

Duckweed Robot Harvester

CS39440 Major Project Report

Author: Benjamin Dunn (bed45@aber.ac.uk)

Supervisor: Dr Fred Labrosse (supervisorid@aber.ac.uk)

6th May 2022

Version 1.0 (Release)

This report is submitted as partial fulfilment of a BSc degree in
Artificial Intelligence And Robotics (GH76)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

I confirm that:

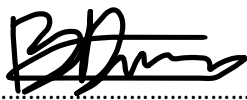
- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work, I understand and agree to abide by the University's regulations governing these issues.

Name: 

Date: 06/05/2022

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: 

Date: 06/05/2022

Acknowledgements

This is not my first time in University. I previously attended the University of Houston and due to my lack of motivation and vision flunked out, leading to a long subsequent period of shame. From 2014 to 2019 I moved between minimum wage jobs with no real direction in life.

When returning to the United Kingdom, I was fortunate to have my wonderful grandparents, Keith and Pamela Brooks, to help support me. Their aid helped set me on the right path.

Savannah Bland, my dear friend and colleague, encouraged my return to higher education and was in large part the spark behind why I have spent the last three years working hard to make something of myself. While I believe we are all capable of doing great things, the support and encouragement from those around you can be immensely valuable, even essential, in taking the first step.

Abstract

Duckweed is an invasive plant species that grows upon the surface of still water such as ponds. The plant is very capable of taking over an entire pond surface with ease, completely clogging the surface and consuming all the nutrients from the water preventing other species from being able to grow. Further to this, it blocks light from other fauna and flora below the surface.

Common tactics to remove duckweed includes manually raking the duckweed out or using a high-powered pump to suck up the surface water and filter out the duckweed. This project analyzed the possibility of developing software for an autonomous water-based robot that could rake the duckweed itself and transport it to a pump, thereby cleaning the body of water and collecting the nutrient rich duckweed for other uses. Most of the work done on this project is largely inspired and by the success and effectiveness of the iRobot Roomba and so therefore largely modeled of early versions of the Roomba. The project used ROS and Gazebo for simulating and determining the effectiveness of the software.

Contents

1. BACKGROUND, ANALYSIS & PROCESS.....	7
1.1. Background	7
1.2. Analysis.....	8
1.2.1. Water Environments.....	9
1.2.2. Boat for Real World Testing	9
1.2.3. Obstacle Avoidance in Water Environments.....	10
1.2.4. Identifying Duckweed and Clearing it.....	10
1.2.5. Returning to the Pump.....	10
1.3. Process.....	10
1.3.1. Project Management Methodology.....	10
1.3.2. Version Control	11
2. DESIGN	12
2.1. Overall Architecture	12
2.2. ROS and Gazebo	13
2.2.1. ROS.....	13
2.2.2. Gazebo	13
2.2.3. Language Choice	14
2.3. Environment Design	14
2.3.1. Gazebo Water Environment Overview	14
2.3.2. Duckweed Representation	14
2.3.3. The Robot Requirements	15
2.3.4. Different Environments Required For Testing	15
2.4. Detailed Design.....	15
2.4.1. Logging GPS Position	15
2.4.2. Moving Around The Environment.....	16
2.4.3. Avoiding Obstacles.....	16
2.4.4. Spiralling Out.....	17
2.4.5. Navigating to Initial Position.....	17
3. IMPLEMENTATION	19
3.1. Topic Layout	19
3.2. Simulation Implementation.....	19
3.2.1. Generating Environments for Simulation.....	19
3.2.2. Summit XL and Launch File.....	21
3.3. Python Script Implementation.....	21
3.3.1. Logging GPS Implementation.....	21
3.3.2. Movement Implementation	21
3.3.3. Moving to the Centre Implementation	22
3.3.4. Obstacle Avoidance Implementation.....	22
3.3.5. Spiralling Out Implementation.....	22
3.3.6. Returning to Pump Implementation.....	23

3.4.	Testing Implementation	23
3.5.	Overall Implementation Success.....	24
4.	TESTING	26
4.1.	Overall Approach to Testing.....	26
4.2.	Simulated Testing	26
4.2.1.	Small Area	26
4.2.2.	Small Area with Obstacles	27
4.2.3.	Medium Area	28
4.2.4.	Medium Area with Obstacles.....	29
4.2.5.	Large Area.....	31
4.2.6.	Large Area with Obstacles	32
5.	CRITICAL EVALUATION	33
5.1.	Requirements.....	33
5.2.	Design and Implementation Reflection	33
5.2.1.	Software Design	33
5.2.2.	Software Implementation.....	33
5.2.3.	Environment Design and Implementation.....	33
5.3.	Testing Reflection	34
5.4.	Future Work.....	34
5.5.	Summary	34
6.	REFERENCES.....	36
7.	APPENDICES.....	38
A.	Third-Party Libraries	38
B.	Third-Party Code.....	38
C.	Third-Party Gazebo Files	39

1. Background, Analysis & Process

1.1. Background

Across the world we are seeing major threats to the stability of our environments. Global warming, overpopulation, ocean acidification, shrinking bio diversity, deforestation; these are just some of the many problems we face, and will continue to face in the foreseeable future [1]. These large problems will demand large societal changes, but fortunately smaller environmental problems can be solved with simpler solutions.

As mentioned in the abstract, duckweed is an invasive species and can rapidly overtake the surface of any still body of water. Duckweed will thrive in nitrate and phosphate rich waters due to their tolerance to these environments [2], unlike most other organisms that exist within British bodies of water. New ponds that are recently created deal with duckweed and similar species in a cyclical nature. They are present for a few months after they initially bloom, and then tend to subside. This will typically continue for several seasons before the ecosystem naturally finds a balance and the duckweed no longer becomes an existential threat. Older still bodies of water that fail to reconcile with duckweed are likely due to a number of issues [2]:

- Fertilizer from nearby gardens/fields is absorbed by the local body of water
- Pollution from livestock or duck faeces
- Addition of nutrients to the environment from water inflow, such as streams connecting to the water body
- Sewage, farmyard run-off
- Additionally anything that consistently causes sediments on the bottom of environment to become agitated. This agitation can potentially cause nutrients from the sediment to be released. Typical culprit behind this involve ducks and bottom-feeding fish.



Figure 1: An example of Duckweed completely dominating an ecosystem

Like algae, an organism that similarly thrives in these nutrient rich waters, duckweed can provide food but additionally shelter for other wildlife [3]. Moderate amounts of duckweed is perfectly acceptable and can help sustain an environment. But when rampant, duckweed

will block light to subsurface wildlife and even cause drops in oxygen levels below the surface, damaging the ecosystem and causing further homogenization of the environment [4].

Due to the large threat duckweed poses to these environments, it must be carefully monitored to ensure it does not get out of control. Unfortunately, most of the accelerants for duckweeds growth cannot be completely solved by individuals and therefore preventing duckweed blooms entirely is an unreasonable solution. Individuals have instead looked to removing and culling the duckweed manually at such times that the duckweed becomes a threat. These methods typically include:

- Manually rake the water's surface
- Using high powered pumps to filter out surface debris
- Applying chemical treatments to the water

Each of these methods have issues [2]. Raking the water's surface manually can become very time consuming depending on the size of the water. In large bodies of water, wading into the water may be required, and if the water is deep enough a boat may be required to get areas otherwise inaccessible. High powered pumps require some set up and so the manual aspect isn't completely circumvented but they are largely effective in cleaning a surface, and for large bodies of water are the current industry standard. The price for such devices however can be a large barrier to entry for many individuals [5] and do require regular maintenance. Lastly, chemical treatments can be applied to the water, but selecting the most appropriate one for an environment can be an entirely new problem. You don't want to be adding chemicals that can have further adverse effects to the environment.

Finding a new cost effective treatment to duckweed without manual labour and fears of further damaging the environment would be hugely beneficial to ecosystems across the world.

1.2. Analysis

The use of automation has forever changed the way countless different industries produce products, and thanks to recent breakthroughs in humanities understanding of robotic applications society is taking automation a step further, replacing old menial day to day tasks with autonomous robots. One of the stronger examples of this is the widespread use of the iRobots Roomba [6]. Across the world people are replacing a time consuming chore with small robots that will now vacuum their households. The main objective of this project is to determine if the laborious task of cleaning away duckweed can be effectively automated, not unlike the success of the Roomba.

From initial background work it was clear that solutions for maintaining ecosystems with duckweed either involved harsh chemicals, expensive and maintenance intensive water filters or large amounts of manual labour. The solution of raking the water body with an autonomous robot is very obviously the easiest solution to implement as the problem has already been largely solved by the Roomba. Transferring the application of the industry standard Roomba onto water very quickly became the priority for this project.

1.2.1. Water Environments

A key difference between the environments Roombas typically move around in and the environment duckweed exists in is water. Unlike flat hard surfaces, water is a fluid with no fixed shape. Fortunately, duckweed will only form on still bodies of water as too much agitation in water creates a hostile environment for duckweed, as agitating the duckweed reduces its ability to photosynthesize, reduces its ability to effectively grow roots, and often leads to the duckweed being pushed to the edges of the water, piling upon itself, leading to self-mulching. Working in still body environments greatly simplifies the environment for our robot and therefore greatly simplifies the project. Wave like environments will not be a problem for the robot and so can generally be treated as if it were in fact a flat solid surface. The major two problems that can be created by this water environment is:

1. Drift. Water, unlike typical hard surfaces robots operate on, will cause the robots forward momentum to be somewhat retained. As long as the robot maintains slow speeds this issue can be minimized as minimizing the speed will minimize the momentum generated. Additionally depending on the robot used, slight contact with the perimeter of the water body or objects within it could be mitigated with the robots design. In other words, design a robot that is built to withstand small collisions with the environment and be able to push itself back into the water if it inadvertently drifts out.
2. Subsurface objects. Large rocks barely below the surface could be hard to detect and avoid. This could be solved with sensors that can detect such objects (sonar). Alternatively a water based robot could be designed to sit mostly upon the surface of the water and so can simply move over these objects with little to no trouble.

1.2.2. Boat for Real World Testing

Early on a small remote control boat was selected for real world testing on the effectiveness of software produced by the project. The Aquacraft Cajun Commander Brushless RTR Airboat [7] covers the requirements of being able to manoeuvre around a still body environment, is made of durable plastic allowing to for small collisions, and its flat bottom hull means it should effectively be able to move over any obstacles just beneath the surface as well as dislodge itself from any surrounding banks of the water body and return to the water. Unfortunately due to hardware shortages the boat was unavailable for purchase and so could not be tested on. In Gazebo a similar scale robot was selected to emulate the size of the Cajun Commander.



Figure 2: The Cajun Commander

1.2.3. Obstacle Avoidance in Water Environments

Determining how to navigate the environment, locate duckweed, clear it and return to the pump for collection was the next step in the project. For obstacles avoidance, use of a depth camera to calculate the distance to the water's surface was determined as an effective way for avoiding obstacles. Any large changes in this depth measurement would suggest obstacles in the way. Using this method obstacles can be recognized and avoided. While it is typical for LIDAR or infrared to achieve this obstacle avoidance, water environments are not conducive for reliable LIDAR/infrared results [8].

1.2.4. Identifying Duckweed and Clearing it

Following the Roombas example, simply covering as much of the water body as possible would effectively clear it of the majority of the duckweed. This makes any algorithm that needs to be created very simple. Unlike clearing dirt from household floors, duckweed is readily visible and so presents a secondary solution to targeting it. Using the camera on the robot, guiding the robot to areas with green could be used to more quickly and efficiently clean areas. It does however run in to problems such as the water appearing green due to reflections on the surface from surrounding vegetation. Instead of trying to solve these new problems, a determination was made to stick with the tried and tested Roomba model.

In order to collect the duckweed, a simple rake/scoop like object could be attached to the front of the Cajun Commander. Along the sides of this rake its advisable to have small plastic walls along the ends to prevent any duckweed being lost while turning.

1.2.5. Returning to the Pump

As part of the project the autonomous robot must be able to navigate itself back to a pump once duckweed has been collected. Two methods could be used in conjunction to solve this problem. Starting the robot near the pump and using onboard GPS will allow it to mark where the pump is located within the environment. When sufficient duckweed has been collected the robot can then navigate its way back to the pump with GPS. Mapping the environment as it moved around would allow for this. Alternatively simple bug algorithms could instead be used to avoid any obstacles and return to the pump [9].

1.3. Process

1.3.1. Project Management Methodology

The programming methodology used for the project was Agile Scrum [10]. Agile Scrum focuses on continuously updating and revising different aspects of a project as needed, unlike other software approaches that attempt to deliver everything at the end of the project. In order to have deliverables over the different versions of the project, different required functionalities must be identified early on. From there, these functionalities are organized and prioritized based upon importance. "Sprints" are used, which are 2 week long periods of development. At the start of this period, planning is done to decide which parts of functionality are left and what will be prioritized. Daily meetings are held to keep the process running and any problems can be brought up and identified to the team to avoid any large problems at the end of development. At the end of a sprint there is a review of the periods work. This review looks at the sprint period as a whole and attempts to identify areas of strength and weakness across the project.

This project was ran by a single person and as such changes to the project process were made. Daily meetings to asses daily work were not held as the only person to communicate with was myself. Sprints were still typically two weeks long, though toward the end of the project timeline (last 6 weeks), they were reduced to a week to allow for greater adaptability and neatly timed up with the weekly meetings with the project supervisor. These meetings helped stand in as the sprint review and allowed for reflection and fresh ideas going into the next sprint.

In general Agile Scrum felt like an effective choice of process for the project, allowing for shifts in priority for the project, recognition of different potential methods for achieving the stated goals, and active testing in the creation of different functionalities. Using this process helped keep the project on track with weekly goals and kept motivation high when seeing the workload reduce over time.

1.3.2. Version Control

GitHub [11] was used for version control throughout the project. Having used it in previous semesters no issues were encountered and helped maintain a smooth process to the project.

Virtual machines (Oracle VM Virtual Box [12]) were used throughout the development process and as such allowed for cloning of different VM's. Cloning them allowed for backups so any catastrophic errors or problems did not cause large quantities of work to be lost.

In both cases no backtracking of software was needed but it was none the less essential to have backups encase of hardware or software faults.

2. Design

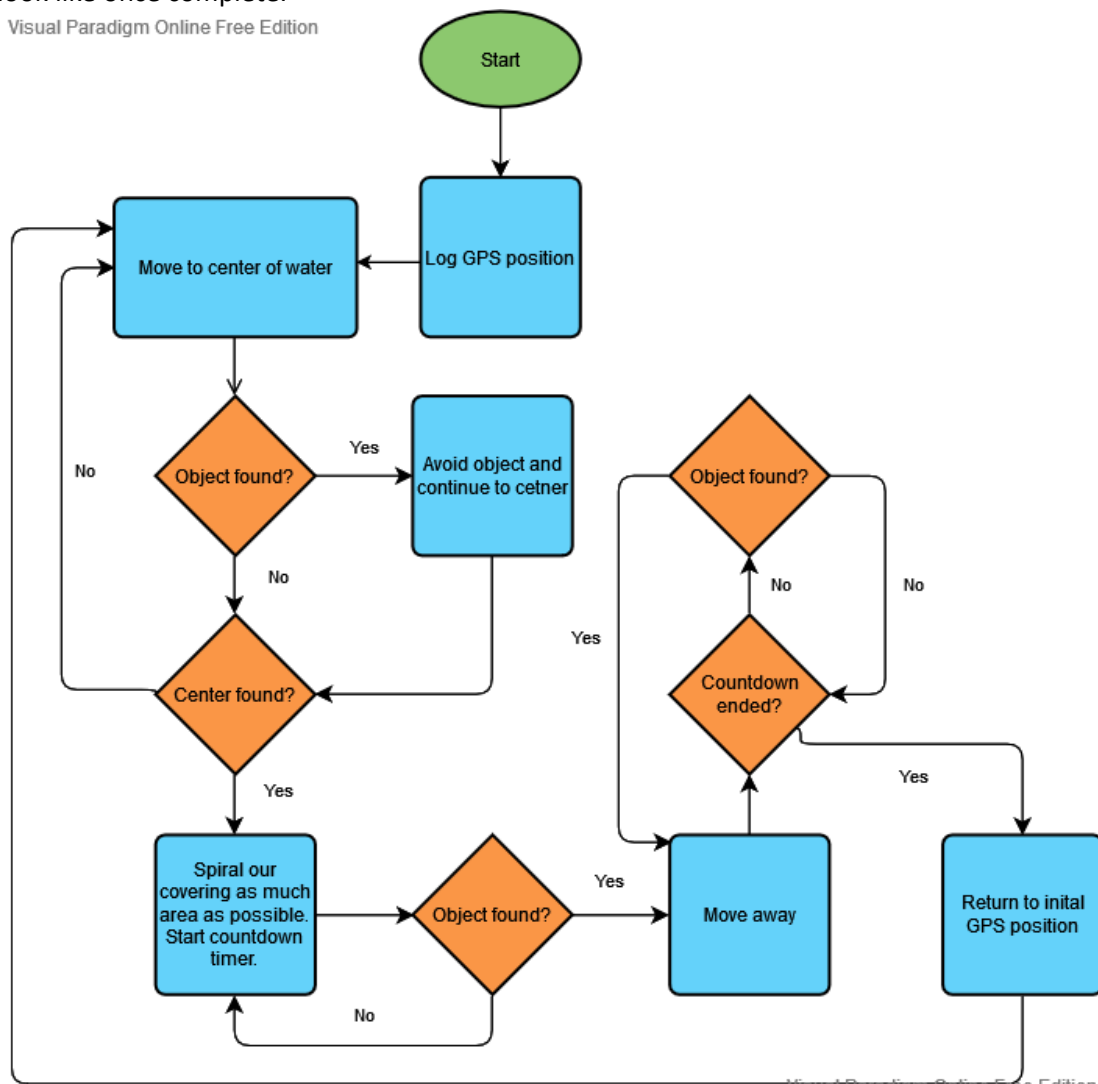
2.1. Overall Architecture

The core of the project and its design was to ensure simple to complex bodies of still water could be cleaned of duckweed. From a design perspective it was imperative to ensure each major feature was sufficiently planned out in terms of their own functionality and how these different parts of the software would interact with one another. As stated in the background section the main functions of the project include:

- Being able to navigate the environment
- Avoid potential obstacles
- Return to the pump intermittently with the gathered duckweed

From this point I began making a simple controller diagram to plan out how these functions would interact with one another and give a visual understanding of how the project would look like once complete.

Visual Paradigm Online Free Edition



Visual Paradigm Online Free Edition

Figure 3: Controller Behavior Diagram

Figure 3 presents the essential parts of the projects required functionality. Several times throughout the project changes to the 'how' it completed tasks were made but in general the overall design did not change and remained the same throughout its development.

The robot begins with initializing the start location. This should be done next to the duckweed pump to allow for its harvesting. Once initialized it will attempt to find the centre of the body of water. After several attempts of estimating the centre with environmental feedback in Gazebo, I determined a simpler and often more accurate way would be to simply ask the user how far they estimate the centre is from the point of initialization. At this point the robot moves forward to reach this centre position. Any objects that are detected along the way are circumvented and movement toward the centre continues. Once movement to centre is completed the robot will move out in a spiral like movement. Additionally it will start a countdown timer to suggest when the robot has collected enough duckweed. Moving in a spiral pattern will help ensure it covers as much area as possible in the initial steps of its gathering. Once an obstacle is detected in this function it will navigate around the environment, moving away from any obstacles until the countdown timer ends. At this point it will return to the initial GPS location marking the pump.

It should be readily apparent that much of the design is modelled of the effectiveness of early iterations of the Roomba. It was important for me to find real world examples of effective cleaning robots and the Roomba was a perfect example. Furthermore, it's very simple design makes the process of emulating it easy.

2.2. ROS and Gazebo

2.2.1. ROS

ROS (Robot Operating System) is a system created to make controlling robotic components from a PC easy via the use of different 'nodes' communicating via publishing and subscribing [13]. More specifically sensors on a robot publish sensor data through a flow of individual messages. Now these messages can be made use of by many different nodes across various systems, including multiple computers or platforms. This is very useful for robotics, allowing easy communication between say an Arduino on a robot, a mobile form and a desktop simultaneously. This streamlined way of working with robots has a steep learning curve to start but can become incredibly useful once understood.

While there are other robotic operating systems on the market, my previous use of ROS last semester gave me some experience with the middleware. Therefore it made sense for me to continue with ROS instead of starting from scratch and trying to learn a different system. ROS is widely used throughout the robotics industry and so makes perfect sense to use in this project.

2.2.2. Gazebo

Gazebo [14] allows users to create 3D environments complete with realistic physics in order for simulations of real world scenarios to be developed. Testing robots within simulated environments avoids any potential damage that could otherwise be inflicted upon your robot (or people) in real world scenarios. Quickly resetting a simulator also allows for faster testing. In conjunction with ROS this powerful testing can be done.

Similarly to ROS, Gazebo was chosen due to previous experience with the simulator. Initial plans were made to create a robot within Gazebo but difficulty in creating this model were experienced – more acutely with getting the GPS plugin to work [15]. This is largely what led to the decision to make use of an already built open source robot.

Environments within the simulator were simpler to create and that process will be detailed later in the design section.

2.2.3. Language Choice

Within ROS two languages are used; either C++ or Python. A useful feature of ROS is its ability to have C++ and Python nodes communicating without issue thanks to the standard ROS messages. Ultimately python was chosen as the language for the project.

From quick research on both languages it was determined that building in Python would allow for fast creation of different functions thanks to the language taking care of large parts of programming by itself. This would give more time to test different functions that would complete the same objective. In hindsight, this was very useful when it came to building the project and deciding on which approaches worked best. Not having to fight with a language and be confused with bugs undoubtedly saved large amounts of time.

While easy to use and learn it was important to consider the fact that Python does run slower compared to C++, something that could be an important factor when it comes to robotics and is largely considered the standard in the industry. However focusing on algorithms rather than language, and previous experience using Python within ROS determined the choice to use Python for the project.

2.3. Environment Design

2.3.1. Gazebo Water Environment Overview

Since this project had no real world testing having good simulations of real world environments was important. Initially the environments were going to be simulated water environments within Gazebo. However typical open source water environments within Gazebo simulate sea based environments or more specifically environments with active waves, something that will be greatly minimized in a still body environment. Having little experience with designing environments within Gazebo and recognizing the reduced importance of including such physics, removing water based physics within the environment would allow the project to move forward with developing other more important aspects of the project. As such the Gazebo environment would be a typical flat land based environment to mimic that of still body water and a simple land based rover would be used to represent the boat.

2.3.2. Duckweed Representation

To represent the duckweed within the environment an initial plan of using small thin cylindrical objects was determined to be effective in representing the duckweed. However creating large amounts of these objects can lead to large amounts of calculations within the simulation, slowing down the program. A simpler and more elegant way of representing the duckweed was required.

The solution was not to represent the duckweed at all. Instead I would track how effective the robot was at covering areas of duckweed by simply tracking what percentage of the area had been covered by the robot. Assuming the robots rake was good at gathering the duckweed in front of it, the problem now became ensuring the robot covered as much area as it could to maximize the gathered duckweed.

2.3.3. The Robot Requirements

The chosen robot would need to fulfil all the functionality stated in the background; being able to navigate the environment, avoid potential obstacles, and return to the pump intermittently with the gathered duckweed. Therefore the robot would need to at the very least be able to move, perceive obstacles in its vicinity, and log its position with GPS. The official ROS website has a large collection of open source robots to be used within Gazebo [16]. Making use of this collection, a suitable robot that could comfortably achieve all the functionality was searched for. The Summit-XL is capable of achieving the objectives of the project thanks to its onboard GPS and camera [17]. Additionally it exhibits similar size specifications as the Cajun Commander [7] in terms of length and width allowing a more accurate simulation.

2.3.4. Different Environments Required For Testing

After deciding to use a land based simulation several different sized environments would need to be created as they would be useful in gauging how effective the software was based on the size of still body. As such I decided creating circular/oval environments could represent small or large ponds. If time permitted I concluded I would create several different shaped ponds with different amounts of variation, some with extreme features (many obstacles, difficult to spot obstacles such as thin reeds, narrow passages formed by islands etc.) and others with very simple layouts. The simple layouts would demonstrate the robot working in perfect conditions, the more extreme featured environments would represent more difficult environments to navigate.

2.4. Detailed Design

2.4.1. Logging GPS Position

The projects use of GPS was always limited to the initial positioning for navigating back to the start and the possibility of using it to recognize where the boundaries of the environment were.

With regards to the initial position, a simple use of the initial GPS data when the robot starts very easily solves that problem. This would be sufficient for later navigating back to the pump.

A more complex use of GPS positioning was formulated for keeping the robot within the bounds of the environment. Similar to the iRobot Terra [18], consideration was made for giving the robot bounds for which to work within, but instead of using wire-free standalone beacons around the perimeter of the environment to help with mapping, simply stating the boundary of the environment using GPS would be sufficient. If at any point the robot encounters these GPS boundaries it would turn away knowing that it can't go past that point. It could be possible that the Cajun Commander would have problems (e.g. getting stuck on land) if it got too close to the edges of a pond. Without real world testing its difficult to

determine, but from reviews it does appear to be able to traverse flat land based environments. If there was a real issue then forcing the robot to work within GPS bounds would be a good solution. Since this can't be ascertained, I refrained from implementing any GPS boundaries.

The projects design for using GPS messages involved reading messages via the *gps/fix* topic.

2.4.2. Moving Around The Environment

The general design around moving through the environment was cut into three main ideas:

1. Move to the centre
2. Spiral outward to cover large amounts of area
3. Move away from obstacles in random angles to cover remaining area

For moving to the centre, simply asking how far the user estimated the centre was from the start, along with the set velocity would allow the robot to calculate if it has moved to the centre.

Spiralling out would simply have the robot incrementing its forward speed over time while maintaining a turn speed. This in theory would create a smooth movement toward the edges of the environment, covering large amounts of the area of the environment.

For navigating the remaining area once an obstacle was encountered while spiralling, use of either mapping of the environment in combination with some form of wavefront algorithm would allow for very high consistent coverage of the environment. The mapping would take into account obstacles as it went and could additionally mark where it had seen green sections along the water's surface and mark them as high priority goals.

In contrast, a secondary idea of simply moving in random angles away from obstacles until sufficient area was covered or a timer expired was considered (the countdown timer described in the controller behaviour diagram). Given the random nature of this method I determined a timer would be a far better choice since you could run into problems where certain areas were difficult to manoeuvre to, causing the robot to potentially take extremely long periods of time to navigate to these areas. This simple method takes no planning and is very easy to understand and implement. The low memory cost helps in keeping actual costs of such a robot low as well, which is imperative in giving any possible robotic product the ability to compete with the industry standard pump/filter systems that dominate it. For these reasons a decision was made to move forward with the random angle method of navigation.

The projects design for moving involved simple messages via the *cmd_vel* topic to move forward and turn.

2.4.3. Avoiding Obstacles

Obstacle avoidance was planned out in three different ways; avoiding obstacles to reach the centre to maximize the effectiveness of the spiralling after, moving away/around obstacles when they are encountered during/post spiralling, and avoiding obstacles when returning to the pump.

As stated in the background analysis, simple use of bug algorithms allows for easy avoidance of obstacles when moving to the centre and when moving back to the pump. Going into implementation these were the main ideas for obstacle avoidance. Possible use of mapping to navigate around obstacles is another obvious choice but for reasons previously stated in 2.4.2 *Moving Around the Environment* mapping was not used.

Moving away from obstacles has already been discussed in the previous section 2.4.2 *Moving Around the Environment*.

When it came to identifying obstacles I determined early on that a depth camera would be most useful in the wet environments the boat would be operating in. Refraction caused by water in the environment can cause infrared lasers and LIDAR systems to give faulty and bad data to the rest of the system, and would be a very unreliable way of detecting obstacles [8]. Therefore using the depth camera to determine changes in distance from the camera to the water's surface was determined to be the best way of identifying obstacles in the environment. Previous experience with depth cameras in ROS also helped inform the decision on potential applications of the technology.

The projects design for detecting objects with the depth camera involved simple messages via the `front_rgb_camera/depth/image_raw` topic.

2.4.4. Spiralling Out

The design for moving in a spiral pattern was discussed in 2.4.2 *Moving Around the Environment*. One issue identified for consistently increase forward speed would mean the boat could potentially get to a point where it was moving too quickly to reliably collect duckweed. Capping this speed and instead incrementally decreasing and increasing the turn velocity would enable the spiral to grow larger without comprising its ability to collect duckweed. So when the max speed is reached, maintain a forward trajectory temporarily and then begin turning again to increase the radius of the circle trajectory it is creating. Then incrementally repeat this to cover more and more area.

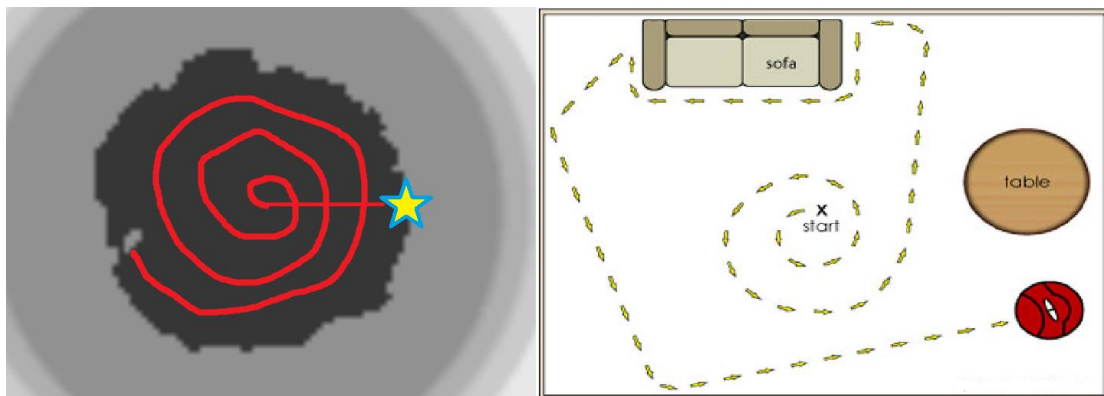


Figure 4: An example of the spiraling method to cover area. The left shows an example in a small pond. The right shows the similar algorithm working for Roomba.

2.4.5. Navigating to Initial Position

Early design for this section included mapping and moving with the navigation stack within ROS [19]. As mapping was dropped early on navigating back to the pump was expected to be handled with a combination of the initial GPS coordinates and obstacle avoidance bug algorithms, much like the initial moving to the centre of the environment.

Another idea was to have a large cyan coloured banner at the pump for the robot to see and navigate to. Cyan was chosen as it is almost none existent in the natural world, so the robot wouldn't get confused when trying to search for the pump. Simply having the robot initialize with a calibration of the banners cyan colour (it could change slightly depending on if the weather is overcast or sunny etc.) at the start would allow it to recognize the pumps location. The issue of obstacles being in the way does add another problem as well as being too far away and the camera not being able to properly recognize the banner.

Because of these potential complications I used the simpler method of bug algorithms and GPS coordinates to return to the pump.

3. Implementation

3.1. Topic Layout

An RQT graph visualizes the application within ROS, in this case the Cajun Command software that has been built. It demonstrates how the topics and nodes are interacting. Figure 5 below illustrates the makeup of the software. The script node `/move_robot_node` subscribes to `/odom`, `/gps/fix`, and `/depth/image_raw`. It publishes out to `/cmd_vel`.

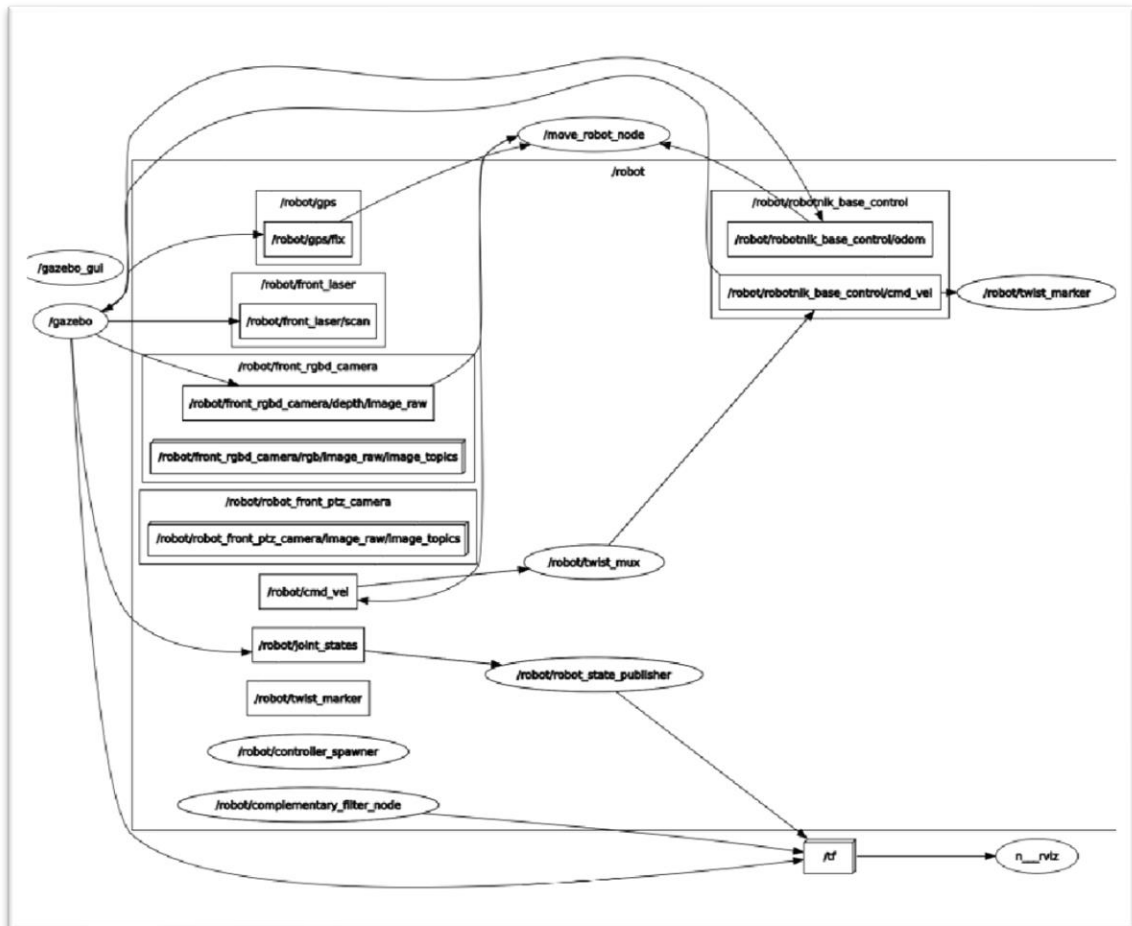


Figure 5: rqt graph showing how topics and nodes interact within the simulation

3.2. Simulation Implementation

3.2.1. Generating Environments for Simulation

Gazebo is capable of making complex physical environments which is invaluable for any simulated testing. From early research it was determined that creating a heightmap based environment would be an appropriate way of modelling the water based environments required for testing. It is often desirable to have a none flat surface over which robots may operate. For this project, the opposite is required but can be achieved via the same methods.

A heightmap is an extrapolation of a 2D greyscale image to create a 3D surface. I started by creating the greyscale image using GIMP[20]. I began by creating a new image and setting an appropriate width and height value, in this case 129x129. Gazebo requires that these values be equal and 2^{n+1} pixels in size.

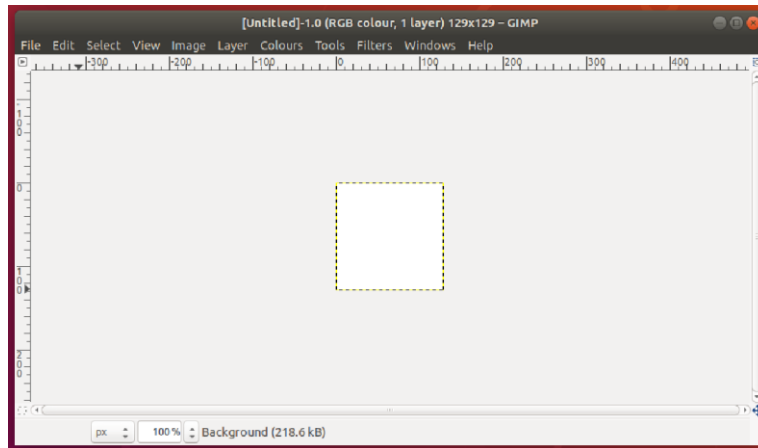


Figure 6: Simple start to a heightmap for Gazebo, currently a flat high surface

Heightmaps work by interpreting each pixel as a height value, where white is high and black is low. So in figure 6 the all-white image produces a high flat plane. Similarly, if it was all black it would be a low flat plane. Since I needed a crater like map to represent a pond for the robot to be tested in I opted to start high and move down lower to create a vertical transition from the water's surface to the banks of the ponds perimeter. Using the airbrush within GIMP allows for smooth transitions from high to low surfaces. Once happy with the created greyscale image it was saved it as a PNG.

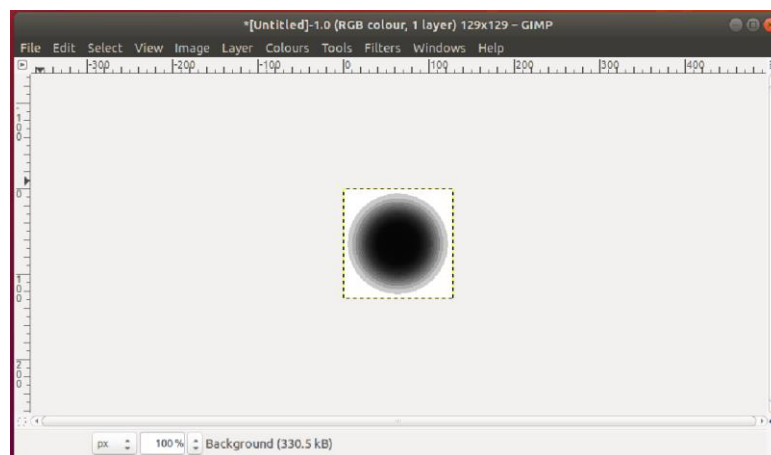


Figure 7: An example of using airbrush in gimp to make a simple greyscale image

Once the greyscale image had been created the next step was to tell Gazebo how to use it. First the model file within Gazebo was located and a copy of an existing model of the ground plane was created as a start point. Models within Gazebo are made up of a *model.config* and *model.sdf* file that explain to Gazebo how the model will interact with other models and what it should look like. The *model.sdf* file septically instructs Gazebo how to create the 3D model. Changing this *model.sdf* file into a heightmap was done by simply copying a *heightmap.world* file from the gazebo directory and overwriting the *model.sdf* file. To convert it from a world to a model, simply removing the world tags and the irrelevant world information provides just the model of the heightmap. Next changes were made to the heightmap *uri* to the greyscale PNG location that was previously created in GIMP. The next change was to alter the size tags to create a suitable sized model. These changes were made to both the *collision* and *visual* tags. Once saved the model was ready to be used in Gazebo.

The process was largely very straight forward and easy to implement once a good tutorial was found.

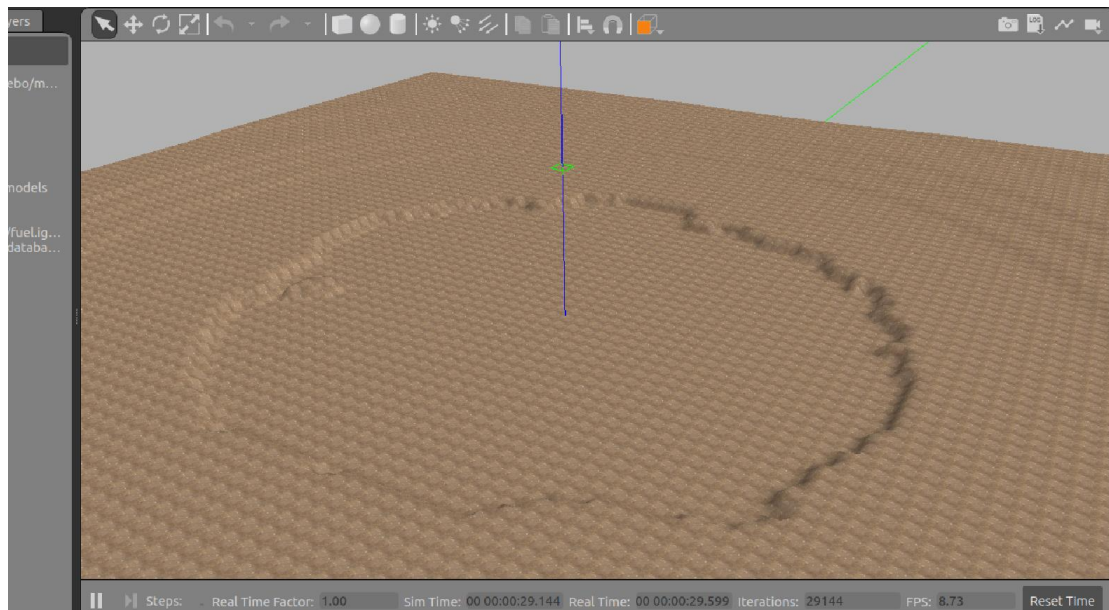


Figure 8: The completed heightmap to represent a simple pond environment

3.2.2. Summit XL and Launch File

The Summit XL rover was taken from the opensource directory provided by ROS on their website [17]. It was effective at completing all the necessary components of the robot and was a good stand in for the Cajun Commander. To place the robot within the generated environment changes were made to one of the launch files that came with the Summit XL packages, removing the models within it (except the robot), inserting the previously created pond model, and moving the robot in position. Once happy, the world was saved. At this point the launch file was copied and changed the copy to launch with the new pond world. A launch file that would be suitable for testing was now ready for use. Further iterations of this pond were made with obstacles within the pond representing potential rocks or vegetation.

3.3. Python Script Implementation

3.3.1. Logging GPS Implementation

Utilizing a simple call-back function, subscribing to the `/robot/gps/fix` topic was a simple process allowing for longitude and latitude data to be fed to other parts of the system. When the robot starts it runs a simple function to log the initial position, thereby marking the position of the pump and the point at which the robot will have to navigate back to after a certain period of time has elapsed. Consideration for mapping the perimeter of the environment and noting the GPS boundaries was made, but ultimately not implemented. The simple method of moving away sufficed for allowing the robot move around the environment. If after further testing it became a problem, this kind of implementation could be useful.

3.3.2. Movement Implementation

Movement of the robot was controlled by publishing messages to the `/robot/cmd_vel` topic. A class was constructed that included all the possible movements that the robot might need to take for the project. This included:

- `moveForward()` – Simply sending a message with positive linear x value and a neutral angular z value allowed for forward movement.

- *moveBackward()* – Simply sending a message with negative linear x value and a neutral angular z value allowed for backwards movement.
- *turn()* – This controlled the robots ability to move away from objects. As such, it would select a random speed between 4.71 and 7.85. These values represent radians per second. Since the rate the robot is ran on is 2 these values should allow the robot to rotate between 135 and 225 degrees. In combination with a random choice between left and right the robot now has a randomized angle and direction for navigating the environment after it encounters an obstacle. A linear speed was kept as the real world boat would maintain some forward momentum when turning.
- *stop()* – This allowed for the robot to set all movements to 0.

3.3.3. Moving to the Centre Implementation

The function *centerPoint()* describes the simple process of moving to the centre of the environment. The centre point is taken from a user input which is then used to calculate a time for the robot to move for in a straight line. This is calculated via the simple equation:

$$time = \frac{distance}{velocity}$$

Knowing the distance to the centre from the user and the velocity of the robot (predetermined by the program) the *centerPoint()* function can be ran until the time is reached. This simple function allows for accurate distance travelled to the centre of the environment.

3.3.4. Obstacle Avoidance Implementation

In order to avoid obstacles a call-back function and subscribing to */robot/front_rgb_camera/depth/image_raw* allowed for the robot to calculate the depth of points in front of it. Initial iterations of this call-back function simply stated the depth at a single point. However this would sometimes lead to the robot colliding with objects just left or just right of this point. To avoid this problem a for loop between a range of horizontal values was created to quickly scan a general area. This helped prevent any collisions with the environment and effectively allowed for obstacle detection. This depth value was then used in a simple function *avoidDepth()* that would either command the robot to move forward if depth was greater than 80cm and turn away from the object if the robot was closer then 80cm.

The call-back function in combination with *avoidDepth()* prevented collisions with obstacles and kept the robot within the bounds of the pond.

3.3.5. Spiralling Out Implementation

Within the movement class there is an additional function controlling the robots movements while spiralling. *spiral()* sets the forward speed at the robots standard 0.2 and the turn speed at 1.0. Using a while loop, the forward speed is then incremented every 0.5 seconds. This allows for a continually larger spiral to be formed by the robot, covering as much area as possible from the centre point. While implementing this function Gazebo exhibited strange behaviour. As the speed increased the front of the robot would begin to rise, defying the gravity physics within the simulation. This would cause issues with object detection since the camera would no longer be able to see the surface of the water. Even more bizarrely, when

the robot would then inevitably move over the boundaries of the pond it would sometimes begin to fly over the surface. After large amounts of testing it was determined that Gazebo was having issues with the high speeds in combination with a high message rate, and the physics was not properly generating an accurate simulation. Therefore linear speed across the robot was initially capped at 0.2 to avoid this issue. Upon use of a different launch file within the summit XL package, the front end stopped moving up and behaved correctly. The gravity however remained a problem and would sometimes have the robot flying.

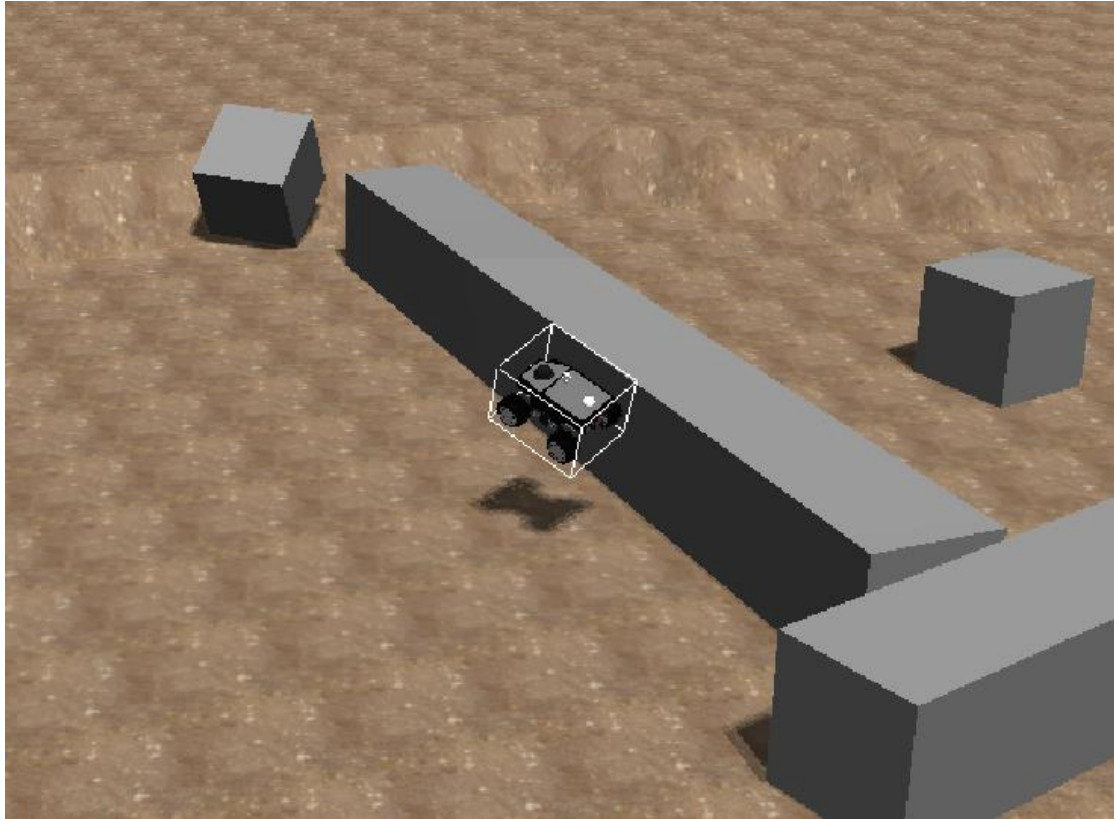


Figure 9: Bug in Gazebo causing robot to lift from the front and eventually fly. I doubt the Cajun Commander would fly in live tests.

Given more time, the function would be altered to maintain the linear speed of 0.2 but periodically stop turning as a way to increase the size of the spiral. This would allow for more effective testing by avoiding the strange Gazebo behaviour. Furthermore in combination with GPS complete circles could be navigated and followed by the robot increasing the radius of the circle just formed, thereby potentially covering more area.

3.3.6. Returning to Pump Implementation

Currently there is no implementation of the robot returning to the pump. Due to lack of time the method for returning the robot to the pump was not implemented.

3.4. Testing Implementation

To test how well the robot was covering the environment a separate script was constructed *testingV3.py*. This script begins by asking for the radius of the pond the robot is tested in. If the big pond environment was being tested the radius of 20 would be applicable. At this point it would construct a 40x40 2D array populated by 0s. These 0's represent a 1m² area that has

not been entered by the robot. The script takes data from `/robot/robotnik_base_control/odom` and uses it to change values within the 2D array to 1. This signifies that a 1m^2 area has been entered by the robot. To account for the area of the pond that does not need to be tracked (the area outside the water) we have to disregard a certain number of the 0s within the array. Summing the total amount of 0's in the initial array we get 1600, next subtract the difference between the initial amount of 0's (1600) and the area of the navigable pond ($A = \pi r^2$) in this case 1256:

$$\text{Total 0's to consider} = \text{current amount of 0's} - (\text{initial amount of 0's} - \pi r^2)$$

From this equation you can estimate the total amount of 0's that should be considered when calculating what percentage of the environment has been covered.

After this a simple division can convert the given value into a percentage of the environment that hasn't yet been covered. As the robot navigates around its environment it will update these values and give a good estimate on what percentage of the environment has been explored and how much has not.

While constructing this test script, a bug was discovered in which Gazebo was sending two separate pieces of odometry data, almost as if there were two separate robots moving around the environment. This "ghost" doubling up data meant half the data being received from the odometry topic was false and did not describe where the robot was in 2D space. This would mean that:

1. The amount of squares actually being covered was less than calculated since if the robot was at 0,0 it might also log that it was at 20,10 as well. In order to account for this the calculation for how many 1s there were in the array is halved.
2. Secondly, since the odometry data was giving values in areas where the robot hadn't been, they were sometimes out of bounds. For this reason when the array is constructed it makes an array larger than is required. This is then taken into account later when calculating percentages for 0s and 1s in the array.

Fortunately, toward the end of the project it was discovered that the error was occurring in the launch file. While it was not clear why it was happening, there was time to use a different launch file where the "ghost" didn't exist and add the models for each environment. When this bug was fixed these sections of the testing script were corrected. The specific areas have been marked with comments to ensure they can be quickly changed if future problems do occur.

3.5. Overall Implementation Success

When compared to the initial set of goals for the project, all goals have been achieved except returning to the initial starting position.

The environment created for the robot to navigate and the robot itself are useful for testing.

The robot moves cleanly through its environment and is capable of recognizing obstacles and avoiding them. It correctly logs its position with the GPS, and with some further work could use this, in conjunction with the timer running in the program, to return at to the start when required.

The testing script does a good job of logging how much of the area has been covered in the large pond environment created. This script is largely tailored for a round circular environment. Obviously in more complicated environments a new sort of testing would need to be done in order to estimate its effectiveness.

The bugs with Gazebo, the faulty odometry data and the faulty physics when moving quickly, were solved and the ways around them have been left as comments in the scripts encase they reoccur. With more time, research into why Gazebo is not simulating these parts appropriately would be explore.

4. Testing

4.1. Overall Approach to Testing

Testing in any software related project is essential for showing what works well and what fails to meet specifications. Tests within this section illustrate the effectiveness of the software within a range of environments, showing both its strengths and weaknesses. For testing, the testing script created in section 3.4 was utilized. This script works best in simple shaped environments and modifications would be required in more complicated shaped environments.

Three different testing environments were constructed within Gazebo, one subsequently larger than the last. Additionally these environments were tested both empty and with obstacles typical in environments the robot would be navigating, such as vegetation or rocks. Experiments were ran for 2, 5, 10 minutes respectively, this was to take in account the battery life of the Cajun Commander and the size of the environments. Modifications to the robot could obviously be made to increase the battery life, but for testing the base robot was singularly considered.

4.2. Simulated Testing

4.2.1. Small Area

	Percentage Covered	Time Taken(seconds)	Notes
Test 1	100.0	55	-
Test 2	100.0	96	-
Test 3	100.0	111	-
Test 4	100.0	43	-
Test 5	100.0	97	-
Test 6	100.0	90	-
Test 7	100.0	52	-
Test 8	100.0	101	-
Test 9	100.0	81	-
Test 10	100.0	70	-

The small area was a simple square environment. The purposes of this was to create the most basic shape possible for simple testing of the robot. Additionally it shows the effectiveness of the robot in urban artificially created water features that are typically angular and simple in shape. The environment was a 6x6 meter square. After ten tests, results show that all tests were completed within their two minute assigned work time. The average time taken to complete the work was 96.6 seconds.

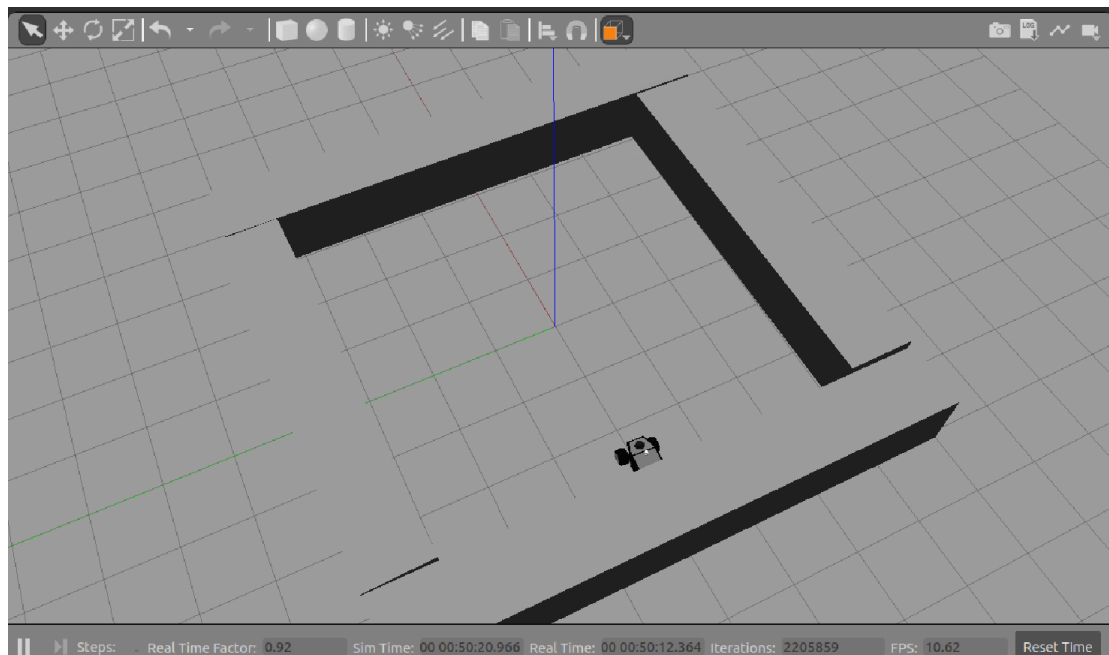


Figure 10: Small area, no obstacles

4.2.2. Small Area with Obstacles

	Percentage Covered (%)	Time Taken(seconds)	Notes
Test 1	93.75	120 (full duration)	-
Test 2	34.375	120 (full duration)	Collided -> camera clipping -> stuck
Test 3	50.0	120 (full duration)	Collided -> camera clipping -> stuck
Test 4	87.5	120 (full duration)	Collided -> wheels flip -> stuck
Test 5	43.75	120 (full duration)	Collided -> wheels flip -> stuck
Test 6	75.0	120 (full duration)	-
Test 7	93.75	120 (full duration)	-
Test 8	62.5	120 (full duration)	Collided -> camera clipping -> stuck
Test 9	65.625	120 (full duration)	-
Test 10	53.125	120 (full duration)	Collided -> camera clipping -> stuck

The small area with obstacles added revealed several problems within the simulation.

The first of these was the camera clipping through objects when too close. Instead of seeing a depth reading of NAN (when below 0.4m the depth returned is NAN) the camera will clip through the object and report a reading as if the obstacles was not there. Because of this the robot will simply maintain a forward direction not knowing there is an obstacle in the way.

The second issue was that sometimes the robot would slightly collide on objects with its wheel, and due to the robot chosen, its wheels would grip on the obstacles (even completely vertical ones) and proceed to move upward. Once the camera correctly saw an obstacle as it turned on the obstacle, it would attempt to move away, now at an angle. The result would have the robot moving around either with its front end pointing upwards or downwards. This would in turn give false depth readings causing the robot to move forward forever (looking at sky so can't see obstacles) or it would spin forever (turning since looking at ground – ground is seen as an obstacles as it is now too close).

After ten tests the average percentage covered was 66%. Of the four tests that did not encounter problems, the average percentage cover was 82%. Assuming the problems with the simulated camera clipping and gravity could be fixed, the percentage complete would no doubt be higher.

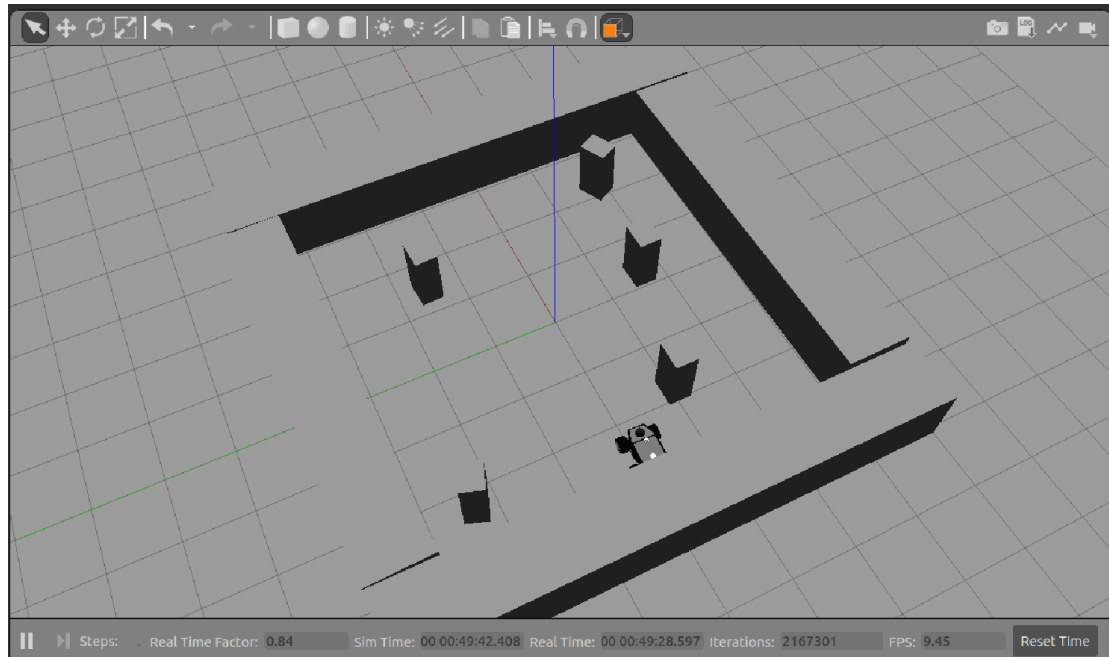


Figure 11: Small area with obstacles

4.2.3. Medium Area

	Percentage Covered	Time Taken(seconds)	Notes
Test 1	100.0	124	-
Test 2	100.0	124	-
Test 3	100.0	124	-
Test 4	100.0	124	-
Test 5	100.0	124	-
Test 6	100.0	124	-
Test 7	100.0	124	-
Test 8	100.0	124	-
Test 9	100.0	124	-
Test 10	100.0	124	-

The medium area was a simple circular environment. The environment has a 9 meter radius, so in turn an area of $254m^2$. After ten tests, results show that all tests were completed within their five minute assigned work time. The average time taken to complete the work was 124 seconds. Since the robot spiralled out from the centre it would consistently make the same path around the environment, hence why every test was identical. In real life, variations from the environment such as wind or the water movements would likely cause far less consistent results. None the less it shows the robot is very well suited for circular like environments due to the effectiveness of the spiral. The medium area did run very slowly, with or without obstacles. With more time it would be helpful to find out why to allow for faster future testing.

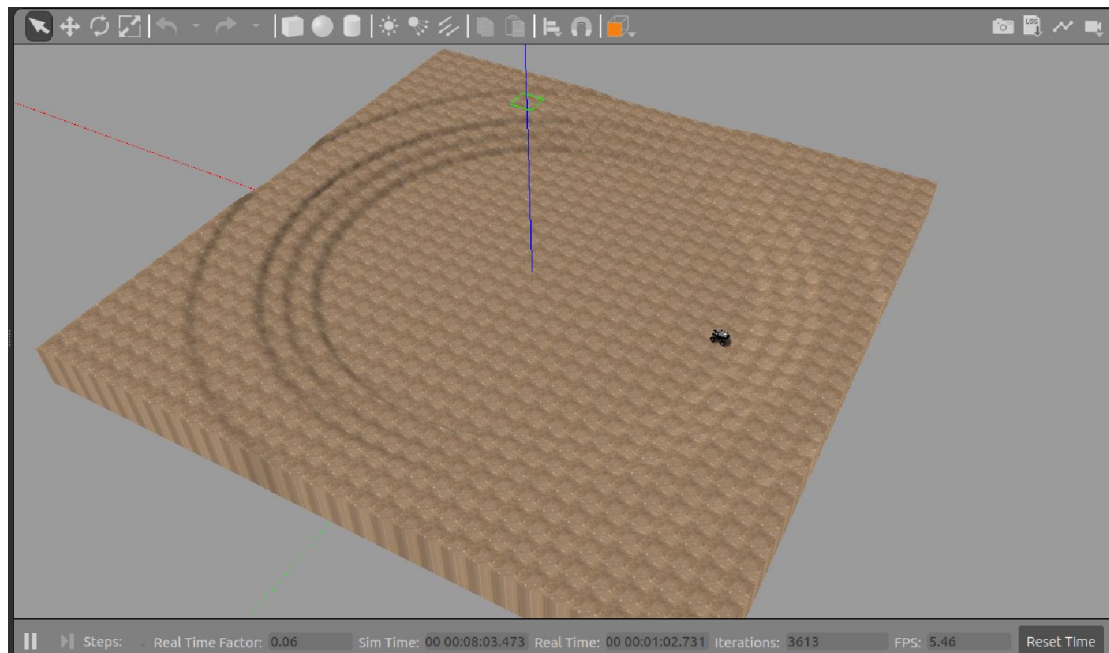


Figure 12: Medium area, no obstacles

4.2.4. Medium Area with Obstacles

	Percentage Covered (%)	Time Taken(seconds)	Notes
Test 1	47.25	300(full duration)	Left environment due to angle with shore
Test 2	83.0	300(full duration)	-
Test 3	73.25	300(full duration)	Collided -> camera clipping -> stuck
Test 4	55.10	300(full duration)	Collided -> camera clipping -> stuck
Test 5	59.85	300(full duration)	Left environment due to angle with shore
Test 6	78.75	300(full duration)	-
Test 7	70.85	300(full duration)	-
Test 8	42.125	300(full duration)	Left environment due to angle with shore
Test 9	45.66	300(full duration)	-
Test 10	62.20	300(full duration)	-

Like with the small area testing, the problems with the gravity and camera were also present. Additionally a new problem occurred, one that would also be a problem within the real world. When the robot approaches the shore at an almost parallel angle, it will simply continue forward, not recognizing the shoreline. This would allow for the robot to leave the parameters of the environment. So while the robot can recognize steep shore lines, ones that are more gradual like in the medium area, cause the robot real challenges in recognizing the boundaries. One way to correct for this would be having the robot scan a larger range of horizontal points on the depth camera images. When the left, centre, and right sides of the image have consistent increasing or decreasing values, it could recognize it is moving along a slope. From this it could infer it needs to move back into the environment.

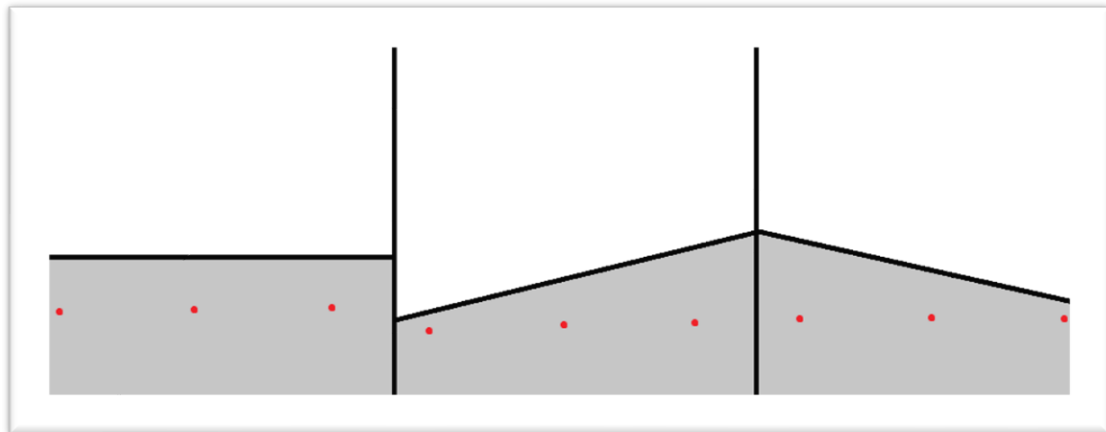


Figure 13: Left image is when robot sees a flat area ahead. Centre image shows robot moving along a slope ascending left to right. Right image shows a robot moving along a slope descending left to right. Dots show hypothesized way to recognize it is moving along the slope by getting distance values for each point and comparing them.

A new obstacle not present in the small area was added, This obstacle was intended to represent vegetation within a water environment, such as reeds. These tall and thin obstacles were often caught thanks to the scanning of several points on the camera image. However due to their thin qualities they did present a problem for being missed and causing collisions, especially when they were not concentrated together in an area. Since they typically are concentrated in natural environments, this is how they were placed the simulations.

After ten tests the average percentage covered was 61.8%. Of the five tests that did not encounter problems, the average percentage cover was 68%. Assuming the problems with the simulated camera clipping and gravity could be fixed, the percentage complete would no doubt be higher. Furthermore correction for the slope bug would again help increase percentage cover.

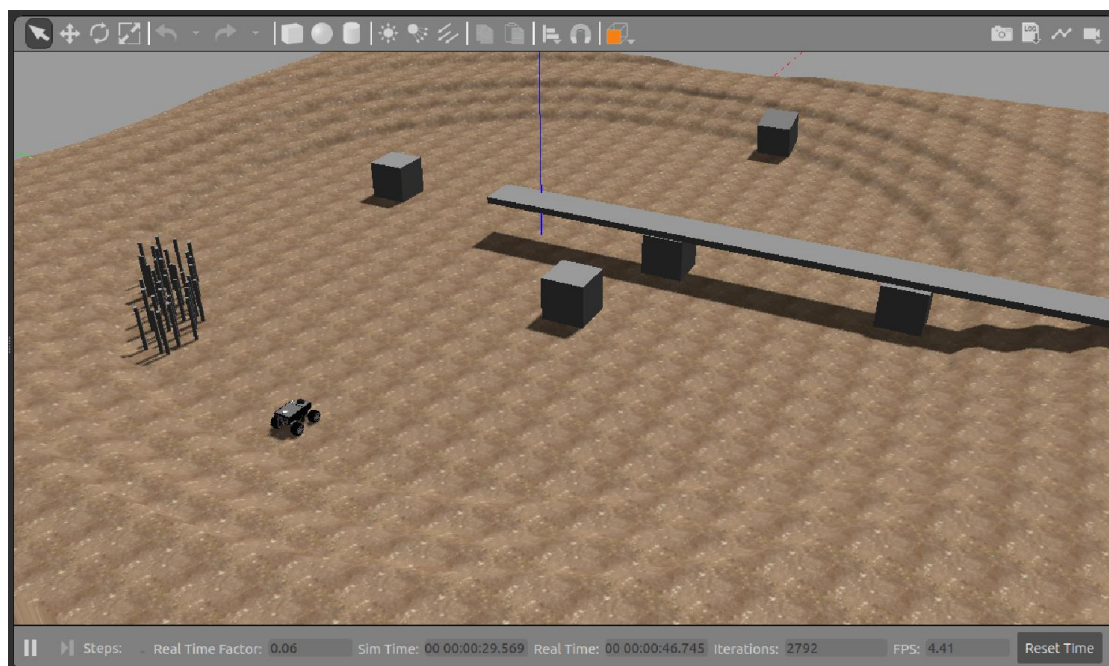


Figure 14: Medium area with obstacles

4.2.5. Large Area

	Percentage Covered (%)	Time Taken(seconds)	Notes
Test 1	72.77	600 (full duration)	-
Test 2	76.43	600 (full duration)	-
Test 3	71.65	600 (full duration)	-
Test 4	69.56	600 (full duration)	-
Test 5	73.16	600 (full duration)	-
Test 6	64.96	600 (full duration)	-
Test 7	72.05	600 (full duration)	-
Test 8	77.62	600 (full duration)	-
Test 9	71.97	600 (full duration)	-
Test 10	73.64	600 (full duration)	-

The large area was a circular like environment with some jagged edges. The purposes of this was to create a realistic environment that didn't have perfect shape to it. The environment has a 20 meter radius, so in turn an area of $1256m^2$. The robot made clean movements to the middle and spiralled appropriately. Unlike the medium sized area, the large area gives the robot more time to spiral, and therefore more time to increase speed. This causes problem with detecting the shore or any obstacles in time to react. For this reason, the robots spiralling function would need to be changed for large sized environments. When testing, I would pause the simulation just after detection and place it back on the ground to allow it to continue its navigation around the environment post spiralling. Unlike the medium area with non-steep shores, the large areas steep shore provided an environment that the robot performed well in since it was consistent in detecting the depth change near the waters perimeter. After running ten tests, no test could complete the area completely. The average percentage covered over the ten tests was 72.38%. Like the medium area, when the robot spiralled out from the centre it would consistently make the same path around the environment until encountering the shore, hence why every test was similar in result.

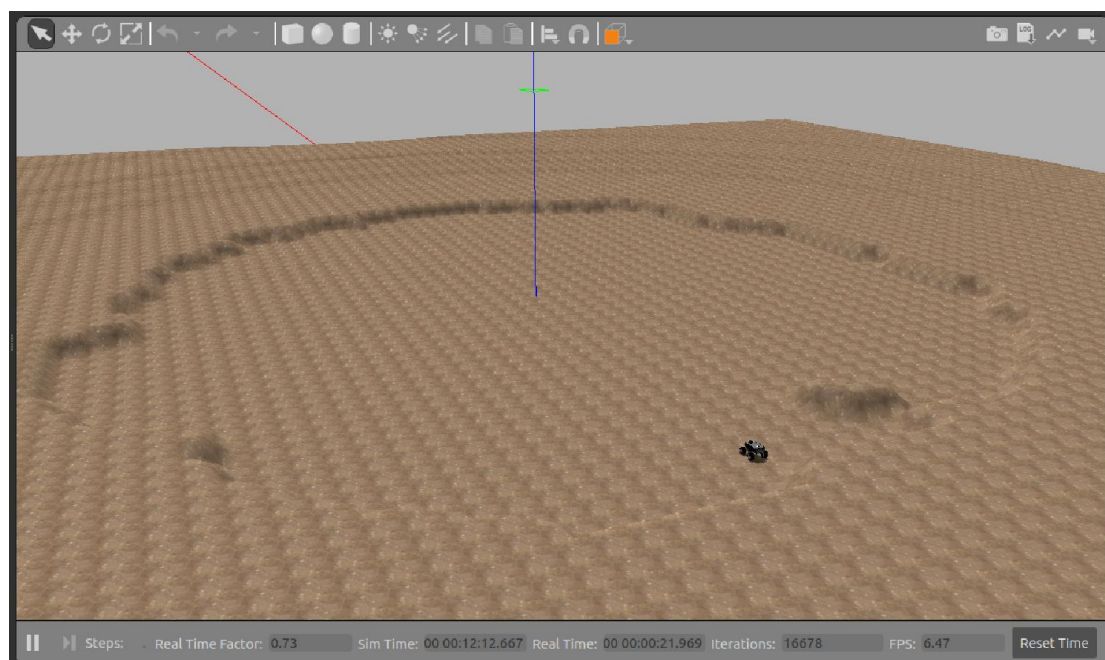


Figure 15: Large area, no obstacles

4.2.6. Large Area with Obstacles

	Percentage Covered (%)	Time Taken(seconds)	Notes
Test 1	46.28	600 (full duration)	Largely stuck in southern half
Test 2	35.47	600 (full duration)	Largely stuck in southern half
Test 3	42.31	600 (full duration)	Largely stuck in southern half
Test 4	40.20	600 (full duration)	Largely stuck in southern half
Test 5	32.34	600 (full duration)	Largely stuck in northern half
Test 6	33.78	600 (full duration)	Largely stuck in northern half
Test 7	45.69	600 (full duration)	Largely stuck in southern half
Test 8	38.00	600 (full duration)	Largely stuck in southern half
Test 9	32.85	600 (full duration)	Largely stuck in northern half
Test 10	33.27	600 (full duration)	Largely stuck in southern half

The large area with obstacles was the large area populated with a bridge going through the centre of the area, rocks, and some vegetation. The purpose of the bridge was to limit the ability to spiral across the area and get a better gauge of how effective randomly turning away from obstacles was. Over ten tests the average percentage cover was 38.02%. Not being able to spiral over a large section of the environment causes much lower rates of coverage when compared to the large area with no obstacles. Unlike the other environments tested in, the robot would sometimes become stuck in certain sections, in this case above or below the bridge. This is likely due to the large bridge supports covering a large area, and therefore having a higher likelihood to make the robot turn back when compared to the smaller obstacles used in previous environments.

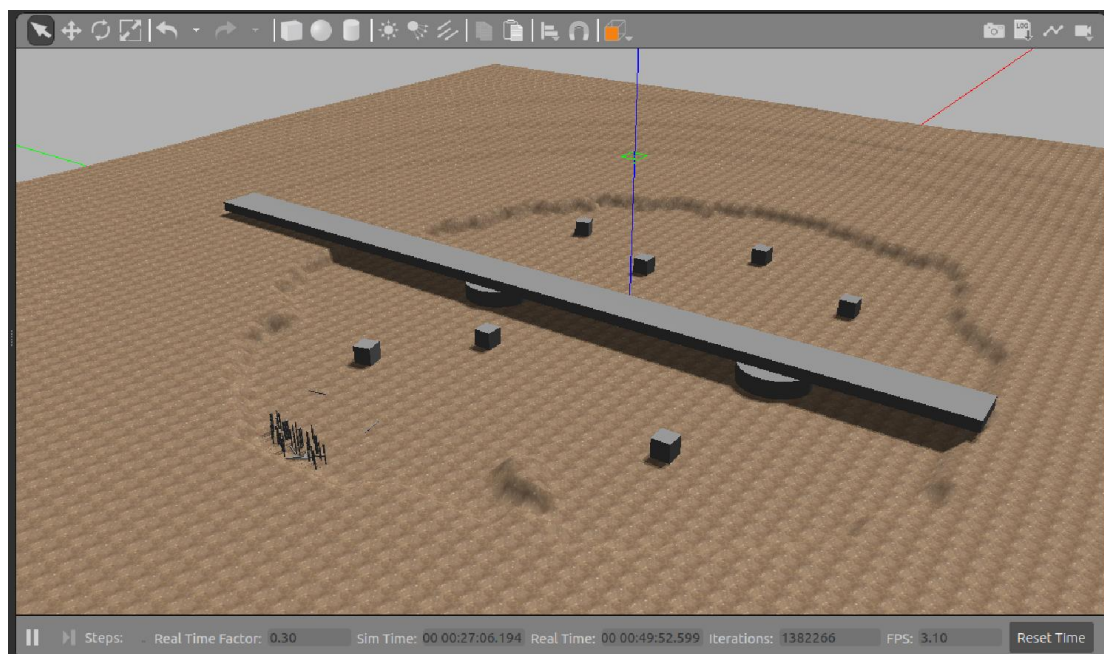


Figure 16: Large area with obstacles

5. Critical Evaluation

5.1. Requirements

The Duckweed Robot Harvester was a project that aimed to clean a still body of water of any duckweed. Overall, the requirements for this project were met, with the exception of returning to the pump. While an important aspect of the project, the functionality and success of the robot in small to medium areas shows there is a strong foundation to build upon. The robot does effectively move to the centre, it will spiral out very smoothly and allow for large initial coverage, and it will correctly recognize any obstacles and move away appropriately.

5.2. Design and Implementation Reflection

5.2.1. Software Design

After completion of testing, it is clear that the robot is most effective in small to medium sized areas. Retrospectively this makes sense considering that the design process was modelled of early versions of the Roomba, something that work inside small areas. If this project was uniquely designed for larger areas, making use of mapping would help provide consistent and effective coverage results. While this might make a final commercial product more cost prohibitive, no one wants to buy something that doesn't work. Otherwise, the robot does show great potential for an automated cleaner in small to medium sized environments.

Another oversight involved shallow gradual shorelines. When initial testing with the depth camera was done a more steep shore was used like in the large Gazebo testing environment. Because of this it struggled to correctly identify less steep gradation. While a solution has been suggested in the testing section, it should have been identified much earlier on given the large amount of water bodies that exhibit this feature. One solution to this, and one already written about in 2.4.1, would be to have GPS boundaries the robot must adhere to and stay within. This would allow for the current obstacle avoidance to remain as it is.

5.2.2. Software Implementation

Python scripts created for use in ROS were all cleanly and coherently written. Comments are clear and thorough, allowing for future potential developers to quickly continue where this project ends.

The ROS node/topic system was used effectively in accessing data required for the robots use. Given more time potential implementation with the navigation stack would be looked into for more efficient navigation of the environments.

5.2.3. Environment Design and Implementation

The different environments, from the small square environment, to the very large complex environment, provided insight into the robots strengths and weaknesses, including some design flaws such as the difficulty in detecting shallows shore lines. Using the heightmap design in Gazebo allowed quick creation of realistic terrain that the robot would likely encounter along the boundaries of a water body. Without this the robot would be limited in very simple simulations when compared to its real world counterpart.

Overall the different environments created within Gazebo were effective at testing how well the robot achieved its task and what scenarios it struggles with.

5.3. Testing Reflection

The testing was done in three different environments, with various features and sizes, ran for different periods of time. Each environment was tested on with and without obstacles. Like previously stated, the different environments in conjunction with testing effectively allowed for discovery of what was effective and what wasn't.

The testing script designed for calculating the total amount of the environment the robot was simple yet effective. In some cases where the robot would leave the environment the scripts arrays would go out of bounds. Simply making the array bigger to compensate for this would ensure it didn't run into exceptions. Changing the percentage calculation with these additional array positions wouldn't be difficult.

Testing could have started earlier in the project to allow for quicker identifications of bugs within Gazebo, most specifically the doubling of odometry data and the gravity physics not working as intended. Having the robot behave strangely due to simulation bugs was stressful to say the least. Fortunately these bugs were circumvented by using a different launch file within the summit packages. Though the robot would still fly sometimes, this was a rare occurrence, and could be fixed by pausing the simulation and placing the robot back on the ground in the correct orientation. From quick google searches, this was a problem in earlier iterations of Gazebo and so could have potentially been fixed by simply updating Gazebo to the latest version but was ultimately not necessary since use of a different launch file fixed the majority of problems.

5.4. Future Work

Future work should centre around adding the ability to return to the pump and adding a permitter to the robots work area via GPS. These additional features would create a finished product that should be able to clean a water body of duckweed and return it for harvesting. Additionally a future look into the cost effectiveness of mapping would be an important consideration. Comparing this intelligent method of navigation with the currently random method would be good at determining which method is more effective and ultimately more viable in a commercial setting, something that should be strongly considered when designing robotics for replacing menial labour.

5.5. Summary

The Duckweed Robot Harvester in its current state effectively covers small to medium sized environments. While not achieving 100% coverage in the small or medium settings with obstacles, it doesn't need to. Its goal should be to remove enough so as to prevent it from overtaking an entire environment. This could be done periodically to help create a balance within an environment instead of simply removing an entire food source from an ecosystem.

Version control with GitHub was only implemented toward the end and should have been used more regularly at the start to avoid any potential issues. Virtual machine cloning was done at the end of every session to ensure any potential faults with the most current version

of the virtual machine wouldn't cause weeks of work to be lost. When combined with GitHub, many backups were available of the project ensuring worst case scenarios wouldn't bring the project to a halt.

Overall the project effectively discusses the environmental issues caused by rampant duckweed and thoroughly delves into the effectiveness of an automated robot duckweed cleaner. From this project it is clear to see there are real possibilities for a water based Roomba coming soon to a store near you.

6. References

- [1] Rinkesh. (2016, December 25). *15 Current Environmental Problems That Our World is Facing - Conserve Energy Future*. Conserve Energy Future. <https://www.conserve-energy-future.com/15-current-environmental-problems.php>
- [2] Frequently Asked Questions Problems with Algae and Duckweed. (2013). In *freshwaterhabitats.org.uk*. <https://freshwaterhabitats.org.uk/wp-content/uploads/2013/09/algae-and-duckweed.pdf>
- [3] Hillman, W. S., & Culley, D. D. (1978). The Uses of Duckweed: The rapid growth, nutritional value, and high biomass productivity of these floating plants suggest their use in water treatment, as feed crops, and in energy-efficient farming. *American Scientist*, 66(4), 442–451. <http://www.jstor.org/stable/27848752>
- [4] Pastierova, Alica & Sirotiak, Maroš & Fiala, Jozef. (2015). Comprehensive Study Of Duckweed Cultivation And Growth Conditions Under Controlled Eutrophication. *Research Papers Faculty of Materials Science and Technology Slovak University of Technology*. https://www.researchgate.net/publication/282895717_Comprehensive_Study_Of_Duckweed_Cultivation_And_Growth_Conditions_Under_Controlled_Eutrophication
- [5] *Oase Skimmer 250*. (n.d.). *Www.swelluk.com*. Retrieved April 28, 2022, from <https://www.swelluk.com/oase-skimmer-250>
- [6] *Roomba® Robot Vacuum Cleaners | iRobot*. (n.d.). *Www.irobot.co.uk*. <https://www.irobot.co.uk/en-gb/roomba.html>
- [7] *Aquacraft Cajun Commander Brushless RTR Airboat | Howes Models*. (2017, May 13). *Howes Model Shop Oxford*. <https://howesmodels.co.uk/product/aquacraft-cajun-commander-brushless-rtr-airboat/>
- [8] Majdalani S, Chazarin J-P, Moussa R. A New Water Level Measurement Method Combining Infrared Sensors and Floats for Applications on Laboratory Scale Channel under Unsteady Flow Regime. *Sensors*. 2019; 19(7):1511. <https://doi.org/10.3390/s19071511>
- [9] McGuire, Kimberly & Croon, Guido & Tuyls, Karl. (2019). A comparative study of bug algorithms for robot navigation. *Robotics and Autonomous Systems*. 121. 103261. 10.1016/j.robot.2019.103261. https://www.researchgate.net/publication/335333655_A_comparative_study_of_bug_algorithms_for_robot_navigation
- [10] Trapani, K. (2018, May 22). *What is Agile/Scrum*. CPrime; cPrime. <https://www.cprime.com/resources/what-is-agile-what-is-scrum/>
- [11] GitHub. (2018). *GitHub*. GitHub. <https://github.com/>
- [12] ORACLE. (2019). *Oracle VM VirtualBox*. Virtualbox.org. <https://www.virtualbox.org/>

- [13] Clearpathrobotics.com. (2015). *Intro to ROS — ROS Tutorials 0.5.2 documentation*. [online] Available at: <https://www.clearpathrobotics.com/assets/guides/melodic/ros/Intro%20to%20the%20Robot%20Operating%20System.html> [Accessed 29 Apr. 2022].
- [14] gazebosim.org. (n.d.). *Gazebo*. [online] Available at: <https://gazebosim.org/>
- [15] wiki.ros.org. (n.d.). *hector_gazebo_plugins - ROS Wiki*. [online] Available at: http://wiki.ros.org/hector_gazebo_plugins [Accessed 29 Apr. 2022].
- [16] robots.ros.org. (2019). *robots.ros.org*. [online] Available at: <https://robots.ros.org/> [Accessed 29 Apr. 2019].
- [17] robots.ros.org. (n.d.). *Summit-XL*. [online] Available at: <https://robots.ros.org/summit-xl/> [Accessed 29 Apr. 2022].
- [18] www.irobot.ie. (n.d.). *Terra: Robot Lawn Mower / iRobot*. [online] Available at: <https://www.irobot.ie/Terra> [Accessed 29 Apr. 2022].
- [19] Ros.org. (2013). *navigation - ROS Wiki*. [online] Available at: <http://wiki.ros.org/navigation>
- [20] The GIMP Team (2019). *GIMP*. [online] GIMP. Available at: <https://www.gimp.org/>.
- [21] wiki.ros.org. (n.d.). *rospy - ROS Wiki*. [online] Available at: <http://wiki.ros.org/rospy/>
- [22] Opencv.org. (2019). *OpenCV library*. [online] Available at: <https://opencv.org/>
- [23] GitHub. (2022). *summit_xl_sim*. [online] Available at: https://github.com/RobotnikAutomation/summit_xl_sim [Accessed 5 May 2022].

7. Appendices

A. Third-Party Libraries

A number of libraries were used when implementing software. These include:

Rospy – Rospy was used to allow for ROS and python to work seamlessly together. From the ROS website, “rospy is a pure Python client library for ROS. The rospy client API enables Python programmers to quickly interface with ROS Topics, Services, and Parameters. The design of rospy favors implementation speed (i.e. developer time) over runtime performance so that algorithms can be quickly prototyped and tested within ROS. It is also ideal for non-critical-path code, such as configuration and initialization code. Many of the ROS tools are written in rospy to take advantage of the type introspection capabilities. Many of the ROS tools, such as rostopic and rosservice, are built on top of rospy”.

The library is open source and it is available from the ROS website [21].

In addition to rospy, other python based libraries used included *math* and *random* to allow for accurate math calculations.

OpenCV – Libraries from OpenCV were essential for utilizing the camera on the robot and correctly estimating distance to obstacles. The library is open source and it is available from the OpenCV website **Error! Reference source not found..**

B. Third-Party Code

Code was reused from a previous semester in building the callback for the depth image. Modifications were made to the callback function provided by Dr Patricia Shaw.

```
def cb_depthImage(image):
    global bridge, x, y, depth
    x = int(greenLocationX)
    y = int(greenLocationY)
    rospy.loginfo('x:%s',x)
    rospy.loginfo('y:%s',y)

    # image msg obtained from callback message for topic
    try:
        cv_image = bridge.imgmsg_to_cv2(image, "32FC1")
        # where x,y is the centre point from the published moment
        depth = cv_image[y][x]
        rospy.loginfo('Depth of point is %s m',depth)
        avoid_depth()
        #if depth < 1.0:
        #    rospy.loginfo('Depth less than 1, too close')
        # For testing/verification:
        cv2.circle(cv_image, (x,y), 10, 255) # draw circle radius 10 at x,y
        cv2.imshow("Image window", cv_image) # display the image
        cv2.waitKey(3)
    except CvBridgeError as e:
        print(e)
```

Figure 17: Original callback used from previous semester

```
def cb_depthImage(image):
    global bridge, y, depth
    y = int(depthHeightInImage)

    # image msg obtained from callback message for topic
    try:
        cv_image = bridge.imgmsg_to_cv2(image, "32FC1")
        # where x,y is the centre point from the published moment
        for i in range(256, 384, 64):
            depth = cv_image[y][i]
            #rospy.loginfo('Depth of point is %s m',depth)
            avoid_depth()
        # For testing/verification:
        cv2.circle(cv_image, (i,y), 10, 255) # draw circle radius 10 at x,y
        cv2.imshow("Image window", cv_image) # display the image
        cv2.waitKey(3)
    except CvBridgeError as e:
        print(e)
```

Figure 18: Modified callback used for project

C. Third-Party Gazebo Files

Use of files from Robotnik Automation GitHub were used for making use of their Summit XL robot within Gazebo [23].