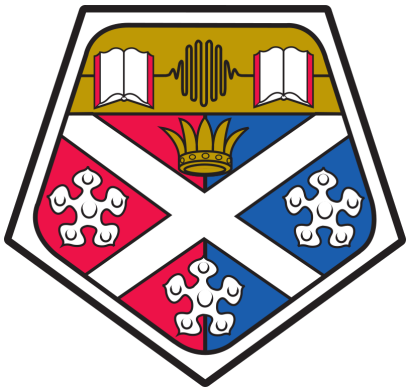
Co-operative Robotics
Using Environmental Sensors

– 19520 Final Report –

Peter De Jonckheere, R. David Dunphy,
Andrew Fagan, Matthew Gaffney,
Kyle Miller

University of Strathclyde, Glasgow



Proposer: Dr John Levine
Supervisor: Dr Marilyn Lennon
16th April 2019

Abstract

Co-operative mobile robotics is a growing field, as multiple autonomous agents can coordinate to solve many tasks faster than a single robot. However, most existing multi-agent systems rely on centralised control and telemetric communication. This is a drawback in environments such as caves or tunnels, where such techniques are restricted due to conditions which limit the range of radio communication.

This project aimed to develop co-operative robots which map and search an area and locate a goal using non-telemetric communication and distributed control, allowing the system to perform in environments with restricted telemetry. In order to simulate these conditions, communication over Wi-Fi was artificially restricted to require line-of-sight. Each agent is able to solve the problem individually; however, the introduction of additional robots to the area should allow them to solve it faster by distributing the task.

Three differential wheeled robots were constructed, with incremental encoders and inertial measurement units used to track their odometry as they traverse their environment. Computer vision is used together with an array of ultrasonic sensors to allow the robots to detect objects, identify other robots, and produce a map of their surroundings. The robots were tested in a custom-made modular testing environment, and their performance was evaluated both individually and co-operatively.

This report presents a detailed analysis of the electrical, mechanical, and software design challenges as well as their solutions. Unforeseen setbacks and technical difficulties delayed the completion of some of the more advanced objectives, which resulted in the AI module not being fully integrated. The system was, however, implemented to a high standard, and its modularity and flexibility will allow future work to be built on the achievements of this project.

Acknowledgements

The authors would like to thank Dr John Levine for the opportunity to work on such an interesting project and Dr Marilyn Lennon for her support throughout the year. Thanks also to Steven Cartwright, whose constant help and advice with the project cannot be overstated. We are indebted to Dr Mark Post, Dr James Irvine, Dr Gordon Dobie and others for providing guidance and expert insight on various topics, and to George Cochrane, Keir Mitchell, and the mechanical workshop team for their help in constructing the maze. The authors would also like to thank friends and family who provided proof reading and moral support. Finally, a heartfelt thanks to Don Eskridge, who divided or united the team depending on the hour of the day.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Using multiple co-operating robots (named and referred to as Blinky, Inky and Clyde) can be a useful strategy when attempting to complete tasks that can be divided and parallelised, such as exploration of a large area. Usually, this requires a centralised control system that can coordinate the robots to accomplish the task. However, in environments where radio communication is impossible or severely restricted, such as underground or in areas with high interference, these systems would be unable to complete their task. In these circumstances, communication may be limited to non-telemetric sensors that require line-of-sight. Therefore, intelligent systems are required so they can co-operate effectively.

Co-operative robotics can be used to solve a variety of tasks more effectively than a more complex individual robot, introducing distribution and redundancy into the system which can be highly advantageous over single agent systems [1]. This redundancy can be mission critical in scenarios where a hazardous environment poses risks to individual robots. Therefore, the loss of any single robot is not fatal to the completion of the task as all the robots are homogeneous and can complete the task individually.

Previous research into co-operative robotics has focused on UAVs [2], non-autonomous agents [3], or made use of extensive communication, such as using the cloud [4]. To simulate non-telemetric communication, this study will use restrict communication to between neighbouring robots, rather than communication over a central server. This application area was inspired in part by a recent incident necessitating cave exploration and rescue in Thailand [5].

To remove the physical complexities of operating on difficult terrain and allow research to focus on communication and problem-solving algorithms, a toy problem has been devised for the robots to solve. This will take the form of a simple maze which contains a target that needs to be found and identified by the robots. The primary aim of the study

is to construct multiple robots which can navigate this maze and coordinate their efforts to find the target more quickly than could be achieved by each robot individually.

An additional requirement is that the robots should be constructed using inexpensive components and a Raspberry Pi single board computer (RPi). This is necessitated by the increased cost implied by the need for multiple robots, in addition to mitigating the risk in the event of the robot failure given the potentially dangerous environments.

1.2 Objectives

- Design a simple differential drive robot capable of exploring and perceiving its environment—Major Objective
- Construct two robots using this design—Major Objective
- Implement a Simultaneous Localisation and Mapping (SLAM) algorithm to allow the robots to explore an area—Major Objective
- Develop a system to allow the robots to interact and communicate with each other—Major Objective
- Implement algorithms to search a maze which can be dynamically parallelised over any number of agents—Major Objective
- Develop a test environment and evaluate the robots' performance—Major Objective
- Add additional robot(s) and evaluate scalability of approach—Optional Objective
- Improve SLAM by adding loop closure between robots—Optional Objective

Chapter 2

Background

2.1 Co-operative Robotics

Robotics is the branch of technology which deals with the design, construction, operation, and application of machines capable of carrying out a complex series of actions automatically [6], [7]. Robotics combines a number of fields from mechanical, electrical and software engineering within a single system to achieve its goals. By combining the three major disciplines of artificial intelligence, operations research, and control theory, a resultant intelligent control system is created [8] which can be used for a wide range of robotic applications.

Co-operative robotics has varied definitions across different papers. One such definition, which generalises co-operative behaviour in robotics, describes it as "joint collaborative behaviour that is directed toward some goal in which there is a common interest or reward" [9]. This description fits the objectives of this study more appropriately than the specific term "swarm robotics", which has a number of additional requirements, including that problem solving should be distributed across the swarm [10]. This definition does not apply to this project, as the robots described herein are capable of operating autonomously and will be able to solve certain tasks individually. In this case, collaboration is used to deliver a performance improvement. For this reason, the more general term of co-operative robotics will be used throughout.

The aim of co-operative robotics is to develop multi-robot systems which have improved performance over single-robot systems [11]. Additionally, by creating a decentralised and distributed system across several homogeneous agents, agent redundancy is introduced which can improve the completion rate of tasks, especially in potentially volatile environments [12], [13]. Co-operative robotics goes beyond the idea of collaborative robotics in requiring an additional aspect of intelligence in the communication and coordination of the individual agents [14].

Co-operative robotics is an area of active research, and is growing in popularity—as the capabilities of the technology increases—as it can be used in a wide-variety of situations. In 2011, co-operative robotics was used as part of the response to the Great Eastern Japan Earthquake. A team from Kyoto University used co-operating, remotely operated, underwater vehicles to search for submerged debris to assist with resuming fishing in the region [15]. Although co-operative, their systems still required a level of human control. Dr Nithin Mathews' research in 2012 provided a communication system without human interaction [16]. This allowed many smaller robots to coordinate to complete tasks that one could not on its own, such as push a larger object. Their system, however, requires a global view of the area to coordinate their efforts. However, since these findings, the progress in the field has accelerated. An excellent example is Sebastien de Rivas' work at Harvard University [17]. In partnership with Rolls Royce, his team have spent the last eight years developing very lightweight co-operating robots. Currently, weighing 1.5 g and measuring 4.5 cm in length, the four-legged micro-robots have eight degrees of freedom, and are being developed with the aim of reducing their size so they can inspect the internal workings of an engine.

2.2 Robotic Control

As the robots in this study are homogeneous and should be able to complete tasks individually, control is limited to the control of a single robot in the complete system. Robotic mechanisms form the control system for a robot and connect the fixed parts of the robot together by joints, allowing motion between these fixed parts [18]. Actuation of the joints, usually by motors, imparts forces on the robot which allow it to move and perform a variety of tasks [18]. The movement of these actuators is influenced by several sensors which provide the system with information about its environment. These sensors can also influence the movement of the actuators through feedback from previous movement instructions [18].

Robots generally have both exteroceptive and proprioceptive sensors. Exteroceptive sensors provide information describing the robot's environment, such as a range sensor. These can take a number of different forms including ultrasound, infrared, Light Detection And Ranging (LIDAR) and stereo computer vision. Each of these sensors use unique methods to determine the distance from the robot to another object in its environment. Ultrasound sensors use high frequency sound waves which reflect from surfaces and return to the sensor. The time taken to return and the speed of sound is used to calculate the distance to the object. Infrared sensors work in a similar fashion but with invisible light instead of sound.

LIDAR is a more advanced use of invisible light to more accurately detect distance using more powerful and precise beams of light than in the infrared case [19]. Binocular

vision allows depth perception to take place similar to that allowed by human vision by perspective-based cues [20], [21].

Proprioceptive sensors can either be related to the actuators or fixed parts of the robot. One such sensor is the encoder which is connected to the wheels and provides only feedback information about this actuator. These sensors are connected directly to the shaft of the motors and provide the control system with the actual distance moved by the motors. This allows adjustments to be made and each of the wheels to be maintained at a constant speed when utilised by the Proportional-Integral-Derivative(PID) controller of a robot.

The Inertial Measurement Unit (IMU) is another example. It provides feedback regarding the acceleration and angular velocity with relation to 6 different axes (x, y, z in both linear and angular planes) or Degrees of Freedom (DOF). This can be combined with information from other sensors in order to reduce the overall error in determining the robot's location relative to its starting position.

2.2.1 PID Controller

As mentioned above, PID controllers are a very common solution to problems where the error of the system (the difference between the current state and the desired state) can be measured. Proportional-Integral-Derivative refers to the function of the error with which the output is calculated. The output of the system is the summation of a term which is proportional to the error, a term which is related to the error integrated over time, and a term which is proportional to the rate of change of the error [22]. This is shown by Equation 2.1.

$$O(t) = k_p e(t) + k_i \int_0^t e(t)dt + k_d \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{2.1}$$

If K_d and K_i were both 0, and only the P term was active, the output would change to reduce the error in the system. For example, with a temperature controller, if the temperature was too low the heater would turn on, with the difference between the actual temperature and the required temperature used to adjust how high the heater was set. In many systems, this simple implementation is adequate, however it has several limitations.

Consider for instance the temperature controller example where heat is being lost in the system is equal to $K_p e(t)$. This would result in a stable state with a potentially large error. To prevent this, the I term can be increased. This will increase the longer an error is present, thus preventing stable state errors.

Again, PI controllers are a common control mechanism, however they often have a problem with overshoot due to system inertia. Once more considering the temperature controller, this is where the heater is set to increase the temperature, and the desired

temperature is reached. The heater turns off, but is still hotter than the rest of the system, causing the system to get too hot. This is especially damaging in systems where the control in one direction is passive, e.g. the system can actively heat up, but must wait for heat loss to cool down. This problem can be mitigated by increasing the D component. This will push the state towards the desired state when the error is increasing (e.g. during over-shoot), and pushes the current state away from the desired state as the error is reducing as to minimise overshoot before it occurs. This does, however, reduce the response speed of the system [23].

One method of tuning a PID controller is the Ziegler-Nichols method [24]. This principle originates from before autonomous robotic control and has been adapted for this application [25]. The method experimentally finds the critical gain, K_u, which is a value of K_p where steady oscillation occurs. The frequency of oscillation, T_u, is then found and these values can then be used to calculate K_i and K_d which results in a tuned system. However, many different formulae and definitions of the critical gain of K_p exist in literature. This leads to the conclusion that significant experimental effort will still need to be invested in the process of tuning the PID.

2.3 SLAM

A key challenge in mobile robotics is for the robot to know its own position in the environment whilst still being able to build a map of its surroundings. This is especially true in the absence of external referencing systems such as GPS to aid in knowing its relative position. This is known as the Simultaneous Localisation And Mapping (SLAM) problem and has been one of the most extensively researched topics in mobile robotics over the last two decades [26]. As the robot's estimate of its position is affected by both the previous state's uncertainty and any errors in the current measurement, the uncertainties compound over time. To rectify this, a map with distinctive landmarks can reduce its localisation error by revisiting these known areas. This is known as loop closure.

SLAM generally rely on sensor fusion algorithms as part of their implementation. These take in readings from an array of sensors and calculate an estimated state change based on the probability of error for each sensor. This mitigates errors, resulting in more reliable estimates than is possible with either one sensor. A standard approach is to use sensor fusion to combine odometry readings from wheel encoders with acceleration information obtained from an inertial measurement unit (IMU) to correct for errors caused by wheels slipping and sensor imperfections.

There are a large variety of solutions to suit various system requirements, these can be categorized as either filtering and smoothing. Filtering creates a state estimation using the current robot position and the map. The estimate is augmented and improved by using the new measurements as they become available. Some popular approaches to

filtering are techniques such as Kalman filters, particle filters and information filters. Smoothing techniques involve a full estimate of the trajectory of the robot from all available measurements. These typically use least-square error minimisation techniques and are used to address the problem known as the full SLAM problem which attempts to map the entire path.

The state of the system is known as x_k which is a function that uses the previous state to determine the next state. As this can not be perfectly accurate, there will be uncertainty in the readings that must be considered. As a result, x_k, which is known as the motion model, is defined as

$$x_k = f(x_{k-1}, q_{k-1}) \,, \tag{2.2}$$

where q_{k-1} is the randomness introduced to the system. As such, this can also be represented by the probability distribution

$$x_k \sim p(x_k|x_{k-1}) \,. \tag{2.3}$$

Both equations imply that the state is stochastic and depends on the previous state. The probability distribution emphasises that the current state is drawn from a distribution of possible states based on the previous state. Given that a perfect sensor is not possible, the current state will also have noise in the reading. This is known as the measurement model and can be defined as

$$y_k = h(x_k, r_k) \,, \tag{2.4}$$

where r represents the uncertainty of the sensor. As before this can be expressed as an uncertainty model:

$$y_k \sim p(y_k|y_{k-1}) \,. \tag{2.5}$$

It is assumed that the motion and measurement models are Markovian in that the current state only depends on the previous state. The measurement model only depends on the current state and no previous values.

By applying Bayes' theorem and marginalisation the current state can be described as

$$p(x_k|y_{1:k-1}) = \int p(x_k|x_{k-1}) p(x_{k-1}|y_{1:k-1}) dx_{k-1} \tag{2.6}$$

$$p(x_k|y_{1:k}) = \frac{p(y_k|x_k) p(x_k|x_{1:k-1})}{p(y_k|y_{1:k-1})} \,. \tag{2.7}$$

Equation 2.6 is known as the predict equation. By integrating over the previous state, all potential outcomes of the state x_k are considered. Equation 2.7 is referred to as the update equation, as the prediction is updated using the new measurement information [27].

One of the most common methods of implementing SLAM is filtering using an Extended Kalman Filter (EKF). An EKF is an efficient, recursive filter that estimates the state of a dynamic system from a series of noisy measurements [28]. This uses the premise of the predict and update equations as joint probability distributions. Given variables defined on a probability space, the joint probability distribution gives the probability that each of the variables falls in any range or set. It uses these techniques to estimate a state vector containing both the location of landmarks in the map and the robot pose [29].

2.4 Computer Vision

Computer vision is the analysis of digital image or video frames to allow a computer system to gain a high-level understanding of the 3D environment contained within the image [30]. A common application for computer vision is the identification and classification of objects. Identification generally involves the recognition of features of the object and the comparison of these features and their relative positions to a known model of the object. Classification usually involves machine learning algorithms to build up a definition of the object based on its visible properties [31].

Computer vision can also be used to triangulate the position of objects in the field of view by measuring the discrepancy in the object's position in the two camera frames given a translation matrix relating the two cameras.

Computer vision can be well integrated with SLAM providing a means of both the measure of distance and the identification of distinctive features in the environment to allow loop closure[32].

2.4.1 Object Detection

CNN

A common method for object detection is to use a Convolutional Neural Network (CNN) [33]. This is a deep learning method that considers the relative position of pixels in the image. It works by scanning a Locally Receptive Field (LRF) across the image, and searching this small group of pixels for patterns at each position as it moves. This results in a set of matrices of outputs, which can then be scanned again. With each iteration, the complexity of pattern which can be detected increases. The output of these convolutional layers is then pooled (combined to reduce size), and used as the input to a neural network which can then classify the image.

Feature Based

Another approach to object detection is feature based. This involves the identification and comparison of key points in an image of the target object and in the camera frame [34]. There are several key point detection algorithms—SIFT, BRISK and ORB, to name a few—each of which has its own way of describing patterns in the pixels and selecting which will be rarer and therefore, more useful when identifying objects. Each also has a method of quantifying the similarity of two matches called a distance metric. The algorithm then looks for features in both the frame and the object image which score low in the distance metric (indicating similarity), filters the matches to remove false positives, and then uses the relative positions of the matches to estimate an outline of the object in the image frame.

2.5 ROS

The Robot Operating System (ROS) is a framework for developing robot software designed to allow flexible software design. It is a collection of tools, libraries and conventions that aim to simplify creating complex and robust robotic systems across a variety of platforms [35]. It was designed with the objective of being as modular and distributed as possible. This modularity allows the user to be able to use as much or as little of ROS as they desire, where their own implementation can be easily fit into the system [36].

ROS uses a peer-to-peer networking topology to allow communication throughout the system. These systems consist of a number of processes that perform the system's computation called nodes which can run across multiple machines. The peer-to-peer topology requires a lookup mechanism to allow processes to find other process' addresses at runtime so they can communicate. Nodes communicate by passing messages that are data structures of typed fields which can include arbitrarily nested structures and arrays [37].

Nodes can use two distinct ROS frameworks, services and topics. Services are synchronous and perform like function calls in traditional programming languages, where only one node in the system can provide a service of a specific name. Alternatively, topics are asynchronous streams of objects published by a node. Other nodes can subscribe by creating a handler function when a new data object is available. Multiple nodes can concurrently publish and/or subscribe to the same topic and a single node may publish and/or subscribe to multiple topics.

ROS was also designed to be language-neutral and supports languages such as C++, Python and Octave. To support this cross-language development, ROS uses a simple, language-neutral Interface Definition Language (IDL) to describe the messages sent between modules [38]. Code generators for each language then generate native

implementations which are serialised and de-serialised by ROS as messages are sent and received. This results in a language-neutral message processing scheme where languages can be used as the programmer prefers based on the requirements of that given module.

There are other alternatives to ROS, such as Yet Another Robot Platform (YARP). YARP was designed to attempt to make robot software more stable and long-lasting without compromising flexibility to change the sensors, processors and actuators. It also communicates using a peer-to-peer topology with an extensible family of connection types. However, YARP is written in C++ and does not support other languages [39]. The YARP model of communication is transport-neutral, meaning the details of the underlying networks and protocols in use are decoupled from the data flow [40]. Compared to ROS, YARP is less widely used. As a result, it does not have the same extensive range of libraries available which implement commonly required functionality for a robotic system.

2.6 AI

Artificial intelligence is another component of intelligent robotic control and is the development of computer systems able to perform tasks normally requiring human intelligence [41]. Searching problems, and the algorithms which solve them, are a common branch of artificial intelligence, into which much research has been undertaken. Search algorithms and optimisation algorithms can be thought of as highly similar, and in the case of a weighted tree search, which maze exploration can be modelled as, either can be used to solve the problem [42].

2.6.1 Maze Exploration

The exploration of an unknown environment is a well-researched problem in robotics. Robot exploration is particularly important for environments that are difficult or dangerous for humans. There are many algorithms that can be used for exploration such as Wall Follower, Trémaux's and Pledge.

One of the simplest exploration algorithms is the Wall Follower algorithm, also known as the right-hand rule, when prioritising turning right, or left-hand rule, when prioritising turning left. If the maze is simply connected, that is there are no loops in the maze, then the Wall Follower algorithm is guaranteed to reach a goal/exit if one exists. Otherwise, the algorithm will return to the starting point having traversed every corridor at least once [43]. The steps of the algorithm are relatively straightforward. The right-hand rule algorithm is shown in Algorithm 1.

These steps ensure a wall is kept on the right hand side of the robot at all times. The left-rule is simply the opposite where the robot turns left if possible to keep the wall on the robot's left. Therefore, this algorithm is inherently inefficient as it exhaustively

Algorithm 1 Wall Follower Algorithm

```
repeat
  if robot can turn right then
    turn right
  else if robot can go straight on then
    go straight on
  else if robot can turn left then
    turn left
  else
    dead end reached, turn
  end if
until goal is reached
```

searches the maze. In many cases, one of these algorithms would be significantly quicker than the other but which cannot be determined without prior knowledge of the maze.

An alternative to the Wall Follower algorithm is Pledge's algorithm. This aims to solve the problem where Wall Follower could be stuck in a loop. An example of which is: if walls form to create a rectangle in the centre of the maze, the robot would continually turn right or left (depending on algorithm) around that rectangle, never finding the exit. Pledge solves this by keeping a count of the number of turns made (if not right-angled turns, then the angle of turn is used instead) as well as the initial direction of travel [44].

Algorithm 2 Pledge's Algorithm

```
Set angle counter to 0
repeat
  repeat
    Walk straight ahead;
  until wall hit;
  Turn right;
  repeat
    Follow the obstacle's wall;
  until angle counter = 0;
until exit found;
```

Using the initial direction and counting the turns made, allows the algorithm to avoid traps such as loops which simple wall followers can be caught in, such as an approximate "G" or "6" shape.

The Trémaux maze-solving algorithm requires the robot to record its path and mark the routes it has taken throughout its navigation routine. Any given path can be unmarked, marked once or marked twice. If a path is marked twice that indicates the robot has travelled down it in both directions. The robot will not travel down any path marked twice for a third time, therefore treating them as dead-ends. Paths without markings are prioritised, before choosing those marked once in search of further unmarked paths. If

a solution is found then the path that is marked only once is the route from the goal back to the start point. Otherwise if no solution is found, all paths in the maze will be marked twice [45]. Importantly, this will likely not find the shortest path, but it will be guaranteed to find one if it exists.

2.6.2 Path Finding

There are many algorithms that can be used to find the shortest path through a graph such as Dijkstra's algorithm and A* search. In the context of this project, path finding is likely to be used once the goal has been found to return the robot back to its starting position. In the case of an unweighted graph (a graph where all edges are equivalent to search), one of the simplest algorithms is breadth-first search. This explores all neighbours at the same depth before moving onto nodes at the next depth level. This would be guaranteed to find a path with the shortest number of edges and is $\mathcal{O}(|V| + |E|)$ where $|V|$ is the number of vertices and $|E|$ is the number of edges [46]. Breadth-first search searches the oldest discovered node first meaning many backtracks are needed to reach the oldest discovered node. As backtracking in physical space is expensive, this results in a very costly search for a single-agent. As more agents are added, the cost of this decreases as less backtracking would take place.

In the case where the edges have a weighting, which in this case would be the distance to next junction, Dijkstra's algorithm is one of the most commonly used methods for finding the length of shortest path between a specified pair of nodes in the graph. The steps for Dijkstra's algorithm are outlined in Algorithm 3.

This version of Dijkstra's algorithm runs in $\mathcal{O}(|V|^2)$ [47]. An improvement to this original algorithm using a min-priority queue implemented using a heap, runs in $\mathcal{O}(|E| + |V|log|V|)$ and was first proposed in [48].

Developed initially as an extension of Dijkstra's algorithm, A* is a path-finding algorithm that typically achieves better performance by use of a heuristic. At each node, A* chooses the node which it believes can be used in the shortest path. It does so by choosing the node with the lowest value of f, where f is the sum of g and h, g being the distance to the current node plus the known distance to that adjacent node, and h the heuristic value from that node to the goal. The heuristic value can be found exactly as the first step of the algorithm though this is very time consuming, so approximate heuristics are used such as the Manhattan distance, Diagonal distance and the Euclidean distance.

Algorithm 3 Dijkstra's Algorithm

Require: G = {vs, es} : G :: Graph, vs :: List(Vertex), es :: List(Edge)
Require: initial ∈ vs : initial :: Vertex
Require: successors :: Vertex → List(Vertex); Finds vertices connected to input
Require: weight :: Edge → Int; Weighting factor applied to edge
Require: edge :: Vertex → Vertex → Edge; Finds edge which links vertices
  for all v ∈ vs **do**
    distance[v] ← ∞
    visited[v] ← False
  end for
  distance[initial] ← 0
  repeat
    next ← v : distance[v] = min(distance) **and** visited[next] = False
    if no available next **then**
      return distance
    end if
    if distance[next] = ∞ **then**
      return null
    end if
    visited[next] ← True
    for all v ∈ successors(next) : visited[v] = False **do**
      if distance[next] + weight(edge(next, v)) < distance[v] **then**
        distance[v] = distance[next] + weight(edge(next, v))
      end if
    end for
  until return condition reached

Chapter 3

Project Management

The project was managed throughout by an elected Project Manager (PM) (Andrew Fagan), who was tasked with ensuring tasks were completed in the timeline decided by the whole group. This allowed every member in the group to have an equal voice in decisions regarding the project while still having a overall manager of the timeline. Using this system, with a minimal number of managers and no strictly assigned teams, provided workflow fluidity and flexibility and thus no work was restricted to a specific subset of the group. This also improved the group's understanding of the project and the aspects which will be detailed herein.

3.1 Project Structure

It was determined early in the timeline of the project that many members of the group were familiar with Agile methodologies [49], and were in favour of adopting such a methodology into the project structure. Upon review, it was decided that this was not fully possible, as iterating many times, particularly on mechanical design aspects would be costly and time consuming. Hence, it was decided to adopt Agile concepts where possible. The project was broken into high level "projects" on GitHub. Within each "project", a broad range of high-level issues were created which team members were able to assign themselves to. These could be viewed on the issues board on GitHub, along with their current status. If issues remained unassigned from the previous group meeting, the PM was tasked with assigning at the next. Individual team members assigned to an issue also had the responsibility of creating additional dependent issues or linking issues together as they felt appropriate, to allow bottlenecks to be identified. These issues can be seen as analogous to the backlog of many Agile methodologies. The PM also had the responsibility of monitoring timelines on issues and moving team members between issues if needed, to spread areas of expertise or solve "blocking" issues, for example.

Team members working on issues had the freedom to approach them as they felt appropriate, which led to some issues such as PCB design and Comms design being

approached in a highly iterative manner, again in line with Agile methodologies. This view was reinforced following feedback from an interim report and presentation. The high level software architecture however remained broadly intact for the entirety of the project, as too many other issues relied on this.

Regular meetings were held with the project supervisor to report progress.

3.1.1 Consultations

Throughout the project several members of academic staff were consulted to obtain an expert's opinion on either discussions or decisions made by the group. Staff from the EEE, CIS and DMEM departments were consulted at various stages in the project, bringing together each of the individual components of the robot.

From the DMEM department, Dr Mark Post was consulted in the planning stages of the project. With a background in robotic vehicles and SLAM, Dr Post was consulted to advise on the possibilities for operating system to use, possible SLAM implementations, potential objectives and timescales for parts of the project. Only one meeting took place as Dr Post moved to The University of York shortly after the meeting, however, the meeting was highly beneficial and gave a vast insight into existing solutions and software, whilst highlighting potential pitfalls which could be encountered throughout the project.

Dr James Irvine, EEE Department, was consulted informally regarding the use of sockets for communication using a Wireless Ad-hoc Network (WANET). Dr Irvine was approached with an assumption that using the Python Socket library to send serialised data between the RPis would be a robust method of communication between the robots. This was confirmed by Dr Irvine with the additional suggestion to serialise the data as JSON objects.

Dr Gordon Dobie, EEE Department, was consulted towards the end of the project regarding the tuning of the PID controller. Members of the group met with Dr Dobie and explained the PID tuning processes undertook to that point. He explained that although PID was conceptually useful, a simpler PD rate control system (c.f. Section 5.3.4) would be more appropriate in this particular case. Following the in-depth investigations made into PID controllers, the group was fairly confident, following some basic guidance provided by Dr Dobie, on implementing the recommended system.

Prior to the arrangement of the meeting with Dr Dobie, Dr Phil Rogers was consulted regarding the tuning of the PID controller. He suggested providing a frame of reference to correct the PID controller and maintain a straight line with regards to its reference frame. While the group waited on the meeting with Dr Dobie, a specialist in robotic control, Dr Rogers' advice was followed and SLAM was progressed in the interim.

3.1.2 Mechanical Workshop

The construction of the modular maze used for testing was outsourced to the mechanical workshop located in the EEE department at the university. A number of discussions with the mechanical workshop were used to specify the design before and during construction. The design was iterated upon multiple times to achieve the best possible outcome. More information can be found in Section 6.1.

3.2 Timeline

The Gantt chart shown here in Figure 3.1 was created using the objectives (Section 1.2) and their respective sub-objectives. The timescales were determined approximately using knowledge obtained from research and consultations, with additional time added to account for possible technical risks. In line with the Agile methodology adopted, the tasks were given appropriate time and a small margin for error within the Gantt chart. For example, assembly of remaining robots was allocated a longer time than expected as this involved lead times for components and spanned the period of time when the labs were closed and construction work was not possible.

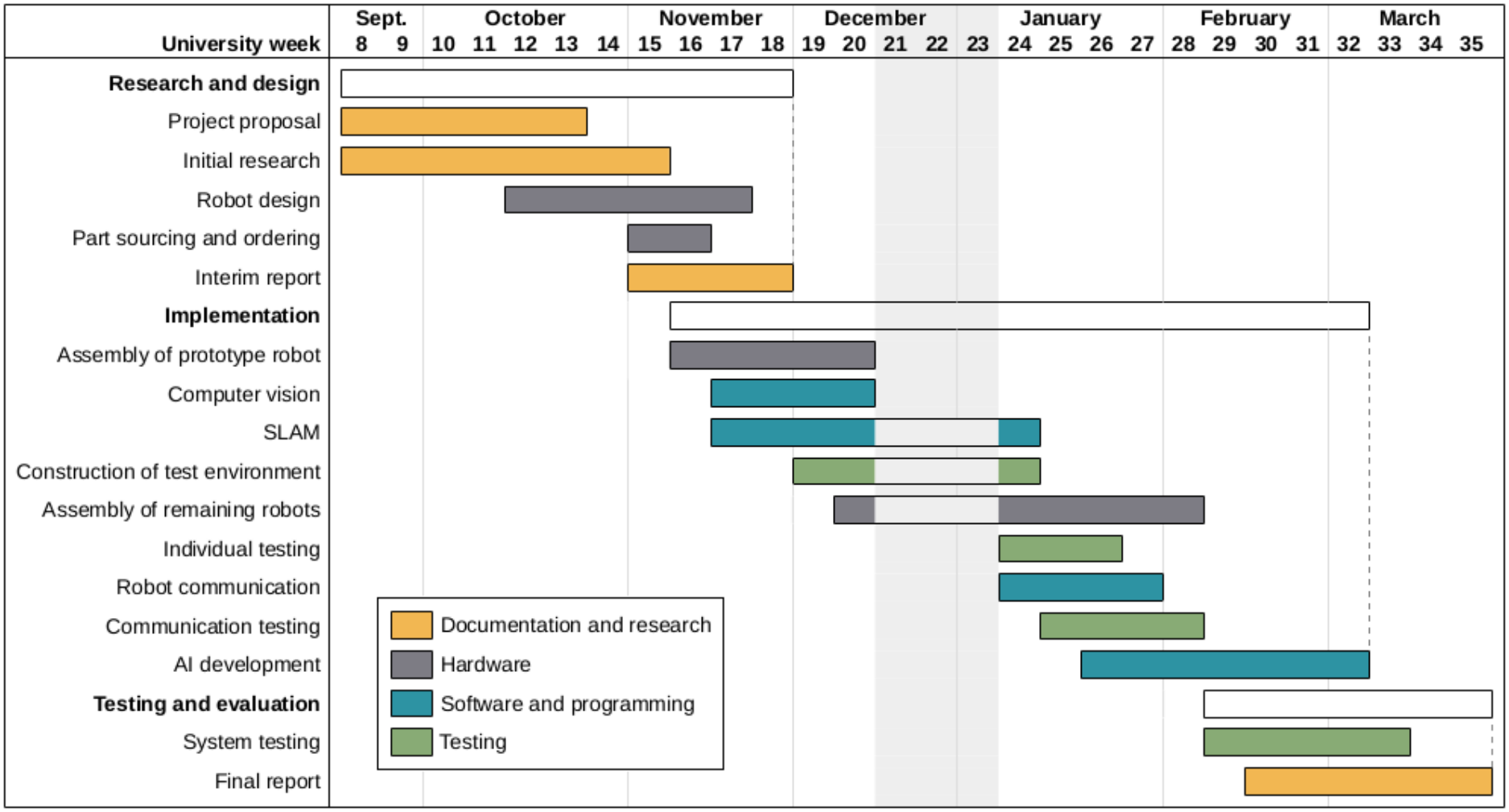


Figure 3.1: Gantt chart

The Gantt chart was adhered to for the most part in the early to mid stages of the project. Towards the latter stages of the project, slippage occurred in the timelines which meant that some of the later objectives were not completed fully. A number of the slippages in the latter stages were due to unforeseeable electrical or mechanical issues. Multiple issues arose including motors leaking grease, various power issues and one RPi being damaged as a result of an electrical short during the testing phase. Several technical slippages occurred in the Gantt chart in the latter stages namely: PID tuning (c.f. Section 5.3) and SLAM, which was eventually spilt into 2 parts (c.f. Sections 5.4 & 5.5).

In hindsight, a revised Gantt chart could have been created in the mid-stages of the project, possibly following feedback from the interim report and presentation to reflect changes in timelines and also the approach taken given presentation feedback.

3.3 Risk Evaluation

A risk evaluation took place at the beginning of the project and was re-evaluated at the time of the interim report. The technical risks evaluated at this time mainly centred on the possible lead times for ordering parts. At the time of ordering, this was thought to be a major issue, and parts for each of the three robots was ordered. One part of the order was not in stock and was back-ordered with an expected lead-time of 5 weeks from the end of December. This happened to not be the case and ordering lead times were not an issue within the project on the whole as other work, such as designing and constructing the modular maze for testing could be carried out in the interim.

It was then expected that component failure would form the second main technical risk of the project. This was partly true, as mentioned in Section 3.2, as several components failed throughout the project and had to either be replaced or fixed, consuming time in the project. This was partly mitigated by the time left at the end of the project, however, the replacement of these parts did impact on system testing time, as the integration testing for each of the replacement components had to be repeated.

Chapter 4

Electrical And Mechanical

4.1 Chassis

The mechanical design of each robot is central to its functionality with several of the sensors reliant on the accuracy of the mechanical construction. As multiple robots are being built, their mechanical similarity is important to guarantee the sensors and software function consistently across each robot. Given this consideration, Printed Circuit Boards (PCBs) are used in the final iteration of the design—with strip board being used for prototyping—as this will provide additional robustness and guaranteed repeatability between robots.

The chassis used for the robots was the pre-built "hobbyist" chassis from Pololu [50]. This chassis was chosen as it ensured manufacturing consistency between robots and allowed for scalability, due to its availability, both of which were essential design considerations. The chassis were accompanied by a number of fitted parts which were also used such as motors, encoders and a power distribution and motor drive board. Using these pre-built solutions reduced the number of parts which had to be created and fitted, simplifying the construction process. The chassis had numerous mounting holes, which again eased construction, as holes did not need to be drilled in the chassis to mount the PCB and other components. A variety of chassis colours were also used to simplify the robot recognition which was required and will be discussed later in Section 5.6.

During integration testing, issues were encountered with some of the motors, as it was noticed that they were leaking oil or grease from their plastic casing. This resulted in the gears making more noise than they did previously and resulted in motor failure when the motors were run for a prolonged period of time without more grease causing the gears to grind and slip. This was investigated through the manufacturer, however, unfortunately no reply has been forthcoming.

To robustly connect each of the components to the chassis, plastic spacers were used. To obtain the required size of 23mm between the chassis and PCB and between the PCB

and the RPi, spacers of 8mm and 15mm were attached using an M3 screw. For the M3 screw to also be used to connect the relevant parts, the standard mounting holes on the RPi had to be widened slightly. The M3 screws were then used to attach the chassis to the PCB and the PCB to the RPi. The 23mm size was determined using the standard header size which would be between both sets of components in addition to allowing for a small gap between the header and the component above. This additional space allowed for airflow which aided in component cooling.

The spacers also allow components to be placed above each other and above the centre of the robot. Maintaining the centre of gravity close to the centre of the robot, and therefore reducing interference of centripetal forces around the centre of the robot, is vital for the accuracy of the IMU (c.f. Section 4.5.2). By using the spacers to maintain a centre of gravity above the centre of the robot, the general balance of the robot was also aided and the drive control of the robot was made simpler.

4.1.1 Drive System

The drive system used for the robots is a differential drive system (DDR). This drives each wheel independently using independent actuators and the wheels are not connected by a single axle [51, p. 146]. When using a two wheeled robot, DDR allows the robot to rotate on the spot around its central axis when the wheels are driven in opposite directions. This provides a high level of mobility which will aid in the sensing and mapping capabilities of the robot. This also allows the corridors of the maze to be narrower than otherwise, allowing more complex mazes to be formed. The chassis chosen is accompanied by a motor and encoder for each wheel which can be used to obtain data for wheel odometry.

4.2 Power Distribution and Motor Drive

The Pololu Power Distribution and Motor Drive board [52] was selected for power distribution across the robot. This board is the main accompanying part for the Romi chassis chosen and, in addition to power distribution, also provides two Texas Instruments DRV8838 [53] motor drivers to control the motors. The board integrates with the chassis easily and regulates the voltage provided by the 6 AA batteries, using an MP4223H switching buck converter [54], to 5 V or 3.3 V at a maximum of 2.5 A. This can be used to power the microcontroller and peripherals used by the robot, with the motors being supplied with the reverse protected voltage before regulation.

The board was also configurable if required, allowing the supply to be divided into nominal 6 V and 3 V supplies. There are also a number of further jumper connections on the board which can be cut for further voltage customisation.

The DRV8838 Motor Driver takes two inputs for each motor for direction and Pulse Width Modulation (PWM), which allows the motor speed to be controlled. The driver also has a sleep input which can be used to coast the motors if required. Each of the motors' respective sleep inputs are connected by default and therefore should not be driven at the same time, as this could result in a short circuit within the motor driver, however the jumper connection between these can be cut to allow the sleep inputs to be driven separately.

The power board also provides a power button and switch to control the power supply. A separate board with only power distribution was available, however, for the additional small cost, integration of motor drivers was deemed to be a cost effective addition given our space and time limitations.

The boards were soldered to the battery contacts and headers soldered onto the boards as the robots were assembled. Once soldered, the power distribution and motor driver board was continuity tested and monitored throughout testing to ensure correct functionality.

4.3 Range Sensors

For the robot to localise itself, it needs to be able to measure the distance between itself and the surrounding environment. By recording the measurements relative to the robot's current position, a point map can be built up over time and a general map of the environment can be constructed.

4.3.1 Design

Infrared (IR), ultrasound and LIDAR distance sensors were all considered. These are all active sensors, meaning that they measure the effect of an output from the sensor, in these cases, IR light, ultrasonic waves and visible light respectively.

Of these three, LIDAR is generally considered to be the most effective as it is the most accurate and has a 360° field of view. However, it is both too expensive and too heavy for the applications of this project, especially considering scalability requirements.

IR transceivers are far more affordable but have a limited cone of vision per sensor as light deviates very little through air and the maximum range of these sensors is approximately 0.5 m [55]. Ultrasonic transceivers have a wider cone of detection and also a large range [56]. In addition to this, they are also cheaper than both IR and LIDAR systems.

Ultrasonic transceivers work by emitting sound at a particular frequency and measuring the time until the reflected wave returns to the sensor. Multiple ultrasonic sensors of the same design have a significant chance to interfere with each other; however,

IR also has this limitation, while also being more susceptible to external interference from ambient light. A possible solution was to make a budget LIDAR unit using a single IR sensor and a servo motor mounted on top of the robot. This solution would also require a budget and software implementation commitment prior to testing which was not deemed worth the potential added risk to the project. Another consideration was that the minimum range of widely available IR sensors is around 10 cm, which would result in a deadband area around the robot in which objects cannot be detected.

For these reasons, ultrasonic sensors were chosen over IR, with the HC-SR04 selected as a suitable model [56]. This sensor has a range of 2 cm to 4 m, and ranging accuracy of up to 3 mm. Due to the deadband in the immediate proximity of sensors, careful consideration had to be given to their placement on the robot (c.f. Section 4.6). The sensor is widely used and was available within the department, mitigating the risk of prolonged delivery lead times.

4.3.2 Implementation

The hardware required for the ultrasonic sensors chosen is relatively straightforward. They are each connected to a 5 V supply and ground, and the other two pins are connected to standard GPIO pins on the RPi. The ECHO pin has to be connected through a voltage divider, as the RPi is only rated at 3.3 V, while the ECHO pin outputs 5 V. The TRIG pin can be connected directly to the RPi, however, as 3.3 V is above the pin's threshold voltage. The sensor operates by emitting a burst of ultrasound when the TRIG pin receives a 10 µs pulse. The ECHO pin then outputs a high signal equal in duration to the flight time of the ultrasound.

Ranges are measured by the `Ultrasonic` class in `crues_sensors/us_node.py`. The range is calculated by the `get_range()` method shown in Listing 4.1. The `start_time` and `stop_time` fields are updated asynchronously by a callback method tied to rising and falling edges on the ECHO pin. The range is then calculated by the equation $2d = tv_s$, where v_s is the speed of sound. `response` and `offset` are used to correct the measured range based on values obtained from testing, such that the returned value represents the distance from the edge of the robot rather than the distance from the sensor itself.

```
1 class Ultrasonic:
2     # ...
3 
4     def get_range(self):
5         """Get range from ultrasonic sensor in metres.
6 
7         :return: (float) Approx. range in metres
8         :except: (UltrasonicTimeout) If module timed out waiting for GPIO input
      change
```

```
9           """
10          self.start_time = -1
11          self.stop_time = -1
12          GPIO.output(self.trig_pin, GPIO.HIGH)
13          time.sleep(self.pulse_duration)
14          GPIO.output(self.trig_pin, GPIO.LOW)
15          time.sleep(self.sensor_timeout)
16          if self.start_time < 0 or self.stop_time < 0:
17              raise UltrasonicTimeout(self.name, self.sensor_timeout, self.
    start_time < 0)
18          duration = self.stop_time - self.start_time
19          distance = duration * SPEED_OF_SOUND * 0.5
20          return self.response * distance - self.offset
```

Listing 4.1: Ultrasonic code for getting range from a sensor

Interference between sensors was a major concern, as sensors can return incorrect ranges if they detect ultrasound from by other sources. Several options were considered for mitigating this problem. Using different frequency ranges for each sensor was one possible solution; however, this was not possible with the chosen components, which lack the hardware necessary to modulate the frequency. The more common solution is sensor synchronisation, with sensors taking it in turn to record a range. Synchronising the sensors on a single robot is a trivial task, which was completed by implementing an `UltrasonicScanner` class, which polls each sensor in turn, waiting for measurement cycle of 50 ms recommended by the manufacturer. This results in an overall cycle of 150 ms for the sensor array, corresponding a maximum polling frequency of around 6.5 Hz.

Synchronisation of sensors between robots is a significantly harder problem, as doing so relies on a global clock between RPis. As the RPi does not have an onboard real time clock (RTC), the time is reset each time the robots are started. This problem was solved by synchronising the RPis clocks on startup using NTP (Network Time Protocol). As the RPi's communicate using WANET, they are not able to connect to the internet in order to communicate with an NTP server; for this reason it was necessary to set up a laptop as an NTP server.

With a shared global time standard, synchronisation becomes less problematic. Each robot has a unique `pulse_offset` parameter, which specifies the offset from each second of the clock. This necessitates that the ultrasonic pulse rate is an integer multiple of Hz. The practical pulse rate for three co-operative robots is therefore 2 Hz. This has implications for mapping, as the resolution of the map depends on the frequency of measurements, as well as for reaction time, placing a limit on the maximum velocity the robots can safely move at.

4.3.3 Testing

The sensors were initially tested using a signal generator and an oscilloscope. An object was then placed in front of the sensors and moved away. The expected output of this test was a linear increase in the pulse width on the echo pin as the distance increased. This test allowed two malfunctioning sensors to be quickly identified. The pulse width could then be used to calculate the measured distance and this compared to the distance found with a measuring tape. This found approximately correct answers, however the precision with which we could measure the pulse width was not great enough for complete testing.

The wave forms produced are shown in Figure 4.1. This shows the shorter echo pulse following the fall of the wider square wave on the trig pin, produced by the signal generator.

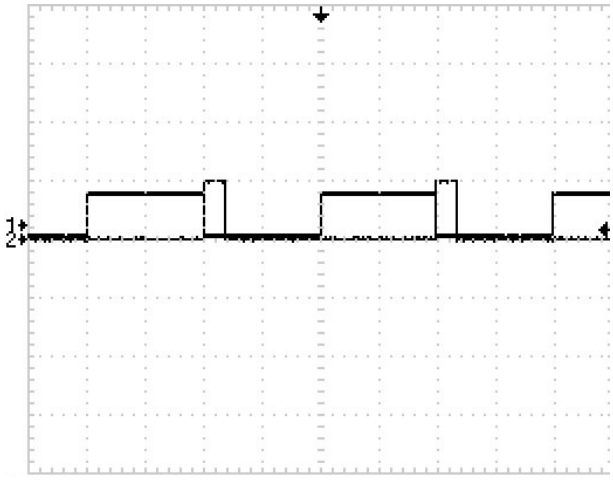


Figure 4.1: Ultrasound response waveform

A circuit was then created to read the values using an Arduino microcontroller to allow the distance readings to be found quickly and the accuracy of the sensors measured. The readings from this were found to be approximately accurate (to within 0.5 cm), although rigorous testing was not performed as the accuracy would be dependent on the temperature due to changes in air pressure, and there was no way of accurately controlling this.

Once the sensor was implemented on the RPi, an experiment was devised to test its level of accuracy. The robot was placed at fixed distances from the wall and at each point 100 readings were recorded. The average response of this experiment across the 1m range can be seen in Figure 4.2.

It can be seen then that the response of the sensor is fairly reasonable producing a linear response with an offset of approximately 18 mm. The values begin to deviate at greater distances, but the robot is unlikely to make measurements over significant distances. Indeed, in normal operation, considering the wide field of view of each sensor,

it is very unlikely that no obstructions will be registered at the extremes of range, and it is unlikely that these values will be as useful as the shorter range ones.

In addition to plotting the average response, Figure 4.3 shows box plots at a subset of each point of measurement, the rest being omitted for readability. It can be seen that the response is fairly consistent at each data point, so a low variance can be safely assumed on these sensors.

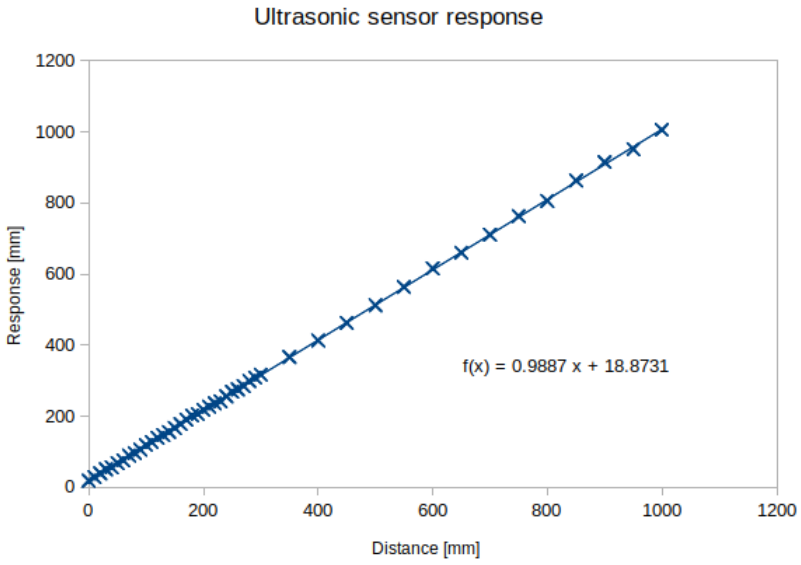


Figure 4.2: Average ultrasound response

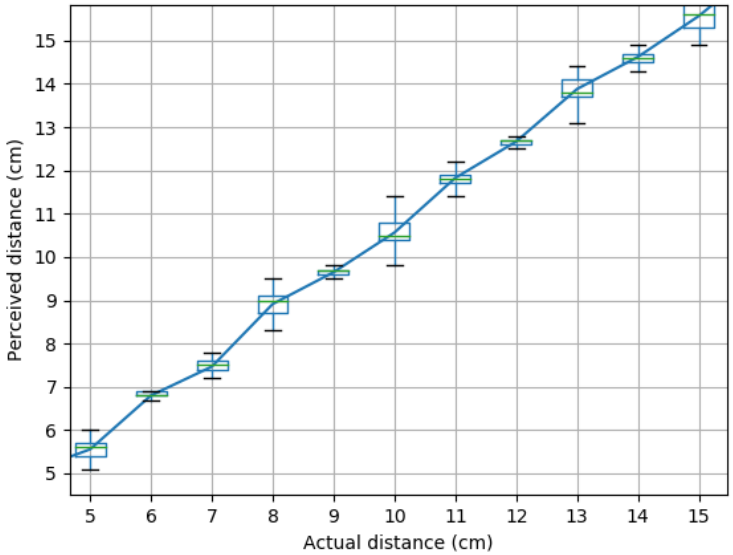


Figure 4.3: Ultrasound response variance

4.4 Encoders

Wheel encoders are required to measure the speed of each wheel independently. This gives the system more control over its path as well as allowing it to perform dead reckoning — meaning an approximate position can be maintained over time — through wheel odometry.

4.4.1 Design

The motors selected had both incremental and absolute encoders available which were suitable, but as only the wheel speed is needed and not the position, the incremental rotary encoders were used. These were hall effect encoders, which measure the changing magnetic field of magnets which are attached to the motor shaft. It has two 6 pole magnets which create 12 counts per revolution of the motor shaft, which, after scaling for the gear ratio of 120:1, results in 1440 counts per revolution of the robot wheel.

The encoders are quadrature encoders, which represent the speed and direction of the wheels with two square waves, where the frequency represents the speed and the leading wave indicates direction.

4.4.2 Implementation

The encoder data was read into the RPi by an encoder class written in Python. This worked by connecting callback methods to both the A and B channel pins. This initialisation is shown in Listing 4.2

```
GPIO.add_event_detect(self.pin$_$a, GPIO.BOTH, callback=self._callback_a)
```

Listing 4.2: encoder callback set-up

This then either increments or decrements a count depending on the direction, as shown in Listing 4.3. Note that `_callback_b()` is identical but with `self._inc()` and `self._dec()` switched.

```
def _callback_a(self, _):
        a, b = GPIO.input(self.pin_a), GPIO.input(self.pin_b)
        if a == b:
            self._inc()
        else:
            self._dec()
```

Listing 4.3: Encoder Callback Function

4.4.3 Testing

The encoders were first tested to ensure correct functionality using a test circuit and connecting the encoder outputs to an oscilloscope. All the encoders were shown to function

correctly outputting square waves in both channels. Channel A led channel B when the motor was moving forwards, and channel B led channel A when going backwards.

An example of the encoder output measured is shown in Figure 4.4 below. This shows the two square waves produced by the two encoders outputs a quarter wavelength out of phase.

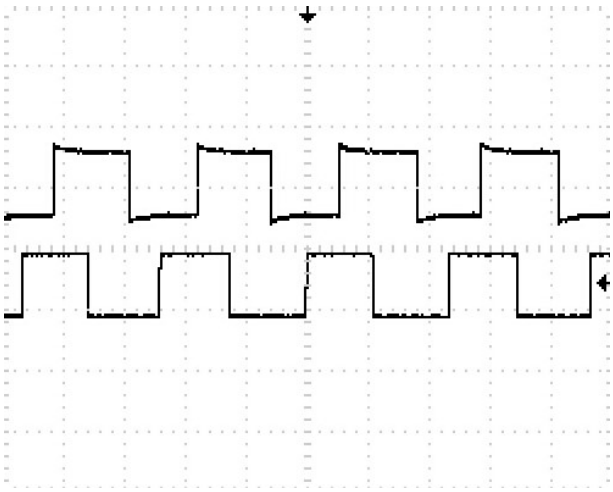


Figure 4.4: Encoder output test waveforms

This experiment allowed us to quickly confirm the correct functionality of all 6 encoders, however did not allow us to test the accuracy. After this, the motors were run at a low speed for several rotations while the system counted pulses. This was not an accurate test of the encoders, as the wheel could not be stopped at an exact integer multiple of rotations, however did verify that the system was approximately outputting 1440 counts per revolution.

4.5 IMU

The IMU consists of a three-axis accelerometer and a three-axis gyroscope which can be used to measure both the linear acceleration and rotational velocity of the robot. This can be integrated to obtain velocity and integrated again to obtain displacement in order to track the robot's position over time, again by performing dead reckoning. It is worth noting that the integration is assuming the behaviour of the system between measurements, which adds to the error within the measurement. However, the system can then perform sensor fusion on this data and the encoder readings to get far more accurate results than is possible with either system independently.

4.5.1 Design

The IMU selected was the MPU-6050 [57]. It was the most accurate sensor of a similar budget with regards to noise, cross axis sensitivity and nonlinearity. Offset tolerance was

also considered but this was less of a concern as this can be compensated for, as long as the offset is not so extreme as to limit the range. It was configured to operate in its smallest range of values ($\pm 250^{\circ}s^{-1}$ and $\pm 2g$ [57]).

The IMU was placed as close to the centre of the axis of the robot as possible, as when it is far from the centre of rotation the centripetal forces acting upon it can interfere and be measured as linear motion away from the centre of the robot.

4.5.2 Implementation

The IMU selected uses the I^2C protocol for data transmission, which integrates well with the RPi as it has dedicated I^2C GPIO pins and existing libraries to facilitate communication.

It should be noted that, while this was the only I^2C device currently used, other devices with conflicting addresses can be handled as the IMU allows the address to be changed through a data pin from 0x68 to 0x69. This would most likely be used if the framework developed was being used on another, more complicated system requiring several IMUs. This pin was not connected to the RPi in this design as it was not yet required.

An `i2c_object` class that used the "smbus" package was created to simplify the communication over the I^2C buses. These were initialised with an I^2C bus and an address, and contained various methods for reading and writing to the I^2C device, such as `read_word()`, shown in Code Listing 4.4.

```
def read_word(self, reg):
    """Read a word from the i2c_object at the register addresses reg and reg+1"""
    h = self.bus.read_byte_data(self.address, reg)
    l = self.bus.read_byte_data(self.address, reg + 1)
    value = (h << 8) + l
    return value
```

Listing 4.4: I2C `read_word()` function

An `IMU` class was then created to read the data from the IMU which has an `i2c_object` instance. Listing 4.5 shows the constructor for the IMU class. Note that the `write_byte()` call is essential to enable the IMU transmissions.

```
def __init__(self, channel=1, address=IMU_ADDRESS):
    ...
    self.address = address
    self.i2c = I2C(self.address, channel=channel)
    self.i2c.write_byte(0x6B, 0x00)  # turns imu on
    self.speed_vect = (0.0, 0.0, 0.0)
```

Listing 4.5: IMU Initialisation Function

As the IMU returns arbitrary values between -32768 and 32767 [57], a constant was required to convert the results to meaningful values. These calculations are shown in Listing 4.6.

```
GYRO_RANGE = 250  # deg/sec
GYRO_DIVISIONS = 32768  # 2 ^15
GYRO_UNITS = (GYRO_RANGE * math.pi * 2.0) / (GYRO_DIVISIONS * 360)  # rad/sec

ACC_RANGE = 2 * 9.81  # m/s^2
ACC_DIVISIONS = 32768  # 2 ^15
ACC_UNITS = ACC_RANGE / ACC_DIVISIONS  # m/s^2
```

Listing 4.6: Calculation for IMU Value to SI unit Conversion Constants

The acceleration values can then be multiplied by the `ACC_UNITS` constant and the result is the acceleration in $\mathrm{m\,s^{-2}}$, and `GYRO_UNITS` can be used similarly to find the rotation in $\mathrm{rad\,s^{-1}}$, as is shown in Listing 4.7.

```
def read_gyro(self):
    x_rot_v = self.i2c.read_signed_word(0x43) * GYRO_UNITS
        # ...
    return -x_rot_v, -y_rot_v, z_rot_v

def read_accel(self):
    x_a = self.i2c.read_signed_word(0x3b) * ACC_UNITS
                # ...
    return -x_a, -y_a, z_a
```

Listing 4.7: Reading and Converting Raw IMU Values

Note that the values returned are negative for the x and y axes. This is because, as will be discussed in Section 4.7.1, the IMU had to be mounted rotated 180° rotated around the z axis relative to the robot frame (i.e. the x axis of the IMU faced the back of the robot).

The final step in reading the IMU results is to integrate the acceleration data as to get the linear velocities. This makes use of the `speed_vect` variable declared in the IMU constructor shown in Listing 4.5. `speed_vect` is incremented by the acceleration multiplied by the IMU rate, as shown in 4.8.

```

def get_speeds(self):
                    # ...
        gyro_speeds = self.read_gyro()
        accel_vect = self.read_accel()
        self.speed_vect = tuple([(1.0 / self.freq) * accel_vect[i] + self.
    speed_vect[i] for i in range(len(accel_vect))])
```

```
7           return self.speed_vect, gyro_speeds
```

Listing 4.8: Integrating Linear Acceleration

4.5.3 Testing

Initial work was performed with the IMU and an Arduino microcontroller before the RPis were acquired. The IMU was connected and various movements were recorded. Python code was then written to interpret and visualise the data as a moving frame. A frame from the output test video is shown in Figure 4.5.

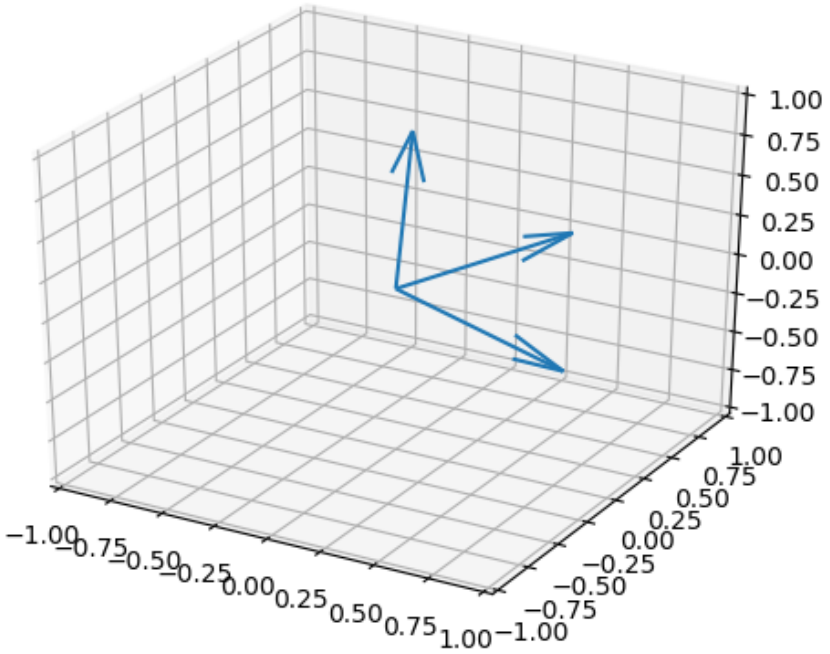


Figure 4.5: IMU visualisation test frame

Various tests were performed using this method testing both the gyroscope and the accelerometer by moving and rotating the IMU. The tests showed reasonable results indicating that the IMU was functioning properly and that the data was being interpreted correctly. Exact measurements of accuracy could not be obtained, however, without a test system for precisely controlling the acceleration and rotations.

4.6 Sensor Placement

The range sensors chosen were ultrasound sensors as these fit the scalability and accuracy requirements of the range sensing. Mechanically these had to be mounted in a consistent manner on each robot to ensure that software results could be replicated between robots with minimal deviation from the standard code. The ultrasonic sensors chosen were HC-SR04 [56].

As can be seen from the datasheet, the sensors have a cone of detection of 30°. This was the main consideration when designing the layout of the sensors at the front of the robot. Due to space restrictions on the front of robot, three sensors were used for an overall cone of detection of approximately 90°. Mitigation of the dead-band was also considered resulting in the placement of the sensor $\approx 30\,\text{mm}$ from the edge of the robot chassis as shown in Figure 4.6.

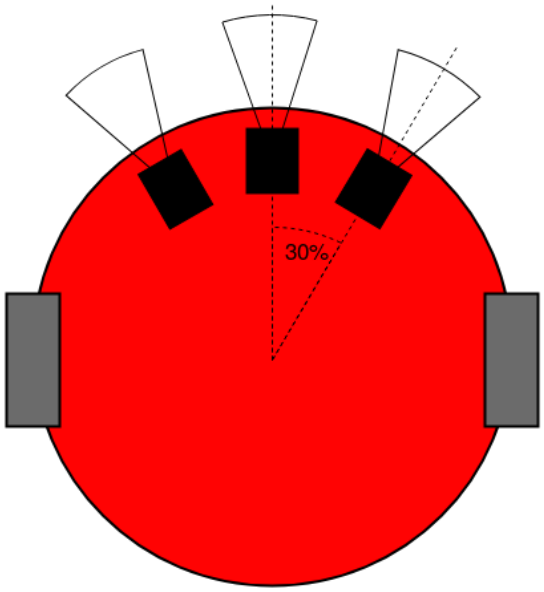


Figure 4.6: Ultrasound sensor layout

The PCB could be designed with these measurements in mind and headers used to ensure the design was modular and sensors could be swapped out if needed.

The IMU had to be placed in the centre/as close to the centre of the robot as possible. This was to ensure the IMU was as close to the centre rotation of the robot as possible, to obtain the most accurate readings possible. By placing the IMU correctly on the robot the need for regular calibration of the measurements is reduced. This heavily influenced the design of the PCB.

4.7 PCB

As precise and consistent placement of components was required to ensure homogeneity of the robots, a PCB was developed for the connection and mounting of the parts. The following describes the rationale and design decisions taken when carrying out the 3-iteration design process for designing the PCB. The final PCB design can be found in the git repository (c.f. Appendix A).

4.7.1 Design

The PCB required to contain the following components:

- Raspberry Pi ribbon cable connector
- Three HC-SR04 ultrasonic sensors
- MPU-6050 IMU
- Left and right encoder connectors
- Motor drive connectors
- Power connectors
- LEDs for debugging

with the following physical design constraints:

- Central ultrasonic sensor at the centre of the robot in the x axis
- Other ultrasonic sensors symmetrical about the x axis
- IMU chip in centre of x and y axes
- Mounting holes for RPi and for mounting PCB to chassis
- Power connectors at fixed position relative to centre to ensure it connects to header on power distribution board
- Motor drive connectors at fixed position relative to centre

As the IMU had to be mounted in the middle of the robot in the x and y axes, and given the motor drive connections to the power distribution board, the IMU had to be positioned backwards, so that the x axis faced towards the back of the robot. This can be compensated for in software, but does need to be considered.

When designing the PCB in the first iteration, an attempt was made to use pins closest to the positions of the components on the robot to achieve the neatest possible design,

and minimise crossing wires and vias. The IMU pins had to be connected to the two I^2C pins of the RPi and, similarly, the motor PWM pins were connected to the two hardware PWM pins of the RPi. The remaining pin choices were made arbitrarily as generic GPIO pins could be used for a variety of purposes.

Some minor changes were made in the connection of the generic GPIO pins in versions 2 and 3 of the PCB to minimise via and path crossing. This can be seen in Table 4.1.

Table 4.1: Pin assignments for iterations of the PCB design

Pin	**Description**	**PCB v1 (Blinky)**	**PCB v2 (Inky)**	**PCB v3 (Clyde)**
GND	Ground	9	9	9
VCC	5 V supply	2	2	2
MLD	Left motor DIR	11	11	11
MLP	Left motor PWM	32	32	32
MRD	Right motor DIR	15	21	19
MRP	Right motor PWM	33	33	33
MS	Motor SLP	13	13	13
ULT	Left ultrasonic TRIG	35	35	35
ULE	Left ultrasonic ECHO	37	37	37
UCT	Centre ultrasonic TRIG	16	16	16
UCE	Centre ultrasonic ECHO	12	12	12
URT	Right ultrasonic TRIG	22	22	22
URE	Right ultrasonic ECHO	18	18	18
ELA	Left encoder A	23	23	21
ELB	Left encoder B	19	19	23
ERA	Right encoder A	24	24	24
ERB	Right encoder B	26	26	26
LG	Green LED	31	31	31
LR	Red LED	29	29	29
SDA	I^2C SDA (IMU)	3	3	3
SCL	I^2C SCL (IMU)	5	5	5

With these constraints the following PCB design shown in Figure 4.7 was created. Note that the peculiar shape is to allow room for the motors and encoders.

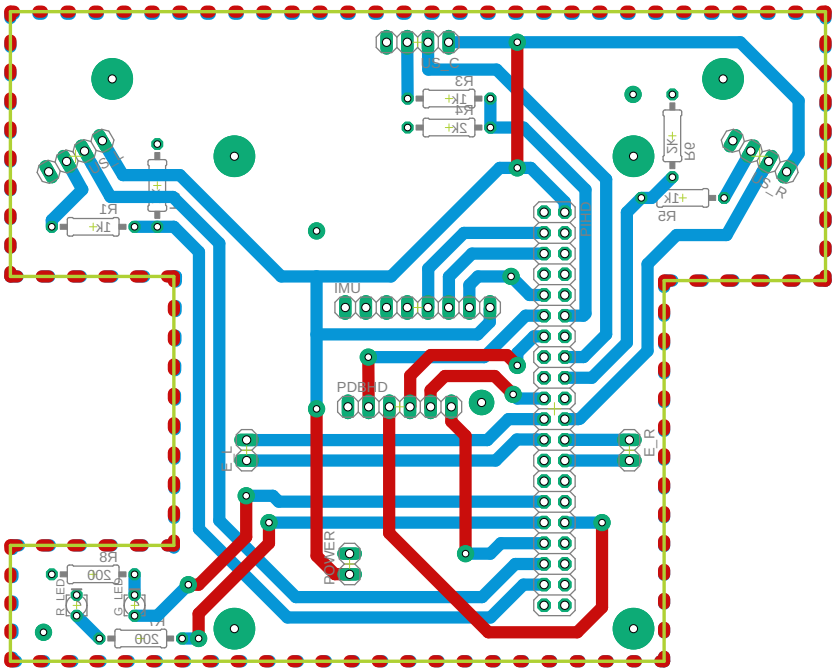


Figure 4.7: Final PCB design

4.7.2 Testing

After the components were soldered into place on the PCB, manual continuity tests were performed with a multimeter to ensure that adjacent pins had not been connected in the soldering process.

The PCB was then mounted and each module test was rerun to ensure that the connections were all correct. This process exposed a flaw as one of the ultrasonic sensors seemed to no longer work. After some investigation it was noticed that the screws being used had slightly larger heads than the screws used for mounting the test strip board and connected two tracks of the PCB. A rubber washer was used as temporary fix and this was corrected in the final PCB design.

Chapter 5

Software

The software architecture was centred on the event-driven implicit implication architectural style [58]. With regards to software architecture, this means modules are signalled to start by other modules and this is propagated through the system with events cascading to trigger other actions within the system.

To implement this architecture, each part of the system can be defined as its own module with a function which triggers its invocation and a function which can broadcast events if required. One way to implement this architecture is to use the publish–subscribe design pattern. In this model, nodes may publish data on communication channels referred to as topics, which triggers all nodes which have subscribed to the topic to take some responding action. This pattern has the advantage that the resulting software has strong support for reuse, as new architecture components can be dynamically added at runtime by subscribing to the appropriate topics. Loose coupling between modules means that maintenance and testing of each module can take place independently and significant changes to the system can also be made without the need for major architectural modifications.

In robotic systems, this data-oriented approach is highly beneficial as each of the sensor and actuator systems can run independently without requiring knowledge of implementation details of other parts of the system. This reduces the risk of system crashes, as module crashes are isolated, increasing robustness.

A true implementation of the event-driven implicit invocation style would have all actions being taken only in the callback responses, with modules being inactive the rest of the time. However, in a dynamic environment such as a robot with many topics being published many times per second, it is inefficient and unsafe to have large amounts of code in the callback functions, which block other operations from taking place. It is therefore often beneficial for nodes to have a polling loop in which they handle computationally expensive functionality, with callbacks only setting up the data required by this loop.

5.1 ROS

In order to implement this architecture, and following extensive research (c.f. Section 2.5), the Robot Operating System (ROS) was selected as a platform. The ROS framework makes it simple to design and implement individual modules as nodes, and uses a central `roscore` control node to manage interactions between nodes, which communicate by publishing and subscribing to topics.

5.1.1 Design

Following the decision to use ROS to implement our chosen architecture, a modular approach was adopted for the design of each of the components within the system. Figure 5.1 shows a system-level block diagram visualising the primary channels of data flow between modules. This structure allows a data-driven approach to module development, making individual modules easily interchangeable due to the high level of decoupling. Sensors, actuators and control nodes can therefore easily be changed at a later date without requiring major changes to the overall software structure.

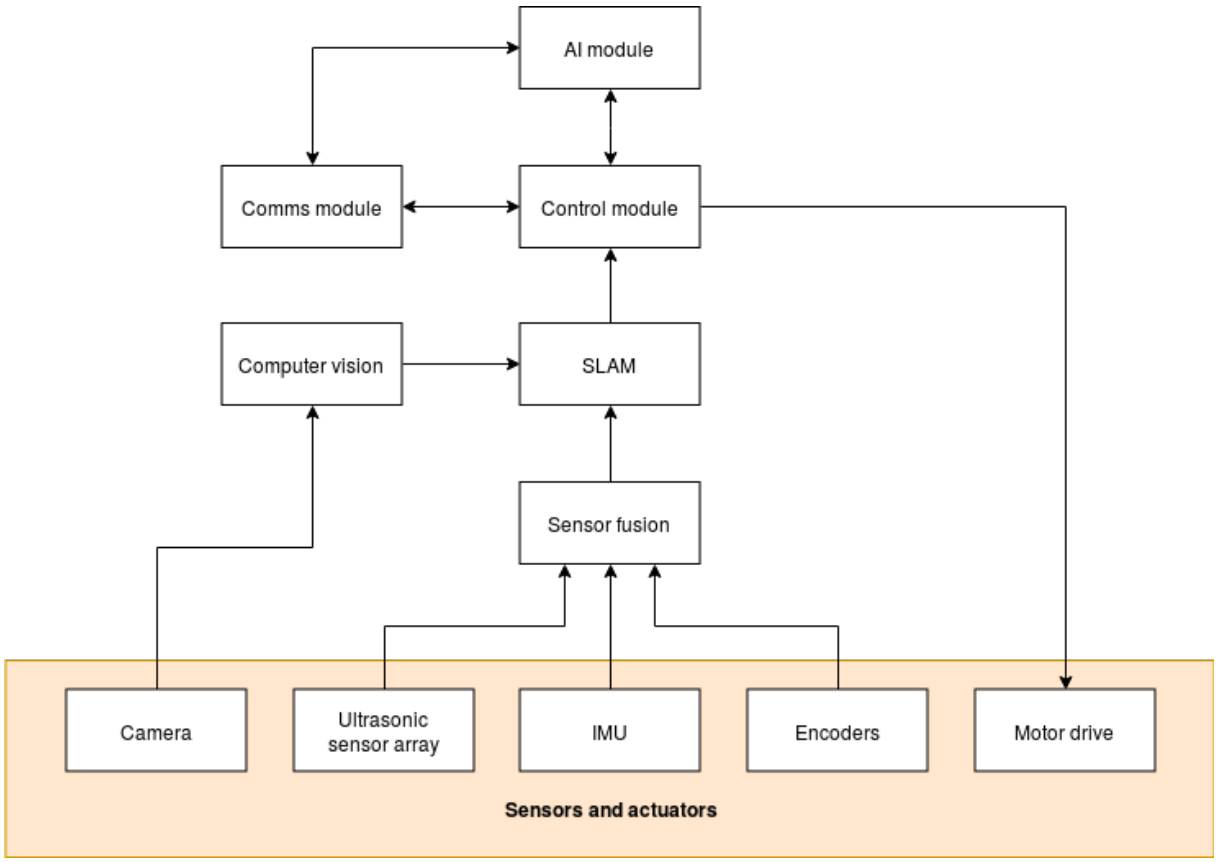


Figure 5.1: High-level software block diagram

ROS provides two client libraries, `rospy` and `roscpp`, which allow programming nodes in Python and C++, respectively. Although the majority of the system will be

programmed in Python, the choice of programming languages means that timing or processing-sensitive operations can be written in a lower-level programming language as required.

Individual nodes can be started from the command line using the `rosrun` tool, or can be included in ROS launch files, which allow multiple nodes to be run simultaneously with any number of parameters. Launch files, which are written in a ROS-specific XML format, enable the creation of multiple configurations of nodes with different parameters, allowing different system setups to be tested easily. Launch files can be run using the `roslaunch package_name file.launch` command.

ROS nodes are created by scripts within ROS packages, which are located in a Catkin workspace. Catkin, a CMake-based build system used to build ROS packages [59], requires a strict directory structure, with each package containing `launch`, `script` and `msg` directories containing launch files, node scripts, and message type definitions, respectively.

5.1.2 Implementation

In order to enable rapid development of the system, ROS packages were primarily written in Python. Four packages specific to the project were created:

- `crues_sensors` for hardware interfaces with sensors and nodes used for processing sensor data and camera imagery
- `crues_actuators` for hardware interfaces with actuators
- `crues_comms` for nodes handling communication between robots
- `crues_control` for control nodes and AI modules, as well as system-level launch and configuration files

In addition, a number of third-party packages were installed to aid with sensor fusion, mapping, and other specific tasks.

Each of the individual modules shown in 5.1 were implemented as a ROS node, generally encapsulated in a single Python script. In order to adhere to best practices and ensure consistency across the code base, each node was represented as class, with an `__init__()` method initialising the node and a `spin()` method which is called to run the node's main loop. In nodes which perform recurring work, such as polling sensors for data, this method uses a `rospy.Rate` object to schedule tasks, whereas nodes that respond to events on subscribed topics simply call `rospy.spin()`, which prevents the script from exiting until a ROS shutdown signal is received.

The `rospy.Publisher` class is used to publish messages of a pre-determined type to a topic with a given name. Similarly, `rospy.Subscriber` is used by nodes wishing to listen for messages on a specific topic. The constructor of this class specifies a callback function,

which receives the message as an argument. By loading parameterised values at startup, the code maintains flexibility, as different values can be used in launch files for different applications.

Listing 5.1 shows an example of a ROS node that publishes at a predefined rate. The rate can be passed to the node as a parameter in the launch file, and in this case defaults to 10 Hz. The data is set by a callback method, which is called by a subscriber to another topic. In general, callback methods were kept very short in order to prevent blocking of the subscriber's thread, with any processor-intensive work executed in the main loop. In this case, the node simply publishes the data received on the subscribed topic unaltered.

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String


class Echo:
    def __init__(self):
        rospy.init_node('node_name')
        self.data = ""
        rospy.Subscriber('input_topic', String, self._callback)
        self.pub = rospy.Publisher('echo_topic', String, queue_size=10)
        self.rate = rospy.Rate(rospy.get_param("~rate", 10))

    def spin(self):
        while not rospy.is_shutdown():
            self.pub.publish(self.data)
            self.rate.sleep()

    def _callback(self, msg):
        self.data = msg.data


if __name__ == '__main__':
    try:
        node = Echo()
        node.spin()
    except rospy.ROSInterruptException:
        pass
```

Listing 5.1: Example ROS node

5.1.3 Software versions

Various third-party software packages were used throughout this project. Below are details and rationale for the choices of software used widely throughout the system; additional packages used for specific modules are detailed in the following sections that concern the respective modules.

Operating system

The Raspberry Pi 3 B+ allows the installation of a number of Linux-based operating systems, of which Raspbian, a derivative of Debian, is the recommended default operating system. Initially Raspbian Stretch was chosen; however, problems were experienced with slow performance and heavy CPU usage. Research suggested that, while ROS officially supports Debian-based systems, performance is significantly optimised in ROS distributions for Ubuntu. For this reason, it was decided to switch to Lubuntu 16.04, a lightweight distribution based on Ubuntu with the LXDE desktop environment. In order to minimise overhead, the desktop environment was disabled. The operating system was installed from an image produced by Ubiquity Robotics (available for download from `https://downloads.ubiquityrobotics.com/pi.html`), which is bundled with various robotics tools, including an installation of ROS Kinetic.

ROS

The selected ROS distribution, ROS Kinetic Kame, was released in 2016. The decision to use this version over more recent releases such as the 2018 Melodic Morenia release was informed by the wide availability of packages for the former distribution, some of which are yet to be released on more recent platforms.

Python

This project currently uses Python 2.7, which will no longer be supported after January 2020 [60]. The reason for this choice was the lack of official support for Python 3.x in ROS Kinetic. While in principle it is possible to use Python 3 with ROS, various compatibility issues may arise [61]. In the interest of stability, it was decided to use the older version. Migration to ROS 2, which is currently under heavy development but will have native support for Python 3, is recommended in future [62].

RPi.GPIO

For hardware interactions with the Raspberry Pi's GPIO pins, RPi.GPIO and WiringPi were considered. RPi.GPIO was selected as it is widely adopted in the Raspberry Pi community, and is used as the basis for a number of other GPIO libraries, such as GPIO

Zero. Over the course of the project, various difficulties were encountered with the library, including problems with the edge detection implementation which resulted in frequent ultrasonic sensor timeouts, and the lack of a hardware PWM interface. Migration to WiringPi, which does not suffer from these problems is suggested in future.

5.1.4 Testing

ROS provides a number of packages for testing and debugging systems. These include `rviz`, which generates spacial visualisations of data published on specific topics, `rqt_plot`, which allows real-time graphing of numerical data, and `rqt_graph`, which displays ROS computation graphs illustrating interactions between nodes. Each of these packages were used extensively in testing both individual nodes and system architectures. Figure 5.2 shows the computation graph of the overall system, generated by `rqt_grpah`, with nodes represented by ellipses and topics shown as rectangles. As can be seen from this graph, the architecture is broadly similar to that of the block diagram of Figure 5.1, with the low level sensor nodes at the base publishing to the more complex nodes, demonstrating the modularity of the system.

In addition to these tools, which were mainly employed on code running on the RPis, a suite of unit tests was written to prevent regressions. In order to allow these tests to run on desktop PCs before deploying to the RPi, the Python `unittest.mock` mocking framework was used to replace ROS functionality that would otherwise require a ROS master node to be running. In addition, a mock version of RPi.GPIO was written in order to allow simulation of hardware interactions in tests. This took the form of a `GPIO_MOCK` Python module, with functions and classes mirroring those in RPi.GPIO. Listing 5.2 shows how to ensure that the correct library is imported depending on the platform the code is running on. This takes advantage of the fact that RPi.GPIO only runs on Raspberry Pis and cannot be installed on desktop computers.

```
1 try:
2     import RPi.GPIO as GPIO
3 except ImportError:
4     from crues_tools import GPIO_MOCK as GPIO
```

Listing 5.2: Import statement for RPi.GPIO library

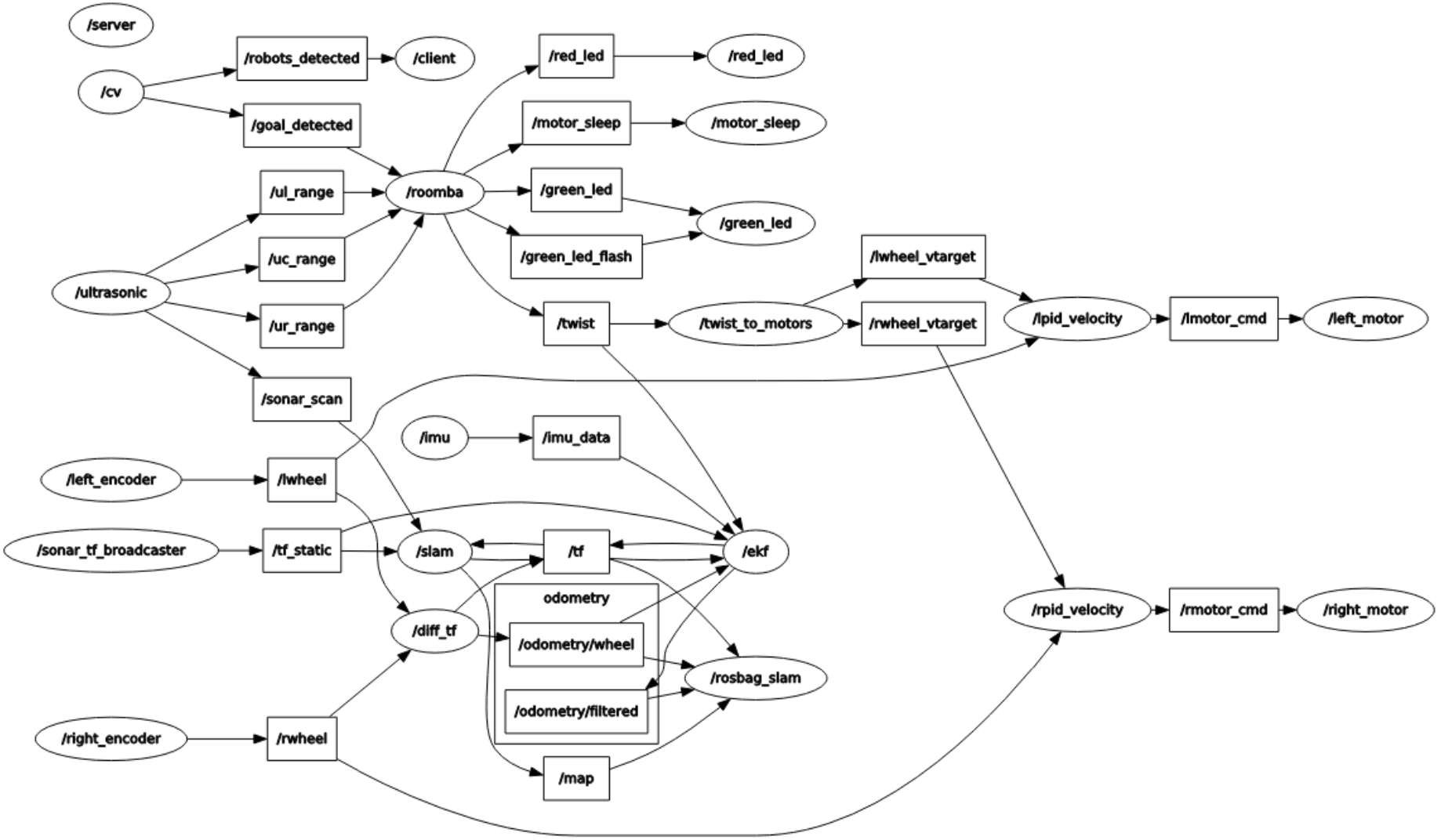


Figure 5.2: ROS computation diagram showing interactions of nodes and topics for overall system

5.1.5 Deployment

Since system testing was closely tied to the hardware of the robot, the ability to rapidly deploy changes to the robot was crucial to the development process. To this end, a shell script (`deploy.sh`) was written to copy any changes to robots connected to the network. The script uses `rsync` to copy any modified source or configuration files in the Catkin workspace, and fetches any output files generated by previous executions of the code. Since the RPis are unable to retain system time between boot cycles, and `rsync` relies on system time to detect file modifications, the script additionally synchronises the system time of the Pi to the host machine. The system time is set using UTC in order to prevent problems caused by time zones and daylight saving time.

The installation of software dependencies proved to be less trivial due to difficulties connecting to the internet and WANET simultaneously. Dependencies were initially installed manually on a single Pi connected to the internet, and were then transferred to the other robots by copying an image of the SD card. This method had the additional benefit of ensuring that all Pis were running the same versions of any dependencies.

5.2 Communication

The communication node's requirements were to allow the robots to be able to send and receive messages to each other. The structure of these messages should allow for any object to be able to be sent regardless of the data types, depth or complexity of the objects. Each robot should also be able to simultaneously listen for incoming messages whilst sending messages to a different robot. With the aim of allowing scalability, the communication system should be able to handle robots listening to multiple clients at the same time as sending messages.

The communication system also requires a network or communication technology to allow all of the robots in the system to be able to communicate with each other. Each of the robots will also require a unique identifier to allow sending messages to specific robots rather than broadcasting all messages to all robots.

5.2.1 Design

In order to implement these requirements, the first decision was the method the robots were going to use to communicate. A few options for this were considered such as Bluetooth and WiFi. After researching and considering these options, the decision was taken to use a Wireless Ad-hoc Network (WANET).

WANET is a wireless network where the nodes can be located anywhere globally and does not require any infrastructure such as a router. The underlying design is such that the nodes believe they are part of a single-hop or multiple-hop wireless network at the physical

layer and the data link layer as part of the MAC sublayer [63]. The wireless channels are often shared and use carrier sense multiple access protocols to handle multiple nodes attempting to use the channel at the same time. This is known as link-level congestion and increases packet service time, decreasing utilization and overall throughput. Although scalability is an important factor, it was deemed a WANET's handling of congestion was sufficient given the anticipated level of communication across the system at any time would be low.

Each node in the WANET is assigned a local IP address. Anything that connects to the network can then send messages through the network provided it knows this address. As a result, any given robot needs a method of finding this address to send messages to other robots. The simplest method for this is to use a lookup table that maps the robot's name to their fixed IP in the network, allowing all nodes in the network access to this.

After deciding to use WANET, a decision had to be made in how to format the data to send over the network. A few options were available such as using JSON, XML or a custom-structure. It was decided the best option was to use JavaScript Object Notation (JSON) objects after consultation with Dr Irvine as discussed in Section 3.1.1. JSON is a standard data-interchange format that is easy for humans to read and write as it uses attribute-value pairs and array data types.

After considering various options for the structure of the communication process, a client-server system was decided upon. The client is the node responsible for sending the messages whilst the server is responsible for listening for incoming messages. As these are separate nodes, they will work independently of each other and will allow the robot to send and receive messages simultaneously.

5.2.2 Initial Implementation

To implement these requirements a `comms.py` file was created which contained the function definitions and address lookup table that the server and client nodes would use to communicate. Firstly, a Python dictionary of the robot's IP addresses and a laptop's IP address for testing purposes was created. A `lookupIP(name)` function and a `lookupname(ip)` function were created for modularity and maintainability. Secondly, functions were required that could create the JSON message (`tojson()`) and extract the data from the message (`fromjson()`).

```
1
2 def tojson(host_ip, hostname, data):
3     # receiver ip and hostname, our ip and hostname, classname
4     sender_ip = socket.gethostbyname(socket.gethostname())
5     header = [host_ip, hostname, sender_ip,
6     Address_lookup.lookupname(sender_ip), type(data).__name__]
7     content = [header, data]
```

```
    message = json.dump(content, separators=(',', ':'))
    return message

def fromjson(packet):
    message = json.loads(packet)
    try:
        target = message[0][0]
        if target == socket.gethostbyname(socket.gethostname()):
            return message[1]
        else:
            return -1
    except IndexError:
        return -2
```

Listing 5.3: JSON Conversion Functions

The header created includes the IP address and name of both the sender and the receiver as well as the type of Python object which the data is. The header and the data to be sent are converted to JSON and returned as a single string. When converting from JSON, the receiver checks that it was the intended recipient of the message, returning -1 if it was not, before returning the data object located at index 1 (header is index 0).

In order to physically send and receive the data the Python Socket library is used [64]. This interface's `socket()` function returns a Socket object whose methods implement the Unix socket system calls.

```
# data is python object to send
def send(hostname, data):
    host_ip = lookupip(hostname) # The remote host
    port_number = 8001 # The same port as used by the server
    message = tojson(host_ip, hostname, data)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    if not s.connect_ex((host_ip, port_number)):
        s.sendall(message) # Sends string (of JSON)
        s.close()
```

Listing 5.4: send() Function

Listing 5.4 shows the `send()` function, which looks up the IP address of the destination and gets the JSON object of the message. It then attempts to connect using the `connect_ex()` which returns 0 if successful. If so, it can then use the Socket methods to send the data and then close the connection.

When listening for incoming messages, the created Socket object must bind to an IP address and port number. The address `0.0.0.0` allows the node to listen to any incoming messages. Listing 5.5 shows the `listen()` function which starts the node listening whilst

setting the number of queued connections allowed (set here to 2 but can be increased when scaling the system to use more agents). Once the socket accepts a connection, packets received can continually be stored until no more data is received, after which the connection can be closed and the data read from the JSON strings and returned.

```
def listen():
    PORT = 8001 # Arbitrary non-privileged port
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('0.0.0.0', PORT))
    s.listen(2)
    conn, addr = s.accept()
    packets = ''
    while 1:
        packet = conn.recv(1024)
        if not packet:
            break
        packets += packet
    conn.close()
    message = fromjson(packets)
    return message
```

Listing 5.5: `listen()` function

5.2.3 Initial Testing

This implementation was set up and initially tested between a laptop and a RPi by remotely connecting to the RPi. When this succeeded in sending multiple messages between each device, a further RPi was remotely connected to and an attempt was made to send messages between RPis. Again this was successful, demonstrating the basic functionality of the communication system.

During more extensive testing, an important issue with the previous code was found. When nodes were sending messages to each other simultaneously at least one of the nodes would fail. For example, if robot A sends a message to robot B but before the message has finished sending, robot B sends a message to robot A; then the receiver of robot B crashes and robot A never receives the message. If robot B were to send to a third robot, C, then robot B's receiver would still fail as would one of either robot B or C.

As a result of this bug, the implementation had to be changed to handle multiple connections properly. Various potential solutions were discussed, such as using multiple ports — so as not to kill the socket connection when attempting to connect again — or to use a multi-connection selector as part of the server to handle multiple requests [65]. Alternatively, if the server can always be listening for incoming messages by utilising multiple threads then there would be no system downtime and thus, each message would

be successfully sent. As a result, a multi-threaded version of the existing code was implemented.

5.2.4 Multi-Threaded Implementation

A multi-threaded implementation was created by defining two classes, namely: `ThreadedTCPServer` and `ThreadedTCPRequestHandler`, to be used by the listener as shown in SocketServer documentation [64].

```
class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):
    def messageHandler(self, x):
        print x

    def handle(self):
        data = self.request.recv(1024)
        message = fromjson(data)
        self.messageHandler(message)

class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass
```

Listing 5.6: ThreadedTCPRequestHandler

The `listen()` function was changed to take in a handler parameter. This was passed a function which is used instead of the `messageHandler()` function in the `ThreadedTCPRequetHandler` before it is used to create a `ThreadedTCPServer` object as shown in Listing 5.7.

```
def listen(handler):
    ThreadedTCPRequestHandler.messageHandler = handler
    HOST, PORT = "0.0.0.0", 8001
    server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
    ip, port = server.server_address
    # Start a thread with the server -- that thread will then start one
    # more thread for each request
    server_thread = threading.Thread(target=server.serve_forever)
    # Exit the server thread when the main thread terminates
    server_thread.daemon = True
    server_thread.start()
    return server
```

Listing 5.7: listen() Function 2.0

The `handler` object passed is a function that will publish the message contents so relevant subscribers can use the data received from the other robot. As a new thread is being created for each request, node crashes from interruptions no longer occur.

5.2.5 Multi-Threaded Testing

In order to test this version, it was important to first ensure that the previously successful tests remained successful. Therefore, the same initial tests were applied to this implementation and were successful as expected. To test whether the new implementation was thread-safe, a number of tests were executed.

The first test involved creating a loop that would send 100 messages from robot A to robot B. When this succeeded without any node crashes, the test was repeated but with robot B sending messages to robot A at the same time. This test would have previously crashed each of the robot's nodes but, with each new connection request handled in a different thread, the test was successful. The test was then repeated once more with robot B sending messages to robot C instead of robot A, again resulting in a successful test with no messages lost.

Finally, a stress test was carried out. The previous test was executed again with the limit on the number of messages being sent removed, and so the node would, in theory, continue to create new threads to listen to the incoming messages. Initially, this worked, however, due to there being no delay between sent messages, thousands of threads were created rapidly. This resulted in the node crashing as the thread limit had been reached, however, this happened after over 50,000 messages. As a result, this was determined to be the upper limit of the system, which is excessive for both this project's, and any reasonable purposes, hence, it was determined that the system was sufficiently thread-safe to be used.

5.3 PID Controller

A PID controller was used for each wheel to ensure a constant output velocity at the target specified by the control node. The PID controller is a feedback loop designed to eliminate errors in the actuator systems, ensuring that the motors correct for any outside forces such as friction that act on the motor. By tuning the gain terms K_p, K_i and K_d for the system in question, a target velocity can quickly be approached without overshooting.

5.3.1 Design

The PID control node from the ROS `differential_drive` package was used [66]. The node subscribes to a `/wheel_vtarget` topic which sets the target velocity of a wheel and a `/wheel` topic which provides the actual current speed of the wheel as determined by the

encoders, and publishes a command value to `/motor_cmd`, which is subscribed to by the corresponding motor drive node. The command is a signed float which is used by the drive node to determine direction and PWM values. Listing 5.8 shows the XML used to create PID nodes for the left and right wheels. The parameters `pid/Kp`, `pid/Ki` and `pid/Kd` are specified in the CRUES configuration file (`crues_control/config/params.yaml`). The publishing rate of the PID nodes was set to the rate of the encoders to avoid unnecessary computation.

```
<node name="lpid_velocity" pkg="differential_drive" type="pid_velocity.py">
    <param name="out_min" type="double" value="-0.5" />
    <param name="out_max" type="double" value="0.5" />
    <param name="rolling_pts" type="int" value="2" />
    <remap from="~Kp" to="pid/Kp" />
    <remap from="~Ki" to="pid/Ki" />
    <remap from="~Kd" to="pid/Kd" />
    <remap from="~rate" to="encoder/rate" />
    <remap from="/wheel" to="/lwheel" />
    <remap from="/motor_cmd" to="/lmotor_cmd" />
    <remap from="/wheel_vtarget" to="/lwheel_vtarget" />
    <remap from="/wheel_vel" to="/lwheel_vel" />
</node>

<node name="rpid_velocity" pkg="differential_drive" type="pid_velocity.py">
    <param name="out_min" type="double" value="-0.5" />
    <param name="out_max" type="double" value="0.5" />
    <param name="rolling_pts" type="int" value="2" />
    <remap from="~Kp" to="pid/Kp" />
    <remap from="~Ki" to="pid/Ki" />
    <remap from="~Kd" to="pid/Kd" />
    <remap from="~rate" to="encoder/rate" />
    <remap from="/wheel" to="/rwheel" />
    <remap from="/motor_cmd" to="/rmotor_cmd" />
    <remap from="/wheel_vtarget" to="/rwheel_vtarget" />
    <remap from="/wheel_vel" to="/rwheel_vel" />
</node>
```

Listing 5.8: PID nodes in ROS launch file

In addition to the PID node, the library has a `wheel-loopback` simulator node, which simulates the movement of wheels. This node was used in order to evaluate PID tuning methodologies ahead of the slower process of tuning the actual values on the physical robot. The various methods discussed in Section 2.2.1 were attempted and outcomes recorded.

5.3.2 Ziegler-Nichols

As most of the research carried out pointed towards the Ziegler-Nichols method as being an effective way of tuning PID, this was first tried directly on the robot. In order to tune the PID in real time, the values of K_p, K_i and K_d were published to the ROS system from a terminal while `roscore` was running. K_p was increased until oscillations were seen in the system — the determination of an oscillation was vaguely defined across literature and hence this was done by eye — and this was set as the critical gain K_u of the system. The period of oscillation T_u was then measured. From these values, a tuning table [67] of tuning rules was used to calculate values for the gains. The results of this process are shown in Table 5.1.

Table 5.1: Ziegler-Nichols PID tuning

Method	K_u	T_u [s]	K_p	K_i	K_d
ZN	775	0.14	465	6642	8.13
ZN	800	0.14	480	6857	8.4
NO-OV	775	0.14	155	2214	7.49
NO-OV	800	0.14	160	2240	7.47

In practice, these values were found to be inaccurate and produce an uneven stop-start motion in the movement of the robot. This was due to these methods providing an aggressive gain and overshoot which resulted in the motors being run at high speed followed quickly by low speed to average to the correct speed. Due to the high current drawn by the motors when stopping and starting, this additionally had implications on the power consumption of the robots, with batteries being drained rapidly. The NO-OV tuning rules, which are intended to limit the amount of overshoot, mitigated this effect slightly, but still did not result in accurate velocities.

5.3.3 Manual Tuning

Due to the problems outlined above, the library simulation was used to find a usable tuning method. A more practical solution was found [68], which approached the problem without calculating values. Using the simulation and output graphs obtained from the ROS `rqt_plot` package to determine correct behaviour, K_p was increased until the system oscillated with a constant amplitude. K_p was then halved and K_i introduced to increase the rate of change in the system. K_d could then be slowly increased to minimise the overshoot in the system.

Additional improvements were achieved by scaling the output of the PID loop in the drive node. As the PWM values used by `RPi.GPIO` are in the range $[0, 100]$ and

the velocities published to `/wheel` are in $\mathrm{m\,s^{-1}}$ (with a typical range of $[0, 0.5]$), high gain values are required to ensure the output is on the correct order of magnitude. A scale factor and an offset were therefore introduced in the motor drive node as shown in Listing 5.9 in order to map inputs from an arbitrary input range to the range $[-100, 100]$. Using -0.5 for `~range_min` and 0.5 for `~range_max` resulted in the lower gain values shown in Table 5.2, and appeared to result in a slight reduction in overshoot.

```
class Motor:
    def __init__(self):
        # ...
        self.range_min = rospy.get_param('~range_min', -1.0)
        self.range_max = rospy.get_param('~range_max', 1.0)
        self.scale_factor = 200.0 / (self.range_max - self.range_min)
        self.offset = (self.range_min + self.range_max) / 2
        # ...

    def _tick(self):
        # ...
        pwm_val = (self.cmd - self.offset) * self.scale_factor
        # ...
```

Listing 5.9: Output scale factor in drive node

Table 5.2: Manual PID tuning results

Method	K_p	K_i	K_d
Before PID scaling	102.5	2.5	0.88
After PID scaling	0.2775	0.0075	0.007

5.3.4 Rate control

These values resulted in far less aggressive motion in the movement of the robot. Despite this, velocity spikes still occurred, and following a consultation with Dr Gordon Dobie (c.f. Section 3.1.1) a rate control implementation was used.

Opposed to traditional PID, Rate Control employs only the K_p and K_d portions of the equation and uses the current velocity as the base rate as opposed to 0 [69]. This allows the controller to maintain the speed more smoothly and fewer spikes in motor current are needed for the required drive [70]. By removing the K_i term, the speed will climb more slowly from start-up and recover from dips in speed more slowly. This is mitigated, however, by using the current speed as the base and therefore, a high gain is not required to increase the speed to the required level.

Implementing this involved modifying the source code for the `differential_drive` package to add an adjustment term to the previous output value. This eliminated the need for steady-state correction, allowing K_i to be set to 0. Tuning the remaining values gave a proportional gain $K_p = 0.176$ and a differential gain $K_d = 0.008$. This was tested and resulted in significantly less aggressive motion of the robot and an increased accuracy in output velocities.

5.4 Odometry and Sensor Fusion

5.4.1 Wheel Odometry

Wheel Odometry is derived in two parts. Firstly, as discussed in Section 5.3, the velocity of each wheel is derived from the output of the encoders. Then, with only these values and the distance between the wheels (usually called the track of the wheels), the linear and angular velocities in two dimensions can be derived. Furthermore, by tracking both of these values over time, an estimate of global displacement can be attained. However, due to the lack of reliability in the encoder data, this estimate is very likely to drift over time.

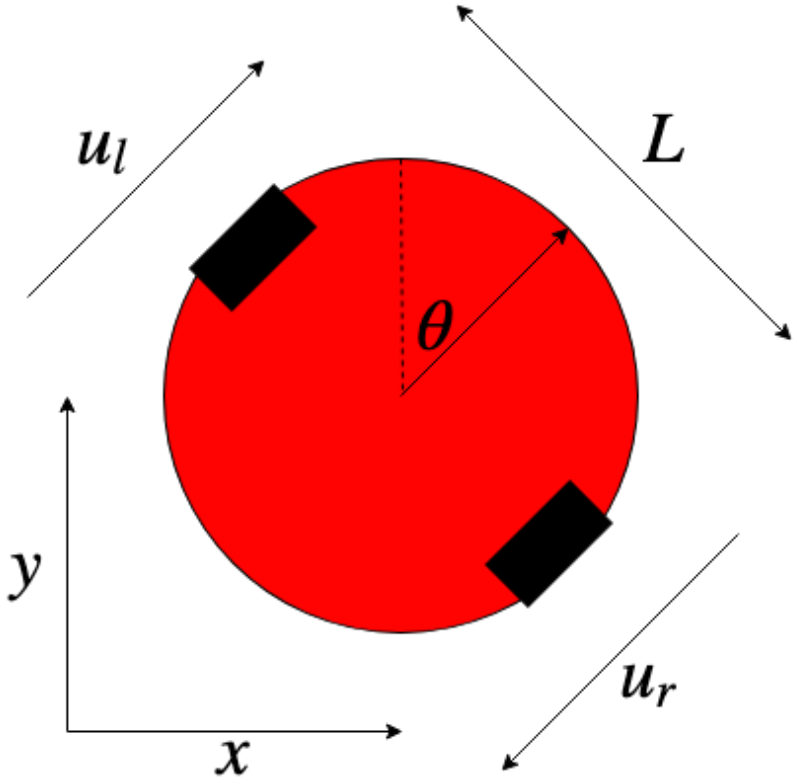


Figure 5.3: The variables used to calculate wheel odometry

Let u_l, u_r and L be the velocities of the left and right wheels, and the wheel track respectively, while (x, y) and θ are the current displacement and rotation from the origin. These values are reflected in Figure 5.3.

This makes derivation of equations for the odometry fairly simple when considering two "states" of movement: moving in a straight line as a result of u_l being equal to u_r; and rotating on the spot as a result of u_l being the inverse of u_r. This makes derivation of equations for the odometry fairly simple.

When moving in a straight line, the linear velocity of the system is simply the velocity of either wheel, in the direction θ, while rotational velocity should be 0. To achieve the latter, the velocity used in the calculation is $0.5(u_l + u_r)$, which is 0 in the case where $u_l = -u_r$ and unchanged otherwise. Splitting the velocities across the axes with the cosine and sine of θ achieves a reasonable result for these velocities.

Likewise, in the case of rotation, the linear velocities should be 0. When the robot rotates at some rate, say ϕ, around its centre, the wheel must move at $u_r = 0.5L\phi$ ($u_l = -u_r$ as the axis of rotation is the centre of the robot). Like in the previous case, the velocity used in the calculation is to be 0 in the case where $u_l = u_r$, so the value used is $0.5(u_r - u_l)$, which is 0 when the velocities are the same, and u_r when they are opposite, preserving the direction of rotation.

The equations derived here are shown in Equations 5.1 to 5.3.

$$\dot{x} = \frac{(u_l + u_r)}{2}\cos(\theta) \tag{5.1}$$

$$\dot{y} = \frac{(u_l + u_r)}{2}\sin(\theta) \tag{5.2}$$

$$\dot{\theta} = \frac{(u_r - u_l)}{L} \tag{5.3}$$

The `differential_drive` has three major components, `twist_to_motors`, `pid_velocity` and `diff_tf`. The first takes in a ROS twist message type, a six dimensional vector of desired velocities in and about three axes ($\dot{x}$, $\dot{y}$, $\dot{z}$, $\dot{roll}$, $\dot{pitch}$ and $\dot{yaw}$). This twist message is the desired velocity of the robot. Since the differential drive system is operating in only two dimensions, the z, *roll* and *pitch* instructions are ignored. This node inverts Equations 5.1 to 5.3 and publishes the desired u_l and u_r values to respective instances of `pid_velocity` nodes as the topics `/lwheel_target` and `/rwheel_target`, which perform PID on each wheel as discussed in Section 5.3.

The final node provided by the package, and the most relevant to odometry, is `diff_tf`, which subscribes to the encoder data of both wheels and performs Equations 5.1 to 5.3 to generate odometry data. This is then published both as an odometry message for the sensor fusion, and as part of the odometry transformation frame.

5.4.2 IMU

As was discussed in Section 4.5, the IMU can also be used to perform dead reckoning, and this can be fused with the wheel odometry to improve the accuracy of the system.

For other nodes to use the IMU data, the readings had to be converted into a ROS IMU message [71, sensor_msgs/Imu.msg] consisting of: a ROS header message [71, std_msgs/Header.msg]; a quaternion representing the orientation; a 3-element vector for the rotational speed; a 3-element vector for the linear acceleration; and covariance matrices for each of these measurements. Listing 5.10 shows the portion of `imu_node.py` used to create IMU messages.

```
def _publish_data(self):
    # ...
    msg = Imu()
    msg.header.frame_id = self.frame_id
    # Indicate orientation unknown:
    msg.orientation_covariance = [-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    msg.linear_acceleration = Vector3(acc[0], acc[1], acc[2])
    msg.angular_velocity = Vector3(gyro[0], gyro[1], gyro[2])
    # ...
```

Listing 5.10: `_publish_data()` in `imu_node.py`

It is important to notice that the covariance matrix for the orientation is set to a -1 followed by 0's. This indicates that values for the orientation are unknown as we are only using a 6-DOF IMU [71, sensor_msgs/Imu.msg].

The ROS spin functionality was used for the IMU, so `_publish_data()` is run at a set frequency. This frequency, along with the topic the IMU data is published to, is set in the launch file, as shown in Listing 5.11.

```
<node name="imu" pkg="crues_sensors" type="imu_node.py">
    <remap from="~rate" to="imu/rate" />
</node>
```

Listing 5.11: IMU node in ROS launch file

IMU Testing

To test the IMU integration with ROS, the ROS IMU visualisation software was used. The robot was then rotated in each axis, and the output visualised. Figure 5.4 shows the IMU RVIZ visualisation alongside the corresponding position of the robot. Note that this only a small selection of the test data, but all results appeared to accurately reflect the movements performed.

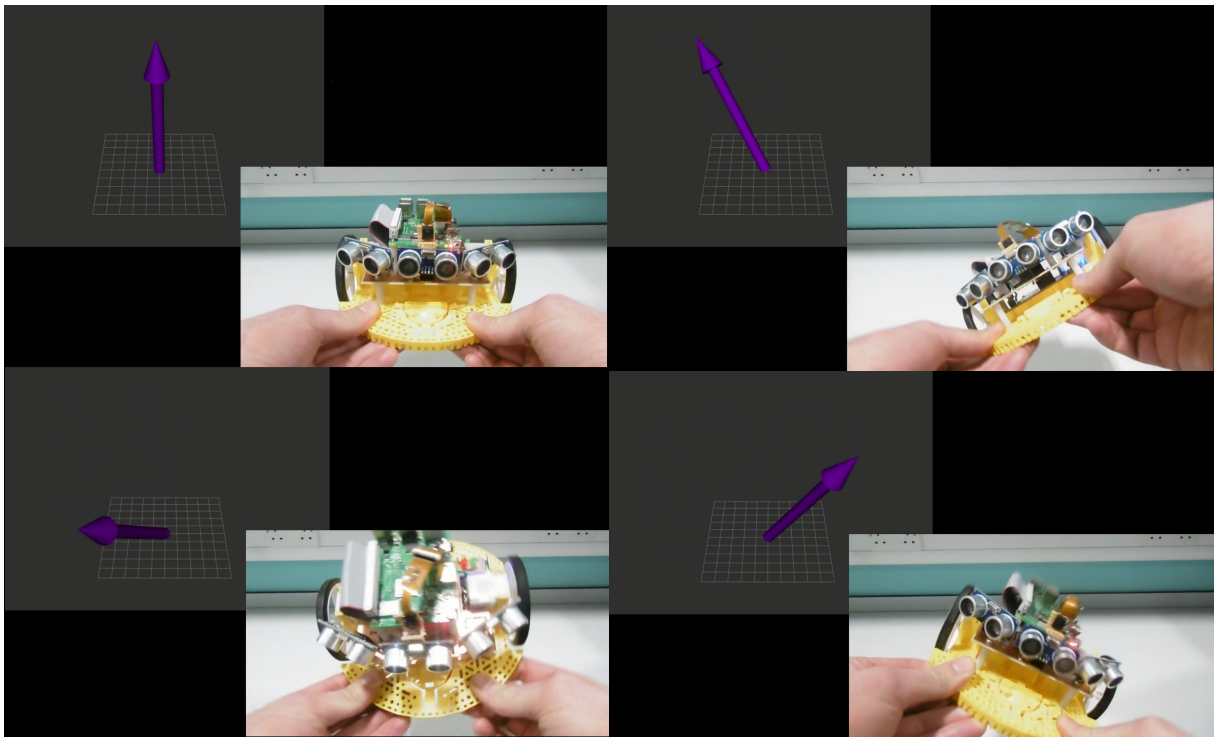


Figure 5.4: IMU RVIZ test

5.4.3 EKF

The ROS `robot_localization` package was used to fuse the sensor data [72]. This package contains implementations of an EKF and a UKF (Unscented Kalman Filter). The latter has the advantage of higher accuracy, particularly for highly non-linear transformations, at the expense of greater computational complexity [73]. In order to limit computational overhead, the EKF implementation (`ekf_localization_node`) was chosen for use with this project. Listings 5.12 shows how an EKF node is created in the main launch file.

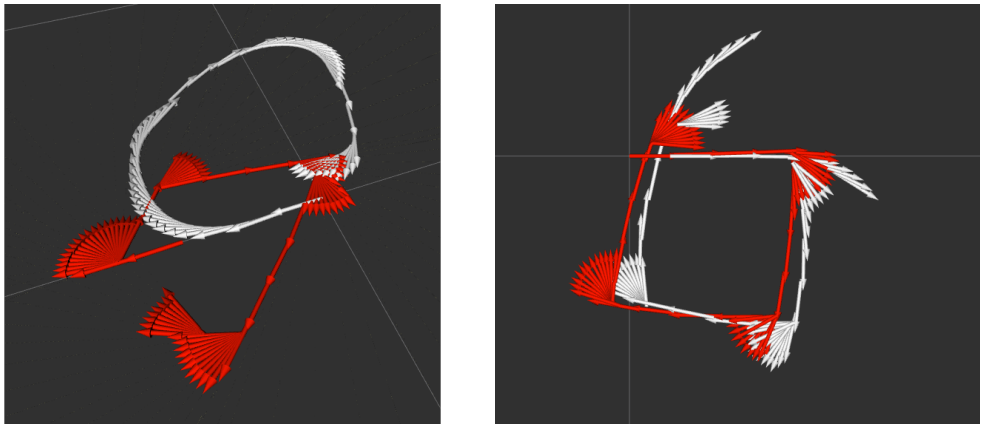
```
<node name="ekf" pkg="robot_localization" type="ekf_localization_node"
      clear_params="true">
    <rosparam command="load" file="$(find crues_control)/config/ekf.yaml" />
    <remap from="/cmd_vel" to="/twist" />
</node>
```

Listing 5.12: EKF node in ROS launch file

The parameters used in the EKF are loaded from the `ekf.yaml` file, which specifies topics and configurations for each of the input sources. The `source_config` parameter for each source (i.e. wheel odometry and IMU data) specifies which of the filter's 15 states the source updates. In the case of odometry, the differential drive restrictions outlined in Section 5.4.1 mean that only the velocity in the X direction relative to the robot's `base_link` frame and the yaw velocity (rotational velocity around the Z axis) can be

updated. The IMU provides values for the yaw velocity and acceleration in the X and Y directions.

Several parameters had to be tuned in order to improve the behaviour of the filter. In particular, the acceleration and deceleration limits (Listing 5.13 ll. 27–28) had to be significantly increased from the default values, as small, light-weight robots are capable of much more abrupt changes in velocity. Figure 5.5a shows the path calculated by the EKF prior to this change, with obvious improvements shown in Figure 5.5b.



(a) Before optimising parameters (b) After optimising parameters

Figure 5.5: Odometry output (wheel odometry in red, filtered odometry in white)

```
frequency: 50  # Frequency at which odometry is published in Hz
two_d_mode: true  # Ignore movement in three dimensions
publish_tf: true

odom0: odometry/wheel  # Topic on which wheel odometry is published
# Which values are provided by wheel odometry
# [x, y, z, roll, pitch, yaw, vx, vy, vz, vroll, vpitch, vyaw, ax, ay, az]
odom0_config: [false, false, false, false, false, false, true, false, false,
    false, false, true, false, false, false]
odom0_queue_size: 2
odom0_pose_rejection_threshold: 5
odom0_twist_rejection_threshold: 1

imu0: imu/data  # Topic on which IMU data is published
imu0_config: [false, false, false, false, false, false, false, false, false,
    false, false, true, true,  false, false]
imu0_queue_size: 5
imu0_pose_rejection_threshold: 0.8
imu0_twist_rejection_threshold: 0.8
imu0_linear_acceleration_rejection_threshold: 0.8
# True iff IMU returns acceleration without gravitational offset
```

```
imu0_remove_gravitational_acceleration: true

use_control: true
stamped_control: false
control_timeout: 0.2
# Which velocities are being controlled [vx, vy, vz, vroll, vpitch, vyaw]
control_config: [true, false, false, false, false, true]
acceleration_limits: [13, 0.0, 0.0, 0.0, 0.0, 34]
deceleration_limits: [13, 0.0, 0.0, 0.0, 0.0, 45]
acceleration_gains: [0.8, 0.0, 0.0, 0.0, 0.0, 0.9]
deceleration_gains: [1.0, 0.0, 0.0, 0.0, 0.0, 1.0]
```

Listing 5.13: EKF YAML file

5.5 SLAM

As discussed in Section 2.3, Simultaneous Localisation and Mapping (SLAM) is an extensively researched and challenging topic. As a result of this, there are many existing implementations of a wide variety of SLAM algorithms. The most commonly used package for this within the ROS framework is `gmapping`, an implementation of a particle filter, which is very well supported by the framework, simply taking and producing standard ROS message types. It also provides a very rich suite of settings which can be used to modify the functionality of the algorithm. The drawback of the package, and the wide variety of settings available in particular, is the somewhat poor documentation it provides, which necessitated a great deal of experimentation to achieve reasonable results, which will be discussed going forward.

5.5.1 Package Overview

The `gmapping` package takes in a `LaserScan` message, as well as odometry data in the form of a transformation frame (`tf`). The odometry data is already produced by the sensor fusion module, so the input parameter of `gmapping` simply needs to be bound the same topic as the output of the EKF.

The scan data is more complex, as the package is intended for use with a LIDAR unit. The outputs of different methods of active range finding are similar enough, however, that a rudimentary laser scan message could be created simply by combining the three ultrasonic readings into a approximation of a LIDAR output. This will likely result in features in the map taking on rounded edges, as the wide range of each sensor will allow it to find the walls of a corner earlier than the point.

The `gmapping` package, while providing some localisation via feature detection, and utilising the odometry in creation of the map, does not actually perform true localisation. Another package, `AMCL`, performs a second iteration particle filtering algorithm on the map already processed by the mapping package, as well as the scan data and odometry, and makes probabilistic approximations of the robot's location in the map. This is mainly used to compensate for odometry drift, as mentioned in Section 5.4.1. A block diagram indicating this relationship can be found in Figure 5.6.

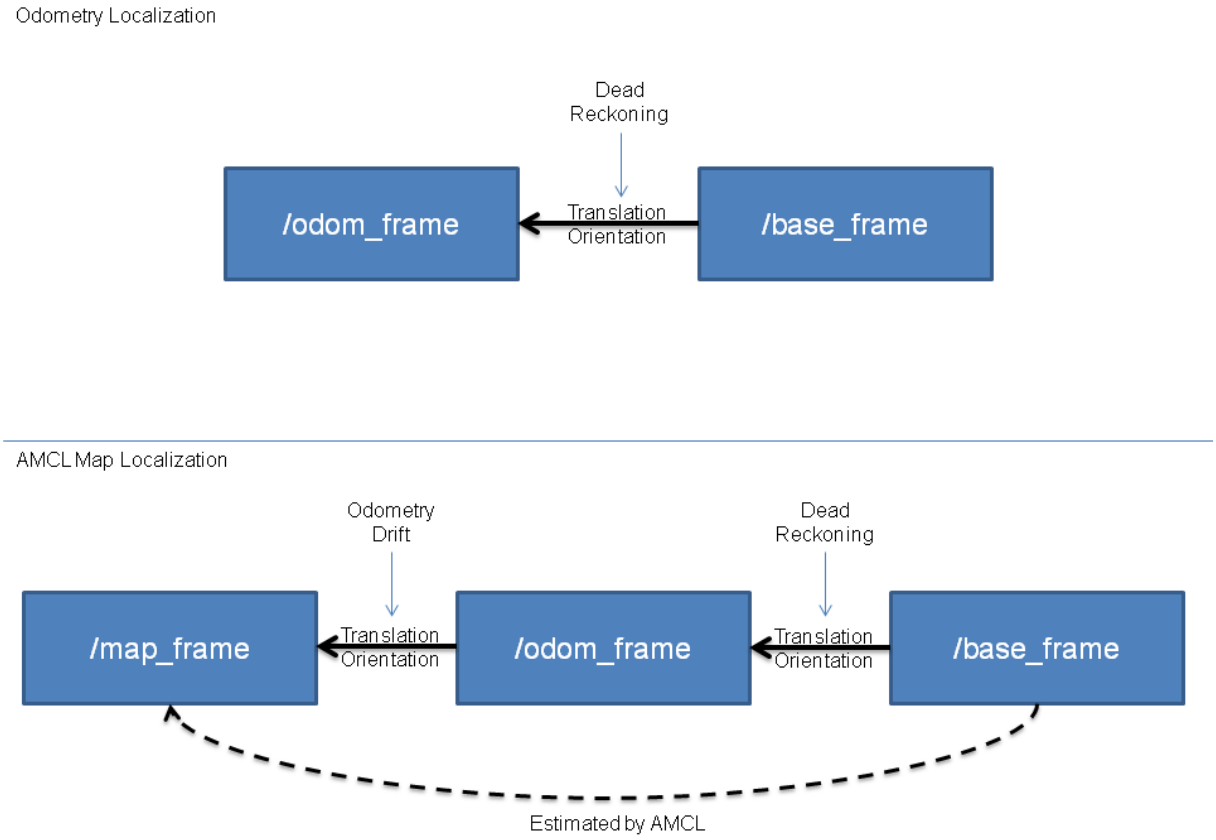


Figure 5.6: AMCL transform diagram [74]

Unfortunately, as particle filtering is an extremely taxing operation, it proved too taxing to run both `gmapping` and `AMCL` at once on the RPi. Indeed, running only the mapping package proved problematic, as certain configurations caused the entirety of the RPi's memory to be used, and led to some other ros packages crashing as a result of being refused memory allocation. As a result of this, two parameters, "particle" and "delta" which respectively govern the number of steps in the filter and granularity of the generated map both had to be reduced. Both of these will reduce the accuracy of the final output, but this appears to be an unavoidable problem.

5.5.2 Tuning

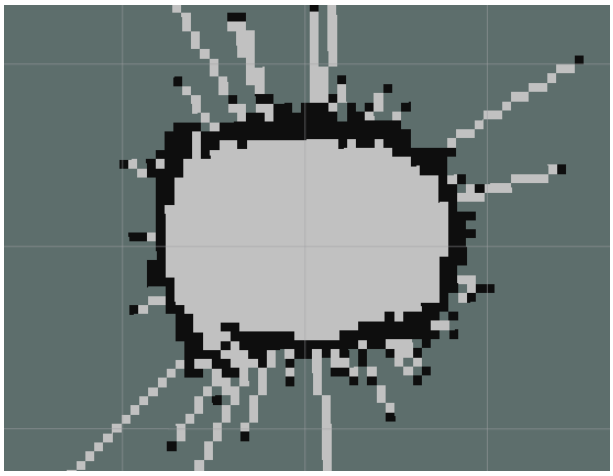
The initial attempts at running `gmapping` presented a number of problems immediately. For one, the map generated was at entirely the wrong scale compared to the odometry data, which created a very chaotic and meaningless map. When displayed alongside the odometry using RViz, the cause of the issue became obvious, that the two values were on different scales. This proved to be a result of a misconfguration, with the odometry returning data in metres and the ultrasonics in millimetres. After fixing this, the map and odometry were on the same scale, but the map was still a formless shape, which turned out to be the result of an offset value in the ultrasonic code, intended to account for the distance between the sensor and the edge of the robot, but which was still on a scale of metres.

Another problem encountered was with the scan matching feature. As a result of expecting to work on a high fidelity LIDAR scan, the mapping algorithm treats almost any scan which looks familiar as a potential loop closure, and revised large parts of the map based on assuming the two positions are the same. This assumption does not work nearly as well when considering a system which reads only three ranges per scan, as a location which results in three similar scan results is much less likely to be the same as one with hundreds. As a result of this scan matching, the robot's position in the map tended to wildly jump around as the map progressed to places it had already been, which made for very poor results, mapping a small arc repeatedly. This was discouraged by setting the minimum heuristic required for values to be matching to 50 instead of the default 0.

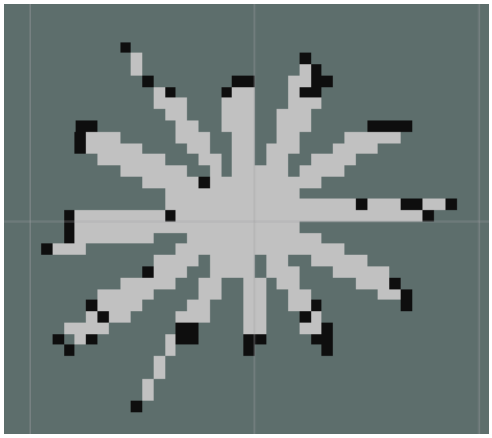
Fixing all of this led to the generation of the map shown in Figure 5.7b when performed on the largest rectangle that could be created in the maze. The robot was placed in the centre and made to rotate slowly on the spot several times. This configuration was chosen to make the results easy to interpret.

This new map was far more accurate, with a correct scale and rough shape, but seemed to be missing many measurements and not updating the map often. This was due again to the assumption of the use of LIDAR, which produces far too much data to process exhaustively, so the mapping algorithm by default only updates the map with the last scan result when the robot has moved 1 unit or rotated by 0.5 radians. This explains the 12 clear branches the map has formed, as the map is only being updated 12 times. It is possible by a normally disabled parameter, `temporalUpdate`, to force the map to be updated at regular time. When this was set to the same rate as the ultrasonic scans were being generated, the map shown in Figure 5.7a was generated.

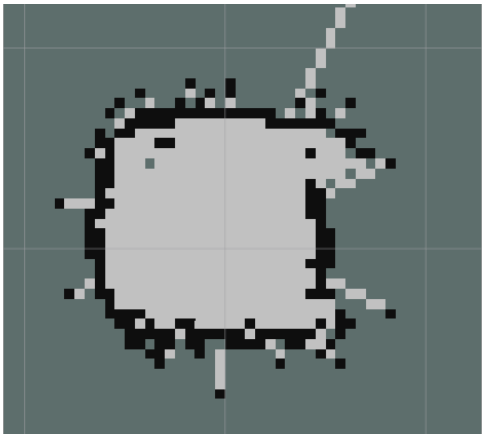
This output is clearly far better, and cursory examination showed the scale to be correct and accurate to within 1 cm. The rounded edges are to be expected, as the ultrasonics have a wide field of view, and therefore reach the walls near the corners before the corner points.



(a) With optimised parameters



(b) Before optimising parameters



(c) Obstacle at right end of map

Figure 5.7: GMapping output

This experiment was repeated on another simple configuration, this time with a large square to the right of the centre. This produced the map shown in Figure 5.7c. This map is, as expected, somewhat less clean around the obstacle, but the obstacle is still clearly present.

5.6 Computer Vision

Computer vision is a cheap and effective way of gaining a high level understanding of the environment (c.f. Section 2.4). In this system, it was used to allow the robots to identify other robots and objectives in the maze. Detecting other robots was an essential component to allow communication between robots, prevent robots mistakenly mapping other robots, and to recognise when ultrasonic interference would occur.

5.6.1 Design

The first detection system considered was a CNN, as discussed in Section 2.4.1. This is a very effective system which has the advantage of being able to classify instances of objects that it has not seen before. This would be essential if the target objects of the robot were not consistent, for instance if it was required to find people in the search space. This was, however, not required in the scope of this project. The major downside to this system would be the time to implement. To construct a CNN for this application would require the collection of a large data set of images of the robots and goals, as well as the time consuming process of tuning the CNN.

Feature-based object detection was also considered. As described in Section 2.4.1, this only requires a picture to be taken of the object and therefore takes far less time to implement than the CNN solution. However, the accuracy and consistency are largely dependent on the number of features detected, and the algorithm's complexity grows relatively quickly. The basic brute force algorithm involves comparing every key point in the object image to every key point in the frame image, which results in $\mathcal{O}(n^2)$ complexity. This can be streamlined, for instance, if it becomes impossible for the distance to be low enough for a pair to be the best match part way through the distance calculation, it does not need to be finished [75]. The calculations for deciding the best outline of the object also become more complicated, as there is a lower true positive to false positive ratio as the best matches (which are found first) are more likely to be true positives. It was decided that this would be too computationally intensive considering the strict constraint of using an RPi.

The method used was by far the simplest considered. It identified the other robots and target by their colour. This was only possible as each robot used had a distinct coloured chassis, which lacks scalability, but was considered an acceptable simplification as this was not the focus of the project.

This works by first converting the images from RGB to Hue-Saturation-Value (HSV) colour space, simplifying the colour detection as the hue of the pixel is determined by a single value instead of the ratio of three values. A colour can then be identified by a range of H values and then minimum S and V values, which is far more intuitive than checking if it falls into a range of ratios between R, G and B values.

5.6.2 Implementation

The `vision_node` works using a ROS spin setup, with a `spin()` function being called at a set frequency until ROS closes.

A `RobotDetector` object is declared which has the range of HSV values for each robot and the goal, and also has a video capture object. It contains a `search()` function which takes a frame as a parameter and returns various details about each robot's presence in

the frame. `search()` first calls `get_colour_mask()` for each range of colour values. This converts the image to HSV, then makes a frame, `mask`, which is white where the pixel is in range and black when out of range. This is shown in Listing 5.14. This also shows the handling of cases where the range of colours crosses the zero point (as zero is adjacent to 255 in the hue value).

```
1
2  def get_colour_mask(self, frame, lower_hsv_bound, higher_hsv_bound):
3      hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
4
5      if (lower_hsv_bound[0] > higher_hsv_bound[0]):
6          mask1 = cv2.inRange(hsv_frame, np.array([0, lower_hsv_bound[1],
       lower_hsv_bound[2]]),
7                              np.array(higher_hsv_bound))
8          mask2 = cv2.inRange(hsv_frame, np.array(lower_hsv_bound),
9                              np.array([179, higher_hsv_bound[1], higher_hsv_bound
       [2]]))
10         mask = mask1 | mask2
11     else:
12         lower = np.array(lower_hsv_bound)
13         upper = np.array(higher_hsv_bound)
14         mask = cv2.inRange(hsv_frame, lower, upper)
```

Listing 5.14: get_colour_mask in RobotDetector

The function then uses erosion and dilation functions provided by opencv to reduce noise in the mask.

The contours in the mask are then iterated through to find the biggest, which is assumed to be the object and the centre point and outline of the contour is found. If no contours are bigger than a fixed size (measured in pixels) the object is assumed to not be in the image frame, and the corresponding `obj_found` variable is set to false. This process is shown in Listing 5.15.

```
1  for c in contours:
2      if cv2.contourArea(c) > max((300, maxsize)):
3          maxsize = cv2.contourArea(c)
4          cx, cy = self.get_centre_point(c)
5          outline = self.get_outline(c)
6          obj_found = True
```

Listing 5.15: Contour teration in `search()`

These parameters are then returned to the `_tick()` function which was called by `spin()`. This then renders debug information to the frame if it's performing a test,

or publishes the detected robot information via a ROS message consisting of a comma separated list of objects detected in the last frame.

5.6.3 Testing

Initial testing of the `vision_node` was performed on a PC with a USB webcam. The testing was performed by using the position information to render labelled rectangles to a real time stream highlighting the position of the objects. The code for this is shown in Listing 5.16.

```
1     def _draw_bounding_rects(self, frame, objects, outlines):
2         for i, o in enumerate(objects):
3             x, y, w, h = cv2.boundingRect(outlines[i])
4             cv2.rectangle(frame, (x, y), (x + w, y + h), tuple(o['rgb']), 2)
5             cv2.putText(frame, o['name'], (x - 20, y - 20),
6                         cv2.FONT_HERSHEY_SIMPLEX, 0.5, tuple(o['rgb']), 2)
```

Listing 5.16: Object highlighting code

An example of this running is shown in Figure 5.8.

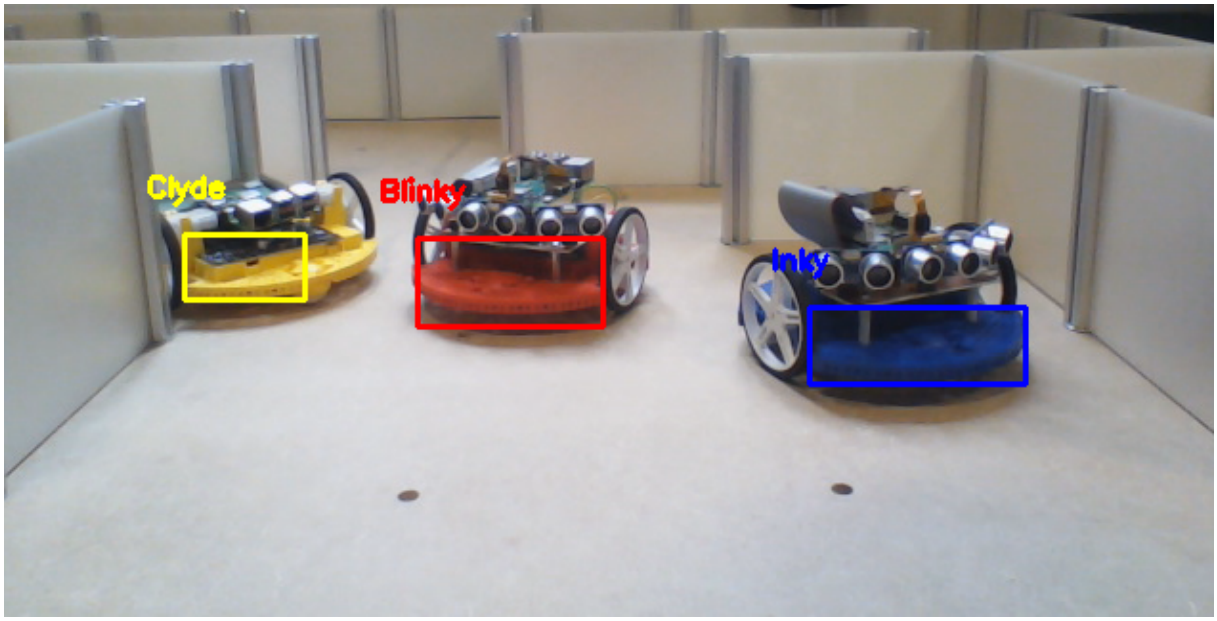


Figure 5.8: Computer vision PC test

The system was then tested on the RPi by monitoring the ROS topic `robots_detected` to test both the integration with the RPi and with ROS. Various detectable objects were moved in and out of the frame and changes in the output were observed. This initially failed as the RPi camera was connected using a CSI port, not a USB port, which OpenCV's VideoCapture function does not support. When running on the RPi, this was replaced with imutil's VideoStream package. Following this small change, the test performed as expected, consistently identifying which objects were in frame.

Functionality was then added to record the feed to observe how the computer vision responded as the robot was traversing the maze, however issues were encountered with frame rate, namely: the functionality being too computationally intensive for the RPi in addition to all other computation. This is therefore only used in debugging, and not usually active.

5.7 Control Modules

High-level control of the robots is handled in control modules in the `crues_control` package. The final control node was envisioned to perform path planning and exploration based on the map acquired from `gmapping`, with an implementation of the algorithms described in Section 2.6 used to direct the robot towards unexplored areas of the map. This was to be done using the `base_local_planner` and `global_planner` packages. Unfortunately, due to time constraints, this objective was not completed.

Separate launch files were created for subsystems which rely on multiple nodes, such as the communications system, so that they can be included in system launch files along with any required parameters in a single line. The main launch file used by every system-level launch is `crues_control/robot.launch`, which sets up all sensors and actuators, as well as nodes required for drive control, sensor fusion, and SLAM. Launching a robot with a specific control node merely requires creating a launch node which includes `robot.launch` as well as a single entry corresponding to the desired control node. Listing 5.17 shows an example of a launch file for a basic controller.

```
<launch>
    <!--Empty controller used for debugging purposes.-->
    <include file="$(find crues_control)/launch/robot.launch" />
    <node name="null_controller" pkg="crues_control" type="roomba_control.py">
        <param name="turn_vel" type="double" value="0" />
        <param name="fwd_vel" type="double" value="0" />
        <param name="obstacle_range" type="double" value="0" />
        <remap from="~rate" to="encoder/rate" />
    </node>
</launch>
```

Listing 5.17: Launch file for null controller

5.7.1 Roomba Controller

A simple control node, referred to as the "Roomba" controller due to its similarity in behaviour to the eponymous robotic vacuum cleaner, was developed in order to demonstrate the functionality of the robot and serve as a benchmark for comparison

with more advanced controllers. This controller navigates by driving straight forward whenever there is no object blocking its path, and turning on the spot whenever it is unable to continue. This results in the robot exploring the maze randomly until the goal is detected by the computer vision node, at which point the robot indicates successful completion of the maze by rapidly turning back and forth on the spot and flashing its green LED. This controller is launched by `roomba.launch` in the `crues_control` package.

Several adjustments were made to improve the performance of the controller. The target velocity for driving forward and rotational velocity for turns are parameterised in the launch file, allowing slower velocities to be specified for maze configurations with narrow passages, where faster velocities are more prone to result in collisions. Collisions were also avoided by ramping the velocity linearly in proportion to the distance from an object at distances of less than 15 cm. This causes the robot to slow down and approach obstacles more accurately, instead of attempting to decelerate instantaneously and overshooting.

Finally, the turning direction was determined based on the range values from the outer ultrasonic sensors. By turning away from the sensor with the lower range (i.e. turning to the right when an obstacle is detected on the left and vice versa), the amount of time spent turning can be reduced. This had the unintended side effect of causing the robot to get stuck in corners, turning first one way then the other. To solve this problem, the turning direction was stored as a field and maintained until the robot was clear of the obstacle. The turning angle is chosen randomly from the range 30° to 60°. If this is insufficient to clear the obstacle, the process is repeated. Randomising the angle has the advantage of preventing the robot from becoming trapped in a repeating pattern. The results of running this system in various maze configurations are presented in Section 6.3.

A variant of this controller was created where instead of turning by a random angle, the robot continues straight as soon as it is parallel to the obstacle it is avoiding. By adjusting the algorithm so that the agent then maintains a constant distance from the sensor pointing towards the wall, a simple wall following algorithm can be implemented, allowing the maze to be searched more systematically (c.f. Section 2.6.1). Without using mapping data, however, it is impossible to avoid getting caught in a loop when the maze walls do not form a simply connected graph.

5.7.2 Debugging Controllers

In addition to the Roomba control node, several simple controllers were created for test and debugging purposes. The simplest of these is the Null controller (`crues_control/null.launch`), which constantly outputs a velocity of 0. This was used primarily for debugging ROS configurations in cases where physical movement of the robot was not required.

Another controller created for a specific use case is the PID tuning controller (`crues_control/pid_tuner.launch`), which launches only those nodes required for tuning the PID on a single wheel (i.e. the corresponding encoder, motor, and PID nodes). This was used to minimise overhead while employing the practical tuning methods described in Section 5.3.

Finally, a Path control node was created (launched by `crues_control/path.launch`), which instructs the robot to follow a predetermined path. Paths can be entered at runtime by publishing distances and angles to the `fwd_cmd` and `turn_cmd` topics, respectively, or can be loaded from text files. The text files take the form of line separated instructions, where each instruction contains a command and a value. Table 5.3 shows the available commands and permissible values. Listing 5.18 shows an example (`square.txt`), which follows a 0.4 m square, with the robot starting and ending in the same pose. This controller was used in order to guarantee consistent test cases for integration tests, ensuring that results from multiple runs are comparable. Several paths were used during the testing and development of various subsystems, such as odometry and SLAM. The example shown below was used during configuration of the EKF, as shown in Figure 5.5.

Table 5.3: Available commands for the Path control node

Command	**Description**	**Argument**
`fwd_vel`	Set forward velocity	Velocity in $\mathrm{m\,s^{-1}}$ (float)
`turn_vel`	Set turn velocity	Velocity in $\mathrm{rad\,s^{-1}}$ (float)
`fwd`	Move forward by the specified distance	Distance in m (float)
`turn`	Turn by the specified angle	Angle in rad (float)
`wait`	Wait for the specified duration	Duration in s (float)
`exit`	Exit after the specified duration	Duration in s (float)

```
fwd_vel 0.15
turn_vel 1
fwd 0.4
turn 90
fwd 0.4
turn 90
fwd 0.4
turn 90
fwd 0.4
turn 90
```

Listing 5.18: Path file for following a 0.4 m square

Chapter 6

System Testing

System testing took place following integration testing and the full construction of the robot. The aim of system testing is to ensure that each of the component parts work as expected when integrated together, and to test and evaluate the system as a whole. A modular maze testing environment was used throughout integration and system testing to allow as many different maze configurations as possible to be tested. Throughout integration the modular maze was used as a "SLAM playground", where various sized boxes or simple two area configurations were used to test the robot's capabilities in these environments and assess the SLAM maps built when using these configurations.

6.1 Modular Maze

The modular maze environment was designed with the aim of being able to system test on as many maze configurations as possible. By creating a modular environment, the maze was also reusable in future iterations of the project, or similar projects which required a secure area to test small autonomous vehicles.

6.1.1 Design

With these broad specifications in mind, a list of requirements for the maze environment was created (c.f. Table 6.1).

These requirements were then used to create a number of approximate ideas which were presented to the mechanical workshop. The maze base would be 1280 mm × 1690 mm × 18 mm, giving 7 columns by 9 rows of peg holes each 12 mm in diameter and 210 mm from centre to centre. The holes drilled into the maze base would be 15 mm in depth leaving a 3 mm section of the base board un-drilled within the hole for the peg to rest on. The pegs would then be 165 mm in height, a protrusion of 150 mm from the top of the base board, with a cross slit cut from the top to roughly 75 mm down the peg (half the visible portion). These slits would be approximately 3 mm wide to allow the

Table 6.1: Modular maze requirements

Requirement	Priority
Be easily modifiable	High
Have enough cells to allow varied mazes	High
Have minimum cell width > diameter of the robot	High
Be accompanied by sufficient "pegs" and "walls" to build complex mazes	High
Wall material be non-translucent and non-reflective	Medium
Base material be non-reflective	Medium
Maze be easily transportable	Medium
Maze base be modular	Low

walls to be slotted in. For this to be accomplished, the walls would be winged—208 mm in length at the top and 196 mm in length 75 mm from the top. This gives 6 mm spaces at either side to allow for the diameter of the peg and means the walls can be slotted between two pegs with ease.

The materials required were researched to obtain the measurements above, such as 12 mm being a standard size for wood dowel and 3 mm and 18 mm respectively being depths of MDF which could be used for the walls and base boards. These requirements, descriptions and accompanying drawings were delivered to the mechanical workshop for construction to begin.

6.1.2 Implementation

The mechanical workshop required a number of changes to the original design in order for the maze to be created.

Foremost, the largest single item of solid wood which could be obtained from suppliers was smaller than the requested measurements of 1280 mm × 1690 mm × 18 mm. Two resolutions were proposed by the mechanical workshop: reduce the width of each cell, or remove a column and row of holes from the maze. Following team consultation, it was decided that reducing the size of each cell would result in more flexibility in the end product, as 3 columns of two cell wide paths could still be created. It is also worth noting that the reduced cell width of 195 mm still met the requirement of the cell width being larger than the diameter of the robot.

Secondly, the materials used for the walls and pegs would not be wood as this would involve many man hours to create the slits in the dowel and manually cut the walls to size. In order to streamline the process, it was suggested by the mechanical workshop,

that the walls be made of acrylic, allowing them to be laser cut, and the pegs be 3D printed, allowing mass printing once a design had been finalised. It was agreed amongst the group that this was a good idea and in order for the pegs to be 3D printed they had to be made using CAD modelling software. None of the group had experience using this software, hence a steep learning curve was involved in carrying out this task.

A peg matching the original specifications was modelled, however upon delivery to the mechanical workshop, the group were informed that this design had to be changed to allow the walls to be cut as rectangles only—with no wings. A new peg design was created which had slits at each 90° interval, running the length of the peg and leaving a central column intact. This allowed the wall to be dropped in without the need for wings. This peg design was also modelled.

Upon returning to the mechanical workshop the group were once again informed that the peg design had to be altered. The base board had been drilled through and so the pegs now required a smaller peg on the bottom and a rim to ensure that they did not fall through the base board. The agreed design was the same as previously with the peg now having a diameter of 15 mm with the small peg at the bottom having the original 12 mm diameter and 15 mm in height. The design was agreed and the peg was modelled.

Midway through modelling the group were called by the mechanical workshop and informed that the acrylic which had been ordered was slightly wider than the originally thought 3 mm and so the slits would have to be widened to 4 mm to allow for this. The model was altered to account for this change and delivered to the mechanical workshop to be 3D printed.

Unfortunately, once printed, it was discovered that the design resulted in a peg that was too fragile and often broke at the base. The mechanical workshop attempted to resolve this issue by increasing the density of the printing, however this was unsuccessful. Eventually, the mechanical workshop concluded that ordering a pre-slitted rod of aluminium and attaching metal pegs to the base to allow them to fit into the base board would be the best solution. This solution was successful and the maze, with 53 pegs and 52 walls, was delivered in full shortly afterwards. An example configuration of the maze is shown in Figure 6.1.

As can be seen from Table 6.2, the requirements laid out in the design phase (c.f. Section 6.1.1) were largely met with the only two requirements not met being of medium or low priority.

Figure 6.1: An example maze configuration constructed using the maze

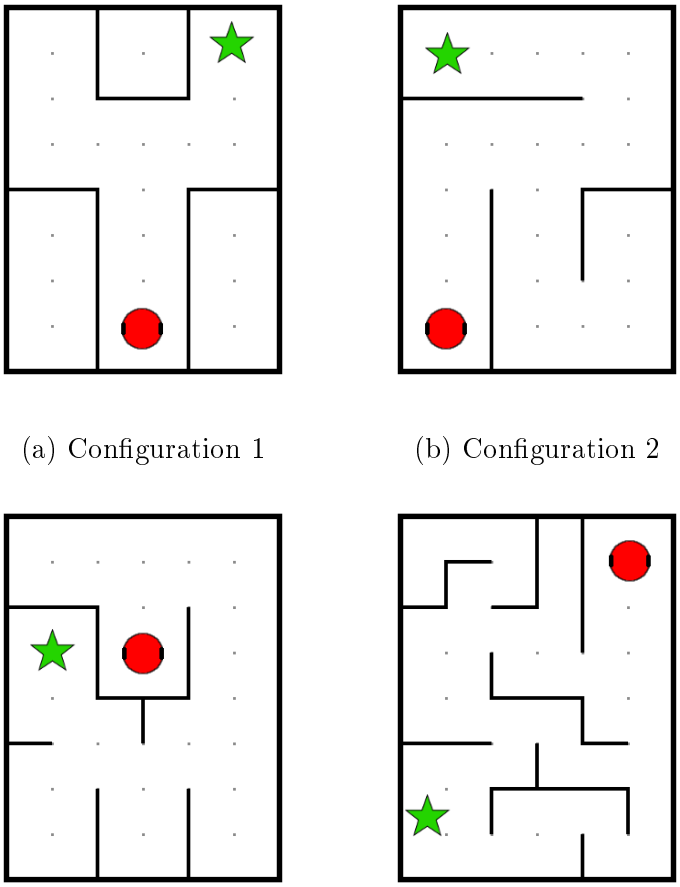
Table 6.2: Modular maze requirements met

Requirement	Priority	Met
Be easily modifiable	High	Met
Have enough cells to allow varied mazes	High	Met
Have minimum cell width > diameter of the robot	High	Met
Be accompanied by sufficient "pegs" and "walls" to build complex mazes	High	Met
Wall material be non-translucent and non-reflective	Medium	Met
Base material be non-reflective	Medium	Met
Maze be easily transportable	Medium	Not Met
Maze base be modular	Low	Not Met

6.2 Testing Strategy

The testing strategy designed for system testing was to develop a number of increasingly difficult mazes and test the capability of the robot to find a goal in each of these mazes. Additional robots will then be added to each of these configurations and the time recorded for quantitative testing of the project.

A number of maze configurations were drawn up to be used as the default test cases for the robots. These configurations are shown in Figure 6.2. It is worth noting that these configurations were created before the cell width of the maze was reduced and therefore, maze configurations with 1 cell width paths are significantly more difficult to traverse than previously thought.



(a) Configuration 1

(b) Configuration 2

(c) Configuration 3

(d) Configuration 4

Figure 6.2: Example maze configurations

6.3 Results

Due to the issues described with PID tuning and SLAM described in Sections 5.3 and 5.5.2 respectively, minimal system testing took place. The system tests which did take place were without the SLAM/AI integration, so the AI had to make decisions without knowledge of the derived map. Despite this, the system was tested on the maze configurations shown in Figure 6.2 with one agent, and then again with two. These results are shown in Table 6.3.

Table 6.3: Results

Maze configuration	**Avg. runtime with one agent [s]**	**Avg. runtime with two agents [s]**
1	57.8	22.7
2	85.2	81.9
3	80.6	71.2
4	134.2	118.3

The results shown are not indicative of the project had it been completed in its entirety, however these results give benchmark scores for the shown maze configurations using a very basic AI module. A major drawback of the lack of SLAM/AI integration is that the same path can be searched multiple times, which resulted in a few instances with unusually slow results. The results do broadly show, however, that multiple agents perform better than a single agent in terms of speed of search. In these cases, this is a simple case of "many hands make light work" rather than anything intelligent that the robots are doing. Nevertheless, the results demonstrate the successful outcomes of the project and vindicate the time taken to research the field of co-operative robotics. Work detailed in further work (Section 8) can use the results shown here as benchmark results and measure the success of refined SLAM and AI implementations against these results to evaluate progress being made.

Chapter 7

Evaluation

Based on the original objectives of the project (c.f. Section 1.2), the project can be deemed as somewhat of a success. Five out of six of the original "Major" objectives have been completed, as well as one of two of the "Optional" objectives. In addition to this, it is believed that with a small number of additional weeks, the remaining "Major" objective could be completed. Given the project aimed to combine three engineering disciplines and tackle complex problems simultaneously while using inexpensive components, a strong effort has been made. As detailed in various previous sections of the report, small setbacks in various aspects of the project caused some of the latter stages of the project to be uncompleted. This was especially true as many of the late-stage elements were sequential, and the issues with the EKF resulted in large delays in the SLAM implementation.

Despite these setbacks, the management of the project was effective and the tasks were predominantly completed to schedule in the early to middle stages of the project. Less management of the project was required in the latter stages as objective related tasks became sequential, and those who were not working on those tasks were assigned secondary objectives pertaining to other deliverables of the project. Git was used effectively, with the issues feature and protected merging used to ensure all members of the team were working on a task and completion of tasks could be monitored by the project manager.

A series of benchmark results have also been obtained and can be used by any future continuations of the project to measure success. The objectives which have been completed were also iterated upon multiple times to complete for each of the robots to ensure the quality of the design.

7.1 Mechanical

Mechanical aspects of the project have been completed to a high standard as is demonstrated by the robustness of the robots when in motion and the durability and modularity of the maze. This is largely due to the decision to complete these tasks first

with a higher priority and allow time for the designs to be iterated upon throughout the project. The first robot was carefully designed to ensure each component mechanically integrated well. This was further iterated upon and improved between robots, leading to a final construction which was consistent and reliable. Given not all objectives were met at the end of the project—and the electrical and software aspects were to be the focus—the decision taken in the early stages to purchase a pre-built chassis, as opposed to create a bespoke chassis was justified, due to the time saved. This would have added unnecessary complication in the beginning of the project which could have caused future tasks to be delayed further.

The only issue which was encountered with the pre-built chassis was the accompanying motors, which were identified as the possible cause of issues throughout the electrical and software implementations. When used for a reasonable period of time, the motors began to leak grease from their plastic casing as described in Section 4.1. As the encoders are on the motor shaft and not the drive shaft, this affected the feedback from the encoders. It is thought that this contributed to increased differences between robots at the end of the project and increased difficulty in tuning the PID controller. If this issue had been known from the start, the motors would have been either rotated in testing to balance the degradation, or different motors would have been used with a bespoke chassis, as finding motors to fit the pre-existing chassis would not have been a suitable solution.

Aside from the robots, the other main mechanical component was the modular maze testing environment. Although constructed by the mechanical workshop, a great deal of effort was put into the design, and subsequent iterations, to obtain the best outcome possible. The modular maze is a reusable "SLAM playground" and is a major successful outcome of the project. Although not particularly transportable, the maze is robust and durable and intended to last and can be used for future projects.

In addition, although not used in the final design, a great deal was learned from using CAD to create designs of pegs which were intended for use in the maze. A number of iterations of the design were created, which were improved both due to the feedback from the mechanical workshop as the specification changed and as the proficiency of using the CAD software increased. It was also worth noting that the CAD designed pegs were not flawed because of the design, but due to limitations in the printing/construction process and as a result were unusable. The mechanical portion of the project can therefore overall be deemed a success as the individual components were made to a high standard of design and construction.

7.2 Electrical

The electrical components of the project were completed to a high standard, however some unforeseen complications caused the completion of these to be delayed. The pre-

built power distribution board chosen had several advantages over a custom built solution. Firstly, it was very easily integrated with the Polulu chassis. Secondly, it saved the significant amount of time it would have taken to build the drive circuit. Lastly, due to manufacturing restrictions on the PCB, there was very little remaining space on the system's main PCB, so using a custom built system would have required the design of a second PCB.

The range sensors used proved to be largely successful, with some caveats. During the unit testing phase, the sensors performed well, giving consistent readings that were accurate to within the tolerable uncertainty. During the SLAM testing, they mostly measured accurate results, however, their limitations started to become apparent. As the distance increased, the widening cone of detection caused some erroneous data to be measured. This is assumed to be the cause of the corners appearing to be rounded in the maze in Figure 5.7a. Another issue with using the ultrasonic sensors is that the RPi used a time shared OS, which meant that the readings from timing the pulse are sometimes inexact. This could be improved with the use of a dedicated timing circuit.

Finally, as there were several range sensors on each robot, taking one measurement sweep took a relatively long time. This is especially the case when multiple robots are synchronising their sweeps to avoid interference. This would have been mitigated if an infrared sensor had been used. This would cost more, but it could be possible to mount the sensor on a servo motor, and move it through 180° sweeps as to not require more than one. This would also remove the timing issue, as they generally return an analogue voltage instead of a pulse time. Initially, this design concept was discounted due to the additional mechanical complexities of mounting the components and cost. However, without the time and the self-imposed constraint of minimising cost, this solution would likely improve the system's performance.

While using an IMU to supplement wheel odometry is a common solution, its effectiveness in this project has not been evident. The IMU returned reasonable results in testing, and the visualisation produced using IMU data integrated with ROS shows promise. However, as the EKF was one of the last aspects of the project to be functional, results from its sensor fusion were not been obtained to measure their accuracy.

Having multiple robots allowed an iterative design process to be carried out, significantly improving the outcome of the PCB. Connections and vias were altered following the first design to be more easily integrable with the mechanical layout.

Time and care was taken to ensure the electrical design of the PCB and any other connections between parts on the robot—such as the ribbon cable connection between the PCB and the RPi—were robust and correct. This resulted in a robot which is both electrically and mechanically sound and easily replicable with time and parts. The robots which possess prototype parts, used in interim stages of the iterative design process,

remain functional and usable. This demonstrates the careful selection and design of each of the parts.

The use of AA batteries throughout the middle stages of the project when the PID controller was being tuned, resulted in a great deal of batteries being consumed. This was due to the high current spikes which were being caused by motor stalls and fewer batteries could have been used if this prolonged process had been foreseen. The consistent current draw of the motors throughout this phase of testing should have been monitored more closely and action to prevent this taken sooner.

Overall, the electrical portion of the project was carried out effectively but did cause some delays which compounded later in the project. A number of blocking tasks in the middle stages of the project required unforeseen parts to be ordered such as ribbon cable and 40-pin IDC connectors which it was thought would be found within the department. This was not the case as 40-pin ribbon cable and connectors are used predominantly for Raspberry Pis, making them more expensive and therefore, not a common order for the department. Hence, these parts had to be ordered in and resulted in the task blocking others longer than expected as these were required for system testing. Towards the end of the project, an RPi was shorted—the cause is unknown, despite investigations and testing—and ceased to function, meaning time was lost to finding the cause of this issue.

7.3 Software

The software architecture was well designed and thought-out making use of the libraries and tools available. By using the ROS framework, the architecture was predominantly handled and did not need to be managed further. ROS also provided access to other tools and libraries such as “differential-drive”, “robot- localization” and “gmapping”. It was thought that these libraries would be “plug and play” with the modules created as part of the electrical sections. Despite designing and implementing the electrical modules to integrate with these libraries, problems were still encountered with integration.

The libraries were far more complex than first thought and were also not designed for the configuration of sensors used in this project. Additional modules had to be created as “adapters” with additional parameters between the libraries and the originally created modules. Each of the libraries used therefore also had to be fully understood, at a source code level, to construct the “adapters” and alter the original modules. This took longer than expected, however did result in more in-depth knowledge of each of the component parts of the software. As the libraries were designed for a variety of other sensors, mainly LIDAR instead of ultrasound, the results obtained were not of as high quality as expected and this impacted the time taken to tune parameters to achieve the best possible outcome. Although this was somewhat achieved for SLAM, this left no time in the project to convert

the SLAM map to a world state which could then be used by an implemented AI to solve the maze intelligently.

Furthermore, the combination of packages and demanding software made it very difficult to run all the code at once on the RPi. The ROS framework together with demanding functions like SLAM and Sensor Fusion proved to be too much for the limited memory and processing power, which made testing fully integrated systems extremely challenging.

Chapter 8

Further Work

As not all of the major objectives of the project were completed, the primary objective going forward would be to implement various AI algorithms and investigate their effectiveness in solving mazes co-operatively. A series of benchmark results have been provided in Section 6.3 to evaluate these by. It is worth noting that A* is likely to be the best path planning algorithm once the maze has been fully or mostly explored. To explore the maze, many different algorithms could be used. Using periodic scans to improve the SLAM map, a frontier-based algorithm could be built upon, using a depth first approach, to avoid doubling back. Robot behaviour on goal discovery would also have to be, explicitly: whether one goal being discovered was an end state — meaning the robot could return to its starting point — or whether to continue searching for more goals.

Various improvements could also be made to the SLAM (c.f. Section 5.5) and Odometry (c.f. Section 5.4). The exploration of other libraries to make improvements to the mapping capabilities of the robot, when using ultrasonic sensors and limited feedback methods, could be explored through testing their performances as part of further work. Two such libraries are: SLAMOT (SLAM with Object Tracking) which could be used to resolve the issue of robots existing in each other's maps, as moving objects would be tracked and remain unmapped; and AMCL for localisation, which compensates for the drift in time between the odometry and reference frames.

The second optional objective, functionality for map sharing and SLAM loop closure across robots, could also be implemented. This would allow robots which have covered different paths to communicate their knowledge of the maps and improve search efficiency. It would also allow both robots to improve the accuracy of their existing map by combining it with that of the other robot. To implement loop closure, reference points would need to be added to the maze environment. These landmarks provide the robots a stationary point from which other measurements can be made relative to, increasing the accuracy of the maps created.

Subsequent to these, more robots could be added to the system in order to stress test the scalability of the project and its components. In areas such as the ultrasonic range finders which are likely to produce bottlenecks in the system with more agents, alternative or improved options should be explored. For example, devising a method for synchronising ultrasonic pulses to prevent interference or utilising more complex and expensive range finders such as LIDAR.

Once these stress tests were successful, the maze environment could be altered to have walls and a floor which more closely simulate a real-life environment. These changes may include adding texture to the walls to evaluate the response of the ultrasonic sensors; using the maze in a darkened environment to evaluate the response of the computer vision node and if additional lighting is required on the robots; and altering the texture of the floor to evaluate the response of the motor control system and the stability of the robots.

Following these evaluations, the system could then either be upscaled in mechanical terms to perform in the field or used directly and tested in the field. Explorations of different real-life environments could be evaluated and the challenges which come with those tackled. This could lead the project to be a cheap, modular option which could be used in a variety of applications such as those described in this report.

Using a similar software and sensing system, underwater or aerial agents could also be created and tested, further evaluating the software system's effectiveness and the overall modularity of the system.

Chapter 9

Conclusion

This project aimed to investigate co-operative robotics within environments with restricted communication by constructing several robots which used scalable methods to map and explore their surroundings and find a goal. The project required the mechanical construction and electrical design of three robots, as well as the development of sensor and control software. Research into these topics provided knowledge which allowed design decisions to be taken for each of the main aspects of the project. Aspects of agile methodology were employed to manage the team over the course of the project and were proven successful with five of the six major objectives completed. Influenced by the design decisions, a pre-built chassis was used as the basis for a mechanically functional robot using odometry, exteroceptive sensors, and actuators to explore its environment. These sensors included ultrasonic sensors, an IMU, encoders and a basic camera, each of which were mounted on the robot via a bespoke PCB. The PCB design was iterated upon throughout the project to achieve a final design which connected each of the components in a mechanically and electrically sound manner.

For the components to function together, a software architecture based on data-oriented software design principles was created. This architecture centred on the Robot Operating System (ROS) framework which uses the publish–subscribe pattern to allow modules in the system to communicate while minimising coupling. Software modules based on ROS node design principles were written for each sensor and actuator, as well as for control structures. Each of these modules was rigorously tested, both in isolation and as part of the overall system.

A flexible communication system was developed that allows robots to communicate directly to a variety of ROS topics, allowing dynamic creation of topics as required. This will allow communication to occur simultaneously at several control levels, from global path planning to sensor synchronisation.

To allow the robot to explore, ROS libraries for differential drive kinematics, sensor fusion, and SLAM were used. Introducing each of these libraries added complexity to the project, and issues were encountered throughout integration. Technical challenges

were encountered with mechanical components, power supply, and the synchronisation of active sensors between robots. Although a number of these issues were overcome, the delays in completing these objectives meant that the final objective—the development of co-operative AI control modules—could not be completed. As a consequence, system testing results obtained using the custom-designed testing environment can only be seen as benchmark results which provide a baseline for future work.

Significant thought has been given to solutions to the outstanding problems, with a clear path outlined for future development of the project. Substantial progress was made in constructing a flexible platform for addressing the remaining challenges. In particular, significant work was accomplished on communication protocols, sensor fusion, localisation and mapping. A combination of the configurable testing environment, the robust mechanical construction of the robot, and the modular software architecture allows for rapid development of additional control systems.

Bibliography

[1] G. Dudek, M. R. M. Jenkin, E. Milios and D. Wilkes, 'A taxonomy for multi-agent robotics', *Autonomous Robots*, vol. 3, no. 4, pp. 375–397, Dec. 1996.

[2] A. Khan, B. Rinner and A. Cavallaro, 'Cooperative robots to observe moving targets: Review', *IEEE Transactions on Cybernetics*, vol. 48, no. 1, pp. 187–198, Jan. 2018.

[3] J. Pliego-Jimenez and M. Arteaga-Perez, 'Telemanipulation of cooperative robots: A case of study', *International Journal of Control*, vol. 91, no. 6, pp. 1284–1299, 2018.

[4] P. M. Wensing and J.-J. Slotine, 'Cooperative adaptive control for cloud-based robotics', in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018, pp. 6401–6408.

[5] H. Cheung and T. Wong. (2018). The full story of Thailand's extraordinary cave rescue, BBC News, [Online]. Available: `https://www.bbc.co.uk/news/world-asia-44791998` (visited on 3rd Apr. 2019).

[6] O. Dictionaries. (2019). Robot | defintion of robot in english by oxford dictionaries, Oxford Dictionaries, [Online]. Available: `https://en.oxforddictionaries.com/definition/robot` (visited on 21st Mar. 2019).

[7] Merriam-Webster. (2019). Robotics | defintion of robotics by merriam-webster, Merriam-Webster, [Online]. Available: `https://www.merriam-webster.com/dictionary/robotics` (visited on 21st Mar. 2019).

[8] G. Saridis, 'Intelligent robotic control', *IEEE Transactions on Automatic Control*, vol. 28, no. 5, pp. 547–557, 1983.

[9] D. Barnes and D. Eustace, 'A behaviour synthesis architecture for mobile robot control', in *Autonomous Guided Vehicles, IEE Colloquium on*, IET, 1991, pp. 3–1.

[10] E. Sahin, 'Swarm robotics: From sources of inspiration to domains of application', in *Swarm Robotics*, 2004, pp. 10–20.

[11] S. Premvuti and S. Yuta, 'Consideration on the cooperation of multiple autonomous mobile robots', in *Intelligent Robots and Systems' 90.'Towards a New Frontier of Applications', Proceedings. IROS'90. IEEE International Workshop on*, IEEE, 1990, pp. 59–63.

[12] R. Beckers, O. Holland and J.-L. Deneubourg, 'From local actions to global tasks: Stigmergy and collective robotics', 1994.

[13] L. E. Parker, 'The effect of action recognition and robot awareness in cooperative robotic teams', in *IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, vol. 3, 1995, pp. 212–219.

[14] Y. U. Cao, A. S. Fukunaga, A. B. Kahng and F. Meng, 'Cooperative mobile robotics: Antecedents and directions', in *Intelligent Robots and Systems 95.'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on*, IEEE, vol. 1, 1995, pp. 226–234.

[15] F. Matsuno, N. Sato, K. Kon, H. Igarashi, T. Kimura and R. Murphy, 'Utilization of robot systems in disaster sites of the great eastern japan earthquake', in *Field and Service Robotics*, Springer, 2014, pp. 1–17.

[16] N. Mathews, A. L. Christensen, R. O'Grady and M. Dorigo, 'Spatially targeted communication and self-assembly', in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 2678–2679.

[17] W. Institute. (2018). Rolls royce and swarm robotics, [Online]. Available: `https://wyss.harvard.edu/media-post/rolls-royce-and-swarm-robots/` (visited on 16th Mar. 2019).

[18] K. M. Lynch and F. C. Park, *Modern Robotics*. Cambridge University Press, 2017, ch. 1, p. 1.

[19] L. UK. (2019). How does LiDAR work?, LiDAR UK, [Online]. Available: `http://www.lidar-uk.com/how-lidar-works/` (visited on 27th Mar. 2019).

[20] J. Read, 'Early computational processing in binocular vision and depth perception', *Progress in biophysics and molecular biology*, vol. 87, no. 1, pp. 77–108, 2005.

[21] J. D. Pfautz, 'Depth perception in computer graphics', University of Cambridge, Computer Laboratory, Tech. Rep., 2002.

[22] K. J. Åström, T. Hägglund and K. J. Astrom, *Advanced PID control*. ISA-The Instrumentation, Systems, and Automation Society Research Triangle ..., 2006, vol. 461.

[23] Y. Chen, D. P. Atherton *et al.*, *Linear feedback control: analysis and design with MATLAB*. Siam, 2007, vol. 14, ch. 6, p. 183.

[24] J. G. Ziegler and N. B. Nichols, 'Optimum settings for automatic controllers', *trans. ASME*, vol. 64, no. 11, 1942.

[25] K. J. Åström and T. Hägglund, 'Revisiting the ziegler–nichols step response method for pid control', *Journal of process control*, vol. 14, no. 6, pp. 635–650, 2004.

[26] G. Grisetti, R. Kummerle, C. Stachniss and W. Burgard, 'A tutorial on graph-based slam', *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[27] M. Kam, X. Zhu and P. Kalata, 'Sensor fusion for mobile robot navigation', *Proceedings of the IEEE*, vol. 85, no. 1, pp. 108–119, 1997.

[28] D. Fox, J. Hightower, L. Liao, D. Schulz and G. Borriello, 'Bayesian filtering for location estimation', *IEEE pervasive computing*, no. 3, pp. 24–33, 2003.

[29] S. Huang and G. Dissanayake, 'Convergence and consistency analysis for extended kalman filter based slam', *IEEE Transactions on robotics*, vol. 23, no. 5, pp. 1036–1049, 2007.

[30] D. H. B. C. M. Brown, *Computer Vision*. Prentice Hall, 1982, p. xiii, 428 pp., ISBN: 0-13-165316-4.

[31] M. Paoletti, J. Haut, J. Plaza and A. Plaza, 'A new deep convolutional neural network for fast hyperspectral image classification', *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 145, pp. 120–147, 2018.

[32] K. L. Ho and P. Newman, 'Loop closure detection in slam by combining visual and spatial appearance', *Robotics and Autonomous Systems*, vol. 54, no. 9, pp. 740–749, 2006.

[33] J. Schmidhuber, 'Deep learning in neural networks: An overview', *Neural networks*, vol. 61, pp. 85–117, 2015.

[34] D. G. Lowe, 'Distinctive image features from scale-invariant keypoints', *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[35] ROS. (2018). About ROS, ROS, [Online]. Available: `http://www.ros.org/about-ros/` (visited on 12th Mar. 2019).

[36] ——, (2018). Is ROS for me?, ROS, [Online]. Available: `http://www.ros.org/is-ros-for-me/` (visited on 12th Mar. 2019).

[37] C. Crick, G. Jay, S. Osentoski, B. Pitzer and O. C. Jenkins, 'Rosbridge: Ros for non-ros users', in *Robotics Research*, Springer, 2017, pp. 493–504.

[38] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler and A. Y. Ng, 'Ros: An open-source robot operating system', in *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, p. 5.

[39] YARP. (2018). Welcome to YARP, YARP, [Online]. Available: https://www.yarp.it/index.html (visited on 13th Mar. 2019).

[40] ——, (2018). What exactly is YARP?, YARP, [Online]. Available: https://www.yarp.it/what_is_yarp.html (visited on 13th Mar. 2019).

[41] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited, 2016.

[42] L. Kanal and V. Kumar, *Search in artificial intelligence*. Springer Science & Business Media, 2012, ch. 1, pp. 29–30.

[43] ArcBotics. (2016). Maze solving, ArcBotics, [Online]. Available: http://arcbotics.com/lessons/maze-solving-home-lessons/ (visited on 20th Mar. 2019).

[44] R. Klein and T. Kamphans, 'Pledge's algorithm-how to escape from a dark maze', in *Algorithms Unplugged*, Springer, 2011, pp. 69–75.

[45] S. Even, *Graph algorithms*. Cambridge University Press, 2011, p. 46.

[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to algorithms*. MIT press, 2009, ch. 22.2, pp. 531–539.

[47] M. Xu, Y. Liu, Q. Huang, Y. Zhang and G. Luan, 'An improved dijkstra's shortest path algorithm for sparse network', *Applied Mathematics and Computation*, vol. 185, no. 1, pp. 247–254, 2007.

[48] M. L. Fredman and R. E. Tarjan, 'Fibonacci heaps and their uses in improved network optimization algorithms', *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[49] Project320. (2019). A beginners guide to understanding the agile method, [Online]. Available: https://project320.com/resources/the-agile-method/ (visited on 10th Apr. 2019).

[50] Pololu. (2019). Romi chassis kits, Pololu, [Online]. Available: https://www.pololu.com/category/203/romi-chassis-kits (visited on 19th Mar. 2019).

[51] T. Bräunl, *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*. Springer Science & Business Media, 2013, 428 pp., ISBN: 978-3-662-05099-6.

[52] Pololu. (2019). Motor driver and power distribtuion board for Romi chassis, Pololu, [Online]. Available: https://www.pololu.com/product/3543/resources (visited on 23rd Mar. 2019).

[53] Texas Instruments. (2019). DRV8838 1.8A low voltage brushed DC motor driver, Texas Instruments, [Online]. Available: http://www.ti.com/product/DRV8838/description (visited on 23rd Mar. 2019).

[54] Mouser Electronics. (2014). MP4423H high efficiency 3A, 36V, synchronous step down converter, Mouser Electronics, [Online]. Available: https://www.mouser.com/datasheet/2/277/MP4423H_r1.0-1384319.pdf (visited on 23rd Mar. 2019).

[55] *Gp2y0a710k0f distance measuring sensor unit*, Sharp. [Online]. Available: https://docs-emea.rs-online.com/webdocs/0d1b/0900766b80d1bdcd.pdf.

[56] *Ultrasonic ranging module hc - sr04*, ElecFreaks. [Online]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf.

[57] *Mpu-6000 and mpu-6050 product specification*, PS-MPU-6000A-00, Rev. 3.4, InvenSense, Jun. 2013.

[58] D. Garlan and M. Shaw, 'An introduction to software architecture', in *Advances in software engineering and knowledge engineering*, World Scientific, 1993, pp. 1–39.

[59] D. Thomas *et al.* (). Catkin: A CMake-based build system that is used to build all packages in ROS, GitHub, [Online]. Available: https://github.com/ros/catkin (visited on 29th Mar. 2019).

[60] B. Peterson. (3rd Nov. 2008). PEP 373—Python 2.7 release schedule, [Online]. Available: https://www.python.org/dev/peps/pep-0373/ (visited on 8th Apr. 2019).

[61] User Beta_b0t. (27th Feb. 2019). How to setup ROS with Python 3, Medium, [Online]. Available: https://medium.com/@beta_b0t/how-to-setup-ros-with-python-3-44a69ca36674 (visited on 8th Apr. 2019).

[62] M. Hidalgo *et al.* (5th Apr. 2019). ROS 2 overview, ROS, [Online]. Available: https://index.ros.org/doc/ros2/ (visited on 8th Apr. 2019).

[63] M. Rajesh and J. Gnanasekar, 'Congestion control in heterogeneous wanet using frcc', *Journal of Chemical and Pharmaceutical Sciences ISSN*, vol. 974, p. 2115, 2015.

[64] Python Software Foundation. (2019). Socketserver - a framework for network servers, [Online]. Available: https://docs.python.org/2/library/socketserver.html (visited on 16th Mar. 2019).

[65] N. Jennings. (2018). Socket programming in python: Multi-connection server, [Online]. Available: https://realpython.com/python-sockets/#multi-connection-client-and-server (visited on 22nd Mar. 2019).

[66] J. Stephan. (2018). Differential-drive - ros wiki, [Online]. Available: http://wiki.ros.org/differential_drive (visited on 19th Mar. 2019).

[67] A. S. McCormack and K. R. Godfrey, 'Rule-based autotuning based on frequency domain identification', *IEEE transactions on control systems technology*, vol. 6, no. 1, pp. 43–61, 1998.

[68] Omega. (2019). Pid tuning - how to tune a pid controller manually?, [Online]. Available: https://www.omega.co.uk/technical-learning/tuning-a-pid-controller.html (visited on 24th Mar. 2019).

[69] D. E. Koditschek, *Quadratic Lyapunov functions for mechanical systems*. Yale University, 1987.

[70] S. Kawamura, F. Miyazaki and S. Arimoto, 'Is a local linear pd feedback control law effective for trajectory tracking of robot motion?', in *Proceedings. 1988 IEEE International Conference on Robotics and Automation*, IEEE, 1988, pp. 1335–1340.

[71] ROS Documentation. (2018). ROS Kinetic API, [Online]. Available: http://docs.ros.org/kinetic/ (visited on 11th Apr. 2019).

[72] T. Moore and D. Stouch, 'A generalized extended Kalman filter implementation for the Robot Operating System', in *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*, Springer, Jul. 2014. [Online]. Available: http://wiki.ros.org/robot_localization (visited on 11th Apr. 2019).

[73] E. Wan and R. Van Der Merwe, 'The unscented Kalman filter for nonlinear estimation', in *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*, Lake Louise, Alta., Canada: IEEE, 2000, pp. 153–158, ISBN: 978-0-7803-5800-3. DOI: 10.1109/ASSPCC.2000.882463. [Online]. Available: http://ieeexplore.ieee.org/document/882463/ (visited on 11th Apr. 2019).

[74] S. Macenski. (24th Sep. 2018). AMCL—ROS wiki, [Online]. Available: http://wiki.ros.org/amcl (visited on 14th Apr. 2019).

[75] G. Bradski, 'The OpenCV Library', *Dr. Dobb's Journal of Software Tools*, 2000.

Appendix A

Repository Structure

The code for this project can be found at:

https://github.com/rddunphy/CRUES

The directory structure is as follows:

- **docs**: Documentation
  - **final**: Latex files of interim report
  - **interim**: Latex files of interim report
  - **img**: images used in both reports
- **rviz**: Rviz configuration files
- **wiring**: Hardware Design Files including PCB files
- **crues_pi**: Source Code
  - **config**: Contains each robot's YAML config files
  - **crues**: Contains helper scripts
  - **ros_pkgs**: Contains the directories of the ROS packages
  - **deploy.sh**: Script to deploy code to RPis