counter) = c_rad;
                            c_counter = c_counter + 1;
                        end
                    end
                end
            end
            end

            %static obstacle constraints:
            if(enable_Static_obs) && ~firsttime && (~isempty(static_obs))
                selected_trajectory = reference_trajectory_los;
                if(~isempty(previous_w_opt))
                    selected_trajectory = previous_w_opt;
                end
                static_obs_constraints = Static_obstacles_check_Iterative(static_obs, selected_trajectory, k);
                static_obs_collection = [static_obs_collection, static_obs_constraints, NaNs]; %#ok<AGROW>
                for i = 1:size(static_obs_constraints,2)
                    static_obs_y1 = static_obs_constraints(1,i);
                    static_obs_x1 = static_obs_constraints(2,i);
                    pi_p = static_obs_constraints(3,i);

                    Static_obs_crosstrack_distance = abs(-(Xk(2)-static_obs_x1) * cos(pi_p) + (Xk(1) - static_obs_y1) * sin(pi_p));
                     g = [g {Static_obs_crosstrack_distance}]; %#ok<AGROW>
                        lbg(g_counter) = 5;
                        ubg(g_counter) = inf;
                        g_counter = g_counter + 1;
                end

    %%              OLD CODE:
    %             [~, cols] = size(Static_obs_constraints);
    %             for i = 1:cols
    %                 g = [g, {(Xk(1:2) - Static_obs_constraints(:,i))'*(Xk(1:2) - Static_obs_constraints(:,i)) - 5^2}]; % Endre constraints
    %                 lbg = [lbg; 0];
    %                 ubg = [ubg; inf];
    %             end
            end

```

```
            loopdata(k+1,:) = [k, xref_i'];

        end

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %% Optimal solution and updating states
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        loopdata(end,:) = [k+1, xref_i'];

        % Truncate lbg, ubg:
        lbg = lbg(1:g_counter-1);
        ubg = ubg(1:g_counter-1);

        % Create an NLP solver.
        prob = struct('f', J, 'x', vertcat(w{:}), 'g', vertcat(g{:}));
    %     options = struct;
        options.ipopt.max_iter = 400;
        options.ipopt.print_level = 0;
        options.ipopt.nlp_scaling_method = 'none';
        options.ipopt.dual_inf_tol = 5;
        options.ipopt.tol = 5e-3;
        options.ipopt.constr_viol_tol = 1e-1;
    %     options.ipopt.hessian_approximation = 'limited-memory';
        options.ipopt.compl_inf_tol = 1e-1;
        options.ipopt.acceptable_tol = 1e-2;
        options.ipopt.constr_viol_tol = 0.01;
        options.ipopt.acceptable_dual_inf_tol = 1e10;
        options.ipopt.acceptable_compl_inf_tol = 0.01;
    %     options.ipopt.acceptable_obj_change_tol = 1e20;
    %     options.ipopt.diverging_iterates_tol = 1e20;

        if(firsttime)
            options.ipopt.max_iter = 200;
            options.ipopt.print_level =  5;
            firsttime = 0;
        end
        solver = nlpsol('solver', 'ipopt', prob, options);

        % Replace w0 with previous_w_opt:
    %     if(~isempty(previous_w_opt)) && feasibility ==
        previous_feasibility && feasibility
        if(~isempty(previous_w_opt)) && feasibility
            endindex = min(size(lbw,1),size(previous_w_opt,1));
            if endindex < size(lbw,1)
                % Add back w0 from NLP construction to fill the gap:
                previous_w_opt(end+1:size(lbw,1)) = w0(size(
                    previous_w_opt,1)+1:end);
                endindex = size(lbw,1);
            end
            w0 = previous_w_opt(1:endindex);
        end

        % Solve the NLP.
        clock = tic;
        sol = solver('x0', w0, 'lbx', lbw, 'ubx', ubw,...
                    'lbg', lbg, 'ubg', ubg);
        Solvertime = toc(clock); %Check here to see how long it took to
```

```
calculate w_opt. if Solvertime exceeds for example 6 seconds we know something might have went wrong.
        w_opt = full(sol.x);
%       w_opt(3:9:end) = wrapTo2Pi(w_opt(3:9:end));

        previous_w_opt = w_opt;
        previous_w_opt_F = w_opt;
        previous_feasibility = feasibility;
        if Solvertime > 30
            previous_w_opt = [];
        end
        %% Variables for plotting
        ploteverything(loopdata,w_opt, vessel, tracks, reference_trajectory_los, c_origins, c_radius, settings, static_obs_collection);

        obstacle_state = true;

        %% Update vessel states
        vessel.eta = w_opt(10:12);
%       vessel.nu = w_opt(4:6);
        vessel.nu = w_opt(13:15);
        vessel.eta_dot = rotZ(vessel.eta(3))*vessel.nu;
        resulting_trajectory = [w_opt(10:9:end), w_opt(11:9:end), w_opt(12:9:end), w_opt(13:9:end), w_opt(14:9:end), w_opt(15:9:end)]';     % TODO
```

```
function F = CasadiSetup(h, N)
import casadi.*

T = h * N;

    %% CasADi setup

    % System matrices.
    x = SX.sym('x',6); % x = [N, E, psi, u, v, r]'
    tau = SX.sym('tau',3); % tau = [Fx, Fy, Fn]';
    xref = SX.sym('xref',6); % xref = [Nref, Eref, Psi_ref, Surge_ref, sway_ref, r_ref]'


%     [R, M, C, D] = SystemDynamics(x, u); % Usikker på hvorvidt det funker
%     å sende CasADi systemer inn i en subfunksjon. Burde jo gå, men lar
%     være for nå.
    % Model Parameters.
    Xu = -68.676;    % Kg/s
    Xuu = -50.08;    % Kg/m
    Xuuu = -14.93;   % Kgs/(m^2)
%     Xv = -25.20;      % Kg/s
%     Xr = -145.3;      % Kgm/s
%     Yu = 90.15;       % Kg/s
    Yv = -8.69;       % Kg/s
    Yvv = -189.08;   % Kg/m
    Yvvv = -0.00613;% Kgs/(s^2) ? Kgs/(m^2)?
%     Yrv = -3086.95; % Kg
%     Yr = -24.09;      % Kgm/s
```

```
%       Yvr = -338.32;   % Kg
%       Yrr = 1372.06;   % Kg(m^2)
%       Nu = -38.00;     % Kgm/s
%       Nv = -97.26;     % Kgm/s
        Nvv = -18.85;    % Kg
        Nrv = 5552.23;   % Kgm
        Nr = -230.19;    % Kg(m^2)/s
        Nrr = -0.0063;   % Kg(m^2)
        Nrrr = -0.00067;% Kgms
%       Nvr = -5888.89; % Kgm

        m11 = 2131.80;   % Kg
        m12 = 1.00;      % Kg
        m13 = 141.02;    % Kgm
        m21 = -15.87;    % Kg
        m22 = 2231.89;   % Kg
        m23 = -1244.35; % Kgm
        m31 = -423.76;   % Kgm
        m32 = -397.64;   % Kgm
        m33 = 4351.56;   % Kg(m^2)

        c13 = -m22*x(5);
        c23 = m11*x(4);
        c31 = -c13;
        c32 = -c23*x(5);

        d11 = -Xu - Xuu * abs(x(4)) - Xuuu*(x(4)^2);
        d22 = -Yv - Yvv*abs(x(5)) - Yvvv*(x(5)^2);
        d23 = d22;
        d32 = -Nvv*abs(x(5)) - Nrv *abs(x(6));
        d33 = -Nr - Nrr*abs(x(6)) - Nrrr*(x(6)^2);


        % System dynamics.
        R = [cos(x(3))     -sin(x(3))     0;...
             sin(x(3))      cos(x(3))     0;...
                0              0          1];
        M = [m11    m12      m13;...
             m21    m22      m23;...
             m31    m32      m33];
        C = [0      0     c13;...
             0      0     c23;...
             c31  c32       0];
%       D = [d11        0           0;...
%             0        d22        d23;...
%             0        d32        50*d33];
%
%       M = eye(3)*1000;
        D = diag([200, 200, 1000]);
%       C = zeros(3);

%       Tau = pickthree(tau); %failed experiement.
        nu_dot = M\(tau -(C+D)*x(4:6));
        nu = x(4:6) + h*nu_dot; % This could almost certainly use a better integrator method.
        eta_dot = R*nu;

        xdot = [eta_dot; nu_dot];
```

```
%     Funker bra:
%     Kp = diag([8*10^-1, 8*10^-1]);
%     Ku = 6*10^2;
%     Kv = 8*10^2;

    % Objective function.
    Kp = diag([8*10^-1, 8*10^-1]); % Tuning parameter for positional reference deviation.
    Ku = 6.7*10^2; % Tuning parameter for surge reference deviation.
    Kv = 7.2*10^2;
%     Kv = 0;
%     Kr = 3*10^2; % Tuning parameter for yaw rate reference deviation.
%     Kt = 10^2;
    R2 = [cos(x(3))    -sin(x(3));...
          sin(x(3))     cos(x(3))];
    Error = R2'*(x(1:2) - xref(1:2));
    Kfy = 1 * 10^-5;

    %Test for heading
    K_phi = 6*10^-5;

    %L = Kp * norm(P - xref)^2 + Ku  * (u(1) - uref(1))^2 + Kr * (u(2) - uref(2))^2;
    %L = (P - xref)'* Kp * (P - xref) + Ku * (u_0'*u_0 - uref(1)'*uref(1))^2;
    %L = (P - xref)'* Kp * (P - xref) + Ku * (u(1) - uref(1))^2 + Kr * (u(2) - uref(2))^2;
    L = Error'* Kp * Error + Ku * (x(4)-xref(4))^2 + Kv * (x(5)-xref(5))^2 + Kfy * tau(2)^2 + K_phi * (ssa(x(3)-xref(3)))^2;% + Kr * (x(6) - xref(6))^2 + Kt * (tau'*tau) + Ku * (x(4) - xref(4))^2;

    % Continous time dynamics.
    f = Function('f', {x, tau, xref}, {xdot, L});

    % Discrete time dynamics.
    M = 4; %RK4 steps per interval
    DT = T/N/M;
    f = Function('f', {x, tau, xref}, {xdot, L});
    X0 = MX.sym('X0',6);
    Tau = MX.sym('Tau',3);
    Xd = MX.sym('Xd',6);
    X = X0;
    Q = 0;
    for j=1:M
        [k1, k1_q] = f(X, Tau, Xd);
        [k2, k2_q] = f(X + DT/2 * k1, Tau, Xd);
        [k3, k3_q] = f(X + DT/2 * k2, Tau, Xd);
        [k4, k4_q] = f(X + DT * k3, Tau, Xd);
        X=X+DT/6*(k1 +2*k2 +2*k3 +k4);
        Q = Q + DT/6*(k1_q + 2*k2_q + 2*k3_q + k4_q);
    end

    F = Function('F', {X0, Tau, Xd}, {X, Q}, {'x0', 'tau', 'Xd'}, {'xf', 'qf'});
```

```
end
```

```
function [flag, dCPA, tCPA] = COLREGs_assessment(vessel,tracks,
    cflag)
%% THIS FUNCTION EVALUATES ONE TARGET SHIP ONLY. TO EVALUATE MORE
    THE FUNCTION MUST BE CALLED FOR EACH TARGET SHIP IN YOUR
    SITUATION.
% a13 = 112.5; % Overtaking tolerance
% a14 = rad2deg(pi/8); % head-on tolerance
% a15 = rad2deg(pi/8); % crossing aspect limit

%% Calculate dCPA and tCPA, check if COLREGs assessment is needed:
% [dCPA, tCPA] = ClosestApproach(vessel.eta(1:2), tracks.eta(1:2),
    vessel.eta_dot(1:2), tracks.eta_dot(1:2));
[dCPAlist, tCPAlist,pos_OS_list, pos_TS_list] = getCPAlist(vessel,
    tracks);

%Keep the lowest dCPA found, this is the only dCPA we're interested
    in
%If there should ever be multiple equally low dCPAs we are in a
    unsupported
%special case that needs more development.
dCPA = min(dCPAlist);
dCPAminlist = find(dCPAlist == dCPA);
tCPA = tCPAlist(dCPAminlist(1));
pos_OS = pos_OS_list(1:3,dCPAminlist);
pos_TS = pos_TS_list(1:3,dCPAminlist);

[TSdCPAlist, TStCPAlist, ts_pos_TS_list, ts_pos_OS_list] =
    getCPAlist(tracks,vessel);
TSdCPA = min(TSdCPAlist);
TSdCPAminlist = find(TSdCPAlist == TSdCPA);

%HACKJOB
%This is a failsafe to prevent MATLAB from throwing an error and
    halting
%the program should any of the Target Ships in the simulation be at
    their
%final destination.
if(~isempty(TStCPAlist))
    TStCPA = TStCPAlist(TSdCPAminlist(1));
    tspos_OS = ts_pos_OS_list(1:3,TSdCPAminlist);
    tspos_TS = ts_pos_TS_list(1:3,TSdCPAminlist);
else
    TStCPA = 0;
    tspos_OS = [0 0 0]';
    tspos_TS = [100 100]';
end
%END of HACKJOB

if TSdCPA < dCPA
    dCPA = TSdCPA;
    tCPA = TStCPA;
    pos_OS = tspos_OS;
    pos_TS = tspos_TS;
end

%Nå vet vi hva dCPA og tCPA er, kan nå sammenligne med en eller
```

```
    annen
    %kvantitet for å se om det er høvelig å sette COLREGs flag på TS.
    %HVIS vi ønsker å sette COLREGs flag må vi også vite hvor OS og TS
        er i
    %forhold til hverandre, og hvilke kurs begge har når vi starter på
        banen
    %som tar oss til denne dCPAen.
    OSareal = vessel.size(1)*vessel.size(2);
    TSareal = tracks.size(1)*tracks.size(2);

    dCPAgrense = (OSareal + TSareal + max(OSareal,TSareal)) / 2; % En
        eller annen funksjon av størrelser
    %Hvis problemet blir unfeasible kan det hende vi blir nødt til å
        senke
    %denne grensen, men det er en funksjon for en annen dag.

    tCPAgrense = 3 * dCPAgrense;



    %% Conduct COLREGs assessment
    if (dCPA < dCPAgrense) && (tCPA < tCPAgrense) && cflag == 0
        % Angles between OS and TS
        phi_1 = rad2deg(pi/8);
    %     phi_1 = rad2deg(pi/15);
        phi_2 = 112.5;

        b0 = rad2deg(wrapTo2Pi(atan2((pos_TS(2)-pos_OS(2)),(pos_TS(1)-
            pos_OS(1))) - wrapToPi(pos_OS(3)))); % Relative from OS to
            TS

        b0_180 = rad2deg(wrapToPi(deg2rad(b0)));

        a0 = rad2deg(ssa(atan2(pos_OS(2)-pos_TS(2),pos_OS(1)-pos_TS(1))
            - pos_TS(3))); % Relative from TS to OS

        % dist = sqrt((tracks.eta(2) - vessel.eta(2))^2 + (tracks.eta
            (1) - vessel.eta(1))^2);
        %a0_360 = rad2deg(wrapTo2Pi(deg2rad(a0)));
        %
        %
        % phi_TS = atan2((vessel.eta(2)-tracks.eta(2)), (vessel.eta(1)
            - tracks.eta(1)));
        % psi_TSR = tracks.eta(3) - vessel.eta(3) - phi_TS;
        %
        % phi_TS = wrapTo2Pi(phi_TS);
        % psi_TSR = wrapTo2Pi(psi_TSR);

        % 1 = HO
        % 2 = GW
        % 3 = SO
        % 4 = OT
        % 5 = SF
        if cflag == 0 %%
            if abs(b0_180) < phi_1 % TS is direcly ahead of OS
                if abs(a0) < phi_1 % TS is facing OS
                    flag = 1;
                elseif a0 > phi_1 && a0 < phi_2 % TS is facing towards
```

```
OS's starboard
                    flag = 3;
                elseif a0 < (-phi_1) && a0 > (-phi_2) % TS is facing towards OS's port
                    flag = 2;
                else                                % TS is facing away from OS
                    flag = 4;
                end
            elseif b0 > phi_1 && b0 < phi_2 %TS is ahead on OS's starboard
                if abs(a0) < phi_1
                    flag = 2;
                elseif a0 > phi_1 && a0 < phi_2
                    flag = 5;
                elseif a0 < (-phi_1) && a0 > (-phi_2)
                    flag = 2;
                else
                    flag = 4;
                end
            elseif b0_180 < -phi_1 && b0_180 > -phi_2 %TS is ahead on OS's port side
                if abs(a0) < phi_1
                    flag = 3;
                elseif a0 > phi_1 && a0 < phi_2
                    flag = 3;
                elseif a0 < (-phi_1) && a0 > (-phi_2)
                    flag = 5;
                else
                    flag = 4;
                end
            else
                if abs(a0) < phi_1
                    flag = 3;
                elseif a0 > phi_1 && a0 < phi_2
                    flag = 3;
                elseif a0 < (-phi_1) && a0 > (-phi_2)
                    flag = 3;
                else
                    flag = 5;
                end
            end
        else % hackjob, needs more work to clear situations properly.
        flag = cflag;
        end


        %% Woerner method
        % if b0 > 112.5 && b0 < 247.5 && abs(a0) < a13
    %       flag = 'SO';
    % elseif a0_360 > 112.5 && a0_360 < 247.5 && abs(b0_180) < a13,
    %       flag = 'GW';
    % elseif abs(b0_180) < a14 && abs(a0) < a14
    %       flag = 'HO';
    % elseif b0 > 0 && b0 < 112.5 && a0 > -112.5 && a0 < a15
    %       flag = 'GW';
    % elseif a0_360 > 0 && a0_360 < 112.5 && b0_180 < -112.5 && b0_180 < a15
```

```
    %       flag = 'SO';
    %
    % else
    %       flag = 'SO';
    else
        flag = cflag;
    end

    if dCPA > (dCPAgrense+30)
        flag = 0;
    end


    end
```