Model-Based Design for Full-Stack Robot Manipulation

by

Katy G. Muhlrad

S.B., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Revised July 30, 2019

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by . . . .
Russ Tedrake
Toyota Professor of EECS, Aero/Astro, and MechE
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Model-Based Design for Full-Stack Robot Manipulation

by

Katy G. Muhlrad

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As robotic manipulation research becomes more prevalent, it is crucial to develop clean, modular, and testable systems for researchers to make advances in their fields of expertise without sacrificing usability for others. It is also important to introduce those methods to beginners as they are entering the field, so they can build upon them in novel research. This thesis introduces a framework, informally called the Manipulation System, built using Drake `Systems` based on work done for the Fall 2018 MIT class Intelligent Robot Manipulation. This thesis also presents the groundwork for performing full-stack robot manipulation tasks and proposes extensions to make the system more usable and accessible to all.

Thesis Supervisor: Russ Tedrake
Title: Toyota Professor of EECS, Aero/Astro, and MechE

Acknowledgments

There are many people I would like to thank for supporting me along the way of bringing this project to life.

Thanks to Russ for starting the Manipulation Station stack and for working with me through many iterations of this project to narrow it down to something reasonable to finish in the short time I had, but still very valuable to future students and researchers. His flexibility and willingness to adapt my project as needed made this project a much better experience.

Thanks to Tomàs and Pang for all their hard work in 6.881 to lay the groundwork of this project. An extra thank you goes to Pang for continuing to support me this semester in remaking the Plan Runner.

A big thank you goes to the Drake Developers at the Toyota Research Institute, especially Eric, Jeremy, and Sean, for quickly answering all of my questions and making changes to Drake that were necessary for me to move forward.

I would also like to thank the rest of the members of the Robot Locomotion Group for always answering my questions and generally creating a warm environment I enjoyed being a part of. I will miss you all and I hope I will get the chance to work with some of you again someday.

And finally, thank you to all of my wonderful roommates, lunch buddies at the Edgerton Center, friends both local and remote, and my family for always being on my side and helping me get through this final year. I couldn't have done it without all the people I care about rooting for me.

Contents

List of Figures

Chapter 1

Problem Statement and Motivation

Imagine your kitchen after cooking a complicated meal. The counter is probably full of leftover ingredients, cooking utensils, and spilled food. While cleaning up may seem like an annoying task, it's not very difficult. You just put the unused ingredients back in the cabinets, put the dirty dishes in the dishwasher, and wipe down the counter once it has been cleared. Simple.

Now imagine trying to program a robot to do the same task. All it must do is put the unused ingredients back in the cabinets, put the dirty dishes in the dishwasher, and wipe down the counter once it has been cleared. But in order to do that, it has to know what all of the ingredients are, where they go, the difference between a clean dish and a dirty dish, how to put a dish in a dishwasher, how to wet a rag to wipe down the counter, and many other pieces of information that you as a human didn't consciously think about when doing your own cleaning. Suddenly this simple task of cleaning up doesn't seem so simple.

The field of robotic manipulation is full of hard, unsolved problems. Perception challenges arise when trying to identify and reason about objects, such as deciding if a carton of milk should go into the fridge or cupboard. Tasks such as loading a dishwasher or wiping down a counter require many - possibly expensive - planning and control computations. Additionally, trying to combine smaller tasks into completing a common goal requires huge system integration efforts.

Because all the subtasks in areas such as perception and planning that go into a full manipulation task are difficult problems on their own, most researchers independently develop algorithms to solve them on very different robot setups. While one group can show an accurate way to identify milk cartons with a particular camera, and another group could write a fast planner to shelve different sizes of cartons, combining the two innovations to make a robot put a carton of milk in a fridge is still a hard problem, especially if the algorithms were not specifically designed to integrate with other researchers' algorithms. This process has become much better in the past decade with the widespread adoption of the Robot Operating System (ROS). However, there are some major disadvantages of ROS which make robust system design difficult.

This thesis presents a model-based systems framework to program full-stack robot manipulation tasks. This work, informally referred to as the Manipulation System, is a clean, deterministic environment where people can develop algorithms related to manipulation to perform real tasks. With working implementations of simple algorithms, researchers can focus on one particular challenge, plug their algorithm into the rest of the system, and immediately run a shelving task without having to do much system integration work.

The structure of the rest of the thesis is as follows. A general overview of systems theory and how it relates to robot manipulation can be found in Chapter 2. Chapter 3 introduces the hardware and simulation setup used throughout the project. Chapters 4 and 5 detail the challenges and system components related to perception and planning, respectively. Chapter 6 discusses using the system as a whole. Finally, Chapter 7 summarizes the work.

Chapter 2

Systems Theory and Model-Based Design

Systems theory is a very broad field that can apply to many different areas in engineering, science, and society [21]. It generally includes strategies to design large, working systems. While there are general principles that apply to all systems, each subfield has its own unique set of challenges. This chapter will discuss a broad overview of systems theory and how it applies to robotics. Then the general framework for the rest of this work will be introduced.

2.1 Systems Theory

Any type of system can grow to be very complex due to interacting requirements, desire for generality, and to maintain high utilization. Trying to design for any one of these sources of complexity often raises multiple problems, such as difficulties scaling up the system, or effects from one part of the system propagating to another seemingly unrelated part of the system. In order to manage these problems, there are many design principles to guide the design process. Some of these include using safety margins, avoiding over generalizing the system, and simplifying subsystems whenever possible [21].

Besides specific design principles, there are some higher level strategies in building large systems, namely modularity and abstraction, layers, and hierarchy. A modular system is one that is split up into discrete components, called modules, that all fit together to accomplish the desired task. These modules can often be exchanged for one another, so the overall system does not depend on the specific implementations of subsystems. For example, often computer parts can be swapped out, so one can upgrade the graphics processing card without replacing any other parts. Abstraction makes modularity formal by specifying the inputs and outputs of each module, and how much communication happens between the modules [21].

Layered and hierarchical systems are both different organizational structures of modules. In layered systems, certain modules are connected to each other in a single layer. They can communicate with other layers only directly above or below them. A hierarchical system has a tree-like structure, and modules can only communicate with their parents and children [21].

2.1.1 Robotics Systems Theory

Robotic systems are required to perform many concurrent calculations while interacting with the real world. Unlike pure software systems, where complexity is mostly limited by human ideas [21], robotic software systems need to be connected with the real world and obey the laws of physics.

Asynchronous Message Passing

A common paradigm for designing robotic software systems is asynchronous message passing, often by using Robot Operating System (ROS). ROS is more of a collection of tools and libraries than an operating system [34].

Using ROS leads to modular systems by writing *nodes*, which are independent programs. Nodes can be designed to do many different things, such as parsing sensor data or sending commands to a robot. The abstraction comes from specifying a set of *topics*, or communication channels between nodes. Each topic stores a single data type, which can be as simple as an integer or as complicated as an entire color image.

Users can organize nodes however they want, including in a layered or hierarchical matter by restricting which nodes communicate with each other.

The topic communication system is all asynchronous, so nodes can send and receive messages through these channels at their own rates. This is good in that it makes it easy for nodes to be independent and easily parallelizable at run time, but it also means the system is not easily testable. In asynchronous message passing, there are no guarantees on messages being sent or received [3]. This makes the system very non-deterministic, because nodes will not necessarily receive the expected input or send the expected output successfully. When working with the real world there will always be an aspect of non-determinism, but if the system is completely simulated, it should be as deterministic as possible. For this same reason, testing nodes can be hard if they all rely on sending and receiving messages, which is expensive to run for testing purposes to ensure basic functionality that does not depend directly on sending or receiving messages.

Model-Based Design

In order to eliminate the non-determinism in simulation and to make systems easier to understand and test, people often use a paradigm called Model-Based Design (MBD) [39]. MBD is about using numerical software models in place of hardware models in order to test individual components much more cheaply than on hardware. MBD often specifically applies to control theory. The block diagram shown in Figure 2-1 is a classic diagram of feedback control. The goal is for the *output signal* y to match the *reference input* r. The difference signal e, or the *error*, between the two signals is fed into a controller, which calculates the appropriate command u to send to the robot. The robot is usually called a *plant*, which is a more general term for a dynamical system. The plant is composed of the actuators which do the work to affect the physical state of the system, which in turn is measured by the sensors. This plant is always an approximation of how the real system behaves. It is important to note that the signals r, u, and y can be vectors representing multiple aspects of the plant, such as both position and velocity.

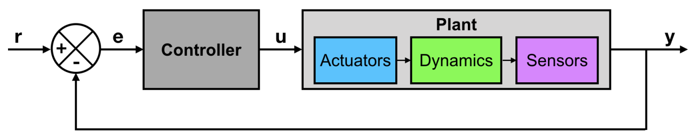


Figure 2-1: A standard block diagram for a feedback control system. The output y is subtracted from the input r to make an error signal e. The error is fed into the controller, which produces a command u to send to the plant.

2.2 Drake `Systems` View

A popular piece of software for implementing systems with MBD is Simulink by MathWorks [29]. However, this software is proprietary and not accessible to all users. This project uses a free and open-source software stack called Drake instead of Simulink to implement MBD. Drake is a software library with tools for modeling and designing advanced dynamical systems [42]. It was originally developed in Prof. Russ Tedrake's Robot Locomotion Group at MIT, but is now being developed by a team at the Toyota Research Institute.

A Drake `System` is a module that has typed *input ports* and *output ports* that can be wired to other `Systems` and/or accessed in surrounding code. Combining `Systems` creates a `Diagram`, which can be thought of as a single, large `System`, which also has input and output ports. A `Diagram` can also contain other `Diagrams`. The input and output ports are strongly typed. Evaluating ports is very fast due to the underlying caching system, whose implementation details are not important to understand when working at the high level in this thesis. Additionally, `Systems`, like any other C++ or Python class, can accept arguments in their constructors, separate from input ports. This project makes heavy use of Drake `Systems` for designing modular and deterministic robotic systems.

Figure 2-2 shows a system port diagram, which will sometimes be referred to as a full system diagram, for a generic `System`. The name of the system is in the center box, which represents the internal code of the module. On the left are the input ports, which are arrows going into the block. The arrows on the right coming out of the system are the output ports. This generic `DrakeSystem` has n input ports and m output ports. There is no rule on how many input and output ports systems have, so any of the following relationships could hold: $n < m$, $n = m$, $n > m$. A `System` can also have no input or outputs ports. Figures 3-5, 3-6, and 3-7 show examples of systems with only input ports and no output ports. Different instances of a certain `System` can also have different numbers of inputs and/or output ports. Every instance of the `PointCloudConcatenation System` shown in Figure 4-4 has a different number of input ports depending on a constructor parameter.

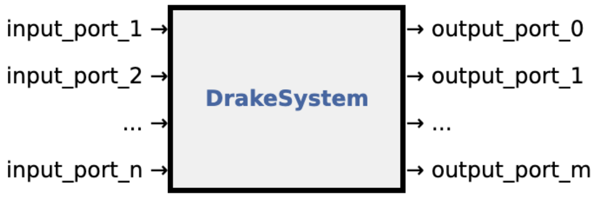


Figure 2-2: The input-output port diagram of a generic Drake `System`. The arrows on the left are the typed input ports of the `DrakeSystem`. The arrows on the right are its output ports

2.2.1 Benefits of the `Systems` View

Writing the software in the Drake `Systems` framework is an intentional design decision. Although it is very different from what many people may be used to, it has clear pedagogical benefits. First, it is very clean and easy to reason about with MBD. Block diagrams such as the one in Figure 2-3 can be constructed, which looks almost identical to the classic feedback control diagram in Figure 2-1. When used, more complicated diagrams such as the one in Figure 6-1 are constructed to give users a good idea of what is going on and how data is being passed around. It also separates pieces of code into specific chunks with only one or two tasks, resulting in very modular pieces that can be reused in other places. This is very similar to the idea of ROS nodes, which people may be more familiar with. However, using Drake `Systems` eliminates the dangers of asynchronous message passing. The advantage of this is that the system is now deterministic for specific inputs, which means it is testable and reproducible.

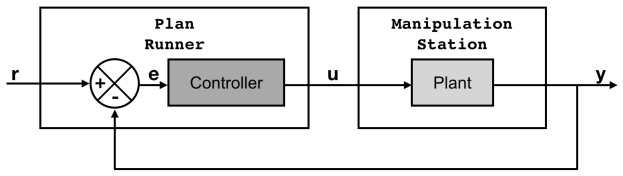


Figure 2-3: A feedback control block diagram as constructed in the Drake `Systems` framework. `PlanRunner` and `ManipulationStation` are `Systems` that will be discussed in detail in later chapters.

Having testable code allows a Continuous Integration system to run on a GitHub repository. Continuous Integration allows developers to write tests the software should always pass. These tests ensure that future updates to the repository do not break important features. Since the `Systems` code is deterministic, it can be tested reliably without relying on external factors such as network communications.

Reproducible code also means users can expect code to run for them exactly as the documentation describes, which reduces the amount of debugging they have to do. It also reduces the number of questions the users ask the original developers, making the system much easier to maintain.

Chapters 3, 4, and 5 will go into detail about many Drake `Systems` useful for robot manipulation, and Chapter 6 will describe ways of putting them all together.

Chapter 3

The Manipulation Station

3.1 6.881

During the Fall 2018 semester at MIT, Prof. Russ Tedrake and Prof. Tomàs Lozano-Pèrez offered a new course titled Intelligent Robot Manipulation, numbered 6.881 [45]. During this course, students explored different aspect of robotic manipulation both in simulation and on real hardware. The assignments revolved around working on the Manipulation Station, a hardware setup provided by Amazon Robotics. Pictures of the software visualization and the real Manipulation Station and are shown in Figures 3-1 and 3-2. The station contains a KUKA IIWA arm with a Schunk WSG-50 parallel jaw gripper across from a three-shelf cabinet. The cage structure around the robot has three Intel RealSense D415 RGB-D cameras mounted at different angles to get a complete view of the table and the cabinet. Assignments given to students using the Manipulation Station included picking up an object and putting it inside the cabinet, locating an object on the table, and opening the cabinet doors.

The software used to interact with the robot was mostly written in Python using Drake. Code was distributed to students in Docker containers, which enabled everyone to run the simulation code without having to install many libraries on their own machines. The code was also set up so switching between controlling the simulated robot for problem sets and the real robot during labs only required changing a Boolean flag.

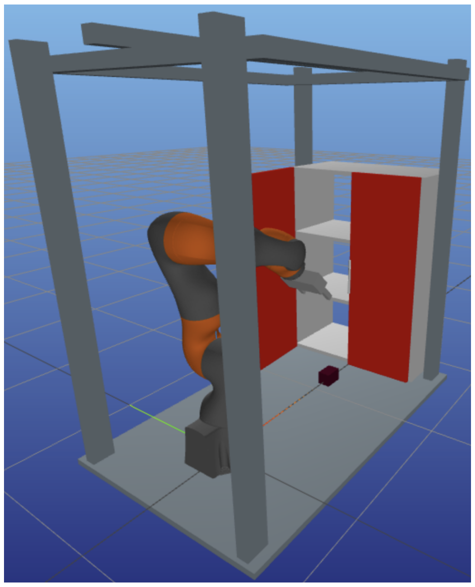


Figure 3-1: The simulated Manipulation Station viewed in MeshCat.

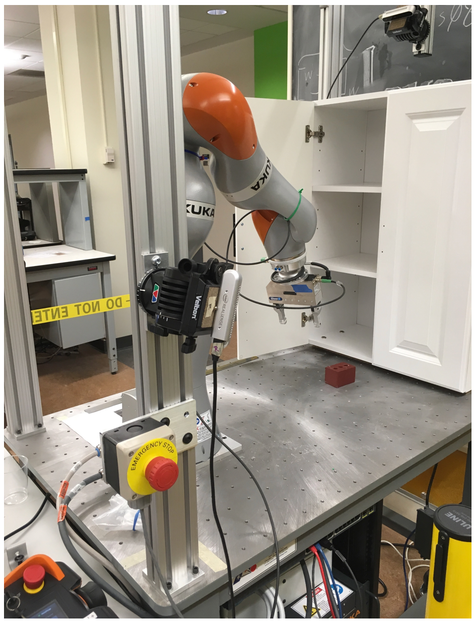


Figure 3-2: The real Manipulation Station in a department teaching lab.

3.2 The `ManipulationStation` Drake `Diagram`

The real Manipulation Station has a software counterpart, `ManipulationStation` which is a Drake `Diagram`. Originally created in Fall 2018 for 6.881, it has since been updated to be more modular and support a second physical workstation setup better suited for clutter-clearing applications.

A `ManipulationStation` has four real-valued vector input ports: the 7 commanded joint positions of the IIWA arm, the 7 commanded joint torques of the arm, the single commanded joint position of the WSG gripper, and the single maximum force threshold of the gripper. The `Diagram` has many output ports, and the exact number depends on how many cameras are added in the software. Each camera has one depth image output port and one color image output port. The complete list of input and output ports is shown in Figure 3-3. The orange ports are only available in simulation for testing purposes, and are not available when running on a physical workstation.

The original `ManipulationStation` was specific to the 6.881 class workstation setup, shown in Figures 3-1 and 3-2. It included the robot, the gripper, the table, the

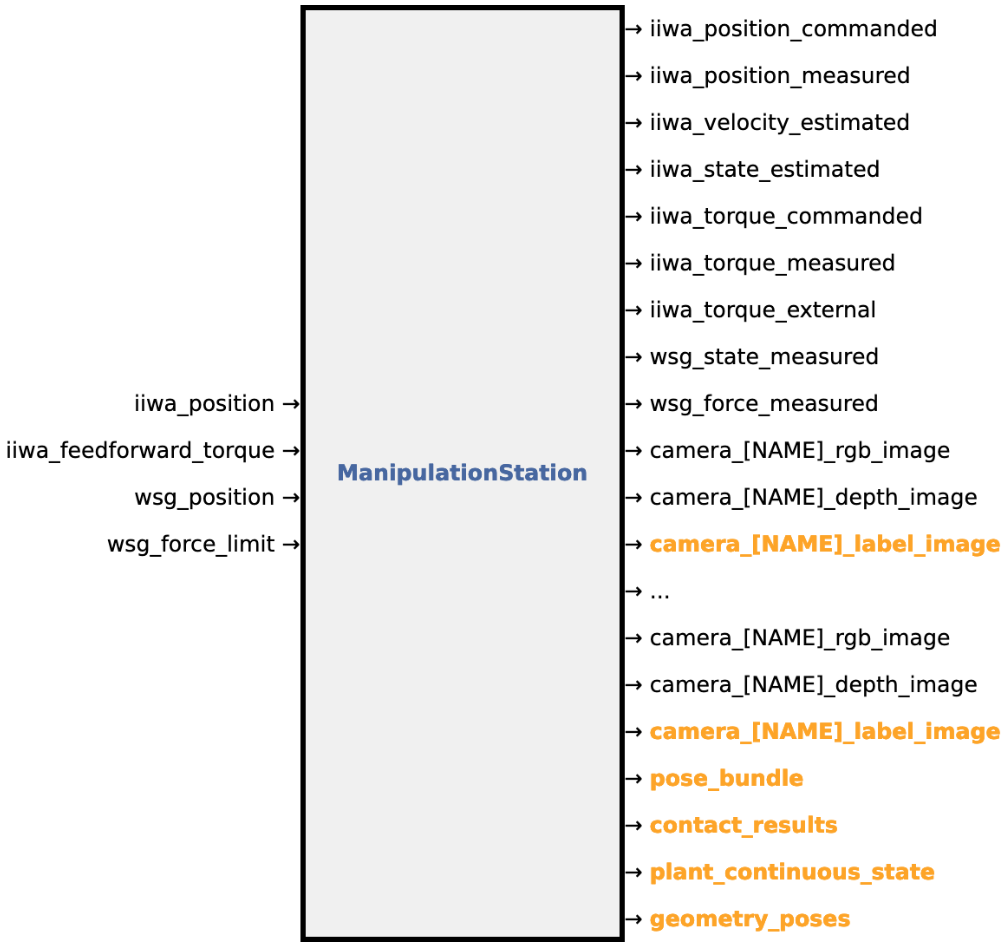


Figure 3-3: The input-output port diagram of a `ManipulationStation`. Image from `https://drake.mit.edu/doxygen_cxx/classdrake_1_1examples_1_1manipulation__station_1_1_manipulation_station.html`, retrieved in March 2019.

cabinet, the surrounding frame, and a single manipuland. This was the only supported setup, and the manipuland had to be added outside of constructing the `Diagram` using an unrelated Drake function. The new `ManipulationStation` is much more modular. There are now two supported physical setups: the original 6.881 workstation, and a cleaner environment with the robot, the gripper, and two bins. Any number of manipulands can be added with a method specific to `ManipulationStation`. This new clutter clearing setup with six objects from the YCB dataset is shown in Figure 3-4. The C++ implementation can be found at `https://github.com/RobotLocomotion/drake/blob/master/examples/manipulation_station/manipulation_station.cc`, and its Python bindings can be found at `https://github.com/RobotLocomotion/drake/blob/master/bindings/pydrake/examples/manipulation_station_py.cc`.

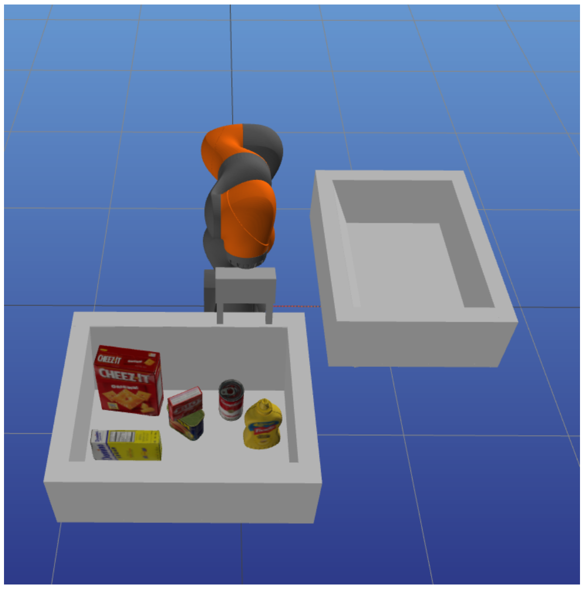


Figure 3-4: The `ManipulationStation` clutter clearing setup with a cracker box, sugar box, mustard bottle, tomato soup can, gelatin box, and a potted meat can.

3.3 Visualization

An important part of running robotic simulations is visualizing relevant data. Most of the visualizations used during 6.881 and throughout the rest of this project are done with MeshCat. MeshCat is a lightweight, JavaScript-based 3D visualizer that runs in web browsers [37]. MeshCat was first incorporated into Drake in the fall for 6.881. It supported viewing the setup, motions of the robot and gripper, and contact forces between two bodies. During Spring 2019, it was extended to visualize point clouds. Each of these pieces is another Drake `System` with only input ports, as shown in Figures 3-5, 3-6, and 3-7. These `Systems` can be inserted into in a large `Diagram` to visualize certain quantities without interfering with other processes. Figure 3-8 shows examples of visualizing the robot workstation, contact forces, and point clouds. The code for all of the MeshCat `Systems` is native

Python that lives at `https://github.com/RobotLocomotion/drake/blob/master/bindings/pydrake/systems/meshcat_visualizer.py`.

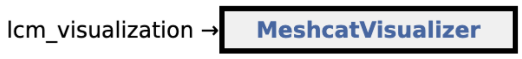


Figure 3-5: The input-output port diagram of a `MeschatVisualizer`

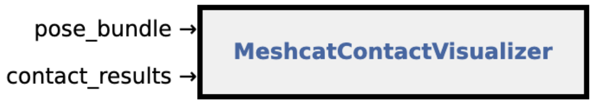


Figure 3-6: The input-output port diagram of a `MeschatContactVisualizer`

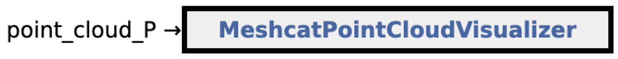


Figure 3-7: The input-output port diagram of a `MeschatPointCloudVisualizer`

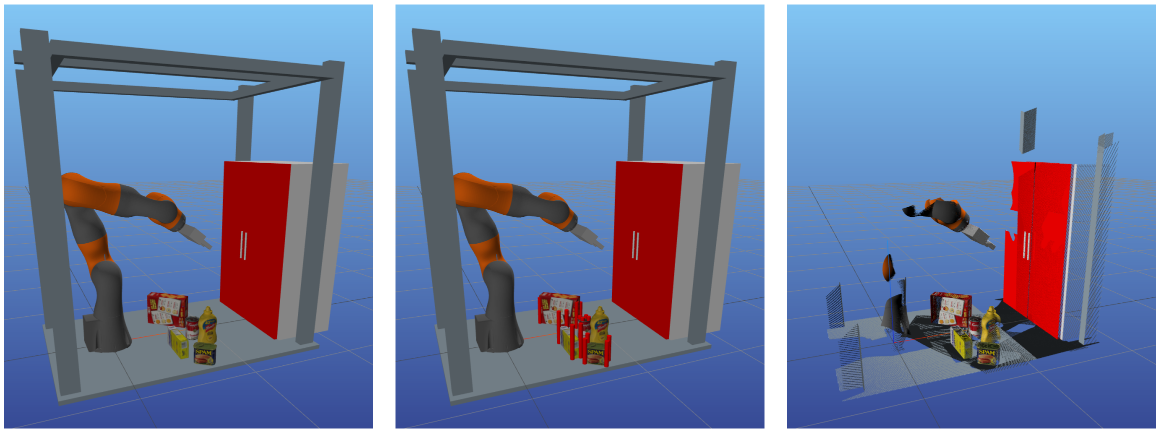


Figure 3-8: The `ManipulationStation` visualized in MeshCat in three ways: (left) the station alone, (middle) the station with contact forces in red arrows, and (right) a point cloud representation of the station.

Chapter 4

Perception

Perception lies at the heart of almost every manipulation task. Take the example from the introduction about cleaning up a dirty kitchen. How would you, as a sighted human, approach the smaller subtask of putting a carton of milk back in the fridge? Chances are, the first step you would take is to look at the kitchen counter for the carton. Without much conscious thought, you would quickly recognize the carton, understand where it is in space, and know how to pick it up. After perceiving the milk carton, the act of moving to pick it up and put it back in the fridge becomes much easier. However, if you had to perform the task blindfolded, it would probably take you much longer to find the milk carton before you could attempt to put it away.

In order for robots to autonomously manipulate objects, they first must be able to perceive them. Perception can refer to a variety of topics, including object class segmentation, object instance segmentation, object recognition, pose estimation, and scene understanding. These topics can be achieved in many different ways utilizing many different sensors. This chapter will focus on object recognition and pose estimation.

4.1 Background and Related Work

4.1.1 The Yale-CMU-Berkeley Object and Model Set

In order to develop perception algorithms, there first must be objects to perceive. Researchers at Yale, Carnegie Mellon, and Berkeley put together a large set of common household objects for other researchers to use in manipulation research, called the YCB Object and Model Set. [8] [7] [6]. The 75 total objects are further classified as food items, kitchen items, tool items, shape items, and task items. The dataset contains the size and weight of each object, as well as various visual representations of all of them. For every object, there are multiple RGB images, depth images, and 3D mesh models. The data is freely distributed, and most of the objects can be easily obtained at grocery stores.

4.1.2 Pose Estimation

The *pose* of an object is its position and orientation in space. In a 2D world, a pose is 3-dimensional, containing the object's x-position, y-position, and rotation about an imaginary z-axis. In a 3D world, a pose is 6-dimensional, containing the object's x-, y-, and z-positions, as well as their rotations about each of those axes, called *roll*, *pitch*, and *yaw*.

One way to estimate object poses is to use geometry. If there exists a 3D model of the object of interest at some known pose, one can find a transform between that and the actual data. This is often done between two point clouds, which are collections of (x, y) in 2D space, or (x, y, z) points in 3D space.

A popular algorithm for finding the transform is called iterative closest point (ICP) [36]. Given an input point cloud, the goal is to find the best transformation that, when applied to the input, results in some known model point cloud. The algorithm is composed of two main steps. The first step finds what are called correspondences between points. This is usually done by finding the nearest neighbor in the model point cloud to the input point cloud. It is assumed that these correspondences should be close together when the two point clouds line up. When individual points are used for correspondences, the algorithm is referred to as point-to-point ICP. The second step estimates the best fit transform between the two point clouds to minimize the distance between all corresponding points. This can be done by calculating the singular value decomposition of the covariance matrix between the input and model point clouds. Once step two is complete and a transform has been estimated, the input point cloud is transformed and both steps are repeated again.

By iterating over these two steps until the error between the input and known point clouds is small, it is possible to get very accurate pose estimates. However, ICP is particularly prone to noise. Because the algorithm is trying to minimize the total distance between all of the points, if the input point cloud has a few points that are very far away from most of the point cloud, the total alignment will be off in order to close the distance between the few outliers and the model point cloud. For this reason ICP is often combined with algorithms for outlier rejection, such as RANSAC. [49]. There are also other variations of ICP that calculate correspondences differently, such as point-to-plane ICP [27].

4.1.3 Object Recognition

Object recognition is identifying a certain set of objects in an image. NVIDIA combined pose estimation with object recognition in one framework. Deep Object Pose Estimation, or DOPE, uses a neural network to both identify objects in images and estimate their poses [25]. The network was trained on a mix of real and synthetic RGB images of YCB objects. The synthetic images were generated with a strategy called domain randomization. By combining photorealistic models and their non-photorealistic counterparts against various backgrounds, they were able to generate a lot of training data with known object categories and 6D poses. This allowed them to train a network to get good classification and pose estimation accuracy without having to generate ground truth pose labels by hand and without the need for depth images.

NVIDIA has provided open-source code for running the network, as well as the final weights they calculated for six YCB objects: the cracker box, the sugar box, the soup can, the mustard bottle, the gelatin box, and the potted meat can [24]. Running their code on an RGB image produces bounding box estimates for all recognized YCB objects in the picture, as well as an estimate of their poses.

4.2 Systems

There are four Drake `Systems` related to manipulation that are dedicated to perception: `DepthImageToPointCloud`, `PointCloudConcatenation`, `DopeSystem`, and `PoseRefinement`. `DepthImageToPointCloud` and `PointCloudConcatenation` are both generic utilities that can be used with many different perception techniques. `DopeSystem` is an adaption of a specific state of the art object recognition and pose estimation algorithm. `PoseRefinement` is a flexible system that lets users experiment with different algorithms that accomplish the same goal.

4.2.1 `DepthImageToPointCloud`

As its name suggests, `DepthImageToPointCloud` takes an input depth image and outputs a point cloud. Its full port diagram is shown in Figure 4-1. By using knowledge about the intrinsic properties of the camera that captured the depth image, such as its width, height, and distortion factors, the depth value at each pixel can be converted into a point in 3D space. The system can optionally take in a color image from the same camera to add color to the point cloud. By default, the output point cloud will be in the camera's coordinate frame, but can be moved by supplying a homogeneous transformation to the system. Each system can only convert one depth image and its corresponding color image into a point cloud. Example color and depth image inputs are shown in Figure 4-2. The C++ implementation of `DepthImageToPointCloud` can be found at `https://github.com/RobotLocomotion/drake/blob/master/perception/depth_image_to_point_cloud.h`. Its Python bindings can be found at `https://github.com/RobotLocomotion/drake/blob/master/bindings/pydrake/perception_py.cc`.

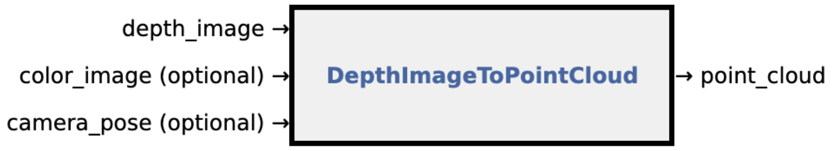


Figure 4-1: The input-output port diagram of a `DepthImageToPointCloud`. Image from `https://drake.mit.edu/doxygen_cxx/classdrake_1_1perception_1_1_depth_image_to_point_cloud.html` retrieved in May 2019.

Example Usage

The Manipulation Station used in 6.881 had three cameras surrounding the robot, one on its left side, one on its right side, and one above the cabinet structure. Figure 4-2 shows the color and depth images from each of these cameras. The depth images are displayed as grayscale images. Darker areas appear closer to the camera, and pure white areas extend beyond the maximum distance read by the camera. Each pair of color and depth images can be fed into a `DepthImageToPointCloud`, which produces the point clouds in Figure 4-3 for the left camera, middle camera, and right camera respectively.

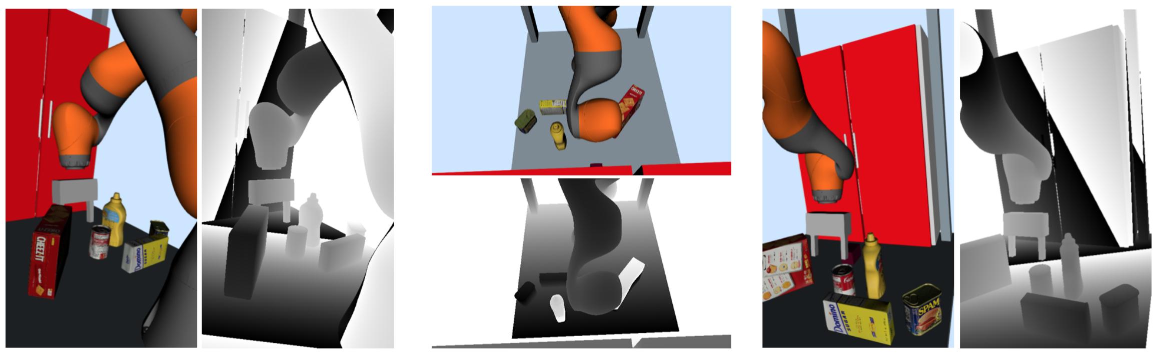


Figure 4-2: The color and depth images from the Manipulation Station's left camera (left), middle camera (center), and right camera (right). Darker shades in the depth images indicate points closer to the camera, and completely white pixels indicate areas beyond the maximum distance measured by the camera.

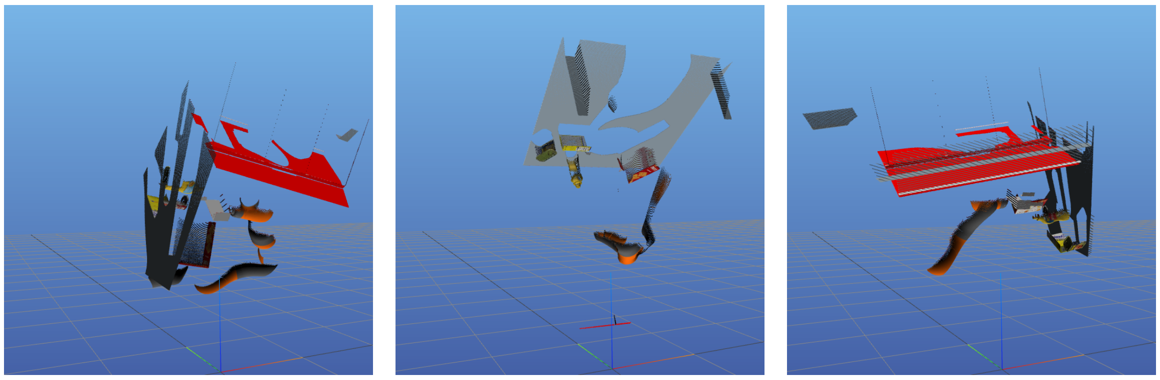


Figure 4-3: The point clouds produced by running each pair of images in 4-2 through a `DepthImageToPointCloud System`. The left, middle, and right images in this figure corresponding to the point clouds from the left, middle, and right cameras on the Manipulation Station respectively.

4.2.2 `PointCloudConcatenation`

`PointCloudConcatenation` is a newly created `System` that makes working with point clouds from multiple sources easier. Its constructor takes in a list of point cloud IDs, which are often the serial numbers of the cameras the point clouds come from. The `System` has an input ports for each of the listed point clouds and their homogeneous transformations to a common frame. It has a single output port containing one point cloud composed of the transformed input point clouds. Figure 4-4 shows its system diagram. The code for this `System` is on the Drake master branch at `https://github.com/RobotLocomotion/drake/blob/master/bindings/pydrake/systems/perception.py`.

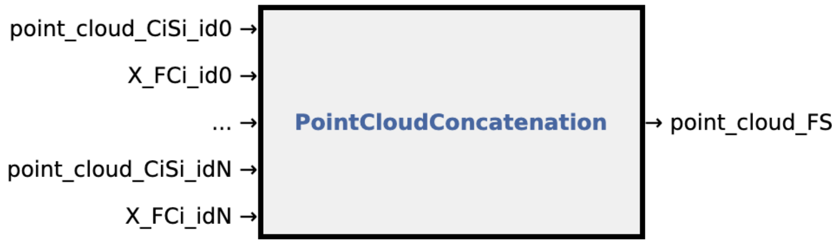


Figure 4-4: The input-output port diagram of a `PointCloudConcatenation`. `Ci` is the frame of point cloud i and `F` is the common frame of the final combined point cloud.

Example Usage

A common use of this `System` is to align point clouds from multiple cameras into the world frame. Figure 4-5 shows the resulting point cloud when the three point clouds in Figure 4-3 are combined using the external camera calibration parameters of the Manipulation Station. This also is an example of how combining different `System`s leads to more useful results than using a `System` on its own, since the point clouds in Figure 4-3 were all originally produced by `DepthImageToPointCloud Systems`.

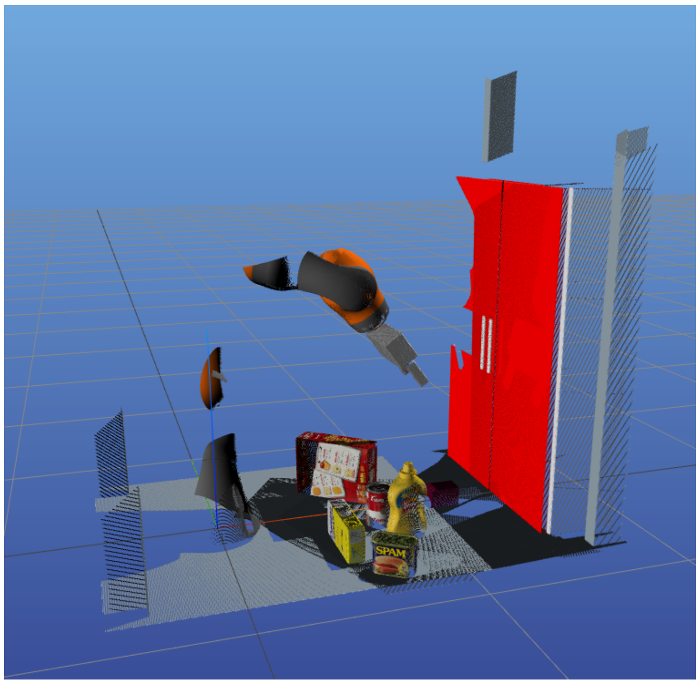


Figure 4-5: A point cloud in the world frame constructing by combining the individual camera point clouds from the Manipulation Station, shown in Figure 4-3.

4.2.3 `DOPE`

A `DopeSystem` (sometimes stylized as `DOPE`) is a Drake `System` that runs the inference step of NVIDIA's DOPE [25]. The code provided by NVIDIA that runs as a ROS process was adapted to run within a Drake `System` without asynchronous message passing. Each `DopeSystem` takes in a path to weights for each object, and a path to a configuration file. It also has a single color image input port. It has two output ports: a pose bundle and another color image. A *pose bundle* is a set of poses, which can have names. The output port contains the estimated pose of each of the objects listed in the given configuration file. The color image output port contains an annotated version of the input color image with bounding boxes surrounding each identified object. See Figure 4-6 for the full port diagram. The `DopeSystem` class can be found at `https://github.com/RobotLocomotion/6-881-examples/blob/km_thesis_work/perception/dope_system.py`.

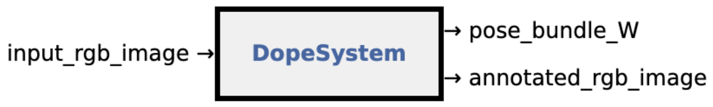


Figure 4-6: The input-output port diagram of a `DopeSystem`.

Example Usage

Figure 4-7 shows an image taken by one of the RealSense D415s on the real Manipulation Station annotated by the original ROS implementation of DOPE. Figure 4-8 shows similar results applied to simulation produced using the `DopeSystem`. Figure 4-9 shows the full 6D pose estimates overlaid on the ground truth objects in two different scenarios. All objects between the two images are in the same location except the potted meat can. Note that DOPE's estimate of the potted meat can's pose is much better in the left image than in the right image. The next `System` presents a way of improving these poses.

4.2.4 `PoseRefinement`

`PoseRefinement` is a new `System` that was based on a similar `System` the author created for 6.881. It is a flexible `System` meant for refining a pose of a single object with a point cloud. The two input ports take in a pose bundle and a point cloud. The constructor requires a model point cloud of the object and a .png of the object texture. The constructor also accepts an optional data structure containing a scene segmentation function and a pose alignment function for a specific object, which

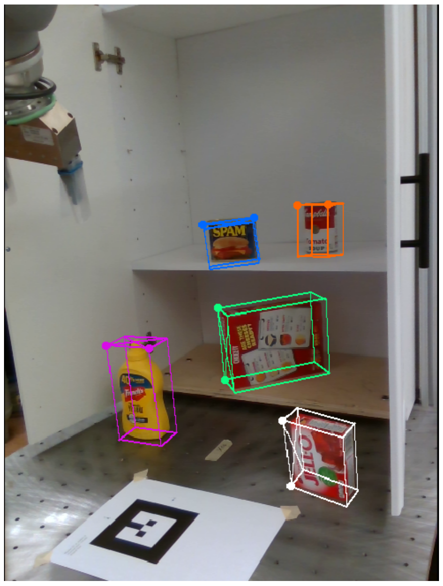


Figure 4-7: An annotated image from one of the Manipulation Station's physical cameras annotated by the original ROS implementation of DOPE.

will be clarified shortly. A `PoseRefinement` has a single output port containing a pose bundle of the calculated poses of all of the objects in the original pose bundle. The input-output port diagram is shown in Figure 4-10. The Python code for `PoseRefinement` can be found at `https://github.com/RobotLocomotion/6-881-examples/blob/km_thesis_work/perception/pose_refinement.py`.

To calculate a refined pose for a single object, first the object is segmented out of the input point cloud, which is usually the point cloud of the entire scene, using either the provided or default implementation of the segmentation function. Then the segmented point cloud is aligned with the model point cloud to estimate the pose of the object using the alignment function. This process is repeated for every object represented in the input pose bundle to form a pose bundle of refined poses to publish on the output port.

The scene segmentation and pose alignment functions have the same method signature. They both take in the scene point cloud, the model point cloud, the object texture image, and an initial pose guess. The scene segmentation function is supposed to remove all points from the scene point cloud that do not belong to the object of interest. The pose alignment function aligns the segmented object with the model

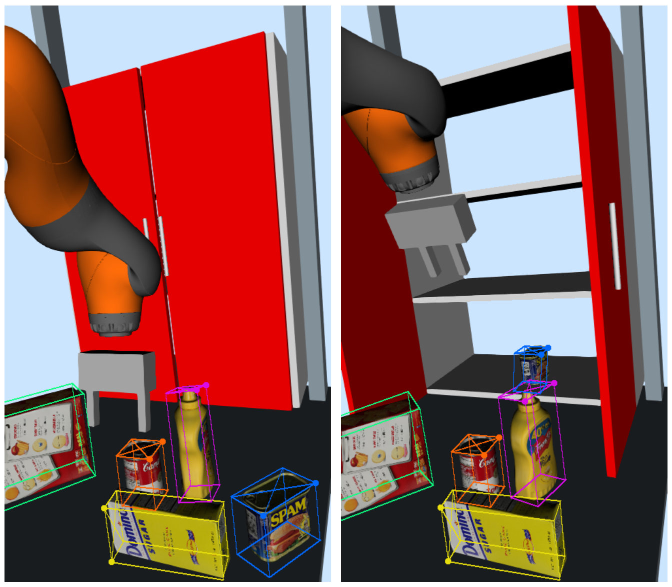


Figure 4-8: RGB images from the simulated Manipulation Station's right camera that have been processed and annotated by `DOPE`.

object.

Users can write their own segmentation and alignment functions, however the `PoseRefinement System` contains default implementations of them both. The default scene segmentation function removes all points that are farther than the size of the largest model dimension away from the initial pose guess. For example, if the dimensions of the model are (1, 2, 1) and the initial pose is located at (0, 3, 4), all points included in the segmented point cloud have x-values between [-2, 2], y-values between [1, 5], and z-values between [2, 6]. The default pose alignment function runs standard point-to-point ICP starting from the initial pose guess.

Example Usage

The best way to use the `PoseRefinement System` is by combining it with all the previously discussed `Systems` as in Figure 6-2. This setup can be used to evaluate different object segmentation and pose alignment functions. The following examples will walk through comparing custom object segmentation functions to the default ones presented above, while using the default pose alignment function.

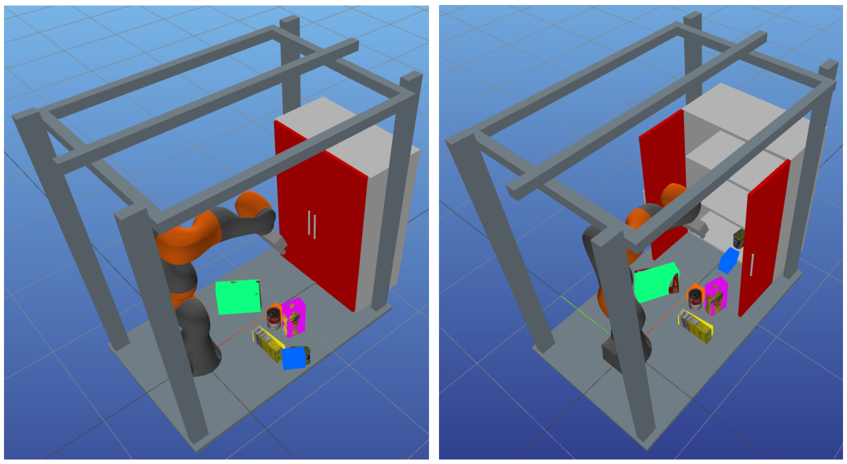
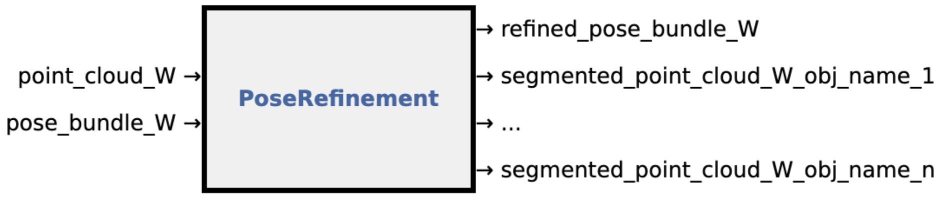


Figure 4-9: The pose estimates of objects computed by DOPE overlaid on the ground truth locations. Note the estimate of the potted meat can's pose is much worse in the right image than in the left image.



PoseRefinement
point_cloud_W
pose_bundle_W
refined_pose_bundle_W
segmented_point_cloud_W_obj_name_1
...
segmented_point_cloud_W_obj_name_n

Figure 4-10: The input-output port diagram of a `PoseRefinement`.

Mustard Bottle Segmentation

DOPE's estimate of the mustard bottle shown in the left image in Figure 4-9 is fairly accurate, but has obvious room for improvement. Figure 4-11 below shows a zoomed in version. The `PoseRefinement System` was used to produce a more accurate pose of the mustard bottle using two different object segmentation functions. The first function was the default implementation described above, which includes every point in a box around the initial pose guess. The second function expanded on the default implementation by filtering out points that are not yellow and removing points that are far away from many other points.

Each of these segmented point clouds is shown in the top two images in Figure 4-12. While both clouds are very different, it is not immediately clear which one is a better candidate for ICP. Figure 4-12 illustrates the full process of segmenting the point cloud, running DOPE to get an initial pose estimate, running ICP to obtain a better pose estimate, and checking the refined estimate with the ground truth data. The default segmentation method resulted in a better final alignment, because the bottom part of the mustard bottle was not included in the segmented point cloud with the custom segmentation function.

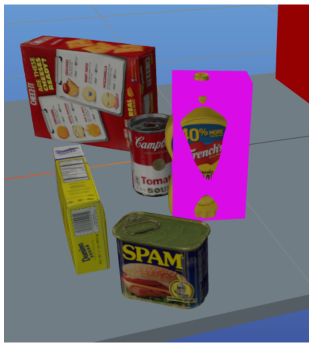


Figure 4-11: DOPE's estimate of the mustard bottle's pose.

Potted Meat Can Segmentation

The two images in Figure 4-9 show the potted meat can in very different locations. DOPE also produces very different pose estimates for each of these locations. The one in the left image is much closer to the ground truth than the one in the right image. Looking back at Figure 4-8, the potted meat can is accurately outlined in images DOPE annotated for each scenario, so the pose discrepancy is not due to poor visibility. In the case of the right image where the can is farther away from the camera and DOPE's initial guess of the can's pose is very far off from the actual object, the default object segmentation function utilized by `PoseRefinement` does not work at all, since the initial guess is too far away from the object to isolate any points. In order to use the `PoseRefinement System` here, a custom segmentation function must be implemented.

The potted meat can was purposefully moved from the first location to the second location by the robot. The segmentation function takes advantage of that knowledge, as well as knowledge about how DOPE behaves for objects farther away from the

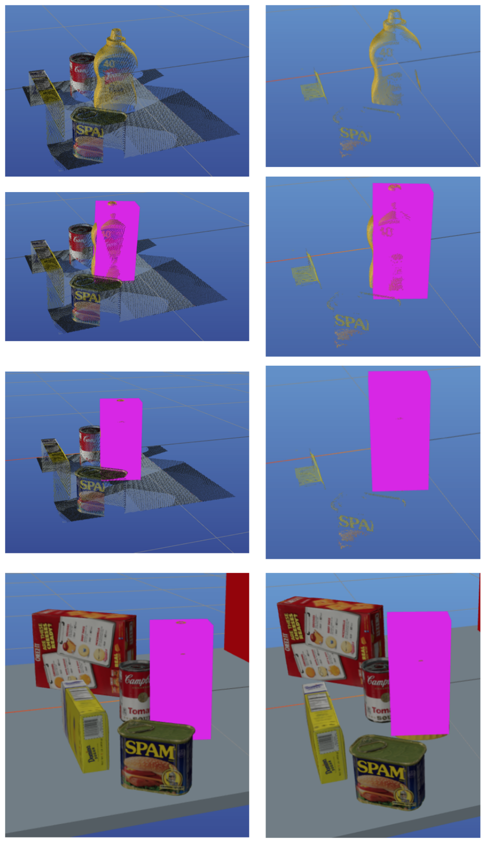


Figure 4-12: A comparison of the default object segmentation function (left) to a custom object segmentation function (right) of the mustard bottle. From top to bottom the images illustrate: the segmented point clouds, DOPE's pose estimate with the segmented point cloud, the refined pose after running ICP against the segmented point cloud, and the final refined pose overlaid on top of the ground truth pose.

camera. First, the segmentation function looks for points in a box 2.5 sizes larger around the initial pose guess than normal, since DOPE is less accurate for objects farther away from the camera. An x-y-z filter is then used on that box of points to isolate points on the bottom shelf, since that is where the robot just placed the object.

Figure 4-13 shows the resulting point cloud from applying this segmentation function (top right) compared to the default segmentation function applied to the original location of the potted meat can (top left). The bottom two images show the final poses produced by `PoseRefinement` after running the default ICP alignment function on the segmented point clouds. The final pose estimates are both very accurate, showing that the specific choice of segmentation function was appropriate in each scenario.

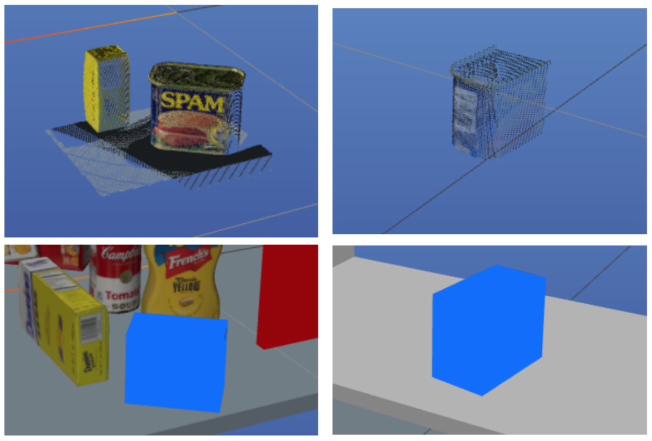


Figure 4-13: A comparison of the default object segmentation function (left) to a custom object segmentation function (right) of the potted meat can in two different locations. The top two images show the segmented point clouds, and the bottom two images show the final refined poses on top of the ground truth data.

Refining Poses from Real Point Clouds

While `PoseRefinement` has not yet been tested on point clouds from the real cameras, there are more expected challenges in getting accurate poses than when running it on simulated point clouds. Figure 4-14 shows one limitation that exists even in the simulation. Although the final aligned pose fits the point cloud well, since the back of the can was invisible to the cameras, the pose was not a perfect fit to the ground truth data. With real point clouds, there is a higher chance of not being able to see parts of objects due to occlusions caused by objects in the world not modeled in simulation (such as the USB cord of the camera). The point clouds in simulation are also noise free, whereas real point clouds have lots of noise. Figure 4-15 shows a point cloud taken with one of the RealSense D415 cameras on the Manipulation Station, which is much noisier than any of the simulated point clouds. To handle the noise, a more sophisticated outlier rejection algorithm such as RANSAC would most likely produce better results than running ICP on a point cloud obtained directly from the cameras.

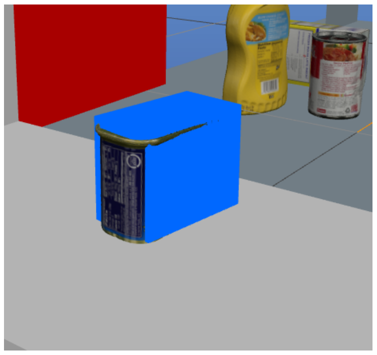


Figure 4-14: Because the back of the potted meat can was not visible to any of the cameras, the final refined pose does not perfectly align with the object, as evidenced when looking at the estimated pose from behind.

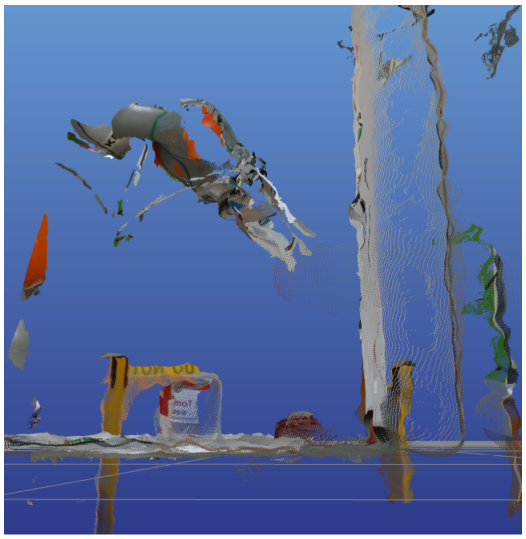


Figure 4-15: A point cloud obtained by a RealSense D415 camera of the Manipulation Station setup shown in Figure 3-2.

Chapter 5

Planning and Control

Now that you've identified and located the carton of milk on your counter, you need to put it back in the fridge. Logically, you would go over to the counter, pick up the milk, walk over to the fridge, open the door, put the milk on a shelf, and then close the fridge door. But just in that seemingly simple sequence, you are making many judgements and are constantly adjusting your actions to match reality. Maybe the milk was heavier than you anticipated and you had to grip it tighter before moving it. Or maybe the fridge was already open so you didn't have to open the door. If there were other things out on the counter, such as a box of sugar or a stack of dirty dishes, you most likely reached around them and didn't hit them at all.

A robot needs to be capable of making all of those small judgements, as well as figuring out how to move its arm through space to get to where it wants. Even if the problem is simplified such that an arm on an immobile base can reach the milk and the fridge, figuring out how to move around in that space is not trivial. Planning and control revolve around allowing a robot to perform these sorts of tasks.

5.1 Background and Related Work

5.1.1 Planning

STRIPS Planning and PDDL

The **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver (STRIPS) was developed by Richard Fikes and Nils Nilsson in 1971 to control Shakey the robot [38]. In this formulation, the world is represented by a set of true or false statements, called *predicates*. The truth values of these statements are changed by performing named atomic *actions*. Each action has a name, a cost, a set of *preconditions*, and a set of *postconditions*. The current state of the world must satisfy all preconditions in order to perform the action. Once the action is finished, the world then contains the values from the postconditions. The truth values of predicates in the world not included in an action's postconditions are not changed after performing that action. The goal in using STRIPS is to find a sequence of actions that turns an initial set of predicates into a desired set of predicates.

One of the most common ways to use STRIPS is through the Planning Domain Definition Language (PDDL, pronounced pid-id-al) [15]. Heuristic forward search algorithms such as A* are often used to generate plans. Planning is usually done using a relaxed version of the problem by ignoring predicates that are removed by performing each action. Different heuristics to inform the search have been developed, including ones called h_max, h_add, and h_FF. h_max returns the maximum cost of performing an action to satisfy a goal condition, h_add takes the total cost of satisfying every goal condition independently, and h_FF uses a greedy backwards-search to bound the cost-to-go [11].

PDDL in 6.881

In the Fall 2018 semester of 6.881, a specific implementation of PDDL, called PDDLStream [13], was used to generate plans to pick up a can of soup and place it in a cabinet, referred to as the *stowing task*. PDDLStream was developed by MIT graduate student Caelan Garrett, who worked on integrating his work with Drake to be used for the class. A video of the KUKA IIWA performing the stowing task using a plan found by PDDLStream can be found at `https://youtu.be/fMBon3IGmeI`.

Behavior Trees

The plans produced by PDDLStream are sent to the robot in a feedforward way. An alternative structure of sending plans to the robot is a behavior tree (BT). BTs are expressive, hierarchical way to control systems, similar to state machines. Although they were originally developed by the computer games industry to control non-playable characters, they have since been adopted by other fields, including robotics, due to their reactiveness and modularity [31]. A simple behavior tree that allows a robot to pick up a can of soup and put it in a shelf in a cabinet is shown in Figure 5-1. Figure 5-2 shows a key explaining the symbols.

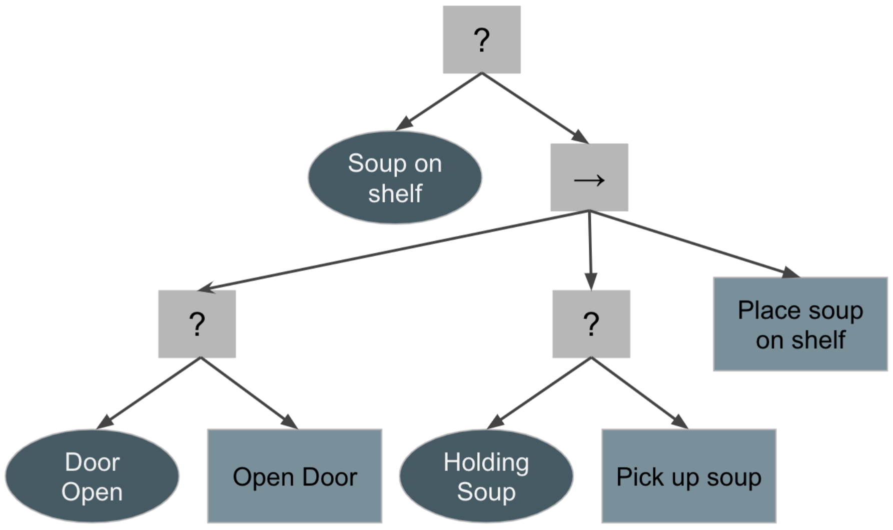


Figure 5-1: A behavior tree that a robot uses to pick up a can of soup and place it inside a cabinet.

Execution starts at the root of the tree, which is the top Selector node in Figure 5-1. The root is the first node to receive a *tick*, which tells the node to run its behavior. Different types of nodes have different responses to ticks, but they will all return a *status* of `RUNNING`, `SUCCESS`, or `FAILURE` when finished.

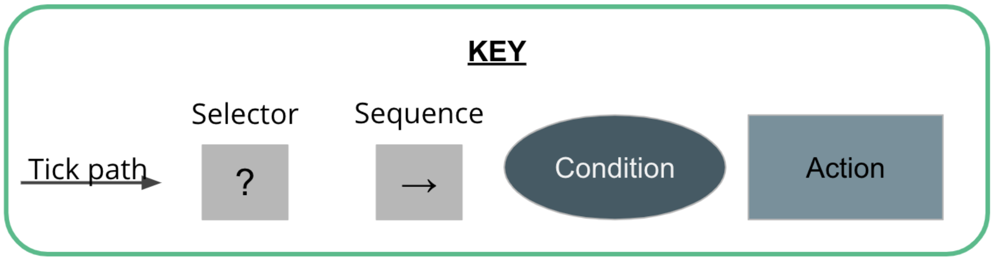


Figure 5-2: An explanation of the nodes and arrows in the behavior tree in Figure 5-1. Selector nodes are also sometimes referred to as Fallback nodes.

Selector nodes (also called *Fallback* nodes) and *Sequence* nodes are both types of control flow nodes. The behaviors they run when ticked consist of ticking their children and returning a status. When a Selector node gets ticked, it ticks its children from left to right and stops once any of them returns `SUCCESS` or `RUNNING`, upon which it returns that same status. If none of its children return `SUCCESS` or `RUNNING`, then the Selector node returns `FAILURE`. Sequence nodes exhibit the opposite behavior. While they still tick their children from left to right, they instead stop and assume the status of the first child that returns `FAILURE` or `RUNNING`. They will only return `SUCCESS` if all of their children return `SUCCESS`.

Condition and *Action* nodes run behaviors that depend on and affect the environment. Condition nodes check conditions and return `SUCCESS` if the conditions are true. Otherwise they return `FAILURE`. When an Action node is ticked, it starts a process and returns the status of that process without blocking the rest of the tree from ticking. If that Action is ticked again, it checks on the process and returns a new status depending on the progress of the process. If the process is running, the node `RUNNING`. Similarly, the node returns `SUCCESS` once the process is finished and `FAILURE` if the process stops without finishing.

BTs are reactive because the root of the tree is ticked on every timestep. Explicit sequences of actions are not generated before executing the task. Instead, the robot performs the appropriate STRIPS action in real time based on different conditions. This also means there is no need to replan if the state of the world changes in an unexpected way, such as if the robot accidentally drops the soup can before placing it in the cabinet. If similar behavior is desired when using a feedforward planning framework such as PDDLStream, a new plan has to be generated every time something unexpected occurs.

Because BTs do not generate feedforward plans, the structure of the tree must be designed. There are a few ways to generate this structure. The first one is designing the tree by hand, such as the one in Figure 5-1. For small tasks, hand-designed trees can be simple yet effective. An alternative is to generate them while running the system using an algorithm such as Planning and Acting Using Behavior Trees (PA-BT) [30]. The key idea of PA-BT is when starting with a set of desired goal conditions, failing conditions are replaced with subtrees that will perform actions to satisfy them. These subtrees are based around STRIPS actions. A BT produced by running PA-BT to accomplish the stowing task is shown in Figure 5-3

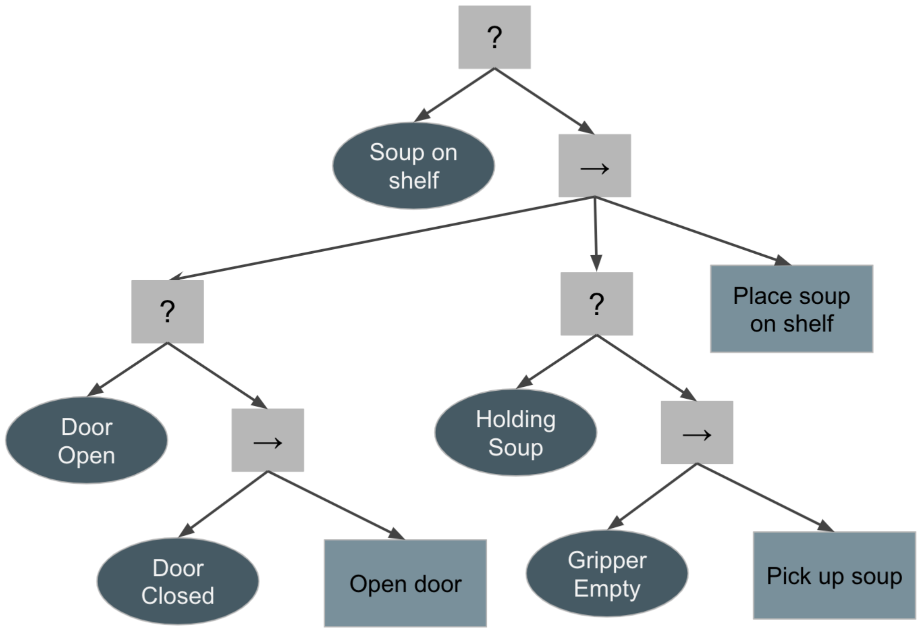


Figure 5-3: A behavior tree produced by running the PA-BT algorithm on the stowing task.

5.1.2 Control

A very broad overview of control theory was given in 2.1.1. The following sections describe more specific controllers

Joint Space Control

A joint space controller is a controller for moving the robot in joint space, which, for the KUKA IIWA, is 7-dimensional for its 7 joints. Joint space refers to the specific angles each joint can go to and the torques they can be moved by. Bringing the current joint positions and torques y to a desired position u can be done by any sort of controller, commonly a proportional-derivative (PD) controller or proportional-integral (PI) controller [17]. For more information about PD and PI controllers, see [10]. Because this feedback controller is embedded in the plant, the joint space controller used in this project is a feedforward controller.

Task Space Control

A task space controller is similar to a joint space controller in that it can use PD or PI controllers to move from the current configuration to a desired configuration, but instead of working with joints directly, it works with a specific point in 3D space. Even though the point exists in x-y-z space, the input and output vectors are actually 6-dimensional by adding the *roll-pitch-yaw* orientation of the point in addition to its location [17]. Unlike the joint space controller, the task space controller that can be implemented with the `System`s described below do use explicit feedback.

Forward and Inverse Kinematics

Forward kinematics (FK) is finding where certain points of a robot are in 6D space given with known angles for all of its joints. Conversely, *inverse kinematics* (IK) is figuring out what joint angles of a robot produce a given 6D pose. This problem is much more difficult and does not always have a unique solution. Calculating IK values is usually an optimization problem of trying to match a FK solution of joint angles to the known point location. [28] describes one such method of performing this optimization. For the specific implementation of computing IK in Drake, see the Drake documentation at [40].

5.2 Systems

The `Systems` in this section use a data structure called `PlanData`. Each `PlanData` instance can represent either a joint space plan or a task space plan. A joint space `PlanData` instance uses a `PiecewisePolynomial` in 7-dimensional space to send joint angles to each of the KUKA's 7 joints. A task space `PlanData` instance contains a 3D `PiecewisePolynomial` for the position in x-y-z space and a 4D `PiecewiseQuaternionSlerp` for orientation of the end-effector. A `PiecewisePolynomial` contains a list of polynomials defined over different time segments. Similarly, a `PiecewiseQuaternionSlerp` contains a list of quaternions to be interpolated with piecewise slerp at different times. More information about these classes can be found in [43] and [44]. Although the `PlanData` struct supports both joint space and task space plans, the systems below only use joint space plans. Task space plans are not supported in Python at this time.

The `PlanData` data structure as well as the `RobotPlanRunner System` were implemented in C++ by Pang Tao, a member of the Robot Locomotion Group. Their Python bindings were written by the author. The C++ code can be found in the directory on a fork of Drake here `https://github.com/pangtao22/drake/tree/robot_plan_runner/manipulation/robot_plan_runner`, and the Python bindings live on the same fork in the directory at `https://github.com/pangtao22/drake/tree/robot_plan_runner/bindings/pydrake/manipulation`.

5.2.1 `RobotPlanRunner`

`RobotPlanRunner` takes in one plan at a time and sends it to the robot. It has input ports for a single `PlanData` instance and the robot's current position, velocity, and external torque. It outputs a position and torque command for the robot. Figure 5-4 shows its full system diagram.

The output position and torque commands are calculated with a `JointSpaceController System`. This `System` is a subsystem of `RobotPlanRunner` and cannot be used directly from Python. Because the IIWA's internal controller is estimated as a joint space feedback controller [41], the `JointSpaceController` simply sends the joint values from the `PiecewisePolynomial` at the appropriate time to the robot.

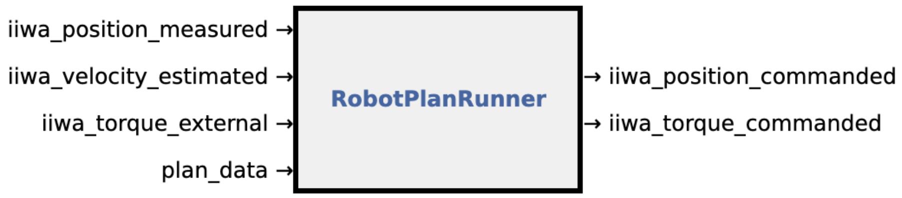


Figure 5-4: The input-output port diagram of a `RobotPlanRunner`.

5.2.2 `BehaviorTree`

The `BehaviorTree System` was designed to send plans to the robot through a `RobotPlanRunner` via an output port for a single `PlanData` instance. It wraps a `py_trees BehaviourTree` object [18] in a Drake `System`. The system diagram for a `BehaviorTree System` is shown in Figure 5-5. The system code can be found at `https://github.com/RobotLocomotion/6-881-examples/blob/km_thesis_work/bt_planning/behavior_tree.py`. To avoid confusing a `py_trees BehaviourTree` object with the `BehaviorTree` Drake `System`, the `py_trees` object will be referred to simply as a `Tree`. The `py_trees` library, written by Daniel Stonier, contains a basic implementation of BTs in Python [18]. In addition to the different node types described in 5.1.1, there is a data structure called a `Blackboard`, which is a key-value store shared by all nodes in a `Tree`.

The `BehaviorTree System` has input ports for the current state of the world: the robot's current position, velocity, and external torque; the gripper's position, velocity, and force; and relevant poses of objects in the world frame. Its constructor takes the

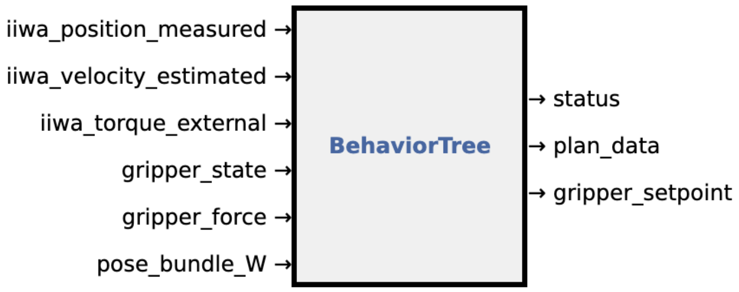


Figure 5-5: The input-output port diagram of a `BehaviorTree`.

root node of a `Tree`, and string names of objects in the world relevant to the stowing task. It periodically ticks the root of its `Tree`, but will first update the `Blackboard` with the values read in from the input ports. The values are written and stored in the `Blackboard` instead of being used directly so that the `System`'s `Tree` can be a native `py_trees` object instead of needing another Drake wrapper. The `Tree` passed into the `System` should look something like the ones in Figures 5-1 or 5-3, where it uses the robot's `Blackboard` variables to calculate the next plan to send to the robot.

The plan calculated by the `Tree` gets written to the `plan_data` output port of the `BehaviorTree`. The `System` also has an output port for the status of the `Tree`, which is an integer representing `SUCCESS`, `RUNNING`, or `FAILURE`. The final output port is the position to set the gripper fingers to, which is not currently included in `RobotPlanRunner`.

Example `BehaviorTree` Use

As described above, the `BehaviorTree System` has variables related to the stowing task. This section describes the "Pick Up Object" behavior, named `PickDrake`, as one of the subtasks. The `Tree` that allows the robot to perform this subtask, shown in Figure 5-6, is a subtree of the one shown in Figure 5-3 with an added Condition to make sure the robot is not moving just before picking up the object.

When ticked, the `py_trees PickDrake Behaviour` solves an IK problem to calculate a joint trajectory from the current position to a position where the gripper is slightly behind the object to pick up. Then the robot holds its position while closing

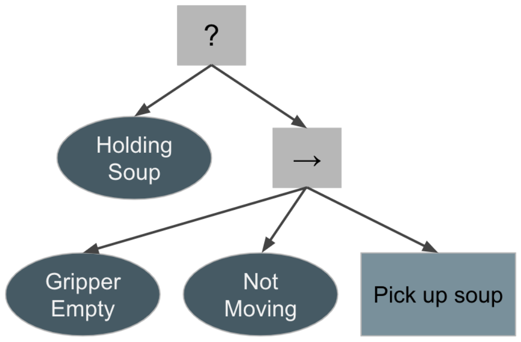


Figure 5-6: The `Tree` given to a `BehaviorTree System` used to test the `PickDrake Behaviour`.

the gripper. A video of the robot performing this task can be found at `https://youtu.be/5cm3LV2dMOo`, and the implementation can be found at `https://github.com/RobotLocomotion/6-881-examples/blob/km_thesis_work/bt_planning/behaviors.py#L105`.

Chapter 6

System Integration

While all of the `Systems` described in the previous chapters individually perform important functions for robot manipulation, they are most useful when combined with other `Systems`. This chapter describes several ways to integrate work from the previous chapters and suggests further extensions of the platform.

6.1 Connecting `Systems` Together

Any two Drake `Systems` with a compatible input/output port type pair can be wired together. Once two or more `Systems` have been wired together, they form a `Diagram`. This section shows some of the ways by which the `Systems` mentioned in the previous chapters can be connected to perform useful manipulation tasks.

6.1.1 Running a Full-Stack Manipulation Task

A `Diagram` such as the one in Figure 6-1 could be used to run a complete manipulation task, including perception, planning, and control. Each block represents a single `System` or `Diagram`, and the arrows indicate port wirings. Each color corresponds to a different type of port. The dark blue ports contain depth images, the green ports contain color images, the yellow ports contain point clouds, the light blue ports contain pose bundles, and the purple ports contain real-valued vectors. Only the names of the real-valued vector output ports are listed, as the rest are self-explanatory based on port type. Note that not all input and output ports of each `System` is displayed. For a full list of ports for each system, see their descriptions in the previous chapters.

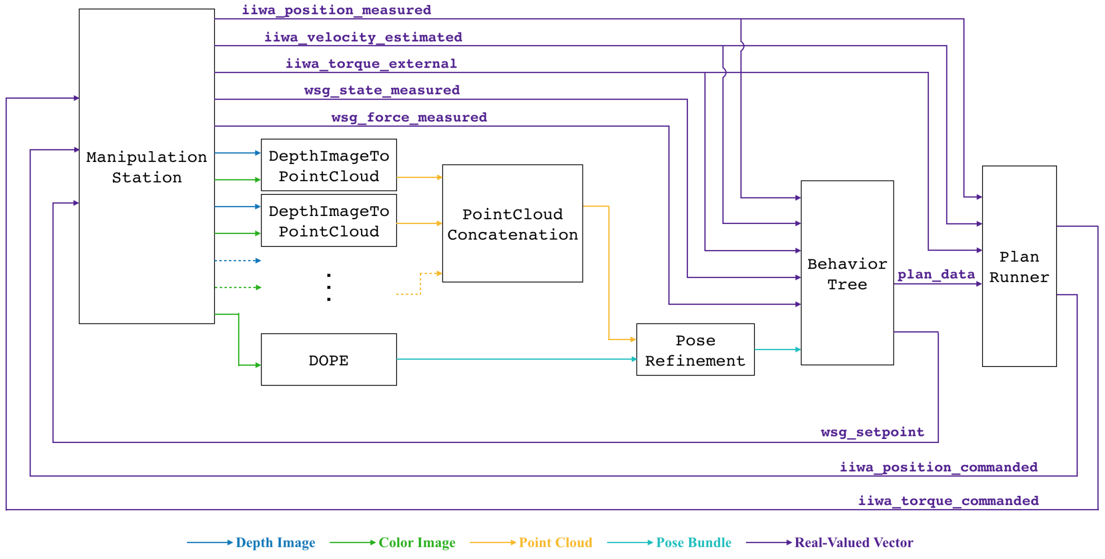


Figure 6-1: A sample `Diagram` of a full-stack robot manipulation task. Note that only the names of the real-valued vector output ports are listed.

This full-stack `Diagram` could work either fully in simulation or on real hardware, because it only uses output ports from the `ManipulationStation` that are available in both (see Figure 3-3). First, a `ManipulationStation` is created, and all of the objects are added to the scene. The depth image and color image output ports from each camera are wired to a separate `DepthImageToPointCloud`, which combines them into a colored point cloud. Separately, a single color image output port of the `ManipulationStation` is wired into a `DopeSystem`. The specific color image port is also usually used as an input to one of the `DepthImageToPointClouds`. The outputs of all of the `DepthImageToPointClouds` are connected to a single `PointCloudConcatenation`, which aligns the individual point clouds into one point cloud in world frame. That merged point cloud is then passed into a `PoseRefinement` system, along with the pose bundle output of the `DopeSystem`. Then, the refined pose information is passed into the `BehaviorTree`, along with the state of the robot and the gripper from the `ManipulationStation`. The `BehaviorTree` produces joint space plans and gripper setpoints. It sends the gripper setpoints directly to the `ManipulationStation` and the plan data to the `PlanRunner`. Finally, the `PlanRunner` sends joint position and torque commands back to the `ManipulationStation` to move the robot. MeshCat `System`s can also be inserted after output ports of the appropriate type without affecting the overall behavior.

6.1.2 Changing Modules

A big advantage of using Drake `System`s is being able to replace any individual `System` within a `Diagram` with a new `System` simply by rewiring the ports. The `Diagram` in Figure 6-1 could work for a many different tasks. Although the current `BehaviorTree` implementation is specific to the stowing task, making small changes to make it either more general or more specific to a different task would be relatively straightforward.

Similarly, replacing one of the perception `System`s, such as `DOPE` or `PoseRefinement` with a `System` containing a different algorithm with the same functionality would be simple. `PoseRefinement` is already a flexible system that lets the user substitute different point cloud segmentation and pose alignment functions without changing the underlying `System`. However, `DOPE` could be replaced with a different `System` for pose estimation, such as the Stochastic Congruent Sets algorithm described in [12]. Some work would be required to wrap the code in a Drake `System`, but an example can be seen from the modified DOPE code that is currently in the `DopeSystem`.

Testing the Perception Stack

Sub-sections of Figure 6-1 are also useful on their own. For example, to test different perception capabilities, the user can construct a `Diagram` such as the one in Figure 6-2. Unlike the one in Figure 6-1, this `Diagram` does not involve robot motion, so perception algorithms can be tested on their own with ground truth data. Figure 6-2 shows one example of explicitly inserting a `MeshcatVisualizer` into the `Diagram`. When running in simulation, the same `MeshcatVisualizer` could instead be inserted from the pose bundle output port of the `ManipulationStation` in addition to a `MeshcatContactVisualizer`. A different `MeshcatPointCloudVisualizer` could also be placed after all the point cloud output ports on either of the `DepthImageToPointClouds` and the

`PointCloudConcatenation`. This was the setup used to produce the bounding boxes in `https://youtu.be/zUS33rvbRsc`. Examples of comparing different object segmentation functions using this setup are described in Section 4.2.4.

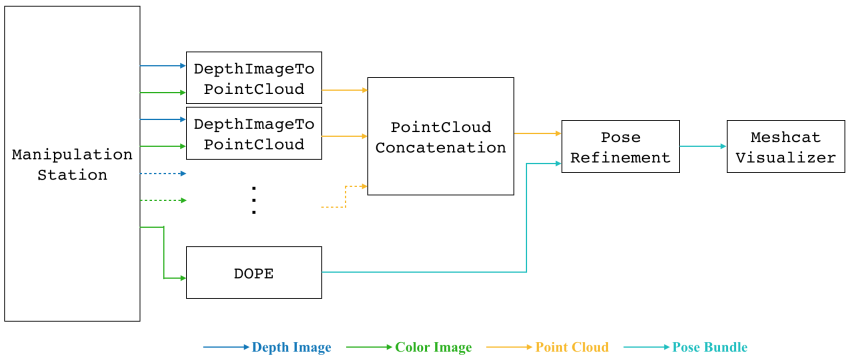


Figure 6-2: An alternative to the block diagram shown in Figure 6-1. Instead of focusing on moving the robot, this setup can be used to easily and quickly test different perception algorithms.

Testing the Planning Stack

Another useful sub-section of Figure 6-1 is shown in Figure 6-3. This `Diagram` can be used to test different planners on ground truth perception data. Note that this setup can only be used in simulation because it requires ground truth poses from the `ManipulationStation` that aren't available on the real robot. This would be the `Diagram` to use when testing a new `BehaviorTree` implementation or an alternative planner such as PDDL. Once again, any new planner would have to be wrapped in a Drake `System`, but all other parts of the `Diagram` would keep the same functionality without any extra work. This is the setup used to debug the existing `BehaviorTree` implementation as described in 5.2.2.

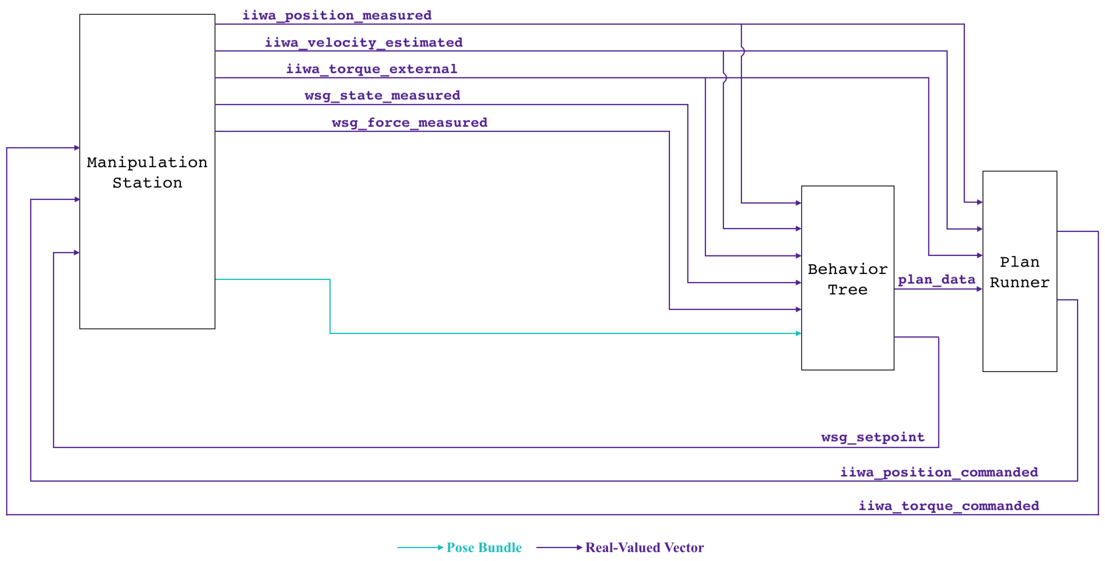


Figure 6-3: An alternative to the block diagram shown in Figure 6-1. Instead of using a perception pipeline to generate poses the behavior tree uses, the ground truth data from the simulator can be used to focus on development of only the planning portions of the problem.

6.2 Possible Extensions

Most of the current algorithms implemented across the various `Systems` that make up the Manipulation System are not the most efficient, state-of-the-art algorithms. The system is meant to be a fully functional start for other users to improve upon. Presented below are two ideas to make use of the Manipulation Station to aid both robotics researchers and students.

6.2.1 Manipulation Benchmarks

Clutter clearing is a common manipulation task where a robot must move every object in front of it into another area, often a bin to the side. A key component to any clutter clearing system is perception, but it can range from only recognizing that there is a pile in front of the robot [4] to recognizing each object to pick up individually [14]. There is also a large emphasis on task-diversity, meaning the robot should be able to pick up many different objects with different colors, shapes, sizes, weights, etc., but only needs to be able to place them in a separate bin, as opposed to a more advanced task such as operating a tool.

With many possible perception systems and objects to manipulate, it is important to have standard comparisons between algorithms to compare which ones perform better in different situations. Especially when researchers develop new methods, being able to compare their results directly against other results can help show the value of their research.

While various groups and competitions have proposed other tasks to benchmark manipulation algorithms [1] [19] [35] [33] [50], no one has proposed a full hardware and software setup to test such algorithms. The Manipulation System could be used as a benchmarking platform to test different aspects of the clutter clearing task. The Manipulation Station already has a simulation setup for this purpose, shown in Figure 3-4. The use of this common platform would allow researchers to directly compare accuracies of algorithms when running on the same hardware. Additionally, many people already use the YCB dataset in their manipulation research [25] [22] [12], which is already supported in Drake.

6.2.2 Cloud Simulation

One of the main advantages of using Drake `Systems` is how easy they are to understand and test. This makes learning about robot manipulation, which is a very broad field, easier than it would otherwise be without being able to reference specific applications of all of the pieces. Besides the complexity of learning all of the different components of manipulation, it can be difficult to learn a specific topic in depth without access to hardware to test it on.

In order to be as accessible as possible, the Manipulation System could be hosted on a cloud platform, such as Amazon Web Services (AWS) [46] or Google Cloud Platform (GCP) [20]. This would make it easy for people to use and develop manipulation algorithms without needing access to specific hardware in terms of robots or computers. Both AWS and GCP allow people to use GPUs for training deep neural networks, which could be used for perception purposes. The result could be similar to AWS RoboMaker [2], a service that was announced in late 2018 that lets people develop ROS code on AWS.

Chapter 7

Conclusion

As evidenced by the breadth of the topics covered in this thesis with only a limited amount of depth into each subject, robot manipulation is difficult. Each of the perception, planning, and control components that make up a complete manipulation task is complicated on its own, and combining them has its own unique set of challenges. While the project was motivated by cleaning a dirty kitchen, only a very small subset of that task, stowing a carton of milk back in the fridge, was covered in this document.

However, the main goal of this project was not to solve all of robot manipulation, but to set up a framework to make it easier for seasoned researchers and beginners alike to home in on their area of choice in an already working system. This was accomplished by using the Drake `Systems` framework to make a clean, modular, and deterministic system for full-stack robot manipulation. By building upon this system, people will be able to collectively program robots to clean your kitchen, possibly faster than you could.

Bibliography

[1] A. H. Quispe, H. B. Amor, and H. I. Christensen. A Taxonomy of Benchmark Tasks for Robot Manipulation. In *Robotics Research*, Springer, pp. 405-421, 2018.

[2] Amazon Web Services. Announcing AWS RoboMaker: A New Cloud Robotics Service. `https://aws.amazon.com/about-aws/whats-new/2018/11/announcing-aws-robomaker-a-new-cloud-robotics-service`, Nov. 2018.

[3] A. Shimi, A. Hurault, and P. Quèinnec. Characterizing Asynchronous Message-Passing Models Through Rounds. *arXiv eprint arXiv:1805.01657*, 2018.

[4] A. ten Pas and R. Platt. Using geometry to detect grasps in 3D point clouds. In the proceedings of the *International Symposium on Robotic Research*, 1-16, 2015.

[5] A. Zeng, S. Song, S. Welker, J. Lee, A. Rodriguez, and T. Funkhouser. Learning Synergies between Pushing and Grasping with Self-supervised Deep Reinforcement Learning. *arXiv preprint arXiv:1803.09956*, 2018.

[6] B. Calli, A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar. The YCB Object and Model Set: Towards Common Benchmarks for Manipulation Research. In the proceedings of the *IEEE International Conference on Advanced Robotics (ICAR)*, 2015.

[7] B. Calli, A. Singh, J. Bruce, A. Walsman, K. Konolige, S. Srinivasa, P. Abbeel, and A. M. Dollar. Yale-CMU-Berkeley dataset for robotic manipulation research. *The International Journal of Robotics Research*, vol. 36, Issue 3, pp. 261 - 268, April 2017.

[8] B. Calli, A. Walsman, A. Singh, S. Srinivasa, P. Abbeel, and A. M. Dollar. Benchmarking in Manipulation Research: The YCB Object and Model Set and Benchmarking Protocols. *IEEE Robotics and Automation Magazine*, pp. 36 - 52, Sept. 2015.

[9] B. Leon, S. Ulbrich, R. Diankov, G. Puche, M. Przybylski, A. Morales, T. Asfour, S. Moisio, J. Bohg, J. Kuffner et al. OpenGRASP: a toolkit for robot grasping simulation. In *Simulation Modeling and Programming for Autonomous Robots*, Springer, pp. 109-120, 2010.

[10] B. Messner, D. Tilbury, et al. Introduction: PID Controller Design. `http://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID`, Accessed May 2019.

[11] C. Betz and M. Helmert. Planning with h+ in theory and practice. *In Proc. of the ICAPS'09 Workshop on Heuristics for Domain-Independent Planning*. 2009.

[12] C. Mitash, A. Boularias, and K. Bekris. Robust 6d object pose estimation with stochastic congruent sets. *arXiv preprint arXiv:1805.06324*, 2018.

[13] C. R. Garrett, T. Lozano-Pèrez, and L. P. Kaelbling. STRIPStream: Integrating Symbolic Planners and Blackbox Samplers. *arXiv eprint arXiv:1802.08705*, 2018.

[14] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, et al. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018.

[15] D. McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2) pp. 35-55. 2000.

[16] D. Morrison, P. Corke, and J. Leitner. Closing the Loop for Robotic Grasping: A Real-time, Generative Grasp Synthesis Approach. In the proceedings of *Robotics: Science and Systems*, 2018.

[17] D. Shiferaw and A. Jain. Comparison of joint space and task space integral sliding mode controller implementations of a 6 DOF parallel robot. *Proceedings of the 11th WSEAS international conference on robotics, control and manufacturing technology*, 2011.

[18] D. Stonier. py_trees: Python implementation of behaviour trees. *GitHub*, `https://github.com/splintered-reality/py_trees/tree/release/0.6.x`. Accessed May 2019.

[19] G. Kootstra, M. Popovic, J. Alison Jorgensen, et al. Visgrab: A benchmark for vision-based grasping. In *Paladyn, Journal of Behavioral Robotics*, vol. 3, no. 2, pp. 54-62, 2012.

[20] Google LLC. Google Cloud Platform Overview. `https://cloud.google.com/docs/overview/`, 2018.

[21] J. H. Saltzer and F. Kaashoek, *Principles of computer system design: an introduction*. Burlington, MA: Morgan Kaufmann, 2009.

[22] J. Mahler, F. T. Pokorny, B. Hou, et al. Dex-net 1.0: A cloud-based network of 3d objects for robust grasp planning using a multiarmed bandit model with correlated rewards. In the proceedings of the *IEEE International Conference on Robotics and Automation (ICRA)*, 2016.

[23] J. Mahler, J. Liang, S. Niyaz, et al. Dex-Net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. In the proceedings of *Robotics: Science and Systems (RSS)*, 2017.

[24] J. Tremblay and S. Birchfield. Deep Object Pose Estimation (DOPE) - ROS inference (CoRL 2018). `https://github.com/NVlabs/Deep_Object_Pose`, 2018.

[25] J. Tremblay, T. To, B. Sundaralingam, et al. Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects. In the proceedings of the *Conference on Robot Learning (CoRL)*, 2018.

[26] K. Hauser. IROS 2016 Grasping and Manipulation Competition Simulation Framework. `https://github.com/krishauser/IROS2016ManipulationChallenge`, Aug. 2016.

[27] K. Low. Linear least-squares optimization for point-to-plane ICP surface registration. *Technical report, TR04-004*, University of North Carolina, 2004.

[28] MathWorks. Modeling Inverse Kinematics in a Robotic Arm. `https://www.mathworks.com/help/fuzzy/modeling-inverse-kinematics-in-a-robotic-arm.html`, Accessed May 2019.

[29] MathWorks. Simulink and Model-Based Design. `https://www.mathworks.com/solutions/model-based-design.html`, Accessed May 2019.

[30] M. Colledanchise, D. Almeida, and P. Ogren. Towards blended reactive planning and acting using behavior trees. *arXiv preprint arXiv:1611.00230*, 2016.

[31] M. Colledanchise and P. Ogren. Behavior trees in robotics and AI: an introduction. *CoRR, vol. abs/1709.00084*, 2017. [Online]. Available: `http://arxiv.org/abs/1709.00084`

[32] M. Gualtieri, A. ten Pas, K. Saenko, and R. Platt. High precision grasp pose detection in dense clutter. *arXiv preprint arXiv:1603.01564*, 2016.

[33] M. Menze and A. Geiger. Object Scene Flow for Autonomous Vehicles. In the proceedings of the *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[34] Open Source Robotics Foundation. ROS Concepts. `http://wiki.ros.org/ROS/Concepts`, June 2014.

[35] O. Russakovsky, J. Deng, H. Su, et al. ImageNet Large Scale Visual Recognition Challenge. In the *International Journal of Computer Vision (IJCV)*, vol. 115, issue 3, pp. 211-252, Dec. 2015.

[36] P. Besl and N. McKay. A method for registration of 3-D shapes. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 239-256, Feb. 1992.

[37] R. Deits. Meshcat-Python. `https://github.com/rdeits/meshcat-python/tree/v0.0.13`, Sept. 2018.

[38] R. E. Fikes and N. J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3-4): 189-208, 1971.

[39] R. Shenoy, B. McKay, and P. J. Mosterman. On Simulation of Simulink Models for Model-Based Design. *Handbook of Dynamic System Modeling*, CRC Press, Chapter 37, 2007.

[40] R. Tedrake and Drake Development Team. InverseKinematics Class Reference. `https://drake.mit.edu/doxygen_cxx/classdrake_1_1multibody_1_1_inverse_kinematics.html`, Accessed May 2019.

[41] R. Tedrake and Drake Development Team. ManipulationStation$<$ T $>$ Class Template Reference. `https://drake.mit.edu/doxygen_cxx/classdrake_1_1examples_1_1manipulation__station_1_1_manipulation_station.html`, Accessed May 2019.

[42] R. Tedrake and Drake Development Team. Model-based design and verification for robotics. `http://drake.mit.edu`, 2019.

[43] R. Tedrake and Drake Development Team. PiecewisePolynomial$<$ T $>$ Class Template Reference. `https://drake.mit.edu/doxygen_cxx/classdrake_1_1trajectories_1_1_piecewise_polynomial.html`, Retrieved May 2019.

[44] R. Tedrake and Drake Development Team. PiecewiseQuaternionSlerp$<$ T $>$ Class Template Reference. `https://drake.mit.edu/doxygen_cxx/classdrake_1_1trajectories_1_1_piecewise_quaternion_slerp.html`, Retrieved May 2019.

[45] R. Tedrake and T. Lozano-Perez, Intelligent Robot Manipulation. `http://manipulation.csail.mit.edu/`, Sept. 2018.

[46] S. Matthew. Overview of Amazon Web Services. *AWS Whitepaper*, April 2017.

[47] S. Ulbrich, D. Kappler, T. Asfour, N. Vahrenkamp, A. Bierbaum, M. Przybylski, et al. The OpenGRASP benchmarking suite: An environment for the comparative analysis of grasping and dexterous manipulation. In the proceedings of the *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1761-1767, 2011.

[48] T. Kluyver, B. Ragan-Kelley, F. Perez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, et al. Jupyter Notebooks - a publishing format for reproducible computational workflows. In the proceedings of the *20th International Conference on Electronic Publishing*, pp. 87-90, 2018.

[49] Y. Guo, Y. Gu, and Y. Zhang. Invariant Feature Point based ICP with the RANSAC for 3D Registration. *Information Technology Journal*, 10: 276-284, 2011.

[50] Y. Sun, J. FalcoEmail, N. Cheng, H. Ryeol Choi, E. D. Engeberg, N. Pollard, M. Roa, Z Xia. Robotic Grasping and Manipulation Competition: Task Pool. In *Communications in Computer and Information Science*, vol 816. Springer, Cham, 2016.