View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DigitalCommons@WPI

Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) Major Qualifying Projects

January 2015

Motion Planning of Intelligent Robots

Parham Salimi
Worcester Polytechnic Institute

Shamiah Dominique McDonald
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Salimi, P., & McDonald, S. D. (2015). *Motion Planning of Intelligent Robots*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/1729

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Motion Planning of Intelligent Robots

A Major Qualifying Project Report:

Submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

In Electrical and Computer Engineering by:

____________________________ ____________________________

Shamiah McDonald Parham Salimi

Date: December 13th, 2014

In partnership with

Shanghai University

Zhiting Qian, Yue Yuan, JianQiang Zhou

Approved by:

____

Professor Yiming (Kevin) Rong, Project Advisor, WPI

____

Professor Xinming Huang, Major Advisor, WPI

Dr. Wang, Co-Advisor, SHU

This report represents the work of one or more WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review.

Abstract

Robotics is a fast growing industry that is used in everyday life. One of the most popular is intelligent mobile robots that are used for basic conventional use. The purpose of this project is to use the Turtlebot 2 to map and navigate its environment, while avoiding obstacles. Also to incorporate human machine interaction by using gesture control. This report details the research, setup, and programming process of the robot.

Acknowledgements

This project was done in partnership with the student, faculty and staff of Shanghai University (SHU). We would like to extend our appreciation and gratitude to our SHU partners and co-advisor for all their help. We also would like to thank Professor Yiming (Kevin) Rong and our school Worcester Polytechnic Institute (WPI) for making this project possible and making the China project center one of the best experiences.

Table of Contents

Abstract 2

Acknowledgements 3

1. Introduction 7

2. Background 8

2.1 History of Robots 8

2.2 Autonomous Robots 9

2.3 Filters 11

2.4 Simultaneous Localization and Mapping (SLAM) 14

2.5 Robot Gesture Control 15

2.6 Robot Voice Control 17

2.7 Turtlebot Research 18

2.7.1 Kobuki 18

2.7.2 Kinect 20

2.7.3 Robot Operating System (ROS) 22

3. Methodology 23

3.1 Environmental Modeling 23

3.2 Navigation 24

3.2.1 Simultaneous Localization and Mapping (SLAM) 24

3.2.2 Gmapping 25

3.2.3 Adaptive Monte Carlo Localization (AMCL) 26

3.3 Human Machine Interaction (HMI) 27

3.3.1 Gesture Control.. 27

3.3.2 Voice Control 28

4. Results and Analysis 30
5. Conclusion 32
Work Cited.. 33
Appendix A 39

How to Setup ROS 39

Turtlebot Bringup Launch Code 42

Navigation (AMCL) Launch Code... 43

Gesture Control 44

Launch Code 44

Python Code 45

Voice Control 48

Launch Code 48

Python Code 48

Appendix B 54

Mapping and Navigation 54

Skeletal Base Tracking and Gesture Control 55

Testing Virtual Turtlebot 57

Appendix C 59

Turtlebot Specifications 59

Project Schedule 60

1. Introduction

Robots are becoming more popular as the decades go by. There are many different uses of robots from manufacturing to search and rescue. Robots are becoming so integrated in our everyday life that we fail realize how much we now depend on them. From completing missions in far outer space to taking over jobs that are deemed too dangerous for humans. Robotics is rapidly growing in this world.

One field of robotics is motion planning, this is the basis of mobile robots. Mobile robots use motion planning to observe their environment and avoid obstacles. There are different algorithms used for path planning, these algorithms help the robot determine the best route possible to avoid one or multiple obstacles. Depending on the type algorithm the best route is then decided by either shortest path, farthest distance from obstacle(s), and etc. There are many different algorithms that can be used to create the best path for the robot.

Most mobile robots are used for mapping and navigation. It’s common for these robots to be used for mostly indoor use, mapping the floor plan of a house or office building. The robot can then navigate through the building using the saved map. The robot has to be given a specific destination point or it can be controlled remotely by keyboard.

In this project we used the Turtlebot 2 for mapping and navigation. We created various maps of different levels of our lab building and directed the Turtlebot to navigate through it to avoid obstacles. Also, the robot was able to self-localize itself in its environment. In addition, we were able to incorporate gesture control to make the robot start and stop following, and to

go into tele-operation mode. We also implemented voice control to initiate the drive forward, back, and stop.

2. Background

2.1 History of Robots

The thought of robots have been around for centuries, starting with ancient mythologies. In ancient myths robots where connected to something spiritual and powerful, like the mechanical handmaidens built out of gold by the Greek god Hephaestus or like the Indian king who hid Buddha relics underground and guarded them with mechanical robots called bhuta vahana yanta. Beside the mythologies people have been testing out robots from the early ages.

Ideas similar to robots have been around since 4th century BC. People have been fascinated with automation for centuries, like the Greek mathematician who proposed a mechanical bird which was propelled by steam. In ancient China a clock tower was built in 1088, it contained mechanical figurines that would chime the hours, by ringing bells and other devices. Throughout the centuries many have drawn sketches of robots or have attempted to build one.

It wasn’t until 1810 that the first humanoid robot was built by Friedrich Kauffman, it was a soldier with a trumpet. The first electronic autonomous robots that could complete complex behavior were built in 1948 and 1949 by William Grey Walter. These robots were named Elsie and Elmer, they were constantly compared to tortoises because of how slow they

moved. These two robots were capable of finding their way back to a charging station when they had a low battery. The first digitally operated and programmable robot was named Unimate and was built in 1954 by George Devol. This robot was sold to General Motors and was installed in their factory to lift hot pieces of metal from a die casting machine and stack them. This robotic arm set the foundations and paved the way for modern robotics.

The Three Laws of Robotics were formulated in 1941 and 1942 by Isaac Asimov, this is how the word "robotics" was created. Norbert Wiener presented the basis of practical robotics in 1948. The way of robotics have been set from the past for us, not only leaving the learning and research for scientist but also inviting amateurs. These beginners can learn and implement code and help take robotics to a different level.

2.2 Autonomous Robots

Science and technology is advancing gradually every day, with autonomous robots being one of the main technologies growing so rapidly. An autonomous robot can achieve tasks/behaviors to a certain extent independent of human interaction. Autonomous robots were mostly used in factories, but now there is a side for building these robots to be used in space explorations, delivery services, and households.

These robots have to learn how to adapt to different situation in their environment. They must learn and gain new knowledge, this helps them adjust to changes in their environment. Autonomous robots live by a certain set of rules, first rule the robot has to be able to gather information from the environment it's in. The second rule is the reason it is autonomous it has to be able to operate without human intervention for certain amount of time. The third rule is

the robot must be able to operate in the environment without the aid of humans. Lastly it must not do harm to itself, people, or property unless it instructed to do so. Autonomous robots like any other machine still require routine maintenance.

Autonomous robots are expanding in their use, from being used in space missions to every day household use. Missions to Mars have been accomplish because of these robots, we were able to gather data and get information back from these robots about a planet that is about 4 months to a year away. Of course the robots do not last long on Mars, because of different obstacles the robot may run into like craters, or simple just breaking down. People are also starting to use autonomous robots in the house for cleaning purposes, like the iRobot that vacuums the house, avoids obstacles and is able to find its charging station when low on battery. They are also robots like the pi-robot and turtlebot that people program to be used as almost personal maids to bring drinks and snacks. These robots are also used for research for mapping a completely new environment and gathering information.

These robots are called autonomous because they can take care of themselves using basic sensory. They can detect when they need to return back to their charging station, while also detecting thermal, optical, and haptic sensing. The sensing capabilities of these robots are advancing; detecting altitude, odor, temperature, sound, touch, and more. These robots must be able to store and learn from the data it collects. For example like the robotic lawn mowers, detect the rate of grass growth and adapt their programming to cut the grass at that rate so it will always remain perfectly cut. In addition of collecting data, robots must be able to execute the task that is given, depending on the situation the robot must react differently.

As stated before autonomous robots can be used indoors and outdoors. While indoor use is easier to program and manage, research have been exploring the advantages of outdoor use. Unmanned aerial vehicles (UAVs) are the most popular when it comes to outdoor navigation, this was achievable because there are less obstacles in the air. Researchers are now looking into autonomous boats and ground vehicles, but the problem with this is the different obstacles possibilities. Weather is a main factor, also 3D terrain, surface density, and lastly the inability of the sensing the environment. With these factors put into place, it makes it harder to create a robot that can stand on its own in environmental situation, where even mankind does not know how to handle it. We don't want to make autonomous robots to independent in fear that they may become to intelligent, and also in fear they may start foraging for their own resources.

2.3 Filters

There are various kind of filters used in robotics, these filters are usually based off of different mathematical equations. Filters are made to help algorithms sort through data points and samples. As stated before there are several different types of filters, but they all narrow down to about four different types. These filters all work differently but achieve the same end goal of filtering, some even work together to get a more accurate result.

The first and most common filter is the Particle filter, which is also known as Sequential Monte Carlo (SMC). This filter implements the Bayesian recursion equation, establishing methodology for generating samples from required distribution. It is set of on-line posterior density estimation algorithms that estimate the posterior density of the state-space, the state of

space can be linear or non-linear. Initial state and noise distribution can be taken in any form required, but does not work well with high dimensional systems. Samples from the distribution are represented by a set of particles, each particle has weight assigned to it. A common issue is weight disparity leading to weight collapse, this can be fixed by including a re-sampling step. This re-sampling step is achieved by replacing particles with negligible weights with new particles. The objective of a Particle Filter is to estimate the posterior density of the state variables given the observation values.

The Kalman Filter also known as the Linear Quadratic Estimation (LQE), is another filter that uses a series of measurements observed overtime. This method allows the filter to produce more precise estimates of unknown variables. It operates recursively on a streams of noisy input data that produces a statistical optimal estimate of the underlying system state. This filter is commonly used in guidance and navigation control in vehicles, aircrafts and space crafts. This filter uses only present input and can run in real-time. There is a two-step process that makes this filter work, the first step is the called the prediction step which produces estimates of the current state variables and the uncertainties. The second step then updates the estimates using a weighted average, more weight is given to the estimates with higher certainty.

As an extension to the previous filter there is the Extended Kalman Filter. This filter is known as the non-linear version of the Kalman Filter, building off the ideas of the previously mentioned filter, but advancing it to be used in non-linear situations. In order for this filter to work it uses the Taylor series expansion to linearize itself, this way it come out with accurate estimates of its non-linear data. Although used very often in today’s world, it is not an optimal

estimator. Extended Kalman Filter is the running standard in GPS and navigation systems. This filter is commonly used and combined with the Kalman Filter, they depend on each other to produce more accurate results.

Last is the Rao-Blackwellized Particle Filter, which is known as Marginalized Particle Filter and Mixture Kalman Filters. Besides the Particles Filter, this filter is very commonly used in robotics. As by the name given above this filter incorporates both the Kalman Filters and particle filter. This filter is based on the Rao-Blackwell theorem, which states any convex loss function improves if a conditional probability is utilized, provided if a process by which a possible improvement of an estimator can be obtained by taking its conditional expectations with respect to a sufficient statistic. This filter improves performance when a linear Gaussian substance is present. It decreases the number of particles necessary to achieve same accuracy with regular Particle Filter. Randomly selects particles according to prior distribution, calculating new weights. It weighs samples and re-sample to obtain an update of distribution of particles. Estimates X(t) using optimal filter, evidence, and previous location, it then simplifies calculations by giving one "free" localization with an optimal filter.

As said before, all four of these filters can be used separately and individually, but they can also all be integrated to work together. These filters tend to be used together manipulated in different kinds of to produce very accurate data. These precise data points are then used with the SLAM algorithms to produce maps and help with navigation. The more accurate the data the more accurate the maps, the more accurate the robot.

2.4 Simultaneous Localization and Mapping (SLAM)

Algorithms are very important in robotics, they are basically like the brains of the robot, breaking down everything for the robot. There are algorithms for path planning, for mapping, and etc. SLAM is one of the most popular algorithms used in robotics, its main responsibilities are to help robots map and localize itself at the same time. Additionally, there are four different types of SLAM, the most common one is gmapping.

Gmapping is one of the more popular SLAM algorithms used in robotics, it is also the default used on the turtlebot. Pairing with the Rao-Blackwellized Particle Filter, it uses the filter to sort out laser data. It then takes into account the altered movements and the recent observations of the robot. By doing this the algorithm decreases the chances of uncertainty in the robots pose for the prediction step of filtering. In addition, it carries out re-sampling operations selectively, reducing the particle depletion problem.

Another SLAM algorithm that uses the Rao-Blackwellized Particle Filter is GridSLAM. This algorithm uses scan matching to correct and minimize odometric errors. A probabilistic model of residual errors is used to resample steps of the scan matching process. By accomplishing this the number of samples required for the algorithm is reduced. GridSLAM is not used as often, because of the many complications that occur when using it.

DP-SLAM is an algorithm that aims to achieve true SLAM without landmarks, this algorithm is very accurate and does not need to complete a loop to correct a map. Additionally, it uses just a normal Particle Filter, to filter out the data. Since, DP-SLAM is so accurate, it only make a single pass over the sensor data. DP-SLAM maintains it accuracy by

keeping uncertainties about a map over several time steps, this prevents errors within the map. This is completed by using a novel map data structure.

Lastly, FastSLAM is the fourth and final algorithm of SLAM that produces very accurate maps in large environments. Using the Extended Kalman Filter (EKF), Kalman Filter, and also a modified Particle Filter that is similar to Monte Carlo Localization (MCL). This algorithm recursively estimates the full posterior distribution of the robot pose and landmark location. It then scales logarithmically with the number of landmarks in the map.

Out of all the four SLAM algorithms FastSLAM and Gmapping are the most common used, Gmapping being the most common out of the two. Gmapping is the most common because is the most simplistic and error free. With our project using the turtlebot Gmapping with the Rao-Blackwellized Particle Filter is the default for the robot. It is very easy to implement on various robots and operating systems.

2.5 Robot Gesture Control

There are various ways to accomplish robotic gesture control, and several more ideas being created. The purpose of using gesture control is to make robot systems more user friendly by providing a more natural way to control them. One of the more common ways is using image processing to complete the task, there are several ways to use image processing for gesture control. One way is to use a webcam to capture real-time video of different hand gestures to generate commands. The images are then processed and sent to the robot where it then determines the command.

Additionally, gesture control can be accomplished by analog flex sensors. These sensors are placed on a glove or a pair of gloves; it then detects and measures the bending of the fingers. The measurements received from the glove is then processed, it then determines what gesture is given by the data obtained. Each gesture command is linked to a finger or multiple finger movements, that data is then transferred to the robot, where the robot then completes the command given.

Microsoft Kinect is most popular today for completing gesture control, using its RGB and Infra-Red camera correspondingly. The Kinect combined with SDK uses skeletal based control, recognizing different joints from the head down to the feet. Every joint is labeled and recognized by the Kinect after you configure the Kinect to recognize you with a pose. You then can initiate follow mode or program in for gesture control. Since every joint is labeled it makes it easier to implement code for gesture control with the Kinect.

Gesture control is becoming increasingly popular in the world of robotics, this being a way to control a robot without actually using a keyboard or joystick. Especially with the use of the Microsoft Kinect or Asus Xtion Depth camera, these devices have made the use of gesture control slightly easier than before. With the use of gesture control becoming more common, human machine interaction has become simplistic, using only ones hands or feet to interact with/control the robot.

2.6 Robot Voice Control

Today in robotics robots can be controlled numerous ways, whether it be through keyboard, joy stick, gesture, or voice. Robots today are becoming more user friendly, becoming simpler to interact with. Instead of the user using code line by line to control robots, user can now use their voice. Voice control makes it easier to interact with robots, and broadens the horizons of who can use robots.

Many voice controlled robots today are used in the home or in medical facilities, though still in the experimental phase, voice control is being improved. Robots can now complete simple task by just using your voice and no further interaction. This lets robots be used by user who may have disabilities that cannot use their hands or feet for keyboard and gesture control. Though voice control only lets a robot complete simple task it's enough to be a great help to someone that may need it.

Just like gesture control, voice control can be integrated with the other ways to control the robot. One can use voice, gesture, and keyboard control on the same robot if the code is implemented correctly. The robot can also seamlessly switch between each different type of control, by using the previous control. For example, one can use voice control to go back into gesture control and vice versa.

There are still some flaws with using voice control

As of now voice control is mostly used for moving the robot from place to place, either by saying a command like "forward, back, stop" or using a complete sentence to tell the robot

what specifically to do like “go get me a soda”. Each command is attached to a different string of code that incurs a different reaction from the robot. If the robot does not understand the command it will then read as an error and will not perform any type of action. The user will then have to repeat themselves more clearly for the robot to understand the command and process the type of action it needs to complete.

The future of voice control looks truly positive, the various different ways that it can be implemented are endlessly. There are so many ways that voice control can be used, not only controlling a robot to complete simple tasks, as moving back and forward, but in doing more advanced jobs. Voice control can be used in the research field to tell a robot to map a specific area, or give the coordinates by voice for search and rescue, the possibilities are endless. Furthermore, as robots are being used more and more in the household, voice control can help in families that have an elderly family member, become a companion, or even take over the simple tasks of an in-home nurse. The uses are vast when it comes to robotics and voice control.

2.7 Turtlebot Research

2.7.1 Kobuki

Kobuki is a mobile base that has motors, sensors and a power source, but although it has all these things it cannot operate by itself, it needs additional software and hardware. For the software it needs one to build their own software or one to implement software from other groups. For the hardware the mobile base needs a platform of an embedded board or a

netbook that runs Windows or Linux. In order for the Kobuki to be completely functional one may want to add additional sensor.

Being just a mobile base, most people setup their own platform based off of the Kobuki creating their own homemade robot with rudimentary functions and goals. These robots tend just to navigate around an open space to more advance function of mapping and navigating through multiple rooms. Most people tend to create their own platform with the Kobuki base, but there are premade platforms like the turtlebot 2.

The Kobuki base comes standard with a decent amount of parts like a brushed DC motor. The motor is controlled by pulse-width modulation and driven with voltage source H-bridge. The base has a 3-axis digital gyroscope, this come with the yaw axis factory calibrated within the range of ±20 degrees/s to ±100 degrees/s. This base is built to be used with a variety of items from custom boards to powering the Kinect. There are many different power and IO ports on this base making it very versatile.

Turtlebot is a common platform used with the Kobuki base, this has helped further the uses of the Kinect and the Kobuki base. With these two machines integrated together, one can accomplish 3D mapping with the Kinects different camera sensors and the mobility of the Kobuki base. The turtlebot 2 can perform various task like mapping, navigation, follow mode, gesture control and etc. Since the turtlebot runs on open software it makes it simple to code and integrate different programs through ROS.

User are not required to use the turtlebot platform, if they do not wish to. Many user create their own platform for the kabuki base making the system more tailored towards

personal needs. This maybe be easier for some users, creating their own platform that they understand and add only the features that are needed for that robots use. The downside of creating a platform is the debugging process and the many errors the user may acquire. Also there are many forums and websites that can give help if one is using the turtlebot or another known robot. When building your own platform there isn't many resources to help when you run into trouble.

Mobile bases like the Kabuki are very popular in robotics because of the way they easily integrate with different devices, and how easily they can be adapted to something new if needed. There is also so many way that mobile bases can be manipulated when used with different machinery. Although they are not great for outdoor use, because of the unevenness of the ground and different surface textures, they are great for indoor use, even when there is slight difference in surface texture.

2.7.2 Kinect

Microsoft developed the Kinect in order to make the gaming experience with the Xbox 360 more interactive. They wanted the user to feel like they were actually in the game, controlling every aspect. Kinect was initially created to go up against the already broad market of motion game control like the Wii remote and PlayStation Move (Eye), these motion game controllers were meant to widen the audience of the gaming universe. While gaming in the past required the user to be seated in front of the TV, the Kinect required users to get up and become active. When Microsoft released the Kinect they did not realize the different capabilities of it.

Kinect was not only bought for its advance in gaming capabilities, people starting buying Kinect because they realized the many different uses of the camera. With its infrared detection, depth, and regular color camera, some users saw that the Kinect could be used for other uses, for example like on robots. It wasn't until February 2012 that Microsoft came out with a Kinect that could be used with windows and from there the research with Kinect grew. SDK on windows was created to help users program and create apps for the Kinect.

There are three different cameras on the Kinect a depth, an infrared and a color, these cameras are what helps a robot see. The Kinect has been used on numerous robots from the pi-robot to the turtlebot, before the Kinect it was very expensive to put a camera on a robot with the same capabilities as the Kinect. There were laser cameras, infrared cameras, and other sensors combined on a robot to help create 2D and 3D maps. So, instead would build there robot blind or use cheap sensors that would not work as great. On top of the Kinect being the eyes of the robot, it has help generate more complicated tasks for the robot.

Everyday robots are developing rapidly, with the help of Kinect robots have been able to complete more complicated goals. With the Kinect one can get a visual of the person standing in front of the sensor in different modes, from infrared to skeletal. It can also recognize the different joints from the head to feet, this makes it easier to program with the Kinect, one can use the different joints labeled and implement them in the code to achieve follow mode or even gesture control. Moreover the Kinect recognize different user, meaning more than one person can control the robot, it can also detect multiple users at once. With these different interfaces it makes it possible to control a robot without the use of any devices.

2.7.3 Robot Operating System (ROS)

Originally developed in 2007 under the name of switchyard the robot operating system (ROS) has been around for a decent amount of years. The purpose of ROS is to make writing robot software easier and less complex, making a collection of libraries, tools, and conventions. ROS can be use on a wide variety of robotic platforms, this makes it simple to program different robots, without the headache of learning a completely different system. With it being open source ROS has helped different groups collaborate and build on top of each other to build better robot applications. ROS is built by the community, not just by one group.

The main client libraries of ROS: C++, Python, and LISP are usually geared to Unix-like systems, mainly because of their requirement for vast amounts of open source software. Ubuntu Linux is supported for ROS, while other operating systems like Microsoft Windows and Mac OS X are listed as experimental. Any operating system that is listed on the ROS website can be used, but for the ones listed as experimental one may run into bugs and errors as they get more into depth with their programming.

Some common packages and program on ROS is ROS visualization tool (RViz), this is a program that lets one see where their robot is at all times. Whether the user is building a map or driving the robot around the room, this program enables the user to see what the robot is seeing. RViz is also very useful for beginners in the ROS system it allows them to simulate different function before even connecting to an actual robot. One can map, navigate, and path

plan with the simulated robot in RViz. This helps a user become used to the application, that way when they use a physical robot, they know how to use the various functions.

ROS offers different packages and libraries for user to integrate with their robot system. Users can also make their own libraries and packages, then share them with ROS community. The point of ROS is for users to learn and share information about robotic codes and programming. It makes it simpler for user to implement different function on their robot, whether it be the pi-robot, turtlebot, or a robot platform they build themselves. ROS is meant for everyone in the robotic world to use.

3. Methodology

3.1 Environmental Modeling

We set numerous goals for our project with the turtlebot, one of the goes being accomplishing environmental modeling. Environmental modeling is when one gets a sense of the area around them. In the case of robotics this means having a robot get an image or build an image of the environment that is around it. This map image can then be saved and used for future purposes.

In order for the turtlebot to accomplish environmental mapping we first have to activate the kabuki mobile base and the Kinect camera through the ROS system. We would also have to open up RViz which stands for ROS visualization tool, this is used on the computer to show the map as it is being built, it also helps control the robot. A map can be built several ways, depending on the way the user setup the robot.

My Partners and I decided to setup the map of our lab hallway manually using a wireless keyboard to drive the robot as it created the map. This was a little hard at first, when creating a map one has to keep a steady speed. Also with the turtlebot using the Kinect, it cannot see when an object is too close to it, to specific 0.5meters or closer. When driving the robot we had to keep close to the wall but not to close. Creating a map can be a long drawn out process depending on the size of room. The hallway we used took about 10-15 minutes to map because all though it was narrow it was very long.

Though mapping may sound easy we did incur some problems. One problem that we encountered is when doors were open, the robot would leave the doorway as an open space on the map. At first we didn't think it was a problem, but when we moved down to navigating the robot it thought that the open space was different path around an obstacle. The turtlebot would then try to investigate the doorway and try to go through it. We then realized that when we our creating maps that we must make sure that we close all doors in the hallway, unless we want the robot to go into that particular room.

3.2 Navigation

3.2.1 Simultaneous Localization and Mapping (SLAM)

First of all to be able to control the motors and wheels of the base controller we need to publish a movement command and be able to translate it to motor signals. ROS uses Twist message types to publish motion commands to the base controller. The topic that the message from the base controller subscribes to is called /cmd_vel which stands for command velocities. The different XYZ coordinate geometry messages are sent to the controller. One is

for the linear velocity and the other is for the angular velocity. For our purposes we will only be using a two-dimensional plane such as the floor, therefore we only need the x linear velocity and z angular velocity. There are three main packages in ROS that we used to help us setup the navigation stack. Move_base is used to drive and control the robot for moving it to a chosen goal. Gmapping is used to create a map of an unknown environment using the scanned data from a depth camera that in our case is the Kinect. The main part of the navigation stack is the AMCL package that enables the robot to localize itself using an existing map. By setting up the navigation stack we will be able to command the robot to go to any location within the map, while avoiding obstacles.

3.2.2 Gmapping

Maps in ROS are basically a bitmap image representing an occupancy grid, where white pixels represent free space, black pixels represent obstacles, and grey pixels indicate "unknown". Therefore it is very easy to build a map using any simple graphical programs, or use a map that someone else has created. Since our Turtlebot is equipped with a depth camera, we can create our own map by using Gmapping. In our case we used slam_gmapping node that combines the data from the depth camera and odometry into an occupancy map.

The strategy of building the map is first to tele-operate the robot using a keyboard or voice control, which we will discuss further in this report. Basically we first tele-operate the robot around an environment that we will be using to navigate in, and then the slam_gmapping node is used against the recorded data to generate a map. The map is then saved in a file and opened through AMCL package to complete our navigation task.

3.2.3 Adaptive Monte Carlo Localization (AMCL)

Monte Carlo Localization (MCL) is a package commonly used in robotics. There is several different types of this package, it is usually used with different algorithms and filters to help robots complete different tasks and goals. The AMCL is one of the ROS packages used on the turtlebot, what is particularly special about this package is that it helps the turtlebot self-localize itself when it is navigating through different environments.

Adaptive Monte Carlo Localization is one of the many ROS packages used in robotics. This package stands out as stated before it lets the robot self-localize itself. This is accomplished by using the current scan of the environment and odometry data taken to pinpoint where the robot is. This then allows us to point and click on our previously built map to a new location. The robot will then travel to the designated destination while avoiding obstacles.

This package is paired with SLAM gmapping, as explained before gmapping is used to build the map of the environment. These two combined lets the robot know the environment it is in and localize itself. The robot can then go to any destination it is given in the mapped environment it is given and successfully avoid obstacles.

3.3 Human Machine Interaction (HMI)

3.3.1 Gesture Control

We were able to enhance the turtlebot with the use of gesture control mode. The way we implemented gesture control was a little different then how it is usually used with robots. Majority of the time when one thinks of robotic gesture control, they think that the robot will move based on a certain gesture, which in most cases is true. We implemented gesture control to recognize a preprogrammed gesture, and based on the gesture given the robot would go into a different operation mode. There were three different modes follow, voice control, and navigation. Depending what gesture was given the robot would then open up the program for whichever mode that needed to be ran.

With our specific code for gesture control when the left hand was raised the turtlebot would run the voice control mode. This then enabled the robot to listen out for keywords, which would tell it what to do next. If the right hand was raised the turtlebot would then run follow mode, which meant the Kinect camera would locate and calibrate the person in sight. The code for follow mode would then run and the robot would follow the person who was insight. The last gesture is arms crossed, when the user insight does this the robot will then run the code and open the application RViz which is used for navigation. The user can then give set navigation goals for the robot and watch it go.

To be able to accomplish gesture control we had to first install the necessary drivers like OpenNI, this package was extremely buggy and did not work alone. We had to fix dome problems and install a complementary bug fix package called NITE, this package was used

with OpeNI to process Kinect data for skeleton tracking and joint tracking. Next we had to develop a code to be able to recognize the gestures we wanted to use, by processing the skeleton data that was acquired by skeleton tracking. This was done but completing some calculations to get the gestures we needed. We would define the gestures and check for the specific gestures in a basic if and else if statement, for example to check if the left hand was up, we would check for normal (confident) position first and compare it to the position where the right hand was raised, based on the shoulder width. The same steps were taken to check if the gesture command is done and then run the appropriate launch file created.

There were some difficulties implanting the gesture control on the turtlebot using the Kinect camera. For some reason the camera would not always recognize a person standing in front of it, sometimes a person would have to stand completely still for the camera to calibrate. Another problem was the camera took a decent amount of time to recognize a gesture, this became gruesome do to the fact that sometimes a person could be standing there for two minutes plus, doing the gesture repeatedly. This was a bug more with the Kinect camera and also our code delaying, while trying to open the correct application to run the different modes.

3.3.2 Voice Control

It was decided during the project that we would add voice control to control our turtlebot. This application can be useful, directing the robot by voice instead of by keyboard teleop or gesture. We also used voice control to transition into other modes like follow and keyboard teleop, this feature allowed our robot to be more interactive with humans. We used a

Bluetooth headset to talk to the robot, this allowed the turtlebot to hear our voice in any location. The only problem with Bluetooth headset is we had to speak directly into the microphone for the robot to hear the command. Voice control could be also be accessed through gesture control like stated in the previous section. Before we could officially implement voice control there were some additional steps that had to be completed.

In order to achieve voice control we had to download some necessary packages onto our platform. The three main packages were gstreamer0.10-pocketsphinx, ros-groovy-audio-common, and libasound2; these packages where used to activate the turtlebot's audio and ability to listen. Next we used the pocketsphinx package form University of Albany's rharmony stack which enabled us to be able to recognize voice commands and create necessary vocabulary needed for our purposes to control the turtlebot. Nav_commands.txt file was modified and the words that we deemed necessary were added to the file.

We used a special website to create and modify the dictionary and pronunciation files (http://www.speech.cs.cmu.edu/tools/lmtool-new.html). The voice_nav.py was modified to add the commands needed to enter the follow and keyboard teleop mode. The OS and signal library was used to launch other files inside the same python code, Os.system was used to launch file name as a string, where Os.kill was used to shut down the current process running; these commands were added by adding a simple if statement. A specific launch file voice.launch was created to run the necessary files used to control the turtlebot through voice commands.

4. Results and Analysis

The turtlebot is capable of mapping its environment and recognizing its obstacles. The robot is able to navigate through its mapped environment and successfully avoid obstacles. It can be successfully controlled by user through keyboard, gestures, and voice. All of these allowed the robot to be able to follow a human in follow mode.

The robot could successfully map its environment, giving the result in an occupancy map, showing the obstacles and free space. This allowed the robot to calculate where it could and could not go. Going into navigation mode, when given set navigation goals the turtlebot could navigate through its known environment avoiding pre-recorded and new obstacles. With keyboard control we could drive the robot around to map its environment and avoid unknown obstacles.

In voice control mode the turtlebot could recognize a human voice, determine and understand what commands had been given. After the command word is identified the robot would perform the giving task, whether it be a driving task or going into a different mode. We used a Bluetooth headset to speak to the robot, so it could pick up the user voice in any environment.

In skeletal based tracking and gesture control mode the turtlebot uses the Kinect camera to track a user. This camera uses the joints of the skeleton to track the user, and maintain focus on the user even when other humans come in and out of view. This skeletal based tracking allows the robot to follow the user around any environment and keep that particular

user in view as long as they do not move to fast. The gesture control mode used the Kinect to determine if a command was given by comparing the position of the user's joints with reference to the shoulders. With this we were able to create 3 different commands that told the robot to go into follow, voice control, or navigation mode.

5. Conclusion

The turtlebot can successfully map and navigate through its environment. The turtlebot is able to autonomously locate itself and navigate through an environment full of obstacles. The user is able to interact with the turtlebot through multiple methods. The location of the robot can be easily controlled by any of the different control methods. This helped enhance the human-machine interaction with the turtlebot.

The robot was able to exhibit and complete the specific tasks that were established for this project. The turtlebot could create a clean and clear occupancy map that allowed the robot to navigate through its environment. The different control methods allowed the robot to drive and switch into other modes. Whether using keyboard teleop, gesture, or voice control; the robot could seamlessly transition between all methods.

Work Cited

www.kobuki.yujinrobot.com/home-en/

www.microsoft.com/en-us/kinectforwindows

www.ros.org

www.ros.org/about-ros/

www.wiki.ros.org

Abbeel, Pieter. "GMapping." (n.d.): n. pag. UC Berkeley EECS. Web. 3 Nov. 2014. <http://www.cs.berkeley.edu/~pabbeel/cs287-fa11/slides/gmapping.pdf>.

Abbeel, Pieter. "Rao-Blackwellized Particle Filtering." (n.d.): n. pag. UC Berkeley EECS. Web. 3 Nov. 2014. <http://www.cs.berkeley.edu/~pabbeel/cs287-fa12/slides/RBPF.pdf>.

"Autonomous Robot." *Wikipedia*. Wikimedia Foundation, 24 Nov. 2014. Web. 25 Nov. 2014. <http://en.wikipedia.org/wiki/Autonomous_robot>.

Butt, Rizwan A., and Syed M. Usman Ali. "Semantic Mapping and Motion Planning with Turtlebot Roomba." *IOP Conf. Series: Materials Science and Engineering 51* 51 (2013): n. pag. *IOP Science*. IOP Publishing, 2013. Web. 2 Nov. 2014. <http://iopscience.iop.org/1757-899X/51/1/012024/pdf/1757-899X_51_1_012024.pdf>.

Claessens, Rik, Yannick Muller, and Benjamin Schnieders. "Graph-based Simultaneous Localization and Mapping on the TurtleBot Platform." (n.d.): n. pag. 23 Jan. 2013. Web. 5 Nov. 2014. <http://airesearch.de/written/Graph-based%20Simultaneous%20Localization%20and%20Mapping%20on%20the%20TurtleBot%20platform.pdf>.

"Controling a Robot Using Voice." *Generation Robots*. N.p., n.d. Web. 2 Dec. 2014. <http://www.generationrobots.com/en/content/59-speech-recognition-system-robot-parallax>.

Eliazar, Austin, and Ronald Parr. "DP-SLAM." *DP-SLAM*. N.p., n.d. Web. 3 Nov. 2014. <http://www.cs.duke.edu/%7Eparr/dpslam/>.

Eliazar, Austin, and Ronald Parr. "DP-SLAM." *OpenSLAM.org*. IJCAI, 2003. Web. 3 Nov. 2014. <http://www.openslam.org/dpslam.html>.

Engebretson, Kelly. "University of St. Thomas Students Lead Robotics Research." *Newsroom*. University of St. Thomas, 15 Aug. 2012. Web. 2 Nov. 2014. <http://www.stthomas.edu/news/school-of-engineering-students-leading-robotics-research-at-st-thomas/>.

"Extended Kalman Filter." *Wikipedia*. Wikimedia Foundation, 22 Sept. 2014. Web. 3 Nov. 2014. <http://en.wikipedia.org/wiki/Extended_Kalman_filter>.

Fardana, A.R., S. Jain, I. Jovancevic, Y. Suri, C. Morand, and N.M. Robertson. "Controlling a Mobile Robot with Natural Commands Based on Voice and Gesture." *Controlling a*

Mobile Robot with Natural Commands Based on Voice and Gesture (n.d.): n. pag. Web. 28 Nov. 2014. <http://home.eps.hw.ac.uk/~cgb7/readinggroup/papers/RobotCommandingByVoiceAndGesture.pdf>.

Grisetti, Giorgio, Cyrill Stachniss, and Wolfram Burgard. "GMapping." *OpenSLAM.org*. In Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2005. Web. 3 Nov. 2014. <http://www.openslam.org/gmapping.html>.

Haehnel, Dirk, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. "GridSLAM." *OpenSLAM.org*. In Proc. of the Conference on Intelligent Robots and Systems (IROS), 2003. Web. 3 Nov. 2014. <http://www.openslam.org/gridslam.html>.

Hendeby, Gustaf, Rickard Karlsson, and Fredrik Gustafsson. "The Rao-Blackwellized Particle Filter: A Filter Bank Implementation." *EURASIP Journal on Advances in Signal Processing*. N.p., 6 Dec. 2010. Web. 4 Nov. 2014. <http%3A%2F%2Fasp.eurasipjournals.com%2Fcontent%2F2010%2F1%2F724087>.

"History of Robots." *Wikipedia*. Wikimedia Foundation, 13 Nov. 2014. Web. 25 Nov. 2014. <http://en.wikipedia.org/wiki/History_of_robots>.

Kalaya, Sithisone, and Hussain A. Alhazmi. "Mobile Robots Exploration and Mapping in 2D." *ASEE 2014 Zone I Conference* (n.d.): n. pag. *ASEE*. ASEE, 2014. Web. 2 Nov. 2014. <http://www.asee.org/documents/zones/zone1/2014/Student/PDFs/249.pdf>.

"Kalman Filter." *Wikipedia*. Wikimedia Foundation, 1 Nov. 2014. Web. 3 Nov. 2014. <http://en.wikipedia.org/wiki/Kalman_filter>.

"Kinect." *Wikipedia*. Wikimedia Foundation, 23 Nov. 2014. Web. 1 Dec. 2014. <http://en.wikipedia.org/wiki/Kinect>.

Kumar, Harish, Vipul Honrao, Sayali Patil, and Pravish Shetty. "Gesture Controlled Robot Using Image Processing." *International Journal of Advanced Research in Artificial Intelligence* 2.5 (2013): 69-77. Web. 28 Nov. 2014. <http://www.ijarai.thesai.org/>.

Li, Maohai, Bingrong Hong, Zesu Cai, and Ronghua Luo. "Novel Rao-Blackwellized Particle Filter for Mobile Robot SLAM Using Monocular Vision." *World Academy of Science, Engineering and Technology* 2.3 (2008): 851-57. *International Science Index*. International Scholarly and Scientific Research & Innovation, 29 Mar. 2008. Web. 4 Nov. 2014. <http://waset.org/publications/15275/novel-rao-blackwellized-particle-filter-for-mobile-robot-slam-using-monocular-vision>.

Lipchin, Boris. "Rao-Blackwellised Particle Filtering." (n.d.): n. pag. Web. 4 Nov. 2014.

Malima, Asanterabi, Erol Özgür, and Müjdat Çetin. "A FAST ALGORITHM FOR VISION-BASED HAND GESTURE RECOGNITION FOR ROBOT CONTROL." (n.d.): n. pag. Web. 1 Dec. 2014. <http://people.sabanciuniv.edu/mcetin/publications/malima_SIU06.pdf>.

Montemerlo, Michael. "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association." *Robotics Institute*. Carnegie

Mellon University, July 2003. Web. 3 Nov. 2014. <http://www.ri.cmu.edu/publication_view.html?pub_id=4434>.

Montemerlo, Michael, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem." (n.d.): n. pag. DARPA's MARS Program. Web. 3 Nov. 2014. <http://robots.stanford.edu/papers/montemerlo.fastslam-tr.pdf>.

"Particle Filter." *Wikipedia*. Wikimedia Foundation, 10 Oct. 2014. Web. 3 Nov. 2014. <http://en.wikipedia.org/wiki/Particle_filter>.

Radhakrishna Rao, Calyampudi. "Rao-Blackwell Theorem." *Scholarpedia*. Eugene M. Izhikevich, Editor-in-Chief of Scholarpedia, the Peer-reviewed Open-access Encyclopedia, 21 July 2008. Web. 4 Nov. 2014. <http://www.scholarpedia.org/article/Rao-Blackwell_theorem>.

"Rao–Blackwell Theorem." *Wikipedia*. Wikimedia Foundation, 22 May 2014. Web. 4 Nov. 2014. <http://en.wikipedia.org/wiki/Rao%E2%80%93Blackwell_theorem>.

"Robot Operating System." *Wikipedia*. Wikimedia Foundation, 24 Nov. 2014. Web. 25 Nov. 2014. <http://en.wikipedia.org/wiki/Robot_Operating_System>.

Sarkka, Simo, Aki Vehtari, and Jouko Lampinen. "Rao-Blackwellized Particle Filter for Multiple Target Tracking." (n.d.): n. pag. Elsevier Science, 22 Sept. 2005. Web. 3 Nov. 2014. <http://www.lce.hut.fi/~ssarkka/pub/nrbmcda-preprint.pdf>.

Tavakolizadeh, Farshid, and Bahador Saket. "2D Mapping Using an Autonomous Robot." *Farshid.ws*. N.p., 22 Jan. 2013. Web. 2 Nov. 2014. <http://farshid.ws/projects.php?id=115>.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. "A Real-Time Algorithm for Mobile Robot Mapping With Applications to Multi-Robot and 3D Mapping." *IEEE International Conference on Robotics and Automation* (2000): n. pag. Apr. 2000. Web. 2 Nov. 2014. <http://robots.stanford.edu/papers/thrun.map3d.pdf>.

Wabale, Amol. "Accelerometer Based Hand Gesture Controlled Robot." *EngineersGarage*. N.p., n.d. Web. 1 Dec. 2014. <http://www.engineersgarage.com/contribution/accelerometer-based-hand-gesture-controlled-robot>.

Appendix A

How to Setup ROS

- Installed the most stable and used version of Ubuntu Linux, version 12.04 (Precise) because it is officially compatible with ROS Groovy.
- Next configure our Ubuntu repositories and set ROS Groovy source list by using the following code:
    - sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
- Setting up the keys:
    - wget http://packages.ros.org/ros.key -O - | sudo apt-key add –
- Installation of ROS groovy full desktop version:
    - sudo apt-get update
    - sudo apt-get install ros-groovy-desktop-full
- Initialize rosdep to give us the ability to easily install dependencies needed for the source packages that we want to use:
    - sudo rosdep init
    - rosdep update
- Next setup the ROS environment to automatically be added to our bash session everytime we launch a package:
    - echo "source /opt/ros/groovy/setup.bash" >> ~/.bashrc
    - source ~/.bashrc

- Now install rosinstall tools such as python which is not included in the normal ROS installation:
    - sudo apt-get install python-setuptools
    - sudo apt-get install python-rosinstall python-rosdep
    - sudo rosdep init
    - rosdep update
- Next step directory needs to be created where we are going to create our packages, and will need to add this directory to the normal ROS path, so ROS will be able to recognize it.
    - mkdir ~/ros_workspace
    - gedit ~/.bashrc
    - Add the following path to the file:
        - Export ROS_PACKAGE_PATH=~/ros_workspace:$ROS_PACKAGE_PATH
    - source ~/.bashrc
- To familiarize with ROS and Linux terminal environment we went through many tutorials that are available on the official wikipage of ROS.
    - Wiki.ros.org
- All of the main codes for ROS packages are both available in C++ and python.
- RViz (ROS visualization tool) is many times through our project to control the robot for navigation and etc.
- Arbotix Simulator is used to simulate the code before running it on the real robot.

- The movement of the turtlebot is controlled by sending Twist messeges which controles the nodes cmd_vel and geometry_msg, here is an example of how it’s done:
    - rostopic pub -r 10 /cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0.5}}'
- Setting up the navigation stack:
    - Gmapping:
        - roslaunch turtlebot_navigation gmapping.launch
    - Base controller:
        - roslaunch turtlebot_navigation move_base.launch
    - Activating kinect:
        - roslaunch turtlebot_bringup 3dsensor.launch
    - Teleoperation:
        - roslaunch turtlebot_teleop keyboard_teleop.launch
    - Created a single launch file which included all the nodes necessary to create a map:
        - roslaunch rbx1_speech mapping.launch
    - To view the map while is being created and save the map:
        - roslaunch turtlebot_rviz_launchers view_navigation
            - Used for both creating the map and navigation
        - rosrun map_server map_server –f my_map
            - This command will save the map in folder called tmp which is a temporary folder in ubuntu. My_map can be called any name.
    - To navigate we need to load the map and run amcl with RViz:

- roslaunch turtlebot_navigation amcl.launch
    - The saved map can be loaded by modifying the amcl launch file
    - When launched before navigating to the goal, the initial position has to be set in rviz using the “set initial position button”.
    - Next we choose a goal in rviz and the robot will navigate through an existing map while avoiding obstacles. Both stationary and mobile obstacles.

Turtlebot Bringup Launch Code

```
<launch>
  <!-- Make sure we can move the robot by simply publishing to the /cmd_vel topic -->
  <remap from="mobile_base/commands/velocity" to="cmd_vel" />

  <!-- Edit and uncomment these parameters after calibration -->
  <param name="turtlebot_node/gyro_scale_correction" value="1.0"/>
  <param name="turtlebot_node/odom_angular_scale_correction" value="1.0"/>
  <param name="turtlebot_node/odom_linear_scale_correction" value="1.0"/>

  <arg name="base"       value="$(optenv TURTLEBOT_BASE kobuki)"/>  <!-- create, 
rhoomba -->
  <arg name="battery"    default="$(optenv TURTLEBOT_BATTERY 
/proc/acpi/battery/BAT0)"/>  <!-- /proc/acpi/battery/BAT0 -->
  <arg name="stacks"     default="$(optenv TURTLEBOT_STACKS circles)"/>  <!-- 
circles, hexagons -->
  <arg name="3d_sensor"  default="$(optenv TURTLEBOT_3D_SENSOR kinect)"/>  
<!-- kinect, asus_xtion_pro -->
  <arg name="simulation" default="$(optenv TURTLEBOT_SIMULATION false)"/>

  <param name="/use_sim_time" value="$(arg simulation)"/>

  <arg name="urdf_file" default="$(find xacro)/xacro.py '$(find 
rbx1_description)/urdf/turtlebot.urdf.xacro'" />
  <param name="robot_description" command="$(arg urdf_file)" />

  <!-- important generally, but specifically utilised by the current app manager -->
```

```
          <param name="robot/name" value="$(optenv ROBOT turtlebot)"/>
          <param name="robot/type" value="turtlebot"/>

          <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher">
            <param name="publish_frequency" type="double" value="10.0" />
          </node>
          <node pkg="diagnostic_aggregator" type="aggregator_node"
name="diagnostic_aggregator" >
            <rosparam command="load" file="$(find turtlebot_bringup)/param/$(arg
base)/diagnostics.yaml" />
          </node>

          <include file="$(find turtlebot_bringup)/launch/includes/_zeroconf.launch"/>
          <include file="$(find turtlebot_bringup)/launch/includes/_mobile_base.launch">
            <arg name="base" default="$(arg base)" />
          </include>
          <include file="$(find turtlebot_bringup)/launch/includes/_netbook.launch">
            <arg name="battery" default="$(arg battery)" />
          </include>
          <include file="$(find turtlebot_bringup)/launch/includes/_app_manager.launch"/>
        </launch>
```

Navigation (AMCL) Launch Code

```
        <launch>
          <include file="$(find turtlebot_bringup)/launch/3dsensor.launch">
            <arg name="rgb_processing" value="false" />
            <arg name="depth_registration" value="false" />
            <arg name="depth_processing" value="false" />
          </include>

          <!-- Map server -->
          <arg name="map_file" default="$(find turtlebot_navigation)/maps/willow-2010-02-
18-0.10.yaml"/>
          <node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />

          <arg name="initial_pose_x" default="0.0"/> <!-- Use 17.0 for willow's map in
simulation -->
          <arg name="initial_pose_y" default="0.0"/> <!-- Use 17.0 for willow's map in
simulation -->
          <arg name="initial_pose_a" default="0.0"/>
```

```
<include file="$(find turtlebot_navigation)/launch/includes/_amcl.launch">
  <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
  <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
  <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
</include>

<include file="$(find turtlebot_navigation)/launch/includes/_move_base.launch"/>

</launch>

<launch>

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
turtlebot_rviz_launchers)/rviz/navigation.rviz"/>

</launch>
```

Gesture Control

Launch Code

```
<launch>
  <arg name="fixed_frame" value="camera_depth_frame" />

  <arg name="debug" value="False" />
  <arg name="launch_prefix" value="xterm -e gdb --args" />

  <param name="robot_description" command="$(find xacro)/xacro.py '$(find
pi_tracker)/urdf/pi_robot.urdf.xacro'" />
  <param name="/use_sim_time" value="False" />

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher">
    <param name="publish_frequency" value="20.0"/>
  </node>

  <include file="$(find openni_launch)/launch/kinect_frames.launch" />

  <group if="$(arg debug)">
    <node launch-prefix="$(arg launch_prefix)" name="skeleton_tracker"
pkg="pi_tracker" type="skeleton_tracker" output="screen"      >
      <param name="fixed_frame" value="$(arg fixed_frame)" />
```

```
            <param name="load_filepath" value="$(find
pi_tracker)/params/SamplesConfigNewOpenNI.xml" />
          </node>
        </group>
        <group unless="$(arg debug)">
          <node name="skeleton_tracker" pkg="pi_tracker" type="skeleton_tracker">
            <param name="fixed_frame" value="$(arg fixed_frame)" />
            <param name="load_filepath" value="$(find
pi_tracker)/params/SamplesConfigNewOpenNI.xml" />
          </node>
        </group>

        <node name="tracker_command" pkg="pi_tracker" type="tracker_command.py"
output="screen">
          <rosparam command="load" file="$(find pi_tracker)/params/tracker_params.yaml"
/>
        </node>
      </launch>
```

Python Code

```
#!/usr/bin/env python



import roslib; roslib.load_manifest('pi_tracker')
import rospy
import pi_tracker_lib as PTL
from pi_tracker.msg import Skeleton
from pi_tracker.srv import *
import PyKDL as KDL
from math import copysign
import time
import os
import signal

class TrackerCommand():
    def __init__(self):
        rospy.init_node('tracker_command')
        rospy.on_shutdown(self.shutdown)

        # How frequently do we publish
        self.rate = rospy.get_param("~command_rate", 1)
        rate = rospy.Rate(self.rate)
```

```
        # Subscribe to the skeleton topic.
        rospy.Subscriber('skeleton', Skeleton, self.skeleton_handler)

        # Store the current skeleton configuration in a local dictionary.
        self.skeleton = dict()
        self.skeleton['confidence'] = dict()
        self.skeleton['position'] = dict()
        self.skeleton['orientation'] = dict()


        self.gestures = {
                'left_hand_up': self.left_hand_up,
                'arms_crossed': self.arms_crossed,
                'right_hand_up': self.right_hand_up
                }


        # Initialize the robot in the stopped state.
        self.tracker_command = "STOP"

        rospy.loginfo("Initializing Tracker Command Node...")

        while not rospy.is_shutdown():
            """ Get the scale of a body dimension, in this case the shoulder width, so that we
can scale
                interjoint distances when computing gestures. """
            self.shoulder_width = PTL.get_body_dimension(self.skeleton, 'left_shoulder',
'right_shoulder', 0.4)

            # Compute the tracker command from the user's gesture
            self.tracker_command = self.get_command(self.tracker_command)

            rate.sleep()

    def left_hand_up(self):
        if self.confident(['right_hand', 'left_hand']):
            if (self.skeleton['position']['left_hand'].y() -
self.skeleton['position']['right_hand'].y()) / self.shoulder_width > 0.7:
                return True
        return False

    def right_hand_up(self):
        if self.confident(['right_hand', 'left_hand']):
            if (self.skeleton['position']['right_hand'].y() -
```

```
self.skeleton['position']['left_hand'].y()) / self.shoulder_width > 1:
                    return True
            return False

        def arms_crossed(self):
            if self.confident(['left_elbow', 'right_elbow', 'left_hand', 'right_hand']):
                if copysign(1.0, self.skeleton['position']['left_elbow'].x() -
self.skeleton['position']['right_elbow'].x()) == \
                    -copysign(1.0, self.skeleton['position']['left_hand'].x() -
self.skeleton['position']['right_hand'].x()):
                    return True
            return False

        def confident(self, joints):
            try:
                for joint in joints:
                    if self.skeleton['confidence'][joint] < 0.5:
                        return False
                return True
            except:
                return False

        def get_command(self, current_command):
            try:
                # Raise right hand to engage base controller
                if self.gestures['right_hand_up']():
                        os.system("roslaunch turtlebot_follower follower.launch")
                        os.kill(os.getppid(),signal.SIGTERM)

                # Raise left hand to engage joint teleoperation
                elif self.gestures['left_hand_up']():
                        os.system("roslaunch rbx1_speech voice.launch")
                        os.kill(os.getppid(),signal.SIGTERM)

                # Cross the arms to stop the robot base
                elif self.gestures['arms_crossed']():
                        os.system("roslaunch pi_tracker amcl.launch
map_file:=/tmp/my_map.yaml")
                        os.kill(os.getppid(),signal.SIGTERM)
                except:
                        pass
        def skeleton_handler(self, msg):
            for joint in msg.name:
                self.skeleton['confidence'][joint] = msg.confidence[msg.name.index(joint)]
                self.skeleton['position'][joint] =
```

```
KDL.Vector(msg.position[msg.name.index(joint)].x, msg.position[msg.name.index(joint)].y,
msg.position[msg.name.index(joint)].z)
                self.skeleton['orientation'][joint] = msg.orientation[msg.name.index(joint)]

    def shutdown(self):
        rospy.loginfo("Shutting down Tracker Command Node.")

if __name__ == '__main__':
    try:
        TrackerCommand()
    except rospy.ROSInterruptException:
        pass
```

Voice Control

Launch Code

```
<launch>

  <node name="recognizer" pkg="pocketsphinx" type="recognizer.py"
output="screen">
    <param name="lm" value="$(find rbx1_speech)/config/nav_commands.lm"/>
    <param name="dict" value="$(find rbx1_speech)/config/nav_commands.dic"/>
  </node>

  <node name="voice_nav" pkg="rbx1_speech" type="voice_nav.py" output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
        <param name="max_speed" value="0.3"/>
        <param name="start_speed" value="0.1"/>
        <param name="linear_increment" value="0.05"/>
        <param name="angular_increment" value="0.4"/>
  </node>

</launch>
```

Python Code

```
#!/usr/bin/env python
```

```
import roslib; roslib.load_manifest('rbx1_speech')
import rospy
import os
from geometry_msgs.msg import Twist
from std_msgs.msg import String
from math import copysign
import signal

class VoiceNav:
    def __init__(self):
        rospy.init_node('voice_nav')

        rospy.on_shutdown(self.cleanup)

        # Set a number of parameters affecting the robot's speed
        self.max_speed = rospy.get_param("~max_speed", 0.4)
        self.max_angular_speed = rospy.get_param("~max_angular_speed", 1.5)
        self.speed = rospy.get_param("~start_speed", 0.1)
        self.angular_speed = rospy.get_param("~start_angular_speed", 0.5)
        self.linear_increment = rospy.get_param("~linear_increment", 0.05)
        self.angular_increment = rospy.get_param("~angular_increment", 0.4)

        # We don't have to run the script very fast
        self.rate = rospy.get_param("~rate", 5)
        r = rospy.Rate(self.rate)

        # A flag to determine whether or not voice control is paused
        self.paused = False

        # Initialize the Twist message we will publish.
        self.cmd_vel = Twist()

        # Publish the Twist message to the cmd_vel topic
        self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist)

        # Subscribe to the /recognizer/output topic to receive voice commands.
        rospy.Subscriber('/recognizer/output', String, self.speech_callback)

        # A mapping from keywords or phrases to commands
            self.keywords_to_command = {'stop': ['stop', 'halt', 'abort', 'kill', 'panic', 'off',
'freeze', 'shut down', 'turn off', 'help', 'help me'],
                                    'slower': ['slow down', 'slower'],
                                    'faster': ['speed up', 'faster'],
                                    'forward': ['forward', 'ahead', 'straight'],
```

```
                                    'backward': ['back', 'backward', 'back up'],
                                    'rotate left': ['rotate left'],
                                    'rotate right': ['rotate right'],
                                            'keyboard':['keyboard'],
                                    'turn left': ['turn left'],
                                    'turn right': ['turn right'],
                                            'follow':['follow me', 'follower mode', 'follow', 'come
with me', 'lets go'],
                                    'quarter': ['quarter speed'],
                                    'half': ['half speed'],
                                    'full': ['full speed'],
                                    'pause': ['pause speech'],
                                    'continue': ['continue speech']}

         rospy.loginfo("Ready to receive voice commands")
         global flag
           flag = False
         # We have to keep publishing the cmd_vel message if we want the robot to keep
moving.
         while not rospy.is_shutdown() and not flag:
           self.cmd_vel_pub.publish(self.cmd_vel)
           r.sleep()

       def get_command(self, data):
         # Attempt to match the recognized word or phrase to the
         # keywords_to_command dictionary and return the appropriate
         # command
         for (command, keywords) in self.keywords_to_command.iteritems():
           for word in keywords:
             if data.find(word) > -1:
               return command

       def speech_callback(self, msg):
         # Get the motion command from the recognized phrase
         command = self.get_command(msg.data)

         # Log the command to the screen
         rospy.loginfo("Command: " + str(command))

         # If the user has asked to pause/continue voice control,
         # set the flag accordingly
         if command == 'pause':
           self.paused = True
         elif command == 'continue':
           self.paused = False
```

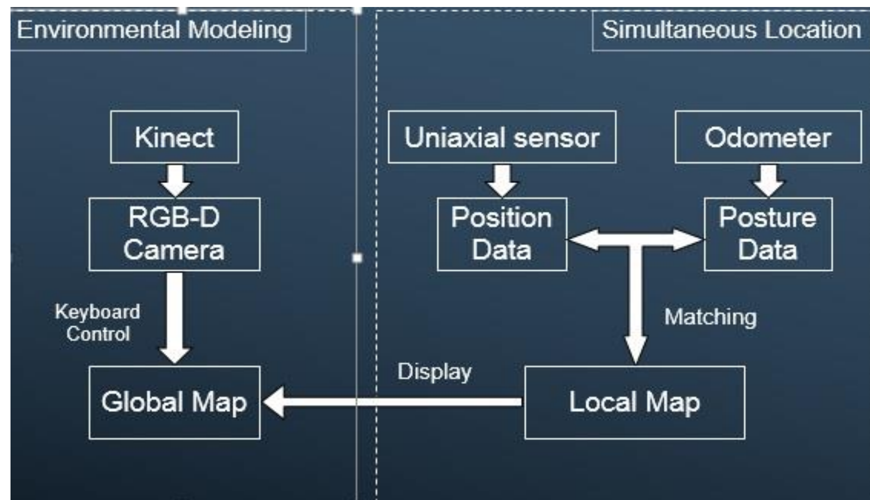
```
# If voice control is paused, simply return without
# performing any action
if self.paused:
    return

# The list of if-then statements should be fairly
# self-explanatory
if command == 'forward':
    self.cmd_vel.linear.x = self.speed
    self.cmd_vel.angular.z = 0

elif command == 'rotate left':
    self.cmd_vel.linear.x = 0
    self.cmd_vel.angular.z = self.angular_speed

elif command == 'rotate right':
    self.cmd_vel.linear.x = 0
    self.cmd_vel.angular.z = -self.angular_speed

elif command == 'turn left':
    if self.cmd_vel.linear.x != 0:
        self.cmd_vel.angular.z += self.angular_increment
    else:
        self.cmd_vel.angular.z = self.angular_speed

elif command == 'turn right':
    if self.cmd_vel.linear.x != 0:
        self.cmd_vel.angular.z -= self.angular_increment
    else:
        self.cmd_vel.angular.z = -self.angular_speed

    elif command == 'follow':
       os.system("roslaunch turtlebot_follower follower.launch")
       os.kill(int(os.getppid), signal.CTL_C_EVENT)

    elif command == 'keyboard':
       os.system("roslaunch rbx1_nav keyboard_teleop.launch")
       os.kill(int(os.getppid), signal.CTL_C_EVENT)

elif command == 'backward':
    self.cmd_vel.linear.x = -self.speed
    self.cmd_vel.angular.z = 0

elif command == 'stop':
```

```
            # Stop the robot!  Publish a Twist message consisting of all zeros.
            self.cmd_vel = Twist()

        elif command == 'faster':
            self.speed += self.linear_increment
            self.angular_speed += self.angular_increment
            if self.cmd_vel.linear.x != 0:
                self.cmd_vel.linear.x += copysign(self.linear_increment,
self.cmd_vel.linear.x)
            if self.cmd_vel.angular.z != 0:
                self.cmd_vel.angular.z += copysign(self.angular_increment,
self.cmd_vel.angular.z)

        elif command == 'slower':
            self.speed -= self.linear_increment
            self.angular_speed -= self.angular_increment
            if self.cmd_vel.linear.x != 0:
                self.cmd_vel.linear.x -= copysign(self.linear_increment,
self.cmd_vel.linear.x)
            if self.cmd_vel.angular.z != 0:
                self.cmd_vel.angular.z -= copysign(self.angular_increment,
self.cmd_vel.angular.z)

        elif command in ['quarter', 'half', 'full']:
            if command == 'quarter':
                self.speed = copysign(self.max_speed / 4, self.speed)

            elif command == 'half':
                self.speed = copysign(self.max_speed / 2, self.speed)

            elif command == 'full':
                self.speed = copysign(self.max_speed, self.speed)

            if self.cmd_vel.linear.x != 0:
                self.cmd_vel.linear.x = copysign(self.speed, self.cmd_vel.linear.x)

            if self.cmd_vel.angular.z != 0:
                self.cmd_vel.angular.z = copysign(self.angular_speed,
self.cmd_vel.angular.z)

        else:
            return

        self.cmd_vel.linear.x = min(self.max_speed, max(-self.max_speed,
self.cmd_vel.linear.x))
```

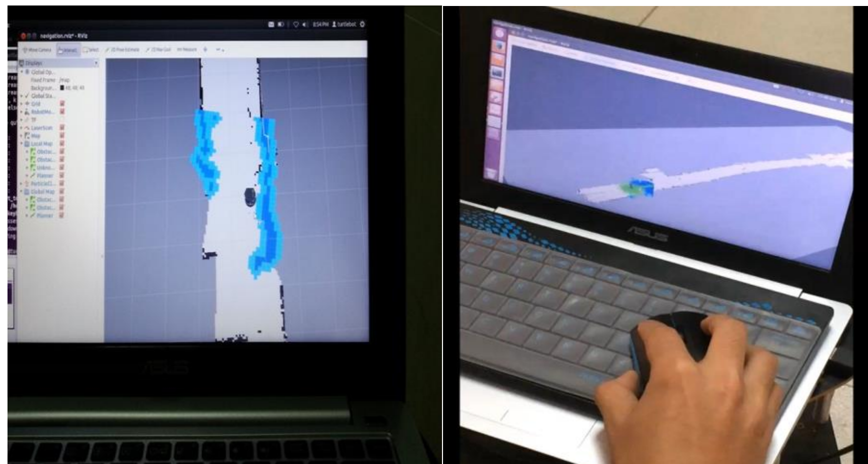
```
            self.cmd_vel.angular.z = min(self.max_angular_speed, max(-
self.max_angular_speed, self.cmd_vel.angular.z))

        def cleanup(self):
            # When shutting down be sure to stop the robot!
            twist = Twist()
            self.cmd_vel_pub.publish(twist)
            rospy.sleep(1)

    if __name__=="__main__":
        try:
            VoiceNav()
            rospy.spin()
        except rospy.ROSInterruptException:
            rospy.loginfo("Voice navigation terminated.")
```

Appendix B

Mapping and Navigation



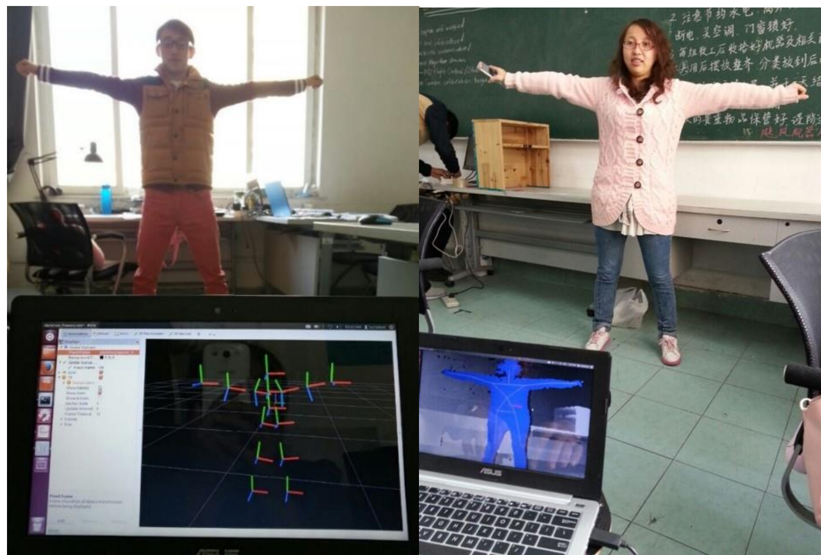
The chart above shows how AMCL works, using environmental modeling and simultaneous localization.



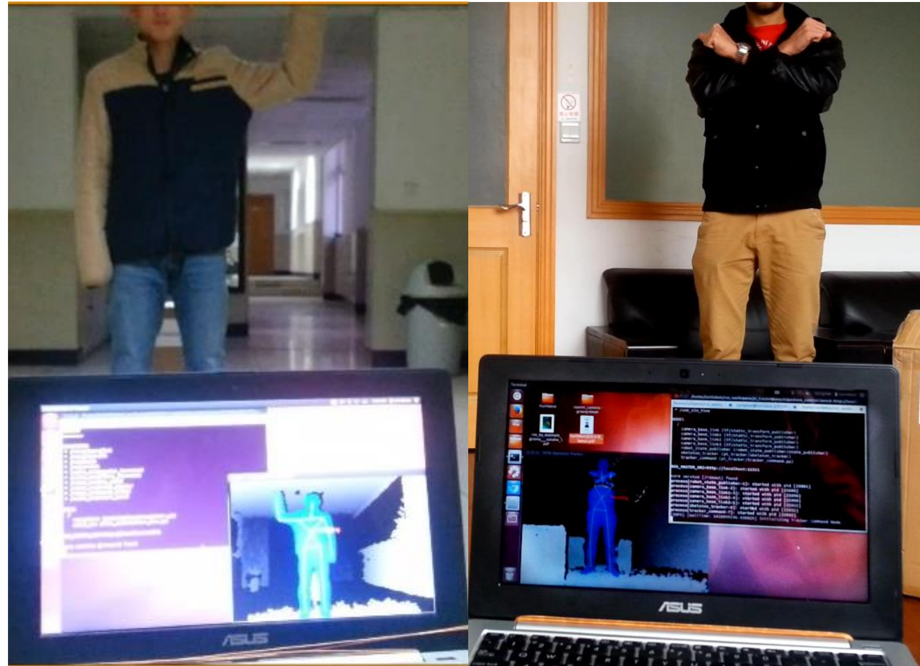
The images above show the occupancy map and how the navigation goal is set for the turtlebot.

These images above show how the turtlebot can avoid obstacles in its environment.

Skeletal Base Tracking and Gesture Control



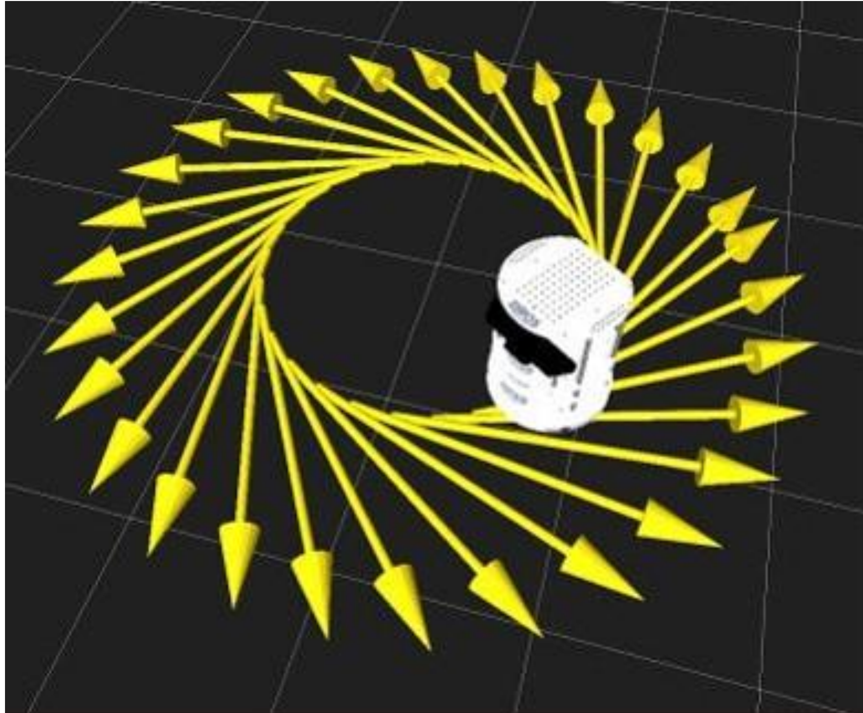
These images show how the Kinect camera is used with the turtlebot to complete skeletal tracking.

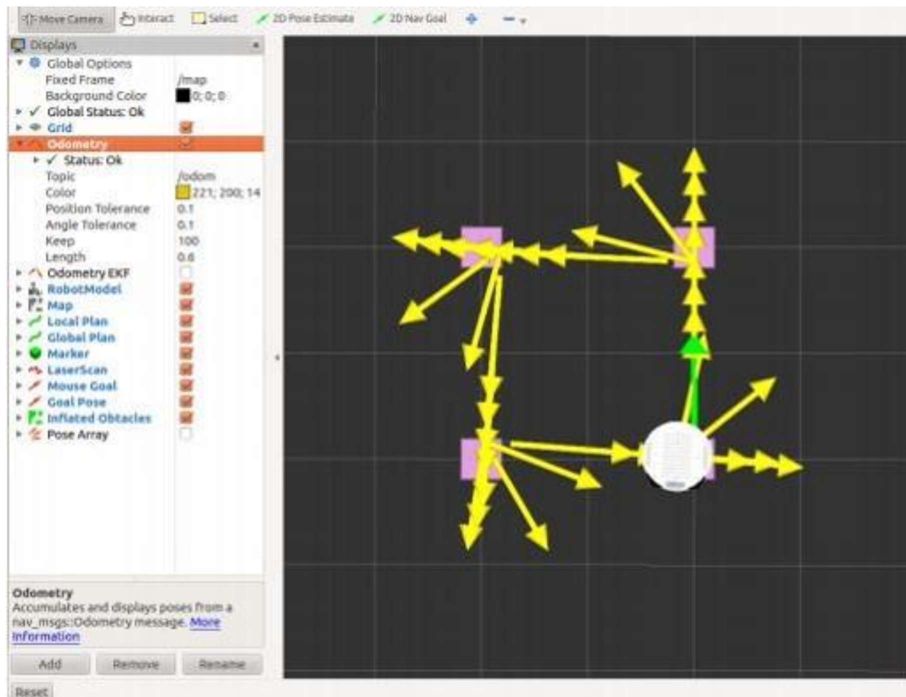


This is what the computer screen looks like when gesture control mode is running. As soon as it recognizes the gesture it runs the launch code for the specific mode.

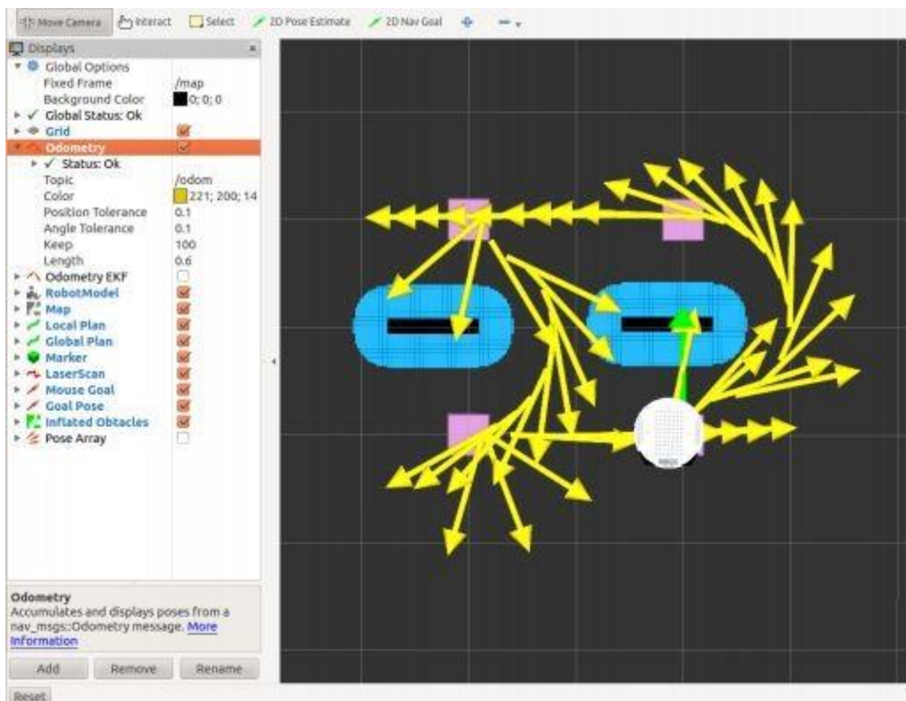
Testing Virtual Turtlebot

These images show what it looks like to test a virtual turtlebot in Arbotix. Arbotix allows programmers to test and run code on the simulated turtlebot to see how it will work in real life.





Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Displays
Global Options
Fixed Frame
/map
Background Color
0; 0; 0
Global Status: Ok
Grid
Odometry
Status: Ok
Topic
/odom
Color
221; 200; 14
Position Tolerance
0.1
Angle Tolerance
0.1
Keep
100
Length
0.6
Odometry EKF
RobotModel
Map
Local Plan
Global Plan
Marker
LaserScan
Mouse Goal
Goal Pose
Inflated Obtacles
Pose Array
Odometry
Accumulates and displays poses from a nav_msgs::Odometry message. More Information
Add
Remove
Rename
Reset



Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Displays
Global Options
Fixed Frame
/map
Background Color
0; 0; 0
Global Status: Ok
Grid
Odometry
Status: Ok
Topic
/odom
Color
221; 200; 14
Position Tolerance
0.1
Angle Tolerance
0.1
Keep
100
Length
0.6
Odometry EKF
RobotModel
Map
Local Plan
Global Plan
Marker
LaserScan
Mouse Goal
Goal Pose
Inflated Obtacles
Pose Array
Odometry
Accumulates and displays poses from a nav_msgs::Odometry message. More Information
Add
Remove
Rename
Reset

Appendix C

Turtlebot Specifications

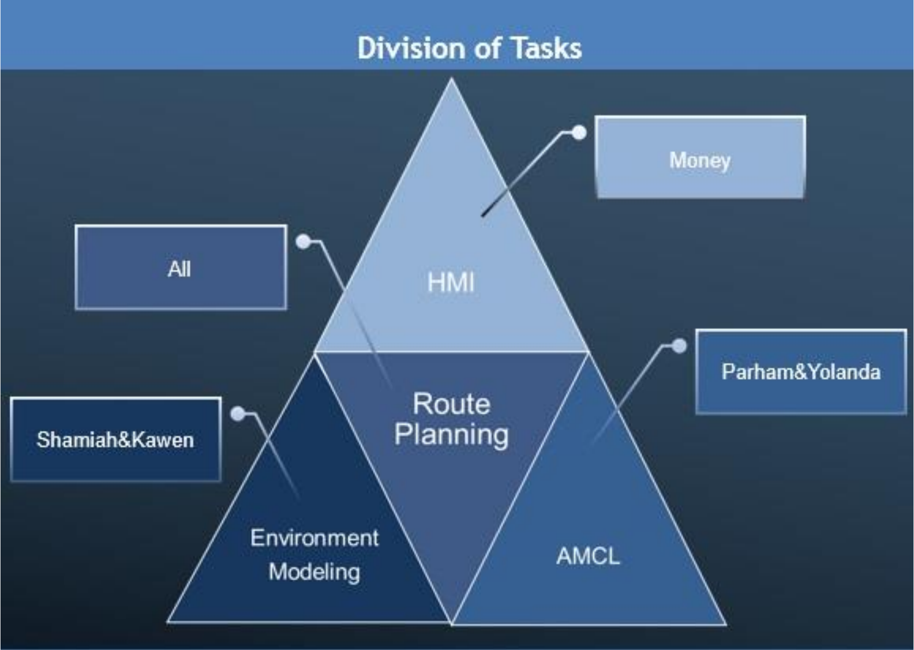
Project Schedule

Gantt Chart

TASK		DETAILS
Functions	Environ-mental Modeling	Learn the Kinect and sensors
		Compare the methods of establishing map
		Simulation debugging
	AMCL (self-localize)	Understanding the theories of 2 location method
		Compare the 2 location methods
		Simulation debugging
	Route Planning	Load the environment map
		Real-time location map
		Avoid collision (Combine inner-sensors with Kinect)
		Simulation debugging
	HMI	Mode 1: Gesture Control
		Mode 2: Remote Control by Keyboard
		Mode 3: Voice Control Bluetooth microphones
Presentation	PPT Preparation	
	Team meeting	
	Weekly Meeting	
	Function combination	
	Final Defence Preparation	
	Final Defence	

October / November / DECEMBER

WEEK 1 | WEEK 2 | WEEK 3 | WEEK 4 | WEEK 5 | WEEK 6 | WEEK 7


Division of Tasks
Money
All
HMI
Parham&Yolanda
Shamiah&Kawen
Route
Planning
Environment
Modeling
AMCL