Multi-tier Robot System

Robot Software Engineering

Samuel Westlake

1. Functionality

1.1.System Aims

This robot system was designed for the task of automatic exploration. This prototype system employs a system that builds an occupancy grid of the robot's surroundings given data from ultrasonic sensors, an Inertial Measurement Unit (IMU) and rotary encoders. The robots motion was controlled by the user or automatically and enabled expansion of the occupancy grid. This interpretation of the environment was displayed in a Graphical User Interface (GUI), in which the position and orientation of the robot were also presented. Furthermore, a camera was mounted on the robot, the pitch and yaw of which could be controlled by the user. The resulting video stream was displayed in the GUI.

1.2.Hardware

All hardware devices were mounted on a four wheeled, two-levelled vehicle with the two inbuilt motors enabling differential drive as shown in Figure 1.

Figure 1: robot platform

1.2.1. Processing Units

This system relied on four processing units:

- A Raspberry Pi 3,
- An Arduino Uno microcontroller,
- A Nexus 5 Android smartphone,
- A laptop.

1.2.2. Sensors

Sensors implemented on the robot included:

- 5 x HC-SR04 Ultrasonic Distance Sensors,
- 1 x IMU gyroscope (integrated in the Nexus 5),
- 1 x webcam,
- 2 x Rotary encoders.

1.2.3. Actuators

Actuators implemented on the robot included:

- 2 x HK-15178 Servos,
- 2 x brushed DC motors.

1.3.Third Party Software

For efficient development some third-party software was implemented.

1.3.1. Robot Operating System

Robot Operating System (ROS) is an open-source collection of software frameworks for robot software development. It is principally used for message-passing between processes, package management, hardware abstraction and low-level device control. For these reasons, it was selected handle such tasks. The main ROS package used was the rosserial_python package for handling message-passing to and from the Arduino. ROS libraries this project was dependent on included:

- rospy,
- std_msgs,
- sensor_msgs,
- message_generation,
- message_runtime.

1.3.2. OpenCV

Some processing of the webcam image was required and OpenCV was chosen to handle this. OpenCV is an image processing library that the developer was already familiar with, meaning its use minimised development time. In addition, this library was open-sourced and so, software expenses were minimised.

1.3.3. PyGame

There were tens of GUI frameworks to choose from for Python, such as Tkinter and PyGUI. However, PyGame was selected was as, unlike Tkinter, it can register parallel keyboard events, which was a desirable quality at the time of selection. In addition, PyGame was familiar to the developer which minimised development time.

1.3.4. Nanpy

Nanpy is a python library that contains ArduinoAPI which allows the Arduino to be used as a ‘slave’ I/O board. This was initially used handle sonar and motor devices connected to the Arduino from the Raspberry Pi however, this method was later dropped as the Aruino PulseIn function, used when detecting ultrasonic echoes, was not supported at the time.

1.4.Interaction with the Environment

The principal form of interaction the robot has with its environment is through ultrasonic distance sensors and motors. Ultrasonic sensors allow the robot to detect the distance to surrounding obstacles and indicate these detections in an occupancy grid. In addition, the robot can move within its environment using electric motors, controlled by the user or automatically. In order to expand the occupancy grid as the robot’s position and orientation changed, an IMU gyroscope and rotary

encoders were used to localise the robot. This meant that obstacles with a relative distance to the robot could be added to the occupancy grid in the world view.

1.5.Real-Time Issues

The principal real-time issue experienced concerned the camera feed. Original implementation for publishing the webcam frames used cv_bridge to convert the captured OpenCV image into a ROS image message. However, this method of passing an uncompressed image over WiFi resulted in significant latency that made out-of-visual-range control of the robot difficult. To solve this, the image was compressed using OpenCV's imencode function before being published as a ROS compressed_Image.

A second real-time issue encountered was the presence of wheel sleep when the robot was twisting using its differential drive setup. This meant that determining the robot's change in direction through rotary encoders on wheel axles would be inaccurate. Therefore, encoders were only used to track linear motion of the robot and tracking the robots instantaneous direction was done through an IMU gyroscope in the Nexus 5.

The third real-time issue concerned the IMU gyroscope in the Nexus 5. Data from this sensor was integrated in real-time before being published to allow the robot's orientation to be tracked. Although, under stable conditions, drift in the gyroscope was negligible, when experiencing sudden changes in orientation some degree of drift occurred. Therefore, over time, the indicated direction of the robot would differ from the actual direction. Implementing a solution to this was not prioritised however, this drift could be corrected by a high-pass filter.

Updating the model of the buggy using the laptop and presenting the subsequent information on the GUI in a timely fashion was an ongoing issue. When plotting the occupancy grid in the GUI, one option considered was to draw the entire occupancy grid to the screen before masking as appropriate to clear space for other widgets. However, this would have resulted in drawing many grid squares unnecessarily. Instead, care was taken as to only plot grid squares that would be seen by the user to minimise rendering time.

1.6.User Interaction

The principal form of interaction from the user was through a keyboard. This was how the robot's drive controls and camera pitch and yaw were be controlled. Whilst in 'Manual' mode, W, A, S, D keys allowed the user to make the robot, drive forward, twist left, twist right and reverse, respectively. This mode could be switched to 'Roam' by clicking a button in the GUI. Furthermore, keypad values 8 and 2 controlled the camera's pitch, 4 and 6 controlled the camera's yaw and key 5 caused the pitch and yaw values to reset to zero.

Interaction from the robot took two forms. Firstly, camera feed from the robot was displayed in the GUI which allowed the user to control the robot without direct visual contact. Secondly, the occupancy grid observed by the robot was plotted in the GUI which also indicated the position of the robot within the environment.

2. Design

2.1.Processes

The deployment diagram in Figure 2 illustrates the devices used in the project and their relative connections other hardware devices. In addition, this diagram shows the software that was executed on each device.

Roscore, necessary for message passing between devices, and the laptop_node, which handled the GUI and computer model of the robot was run on the laptop. Separate python scripts to handle the webcam, encoders, and servos were run on the Raspberry Pi along with the serial_node which was required for communication with the Arduino. Software for handling the IMU gyroscope was run on the Nexus and all interactions between devices were handled by ROS.

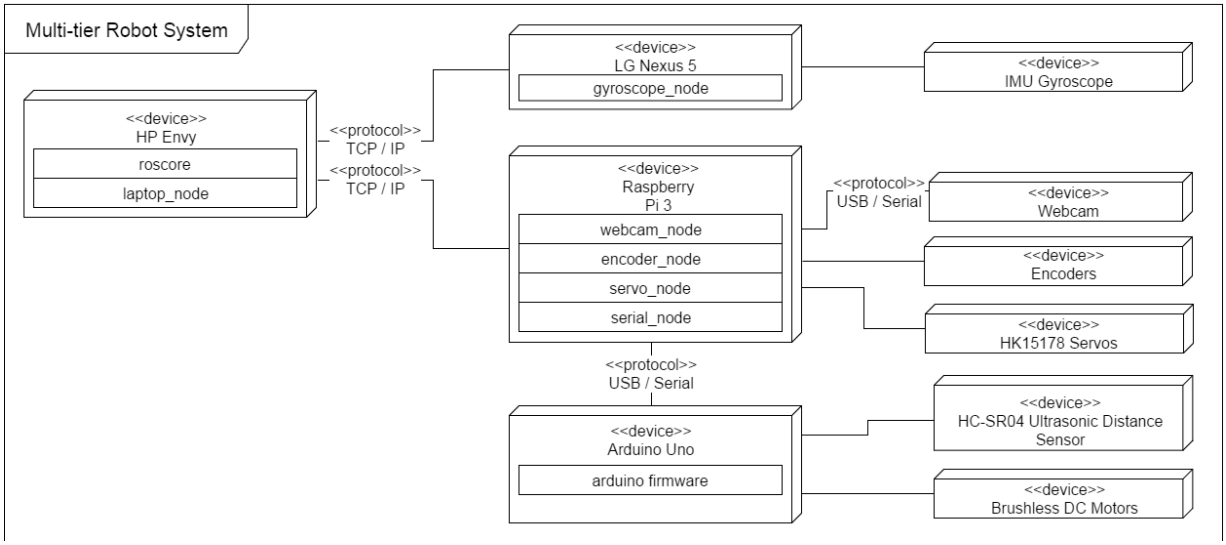

Figure 2: deployment diagram of the multi-tier robot system.

2.2.Process Interactions

An outline of the processes associated with each node, and their subsequent interactions is shown in Figure 3. As illustrated in the diagram, the gyroscope node, serial node and encoder node all contribute to the computer model of the robot, whilst the web cam node supplies a video stream to the GUI. Furthermore, upon handling keypress events, the laptop node passess messages to the servo node and serial node to implement servo and motor controls respectively. Upon a disconnection between the laptop and serial node, condition 1 in Figure 3 is acted as a fail-safe mechanism to stop the robot's motors after 10 cycles of the program's main loop.

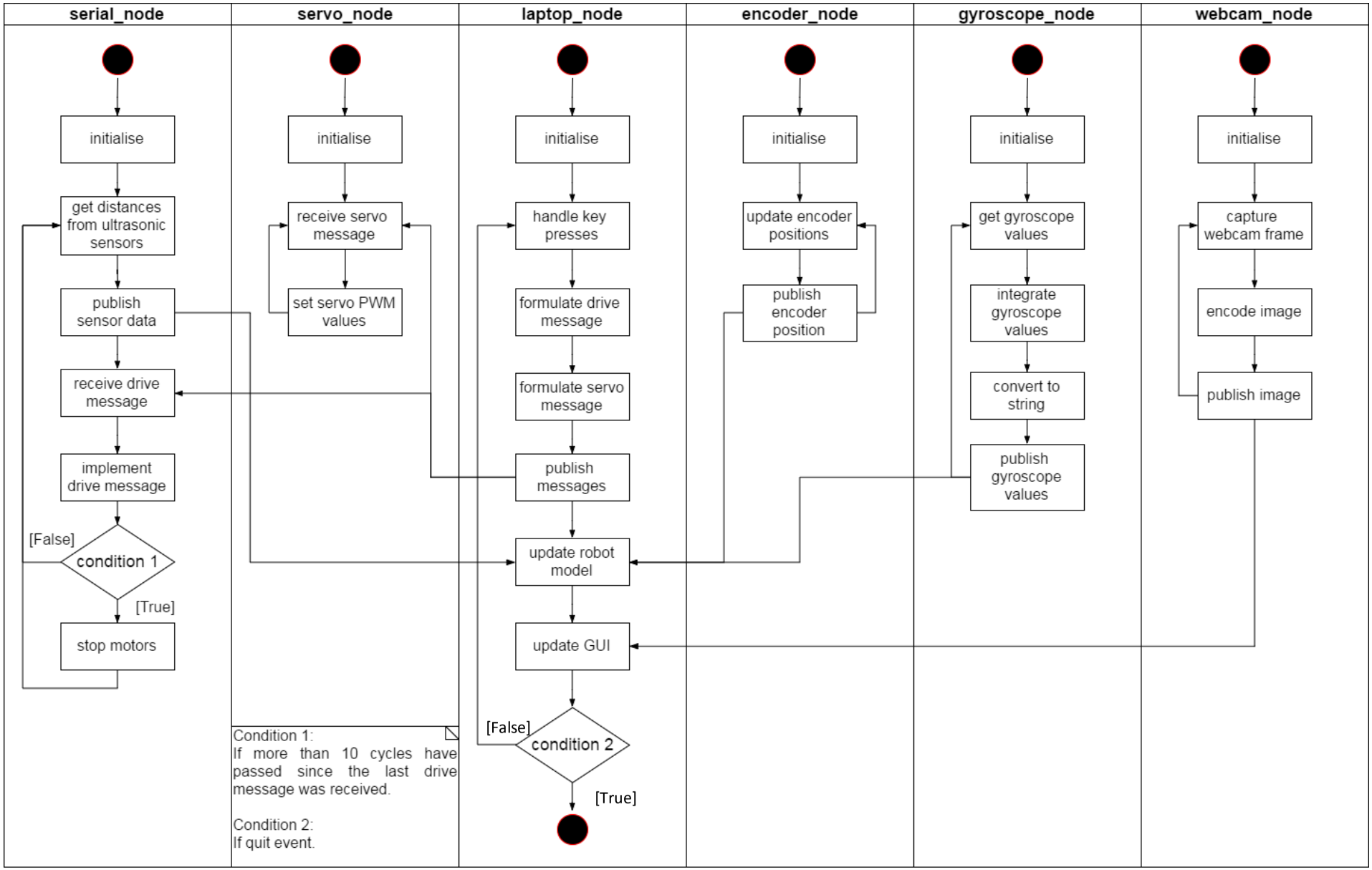

Figure 3: activity diagram for the multi-tier robot system.

2.3.Process Structures

The core component of this project was the LaptopNode which hosts both the GUI and the computer model of the robot and its environment. Figure 4 shows a class diagram for the LaptopNode and illustrates this relationship.

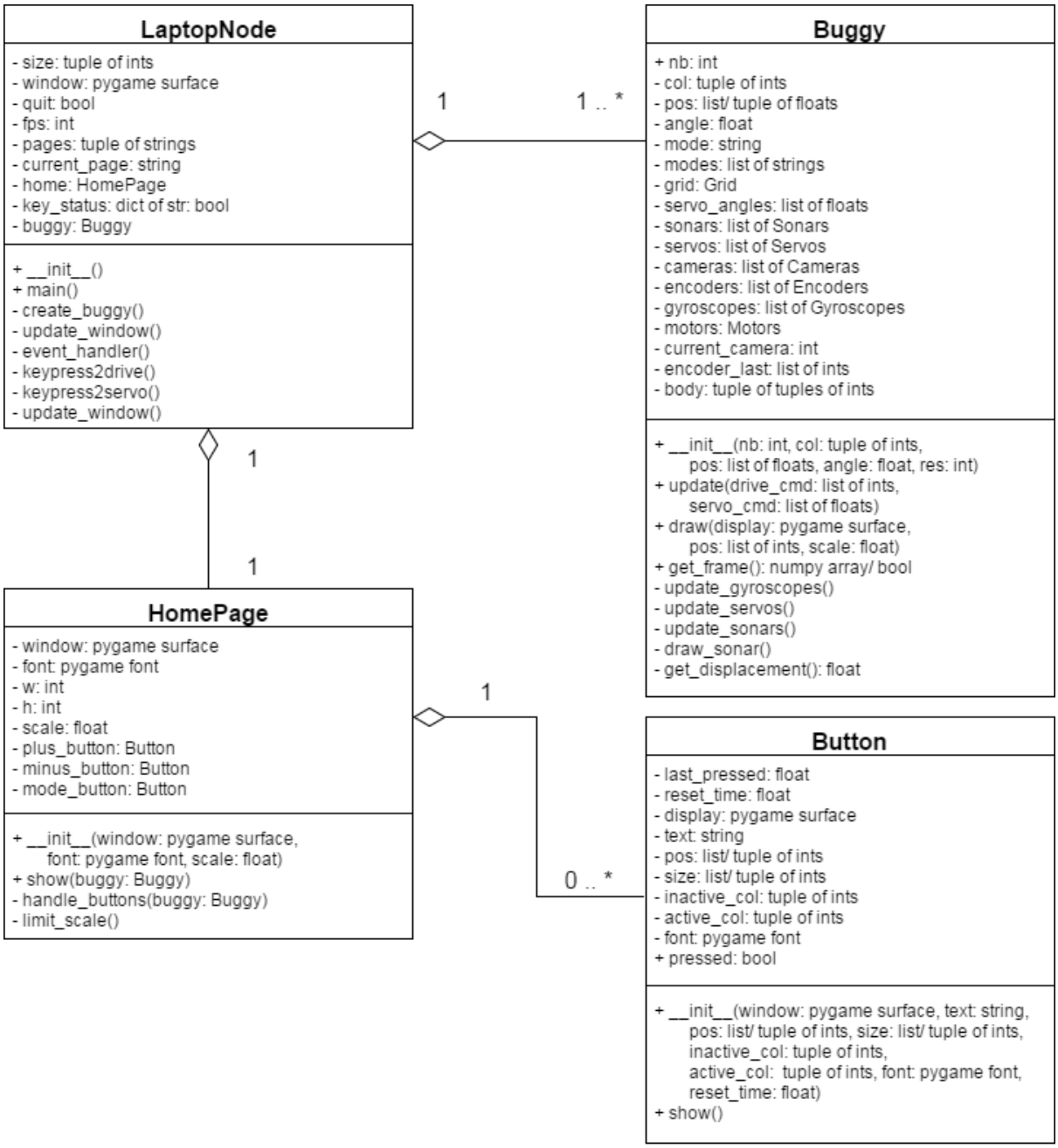

Figure 4: UML class diagram to show attributes of the LaptopNode.

In order to effectively process multiple difference sensors and actuators, a class was written for each device type. Figure 5 gives an overview of these classes and their relationship with the Buggy class. The Grid class was crucial for tracking detected objects in the environment and also handled representing this environment in the GUI. Motors object handled the publication of drive messages to the robot.

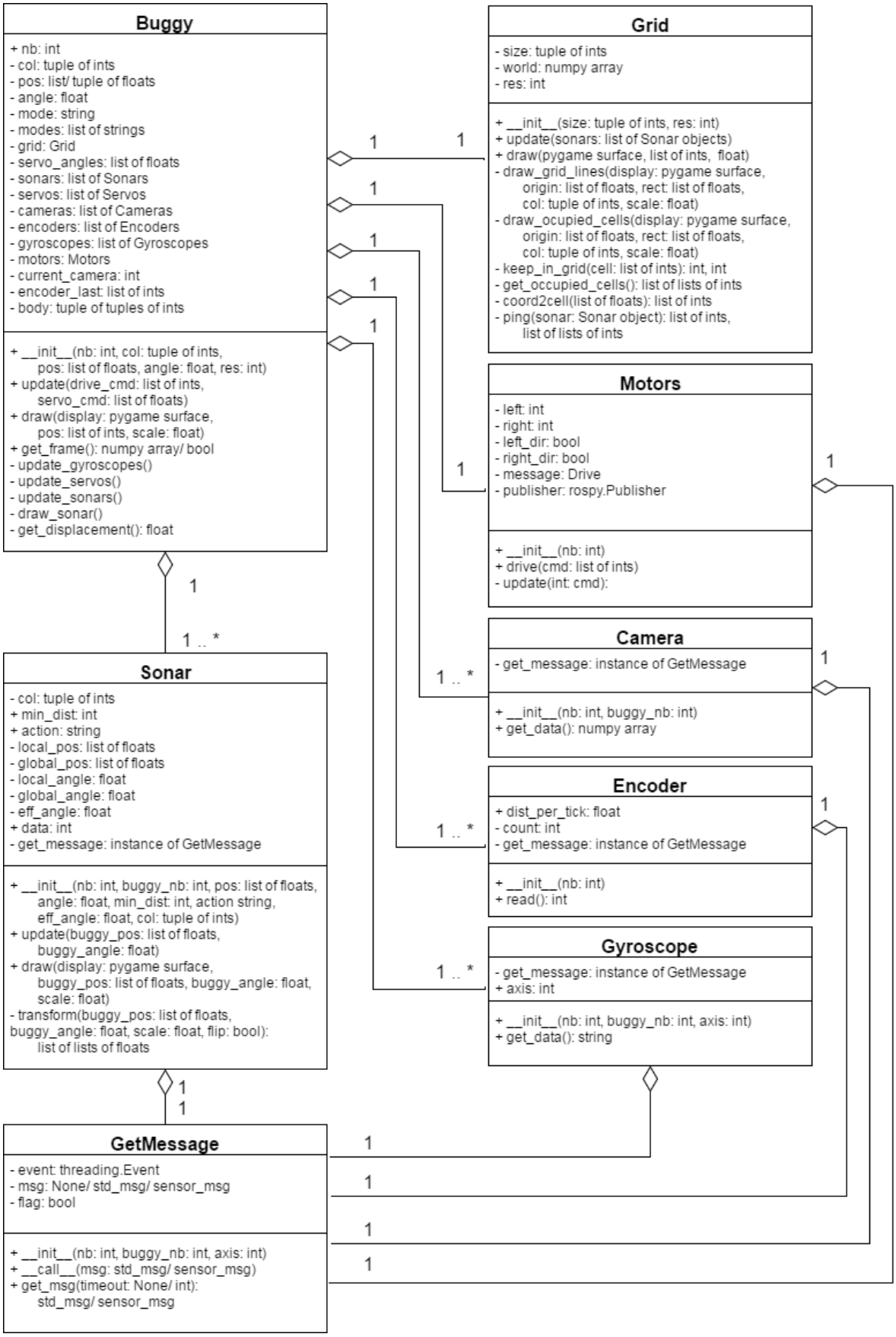

Figure 5: UML class diagram showing attributes of the buggy.

Figure 6 contains class diagrams for the three nodes that handle data on the Raspberry Pi.

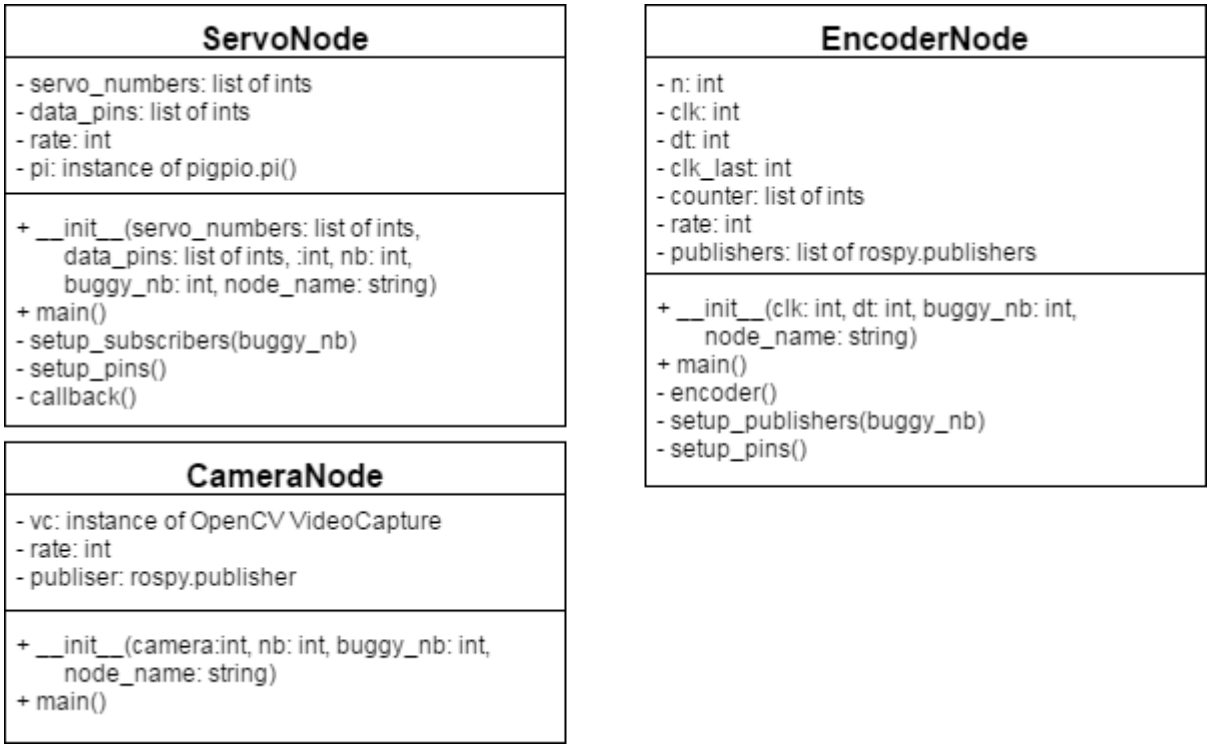

Figure 6: UML class diagram showing nodes that run on the Raspberry Pi.

Two classes were written in Android to read, process and publish gyroscope data from the Nexus 5's IMU and are described in Figure 7.

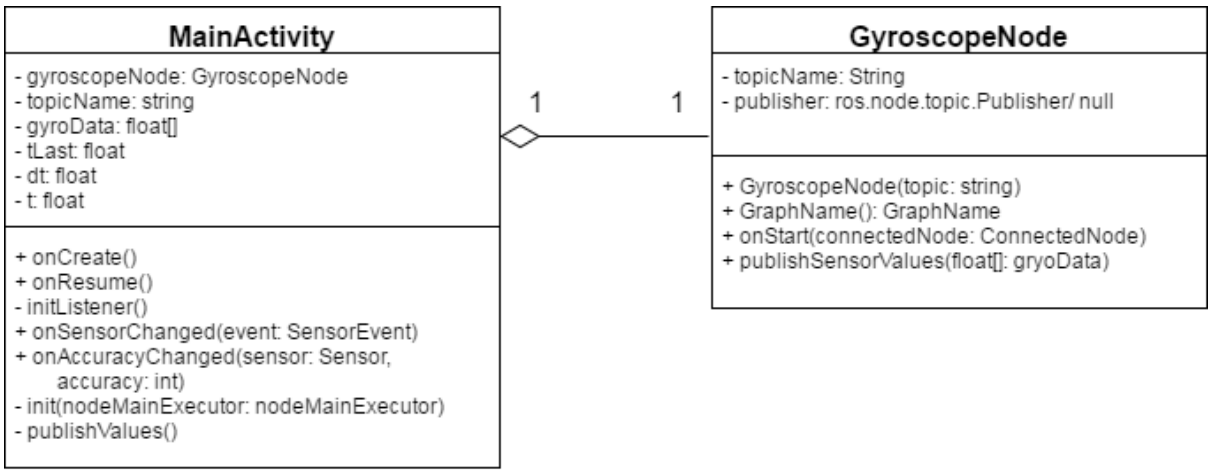

Figure 7: UML class diagram showing Android classes for the Nexus 5.

One class was written for the Arduino firmware, shown in Figure 8, which, once instantiated with the trigger and echo pin values of an ultrasonic sensor, handled pinging the sensor and converting the subsequent ping time into a distance value.

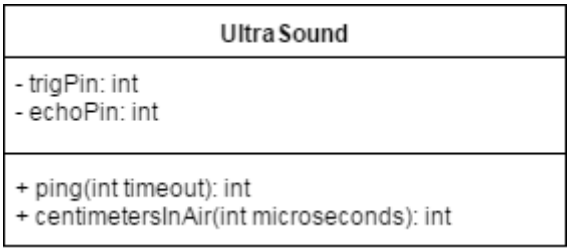

Figure 8: UML class diagram of the UltraSound class.

2.4.Software Design Patterns

The observer pattern is the software design pattern whereby an object maintains a list of its dependents that are automatically notified, usually through calling one of their methods, if a state changes. This was implemented through the use of ROS in which roscore maintained a list of nodes and their subscribed topics and triggers a callback function upon a change to that topic.

Component-based software engineering is a reuse-based approach to defining and implementing loosely coupled independent components within a system. This approach was considered throughout the project as demonstrated by the development of individual classes to handle each sensor/ actuator type. The result of this was an inherently modular and loosely coupled software system, where changes to one class were unlikely to affect others.

The following philosophies from the agile manifesto were also applied:

- Working software over comprehensive documentation,
- Responding to change over following a plan.

The principal of emphasising working software existed throughout the project and this linked well with the concept of a Least Viable Product (LVP). Priority is given to producing software that met the minimum benchmark for functionality and once this stage was reached, the project goals were reassessed to take account for encounter problems and lessons learnt.

3. Performance

The robot constructed as a result of this project is shown in Figure 9.

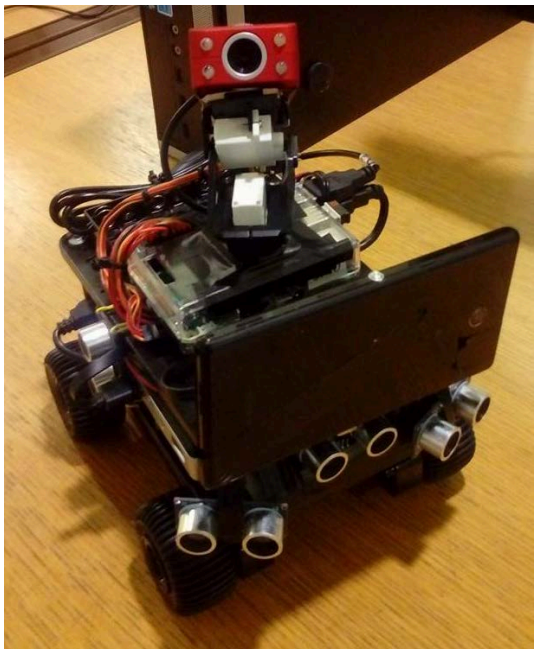

Figure 9: the multi-tier robot.

3.1.Robustness of Interaction with the Environment

Effective and reliable interaction with the environment required seamless integration of multiple sensors, actuators and processors. In addition, the software needed to be robust enough to continuously process a large streams of different data under real-world conditions.

3.1.1. Driving

The principal way in which the robot interacted with its environment was through driving and the robot had two different behaviors concerning this. In 'Manual' mode the robot was controlled primarily by the user through keyboard presses. However, user controls were overridden by automatic responses that were predetermined in the configuration file and triggered by the ultrasonic distance sensors. However, in 'Roam' mode, the robot simply drove forward continuously until automatic responses were required to amend the robot's course.

Initially, problems would occur if the user and robot became disconnected. For example, if the user was commanding the robot to drive forward whilst the GUI was exited, the robot would continue to drive forwards until the connection was re-established. This problem was solved with the addition of a 'cycle counter' in the Arduino. The counter was set to a positive integer upon receipt of a new drive message and this counter value was reduced by one in each subsequent cycle of the main loop. If the counter fell to zero, the robot's motors were stopped.

This was an effective and reliable solution when a value of 10 was used as the initial counter value upon receipt of a message. Higher values such as, 100, resulted in an undesirably long delay between disconnection and braking whilst lower values would result in stuttering.

3.1.2. Automatic Responses

Device specific automatic responses were determined in the configuration file. For each ultrasonic device, the user could input a response and a distance value, below which the response would be implemented, overriding all over drive commands. Possible responses included:

- Forwards
- Reverse
- Left
- Right

For example, the forward, right hand ultrasonic distance sensor was given a distance trigger of 20 cm and an action of “Left”.

In general, these responses effectively guided the robot away from detected obstacles. However, problems occurred due to shallow angles of incidence and objects with non-acoustic properties. Approach of an obstacle were angles of incidence are a shallow is illustrated in Figure 2, where the signal echo cannot be detected by the device. These problems cannot be solved in the software domain and instead, require additional sensors.

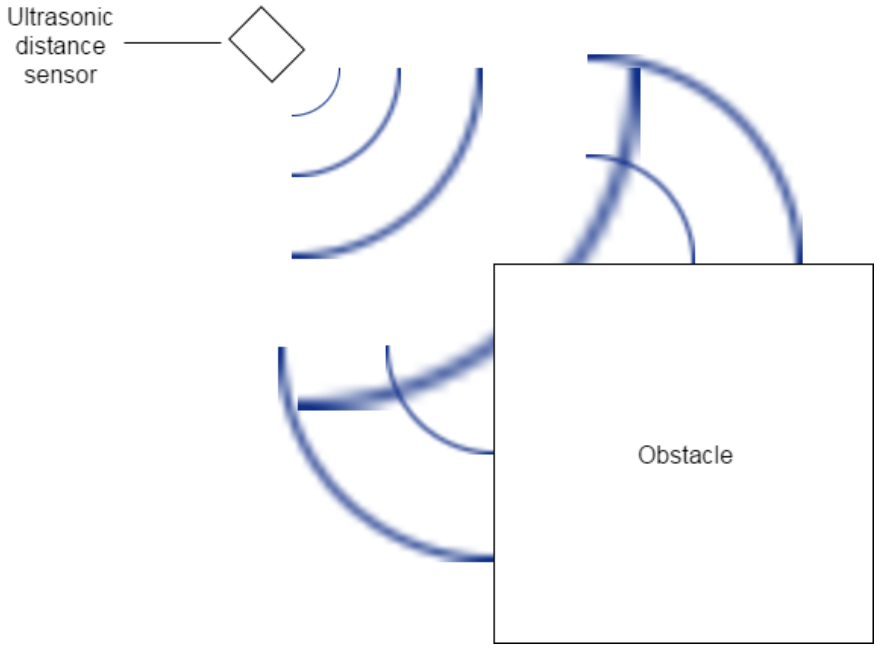

Figure 10: ultrasonic sensor signal at shallow angles of incidence.

3.2.Effectiveness of Simultaneous Localisation and Mapping

The most complex interaction with the environment that this robot demonstrated was its mapping functionality. The largest challenge associated with this was maintaining a world view of the environment whilst the robot’s position and orientation were continuously changing. Given data from the encoders and gyroscope, the robot’s position, relative to its initial position, were tracked. Given this global orientation of the robot and the local orientations of each ultrasonic sensor, the global position of each detected obstacle was calculated and converted into coordinates in the occupancy grid. In addition, coordinates in the grid through which the ultrasonic signal passed were calculated to remove obstacles from the occupancy grid that can no longer be detected.

The robot was able to localise itself relative to its starting position. However, inaccuracies propagated from the encoders and wheel slip and, with continuous movement, the accuracy of localisation reduced.

Without reliable localisation, the position of previously observed obstacles would also drift with time however, as the map was continuously updated, obstacles in immediate proximity to the robot were accurately indicated in the occupancy grid. In order to demonstrate this, the robot was placed in a 2m x 2m pen and commanded to twist on the spot, as shown in Figure 3.

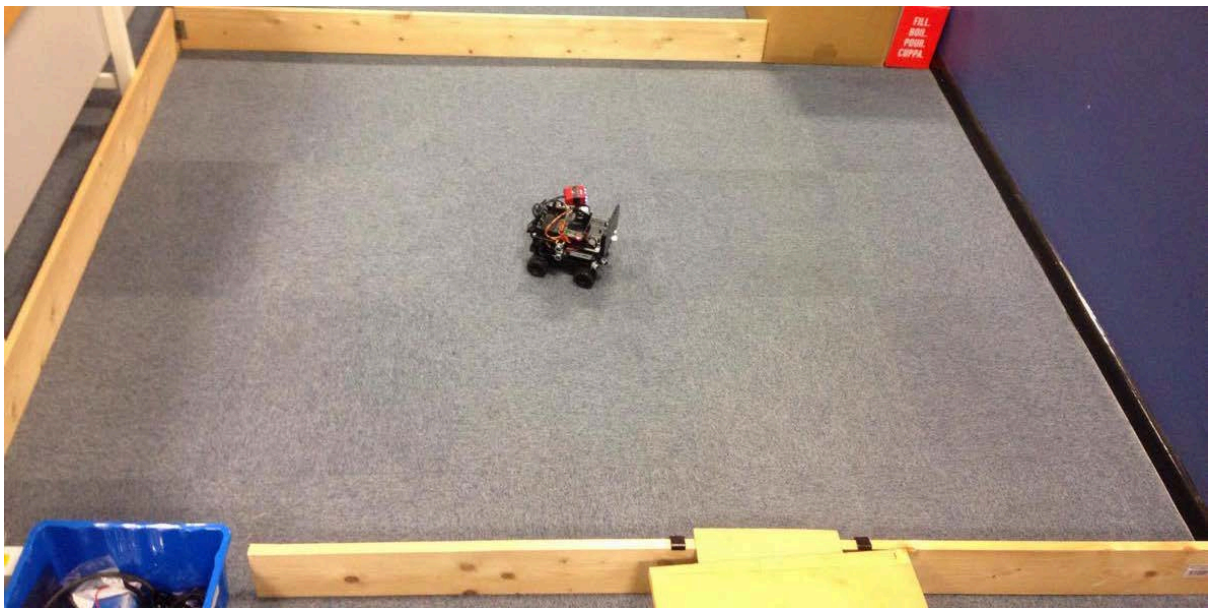

Figure 11: testing the robot mapping capabilities in a square pen.

For two tests, the resultant occupancy grids displayed by the GUI are shown in Figure 4 and Figure 5. The current data reported by ultrasonic devices shown in green, occupied grid squres represented in red and each squre in these grids represented 20cm x 20cm of the real environment. In each test, the robot successfully observed that it was enclosed from all directions and that its enclosure was a rectangular shape. Furthermore, the size of the squre pen represented in the occupancy grid, *c.a.* 2m x 2m, corresponded well with the real size of the enclosure.

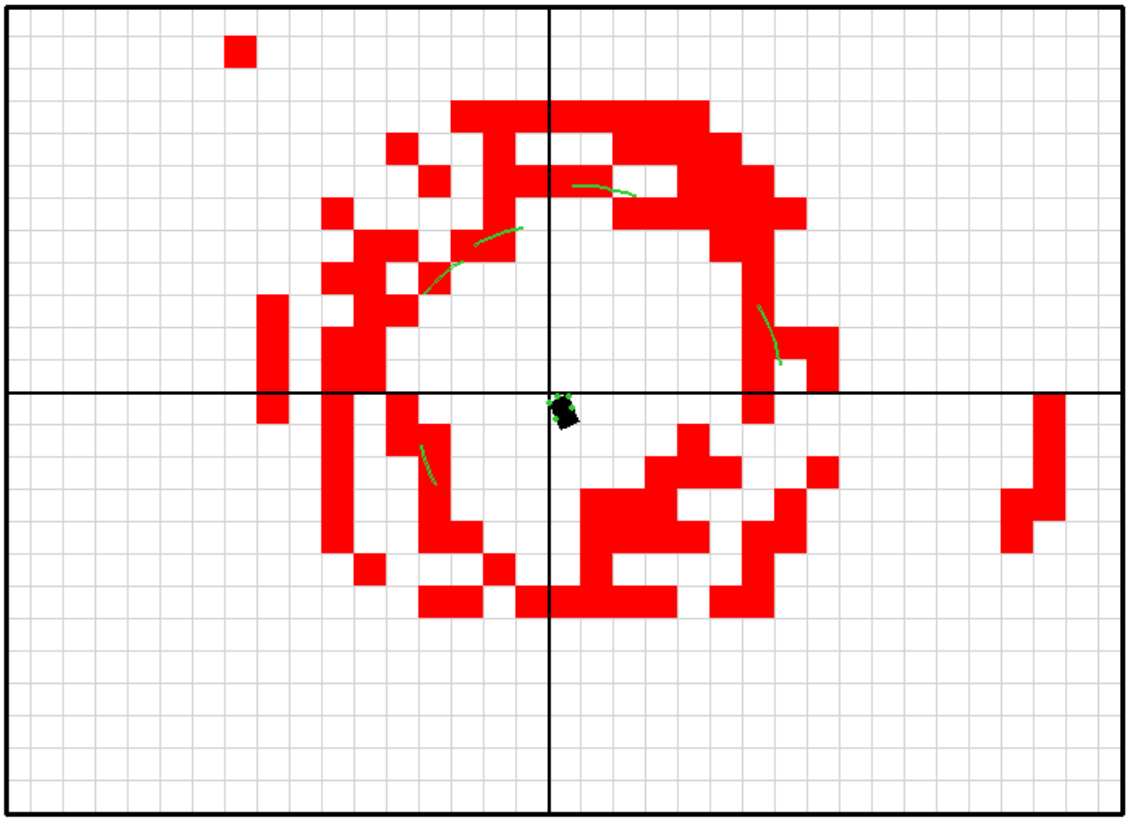

Figure 12: occupancy grid produced after mapping test number 1.

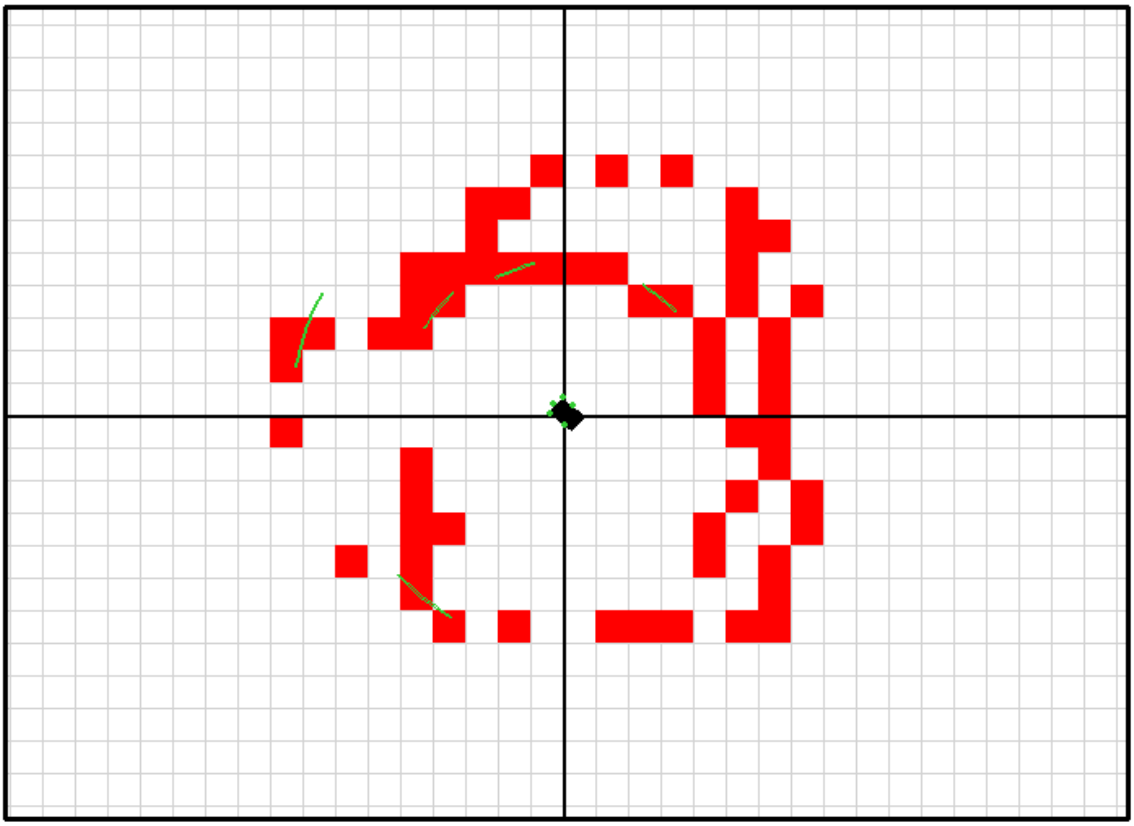

Figure 13: occupancy grid produced after mapping test number 2.

However, this occupancy grid was of finite size and therefore, it was possible for the robot to ‘leave’ the grid world. In this scenario, the software was robust and all normal operation continued minus expansion of the occupancy grid. This was because obstacles detected by the ultrasonic sensors outside the ranges of the occupancy grid were ignored. It was possible for the robot to ‘re-enter’ the grid world, at which point, full mapping capability resumed.

3.3.Ease of Use

3.3.1. Setup

Many sensor configurations were pre-determined by the user in a yaml file. Given these configurations, for each named device, the computer model of the robot automatically initialises an appropriate object to handle properties of that device.

Regarding ultrasonic sensors, the number of devices was specified as well as their corresponding position and orientation on the robot. In addition, the conditions to trigger an automatic response and the subsequent action to be taken are specified in this file. For each servo on the robot, the axis of rotation was specified as well as the minimum and maximum permissible angles. Other devices specified in the configuration file include: encoders, cameras and gyroscopes.

The main reason for this setup was scalability. Sensors and actuators could easily be added to the robot with no need for additional code. Furthermore, the option to control multiple robots from a single instance of the GUI, shown in Figure 6, remains open. This would be possible with only slight additions to the software as multiple robots could be defined in the configuration files and thus, multiple Buggy objects could be initialised.

1.1.1. Operation

The GUI can be seen in Figure 6, and shows the occupancy grid built by the robot model and the camera feed relayed to the user. The green button, currently labeled ‘Roam’, allows the user to toggle between robot behaviours and some interaction with the map is possible, using the zoom buttons.

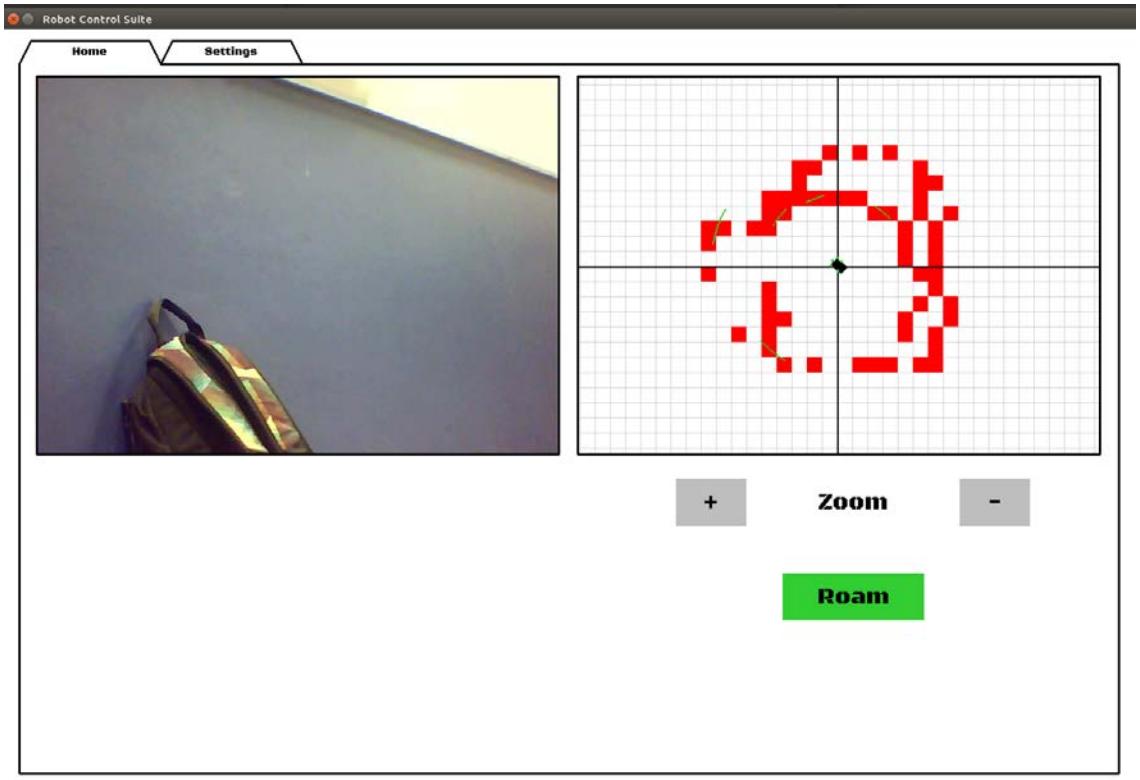

Figure 14: the Robot Control Suit GUI created to interface with the robot.

4. Personal Reflection

This project was a welcomed opportunity to learn a multitude of software engineering concepts including object-oriented programming, ROS, Android and Arduino. The use of classes to abstract robot components proved a successful aspect of the project design which culminated in a complex but robust system. In addition, the use of ROS meant message passing was handled with ease, allowing more time to be allocated to expanding the robot system functionality and robustness.

However, difficulty was experienced when powering mobile processing devices. A rechargeable was used to power the Raspberry Pi but the battery case provided for powering the robot's motors was not sufficient.

Despite allocating generous proportions of time to each stage of development, time taken to complete each stage was almost always underestimated. This must be noted when planning future projects, in particular, more time should be allocated to performance and robustness testing.

The concept of LVP and some principles from the agile manifesto will be applied to future projects. These emphasis of these philosophies on producing a working, presentable product quickly proved effective for rapid software development in this unfamiliar situation.

5. GitHub Repository

Version control was handled throughout this project by GitHub. The repository for this project can be found at:

https://github.com/samuelwestlake/Multi-tier-Robot-System