NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

UTILIZING ROBOT OPERATING SYSTEM (ROS) IN ROBOT VISION AND CONTROL

by

Joshua S. Lum

September 2015

Thesis Advisor: Xiaoping Yun
Co-Advisor: Zac Staples

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
UTILIZING ROBOT OPERATING SYSTEM (ROS) IN ROBOT VISION AND CONTROL

5. FUNDING NUMBERS

6. AUTHOR(S) Lum, Joshua S.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The Robot Operating System (ROS) is an open-source framework that allows robot developers to create robust software for a wide variety of robot platforms, sensors, and effectors. The study in this thesis encompassed the integration of ROS and the Microsoft Kinect for simultaneous localization and mapping and autonomous navigation on a mobile robot platform in an unknown and dynamic environment. The Microsoft Kinect was utilized for this thesis due to its relatively low cost and similar capabilities to laser range scanners. The Microsoft Kinect produced three-dimensional point-cloud data of the surrounding environment within the field-of-view. The point-cloud data was then converted to mimic a laser scan. The odometry data from the mobile robot platform and the converted laser scan were utilized by a ROS package for simultaneous localization and mapping. Once self-localization and mapping were achieved, a ROS navigation package was utilized to generate a global and local plan, which translated to motor velocities in order to move the robot to its objective. The results demonstrated that simultaneous localization and mapping and autonomous navigation can be achieved through the integration of ROS and the Microsoft Kinect.

14. SUBJECT TERMS Robotics, mobile robots, Microsoft Kinect, Pioneer P3-DX, ROS, SLAM, autonomous navigation			**15. NUMBER OF PAGES** 91
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

UTILIZING ROBOT OPERATING SYSTEM (ROS) IN ROBOT VISION AND CONTROL

Joshua S. Lum
Captain, United States Marine Corps
B.S., U.S. Naval Academy, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Author: Joshua S. Lum

Approved by: Xiaoping Yun
Thesis Advisor

Zac Staples
Co-Advisor

R. Clark Robertson
Chair, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Robot Operating System (ROS) is an open-source framework that allows robot developers to create robust software for a wide variety of robot platforms, sensors, and effectors. The study in this thesis encompassed the integration of ROS and the Microsoft Kinect for simultaneous localization and mapping and autonomous navigation on a mobile robot platform in an unknown and dynamic environment. The Microsoft Kinect was utilized for this thesis due to its relatively low cost and similar capabilities to laser range scanners. The Microsoft Kinect produced three-dimensional point-cloud data of the surrounding environment within the field-of-view. The point-cloud data was then converted to mimic a laser scan. The odometry data from the mobile robot platform and the converted laser scan were utilized by a ROS package for simultaneous localization and mapping. Once self-localization and mapping were achieved, a ROS navigation package was utilized to generate a global and local plan, which translated to motor velocities in order to move the robot to its objective. The results demonstrated that simultaneous localization and mapping and autonomous navigation can be achieved through the integration of ROS and the Microsoft Kinect.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application program interface
CMOS	Complementary metal–oxide–semiconductor
GPS	Global positioning system
IP	Internet protocol
RGB	Red, green, blue
ROS	Robot Operating System
SLAM	Simultaneous localization and mapping
SSH	Secure shell
URDF	Unified robot description format
XML	Extensible markup language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

First and foremost, I would like to thank my wife, Katelynn, for putting up with all the late nights and early mornings, all while carrying and raising our son. No one can ever question your strength, will, and patience for having to put up with me, and I thank God for your support and encouragement. Next, I would like to recognize my thesis advisors, Professor Xiaoping Yun and Commander Zac Staples. Professor Yun, you have taught me so much about robotics and have helped develop a stronger desire to continue to pursue research in the field. Commander Staples, thank you for introducing me to ROS and guiding me in my first tentative steps in the Linux world. I never thought I would say this, but I actually feel more comfortable with the black screen and white letters now. I would also like to thank James Calusdian for his tireless efforts in supporting me and all the students in the laboratory, Jeremy Dawkins for his aid in ROS networking, and Bill Stanton for providing lab support from the United States Naval Academy. Additionally, I would like to thank my thesis processor, Michele D'Ambrosio, for tediously reviewing this thesis. Finally, I thank all those who provided assistance from the ROS Answers community.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION FOR RESEARCH

Autonomous navigation is considered to be one of the most challenging competences of mobile robotics and requires four main components: perception, localization, path planning, and motion control. For a robot to navigate, it must be able to sense its environment in order to detect obstacles and a clear path to its goal. It must also be able to identify or estimate its location relative to its objective and any obstacles that may exist in its surroundings. While a robot may know its location in relation to its goal, it must have the ability to determine a feasible and optimal path from its current location to the desired goal while avoiding obstacles. Finally, the robot must have the means to maneuver to its goal.

Within robotics, simultaneous localization and mapping (SLAM) is the problem of using sensors to construct a map of an unknown environment while simultaneously keeping track of its own location. It is a two-pronged issue, made complex because the two parts of the problem rely on each other. In order to construct a map of its environment, a robot must have a good estimation of its location. For self-localization, a robot must have a relatively accurate map. The problem is further complicated in a dynamic environment in which moving people or objects may exist. SLAM can be simplified using a global positioning system (GPS); however, GPS is typically not feasible indoors or the robot may be in an environment where GPS services have been denied.

In order to conduct both autonomous navigation and SLAM, a robot must have a capable sensor suite and the ability to process the data from the sensor, estimate its location, construct a map, identify an optimal and feasible path from its current location to the goal, and maneuver to the objective. This requires a robust software framework to allow the robot platform, controllers, and sensors to work harmoniously in order to achieve the objective.

As humans, we have the natural ability to connect with our world through our senses and to immediately process the information to make decisions almost subconsciously. With the use of binocular vision, we are able to identify objects, determine distances to obstacles, and mentally locate ourselves within a room. Even if the lights of the room are suddenly turned off, we are able to cross the dark room while avoiding the obstacles that were once visible because we are able to estimate our position by the length of our strides and the natural accelerometer within our inner ear. Our brain links all of our senses together and allows us to function more effectively and in a more versatile way than any machine; however, as technology improves, machines and robots draw ever closer to humans in their ability to perform tasks. While each simple task that we perform is a complex and daunting task for a robot, research and development teams are constantly working to improve the field of study to close the gap between robots and humans.

Many consider the brain and the eye to be the most complex parts of the human body. They allow humans the best methods to perceive their environment. In robotics, there are many different sensors that allow the robot to sense its surroundings for the purpose of navigation. The most widely used sensor, because of its accuracy and ability to be used at short and long distances, is the laser-range scanner. Another type of sensor that is widely used, because of the relatively low cost, is the sonar-range finder. The limiting factor between these two popular range sensors is high cost or low accuracy. With the development of the Microsoft Kinect and the subsequent release of its software development kit, it has become one of the chief sensors for robotic researchers and developers due to its relative low cost and the capabilities and accuracy of its depth camera.

Robot Operating System (ROS) is becoming a widely popular method for writing robot software, primarily because of its flexibility, robustness, and modularity. Additionally, ROS is completely open-source, creating an environment in which the spread of knowledge and learning is prevalent within the robotics community. Because of these qualities, each modular part of ROS has a plug-and-play feel, allowing users,

developers, and researchers to pick whichever packages are best for their robots and the ability to configure them in a simple manner.

B. BACKGROUND

The topic of mobile robotics has recently become even more widespread and popular, especially with the progress of technology and the increasing availability of robot platforms and new robotic software architecture. Many autonomous and semi-autonomous robots are being developed for specific purposes. Some examples of the uses of mobile robots are space and sea exploration, elderly or disabled person assistance, janitorial services, manufacturing, and even autonomous cars that operate on roadways with human-driven cars and pedestrians.

Typically, when robots are designed, software developers and programmers must write programs and code that is specifically designed for that particular robot. Because different robot designs typically contain different controlling software, robot developers must write diverse programs to meet the needs of each robot design. These programs can often be used only by that robot design and are not modular in nature. Creating modularity in terms of hardware is quite a simple task, but designing modularity in software can be extremely difficult [1].

One particular area of importance for employing a mobile robot that can conduct SLAM and autonomous navigation is the sensor systems utilized to gather information about the robot's surroundings. Popular range sensors include the laser range finder, the laser range scanner, and the sonar array. The Microsoft Kinect offers an infrared depth sensor, which offers a cheap yet relatively accurate solution for a robot to sense its environment.

This thesis stems from a thesis [2] completed within the Naval Postgraduate School's Electrical and Computer Engineering Department in which the capabilities of the Microsoft Kinect and its ability to detect thin or narrow obstacles, which were undetectable by the sonar-range sensors of the Pioneer P3-DX mobile robot platform, were investigated. An algorithm to process and analyze the point-cloud data from the Microsoft Kinect was presented, and the point-cloud data was transformed into a two-

dimensional map of the local environment in order to conduct obstacle avoidance. MATLAB was utilized to process the captured point-cloud data, conduct obstacle avoidance, and control the Pioneer P3-DX mobile robot [2].

In this thesis, we seek to investigate further the capabilities of the Microsoft Kinect in conducting SLAM and autonomous navigation when integrated with the robust and flexible software framework that ROS provides.

C. PURPOSE AND ORGANIZATION OF THESIS

The purpose of this thesis is to investigate the feasibility of the integration of ROS and the Microsoft Kinect on a mobile robot platform for SLAM and autonomous navigation without the use of a GPS or simulated indoor GPS. This thesis is divided into five chapters. An explanation of ROS, the Microsoft Kinect, and the Pioneer P3-DX mobile robot are provided in Chapter II. The integration of ROS software with the Microsoft Kinect and Pioneer P3-DX, as well as the approaches used for SLAM and autonomous navigation, are discussed in Chapter III. The focus of Chapter IV is the results of experimentation and the effectiveness of the integration of ROS packages and the Microsoft Kinect. A conclusion and a discussion of future work which can be developed from this project are provided in Chapter V.

II. DESIGN

The focus of Chapter II is the descriptions of the software and hardware utilized in this thesis. Within the chapter, a detailed explanation of ROS, the Pioneer P3-DX mobile robot platform, and the Microsoft Kinect depth sensor can be found.

A. ROBOT OPERATING SYSTEM

ROS is a Linux-based, open-source, middleware framework for modular use in robot applications. ROS, originally designed by Willow Garage and currently maintained by the Open Source Robotics Foundation, is a powerful tool because it utilizes object-oriented programming, a method of programming organized around data rather than procedures in its interaction with data and communication within a modular system [3]. ROS is divided into three conceptual levels: the filesystem level, the computation graph level, and the community level.

1. Filesystem Level

The filesystem level is the organization of the ROS framework on a machine. At the heart of the ROS's organization of software is the package. A package may contain ROS runtime execution programs, which are called nodes, a ROS-independent library, datasets, configuration files, third-party software, or any software that should be organized together [4]. The goal of the packages is to provide easy to use functionality in a well-organized manner so that software may be reused for many different projects. This organization, along with object-oriented programming, allows packages to act as modular building blocks, working harmoniously together to accomplish the desired end-state. Packages typically follow a common structure and usually contain the following elements: package manifests, message types, service types, headers, executable scripts, a build file, and runtime processes [4]. Package manifests provide metadata about a package, such as the name, author, version, description, license information, and dependencies. Packages may also contain message types, which define the structure of data for messages sent within ROS, and service types, which define the request and response data structures for services. Also within the filesystem level are repositories,

which are a collection of packages sharing a common version control system. Both packages and repositories help make ROS a modular system.

2. Computation Graph Level

The computation graph level is where ROS processes data within a peer-to-peer network. The basic elements of ROS's computation graph level are nodes, messages, topics, services, bags, Master, and Parameter Server. Nodes are the small-scale workhorses of ROS, subscribing to topics to receive information, performing computations, controlling sensors and actuators, and publishing data to topics for other nodes to use [5]. The `rosnode` tool is a useful command-line tool for displaying information about ROS nodes. The command, `rosnode list`, displays all active nodes running on the ROS Master. A package may have many nodes within it to accomplish a group of computations and tasks, in which they all communicate with each other through topics and services via messages.

The primary method in which nodes pass data to each other is by publishing messages to topics. A message is simply a structuring of data so it is in a useful, standard format for other nodes to use. Standard types, such as integer, floating point, and Boolean, are supported as well as arrays. A standard message utilized in this thesis is the `sensor_msgs/PointCloud2` [6], which can be found in Figure 1. The command `rosmsg list` prints all messages available to the ROS Master. The key to the modularity of ROS is the method in which nodes typically communicate with each other through topics.

```
std_msgs/Header header                # Time stamp and frame ID of message.
unit32 height                         # Pixel height of image.
unit32 width                          # Pixel length of image.
sensor_msgs/PointField[] fields       # Describes the channels and their layout.
bool is_bigendian                     # Is data big endian?
uint32 point_step                     # Length of point in bytes.
uint32 row_step                       # Length of ros in bytes.
uint[] data                           # Actual point data (size is row_step*height).
bool is_dense                         # True if no invalid points.
```

Figure 1. Message format utilized in PointCloud2 message.

Rather than communicating directly with each other, nodes usually communicate through topics. Topics are named hubs in which nodes can publish and subscribe and are the crux of what makes ROS an object-oriented and modular environment [7]. Nodes that generate data are only interested in publishing that data, in the correct message format, to the correct topic [7]. Nodes that require data simply subscribe to the topics of interest to pull the required information. Multiple nodes may publish or subscribe to a single topic as shown in Figure 2. This method of publishing and subscribing to topics decouples the production of information from the consumption of information. It allows nodes within different packages to work harmoniously with each other even though they may have different origins and functions. The `rostopic` command-line tool is useful for displaying debugging information about ROS topics. To display all active topics, the command `rostopic list` is utilized. The command `rostopic info` *`<topic_name>`* prints the message type accepted by the topic and publishing and subscribing nodes. Another useful command-line tool is `rostopic echo` *`<topic_name>`*, which prints messages published to a topic. The commands `rostopic hz` *`<topic_name>`* and `rostopic bw` *`<topic_name>`* displays the publishing rate and the bandwidth used by a topic, respectively. Additionally, data can be manually published to a topic by using the `rostopic pub` *`<topic_name>`* command.

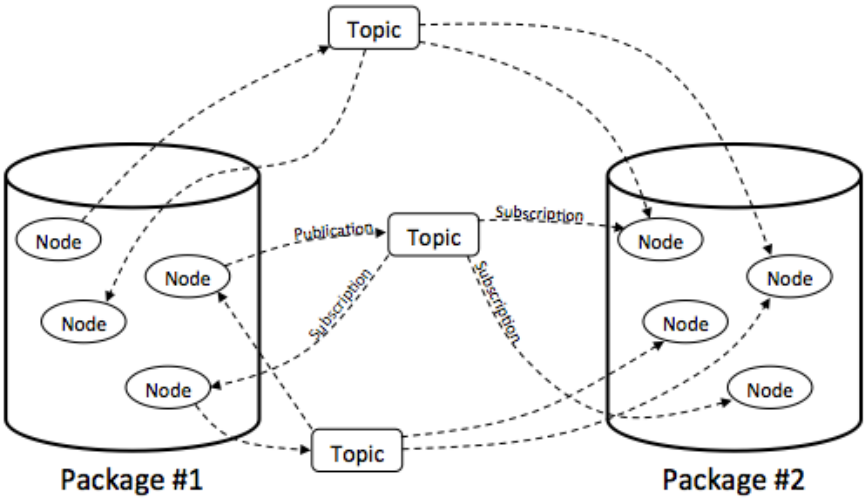

Figure 2. Model of how the ROS nodes publish and subscribe to topics.

In addition to publishing messages to topics, nodes can also exchange a request and response message as part of a ROS service. This is useful if the publish and subscribe (many-to-many) communication method is not appropriate, such as a remote procedure call. A ROS node that provides data offers a service under a string name, and a client node that requires data calls the service by sending the request message and awaiting the response [8]. Active services can be displayed by utilizing the command `rosservice list`, and information about a service can be found by using `rosservice info` *`<service_name>`*.

Bags are a method for recording and storing ROS message data. This is a powerful tool that allows users to store, process, analyze, and visualize the flow of messages. Bags are created utilizing the `rosbag` tool, which subscribes to one or more ROS topics and stores message data as they are received. This stored data can be replayed in ROS to the same topics, as if the original nodes were sending the messages. This tool is useful for conducting experiments using a controlled set of data streams to test different algorithms, sensors, actuators, and controllers. To record data, the command `rosbag record` *`<topic_names>`* should be used. To view information about a `bagfile` already created, the command `rosbag info` *`<bag_file>`* should be utilized. The command `rosbag play` *`<bag_file>`* can be used to publish messages from topics just as if they were being played for the first time. When rosbag is utilized to play data, the time synchronization is based on the global timestamp when the `bagfile` was recorded. It is recommended that when playing back data using `rosbag play` to use `rosparam set sim_time true` and `rosbag play` *`<bag_file>`* `--clock` in order to run the recorded system with simulated timestamps.

A launch file is method of launching multiple ROS nodes, either locally or remotely, as well as establishing parameters on the ROS Parameter Server. It is useful for running large projects, which may have many packages, nodes, libraries, parameters, and even other launch files, which all can be started via one launch file rather than individually running each node separately. The `roslaunch` tool uses extensible markup language (XML) files that describe the nodes that should be run, parameters that should be set, and other attributes of launching a collection of ROS nodes [9]. The roslaunch tool

is utilized by using the command `roslaunch <package_name> <file.launch>`. Examples of a roslaunch XML file can be found in Appendices A and B.

The ROS Master acts as a domain name system server, storing topic's and service's registration information for ROS nodes. ROS Master provides an application program interface (API), a set of routines and protocols, tracking services and publishers and subscribers to topics. A node notifies ROS Master if it wants to publish a message to a topic. When another node notifies the master that it wants to subscribe to the same topic, the master notifies both nodes that the topic is ready for publishing and subscribing. The master also makes callbacks to nodes already online, which allows nodes to dynamically create connections as new nodes are run [10]. The ROS Master is started with the command `roscore` and must be used to run nodes in ROS. The ROS Master also provides the Parameter Server. The ROS Parameter Server can store integers, floats, Boolean, dictionaries, and lists and is meant to be globally viewable for non-binary data [11]. The parameter server is useful for storing global variables such as the configuration parameters of the physical characteristics of a robot. ROS parameters can be displayed by utilizing the command `rosparam list`. A user can also set a parameter from the command line by using `rosparam set <parameter_name> <parameter_value>`. Parameters can also be loaded from a `.yaml` file by using the command `rosparam load <parameters.yaml>`.

An example of how a node is used in this thesis is `openni_camera`, which is the driver for the Microsoft Kinect. The node runs the Microsoft Kinect, extracts data, and publishes the captured data via messages such as `sensor_msgs/CameraInfo` and `sensor_msgs/PointCloud2` to various topics such as `rgb/camera_raw`, `depth/image_raw`, and `ir/image_raw`. Then other nodes, such as `openni_tracker`, subscribe to those topics and conducts processes and computations.

Names have an important role within ROS. Every node, topic, service, and parameter has a unique name. This architecture allows for decoupled operation that allows large, complex systems to be built. ROS supports command-line remapping of names, which means a compiled program may be reconfigured at runtime to operate in a

different computation graph topology [12]. This means that the same node can be run multiple times, publishing difference messages to separate topics.

3. Community Level

The ROS Community Level consists of ROS distributions, repositories, the ROS Wiki, and ROS Answers, which enable researchers, hobbyists, and industries to exchange software, ideas, and knowledge in order to progress robotics communities worldwide. ROS distributions are similar to the roles that Linux distributions play. They are a collection of versioned ROS stacks, which allow users to utilize different versions of ROS software frameworks. Even while ROS continues to be updated, users can maintain their projects with older more stable versions and can easily switch between versions at any time.

ROS does not maintain a single repository for ROS packages; rather, ROS encourages users and developers to host their own repositories for packages that they have used or created. ROS simply provides an index of packages, allowing developers to maintain ownership and control over their software. Developers can then utilize the ROS Wiki to advertise and create tutorials to demonstrate the use and functionality of their packages. The ROS Wiki is the forum for documenting information about ROS, where researchers and developers contribute documentation, updates, links to their repositories, and tutorials for any open-sourced software they have produced. ROS Answers is a community-oriented site to help answer ROS-related questions that users may have.

4. Other ROS Concepts

a. Unified Robot Description Format

The unified robot description format (URDF) package contains an XML file that represents a robot model. The URDF is another tool within ROS that makes it a modular system. Rather than creating a unique process for different styles of robots, nodes are created without regard for the robot that will utilize them. The URDF file provides the necessary, robot-specific, information so nodes may conduct their procedures. A URDF file is written so that each link of the robot is the child of a parent link, with joints

connecting each link, and joints are defined with their offset from the reference frame of the parent link and their axis of rotation [13]. In this way, a complete kinematic model of the robot is created. A tree diagram can be visualized utilizing the `urdf_to_graphiz` tool as is shown in Figure 3. The URDF can be parsed and checked by utilizing the commands shown in Figure 4.

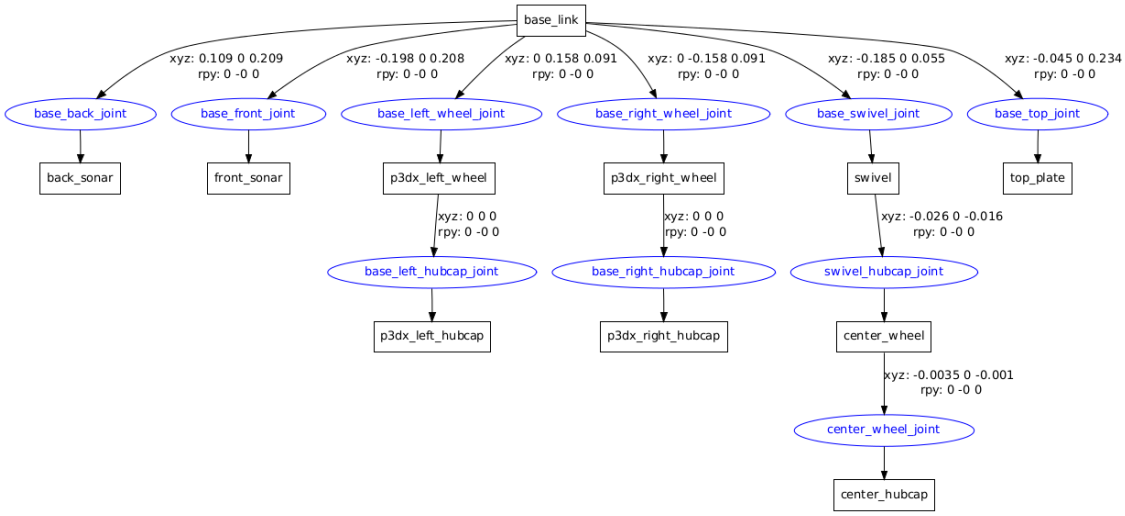

Figure 3. A diagram of URDF of Pioneer P3-DX is shown utilizing the `urdf_to_graphiz` tool.

```
# Create the urdf from .xacro file
$ rosrun xacro xacro.py ~/<filepath>/<file_name.xacro> -o ~/<filepath>/<file_name.urdf>

# Attempt to parse urdf file and, if successful, print description of kinematic chain
$ check_urdf <file_name.urdf>

robot name is: pioneer3dx
---------- Successfully Parsed XML ----------
root Link: base_link has 8 child(ren)
    child(1):  back_sonar
    child(2):  front_sonar
    child(3):  swivel
        child(1):  center_wheel
            child(1) center_hubcap
    child(4):  top_plate
    child(5):  left_hub
        child(1):  left_wheel_joint
    child(6):  right_hub
        child(1): right_wheel_joint
```

Figure 4. Commands used to create, parse, and check a URDF file.

b. Coordinate Frames and Transforms.

A robotic system typically has many three-dimensional coordinate frames that change over time. The `tf` ROS package keeps track of multiple coordinate frames in the form of a tree structure. Just as the URDF manages joints and links, the `tf` package maintains the relationships between coordinate frames of points, vectors, and poses, and computes the transforms between them. The `tf` package operates in a distributed system; all ROS components within the system have access to information about the coordinate frames. The transform tree can also be viewed by developers for debugging by utilizing the `view_frames` tool as shown in Figure 5. Additional command-line tools for the `tf` package are `rosrun tf tf_monitor`, `rosrun tf tf_echo` *`<source_frame> <target_frame>`*, and `roswtf`, which, respectively, monitors delays between transforms of coordinate frames, prints transforms between coordinate frames, and aids in debugging [14].

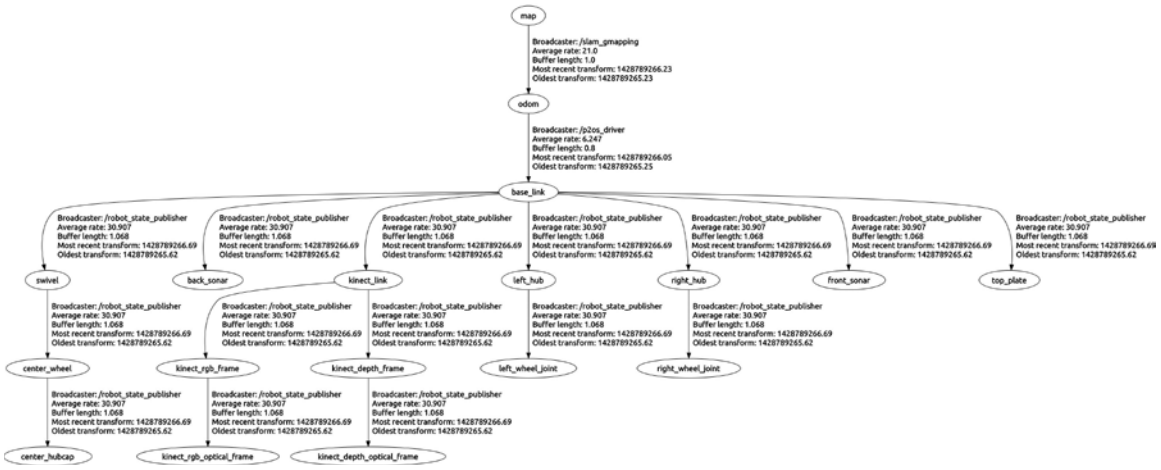

Figure 5. Tree diagram of transforms used by navigating Pioneer P3-DX is shown utilizing the `view_frames` tool.

c. Visualization

The `Rviz` package, developed by Willow Garage, comes standard with ROS and is a powerful visualization tool that allows users to visualize data from sensors, maps, and a robot's internal state [15]. Used to its fullest capacity, it can allow users to view what the

robot is seeing, thinking, and doing. `Rviz` subscribes to sensor topics such as `/LaserScan`, `/PointCloud2`, and `/Camera` as well as topics such as `/tf` and `/map`. Additionally, a URDF file can be utilized to visualize a robot in three-dimensional space.

5. Basic ROS Commands

ROS provides users with a variety of tools in order to make navigation through the ROS filesystem and debugging as simple as possible. A few basic ROS commands utilized within this thesis are shown in Table 1.

Table 1. A few basic ROS commands.

roscore	Starts ROS Master.
rosrun <pkg_name> <node_name>	Starts executable node.
roslaunch <pkg_name> <launch_file>	Starts launch file.
rostopic list	Lists all active topics.
rostopic info </topic_name>	Provides data on topic such as type, subscribers and publishers.
rostopic echo </topic_name>	Prints topic messages to screen.
rostopic hz </topic_name>	Prints publishing rate to screen.
rosnode list	Lists all nodes running.
rosnode info <node_name>	Provides data on node such as publications, subcriptions, services, and Pid.
Rosmsg show –r <msg_type>	Prints raw message text.
rospack find <package_name>	Prints file path to package.
rosrun rqt_graph rqt_graph	Tool to visualize graphical representation of active packages, nodes, and topics.
rosrun rviz rviz	Starts ROS visualization tool.
Rosbag record –O <filename> </topic>	Starts rosbag tool to record data from a desired topic.

B. HARDWARE

The ground, mobile robot platform utilized for this thesis was the Pioneer P3-DX designed by Adept MobileRobots. Mounted onto the P3-DX were the Microsoft Kinect and the computer processing unit.

1. Pioneer P3-DX

The *Pioneer 3 Operations Manual* [16] states that the P3-DX is small, measuring 45.5 cm in length, 38.1 cm in width, 23.7 cm in height, and weighs only 9 kg. It has two differential drive wheels and a small, free-rotating castor wheel, making it capable of completing a zero radius turn. It is capable of traveling forward and backward on level ground at a speed of 1.5 meters per second and has a rotation speed of 300 degrees per second. Its power supply consists of one to three 12 V DC, sealed lead/acid batteries, which gives it a maximum run time of four to six hours [16]. The P3-DX comes standard with two sensor arrays, one oriented to the front of the robot and the other oriented to the rear, each with eight sonar sensors. The sonar sensors provide 360° range data to a maximum of five meters utilizing time of flight computations. Each sonar transducer produces a sound wave in sequence at a rate of 25 Hz for each array [16]. Range data is determined from the amount of time it takes for the emitted sound wave to travel from the sonar sensor, reflect off an object, and return to the sonar sensor. Additionally, the P3-DX has two segmented bumper arrays, which sense if the robot has come into contact with an obstacle.

2. The Microsoft Kinect

The Kinect sensor, developed by Microsoft and PrimeSense, is a natural interaction device that allows for a more natural connection between humans and computers by capturing three-dimensional data of its environment and body movements of humans. In June 2011, Microsoft released the software development kit for non-commercial use. The Open Natural Interaction (OpenNI) framework, which focused on improving interoperability of natural user interfaces, provided open-sourced APIs that allowed public access to natural interaction devices such as the Microsoft Kinect [17].

Due to its relatively low price and the ability to utilize open-source software to interact with the sensor, the Kinect is a viable alternative to other depth-finding sensors such as laser-range scanners and sonar sensors. Comparable laser-range scanners can cost more than $2,000 and often only produce a two-dimensional “slice” of the environment. More expensive alternatives can offer three-dimensional scans but cost as much as $5,500. A few of the advantages that laser range scanners offer are accuracy and range. Many laser range finders grant an accuracy of $\pm$30-40 mm with maximum depth sensing capabilities ranging from 25 m to 100 m. Sonar sensors offer a cheaper alternative, with prices between $25 and $50, but only offer range data for a single vector. In order to capture even a very limited three-dimensional view of its environment, a robot has to use an array of many sonar sensors [18].

According to [19], the Microsoft Kinect is composed of a red-green-blue (RGB) camera and a depth sensor. The RGB camera captures color images of its environment. In [19] it further explains that the depth sensor is composed of an infrared laser projector and a complementary metal–oxide–semiconductor (CMOS) sensor. The infrared source projects an infrared light-coded image into the scene, and the CMOS sensor captures the image of the reflections of the coded, infrared, laser speckle, the deformation of the coded pattern of infrared light from objects. Additionally, the CMOS sensor and infrared source are separated laterally on the sensor by 7.5 cm, which allows for stereo triangulation, which is similar to the binocular vision that humans and many animals use to determine depth. Through the returned light-coded image and stereovision, the processor computes the position and depth through statistical analysis, producing an array of voxels, a three-dimensional point cloud of the scene. Both the color video-stream and the point cloud are captured at a frame rate of 30 Hz.

One side effect of utilizing stereo triangulation is known as the parallax effect, the difference in the apparent position of an object due to varying lines-of-sight. An example of this is the shift in view when one focuses on an object and alternately covers one eye then the other. The view of the object shifts slightly due to a change in the line-of-sight of the object from one eye to the other. With the Microsoft Kinect, the parallax effect causes shadow-like areas of non-returns around objects as seen in Figure 6, since the coded

infrared light cannot be captured if blocked by an object. As the distance of the object from the viewer increases, the effect of parallax decreases.

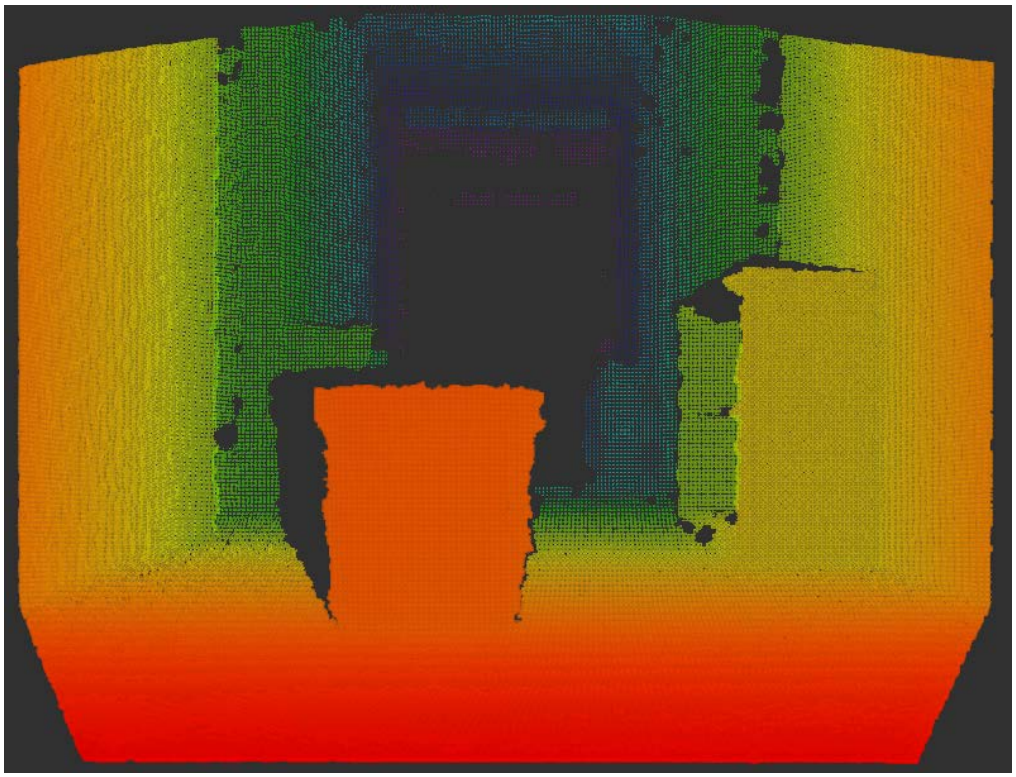

Figure 6. Image captured by the Microsoft Kinect depth camera showing an example of the parallax effect. The gray "shadows" are created by objects blocking the coded infrared projection from the offset projector from being captured by the CMOS sensor and results in a non-return.

The Kinect produces a VGA-sized (640 pixels × 480 pixels) resolution and has a 57° horizontal and 43° vertical field-of-view. The depth sensor is able to provide a depth *z* resolution of ±1.0 cm and an *x*/*y* resolution of ±0.3 cm and operates optimally between 0.8 m and 3.5 m, although it can operate at a maximum range of approximately 6.0 m [20].

In addition to capturing a three-dimensional point cloud of its environment, the Microsoft Kinect is capable of tracking human body positions within the field-of-view. Microsoft, through the study of more than 15 body-types and hundreds of thousands of body positions, utilized random decision-making trees, probability distributions, and machine

learning to teach the Kinect to recognize the human form. Because of this, the Kinect is able to infer body position even if a person's body is partially hidden from view.

3. Computer Processing Units

A SlimPro 675FP fanless mini-computer, running the Linux kernel, Ubuntu Precise Pangolin 12.04, was mounted to the top of the Pioneer P3-DX and was connected to the robot through its serial port. The processing unit was used to run the ROS packages required for the Pioneer P3-DX, the Microsoft Kinect, SLAM, and autonomous navigation. It also communicated wirelessly, through a wireless access point, to an ASUSPRO Advanced Notebook, which also ran ROS on Ubuntu 12.04. The base laptop was utilized to remotely launch the ROS packages on the robot's processing unit, check the robot's real-time diagnostics, visualize maps built through simultaneous localization and mapping (SLAM) as the robot autonomously navigated its environment, and, if necessary, control the robot through keyboard inputs. The Pioneer P3-DX, with forward-mounted Microsoft Kinect and SlimPro processing unit, is shown in Figure 7.

Figure 7. The Pioneer P3-DX with mounted SlimPro mini-computer and Microsoft Kinect depth sensor.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SYSTEM DEVELOPMENT AND INTEGRATION

The process of configuring the Pioneer P3-DX with Microsoft Kinect into a system capable of conducting SLAM and autonomous navigation can be divided into four parts. The first is the installation and configuration of ROS onto the robot's processing unit as well as the base laptop, to include establishing a wireless network connection between the two Ubuntu 12.04 machines and ensure ROS is properly communicating. The second is to install and configure the ROS driver nodes for the Pioneer P3-DX. The third is to install and configure the ROS driver nodes for the Microsoft Kinect. The fourth is to install, configure, and test the ROS navigation stack, to include the SLAM packages and navigation control packages.

A. INSTALLING AND CONFIGURING ROS

ROS is supported on Ubuntu and experimentally on OS X, Arch Linux, and Debian Wheezy. For this thesis, Ubuntu 12.04, Precise Pangolin, was utilized. Before installing ROS, Ubuntu must be properly configured to accept the four types of repository components: main, officially supported software; restricted, supported software not available under a completely free license; universe, community maintained software; and multiverse, software that is not free. This is done from the Software Sources interface, which can be accessed through the Ubuntu Software Center.

Next, the appropriate ROS keys must be downloaded and the Debian package index updated. Once the keys have been downloaded, ROS can be installed. For this thesis, ROS Hydro was installed. The full ROS desktop install downloads all packages and libraries. This can be accomplished by running the commands found in Figure 8.

```
# Set up ROS keys
$ wget https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -O | sudo apt-key add -

# Update Debian package index
$ sudo apt-get update

# Full ROS desktop install
$ sudo apt-get install ros-<ros_distro>-desktop-full

# Initialize rosdep
$ sudo rosdep init
$ rosdep update

# Add ROS variable to .bashrc
$ echo "source /opt/ros/<ros_distro>/setup.bash" >> ~/.bashrc
$ source ~/.bashrc

# Download rosinstall command-line tool
Sudo apt-get install python-rosinstall
```

Figure 8. Commands used to download ROS. Replace *<ros_distro>* with appropriate ROS distribution, for example: `hydro`.

In order to allow the robot's processing unit to communicate with the base laptop for remote launch operations and to visualize the robot's operations through Rviz, the two computers must be properly configured on a wireless network adaptor. In order to establish a secure, encrypted connection, `openssh-server` and `openssh-client` must be downloaded on the robot's processing unit and the base laptop. The appropriate commands to download the secure shell protocol are shown in Figure 9.

```
# Download secure shell protocol
$ sudo apt-get install openssh-client
$ sudo apt-get install openssh-server

# Add IP address, associated hostname, and remote machine to /etc/hosts configuration file.
$ ifconfig      # Note IP address on wireless access point
$ cd /etc/
$ nano hosts

# Start a secure shell session
```

Figure 9. Command utilized to download the secure shell protocol in order to initiate a secure shell session on a remote machine.

Once the `openssh-server` and `openssh-client` have been downloaded, it is necessary to add each computer's IP address on the wireless access point, create a hostname, and add the remote machine's information to the `/etc/hosts` configuration

file. Utilizing the command `ifconfig` in the command-line of the terminal will display the IP address that each machine has been given on the wireless access point. Once the IP address has been identified, it is necessary to add that IP address and a hostname, as well as the IP address and hostname of the remote machine to the robot's processing unit and the base laptop. This can be accomplished by editing the `/etc/hosts` configuration file and adding the appropriate data as seen in Figure 10. Once the configuration file has been edited, a secure shell session can be started on the remote machine by utilizing the command `ssh remote_machine@hostname`. Once accomplished, programs may be created, edited, and run remotely.

```
# Add IP address, associated hostname alias to /etc/hosts configuration file
$ ifconfig      # Note IP address on wireless access point
$ cd /etc/
$ sudo nano hosts

# Edit /etc/hosts on base_laptop as shown:
127.0.0.1       localhost
<ip_address>    base@base_laptop
<robot_ip>      SlimPro@p3dx            p3dx

# Edit /etc/hosts on robot as shown:
127.0.0.1       localhost
<ip_address>    p3dx
<base_ip>       base@base_laptop        base_laptop
```

Figure 10. Procedure to edit `/etc/hosts` configuration file to initiate a secure shell session from a remote machine.

Once each of the machines have been properly configured, it is necessary to ensure ROS is properly configured to operate on multiple machines so all machines can see all topics in real-time. It is of note that only one ROS Master is necessary to be running, even across multiple machines. Let it be assumed that the hostname for the base laptop is `base` with the alias `base_laptop` and IP address of `192.168.0.100`, and the hostname for the machine onboard the P3-DX is `SlimPro` with the alias `p3dx` and IP address of `192.168.0.101`. It is desired that the ROS Master be run on the base laptop. To test the connection between the two computers and ensure ROS is properly communicating across them, we use the nodes `talker` and `listener` from the `rospy_tutorial` package, two standard packages within the ROS installation. From

the base laptop, the ROS Master is run by utilizing the command `roscore`. Next, it is necessary to check the `ROS_MASTER_URI`, which informs nodes where to find the master. This can be accomplished by using the command `export ROS_MASTER_URI`. Let it be assumed the `ROS_MASTER_URI` is `http://192.168.0.100:11311`. Next, on the base laptop, the node `listener.py` is run by utilizing the command `rosrun rospy_tutorials listener.py`. Next, on the Pioneer's processing unit, the ROS_MASTER_URI is configured to match the master that was run on the base laptop by using the following command: `export ROS_MASTER_URI= http://192.168.0.100:11311`. Finally, the `talker` node is started by utilizing the command `rosrun rospy_tutorials talker.py`. If the network and ROS have been configured correctly, the test message "`hello world`" with a counter appears on the base laptop. This process is shown in Figure 11.

```
# On the base laptop:
$ export ROS_MASTER_URI=http://base_laptop@base:11311
$ roscore

# Start listen node (in a separate terminal window)
$ rosrun rospy_tutorials listener.py

# On the Pioneer P3-DX processing unit:
$ export ROS_MASTER_URI=http://base_laptop@base:11311
$ rosrun rospy_tutorials talker.py
```

Figure 11. Procedure to test if ROS has been properly configured for use on multiple machines.

Once both the robot and the base laptop have been properly configured to communicate with each other, a secure "tunnel" must be established to allow the base laptop to remotely start tasks on the robot. In order to do this, the `command ssh SlimPro@p3dx` must be implemented. The user is prompted for the password of the robot's computer. Once given, the user is utilizing a terminal window as if it is on the remote machine.

B. P2OS STACK

Once ROS has been successfully installed, it is necessary to install the appropriate drivers in order for ROS to communicate with and control the Pioneer P3-DX. There are two main drivers available for the Adept MobileRobots Pioneer family of robots, the `p2os` and `ROSARIA` stacks. For this thesis, the `p2os` stack was chosen as the driver for the P3-DX. The packages that comprise the `p2os` stack are `p2os_driver`, `p2os_launch`, `p2os_teleop`, `p2os_urdf`, and `p2os_msgs`.

The `p2os_driver` package is the main package of the `p2os` stack and contains nodes, libraries, and parameters that are essential for ROS to interface with the Pioneer P3-DX's client-server Advanced Robot Control and Operations Software (ARCOS). The package `p2os_driver` receives linear and angular velocity commands by subscribing to the ROS topic `/cmd_vel` and sends the necessary motor commands to the P3-DX. Additionally, the `p2os_driver` package extracts motor encoder information and publishes position, orientation, and velocity in the form of an odometry message from [21], as seen in Figure 12, to the ROS topic `/pose`.

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Pose position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
```

Figure 12. Standard message format for the ROS odometry message, which sends *x*, *y*, and *z* position, orientation, and linear and angular velocities with covariance.

The p2os_driver package is also responsible for publishing the transforms of the robot to the /tf topic for other ROS nodes to utilize. Additionally, the p2os_driver package publishes useful information about the Pioneer P3-DX such as its battery state, digital input/output voltage, and analog input/output voltage [22]. The p2os_driver package utilizes the URDF from the p2os_urdf package. For this thesis, the Microsoft Kinect was added to the Pioneer's URDF model. The URDF is responsible for establishing the transforms for each joint and link of the robot so the p2os_driver package can publish it to the /tf topic. The p2os_driver package also utilizes message formats that are specific to the p2os stack, which are located in the p2os_msgs package.

The p2os_launch package contains useful ROS launch files for running multiple nodes of the p2os stack to systematically start certain operations of the robot. Some of the launch files run necessary parameters for proper navigation of the Pioneer P3-DX, while others are used for running sensors such as the Hokuyo laser range scanner [22]. For this thesis, a master launch file, which can be found in Appendix A, was created in order to launch all the necessary nodes to conduct SLAM, autonomous navigation, and human form tracking with the Pioneer P3-DX and Microsoft Kinect.

The commands that must be utilized in the shell command-line in order to download, build, and configure the p2os stack are seen in Figure 13. The p2os_driver node can also be started by using the roslaunch tool as seen in Appendix A.

```
# Install p2os stack
$ sudo apt-get install ros-<ros_distro>-p2os-driver ros-<ros_distro>-p2os-teleop ros-
<ros_distro>-p2os-launch ros-<ros_distro>-p2os-urdf ros-<ros_distro>-pr2-controller ros-
<ros_distro>- joystick-drivers

# Download and build the p2os repository
$ cd ~/catkin_ws/src
$ git clone https://github.com/allenh1/p2os.git
$ cd ~/catkin_ws
$ source /devel/setup.bash
$ catkin_make

# Modify permissions to serial or USB port
$ sudo chmod 777 -R /dev/ttyS0 #(or ttyUSB0)

# Run p2os_driver node
$ rosrun p2os_driver p2os_driver

# Publish enable_motor message (in a separate terminal window)
$ rostopic pub /cmd_motor_state p2os_driver/MotorState 1
```

Figure 13. Commands utilized to download and build the p2os stack.

In order for the Pioneer P3-DX to be able to communicate to the SlimPro mini-computer, it is necessary to configure the system in order to allow the robot to have access permissions via the serial or USB ports. To properly set up permissions to allow the P3-DX to connect to its processor, the robot must be powered, and the last command shown in Figure 13 must be utilized in the terminal window [23]. The program `sudo` must precede the command `chmod` in order to run the `change mode` program with the security privileges of the superuser or root. The `chmod` command allows users to adjust access permissions for the owner, a group, or the general public. The numerals `777` give different levels of permissions for the different users as seen in Table 2, with the first digit being the owner, the second digit being a group, and the third digit being the general public.

Table 2. Table of numerical permissions, with 7 being the most permissive and 0 being the most restrictive.

#	Binary	rwx
0	000	No permissions
1	001	Execute
2	010	Write
3	011	Write and execute
4	100	Read
5	101	Read and execute
6	110	Read and write
7	111	Read, write, and execute

C. OPENNI STACK

The packages responsible as the driver for the Microsoft Kinect and converting raw depth, RGB, and infrared streams to depth images, disparity images, and point clouds were `openni_camera` and `openni_launch`. Within the `openni_camera` package,

the `openni_node` acts as the driver for a camera. After capturing the stream from a camera, the `openni_camera` package publishes `camera_info` and `image_raw` data to ROS topics for the RGB camera, depth camera, depth registered camera, and infrared camera.

The `openni_launch` package contains the necessary launch files to simultaneously start the device driver and the processing nodes which convert the raw RGB and depth images to useful products such as point clouds [24]. Additionally, it produces depth registered data. The depth and color images from the Microsoft Kinect are captured from two, separate, slightly offset sensors; therefore, oftentimes, the pixels from the RGB camera and the depth camera do not overlap perfectly. A registered depth image is built by calculating, for each pixel in the depth image, the three-dimensional position and projecting it onto the image plane of the RGB camera. The registered depth image has each pixel aligned with its counterpart in the RGB image as shown in Figure 14.

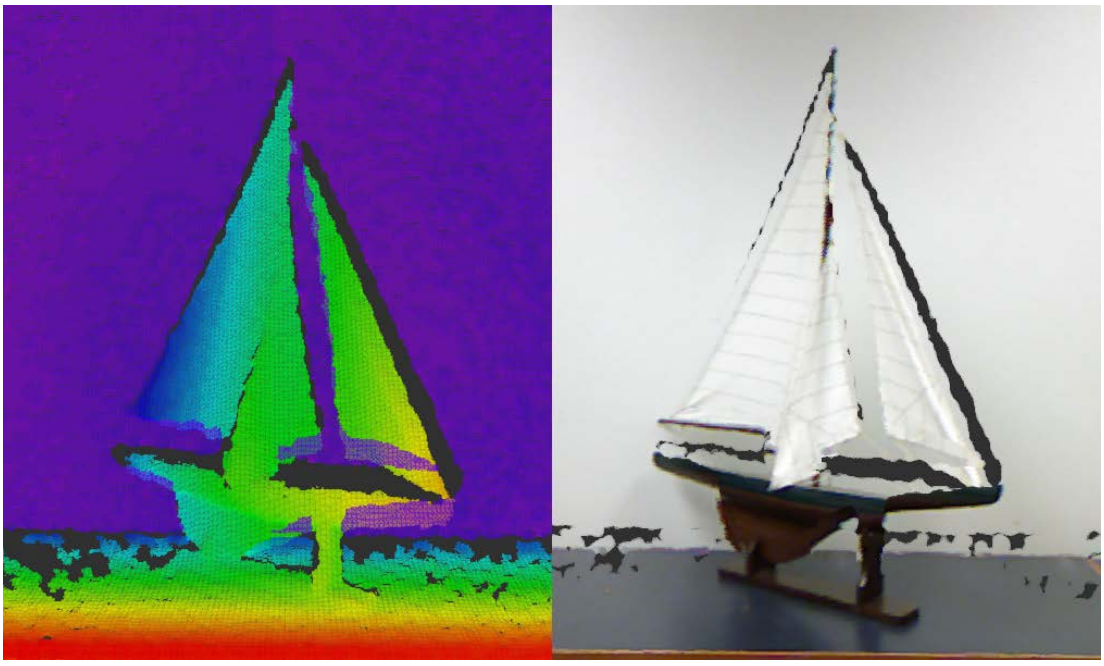

Figure 14. The left image was captured by the Microsoft Kinect's depth camera, which shows pixels with maximum range marked as purple and minimum range marked as red. The right image demonstrates depth registration.

The launch package for `openni_camera` senses the type of device utilized and adjusts its parameters to match the device. The `openni_camera` package can interface with the following depth sensors: ASUS Xtion PRO, ASUS Xtion PRO Live, and PrimeSense PSDK 5.0, as well as the Microsoft Kinect [25].

The `openni_tracker` package detects and tracks a human within the field-of-view of the Kinect. Utilizing `openni_tracker`, the Microsoft Kinect can track up to six users and provide detailed, joint tracking for up to two user's skeletons simultaneously. Once the `openni_tracker` package has identified a human form, the user can initiate skeleton tracking by performing the "psi pose." The package publishes tracked skeletons in the form of a set of transforms through the `/tf` topic, tracking head, neck, torso, and left and right shoulders, elbows, hands, hips, knees, and feet. All `openni_tracker` transforms are published from the camera frame as the parent frame, which is published by `openni_camera` [26]. The commands that must be utilized in the shell command-line interface in order to download and run `openni_camera`, `openni_launch`, and `openni_tracker` are seen in Figure 15. The nodes can also be run from a `roslaunch` file as seen in Appendix A. The transforms published by the `openni_tracker` package can be visualized using Rviz as seen in Figure 16.

```
# Install openni_camera, openni_launch, and openni_tracker
$ sudo apt-get install ros-<ros_distro>-openni-camera ros-<ros_distro>-openni-launch ros-
<ros_distro>-openni-tracker

# Build openni stack
$ cd ~/catkin_ws
$ catkin_make

# Run openni_launch
$ roslaunch openni_launch openni.launch camera:=kinect depth_registration:=true

# Run openni_tracker (in a separate terminal window)
$ rosrun openni_tracker openni_tracker

# View feed from Microsoft Kinect (in a separate terminal window)
$ rosrun rviz rviz
```

Figure 15. Commands utilized to download and run `openni_camera`, `openni_launch`, and `openni_tracker` as the driver and processors for the Microsoft Kinect.

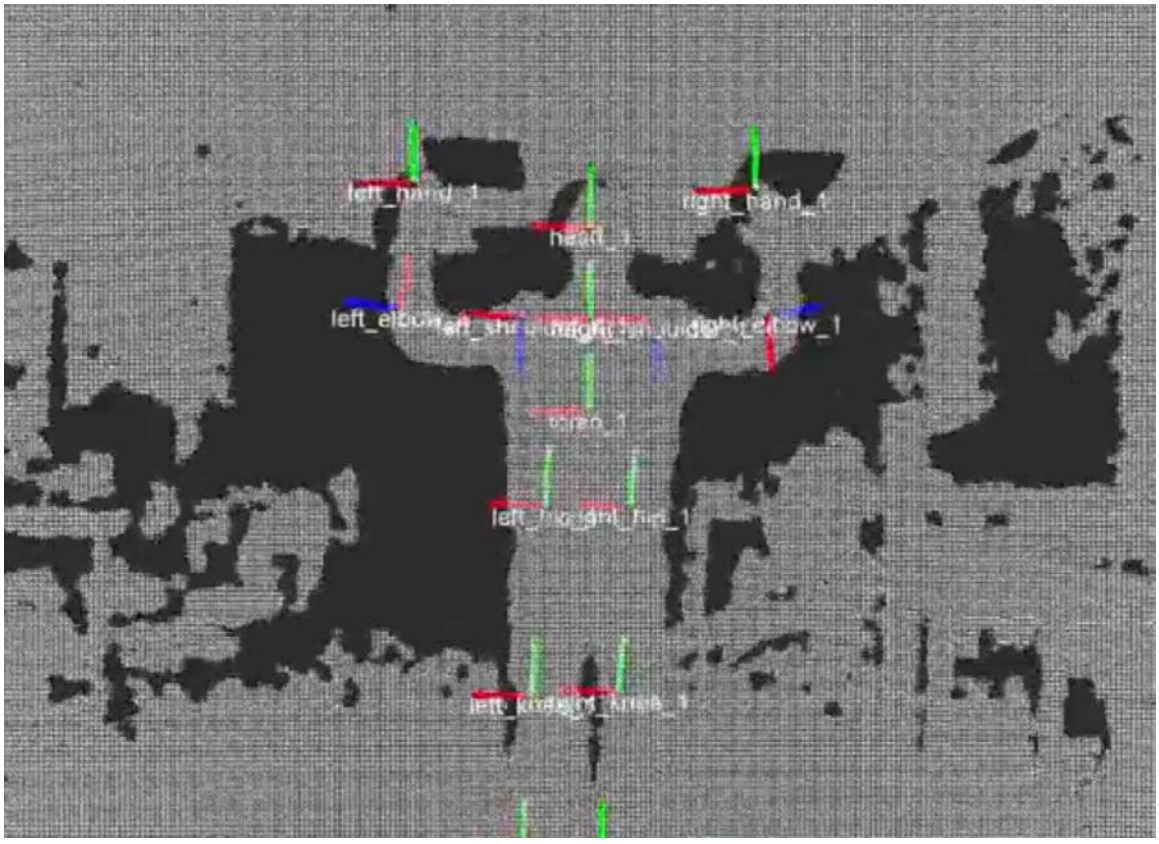

Figure 16. The transforms produced by the `openni_tracker` package. This is the psi pose used for joint tracking calibration.

D. NAVIGATION STACK

The ROS navigation stack requires the robot to publish information about the relationships between coordinate frames using the ROS `/tf` topic, a sensor publishing data about the environment, and odometry information about the orientation, position, and velocity of the robot.

1. Sensor Information

The `gmapping` package within the navigation stack is responsible for providing laser-based SLAM. Since the `openni_camera` package provides point-cloud data from the Microsoft Kinect, the `depthimage_to_laserscan` package was utilized to meet the `gmapping` package's requirements for laser-based range data. The `depthimage_to_laserscan` package converts a horizontal slice of the point-cloud

data into depth laser scan, formatted as a `sensor_msgs/LaserScan.msg` message as demonstrated in Figure 17, where the range colored line is overlaid across the depth-registered point cloud [27].

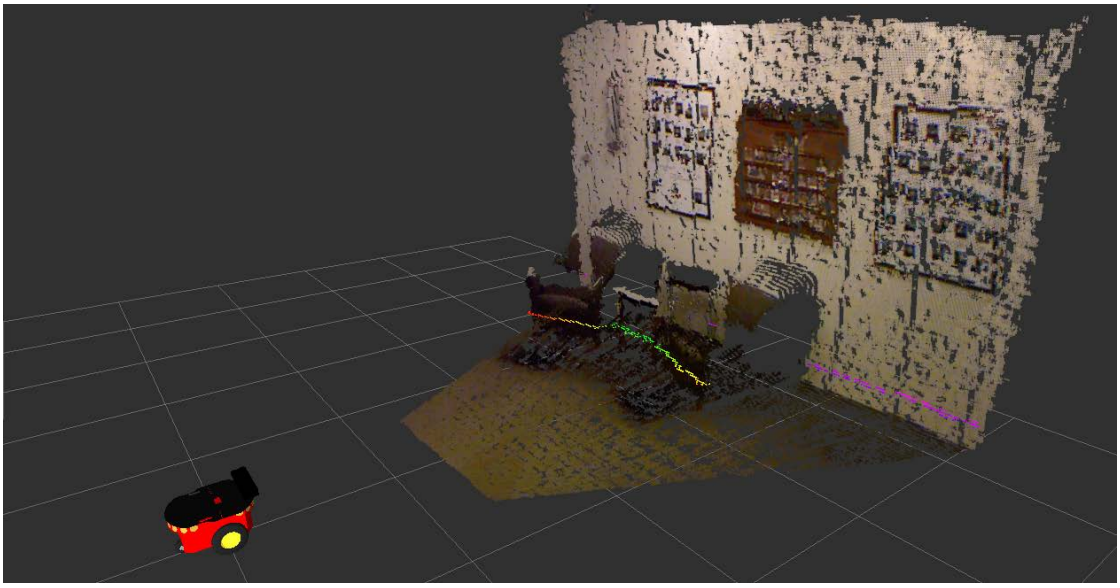

Figure 17. Depth registered point cloud with converted laser scan (red shows minimum range and purple shows maximum range). The mesh of the Pioneer P3-DX is created by the URDF.

The commands that must be utilized in the shell command-line interface in order to download and run `depthimage_to_laserscan` are shown in Figure 18. To properly convert point-cloud data to laser-scan data, the `depthimage_to_laserscan` node must subscribe to the ROS topics with the camera's depth image as well as the camera's info. For this thesis, those two topics were `/kinect/depth/image_raw` and `/kinect/depth/camera_info`. The topics are included as arguments when the `depthimage_to_laserscan` node is run within the command line, as in Figure 18, or in a `roslaunch` file, as in Appendix A. The `depthimage_to_laserscan` node's output is published to the `/scan` topic as a `sensor_msgs/LaserScan.msg` message.

```
# Install depthimage_to_laserscan package
$ sudo apt-get install ros-<ros_distro>-depthimage-to-laserscan

# Build depthimage_to_laserscan package
$ cd ~/catkin_ws
$ catkin_make

# Run openni_launch
$ roslaunch openni_launch openni.launch camera:=kinect depth_registration:=true

# Run depthimage_to_laserscan (in a separate terminal window)
$ rosrun depthimage_to_laserscan depthimage_to_laserscan image:=/kinect/depth/image_raw

# View feed from Microsoft Kinect (in a separate terminal window)
$ rosrun rviz rviz
```

Figure 18. Commands utilized to download, run, and view `depthimage_to_laserscan`; the package that converts point-cloud depth images to range-finding laser scans.

2. Odometry Information

In robotics, odometry information refers to the estimated pose, position, orientation, and velocity, of a robot in free space and is required by the navigation stack in order to conduct SLAM. Odometry information is typically determined through kinematics from the encoder counts of the robot's motor shafts. In [28], the robot's pose p, position and heading in the world coordinate frame, is calculated utilizing encoder dead-reckoning. First, the distance each wheel rotates over the ground is calculated from

$$\Delta^r s_{r/l} = \Delta e_{r/l} \frac{\pi w}{e_{rev}} \tag{1}$$

where $\Delta^r s_{r/l}$ is the change in distance that the right or left wheel rotates over the ground since the last sample, $\Delta e_{r/l}$ is the change in encoder counts since the last sample, e_{rev} is the number of encoder counts for one wheel revolution, and w is the diameter of the wheel. Next, the distance the robot has traveled $\Delta^r s$ since the last sample with respect to the robot's coordinate frame can be determined from

$$\Delta^r s = \frac{\Delta^r s_r + \Delta^r s_l}{2}, \tag{2}$$

and the change in the robot's heading $\Delta^r \psi$ since the last sample with respect to the robot's coordinate frame can be determined from

$$\Delta^r \psi = \frac{\Delta^r s_r - \Delta^r s_l}{a} \tag{3}$$

where a is the length of the wheel base. With Equations (2) and (3), the robot's pose in the world-coordinate frame can be determined from

$${}^w p_{k+1} = \begin{bmatrix} {}^w x_{k+1} \\ {}^w y_{k+1} \\ {}^w \psi_{k+1} \end{bmatrix} = \begin{bmatrix} {}^w x_k \\ {}^w y_k \\ {}^w \psi_k \end{bmatrix} + \begin{bmatrix} \Delta^r s \cos\left({}^w \psi_k + \frac{\Delta^r \psi}{2} \right) \\ \Delta^r s \sin\left({}^w \psi_k + \frac{\Delta^r \psi}{2} \right) \\ \Delta^r \psi \end{bmatrix} \tag{4}$$

where ${}^w x$ and ${}^w y$ represents the robot's position in the world coordinate frame and ${}^w \psi$ is the heading in the world coordinate frame. The subscript k denotes the last sample taken.

The `p2os_driver` package extracts the encoder information from the Pioneer P3-DX, calculates odometry data, and publishes the data over the `/tf` and `/pose` topic as a `nav_msgs/Odometry.msg` message. Within the navigation stack, the node `slam_gmapping` subscribes to the `/tf` topic and receives the data.

3. Transform Configuration

When working with mobile robots, it is crucial that the robot is aware of itself as well as the surrounding environment. To be able to sense an obstacle, it is not only enough that the robot is able to see the obstacle, but it must be able to calculate its geographical relationship to the obstacle. The `openni_camera` package gives the location (*x*, *y*, *z*) of the obstacle with respect to the Microsoft Kinect's coordinate frame, but in order to navigate around an obstacle, the robot must be able to calculate the obstacle's location relative to the robot's coordinate frame. The position of an object relative to the coordinate frame of a sensor can be defined as the position vector

$${}^S P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{5}$$

where P_x, P_y, P_z are the individual elements giving the orientation of the vector from the sensor to the obstacle. To calculate the position vector in the robot's coordinate frame, that is, from the robot to the object, we must use a rotation matrix ${}^{R}_{S}R$, which defines the principal directions of a coordinate system relative to another. In this case, the rotation describes the sensor's coordinate frame to the robot's coordinate frame. The transformation of the position vector to the obstacle from the sensor's coordinate frame to the robot's coordinate frame is given by

$$ {}^{R}P = {}^{R}_{S}R\,{}^{S}P + {}^{R}P_{S_{org}} \tag{6} $$

$$ {}^{R}_{S}R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{7} $$

where ${}^{R}P$ is the position vector of the object relative to the robot, ${}^{R}_{S}R$ is the rotation matrix from the sensor's coordinate frame to the robot's coordinate frame, ${}^{S}P$ is the position vector to the object from the sensor's coordinate frame, and ${}^{R}P_{S_{org}}$ is the position vector to the origin of the sensor's coordinate frame from the robot's coordinate frame. Equation (6) can be compacted to

$$ \begin{bmatrix} {}^{R}P \\ 1 \end{bmatrix} = {}^{R}_{S}T \begin{bmatrix} {}^{S}P \\ 1 \end{bmatrix} \tag{8} $$

where

$$ {}^{R}_{S}T = \begin{bmatrix} & {}^{R}_{S}R & & {}^{R}P_{S_{org}} \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} & & & 1 \end{bmatrix} \tag{9} $$

represents the position and orientation of the sensor coordinate frame's origin relative to the robot's coordinate frame. In Equation (9), the position vectors and rotation matrix are extended and made homogeneous. Utilizing transformation matrix multiplication

$$ {}^{C}_{A}T = {}^{C}_{B}T\,{}^{B}_{A}T \tag{10} $$

and Equation (8), we can find the position vector through multiple layers of coordinate frames.

The `p2os_urdf` package is responsible for generating the relationships between all joints and links of the Pioneer P3-DX. The `p2os_driver` package takes these relationships from the Pioneer's URDF and publishes them to the `/tf` ROS topic so all other nodes can utilize them. Whenever a node desires to know the relationship between two objects, it listens to the `/tf` topic and follows the transform tree, as seen in Figure 5, between the two objects. When traveling up the tree, the inverse transform is used, and when traveling down the tree, the value of the transform is used as in Equation (10).

In order for the `p2os_driver` package to broadcast the proper position and orientation of the Microsoft Kinect relative to the Pioneer P3-DX, the Kinect must be added to the `p2os_urdf` package's XML file for the P3-DX. A generalized format for the URDF's XML file is shown in Figure 19. The portion of the code added to the `p2os_urdf/defs/pioneer3dx_body.xacro` file is found in Appendix E.

```
<?xml version="1.0"?>

<robot name="my_robot">
    <link name="base_link"/>

    <joint name="manipulator_base_joint" type="continuous"/>
        <parent link="base_link"/>
        <child link="manipulator_base"/>
        <origin xyz="0.5 0 0.8" rpy="0 0 0"/>
    </joint>

    <link name="manipulator_base"/>

    <joint name="manipulator_upper_arm_joint" type="continuous"/>
        <parent link="manipulator_base"/>
        <child link="manipulator_upper_arm"/>
        <origin xyz="0.0 0 0.4" rpy="0 0 0"/>
        <axis xyz="0 0 1"/>
    </joint>

    <link name="manipulator_upper_arm"/>

    <joint name="manipulator_lower_arm_joint" type="continuous"/>
        <parent link="manipulator_upper_arm"/>
        <child link="manipulator_lower_arm"/>
        <origin xyz="0.3 0 -0.2" rpy="1.57075 0 0"/>
        <axis xyz="0 1 0"/>
    </joint>

    <link name="manipulator_lower_arm"/>

    <joint name="sensor_joint" type="fixed"/>
        <parent link="base_link"/>
        <child link="sensor"/>
        <origin xyz="-0.2 0 0.6" rpy="0 0 0"/>
    </joint>

    <link name="sensor"/>
</robot>
```

Figure 19. An example of the XML format for a URDF of a generic robot with a manipulator arm.

4. SLAM – gmapping

In robotics, SLAM is the problem of utilizing sensors to construct a map of an unknown environment while simultaneously keeping track of the robot's location. For humans, SLAM comes naturally. Even the most directionally challenged people can use their senses to find recognizable landmarks from which they can identify their location; however, designing a robot without the use of a GPS device to identify its location and simultaneously building a map is a complex problem. For localization, a robot requires a consistent and accurate map; for constructing a map, a robot needs a good estimate of its location. Given a series of sensor observations, $z_{1:t}$, and odometry measurements, $u_{0:t}$, the SLAM problem is to compute an estimate of the robot's location, $x_{1:t}$, and a map, m, of its environment. There are many popular statistical techniques to accomplish this problem to include extended Kalman filter, particle filter, and range-scan matching. Within the ROS community, several open-sourced SLAM implementations are available such as `hector_slam`, `cob_3d_mapping_slam`, `gmapping`, and `mrpt_slam`. For this project, the `gmapping` package was utilized.

The `gmapping` package uses a Rao-Blackwellized particle filter. In order to reduce the common problem of particle depletion associated with the Rao-Blackwellized particle filter, the `gmapping` package employs an adaptive resampling technique [29] [30] [31]. The `gmapping` package used a two-dimensional occupancy grid method to construct a map. Using sensor stream data, it either inserts an obstacle into a cell or clears a cell. Clearing a cell consists of ray-tracing through a grid for each successful laser-scan sample. GMapping also utilizes scan matching, comparing current laser scans to previous laser scans in order to reduce and/or correct odometry drift errors. As the likelihood of scan matching an obstacle with the same obstacle in a previous scan increases, the `slam_gmapping` node registers that obstacle on the map, while those scans with a low likelihood begin to clear the occupancy grid [32]. In order to insert data into a map, the `slam_gmapping` node makes extensive use of the ROS `/tf` topic to identify the geographical relationship of an obstacle, as seen from the sensor coordinate frame, and place it in the correct position with respect to the map coordinate frame [33]. By its

utilization of ROS `/tf` topic, the `slam_gmapping` node is able to construct and publish the map to the `/map` topic. The `gmapping` package requires transforms from the sensor source to the `base_link` and broadcasts the transform from the `/map` to the `/odom` frames. The commands utilized in order to download and run the gmapping package are shown in Figure 20. The code to run the slam_gmapping package from a roslaunch file can be found in Appendix B, Section 2.

```
# Install gmapping package
$ sudo apt-get install ros-<ros_distro>-gmapping ros-<ros_distro>-slam-gmapping

# Build gmapping package
$ cd ~/catkin_ws
$ catkin_make

# Run slam_gmapping node
$ roslaunch gmapping slam_gmapping scan:=scan odom_frame:=odom
```

Figure 20. Command utilized to install and run `gmapping` package, subscribe to the `/scan` topic and publish the map in relation to the odometry frame.

5. Autonomous Navigation – move_base

The `move_base` package lies at the heart of the navigation stack. It maintains a global and local costmap through the `costmap_2d` node as well as links together a global and local planner to accomplish the global navigational task. It is also responsible for publishing velocity commands to the robot via the `/cmd_vel` topic.

The `costmap_2d` package uses the developed map from the `gmapping` package, via the `move_base` package, and data from sensor sources to develop a `global_costmap` and a `local_costmap`. A costmap is a type of occupancy grid, but unlike the occupancy grid developed by the `slam_gmapping` node, each cell of the costmap not only is marked as free, occupied, or unknown but also has a cost value between 0 and 254 associated with it [34]. As obstacles are identified and the associated cells are marked as occupied, the surrounding cells are also given a cost based on the shape, dynamics, and orientation of the robot. With costmaps, as in Figure 21, the location of the robot is considered to be the size of a single cell and identified obstacles

are inflated by increasing the cost of surrounding cells to account for the footprint of the robot, depending on the robot's orientation. This is often known as configuration space as shown in Figure 22. The `costmap_2d` package publishes global and local occupancy grid and occupancy grid updates to the `move_base` package.

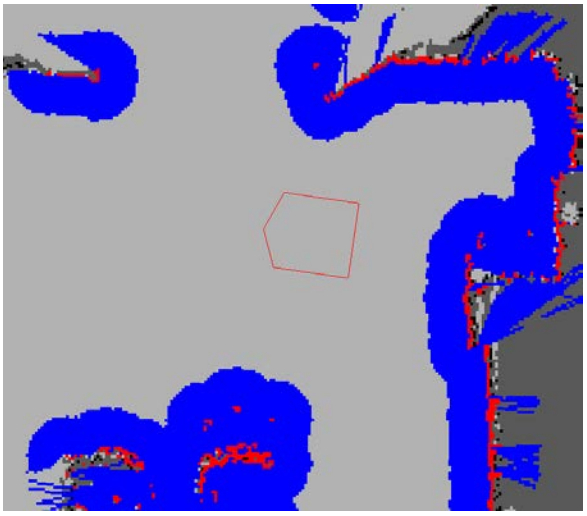

Figure 21. Depiction of a costmap, where the cells marked in red are considered to be obstacles, cells marked in blue represent obstacles inflated by the inscribed radius and orientation of the robot, and cells marked in gray are considered to be free space. To avoid obstacle collision, the center point of the robot should never cross a blue cell, from [34].

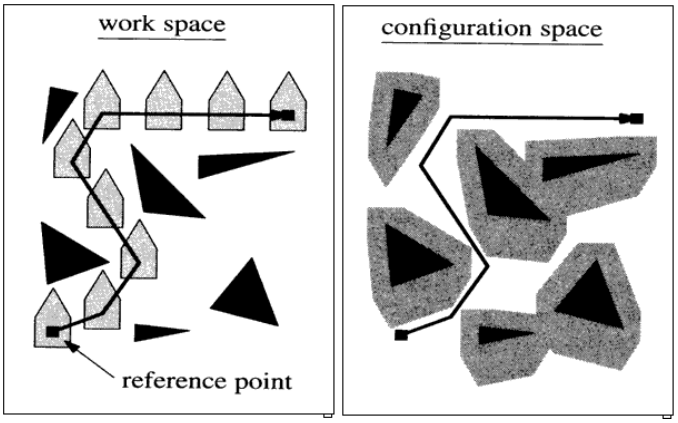

Figure 22. The difference between work space and configuration space. Note the inflation of the obstacles and compaction of the robot to a single reference point in configuration space, from [35].

With the `global_costmap` and a given goal position and orientation, the `move_base` node creates a global path through the `global_planner` package. The `global_planner` package can utilize several different path planning algorithms such as Dijkstra's algorithm, quadratic or non-quadratic potential field algorithms, and the A* algorithm, examples of which can be found in Figure 23, depending on the parameters set by the user. The global path is published by the `global_planner` package to the `/move_base/TrajectoryPlannerROS/global_plan` topic via the ROS-standard `nav_msgs/Path.msg` message.

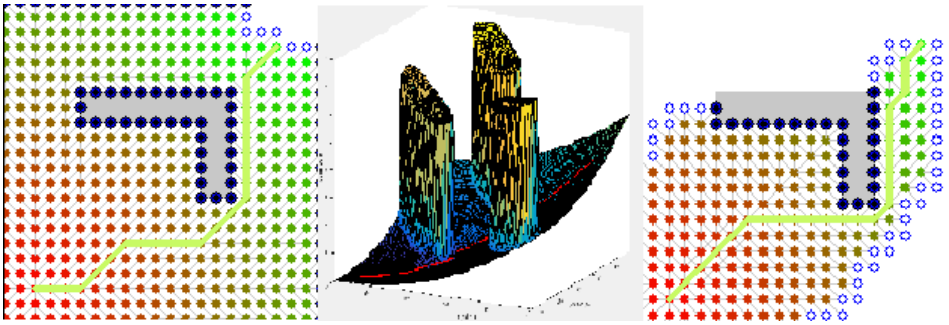

Figure 23. Example images of path planning algorithms that can be used in the `global_planner` package, from left to right, Dijkstra, potential field, A*, from [36].

To edit the parameters of the global`_costmap` and `local_costmap`, parameter files were created and accessed in the navigation launch file. Three `.yaml` parameter files were created, one for common costmap configurations which apply to the global and local costmaps, one for global costmap configurations, and one for local costmap configurations. The configuration files can be found in Appendix C.

With the aid of odometry data and the `local_costmap`, the `base_local_planner` package develops a local trajectory, serving as a connection from the global path planner and the robot through the `move_base` package. Its end-state is to provide dx, dy, and $d\theta$ velocities to the `move_base` package to send to the robot. The internal process of the `base_local_planner` package is to discretely sample the robot's control space (dx, dy, $d\theta$), perform forward simulation for each sampled

velocity, evaluate each trajectory for characteristics such as proximity to obstacles, proximity to goal, proximity to the global path, and speed, select the highest-scoring trajectory, and send the associated velocity to the `move_base` package for the robot [37]. A parameter file, as found in Appendix C, was also used to configure the `base_local_planner` package and how it produces trajectories for the robot.

Not only does the `move_base` node act as a central line to the `global_costmap` package, the `local_costmap` package, the `global_planner` package, and the `base_local_planner` as shown in Figure 24 but is also responsible for performing recovery behaviors if the robot perceives itself as stuck [38]. If the robot goes into recovery procedures, it first clears obstacles from the robot's map in the surrounding area and then attempts to perform an in-place rotation to 'visually' clear the surrounding space with its sensors. Should the first action fail, it then clears obstacles in a larger area from the robot's map and conducts another clearing rotation. If the last action fails, then it finally aborts the mission and reports to the user that the robot considers its goal infeasible. These recovery procedures are shown in Figure 25.

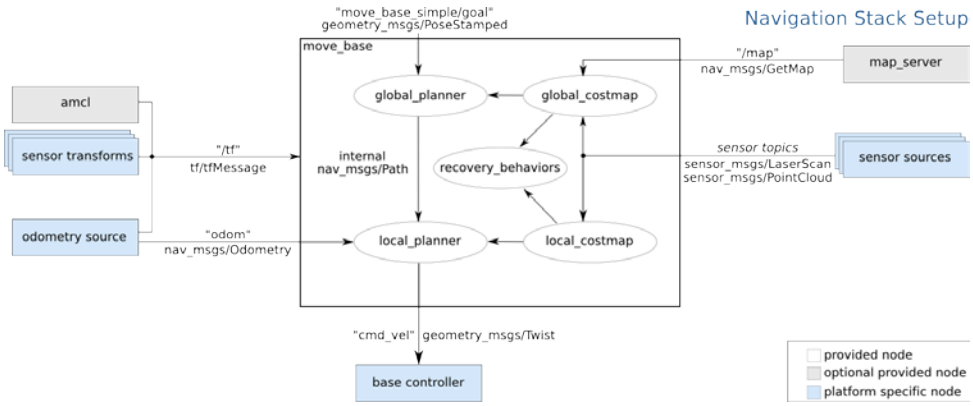

Figure 24. Flow chart of the internal communications in the `move_base` package, from [38].

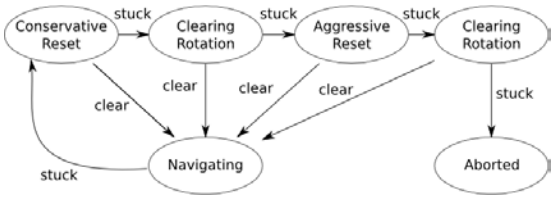

Figure 25. The navigation stack's `move_base` package goes into recovery procedures should the robot become stuck, from [38].

THIS PAGE INTENTIONALLY LEFT BLANK

IV. RESULTS

The integration of ROS with the Microsoft Kinect, the Pioneer P3-DX, and the ROS packages that conduct SLAM and autonomous navigation are discussed in this chapter, which is divided into three parts. First, the robot was remotely teleoperated by keyboard while conducting SLAM in order to create a map of the environment. Second, the robot conducted autonomous navigation with the pre-constructed map, avoiding obstacles while achieving a goal position and orientation. Third, the robot utilized SLAM and autonomous navigation in an unknown and dynamic environment, simultaneously building a two-dimensional map of the surroundings while attaining the goal pose. All data from the environment was obtained through the Microsoft Kinect, and motor commands were given to the Pioneer P3-DX utilizing ROS.

A. MAPPING

The first phase in the integration of the Microsoft Kinect on a mobile robot platform and ROS in conducting SLAM and autonomous navigation operations was to test the ROS `gmapping` package in mapping an unknown environment. This was accomplished by remote operating the control of the mobile robot platform and capturing data of the environment with the Microsoft Kinect.

The teleoperation node, after [39], which can be found in Appendix D, received keyboard inputs from the W, A, S, D, and SHIFT keys and published desired velocities to the `/cmd_vel` topic. As the p2os_driver package received commanded velocities from the `/cmd_vel` topic, it also produced odometry data, which was published over the `/tf` topic in the form of a `nav_msgs/odometry.msg` message. The odometry data was utilized by the `gmapping` package and paired with scan matching for self-localization. The commands given to start the packages required to conduct teleoperated SLAM with the Pioneer P3-DX can be found in Figure 26 and Appendices A and B.

```
# Start ROS Master
$ roscore

# Start secure connection to robot from base laptop (in a separate terminal window)
$ ssh SlimPro@p3dx

# Start roslaunch file that runs p2os stack, openni stack, and depthimage_to_laserscan node
$ roslaunch follow_me follow_me.launch

# Start keyboard teleoperation node (in a separate terminal window)
$ rosrun follow_me follow_me_teleop_keyboard

# Start rosbag record (in a separate terminal window)
$ rosbag record -O follow_me_map /scan /tf /rosout

# Once environment has been captured, stop recording with rosbag
# Set simulation time, start gmapping, start recorded data
$ rosparam set sim_time true
$ rosrun gmapping slam_gmapping scan:=scan odom_frame:=odom map_update_interval:=8.0
maxUrange:=6.5 maxRange:=8.0
$ rosbag play follow_me_map.bag --clock

# Once recorded data has completed, convert map to .png image file and .yaml parameter file
$ rosrun map_server map_saver
```

Figure 26. Command-line inputs to record data from environment, conduct post-capture SLAM, and save map data.

The robot was wirelessly driven throughout the environment as the `openni_camera` package extracted streaming depth and RGB images from the Microsoft Kinect. The `openni_camera` package published the steaming depth image and associated camera information via the topics `/kinect/depth/image_raw` and `/kinect/depth/camera_info`. The `/depthimage_to_laserscan` node subscribed to the topics and converted the point-cloud data to a `sensor_msgs/LaserScan.msg` message published via the `/scan` topic as can be seen in Figure 27. With the estimated position of the Pioneer P3-DX through the `/tf` topic and range data collected at 30 Hz from the `/scan` topic, the `slam_gmapping` node was able to utilize particle filtering and scan matching in order to conduct SLAM.

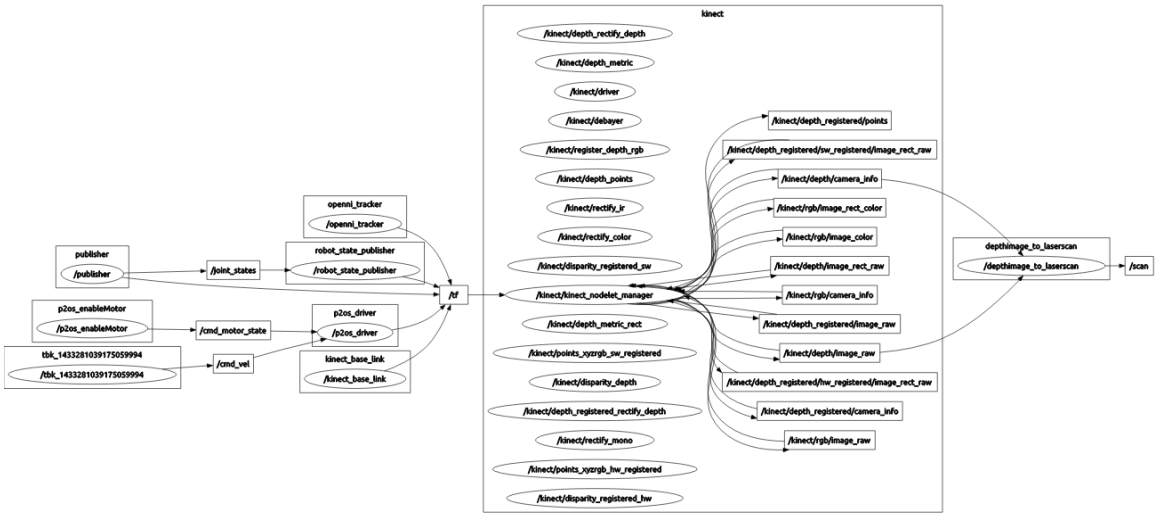

Figure 27. Image depicting communication between nodes and topics utilizing the tool `rqt_graph`.

It was found that smoother teleoperation control of the Pioneer P3-DX resulted in more accurate results in the construction of the map. Additionally, several of the parameters of the `slam_gmapping` node were optimized as can be found in the navigation launch file in Appendix B, Section 1. The map, shown in Figure 28, was constructed by recording the data from the `/tf` and `/scan` topics with the `rosbag` tool utilizing the lines of code as found in Figure 26. Note the error in loop closure, which can be attributed to odometry drift errors.

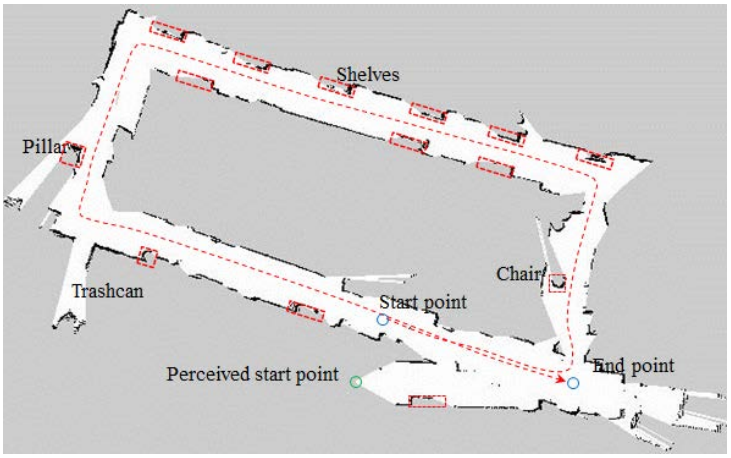

Figure 28. This image is a map of the interior of an office building created by the teleoperated Pioneer P3-DX with Microsoft Kinect and `gmapping` package.

B. AUTONOMOUS NAVIGATION WITH MAP

The next phase in systematically integrating ROS with the Microsoft Kinect and Pioneer P3-DX for SLAM and autonomous navigation was to test and configure the `move_base` package by conducting autonomous navigation in a known environment. This was accomplished by utilizing the map created from the previous phase for the `move_base` package to utilize for autonomous navigation.

In order to utilize the map created from teleoperated SLAM, the map was loaded through the `map_server` package. This was accomplished by running the `map_server` package in the navigational launch file that can be found in Appendix B, Section 1. Since the `move_base` package does not provide its own localization, the `amcl` package, a ROS localization package, was utilized. The `amcl` package uses the adaptive Monte Carlo localization, which uses a particle filter to track the pose of a robot against a known map [40]. The commands given to start the required packages for the Pioneer P3-DX to conduct autonomous navigation are shown in Figure 29.

```
# Start ROS Master
$ roscore

# Start secure connection to robot from base laptop (in a separate terminal window)
$ ssh SlimPro@p3dx

# Start roslaunch file that runs p2os stack, openni stack, and depthimage_to_laserscan node
$ roslaunch follow_me follow_me.launch

# Start roslaunch file that loads the map and runs the move_base package
$ roslaunch follow_me_2dnav move_base_map.launch

# Start rviz (in a separate terminal window)
$ rosrun rviz rviz

# Once map is displayed in Rviz, input initial and goal positions and orientations
```

Figure 29. Command-line inputs to load map and start autonomous navigation.

The goal position and orientation were input through Rviz and published to the `/move_base_msgs/MoveBaseActionGoal` topic. The `move_base` package received the initial position and goal data and its approximate location through particle filtering from the `amcl` package. Through the `global_planner`, the `move_base` node

developed a feasible path from its location to the goal utilizing the global costmap. Then the `local_planner`, utilizing the local costmap, developed short-term trajectories and determined the optimal velocities to send to the robot via the `/cmd_vel` topic in order to get the robot on the global path while avoid obstacles. The map, goal, costmap, global path, mesh of the robot, point cloud, and converted laserscan were able to be viewed in real-time utilizing Rviz as can be seen in Figure 30 and Figure 31.

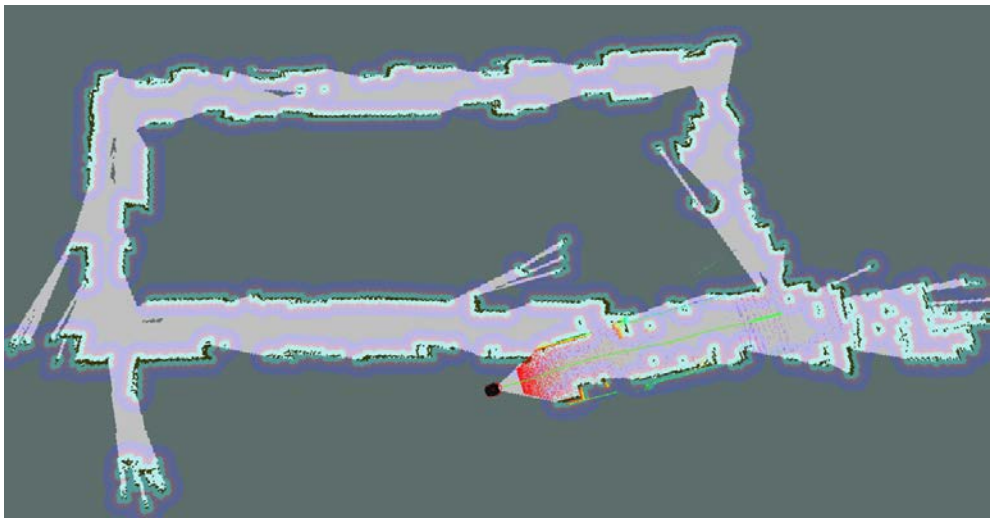

Figure 30. Image of Pioneer P3-DX conducting autonomous navigation on a pre-constructed map.

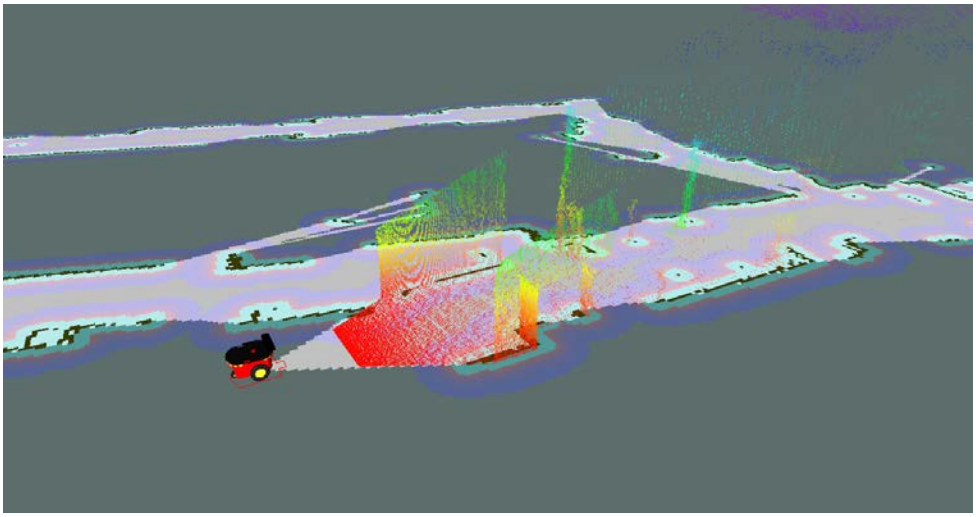

Figure 31. Image of the Pioneer P3-DX with the point-cloud data from the Microsoft Kinect on a known mapped environment.

During navigation through a previously constructed map, the robot behaved as expected with only a few shortfalls. When attempting to self-localize, the robot would often go into recovery procedures, conducting several in-place rotations; however, these in-place rotations would often distort the robot's self-localization because of increasing errors in the odometry data, resulting in a skewed view of the local map compared to the global. The `local_planner` would then make trajectory determinations based on the local costmap, even though its physical orientation concurred with the global costmap. These errors were most likely caused by errors in odometry data, as well as increasing latency in the `/tf`, as the robot's awareness of its location from odometry data did not concur with sampled scans.

Several parameters were adjusted to optimize the navigation such as velocity inputs, expected sensor sampling rates, planner frequency, controller frequency, and weights for the goal, global path, and obstacle avoidance, which improved the ability of the robot to achieve its goal while avoiding obstacles and increasing the accuracy matching scans to the given map. In adjusting some of these parameters, the speed of the robot and map update rate were exchanged in order to decrease latency, improve the robot's ability to avoid obstacles, and increase the controller frequency. Despite the occasional local costmap errors, the `move_base` node, given a goal, map, and sensor data from odometry and point-cloud converted laser scans, was able to effectively navigate throughout a dynamic environment, navigating through doorways, around corners, and moving obstacles.

C. SIMULTANEOUS LOCALIZATION AND MAPPING

In order to do autonomous navigation and SLAM in an unknown environment, both the `move_base` and `slam_gmapping` nodes were run simultaneously. Rather than the `global_costmap` node operating from a previously constructed map, the `global_costmap` node received the map data and localization information as it was being constructed from the `/map` topic, which was being published as a `nav_msgs/OccupancyGrid.msg` by the `slam_gmapping` node. The `global_costmap` node created a costmap from the map data, and, from the costmap and

user-input goal, the `global_planner` calculated a global path. Additionally, the `local_costmap` node received information from the converted laser scan and the global costmap to develop a local costmap. The `local_planner` node, based on data from the local costmap, conducted trajectory simulations, which translated to motor velocities that the `move_base` node published on the `/cmd_vel` topic. The `p2os_driver` node, which subscribed to the `/cmd_vel` topic, sent the appropriate motor commands to the Pioneer P3-DX. The entirety of the system, nodes publishing and subscribing to topics, is shown in Figure 32. The `/map` topic was visualized within Rviz in real-time from the base laptop.

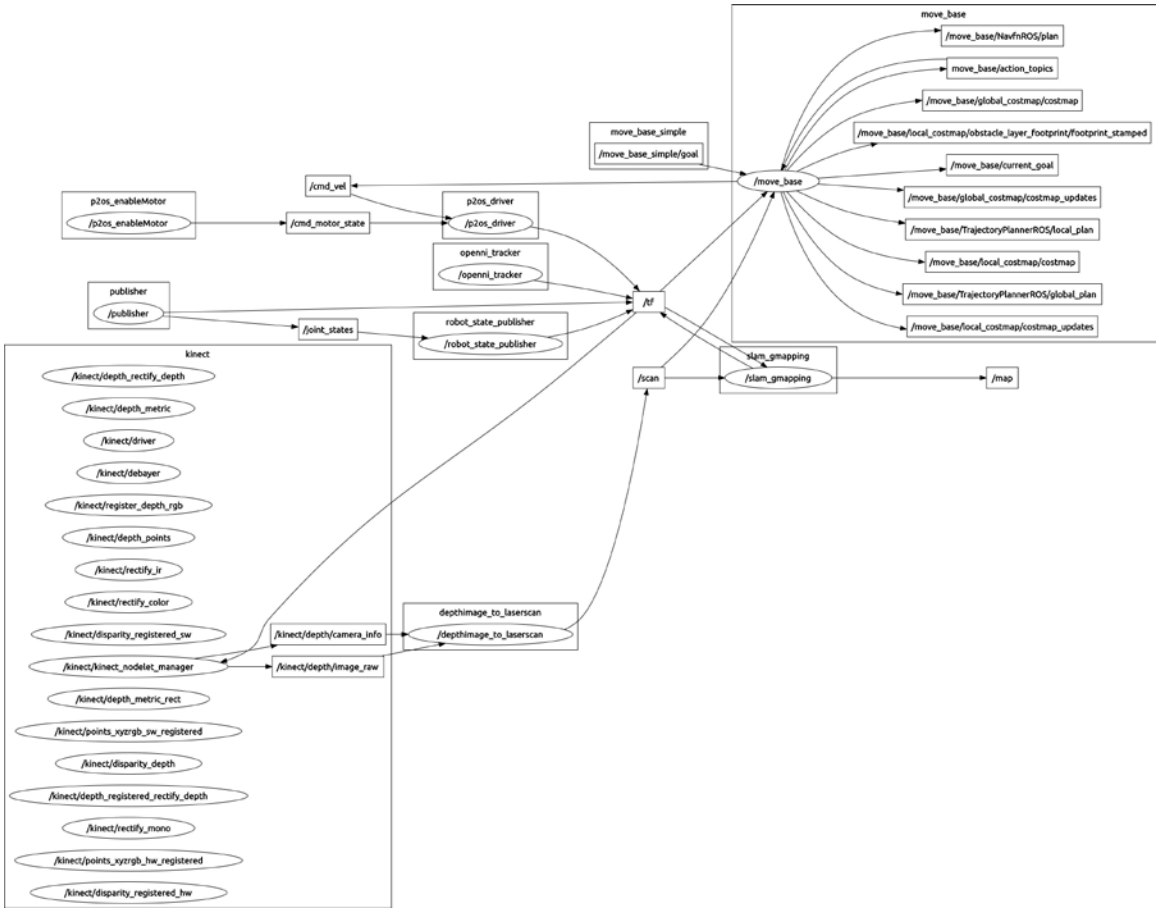

Figure 32. Graphical representation of active nodes communicating via topics while the robot was conducting autonomous navigation and SLAM.

Through the graphical interface, Rviz, the goal position and orientation were sent to the robot through the `/move_base_simple/goal` topic. Using the `slam_gmapping` node and `move_base` package, the robot was able to maneuver through the unknown environment towards the goal, constructing the map as it identified obstacles with the converted laser scans as seen in Figure 33 and Figure 34. The global costmap can be seen in Figure 33, where yellow represents obstacles, red identifies where the robot would strike identified objects if its center point intersected, and blue shows the cost inflation radius of the obstacle. An input goal, the global path, and the local costmap, which is highlighted around the robot in the lower-right corner of the image, and the entirety of the environment mapped by the robot while conducting SLAM and autonomous navigation is shown in Figure 34. The map in Figure 34 was created by the `slam_gmapping` node as the robot autonomously navigated to goals which were input through Rviz. These goals were given incrementally to the robot. As the robot achieved its navigational goal, another goal was input through Rviz. The robot was able to autonomously navigate to goals from one end of the hallway to the other, around corners, and through doorways. If the robot found itself stuck between obstacles, it conducted recovery behaviors, making several in-place rotations in an attempt to identify a feasible path towards the global path. Navigation improved as more of the environment became a part of the map because the robot was able to better estimate its location through scan matching. As can be seen when comparing Figure 34 with Figure 35, which was created by teleoperated SLAM, it was found the robot was able to create a fairly accurate map while conducting SLAM and autonomous navigation.

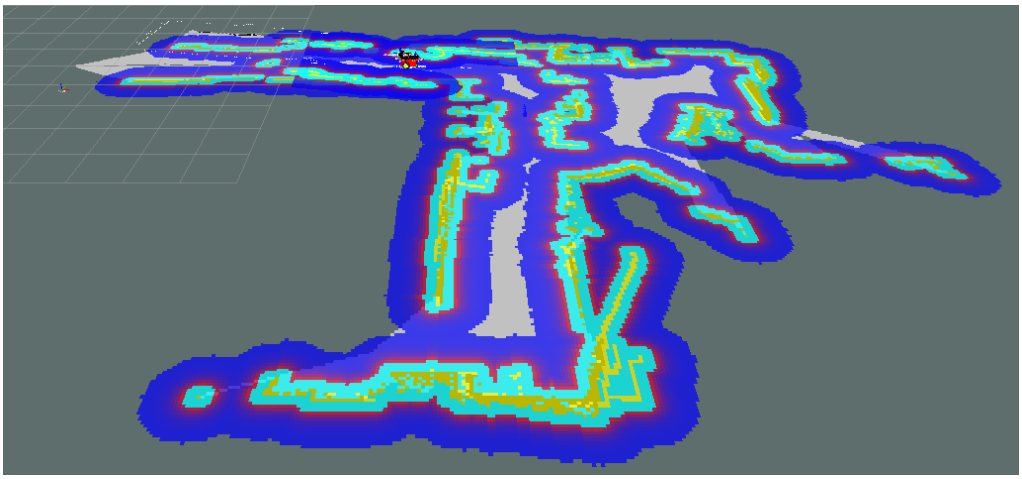

Figure 33. This image shows the global costmap while conducting autonomous navigation and SLAM of the interior of an office building.

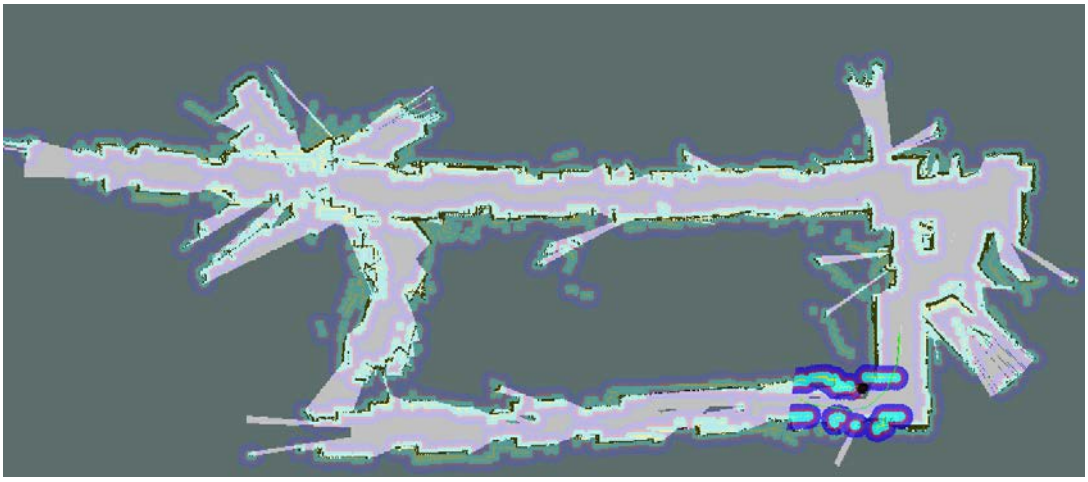

Figure 34. This image depicts the results of a SLAM constructed map of the interior of an office building. Note the rolling local costmap highlighted around the robot. Also, note the green global path leading to the goal position and orientation.

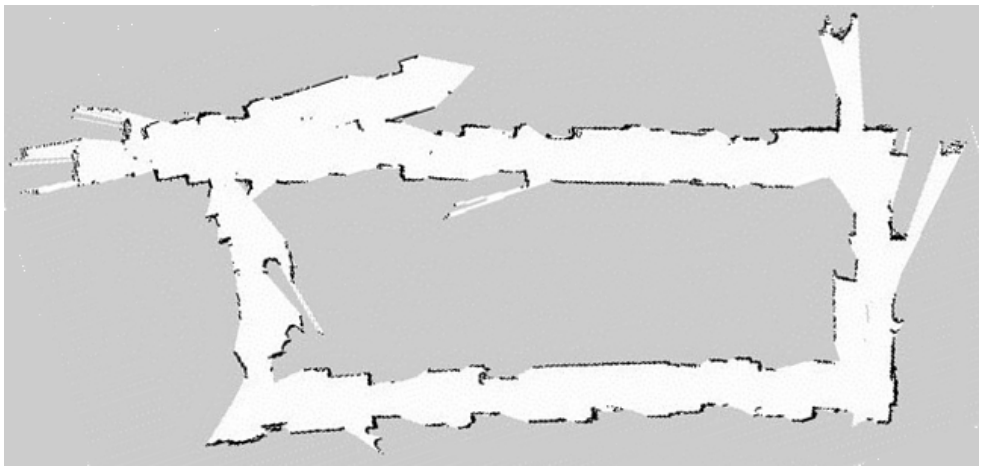

Figure 35. Image of same office building produced by teleoperation SLAM.

As with navigation utilizing the pre-constructed map, the robot still occasionally lost track of its pose, specifically its orientation. As before, these errors can be attributed to increasing odometry drift error over time. When conducting in-place rotations, the orientational error due to odometry often got worse, causing the robot's local costmap to become skewed relative to the global costmap. When the local costmap became skewed, it often cleared cells that were previously marked as an obstacle or placed an obstacle where there was actually free space in the environment as in Figure 36. This caused the global and local planners to create paths to go around obstacles that did not exist within the environment or attempt to go through obstacles that the planner considered to be free space.

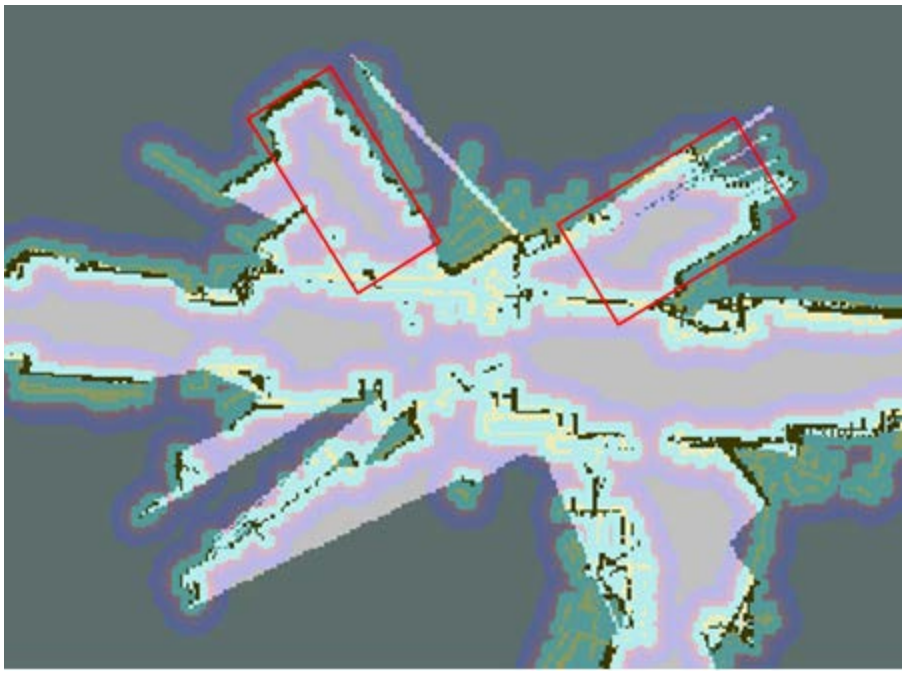

Figure 36. Image of costmap after the `move_base` package attempted in-place rotations during recovery procedures. The false hallways, highlighted in red, were produced from the skewed local map, placing inaccurate obstacles.

In order to attempt to rectify this issue, the `planner_frequency` parameter was changed. The default `planner_frequency` is set so that the global planner only creates a new global path when a new goal is received by the `move_base` package. It was adjusted to recalculate the global path at a rate of 0.1 Hz. While this increased the computational burden on the robot's processor, it allowed for adjustments to the global path to be made during operation. This allowed the robot to continually reevaluate its global path so when false obstacles were cleared or obstacles were mistakenly cleared in the local costmap, the robot could identify a new global path. Within the `costmap` nodes, the `obstacle_range` and `raytrace_range` were adjusted specifically for the capabilities of the Microsoft Kinect to better optimize the placement and removal of obstacles from the costmaps. Additionally, some of the `gmapping` package's parameters were adjusted, specifically `map_update_interval` and `minimumScore`. The `map_update_interval` parameter was changed to provide more frequent updates to the `slam_gmapping` occupancy grid. The `minimumScore` parameter was adjusted to

improve scan matching. Despite the issues with odometry errors, the robot was able to successfully autonomously navigate to the desired goal in an unknown and dynamic environment while conducting SLAM.

V. CONCLUSIONS

A. SUMMARY

The processes of the integration of ROS with the Pioneer P3-DX mobile robot platform and the Microsoft Kinect depth sensor to conduct SLAM and autonomous navigation in an unknown and dynamic environment were investigated in this thesis. The basic concepts of ROS and the capabilities of the hardware, namely the Pioneer P3-DX mobile robot and the Microsoft Kinect depth sensor, were explored. The appropriate ROS packages were downloaded, configured, and tested for the mobile robot, depth sensor, SLAM, and autonomous navigation. The SLAM components were tested as the mobile robot was wirelessly teleoperated from keyboard commands, creating a map of the environment. Next, the autonomous navigation components were investigated using the previously created map, the robot demonstrating the ability to avoid static and dynamic obstacles and achieve its goal. And finally, with the integration of ROS, the Pioneer P3-DX, and the Microsoft Kinect, autonomous navigation and SLAM in an unknown environment were successfully conducted resulting in the robot achieving its goal while identifying and mapping obstacles. As tests were conducted, parameters of the packages were adjusted to improve the system.

In conclusion, autonomous navigation and SLAM using ROS and an affordable depth sensor such as the Microsoft Kinect was achieved. ROS, with its modular architecture and object-oriented programming, provided a simple, yet robust, framework for the mobile robot platform, depth sensors, map construction packages, and autonomous navigation controllers to work harmoniously together in order to accomplish the objectives. Utilizing the `slam_gmapping` node, we constructed maps in real-time through teleoperation and autonomous navigation. The `move_base` package was adept in receiving information from a stored map, the `slam_gmapping` node, and scanned data, generated costmaps, calculated optimal local and global paths, and controlled the robot through a dynamic environment to reach its goal. With the modular aspect of ROS, as long as messages are published to topics with the correct format, the results obtained in this thesis research could be reproduced utilizing many different mobile robot platforms,

including other commercial off-the-shelf or laboratory-built robots, as well as different types of depth sensors.

B. FUTURE WORK

Since the field-of-view of the Microsoft Kinect is relatively narrow compared to a laser range scanner, the results obtained in this thesis research could be improved upon with the addition of multiple sensors, greatly improving the robot's ability to sense its environment. For example, the Pioneer P3-DX's array of 16 range-finding sonar sensors could be incorporated into the project, which would allow the robot to sense surrounding obstacles. Additionally, another depth camera could be added to the robot or within the environment of the mobile robot.

As well as increasing the number of sensors for the robot to sense its surroundings, another direction for improvement could be a better division of labor between the robot's processing unit and the base laptop. With latency being partially responsible for faulty navigation, a more effective spread loading of the large amount of computations associated with point-cloud data, SLAM, and controlling autonomous navigation should be investigated.

There are several three-dimensional SLAM packages available in ROS such as `cob_3d_mapping_slam`, `mrpt_ekf_slam_3d`, `RGBD SLAM`, and `octomap` packages. They were not chosen for this thesis research due the already strained computational load on the robot's processing unit; however, three-dimensional SLAM could be another interesting way ahead with this project, an example of which is shown in Figure 37.

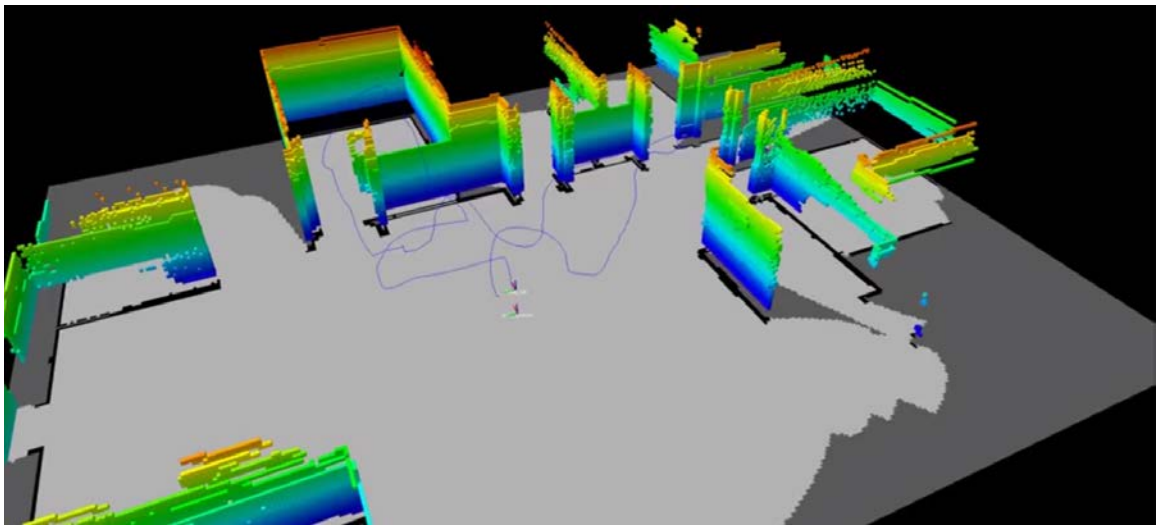

Figure 37. Example of three-dimensional SLAM utilizing the `octomap` package, from [41].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MASTER LAUNCH CODE

Master launch file -

(~/catkin_ws/src/follow_me_main/follow_me/launch/follow_me.launch)

```
<launch>
<!-- Defining the arguments -->
        <arg name="urdf" default="true" />
        <arg name="P2OS_Driver" default="true" />
        <arg name="enableMotor" default="true" />
        <arg name="keyboard_control" default="false" />
        <arg name="Kinect" default="true" />
        <arg name="skeleton_tracking" default="true" />
        <arg name="ConvertToLaserScan" default="true" />
        <arg name="RViz_Robot_View" default="false" />

        <arg name="fwd_vel_test" default="false" />


  <!-- Start p2os URDF -->
        <group if="$(arg urdf)" >
                <include file ="$(find p2os_urdf)/launch/pioneer3dx_urdf.launch" />
        </group>

  <!-- Start the p2os ROS Driver -->
        <group if="$(arg P2OS_Driver)" >
                <node pkg="p2os_driver" name="p2os_driver" type="p2os_driver" />
        </group>

  <!-- Enable p2os Motor State -->
        <group if="$(arg enableMotor)" >
                <node pkg="follow_me" name="p2os_enableMotor" type="p2os_enableMotor" />
        </group>

  <!-- Start keyboard teleop -->
        <group if="$(arg keyboard_control)" >
                <node pkg="follow_me" name="follow_me_teleop_keyboard"
type="follow_me_teleop_keyboard" />
        </group>

  <!-- Start the Kinect (openni_camera) -->
        <group if="$(arg Kinect)" >
                <include file="$(find openni_launch)/launch/openni.launch">
                         <arg name="camera" value="kinect" />
                         <param name="depth_registration" value="true" />
                </include>
        </group>

  <!-- Start Human Tracking (openni_tracker) -->
        <group if="$(arg skeleton_tracking)" >
                <node pkg="openni_tracker" name="openni_tracker" type="openni_tracker"
output="screen" >
                        <param name="camera_frame_id" value="kinect_depth_frame" />
                </node>
        </group>

  <!-- Start depthimage_to_laserscan -->
        <group if="$(arg ConvertToLaserScan)" >
                <node pkg="depthimage_to_laserscan" name="depthimage_to_laserscan"
type="depthimage_to_laserscan" args="image:=/kinect/depth/image_raw" respawn="true" >
                        <param name="scan_height" value="200" />
                        <param name="scan_time" value="0.125" />
                        <param name="range_min" value="0.45" />
```

```
                        <param name="range_max" value="7.0" />
                        <param name="min_height" value="0.05" />
                        <param name="max_height" value="1.0" />
                        <param name="output_frame_id" value="/kinect_depth_frame_laserscan" />
                </node>
        </group>

  <!-- Start Rviz for display of Robot Model with PointCloud -->
        <group if="$(arg RViz_Robot_View)" >
                <node pkg="rviz" name="rviz" type="rviz" args="-d
/home/ecejames01/.rviz/robot_view.rviz" />
        </group>


  <!-- Start moving Pioneer forward at 0.5 m/s (Test node) -->
        <group if="$(arg fwd_vel_test)" >
                <node pkg="follow_me" name="twist_test" type="twist_test" />
        </group>

</launch>
```

APPENDIX B. NAVIGATION LAUNCH CODE

Navigation launch file with pre-constructed map –
(~/catkin_ws/src/follow_me_main/follow_me_2dnav/launch/move_base_map.launch)

```
<launch>

  <!-- Start the map server -->
       <node pkg="map_server" name="map_server" type="map_server" args="`rospack find
follow_me_2dnav`/launch/mylaserdata_1503201800.yaml" />

  <!-- Run AMCL -->
       <include file="$(find amcl)/examples/amcl_diff.launch">
               <param name="transform_tolerance" value="0.2" />
               <param name="recovery_alpha_slow" value="0.001" />
               <param name="use_map_topic" value="false" />
               <param name="laser_min_range" value="1.0" />
               <param name="laser_max_range" value="7.0" />
               <param name="laser_likelihood_max_dist" value="2.0" />
               <param name="odom_model_type" value="diff" />
               <param name="odom_frame_id" value="odom" />
               <param name="base_frame_id" value="base_link" />
               <param name="global_frame_id" value="map" />
       </include>

  <!-- Start navigation stack -->
       <node pkg="move_base" name="move_base" type="move_base" respawn="false" output="screen" >
               <rosparam command="load" file="$(find
follow_me_2dnav)/params/costmap_common_params.yaml" ns="global_costmap"/>
               <rosparam command="load" file="$(find
follow_me_2dnav)/params/costmap_common_params.yaml" ns="local_costmap" />
               <rosparam command="load" file="$(find
follow_me_2dnav)/params/local_costmap_params.yaml" />
               <rosparam command="load" file="$(find
follow_me_2dnav)/params/global_costmap_params.yaml"/>
               <rosparam command="load" file="$(find
follow_me_2dnav)/params/base_local_planner_params.yaml" />
               <param name="controller_frequency" type="double" value="20.0" />
               <param name="planner_patience" value="5.0" />
               <param name="controller_patience" value="15.0" />
               <param name="conservative_reset_dist" value="5.0" />
               <param name="recovery_behavior_enabled" value="true" />
               <param name="clearing_rotation_allowed" value="true" />
               <param name="shutdown_costmaps" value="false" />
               <param name="oscillation_timeout" value="0.0" />
               <param name="oscillation_distance" value="0.5" />
               <param name="planner_frequency" value="0.0" />

       </node>

</launch>
```

Navigation launch file with pre-constructed map –
(~/catkin_ws/src/follow_me_main/follow_me_2dnav/launch/move_base.launch)

```
<launch>

  <!-- Start gmapping-->
        <node pkg="gmapping" name="slam_gmapping" type="slam_gmapping" output="$
                <param name="map_update_interval" value="8.0" />
                <param name="maxUrange" value="6.5" />
                <param name="maxRange" value="8.0" />
                <param name="odom_frame" value="odom" />
        </node>

  <!-- Start navigation stack -->
        <node pkg="move_base" name="move_base" type="move_base" respawn="false"$
                <rosparam command="load" file="$(find follow_me_2dnav)/params/c$
                <rosparam command="load" file="$(find follow_me_2dnav)/params/c$
                <rosparam command="load" file="$(find follow_me_2dnav)/params/l$
                <rosparam command="load" file="$(find follow_me_2dnav)/params/g$
                <rosparam command="load" file="$(find follow_me_2dnav)/params/b$
        </node>

</launch>
```

APPENDIX C. NAVIGATION PARAMETER CODE

Common costmap parameter .yaml file

```
obstacle_range: 4.0
raytrace_range: 5.0

footprint: [ [0.254, -0.0508], [0.1778, -0.0508], [0.1778, -0.1778], [-0.1905, -0.1778], [-0.254,
0], [-0.1905, 0.1778], [0.1778, 0.1778], [0.1778, 0.0508], [0.254, 0.0508] ]
inflation_radius: 0.55

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan, topic: scan, marking: true,
clearing: true, expected_update_rate: 0.2, obstacle_range: 5.0, raytrace_range: 5.0}
```

Global costmap parameter .yaml file

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  transform_tolerance: 0.5
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: true
```

Local costmap parameter .yaml file

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
  rolling_window: true
  width: 10.0
  height: 10.0
  resolution: 0.05
```

Base local planner parameter .yaml file

```
TrajectoryPlannerROS:
  max_vel_x: 0.8
  min_vel_x: 0.1
  max_rotational_vel: 0.8
  min_in_place_rotational_vel: 0.3
  escape_vel: -0.2
  acc_lim_x: 2.5
  acc_lim_y: 0.0
  acc_lim_th: 3.2
  holonomic_robot: false

  yaw_goal_tolerance: 0.15
  xy_goal_tolerance: 0.1
  latch_goal_tolerance: true

  sim_time: 2.0

  meter_scoring: true
  pdist_scale: 0.6
  gdist_scale: 0.8
  occdist_scale: 0.05
  dwa: true
  global_frame_id: odom
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. KEYBOARD TELEOPERATION CODE

Keyboard teleoperation node –

(~/catkin_ws/src/follow_me_main/follow_me/src/follow_me_teleop_keyboard.cpp)

```
#include <termios.h>
#include <signal.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/poll.h>

#include <boost/thread/thread.hpp>
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>

#define KEYCODE_W 0x77
#define KEYCODE_A 0x61
#define KEYCODE_S 0x73
#define KEYCODE_D 0x64
#define KEYCODE_Q 0x71
#define KEYCODE_E 0x65

#define KEYCODE_A_CAP 0x41
#define KEYCODE_D_CAP 0x44
#define KEYCODE_S_CAP 0x53
#define KEYCODE_W_CAP 0x57

class ErraticKeyboardTeleopNode
{
        private:
                double walk_vel_;
                double run_vel_;
                double yaw_rate_;
                double yaw_rate_run_;

        geometry_msgs::Twist cmdvel_;
        ros::NodeHandle n_;
        ros::Publisher pub_;

        public:
                ErraticKeyboardTeleopNode()
                {
                        pub_ = n_.advertise<geometry_msgs::Twist>(“cmd_vel,” 1);

                        ros::NodeHandle n_private(“~”);
                        n_private.param(“walk_vel,” walk_vel_, 2.0);
                        n_private.param(“run_vel,” run_vel_, 2.5);
                        n_private.param(“yaw_rate,” yaw_rate_, 0.75);
                        n_private.param(“yaw_rate_run,” yaw_rate_run_, 1.5);
                }

                ~ErraticKeyboardTeleopNode() { }
                void keyboardLoop();
                void stopRobot()
                {
                        cmdvel_.linear.x = 0.0;
                        cmdvel_.angular.z = 0.0;
                        pub_.publish(cmdvel_);
                }
};
```

```
ErraticKeyboardTeleopNode* tbk;
int kfd = 0;
struct termios cooked, raw;
bool done;

int main(int argc, char** argv)
{
        ros::init(argc,argv,"tbk," ros::init_options::AnonymousName |
ros::init_options::NoSigintHandler);
        ErraticKeyboardTeleopNode tbk;

        boost::thread t = boost::thread(boost::bind(&ErraticKeyboardTeleopNode::keyboardLoop,
&tbk));

        ros::spin();

        t.interrupt();
        t.join();
        tbk.stopRobot();
        tcsetattr(kfd, TCSANOW, &cooked);

        return(0);
}

void ErraticKeyboardTeleopNode::keyboardLoop()
{
        char c;
        double max_tv = walk_vel_;
        double max_rv = yaw_rate_;
        bool dirty = false;
        int speed = 0;
        int turn = 0;

        // get the console in raw mode
        tcgetattr(kfd, &cooked);
        memcpy(&raw, &cooked, sizeof(struct termios));
        raw.c_lflag &=~ (ICANON | ECHO);
        raw.c_cc[VEOL] = 1;
        raw.c_cc[VEOF] = 2;
        tcsetattr(kfd, TCSANOW, &raw);

        puts("Reading from keyboard");
        puts("Use WASD keys to control the robot");
        puts("Press Shift to move faster");

        struct pollfd ufd;
        ufd.fd = kfd;
        ufd.events = POLLIN;

        for(;;)
        {
                boost::this_thread::interruption_point();

                // get the next event from the keyboard
                int num;

                if ((num = poll(&ufd, 1, 250)) < 0)
                {
                        perror("poll():");
                        return;
                }
                else if(num > 0)
                {
                        if(read(kfd, &c, 1) < 0)
                        {
                                perror("read():");
                                return;
```

```
        }
}
else
{
        if (dirty == true)
        {
                stopRobot();
                dirty = false;
        }
continue;
}

switch(c)
{
        case KEYCODE_W:
                max_tv = walk_vel_;
                speed = 2;
                turn = 0;
                dirty = true;
                break;
        case KEYCODE_S:
                max_tv = walk_vel_;
                speed = -1;
                turn = 0;
                dirty = true;
                break;
        case KEYCODE_A:
                max_rv = yaw_rate_;
                speed = 0;
                turn = 1;
                dirty = true;
                break;
        case KEYCODE_D:
                max_rv = yaw_rate_;
                speed = 0;
                turn = -1;
                dirty = true;
                break;
        case KEYCODE_Q:
                max_tv = walk_vel_;
                speed = 1;
                turn = 1;
                dirty = true;
                break;
        case KEYCODE_E:
                max_tv = walk_vel_;
                speed = 1;
                turn = -1;
                dirty = true;
                break;

        case KEYCODE_W_CAP:
                max_tv = run_vel_;
                speed = 1;
                turn = 0;
                dirty = true;
                break;
        case KEYCODE_S_CAP:
                max_tv = run_vel_;
                speed = -1;
                turn = 0;
                dirty = true;
                break;
        case KEYCODE_A_CAP:
                max_rv = yaw_rate_run_;
                speed = 0;
                turn = 1;
```

```
                        dirty = true;
                        break;
                case KEYCODE_D_CAP:
                        max_rv = yaw_rate_run_;
                        speed = 0;
                        turn = -1;
                        dirty = true;
                        break;

                default:
                        max_tv = walk_vel_;
                        max_rv = yaw_rate_;
                        speed = 0;
                        turn = 0;
                        dirty = false;
        }

        cmdvel_.linear.x = speed * max_tv;
        cmdvel_.angular.z = turn * max_rv;
        pub_.publish(cmdvel_);
    }
}
```

APPENDIX E. MICROSOFT KINECT URDF CODE

Code added to p2os_urdf to include the Microsoft Kinect – (~/catkin_ws/src/p2os/p2os_urdf/def/pioneer3dx_body.xacro)

```
<!--Kinect Sensor-->
        <joint name="kinect_joint" type="fixed">
                <origin xyz="0.1397 0 0.2677" rpy="0 0 0" />
                <parent link="base_link" />
                <child link="kinect_link" />
        </joint>

        <link name="kinect_link">
                <inertial>
                        <mass value="0.001" />
                        <origin xyz="0 0 0" />
                        <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
                                 iyy="0.0001" iyz="0.0"
                                 izz="0.0001" />
                </inertial>
                <visual>
                        <origin xyz="0 0 0.028575" rpy="0 0 ${M_PI/2}" />
                        <geometry>
                                <box size="0.27796 0.07271 0.0381" />
                        </geometry>
                        <material name="Blue" />
                </visual>
                <collision>
                        <origin xyz="0 0 0" rpy="0 0 0" />
                        <geometry>
                                 <box size="0.27796 0.07271 0.073" />
                        </geometry>
                </collision>
        </link>

        <joint name="kinect_rgb_joint" type="fixed">
                <origin xyz="0.01905 -0.0125 0.02794" rpy="0 0 0" />
                <parent link="kinect_link" />
                <child link="kinect_rgb_frame" />
        </joint>

        <link name="kinect_rgb_frame">
                <inertial>
                        <mass value="0.001" />
                        <origin xyz="0 0 0" />
                        <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
                                 iyy="0.0001" iyz="0.0"
                                 izz="0.0001" />
                </inertial>
        </link>

        <joint name="kinect_rgb_optical_joint" type="fixed">
                <origin xyz="0 0 0" rpy="${-M_PI/2} 0 ${-M_PI/2}" />
                <parent link="kinect_rgb_frame" />
                <child link="kinect_rgb_optical_frame" />
        </joint>

        <link name="kinect_rgb_optical_frame">
                <inertial>
                        <mass value="0.001" />
                        <origin xyz="0 0 0" />
                        <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
```

```
                        iyy="0.0001" iyz="0.0"
                        izz="0.0001" />
        </inertial>
</link>

<joint name="kinect_depth_joint" type="fixed">
        <origin xyz="0.01905 0.0125 0.02794" rpy="0 0 0" />
        <parent link="kinect_link" />
        <child link="kinect_depth_frame" />
</joint>

<link name="kinect_depth_frame">
        <inertial>
                <mass value="0.001" />
                <origin xyz="0 0 0" />
                <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
                        iyy="0.0001" iyz="0.0"
                        izz="0.0001" />
        </inertial>
</link>

<joint name="kinect_depth_optical_joint" type="fixed">
        <origin xyz="0 0 0" rpy="${-M_PI/2} 0 ${-M_PI/2}" />
        <parent link="kinect_depth_frame" />
        <child link="kinect_depth_optical_frame" />
</joint>

<link name="kinect_depth_optical_frame">
        <inertial>
                <mass value="0.001" />
                <origin xyz="0 0 0" />
                <inertia ixx="0.0001" ixy="0.0" ixz="0.0"
                        iyy="0.0001" iyz="0.0"
                        izz="0.0001" />
        </inertial>
</link>
```

LIST OF REFERENCES

[1] H. Wei, Z. Huang, Q. Yu, M. Liu, Y. Guan, and J. Tan, "RGMP-ROS: a Real-time ROS architecture of hybrid RTOS and GPOS on multi-core processor," in *Proc. of 2014 IEEE International Conference on Robotics & Automation*, Hong Kong, China, 2014, pp. 2482–2487.

[2] T. K. Calibo, "Obstacle detection and avoidance on a mobile robotic platform using active depth Sensing," M.S. thesis, Dept. Elect. Eng., Naval Postgraduate School, Monterey, CA, 2014. [Online]. Available: http://calhoun.nps.edu/handle/10945/42591

[3] D. Thomas. (2014, May 22). ROS/introduction. [Online]. Available: http://www.ros.org/wiki/ROS/Introduction

[4] J. Bohern. (2014, Mar. 14). Packages. [Online]. Available: http://wiki.ros.org/Packages

[5] A. Romero. (2014, Jan. 21). ROS/concepts. [Online]. Available: http://wiki.ros.org/ROS/Concepts

[6] D. Thomas. (2014, Jan. 27). sensor_msgs/PointCloud2. [Online]. Available: http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html

[7] D. Forouher. (2014, June 1). Topics. [Online]. Available: http://wiki.ros.org/Topics

[8] K. Conley. (2012, Feb. 3). Services. [Online]. Available: http://wiki.ros.org/Servies

[9] S. Bishop. (2015, Jan. 8). Roslaunch/XML. [Online]. Available: http://wiki.ros.org/roslaunch/XML

[10] L. Joseph, Learning Robotics Using Python, Birmingham, UK: Packt Publishing, 2015, pp. 56-57.

[11] D. Thomas. (2013, Aug. 7). Parameter server. [Online]. Available: http://wiki.ros.org/Parameter%20Server

[12] I. Saito. (2015, May 2). Remapping arguments. [Online]. Available: http://wiki.ros.org/Remapping%20Arguments

[13] I. Saito. (2015, Feb. 21). Urdf/Tutorials/Create Your Own URDF File. [Online]. Available: http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file

[14] D. Stonier. (2014, Sep. 1). ROS/ tf. [Online]. Available: http://wiki.ros.org/tf

[15] B. Chretien. (2014, June 2). ROS/rviz. [Online]. Available: http://wiki.ros.org/rviz

[16] MobileRobots Inc. (2006). *Pioneer 3 Operations Manual*, MobileRobots Inc., Amherst, NH.

[17] H. M. Kahily, A. P. Sudheer, and M. D. Narayanan, "RGB-D sensor-based human detection and tracking using an armed robotic system," in *Proc. of the 2014 Int. Conf. on Advances in Electronics, Computers and Communications*, Delhi, India, 2014.

[18] A. Oliver, S. Kang, B. C. Wünsche, and B. MacDonald, "Using the Kinect as a navigation sensor for mobile robotics," in *Proc. of 27th Conf. on Image and Vision Computing*, Dunedin, New Zealand, 2012.

[19] Y. Wang, C. Shen, and J. Yang, "Calibrated Kinect sensors for robot simultaneous localization and mapping," in *Proc. of 19th International Conf. on Methods and Models in Automation and Robotic*s, Miedzyzdroje, Poland, 2014, pp. 560-565.

[20] K. Kamarudin, S. M. Mamduh, A. Y. M. Shakaff, S. M. Saad, and A. Zakaria, "Method to convert Kinect's 3D depth data to a 2D map for indoor SLAM," in *Proc. of IEEE 9th International Colloquium on Signal Processing and its Applications*, Kuala Lumpur, Malausia, 2013, pp.247-251.

[21] nav_msgs/Odometry Message. (2013, Jan. 11). [Online]. Available: http://docs.ros.org/diamondback/api/nav_msgs/html/msg/Odometry.html

[22] H. Allen, D. Feil-Seifer, A. Synodinos, B. Gerkey, K. Stoy, R. Vaughan, A. Howard, and T. Hermans. (2012, Sep. 29). ROS/ p2os_driver. [Online]. Available: http://wiki.ros.org/p2os_driver

[23] T. Sweet. (2014, Feb. 15). p2os/Tutorials/p2os with local computer USB communication. [Online]. Available: http://wiki.ros.org/p2os/Tutorials/p2os%20with%20local%20computer%20USB%20communication

[24] J. Kammerl and J. Binney. (2013, Nov. 14). ROS/openni_launch. [Online]. Available: http://wiki.ros.org/openni_launch

[25] P. Mihelich, S. Gedikli, and R. B. Rusu. (2014, Oct. 24). ROS/ openni_camera. [Online]. Available: http://wiki.ros.org/openni_camera

[26] T. Field. (2013, Mar. 5). ROS/openni_tracker. [Online]. Available: http://wiki.ros.org/openni_tracker

[27] K. Franke. (2013, Mar. 10). ROS/depthimage_to_laserscan. [Online]. Available: http://wiki.ros.org/depthimage_to_laserscan

[28] R. Siegwart and I. R. Nourbakhsh, Introduction to Autonomous Mobile Robots, Cambridge, MA: The MIT Press, 2004.

[29] R. Tang, X. Chen, M. Hayes, and I. Palmer, "Development of a navigation system for semi-autonomous operation of wheelchairs," in *Proc. of the 8th IEEE/ASME Int. Conf. on Mechatronic and Embedded Systems and Applications*, Suzhou, China, 2012, pp. 257-262.

[30] G. Grisetti, C. Stachniss, and W. Burgard, "Improving grid-based SLAM with Rao-Blackwellized particle filters by adaptive proposals and selective resampling," in *Proc. of the 2005 IEEE Int. Conf. on Robotics and Automation*, Barcelona, Spain, 2005, pp. 2432-2437.

[31] N. Kwak, I. Kim, H. Lee, and B. Lee, "Analysis of resampling process for the particle depletion problem in FastSLAM," in *Proc. of 16th IEEE Int. Conf. on Robot & Human Interactive Communication*, Jeju, Korea, 2007, pp. 200-205.

[32] H. W. Keat and L. S. Ming, "An investigation of the use of Kinect sensor for indoor navigation," in *Proc. of the 2012 IEEE Region 10 Conf. on Sustainable Development through Humanitarian Technology*, Cebu, Philippines, 2012.

[33] J. Santos. (2014, Aug. 6). ROS/gmapping. [Online]. Available: http://wiki.ros.org/gmapping?distro=hydro

[34] W. Woodall. (2015, May 6). ROS/costmap_2d. [Online]. Available: http://wiki.ros.org/costmap_2d

[35] Geometric reasoning and applications: robot motion planning. (n.d.). [Online]. Available: http://www-scf.usc.edu/~peiyingc/gra_planning.html. Accessed May 5, 2015.

[36] Motion planning. (n.d.). *Wikipedia*. Available: http://en.wikipedia.org/wiki/Motion_planning. Accessed May 5, 2015.

[37] D. V. Lu and M Ferguson. (2014, Oct. 5). base_local_planner. [Online]. Available: http://wiki.ros.org/base_local_planner

[38] E. Marder-Eppstein. (2014, Aug. 1). ROS/move_base. [Online]. Available: http://wiki.ros.org/move_base

[39] arebgun/erratic_robot. (2011, Jan. 3). GitHub. [Online]. Available: https://github.com/arebgun/erratic_robot/blob/master/erratic_teleop/src/keyboard.cpp

[40] B. P. Gerkey. (2015, Feb. 28). ROS/amcl. [Online]. Available: http://wiki.ros.org/amcl

[41] (2013, Aug. 22). “3D SLAM on Simulated Multirotor UAV.” [YouTube video]. Available: https://www.youtube.com/watch?v=quqF5_ZE_fI. Accessed May 5, 2015.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California