Visual Prediction from Active Manipulation with a UR-10 Robotic Arm

Faiz Mirza

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2018

Abstract

Children learn about the causal structure of the world and the effects of their actions at a relatively young age, from a combination of active trial and error and observation of others movements. We would like to explore how to achieve some of these capabilities in robots, allowing them to more autonomously learn about objects and their behavior in specific environments. A key objective of this project is to collect data from robots performing diverse manipulation tasks, and to train predictive models of the effects of the various actions. Starting with simple regression models that predict a next state from an initial state and robot action, we plan multi-action paths between two object states. We show that computing such plans with an informed model outperforms a baseline model without this information. We view this as a step towards a robot that can continually "Learn by Doing".

Acknowledgements

I'd like to give thanks to the members of the RAD Lab, in particular Daniel for their help with this study and with setting up the UR-10. Thank you to my family who always supports me, and to my friends who acted interested when I talked to them about this project. Thank you to my dog, Hazel.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Faiz Mirza*)

Table of Contents

Chapter 1

Introduction

1.1 Motivation

It's the year 2017 and robots are firmly ingrained as part of the present and the future. While not yet quite as ubiquitous in everyday life as a science fiction movie would make one believe, robots are starting to become normalized in many different industries. Robots lead the way in space travel, with rovers exploring the moon and Mars sending data from afar. In the field of medicine, robotic surgery is right on the cutting edge of medical technology. In some parts of the world such as Korea and Japan, humanoid robots are already caring for young children and the elderly [1]. Perhaps most well known of all, Roombas, robotic vacuum cleaners, have become relatively common in consumer homes. Robots are establishing themselves in a multitude of industries, pushing past direct control and functioning autonomously.

The future of robotics is to create autonomous robots. By doing so, robots are able to perform their tasks without a human constantly telling them how to do each and every sub task. Instead, instructions develop into more abstract and high-level directions, leading to robots that are more consumer friendly. Instead of telling the robot how to perform a task, the robot can simply be told what task to perform and should use its sensory capabilities to calculate on its own how exactly to perform the given task. These higher level tasks can range from "pick up the banana", to "move to coordinate X, Y", and can even be as high level as "drive to Edinburgh airport". The fundamental task for an autonomous robot is motion planning, whether that be planning a path for the robot itself or planning a trajectory for a manipulation device controlled by the robot.

Motion planning is the process by which an autonomous robot finds a path to a goal

state. Given an initial state and a goal state, motion planning involves finding the series or set of joint torques, motor speeds, wheel angles, etc., that need to be set to reach the goal state in an optimal way. This state can include position, orientation, objects in the world, battery charge level, or any number of other variables. For any motion planning algorithm, the relationship between the parameters (torques/speeds/angles) and changes in state must be established. If the motion planning takes place in an ideal environment without interacting with the world around it, such as a robot manipulator moving to another location without manipulating an object, the relationship between parameters and state can be established with kinematic equations [2].

Inverse kinematics can provide exactly where the end effector of the robot is, given any set of parameters. However, inverse kinematics alone cannot be used to determine the effect that the manipulator will have on external objects or the world. For example, a robot cannot know the exact final state of a ball that it pushes without knowing the size and weight of the ball, the friction of the floor, and many other physical properties about itself, the ball, and the world around them. In addition, while the robot can provide the exact position of the end effector relative to the base of the robot unless the base is fixed, the position of the end effector relative to the world can only be approximated. In order to predict the current state of the environment, the results of an action, or position of a robot, robots need models for the environment and for the behavior of objects they interact with [3].

The basic goal of robotics is to automate tasks that a human would ordinarily have to do manually. This begs the question, how do human beings estimate the state of their environment and the results of their actions? If a robot is built to replace a human doing the same task, it logically follows that a human will need the same information that a robot would need to perform the task. Identifying how a human predicts the physical properties of objects and the environment will guide a method for establishing models for a robot to do the same. One explanation for human physical scene understanding is that people have internal models for predicting physics, sensing their environment, and even for motor control [4]. The types of models a person uses, whether that be a feed forward model, a feedback model, or an entire internal physics engine, are still a topic of research and discussion, however, it is generally agreed that humans use some form of internal models, just as an autonomous robot would need.

People are not computers, and while a human might have internal models, these are not models that somebody programmed into the person. These physics and motion models must have been acquired at some point in the person's early life. Studies show

that infants acquire and refine their internal models regarding the physics of objects when they pick them up and manipulate them [5]. Over time, infants and children intuitively know how to move objects to achieve a desired result. Ideally, robots should also be able to acquire models for their environment by interacting with that same environment. By building the robot's models based on data collected by the robot itself, the robot is able to "learn by doing", taking inspiration from the way humans learn.

1.2 Research Goals

In our study, we wish to demonstrate the ability of a robot to acquire models for the behavior of objects in response to that robot's actions entirely from manipulating those objects. Using this robot object interaction data, collected in a relatively random manner, we build a machine learning model to provide predictions on the end state of the environment given the start state and robot action. With this model established, we then close the loop by using the model to predict the action required to reach the given end state from the current state. This end state could be reached in one action or a series of actions taken by the robot. We claim that an experimentally learned model will perform better than a baseline random model as well a simple linear extrapolation. This will be demonstrated by predicting the results of UR-10 manipulations as well as using the robot to move objects to a designated point efficiently. This will show that, like a young human being, a robot can also be programmed to learn from the random interactions it has with the world.

1.3 Experimental Outline

Robots learning from experience is not a novel concept, as it is the most obvious cross section of robotics and machine learning. It is a logical step in the development of autonomous robotics, and in Chapter 2 of this paper, we discuss other studies that have been done on this topic, studies that motivate this study, and adjunct topics as well. There are also other studies that take motivation from different models of human learning, attempting to make robots understand their environment in completely different ways than our"learn by doing" process. We will also look at some models and algorithms that have been used for other tasks that are applicable to robots learning from experience.

In order to demonstrate the capability of robots to acquire models from manipulation data, we first need to implement a pipeline capable of collecting the data. To do this, we use a UR-10 robotic arm mounted upside down in a 2m x 2m x 1.5m frame. Attached to this arm is a Robotiq 2 Finger Gripper which is used as the end effector of the robot arm. In addition, the frame has 4 Microsoft Kinects, 1 in each corner, which are used as the vision system to detect and identify the objects being manipulated. In Chapter 3, we discuss the methodology behind setting up the software pipeline for the robot's motion, vision, and data recording, the mathematics behind the robot's motion, and the choices that we make regarding types of data collected.

Once we collect a sufficiently large and varied corpus of object manipulation data, we use this data to train, test, and improve a predictive linear regression model that could then predict the effect of a robot's action on an object. The first part of Chapter 4 describes the various models that we try, how they are implemented, and the type of models that perform the best and will be used moving forward. Finally, as we will talk about in the later half of Chapter 4, the model that we define is used in order to close the loop. This allows the robot to manipulate the object in a desired way by using the inverted model to provide the action or actions that need to be performed for the object to move in such a fashion.

In chapter 5, we present an evaluation of the model. We evaluate by comparing the accuracy of our model against that of the baselines, showing that our proposed data-driven approach achieves 81% higher accuracy than the linear extrapolation baseline for one dataset, and 67% higher accuracy for another. We will analyze the quality of our experiments, as well as problems that were encountered and how those problems were avoided or fixed. In addition, we will look at further experiments that can be done and how this project could have been made more interesting or extended given additional time, and further studies that seem like logical follow-ups to robots learning from experience.

Chapter 2

Related Works

2.1 Human Object Manipulation Model Learning

One of the most basic goals of robotics is to create machines that are able to emulate human actions. An assembly line robot replaces a factory worker and a self-driving car replaces a human taxi driver. These robots are less injury prone and more efficient than humans, they do not need breaks and can work 24 hours a day. Yet the standard to which most robots are compared is human performance, and building machines that think and perform like humans is often the overarching goal [6]. Therefore, in order to build a robot that learns, it is necessary to look at how human beings learn. Specifically for understanding the physical properties of objects and the environment, there are two primary theories. One explanation involves humans inherently having internal models for dynamic scene understanding, which they use to make decisions based on simulation. Studies have shown how people have general intuitive understandings of physics [7, 8] and how even early infants can understand physical situations such as occlusion and containment [9]. The other explanation of human physical scene understanding is data driven, where repeated interactions with objects over time will create and refine an internal model. This idea has been shown in a paper looking at babies learning to grasp [10], as well as a general study of infant intuitive learning [5]. Most studies admit that human beings actually use some combination of the two theories, using the data acquired from interactions to improve and refine the innate model used for simulation. For now, most studies approach robotic learning from either one approach or the other, including this study.

2.2 Simulation-Based Predictive Physics Models

This study examines a data driven approach to robot learning, emulating one of the ways in which human being are thought to learn about their environment. Research theories now show that human beings, in fact, use data to augment and refine an internal base model. The underlying model works almost like a video game's physics engine. Given a dynamic scene situation, the internal engine is capable of running simulations in the person's mind and making a subconscious statistical decision about the situation. Similarly, by using a game engine to run simulations, a robot can also be given an understanding similar to that of a human. One such engine, called "Galileo", gave a system the ability to predict the outcome of various dynamic scenes, such as an object rolling down a slope [4]. The system could predict the resulting state of the scene with accuracy comparable to human beings. In addition, the study found that the mistakes and errors made by the simulation were parallel to those made by actual human beings. One could argue this as a positive, since performance is similar to a humans, or a negative, since this system arguably does not improve upon human performance.

A similar but more specific study looked at using a physics simulation engine to make predictions about a billiards game [11]. By running enough simulations, the system was able to make predictions about the final positions of each ball given the initial state and the forces applied. Several other such experiments used the simulations and physics engine to predict the results of a single image of a dynamic scene. In one case, the engine is used to decide if objects in a real world image are stable or will fall [12]. The applications of a study like this would allow a robot to catch falling objects, or even just provide a warning so that a human will catch the object or prevent the fall in the first place. In another, more theoretical study, stacks of wooden blocks were defined in simulation with the goal of predicting stability [13]. Humans often use intuitive belief to predict the stability of a situation, such as stacking dishes or playing Jenga. The last two studies tried to replicate this idea with images and simulations. Each of these simulation-based studies were able to build systems that made predictions equal to or more accurate than human predictions, with types of errors similar to humans' errors as well.

There are also downsides to simulation based systems such as the ones above. One of the primary criticisms is the difficulty in using physics engines. While smaller, more user-friendly engines do exist, they are usually narrow in focus and specifically created to simulate one kind of task. The more complex and general physics engines often re-

quire many hours just to create a few seconds of a simulation for the robot to make predictions with. In addition, simulation data often does not translate perfectly to the real world, as the engine sometimes must oversimplify properties in order to make calculations faster [14]. For example, while in a simulation, a box's mass might be spread evenly throughout the area, the actual box might not be perfectly balanced that way. This could affect the accuracy of a robot's manipulation of that box. This implies there is a limit to accuracy that can be reached through simulation based methods, however, there are studies looking at ways to surpass these limitations [15]. In comparison, the data-driven approach does not suffer from these limitations, although it suffers from downsides of its own.

2.3 Robot Model Acquisition through Observation

There are two ways in which human beings can collect data about a specific task. Either they learn by watching someone else perform the task, or they learn by trying the task themselves. Learning by observation is translated to machines by having it learn from images or videos. For example, by providing the machine with simulation videos of wooden block stacks, which collapsed or did not. From this data, the machine can predict the stability of new simulation and could even translate this predictive ability to real world videos of wooden block towers [16]. In a more similar study to this one, they examined the ability of a machine to learn the effects of forces. Given a single image and a force to be applied, such as pushing a cup off a table, they were able to have their system predict the trajectory of the cup, reaching the end of the table and falling off [17]. These systems learn their predictions from passive image data and are purely sensory machines with no actions performed in response. In our study, we wish to actively collect the data that we will use and train the predictive model, as well as use that model to actually perform desired movements.

2.4 Robot Model Acquisition through Interaction

Several studies serve as the direct inspiration for this study, most of which look at some aspect of "learning by doing". The first uses repeated manipulation data as a replacement for a vision system. This robot tries to learn the visual representations of an object by poking, pushing, and picking the object [18]. Perhaps the most similar to ours is a study in wish they build a model to predict the effects of a poke on an object. They train this model by using data collected from randomly poking the object [3]. This study greatly resembles ours, however, we wish to use pushes rather than pokes, as well as wish to close the loop and be able to use the model to push an object to a desired position. Several other experiments build upon the poking robot, with one using data from multiple types of actions, pushing and grasping, to train both actions. This allows them to use more data as they do not need to separate the corpus into action based subsections [19]. Another study improved upon the data collection process by making it fully automated and allowing the robot to collect grasping data for over 700 hours [20]. Doing this gave them far more data and a far more complex model than we will build here. Each of these papers looked at model acquisition by interacting with their environment, and similar to this study, "Learn by Doing"

Chapter 3

Data Collection

3.1 Preparation

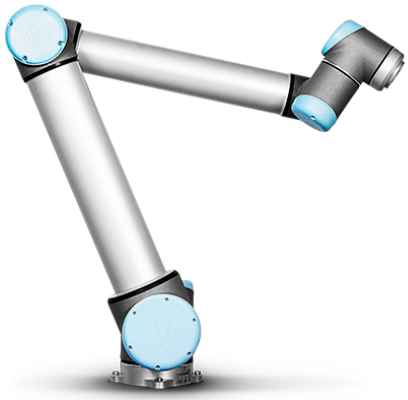
3.1.1 Requirements

We want to build a model capable of predicting the effects of a robot's actions on a given object. Before a data-driven predictive model can be trained, we need data. The data that we need in order to predict an object's state after being manipulated by a robot includes the initial state, the action taken, and the final state. Because we do not have such a corpus of data, it is necessary to create a new data set with the required attributes. Ideally, we need to collect a large and varied corpus of object manipulation data, which will include the initial pose of the object, the final pose of the object, the action performed, the identity of the object, and images from various angles for each trial. To do so, we will use a Universal Robots robotic arm mounted in a work cell, several types of cameras, two different types of surfaces and three different types of objects. Each trial will consist of the arm pushing the given object, giving a corpus of push data with three objects, a blue rubber duck, a red ball, and a yellow lego block.

3.1.2 Materials

To begin our study, we are supplied with a UR-10 robotic arm with 6 joints. The arm is mounted upside-down 1.43 meters above the work surface in a 2 meter by 2 meter frame with a small pillar in each corner. The work surface is composed of 2 halves, each of which can be reversed to alternate between a wooden surface and a rubbery surface. While we initially thought that the arm was mounted in the center, the arm is slightly offset and actually mounted 6 centimeters more towards one side in the Y

direction. The arm is connected to a computer and a control pad used to power on the robot, move it around manually, and even create simple programs in a built in programming language. This tablet also includes a bright red emergency stop button, which is kept close at hand during all data collection and trials in case of a dangerous situation.



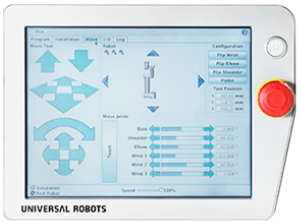


Figure 1 and 2: UR-10 arm and Control Pad with emergency stop

Attached to the end of the UR-10 robot arm is a Robotiq 2 finger gripper, which has a full range of motion between open and closed. This gripper is also controlled by the UR-10 control pad. Mounted from each of the four corner pillars of the frame is a Microsoft Kinect angled towards the work surface. Each is mounted about 103 cm above the work surface and angled towards the center of the table. These are connected to a computer with a specifically ordered USB card that allows it to connect with 4 kinects at a time, as only one can be connected to a motherboard at a time otherwise.

Figure 3 and 4: Robotiq 2 Finger Gripper and Microsoft Kinect

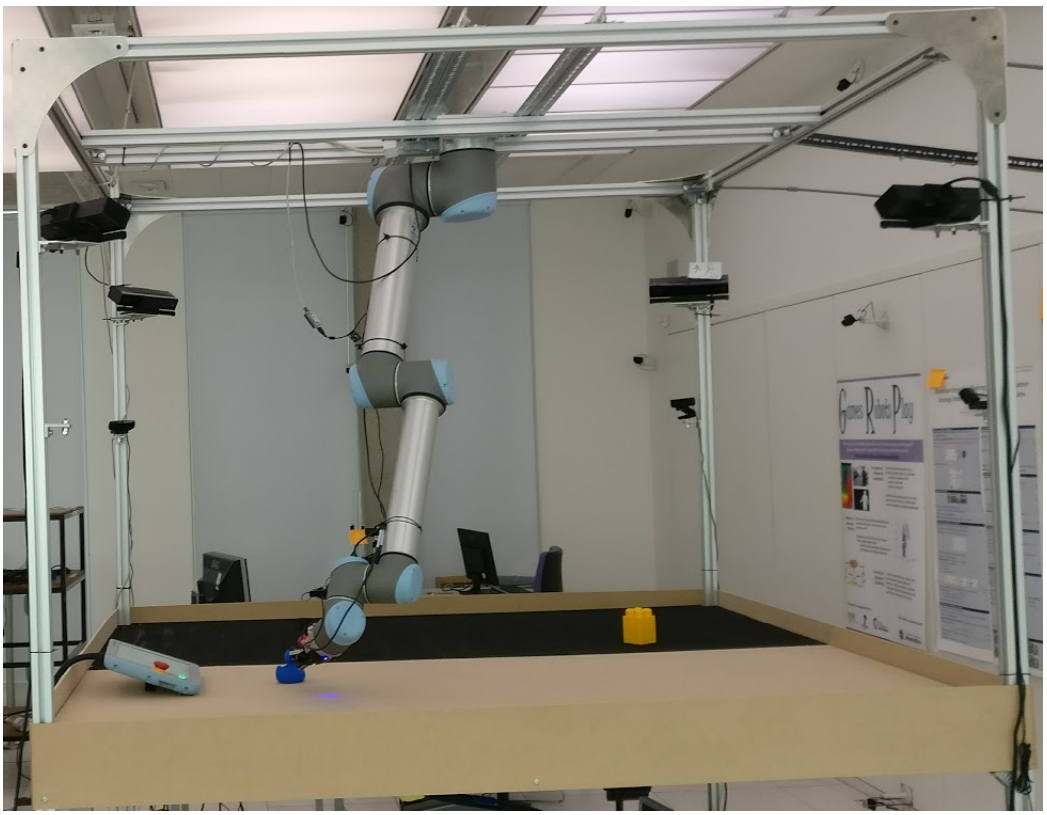


Figure 5: UR-10 Workspace and Frame Setup

3.1.3 Necessary Tasks

In order to begin collecting data with the UR-10 arm, we need to be able to push an object at its initial position and record all attributes required while doing so. The arm needs to be able to push at a given position, and the cameras need to be able to provide the present position of the object so that it can be pushed and recorded. Given a set of coordinates, the system is able to calculate the joint angles required to make the end effector reach that position, as well as calculate the most efficient path to those joint angles. Given that we now have a system that could push an object at a specified location, we need to be able to find the object's location using computer vision. Using point clouds from the kinects, the vision system must provide the present position of the object specified. We also require software to record from both the wrist camera and side camera for each trial. Lastly, we combine each of these subtasks into a system that could find the location of the object, push it, and record the entire push.

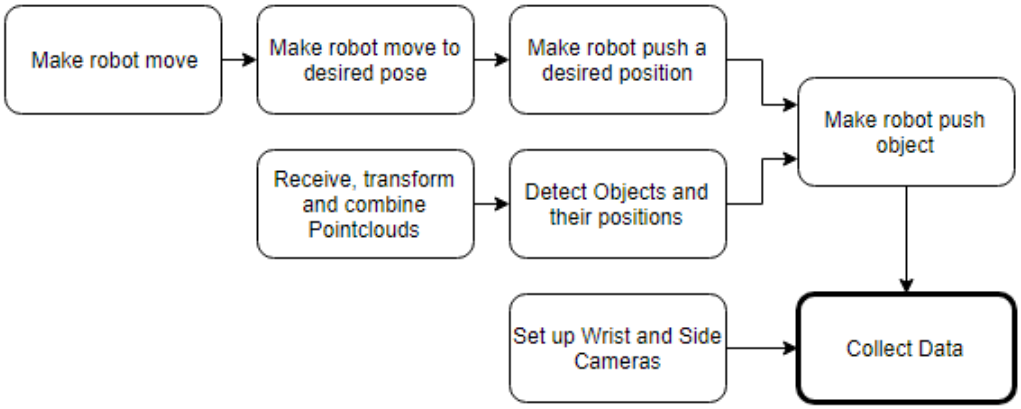


Figure 6: Data Collection Sub-Tasks

3.2 Robot Setup

3.2.1 ROS

ROS, or Robot Operating System, is essentially a collaborative library or framework for working with robots. Each sub-system is run by ROS and called an ROS Node, and these Nodes communicate between each other with ROS Topics. The Nodes can publish Messages on these Topics and other Nodes can subscribe to these Messages, taking a responsive action to the Messages it receives. This works well for a system with many subtasks, sensors, actuators, visualizations or even communication between subsystems being run on multiple different computers. ROS works with C++ and Python files, and has many tools and pre-built, downloadable packages that we worked with in this study [21].

The first step after becoming familiar with ROS is to get ROS to communicate with the UR-10 arm. For this, we use the ROS package UR Modern Driver, which provides several Nodes necessary for input and output to the robot. The modern driver takes in the robot's IP address and establishes a persistent connection to the robot [22]. The driver starts an ROS Node for the motion controller, which takes in commands and actually actuates them on the robot. The driver comes with 2 available motion controllers, a position based and a velocity based trajectory controller. In addition, the modern driver package also starts a node that consistently publishes the joint states of the robot, including the joint angles, so other nodes know the current state of the robot.

3.2.2 MoveIt!

We now have a framework on top of which we can build our robot control system, ROS. There are Nodes that can communicate with the robot, providing the joint state of the robot and allowing us to send movement commands to the robot through the trajectory controller. Our next task is to use inverse kinematics to find the joint angles required to move to a desired x, y and z coordinate. Fortunately, these calculations are abstracted away for us by an ROS package called MoveIt!, which handles almost every aspect of motion planning internally [23]. MoveIt! adds several more Nodes to our ROS system, primarily the ”move_group” Node, which provides a C++ interface using which we cam plan and execute motions. This Node communicates with the trajectory controller provided by the UR driver to execute these requested actions and uses information provided by the joint state publisher to do so. If the motion is possible, using MoveIt! allows us to easily move the arm to a desired pose without having to implement the kinematics.

In order to do this planning for us, MoveIt! also requires information about the robot for which it is motion planning. This information is read from the ROS Param Server, an ROS tool that stores all the parameters of our system so that they can be read by any Node that requires the information. In this case, MoveIt! reads the robot URDF, SRDF, and any configuration options that are uploaded, such as joint limits, which kinematic solver to use, etc. In our specific case, we decide to limit the joint angles to $[-\pi, \pi]$, since it is commonly reported that MoveIt! sometimes does not work well with the full joint range of $[-2\pi, 2\pi]$ with UR robots [22]. We also use the kinematic solver provided by Universal Robots, the UR Kinematics package.

3.2.3 Robot Description

The URDF or Unified Robot Description Format file for a robot is essentially the way that ROS describes the entire robot in an XML format. The file contains information about each link of the robot, their inertial, kinematic and dynamic properties, and even model files so that they can be visualized in simulation. The file also describes how these links fit together, provides names for each joint, and what type of joint (revolute or fixed) each should be. Universal Robots provides the URDF for the UR-10 arm in a package called UR Description, and Robotiq provides one for the gripper in another package called Robotiq C2 Model Visualization. We created a new URDF which imports both the arm and the gripper, attaches them, as well as offsets the end-

effector of the arm 16.5 cm, the distance from the end of the arm and the tip of the gripper. This URDF file is used by MoveIt! to understand the robot it is planning for and to successfully plan and execute commands.

The MoveIt! package comes with a setup wizard which allows you to create a specific package for your robot based on the robot's URDF file. The wizard helps you to define the end-effector as well as the joints for which you want planning to occur, and generates all the ways that the robot can collide with itself. This information is primarily stored in a generated SRDF or Semantic Robot Description Format. The file provides a list of collisions to ignore, such as when two links are connected, as well as the grouping of the joints, a list of predefined positions, and other useful information not found in the URDF. All three of these, the URDF, the SRDF and configuration files are uploaded to the ROS parameter server and used by the move_group Node and other Nodes.

3.2.4 Robot Visualization

Before testing the ROS setup on the very expensive UR-10, we want to able to plan and execute motion on a simulation of the robot to confirm that the system was working correctly. ROS has another tool, Rviz, a visualization tool that shows sensory information as well as state information about the ROS system. The MoveIt! setup wizard automatically generates a Rviz demo file to work with which has a fake joint controller and a fake joint state publisher that imitates the actual robot. When sending motion planning commands to the robot, the planning or execution instead happens only in Rviz, where the pose can be sanity checked to determine validity. Rviz will also render any collision objects that were added, such as the frame that was defined to avoid collisions, and can also be used to visualize the point cloud data from the kinects.

One of the features of MoveIt! and Rviz is the ability to define collision objects, objects which the robot will avoid when motion planning. The frame in our setup includes the 2m x 2m ceiling and table a distance of 143 cm apart, as well as four 4 cm x 4 cm pillars that were 143 cm in length. Later the workspace was updated to have 5 cm high walls along the perimeter of the table. This frame is defined using MoveIt! and adds to its planning context. In order to guarantee that these collision objects are loaded before any robot action is taken, the Node that defines the collision objects and publishes them is created during the instantiation of the move_group Node. Any Nodes that are created later that attempt to issue commands to the robot or to MoveIt! first

check for the existence of the frame in the planning context to avoid a collision.

3.2.5 Gripper

One remaining motion planning task is left incomplete: opening and closing the gripper using ROS. We found that it is not possible in the current setup to communicate with the gripper using ROS. In order to do so, the gripper needs to be connected directly to a PC with ROS installed, which we cannot do on the UR-10 PC. We also cannot connect the gripper directly to another PC as there are other studies that require the gripper to controllable from the control pad. In addition, it was not necessary for our study to change the gripper position. We are able to keep the gripper consistent and closed, rendering the functionality unnecessary.

3.2.6 Action Server

From an efficiency point of view, it does not make a lot of sense to create a new Node every time the arm needs to be moved or needs to push an object. The startup time it takes to start up the Node and connect to move_group is non-negligible. Instead of starting up a new instance of the user interface Node with differing parameters, we take advantage of the ROS framework and create a persistent Action Server, a Node that connects to MoveIt! and subscribes to Messages sent to the server. Beyond just abstracting away the kinematics, this abstracts away all the move_group interfacing and allows for a simple one line command to move to or push a given position. It also allows for a single node to handle the many action types, taking the type of planning as an argument using the ROS Server Client framework.

To have a server and client communicate in ROS, there must first be a custom Message type implemented so that the server knows what to listen for and the client knows what to send. For the purposes of data collection, only pushing is needed as an action, however, in order to create a more well-rounded interface, the Message provides options for executing a simple move, a push, only planning a move and saving the plan, as well as executing a previously planned and saved plan. The Message also needs to provide the arguments necessary for the action, a pose for a move or a position, direction, and distance for a push. The Client must be made aware of the results of its request, so the Message also includes a response with a boolean success and a string for an error message, if one is needed. The Action Server model simplifies our subsystem so that we can now push an object at a provided position with a single line of

code, given.

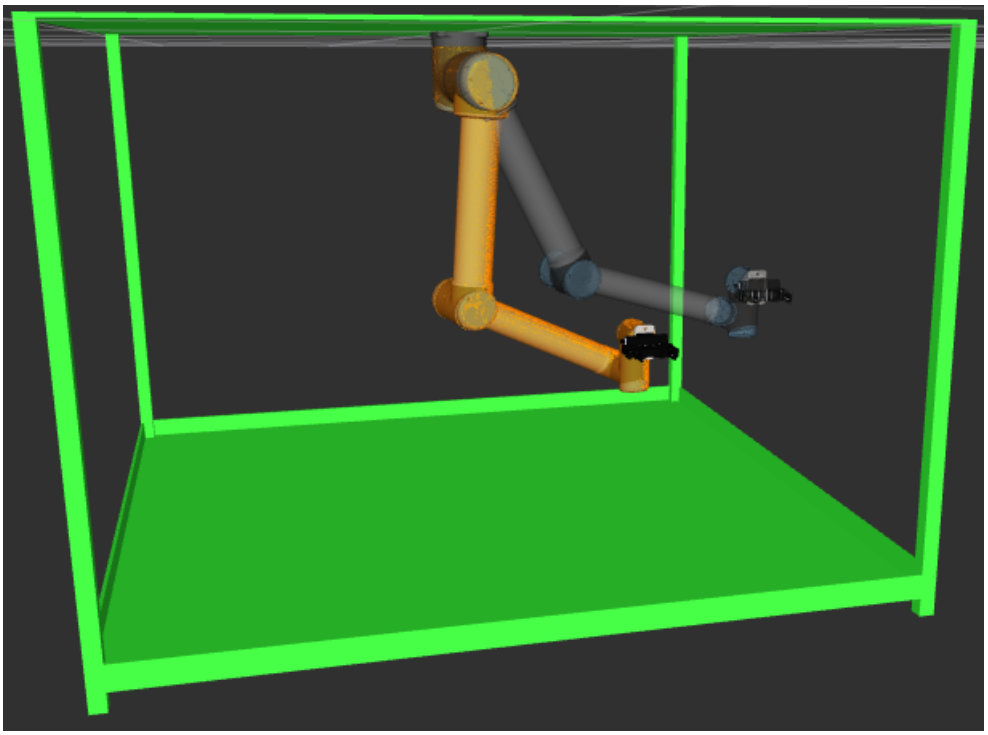


Figure 7: Rviz Visualization of Workspace

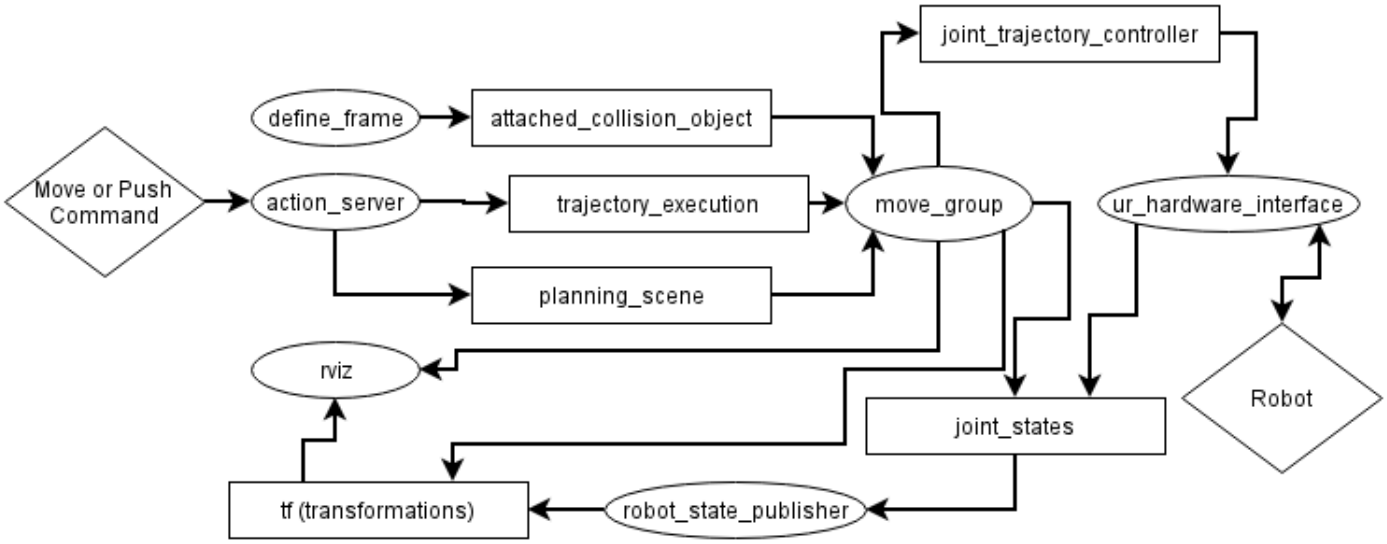


Figure 8: Robot Motion ROS Graph

Ovals are Nodes and Rectangles are Topics

3.3 Motion Planning

3.3.1 Kinematics

The overarching idea behind motion planning is to have the end-effector make a desired movement by moving the joints of the robot. This involves calculating the relationship between the joint angles and the pose of the end-effector. Given that we know the joint angles, we want to know the pose of the end-effector, which can be calculated using a Kinematic Map. Given we change those joint angles, we want to know how the pose will change, which can be calculated using the Jacobian. Using that information, we want to be able to find the change in joint angles needed to make a desired movement.

Each joint of the arm has a displacement and a rotation compared to the previous joint in the chain, the pose of the first joint being relative to the world frame. These relative poses can be described using Transformation Matrices, and these transformation matrices can be multiplied together in a chain in order to describe the overall position of the end-effector relative to the world's coordinate frame. Each displacement or link transform is of the form:

$$T_{i\to j} = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ where d_x, d_y, d_z represent the displacement in each direction

Rotation or joint transforms around the x, y and z axis respectively are given as

$$T^x_{i\to i'}(q) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) & 0 \\ 0 & sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T^y_{j\to j'}(q) = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\theta) & 0 & cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T^z_{k\to k'}(q) = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 & 0 \\ sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ where θ is the angle of rotation

The Kinematic map $\phi : q \to x$, the transformation from the end-effector frame to the world frame, can then be calculated by multiplying the entire chain of transformations.

$$\phi = T_{W\to eff} = T_{W\to 1} * T_{1\to 1'}(q_1) * T_{1'\to 2} * T_{2\to 2'}(q_2) * T_{2'\to 3} * T_{3\to 3'}(q_3) * T_{3'\to eff}$$

The next step is to calculate the results of changing each joint angle, specifically what effect changing the joint angles has on the pose of the end effector. This is defined

by the Jacobian $J(q)$, a set of partial differential equations.

$$J(q) = \frac{\partial \phi(q)}{\partial q}$$

This can be simplified so that we do not need to calculate derivatives. a_i represents the axis of rotation, x, y, or z, and p_i represents the position of the ith joint relative to the world. The Jacobian of position is simplified as:

$$J(q) = \begin{bmatrix} [a_1 \times (p_{eff} - p_1)] \\ [a_2 \times (p_{eff} - p_2)] \\ \dots \\ [a_n \times (p_{eff} - p_n)] \end{bmatrix}^T$$

The Vector Jacobian can also be calculated and used, and will give the change in velocity of the end effector given a change in joint velocities. For the purposes of this study, velocity information was unnecessary, as the goal was always to move to a desired position regardless of speed.

Both the Kinematic Map and the Jacobian make up forward kinematics, situations where the joint parameters are given and we calculate the position and rotation of the end-effector of the arm. What we need in order to move the arm correctly is Inverse Kinematics, which gives us the joint angles needed for a desired end-effector pose. Ideally, since $\delta x = J(q)\delta q$, $\delta q = J(q)^{-1}\delta x$, however, the Jacobian is not necessarily invertible. In addition, there may be multiple paths to the desired pose, and we want to choose the most optimal. We want our path such that at each time step we minimize the cost function including the angular distance from our previous state as well as the distance from our goal position. Under the assumption that the kinematic map is locally linear:

$$f(q_{t+1}) = ||q_{t+1} - q_t||_W^2 + ||\phi(q_{t+1}) - x^*||_C^2 \qquad \delta q = J^{\#}\delta x$$

Where W generalizes the metric to joint space and C acts as a regularizer.

$$J^{\#} = (J^T C J + W)^{-1} J^T C$$

Generally $W = I$ and $C \to \infty$, so $J^{\#} = J^T (J J^T)^{-1}$, the pseudo-inverse, is used. This allows us to calculate the change in joint angles needed in order to reach a desired end-effector pose [24].

3.3.2 Push Implementation

Having established that the Action Server spoke to MoveIt!, which spoke to the UR driver, which in turn spoke to the robot, making the robot and its visualization move is a single line of code. The user provides the X, Y, Z coordinates of the desired position, as well as the rotation in Roll, Pitch, and Yaw format. This establishes a complete path for the user to provide a full pose and has the robot arm plan and execute a trajectory to that pose without colliding with itself, the table, or the frame, and without worrying about the kinematics or connecting to the robot.

In addition to simply moving to a given X, Y and Z coordinate, one of the interactions that we want to have the system perform, and the most essential, is to push an object. In other words, we are given a set of coordinates X, Y, Z where an object is placed, as well as a direction that we want the object pushed. The arm needs to move behind the object and then push it forward in the given direction, ideally maintaining the gripper in a relatively static pose. This essentially simplifies to two motions, the first, to reach the initial pose while avoiding the object, and the second, a simple motion forward to reach a pose on the other side of the object.

The object to be pushed is at a given location (X,Y,Z) and the direction we wish to push the object is an angle θ which is specified to be in radians. The command also takes a distance D as an argument, although for all data collection in this study, that distance is held consistent at $10cm$. After a bit of experimenting, we found that a distance of 10 cm behind the object serves as a good initial pose. To calculate the coordinates that are a distance of 10 cm away from the object at angle θ, (X_0,Y_0), we used the equations:

$$X_0 = X - 0.1 * cos(\theta) \qquad Y_0 = Y - 0.1 * sin(\theta)$$

Where X,Y are the coordinates of the object we are trying to push. In addition, we keep the Z component of the initial position, Z_0, equal to the Z component of the object so that we move in a horizontal line when pushing the object.

We once again define the rotation in terms of Roll, Pitch, and Yaw, RPY, as these are the easiest rotation representations for humans to understand. We initially planned to keep $P = 0.0$, since we want the object to be pushed with the front of the gripper, however, found that the thickness of the wrist prevented the arm from being able to push shorter objects. To compensate and allow the arm to push objects close to the ground, we instead decided to push with the gripper tilted downwards at a 45° angle,

$P = \pi/4$ radians. To make the arm face the object, we set $Y = \theta$, the direction we want to push the object, and lastly, we set $R = \pi$ so that the wrist mounted camera is correctly oriented. These RPY values remain the same throughout the push, for both the initial and final positions.

$$R = \pi \qquad P = \pi/4 \qquad Y = \theta$$

Before we plan or execute this motion, we want to create a constraint so that our motion to get to the initial pose will not collide with the object we later want to push. To do this, we create a collision object covering the object, a hemisphere or dome with a radius of 0.05. While this does not necessarily cover the entire object, it discourages movement around that area. This requires the UR-10 to plan around the object and its dome shield during the initial movement, allowing our push to be the only interaction that we have with that object during that specific trial. Once we reach our initial pose, we can remove the collision object from the planning scene so that we can interact with and push the object during our movement from initial to final pose.

To calculate our final pose, we use equations similar to our initial pose, except instead of subtracting X and Y distances from the objects position, we add them to the objects position. These final coordinates are calculated using the equations:

$$X_f = X + 0.1 * cos(\theta) \qquad Y_f = Y + 0.1 * sin(\theta)$$

3.4 Object Detection

3.4.1 Point Clouds

A point cloud is a representation of a 3-dimensional image as a list of points, each with several characteristics, such as x, y, z position and rgb color. PCL or Point Cloud Library is an independent library for working with point clouds. It provides algorithms for filtering, and processing point clouds as well as performing most computer vision tasks on them[25]. The four Microsoft Kinects that are mounted from each of the four pillars in the workspace frame each provide a point cloud when they interface with ROS at a rate of 5 frames per second. The majority of the work done with the kinects was done prior to this study, so we assume that the Kinect set up is given and we take the point clouds provided to ROS by them at face value. In addition, we also take the calculated transformations for each kinect at face value, transforms that change each point cloud to the same world frame of reference.

We now have 8 new ROS Nodes, 1 for each kinect that provides point clouds, as well as 1 that constantly publishes the transform for each kinect. These point clouds can be viewed in Rviz, which automatically subscribes to the transformation for each and can display all four, which collectively give a full image of the workspace, arm, objects, and a lot of excess area. These kinects do not provide perfect information, and one of the major limitations of the setup is the necessity to recalibrate the kinects very often. In addition, in order to use PCL functions with the provided point clouds, the cloud needs to be changed from the ROS point cloud format to the PCL format, filtered, and then converted back to the ROS point cloud Message. Nonetheless, using these 4 point clouds, we combine all the points into one cloud and find the position of each of the three predefined objects by filtering these points.

3.4.2 Point Filtering

Before starting to filter the points, the four point clouds are combined into one point cloud, otherwise, each would have to be filtered and segmented separately. This is done by creating a new point cloud and adding all points from each of the four original clouds to the new point cloud. In order to simplify the object detection as much as possible, the objects chosen for data collection are all distinct, single color objects. They include a blue rubber duck, a red ball, and a large yellow lego block. In order to segment out the objects, we first filter out all points outside of the area of interest. If the X, Y, Z position of the point is outside of the 2 meter x 2 meter workspace or more than about 10 cm above the table, the point is of no interest to the system and is filtered out. The filtering is done using the simple passthrough filter provided by PCL and returns a greatly simplified point cloud. This filtered point cloud is then published as an ROS Topic so that it can later be recorded as part of our data collection.

For each of our three objects, another passthrough filter, this time with rgb color, is used to remove all points that are not similar to the color of the object in question. For each object we experiment with RGB value filters, examine the resulting set of points using Rviz, and further tweak our filter values, until for each object filter, only points belonging to that object remain. The RGB values that are used to filter each object were:

Object	Red	Green	Blue
Red Ball	$R > 60$	$B < 20$	$G < 75$
Blue Duck	$R < 75$	$B > 75$	$G < 75$
Yellow Lego Block	$R > 100$	$B < 75$	$G > 100$

Figure 9: RGB Value Chart

In order to remove any chance of stray blue, red, or yellow points around the work surface, any points considered outliers from the mean are also removed, which helps to make the only cluster of points for each object validly belong to that object. Finally, for each object, the average position is found by averaging the X values, the Y values and the Z values and combining these 3 values into an average coordinate position. These 3 positions for 3 objects are constantly published by the point cloud filtering Nodes that we created so that at any given moment, any ROS Node can subscribe to the position of any of the defined objects, even all 3 at the same time.

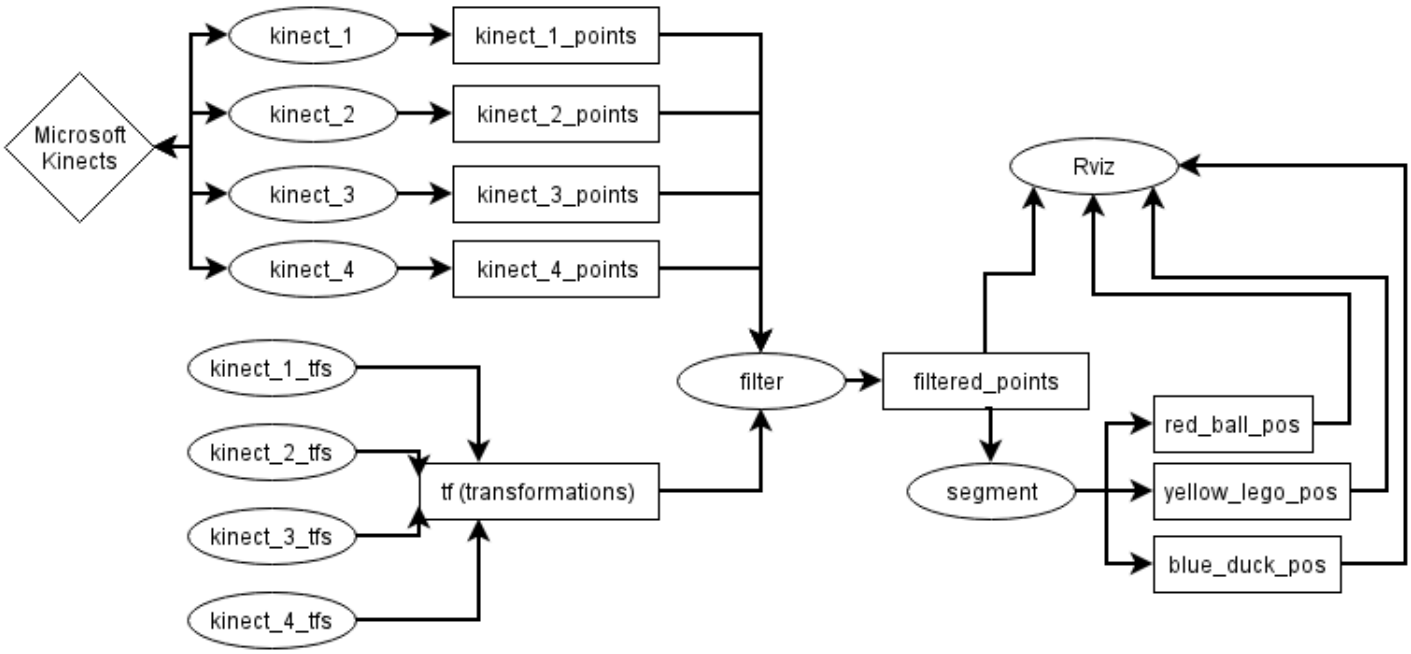


Figure 10: Robot Vision ROS Graph

Ovals are Nodes, Rectangles are Topics

3.5 Collecting Data

At last, with a vision system providing the coordinates of each object and a motion planning system capable of pushing an object at provided coordinates, the full data collection system is almost complete. We can now create a final Node to run a trial, which would take the object we wish to push as an argument, either the rubber duck, the red ball, or the yellow block. The Node will then choose a direction randomly to push the object, subscribe to the Topic publishing the objects position, and command the arm to push the object in the randomly chosen direction. Before the trials can begin, there are several choices that needed to be made.

Firstly, there are two possible trajectory controllers that could be used. The position based controller is implemented as more accurate, while the velocity based controller reacts more quickly. After experimenting with both, it was found that the velocity based controller caused the arm to vibrate and oscillate slightly, in a very undesirable way. Because of this, we chose to stick with the more accurate position based controller. Next, the distance to push the object each time needs to be constant, so we determined 10 cm to be an ideal distance to push. The trial Node now passes a full X, Y, Z, θ, D list of arguments to the push action server Node.

In addition to the 4 kinects, we also decided that storing some ordinary camera data for each trial might be helpful in future experiments. The robot already has a camera mounted on its wrist, and an additional camera is placed on a tripod to capture a side view of the workspace. Both cameras are interfaced with using an ROS package, usbcam, which starts up the cameras and publishes their raw image data on ROS Topics [21]. The filtered point cloud data from the 4 kinects, which include all points within the 2m x 2m x 10cm area of interest, are also published on an ROS Topic.

In order to record each trial, another ROS tool, rosbag, is used to record the published data of chosen Topics over a period of time [21]. Each rosbag stores the data from one single push trial. The Topics that are recorded during each trial include the image data from both cameras, the filtered point cloud, the detected positions of each object, the transform data for the kinects (necessary if we later want to visualize the point cloud in Rviz), and the direction of the push. The rosbag recording is started and stopped independently of each push, as the length we wish to record might vary depending on planning time, movement time, and for some objects, roll time. Each rosbag lasts between 5 and 10 seconds and is usually about one gigabyte in size.

For each object, the red ball, blue rubber duck, and yellow lego block, 50 push

data points are collected for each of the two types of surfaces. After these solo data points, multi object data are attempted, and we collect 100 more data points where we push the blue duck off of the yellow lego block. For the purposes of this study moving forward, we use the yellow lego block data on the wooden surface to build a predictive model as well as invert this model for object motion path planning. Direct pushing data is a good starting basis for this study as it is the most simplistic direct arm to object interaction possible. From here we can move to more complex interactions, and the data collection system that we have built is easily able to transition to any multi object interactions and is easily adapted to work for other action beyond pushing. The multiple surfaces provide breadth to our data, giving us more options for future studies without necessarily needing to change the data collection portion of the system.

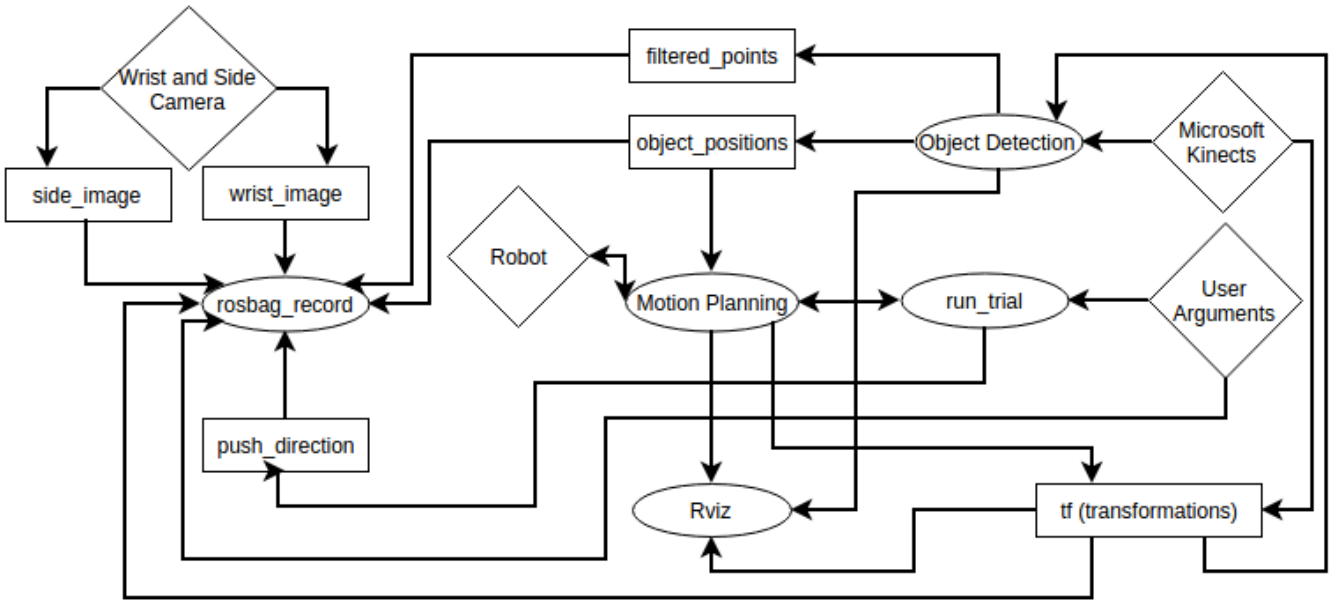


Figure 11: Data Collection ROS Graph

Ovals are Nodes, Rectangles are Topics

Chapter 4

Learning from Data

4.1 Predictive Model

4.1.1 Linear Regression

Linear Regression is one of the most basic forms of data fitting. The basic idea is to find the parameters w and b of the function

$$f(x;w,b) = w^T x + b$$

that maximize the accuracy of the model [26]. x is an $N \times D$ matrix with N data points which each have D independent variables, and $f(x;w,b)$ is the N length column vector of predictions. w and b can be combined into a single vector to solve for by making $b = w_0$ and adding a feature to x which was always 1. Solving for the w that provides the highest accuracy is a matter of minimizing the error function

$$E(w) = \sum_{n=1}^{N} [y^{(n)} - f(x^{(n)};w)]^2 = (y-f)^T (y-f)$$

where f is the vector of predictions and y is the vector of actual values [26]. Minimizing the squared difference between the predicted and actual values on the training data makes the resulting w, or weights the best parameters for this model. Finding this ideal solution is trivial in MATLAB or using Python libraries SciPy and Numpy.

In the particular case of this study, there are 3 independent variables or features: the initial X coordinate X_0, the initial Y coordinate Y_0, and the push direction θ. There are also 2 dependent variables or predictions, the final X coordinate X_f and the final Y coordinate Y_f. These 5 variables need to be extracted from each rosbag created during data collection. For each rosbag, the data that is needed for this particular study

includes only the push direction and the position of the specific object for which the model is being built. The push direction Message is only published once per rosbag, so is very straight forward to extract. For the object position, say the yellow lego block, the Topic extracted from the rosbag will provide a vector of Messages in chronological order. By saving the first and last Messages in the vector, we store the initial and final position of the yellow lego block.

4.1.2 Basis Functions

There are often cases where some function of a feature would make a useful additional feature for the model. For instance, x^2 might be more useful as a feature than x, or the absolute value of x might give a better model than x alone. The features can also be multiplications or divisions of several different features. For all such cases, the new dependent feature is a Basis Function [26]. By applying it to every data point and adding it as a new feature, the same basic linear regression solver can be used. Given that the initial input data is X, the new input matrix might look like Φ.

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} \\ \dots & \dots & \dots \\ 1 & x_1^{(N)} & x_2^{(N)} \end{bmatrix} \rightarrow \Phi = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & (x_1^{(1)})^2 & (x_2^{(1)})^2 & x_1^{(1)} * x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & (x_1^{(2)})^2 & (x_2^{(2)})^2 & x_1^{(2)} * x_2^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} & (x_1^{(3)})^2 & (x_2^{(3)})^2 & x_1^{(3)} * x_2^{(3)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_1^{(N)} & x_2^{(N)} & (x_1^{(N)})^2 & (x_2^{(N)})^2 & x_1^{(N)} * x_2^{(N)} \end{bmatrix}$$

4.1.3 Choosing a Model

For the predictions needed for this study, two models are actually necessary, one each for X_f and Y_f. Since they solve similar problems, we make the assumption that both dependent variables use the same type of model, with the same basis functions, just different weights. Because linear regression does not have any hyperparameters, and because the data size is so small, there is no validation set. The data is split 80:20 into training and testing data and the performance on both sets is measured and reported below to compare the various models. Training set accuracy is more important in this case, as we did not want to overfit to the very small testing set. A linear regression model with no basis functions is the initial attempt to fit the data, and acts as a baseline for all other models and basis functions. The first attempt to improve upon this took advantage of the fact that the push direction θ is an angle and the sin and cosine of

this angle provide an indication of whether the push is more in the X direction or the Y direction. From here, different combinations of the various features are compared with, including

$$\Phi_0 = \begin{bmatrix} 1 & X_0 & Y_0 & \theta \end{bmatrix} \qquad \Phi_1 = \begin{bmatrix} 1 & X_0 & Y_0 & \theta & cos(\theta) & sin(\theta) \end{bmatrix}$$

$$\Phi_2 = \begin{bmatrix} 1 & X_0 & Y_0 & \theta & cos^2(\theta) & sin^2(\theta) \end{bmatrix} \qquad \Phi_3 = \begin{bmatrix} 1 & X_0 & Y_0 & \theta & X_0 cos(\theta) & Y_0 sin(\theta) \end{bmatrix}$$

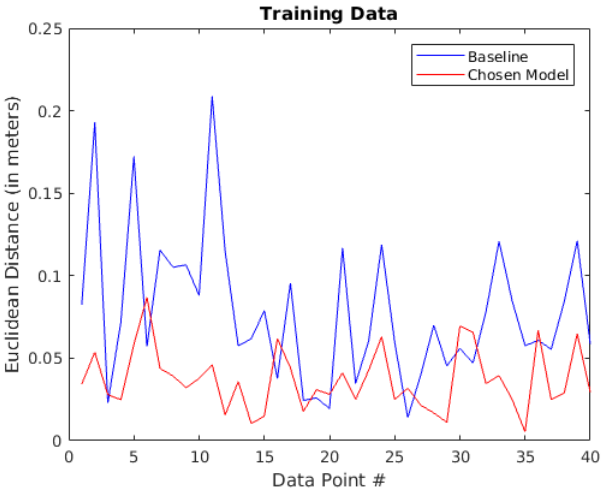
	Φ_0	Φ_1	Φ_2	Φ_3
X Model	0.0805	0.0331	0.0772	0.07
Y Model	0.0495	0.0302	0.0486	0.0367
Combined	0.0895	0.0453	0.0840	0.0643

Figure 12: RMS Error of Training Data

	Φ_0	Φ_1	Φ_2	Φ_3
X Model	0.0717	0.0602	0.0724	0.0588
Y Model	0.0304	0.0255	0.0379	0.0272
Combined	0.0851	0.0522	0.0961	0.0721

Figure 13: RMS Error of Test Data

Based on these results, while Φ_1 and Φ_3 provided similar results on the testing data, we chose to continue with Φ_1 as it performed far better on the training data, as well as being the slightly more simple model. The benefits of using the basis function model over the baseline were evident regardless, especially in the training data. The combined models are compared by looking at the euclidean distance between the prediction point and the actual point.



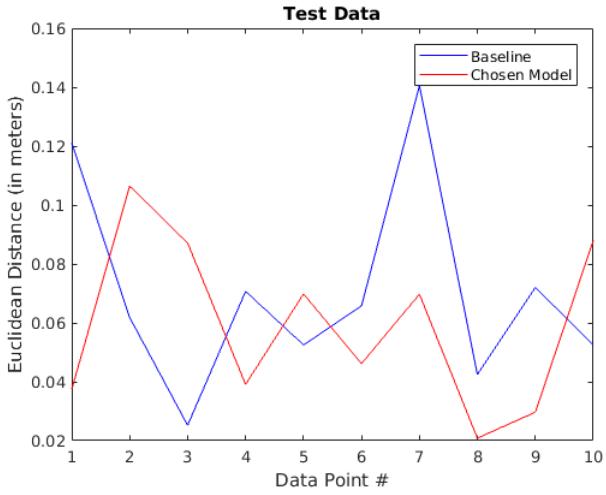


Figure 14 and 15: Euclidean Distance between predicted and actual final position

Yellow Lego Block on Wooden Surface

4.2 Finding an Optimal Path

4.2.1 Systems of Equations

Linear Regression is the process of finding the weights of each feature that maximize the accuracy of the model. Given that those weights are found, the weight computation is with respect a linear model structure of the form

$$y = w_0 + w_1 x_1 + w_2 x_2 ...$$

where x is the feature vector, w is the weight vector, and y is the dependent linear combination. If there are multiple such linear or even non-linear equations, they together form a system of equations. There are three possible situations given a system of equations. If there are less independent variables than equations, the system will have infinite solutions, and if there are more independent variables than equations, there will be no solution. However, if the number of equations is the same as the number of independent unknown variables in the system, the system has one solution. This solution can be solved for using various methods, such as substitution or matrix manipulation[27].

The model chosen above provides 2 equations, one each for the X_f and Y_f coordinates. While the data includes 5 features, the later 2 are dependent on the push direction as they are basis functions. Both of these provide an additional equation to the system of equations.

$$X_f = w_0 + w_1 X_0 + w_2 Y_0 + w_3 \theta + w_4 u + w_5 z \qquad u = cos(\theta)$$

$$Y_f = v_0 + v_1 X_0 + v_2 Y_0 + v_3 \theta + v_4 u + w_5 z \qquad z = sin(\theta)$$

Given the final coordinates or goal position (X_f, Y_f), as well as the weight vectors w and v, this system of equations has infinite solutions, as there are 5 unknowns and only 4 equations. The solution we used for this is to discretize the domain of one of the unknown variables into N possible values and find the N solutions to the system. In this case, we will use the push direction θ and test 32 values split evenly between 0 and 2π.

4.2.2 Routing Algorithms

Suppose there is an object at a given position (X_0, Y_0) which needs to be moved to another position (X_f, Y_f). If the above system of equations is solved, the 32 solutions

each represent a position from which the object can be pushed and reach (X_f, Y_f). Each of these 32 positions has 32 of its own positions from which it can be reached, and if we continue like this, a graph can be created with infinite nodes where one of the nodes will happen to be (X_0, Y_0) (within a set tolerance range). This provides a series of pushes that would ideally move the object from (X_0, Y_0) to (X_f, Y_f). The goal of a routing algorithm would then be to find the path within the graph that would move the object in the fewest number of pushes.

There are several types of algorithms that can be used for finding the shortest path. A DFS, or depth first search, will follow a path until it reaches a terminal node before moving back up to the previous node. DFS was a bad choice for this case, as the number of nodes and the path lengths were not limited and DFS could have run forever [28]. BFS, or breadth first search, will investigate all the children of the start node before following a path any deeper. BFS is guaranteed to find the shortest path possible, but will search in an uninformed way, and is therefore slow. BFS looks at every single node, even if it looks as if the object is being pushed away from the goal [28].

A* search algorithm takes into account whether each movement is a step away from or towards the goal. It searches nodes in increasing order of cost, where cost is determined by

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the actual number of pushes it would take to get from the current node to (X_f, Y_f) and $h(n)$ is an estimate of the number of pushes it would take to get from (X_0, Y_0) to the current node. $h(n)$ is called the heuristic, and choosing a good heuristic to estimate the remaining cost to the goal node is key in making A* work well [29]. It is a matter of choice whether to start searching at (X_f, Y_f) and work backwards or start at (X_0, Y_0) and work forwards. In our case, the algorithm starts at (X_f, Y_f) simply because that is how the idea was incepted.

4.2.3 Full A to B object movement

Data collection has been implemented, data has been collected, a model has been trained using that data, and now that model can be inverted and used to find the best series of actions to move an object to a given position. We solve the systems of equations that inverting the model with different variables gave and use A* to find the shortest path through these solutions from (X_0, Y_0) to (X_f, Y_f). This provides a sequence of push

directions to perform in order to move the object to the goal position. If the model was very accurate, then simply following this series of pushes would get the object to the goal position.

In order to use A* to find the shortest path, the algorithm needs a defined heuristic, an estimate of how close to the target the current node is. Since our cost for each push is a consistent 1, as we are trying to minimize the number of pushes, the estimator needs to also estimate number of pushes. It is also better for the heuristic to underestimate the number of pushes than over, since if $h(n)$ overestimates the cost, A* will not always find the shortest path [29]. Since each push is 10 cm, we defined the heuristic to be the euclidean distance to the goal node, divided by 10 cm. This provided an approximation of the number of pushes still required to reach the goal, and could never overestimate.

$$f(n) = g(n) + \frac{\sqrt{(X_n - X_0)^2 + (X_n - X_0)^2}}{10cm}$$

The data trained model that is used, while more accurate than the baseline, is not so accurate that it can be used for path planning without adjusting as the object moves. Instead of simply following the calculated path, each time the path is calculated, the first action in the list is performed, and then the path is recalculated so that any errors in the last action or inaccuracies in the model can be corrected for. At each step, the robot calculates the first move in the optimal path, performs it, and then repeats this process until the object is within range of the goal position. This tolerance range was experimentally determined, with the aim being to keep the range as low as possible to allow for object movements as accurate as possible. This implementation allows the robot to use the data it has collected and built a model with to come full circle and push objects as desired.

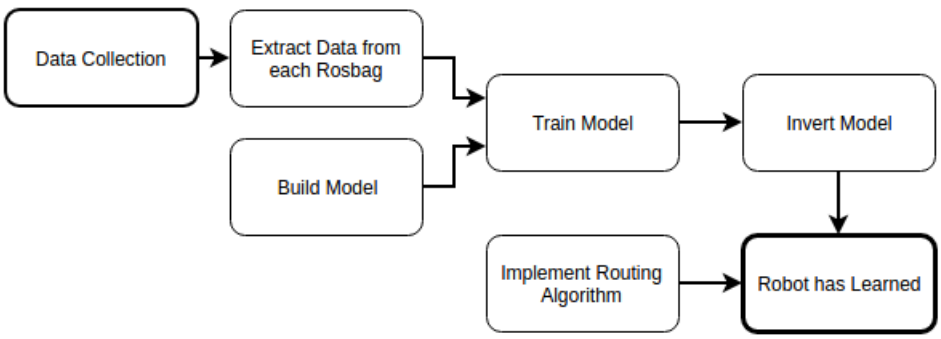


Figure 16: Path Finding Sub-Tasks

Chapter 5

Critical Analysis

5.1 Data Collection

5.1.1 Results

There is not a clearly defined metric to evaluate the performance of the data collection portion of this study. Subjectively, the data collection presented many challenges and decisions that needed to be made, however, the quality of the data collected seems to be a decent representation of the physical properties of the system. As discussed above, we chose to collect data for a ball, a rubber duck, and a lego block. Each of these was primarily chosen due to it bright coloring, as vision and object segmentation was not intended to be a large or difficult portion of this study, and color segmentation is a relatively easily implemented filter. For each of the 3 objects and each of the 2 types of surfaces (wood and rubber), 50 data points were collected. In addition, several data points were collected for multi object interaction, such as pushing the duck off of the lego block. While these other interactions were not the focus of the later path-finding or predictive model portion of the study, it would not be difficult to change the predictive model to be based upon more complex interactions.

5.1.2 Sources of Uncertainty

While it is difficult to judge the quality of collected data independent of the model it trains, it is easy to identify shortcomings, issues, and possible improvements in the data collection process. To begin with, the motion planning system is not flawless. The area where the arm can successfully push objects is limited, and many times the arm fails to successfully push an object. These types of failures, where the arm does

move at all, are not counted as data points and are not recorded. In addition, if the arm successfully moves to its initial position and then fails to find a valid motion plan for the actual push, the data point is also discarded or not recorded. The initial movement of the arm is highly unpredictable, as sometimes reaching even a very similar position requires a readjustment of all six joints. Furthermore, if a joint needs to be below π for a start position and needs to change to close to above π, the joint limits imposed would require the joint to turn almost 360 degrees to reach an angle that should theoretically be very close by. Thus the path taken to realize the initial position of the push is of no consequence to this study and so only the push itself is recorded.

There were some types of failures that are in fact valuable data and not failures at all. In these situations, the object is not moved or pushed in an expected way, but the behavior is still interesting. In the case of the yellow lego block, if the detected position is slightly offset, the block will be pushed off center and might slip to the side of the gripper rather than be pushed forward. The push can also have been too high and just knocked the block over, or the block's friction can have caused enough force that the arm gives up on pushing it. If the arm's position deviates too far from where it is expected to be, i.e. if it encounters too much resistance, it will emergency stop itself, as the controller believes the arm has hit something. This situation is common with the red ball and yellow lego, however, the blue duck is squishy and provides some valid data where the duck simply gets trapped under the gripper.

Many of the above situations, while valid data, are unexpected and nonideal behaviors. These behaviors are all primarily caused by either inaccuracy in the vision system or in the motion planning system. In the motion planning system, many of the inaccuracies have to do with the joint limits or inability to find a motion plan as discussed above. In addition to these, there are several other motion related failures or inaccuracies that contribute. When building the collision frame for the robot, the ceiling, table, walls, and pillars are all accurately defined, however, there are some details unaccounted for. Each pillar has mounted cameras, as does the wrist of the robot, and the wires for both the camera and the gripper travel up the arm. All of these are not modeled as collision objects but remain well withing the range of the arm, and so are at risk of collision with the arm. It is highly unlikely that the arm will plan near the pillar mounted cameras, however on several occasions, the emergency stop was used to prevent a collision. Emergency stopping was also used a few times when wires became caught or if the wrist mounted camera collided with a part of the arm, as the robot is not aware of the possible self-collision.

When it comes to the vision portion of the data collection system, the majority of the implementation is provided for us. Like the motion planning system, there are shortcomings in using the kinects. While combining the 4 kinects gives a full and clear representation of the workspace, the kinects have to be calibrated very well in order for each of them to provide matching point clouds. This is often not the case, as small movements caused by things such as someone bumping a kinect or vibrations caused by the robot moving around, could cause the kinects to no longer match up perfectly. When looking at the point clouds, one can at times clearly see that there is a discrepancy in the detected object positions. While often averaging the points still gives a reasonable position, it is not always the center of the object, and so the behavior of the push is not always as anticipated.

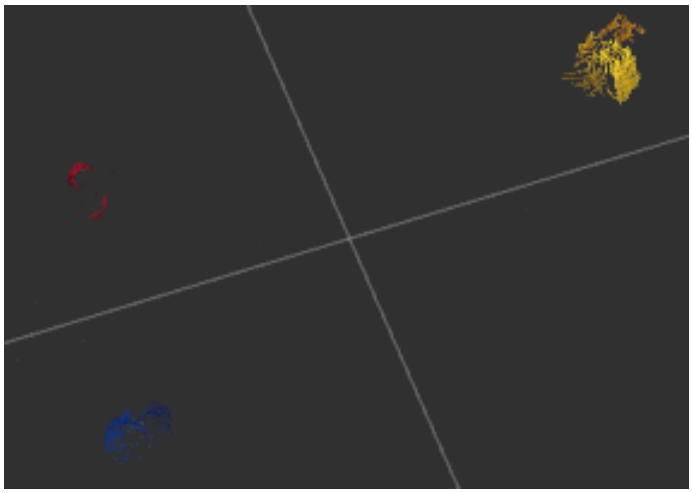


Figure 17: Objects segmented from Point Clouds

We can see that the kinects do not line up perfectly

5.1.3 Possible Improvements

Given more time or another study, there are improvements that can be made to the design of the data collection system, as well as fixes for the sources of uncertainty and failure. At the moment, every time the 4 kinects need recalibration, it is a manual process that can take some time, and only a few people, primarily uninvolved with this study, know how to perform the recalibration. Using a piece of wood with various symbols on it, each kinect has a transformation calculated to change all four kinects to the same frame of reference. In the future, our goal is to automate the recalibration process, creating a script that will perform the calculations and find the transformations for each kinect quickly and autonomously so that the points all correspond to each other

correctly. It would be as simple as placing the calibration wood panel on the table and running the script. By making the kinect calibration far easier and convenient, it becomes more reasonable to just calibrate whenever the kinects are even slightly out of sync. This would repair a myriad of issues and would make the data collected much more consistent. Instead of the push location on the object varying, if the kinects all line up, the center of the object is found far more accurately. Objects are less likely to get stuck, slip to one side, or get knocked over. If these issues still occur, it would not be due to the confounding variable of where on the object the arm pushes.

In addition to improving the vision system, we can also improve our motion planning in order to avoid errors such as hitting the cameras or kinects. These are easily rectified by creating a more accurate collision object frame. Although the collision object creation process is primarily for simple shapes, given enough time, objects such as wrist cameras and pillar cameras could be added to the frame, allowing the robot arm to plan around them and never risk a collision. These would not have to be exact, and if these collision objects took up more space than the actual object, it would do no harm and leave room for slight variations or tilting of the various cameras. Solving a similar problem, there is likely a way to connect the wires of the gripper and wrist camera so that they can never get caught on any portion of the arm. The wiring now is decent, however, there is definitely room for improvement, and improving will further increase both the quality of our data and the ease of data collection.

As a final possible improvement to the data collection design, instead of choosing the push direction at random, we might impose some limitation on the possible directions and choose the direction pseudo-randomly. If the object is close to the upper edge of the work space, the arm should probably not push further up, as the object might become unreachable in the subsequent push. That requires human intervention, moving the object back to a reachable location. If we impose that the object should attempt to remain within the limits of what is easily reachable by the robot, data collection will require less human intervention, and ideally, given a refined enough system, be able to continuously collect data with no required input from the user.

5.2 Predictive Model

5.2.1 Results

The relatively simple linear regression models that were trained to predict the final X and Y coordinates performed well when compared to a baseline random model. The

two models, one for the X and one for the Y, also outperformed a linear extrapolation of the data, a far better baseline. Given that the push distance was held constant at 10 cm, the linear extrapolation baseline simply added 10 cm to the initial position in the push direction. In an ideal situation, if the robot was told to push an object 10 cm at angle θ, the final location would be 10 cm away at angle θ. Therefore this is a good baseline, and out-performing it proved that there is benefit to acquiring a data based model rather than simply extrapolating from the initial coordinates and direction. For analyzing the results of the predictive model, the Yellow Lego data from the wood surface was used.

	Random Baseline	Linear Extrapolation	Data Trained
X Model	0.8892	0.0369	0.0434
Y Model	0.5699	0.1014	0.0337
Combined	0.2936	0.0947	0.0522

Figure 18: Root Mean Squared Error of Various Models (in meters)
Yellow Lego Block on Wood Surface with Testing Data

Overall the combined data trained model performed 81% better than the baseline when the euclidean distance is used as the error metric. One can see that our model for both the X and Y directions highly out performed the random baseline, where for both X and Y, a number was randomly chosen from a uniform distribution between -1 and 1, the dimensions of our workspace. This was a rather low bar, however, the trained Y model also out-performed the linear extrapolation model, a much more reasonable approach and a much higher bar. While the X model did not out-perform linear extrapolation in isolation, the difference was relatively small. When instead taking both models into account and looking at the residual euclidean distance from the predicted to actual final position of the object, the data trained model overall performed better than linear extrapolation on both the training data and the testing data. Figure 19 shows that on the training data, 87.5% of the predictions were more accurate with the data driven model than with linear extrapolation. Figure 20 shows a similar result on the test data, with 70% of the data points favoring the model's predictions.

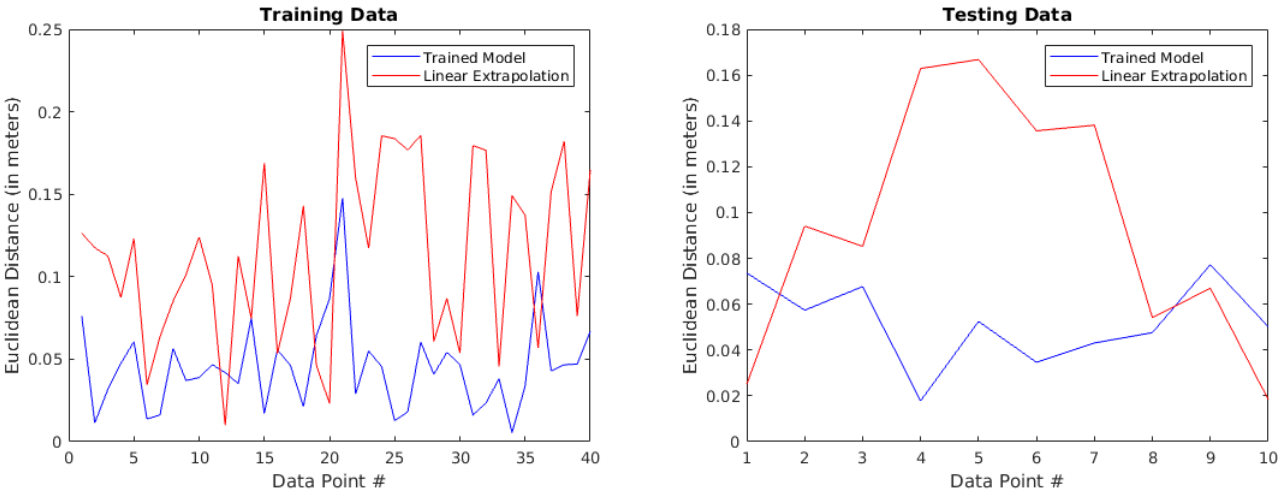


Figure 19 and 20: Distance between predicted and actual position

Yellow Lego Block on Wood Surface

When instead training the model using push data from the blue rubber duck on the rubber surface, the model continued to provide higher prediction accuracy, with the combined model providing 67% higher accuracy than linear extrapolation.

	Random Baseline	Linear Extrapolation	Data Trained
X Model	0.4700	0.0620	0.0602
Y Model	0.6036	0.0874	0.0255
Combined	0.3094	0.0996	0.0594

Figure 21: Root Mean Squared Error of Various Models (in meters)

Blue Rubber Duck on Rubber Surface with Testing Data

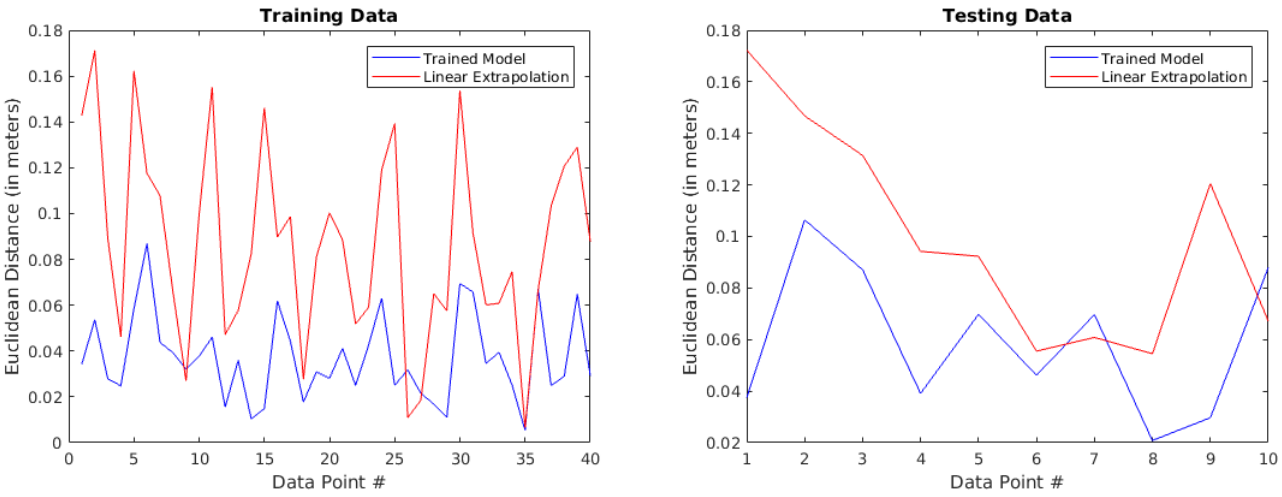


Figure 22 and 23: Distance between predicted and actual position

Blue Rubber Duck on Rubber Surface

5.2.2 Sources of Uncertainty and Improvements

One of the primary reasons that a model as simple as linear regression was chosen as the predictive model is the fact that there is was relatively small amount of data. 50 data points is not a large amount, and left only 40 training points and 10 test points, not a very adequate number to create a fully representative model. Given more time, more data points for each object and surface could greatly improve the accuracy of our predictive model. In addition, all of the sources of uncertainty during data collection carried over and caused uncertainty in the predictive model. By making the improvements discussed on the data collection system, the data collection process can become essentially automated and the quantity of data in addition to the quality will be improved. Both of these would increase the predictive capabilities of a better trained model.

Both the predictive model and linear extrapolation seem to provide far more accurate results on the Y coordinate than the X coordinate. We believe this is primarily due to the locations at which data was collected. While the X coordinates varied a lot and may need more representative data, the Y coordinates were often very similar, as the chosen coordinates are usually those within arm's reach of the computer, in case a human needed to intervene. Again, collecting data with more quality and quantity, especially in an autonomous series such as described above, could provide a better and more accurate predictive model.

Another possible change is to use the data for all surfaces and objects to train one universal predictive model, with categorical features describing which object or surface the data originated from. This would increase the number of features, and by using a more complex model than just regression, allow the model to use all of the data collected as opposed to just a small portion on many smaller models. This could be further extended to using data from other actions, such as pick and place or multi-object actions. Overall, improvement to this model will come from an increase in the data used for training, regardless of the method behind increasing this quantity, as well as a refinement of the data collection process.

5.3 Path Finding

5.3.1 Results

For the most part, since the quality of the data driven model is assessed in isolation, and A* is guaranteed to find the shortest path, the quality of the path finding system was not in question. We do, however, compare the number of pushes actually taken to the number that an object would take if it wad pushed in a direct path along the line formed between (X_0, Y_0) and (X_f, Y_f). We also compare the number of pushes actually taken to the number of pushes predicted during the first iteration of model inversion, as a further indicator of the of the accuracy of the predictive model.

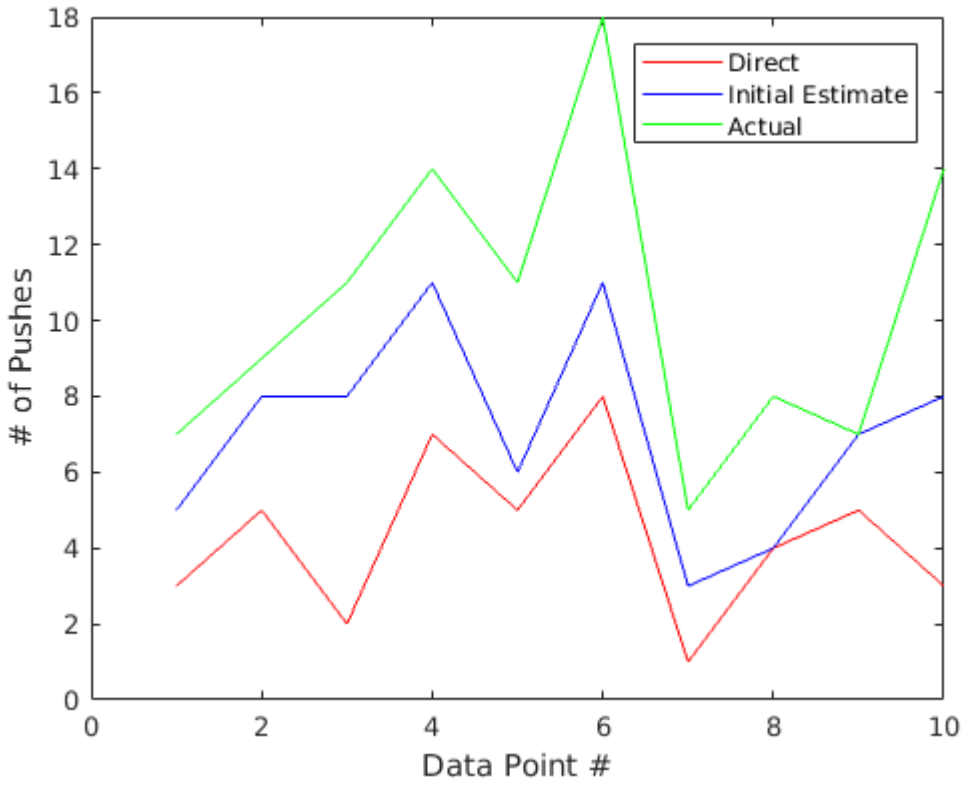


Figure 24: Number of Pushes to move object to (X_f, Y_f)

Yellow Lego Block on Wooden Surface

5.3.2 Sources of Uncertainty and Improvements

As before, all the sources of uncertainty and causes for failure for the previous potions of this study still cause uncertainty and failure here. The path finding portion of the study is dependent on the data collection and the model, so inaccuracies in data collection, lack of data, and motion planning failures all contribute to the inability of the robot to find the best path to move the object in. The motion planning portion in particular came into question again, as the robot is not good at planning near the exact center of the workspace, and for many motions, the most direct and simple path is through the center of the workspace.

As discussed previously, the UR-10 does not have unrestricted motion capabilities, and there are many positions that cannot be reached. There are also many motion plans that cannot be executed by the arm. While the route planning may find potential mathematically efficient routes, the arm is not necessarily able to perform the first action in the list. Because of this, if the first action of the shortest path is not able to be executed, we try to instead follow the next shortest path that did not begin with that same node. This causes fewer failures, but sometimes the object is pushed somewhere where no direction of push can reach it and gets stuck.

As a general rule, we try to limit valid push locations to be within 60 cm of the center of the robot. A more ideal solution would be to identify and preprocess what pushes are possible and what pushes are difficult for the robot. As a step towards this, we ran a simulation and had the simulated robot push in all 32 directions every 5 centimeters and created a heatmap of what positions the robot arm favored and was able to push more easily.

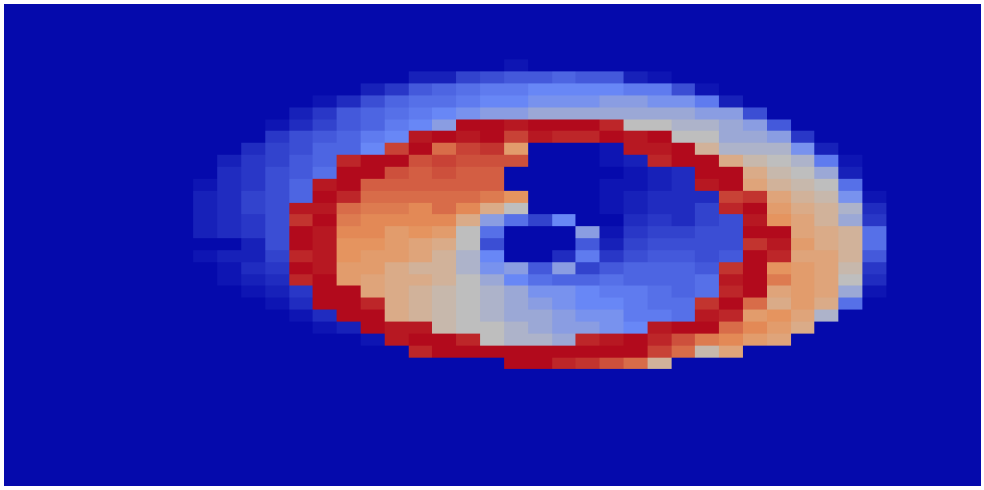


Figure 25: HeatMap of UR10 push capability

5.4 Future Works

There are many other possible studies that stem from these ideas. In this study, we look primarily at the pushing of a single object directly, somewhat like a game of minigolf. To further that analogy, an interesting addition would be to add minigolf-like obstacles to the workspace. In addition to learning the basic physical properties of the ball, the model would also learn what areas to avoid due to obstacles, albeit this would most likely require far more data. Even keeping the surface type inconsistent or adding slopes and valleys would make the terrain interesting to learn. One way to make the robot get better over time in a scenario like this is to collect data actively as it attempts to reach the goal. Through trial and error, the object will eventually reach the goal and collect data along the way.

Another interesting twist would be to add multiple objects to the workspace. Moving away from minigolf, we could instead look at a game like pool or billiards. Instead of directly pushing the object, the robot could perhaps use the ball to push the rubber duck to a goal state. There could be many different balls and the robot could even learn an actual game of pool. Some situations that could arise in such a study include rebounds off the walls, unexpected collisions, or balls being pushed outside of reachable areas. Other multi-object interaction studies would also further the system, and adapting the model for more complex interactions would not require a complete overhaul.

The robot can learn the physical properties of the objects it interacts with, even multiple at a time. It can learn about its environments, obstacles, boundaries or even terrain. A logical possible extension would be to have the robot learn gravity, a physical property of the world as a whole. Using a stack of blocks or objects as the subject, the robot could simply push the stack at varying heights and track the final locations of each object as the stack topples. This would imbue in the robot model an understanding of the acceleration caused by gravity and how far away things will fall depending on the height they began at. All of these, whether the robot is learning object, workspace, or world properties, require an accurate vision system able to segment and track all objects in the trial. They require a motion system able to accurately push (or even pick up and move in future studies) the tracked objects around. Most of all, the future work will revolve around collecting more data, increasing the accuracy of the model and of all of the further applications as well.

5.5 Conclusion

In this study, we wished to emulate the way that a human being learns on a robot. Essentially, we wanted the robot to ”learn by doing”, obtain the models that it required to successfully manipulate and plan with objects entirely through the act of manipulating those objects. To accomplish this, a data collection pipeline using the UR-10 robot arm and several cameras and kinects was established to obtain a corpus of object manipulation data. This data, or a portion of it, was used in order to train a linear regression model used to predict the final position of an object given the initial position and the push direction. The high accuracy of this model is the main goal of this study, and our initial claim that this model would out-perform both the linear extrapolation and random baselines was shown to be true.

We then inverted this model, using it and a given goal position to find all the possible initial positions that can reach that goal in 1 push, then extrapolating backwards and making a graph of the pushes. We used a shortest path finding algorithm, A* to find the path with the least number of pushes to move the object from its current position to the desired goal. After running full closed loop path planning trials, we looked at all the causes of failure and uncertainty in our experimental design. Analyzing these weaknesses in our system allowed us to look at future improvements that can be made to greatly augment both the accuracy of the predictive model and the path finding ability of the closed loop solution. By attempting to imitate human learning, we have made robot learning more accurate and self-contained. We have shown that by learning from self-collected data, a robot is able to learn a predictive model more accurate than a mathematically sound linear extrapolation model. From this, we have made it possible for a machine to learn the uncertainties of its environment, such as where it can and cannot move, and how different surfaces and objects react differently to manipulation. Our study has shown that like a human being, a robot can “learn by doing”.

Bibliography

[1] S. Šabanović, "Inventing Japan's 'robotics culture': The repeated assembly of science, technology, and culture in social robotics," *Social Studies of Science*, vol. 44, no. 3, pp. 342–367, 2014.

[2] D. Tolani, A. Goswami, and N. I. Badler, "Real-Time Inverse Kinematics Techniques for Anthropomorphic Limbs," *Graphical Models*, vol. 62, pp. 353–388, 9 2000.

[3] P. Agrawal, A. Nair, P. Abbeel, J. Malik, and S. Levine, "Learning to Poke by Poking: Experiential Learning of Intuitive Physics," *arXiv*, 2016.

[4] J. Wu, I. Yildirim, J. Lim, W. Freeman, and J. Tenenbaum, "Galileo : Perceiving Physical Object Properties by Integrating a Physics Engine with Deep Learning," *Advances in Neural Information Processing Systems 28 (NIPS 2015)*, pp. 1–9, 2015.

[5] L. Smith and M. Gasser, "The Development of Embodied Cognition: Six Lessons from Babies,"

[6] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building Machines that learn and think like people," *arXiv*, pp. 1–54, 2016.

[7] P. W. Battaglia, J. B. Hamrick, J. B. Tenenbaum, P. W. Battaglia, J. B. Hamrick, J. B. Tenenbaum, and R. M. Shiffrin, "Simulation as an engine of physical scene understanding Citation Detailed Terms Simulation as an engine of physical scene understanding," *Proceedings of the National Academy of Sciences*, vol. 110, no. 45, pp. 18327–18332, 2013.

[8] T. Ullman and N. Goodman, "Learning physics from dynamical scenes,"

[9] R. Baillargeon, "Infants' physical world," *Current Directions in Psychological Science*, vol. 13, no. 3, pp. 89–94, 2004.

[10] E. Oztop, N. S. Bradley, and M. A. Arbib, "Infant grasp learning: A computational model," *Experimental Brain Research*, 2004.

[11] K. Fragkiadaki, P. Agrawal, S. Levine, and J. Malik, "Learning Visual Predictive Models of Physics for Playing Billiards,"

[12] B. Zheng, Y. Zhao, J. C. Yu, K. Ikeuchi, and S. C. Zhu, "Detecting potential falling objects by inferring human action and natural disturbance," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2014.

[13] W. Li, S. Azimi, A. Leonardis, and M. Fritz, "To Fall Or Not To Fall: A Visual Approach to Physical Stability Prediction,"

[14] E. Davis and G. Marcus, "The scope and limits of simulation in automated reasoning," 2016.

[15] Goodman Noah D. and M. C. Frank, "Relevant and Robust A Response to Marcus and Davis (2013)," *Psychological Science*, vol. 26, no. 4, pp. 539–541, 2015.

[16] A. Lerer, S. Gross, and R. Fergus, "Learning Physical Intuition of Block Towers by Example,"

[17] R. Mottaghi, M. Rastegari, A. Gupta, and A. Farhadi, "What happens if... learning to predict the effect of forces in images," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.

[18] L. Pinto, D. Gandhi, Y. Han, Y.-L. Park, and A. Gupta, "The Curious Robot: Learning Visual Representations via Physical Interactions,"

[19] L. Pinto and A. Gupta, "Learning to Push by Grasping: Using multiple tasks for effective learning," *arXiv*, 2016.

[20] L. Pinto and A. Gupta, "Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2016.

[21] "ROS."

[22] T. T. Andersen, *Optimizing the Universal Robots ROS driver.* Technical University of Denmark, Department of Electrical Engineering, 2015.

[23] I. A. Sucan and S. Chitta, "MoveIt!."

[24] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. London: Springer-Verlag, 2009.

[25] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL),"

[26] K. P. Murphy, *Machine Learning: a Probabilistic Perspective*. MIT press, 2012.

[27] B. Noble and J. W. Daniel, *Applied linear algebra*, vol. 3. Prentice-Hall New Jersey, 1988.

[28] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering Route Planning Algorithms," in *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation* (J. Lerner, D. Wagner, and K. A. Zweig, eds.), pp. 117–139, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[29] W. Zeng and R. L. Church, "Finding shortest paths on real road networks: the case for A*," *International Journal of Geographical Information Science*, vol. 23, no. 4, pp. 531–543, 2009.