A Student Project using Robotic Operating System (ROS) for Undergraduate Research

Dr. Stephen Andrew Wilkerson P.E., York College of Pennsylvania

Stephen Wilkerson (swilkerson@ycp.edu) received his PhD from Johns Hopkins University in 1990 in Mechanical Engineering. His Thesis and initial work was on underwater explosion bubble dynamics and ship and submarine whipping. After graduation he took a position with the US Army where he has been ever since. For the first decade with the Army he worked on notable programs to include the M829A1 and A2 that were first of a kind composite saboted munition. His travels have taken him to Los Alamos where he worked on modeling the transient dynamic attributes of Kinetic Energy munitions during initial launch. Afterwards he was selected for the exchange scientist program and spent a summer working for DASA Aerospace in Wedel, Germany 1993. His initial research also made a major contribution to the M1A1 barrel reshape initiative that began in 1995. Shortly afterwards he was selected for a 1 year appointment to the United States Military Academy West Point where he taught Mathematics. Following these accomplishments he worked on the SADARM fire and forget projectile that was finally used in the second gulf war. Since that time, circa 2002, his studies have focused on unmanned systems both air and ground. His team deployed a bomb finding robot named the LynchBot to Iraq late in 2004 and then again in 2006 deployed about a dozen more improved LynchBots to Iraq. His team also assisted in the deployment of 84 TACMAV systems in 2005. Around that time he volunteered as a science advisor and worked at the Rapid Equipping Force during the summer of 2005 where he was exposed to a number of unmanned systems technologies. His initial group composed of about 6 S&T grew to nearly 30 between 2003 and 2010 as he transitioned from a Branch head to an acting Division Chief. In 2010-2012 he again was selected to teach Mathematics at the United States Military Academy West Point. Upon returning to ARL's Vehicle Technology Directorate from West Point he has continued his research on unmanned systems under ARL's Campaign for Maneuver as the Associate Director of Special Programs. Throughout his career he has continued to teach at a variety of colleges and universities. For the last 4 years he has been a part time instructor and collaborator with researchers at the University of Maryland Baltimore County (http://me.umbc.edu/directory/). He is currently an Assistant Professor at York College PA.

Dr. Jason Forsyth, York College of Pennsylvania

Jason Forsyth is an Assistant Professor of Electrical and Computer Engineering at York College of Pennsylvania. He received his PhD from Virginia Tech in May 2015. His major research interests are in wearable and pervasive computing. His work focuses on developing novel prototype tools and techniques for interdisciplinary teams.

Cara Sperbeck, York College of Pennsylvania

Cara Sperbeck (csperbeck6@gmail.com) is currently an undergraduate senior pursing her BS in Computer Engineering from York College of Pennsylvania. She has worked at Northrop Grumman as a Digital Technology FPGA firmware designer (co-op) and Intelligent Automation Inc. as a Robotics and Software Control Engineering Intern. Cara has received the Who's Who Among Colleges and Universities Award in January 2017 and was inducted into Alpha Chi National Honor Society for being in the top 5% in York College Junior Class in October 2016. She has also received the Engineering Society of York (PA) Award for being the top Sophomore Electrical/Computer Engineer in April 2016. Cara's current research interests include Digital Design, FPGA design, Robotics, and Computer programming.

Mr. Matthew Jones, York College of Pennsylvania

Major: Computer Science Minor: Mathematics Areas of interest: machine learning, web development

Mr. Patrick David Lynn, York College of Pennsylvannia

Senior Computer Engineer

A Student Project using Robotic Operating System (ROS) for Undergraduate Research

In this student led undergraduate research paper we present a robotics project using the Robot Operating System (ROS). The purpose of this student paper is to document their learning path and steps taken for a project using three related, yet independent student projects so that others might benefit from the details. The students worked together initially to learn enough about ROS and it's development environment so that they might employ it. Thus far the use of ROS has primarily been focused on graduate studies where improvements to the underlying algorithms and techniques have been made. In this undergraduate approach no such attempts are made in improving the foundation algorithms already developed by top researchers and schools. Rather, the students employed published techniques to provide the foundation of their work. Specifically, the project used the Turtle-bot architecture and modules within ROS to create the components of a cooperative robotic mission. The crux of the mission is for one of the robots to autonomously explore and map an area of the engineering building while leaving bread-crumbs behind for another robot to follow. A third robot comes behind the second and uses the information from the first two to locate tags distributed throughout the building. Each student made a portion of this project, that could stand alone, so that others could use these individual modules and details for their own projects without redoing what had been done here. The three parts were broken into mapping, tag recognition with robot leader follower operations, and object location and RF tag reading. It enabled students to use the existing sensors on the turtle-bot, while incorporating new devices to complete their particular missions. In this paper the students detail the learning path that was required to bring their individual technologies for their sub project to fruition. Using their initial code and techniques will enable others to duplicate and expand at a quicker pace. We have already seen this in the second semester as new students are tackling tasks of increased difficulty building on what was done here. Furthermore, the students detail the methods used and the code that they wrote to accomplish the tasks described. Finally, the students identified the technologies required to learn and the research they did as a criteria of merit. Additionally, a series of *Youtube* instructional videos and a file repository are available for others to use. All of the code developed is given on google drives and GitHub for others to use and is referenced at the end of this paper.

Introduction

One of the difficulties with robotics is that it usually requires intricate knowledge of hardware level details in order to even begin developing robotic systems (Quigley 2009). Whether it's communicating with motor controllers, utilizing servos, interfacing with sensors or implementing hardware acceleration for improved computing, learning robotics often requires graduate level knowledge of these topics. This means that some robotics topics and capability

can be out of reach for most undergraduates, due to its steep learning curve. Fortunately, ROS has numerous libraries and "how to examples" making it easy for students to learn and exchange examples (Cousins et. al. 2010). When robotics is applied to a problem such as navigating a hallway in order to deliver a soda to someone, or following a known target, these problems are well suited for undergraduates. How the robot's environment influences its decision-making is a problem that can be researched and tackled by undergraduates. So, it would be beneficial to have a way to work with robots that was within the grasps of undergraduate studies. The algorithms and theory surrounding tasks involving in-depth knowledge of the hardware and software systems are all taken care of by ROS. This is the void that ROS fills. ROS simplifies robotics by abstracting away the hardware/software level issues and allows robotic engineers to focus directly on solving problems. ROS does this by utilizing Ubuntu as an operating system and implementing a set of common libraries that allows programmers to easily get robots to communicate with other ROS enabled devices. This simplifies the process for adding functionality to a robot, including its sensory equipment or servo motors. ROS requires much less knowledge to learn than diving right into hardware level details and therefore, ROS makes the learning curve much more manageable for undergraduates (Martinez et. al. 2013).

When you couple ROS with project learning (Qidwai 2011) you create an environment for undergraduate computer engineer to really expand their understanding of computation systems and utilize much of the engineering theory that has been learned in class. With project based learning the student gains a more complete understanding of the problem by necessity, because they are the only one working on the project. This means that the student fully conceptualizes the problem set in order to solve it, often leading the student to becoming some what of an expert of the topics involved. Furthermore, project based learning allows students to learn at their own pace. Since students do not all learn at the same rate, project based learning allows students to be unhindered by pacing issues that are commonplace in traditional classroom settings. But most importantly, project based learning mirrors industry, which forces the student to apply theory in a pragmatic way. In industry you are not given monthly tests in order to prove your understanding; in industry you tackle projects, in teams or individually, focusing on the task at hand until the problem is solved. This is the part of industry that project based learning best represents and is its most valuable asset.

In order to get everyone on the same page, a course of study was undertaken to familiarize students with the basic tools needed in this independent study project. In some cases the students had no prior experience with linux, ubuntu, and had never heard of ROS or the Turtlebot/Kobuki. To do this the first 6 weeks were spent doing the following:

- **Learn Linux, Ubuntu, ROS Setup.**

To this end, students rebuilt the ubuntu laptops that ran the Turtlebots. They needed to build from scratch the OS using ubuntu 14.04 and then load ROS, Turtlebot and other associated libraries on the laptops. We had two different Asus laptops for this purpose that were both 32 bit systems and 64 bit systems. Initially the students used the existing school wireless network to run their experiments. However, it became obvious that this was insufficient and an independent robot only network was built by the students for robot experiments.

- **Learn the Robotic Operating System (ROS) and its development environment by developing applications in C++ or Python.**

For this portion we employed the book "A Gentle Introduction to ROS," (O'Kane 2014). Students worked through the chapters in the first couple of weeks; this gave them the ability to write both C^{++} and Python programs that accessed the information within the publisher subscriber ROS environment. This course of study also enabled the students to better understand the tutorial examples that were available online. Without this asset it would have been difficult to fully understand the Turtlebot examples or how to access ROS topics, messages and other vital information within the ROS framework.

- **Learn how others have built ROS enabled programs for robotic systems by recreating and extending several existing github ROS examples.**

For this portion we relied on internet examples. In particular the Turtlebot has a number of tutorials on learning Turtlebot†. Additionally they have a series of challenges to further understanding of the Turtlebot. However, these tutorials are step by step examples that do not really teach you the workings of ROS, the Turtlebot, or how the publisher subscriber works. It is our assertion here that becoming proficient with writing publisher subscriber programs, launch files and the like can only be accomplished through examination of the details of the programs needed to support program development. ROS is more than just a library and includes a central server, command line tools, graphical tools and a build system. Our hypothesis is by combining several resources ROS can be used for undergraduate research efficiently. What follows here are the three projects that make up our overall goal stated in the beginning of this paper.

Individual projects:

1. Map making.

The Kobuki Turtlebot, a robot that relies on a relative coordinate system and dead reckoning to drive autonomously using a blueprint map so the robot knows its location and destination. The Turtlebot will simultaneously and computationally construct a map of the environment, as well as its known relative position, in accordance with its location, by using Simultaneous Localization And Mapping (SLAM) (Ghani et. al. 2014). Turtlebot uses the SLAM algorithm called GMapping. Using GMapping, the robot analyzes an existing map to find the best route to get to a destination(Schmidt et. al. 2012). If multiple routes exist there are existing algorithms to help the robot make a decision.

In this paper, we document our findings of the many deficiencies in this method of "Robot-made" mapping, and then we propose a method that does not have these same deficiencies. We present a method where the floor plan could be converted to the map file format that the robot understands. If a raster image of the floor plan exists, then this process will work. The time consuming issues and lack of accuracy problems found with robot-made mapping is eliminated. We will also compare how the robot drives and performs on the maps generated from both map generation types.

† Turtlebot learning: http://learn.Turtlebot.com/

Robot made map

Creating a robot-made map is a 4 step process. First, the user loads the GMapping demo software. Second, RViz, Turtlebot's front-end live map visualization tool, needs to be loaded so the user can visualize the map as they create it (See http://learn.Turtlebot.com/2015/02/01/11/ for more details). Third, the user drives the Turtlebot around the desired environment until the user is satisfied with the map coverage, as seen in RViz‡†. Lastly, the user saves the map to the desired location in the computer's file system. This last step creates and saves the map with the output in the form of a "pgm" image of the map, as well as a "yaml"(Yet Another Markup Language) configuration file (Sumaray et. al. 2012). The "pgm" is an image that shows the blueprint of the environment that the Turtlebot just traversed, and the "yaml" file is the configuration file for the map. The "yaml" file is also used to load a map onto the Turtlebot, linking the "pgm" blueprint, as well as the other meta information. The "yaml" contains information such as the base path of the "pgm" blueprint associated with the "yaml," the origin (the coordinates of the Turtlebot when the map making process began), and the image resolution.

Maps made by Turtlebot using GMapping are far from perfect. When viewing the map being made in RViz, the user can sometimes see the map rotate at an angle or shift, either horizontally or vertically, in the middle of making a map. This can be due to the robot's "confusion" from poor odometry. If the robot's wheels slip in the middle of making a map, the new distance between the map origin and the current position is not equivalent. The robot will now believe that it is 'x' amount of wheel turns ahead of its actual physical position. In addition, depending on the camera, depth sensors, and laser range finders used on the robot, the accuracy of the robot-made map will vary. Furthermore, when driving the Turtlebot around, the user will see that the map will have missing sections that the robot does not see, as shown in Figure 1.1. The user will need to drive the robot over the missing map section many times before the robot recognizes the spot. The Turtlebot odometry was poorest when Turtlebot drives slowly over slippery cement floors, so in similar conditions the user should try to ensure to move at a decent pace when making the map. They should drive the perimeter first, and then finish in the middle of the hallway or room. Turtlebot uses dead reckoning to match up places and objects it has seen in relation to its current position.

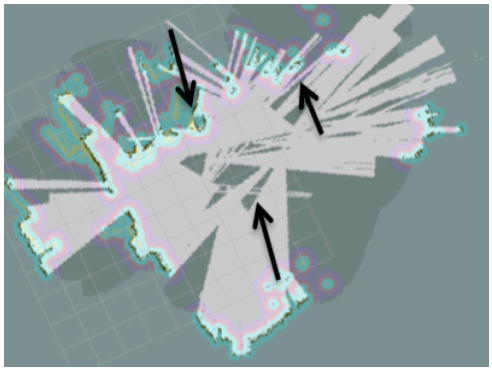


Figure 1.1 Robot-made map created with GMapping illustrating the gaps made in the middle of the map.

‡† RViz is a GUI used by ROS applications for more info see: http://wiki.ros.org/rviz/Tutorials

Another potential disadvantage of creating a robot-made map is that it can be very time consuming if the user is attempting to make a large map. In the section of map in the blue box in Figure 1.3, a relatively small hallway took about a half hour to create by driving the robot around. The whole map shown in Figure 1.3 is only the section of map in the red box in Figure 1.2, the entire floor plan. That red box only encompasses the portion of the hallways that are the perimeters around the map, and it does not include the rooms. The rooms in the middle were not mapped in this process. The perimeter took approximately 4 hours to map by robot. By extrapolation, the entire floor plan would take around 40 hours. Therefore, the user should consider the battery lifetime for the robot, robot laptop, and user laptop when creating the map. In this case, small areas of the map should be completed at a time. Photo-editing software, such as Gimp on Linux, can be used to stitch the multiple map sectors together to create a large master map, although this introduces human error as well as bot error, and a new "yaml" file will need to be created in this case for the new map. In addition, if the accuracy of the bot map is poor, or a map shift occurs and is not recognized and fixed during the map creation, then stitching portions of the map together will not be possible. That sector will need to be re-mapped and re-stitched together. However, a shifted map may be stitched in the beginning, and the user may not realize that the basis of the master map is now distorted, which is very plausible depending on the floor layout being mapped. It is also possible that a newly inserted map section could be correctly mapped, but may not fit into the current master map because of the original shifted map. This can get the user stuck in a loop, not knowing which sectors are correctly mapped and inserted into the stitched map.

Figure 1.2 *Floor Plan Map*

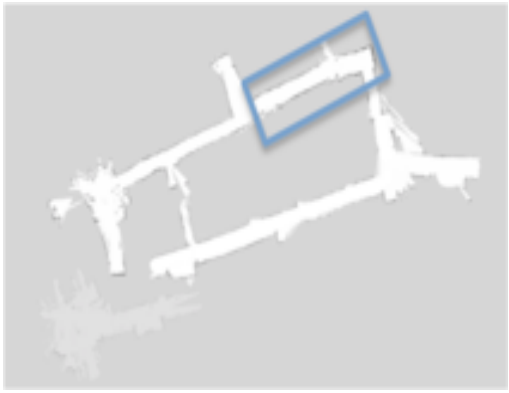


Figure 1.3 *Robot-Made Map*

Using a building floor-plan to create a ROS map.

If the user has access to a floor plan of the building, it could be converted into the appropriate "pgm" and "yaml" that the robot reads as a map; then the time consuming and lack of accuracy issues would be eliminated. We created a python program, MakeROSMap.py[‡], that is designed to allow the user to input a floor plan of any picture type ("jpeg", "png", etc.), and then the program will convert these pictures into a ROS map (the "pgm" and "yaml" files)

‡ To download code see: https://drive.google.com/file/d/0B2AcDRX3bKLVdjhPU1B2UUNRaDA/view

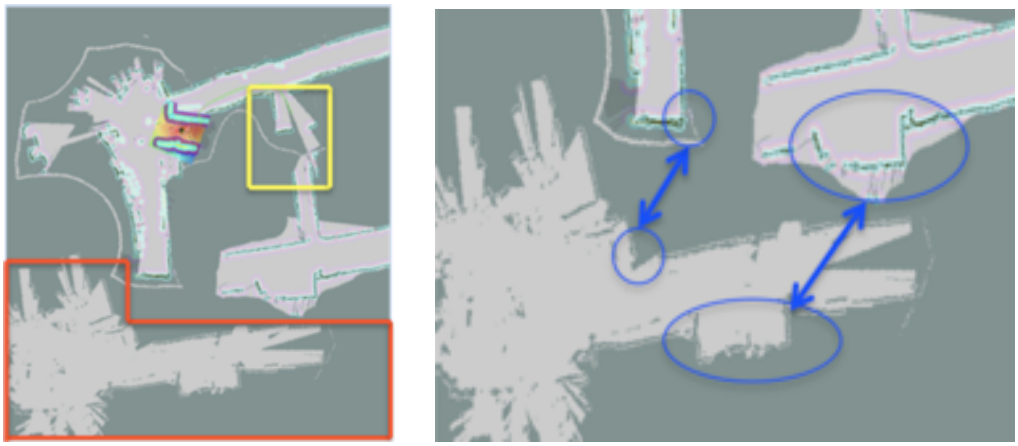
needed by Turtlebot. The user provides the program with 4 (x, y) coordinates: 2 in the x direction and 2 in the y direction. The user then inputs the x and y distances between the two points in order for the program to scale the floor plan correctly. Without appropriate map scaling, the Turtlebot will be at a different point on the map displayed in RViz than in its actual environment. The map needs to be scaled to the Turtlebot's environment (See https://www.youtube.com/channel/UCQgDH1KtuoDZkafNM0QiC-A for additional details). Furthermore, the program will output a "yaml" file for the user to associate with the map.

There are two main concepts to note when running this program. First, when selecting the two coordinate points, for both x and y, the user should ensure that the points selected are linear with respect to each other. The code then calculates the distance between the two points, assuming the points are horizontal and vertical to one-other. Without this close linear approximation, the map will not be scaled appropriately to the map's physical location. Second, the program will ask the user where to save the program's output (the map "pgm" and "yaml"), and it will save it to that location. If the program is not run on the Turtlebot laptop, the "pgm" and "yaml" files will be saved to the right location on the file system on which the program is run, but it will be on the wrong computer. When loading a map onto Turtlebot, the map must reside on Turtlebot for it to work accurately. The user needs to save the program output to the desired location, and then transfer the output to the Turtlebot. This will allow for the program to write the correct data to the "yaml" file.

Map options comparison.

Based on our test results, creating a map from a building's floor plan is much quicker than creating a robot-made map using SLAM and GMapping. We found that it will take the user only a few minutes to input the original floor plan and have a completed ROS map converted and resized. As stated above, creating a map by driving the robot around can take many hours, whereas running this program can take less than 5 minutes. To determine which type of map, either a robot-made map or a floor plan map, the robot drives better on, the Turtlebot was sent to different locations separately on both maps, and the performance was analyzed.

On the robot-made map, Turtlebot drove rather well despite its imperfections. Our observations show that the Turtlebot relies on both cameras and sensors in addition to the map, leading for the performance to be better than if Turtlebot was relying solely on the map. There are 3 main problems that were observed when driving Turtlebot on this robot-made map. The maps shown in Figures 1.4a,b were previously edited in Gimp as explained earlier. Any missing gaps were filled in and small sections of the map were stitched together to create a master robot-made map.



Figures 1.4 a,b *Highlighting problems encountered in the robot-made mapping method*

The first problem encountered in this map was the stitching process. Figure 1.4a shows 2 of the problems highlighted in the yellow and red boxes. The yellow box shows that the hallway has almost doubled in width, which occurred when Turtlebot shifted its map's coordinate system during the map making process. A portion of the hallway is long enough to connect with the hallway on the other side, but they unfortunately do not line up. Another robot-made map of this section would need to be created and re-inserted to fix this error. The red box shows a section of the map that was created and was ready to be stitched into the main map. However, as seen in Figure 1.4b, the two sections in the blue circles should line up. When trying to fit them together in Gimp, one circle's corners aligned correctly, while the others did not, and the map was significantly rotated, preventing it from being stitched in. Therefore, this robot-made map, which took around an hour to create, would need to be re-done. It will be hard to notice if the error is in the main map or in the map section that is about to be stitched. If it is the former, the problem will escalate, and the main map would need to be scratched.

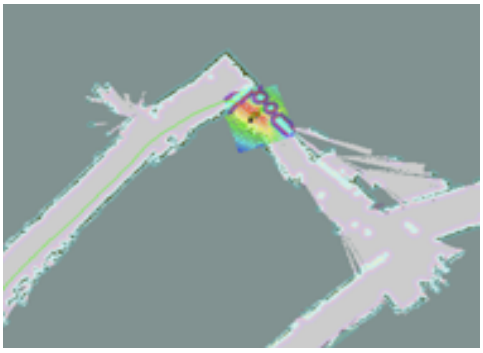


Figure 1.5 *Corners that do not match up for the stitching*

Secondly, the robot seems to lose its position during cracks or unevenness in the ground. Figure 1.5 demonstrates how the robot rotated its position from where it should be on the robot-made map. It is possible for the robot to recover from this rotation, but the recovery rates from this experimentation do not seem to be high.

The last issue in the robot-made mapping process can be seen in Figure 1.5. The

dimensions of the upper hallway are longer than in reality. The robot's map shows that it did not reach the place to turn left, but the robot sees a wall in front of it. This could either be a bot made map error, if there was a wheel slip during map making, leaving the robot to believe it was in a different location, or a human-made map stitching error. The robot now cannot drive on this section of the map because it will not be able to turn the corner. It will get stuck and start rotary behavior, but it will not be able to recover and find where it needs to go.

Maps created from building floor plan.

There were two noticeable occurrences that happened when using a converted floor plan map. First, similar to the robot-made map, the robot tended to lose positions around cracks in the floor. One difference, however, is that the robot rotated its location and position in the robot-made map, whereas the robot simply jumped forward on the map in the floorplan map. In both cases, the robot seemed to not recover.

The other behavior noticed on the floor plan map was that Turtlebot had trouble navigating tight corners. Figure 1.2 shows the floorplan. Turtlebot had trouble navigating the corner highlighted by the blue box. If the route Turtlebot decided was the shortest included turning the tight corner, it would re-route itself using a different path, but it would ultimately reach the destination. Turtlebot was able to turn around corners with a 'T" shape best, but had more trouble when the corner was an "L" shape (a corner that needs to be taken on this map). This could not be tested on the robot-made maps because, as stated above, the hallway was longer in the map than Turtlebot saw in the environment. Turtlebot would not turn this corner of the other map because when the robot was at the wall, the map told it to go a few feet farther before turning. This does not occur with the floor plan generated maps because the floor plan is scaled appropriately in the python script.

2) Leader Follower.

As a starting point for this portion of the work we require the basic ArUco Tag identification and location software example provided by The United States Military Academy's github repository†. Their work was a continuation of other contributions that can be found on the web. Figure 2.1 shows the published information from the libraries. As can be seen in the figure, specific tags are identified and that information is published. Moreover, the vector data for the ArUco tag and its orientation are given as well as the usb camera's vector and direction. Additionally a distance vector between the two objects is estimated in vector form. The information is from a capture window using RViz.

† For ArUco Marker detection.see: https://github.com/westpoint-robotics/aruco_ros

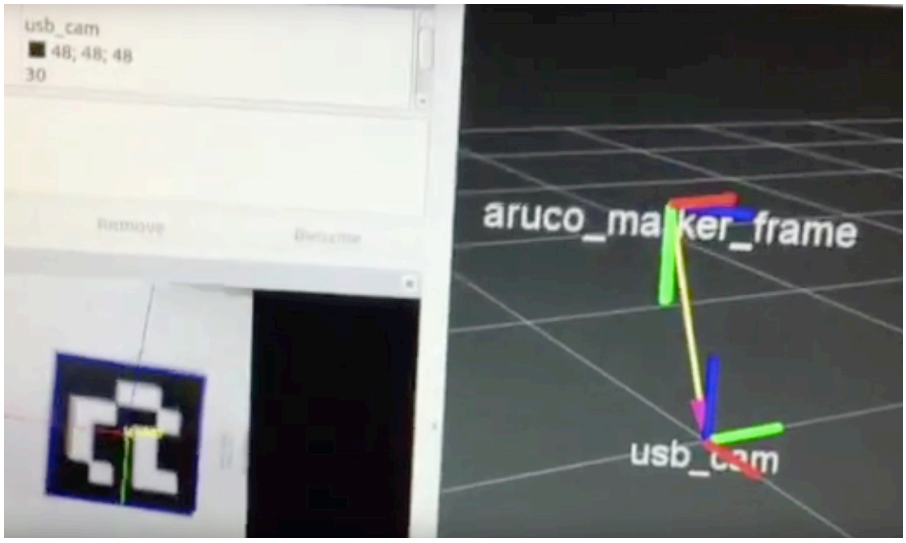


Figure 2.1 ArUco tag identification and Vector directions as viewed in RVIZ

Using ROS and project based learning, a follower routine based on the ArUco software library (Ladelfa et.al. 2014) was used. This project was slightly more complicated due to the need for a control algorithm to coordinate the Kobuki's movement with the robot carrying the symbol it was following. Another complication was that the vector data is not perfect and required some form of compensation for spurious entries. To that end the student researched and used a Kalman filter on the vector data. For the actual movement a Proportional Integral Derivative (PID) controller was used. Once again the student needed to research and understand how to implement the PID constants, since no mathematical model had been developed and there wasn't sufficient time to create one. Traditional experimental methods were used to obtain suitable gain constants. In the end, the Kobuki Turtlebot would follow the marker with good results provided that the leader did not go too fast or turn too quickly. This leaves open the possibility for future improvements and additional student work.

At the start of the semester, one of the students commented that “I had only a basic understanding of the Linux operating system.” Further, he did not know anything about ROS or any of the tools that would be needed to implement this portion of the project design. In fact none of the students had knowledge of many of the sciences that would be required to move these project forward. Patrick is also the only student that can continue his work in the area for a second semester due to other academic requirements for graduation. Ultimately we will be attempting to move the navigation onto a quad copter in the second semester. This is by far a more difficult task as we will have no localization through GPS while indoors. Originally, efforts in this area used a Vicon or equivalent system to localize their quad copters. This was done with great accuracy at the Grasp Lab at U-Penn (Mahony et. al. 2012). Subsequently others have used cameras and other means to stabilize quad copters for indoor flight. Along those lines we will use the ArUco tags as visual markers that the quad copter can follow and a sonar device to control altitude. Hence, this was a logical progression towards our end goal of hovering a quad indoors (Torn et. al. 2014).

3) RF tags and object Location (Target Tracking and Simple Navigation)

Another challenge undertaken by the students was simple target tracking using computer vision to locate and scan an RFID tag. By using the on-board Turtlebot camera and an off-the-shelf RFID reader, the bot can search for distinct objects in a space and navigate towards them. The bot needed to realize it had reached the target so that the RFID tag could be scanned[‡†]. This ability is a useful precursor to localization in navigation tasks as it provides an absolute and accurate reference for position. Figure 3.1 shows the Kobuki and box it needed to find. The bot's navigation was tied to the location observed and the robot moved towards the object until the RF tag could be detected. In future applications we will include an arm so that the bot can retrieve the object from the floor. We also made use of color to help find the box and in this work the boxes were red.

Figure 3.1 Bot and object recognition.

For this application two ROS nodes were used with each running on separate hardware: the Turtlebot itself and an external laptop to perform the computer vision tasks. On the Turtlebot node the hardware would continuously scan for an RFID tag, and when discovered, would publish that information over a ROS topic. The onboard camera information was published onto its own topic to be processed by the laptop node. This was done to take advantage of the higher performance and utilize OpenCV[††]on the laptop node. Figure 3.2 shows the OpenCV vision recognition of the box. To simplify the task, we started by using a red box and color recognition to localize the box. In subsequent projects we will use edge detection and shape rather than color alone further complicating the process and leading to a more indepth understanding of the power of OpenCV. In general, the bot would rotate on its axis until the target object was seen. Then the bot would center the object while moving forward. For our application the target was a bright red box. OpenCV and Python were used to filter the incoming camera data, remove all colors that were not near the target color, and then perform blob detection to find the likely position of the box. Based upon the box's location in the camera screen, velocity commands were sent to the Turtlebot to move closer to the target. As the blob grew closer the bot would continue forward until the blob exited the lower part of the screen. At this point the bot would decrease velocity until the RFID was detected.

‡† See: https://learn.sparkfun.com/tutorials/sparkfun-rfid-starter-kit-hookup-guide
†† See: http://opencv.org/

Figure 3.2 Box location using OpenCV libraries and color recognition.

Overall the application performed well but the bot movement was not smooth. Furthermore, a fixed color threshold used for blob detection saw reduced performance if other similarly colored objects were present. One drawback of this application is the "lag" caused by the communication between the two nodes. This was necessary because the netbook controller used with the Turtlebot lacked sufficient computing power to efficiently use the computer vision algorithm. While the Turtlebot and the laptop were on their own private network, the latency between the two also cause the bot to overshoot its target. Future efforts for this project will make sure that each node runs on the same hardware platform thereby ensuring smooth and accurate operation.

Conclusion

The purpose of this paper was to document the progress we made in one semester toward projects that would contribute to the growing body of open source work with ROS and the Kobuki Turtlebot. The students made good progress and their individual projects are being used by new students in our second semester, with ROS for project based learning. The step by step details of the students work made it far easier for the second semester students to duplicate and understand what had been done and use it rapidly. For example, a student this semester is using the map making details and incorporated voice commands to allow visitors to ask the robot where a particular office is and have the robot take them to their destination. Another is using what was learned with the maps to check for open doors after hours. As this is a work in progress the overall educational experience along with associated details will be detailed by faculty researchers in a subsequent publication.

In order for this publication to merit consideration we include all of the code with some instructions on the web (see Table 1 below). Further we have included some instructional videos on the web and as of this writing they have been viewed approximately 100 times. This is after 3 months of being available. Nonetheless, this data is subjective at best as we do not yet know who, if any, used these findings to move their projects forward. The details in this publication will allow others to avoid some of the issues found when using standard Turtlebot routines. It will also provide an initial source of estimation into the utility of the approach of using ROS and PBL as an undergraduate research catalyst.

For this initial report all of our original project goals were not realized to the degree we had hoped. However, the idea to use PBL did prove useful in terms of student motivation and learning objectives. For this project all of the students were upper level (near graduation), juniors and senior students. Moreover, they were selected based on prior performance in classes and projects at the school. This helped our success for this initial attempt of using ROS as an undergraduate research tool. As we continue to grow our program more data will become available with regards to freshmen and sophomores that will be included in subsequent faculty papers. Duplicating this work with no prior knowledge of Ubuntu, ROS or the Turtlebot among other issues would take any undergraduate student many months to accomplish. These three students were able to do this using the course of study they described and that is another measure of merit. Furthermore, the paper provides links to code and instructional videos that the students made. In the simplest form the Github repositories can be git-cloned onto a ROS enabled Ubuntu system with the correct libraries and will work immediately. Improvements to all of the approaches are possible. The details in the paper along with Github examples and videos will benefit other student programs doing undergraduate research on robots using ROS and PBL.

Table 1. ROS and Turtlebot resources.

Description	**Link**
YCP Robotics ArUco Move Overview	https://www.youtube.com/watch?v=C65wWCgdhb8&list=PL8TAyn8EupeiqkQE8NklYT6rKvYsEz9pG&index=3
YCP Robotics Map Making Overview	https://www.youtube.com/watch?v=mfZfx1gtiBk&list=PL8TAyn8EupeiqkQE8NklYT6rKvYsEz9pG&index=4
How to create a ROS Map using a Buildings Floor Plan	https://www.youtube.com/watch?v=ySlU5CIXUKE
Autonomous Driving Robots with code by Inputting Coordinates on Map	https://www.youtube.com/watch?v=Vd2fqvcC6MQ
Autonomous Driving With code Basic	https://www.youtube.com/watch?v=n8hmtVMMjaQ
Autonomous Robot Driving with RViz and Turtlebot	https://www.youtube.com/watch?v=bx5-dwQnZzA
Creating a Robot-Made Map in ROS	https://www.youtube.com/watch?v=pXllj9Q1bE0
Map Making Python code	https://drive.google.com/file/d/0B2AcDRX3bKLVdjhPU1B2UUNRaDA/view
GitHub ArUco Move Repository	https://github.com/plynn17/Aruco_move_ros_pkg
ROS-RFID Finder Repository	https://github.com/mrjones2014/ROS-RFID-Finder

References

1. Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." *ICRA workshop on open source software*. Vol. 3. No. 3.2. 2009.
2. Cousins, Steve, et al. "Sharing software with ros [ros topics]." *IEEE Robotics & Automation Magazine* 17.2 (2010): 12-14.
3. Martinez, Aaron, and Enrique Fernández. *Learning ROS for robotics programming*.

Packt Publishing Ltd, 2013.
4. Qidwai, Uvais. "Fun to learn: Project-based learning in robotics for computer engineers." *ACM Inroads* 2.1 (2011): 42-45.
5. O'Kane, Jason M. "A gentle introduction to ros." (2014): 1564.
6. Ghani, Muhammad Fahmi Abdul, Khairul Salleh Mohamed Sahari, and Loo Chu Kiong. "Improvement of the 2D SLAM system using Kinect sensor for indoor mapping." *Soft Computing and Intelligent Systems (SCIS), 2014 Joint 7th International Conference on and Advanced Intelligent Systems (ISIS), 15th International Symposium on*. IEEE, 2014.
7. Schmitt, Simon, et al. "A reference system for indoor localization testbeds." *Indoor Positioning and Indoor Navigation (IPIN), 2012 International Conference on*. IEEE, 2012.
8. Sumaray, Audie, and S. Kami Makki. "A comparison of data serialization formats for optimal efficiency on a mobile platform." *Proceedings of the 6th international conference on ubiquitous information management and communication*. ACM, 2012.
9. La Delfa, Gaetano Carmelo, and Vincenzo Catania. "Accurate indoor navigation using smartphone, bluetooth low energy and visual tags." *Proc. 2nd Conf. on Mobile and Information Technologies in Medicine*. 2014.
10. Mahony, Robert, Vijay Kumar, and Peter Corke. "Multirotor aerial vehicles." *IEEE Robotics and Automation magazine* 20.32 (2012).
11. Toma, Antonio, et al. "The Vision-Based Terrain Navigation Facility: A Technological Overview." *International Workshop on Modelling and Simulation for Autonomous Systems*. Springer International Publishing, 2014.