Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Predictive Planning with Self-Explored Push Dynamics

proposed by

Lars Henning Kayser

M.Sc. Informatik, Mat-Nr. 6314876
1kayser@informatik.uni-hamburg.de

on August 24, 2018

Faculty of Mathematics, Informatics and Natural Sciences
Department of Informatics
Master of Science Informatics

First Supervisor: Prof. Dr. Jianwei Zhang
Second Supervisor: Dr. Matthias Kerzel

Abstract

Humans use a variety of motor skills without understanding the underlying physics. Throwing a ball seems just as easy as pushing objects around on a table. The former can be described by simple mechanics. The latter is often indeterminable. Aspects like friction distribution and certain surface properties are practically unmeasurable. This uncertainty makes pushes hard to predict and difficult for robots. This approach implements goal-directed pushes based on prediction models. The robot collects samples by randomly pushing target objects. The data is used to train forward models that implicitly learn the friction coefficients. These models are applied in motion planners to generate collision-free push plans. However, the plans are not executable in an open-loop fashion. Accumulation of setup and prediction errors often lead to deviating object paths. This can be avoided by a closed-loop execution using Model Predictive Control (MPC), as shown with different planning strategies. The best performing strategy enforces subsequent pushes to be similar, to smooth the trajectory.

Zusammenfassung

Menschen besitzen motorische Fähigkeiten, ohne Kenntnis der zugrundeliegenden Physik. Bälle zu werfen, scheint etwa so einfach, wie Gegenstände über den Tisch zu schieben. Ersteres lässt sich ballistisch beschreiben. Letzteres ist nahezu unlösbar. Der Grund ist, dass Reibungskräfte und -koeffizienten praktisch nicht bestimmbar sind. Schieben ist daher eine große Herausforderung für Roboter. Diese Arbeit beschreibt zielgerichtetes Schieben mithilfe autonom gelernter Modelle. Der Roboter sammelt Daten, indem er Gegenstände zufällig anschiebt. Die daraus gelernten Dynamikmodellen beschreiben implizit die Reibung. Diese werden zum Planen von kollisionsfreien Schiebebewegungen verwendet. Die Pläne sind jedoch nicht direkt ausführbar. Fehler im Modell oder Umgebung lassen den Gegenstand vom Zielpfad abweichen. Per *closed-loop* Steuerung mittels *Model Predictive Control* (MPC) kann dies verhindert werden. Dieses wird am Beispiel unterschiedlicher Planungsstrategien gezeigt. Für gleichmäßige Trajektorien erweist sich das Planen mit aufeinanderfolgend ähnlichen Schiebebewegungen am geeignetsten.

Contents

Tables

Figures

Algorithms

1 Introduction

The interest for autonomous robots in industry and consumer market is widely focused on adaptability. Basic robot skills are desired, such as grasping, moving or throwing of arbitrary objects. These tasks require an abstract model of the problem and are solved either analytically or by approximation. The ballistic trajectory of a thrown object is analytically solvable given weight and velocity of an object at release. An example of a problem where no exact analytical solution exists is that of pushing objects on a flat surface. While humans are able to intuitively move any item by means of the finger tip, the exact process is indeterminate. The reason is that pushes rely on friction models and object properties that are practically unmeasurable. There are analytical push models, but even if all required information was available, the execution would still be unstable. The slightest errors in the model, setup calibration or robot control severely affect the results. This is why goal-directed and collision-free pushing is still a challenging problem.

The proposed approach attempts to learn forward push models for predictive manipulations. More specifically, the approach consists of the following steps:

1. Autonomous exploration of push effects
2. Learning of a push prediction model
3. Applying the model for push planning
4. Closed-loop execution of the plans

1.1 Motivation

An interesting pattern with human motor skills is the ability to predict complex physical processes without realising it. When pushing an object on a table the movement is controlled by sight and force sensation. During the movement, we might adjust the push direction online to counteract undesired drifting of the object. Even non-rigid objects can be manipulated easily that way.

That suggests that it is possible to learn implicit models of the physical process by observation. This work intents to realize push capabilities by automated self-learning. The robot should autonomously explore different pushes on a target object to collect data that can be used for training prediction models. Ultimately, the robot should be able to move an object towards a goal without colliding to obstacles. That problem resembles control-based motion planning, where pushes are the controls and the object trajectory the motion. The approach describes planners with different exploration strategies using the prediction models for goal-directed control sampling. The resulting plans are correct,

however unfeasible to execute in an open loop. Therefore, a closed-loop model predictive control approach is applied in order to realize goal-directed push movements.

1.2 Related Work

In robotics, push manipulations have been researched extensively for over 30 years. There exist many approaches with different objectives and applications that try to solve this task. Two terms are often used to describe the corresponding scenario:

(Quasi)-Static pushing: The velocities of the pusher are low so that effects of inertia are neglected.

Stable pushing: The object is stabilized by using restricting pusher shapes or wider contact regions

The following approaches are separated into analytical and predictive approaches. This is only an approximate classification based on the corresponding focus. Several works use hybrid methods or are additionally based on physical simulation.

1.2.1 Analytical Approaches

Mason et al. [Mas86] describe the theoretical foundations of push mechanics. They show that the instant movements of an object can be calculated from the support friction distribution of the object and the velocity or force vector of the pusher. They also present analytical algorithms for computing the translation and rotation of an object.

Lynch et al. use these fundamentals to approximate the objects center of rotation [Lyn92]. They apply this model in a planning algorithm for stable push operations [LM96]. The planner is based on exploring small pushes in a neighborhood search.

Ruiz-Ugalde et al. [RCB11] present a learning architecture that predicts the center of rotation given the push force vector. The support friction distribution is learned implicitly, however limited to primitive shaped objects. They show that their model is applicable to find rotation centers for sampling directed pushes.

1.2.2 Predictive Approaches

Salganicoff et al. [Sal+93] present a learning method for predicting the object rotation. A simple control policy minimizes the angle between a path towards a goal which allows fast and smooth trajectory executions. However, the policy is limited to stable pushes.

Walker et al. [WS08] conceptualize a push learning architecture for non-primitive objects. Their approach is to model the objects shape with splines and distribute push points around it. A push point maps the surface normal to an object movement from orthogonal pushes. Goal directed pushes are realized by sampling the contact points. The approach is limited to round objects and does not generalize to objects of different shape or size.

Lau et al. [LMI11] present a learning-based approach where a mobile disk shaped robot pushes objects towards a goal. The learning mechanism relies on letting the robot explore random pushes. A push policy minimizes the distance between object and goal by sampling adjacent pushes close to the contact point. If a push maximizes the goals distance, a new contact point is sampled to rearrange the movement. This method appears to be efficient even though the orientation of the object is neglected.

Hermans et al. [Her+13] introduce a data-driven learning method similar to Salganicoff et al. [Sal+93]. Their approach is to learn push operations as a function of the object shape by using locally weighted kernel regression. That allows using irregular objects like a camcorder, a brush or toothpaste. They also introduce a specific score to make stability and accuracy of pushes more comparable.

Yang et al. [YSG14] compare different neural network architectures for predicting push effects from images.

Byravan et al. [BF16] present a deep learning architecture that models the dynamics of the robot and its environment. Their implementation of SE3-Nets - specialized networks for 3-dimensional Euclidean transformations - is capable of predicting any movement based on learned physics. In addition, the output of the system is used to produce 3D images of the approximated scene.

Agrawal et al. [Agr+16] apply a deep learning architecture for learning push policies from raw images. They use Convolutional Neural Networks (CNN) to learn the effect of random pokes (short pushes) with arbitrary objects based on depth images. The models are used to sample goal-directed pokes to move an object to a desired position and orientation.

Kopicki et al. [Kop+17] present a machine learning method for predicting object movements in 3D. The object shape is modeled by multiple transforms, where each is used to train separate predictors. The object movement is computed by a probabilistic density estimation over all predictions. They show that their models are transferable to different objects and also prove that their method enables interpolation between push actions.

1.3 Outline

Chapter 2 introduces the theoretical and conceptual foundations of this work. That includes basic push mechanics, machine learning with neural networks and sampling-based motion planning. The approach is described in chapter 3. Starting with a push specification, this chapter describes sample exploration, prediction, planning and control methods. The robot setup and implementation of said methods are described in chapter 4. An analysis of the experiments and results are given in chapter 5. Chapter 6 interprets the findings of this work and the implications for this research field and applications.

2 Theory and Background

Section 2.1 introduces a theoretical model for pushing and describes its limitations. On the basis of this model assumptions are made regarding the push specification and execution in this work. Section 2.2 introduces the methods used for push prediction. This includes an outline about neural networks and the automated process of hyper-parameter optimization. The predictive planning approaches are using sampling-based motion planners. Details about exploration methods and implementation are given in section 2.3.

2.1 Push Mechanics

All pushes in this approach are executed on a 2D surface and follow the quasi-static assumption. That is, the velocities of the push movements are so slow, that effects of inertia can be neglected. A theoretical model for quasi-static pushes is given by Mason et al. [Mas86], which is also the source for the concepts described in this section.

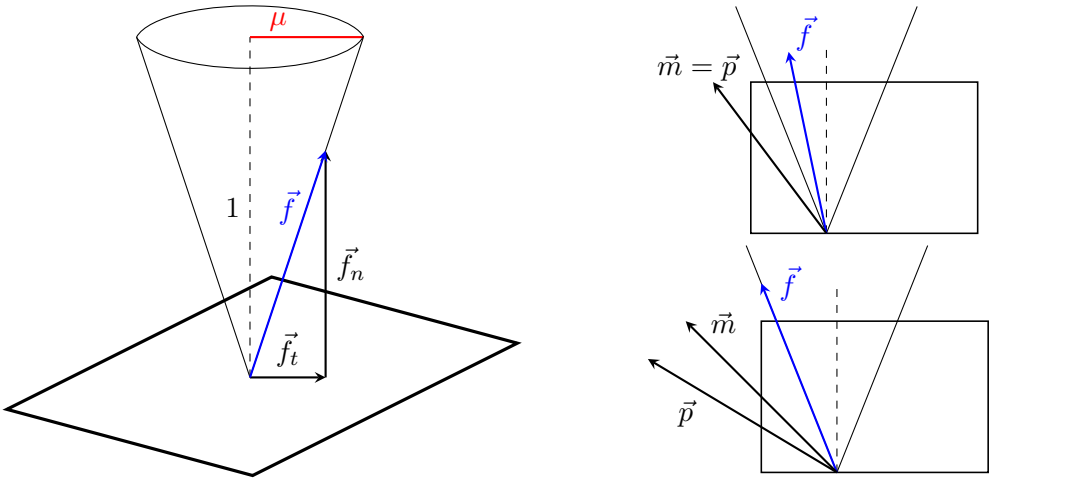
When pushing, the object moves as a result of all applied forces. These are gravity, the push contact force and occurring friction forces between the object and the pusher as well as the surface. In order to understand push effects, the friction forces must be defined.

Elementary laws of friction were first stated by Amonton and later by Coulomb. Contact forces between objects are described as tangential and normal forces. Tangential forces are the result of friction and occur opposed to lateral forces, for instance, when sliding an object on a surface. The relation between tangential and normal force has a geometrical representation, the friction cone depicted in figure 2.1a. The cone is a friction model of a single contact point on a surface. Friction force $\vec{f}$ is composed of tangential force $\vec{f_t}$ and normal force $\vec{f_n}$. The normal force $\vec{f_n}$ corresponds to the pressure at the contact point, for instance, the weight of an object on a table. The tangential $\vec{f_t}$ occurs opposed to lateral forces and proportional to $\vec{f_n}$ by a factor μ. This factor μ is the friction coefficient which depends on the materials of the objects in contact. There are two different instances of μ, according to whether the point is sliding on the surface or not. During motion μ corresponds to the *dynamic friction coefficient*, otherwise to the *static friction coefficient*. Both coefficients describe the same forces in different scenarios, and can actually have different values. An example is a sled standing on a snow-covered surface. If the sled is packed with heavy weights, it can be hard to bring it into motion. Once it is moving, it's usually no problem to accelerate it. The reason is the static friction coefficient is higher than the dynamic coefficient. As an effect, the opposing tangential force is higher when the sled is standing.

The static friction coefficient defines the force that can be applied from the side without moving an object. The friction cone then includes all possible force vectors that don't

result in changing the contact point. If the lateral force exceeds $\vec{f_t}$, movement occurs. In that case, μ becomes the dynamic friction coefficient which defines $\vec{f_t}$ during sliding. When sliding, the outer hull of the cone spans the space of possible force vectors $\vec{f}$. Interestingly, the occurring friction forces are unrelated to the sliding velocity.

The friction cone model can be applied to determine the contact force between pusher and object surface. For simplicity, it is assumed that the dynamic and static friction coefficients are equal. Figure 2.1b shows how the contact force is directed during pushing. The upper image depicts a push where the pusher has a stable contact point. The lower image shows how the contact force is directed when the pusher slides along the object during the movement. $\vec{p}$ is the movement of the pusher, while $\vec{m}$ is the movement of the contact point on the object. If the contact force is directed within the friction cone, then $\vec{p}$ and $\vec{m}$ are aligned so the pusher sticks to the contact point. If the contact force is at the limit of the friction cone, the pusher slides along the surface during the movement. In that case, $\vec{p}$ and $\vec{m}$ are deviating.



(a) Friction Cone

(b) Direction of contact force on the object

Figure 2.1: Friction cone and contact forces when pushing [Mas86]

Using this friction model, translation and rotation of an object can be described, given the following properties:

- Friction coefficients between surface and object
- Friction coefficients between object and pusher
- Pressure distribution of the object on the table

Translation

Pushes that result in linear translations of the object can be described analytically. The friction forces between object and surface can be reduced to a single force at the *center*

of friction (COF). The COF is the centroid of the pressure distribution if the object is homogeneous. All pushes where the force vector points through the COF result in linear movements of the object. Since the force vector aligns with the movement if the contact point is stable, this method also applies to the motion vector.

There is no analytical solution for non-linear translations. However, they can be determined as part of rotating movements.

Rotation

A rotating movement is described by its path around an instantaneous center of rotation (COR). It can be solved numerically, given the push contact point and the push direction angle. The procedure to retrieve the COR exceeds the scope of this introduction. A detailed description can be found in the main resource, Mason et al. [Mas86].

The method relies on creating a continuous function of all rotation centers at a contact point parameterized by the direction angle. The instant rotation center can be determined geometrically, by plotting the function along with the object as shown in figure 2.2. L and R represent the edges of the friction cone and $\vec{p}$ is the push direction vector. The COR candidate $C_{\vec{p}}$ lies at the crossing between the function and a vertical line to $\vec{p}$. However, the range of feasible rotation centers is delimited by C_L and C_R, corresponding to the friction cone. If the candidate is outside of this range, the nearest border is the result. The friction cone limits the tangential force and surplus lateral forces only lead to slippage.

Analogous to translations, the rotation can be determined by the motion vector if the contact point is stable.

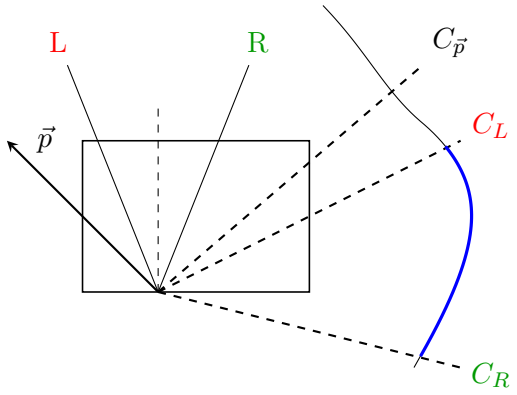


Figure 2.2: Center of Rotation [Mas86]

Limitations and Assumptions

As stated, all solutions require exact values of friction coefficients and the pressure distribution of the object. These are generally unmeasurable, except the object is primitive and homogeneous. If those properties could be measured, the solutions for translation

and rotation would still be inaccurate. The methods only apply to pushes that are performed at surface level. Pushes above the surface change the pressure distribution of the object and render the friction forces indeterminable. The slightest irregularities in the object shape, support surface or push control would invalidate the results.

There are further considerations involving the robot setup. The robot in the approach setup is position controlled. The joint controllers actuate movements with approximate velocity and acceleration targets. There is no control over the applied forces. Force control could be emulated by using a force torque sensor. The sensor needs to be exactly calibrated and should offer an update rate that matches those of the joint controllers. However, it's questionable if the accuracy would be sufficient for push controls. Another problem is the stabilizing behavior of the joint controllers. If an external force presses against the manipulator, the joint controllers correct deviating joint positions. For pushing this means the applied contact force affects the end effector trajectory. Depending on the weight of the object, this results in slight jiggling of the pusher, especially when pushing heavy objects. Additionally, bending of the pusher or object can aggravate this problem.

It's not practicable to consider these issues in the push method. Based on the limitations of the robot following assumptions are made:

1. The pusher movement is linear with continuous velocity
2. Pusher and object are in steady contact during a push
3. The pusher movement directs the contact force
4. The contact force is continuous

Assumption (1) states that jiggling or deviations of the end effector can be neglected. (2) is an extension of the quasi-static assumption regarding (1). Assumption (3) directly follows from (1) and (2) and implies that push movements produce repeatable results. Lastly, (4) is derived from the continuous velocity assumption. If the contact force was not continuous, the resulting movements were indeterminate.

The assumptions state that position controlled pushes actuate sufficiently stable and reproducible forces. Even if the push controls are not fully precise, the effects should be negligible compared to other factors like object localization error or calibration offsets.

2.2 Learning and Prediction

As stated above, push effects are highly dependent on the friction coefficients. These are practically immeasurable, they could, however, be approximated by analysing the outcome of different pushes. For instance, the center of friction could be determined easily by a simple strategy. If multiple pushes are known to produce linear translations, the COF coincides with the intersection of the push directions. Similarly, the COR could be measured and used for approximating the friction distribution.

Therefore, it is feasible to treat the friction model as a black box and focus directly on the outcome. This work uses neural networks to implicitly learn a friction model by training push effects. The considered and used methods and concepts are introduced in this section. That includes the hyper-parameter optimization process, an automated method for configuration tuning.

2.2.1 Neural Networks

Many approaches towards learning push effects have been proposed. They vary in problem specification and scenario and are thus using different learning architectures. Depending on the approach these include Multi Layer Perceptrons (MLP), Recursive Neural Networks (RNN) or Convolutional Neural Networks (CNN). RNNs are used to learn a sequence of actions where an inner state is considered. An example is the learning of push control policies for following a trajectory. CNNs are often applied for end-to-end learning, where the camera image is directly used as input. Both architectures are applied to problem scenarios that exceed the mere approximation of friction models.

This work attempts to approximate only immeasurable push features in order to realize predictive planners. Since pushes fulfill the Markov assumption, ordinary MLPs are used. In order to increase prediction accuracy and robustness, different structures and configurations are considered. Below is a short introduction into the used concepts.

Normalization

Input and output features can have arbitrary shapes and value distributions. Normalizing the values to a certain range usually improves the learning results. Typical ranges are [0,1] or [-1,1]. Depending on the value distribution, different normalization methods are preferred. Min-Max normalization is a scaled mapping between source and target ranges. This is usually applied for uniform value distributions. Normal distributions are preferably normalized by methods using the standard deviation, like Z-Score.

Activation Functions

In this work the Linear, Rectifier (*relu*) and Tangens Hyperbolicus (*tanh*) activation functions are considered. All of these represent continuous activations, fitting to the expected push features and learning problem. For this reason, other activations like Softmax, Softplus or Softsign are not applied.

Optimizers

An optimizer defines learning behavior and is chosen according to the training data. The proposed architectures use RMSProp [TH12], Adam and its variant Nadam [Rud16]. They are assumed to excel especially in small datasets compared to ordinary stochastic gradient descent (SGD). The difference between the optimizers is defined by their momentum.

Regularization

Regularization methods are approaches to prevent overfitting in the network. If a network is too expressive for certain features, undesired noise or systematic errors are learned as well. Dropout regularization partially disables the activation of different layers during training. That enforces redundant feature representations and balances the weight distribution. Other regularization methods used in this work are L1 and L2 regularization.

Loss Functions

The loss function defines the prediction error and is used as a learning objective. Typical loss functions are *mean squared error* or *LogCosh*. They compute a common error value given the difference between target and prediction result. Some learning problems require weighted loss functions, in particular, if the output features require different accuracies. A weighted loss function is also applied in the push predictors introduced in the method section 3.3.

Training and Validation

There are different strategies and protocols on how to train neural networks. In general, the training is run in multiple epochs with intermittent validation phases. The training itself is run by iterative steps or in batches. Batch training involves applying groups of training data at the same time, smoothing the learning curve. Iterative training is more prone to local minima, it can, however, be applied online. This is also a reason why all presented networks are using iterative training protocols.

2.2.2 Hyper-parameter Optimization

Hyper-parameter optimization is the automated search for architecture and learning parameters in order to increase prediction accuracy. These parameters can apply to all aspects of the model, like number of layers in a neural network or regularization methods. The optimization algorithm stochastically samples configurations from a defined parameter space and applies them to new models. The model with the minimal loss defines the best parameter configuration.

A configuration space is defined as a set of probabilistic value ranges of the parameters. These are stochastic expressions of continuous numbers, integers or arbitrary types.

Choice: A uniform distribution over a defined set of arbitrary expressions.

Random Integer: A random integer between 0 and an *upper* value.

Uniform Distribution: A uniform distribution of real values in a range between given min and max values, denoted as $U(min, max)$.

Normal Distribution: A normal distribution of real values defined by mean μ and standard deviation σ, denoted as $N(\mu, \sigma)$.

Log Distributions: Computes the exponent of a normal or uniform distribution.
For instance, log uniform corresponds to the function exp(uniform(min, max)).
Here the logarithm of the return value is uniformly distributed.
This is analogous to the definition for the log normal distribution.
A log distribution is denoted as $logU(min, max)$ and $logN(\mu, \sigma)$ respectively.

Quantized Distributions: All distributions with or without log can be quantized.
All resulting values are rounded by a given a step size.
For example, a step size of 1 means all values are rounded to integers.
Quantized distributions are denoted as $qU(min, max, step)$ or $qlogU(min, max, step)$ for uniform distributions. The notation for normal distributions is defined analogously.

2.3 Sampling-based Motion Planning

In robotics motion planning methods are used for computing collision-free robot movements. The planning problem is defined as finding a sequence of robot controls that result in a continuous path from start to goal state. A collision-free path corresponds to a path of collision-free states. The states of a robot conform to its joint space while the controls are instructions for how the joints should move. Since the state space grows exponentially with the number of joints, planning algorithms can become very costly.

Sampling-based Motion Planners are efficient methods for solving this problem. In contrast to other approaches like *Cell Decomposition* or *Potential Fields*, their focus relies on randomly exploring the complex state space by growing a graph representation. This is done by sampling collision free states and connecting them following a certain heuristic. The planning problem is solved if start and goal state are connected by the graph.

There are many methods on how the graph is grown and what states are sampled and connected. Prominent algorithms are PRM, KPIECE and RRT. All three of them exist in many variations.

PRM aims at mapping the collision free state space by an equally distributed graph. This is especially useful in applications where multiple planning attempts are queried.

KPIECE and RRT are both tree-based planners, meaning they are growing acyclic graphs beginning at the start state. The exploration is biased towards regions that are sparsely represented or around the goal state. That makes them very efficient for single-query scenarios, since the tree only consists of path candidates. This is one of the reasons why RRT is used in the scope of this work.

Described planners and all variants presented in the method section are implemented using the Open Motion Planning Library (OMPL) [PC18]. Terms and concepts of OMPL are therefore briefly introduced in section 2.3.1 below. A sample implementation of control based RRT described in section 2.3.3.

2.3.1 Planning Concepts in OMPL

OMPL is a collection of motion and control planning implementations for robotics. It offers convenient interfaces for generic planning problems and uses abstractions for different planner components. These components define how the algorithms work in detail, particularly, they affect the objective and exploration strategy. The following concepts are widely used in OMPL planners, and are further referenced in the method section 3.

State Space

The state space consists of a set of value range specifications that define all valid states. In addition, a unified distance metric is defined over the space, so that proximity between states can be compared. There are multiple predefined state spaces, as for instance, the SE(2) space representation for object poses used in this work.

State Sampler

The state sampler generates random states that are used for exploration. This incluedes goal biased sampling, which is used in all tree-based planning algorithms.

Control Space

The control space is a set of value ranges that define the space of applicable control instructions. Controls along with corresponding durations initiate a transition from one state to another. The actual transition method is defined by the *state propagator*.

Control Sampler

The control sampler is used for sampling or generating controls. Depending on the planner, different kinds of control samplers are used. Based on the functionality the samplers can be divided into three groups.
1. *Random Control Sampler:* Uniform random sampling of control values
2. *Directed Control Sampler:* Sampling of a control that minimizes the distance between a start and a target state.
3. *Constrained Control Sampler:* Sampling of controls based on arbitrary constraints on the control values

State Propagator

The state propagator defines how state transitions are performed. By default, it computes a successor state given start state, control and duration. The *steered state propagator* is a variant that computes control and duration to get from one state to another.

State Validity Checker

The state validity checker is used to define whether a given state can be used for exploration. This is used to check state space bounds, object collisions or other arbitrary conditions.

2.3.2 Planner and Configuration

In addition to the described concepts, the planner is configured by a set of parameters. These play a role for exploration as well as solution accuracy.

goal distance: Threshold distance that defines if the goal is reached.
goal bias: Ratio of states that are sampled at the goal.
min/max control duration: Defines the range of the control duration sampling.

Some planners explore new states by repeatedly applying controls as long as the successor states are valid. Two additional parameters define this behavior:

propagation step size: Fixed duration that is used for repeated control step execution.
set intermediate states: Defines if visited states are added to the exploration graph.

2.3.3 Implementation of RRT

RRT uses a uniform state sampler with goal bias and a directed control sampler. The path objective is defined by start and goal states as well as a goal distance threshold. At the beginning, the planning tree is initialized with start state as root. The exploration is run in a loop of sampling random states and propagating directed controls towards them. Algorithm 2.1 is a simplified version of the exploration process.

Algorithm 2.1: Simple Control RRT

```
1  simpleControlRRT(start, goal, threshold=0.05) {
2
3    planningTree = initializeTree({start})
4
5    while (true) {
6      s_rand = sampleStateWithBias()
7      s_near = planningTree.findNearest(s_rand)
8      control = sampleDirectedControl(s_near, s_rand)
9      s_next = propagate(s_near, control)
10
11     while (isValid(s_next)) {
12       planningTree.connect(s_near, control, s_next)
13
14       if (getDistance(s_next, goal) < threshold)
15         return true
16
17       s_near = s_next
18       s_next = propagate(s_near, control)
19     }
20   }
21 }
```

3 Method

This chapter gives a detailed description of the methodology of the approach. In section 3.1 the applied pushes are specified and general terminology introduced. Section 3.2 describes the process of collecting push samples. That includes the execution, data structures, target objects and sampling protocols. Section 3.3 contains design and optimization process of different push prediction models. The predictive planners and closed-loop control methods are described in section 3.4.

3.1 Push Method

There are plenty of different ways on how objects can be manipulated by pushes. The shape and number of contact points and the motion of the pusher defines how the object moves. This approach is restricted to single contact point quasi-static pushing, as described in the theory section 2.1. Also, the pusher motion is always linear regarding the world or surface frame.

This detail is important, since curved push movements are justified as well for executing pushes with rotations. Consider pushing an object around in a circle. The center of the circle aligns with the center of rotation which is fixed regarding the object frame. This movement can be created by a continuous sequence of the same push. Being able to perform this movement is a feasible requirement, which is limited by the accuracy of the prediction.

It's not feasible to predict exact push effects because of the following problems:

- Friction coefficients between object and surface are unknown
- Friction coefficients between pusher and object are unknown
- Pressure distribution of the object is unknown
- The push can only be applied above surface level, rendering the pressure distribution indeterminate

The first three properties are necessary to determine all acting forces which define the movement of the box. But even if they were measurable, the last point would invalidate any analytical solution.

Linear pushes offer the advantage that they reduce the dimensionality of the problem. It allows comparing the same pushes with different distances, and might even enable interpolation. That way circular movements could be approximated by a sequence of very small linear pushes.

Push Specification

Corresponding to the problem scenario, the target object is rigid and positioned on a surface. The object pose P is defined by a frame in reference to the surface frame. The Z-axes of both frames align with the surface normal of the table. The object frame is defined in SE(2), composed of a 2D position and an orientation around the normal. Therefore, an object pose is represented by three values P_x, P_y, P_γ.

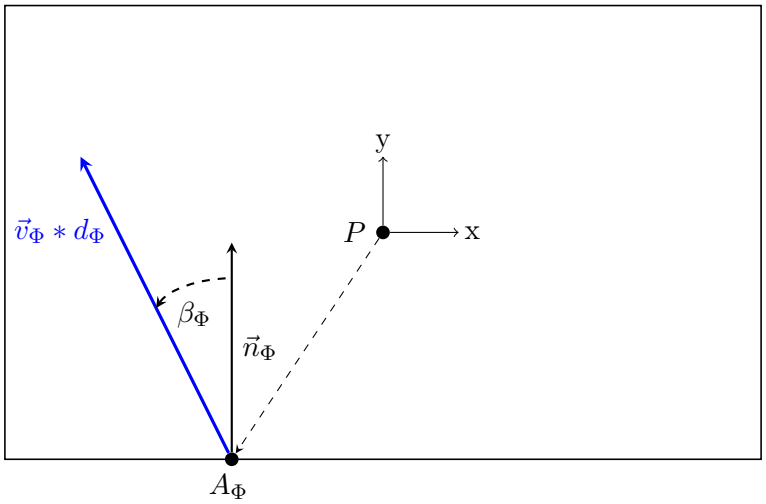


Figure 3.1: Illustration of the Push Specification

A push Φ is specified in reference to the object frame. The contact point is called *approach point* A_Φ and is a 2D position that aligns with the border of the object shape. The *approach normal* $\vec{n}_\Phi$ is the inwards directed normal vector of the object shape surface at A_Φ. This vector works as a reference for the *push direction* vector $\vec{v}_\Phi$. To simplify discussion and visualization $\vec{v}_\Phi$ is also represented as *push angle* β_Φ. β_Φ is the angle between $\vec{v}_\Phi$ and $\vec{n}_\Phi$. Finally, the *push distance* d_Φ defines the distance of the actual movement of the pusher. Figure 3.1 illustrates the introduced terms in an example.

Terminology

approach point	A_Φ
approach normal	$\vec{n}_\Phi$
push direction	$\vec{v}_\Phi$
push distance	d_Φ
push angle	β_Φ

3.2 Sample Collection

The collection of push samples is run autonomously by the robot. The process consists of repeatedly executing random pushes and saving the results, which are the object poses before and after the execution as well as the push specification itself. The structure of the data is described in 3.2.1. Additionally, each push is documented by multiple camera snapshots for post processing and analysis.

The collected data is separated into multiple datasets, each defined by different exploration protocols. These vary in the used sampling methods and target object and are described in 3.2.2.

However, it is not feasible to run a fully random exploration process. At some point, the object might be pushed off the table or out of reach so that user intervention is required. This is prevented by a safety measure that affects all sampling methods which must be considered when evaluating the data. Safety measure and potential impact are described in 3.2.3.

3.2.1 Sample Structure

During the exploration process information for each successful push sample is recorded. Each record contains two types of data: object locations and executed push. Object locations are saved at timestamps before and after the push execution. The data is structured following the specification from section 3.1. That is a push $\Phi : P \rightarrow P'$ is recorded as:

Push Φ*:* A_Φ, $\vec{n}_\Phi$, $\vec{v}_\Phi$, d_Φ
Start Pose P*:* P_x, P_y, P_γ
Result Pose P'*:* P'_x, P'_y, P'_γ

Camera images are saved for later review of single records. The snapshots are taken at three times per push: one before, one at contact and one after the execution. This should allow detecting failed attempts and possibly skewed results as well. Examples of the snapshots are shown in section 4.1.2.

3.2.2 Exploration Protocols

The exploration process is run in four epochs. The respective protocols are shown in table 3.1. They describe the value ranges used for push sampling and the applied target object. All random values are uniformly distributed, denoted by $U(min, max)$.

Push approach A_Φ, approach normal $\vec{n}_\Phi$and push direction $\vec{v}_\Phi$are sampled equally in all protocols. Given the shape of the object, the border is mapped to a value range from 0 to 1. This range is used to sample a random A_Φ. The value of $\vec{n}_\Phi$ is determined by A_Φ and the object shape.

The push direction $\vec{v}_\Phi$ is set by sampling the angle representation β_Φ. The angle ranges between -0.5 and 0.5 rad which is approximately +-29°. The limits should keep the pusher from slipping too much from the contact point.

Variable	Protocol 1	Protocol 2	Protocol 3	Protocol 4
A_Φ, $\vec{n}_\Phi$	$U(0,1)$ * shape	$U(0,1)$ * shape	$U(0,1)$ * shape	$U(0,1)$ * shape
$\vec{v}_\Phi(\beta_\Phi)$	$U(-0.5,0.5)$ rad	$U(-0.5,0.5)$ rad	$U(-0.5,0.5)$ rad	$U(-0.5,0.5)$ rad
d_Φ	3 cm	$U(0.5,3)$ cm	$U(0.5,3)$ cm	$U(0.5,3)$ cm
object	A	A	B	C

Table 3.1: Push Exploration Protocols

Protocol 1 has a fixed push distance of 3cm. The intention is to have a reference dataset for separate analysis of distance and push approach. All other protocols define a sampling range from 0.5 to 3cm. The minimum matches the approximate accuracy of the setup. The maximum is set to prevent the result distribution from becoming too sparse.

Another difference between the protocols is the object version. All experiments are run with the same object shape, cardboard box with dimensions 23cm x 16.2cm x 11.2cm. Object A is the empty box, and B and C are modified versions with weights inside. The weights at the corners of the box so that the pressure distribution is severely altered. The assumption is that this can be confirmed during the analysis by completely altered results. Figure 3.2 depicts the object modifications B and C.

(a) Modification B (b) Modification C

Figure 3.2: Object Modifications

Object modifications B and C consists of weights being put into the corners of the box. For the weight in version B a single wood block is used. Version C contains two metal plates fixed into adjacent corners as shown in (b).

3.2.3 Safety Measure

As a safety measure, the push sampling is restricted depending on the position of the object. The idea is that if the object moves too close to the surface border it is being pushed back towards the table center. Since this resembles the problem intended to be solved, a simple heuristic must suffice.

The table surface is separated into three nested zones around the table center. The inner zone is the space where the object is supposed to be, so sampling is unrestricted.

Surrounding the inner zone is the safety zone. If the box is located inside that zone pushes are restricted by their direction regarding the table frame. The assumption is if the push movement is directed towards the center, the object at least won't move away from it. Ideally, the object is being pushed back into the inner zone so that fully random exploration can continue. In case the object is still being pushed further outside, the safety zone is surrounded by the emergency zone. To prevent the object from falling off the table the exploration process is aborted and needs to be restarted.

The critical point of this safety measure is the restricted sampling in the safety zone. The sampler rejects all pushes that are not directed towards the table center within a tolerance of 20°. It is a founded concern if this restriction affects the randomness of the push sampler. Since all pushes are specified regarding the object frame, the restriction depends on the orientation of the object. Assuming the orientation is fully random, the restriction has no impact on the push sample distribution within the safety zone. On that condition, the pushes within the safety zone are recorded along with the exploration process. Figure 3.3 shows the top view of a table with safety zones and restricted pushes in an example.

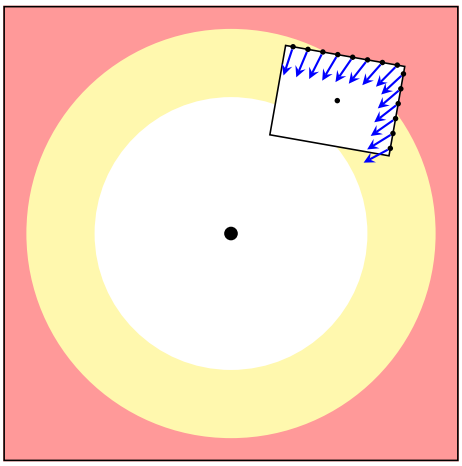


Figure 3.3: Table surface with Safety Zones - The blue arrows indicate restricted pushes

3.3 Learning to Predict Push Effects

Section 3.3.1 defines the learning problem and specifies prediction function and feature vector signatures. Section 3.3.2 describes the introduction of the SE(2) distance as a custom loss function. Since the predicted output represents a SE(2) state, this loss is expected to better represent the learning objective. Section 3.3.3 introduces several architecture prototypes and training methods. These prototypes are used as a foundation for the subsequent hyper-parameter optimization described in section 3.3.4.

3.3.1 Prediction Problem

The relation between object poses on a surface and applicable pushes can be defined as a dynamical system. If object poses are the states of the system, then push Φ is a transition function between states P and P':

$$P \xrightarrow{\Phi} P' \tag{3.1}$$

Let T denote the transformation from P to P'. Under the assumption that the system of push operations satisfies the Markov property, the transition function can be simplified:

$$\begin{aligned} & P \xrightarrow[M]{\Phi} P' \\ \equiv\ & P \xrightarrow[M]{\Phi} P * T \\ \equiv\ & I \xrightarrow[M]{\Phi} I * T \\ \equiv\ & I \xrightarrow[M]{\Phi} T \end{aligned} \tag{3.2}$$

A forward push model p approximates the transition function $\xrightarrow[M]{\Phi}$ corresponds to the following function signature:

$$\begin{aligned} p : \Phi &\xrightarrow{\approx} T \\ &\equiv \\ (A_\Phi, \vec{n}_\Phi, \vec{v}_\Phi, d_\Phi) &\xrightarrow{\approx} (x_T, y_T, \gamma_T) \end{aligned} \tag{3.3}$$

This signature specifies the input and output parameters of the prediction function. In order to approximate this function, the parameters need to be represented as normalized input and output features. Push approach point A_Φ is decomposed into X and Y components. The orientation vectors $\vec{n}_\Phi$ and $\vec{v}_\Phi$ are represented by their angle representations α_Φ and β_Φ. Push distance d_Φ, x_T, y_T and γ_T are already scalar values and can remain as such. This results in the feature vector signatures used for all prediction architectures.

$$\begin{aligned} X{:}&\quad x_A, y_A, \alpha_\Phi, \beta_\Phi, d_\Phi \\ Y{:}&\quad x_T, y_T, \gamma_T \end{aligned}$$

3.3.2 SE(2) Distance as Loss Function

Sophisticated machine learning models like neural networks usually optimize a specific loss function that measures the prediction error as a single loss value. Loss functions like mean squared error rely on averaging the absolute errors of all features equally. This is a problem if the scaling of the output features is not equal, since some feature might then be trained more accurately than others. This is especially true, if the features are not homogenic, like it is the case with the prediction of SE(2) Poses by X, Y, Yaw features. While X and Y translation might be weighted equally, the Yaw value is difficult to relate as an equitable distance value.

A common metric for the SE(2) distance is the weighted sum between translation and rotation distance:

$$D_{SE(2)} := \sqrt{D_X^2 + D_Y^2} + 0.5 * D_{Yaw} \tag{3.4}$$

Using this loss function for optimization SE(2) pose prediction is assumed to produce advantageous results:

- The errors of the features is weighted implicitly and unrelated to scaling
- The loss function represents the real problem, minimizing the distance between poses

However, the ratio of 2:1 is not necessarily the best weighting of the features in every scenario. Considering that the maximum yaw distance is limited to π fine tuning the weight to level the maximum expected translation distance might produce even better results.

3.3.3 Architecture Prototypes

With the goal to analyse the learning problem and improve prediction accuracy multiple approaches are tested and compared. This section presents three different prototype architectures. Each architecture takes a different focus regarding optimization objective and regularization. Also, they form the foundation for the hyper-parameter optimization process described in section 3.3.4. All prototypes are trained using min-max normalization for input and output features.

Prototype $B0$ - Separated Models

This prototype is intended to serve as a baseline reference for accuracy and robustness of later models. The output features are predicted by separate architectures. X- and Y-translations are approximated by linear regression models using least squares approximation.

The Yaw-predictor is a simple MLP with a single hidden layer of 100 units size. The MLP optimizes the squared loss using a *Limited-memory BFGS* (lbfgs) solver [Byr+95]. This solver is expected to converge faster given the comparably small datasets. The training is run for 200 iterations with a batch size of 200 and *l2* penalty set to 0.0001. All models are trained using the normalized push specification as input and only the corresponding feature as output.

The choice of the architectures is based on different expectations. Using separate predictors should allow isolated analysis of the prediction accuracy. That means the prototype not only serves as a baseline for overall accuracy but also for the accuracy of single features. This design simplifies the prediction problem by reducing the dimension of the output function, even if this increases the risk for overfitting. The results are assumed to give deeper insight into the problem and learning behavior.

Considering later approaches are implemented using single neural networks, several questions arise regarding the interdependence of the output features:

- How much does the prediction accuracy improve, if at all?
- How strong is the regularizing effect of weighting the output features against each other?

Prototype N_1 - Simple Neural Network

This approach uses a single neural network for all output features. It only has a single hidden layer of 100 units and uses *relu* activation. The network uses *Adam* [KB14] optimization and is not regularized by any measure. As loss functions both the *mean squared error* (MSE) and SE(2) distance (see 3.3.2) are tested. The training is run for 20 epochs of 500 training steps and 100 validation steps.

Prototype N_2 - Regularized Neural Network

In contrast to Prototype 2 this network has three hidden *relu*-layers with dropout and $l2$ regularization. The layer sizes are 128, 64 and 32 in sequential order. The dropout is applied after the first and second hidden layer with a weight of 0.3. Other than that the network is trained and configured like the second prototype. The optimizer is *Adam*, loss functions are MSE and SE(2) distance and training is run for 20 epochs with 500 training and 100 validation steps.

3.3.4 Hyper-parameter Optimization

Based on the tested prototypes, the hyper-parameter search optimizes a multi-layer MLP with dropout regularization.

The architecture of the neural net is defined by the number and sizes of hidden layers. Up to 4 hidden layers are initialized while each has its own unit count, dropout and activation function. The unit count is sampled as a 2 based exponent, from 2^4 to 2^10. This is more efficient sampling the layer sizes from a quantized distribution directly, since the value range is dissected into just a few steps. To counteract overfitting caused by too high layer resolution, each layer is regularized by a dropout layer. The dropout is scaled from 0.0 to 0.5. This range practically makes the dropout layers optional, in case the optimization process results in values approaching 0. Higher values are not necessary since in that case the layer size could be reduced as well.

The learning rate is set as the product of a randomized multiplier and a default value of 0.001. The default is a generic value that is feasible for this kind of architecture. The multiplier is sampled from a log uniform distribution from -0.5 to 0.5. This conforms to a value range between 0.61 and 1.65 with a much higher density in the lower range. That means the learning rate is effectively sampled between 0.00061 and 0.00165 which is a comparably large search space.

Next to dropout regularization $l2$ regularization is used as as well. The $l2$ penalty weight is sampled analogous to the learning rate from a log uniform distribution and a

default. With a base of 0.0007 and a multiplier log between -1.3 and 1.3, the effective values range from 0.00019 to 0.00257.

Sampling the learning rate and $l2$ weight from log uniform distributions allows more efficient search in higher value ranges. This choice is based on the assumption that the optimized values would be low while still considering higher results. Table 3.2 shows the final version of the hyper-parameter space.

Parameter	Domain
optimizer	*Adam, Nadam, RMSProp*
learning rate	0.001 * log$U(-0.5, 0.5)$
L2 weight	0.0007 * log$U(-1.3, 1.3)$
input activation	*linear, tanh, relu*
hidden layers	1 to 4
per layer	
- units	$2^{qU(4,10)}$
- dropout	$U(0.0, 0.5)$
- activation	*Linear, Tanh, ReLu*

Table 3.2: Hyper-parameter Space

Other considered parameters include the loss function (see 3.3.4) and training protocol (see 3.3.4). In the final version, the loss function is set to SE(2) loss. Alternatives and the decision process is described below in section 3.3.4.

The training protocol is specified to 20 epochs with 500 training- and 100 validation steps. That is the same protocol as used for the neural network prototypes. The reasoning behind this is described in section 3.3.4.

Also analogous to the prototypes, input and output vectors are min-max normalized. Based on the results of early experiments, this results in a much higher prediction accuracy compared to standard deviation based methods like z-score.

The optimization process is run for 600 trials both with and without filtering failed attempts. The architecture of the best resulting model is presented in section 3.3.5 below. This model is also used for the predictive planning approaches described in section 3.4.

Loss Function

Adjusting the loss function is generally a very powerful method for optimizing neural networks. For regression problems, there are various options for how errors are weighted to compute the loss. Since the loss function measures the prediction error, it is a critical factor for what the network is actually learning. *Mean Squared Error* (MSE) and its variants all weight the features equally with slight alternations how outliers are handled. The *LogCosh* function, for instance, is similar to MSE but reduces the impact of occasional outliers. In order to find a good loss function early prototypes of the networks are trained with *MSE, MAE, MAPE, MSLE* and *LogCosh*. However, when using different

loss functions the performance of the models is not really comparable. A common loss or distance criterion is required for this. As a suitable method, the SE(2) distance loss (see 3.3.2) is chosen as the only loss function. This also gives the advantage that the output features are weighted so that a geometrical distance criterion is minimized.

Training Protocol

The training protocol defines how training and validation steps are run. This includes alternating between batch and iteration based training, as well as adjusting batch size, iteration count and validation. Since the training protocol has a big influence on the time a network is trained, this is a major factor for efficient hyper-parameter optimization. To reduce the training time and still ensure convergence of the models the training protocol is prespecified. That way the emphasis is put on architecture optimization and the models are more comparable.

Outlier Removal

It is questionable if outliers should be excluded from training. These are almost exclusively failed attempts with no object movement. Due to inaccuracy the pusher would primarily fail to hit the corners of the object. Removing the outliers could therefore improve the friction model representation. In return, the failed attempts are valuable indicators for insecure pushes. Training the models with the unfiltered data could result in a model that avoids the corners, which could increase execution robustness.

The optimization process is run with filtered and unfiltered data. The filter removes samples if the SE(2) distance of the object movement is below a threshold of 0.005.

3.3.5 Predictive Sampling

A low prediction error is crucial for predictive sampling, in order to increase planning accuracy. For that, the best results of the optimization process should be applied. However, the winning architectures are not necessarily the best performing overall, which is further described in section 5.2. In order to increase robustness, all presented planners are using the prototype N_2.

3.4 Push Planning

The problem of finding a sequence of pushes that move an object from start to goal resembles a classical motion planning problem. The state space is the space of object poses and the control space is the space of possible push operations. Therefore, the following planning problem is considered:

Find a sequence of pushes that produce a path of object poses from start to goal.

Section 3.4.1 specifies the state and control space that are used for path exploration. In section 3.4.2 different planners are presented using distinct exploration strategies. An exploration strategy is defined by the corresponding *state propagator* and *control sampler*. This relates to the concepts *steered* state propagator as well as the *directed* and *chained* control samplers. Functionality and implementation of these are described in section 3.4.3. In order to investigate planning performance and compare the different strategies, the planners are initialized using different parameter configurations. The applied configuration setups are stated in section 3.4.4. Finally, the execution of push plans by means of an MPC approach is discussed and described in section 3.5.

3.4.1 State and Control Space

In order to apply generic motion planners, state and control space need to be specified. The object poses are in SE(2) and are restricted to the table surface. SE(2) is applicable as a state space and the surface bounds define the value ranges of the X and Y components. Also, the goal objective is defined by the SE(2) distance between object and goal.

The control space is defined as the space of scaled and normalized push control vectors. A control vector contains three decimal values scaled from 0.0 to 1.0. Considering that the shape of the object is known, the vector can be mapped to a push specification. A_Φ and $\vec{n}_\Phi$ are determined by mapping the first control to the object shape. This is analogous to the random push sampling described in the exploration section 3.2. The push direction $\vec{v}_\Phi$ is retrieved by normalizing the second control into the angle range. The third control is mapped to the push distance d_Φ by normalizing it into the distance range. The table below summarizes the value mapping for push controls.

Control	Push	Control Normalization
1	A_Φ, $\vec{n}_\Phi$	Shape boundary
2	$\vec{v}_\Phi$	from β_Φ: -0.5 to 0.5 rad
3	d_Φ	1cm to 3 cm

3.4.2 Planning Strategies

All planning strategies are implemented using the tree-based algorithms RRT [KL]. The strategies allow experimenting with different exploration methods. They are defined by the applied state propagators and control samplers.

A state propagator determines if the state space is explored randomly or goal-directed. A control sampler has a more local impact on the appearance of the planning tree. For instance, random sampling creates high frequent changes in direction while directed sampling produces rather smooth paths.

Since the controls are pushes and the states poses all propagators compute the successor states by using the forward push prediction models (see 3.3). Due to the lack of inverse push models the *Directed-* and *Steered StatePropagator* use other methods for computing the directed controls. Next to a random sampling approach similar to that used in the POC (see 3.5) different directed sampling algorithms are used and tested. However, since KPIECE only supports the *Default StatePropagator* only RRT exploits directed exploration.

For the pushing scenario, the duration is not easily applicable as a continuous time value. That would require interpolating between pushes, which is assumed to be very inaccurate. It is much more practicable to define the duration as the number of executed push controls. In combination with *propagation step size* set to 1, the planner uses push sequences for exploration.

Random Exploration

This strategy is the default behavior of the used OMPL implementation of RRT. It uses the *default state propagator* combined with *random control sampling*. That means all exploration steps are run in random directions. The only goal-directed element is the goal biased state sampling, which defines the states from where the exploration is progressed.

Figure 3.4 shows an example of the exploration method. The vertices r_1, r_2, r_3 are sampled target states and s_1, s_2, s_3 states that are reached by random controls. Close

State Propagator: **default**
Control Sampler: **random**

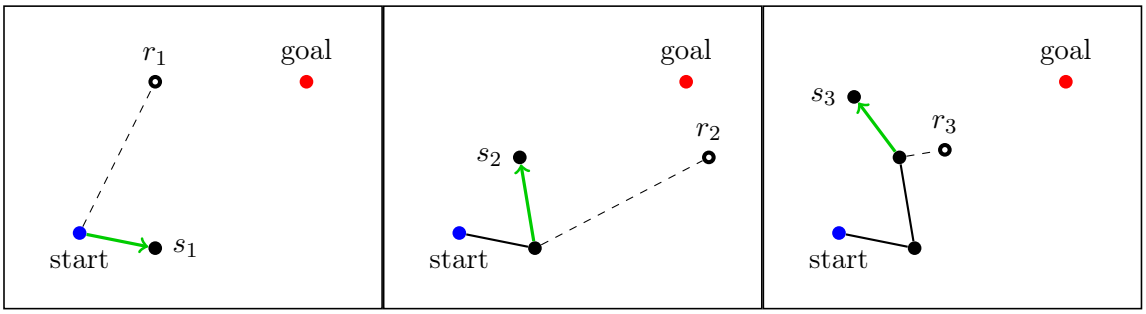


Figure 3.4: Random Exploration

Directed Exploration

Directed exploration is similar to *Random Exploration*, just that the sampled pushes are directed and not random. When a new state is sampled, the controls are sampled to minimize the distance towards it. Since state sampling and control sampling is goal-directed, this strategy is assumed to converge faster than the random approach. The tree growth is a direct result of the state sampling which allows more efficient use of the goal bias. An example of the strategy is shown in figure 3.5.

State Propagator: **default**
Control Sampler: **directed**

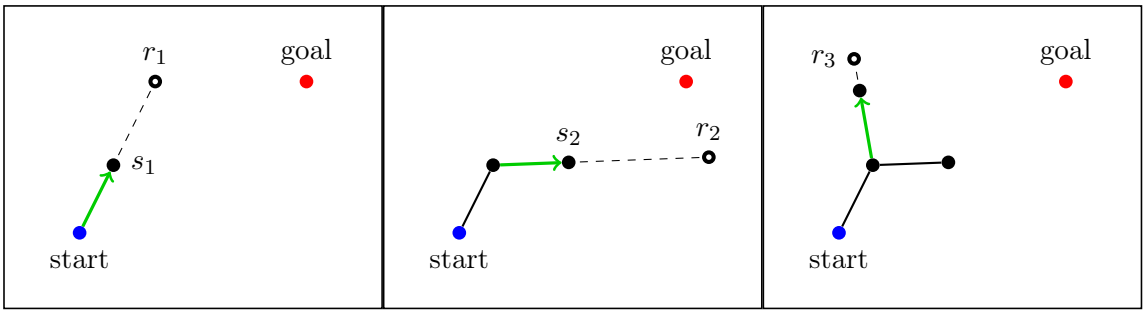


Figure 3.5: Directed Exploration

Steered Exploration

This exploration strategy uses a *steered state propagator* with *random control sampling*. A steered propagator generates a control and a duration to reach a goal state from a given start state. When a new state is sampled, the steered propagator attempts to include it into the planning tree. The closest existing state is selected as the starting point for steered controls to reach the new state. Figure 3.6 below illustrates this strategy.

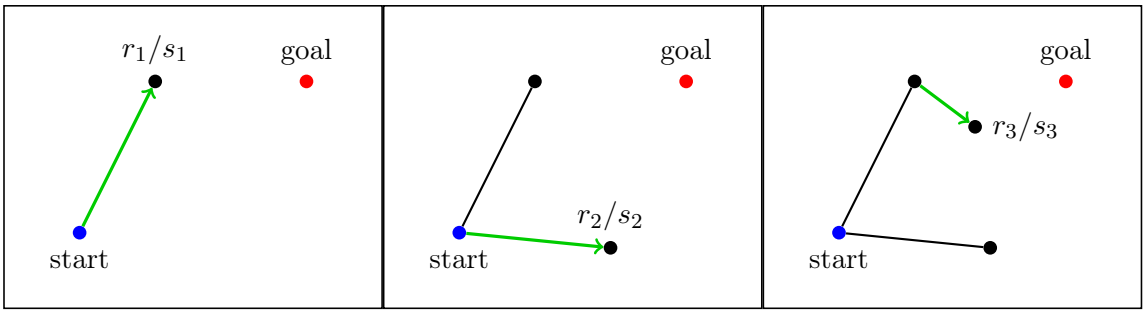


Figure 3.6: Steered Exploration

It is reasonable to assume that directed control sampling would be more appropriate. However, the pushing scenario is a special case. When pushing the steering conforms to a sequential execution of the same push. Linear pushes cause the object to move around a rotation center, analogous to the instant center of rotation described in 2.1. That means that all steered paths would appear either as straight lines or partial circles. The length of the paths is dependent on the control duration which equals the number of push executions. Directed control sampling would allow pushes that are directly pointed towards the target, hence invalidating all curved solution. Random sampling allows all directions at the beginning of a steered path.

This strategy is expected to explore the state space in bigger leaps than the random strategy. It's even possible that steer controls span the full distance between state space boundaries.

State Propagator: **steered**
Control Sampler: **random**

Chained Exploration

The term *Chained* corresponds to the control sampling method, where previous controls influence the next solution. The control sampler used in this strategy samples approach points from the neighborhood of the last push. The intention is that the sequential execution of similar pushes should lead to continuous exploration paths. To enforce directed exploration steps, the chained controls are sampled to minimize the distance towards the target state. In combination with the default state propagator, this should result in uniform exploration with smooth solutions. An example is shown in figure 3.7.

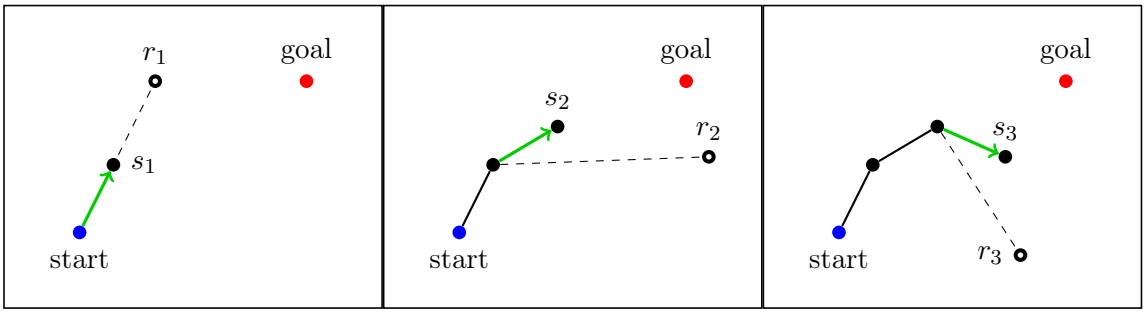


Figure 3.7: Chained Exploration

When planning with the box object, this means that consecutive pushes are not forced to be continuous. Pushes close to a corner of the box allow the sampling of pushes on the adjacent side. Solutions could therefore also include abrupt direction changes.

State Propagator: **default**
Control Sampler: **chained & directed**

3.4.3 Push Sampling

The introduced planning strategies are based on sampling directed, steered and chained pushes. Directed and steered controls are specified as part of the state propagation interface in OMPL [Suc18b]. Chained controls are based on the *sampleNext()* function in the control sampler specification [Suc18a], which considers previous controls. While directed and chained push samplers are ordinary control samplers, steering is implemented as a full state propagator. The reason is that steered pushes require a duration value which depends on the state context. The implementation of the sampling algorithms is described in the following sections.

Directed Control Sampler

Directed pushes are sampled so that they minimize the SE(2) distance between start and target states. When called, the algorithm iteratively generates random pushes and predicts the successor state. The push that produces the closest successor state to the goal is returned as the solution. The sampling count k is determined by the corresponding planner configuration. Increasing values lead to better results while equally impairing sampling time.

Algorithm 3.1: Directed Push Sampler

```
1  sampleDirectedPush(start, goal, k=100) {
2    bestPush = None
3    bestDistance = getDistance(start, goal)
4    for (i=0 to k) {
5      push = sampleRandomPush()
6      nextPose = start * predictPose(push)
7      nextDistance = getDistance(nextPose, goal)
8      if (nextDistance < bestDistance) {
9        bestDistance = nextDistance
10       bestPush = push
11     }
12   }
13   return push
14 }
```

Chained Control Sampler

Exploration with chained controls enforces newly sampled controls to have an approach point close to previous controls. In the attempt to enable smooth object paths adjacent pushes are supposed to be similar to produce similar movements. Algorithm 3.2 is an example implementation of a chained control sampler with distance optimization. The procedure is very similar to the ordinary directed control sampler. The difference is that the previous push is used to sample a similar push approach, as shown in the lines 6 to 8.

Algorithm 3.2: Chained Push Sampler

```
1  sampleChainedPush(lastPush, start, goal, k=100) {
2    bestPush = None
3    bestDistance = getDistance(start, goal)
4    for (i=0 to k) {
5      push = sampleRandomPush()
6      if (lastPush != None) {
7        push.approach = sampleFromNeighborhood(lastPush.approach)
8      }
9      nextPose = start * predictPose(push)
10     nextDistance = getDistance(nextPose, goal)
11     if (nextDistance < bestDistance) {
12       bestDistance = nextDistance
13       bestPush = push
14     }
15   }
16   return push
17 }
```

Steered Propagator

This strategy uses a minimal distance strategy similar to the directed control sampler. If a random push minimizes the distance from start to goal it is selected as a steering candidate. Each push control candidate is tested if a sequence of executions pushes the object from start to goal. This is done by repeatedly computing successor states beginning from the start state and comparing the new goal distances. Decreasing push distances suggest that the push sequence moves the object towards the goal. If at some point the goal distance is within the goal threshold the candidate is returned as a successful solution. Otherwise, there will be a step that increases the goal distance. In that case the candidate is discarded and a new push is sampled. The control duration is determined by the number of propagated steps.

Algorithm 3.3 is a pseudo-code implementation of the described procedure. Different from control samplers, a state propagator applies the push control together with a duration for state propagation. If no solution is found, the exploration would continue with a new sampled goal state.

Algorithm 3.3: Steered Propagator

```
steerPushControl(start, goal, k=100, threshold=0.05) {
  startDistance = getDistance(start, goal)

  for (i = 0 to k) {
    duration = 0.0
    push = sampleRandomPush()
    transform = predictPushEffect(push)

    pose = start
    distance = startDistance
    minDistance = startDistance

    while(distance <= minDistance) {
      minDistance = distance

      pose = pose * transform
      duration += 1.0
      distance = getDistance(pose, goal)

      if (distance < threshold) {
        propagate(push, duration)
        return True
      }
    }
  }

  return False
}
```

3.4.4 Planner Configurations

The introduced planning strategies rely on very different exploration methods. Therefore, it is unfit to compare all planners using the same configuration. If one planner excels with certain parameters set, that does not imply another will as well. In order to gain insight on a planners' performance, each planner is tested with varying parameters. The focus is put on parameters that alter exploration behavior, like *goal bias*, *max control duration* and *intermediate states*. The other parameters are fixed to the same values in all planning instances. *Goal threshold* is the common planning objective and essential for comparability. *Min control duration* and *Propagation step size* are both set to 1.0 due to stepwise pushing. Table 3.3 lists the parameter space of the tested configurations.

3.5 Plan Execution

A common and expected issue with predictive planning is the inherent inaccuracy. This is especially prominent in the pushing scenario. The prediction error would accumulate during the execution so that the object departs from the target trajectory. Even if there was a perfect push prediction model of the object, there would always occur some error in the setup. These could be caused by calibration offsets, camera distortion, uneven

Parameter	**Tested Values**
Goal Threshold	0.05
Goal Bias	0.05, 0.1, 0.25, 0.5, 0.75
Min Control Duration	1.0
Max Control Duration	1.0, 5.0, 10.0, 25.0, 50.0, 100.0
Propagation Step Size	1.0
Intermediate States	enabled, disabled

Table 3.3: Planner Configurations

surface or bending hardware. This renders predictive plans unfit for open-loop execution. A good plan execution strategy should, therefore, adapt to prediction errors in a closed-loop fashion. An example for that is *Model Predictive Control* (MPC) - a control method that relies on iterative replanning.

This section describes the approach of realizing goal-directed push operations based on closed-loop controls. First, a simple proof of concept is presented that uses directed sampling for greedy distance minimization. Afterwards, an MPC implementation is described that generates the push controls based on the different push planners. Both approaches are tested under the aspects of robustness, efficiency and collision avoidance.

Greedy Distance Minimization

This approach attempts to move an object from a start to goal using directed pushes. The algorithm repeatedly samples pushes that minimize the distance towards the goal and executes them. Push sampling is realized using the directed control sampler described in section 3.4.3. After each push execution, the new object pose is queried. The goal objective is fulfilled if the SE(2) distance between object and goal pose is under a certain threshold. Otherwise, the loop continues by sampling and executing the next push. An implementation of the proof of concept prototype is given in algorithm 3.4.

Algorithm 3.4: Greedy Distance Minimization

```
1 pushObjectToGoal(start, goal, threshold=0.05) {
2   pose = start
3   while (getDistance(pose, goal) > threshold) {
4     push = sampleDirectedPush(pose, goal)
5     executePush(push)
6     pose = getCurrentObjectPose()
7   }
8   ... // goal is reached
9 }
```

Plan-based MPC

The greedy approach shows how the goal objective can be fulfilled using a simple sampling strategy. Since it only uses a local criterion, this method lacks the capability for collision avoidance. An MPC approach based on push plans is expected to function as a local and global control strategy at the same time.

By only executing the first control step of each new plan following it can be assumed that:

1. Each plan leads the object towards the goal
2. Each step moves the object along a path towards the goal

Likewise, following assumptions can be made on collisions:

1. The first control of a collision-free path is itself collision-free
2. Iterative execution of collision-free controls creates a collision-free path

MPC only guarantees to produce a sequence of valid steps along goal-directed collision-free paths. It is not guaranteed that this sequence converges. An example where MPC converges to a local minimum can be easily constructed. Consider an obstacle being placed between object and goal. The planned paths could pass the obstacle alternately on the right or left side. Since only the first control step is executed each time, the object would be pushed back and forth without moving towards the goal. However, MPC appears quite reliable in many practical scenarios. Also, the *repeatability* of the planned solutions is important. If adjacent plans always lead to similar solutions, MPC does not get stuck in local minima and creates consistent trajectories.

A pseudo-code implementation of a simple MPC approach is shown in figure 3.5.

Algorithm 3.5: Plan-based MPC

```
1  runMPC(start, goal, threshold=0.05) {
2    planner = initializePlanner()
3    pose = start
4    while (getDistance(pose, goal) > threshold) {
5      plan = planner.plan(pose, goal)
6      if (plan.isValid()) {
7        push = plan.getPushControls().first()
8        executePush(push)
9        pose = getCurrentObjectPose()
10     }
11   }
12   ... // goal is reached
13 }
```

Multi-step MPC

Predictive planning is computationally very costly. Also, replanning is redundant if the object actually moves along the target trajectory. An optimized MPC should execute as much of a plan as possible. Since trajectories consist of sequences of pushes and states, a simple heuristic can be used. A multi-step MPC successively executes the pushes and compares the object locations with the target states. If a certain distance threshold to the target state is exceeded, this process continues with a new plan. Algorithm 3.6 implements this behavior.

Algorithm 3.6: Multi-step MPC

```
1  runMultiStepMPC(start, goal, threshold=0.05) {
2    planner = initializePlanner()
3    pose = start
4    while (getDistance(pose, goal) > threshold) {
5      plan = planner.plan(pose, goal)
6      for (step=0 to plan.pushes.size()) {
7        push = plan.pushes[step]
8        executePush(push)
9        pose = getCurrentObjectPose()
10       target = plan.poses[step+1]
11       if (getDistance(pose, target) > threshold)
12         break
13   }
14   ... // goal is reached
15 }
```

4 Setup & Implementation

All experiments are executed using a UR5 manipulator [Rob18b] that is mounted to the wall. A table is placed in front of the robot so that objects on the surface are reachable in all positions. The robot is equipped with a 3-finger adaptive gripper from Robotiq [Rob18a]. Single contact pushes are enabled by a custom pusher tool attached to the gripper. Design and images are described in section 4.1. A Kinect2 camera [XBO18] is placed directly in front of the table, facing the surface and robot mount. The object localization is realized using april tag detection.

The software setup is built on the Robot Operating System (ROS) [Fou18]. ROS is an open source infrastructure for integration of hardware drivers and robot capabilities that is widely used in research and industry. Collision free motion planning is realized with the MoveIt! framework [SC18] using the *planning scene*. This is a world representation that includes the robot model as well as visual and collision geometry. It allows dynamic modifications of the planning environment including collision and visibility checks. Just as the push planning approaches the motion planners in MoveIt! are implemented using the Open Source Motion Planning Library (OMPL)[PC18].

Section 4.1 describes the application of pushes in the setup. This includes the pusher tool, control procedures and optimizations. Object localization and improvements are described in section 4.2. Details about the implementation of learning and planning is given in section 4.3.

4.1 Push Setup and Execution

Accurate push operations can only be performed as long as the pusher is rigid and allows hitting the contact point. The attached gripper has adaptive fingers with passive joints that move when an external force is applied. Also, the finger tips are polygonal and too large for the required accuracy. Even if accuracy was not a problem, the polygonal shape would cause the object to align with the dominant edge, invalidating the push in the process.

To enable accurate push operations a custom pusher tool is applied to the gripper. The tool is a 3D-printed device designed in OpenSCAD [Kin18]. It consists of a rounded stick attached to a broader base. The rounded stick resembles a finger tip and allows neglecting the rotation around the length axis. For attaching the tool to the gripper, the base is shaped to fit into the palm. By closing the fingers to a pinch grasp, the pusher tool is fixed and can be used as the new end effector. RViz visualization and an image of the attached pusher is shown in figure 4.1.

The modified end effector extends the kinematic chain of the robot. When the pusher is attached the end effector frame is moved to the pusher tip. The Z-axis of the frame

Figure 4.1: Pusher tool attached to gripper

points towards the palm so that it aligns with the surface normal when pushing. That allows describing pushes by their X and Y translations. The Z coordinate defines the distance from the table. In the planning scene, a mesh model of the pusher is attached to the gripper for collision avoidance.

4.1.1 Push Execution

The push execution itself is implemented as a cartesian path along three waypoints, *Start, Target, Retreat*. Figure 4.2 illustrates path and waypoints in an example.

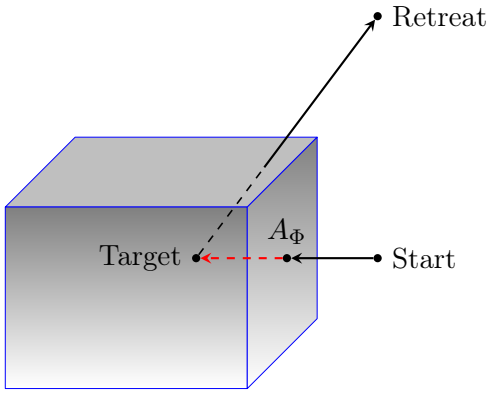


Figure 4.2: Push Execution Waypoints

They waypoints are positioned regarding the push approach A_Φ and direction $\vec{v}_\Phi$. The actual push is performed by a linear movement from *Start* to *Target* crossing A_Φ. The distance between *Start* and A_Φ is added to avoid hitting the object too early when moving the pusher into position. The *Retreat* leads the pusher away from the object vertically above *Start*. The height of *Retreat* ensures a clear view from the camera for localization.

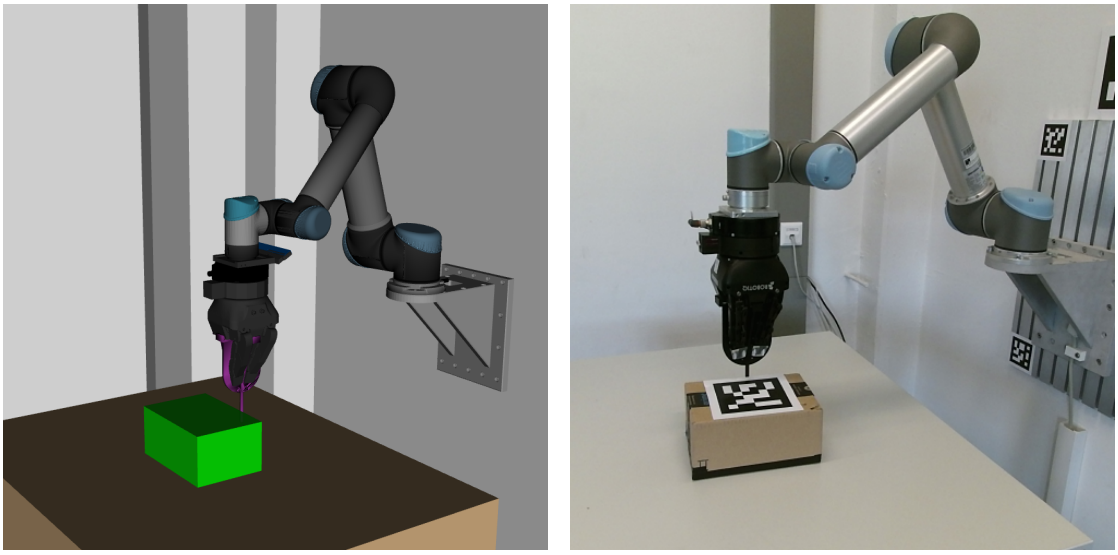
Movements before and after the push are planned using ordinary motion planning while avoiding collisions with the object.

4.1.2 Sample Exploration

The autonomous exploration process is realized as an ongoing loop of instructions. These include planning and execution runs as well as retrieval and saving of data. Each iteration consists of the following sequence of steps:

1. Localize object pose P
2. Add object to collision scene
3. Sample random push Φ
4. Move pusher to *Start*
5. Take snapshot 1
6. Remove object from collision scene
7. Plan Φ
8. Execute Φ (take snapshot 2 at contact)
9. Take snapshot 3
10. Localize object pose P'
11. Save Φ, P, P' and snapshots

At the beginning of every iteration, the target is localized and added to the planning scene as collision object. That ensures that the pusher can be moved to the start pose without moving the object. After the push is sampled, the collision object is removed from the planning scene to allow the push execution. The push is planned and executed as described in section 4.1.1. Camera snapshots are taken before, during and after the execution to document the push. Figure 4.3b shows is an example of a snapshot taken at step 8 during the execution. After the execution, the object is localized again and the results and snapshots are saved.



(a) Screenshot taken in RViz (b) Snapshot taken from Kinect2 camera

Figure 4.3: Workspace during push operation

Optimizing the Exploration Process

An important aspect when collecting larger datasets from robot experiments is the needed amount of time. The initial implementation of the exploration process runs pushes with an average duration of 18 seconds. The major factors are the execution times of the free motion towards the start pose (step 4) and the push execution itself (step 8). Besides increasing the velocity and acceleration limits of the robot, the following measures reduce the execution time to only 8 seconds per push.

1. Optimistic pre-planning: The default trajectory execution function in MoveIt! is blocking. That means the program flow is interrupted as long as the movement endures. Computing tasks like the Cartesian push planning are run after the execution, delaying the time until the push can be executed. By changing to asynchronous execution, the program flow can continue so that the time during the path execution can be used for further computations. The push is planned before the robot even reached the start state, using the expected joint configuration. A callback is triggered when the trajectory is complete to synchronize the program flow before executing the push. If the push plan fails, the executed trajectory is reversed so that a new push attempt can be started.

2. Constrained Motions: Motions between pushes are planned using *RRT-Connect* [KL], a bidirectional implementation of RRT. The planner is used because of the short planning times. However, some trajectories can be awkwardly long and complicated which leads to lengthy execution times. Usually, the problem is a flipped joint in the target state, requiring the arm to turn the joint around. This is a common issue when using non-optimizing motion planners. Since optimizing planners require longer planning times,

another approach is followed. The idea is to restrict the motion range to enforce simple and short trajectories. An orientation constraint for the end effector lets the pusher point downwards during all motions. This orientation is the same used for pushing, so that movements between pushes appear more natural. Flipped joint targets are still technically possible, but are practically avoided by the restricted motion space. A drawback of constrained planning is the increased planning time, which can be up to a minute in extreme cases. The reason is inefficient rejection sampling of joint configurations. A solution is presented by Sucan et al [SC12], by priorly constructing a constrained joint space approximation and using that for sampling. MoveIt! offers an interface to compute a space approximation for given constraints [Lau18] which can be saved to a database. The database is loaded when launching MoveIt! along the robot setup and can from then on be used for constraint planning. This reduces planning times of constraint motions to less than a second. In combination with the reduced execution time, this accounts to the major part of the speedup.

4.2 Accurate Object Localization

Object localization is accomplished by tracking an April tag attached to the object as shown in figure 4.3b. The detected pose is transformed from the camera frame to the surface frame and adjusted to the surface normal by removing rotations around X- and Y-axes. The adjustment is valid since the object is known to be lying flat on the surface. Rotations other than $P_\gamma$are therefore caused by detection or calibration errors.

A major factor for localization and setup accuracy is the camera frame. The camera is movable and the frame is set dynamically when launching the setup. It is adjusted by an April tag on the wall with a fixed transform to the robot model. Since tag detections can be noisy, a low pass filter is used to smooth the position. Still, small detection errors regarding the orientation of the wall tag lead to up to 3 centimeters position offset of the object. The detection error of the object tag itself is added on top of the already noisy recognition. This leads to invalid push results that skew the collected sample data.

Different measures are implemented to reduce this problem.

1. Camera Calibration

The first optimization step is to reduce potential image distortion. This is done by recalibrating the camera with the *Kinect2 Calibration* package [Wie18]. As a result, the object position offset is reduced to about 2.5 centimeters.

2. AprilTags 2 Upgrade

Another important step is the upgrade of the April tag detection method. In the initial implementation uses the April tag algorithm from the ROS-package *apriltags_ros* [Wil18]. The new algorithm *AprilTag 2* presented by Wang et al [WO16] promises improved robustness and detection rate. The package *apriltags2_ros* [Mal18] works as a drop in replacement for *apriltags_ros*. In the setup this upgrade increases the detection rate of a single tag by a factor of five, from 3 Hz to 15 Hz. Also, the detection rate is less affected by multiple tags in the image. Even with five tags in the image, the rate is still at 12 Hz.

This allows increasing the low pass filter weight while keeping the position delay low. A higher filter weight corresponds to averaging over more detections, which in return increases the accuracy.

Another feature of *apriltags2_ros* are tag bundles. A group of tags can be defined as a bundle with fixed transforms to a certain frame. The detection returns the bundle frame by averaging all tags belonging to the group. Changing the camera localization from a single wall tag to a bundle increases the setup accuracy in multiple ways. The bundle tags are attached to the four corners of the robot mount plate, as shown in 4.4b. The distance between the tags dramatically reduces orientation errors, a major factor for noise. Since not all of the tags need to be visible, the robot is less likely to obscure the view. Additionally, fixing the tags to the mount plate avoids another problem in the setup. The wall holding the plate and wall tag is slightly tilted, which is not represented in the robot model. Adjusting the camera after the tilted april tag produces a localization offset. In effect, the robot model does not match with the camera image as shown in figure 4.4a. By adjusting the camera with the exactly measured mount plate, robot model and camera image are aligned.

Comparison

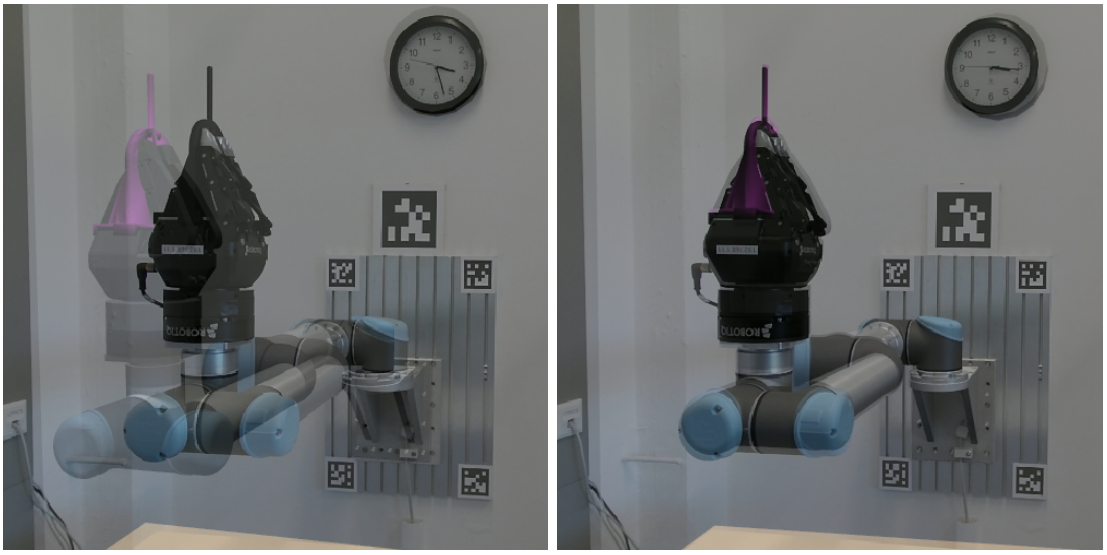
In short, the steps for improving setup and localization accuracy are:

1. Camera Calibration
2. Upgrade to *AprilTags 2*
3. Stronger low pass filtering with higher detection rate
4. *Bundle Detection* for camera localization

The overall improvement reduces the end effector offset from 3cm to about 0.3cm. An example of the optimized setup is shown in figure 4.4b.

4.3 Prediction and Planning

All learning related software is implemented in python. Data preprocessing and analysis is performed with the package *pandas* [PyD18]. The baseline prototype is implemented using the machine learning library *scikit-learn* [Ped+18], and all neural netwoks are compiled with Keras [Cho+15]. The hyper-parameter optimization process is run using *Hyperopt* [BYC13][BYC18]. Trained models are stored, so that they can be loaded and used for predictive planning. Prediction requests are implemented in two different ways. The initial python implementation uses ROS-nodes to enable access from the planners that are implemented in C++. A more efficient implementation in C++ loads the models directly and computes the results using vectorization. This improvement is described below in section 4.3.



(a) Model overlay without optimization (b) Model overlay of the optimized setup

Figure 4.4: Comparison of the RViz model overlays before and after the improvements

The planning architectures are implemented using the OMPL framework, including all components described in the method section 3.4. Planning attempts can be run with help of interactive markers in RViz. The markers are shaped like the target object and can be freely moved on the table model to define start and goal state. A context menu offers options to reset the markers, to start a planning attempt or to execute a successful plan. A successful planning attempt is visualized including the trajectory and the complete planner graph. Examples for this are shown in the analysis chapter in section 5.3.

Increasing Prediction Efficiency

A critical factor for the performance of sampling based motion planners is efficient control sampling. This is especially so if the planners rely on predictive sampling. Some planning attempts require thousands of push predictions, rendering the prediction time as a major factor for the overall planning time.

The initial predictor is a python ROS-node that accepts requests through a service protocol. The prediction itself is hard to optimize since it is implemented within the Keras framework. Instead, the ROS infrastructure is assumed to cause delays in high frequent throughput scenarios. The attempted solution avoids the message infrastructure and by directly running predictions in C++.

The C++ implementation loads the model description and builds the architecture using vectors in Eigen[G+10]. Each layer is initialized with the weights of the trained model and assigned to an activation function. Input and output vectors of the model are normalized and scaled as priorly specified in the export file. The prediction is run by sequentially computing the activation output of each layer and feeding it as input for

the next one.

An additional improvement accounts to planners that rely on the repeated propagation of the same push. Previous results are cached so that repeated requests can be handled without running the prediction.

5 Analysis

An essential factor for accurate and robust prediction models is the quality and amount of training data. If the dataset is skewed or flawed then this is what the model will learn. The result is overfitting which limits the transferability of the predictor.

Most of these problems can be reduced by increasing the size of the training data. With increasing sample count the value distributions generally represent the real problem better.

However, this strategy is limited when using data from robot experiments as it is the case in the pushing scenario. The process of collecting the data takes time and is even costly in many cases. Gathering tens or hundreds of thousand data samples is often not a feasible option. Furthermore, the constructed nature of setups and experiments are prone to skewed data which increases the risk of overfitting. Examining and optimizing the aggregation process is critical to acquire suitable datasets.

In order to verify the quality and suitability of the push effect data, at first the exploration process is evaluated in section 5.1.. This includes the sampling method as well as the resulting push effect feature distributions. Based on this results, the proposed prediction architectures are evaluated in section 5.2. The focus lies on the used regularization methods, the applied SE(2)-loss function and the results of the hyper-parameter optimization.

The push models are the basis for the predictive samplers used for planning. Assuming the predictors are sufficiently accurate, planners and exploration strategies are analyzed and compared in section 5.3.

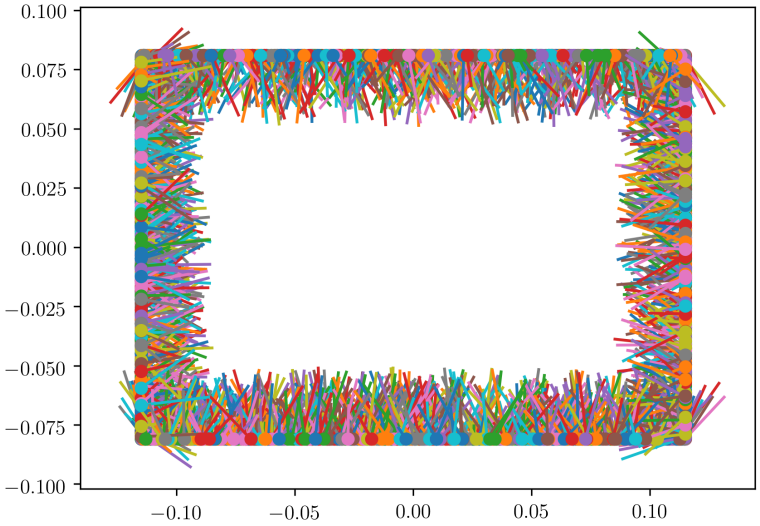
Section 5.4 describes the results of open-loop and closed-loop plan execution. The feasibility of a planner is determined by its performance in the MPC approach.

5.1 Exploration Method

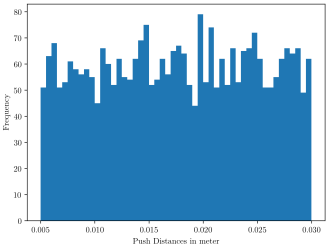
The quality of the exploration method is determined by the sampling method and the setup accuracy. Since the sampling method is supposed to generate random pushes, the distribution of the executed pushes should be random as well. This is a reasoned concern considering the implemented safety measure influences the sampling. Figure 5.1a visualizes the sampled and executed pushes applied to the box object. The border of the object is completely covered by approaches and push distances and direction appear random.

Hereafter, the edges of the box are called regarding their position towards the object frame: *bottom*, *right*, *top*, *left*.

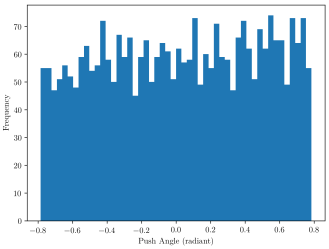
Below is a comparison of the push counts per side:



(a) Visualization of push approaches applied to the box. The length of the bars corresponds to the push distance.



(b) Push Distances



(c) Push Direction Angles

Figure 5.1: Visualization of push samples that where applied to the box

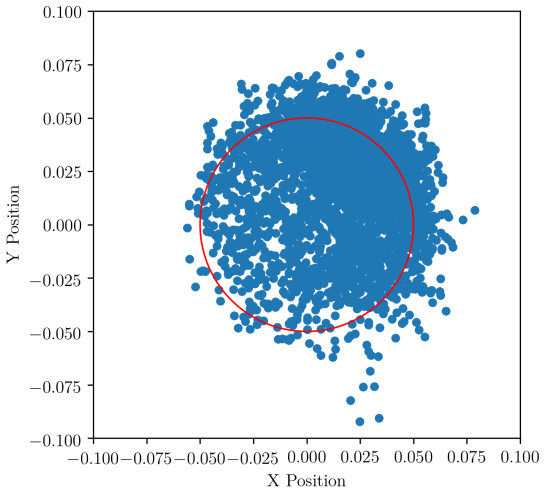
Side	Bottom	Right	Top	Left
Count	1080	657	713	521

Considering *bottom* and *top* are longer than the other sides, it is reasonable that they are sampled more often. This does not explain the preference of the *bottom* side. While the approach point distribution is visibly skewed, the push directions and distances are random random, as shown in figures 5.1b and 5.1c.

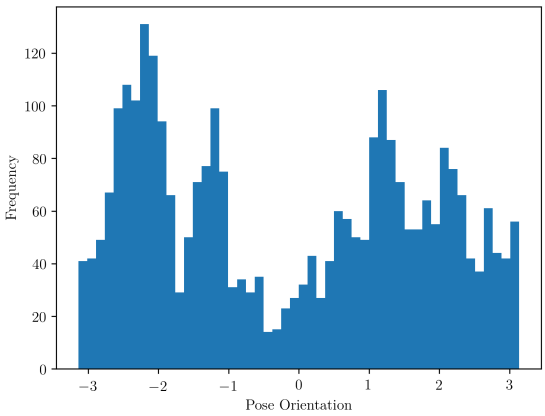
A factor that could skew the box approach distribution is the safety constraint. Figure 5.2a shows the box positions before each push. The red circle marks the border of the safety zone. If the box is outside the zone, the push sampler rejects all samples that are not approximately directed to the table center. From the visualization, it is obvious that the box is predominantly located in the positive XY-quadrant and even outside the border. The safety measure works well since there is not a single case where the box enters the emergency zone, stopping the exploration.

However, there are either more pushes in the corresponding direction or the pushes move the box further. Varying box translations could be caused by a constant offset in the object localization or the end effector position. Both factors would produce the same skew in the data.

Figure 5.3a visualizes the push direction in the surface frame plotted against the translation distance. In a perfect setup, these factors should be unrelated. The clustered region at the bottom right of the plot indicates that pushes directed in an angle between 3 to 5 radiant result in lower distances. In the opposite direction the distance appears to be shifted towards higher values.

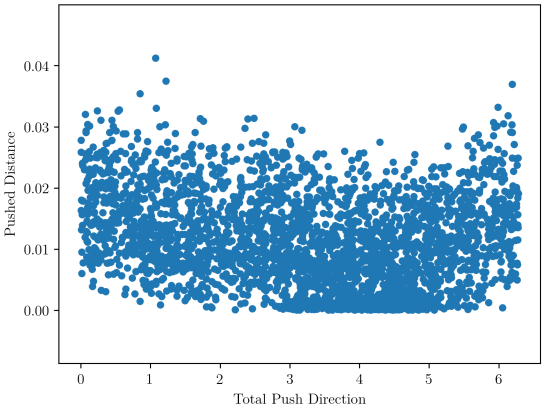


(a) Sampled Object Positions - The red circle is the safety zone

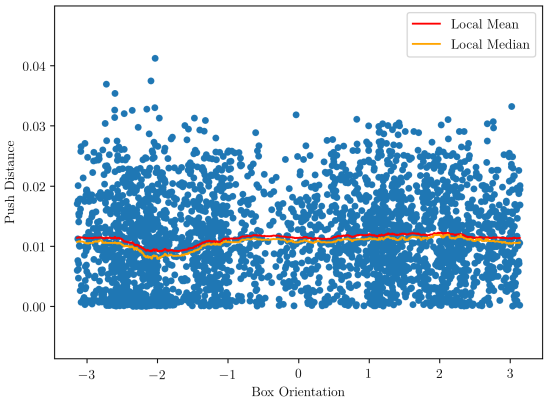


(b) Histogram of the box orientations

This offset can be problematic considering that the translation distance should be predicted. Also, this kind of error could skew the object rotation as well.



(a) Push direction angle vs distance



(b) Object orientation vs pushed distance

Figure 5.3: Impact of push direction and box orientation on the pushed distance

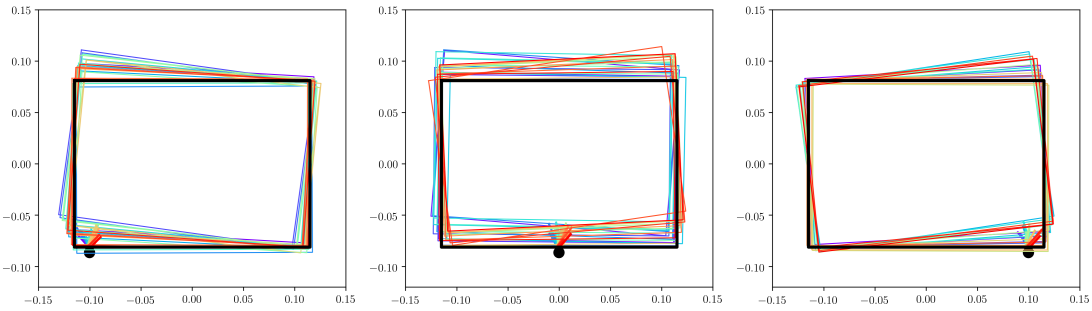
An important factor if the offset results in skew or noise is the distribution of object orientations. Since push effects are referencing the object frame, random orientations indicate random offset directions and therefore noise. Figure 5.3b shows that the object orientation has only a slight impact on the pushed distance. As a consequence it can be assumed that the offset mostly appears as noise in the push features.

The exploration analysis shows following results:

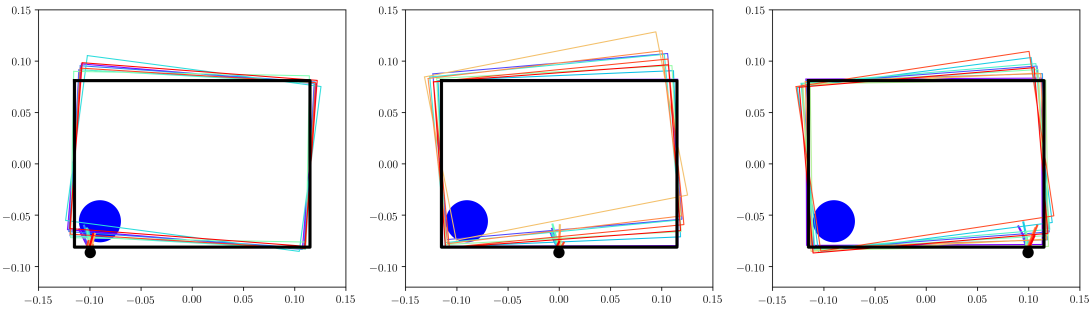
- The distribution of object positions is shifted towards the border of the safety zone
- The shift is caused by a systematic offset, probably in the object localization or end effector position
- The offset influences the push effects regarding the table frame
- Since the object orientation is random, the offset mostly appears as noise in the push effects

5.1.1 Feature Analysis

When pushing objects with the finger tip, it is easy to predict the general movement of the object. Considering pushes applied to the box, naive statements can be made about direction and rotation. For instance, pushing the left side of an edge should rotate the box clockwise. Slight changes of the push direction or approach should lead to slight alterations of the movement. Since the box is symmetric, there are symmetric pairs of pushes where the movements mirror each other. The absolute distances and rotations of symmetric pushes should be equal.



(a) Box transforms from different approach points



(b) Box transforms with weight attached to the object

Figure 5.4: Box approach effects

These statements describe features of push dynamics and can be verified in the datasets. Figure 5.4a visualizes the object transformation caused by different push approach points

and varying push angles. As expected the box rotates according to the push approach and push angle. The effects of the symmetric approaches on the left and right side of the edge roughly mirror each other.

For comparison, dataset 3 is recorded with object modification B, which is the box with a weight attached to the corner. Figure 5.4b shows how the changed weight distribution influences friction and therefore push effects.

Considering the sample data confirms the expected movement, there should exist corellations between push and effect features. They indicate how accurate push effects can be predicted. In the best case feature corellations resemble continuous functions. Obviously, push prediction is a multidimensional problem but isolated examination of push features should give deeper insight about their influence. This conforms to inspecting the partial derivatives of single push features.

Figure 5.5 contains the pairwise plots between push and effect features from dataset 1 and 2 combined. X- and Y- values of the push approach, push direction and distance are mapped against X- and Y-translation, rotation and moved distance of the box. To isolate the single features, the datasets are filtered per row as listed in table 5.1.

	Approach Point	Push Distance	Push Direction
Approach X	from bottom	3cm	orthogonal
Approach Y	from left	3cm	orthogonal
Direction	from bottom center	3cm	-
Distance	from bottom center	-	orthogonal

Table 5.1: Feature Map Row Filters

At first sight most plots seem considerably noisy. Out of the push effects, the rotation angle appears to have the most distinctive corellations. While the rotation is strongly defined by push approach and direction, the push distance is positively correlated to noise. For comparison, figure 5.6 depicts the impact of the approach point on the top and bottom sides of the box mapped to the resulting rotation.

The object translation also seems noisy and random at first. However, comparing the translations with the distance values indicates feature dependencies. For instance, in the first row, the translation distance almost exactly resembles the Y-translation, since the X-translation is just too low. Conversely, in the second row, the X-translation dominates the overall translation distance. The reason is the direction of the movement is defined by the side from which the box is pushed.

If a box is pushed from the bottom towards the positive Y-axis then this dominates overall distance. The same effect can be observed when looking at the push direction where the push approaches are also filtered to one side.

While the push distance is positively correlated to the translation distance, the is push direction seems to have no apparent impact. The maximum distance conforms to the fixed push distance of three centimeters in the first three rows. Interestingly there is a difference in the distance values comparing the X- and Y-approaches. Pushes plotted in

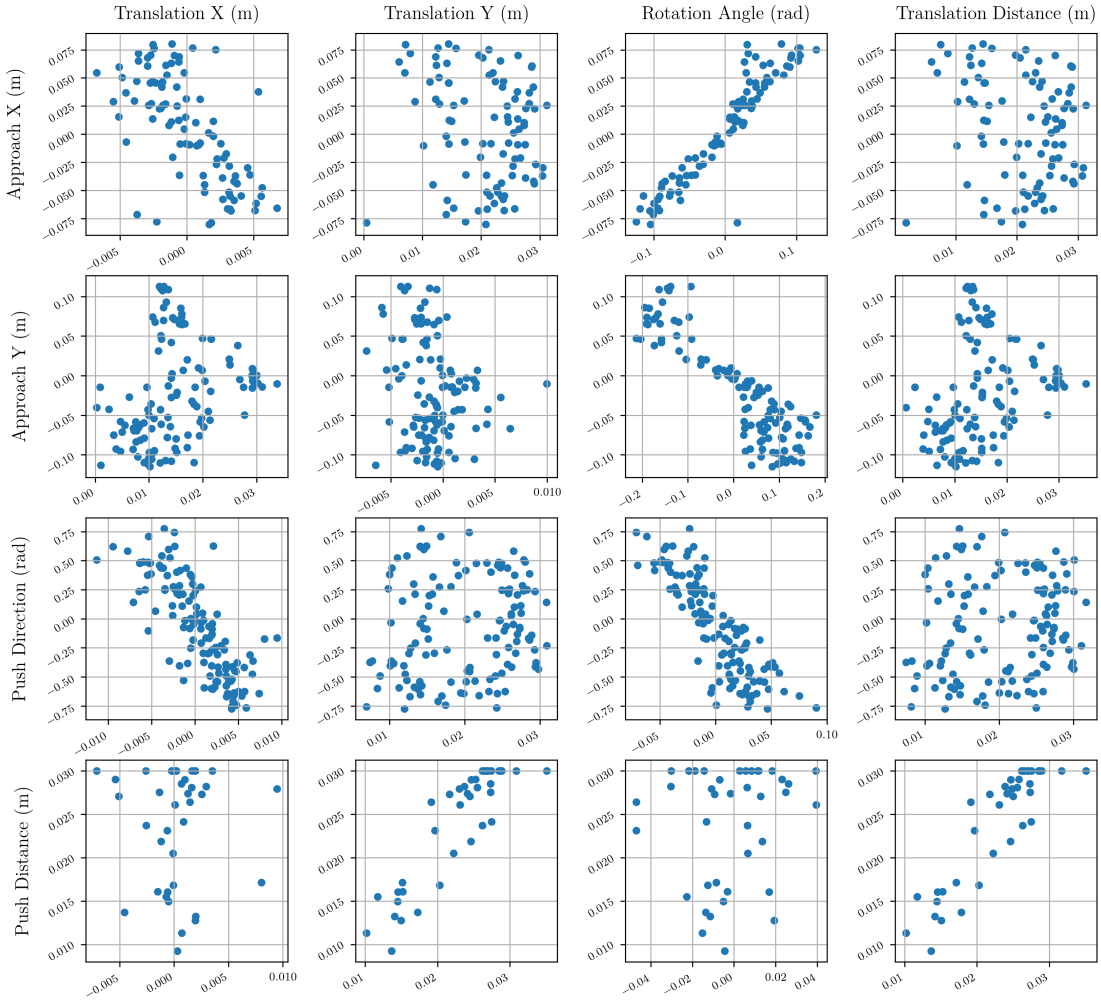
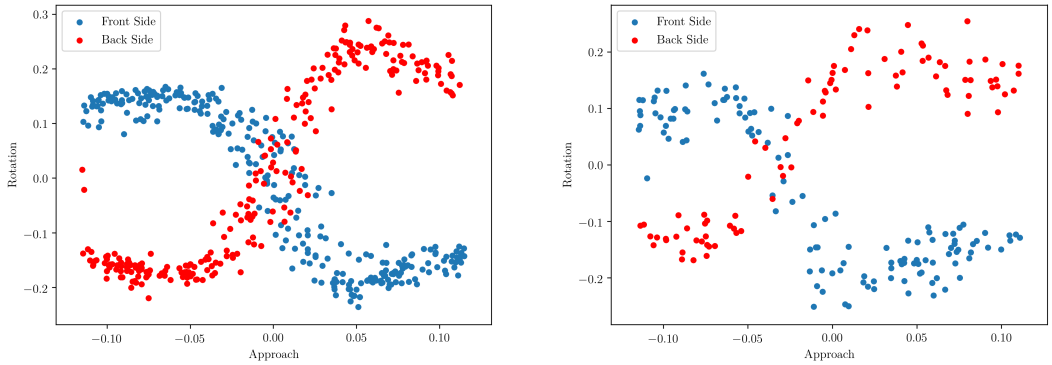


Figure 5.5: Feature Map between push and effects

Approach Y are applied to the shorter left side and thus are more prone to rotating the box when deviating from the center. This results in a higher deviation of the rotation angle and lower translation distances.

With the goal of predicting push effects in mind, several assumptions can be made. The rotation angle should be well predictable, since multiple push features correlate with it. However, the accuracy of is expected to decrease with higher push distance. The push approach is a good indicator for the direction of translations. In return, the translation distance could be hard to predict due to noise and less correlations. Ultimately, the prediction accuracy is not limited on single features but rather on their weighted combination.



(a) Box object

(b) Modified object

Figure 5.6: Approach point in relation to object rotation

5.2 Push Models - Learning and Prediction

The feature analysis outlines the relations between push and object movement. The correlations of the object rotation seem more distinct than the translation, which indicates better predictability. Obviously, there is no metric to directly compare rotation and translation accuracy. It should, however, be possible to gain insight about the relative accuracy by comparing the architectures. The baseline prototype B_0 uses separate predictors while the neural networks N_1 and N_2 optimize the SE(2)-loss. While B_0 should approximate only direct correlations, N_1 and N_2 learn a weighted representation of the output.

The assumption is that the SE(2)-loss works as a kind of regularization. If push effects belong to a multivariate distribution, the neural networks could produce more feasible results.

Lastly the different prototypes are examined for systematic errors and overfitting. In particular, the effect of regularization methods is analyzed by comparing N_1 and N_2.

5.2.1 Prediction Accuracy

Figure 5.7 depicts the prediction error of the prototypes trained on the unfiltered data. B_0 has a distance error of 1 cm, which is about twice the setup accuracy. The rotation is predicted with an error angle of 0.28 rad or 16°.

When looking at the neural networks, the gap between translation and rotation is even larger. Both N_1 and N_2 have a higher translation accuracy than the baseline while the rotation error is increased. In other words, the neural networks predict rotations relatively less accurate than translations. Considering the SE(2)-loss weights yaw and distance in a ratio of 1:2, the rotation error is 40% higher than the translation error. This indicates that the weight ratio is not optimal for the feature distribution. Otherwise, the neural network prototypes should reach the accuracy of B_0. In the end, N_1 and N_2 have a lower SE(2)-loss, which is no surprise since B_0 is not optimized for it.

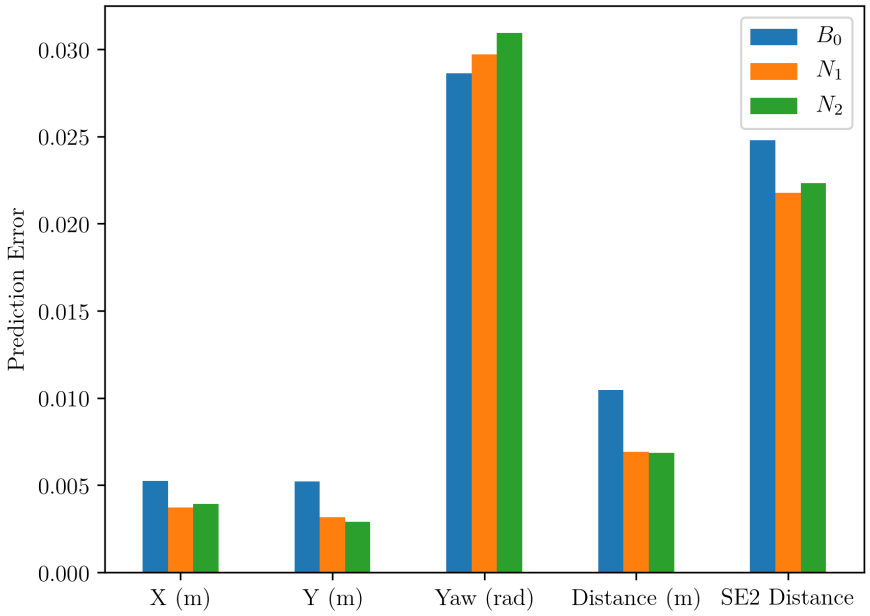


Figure 5.7: Models Error on the Test Set

5.2.2 Hyper-parameter Optimization

There are two winning architectures H^* and H_f^*, corresponding to the filtered and unfiltered dataset. They are the results with the lowest validation loss after 600 optimization trials per session. The minimal validation loss of H^* is 0.0176 and that of H_f^* is only 0.0148, which is considerably lower than those of the prototypes.

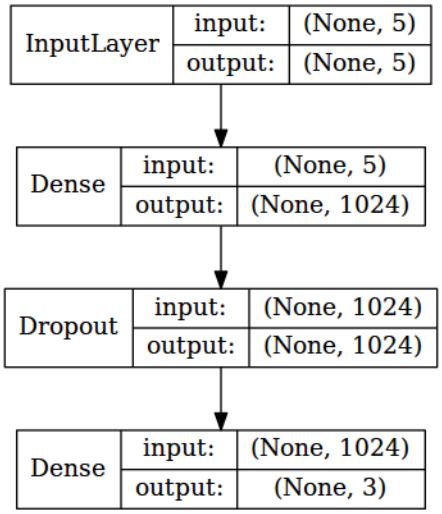
Interestingly, the two architectures are very different. Below is a depiction of the results.

It is apparent how the filtering influences the network structure. This affects the number of layers, unit sizes and regularization methods. H^* has only a single hidden layer with the maximum unit size of 1024. Dropout is practically disabled with a value of 0.0005 and the l2 penalty weight is not too high either. In combination with a high learning rate, H^* is a comparably flexible and fast learning architecture.

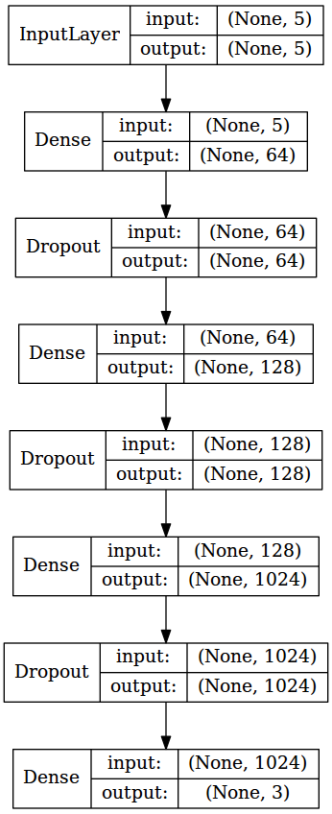
In contrast, H_f^* has three hidden layers with increasing unit counts. The dropout in the first and third layer are active with values of 0.1444 and 0.0607. Structure and regularization indicate a multi-scaled feature extraction that is not present in H^*. Also, the learning rate is much lower, which indicates overall slower and more robust learning.

However, the architectures are not necessarily the best performing ones. The optimization process compares the validation loss of single training runs. A single training run is not significant enough to conclude about the overall accuracy. Especially in less regularized networks, there is always the chance of randomness. Therefore, the accuracy has to be observed over multiple runs.

The average validation loss of H^* over 20 training runs is 0.0182. That is actually



(a) Architecture of H^* - optimizer: *Adam*, learning rate: 0.00155, activation: *ReLu*, dropout: 0.0005, l2: 0.00024



(b) Architecture of H_f^* - optimizer: *Adam*, learning rate: 0.00061, activation: *ReLu*, dropout: [0.1444, 0.0004, 0.0607], l2: 0.00021

higher than those of N_1 and N_2 with 0.0178 and 0.0179. The same is true for the models trained on the filtered data. H_f^* has an average validation loss of 0.0167 while that of N_1 and N_2 is 0.0176 and 0.0152. In average N_1 and N_2 actually perform better than H^* and H_f^* respectively.

Interestingly, there are similarities between the optimized models and the prototypes. H^* has a similar flat structure as N_1 while H_f^* uses multiple layers with dropout like N_2. The results of the prototypes match this resemblance. N_1 performs better with the unfiltered data while N_2 excels with the filtered data.

That does not mean that the prototypes are optimal. In fact, probably many of the optimized models perform better. The problem is, that the validation loss can be influenced by chance. Also, overfitting is not really avoided since many of the models show much lower training accuracy. Ideally, the optimization process should include multiple training runs and use an objective that punishes overfitting.

5.3 Planning Approaches

The planners described in section 3.4 are separated in to four exploration strategies.

- Random Exploration
- Directed Exploration
- Steered Exploration
- Chained Exploration

They are named after their combination of state propagators and control samplers. The *Random* planning strategy uses the default state propagator and random control sampler. *Directed* planning uses a directed control sampler instead. *Steered* planning is based on a steering state propagator. The *Chained* planning strategy relies on the constrained control sampler that prefers sequences of similar pushes. All predictive planners are using the prototype N_2, assuming that the predictions are sufficiently accurate.

Independent of configurations, these strategies show distinguishable planning results. Examples are visualized in RViz, including planning tree, solution paths and obstacles. Start and goal states are marked as blue and red boxes respectively.

Random Exploration

The random exploration strategy is very time efficient, since the random control sampler does not rely on prediction. The planning tree spreads into all directions almost equally, only influenced by the goal bias.

Even if the single exploration steps are undirected, the path always converges to the goal. The solutions contain detours since adjacent pushes are completely unrelated. This is mostly dependent on the *maximum control duration* parameter, which defines how often successful controls are propagated. Figure 5.9 shows three planning trees with duration limits of 1, 10 and 25 from left to right.

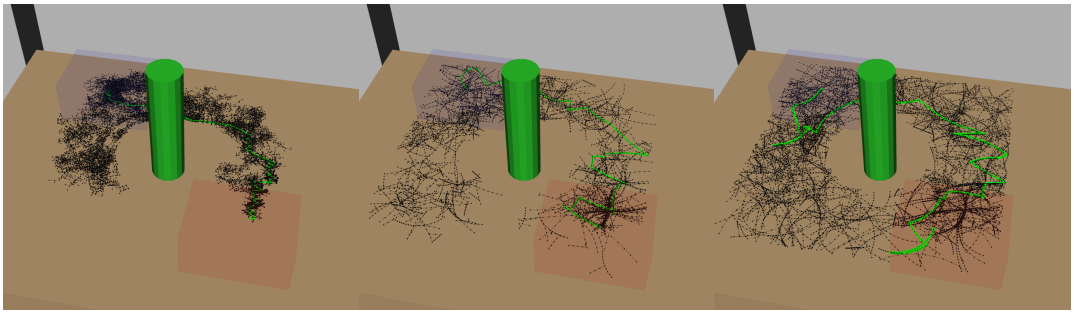


Figure 5.9: Random Plans with duration limit set to 1, 10, 25 (left to right)

In all plans the goal bias is set to 0.2. It is apparent how higher duration limits cause longer exploration leaps but also more detours. The leftmost plan with the duration limit

set to 1 is restricted to local exploration only. Since no control is propagated repeatedly, all new states are sampled within proximity of existing ones. Accordingly, the planning tree is dense with very short connections.

With larger limits, the exploration tree is further distributed over the reachable space. Controls are further propagated, however, not necessarily goal-directed. Also, the resulting solution path contains much more detours. At first sight, the trees seem to contain shorter paths. However, these are apparently not feasible because of the orientation difference between states.

Directed Exploration

This strategy produces similar results to the random exploration. However, directed sampling produces planning trees that span the explored space much more efficiently. Especially with higher max control duration values, this planner produces fairly usable plans. Examples are shown in figure 5.10. The planners are initialized with a goal bias of 0.2 and duration limits of 1, 10 and 25 respectively. The first two results resemble those of the random approach. Higher duration limits apparently allow better use of the directed controls. The planning trees seem better distributed and the solution paths smoother.

Figure 5.10: Directed Plans with duration limit set to 1, 10, 25 (left to right)

Steered Exploration

Steering seeks to connect newly sampled states directly to the exploration tree. Combined with a high goal bias, this approach can be very efficient in some scenarios. If there are no obstacles, the steering produces simple curved paths with few detours. The reason is that many states can be connected by a sequences of the same control, which is exactly the purpose of steering.

However, the results are strongly affected by obstacles. The planner can't exploit the high goal bias in that case and is forced to explore different other regions. This can lead to high planning times, since steering relies on predicting high numbers of push controls. With lower goal bias the tree spreads into all directions, aggravating the problem. Then

the solutions are comparable to those of random planning with a high duration limit. Figure 5.11 shows an example of a steered plan around an obstacle.

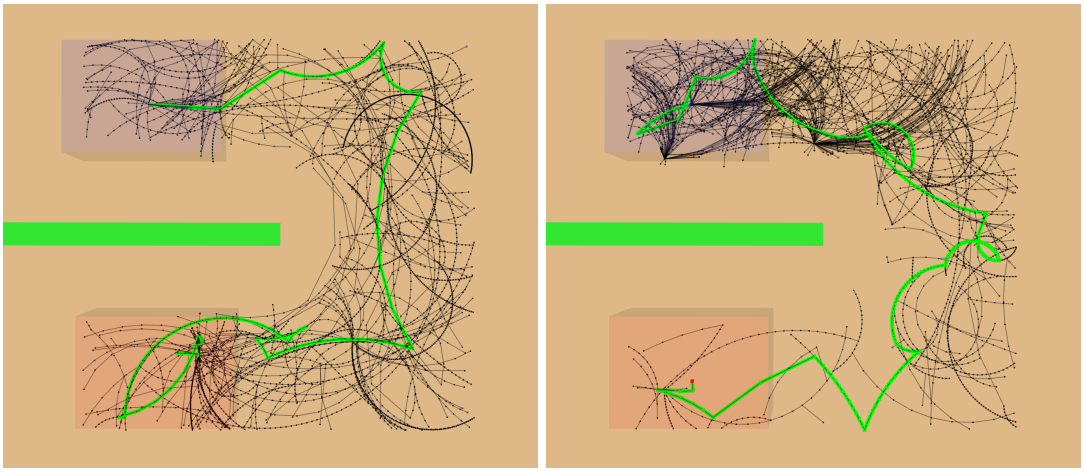


Figure 5.11: Steered Planning with goal bias set to 0.2 (left) and 0.5 (right)

Chained Exploration

The intention behind chained controls is to resemble continuous push behavior. Adjacent controls are sampled within an approach point neighborhood to reinforce similar pushes. As a result, the planning trees contain paths that are goal-directed and smooth at the same time. Most solutions consist of longer continuous segments that are pushed from the same side of the object. By allowing the approach point to cross the edge of the box, abrupt direction changes are possible as well. Figure 5.12 shows two plans with a high goal bias of 0.75. The max control duration is set to 2, so controls are only propagated twice.

The boxes indicate the objects position during the motion. Corresponding to the side from where the box is pushed, both paths can be divided into three segments. That is pushes from the left side move the object to the right. Then the object is pushed down past the obstacle. The last leap is a rather straight sequence of pushes towards the goal. This is especially apparent at the right image.

This behavior almost appears strategic, albeit the planning tree shows how random the exploration is performed. In fact, this approach is very similar to the directed exploration, just with smooth paths.

Similar to the steered approach, this method excels at push plans without obstacles. When obstacles are involved, the planning time and exploration complexity increases significantly. The reason is the directed and constrained sampling of new pushes which requires many predictions. The generated solutions are generally simple and consist of several continuous segments of pushes. Also, the generated plans often look similar, which is important for closed-loop execution.

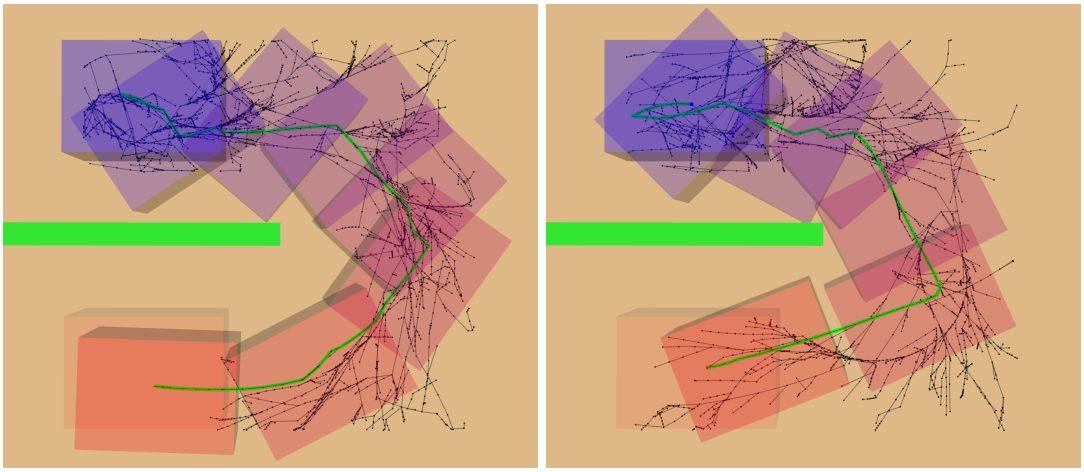


Figure 5.12: Chained sampling, goal bias 0.75, max control duration 2

5.4 Plan Execution

The last section shows that predictive motion planning can produce feasible goal-directed trajectories. As expected, these can not be executed in an open loop. Figure 5.13 shows how the execution of a comparably simple plan can fail. The green line is the target trajectory leading to the goal position marked by the red box. The orange line is the actual path of the object during execution, ending in the position marked by the green box. This example of path deviation highlights the effect of error accumulation. The target trajectory consists of 20 waypoints that are reached by an equal number of pushes. Since the final object orientation deviates about 80°, the average orientation error per push is 4°. This value is 2.5 times higher than the average prediction error, which indicates small inaccuracies in the setup. Some imperfections can't be avoided, especially in dynamic robot environments. Therefore, closed-loop execution of predictive plans is essential.

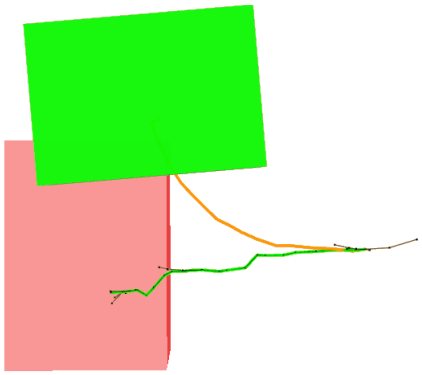


Figure 5.13: Deviating path from open-loop execution

5.4.1 Greedy Distance Minimization

This approach attempts to examine three main questions:

- Does the prediction accuracy suffice for directed pushes?
- Is the SE(2) distance a valid cost function?
- Will this method move the object to the goal target?

Since no planning and obstacle avoidance is implemented in this method, the experiments are comparably simple. Linear translations and in place rotations are tested with arbitrary goals.

Linear translations are mostly sampled directly towards the goal, as expected. All approach points are applied to the goal averted side, mostly centered, and the object always reaches the target. However, in many cases, the pushes are repeatedly sampled close to the corners of the box, leading to some kind of walking movement.

The first suspicion is that predictions indicate higher translation distances at the corners. This can be disproved by analysing the different predictors. The problem appears to be caused by the following behavior. When a prediction error causes a slight rotation of the object, the rotation distance towards the goal increases. The optimal push would countersteer this rotation in the attempt to minimize the SE(2) distance. If the object rotates stronger than predicted, and into the other direction, the next push would again correct this overshooting rotation. This escalates and the push approaches drift towards the edges.

Eventually, this issue is an effect of accumulated prediction errors, even though the predictions are made independently. At the same time it shows that the SE(2) distance is valid as a cost function, otherwise, the rotation error would not have this strong effect. Its arguable that reducing the weight of the rotation would improve this problem.

This problem does not appear with in-place rotations. All pushes are applied at diagonally opposed corners, so that the box indeed rotates in place. By alternating the corners the pushes stabilize the position of the box. No error accumulation is observed.

Even when pushing the object towards arbitrary goals, there is seldom the case of repeated overshooting. Two kinds of push movements can be observed often.

1. Execution of similar pushes that move the object in a circle
2. Alternated rotation and translation by switching between two approach points

The first behavior corresponds to movements around a certain center of rotation. In that case the local optimal push resembles the global one. The second behavior could be caused by too narrow rotation centers or absence of any valid ones.

Concluding, the greedy approach shows that prediction and cost function are feasible, since the object successfully reaches the goal. However, even if the results meet the assumptions, there are unexpected effects of accumulated prediction errors. These could affect performance and accuracy of further execution approaches, and might be even more problematic in continuous pushing scenarios.

5.4.2 Model Predictive Control

Two closed-loop MPC approaches are presented, in order to perform goal-directed push operations. Simple MPC repeatedly samples plans and only executes the first push. Multi-step MPC executes a trajectory and only computes a new plan if the object deviates from the path. Both approaches are tested using random, directed, steered and chained planning strategies.

If no obstacle is present, simple MPC moves the object directly towards a goal if directed, steered or chained planning is used. With obstacles, the execution of random, directed and steered planning is problematic. The first controls are too random to actually move the object along a certain path. The chained planner allows moving the object past the obstacle, but often gets stuck in local minima. Then the object is pushed back and forth, since adjacent plans are too diverse.

Multi-step MPC performs far better than simple MPC. Depending on the threshold the first couple of steps of a plan are executed. If the threshold is set to value of the goal distance threshold, the object deviates from the path after about three pushes. Higher values increase the deviation tolerance and improve the process flow. In return they increase the risk for object colisions. In fact, obstacles are an issue for predictive planning in general. If collision checking uses exact geometry of the objects, plans often lead the object very close to obstacles. Small prediction errors cause the object to be pushed into obstacles, which invalidates the process. This could be avoided by optimizing planners or simple heuristics like pushing the object away from obstacles if they are too close.

In comparison, the multi-step approach is superior to simple MPC, since successive controls are less likely to be opposing. In combination with chained planning this is a promising approach to realize goal-directed push operations.

6 Discussion

This chapter discusses the methods and results of the presented approach. This includes alternative approaches as well as potential future work.

Pushing

The introduction of a unified push specification serves the purpose of streamlining the process from exploration to execution. Restricting the push method to linear movements applied at always the same height simplifies the problem drastically.

Obviously, pushes can be executed in all kinds of variants. The pusher could draw curved movements or alter the height of the contact point. This would be useful for continuous pushes or irregularly shaped objects. However, a larger action space creates a much more complex prediction problem. This would require extended exploration and much more sophisticated learning architectures. Restricting the action space allows short learning times and potentially more accuracy.

Another aspect of the used push method is the single contact point. Considering the goal is to perform predictable and stable pushes, multiple contacts could be more suitable. Already a second contact point would stabilize the object so that undesired rotations are prevented. It can be assumed that suitable learning architectures would be similar to those used in this approach. In return, this approach would increase the control space. In order to hit both contact points, the orientation of the pusher needs to be controlled as well. Depending on the robot setup and end effector range this strongly restricts transferability.

SE(2) states are a simple but accurate representation for object poses and movements. They can be used for all scenarios where the object is rigid and stays on the surface without tipping over. For pushes that roll or overturn the object, the state space could be extended to SE(3). Both representations are unspecific to the pushing problem. An example for a specific representation for push effects is the center of rotation. Along with the moved distance, this allows expressing push effects with only two features. Since the representation is push specific, this could enforce predictions to be feasible.

In addition, this representation could be used to generate continuous object trajectories. Such trajectories could be described as continuous functions of rotation centers. These could be mapped to a continuous sequence of pushes which in return specifies the pushers' trajectory. Obviously, this application exceeds the scope of simple pushing and rather constitutes a control problem. Planning and actually following such trajectories requires methods that are completely different to this approach, and might be limited to specialized hardware with nearly flawless calibration. Equally, it would be considerably more challenging to realize such an approach as a robust and transferable application.

Exploration

Letting the robot autonomously explore push effects is a practicable way to gather training data. It can also easily be integrated with the learning procedure. If required, learning could be performed online or even stopped, after a certain prediction accuracy is reached.

An observed problem with the exploration procedure is that the quality of the collected data relies on the exploration method. If the exploration method is flawed, the data might be as well. Using the data for prediction models could lead to overfitting on the setup. In order to eliminate potential skews, frequent checks and corrections are required. That limits the autonomy of the robot and costs time. A possible solution would be to collect and integrate data sets from different setups.

Another concern of autonomous exploration is the robustness and safety of the method. Ideally, the exploration process can run indefinitely without aborting or the need for intervention. Since there is no bug-free system, appropriate safety and error handling is required. Heuristics like the proposed safety zones are practicable but can in return impair the exploration process. A moderate solution would include continually adjusting the object position while ensuring that the explored push distribution is still random.

Further, this could be extended to perform strategic exploration methods when using online learning. For instance, the exploration process could focus on pushes where the current prediction accuracy is rather low. This could be implemented as an optimizing process that terminates when the robot has learned a sufficiently good model. The reduced redundancy would result in shortened exploration runs and possibly more accurate results.

Prediction

The proposed predicted function is restricted to a single object shape. Generic push learning would include learning a representation of the object shape and appearance along with the push. That requires more sophisticated architectures like for instance CNNs or SE(3)-Nets. This is necessery for robust applications involving unknown objects.

Another method to increase adaptability to unknown objects is to factor in analytical information. Push effects partially depend on the center of friction. The center of friction can be approximated by multiple pushes from different directions. If pushes applied to an unknown object are predicted with a large error, feeding in the corrected center of friction could improve the results. It is equally possible to create a weight distribution of the object by using available force torque sensors. The weight distribution influences the friction distribution and could complement the learned model.

A different learning problem is the scenario of sequential or continuous pushing. Instead of a single push prediction, the effects during a sequence of pushes could be learned by RNNs. This allows feeding in the observed prediction error, so that the model corrects the movement. In return this scenario limits planning approaches and might be too inefficient for online use.

Planning and Execution

Using motion planners for predictive push planning is an efficient way to compute object trajectories. Even though the solution paths are not directly executable, they can be used for closed-loop control strategies. However, there are several points to be considered.

Often the solutions are long-winded and awkward. This is a general issue in motion planning that can be solved by using optimizing planners like RRT* or STOMP.

Another issue is that some planners are inefficient because of the need for predictive sampling. Using an inverse model could solve this problem, although, pushes are potentially ambiguous.

The MPC approach is a promising attempt for closed-loop plan execution. A major problem is that consecutive plans are unrelated and lead to inefficient trajectories. Ideally, the planner should create repeatable solutions that MPC converges to a certain path. This could be realized by reusing the planning tree or by implementing a closed-loop planner, similar to the approach of Kuwata et al. [Kuw+09].

6.1 Conclusion

The presented approach attempts to integrate solutions for all subtasks necessary to execute self-learned push operations. For the applied scenario, this exposes certain challenges and limitations. It shows that planar push effect models of rigid objects can be learned by exploration. Extending this to non-rigid objects or non-planar movements increases the problem space drastically. This arguably requires a vastly extended exploration process and much more complicated learning architectures. Eventually, it is a trade-off between expressiveness of the model and time or computational efficiency.

Applying prediction models for sampling-based motion planning is most of all an efficient method for finding goal directed push trajectories. The planners can be designed to meet the requirements to produce collision-free, smooth or even continuous paths. Even if more complicated push effect models with higher dimensions were applied, they would still be feasible solution. The biggest limitation affects plan execution.

Experiments with direct plan execution show how accumulated errors invalidate open-loop controls. In order to safely move objects along prolonged and collision free paths, closed-loop approaches like MPC seem promising. However, the performance strongly depends on the underlying planner. For efficient and goal directed push operations, MPC requires stable and repeatable planning results.

Bibliography

[1] Pulkit Agrawal et al. "Learning to Poke by Poking: Experiential Learning of Intuitive Physics". In: *CoRR* abs/1606.07419 (2016). arXiv: 1606.07419. URL: http://arxiv.org/abs/1606.07419.

[2] J. Bergstra, D. Yamins, and D. D. Cox. *Hyperopt - Distributed Asynchronous Hyperparameter Optimization in Python*. 2018. URL: http://hyperopt.github.io/hyperopt/ (visited on 07/26/2018).

[3] J. Bergstra, D. Yamins, and D. D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, pp. I-115–I-123. URL: http://dl.acm.org/citation.cfm?id=3042817.3042832.

[4] Arunkumar Byravan and Dieter Fox. "SE3-Nets: Learning Rigid Body Motion using Deep Neural Networks". In: *CoRR* abs/1606.02378 (2016). arXiv: 1606.02378. URL: http://arxiv.org/abs/1606.02378.

[5] Richard H. Byrd et al. "A Limited Memory Algorithm for Bound Constrained Optimization". In: *SIAM Journal on Scientific Computing* 16.5 (Sept. 1995), pp. 1190–1208. DOI: 10.1137/0916069.

[6] François Chollet et al. *Keras*. 2015. URL: https://keras.io (visited on 07/26/2018).

[7] Open Source Robotics Foundation. *ROS - Robot Operating System*. 2018. URL: http://www.ros.org/ (visited on 07/22/2018).

[8] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org (visited on 07/30/2018).

[9] T. Hermans et al. "Learning Stable Pushing Locations". In: *IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL-EPIROB*. 2013. URL: http://www.ias.tu-darmstadt.de/uploads/Team/TuckerHermans/hermans-icdl2013.pdf.

[10] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (Dec. 22, 2014). arXiv: http://arxiv.org/abs/1412.6980v9 [cs.LG].

[11] Marius Kintel. *OpenSCAD - The Programmers Solid 3D CAD Modeller*. 2018. URL: http://www.openscad.org/ (visited on 07/22/2018).

[12] Marek Kopicki et al. "Learning modular and transferable forward models of the motions of push manipulated objects". In: *Autonomous Robots* 41.5 (June 2017), pp. 1061–1082. ISSN: 1573-7527. DOI: 10.1007/s10514-016-9571-3. URL: https://doi.org/10.1007/s10514-016-9571-3.

[13] J.J. Kuffner and S.M. LaValle. "RRT-connect: An efficient approach to single-query path planning". In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*. IEEE. DOI: 10.1109/robot.2000.844730.

[14] Y. Kuwata et al. "Real-Time Motion Planning With Applications to Autonomous Urban Driving". In: *IEEE Transactions on Control Systems Technology* 17.5 (Sept. 2009), pp. 1105–1118. DOI: 10.1109/tcst.2008.2012116.

[15] Manfred Lau, Jun Mitani, and Takeo Igarashi. "Automatic learning of pushing strategy for delivery of irregular-shaped objects". In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, May 2011. DOI: 10.1109/icra.2011.5979740.

[16] Mike Lautman. *Moveit! Tutorials - Planning with Approximated Constraint Manifolds*. 2018. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/doc/planning_with_approximated_constraint_manifolds/planning_with_approximated_constraint_manifolds_tutorial.html (visited on 07/23/2018).

[17] Kevin M. Lynch and Matthew T. Mason. "Stable Pushing: Mechanics, Controllability, and Planning". In: *The International Journal of Robotics Research* 15.6 (Dec. 1996), pp. 533–556. DOI: 10.1177/027836499601500602.

[18] K.M. Lynch. "The mechanics of fine manipulation by pushing". In: *Proceedings 1992 IEEE International Conference on Robotics and Automation*. IEEE Comput. Soc. Press, 1992. DOI: 10.1109/robot.1992.219921.

[19] Danylo Malyuta. *apriltags2_ros - A ROS wrapper of the AprilTags 2 visual fiducial detection algorithm*. 2018. URL: http://wiki.ros.org/apriltags2%5C_ros (visited on 07/21/2018).

[20] Matthew T. Mason. "Mechanics and Planning of Manipulator Pushing Operations". In: *The International Journal of Robotics Research* 5.3 (Sept. 1986), pp. 53–71. DOI: 10.1177/027836498600500303.

[21] F. Pedregosa et al. *scikit-learn - Machine Learning in Python*. 2018. URL: http://scikit-learn.org/stable/ (visited on 17/23/2018).

[22] Physical and Biological Computing Group - Department of Computer Science - Rice University. *OMPL - The Open Motion Planning Library*. 2018. URL: https://ompl.kavrakilab.org/ (visited on 07/17/2018).

[23] PyData. *Python Data Analysis Library*. 2018. URL: https://pandas.pydata.org/ (visited on 08/18/2018).

[24] Robotiq. *3-Finger Adaptive Robot Gripper*. 2018. URL: https://robotiq.com/products/3-finger-adaptive-robot-gripper (visited on 07/22/2018).

[25] Universal Robots. *UR5 Robot*. 2018. URL: https://www.universal-robots.com/products/ur5-robot/ (visited on 07/22/2018).

[26] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: (Sept. 15, 2016). arXiv: http://arxiv.org/abs/1609.04747v2 [cs.LG].

[27] Federico Ruiz-Ugalde, Gordon Cheng, and Michael Beetz. "Fast adaptation for effect-aware pushing". In: *2011 11th IEEE-RAS International Conference on Humanoid Robots*. IEEE, Oct. 2011. DOI: 10.1109/humanoids.2011.6100863.

[28] Marcos Salganicoff et al. "A Vision-Based Learning Method for Pushing Manipulation". In: *IRCS TECHNICAL REPORTS SERIES*. 1993.

[29] Ioan Sucan. *OMPL Control Sampler Class Reference*. 2018. URL: https://ompl.kavrakilab.org/classompl%5C_1%5C_1control%5C_1%5C_1ControlSampler.html (visited on 08/21/2018).

[30] Ioan Sucan. *OMPL State Propagator Class Reference*. 2018. URL: https://ompl.kavrakilab.org/classompl%5C_1%5C_1control%5C_1%5C_1StatePropagator.html (visited on 08/21/2018).

[31] Ioan A. Sucan and Sachin Chitta. "Motion planning with constraints using configuration space approximations". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, Oct. 2012. DOI: 10.1109/iros.2012.6386092.

[32] Ioan A. Sucan and Sachin Chitta. *MoveIt!* 2018. URL: https://moveit.ros.org/ (visited on 07/22/2018).

[33] T. Tieleman and G. Hinton. "Rmsprop: Divide the gradient by running average of its recent magnitude". 2012. URL: https://www.coursera.org/lecture/neural-networks/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude-YQHki.

[34] Sean Walker and J. Kenneth Salisbury. "Pushing using learned manipulation maps". In: *2008 IEEE International Conference on Robotics and Automation*. IEEE, May 2008. DOI: 10.1109/robot.2008.4543795.

[35] John Wang and Edwin Olson. "AprilTag 2: Efficient and robust fiducial detection". In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2016.

[36] Thiemo Wiedemeyer. *Kinect2 Calibration*. 2018. URL: https://github.com/code-iai/iai_kinect2/tree/master/kinect2_calibration (visited on 2018).

[37] Mitchell Wills. *apriltags_ros - A ROS wrapper for the AprilTags C++ library*. 2018. URL: http://wiki.ros.org/apriltags%5C_ros (visited on 07/24/2018).

[38] Microsoft - XBOX. *Kinect Camera*. 2018. URL: https://support.xbox.com/en-US/browse/xbox-one/accessories/kinect (visited on 07/22/2018).

[39] Haolin Yang, Fuchun Sun, and Di Guo. "Pushing operation of manipulator based on experience learning: Position prediction of an object and pushing analysis". In: *2014 International Conference on Multisensor Fusion and Information Integration for Intelligent Systems (MFI)*. IEEE, Sept. 2014. DOI: 10.1109/mfi.2014.6997642.