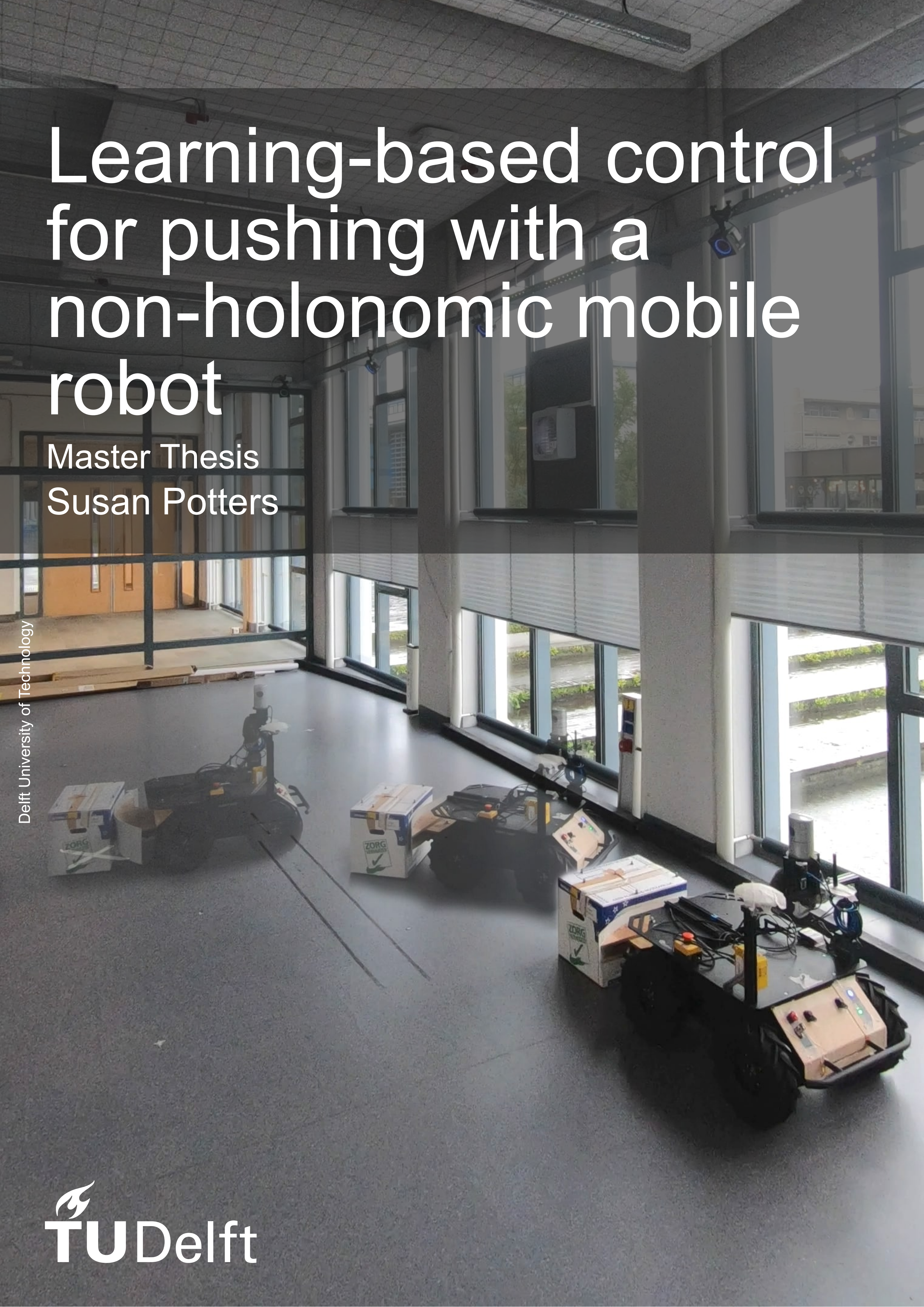


# Learning-based control for pushing with a non-holonomic mobile robot

Master Thesis

Susan Potters

Delft University of Technology

TU Delft

# Learning-based control for pushing with a non-holonomic mobile robot

by

Susan Potters

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on November 24, 2022 at 10:00 AM.

| | | |
|---|---|---|
| Student number: | 4570642 | |
| Project duration: | December 1, 2021 – November 24, 2022 | |
| Thesis committee: | Prof. Dr. Ir. M. Wisse | TU Delft, supervisor |
| | Ir. Y. Tang | TU Delft, daily supervisor |
| | Dr. J. Alonso-Mora | TU Delft |
| | Dr. L. Ferranti | TU Delft |

TU Delft

# Abstract

Mobile robots are getting more common in warehouses, distribution centers and factories, where they are used to boost productivity. At the same time, they are moving into the everyday world, where they need to operate in uncontrolled and cluttered environments. In order to extend the set of tasks that a robot can autonomously accomplish in such environments, we can equip mobile bases with a pushing skill. Possible use cases are delivering packages by pushing objects to a goal location, or clearing a path by pushing movable obstacles out of the way.

This thesis considers pushing with a non-holonomic mobile robot in cluttered environments. We develop a learning-based method, based on ensembles of probabilistic neural networks for learning the object's dynamics. The models capture the inherent stochasticity of pushing from the variability of friction. In addition, the ensembles can capture the uncertainty that arises from a lack of data, so we can reason about which regions to avoid during the pushing and do not come into unrecoverable states.

The model is combined with a Model Predictive Path Integral (MPPI) controller. Each timestep the controller optimizes for a finite time horizon, which allows us to recover from slight model inaccuracies. The MPPI controller is capable of handling complex cost functions, which allows us to include the estimated uncertainty from the model to avoid uncertain regions. For obstacle-aware pushing, we provide the MPPI controller with a costmap grid, that contains the free pushing space, which the robot and object can use to maneuver to the goal location. Both the robot and object are constrained to keep within the free pushing space, which is constructed using LiDAR measurements.

We verify the method in simulation and compare it to two baseline methods: the state-of-the-art adaptive pushing controller proposed by Krivic et al. [31] and a pushing policy trained with the model-free reinforcement learning baseline Soft Actor-Critic (SAC) [23]. All three baselines reach comparable success rates in simulation of 100%, 98% and 96%, respectively. Furthermore, the learning-based MPPI shows improvements of 50% and 23% in accuracy compared to the baselines. The results are consistent for pushing in cluttered environments, where we show an increase of accuracy by 43%, compared to [31].

Lastly, we validate the approach in a real-world setting, where the robot and object are tracked with a motion capture system. The approach has an overall success rate of 94% and performs successful pushing in a cluttered environment.

# Contents

# 1 Introduction

Mobile robots are increasingly being used in applications beyond structured environments. They are moving into the everyday world, where they need to operate in uncontrolled and cluttered human-inhabited environments. To complete tasks autonomously in these environments, robots need to be equipped with manipulation skills that can adapt to unforeseeable circumstances.

Pushing is one such a manipulation skill, that extends the set of tasks that a robot can autonomously accomplish. Objects may be too large or heavy for pick-and-place operations, or the mobile robot may not be equipped with a manipulator in order to save costs or space. In these cases, the mobile base itself can be used for pushing. Mobile robots can clear a path by pushing movable obstacles out of the way, which could be useful in rescue missions or for household robots. Another application could be in automated warehouses or storage facilities, where heavy packages need to be transported or fallen objects may block the passage for all unmanned vehicles. By quickly pushing the object out of the way with a mobile base, there would be no need for human intervention and production is not obstructed.

In this thesis, we consider pushing with the Husky UAGV (Unmanned Autonomous Ground Vehicle), a mobile robot with an add-on round bumper (Figure 1.1). It is a non-holonomic robot, since the contact between the ground and the wheels is pure rolling without sideway slipping, and therefore the robot cannot move side-by-side. The objective for the Husky is to deliver packages to a goal location in cluttered environments, by controlling the motion of the object with the round bumper.

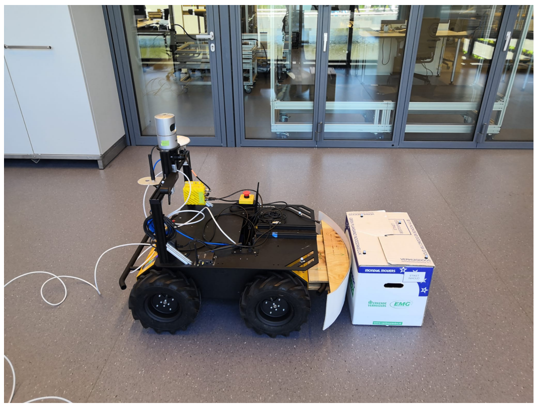


**Figure 1.1:** The Husky UAGV with a circular bumper.

## 1.1. Background

Current research into pushing focuses on modelling the object dynamics, and uses it to plan and control the object's motion during the pushing [34, 12, 2]. There are multiple options for modelling the object's dynamics, including the use of analytical, hybrid and data-driven models. While analytical models have been successfully used for pushing with a manipulator [34], they rely on assumptions that may not in the real world. In addition, they require extensive knowledge of the robot and object, including information or assumptions on the mass distribution and friction coefficients. Hybrid models alleviate this problem, as they try to estimate the unknown parameters of analytical models [49]. In contrast, recent data-driven models have proven to be more accurate than analytical models, even with limited data available [6], and have shown successful pushing in combination with control methods [12, 2].

For planning and controlling the pushing, models are typically combined with global and local planning methods. In cluttered environments, a global planning method is often used to find a feasible path [38, 1, 64, 40]. A local planner is then used to move between waypoints, while taking into account the dynamics of the pushed object [12, 29, 31]. Performance of these methods is thus dependent on the accuracy of the model, and the local planner's ability to cope with inaccuracies.
Another line of work investigates learning a pushing policy without any object model, using model-free reinforcement learning [11, 15, 10]. They reason that learning a model-free pushing policy is simpler than learning complex object dynamics. While these methods have shown successful real-world pushing with a manipulator, they require more data samples than model-based methods and are dependent on the quality of the simulation environment.

Model-based control and model-free reinforcement learning have shown successes in pushing with a manipulator and holonomic mobile bases [31, 11, 10, 12]. However, they often cannot be directly transferred to pushing with non-holonomic mobile bases. Due to the non-holonomic constraints, the robot cannot move freely around the object. This increases the difficulty of the pushing problem, as the robot cannot easily reposition itself behind the object. In addition, many works lack testing in real-world scenarios, as they focus on pushing in simulation [15, 40, 64, 35, 1, 41].
Since the Husky cannot move smoothly around the object, it is desirable to avoid repositioning maneuvers. To achieve precise real-world pushing with a non-holonomic robot, we need an accurate model of the object, in combination with a controller that can cope with slight inaccuracies. We use a learning-based control method, where the object dynamics are learned and combined with a Model Predictive Control approach.

## 1.2. Research Objective

The current state-of-the-art for pushing with a holonomic mobile base is by Krivic et al. [31], who learn local inverse models of the object-robot interaction and introduce pushing corridors for obstacle avoidance. They hypothesize that only partial knowledge of the object dynamics is needed for successful control, and rely on relocating movements if the object is pushed to an undesirable region. Repositioning is however more time-intensive for non-holonomic robots, as they cannot smoothly circle around the object. Furthermore, there is room for improvement in terms of pushing time and accuracy, since the method makes use of approximate models.

This thesis considers the problem of pushing objects to a goal location with a mobile robot. The objective is to develop a method suitable for real-world pushing with a non-holonomic base in cluttered environments. We model the object dynamics with probabilistic neural network models, and combine it with Model Predictive Path Integral (MPPI) control. For obstacle avoidance, we provide the MPPI controller with an occupancy grid containing the free pushing space that the robot and object can use to maneuver to the goal location [31]. Its performance is compared to the baseline by Krivic et al. [31] and a model-free reinforcement learning baseline, based on the success rate, accuracy, pushing time and path lengths. We also validate the approach with real-world experiments, where the robot and object are tracked with a motion capture system.

Summarizing, the main goal of this thesis is to develop a method for pushing objects to goal locations with a non-holonomic robot in real-world cluttered environments. To achieve this goal, the following objectives were realized:

- A dataset was collected in simulation, for learning object dynamics.
- The MPPI controller was implemented in combination with the trained probabilistic neural networks, to perfom pushing with the non-holonomic mobile robot in simulation.
- Two baseline methods were implemented (the adaptive controller from Krivic et al. [31] and a model-free pushing policy trained with SAC), to perform pushing in simulation and to compare them in terms of success, accuracy, pushing time and path lengths.
- A global planning method was implemented for obstacle avoidance, by providing the MPPI controller with an occupancy grid that contains the free pushing space for the robot and object.
- A dataset was collected in the motion capture lab, for learning real-world object dynamics.
- A ROS architecture was developed for pushing with the Husky in the motion capture lab.
- Pushing experiments were performed in the motion capture lab, to validate the method.

## 1.3. Structure

This thesis is structured as follows. Chapter 2 covers related work in pushing models, planning and control. Chapter 3 illustrates how we train the object model, while Chapter 4 discusses how this model is used for planning and control, which baselines we compare with, and elaborates on the pushing experiments. Chapter 5 provides results on the pushing experiments in simulation and the real world. Lastly, we discuss and conclude the results of the thesis in Chapter 6.

# 2 Related Work

Without grasping an object to affix it to a manipulator, accurate manipulation is hard to achieve. Since the object is not firmly grasped during pushing, the object can roll, slide and break contact freely with the robot. This nonprehensile manipulation is underactuated: there are more degrees of freedom than actuators. Specifically, the number of actuators of the nonholonomic robot do not match the degrees of freedom of the manipulated object. To perform pushing manipulation, the object dynamics need to be taken into account, as well as the robot and object geometries and friction [36].

Humans make internal models of physical interactions, as our brains work as simulators: they enable us to predict outcomes of interactions during pushing, pulling and grasping [17, 7]. During our lives, we accumulate experience in manipulation tasks and improve our internal models, while we can also react smoothly on mistakes by correcting our motions. Robots do not have the same lifelong experience and therefore it is hard to accurately predict the object's dynamics, or recover from modelling mistakes.

Current research therefore focuses on modelling the object's behaviour when pushed, and using this information to plan and control the push manipulation. For modelling the object dynamics, various types of models exist: *analytical*, *hybrid* and *data-driven* models (Section 2.1). To control the object, these models can be used in combination with a *classical* planning and control method (Section 2.2.1). Alternatively, model-free *reinforcement learning* algorithms can be used to learn a pushing policy (Section 2.2.2).

## 2.1. Push Manipulation Models

The non-holonomic mobile base constrains the motion of the robot in the manner that certain goal locations can be reached. As a result, we would like the robot to stay in contact with the pushed object, as contact loss can result in elaborate and time-intensive repositioning maneuvers. In order to avoid contact loss, we need an accurate model of the object behaviour, that is applicable in real-world conditions. We discuss analytical, hybrid and data-driven models, and their suitability for our application.

**Analytical models**

Early studies [37, 34] on planar pushing use analytical models, derived from classical mechanics. By incorporating the quasi-static assumption [39], which assumes that objects' velocities are small enough that inertial forces can be neglected, the problem is simplified. Analytical models do not take the stochasticity of pushing into account [62]. They also require knowledge about object properties, such as the mass, mass distribution, and friction, which can be hard to measure and limit the applicability to real-life scenarios.

A first study done by Mason [37] assumes the mentioned quasi-static assumption, and analyzes the sense of rotation of an object when pushed. Given the contact point, contact normal, friction coefficient between pusher and object, push direction and center of friction, the general rotation direction of the object could be determined. This method was called the 'voting theorem'. However, to find the exact motion of the object, the pressure distribution also needs to be considered.
Goyal et al. [20] showed that using the limit surface, relations between the friction load and motion of a sliding object can be found. The general idea is that we can create a mapping between the instantaneous velocity of a sliding object and the corresponding frictional forces, and find out how an object will slip when a force is applied. The mapping process between the forces and velocities gives the limit surface. However, the pressure distribution here is assumed known, limiting the use of the model in real-life scenarios, since the pressure distribution is generally unknown and might change during object motion.
Lee and Cutkosky [32] built upon this by approximating the limit surface of a point contact as an ellipsoid, without assumptions on the pressure distribution. Using the ellipsoid approximation, Lynch et al. [34] introduced an open-loop push controller using tactile feedback, yielding good results for quasi-static point pushing. While they showed that an analytical model can be successfully used for pushing with a manipulator, they require extensive information of the pusher and object, including friction coefficients, mass, mass distribution and dimensions. It limits the applicability to real-world pushing, as the friction coefficients and mass distribution are difficult to measure. Moreover, the assumptions that analytical models rely on, such as the quasi-static assumption, may not hold during real-world pushing and cause inaccuracies.

**Hybrid models**
Hybrid models alleviate some of the issues of analytic models, as they estimate parameters of analytical models, that would otherwise have to be assumed or known. In contrast, they are still based on assumptions, which possibly do not hold in the real world.
Sabbagh Novin et al. [49] introduced a hybrid model, that uses Bayesian regression to estimate model parameters from force and motion data. These parameters are implemented in the analytical model, which is used in combination with an MPC for planning push actions. As they consider legged objects, they first grasp the object before pushing, to ensure contact.
Adjusting online to novel objects was included by Gao et al. [19]. A fully connected neural network was trained from pre-collected data and a low-dimensional analytical parameter is learned online from outputs of an analytical model. This enables the neural network to adapt online to novel objects.

**Data-driven models**
Alternatively, data-driven models are being used to capture pushing dynamics, learning the model only from data. These models have the potential to outperform analytical models, even with as few as a hundred data samples [6].
Deep learning architectures, such as Long Short Term Memory (LSTM) Networks [12], Mixture Density Networks [2] and Graph Neural Networks [48], have shown accurate results that can be used for planning and control on real robot manipulators.
Cong et al. [12] used a LSTM Network to fit the object dynamics, after collecting data in a physics simulator. They employ domain randomization to close the sim-to-real gap, so they train on many different objects to achieve successful pushing in the real world. Arruda et al. [2] incorporate uncertainty in their predictions by Mixture Density Networks and use it for planning robust paths. In contrast, Paus et al. [48] focus on explainability of their model and use a Graph Neural Network.

Data-driven models have proven to be more accurate than analytical models [6], and have shown successful learning of dynamics and pushing in real-world environments with manipulators. Since the non-holonomic robot cannot easily reposition itself behind the object, it is important that the object's trajectories are modelled as accurately as possible, which data-driven models show the most promise for.

## 2.2. Planning and Control for Push Manipulation

Using the object dynamics model, we can generate pushing plans with classical planning and control methods. Alternatively, a model-free reinforcement learning approach can be used to learn a pushing policy, while also taking into account the layout of the environment for obstacle avoidance.

### 2.2.1. Classical Planning and Control

Classical planning and control methods are generally divided into a global and local planning part. First, the global planner finds a feasible path, that avoids collisions. Next, the local planner is used to follow the global path, while also taking into account the object's dynamics for pushing. We discuss works that provide improvements in local and global planning for pushing.

**Global Planning**

Global planning methods include graph-search and [38] and sampling-based algorithms, such as Rapidly-exploring Random Trees (RRT) [64]. Both methods try to find a feasible path, consisting of waypoints from the initial location to the goal and need information on the full environment.

As an example of graph-based search, Mason and Lynch [35] were able to reorient and translate objects simultaneously, using the concept of stable pushing in combination with best-first search (using a variation of Dijkstra). When pushing with a point contact, the resulting motion is usually not predictable and we do not know if the object will stick or slide. However, for a line contact that has two or more contact points, a predictable sticking contact can be established. It is possible to find pushing actions where the object remains fixed to the pusher, which is called stable pushing. They introduced a planner that finds stable pushing paths among multiple obstacles by best-first search. Specifically, they choose a discrete set of possible line contacts for pushing, and calculate the set of stable pushing directions from these contacts. These directions are then used in the graph-search, after which open-loop plans were generated and executed. This approach was extended for the case of a robot pushing a disk-shaped object with a point contact by Agarwal et al. [1].

Rapidly-exploring Random Trees is a samping-based algorithm, that finds a path to the goal by randomly sampling nodes. The probabilistic motion planning algorithm is used by Miyazawa et al. [41] for sampling fingertip locations, to obtain feasible paths relatively quick. However, due to the probabilistic nature of the planner, the generated plans can lead to sub-optimal behaviour. Zito et al. [64] used RRT as a global planner, that generates a set of nodes. A local planner using depth-first search in the action space of the robot, is then used for low-level control between these nodes. A physics simulator is used as a forward model, to evaluate the outcome of a pushing action. It finds locally appropriate pushes to reach the next node, as found by the global planner. Mericli et al. [40] extend RRT with experience from their probabilistic model, which is based on observing and memorizing the motion characteristics of an object. Observed trajectories are used as building blocks for extending the tree towards the goal, so the resulting path is safe and achievable. However, all these works lack real-world experiments, that validate the method developed in simulation.

**Local Planning**

Alternatively, local planning methods can better handle uncertainties in the environment, but are more complex to design, require more memory and may need a learning phase. Examples that were found in pushing literature, that focus on local push planning, include adaptive controllers [31], Model Predictive Control [8, 29, 12, 2] and Potential Fields [30, 28].

The Artificial Potential Field method was implemented on a robot arm [30], to achieve real-time low level control in cluttered and evolving environments. Obstacles are considered repulsive, while the goal is assigned an attractive force. The robot can then move towards the target, while avoiding obstacles. Similarly, Igarashi et al. [28] used a dipole field for obstacle avoidance for a simple mobile robot. Dipole-like fields guide the robot to stay behind the object and bring it to a goal. A drawback of this approach is that the robot may start orbiting around the object.

Krivic et al. [31] introduce the concept of pushing corridors to guarantee collision-free paths. The corridor consists of the free space that can be used for pushing. A local adaptive pushing controller is used, which consists of a feedback and feedforward term. Object behaviour is collected online to learn an inverse model of the interaction between the robot and object. Based on this model, the feedback term is adjusted for constant deviations in object movement directions, that can be caused by non-uniform distributions of friction or mass. The controller is used for pushing objects inside the pushing corridors to the goal location. It is the current state-of-the-art for pushing objects in a cluttered environment with a mobile base, but their approach considers a holonomic robot and uses a relatively simple model.

Model Predictive Control (MPC) methods can optimize for the current timestep, while taking into account the next timesteps and constraints [29, 8, 49]. By re-planning every timestep, inaccuracies from the model can be compensated. For example, Bertoncelli et al. [8] constrain the robot's velocity to avoid slippage of the object and Zarei et al. [29] track a path among obstacles while eliminating the disturbances from inaccurate friction estimation. These works use Model Predictive Control in combination with an analytical model, while [5] linearizes the motion equations obtained from Gaussian Processes along a nominal trajectory for use with a MPC. These methods generally find a global path first, and track this path with the MPC, while taking account uncertainties.
Cong et al. [12] used a neural network dynamics model, combined with a Model Predictive Path Integral (MPPI) controller. This model-based approach is compared with a model-free reinforcement learning algorithm, where Deep Deterministic Policy Gradient (DDPG) is used to learn a pushing policy. They find that the model-based method performs better than model-free reinforcement learning in terms of success rate. A similar approach was taken by Arruda et al. [2]: a learned forward model was combined with MPPI control to perform a pushing task with a robot arm. However, both methods focus solely on pushing with their local planners, not taking into account cluttered environments.

We discussed several classical planning and control methods, consisting of global and local planners. Many works focus on pushing in simulation, and lack extensive testing in real-world conditions [40, 64, 35, 1, 41]. The current state-of-the-art for real-world pushing with a mobile robot is the adaptive controller from Krivic et al. [31], but it considers holonomic robots, that can freely move around the object and thus needs a less accurate model. In contrast, Model Predictive Control methods in combination with complex neural networks have shown accurate pushing in real-world conditions [12], but have not been used in the context of pushing with mobile robots in cluttered environments.

### 2.2.2. Reinforcement Learning

Lastly, instead of learning a dynamics model, some studies take a model-free reinforcement learning approach to learn a pushing policy. This method can also work as a local planner for obstacle avoidance [15], and could potentially perform end-to-end planning, unifying global and local planners [25].

Clavera et al. [10] learn a pushing policy in simulation with reinforcement learning (TRPO). They explore different reward functions in simulation. Since using a sparse objective results in slow learning, they guide their policy by non-sparse objectives and reduce the weights of those terms as learning progresses. On the real manipulator, the method reaches a success rate of 70% for pushing objects from various initial positions and orientations.
Cong et al. [11] use Soft Actor-Critic (SAC) for training a pushing policy in simulation. They combine vision data into their approach and train a Variational Autoencoder (VAE) to extract task-relevant parts of the image. The output of the VAE is then combined with state information, and used for training the policy with domain randomization. The method works in a real-world environment without additional finetuning, and is robust to pushing unseen, but similar, objects.
Dengler et al. [15] also use a VAE, coupled with model-free reinforcement learning for better scene understanding. While the above methods solely focus on pushing, their approach is also focused on obstacle avoidance and works as a local planner. They employ curriculum learning, so the robot learns obstacle avoidance tasks with increasing difficulty.

Although the discussed model-free reinforcement learning methods show good results, they are generally more data-intensive than model-based methods, require domain generalization to work in real-world environments, and have only been used for pushing with manipulators. Furthermore, performance in the real-world depends on the quality of the simulation environment. If there is a big gap between the simulation and the real world, the policy will not work directly, without additional sim-to-real strategies or real-world training.

## 2.3. Discussion

The discussed methods have been widely used in the field of pushing with robot arms. However, they often cannot be directly transferred to pushing with non-holonomic mobile bases, since the robot cannot move freely around the object. Most similar to this work are the pushing solutions from [2] and [31]. Arruda et al. [2] include uncertainty in their approach for modelling the object dynamics with Mixture Density Networks and combine it with a MPPI controller. However, their solution was implemented for a manipulator and does not work in cluttered or real-time environments. Krivic et al. [31] focus on real-life pushing with a mobile base, by introducing pushing corridors to guarantee collision-free paths. In addition, they learn local inverse models of the object-robot interaction for local planning. However, their approach is only suitable for holonomic bases: their local inverse models are simple and based on the error of the object movement direction, often resulting in cases where the robot needs to reposition itself behind the object. They use a holonomic robot, that can freely move around the object, but repositioning is not as time-inefficient for non-holonomic robots.

Due to the non-holonomic mobile base we use, we would like to avoid repositioning as it is time intensive. To achieve this, we need to use an accurate model and combine it with a controller that can cope with inaccuracies and constraints. Based on the literature, we find that data-driven models have shown to be more accurate than analytical models and MPC controllers can cope with constraints and disturbances. We choose to compare to two baselines: the state-of-the-art pushing controller by Krivic et al. [31] and a model-free reinforcement learning baseline.

# 3 Dynamics

This chapter discusses the robot dynamics in Section 3.1 and how we learn the object dynamics from data in Section 3.2. We discuss data collection and processing in the simulation and the motion capture lab in Section 3.3. Section 3.4 elaborates on choosing the hyperparameters for training the final models.

## 3.1. Robot Dynamics

Figure 3.1 shows a topview of the Husky with a circular bumper and a rectangular object. The coordinate frames are given by $W$, $R$ and $O$, for the world, robot and object frame. The general dynamics of a non-holonomic robot are described by the following equations:

$$\begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \\ \dot{v}_r \\ \omega_r \end{bmatrix} = \begin{bmatrix} v_r \cos\theta_r \\ v_r \sin\theta_r \\ \omega_r \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ a_\text{r} \\ \xi_r \end{bmatrix} \tag{3.1}$$

Where $\mathbf{x}_r = [x_r, y_r, \theta_r, v_r, \omega_r]^T \in \mathbb{R}^5$ is the robot's state and $\mathbf{u}_r = [a_r, \xi_r]^T \in \mathbb{R}^2$ is the robot's action in the world frame. Its state is described by the position $[x_r, y_r]^T$, angle $\theta_r$, linear and angular velocity $[v_r, \omega_r]^T$. The robot's action $\mathbf{u}_r$ is described by the linear and angular accelerations.

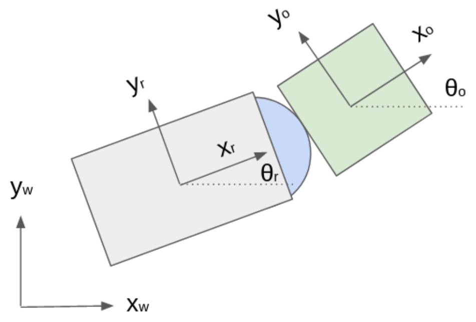


**Figure 3.1:** Husky and object with respective coordinate frames and positive angles indicated.

## 3.2. Object Dynamics

The object dynamics are learned from data with probabilistic neural networks [9], which are discussed in Section 3.2.1. The details of the learning process are then discussed in Section 3.2.2.

### 3.2.1. Neural Network Architecture

Standard neural networks are deterministic and do not account for model or data uncertainty. Models for regression are trained with the Mean Squared Error (MSE) loss, which is defined as the squared difference between the true and predicted values:

$$L_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3.2}$$

Where $y_i$ is the training target and $\hat{y}_i$ is the predicted value by the neural network.

However, a deterministic neural network cannot capture the stochasticity of pushing, and can start to overfit when there is a limited amount of data. To capture the stochasticity of pushing, which is also called *aleatoric* uncertainty, we use probabilistic neural networks [9]. Moreover, the uncertainty from a lack of data is called *epistemic* uncertainty, and is captured with ensembles of neural networks.

The probabilistic neural networks predict parameters of a parameterized (Gaussian) distribution with a neural network and thus model the aleatoric uncertainty. The loss function used for training is the Gaussian negative log-likelihood, which is described by:

$$L_{GLL} = \sum_{i=1}^{N} \text{log}(\sigma^2(s_i)) + \frac{(y_i - \mu^2(s_i))}{\sigma^2(s_i)} \tag{3.3}$$

Where $s_i$ is the input state of the neural network and $y_i$ is the training target. $\mu(s_i)$ and $\sigma(s_i)^2$ denote the mean and variance as predicted by the neural network.

Epistemic uncertainty arises from lack of data, and vanishes with unlimited data. It can be captured with accurate Bayesian neural network inference [44], but approximate methods are simpler and faster to train [18]. The epistemic uncertainty can also be captured with ensembles of neural networks, where the neural networks all have different random weight initializations and use a different batch of training data every epoch [9]. This method is simpler, needs no additional hyperparameter tuning, and is therefore used for modelling the object dynamics.

For modelling both types of uncertainty, the methods can be combined into an ensemble of $M$ probabilistic neural networks (Figure 3.2) with different weight initializations, from which the mean (Equation 3.4) and variance (Equation 3.5) can be calculated from all networks and input **s** by [57]:

$$\hat{\mu}(\mathbf{s}) = \frac{1}{M}\sum_{m=1}^{M}\mu_m(\mathbf{s}) \tag{3.4}$$

$$\hat{\sigma}^2(\mathbf{s}) = \frac{1}{M}\sum_{m=1}^{M}(\sigma_m^2(\mathbf{s}) + \mu_m^2(\mathbf{s})) - \hat{\mu}^2(\mathbf{s}) \tag{3.5}$$

We learn the object dynamics with ensembles of neural networks, where each probabilistic neural network outputs a Gaussian distribution $N(\mu(\mathbf{s}^t), \sigma^2(\mathbf{s}^t))$. The mean $\hat{\mu}(\mathbf{s}^t)$ and variance $\hat{\sigma}^2(\mathbf{s}^t)$ of these predictions are calculated with Equations 3.4 and 3.5, respectively.

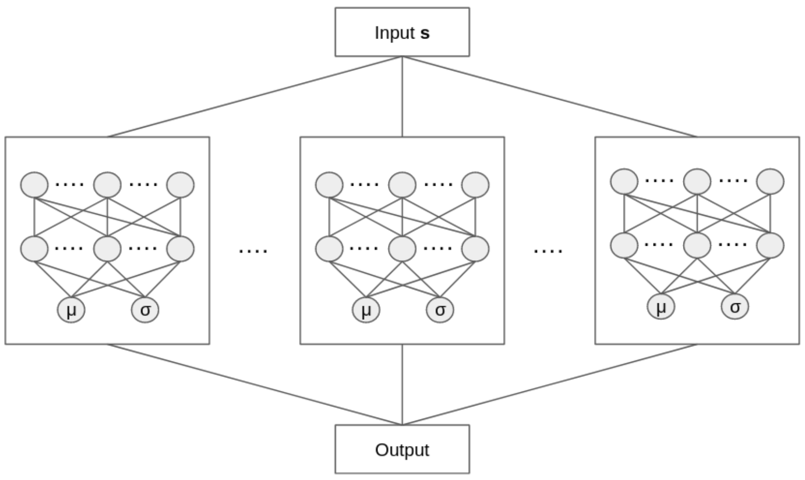


**Figure 3.2:** Ensemble of $M$ probabilistic neural networks, with input **s** and outputs $\hat{\mu}$ and $\hat{\sigma}$ .

### 3.2.2. Learning the Object Dynamics

The object state is described by $\mathbf{x}_o = [x_o, y_o, \theta_o, v_{xo}, v_{yo}, \omega_o]^T \in \mathbb{R}^6$. Here, $[x_o, y_o]^T$ and $[v_{xo}, v_{yo}]^T$ are the object's position and velocity in the world frame, respectively. $\theta_o$ and $\omega_o$ represent the object's angle and angular velocity.

Instead of trying to describe the object dynamics with an algebraic model, we use the probabilistic neural networks with parameters $\theta$ to learn the motion. Specifically, we learn the state difference between $\mathbf{x}_{o,t+1}$ and $\mathbf{x}_{o,t}$, so we can update the object state recursively. Learning the state difference, instead of the next state directly, means that the neural network is less likely to overfit on the world coordinates used for training. In addition to this, learning state differences is easier when $\mathbf{x}_{o,t+1}$ and $\mathbf{x}_{o,t}$ are very similar and a small timestep is used [43]. The discrete-time dynamics of the object are described by:

$$\mathbf{x}_{o,t+1} = \mathbf{x}_{o,t} + \dot{\mathbf{x}}_{o,t}\Delta t \tag{3.6}$$

Where we learn the second part of the equation, that updates the state:

$$\mathbf{x}_{o,t+1} = \mathbf{x}_{o,t} + f_\theta(\mathbf{x}_{r,t}, \mathbf{x}_{o,t}, \mathbf{u}_{r,t}) \tag{3.7}$$

$\mathbf{s}^t$ denotes the input state of the neural network, which consists of the current robot $\mathbf{x}_{r,t}$ and object $\mathbf{x}_{o,t}$ state, as well as the action $\mathbf{u}_{r,t}$. Since the real robot can only receive velocity commands, we choose to convert the acceleration to a velocity action for learning in simulation. In world coordinates, the full state for learning would be:

$${}^W\mathbf{s}_t = [{}^W\mathbf{x}_{r,t}, {}^W\mathbf{x}_{o,t}, {}^W\mathbf{u}_{r,t}]^T \tag{3.8}$$

However, learning the object dynamics in the world frame decreases performance, since the robot and object positions in the world frame make it harder for the network to generalize the motion to other positions in the world frame. Therefore, we use learning in the object frame for increased accuracy and generalization:

$${}^O\mathbf{s}_t = [{}^O\mathbf{x}_{r,t}, {}^O\mathbf{x}_{o,t}, {}^O\mathbf{u}_{r,t}]^T \tag{3.9}$$

Where we transform from the world to the object frame with:

$$
{}^{O}\mathbf{x}_{r,t} = \begin{bmatrix} {}^{O}x_{r,t} \\ {}^{O}y_{r,t} \\ {}^{O}\theta_{r,t} \\ {}^{O}\dot{x}_{r,t} \\ {}^{O}\dot{y}_{r,t} \\ {}^{O}\omega_{r,t} \end{bmatrix} = \begin{bmatrix} ({}^{W}x_{r,t} - {}^{W}x_{o,t})\cos(\theta_{o,t}) + ({}^{W}y_{r,t} - {}^{W}y_{o,t})\sin(\theta_{o,t}) \\ -({}^{W}x_{r,t} - {}^{W}x_{o,t})\sin(\theta_{o,t}) + ({}^{W}y_{r,t} - {}^{W}y_{o,t})\cos(\theta_{o,t}) \\ \theta_{r,t} - \theta_{o,t} \\ {}^{W}\dot{x}_{r,t}\cos(\theta_{o,t}) + {}^{W}\dot{y}_{r,t}\sin(\theta_{o,t}) \\ -{}^{W}\dot{x}_{r,t}\sin(\theta_{o,t}) + {}^{W}\dot{y}_{r,t}\cos(\theta_{o,t}) \\ \omega_{r,t} - \omega_{o,t} \end{bmatrix} \tag{3.10}
$$

$$
{}^{O}\mathbf{x}_{o,t} = \begin{bmatrix} {}^{O}\dot{x}_{o,t} \\ {}^{O}\dot{y}_{o,t} \\ {}^{O}\omega_{o,t} \end{bmatrix} = \begin{bmatrix} {}^{W}\dot{x}_{o,t}\cos(\theta_{o,t}) + {}^{W}\dot{y}_{o,t}\sin(\theta_{o,t}) \\ -{}^{W}\dot{x}_{o,t}\sin(\theta_{o,t}) + {}^{W}\dot{y}_{o,t}\cos(\theta_{o,t}) \\ {}^{W}\omega_{o,t} \end{bmatrix} \tag{3.11}
$$

$$
{}^{O}\mathbf{u}_{r,t} = \begin{bmatrix} {}^{O}a_{rx,t} \\ {}^{O}a_{ry,t} \\ {}^{O}\xi_{r,t} \end{bmatrix} = \begin{bmatrix} {}^{W}a_{rx,t}\cos(\theta_{o,t}) + {}^{W}a_{ry,t}\sin(\theta_{o,t}) \\ -{}^{W}a_{rx,t}\sin(\theta_{o,t}) + {}^{W}a_{ry,t}\cos(\theta_{o,t}) \\ {}^{W}\xi_{r,t} \end{bmatrix} \tag{3.12}
$$

We retrieve the predicted state difference in the object frame from the ensemble of $M$ neural networks:

$$
\hat{\mu}({}^{O}\Delta\mathbf{x}_{o,t+1}|{}^{O}\mathbf{s}_t) = \frac{1}{M}\sum_{m=1}^{M}\mu_m({}^{O}\Delta\mathbf{x}_{o,t+1}|{}^{O}\mathbf{s}_t;\theta_m) \tag{3.13}
$$

Next, we transform the predicted state difference back to the world frame, and add it to the previous state:

$$
{}^{W}\Delta\mathbf{x}_{o,t+1} = \begin{bmatrix} {}^{W}\Delta x_{o,t+1} \\ {}^{W}\Delta y_{o,t+1} \\ {}^{W}\Delta\theta_{o,t+1} \\ {}^{W}\Delta\dot{x}_{o,t+1} \\ {}^{W}\Delta\dot{y}_{o,t+1} \\ {}^{W}\Delta\omega_{o,t+1} \end{bmatrix} = \begin{bmatrix} {}^{O}\Delta x_{o,t+1}\cos(\theta_{o,t}) - {}^{O}\Delta y_{o,t+1}\sin(\theta_{o,t}) \\ {}^{O}\Delta x_{o,t+1}\sin(\theta_{o,t}) + {}^{O}\Delta y_{o,t+1}\cos(\theta_{o,t}) \\ {}^{O}\Delta\theta_{o,t+1} \\ {}^{O}\Delta\dot{x}_{o,t+1}\cos(\theta_{o,t}) - {}^{O}\Delta\dot{y}_{o,t+1}\sin(\theta_{o,t}) \\ {}^{O}\Delta\dot{x}_{o,t+1}\sin(\theta_{o,t}) + {}^{O}\Delta\dot{y}_{o,t+1}\cos(\theta_{o,t}) \\ {}^{O}\Delta\omega_{o,t+1} \end{bmatrix} \tag{3.14}
$$

$$
{}^{W}\mathbf{x}_{o,t+1} = {}^{W}\mathbf{x}_{o,t} + {}^{W}\Delta\mathbf{x}_{o,t+1} \tag{3.15}
$$

Doing this recursively, we can retrieve the object's position for a length of time. Figure 3.3 depicts the process of retrieving data, inputting it in the model and predicting the state difference.

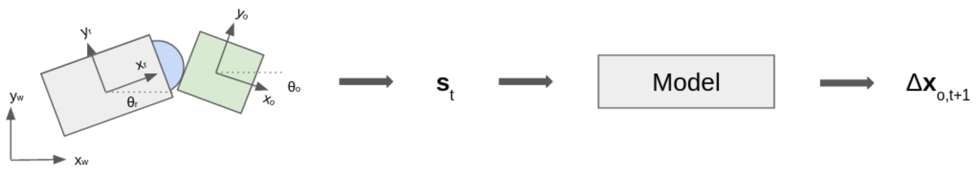


**Figure 3.3:** Process of retrieving data from the system, converting it to the input state, inputting it in the neural network and retrieving the state difference.

## 3.3. Data Collection

For the neural network to learn the object dynamics, we need data of the robot-object interactions. Section 3.3.1 discusses the data collection method in simulation, while Section 3.3.2 discusses the method used in the motion capture lab.

### 3.3.1. Simulation

The simulation we use is from Tang et al. [55] and uses Differential Algebraic Equations (DAE) to compute the object displacement. We choose this simulation environment over Gazebo or Isaac Gym [33], as it is not a black box and gives us insight in the underlying dynamics.

For collecting robot-object interaction data in simulation, we place the box-shaped object in the environment with workspace $ws$ at a randomly initialized position and angle (Table 4.2). We position the robot and object to be in contact. The box used for data collection is 0.48 x 0.32 x 0.32 m (L x W x H), has a weight of 4 kg and a friction coefficient of 0.3.

**Table 3.1:** Minimum and maximum values for randomized variables during data collection.

| | Minimum value | Maximum value |
|---|---|---|
| ${}^W x_o$ | $-ws_x$ | $ws_x$ |
| ${}^W y_o$ | $-ws_y$ | $ws_y$ |
| ${}^W \theta_o$ | atan2$({}^W y_o, {}^W x_o) - 30°$ | atan2$({}^W y_o, {}^W x_o) + 30°$ |
| ${}^W \theta_r$ | ${}^W \theta_o - 5°$ | ${}^W \theta_o + 5°$ |
| ${}^O y_r$ | $-\frac{1}{2} L_{box}$ | $\frac{1}{2} L_{box}$ |

The MPPI controller is used for navigating the robot, instead of the object, to goal location $[0.0, 0.0]$ m. Since the robot navigates to the goal location, the robot and object will eventually lose contact. If contact is lost, we stop the current simulation and reset the environment. This data collection method was chosen, since it is simple and gives us meaningful interaction data, that is sufficient for learning the object model. We save the robot state $\mathbf{x}_r$, object state, $\mathbf{x}_o$ and robot action $\mathbf{u}_r$ every timestep of 0.1 s, and collect 8440 data samples (the same as collected for the real-world experiments).

### 3.3.2. Motion Capture

Data collection for real-world pushing is done in a lab (Figure 3.4) equipped with a OptiTrack Motion Capture System (mocap). Reflective markers are attached to the robot and object, which are used for tracking their 3D positions and orientations at 120 Hz with infrared technology [47]. To ensure that the mocap system can always construct a 3D rigid body, we attach at least three markers to the robot and object, and make sure they form an unique shape. From the mocap system, the marker's locations are communicated to a desktop with the Motive software [46], after which we use the ROS package mocap_optitrack [21] to translate the data to a ROS format.

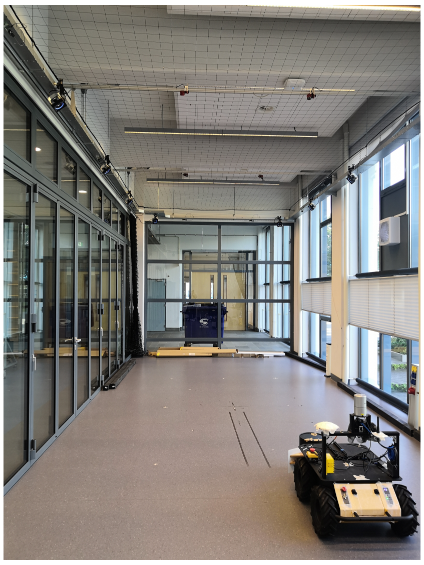


**Figure 3.4:** Lab with the Motion Capture System.

After we collect the positions from the mocap_optitrack node, we retrieve the robot's and object's velocity from the bebop2_state_estimator package [63], that uses an Unscented Kalman Filter (UKF). To retrieve the control commands that are sent to the Husky, we subscribe to its velocity controller. A data_saver node collects the robot state, object state and robot action every 0.1 seconds and saves them. Figure 3.5 illustrates the whole process of collecting data.

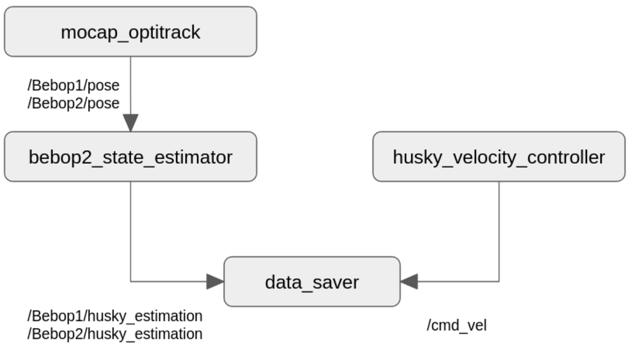


**Figure 3.5:** Scheme of all ROS packages used for data collection.

We manually control the robot to perform pushes for collecting the data. To ensure we create meaningful data that captures different pushing conditions, we vary the initial angle of the object relative to the robot, and the goal location. The pushed object is a paper box with size 0.48 x 0.32 x 0.32 m (L x W x H) and a weight of 4.0 kg. The bounds on the linear and angular speed during the experiment are $[0.0, 0.4]$ m/s and $[-1.0, 1.0]$ rad/s, respectively.

**Data Processing**

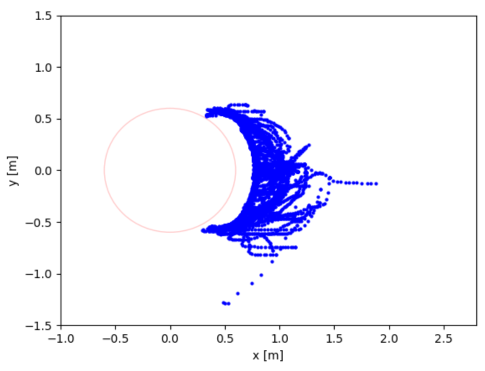
After collecting the data, we process it before training our object dynamics model. First, there are instances in the data where the robot and object are not in contact, when the object slipped away from the bumper during the pushing or when the robot approaches the object. These need to be filtered out, which is done by setting a distance threshold. We calculate the distance between the robot and object $d_{ro}$ and check if there is contact:

$$d_{ro} < \frac{L_{husky}}{2} + \frac{W_{box}}{2} + \epsilon \tag{3.16}$$

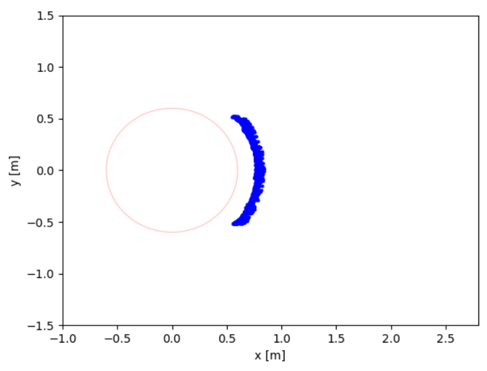
Where $L_{husky}$ is the robot's length and $L_{box}$ the width of the box. If the contact point is at the center of the box, the distance between the robot and object would be $\frac{L_{husky}}{2} + \frac{W_{box}}{2}$. However, other contact configurations are possible, the motion capture system is not perfectly accurate and the tracked points on the rigid bodies were not perfectly in the center. Therefore, we introduce a slacking parameter $\epsilon$, which was set by looking at the data. To further ensure the robot and object have not lost contact, we check the y-position of the box relative to the robot, which should be lower than the bumper's width $W_{bumper}$:

$$|{}^{R}y_o| < \frac{W_{bumper}}{2} + \epsilon \tag{3.17}$$

Figure 3.6 illustrates the relative positions of the object, in the robot frame. Before data processing, the object and robot are not always in contact. We filter these instances out, and are left with data points where the robot and object are in contact.



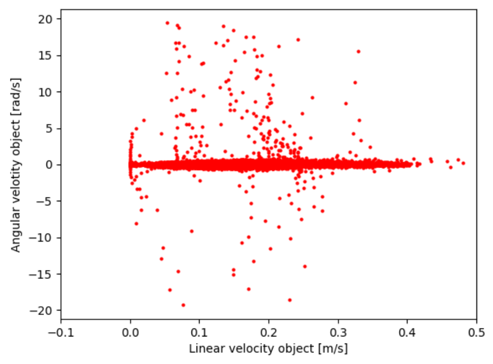
**(a)** Object positions in the robot frame, before preprocessing.



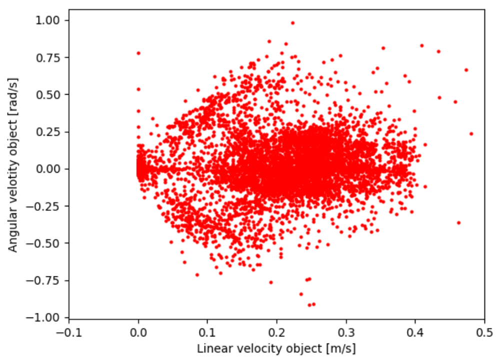
**(b)** Object positions in the robot frame, after preprocessing.

**Figure 3.6:** Object location relative to the robot, before and after data preprocessing. The red circle illustrates the robot, while the blue dots illustrate the relative object positions, in the robot frame.

In addition, there were some high peaks observed in the obtained angular velocity from the UKF filter, that are not physically possible. When the robot abruptly starts driving, the detected velocity is sometimes much higher than the maximum speed of the robot. These moments were removed from the data as well, by setting a velocity threshold. Figure 3.7 and 3.8 show the linear and angular velocities of the object and robot, respectively, before and after preprocessing. It is clear that there are outliers in the angular velocity, with velocities up to 20 rad/s. These are removed with the velocity threshold, after which plausible velocities remain.

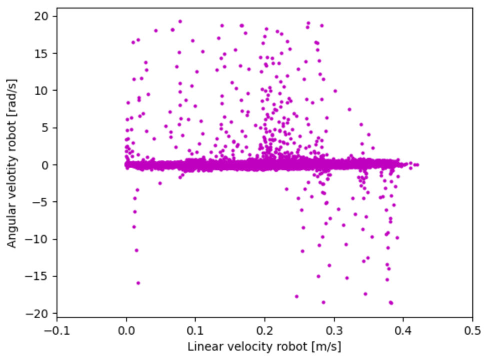


**(a)** Linear and angular velocity of the object, before data preprocessing.

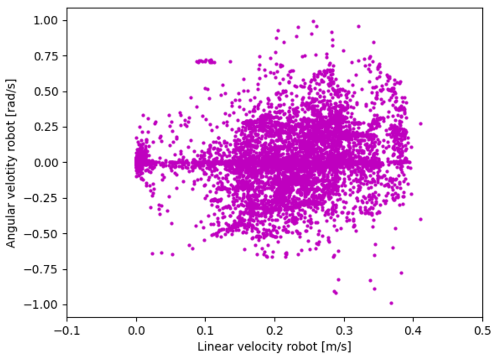


**(b)** Linear and angular velocity of the object, after data preprocessing.

**Figure 3.7:** Velocity data of the object before and after data preprocessing.



**(a)** Linear and angular velocity of the robot, before data preprocessing.



**(b)** Linear and angular velocity of the robot, after data preprocessing.

**Figure 3.8:** Velocity data of the robot before and after data preprocessing.

## 3.4. Hyperparameter Optimization

Before using the model for pushing, we perform hyperparameter optimization. We use an ensemble of probabilistic ensembles, as described in Section 3.2.1. From the collected data, we use 70% for training, 10% for validation, 20% for testing and choose the model with the lowest validation error. Table 3.2 contains the hyperparameter values that were tried. Searching for the optimal combination was done in a subsequent manner: we vary the values of one parameter, see which one reaches the lowest error, freeze the parameter and reiterate for the next parameter.

**Table 3.2:** Hyperparameter values used for optimization

| Hyperparameter | Values |
|---|---|
| Neurons | 128, 256 |
| Batch size | 32, 64 |
| Learning rate | 0.0001, 0.0005 |
| Layers | 3, 4 |
| Number of ensembles | 3, 6, 12 |
| Non-linear layers | ReLU, sigmoid |

In addition to finding the most optimal hyperparameter combination used for the probabilistic ensembles, we also performed a rough hyperparameter optimization when comparing networks (Appendix A).

# 4 Planning and Control

This chapter discusses how the learned object dynamics are used for pushing control in Section 4.1 In Section 4.2 we describe the two baselines that we compare with. Section 4.3 provides detailed descriptions on how the pushing experiments were performed in simulation and in the motion capture lab.

## 4.1. Planning and Control with the Learned Dynamics

The learned object dynamics are combined with Model Predictive Path Integral (MPPI) control [61], which is a sampling-based Model Predictive Control method. Section 4.1.1 first briefly introduces Model Predictive Control, Model Predictive Path Integral control, and their suitability for pushing with learned dynamics. Section 4.1.2 and 4.1.3 then elaborate on the implementations for pushing in uncluttered and cluttered environments, respectively.

### 4.1.1. Background

Chapter 2 discussed the advantages of Model Predictive Control methods, and why they are suited for pushing. They optimize for the current timestep, take future timesteps into account and by replanning every timestep inaccuracies from the model can be compensated. While it is difficult to describe or learn the object dynamics accurately for multiple timesteps in the future, Model Predictive Control methods have the potential to handle these disturbances. We first introduce conventional Model Predictive Control, and discuss its suitability for pushing with learned dynamics. Then, we discuss Model Predictive Path Integral control, which is a sampling-based MPC method.

#### Model Predictive Control

Model Predictive Control (MPC) is an advanced control method used for controlling complex problems. Figure 4.1 shows the workings of an MPC, which consists of the three following steps for a robot at current timestep $k$:

1. A dynamics model predicts the states at future timesteps, for the prediction horizon $P$.
2. By minimizing a cost function and taking into account constraints, a sequence of optimal control actions is calculated for the control horizon $M$.
3. The first control action is implemented on the robot, after which the horizons are shifted one timestep into the future and we re-optimize.

By taking into account the future timesteps, MPC can anticipate future incidents and act accordingly. Another advantage is that constraints can be included into the problem in a simple manner.
Since MPC is an optimization-based method, we find a locally optimal solution for the horizon, but it is not necessarily a globally optimal solution. Also, the method can get computationally expensive, based on the complexity of the problem and its performance is heavily dependent on the accuracy of the model.

Preliminary tests showed that Model Predictive Control with the neural network dynamics was often not able to find a feasible solution in time, which ultimately leads to loss of contact between the robot and object. Therefore, Model Predictive Path Integral control was implemented. It can handle the complex neural network dynamics better, since it does not need differentiation to come to a solution.

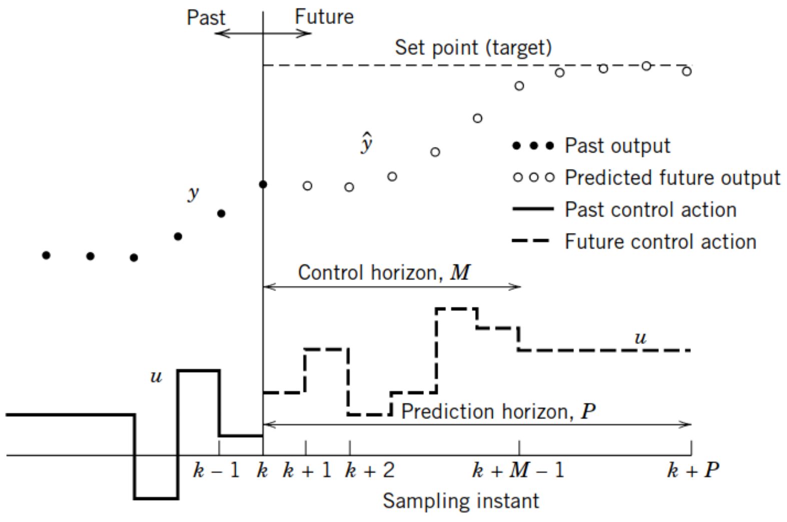


**Figure 4.1:** MPC concept [52].

**Model Predictive Path Integral Control**
Model Predictive Path Integral control is well-suited for complex problems, where differentiating the dynamics model is not computationally feasible. MPPI is a sampling-based approach and does not require differentiating to come to an optimal solution. Instead, it performs many trajectory rollouts in parallel and can thus handle non-linear dynamics and non-differentiable cost functions without any approximations.

As such, MPPI control has been used in various tasks, where the dynamics are approximated by a neural network. Williams et al. [61] use a neural network for predicting vehicle dynamics for racing purposes. Nagabandi et al. [42] train neural networks in a model-based reinforcement context for dexterous manipulation. Both combine their models with MPPI for successful real-time action selection. Lastly, various works show that MPPI can be used for systems with stochastic dynamics [43, 56].

The idea of MPPI control is based on Monte Carlo simulation and sampling as many trajectories as possible in real-time. The trajectories are compared to each other based on a predetermined cost function, to find the most optimal action.

We consider the robot and object systems with state $\mathbf{x}_t = [x_r, y_r, \theta_r, v_r, \omega_r, x_o, y_o, \theta_o, v_{xo}, v_{yo}, \omega_o]^T$, consisting of the robot and object states, which are propagated:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta\mathbf{x}_{t+1} \tag{4.1}$$

The robot states are propagated with the robot dynamics from Equation 3.1, which are then discretized with RK4. The object states are propagated with the neural network expectations, as is done in [43, 56]. Chua et al. [9] report that more advanced uncertainty propagation techniques, such as their proposed trajectory sampling, seem to offer minor improvements, while they increase computational time significantly. The state updates for the robot and object are thus defined as:

$$\Delta\mathbf{x}_{o,t+1} = \hat{\mu}(\mathbf{x}_t, \mathbf{u}_t + \delta\mathbf{v}_t) \tag{4.2}$$

$$\Delta\mathbf{x}_{r,t+1} = f_{RK4}(\mathbf{x}_{r,t}, \mathbf{u}_t + \delta\mathbf{v}_t) \tag{4.3}$$

$$\delta\mathbf{v}_t \sim N(0, \mathbf{\Sigma}) \tag{4.4}$$

Here, $\mathbf{x}_t \in \mathbb{R}^n$ denotes the state vector at timestep $t$, $\mathbf{u}_t \in \mathbb{R}^m$ denotes the control input and $\delta\mathbf{v}_t \in \mathbb{R}^m$ is a Gaussian noise vector with zero mean and $\mathbf{\Sigma}$ variance. In our case, $\mathbf{\Sigma} = [\Sigma_v, \Sigma_\omega]^T$, so we can tune the variance for the linear and angular robot action.
The goal of the optimal control problem is then to find the control sequence, for which the cost over all trajectories is minimized. The cost function is defined as:

$$C(\tau_{t,k}) = \phi(\mathbf{x}_T) + \sum_{t=i}^{T-1} q(\mathbf{x}_t, \mathbf{u}_t + \delta\mathbf{v}_t) \tag{4.5}$$

Where $\phi(\mathbf{x}_T)$ denotes the terminal cost and $q(\mathbf{x}_t, \mathbf{u}_t)$ denotes the running cost. The cost function should be minimized and thus we can write the optimal control problem as:

$$J = \min_{\mathbf{u}} \mathbb{E}[\phi(\mathbf{x}_T) + \sum_{t=i}^{T-1} q(\mathbf{x}_t, \mathbf{u}_t + \delta\mathbf{v}_t)] \tag{4.6}$$

From [60], we use importance sampling based on the cost, to give weights to all $K$ rollouts and use it to iteratively update the control sequence:

$$\mathbf{u}_t \leftarrow \mathbf{u}_t + \frac{\sum_{k=1}^{K} exp(\lambda^{-1}C(\tau_{t,k}))\delta\mathbf{v}_{t,k}}{\sum_{k=1}^{K} exp(\lambda^{-1}C(\tau_{t,k}))} \tag{4.7}$$

Where $\lambda$ is a hyperparameter to be tuned, called the temperature. Each timestep, we perform $K$ rollouts of $T$ timesteps, optimize the control sequence and execute the first command. The rest of the calculated control sequence, of length $T-1$, is used as input for the next timestep.

### 4.1.2. Implementation for Pushing

The MPPI controller uses the robot and object dynamics to propagate trajectories, and finds the most optimal robot action based on the cost function. It uses Equation 3.15 to propagate the object trajectories in a recursive manner, and propagates the robot's trajectories with the robot dynamics from Equation 3.1, which are then discretized with RK4.

The running cost of the controller is defined as:

$$\begin{aligned} q(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}, \mathbf{u}_{r,t}) = C_{vel}(\mathbf{x}_{r,t}) + C_{var}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}, \mathbf{u}_{r,t}) + C_a(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) \\ + C_{p,o}(\mathbf{x}_{o,t}) + C_{p,c}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) + C_{p,a}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) \end{aligned} \tag{4.8}$$

Which consists of a velocity penalizing cost $C_{vel}$, a variance penalizing cost $C_{var}$ and a cost that encourages the object angle to be directed towards the goal $C_a$. In addition, three costs are added that give big penalties for unwanted events. $C_{p,o}$ gives a big penalty for overshooting the goal, $C_{c,o}$ penalizes contact points that are not possible based on the object's geometry and $C_{p,a}$ penalizes too large angles between the robot and object, so the object does not slip away from the bumper. In the cost function, $\mathbf{C} = [C_1, C_2, ..., C_7]^T$ are are tuned weights. $C_{vel}$ slightly penalizes the robot's velocity, so it will not move too fast:

$$C_{vel}(\mathbf{x}_{r,t}) = C_1(v_{r,t})^2 + C_2(\omega_{r,t})^2 \tag{4.9}$$

$C_{var}$ describes the uncertainty of the object's state at timestep $t$. It therefore penalizes trajectories, where the prediction and real behaviour of the object are likely to differ much. Including this cost minimizes the risk of pushing the object in a way that the robot cannot recover from and thus produces more robust paths. It is defined as:

$$C_{var}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}, \mathbf{u}_{r,t}) = C_3\hat{\sigma}^2(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}, \mathbf{u}_{r,t}) \tag{4.10}$$

$C_a$ minimizes the difference between the current object angle, and the optimal angle towards the goal. It therefore encourages the object to be directed to the goal:

$$C_a(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) = C_4||\theta_o - \text{atan2}(y_{o,g} - y_{o,t}, x_{o,g} - x_{o,t})|| \tag{4.11}$$

Where $x_{o,g}$ and $y_{o,g}$ are the $x$ and $y$ position of the goal in the world frame, respectively. Lastly, we describe the three penalizing costs for unwanted behaviour:

$$C_{p,o}(\mathbf{x}_{o,t}) = \begin{cases} C_5 & \text{if } ||\mathbf{p}_{o,t} - \mathbf{p}_{o,g}|| - ||\mathbf{p}_{o,t-1} - \mathbf{p}_{o,g}|| > 0 \text{ and } ||\mathbf{p}_{o,t} - \mathbf{p}_{o,g}|| < 1 \\ 0 & \text{else} \end{cases} \tag{4.12}$$

Where $\mathbf{p}_{o,t}$ is the position of the object at timestep $t$ and $\mathbf{p}_{o,g}$ is the object's goal position. The overshooting cost $C_{p,o}$ gives a big penalty cost if the object's location is within 1.0 m of the goal, and the predicted motion moves the object further away from the goal. If the object's position is predicted to be outside the Husky's bumper, a big penalizing cost is given by $C_{p,c}$:

$$C_{p,c}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) = \begin{cases} C_6 & \text{if } ({}^{R}y_{o,t} - {}^{R}y_{r,t}) > \frac{1}{2}W_{husky} \\ C_6 & \text{if } ({}^{R}y_{o,t} - {}^{R}y_{r,t}) < -\frac{1}{2}W_{husky} \\ 0 & \text{else} \end{cases} \tag{4.13}$$

$C_{p,a}$ gives a big penalty if the angle between the robot and object is larger than the angle of the bumper:

$$C_{p,a}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) = \begin{cases} C_7 & \text{if } \theta_r - \theta_o < -35^\circ \\ C_7 & \text{if } \theta_r - \theta_o > 35^\circ \\ 0 & \text{else} \end{cases} \tag{4.14}$$

The terminal cost is defined as:

$$\phi(\mathbf{x}_{r,T}) = C_8||\mathbf{p}_{o,T} - \mathbf{p}_{o,g}|| \tag{4.15}$$

Where $C_8$ is a tuning weight, $\mathbf{p}_{o,T}$ is the position of the object at the last timestep $T$ and $\mathbf{p}_{o,g}$ is the object's goal position.

## 4.1.3. Implementation for Obstacle-Aware Pushing

For obstacle-aware pushing, we describe the free pushing space with the corridors from Krivic et al.[31]. They are constructed from LiDAR measurements from the robot, according to the following steps, that are visualized in Figure 4.2:

1. Receive raw sensor data from the LiDAR.
2. Create a costmap with an inflation layer from the raw sensor data.
3. Create a Generalized Voronoi Grid [54] from the free space of the costmap.
4. Find a path from start to goal along the Generalized Voronoi Grid using A* [24].
5. Find the pushing corridor by fitting circles along the found path, that intersect with the occupied space.
6. Normalize the corridor by dividing by the distance to the path.

Next, the cost function is adjusted such that the robot and object stay inside the corridor:

$$\begin{aligned} q(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}, \mathbf{u}_{r,t}) = {} & C_{vel}(\mathbf{x}_{r,t}) + C_{var}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}, \mathbf{u}_{r,t}) + C_{a,grid}(\mathbf{x}_{o,t}) \\ & + C_{p,o}(\mathbf{x}_{o,t}) + C_{p,c}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) + C_{p,a}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) + C_{grid}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) + C_{a,grid}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) \end{aligned} \tag{4.16}$$

Here we introduce the new costs $C_{grid}$ and $C_{a,grid}$:

$$C_{grid}(\mathbf{x}_{o,t}, \mathbf{x}_{r,t}) = C_9 M(x_{r,t}, y_{r,t}) + C_{10} M(x_{o,t}, y_{o,t}) \tag{4.17}$$

$$C_{a,grid}(\mathbf{x}_{o,t}) = C_{11}||\theta_{o,t} - \theta_{path,t}|| \tag{4.18}$$

Where $M()$ looks up the value of the normalized pushing corridor for the given coordinate. It encourages the robot and object to stay close to the middle of the corridor. $C_{a,grid}$ encourages the object to be directed towards the intermediate pushing goal with $\theta_{path,t}$ [31]. The terminal cost stays the same as in Equation 4.15 and $C_9$, $C_{10}$, $C_{11}$ are tuned weights.

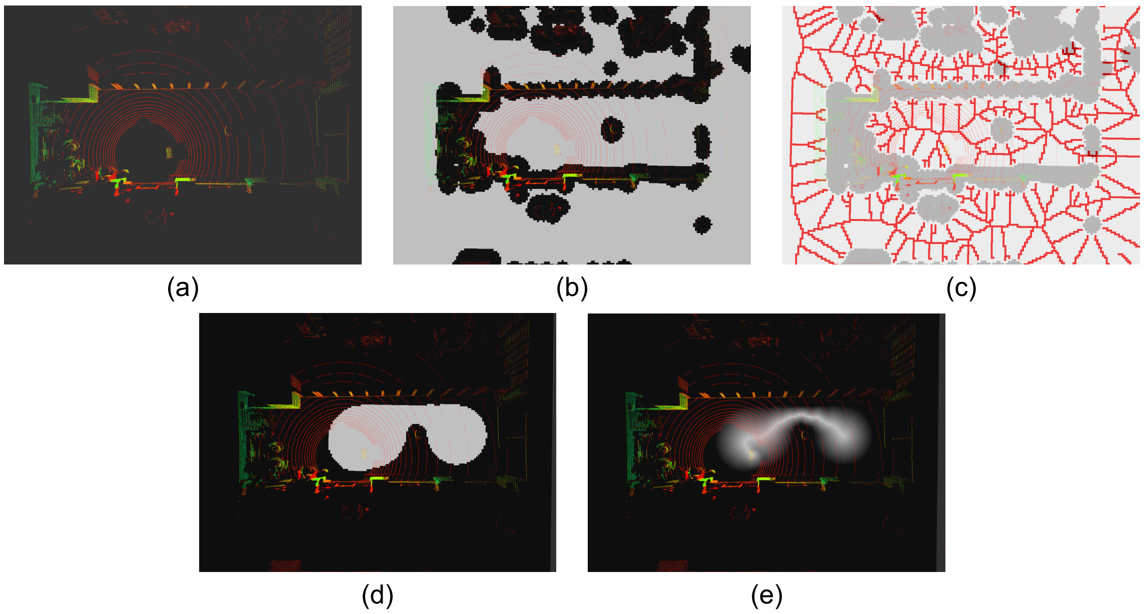


**Figure 4.2:** **(a)** Raw sensor data from the LiDAR. **(b)** Costmap. **(c)** Voronoi grid. **(d)** Pushing corridor. **(e)** Normalized pushing corridor.

## 4.2. Baselines

We compare our method to two baseline methods, based on the performance criteria outlined in Section 4.3. The first baseline is the learning-based controller of Krivic et al. [31], which is a state-of-the-art pushing controller. We choose to compare to this method, so we know how the method performs relative to the current state-of-the-art. The method is first explained in detail in Section 4.2.1. Then, we discuss the model-free reinforcement learning algorithm SAC and how it is used to learn a pushing policy as a second baseline in Section 4.2.2. Model-free reinforcement learning algorithms have shown promising results for pushing with a manipulator [10, 11, 15], which is why it is used as an additional baseline for pushing with a non-holonomic robot.

### 4.2.1. Adaptive Pushing Controller

The adaptive pushing controller by Krivic et al. [31] consists of a learning module that adapts parameters of a motion controller, based on past object-interaction data. They assume that objects can be approximated by a circle, and ignore angular velocities.
The input of the controller is the error of the movement direction $\gamma$, where the desired movement direction is defined as the direction between the object and goal:

$$\gamma = \text{atan2}(y_{o,g} - y_{o,t-1}, x_{o,g} - x_{o,t-1}) - \text{atan2}(y_{o,t} - y_{o,t-1}, x_{o,t} - x_{o,t-1}) \tag{4.19}$$

They base their pushing strategy on blending pushing and relocating actions for a feedforward controller, as successful pushing actions consist of relocating behind the object and moving forward.

$$\mathbf{v}_{ref} = \psi_{push}(\alpha)\hat{\mathbf{d}}_{push} + \psi_{relocate}(\alpha)\hat{\mathbf{d}}_{relocate} \tag{4.20}$$

Where $\psi_{push}(\alpha)$ and $\psi_{relocate}(\alpha)$ are activation functions, that should make the transitions between pushing and relocating smooth. $\hat{\mathbf{d}}_{push}$ and $\hat{\mathbf{d}}_{relocate}$ are unit vectors that determine the directions for pushing and relocating:

$$\hat{\mathbf{d}}_{push} = sgn(\cos(\alpha)) \tag{4.21}$$

$$\hat{\mathbf{d}}_{relocate} = sgn(\cos(\alpha_p - \alpha)) \tag{4.22}$$

Where $\alpha$ is the current angle to the goal, so $\alpha = \text{atan2}(y_{o,g} - y_{o,t}, x_{o,g} - x_{o,t})$ and $\alpha_p$ is the value of $\alpha$ for which the robot should move to advance the object towards the goal. This value is learned online. If the robot is behind the object, so $\alpha = 0$, the object should advance to the target. However, based on the object and environment, the robot might need to adjust its behaviour from this tactic. Therefore, they model the values of $\alpha$ for which the object moves towards the goal with a Von Mises distribution:

$$p(\alpha|\mu, \kappa) = \frac{e^{\kappa \cos(\alpha - \mu)}}{2\pi I_0(\kappa)} \tag{4.23}$$

Where $I_0(\kappa)$ is a modified Bessel function of order zero. $\kappa$ is a concentration parameter and $\mu$ is the mean direction of the distribution, which are both estimated by Maximum a-posteriori (MAP) estimation. They save $\alpha$ of successful pushes: $A_k = [\alpha_1, \alpha_2, ..., \alpha_k]$, and use Bayes' rule to estimate the model:

$$p(\mu, \kappa|A_k) = \frac{p(A_k|\mu, \kappa)p(\mu, \kappa)}{p(A_k)} \tag{4.24}$$

Where $p(\mu, \kappa|A_k)$ is the posterior, $p(A_k|\mu, \kappa)$ is the likelihood of the data in $A_k$ and $p(\mu, \kappa)$ is the prior. The prior of the von Mises distribution is initialized with values that correspond to a circular object with uniform mass and friction distributions. The activation functions are defined as:

$$\psi_{push}(\alpha) = p(\alpha|\mu, \kappa) \tag{4.25}$$

$$\psi_{relocate}(\alpha) = \sqrt{1 - \psi_{push}(\alpha)^2} \tag{4.26}$$

Then, they compute the the robot's velocity command with the feedforward term of $\mathbf{v}_{ref}$ and a feedback term:

$$\mathbf{v}_u = \mathbf{v}_{ref} - K_\mu \mu_\gamma - K_\gamma \gamma \tag{4.27}$$

Where $K_\mu$ and $K_\gamma$ are constants. Lastly, we convert the robot's control commands to $\mathbf{u}_r = [v_r, \omega_r]$ for a non-holonomic base, based on [14]:

$$v_r = \cos(\theta_r)v_{u,x} + \sin(\theta_r)v_{u,y} \tag{4.28}$$

$$\omega_r = \text{atan2}(v_{u,y}, v_{u,x}) \tag{4.29}$$

These inputs are used to control the robot in simulation, so we can compare the state-of-the-art controller to ours for a non-holonomic robot. Since our method does not make use of repositioning and cannot move sideways, we set $\psi_{relocate}$ to zero, to better compare the methods.

### 4.2.2. Reinforcement Learning for Pushing

As the second baseline, we use model-free reinforcement learning for learning a pushing policy. First, we give background information on reinforcement learning and the Soft Actor-Critic algorithm, that is used for learning the pushing policy (Section 4.2.2). Next, we define the implementation of the pushing policy in Section 4.2.2.

#### Background

Reinforcement learning focuses on training agents to reach a goal or learn a certain skill, based on interactions with the environment [53]. Here, we consider an agent that we train to complete the pushing task successfully. The agent interacts with the environment and can take a certain action $a_t \in A$ in its surroundings at a time step $t$. The state $s_t \in S$ defines the current state the agent is in. Based on the state and past experience, the agent chooses an action which gives a reward $r_t \in R$. This interaction between agent and environment is illustrated in Figure 4.3. The objective is to maximize the overall reward and learn an optimal policy $\pi^*$.

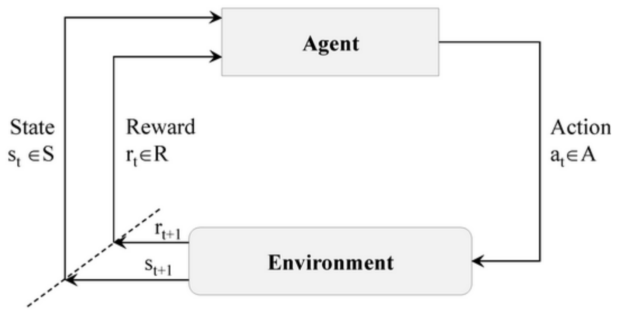


**Figure 4.3:** The interaction between agent and environment in reinforcement learning [3].

The Markov Property holds for the whole process of reinforcement learning. It says that the current state and action of the agent only depend on the previous state, not on other past states or actions. A Markov Decision Process (MDP) is defined by $(S, A, T, R, \gamma)$, where $S$ is the set of possible states, $A$ is the set of possible actions, $T$ is the transition probability function, $R$ is the reward function and $\gamma$ is the discount factor [58].

Model-free reinforcement learning is focused on learning directly from experiences, without the need of a model. In other words: we optimize the policy by trial and error, as learning a successful policy may be easier than learning a transition model in some cases, when the environment has complex dynamics. So, learning a pushing policy might be easier than learning the object dynamics and using it for planning.

However, finding a suitable reward function that directs the learning is of importance, but cannot always be easily obtained. Other downsides of this approach are that the policy cannot easily generalize to new tasks and a substantial amount of data is needed in most cases. An additional consideration is the sim-to-real gap: if the simulation does not perfectly capture the real world dynamics, the policy learned in simulation might not work as well in the real world.

Model-free reinforcement learning can be split up again in three directions: *value-based*, *policy-based* and *actor-critic* methods. Figure 4.4 shows a visualization of the different methods in reinforcement learning and how they are correlated. It shows that actor-critics are a combination of policy-based and value-based reinforcement learning.

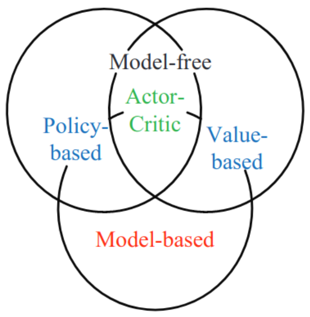


**Figure 4.4:** Classification of different reinforcement learning methods [27].

Value-based methods make use of a value or action-value function, without an explicit policy function to learn a task. Using the Bellman equations [53], the optimal value function is learned and the optimal policy is implicitly found. However, this procedure can be computationally expensive for continuous action spaces, as we need to optimize for the action in every state encountered. At the same time it is more sample efficient and stable than policy-based reinforcement learning [45].

In policy-based reinforcement learning, the optimal policy is approximated directly. One of the biggest advantages of policy gradient algorithms is that they are suited for continuous action spaces. However, the estimated gradient can have a large variance and it is calculated without information about past estimates, which can result in slow learning [22]. In addition, taking too large steps may break the policy or we may end up in a local minimum.

*Actor-Critic Reinforcement Learning*
Actor-critic architectures take the best from policy-based and value-based methods by combining them. Parameterizing the actor gives the benefit of computing continuous actions easily, while the critic gives the benefit of low-variance for the expected return estimates. In practice, this results in a speed-up of learning, such that actor-critics usually have good convergence [22].

Figure 4.5 shows the architecture of an actor-critic; the actor represents the policy while the critic represents the value function. The actor produces control inputs from current state $s_t$, from which the reward $r_t$ is calculated. The critic evaluates the rewards and adapts the value function estimate accordingly. Essentially, it solves the Bellman equation for the policy, after which the actor is updated based on information from the critic.

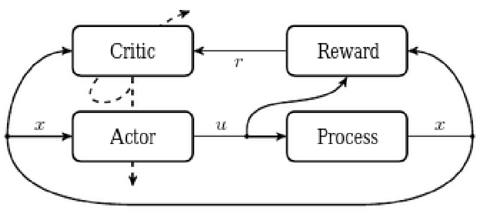


**Figure 4.5:** Actor-critic architecture [22].

*Soft Actor-Critic*
Soft Actor-Critic (SAC) [23] is based on the concept of entropy regularization, where the actor optimizes a policy to maximize both the expected return and entropy. The entropy $H$ is considered a measure of randomness of the policy, and a trade-off exists between the expected return and entropy. By rewarding the policy for high entropy, the state space is better explored and early convergence to a bad local optimum is prevented. The entropy $H$ can be written as:

$$H(P) = \mathbb{E}_{x \sim P}[-log(P(x))] \tag{4.30}$$

With $P$ as a probability distribution and $x$ a random variable. An additional reward is given that is proportional to the entropy at that timestep.

$$\pi^* = \arg\max_{\pi_\theta} \mathbb{E}_{\tau \sim \pi_\theta}[\sum_{t=0}^{\infty} \gamma^t (r_t + \alpha H(\pi_\theta(\cdot|s_t))] \tag{4.31}$$

Where $\alpha$ is the trade-off parameter that controls the exploitation versus exploration. A higher entropy gives more exploration, whereas a lower entropy results in more exploitation. Additionally, SAC incorporates a double Q-network and uses the minimum Q-value from the two networks. It also uses target networks that are updated less frequently for increased stability.

Since SAC is a state-of-the-art model-free reinforcement learning algorithm, and open-source implementations are available [26], we use it to learn the pushing policy. During preliminary tests, Proximal Policy Optimization (PPO) [50] was also considered. However, it showed a less consistent performance than SAC.

**Implementation of Soft Actor-Critic for Pushing**
Next, we define how to set up model-free reinforcement learning with SAC in the pushing environment.

We set up the action space to be the linear and angular acceleration of the robot with minimum and maximum values of -1.0 and 1.0, respectively. If an action results in exceeding the maximum linear and angular robot speed of $[0.0, 0.5]$ m/s $[-0.5, 0.5]$ rad/s, we clip the action such that the robot's maximum speed is not exceeded.

The observation space consists of 16 values (Table 4.1). These include the robot position and velocity ${}^{O}\mathbf{x}_{r,t}$, the object's velocity ${}^{O}\mathbf{v}_{o,t}$ and the previous robot action ${}^{O}\mathbf{u}_{r,t-1}$, all in the object frame. To give the agent an idea of the goal, we also include the distance between the goal and object ${}^{O}\mathbf{d}_{g,t}$ normalized by the start distance, the angle to the goal $\theta_{g,t}$ , and the error between the object's angle and the angle to the goal $\theta_{g,t}-\theta_{o,t}$, all in in the object frame. $\theta_{g,t}$ is defined as atan2$(y_{g,t}-y_{o,t}, x_{g,t}-x_{o,t})$.

**Table 4.1:** Observation space SAC for pushing.

| Observation | Size |
|---|---|
| ${}^{O}\mathbf{x}_{r,t}$ | 6 |
| ${}^{O}\mathbf{v}_{o,t}$ | 3 |
| ${}^{O}\mathbf{u}_{r,t-1}$ | 3 |
| ${}^{O}\mathbf{d}_{g,t}$ | 2 |
| $\theta_{g,t}$ | 1 |
| $\theta_{g,t}-\theta_{o,t}$ | 1 |

The reward function consists of the following elements:

$$r_{goal} = \begin{cases} 500 & \text{if goal reached} \\ -d_{g,t} & \text{otherwise} \end{cases} \tag{4.32}$$

Where $r_{goal}$ is the reward for advancing to the goal. If the goal is reached (the object is within 0.1 m of the goal), a big reward is given and the episode is reset. If the object is not at the goal, we give a non-sparse reward $d_{g,t}$ that penalizes larger distances between the goal and object. The distance between the object and the goal is normalized by dividing by the initial distance and scaling it between 0 and 1, to make sure we give equal rewards for different initial conditions. We also define three cases for which we give a penalty:

$$r_{penalty} = \begin{cases} -50 & \text{if object or robot outside workspace} \\ -150 & \text{if object and robot lose contact} \\ -150 & \text{if number of episode steps > 120} \end{cases} \tag{4.33}$$

Where $r_{penalty}$ gives penalties for exceeding the workspace, losing contact and when the episode takes too long. The total reward is thus:

$$r_{total} = r_{goal} + r_{penalty} \tag{4.34}$$

We use stable baselines' implementation of SAC [26] and train the algorithm for one million timesteps with the standard hyperparameters. The model with the highest reward is saved and used for testing.

## 4.3. Pushing Experiments

To evaluate the pushing performance, we perform pushing experiments, that consider multiple goal locations with different distances and turning radii. By using multiple goal locations, we can compare performance for varying goal location complexities, and performance across methods.

For the pushing tests without obstacle avoidance, we choose six goal locations for testing in simulation and in the motion capture room, based on the lab's dimensions of 9.0 × 4.0 × 2.2 m. The six goal locations include a straight-line push $[3.0, 0.0]$ m, left turns $[2.0, 1.0]$ m, $[4.0, 2.0]$ m, $[5.5, 2.0]$ m and right turns $[3.0, -1.0]$ m, $[3.0, -1.5]$ m. They capture different distances and turning radii. Figure 4.6 shows the goal locations relative to the object.

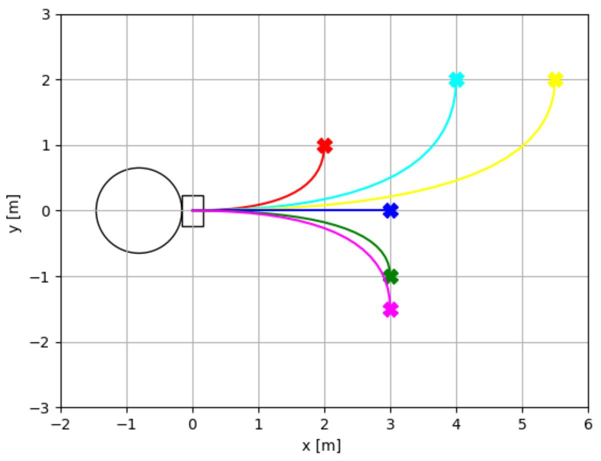
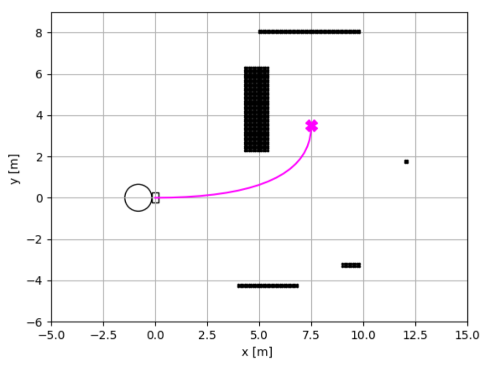
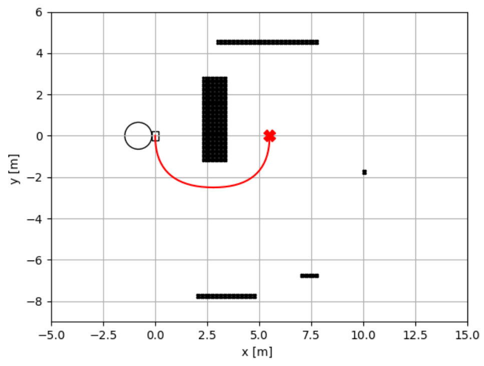


**Figure 4.6:** The various pushing goals visualized in color. The black circle depicts the robot and the black rectangle depicts the object to be pushed.

For pushing with obstacle avoidance, we use the goals $[5.5, 2.0]$ and $[5.5, 0.0]$ in the motion capture lab. In simulation we use $[5.5, 0.0]$ and $[7.5, 3.5]$, to include an easier pushing environment. The pushing environment in simulation is filled with rectangular objects for simplification. Figure 4.7 shows the pushing environments and goal locations as used in simulation.



**(a)** Obstacle avoidance environment 1, going to goal location $[7.5, 3.5]$ m.



**(b)** Obstacle avoidance environment 2, going to goal location $[5.5, 0.0]$ m.

**Figure 4.7:** Environments used in simulation, filled with rectangular obstacles.

**Performance Criteria**

For each environment and method, we run multiple simulations with randomized initial conditions. To efficiently compare across methods, we use the same seeds for randomizing the baselines and our method. To evaluate the performance of the different methods, we compare various metrics:

- Success rate
- Accuracy
- Pushing time
- Robot path length
- Object path length

First of all, we evaluate the success rate, which is based on the end position of the object. The distance between the end and goal position of the object should be below a set threshold. Furthermore, the object must be delivered within a certain timeframe and the object cannot lose contact with the forward bumper of the robot. If all these conditions are met, the pushing is considered successful.

Next, we compare the accuracy, which is the distance between the end and goal location of the object. The controller should not only deliver the package accurately, but do it in as little time as possible, with the least amount of effort. Therefore, we also compare the pushing time and path lengths. The pushing time is defined as the time between the first and last control input. For calculating the path lengths, we sum up the distance between all locations.

### 4.3.1. Simulation

We run thirty simulations for each goal position, where the initial conditions for the robot and object are randomized each run. We create the different initial conditions by uniformly sampling the initial object $\theta_o$ and robot angles $\theta_r$ and the y-position of the robot, relative to the object ${}^O y_r$. The x-position of the robot is chosen such that the robot and object start in contact. By randomizing these values, we make sure that different start configurations are evaluated, where the relative angles and positions of the robot and object differ. Table 4.2 shows the minimum and maximum values for the randomized initial conditions.

**Table 4.2:** Minimum and maximum values for randomized variables during testing.

| | Minimum value | Maximum value |
|---|---|---|
| $\theta_r$ | -5 ° | 5 ° |
| $\theta_o$ | -30 ° | 30 ° |
| ${}^O y_r$ | $-\frac{1}{2}L_{box}$ | $\frac{1}{2}L_{box}$ |

### 4.3.2. Motion Capture

Real-world pushing experiments are performed in the motion capture lab, so we can acquire the groundtruth states of the robot and object. We do experiments with and without obstacle avoidance, to evaluate the pushing controller in both scenarios. Since there is only limited time available in the motion capture lab, the amount of experiments per goal location is reduced to eleven, with different initial conditions. These are created by hand, by placing the object randomly against the bumper of the robot.

## 4.4. Implementation Details

For further clarification, we provide detailed explanations of the code implementations. Links to the code can be found in Appendix C.

The code for training the dynamics model is based on the probabilistic neural networks as used by Chua et al. [9] in Pytorch [59]. The MPPI controller is based on Williams et al. [61] and is written in Python for our robot. The dynamics and cost functions are implemented as stated in Section 3.2 and Section 4.1. Next, we define the global planning for obstacle avoidance in simulation (Section 4.4.1) and in the motion capture lab, as well as the ROS architecture (Section 4.4.2),.

### 4.4.1. Simulation

Since we do not receive LiDAR measurements in simulation, we create a costmap from the rectangular objects, including an inflation radius, in the environment ourselves. A Voronoi grid is computed in Python [51], and we search along the grid to find the shortest path between start and goal, using A*. Lastly, we compute the corridor and the normalized corridor as discussed in Section 4.1.3, which are used in the MPPI controller.

### 4.4.2. Motion Capture

To control the robot in the Motion Capture lab, a ROS architecture was created. Figure 4.8 illustrates the ROS packages used for the experiment.

For testing the controller in an obstacle-free environment, we only require the robot and object states. The packages mocap_optitrack [21] and bebop2_state_estimator [63] send the states to the mppi_push package, that uses the information to calculate trajectory rollouts and choose the most optimal action. From the mppi_push package, the velocity command is sent to the Husky and all topics are sent to the data_saver, that saves the trajectories and actions.

For including obstacle avoidance, we add an extra function to compute the free pushing space. We project the LiDAR points to a 2D occupancy grid, and inflate the obstacles in a 2D costmap with an inflation radius, using the costmap_2d package [13]. From the costmap, the planner node computes the pushing corridors: it is based on the voronoi_planner node [16] that uses the dynamic_voronoi package [4] to compute the Generalized Voronoi Diagram from the costmap, and finds a path along the diagram. We add functions for finding the pushing space around the path and normalizing it. The normalized corridor is then sent to the pushing controller, that sends a velocity command to the Husky.

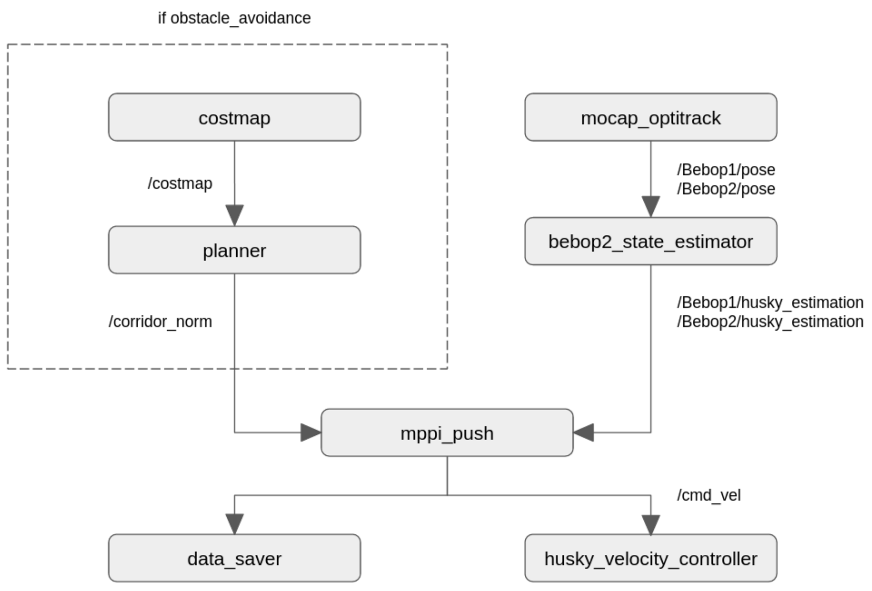


**Figure 4.8:** Scheme of ROS packages for pushing experiments.

# 5 Results

This section describes the results from simulation and real-world experiments. Section 5.1 presents results from the experiments in simulation, including test results of the dynamics model and pushing to various goal points in an empty and cluttered environment. The pushing results are compared to the baseline methods. Section 5.2 contains results of pushing experiments with the real robot in the motion capture lab.

## 5.1. Simulation experiments

First, we discuss the modelling errors of the obtained dynamics model in Section 5.1.1. This model is used in combination with the MPPI controller for pushing to various goal points in uncluttered and cluttered environments in Section 5.1.2 and 5.1.3, respectively.

### 5.1.1. Dynamics

From the 422 collected trajectories (8440 data samples), 70% of the data was used for training, 10% for validation and 20% for testing. We tune the hyperparameters and choose the configuration that reaches the lowest validation error. Lastly, we train a model on the full dataset with the found hyperparameters (Table 5.1), which is later used for pushing.

**Table 5.1:** Hyperparameter values used for training the simulation model

| Hyperparameter | Values |
|---|---|
| Neurons | 128 |
| Batch size | 64 |
| Learning rate | 0.0005 |
| Layers | 4 |
| Number of ensembles | 3 |
| Non-linear layers | ReLU |

Since we use the dynamics model for predicting trajectory rollouts, we evaluate its performance based on the accumulating error. The controller's inference time increases with longer trajectory rollouts, and therefore we are only interested in the near future. Based on later pushing experiments in simulation, we choose to evaluate the controller, and thus the model, for twenty timesteps (the next 2 seconds).

Figure 5.1 shows the accumulating error for twenty timesteps, based on the testing data in simulation. It shows an exponentially accumulating error, of 20 mm and 1.5 degrees for the position and orientation of the object after twenty timesteps. Appendix A compares test results from different neural networks architectures (Mixture Density Network, LSTM).

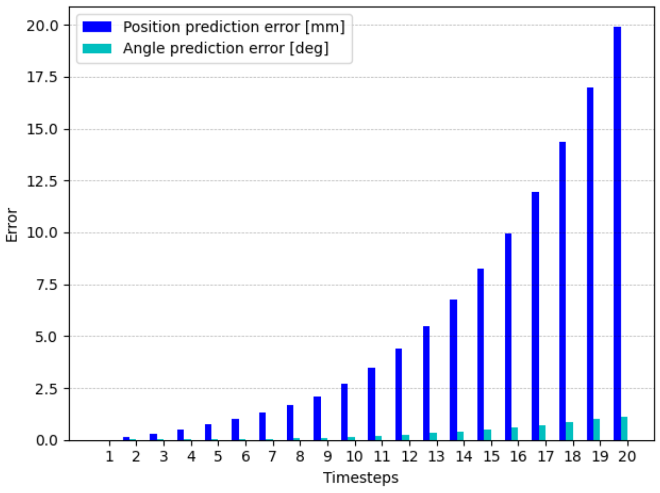


**Figure 5.1:** Accumulating error for twenty timesteps, based on testing data in simulation.

### 5.1.2. Pushing

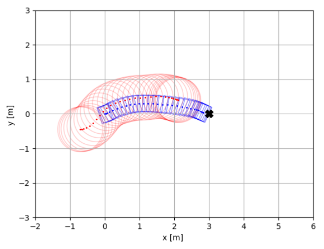
Next, we evaluate how the combination of the model and the MPPI controller perform in simulation for the six different goal locations. For each goal, we run thirty simulations with randomly initialized start conditions (Table 4.2).
The MPPI controller is run with a horizon of twenty timesteps and 150 trajectory samples. In addition, we use $\lambda = 2.0$ and set the sampling variances $\Sigma_v, \Sigma_\omega$ to 0.1 and 2.0, respectively. These variables were set by trial and error and allow the controller to run in real time.

We evaluate if the pushing is successful based on the end position of the object. It should be below a threshold, which is set at 1.0 m. In addition, the object and robot must not lose contact during the pushing, the object should be delivered within thirty seconds and the object cannot leave the forward bumper of the robot. If all these conditions hold, the pushing is considered successful.
The learning-based MPPI controller's performance is compared to the two baseline methods. We measure the success rate, accuracy, pushing time and the robot and object path lengths (Table 5.2). Figure 5.3 shows one trajectory for each goal point and method. All controllers were run with bounds on the linear and angular velocity of $[0.0, 0.5]$ m/s and $[-0.5, 0.5]$ rad/s.

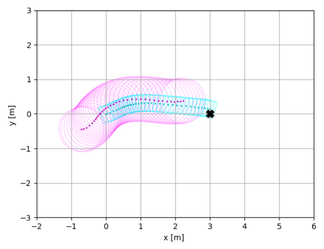
The learning-based MPPI controller has an overall succes rate of 100%, while Krivic et al. [31] and SAC reach slightly lower success rates of 98% and 96%, respectively.
We observe that the pushing time of SAC is on average 10% shorter than our method, but its accuracy is 50% lower and its path length is on average slightly longer (increase of 5% and 9% for the object and robot path lengths). The method strives to get close to the goal as fast as possible, and its velocity increases fast to the maximum value. However, the method does not choose a more optimal or shorter path than the other methods and reaches a lower accuracy.
The method from Krivic et al. [31] is a more conservative method, with longer pushing times and a higher accuracy than SAC. Compared to the learning-based MPPI, it has a 23% lower accuracy and a 12% higher pushing time. In addition, the robot and object path lengths are 6% and 3% longer, respectively. Planning with a more accurate model including an uncertainty estimate, gives the learning-based MPPI the ability to push the object with higher accuracy and lower pushing times than Krivic et al. with their approximate inverse models or the model-free policy.
Lastly, we observe that that success rates of SAC and Krivic et al. [31] are lowest for the goal at $[2.0, 1.0]$. Since the goal is relatively close, it can be hard to reach from certain initial conditions and may involve sharp turns. It shows us that the learning-based MPPI can better perform pushes with sharp corners.

**Table 5.2:** Success rate, accuracy, pushing time and path length results for pushing in simulation with a box-shaped object to various goal points.

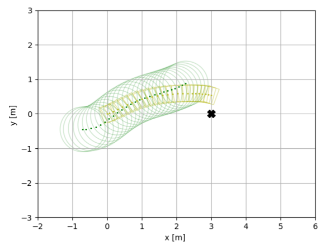
| Goal | | Success | Accuracy [m] | Time [s] | Path length$_O$ [m] | Path length$_R$ [m] |
|---|---|---|---|---|---|---|
| (3.0, 0.0) m | Ours | 1.0 | **0.11** $\pm$ **0.04** | 8.9 $\pm$ 1.0 | **2.94** $\pm$ **0.07** | **3.09** $\pm$ **0.17** |
| | Krivic et al. [31] | 1.0 | **0.11** $\pm$ **0.04** | 8.3 $\pm$ 1.9 | 3.02 $\pm$ 0.06 | 3.11 $\pm$ 0.14 |
| | SAC | 1.0 | 0.26 $\pm$ 0.16 | **7.4** $\pm$ **1.0** | 3.06 $\pm$ 0.18 | 3.19 $\pm$ 0.27 |
| (2.0, 1.0) m | Ours | 1.0 | **0.17** $\pm$ **0.13** | 9.6 $\pm$ 2.2 | 2.51 $\pm$ 0.43 | 2.87 $\pm$ 0.62 |
| | Krivic et al. [31] | 0.90 | 0.22 $\pm$ 0.11 | 9.7 $\pm$ 3.3 | **2.41** $\pm$ **0.22** | **2.74** $\pm$ **0.43** |
| | SAC | 0.77 | 0.18 $\pm$ 0.13 | **7.6** $\pm$ **1.4** | 2.47 $\pm$ 0.29 | **2.74** $\pm$ **0.45** |
| (4.0, 2.0) m | Ours | 1.0 | 0.12 $\pm$ 0.04 | 13.8 $\pm$ 1.7 | **4.63** $\pm$ **0.23** | **4.96** $\pm$ **0.39** |
| | Krivic et al. [31] | 0.97 | **0.08** $\pm$ **0.04** | 16.4 $\pm$ 4.2 | 4.84 $\pm$ 0.17 | 5.16 $\pm$ 0.29 |
| | SAC | 1.0 | 0.34 $\pm$ 0.19 | **13.1** $\pm$ **1.9** | 4.88 $\pm$ 0.37 | 5.30 $\pm$ 0.59 |
| (5.5, 2.0) m | Ours | 1.0 | **0.11** $\pm$ **0.03** | 16.0 $\pm$ 0.87 | **5.97** $\pm$ **0.11** | **6.26** $\pm$ **0.21** |
| | Krivic et al. [31] | 1.0 | **0.11** $\pm$ **0.16** | 20.1 $\pm$ 3.4 | 6.26 $\pm$ 0.13 | 6.61 $\pm$ 0.23 |
| | SAC | 1.0 | 0.22 $\pm$ 0.14 | **15.2** $\pm$ **1.5** | 6.22 $\pm$ 0.35 | 6.52 $\pm$ 0.50 |
| (3.0, -1.0) m | Ours | 1.0 | **0.11** $\pm$ **0.05** | 10.6 $\pm$ 1.2 | **3.24** $\pm$ **0.13** | **3.57** $\pm$ **0.26** |
| | Krivic et al. [31] | 1.0 | 0.31 $\pm$ 0.03 | 11.8 $\pm$ 2.5 | 3.28 $\pm$ 0.13 | 3.59 $\pm$ 0.28 |
| | SAC | 1.0 | 0.32 $\pm$ 0.25 | **9.5** $\pm$ **1.4** | 3.45 $\pm$ 0.41 | 3.80 $\pm$ 0.61 |
| (3.0, -1.5) m | Ours | 1.0 | **0.12** $\pm$ **0.05** | 12.0 $\pm$ 1.3 | **3.55** $\pm$ **0.20** | **3.96** $\pm$ **0.34** |
| | Krivic et al. [31] | 1.0 | 0.13 $\pm$ 0.08 | 14.1 $\pm$ 3.7 | 3.69 $\pm$ 0.20 | 4.09 $\pm$ 0.34 |
| | SAC | 1.0 | 0.16 $\pm$ 0.14 | **11.0** $\pm$ **1.6** | 3.94 $\pm$ 0.35 | 4.40 $\pm$ 0.50 |



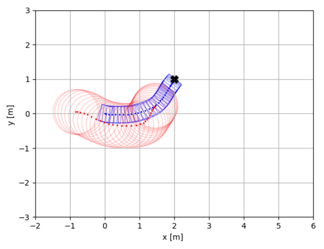
**(a)** Pushing to goal location (3.0, 0.0) m with our method.



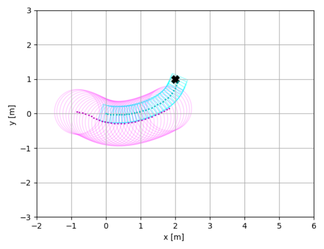
**(b)** Pushing to goal location (3.0, 0.0) m with Krivic et al. [31].



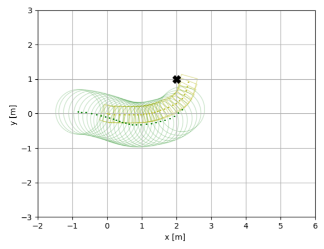
**(c)** Pushing to goal location (3.0, 0.0) m with the model-free baseline method.



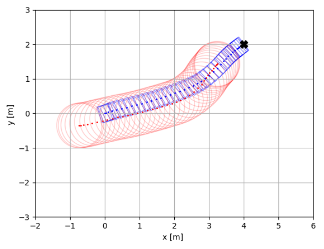
**(d)** Pushing to goal location (2.0, 1.0) m with our method.



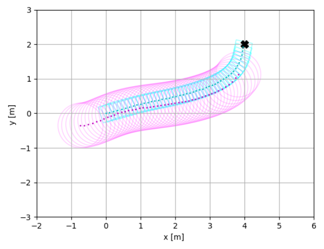
**(e)** Pushing to goal location (2.0, 1.0) m with Krivic et al. [31].



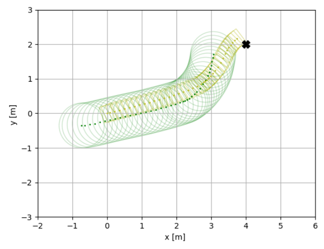
**(f)** Pushing to goal location (2.0, 1.0) m with the model-free baseline method.



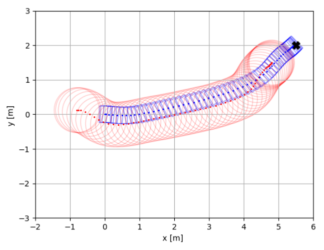
**(g)** Pushing to goal location (4.0, 2.0) m with our method.



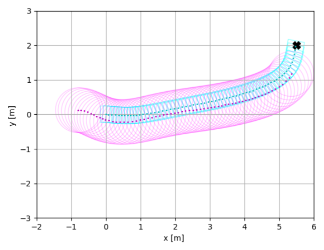
**(h)** Pushing to goal location (4.0, 2.0) m with Krivic et al. [31].



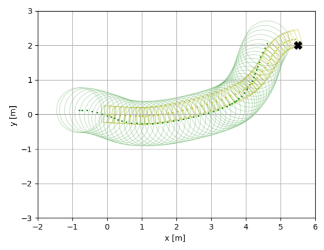
**(i)** Pushing to goal location (4.0, 2.0) m with the model-free baseline method.



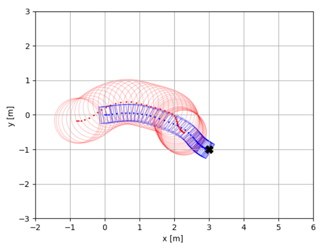
**(a)** Pushing to goal location (5.5, 2.0) m with our method.



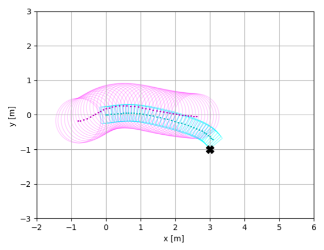
**(b)** Pushing to goal location (5.5, 2.0) m with Krivic et al. [31].



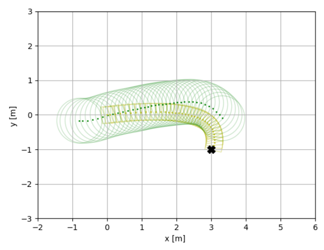
**(c)** Pushing to goal location (5.5, 2.0) m with the model-free baseline method.



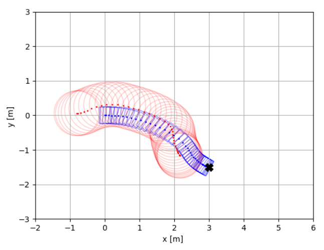
**(d)** Pushing to goal location (3.0, -1.0) m with our method.



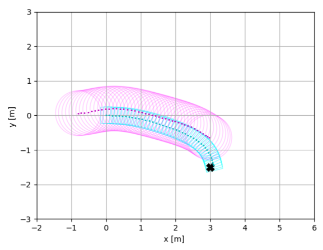
**(e)** Pushing to goal location (3.0, -1.0) m with Krivic et al. [31].



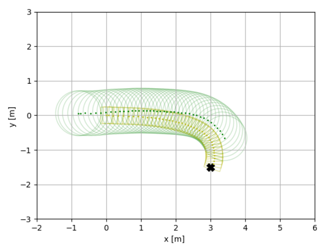
**(f)** Pushing to goal location (3.0, -1.0) m with the model-free baseline method.



**(g)** Pushing to goal location (3, -1.5) m with our method.



**(h)** Pushing to goal location (3, -1.5) m with Krivic et al. [31].



**(i)** Pushing to goal location (3, -1.5) m with the model-free baseline method.

**Figure 5.3:** Examples of pushing trajectories for pushing with a box-shaped object in simulation.

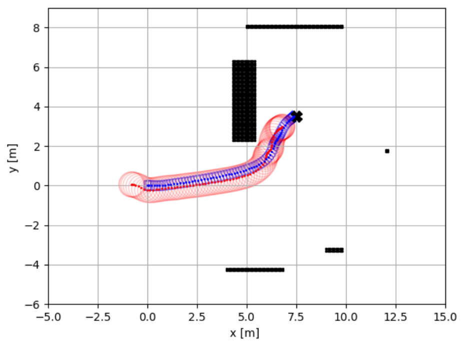
### 5.1.3. Obstacle-aware Pushing

Section 5.1.2 showed the limited pushing accuracy of SAC (especially in sharp corners), therefore we do not include it in our further experiments for obstacle-aware pushing. In order to validate our method, we create an environment filled with obstacles and test both methods for two different goal locations (Figure 5.4). Here, one goal location is 'easy', where the robot needs to avoid an obstacle when turning towards the goal. The second goal location is harder, where the robot and object need to go around the obstacle, while there is not a lot of space to maneuver. The second environment is more close to the layout of the motion capture room, where the real experiments are done, so we can compare results later.
Again, we collect thirty runs for both goals with randomly initialized conditions according to Table 4.2. In addition to the criteria for successful pushing named in Section 5.1.2, the robot or object cannot collide with the obstacles at any point during the pushing. Furthermore, we extend the maximum pushing time to ninety seconds, since the robot needs to cover more distance.

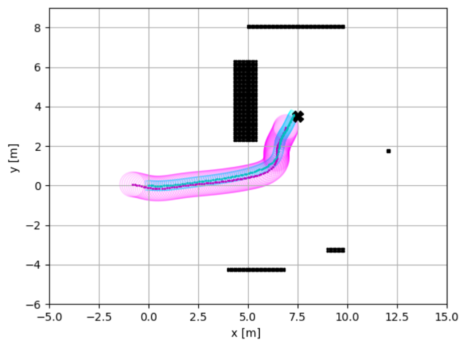
Table 5.3 illustrates the pushing performance of Krivic et al. [31] and our method. The success rates of the methods for both goal points are similar, but [31] is 43% less accurate and 44% slower than ours on average. This is also illustrated in Figure 5.4, where we can clearly see the paths executed by our controller are shorter and thus less conservative.

**Table 5.3:** Success rate, accuracy, pushing time and path length results for pushing in simulation with a box-shaped object to various goal points avoiding obstacles.

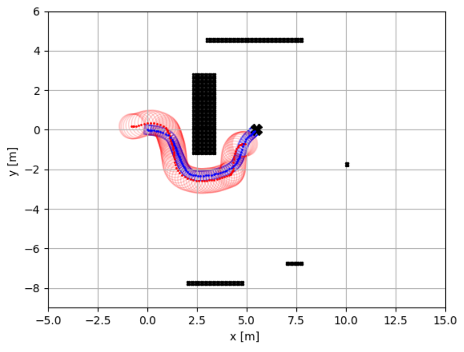
| Goal | | Success | Accuracy [m] | Time [s] | Path length$_O$ [m] | Path length$_R$ [m] |
|---|---|---|---|---|---|---|
| (7.5, 3.5) m | Ours | 1.0 | **0.16 ± 0.04** | **22.7 ± 10** | **8.79 ± 0.11** | **9.15 ± 0.21** |
| | Krivic et al. [31] | 1.0 | 0.21 ± 0.02 | 34.7 ± 20 | 9.10 ± 0.10 | 9.48 ± 0.16 |
| (5.5, 0.0) m | Ours | 0.67 | **0.18 ± 0.04** | **26.2 ± 4.4** | **7.85 ± 0.65** | **8.83 ± 0.84** |
| | Krivic et al. [31] | 0.67 | 0.37 ± 0.09 | 52.7 ± 5.7 | 8.87 ± 0.36 | 10.09 ± 0.58 |



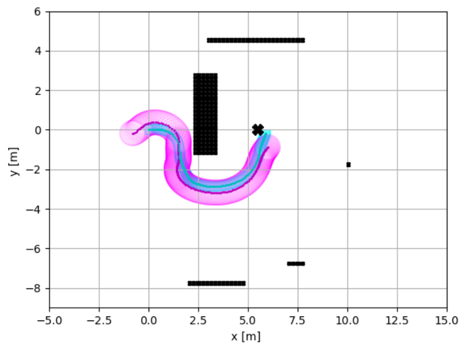
**(a)** Pushing to goal location (7.5, 3.5) m with our method.



**(b)** Pushing to goal location (7.5, 3.5) m with Krivic et al. [31].



**(c)** Pushing to goal location (5.5, 0.0) m with our method.



**(d)** Pushing to goal location (5.5, 0.0) m with Krivic et al. [31].

**Figure 5.4:** Examples of pushing trajectories for obstacle-aware pushing with a box-shaped object.

## 5.2. Real robot experiments

This section shows the results obtained from testing with the real robot in the motion capture environment. Section 5.2.1 discusses the modelling errors of the model, that is used for pushing in Section 5.2.2 and 5.2.3. Appendix B includes results from additional experiments, for pushing with a cylinder-shaped object and pushing a different box.

### 5.2.1. Dynamics

We use the same amount of data for training, validating and testing the real-world object dynamics model and evaluate it again on the accumulating error of twenty timesteps (Figure 5.5). Instead of three ensembles, we use six for increased accuracy (Table 5.4). It shows a position error of 20 mm and an angular error of 2 degrees after twenty timesteps.

**Table 5.4:** Hyperparameter values used for training the real-world box model

| Hyperparameter | Values |
|---|---|
| Neurons | 128 |
| Batch size | 64 |
| Learning rate | 0.0005 |
| Layers | 3 |
| Number of ensembles | 6 |
| Non-linear layers | ReLU |

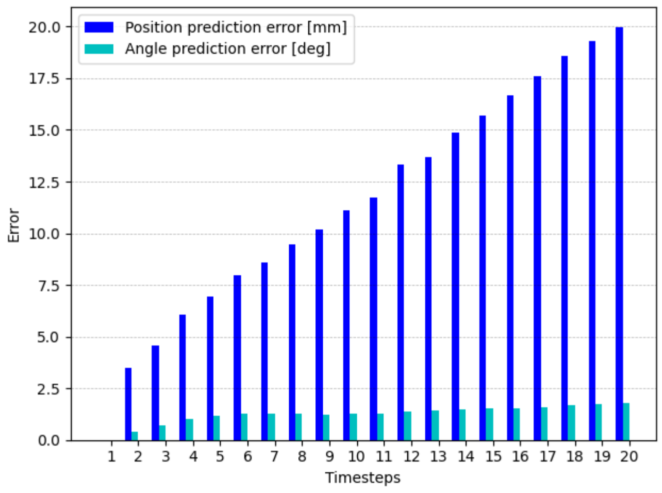


**Figure 5.5:** Accumulating error for twenty timesteps, based on real world testing data for the box-shaped object.

**Difference Simulation vs. Real World**
The simulation that was used for early verification of the method is based on Differential Algebraic Equations. Therefore, we can verify how well the analytical model performs in real-life conditions, by comparing the outcomes to real-world data. Figure 5.6 illustrates the twenty timestep error for the real world test data (the same as used for evaluating the neural network model), when predicting the object's displacement with the analytical model.

We observe that the error for the analytical model is higher than when we train a model on the real-world data. This finding agrees with literature [6], where learned models outperform analytical models in real-world pushing environments.

The results from simulation showed an exponentially increasing error, whereas Figure 5.5 and 5.6 show a linear increase. The trajectories in simulation were collected by navigating the robot to a goal, without any constraints to keep the object in front of the robot. Therefore, the object slipped away quickly during the trajectories. The data in the real world was collected by controlling the robot to create push trajectories. These trajectories thus contain less data samples, where the object is at the edges of the bumper and is on the verge of losing contact. We believe that the difference in data collection is the reason that the error plots for simulation and the real world differ. However, we show that both methods of collecting data are sufficient for training an object dynamics model, that can be used for pushing.

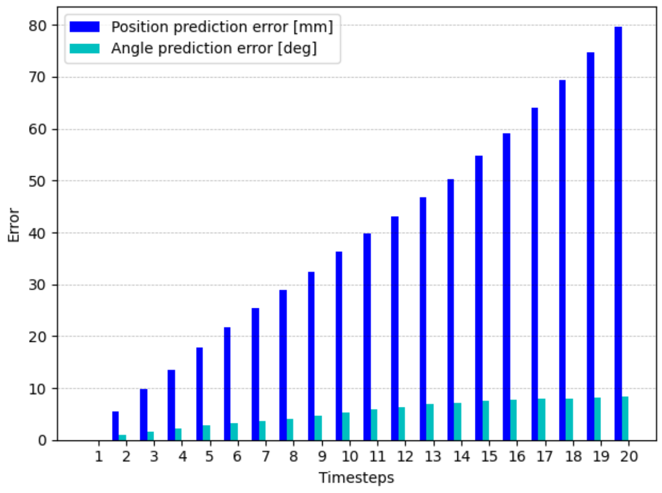


**Figure 5.6:** Accumulating error for twenty timesteps, based on real world testing data for the box-shaped object calculated from the analytical model.

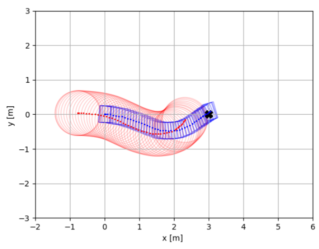
### 5.2.2. Pushing

We test the pushing performance with the same six goals as used in the simulation and initialize the robot and object start conditions randomly. Instead of running thirty experiments per goal, we reduce this amount to eleven due to time restrictions in the motion capture lab. In addition, we lower the bounds on the linear and angular velocity to $[0.0, 0.3]$ m/s and $[-0.3, 0.3]$ rad/s for safety reasons.
The controller has a prediction horizon of twenty timesteps, we use 150 trajectory samples, and $\lambda = 2.0$. The sampling variances $(\Sigma_v, \Sigma_\omega)$ are set to 0.5 and 2.0, respectively. The earlier defined conditions (Section 5.1.2) for successful pushing hold.

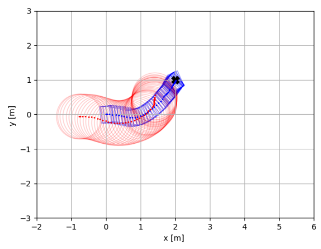
Table 5.5 illustrates the pushing performance in terms of success, accuracy, pushing time and path lengths. The success rate of 93% and overall accuracy of 0.33 m are lower than in simulation. The pushing time is higher than in simulation, which is partly due to the lower maximum velocity. The object's and robot's path length are 6% and 3% longer than in simulation. The slightly longer path lengths and lower accuracy compared to simulation are to be expected. The simulation's dynamics are based on assumptions and approximations of the real-world dynamics, which are more complex and harder to describe or learn.
Figure 5.7 shows example trajectories for each of the different goals, from different initial conditions.

**Table 5.5:** Success rate, accuracy, pushing time and path length results for pushing with the real robot and a box-shaped object to various goal points.

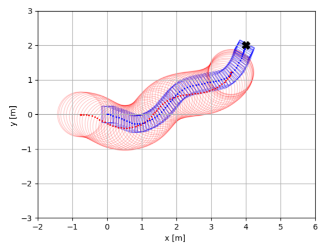
| Goal | Success | Accuracy [m] | Time [s] | Path length$_O$ [m] | Path length$_R$ [m] |
|---|---|---|---|---|---|
| (3.0, 0.0) m | 1.0 | 0.25 $\pm$ 0.13 m | 14.4 $\pm$ 0.7 | 3.19 $\pm$ 0.15 | 3.31 $\pm$ 0.17 |
| (2.0, 1.0) m | 1.0 | 0.31 $\pm$ 0.28 m | 15.4 $\pm$ 2.3 | 2.68 $\pm$ 0.38 | 2.91 $\pm$ 0.41 |
| (4.0, 2.0) m | 0.82 | 0.36 $\pm$ 0.18 m | 21.6 $\pm$ 1.6 | 5.21 $\pm$ 0.28 | 5.35 $\pm$ 0.31 |
| (5.5, 2.0) m | 0.91 | 0.42 $\pm$ 0.26 m | 27.5 $\pm$ 1.9 | 6.36 $\pm$ 0.33 | 6.58 $\pm$ 0.36 |
| (3.0, -1.0) m | 0.91 | 0.31 $\pm$ 0.15 m | 14.7 $\pm$ 1.9 | 3.34 $\pm$ 0.22 | 3.45 $\pm$ 0.29 |
| (3.0, -1.5) m | 0.91 | 0.34 $\pm$ 0.13 m | 15.3 $\pm$ 1.9 | 3.52 $\pm$ 0.32 | 3.62 $\pm$ 0.40 |



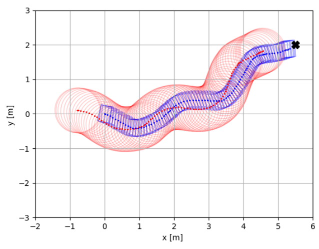
**(a)** Pushing to goal location (3.0, 0.0) m.



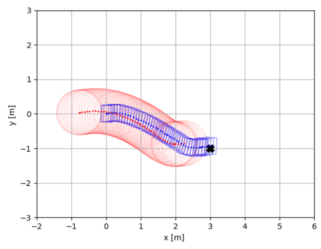
**(b)** Pushing to goal location (2.0, 1.0) m.



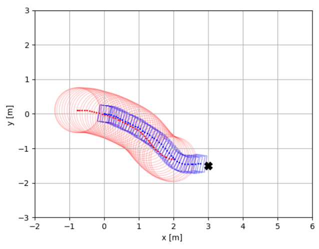
**(c)** Pushing to goal location (4.0, 2.0) m.



**(d)** Pushing to goal location (5.5, 2.0) m.



**(e)** Pushing to goal location (3.0, -1.0) m.



**(f)** Pushing to goal location (3, -1.5) m.

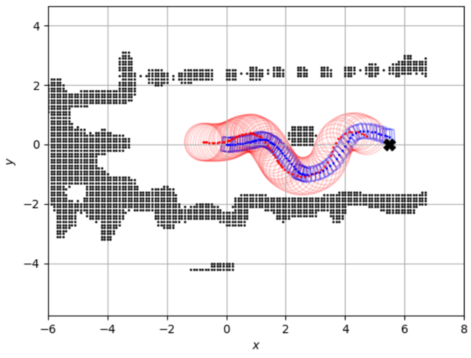
**Figure 5.7:** Examples of real-world pushing trajectories for pushing with a box-shaped object.

### 5.2.3. Obstacle-aware Pushing

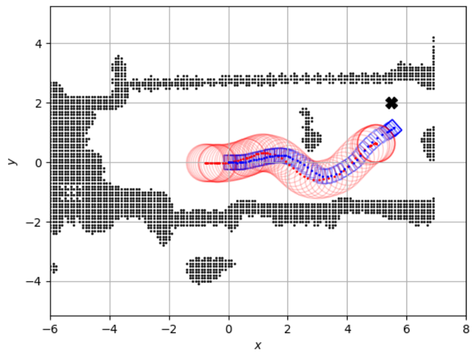
Next, we test the pushing in an environment filled with an obstacle and two different goal locations. Figure 5.8 shows example trajectories including the occupancy grid from the LiDAR measurements, showing the walls of the room and the obstacle. Table 5.6 shows that the success rates are similar to the difficult environment in simulation, because of the limited pushing space in the motion capture room. The robot does not have enough space to perform successful pushing from all initial conditions.

**Table 5.6:** Success rate, accuracy, pushing time and path length results for pushing with the real robot and a box-shaped object to various goal points with obstacles.

| Goal | Success | Accuracy [m] | Time [s] | Path length$_O$ [m] | Path length$_R$ [m] |
|---|---|---|---|---|---|
| (5.5, 0.0) m | 0.55 | $0.35 \pm 0.21$ | $16.1 \pm 3.8$ | $5.79 \pm 0.53$ | $5.89 \pm 0.66$ |
| (5.5, 2.0) m | 0.45 | $0.67 \pm 0.19$ | $19.7 \pm 2.2$ | $6.24 \pm 0.38$ | $6.43 \pm 0.46$ |



**(a)** Pushing to goal location (5.5, 0.0) m.



**(b)** Pushing to goal location (5.5, 2.0) m.

**Figure 5.8:** Examples of pushing trajectories for obstacle-aware pushing with a box-shaped object.

# 6 Conclusion

## 6.1. Summary

Mobile robots are increasingly being used in production processes, such as warehouses and factories for running processes smoother, faster and more accurate. They are expected to complete tasks such as loading, stacking, and transporting packages autonomously in complex and uncontrolled environments. By equipping mobile robots with a pushing skill, we are extending the set of tasks that a robot can autonomously accomplish. Objects can be pushed with the mobile base to goal locations, or obstacles can be pushed out of the way to clear paths.

This thesis considers pushing objects to goal locations with a non-holonomic mobile robot in real-world environments. The non-holonomic robot is constrained in its motion, so losing contact and repositioning should be avoided for time-efficiency. We present a learning-based method, where object dynamics are learned with an ensemble of probabilistic networks. It captures the inherent stochasticity of pushing, as well as uncertainty from lack of data. It is combined with a MPPI controller [61], that takes this uncertainty into account in the planning of push actions. Since the controller optimizes for a finite time horizon at each timestep, we can recover from slight model inaccuracies. For obstacle-aware pushing, we provide the MPPI controller with a costmap grid, that contains the free pushing space and constrain the robot and object to it.

The method is first verified in simulation for six different goal locations. Its performance is compared to two baseline methods: the state-of-the-art pushing controller proposed by Krivic et al. [31], adapted for a non-holonomic mobile robot, and the model-free reinforcement learning baseline SAC.
The three methods perform comparably in terms of success rates: the learning-based MPPI reaches a success rate of 100% in simulation, while SAC and Krivic et al. [31] reach success rates of 98% and 96%, respectively. Compared to SAC, the learning-based MPPI performs pushing tasks with an overall increase of 50% in accuracy, but a 10% longer pushing time. Moreover, it increases accuracy by 23% and decreases the pushing time by 12%, compared to Krivic et al. [31]. The probabilistic model can reason about the uncertainty and adjusts its plan and speed accordingly, therefore it performs less conservative pushing than [31]. This was also shown for pushing in cluttered environments, where Krivic et al. [31] and the learning-based MPPI show comparable success rates, but the learning-based MPPI increases accuracy with 43% and decreases pushing time with 44%.

We validate the approach in a real-world environment, where the object and robot are tracked with a motion capture system. The method works consistently for all six goal points with an overall success rate of 94%. Obstacle-aware pushing achieves a success rate of 50%. Due to the constrained space in the motion capture room, the robot cannot recover from all initial conditions.

In conclusion, we develop a method for accurate real-time and real-world pushing with a non-holonomic mobile robot. The method has shown improved accuracy and lower pushing time compared to baselines in simulation, and shows comparable performance in the real world.

## 6.2. Discussion and Future Work

During the real-world experiments in the motion capture lab, we saw that the available space was too limited for the robot to be able to successfully push in the cluttered environment from all initial conditions. This was also verified in simulation, where the robot could not avoid the obstacles in the difficult environment from all initial conditions, similar to the motion capture lab. Further work could involve investigating which goal locations are possible for the robot to reach, and which not.

Furthermore, a limited amount of experiments was performed per goal location (thirty experiments in simulation, and eleven experiments in the real world). For this amount of experiments, there was no clear significant difference between our method and the baselines. A significant difference might become apparent if more experiments were performed.

Further work should include data collection in simulation with a pushing controller, instead of the method currently used. These trajectories might be more informative, improve pushing performance, and might explain the difference between the error plots for simulation and the real world. For example, the pushing controller by Krivic et al. [31] could be used for data collection. Another option would be to employ an iterative strategy, where we first collect data with our method, train the dynamics model, collect new trajectories with the pushing controller, and retrain the dynamics model.

Another limitation in this work is the limited parameter tuning of the model-free reinforcement learning baseline. Standard hyperparameters were used, and performance could possibly be further optimized if the hyperparameters are better tuned. Further research could also investigate the sim-to-real gap, to see if the method can work in real-world experiments. If this proves to be possible, the LiDAR measurements or occupancy grid could be included into the network, which can then be trained in an end-to-end manner to perform pushing with obstacle avoidance.

Further work should also include improving the model for more complex objects, such as the cylinder. Experiments showed a drastic decrease in success rate for pushing the cylinder. This is probably due to the lower accuracy of the model, and the fact that the cylinder-shaped object can roll. We showed a success rate of 87% for pushing a similar box with slightly different parameters. Further work could include improving the method for similar objects with different properties, for instance by domain randomization or an additional online learning method.

Lastly, it would be interesting to evaluate the method's performance outside of the motion capture room. The mobile robot would need to localize itself using SLAM and recognize the object's position from Aruco Markers or a computer vision model. This would introduce additional noise in the states, compared to the states received from the motion capture system, which can influence performance.

# References

[1] P. Agarwal et al. “Nonholonomic path planning for pushing a disk among obstacles”. In: *Proceedings of International Conference on Robotics and Automation*. Vol. 4. IEEE. 1997, pp. 3124–3129.

[2] E. Arruda et al. “Uncertainty averse pushing with model predictive path integral control”. In: *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*. IEEE. 2017, pp. 497–502.

[3] A. Azar et al. “Drone Deep Reinforcement Learning: A Review”. In: *Electronics* 10.9 (2021), p. 999.

[4] C. Sprunk B. Lau and W. Burgard. *dynamic_voronoi.* URL: `http://wiki.ros.org/dynamicvoronoi` (visited on 10/05/2022).

[5] M. Bauza, F. Hogan, and A. Rodriguez. “A data-efficient approach to precise and controlled pushing”. In: *Conference on Robot Learning*. PMLR. 2018, pp. 336–345.

[6] M. Bauza and A. Rodriguez. “A probabilistic data-driven model for planar pushing”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 3008–3015.

[7] A. Berthoz. *The brain’s sense of movement*. Vol. 10. Harvard University Press, 2000.

[8] F. Bertoncelli, F. Ruggiero, and L. Sabattini. “Linear time-varying MPC for nonprehensile object manipulation with a nonholonomic mobile robot”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 11032–11038.

[9] K. Chua et al. “Deep reinforcement learning in a handful of trials using probabilistic dynamics models”. In: *Advances in neural information processing systems* 31 (2018).

[10] I. Clavera, D. Held, and P. Abbeel. “Policy transfer via modularity and reward guiding”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 1537–1544.

[11] L. Cong et al. “Reinforcement Learning With Vision-Proprioception Model for Robot Planar Pushing”. In: *Frontiers in Neurorobotics* 16 (2022).

[12] L. Cong et al. “Self-Adapting Recurrent Models for Object Pushing from Learning in Simulation”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 5304–5310.

[13] M. Ferguson D. Lu and A. Hoy. *costmap_2d.* URL: `http://wiki.ros.org/costmap%5C_2d` (visited on 10/05/2022).

[14] A. De Luca and G. Oriolo. “Local incremental planning for nonholonomic mobile robots”. In: *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. IEEE. 1994, pp. 104–110.

[15] N. Dengler, D. Großklaus, and M. Bennewitz. “Learning Goal-Oriented Non-Prehensile Pushing in Cluttered Scenes”. In: *arXiv preprint arXiv:2203.02389* (2022).

[16] R. Fedorenko. *voronoi_planner.* URL: `http://wiki.ros.org/voronoi%5C_planner` (visited on 10/05/2022).

[17] J. Flanagan, M. Bowman, and R. Johansson. “Control strategies in object manipulation tasks”. In: *Current opinion in neurobiology* 16.6 (2006), pp. 650–659.

[18] Y. Gal, J. Hron, and A. Kendall. “Concrete dropout”. In: *Advances in neural information processing systems* 30 (2017).

[19] H. Gao, Y. Ouyang, and M. Tomizuka. “Online learning in planar pushing with combined prediction model”. In: *2021 American Control Conference (ACC)*. IEEE. 2021, pp. 2529–2536.

[20] G. Goyal, A. Ruina, and J. Papadopoulos. “Planar sliding with dry friction part 1. limit surface and moment function”. In: *Wear* 143.2 (1991), pp. 307–330.

[21] K. Gräve and A. Bencz. *mocap_optitrack*. URL: `http://wiki.ros.org/mocap_optitrack` (visited on 10/04/2022).

[22] I. Grondman et al. “A survey of actor-critic reinforcement learning: Standard and natural policy gradients”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307.

[23] T. Haarnoja et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[24] P. Hart, N. Nilsson, and B. Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[25] Z. He, J. Wang, and C. Song. “A review of mobile robot motion planning methods: from classical motion planning workflows to reinforcement learning-based architectures”. In: *arXiv preprint arXiv:2108.13619* (2021).

[26] A. Hill et al. *Stable Baselines*. `https://github.com/hill-a/stable-baselines`. 2018.

[27] J. Hua et al. “Learning for a robot: Deep reinforcement learning, imitation learning, transfer learning”. In: *Sensors* 21.4 (2021), p. 1278.

[28] T. Igarashi, Y. Kamiyama, and M. Inami. “A dipole field for object delivery by pushing on a flat surface”. In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 5114–5119.

[29] Zarei K., H. Karimpour, and M. Keshmiri. “Robotic box pushing under indeterminate anisotropic friction properties”. In: *International Journal of Dynamics and Control* 9.3 (2021), pp. 872–884.

[30] O. Khatib. “Real-time obstacle avoidance for manipulators and mobile robots”. In: *Autonomous robot vehicles*. Springer, 1986, pp. 396–404.

[31] S. Krivic and J. Piater. “Pushing corridors for delivering unknown objects with a mobile robot”. In: *Autonomous Robots* 43.6 (2019), pp. 1435–1452.

[32] S. Lee and M. Cutkosky. “Fixture planning with friction”. In: *Journal of Engineering for Industry* 113.3 (1991), pp. 320–327.

[33] J. Liang et al. “Gpu-accelerated robotic simulation for distributed reinforcement learning”. In: *Conference on Robot Learning*. PMLR. 2018, pp. 270–282.

[34] K. Lynch, H. Maekawa, and K. Tanie. “Manipulation and active sensing by pushing using tactile feedback.” In: *IROS*. Vol. 1. 1992, pp. 416–421.

[35] K. Lynch and M. Mason. “Stable pushing: Mechanics, controllability, and planning”. In: *The international journal of robotics research* 15.6 (1996), pp. 533–556.

[36] K. Lynch and T. Murphey. “Control of nonprehensile manipulation”. In: *Control problems in robotics*. Springer, 2003, pp. 39–57.

[37] M. Mason. *Manipulator grasping and pushing operations*. 1982.

[38] M. Mason. “Mechanics and planning of manipulator pushing operations”. In: *The International Journal of Robotics Research* 5.3 (1986), pp. 53–71.

[39] M. Mason. “On the scope of quasi-static pushing”. In: *International Symposium on Robotics Research, 1986*. 1986, pp. 229–233.

[40] T. Meriçli, M. Veloso, and H. Akın. “Push-manipulation of complex passive mobile objects using experimentally acquired motion models”. In: *Autonomous Robots* 38.3 (2015), pp. 317–329.

[41] K. Miyazawa, Y. Maeda, and T. Arai. “Planning of graspless manipulation based on rapidly-exploring random trees”. In: *(ISATP 2005). The 6th IEEE International Symposium on Assembly and Task Planning: From Nano to Macro Assembly and Manufacturing, 2005.* IEEE. 2005, pp. 7–12.

[42] A. Nagabandi et al. “Deep dynamics models for learning dexterous manipulation”. In: *Conference on Robot Learning*. PMLR. 2020, pp. 1101–1112.

[43] A. Nagabandi et al. “Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 7559–7566.

[44] R. Neal. *Bayesian learning for neural networks*. Vol. 118. Springer Science & Business Media, 2012.

[45] H. Nguyen and H. La. “Review of deep reinforcement learning for robot manipulation”. In: *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE. 2019, pp. 590–595.

[46] OptiTrack. *Motive: Optical motion capture software.* URL: `https://optitrack.com/software/motive/` (visited on 10/04/2022).

[47] OptiTrack. *Optitrack prime 17w tech specs*. URL: `https://optitrack.com/cameras/prime-17w/specs.html` (visited on 10/04/2022).

[48] F. Paus, T. Huang, and T. Asfour. “Predicting pushing action effects on spatial object relations by learning internal prediction models”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2020, pp. 10584–10590.

[49] R. Sabbagh Novin et al. “A model predictive approach for online mobile manipulation of non-holonomic objects using learned dynamics”. In: *The International Journal of Robotics Research* 40.4-5 (2021), pp. 815–831.

[50] J. Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[51] Scipy. *Spatial data structures and algorithms*. URL: `https://docs.scipy.org/doc/scipy/tutorial/spatial.html` (visited on 10/04/2022).

[52] D. Seborg et al. *Process dynamics and control*. John Wiley & Sons, 2016.

[53] R. Sutton and A. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[54] O. Takahashi and R. Schilling. “Motion planning in a plane using generalized Voronoi diagrams”. In: *IEEE Transactions on robotics and automation* 5.2 (1989), pp. 143–150.

[55] Y. Tang, M. Wisse, and W. Pan. *Submitted to 2023 IEEE International Conference on Robotics and Automation (ICRA)*.

[56] C. Tao et al. “Path Integral Methods with Stochastic Control Barrier Functions”. In: *arXiv preprint arXiv:2206.11985* (2022).

[57] M. Valdenegro-Toro and D. Mori. “A Deeper Look into Aleatoric and Epistemic Uncertainty Disentanglement”. In: *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE. 2022, pp. 1508–1516.

[58] M. Van Otterlo and M. Wiering. “Reinforcement learning and markov decision processes”. In: *Reinforcement learning*. Springer, 2012, pp. 3–42.

[59] Q. Vuong. *Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models*. URL: `https://github.com/quanvuong/handful-of-trials-pytorch` (visited on 10/04/2022).

[60] G. Williams, A. Aldrich, and E. Theodorou. “Model predictive path integral control: From theory to parallel computation”. In: *Journal of Guidance, Control, and Dynamics* 40.2 (2017), pp. 344–357.

[61] G. Williams et al. “Information theoretic MPC for model-based reinforcement learning”. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 1714–1721.

[62] K. Yu et al. “More than a million ways to be pushed. a high-fidelity experimental dataset of planar pushing”. In: *IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2016, pp. 30–37.

[63] H. Zhu and J. Alonso-Mora. “Chance-Constrained Collision Avoidance for MAVs in Dynamic Environments”. In: *IEEE Robotics and Automation Letters* 4.2 (2019), pp. 776–783.

[64] C. Zito et al. “Two-level RRT planning for robotic push manipulation”. In: *IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2012, pp. 678–685.

# A Comparison Networks

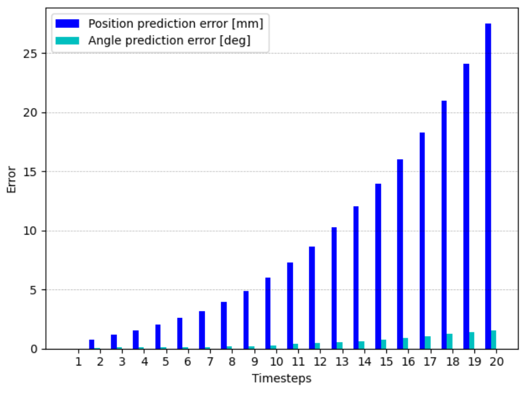
We compare various different architectures for modelling the object dynamics: Long Short Term Memory (LSTM) Networks, Mixture Density Networks (MDN), deterministic and probabilistic neural networks. The networks are compared based on their twenty timestep accuracy and the inference time. First the network's hyperparameters are optimized so it reaches a low validation error. Then, we compare the twenty timestep accuracy on the test set (Figure A.1) and the average inference time for these predictions (Table A.1).

The hyperparameters for training the (ensembles) of deterministic and probabilistic networks are the same as in Table 5.1. For the LSTM architecture, we use four LSTM layers of 128 neurons, followed by a linear layer of 128 neurons. For training we use a learning rate of 0.001 and a batch size of 64. The MDN architecture consists of three fully connected layers of 128 neurons and a mixture model with 35 gaussians. For training, a learning rate of 0.0001 and a batch size of 64 are used. A rough hyperparameter optimization was done to come to these values. Both networks are trained for 1000 epochs, from which we save the model with the lowest validation error.

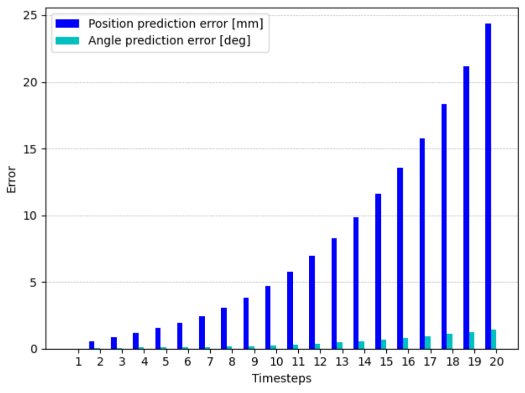
The Mixture Density Network and LSTM architectures have a slightly higher inference time than the deterministic and probabilistic networks. Since the controller needs to sample as many trajectories as possible, it is important that the inference time is as low as possible. We can parallelize the inference for the ensembles, which results in a relatively low increase in inference time when adding models to the ensemble. In addition, the LSTM and MDN have slightly higher error after twenty timesteps (Figure A.1)

**Table A.1:** Comparison average inference time

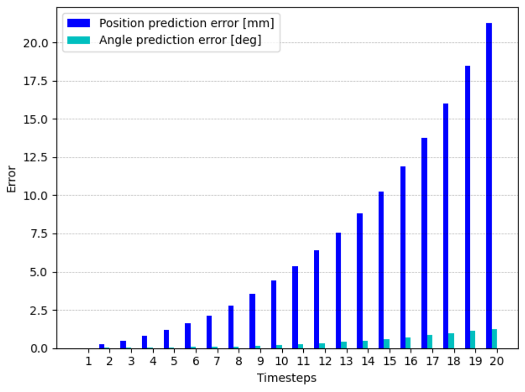
| Network | Average inference time |
|---|---|
| LSTM | 0.000443 s |
| Deterministic NN | 0.000294 s |
| Deterministic NN (ensemble) | 0.000373 s |
| Probabilistic NN | 0.000376 s |
| Probabilistic NN (ensemble) | 0.000396 s |
| Mixture Density Network | 0.000408 s |



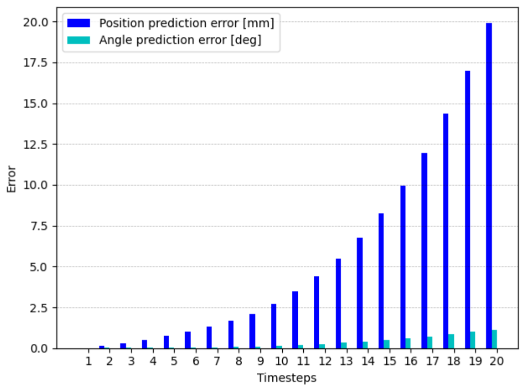
**(a)** Error bar for one deterministic neural network.



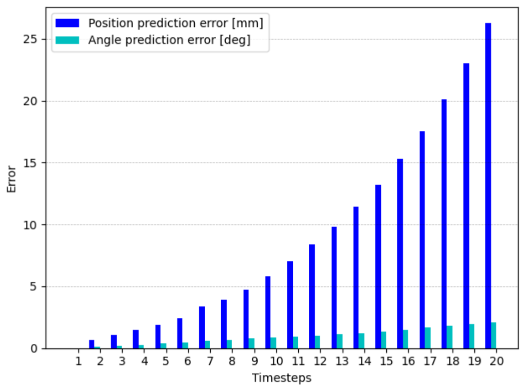
**(b)** Error bar for an ensemble of 3 deterministic neural networks.



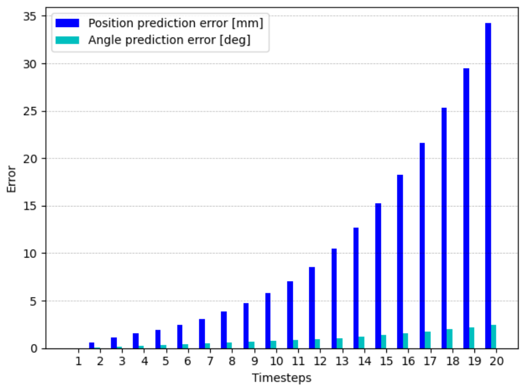
**(c)** Error bar for one probabilistic neural network.



**(d)** Error bar for an ensemble of 3 probabilistic neural networks.



**(e)** Error bar for the Mixture Density Network.



**(f)** Error bar for the LSTM network.

**Figure A.1:** Comparisons of 20 timestep error, for different architectures.

# B Additional Experiments

On top of the experiments with the box, we verify performance of the method on a cylinder-shaped object that can roll off the bumper and test if the learned model of the box generalizes to another box with a different weight, material and a slightly different shape.

## B.1. Cylinder

The cylinder object (Figure B.1) can roll, which makes dynamics more difficult and the object harder to control. We collect 11240 data samples and train a model with twelve ensembles (Table B.1). Figure B.2 illustrates the accumulating error, which is almost double the error for the box model.

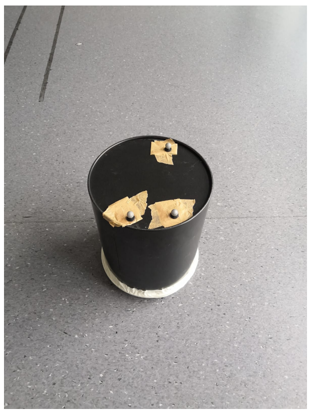


**Figure B.1:** Cylinder object.

**Table B.1:** Hyperparameter values used for training the real-world cylinder model

| Hyperparameter | Values |
|---|---|
| Neurons | 128 |
| Batch size | 64 |
| Learning rate | 0.0005 |
| Layers | 3 |
| Number of ensembles | 12 |
| Non-linear layers | ReLU |

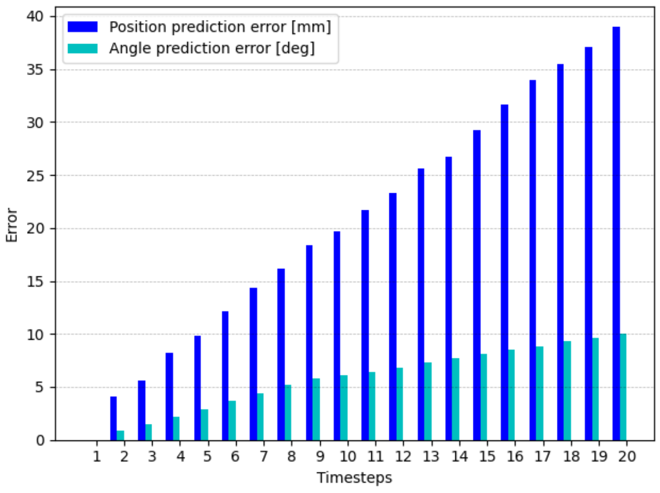
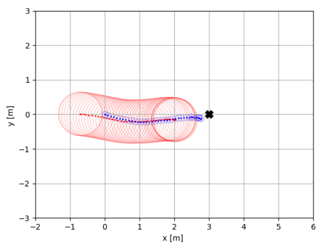


**Figure B.2:** Accumulating error for twenty timesteps, based on real world testing data for the cylinder object.

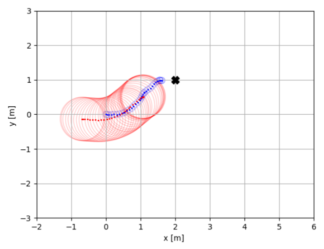
Due to the increased amount of ensembles, we reduce the number of sampled trajectories in the MPPI to eighty. In line with the modelling results, pushing performance declines as well (Table B.2). The overall success rate declines to 44%, as the dynamics model cannot predict well how the cylinder is going to behave. Due to the rolling behaviour, the dynamics are harder to describe and predict. In addition, the system is more unstable, as the cylinder can roll off the bumper (not only slide), which makes the pushing task harder. From the experiments, we observe that the controller cannot always recover if the cylinder rolls too far off the bumper. Figure B.3 shows successful trajectories for each of the goals. These trajectories illustrate that the robot can do successful pushing, when the object's (start) location is at the center of the bumper. However, it cannot cope when the object's (start) location is too far off-center. The cylinder would then roll off the bumper, and the robot cannot compensate for that quickly enough.

**Table B.2:** Success rate, accuracy, pushing time and path length results and accuracy results for pushing with the real robot and a cylinder-shaped object to various goal points.

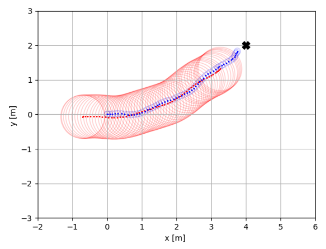
| Goal | Success | Accuracy [m] | Time [s] | Path length$_O$ [m] | Path length$_R$ [m] |
|---|---|---|---|---|---|
| (3.0, 0.0) m | 0.55 | 0.26 $\pm$ 0.16 | 12.2 $\pm$ 0.7 | 2.85 $\pm$ 0.12 | 2.98 $\pm$ 0.16 |
| (2.0, 1.0) m | 0.55 | 0.34 $\pm$ 0.10 | 10.7 $\pm$ 1.0 | 2.17 $\pm$0.24 | 2.34 $\pm$ 0.20 |
| (4.0, 2.0) m | 0.36 | 0.31 $\pm$ 0.05 | 18.9 $\pm$ 1.6 | 4.50 $\pm$ 0.08 | 4.43 $\pm$ 0.08 |
| (5.5, 2.0) m | 0.55 | 0.54 $\pm$ 0.17 | 23.0 $\pm$ 1.0 | 5.83 $\pm$ 0.18 | 5.76 $\pm$ 0.21 |
| (3.0, -1.0) m | 0.36 | 0.33 $\pm$ 0.10 | 12.0 $\pm$ 0.6 | 3.05 $\pm$ 0.21 | 3.02 $\pm$ 0.19 |
| (3.0, -1.5) m | 0.27 | 0.30 $\pm$ 0.20 | 13.9 $\pm$ 0.9 | 3.24 $\pm$ 0.19 | 3.15 $\pm$ 0.19 |



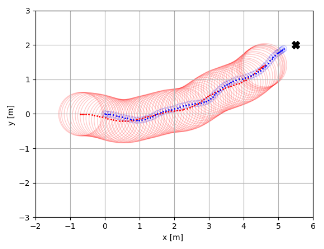
**(a)** Pushing to goal location (3.0, 0.0) m.



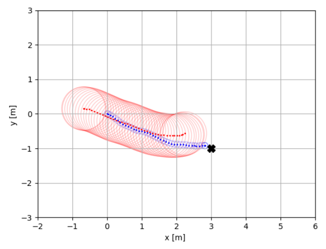
**(b)** Pushing to goal location (2.0, 1.0) m.



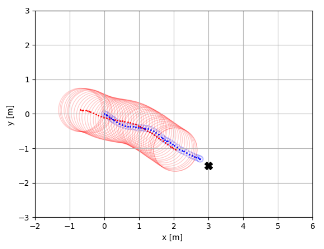
**(c)** Pushing to goal location (4.0, 2.0) m.



**(d)** Pushing to goal location (5.5, 2.0) m.



**(e)** Pushing to goal location (3.0, -1.0) m.

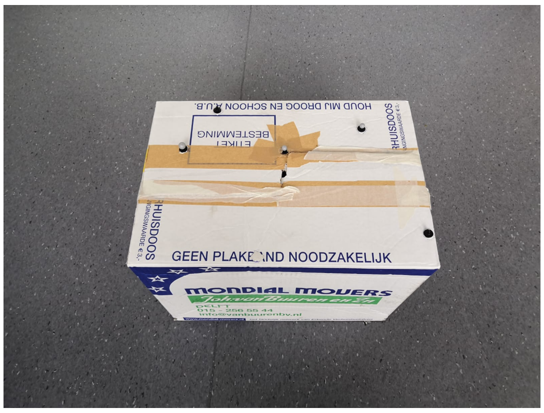


**(f)** Pushing to goal location (3, -1.5) m.

**Figure B.3:** Examples of pushing trajectories for pushing with a cylinder-shaped object.

## B.2. Extra box

Lastly, we test the pushing performance for the trained model on another box, with a different material and slightly different shape (Figure B.4).



**(a)** Box used for collecting training data.

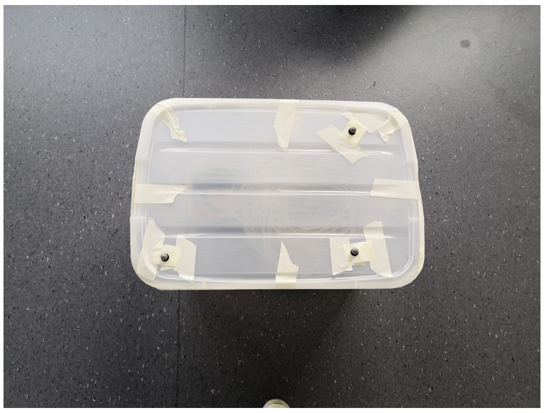
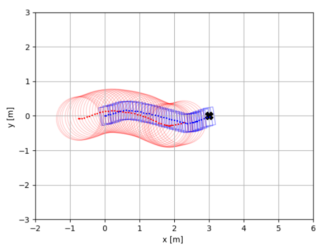

**(b)** Box used for testing.

**Figure B.4:** The paper and plastic boxes.

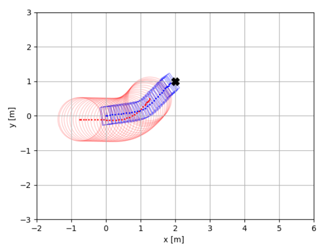
We show that the model still makes meaningful enough predictions to push the object with a 87% success rate and comparable accuracy (Table B.3). Figure B.5 shows successful trajectories for each goal location with the extra box. We show that even though the geometry and material of the boxes are different, the model does not collapse and can make meaningful predictions. In addition, the MPPI controller can compensate for inaccuracies of the model, resulting in successful pushing. It shows us that the method is robust to slightly different conditions, and can cope with uncertainties.

**Table B.3:** Success rate, accuracy, pushing time and path length results for pushing with the real robot and an extra box-shaped object to various goal points.

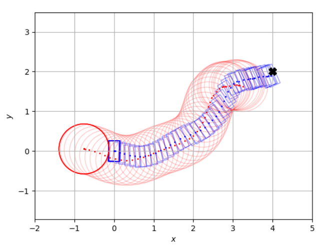
| Goal | Success | Accuracy [m] | Time [s] | Path length$_O$ [m] | Path length$_R$ [m] |
|---|---|---|---|---|---|
| (3.0, 0.0) m | 1.0 | 0.26 $\pm$ 0.17 | 13.8 $\pm$ 1.3 | 3.17 $\pm$ 0.30 | 3.27 $\pm$ 0.35 |
| (2.0, 1.0) m | 1.0 | 0.31 $\pm$ 0.15 | 10.7 $\pm$ 1.8 | 2.47 $\pm$ 0.31 | 2.55 $\pm$ 0.33 |
| (4.0, 2.0) m | 0.91 | 0.32 $\pm$ 0.15 | 20.1 $\pm$ 1.5 | 4.88 $\pm$ 0.38 | 5.01 $\pm$ 0.35 |
| (5.5, 2.0) m | 0.91 | 0.50 $\pm$ 0.15 | 23.4 $\pm$ 1.2 | 6.02 $\pm$ 0.19 | 6.24 $\pm$ 0.36 |
| (3.0, -1.0) m | 0.55 | 0.27 $\pm$ 0.11 | 13.4 $\pm$ 1.4 | 3.32 $\pm$ 0.28 | 3.42 $\pm$ 0.34 |
| (3.0, -1.5) m | 0.82 | 0.32 $\pm$ 0.24 | 15.0 $\pm$ 1.1 | 3.50 $\pm$ 0.21 | 3.60 $\pm$ 0.25 |



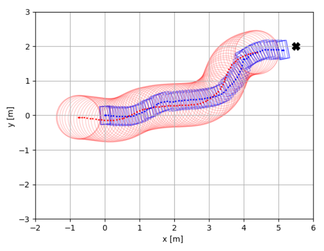
**(a)** Pushing to goal location (3.0, 0.0) m.



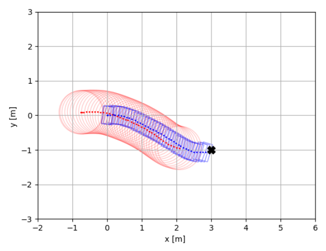
**(b)** Pushing to goal location (2.0, 1.0) m.



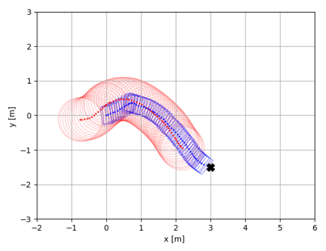
**(c)** Pushing to goal location (4.0, 2.0) m.



**(d)** Pushing to goal location (5.5, 2.0) m.



**(e)** Pushing to goal location (3.0, -1.0) m.



**(f)** Pushing to goal location (3, -1.5) m.

**Figure B.5:** Examples of pushing trajectories for pushing with the extra box.

# C
# Code

The code that was developed for this thesis can be found in the following Github repositories:

```
https://github.com/SusanPotters/mocap_experiment_push_control
https://github.com/SusanPotters/learning_pushing_control
```

The first link refers to the code used for the experiments in the motion capture lab. It contains the ROS packages described in Section 4.3.2. The second link refers to the code for simulated experiments.

The code for the simulation baselines can be found in the following repositories:

```
https://github.com/SusanPotters/adaptive_pushing_control
https://github.com/SusanPotters/sac_pushing_control
```

The repositories are currently private, if you would like access contact me.