QUEEN'S UNIVERSITY BELFAST

SWARM ROBOTICS

BEng Final Year Project Report

Tolu Omitola

April 2023

Abstract

This project covers the implementation of a swarm robotics system, created using the RP2040 based Raspberry Pi Pico microcontrollers, programmed in the embedded Python language MicroPython. The three-robot large swarm demonstrates foraging behaviour which is achieved by implementing the main principles of swarm robotics to achieve autonomy and communication within the swarm.

First a literature review covers the current implementations of swarm robotics and their common characteristics, which is followed by the ideas used to develop the project inspired by the research. The hardware and software implementations are covered, with all components used provided with website links and code written is made available in the appendix for future use of recreating the system. Testing is done to examine the capability of the system, with the limitations of the system and challenges faced during the creation of the system discussed. Future improvements on the mechanisms, hardware, software and testing platforms are suggested for building on the foraging capabilities the system demonstrates.

Declaration of Originality

I declare that this report is my original work except where stated.

Project Specification

The following specification is as set by the university

Autonomous systems and robotics is one of the most important strategic areas of importance as outlined by the UK government. A group of simple inexpensive robots working cooperatively can carry out and complete a mission in a reduced amount of time with less effort. Cooperative robotics or cobotics poses a number of challenges such as collision-avoidance, path-planning and group formation. In this project, the aim is to investigate cooperative robotics for a large group of autonomous vehicles in both structured and unstructured environments.

The project will make use of small-footprint platforms (e.g. Raspberry Pi, Arduino or similar) to develop low cost robotic platforms to test swarm robotics. Various sensing hardware such as ultrasonic and orientation sensors will be integrated to detect other cooperative users and the environment. The project will require working knowledge of C/C++ programming, Java, Python or any other programming language. Knowledge of PCB design and embedded hardware will be beneficial, although not essential.

Objectives

1. Investigate cooperative robotics, in particular swarm robotics and review existing work in the area.
2. Develop swarm robotics for a structured workplace followed by an unstructured environment.
3. Develop a laboratory-based platform to demonstrate the concept of cooperative robotics.

Acknowledgements

Thank you to Dr Wasif Naeem for the meetings that helped monitor the progress of the project on a regular basis.

Thank you also to the technicians on the 9th floor of the Ashby building: Jim Norney for ordering parts and components, Gerry Rafferty for manufacturing the PCBs and 3D printing parts, and Tony Boyle for providing continual assistance – both technical and clerical – over the course of the project.

Contents

1 Introduction

Swarm robotics is a field in mobile robotics which takes an approach to the coordination of multiple robots as a system which consist of large numbers of mostly simple physical robots [1]. The basic requirements of swarm robotics are that the agents (robots) of the swarm are homogeneous, utilise decentralised communication methods and work together in a cooperative manner that the individual agents would not be able achieve by themselves.

Due to the number of agents typically used in a swarm (N>10), the robots aim to be low cost with minimum capabilities for sensing, actuating and communication. This is done by using time of flight, ultrasonic, IR or even cheap cameras for agents to sense the world around them, DC motors suited for the agents' task, and RF, Bluetooth, WLAN or IR line of sight technology [2] for intra-swarm communication.

Research in the field takes inspiration from behaviours in biological systems that can be found in nature, from cellular organisms to packs of mammals. However, research in the field of swarm robotics focuses on insect behaviour due to their large size, relatively simple behaviours, but large potential in the tasks they can accomplish. A common behaviour observed among insect swarms are those of foraging when collecting resources for a colony.

Foraging behaviours are carried out in different ways among different species depending on the goal they aim to complete. This can be flocking to a resource location, following pheromone scents to locate the shortest paths to resources or working together to carry larger loads.

Swarm robotics are suited for foraging tasks by virtue of their structure and have applicable real world uses in tasks such as in mining, search and rescue, or exploration and mapping. On an individual level, each agent will be able to navigate and detect "resources" and will be able to communicate to the other agents on its success at locating the resource.

This project aims to use develop a low cost swarm system with local communication among agents in order to achieve an aggregation foraging task, employing the appropriate hardware and software required, and develop a testing environment that can be used to observe the swarm's success in achieving its goal. Limitations of the system and improvements that can be made are evaluated for improving the functionality of the swarm in a more sophisticated manner.

Section 2 reviews existing literature on swarm robotics, algorithms that are normally used, and current foraging methods.

Section 3 covers the proposed ideas for this project.

Section 4 gives and overview of the project development, schedule and cost.

Sections 5 and 6 covers the implementation of the project, including hardware choices and code used for individual and swarm behaviours.

Section 7 covers the testing implemented to observe the functionality of the system.

Sections 8 and 9 gives a breakdown of the challenges and limitations faced during development of the final system and of the previous prototypes.

Section 10 suggests future improvements that can be made to the system on swarm and individual agent levels.

Section 10 concludes the report and discusses the implementation of the foraging algorithm on the system.

2 Literature Review

2.1 Swarm robotics characteristics

Among the papers read for the literature review, underlying features appear among the design of robot swarms.

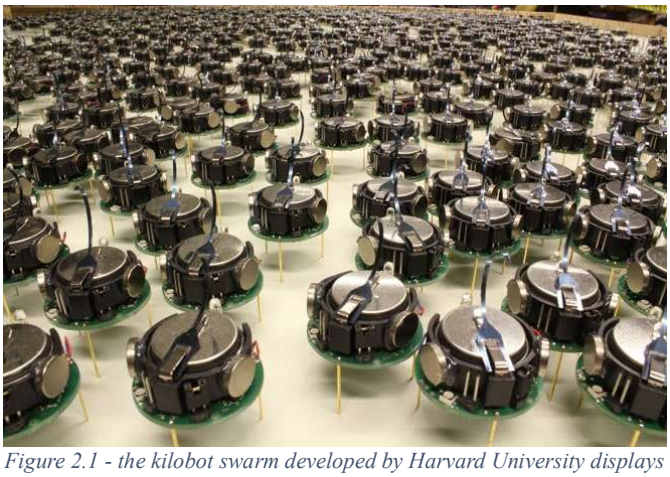


Figure 2.1 - the kilobot swarm developed by Harvard University displays all the main charactersistcs of a swarm robotics system

2.1.1 Scalability

Robot swarms usually have a minimum of three agents up to one hundred in practical applications [3]. In many cases, the swarm is simulated through MATLAB or ARGoS to simulate hundreds to thousands of agents to test and compare different swarm algorithms.

2.1.2 Homogeneity

Agents in the swarm are generally homogeneous in nature, due to the nature of research taking inspiration from biological systems in which agents that operate in a swarm are of the same species. Designing the behaviours of swarm robots becomes more manageable when all agents are the same.

2.1.3 Autonomy

All agents in the swarm can act independent of human interaction in order to complete their set task. This is both on an individual scale and swarm scale. Each agent is equipped with onboard sensors and actuators for mobility and object detection.

2.1.4 Intra-swarm Communication

The agents must be able to communicate with other agents in the swarm, directly or indirectly, in order to form collective behaviours. Direct methods have been done through RF methods such as RFID, Bluetooth, or WLAN. These direct methods can be used by agents to directly tell other agents information about its own behaviour, environment observations and location relative to the swarm. Indirect methods, also known as stigmergy, can be used by individual agents in an analogous manner to ants or bees. In this method, pheromones are left behind by agents to signal to others the shortest path to resources [4].

2.1.5 Decentralised system

As implied by the communication characteristics, robot swarms aim to utilise decentralised methods of control rather than central control, meaning if some agents become non-functional the swarm is robust enough to continue to operate.

2.2 Applications

Robotic swarm applications have been implemented in various real-world situations and industries. Some of these are explored in the following subsections.

2.2.1 High-risk tasks

Swarm robotics have been be used as detect mines on a battlefield and subsequently demine them [5] as well as scouting combat zones using multiple UASs (unmanned air systems) and UGSs (unmanned ground systems) [6].

2.2.2 Exploration and mapping

Multiple agents with limited sensing capabilities can be used to create a map of an area by collating shared information [7] or even in space exploration applications such as in the Cluster II space mission [8].

2.2.3 Emergency and rescue

A team of robots can be used in urban search and rescue (USAR) operations to search for survivors after natural disasters or structural failures. This has been implemented in projects such as SMAVNET [9] that employs 10 UAVs to assist rescuers in disaster areas and the GUARDIANS where UGVs are used in places where humans would be ineffective or prohibited. The agents in GUARDIANS are able to complete their task without direct communication by using potential fields of navigation to decide their behaviours. [10]

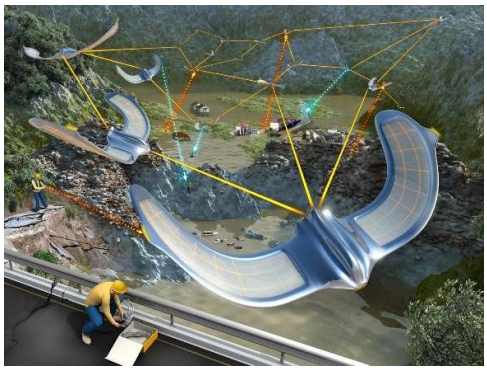


Figure 2.2 – the SMAVNET swarm system implements UAVs in search and rescue operations

2.2.4 Entertainment

UAVs are used in swarms from the hundreds [11] up to the thousands [12] to perform lightshows that use colourful LEDs and accompanied by music. This is a less accurate application of swarm intelligence however, as the swarm is centrally controlled by radio and have pre-planned movements.

2.3 Swarm robotics algorithms

Various algorithms have been created for developing swarm intelligence (SI) in the field of swarm robotics. Swarm intelligence refers to the collective behaviour that emerges from the agents in the swarm in reaction to each other and their environment. Sophisticated algorithms have emerged as a requirement of controlling swarms that perform complex tasks to optimise performance without supervision to improve their response time and noise tolerance [13].

2.3.1 Particle Swarm Optimisation

The particle swarm optimisation (PSO) algorithm is among the most popular of SI algorithms. It is an optimisation technique designed to mimic a flock of birds that communicate with each other. This is achieved by dispersing a swarm in an area to find a global minima of a problem. In the case of swarm robotics this is done by defining each agent as a particle whose motion is directed by its velocity, the agent's best position found, and the best position among its neighbours [14]. With this, the agents are able identify the location in an area that minimises a predetermined cost function.

However, in real world applications PSO is not entirely practical as: the agents of the swarm will not be able to find the actual minimum cost location due to kinematic constraints in the physical robots; the limited rate of communication among the swarm due to delays in their

software; changes in the environment that PSO does not account for; obstacles can obstruct the location of the minimum cost.

2.3.2 Ant Colony Optimisation

Ant Colony Optimisation (ACO) is a heuristic search-based algorithm inspired by the behaviours of ants in nature, where pheromones are left behind to guide other ants to a food source. In the field of swarm robotics this is done by multiple agents released into an area where a food source is known to be located. Each one leaves a marker, such as a beacon, during their pathing. This is done over many iterations until the shortest pheromone path between their "nest" and "source" is identified, in which the rest of the colony follows suit [15].

In real world applications, ACO is not completely practical due to the limit on the amount of beacons each agent can leave behind, changes in environment that will require re-optimisation of the pathing and the power consumption of each robot that may diminish before an optimal path to the source can be found.

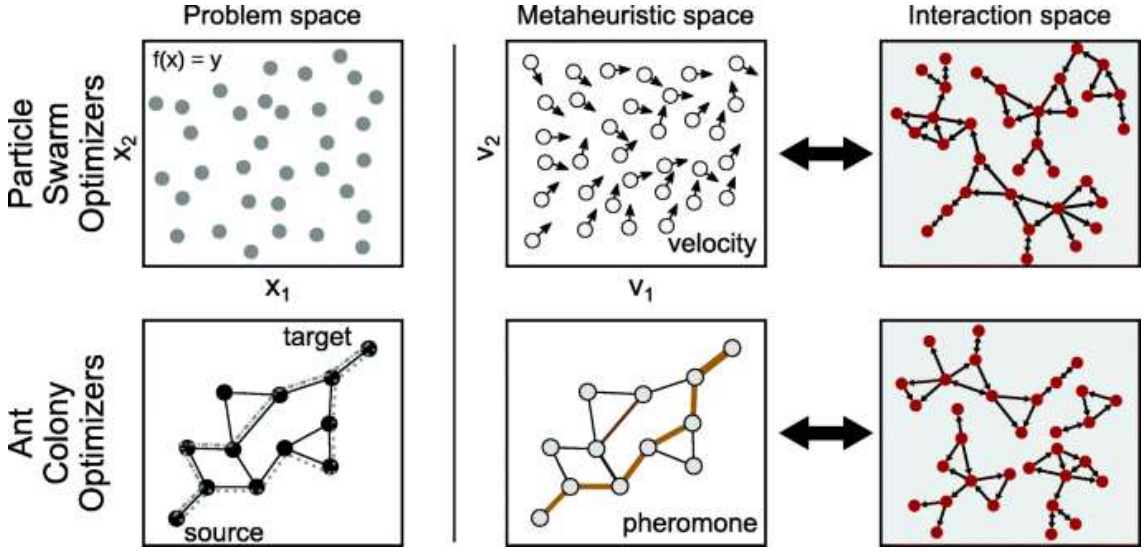


Figure 2.3 - visualisations of the popular PSO and ACO algorithms

2.3.3 Artificial Bee Colony Optimisation

The artificial bee colony optimisation (ABCO) is an SI algorithm based on the behaviour of bee colonies in which an employee bee informs onlooker bees on the hive's nectar supply, which scout bees (who are former employees) are tasked with locating and replenishing from new sources [15]. In robotic swarms this is fairly simple in execution, with scout agents locating the sources then communicating to the other agents on its location and the amount of "nectar" located at the source.

2.3.4 Swarm Gradient Bug Algorithm

An algorithm designed specifically for physical robot systems as a minimal navigation solution for exploring an unknown and unstructured environment, then returning to their departing point after a requirement has been filled.

The swarm gradient bug algorithm (SGBA) was developed by TU Delft in order to create a low resource demand navigation strategy as an alternative to Simultaneous Localisation and Mapping (SLAM) [16]. The main functions of the agents in the SGBA are to travel in a given location, detect objects and use a wall following function to travel around them, determine an

estimate of their location using odometery, avoid and communicate with other agents, and return to their departure point.

Using this method, multiple tiny flying robots are used to navigate a large unknown area in a GPS denied location, gather images with onboard cameras, and return to their starting location before onboard power is depleted.

3 Proposed Ideas

After reviewing literature on swarm robotics and considering the project specifications, three ideas were considered when developing a robotic swarm system.

3.1 Idea 1 – Swarm Drones

Based off research done at TU Delft using Crazyflie microdrones to navigate a GPS-denied area, custom drones will be created using small form factor microcontrollers as flight controllers, using 2-way RF transceivers to communicate to each other in order to navigate a room.

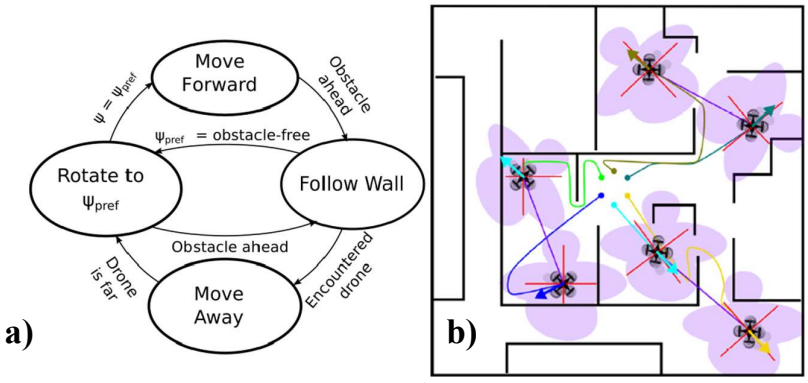


Figure 3.1 – TU Delf's implementation of SGBA to explore an unknown environment

Using Delft's paper [16] to plan the algorithm that needs to be implemented and an online tutorial on how to make an drone using a small formfactor microcontroller [17], 3-4 drones will be deployed that can navigate a room while communicating with each other to cover as much area as possible using SGBA, with autonomous flight and travel, as well as obstacle detection. If time permits a colour sensor, or possibly a camera, is to be added to detect a certain object (e.g. a red ball) that, if found by one drone, it communicates to the others that the "target" has been found, calling them over and creating a formation with the discoverer becomes the "leader" as they continue to follow the target together until it travels too fast/far to follow or the drones run out of battery, where they will then land themselves.

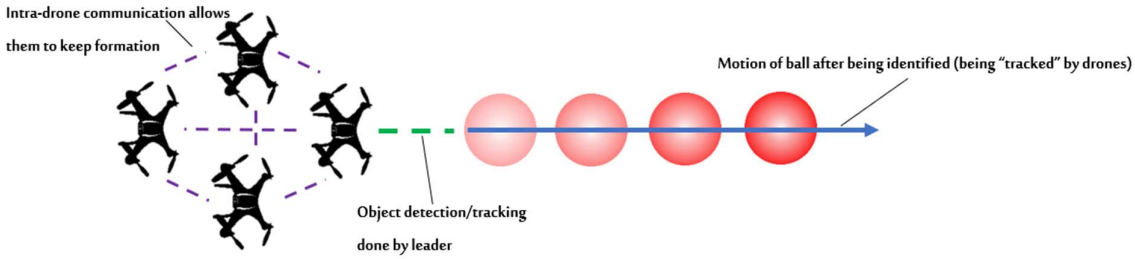


Figure 3.2

3.2 Idea 2 – Swarm flocking with UGVs

If it becomes apparent that too many challenges creating drones hinder the progress of the project, the previous idea will be similarly implemented only using UGVs and RGB sensors or cameras for object detection. A change to Idea 1 is that once the target has been found, the mobile robots will swarm to the target. For this, their relative dead reckoning from their starting position would have to be calculated from their kinematics.

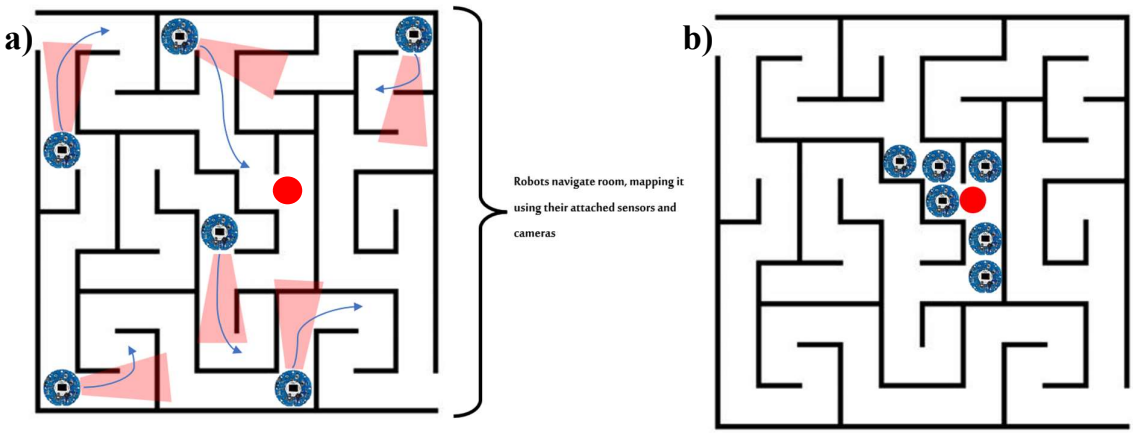


Figure 3.3

The other robots would have to know the 2D kinematics of each of the other agents in the swarm to get an estimation of their position over time. The kinematics of a 2-wheeled drive (2WD) mobile robot can be defined as:

$$\dot{x} = v * cos\theta$$

$$\dot{y} = v * sin\theta$$

Equation 1

Where $\dot{x}$ and $\dot{y}$ are the 2D horizontal and vertical positions respectively of the robot, relative to their starting position, and θ is the pose of the robot, relative to its starting position.

3.3 Idea 3 – Swarm Foraging

After developing the UGV idea, deriving and updating the odometery of each robot among the swarm proved difficult with the chosen platform, so an alternate idea was developed. In this case, there are multiple "food source" targets that are represented by coloured LEDs. Three robots are preloaded with knowledge of how many sources there are to find, and on startup use a random walk algorithm to leave their starting position – or "nest" – and explore their environment. On locating a target, the finding robot picks it up, transmits the knowledge of the target being located to the other robots through an RF transceiver, returns it to the collective starting point, then returns out to the field to continue searching for the remaining food sources. Once all targets have been found, the robots stop their current task and return to their nest.

The return position can be located by determining on wheel odometery and how many turns were taken. This is not ideal as it this does not optimise the shortest path back to the nest, but it is the most practical and achieves the task.

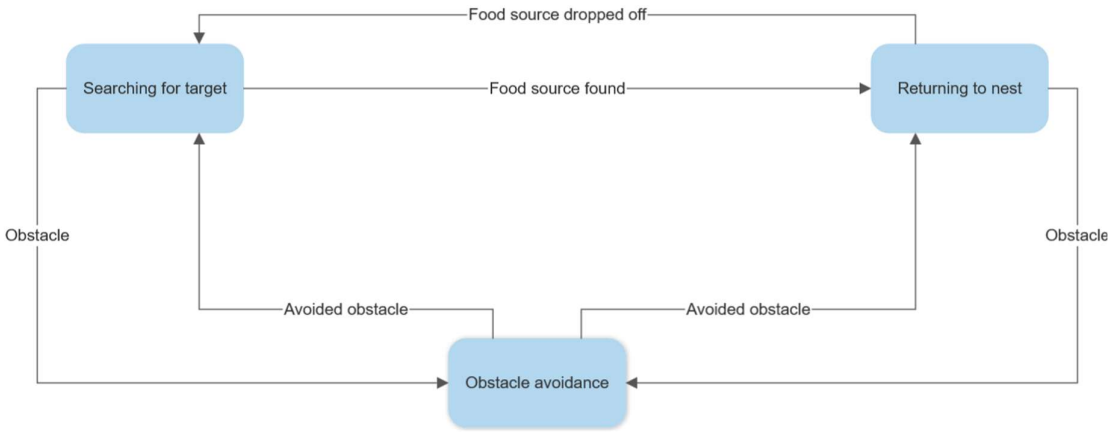


Figure 3.4 –a generic foraging algorithm based on SGBA can be found in [4]

However, adjustments to the UGV hardware and software would have to be made to be used in unstructured environments, as new obstacles can block agents' return paths. This and additional limitations are expanded upon in Section 8.2.

3.4 Project choice

Idea 3 in Section 3.3 was developed upon as the previous ideas had their own challenges that were beyond the scale of the project. These challenges are expanded upon in Section 9.

Idea 3 in Section 3.3 uses a generic, simplified version of the SGBA methodology, implementing the key components of swarm robotics that includes:

- **Homogeneity** – all robots use the same hardware and robot base in order to carry out the same task, while displaying the same behaviours.
- **Autonomy** – all robots navigate the area and identify targets without human intervention.
- **Communication** – robots communicate their current behaviour determined by if they have located a target.

3.4.1 Foraging

There are different tasks in which foraging can be implemented by using a swarm of robots. This includes:

- **Aggregation** – agents gather at a target location; useful for returning to a "nest" once foraging target objects have been found.
- **Flocking** – agents move together to location; useful for exploration or returning target objects to nest.
- **Object clustering** – agents collect target objects without having to return to a nest.
- **Chain formation** – agents form a shortest path between two locations using a chain of robots and forage via communication e.g. transporting target objects down chain.
- **Self-deployment** – agents scatter themselves in an unknown location for locating food sources or exploration of an environment.
- **Collaborative manipulation** – multiple agents work together to transport a large object that one agent would not be able to transport by themselves.

This project focuses on using the aggregation and self-deployment methods of foraging to locate target "food" sources, being coloured LEDs in this case, then carrying them back to the

nest before going out again to find other targets until a predetermined number of targets are found.

3.4.2 Applications

Applications of foraging are numerous, with some already mentioned such as in USAR and in high-risk tasks. Applications of foraging also include collective transport in order to move larger objects that a single agent would not be able to transport [18] and shortest pathfinding that can be used to locate payloads in an efficient manner [19].

4 Project Development

4.1 Swarm agents – mobile robots

The agents of the swarm are comprised of two-wheel drive robots, with their motors powered by 6V through AA batteries. Onboard components include a Raspberry Pi Pico as the microcontroller, a motor driver to power the motor and Inertial Motion Unit (IMU) for control purposes, a motor driver to power the motor, an RF transceiver for communication purposes, ToF sensors for wall detection and an RGB sensor for proximity detection to avoid objects and colour detection to identify targets.

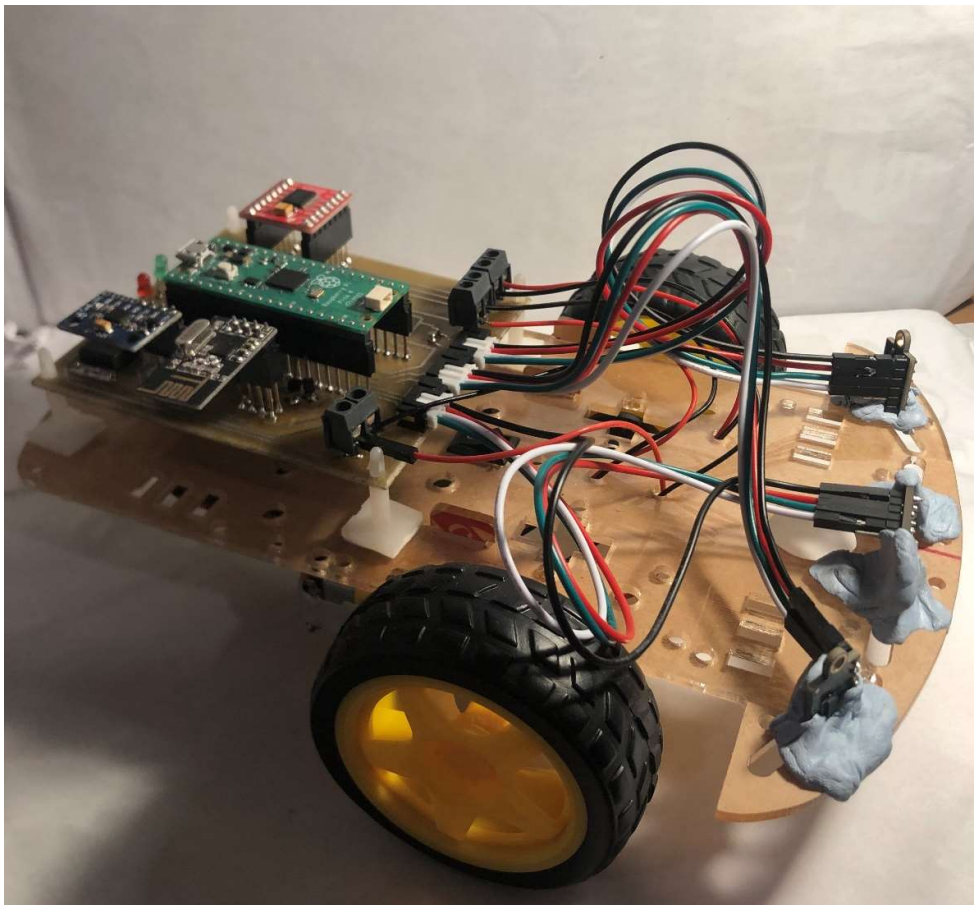


Figure 4.1 – One of the assembled homogeneous robots used in the swarm

Robot
• 220x140mm acrylic body • Raspberry Pi Pico microcontroller (3.3V input) • TB6612FNG motor driver (6V input) • 4x AA batteries
Input Sensors
• APDS-9960 colour, proximity, and gesture sensor • 2x VL53L0X time-of-flight sensors • MPU6050/MPU9250 inertial motion unit
Outputs
• 2x 1:48 6V motors • NRF24L01 2.4GHz transceiver • 2xColoured 2V LEDs

Table 4.1

4.2 Costs and Budgeting

The table below outlines the total cost of the swarm.

Part	***Component Name***	***Vendor***	***Cost per unit***	***Quantity***	***Total***
Motor driver	TB6612FGN	The Pi Hut	£6.30	3	£18.90
Chassis and motor	DIY 2WD Smart Robot Car Chassis Kit for Arduino	Amazon	£10.99	3	£32.97
Microcontroller	Raspberry Pi Pico H	Pimoroni	£5.40	3	£16.20
Time of Flight sensor	VL53L0X	eBay	£5.52	6	£33.12
IMU sensor	MPU6050	eBay	£3.45	3	£10.35
Colour sensor	APDS-9960	Mouser	£6.60	3	£19.80
RF communicator	NRF24L01	Amazon	£1.20	3	£3.59
JST cable	JST PH 2mm 4 pin to female socket cable	Mouser	£1.24	6	£4.96
JST connector	JST PH 2mm 4-pin Vertical Connector	Mouser	£0.29	6	£1.16
		Cost per robot	**£48.04**	**Total**	**£144.12**

Table 4.2

The cost for the swarm totals to £144.12, or £48.04 per agent. For the project, and through the iteration of the idea prototypes, the cost was a bit larger than the total above due to additional components being used in unused prototypes. Also, some components such as additional IMUs and batteries were provided by the university.

Unused and unpaid for components and their costs are listed in Appendix A.

4.3 Scheduling – Gantt Chart

A Gantt chart was created to schedule and keep track of progress made over the course of project.

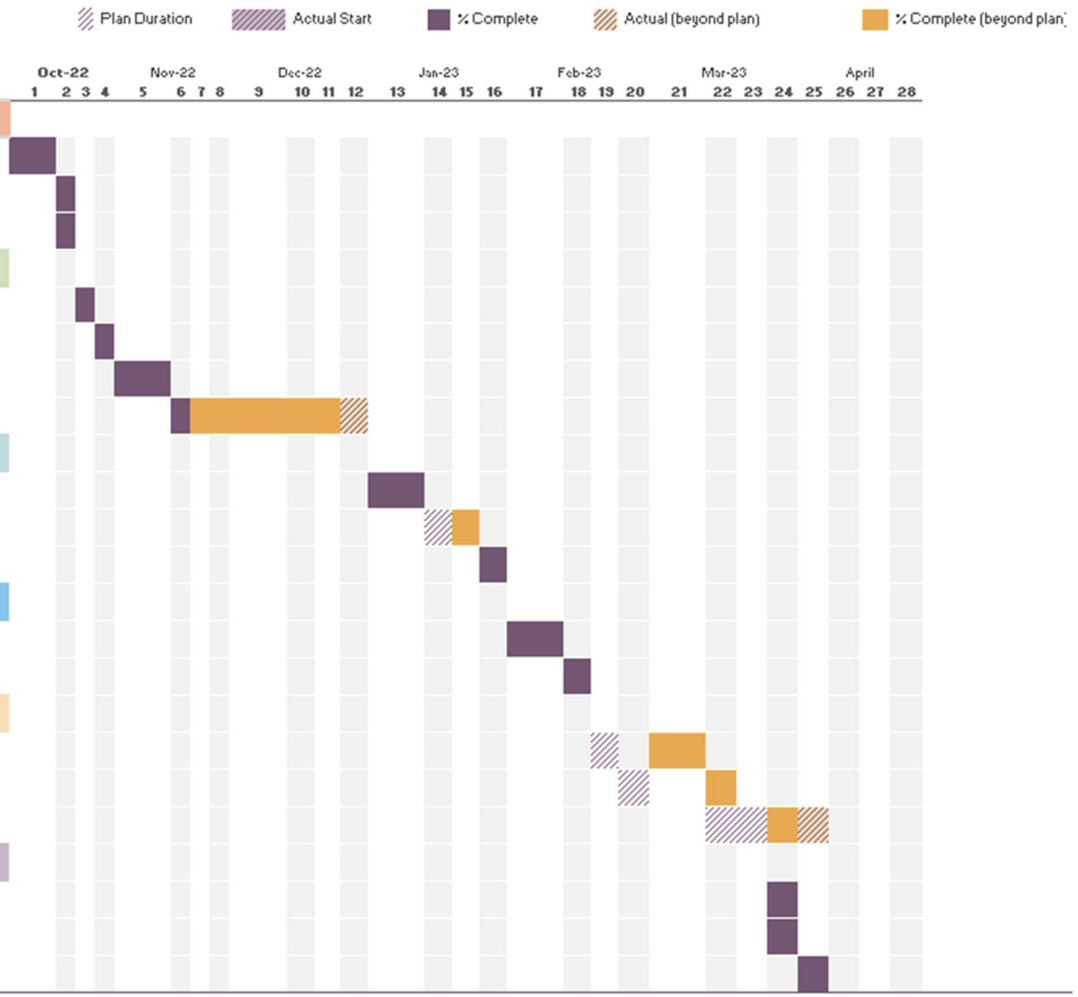
Due to the change in project ideas twice, the schedule had to be adjusted over the duration to take into account the new tasks that needed to be completed. After the Christmas break, the choice of agents to be used in the project had to be changed but did not have a great effect on the overall schedule of the project.

The main problem over the course of the project was time management and task prioritisation but overall project was able to be completed within the set time frame.

Final Year Project

Swarm Robotics — **Tolu Omitola**

TASK	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Research					
Read papers on swarm robotics theory	1	1	1	1	100%
Read articles on existing swarm robotic projects	2	1	2	1	100%
Draw up project proposition	2	1	2	1	100%
First prototype					
Determine components needed and order them in	3	1	3	1	100%
Get all sensor modules working with Pico	4	1	4	1	100%
Implement PWM with Motor	5	1	5	1	100%
Get PID controller working with IMU, ToF and motors	6	1	6	7	90%
Second Prototype					
Determine new components needed and order them in	13	1	13	1	100%
Build robot and get all sensors working with motors	14	1	15	1	100%
Implement PID and obstacle avoidance	16	1	16	1	100%
Swarm and communication					
Create other 2 robots and RF modules to each to allow for communication	17	1	17	1	100%
Create PCBs	18	1	18	1	100%
Positioning and odometery					
Implement double intergation to estimate position	19	1	21	1	100%
Use position esimate to attract other robots using RF	20	1	22	1	50%
Implement KF, EKF and IEKF to get a better estimation of poiziton	22	2	24	2	30%
Foraging					
Communicate when targets are found	24	1	24	1	100%
Retrace steps to reach nest	24	1	24	1	100%
Cooperate with swarm to find all targets and return to base once completed	25	1	25	1	100%



5 Hardware

The hardware was chosen based on the basic requirements of an autonomous robot that would be used in a swarm, focusing on low cost, flexibly replaceable/interchangeable components. This section focuses on the hardware chosen for the project and the justification for each part.

5.1 Complete Robot

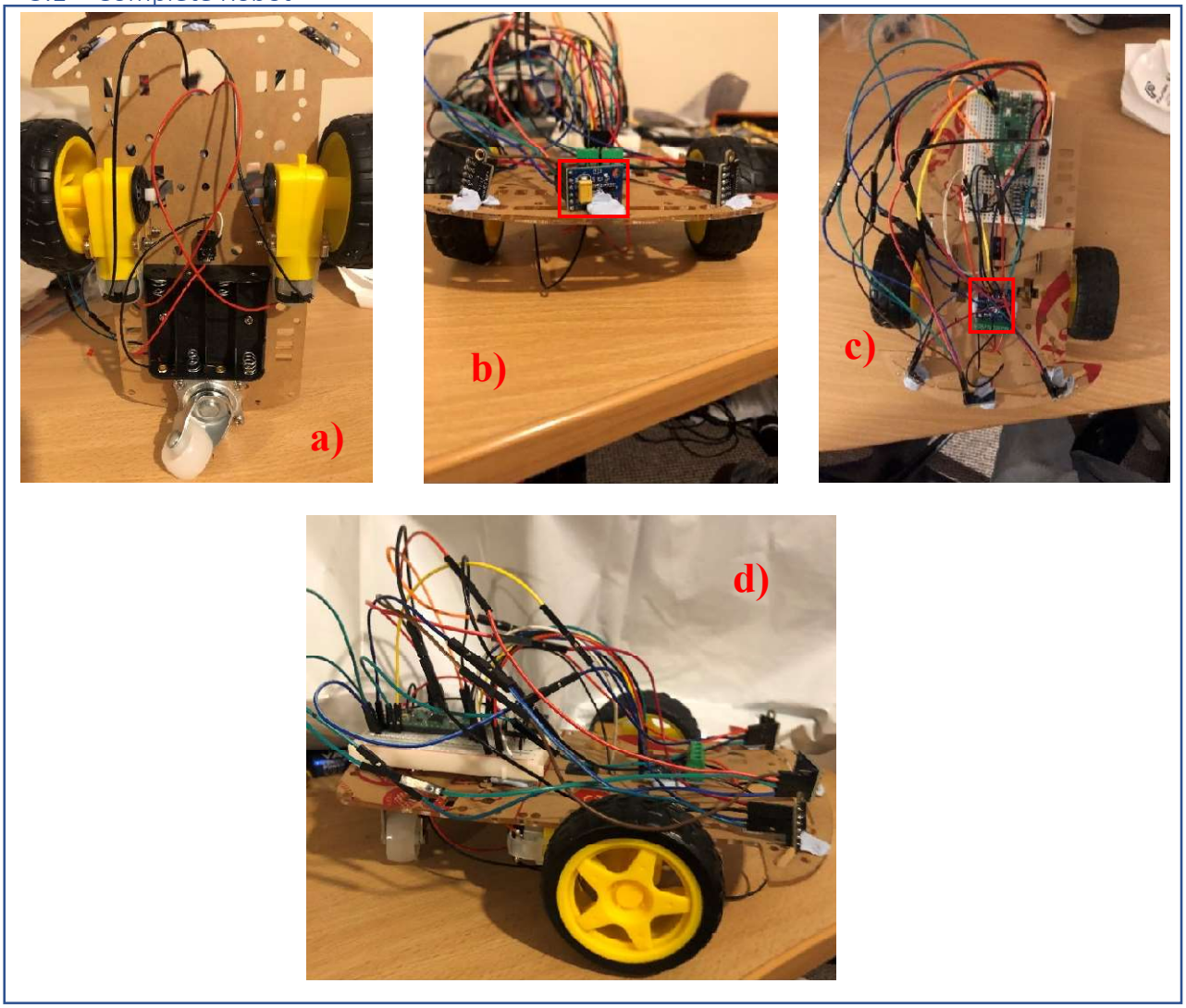


Figure 5.1 – The breadboard prototype robot showing the a) bottom, b) front, c) top and d) side views

First, a breadboard prototype was created using readily available components that were not used in the final design, but provided a good alternatives for what would eventually be used. The replaced components were an L9110S motor driver highlighted in Figure 5.1c) and a TCS34725 colour sensor in Figure 5.1b).

The final PCB was designed with modularity in mind. In this manner the components were able to be plugged in and replaced as desired. Three PCBs were manufactured for use with the swarm robots, as shown in figure 5.2.

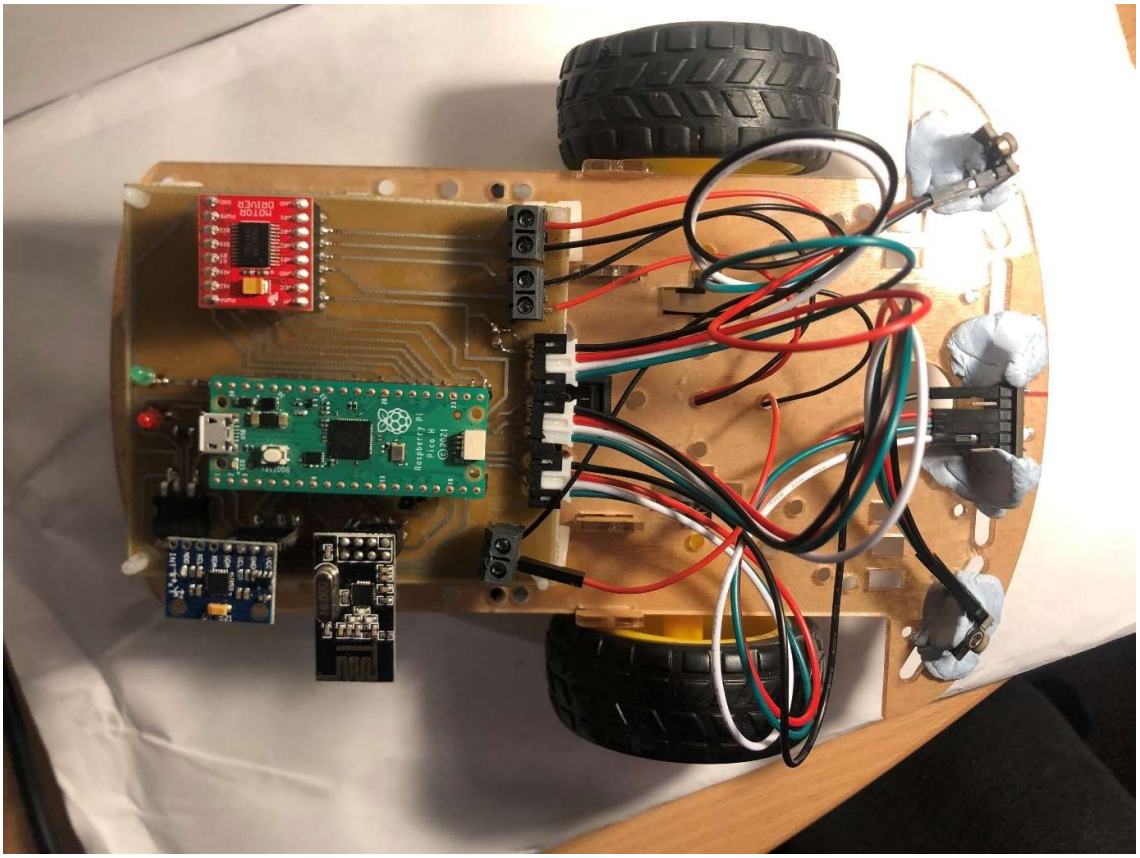


Figure 5.2 – The final PCB mounted on the robot

5.2 Required Components

The following components are based on fulfilling the basic requirements for a grounded autonomous mobile robot. A few of the following required components were chosen based on the parts left over from previous ideas but proved to be useful for meeting the requirements of an autonomous robot.

5.2.1 Chassis

The main component of a mobile robot is the chassis. This is required for the components to be placed, motors to be attached and sensors to be mounted.

5.2.2 Motors

A wheeled drive system is one of the most common methods used to control the movement of a mobile robot. A 2-wheel drive system is the simplest out of the wheeled drive methods that exist, allowing for quick and effective implementation for the planned task.

5.2.3 Microcontroller

Microcontrollers provide the main control of a mobile robot system, able to combine all inputs, outputs and determine how to process the information to control the system.

5.2.4 Voltage regulator

As the motor has a different operating voltage requirement to the microcontroller and other components, a voltage regulator is necessary to manage the power requirements demanded by the system.

5.2.5 Motor driver

As microcontrollers have a limited voltage output, and DC motors have a high current draw demand, a motor driver is a necessity as it allows for direct motor control through PWM signals using a microcontroller while drawing current directly from the main supply. This allows the motors to operate at higher RPMs than if they were to be controlled directly from a microcontroller that have output voltages of up to 5V.

5.2.6 IMU

To provide precise control of the movement of a mobile robot, an IMU is able to provide angular velocities and planar acceleration to the microcontroller, which can be used to determine its pose and speed to determine the PWM signals that need to be provided to the motors.

5.2.7 Colour sensor

In order to detect the "food sources" in the task, the robots need some way of determining if such a source has been found. RGB sensors provide a simple, low cost method in deciding if predefined targets have been located.

5.2.8 Distance sensors

For obstacle avoidance, distance sensors are useful for determining the exact distance from an object. Multiple can be used to cover different sides of a robot, and even determine angles to objects by fusing sensor information.

5.2.9 RF communication

This method of wireless communication is chosen as a requirement due to the cost and low power consumption of RF transceivers. Other methods of wireless communication such as Bluetooth, while exhibiting lower power consumption, have lower ranges and number of connections than RF. RF also does not require line of sight communication, making it suitable in an environment where obstacles are present.

5.2.10 LEDs

LEDs are used for human observers to determine the current behaviour of the robot, and not necessary for intra-swarm communication.

5.3 Chosen parts

The choice of building the robot was determined by the implementation of the first idea. As many of the necessary components needed to build the hardware for an autonomous ground vehicle had already been purchased (microcontroller, IMU, colour and distance sensors, and RF modules), it was decided that it would be cost effective to use these parts in a new robot.

The choice of components also depended on their compatibility with each other.

In this subsection, the products chosen for component are given, along with a justification for the choice made. The cost and vendors for each component can be found in table 4.2.

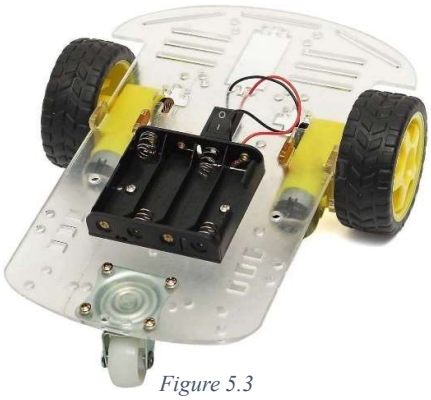


Figure 5.3

5.3.1 DIY 2WD Smart Robot Car Chassis Kit for Arduino

Features:

- Dimensions – 220x140mm
- Two gear motors – 3 to 6V DC operational
- Battery 4xAA battery pack
- Speed encoder

Justification

With a relatively cheap price tag of £10.99, the kit comes covers two of the components required in the chassis and motor, while also providing additional parts that are the encoder disks and battery pack. This kit is easy to assemble, the motors operate within a suitable operational range and the frame is strong enough to support multiple components to be mounted on top of it.

5.3.2 Raspberry Pi Pico

Features:

- Form factor – 21x51mm
- Dual-core 133 MHz ARM processor
- 264KB on-board SRAM and 2MB on-board flash
- Input power of 1.8-5.5V DC
- Programmable in C/C++ and embedded Python (MicroPython)
- 26 GPIO pins – 3 analogue inputs, 2 SPI controllers, 2 I2C controllers and 16 PWM channels.

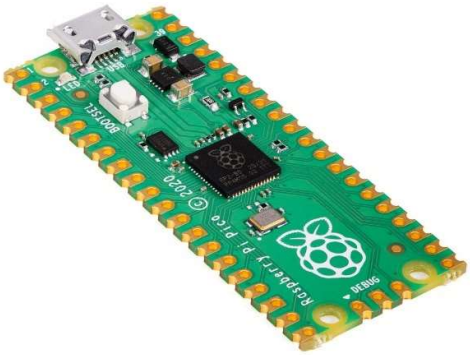


Figure 5.4

Justification:

The Raspberry Pi Pico is a microcontroller that uses Raspberry Pi's on RP2040 chip that is faster in processing speed and larger in memory than competing microcontrollers in its price range. It is well documented and supported, with many libraries being made available from a dedicated open-source community. It is programmable in both low and high level languages, making it an accessible platform to use straight away, providing the possibility of creating more time and memory efficient programs too. Due to the simplicity of MicroPython and the supported libraries for commonly used sensors and peripherals, the Pico was chosen as the microcontroller for the project as it is more efficient power wise than its competitors, that operate at a minimum of 5V, while also offering much better processing power which is useful for a robot that has to fulfil many functions during operation.

5.3.3 TB6612FNG Motor Driver

Features:

- Controls up to two DC motors
- Decoupling capacitors on supply lines
- 2.7 to 5V DC logic voltage and up to 15V DC motor voltage
- Standby control to save power
- Backwards, forwards and stop control modes
- Built-in thermal shutdown circuit and low voltage detecting circuit

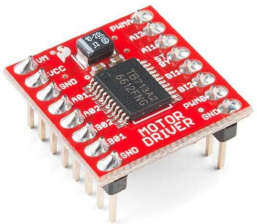


Figure 5.5

Justification:

The TB6612FNG is a motor driver that is known to be compatible with the motor driver. This allows the motors to operate at a higher RPM as they are powered by the 6V battery pack, rather than the 3V maximum output Pico. The TB6612FNG utilises MOSFETs rather than BJTs making it more than 90% efficient than BJT based drivers when supplying voltage to motors, it is a small module that requires no heatsink, and can supply up to 3.2A peak current. As the maximum operating voltage of the chosen motors are less than the maximum supply

voltage of the motors, it is perfectly suitable for the robot making larger and less efficient drivers incomparable choices.

5.3.4 MPU6050/MPU9250

Features:

- Low-cost, low-power, high performance IMU
- 3-5V operating voltage; 1.3mA idle; 10mA peak operating current draw
- 6 degrees of freedom – three gyroscope and three accelerometer outputs (9 DoF for MPU9250 that includes a magnetometer)
- Programmable for different ranges of acceleration and gyroscope to adjust precision
- I2C compatible
- Independent I2C that can be used to add external sensors to it

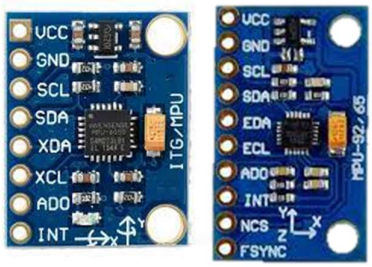


Figure 5.6 – MPU6050 left, MPU9250 right

Justification:

Due to the small form factor, low-cost and low power consumption of the MPU6050 and MPU9250, it was chosen as the IMU for the system. Although it can have noisy measurements, for the scale of the robots being used and the low requirement for accuracy, the sensor is more than suitable. The Pico also has supported libraries for both IMUs, making it easy to integrate with the robot system to provide direct yaw monitoring for PID control.

5.3.5 APDS-9960

Features:

- Small package, lower power consumption
- RGB light sensing with integrated UV-IR block filter
- I2C compatible
- Proximity detection
- 2.4-3.6V operating voltage; 0.2mA operating current draw

Figure 5.7

Justification:

A low power colour sensing module that is also supported by the Pico's MicroPython platform. Delivers a low cost solution for colour detection which will be used to determine if targets have been found. Its inbuilt proximity detection makes it useful to detect walls in front of the robot too. The colour detection is dependent on the luminosity of the object it is directed at, as the module does not feature an inbuilt LED to illuminate detected objects. This sensor was chosen due to its consistent accuracy at detecting colours compared to other tested modules at its price range.

5.3.6 VL53L0X

Features:

- 2.6-5.5V operating voltage; 10mA operating current
- Measures range up to two metres
- I2C interface, with programmable address to allow multiple sensors to be used on the same controller
- Reset and interrupt GPIO pins

Figure 5.8

Justification:

The VL53L0X is a time-of-flight (ToF) sensor which uses a laser rather than infrared or ultra sonic like other popular sensors. ToF is a lot more accurate than other types of low-cost distance ranging sensors but cost slightly more. Originally meant to be used as a ranging sensor for the first prototyped robots, this sensor proved to be useful in the final iteration as walls on the left and right sides of the robot can be detected and avoided as needed.

5.3.7 NRF24L01

Features:

- Operates in the 2.4 to 2.5GHz frequency band range
- Programmable adjustable data rates and maximum output power
- Low operating voltage and current
- Standby current
- 100m communication open space range and can operate through walls
- SPI interface
- 3-5V operating voltage; 11.3mA current draw during transmissions; 13.5mA current draw during receptions

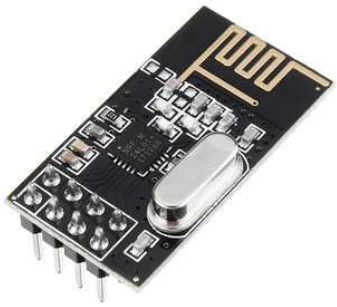


Figure 5.9

Justification:

The NRF24L01 is an RF transceiver module, meaning only one module is needed to both receive and transmit messages. It is a low cost, widely supported RF module, commonly used in microcontroller projects. Each module can communicate with up to 6 other units at the same time, making it useful for building an intra-swarm network.

5.3.8 LD33CV

Features:

- Low dropout voltage
- Output current up to 800mA
- Fixed output voltage – 3.3V
- Internal current and thermal limit
- Supply voltage rejection – 75dB

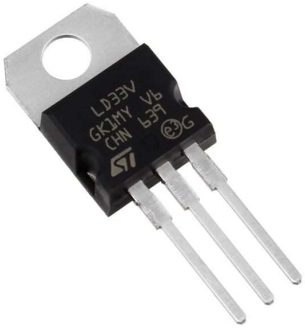


Figure 5.10

Justification:

The LD33CV was the voltage regulator provided by the Ashby workshop, which proved to be very suitable for the project. Due

to the TO-263 package format, the regulator fit nicely onto the PCB, while having a high operational temperature range, from -40 up to 150°C.

5.3.9 Kingbright2 V LEDs

Features:

- Long lifetime
- Low power consumption (maximum 105 mW)
- Standard through hole package

Figure 5.11

Justification:

Suitably visible LEDs that will be able to indicate to human observers the current behaviour of the robot and also bright enough to be detected by the APDS-9960 for object detection in testing stages.

5.4 Hardware Design

The hardware design was divided into two stages: breadboard prototyping then which was then transferred to a PCB design.

5.4.1 Breadboard design

The breadboarding stage proved to be beneficial as being able to quickly interchange components and determine faults in the system was simple, allowing for logical debugging. Figure 5.12 below shows the breadboard set up used in the prototyping stage.

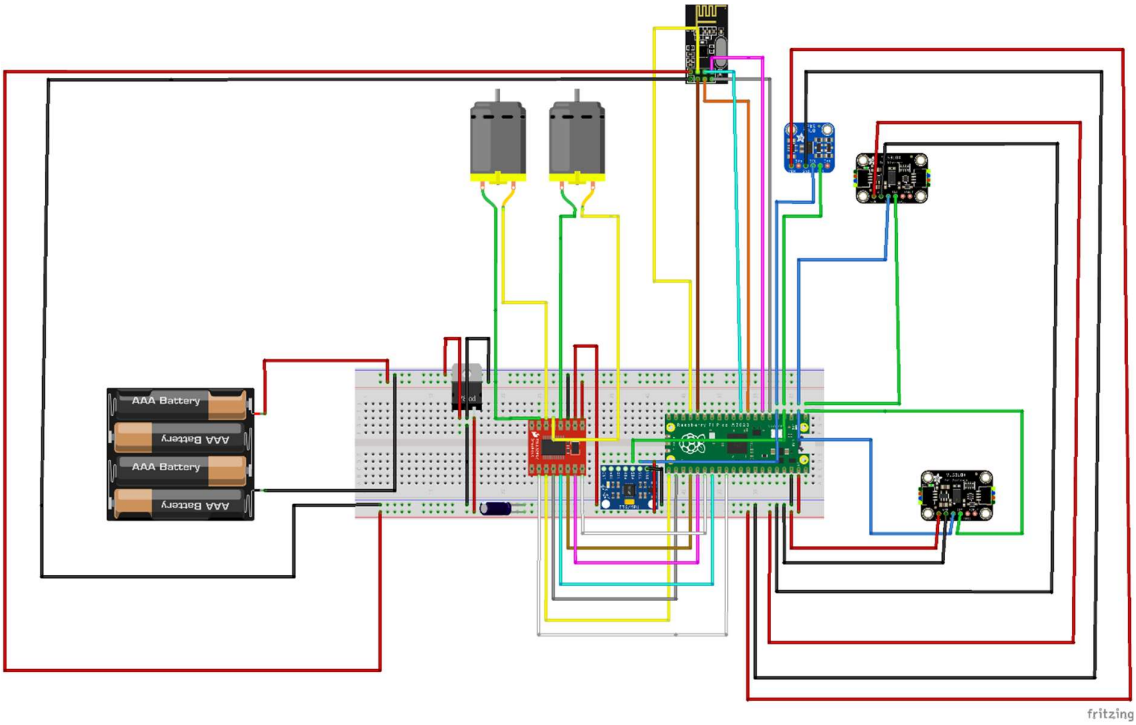


Figure 5.12 – The breadboard prototype layout using final components

However, the breadboard would not be suitable for the final design due to the large amount of wires used that can easily become loose during the operation of the robots. Power distribution

is also suboptimal when using long lengths of wire to connect components, so a Printed Circuit Board (PCB) was designed for the final system.

5.4.2 PCB design

A complete schematic for the board was designed using EAGLE with the actual components in mind so it could be visualised how they connect together. Then, using the complete schematic, a new one was created with the consideration of modularity. This schematic uses female headers and JST plugs for modules that are interchangeable. These interchangeable components include:

- Microcontroller
- Motor Driver
- IMU
- RF transceiver
- RGB and ToF sensors

The PCB board was then created using the .brd creation function of EAGLE, where the dimensions were designed to fit inside the width of the chassis while also being sort enough for the power switch button to be accessible (100mm from switch to chassis edge). A ground plane was included to incorporate a common ground among the components without having to worry about trace widths.

Below is the schematic for the circuit and the PCB design. The header-based schematic can be found in the Appendix B.

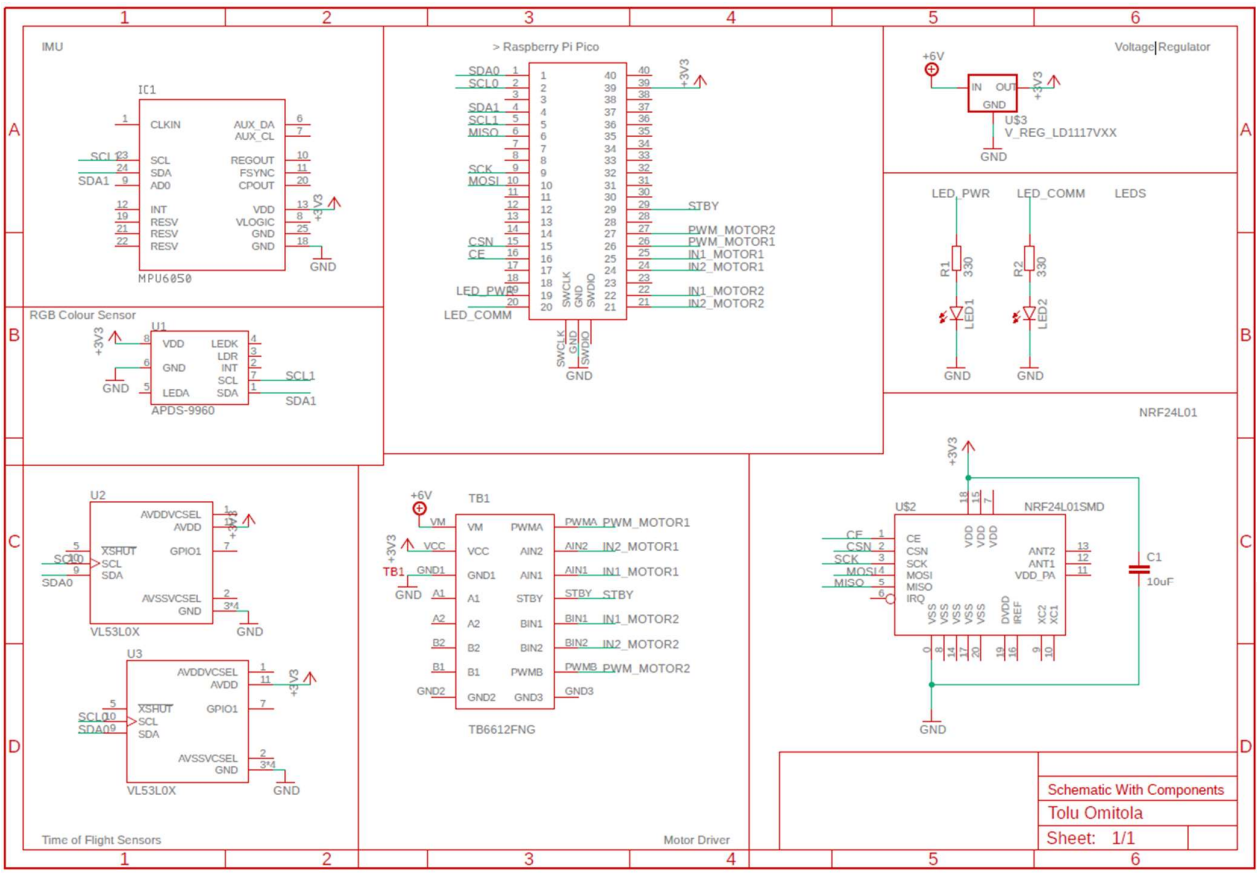


Figure 5.13 – Eagle Schematic layout of circuit design

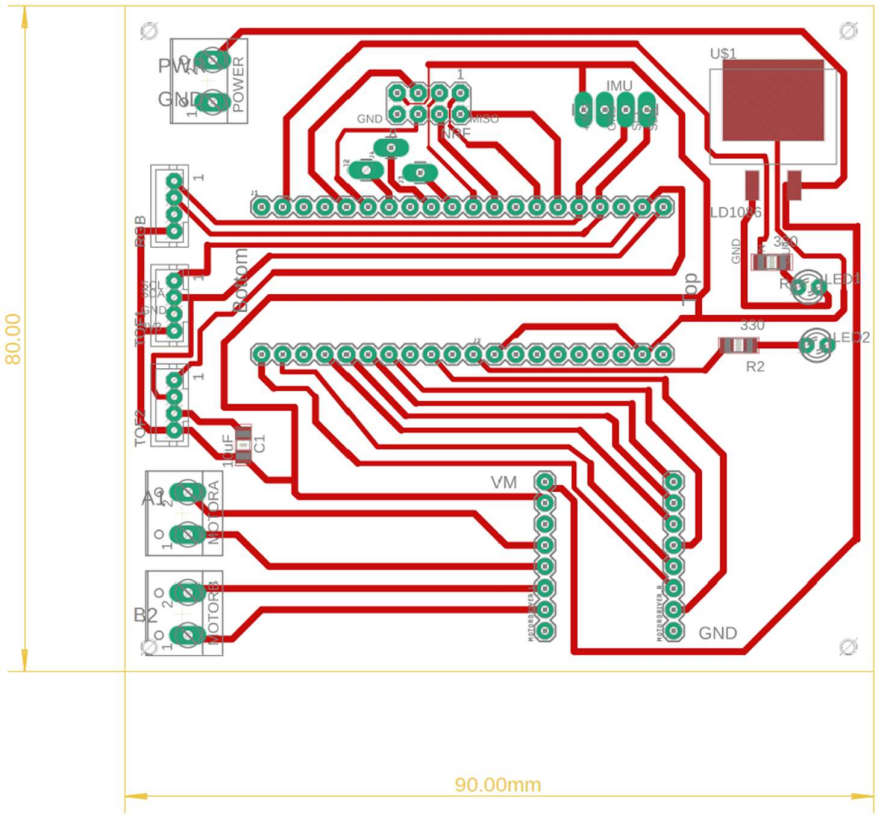


Figure 5.14 – Eagle PCB layout that uses headers for replaceable components

5.5 Review of hardware design

With the PCB printed, less hardware-based errors occurred during further testing, and with the modular design, switching out components became simple. Creating additional robots to add to the swarm also became easier as the there was less set up time.

To put together the PCB, a soldering iron, solder and flux paste was used to solder:

- Female headers – for the microcontroller, motor driver, RF module and IMU
- Screw terminals – for the power supply and motors
- JST headers – for the RGB and ToF sensors
- LEDs
- Voltage regulators
- Capacitors and resistors

5.5.1 Possible modifications

A noticeable problem with the hardware was that no metal fixings were used to mount the input sensors, with adhesives used as a substitute. Using suitable metal brackets to fasten these in place would be a provide a better alternative.

Figure 5.16 – bracket that can be used to fasten sensors onto robot

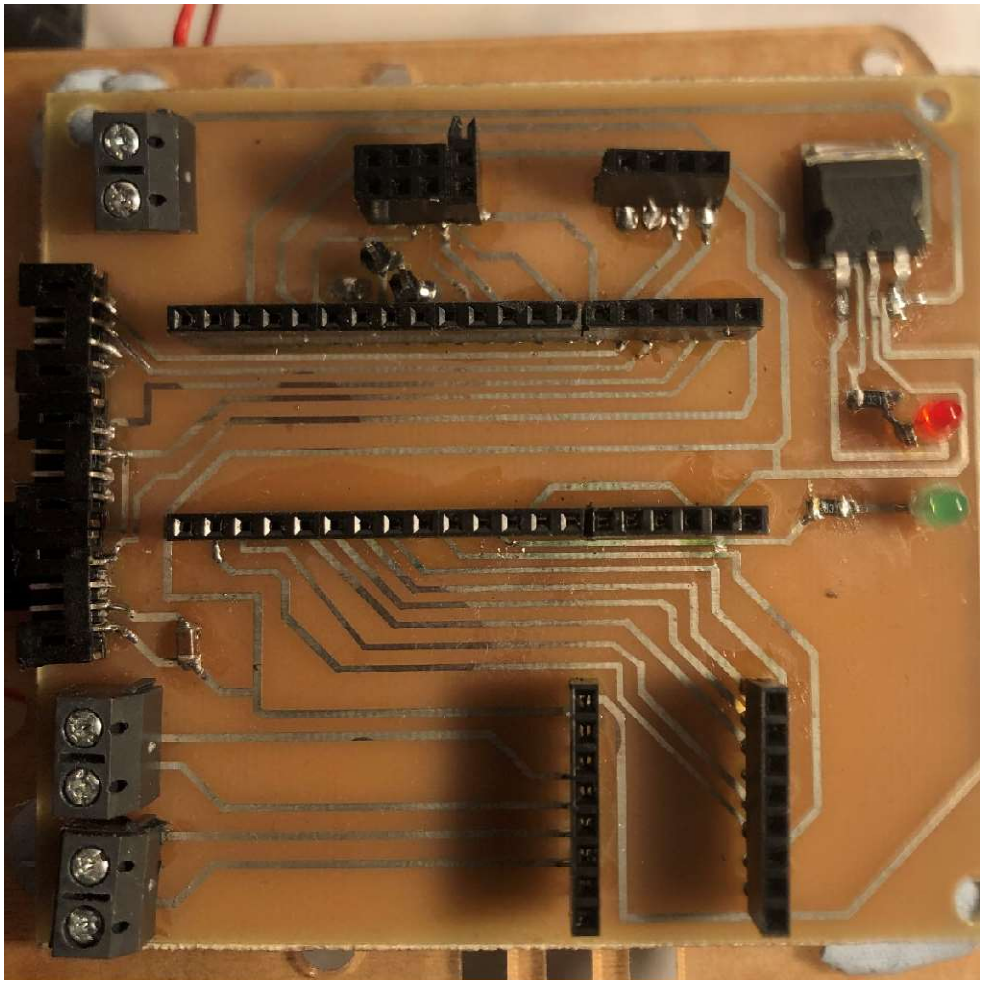


Figure 5.15 – Final Printed Circuit board with resistors, capacitor, screw terminals, headers, LEDs and voltage regulator soldered on

The robots also do not have any way to transport the targets back to the “nest”. Incorporating a non-moving or a mechanical scooping mechanism would provide a modification that can be made to the hardware to make transporting the targets possible. This is further discussed in Section 10.

6 Software

In this section an overview of the software used for the swarm is given. The methods used are described at a high level with flowcharts for visualisation.

6.1 IDE

The chosen integrated development used during the project for writing, managing, and uploading the code was Thonny. This is due to its seamless integration with MicroPython on RP2040 chips. Other viable IDEs suitable for the Raspberry Pi Pico are Visual Studio and Pycharm. However, the additional features included in these IDEs are essentially syntax verifiers that can identify errors in code before they are uploaded. Any issues that occur at runtime are shown in their debugging windows after execution, a feature which Thonny also provides. Because of this, it was decided that using Thonny would be less time consuming and much simpler to use.

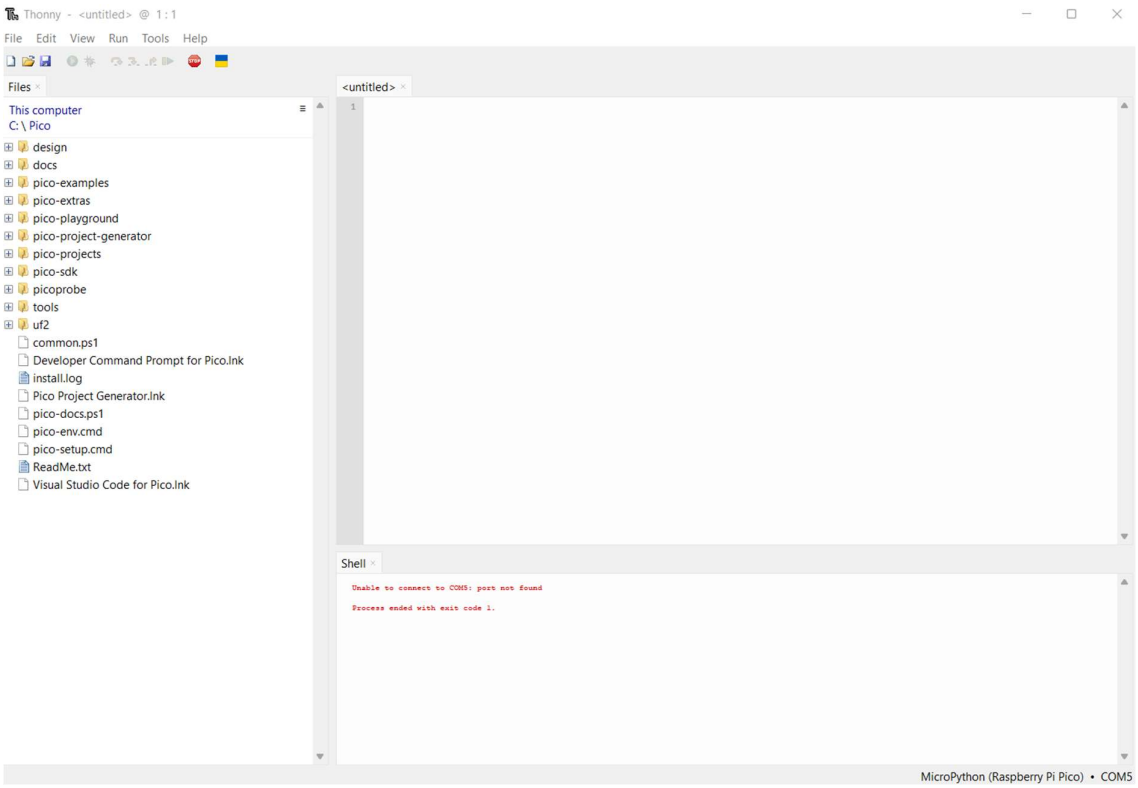


Figure 6.1 – The Thonny IDE

6.2 Overview of functions

The functions of the swarm can be summarised as follows:

- **Main loop** – the main loop executes the code until targets have been found, then executes a separate a new set of code once all targets are located. Utilises all the following functions below.
- **RF functions**
  - setup() – sets up the transceiver for use.

    - sendColour() – one of the agents sends the targets it has found and receives a response on the rest of the swarms colours, then sends another message to update the swarm on all targets found.
    - recColour() – the rest of the agents send their targets to the head of the swarm and receives a message containing the targets found by the rest of the swarm..
- **Movement control**
    - forwardControl() – calculates the PID values needed to adjust the speed of each motor, so the agents move in a straight line.
    - moveForward() – causes the agent moves forward.
    - moveBack() – causes the agent to move backwards.
    - turnR() – causes the agent to turn right.
    - turnL() – causes the agent to turn left.
    - turnAngle() – provides accurate turning based on the gyroscope measurement from the IMU sensor to turn a given angle left or right.
    - stop() – causes the agent to stop moving.
- **Sensor functions**
    - wallDetectLeft() – uses ToF sensor to detect walls on left of agent.
    - wallDetectRight() – uses ToF sensor to detect walls on right.
    - wallDetectFront() – uses the APDS-9960's proximity sensor to detect walls on in front of agent.
    - detectColours() – uses the APDS-9960's RGB sensor to detect the coloured targets.
- **Foraging functions**
    - updateTargets() – uses the aforementioned RF functions to update the agent on the targets that have and haven't been found.
    - returnHome() – returns the agent home once it has located a target or once all the targets have been located by the rest of the swarm.
    - resetPathing() – clears the array used for pathing back to the nest once has executed its returnHome code

6.3 Function implementation

6.3.1 Initialise heading

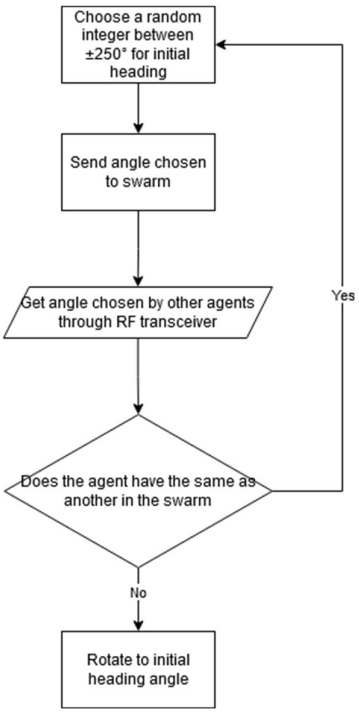


Figure 6.2

On start up the agents wait until they have each received an expected signal form the rest of the swarm. Then, they swivel at an angle between ±250° from their initial one, make sure that they do not share a starting heading angle with another agent, and then start the execution of the main loop.

6.3.2 Main loop

In the main loop, the agents move forward until they encounter an obstacle in their path.

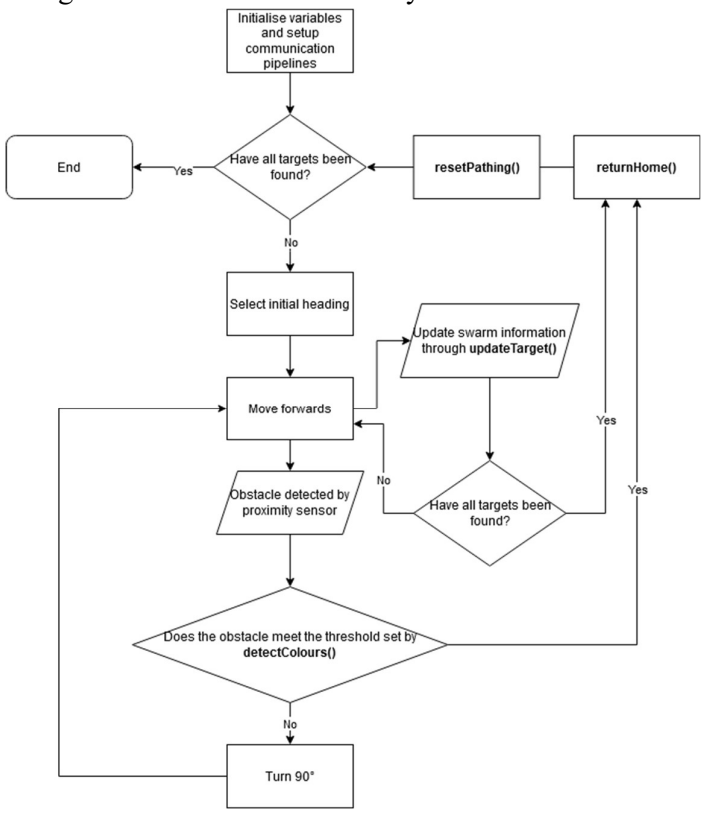


Figure 6.3

After checking if the colour of the object matches that of an expected target, they either turn right 90° and continue moving forward or execute their return home code. The targets in this case are two coloured LEDs – one red and one blue. The main loop uses the **detectColours()** method to determine if the threshold for determining a coloured LED has been met.

If an object is detected to close to the left or right of an agent, it continually adjusts its driving in the opposite direction by one degree until it is no longer within range of detected object.

All the while they detect if walls are too close to the left or right of the agent. At each turn they store how long they have been travelling for before reaching an obstacle, that is used for retracing a path back to the nest. All the while, the **updateTargets()** method runs during operation so the swarm stays updated on the amount of targets found.

6.4 updateTargets()

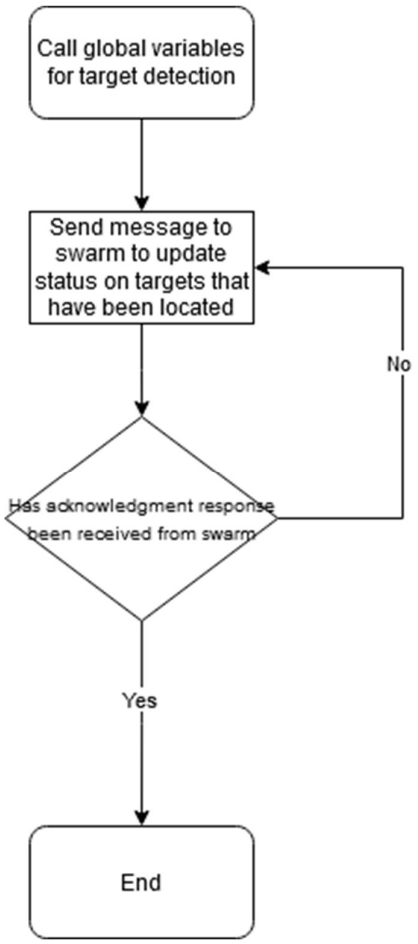


Figure 6.4

The updateTargets method updates each agent on the targets that have been located by the swarm by an agent sending the swarm a message of the targets it has located and receiving a message back about the targets the rest of the swarm has located. After returning to the nest and while out on the field, the agents are able to determine if they need to continue their search, return to the nest or stay put depending on the message received by the rest of the swarm.

6.4.1 returnHome()

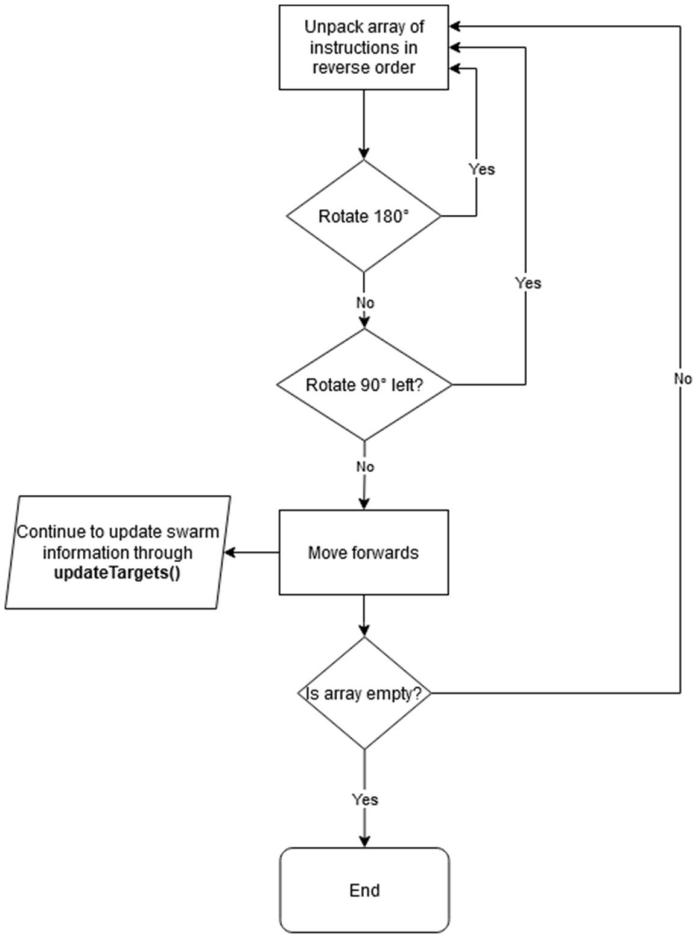


Figure 6.5

The returnHome method uses the array created by the agents while searching for their targets. Upon executing the method, the array is unpacked. As mentioned earlier, when the agent encounters an object, it stores the time it took to reach the object while moving forwards which is stored in the pathing array. Once the main loop executes the returnHome method upon finding a target or receiving the message that all targets have been located, the agents execute the array in reverse order, turning left where it went right originally. If an obstacle is detected before the array has finished its current move forward command, the method interrupts it, moves left to avoid the obstacle, then moves onto the next instruction. The **resetPathing()** method is called immediately after the returnHome method in the main loop. This clears the array and allows new pathing values to be stored when the agent goes searching again.

6.5 Review of software design

Overall, the implementation of the software was successful, allowing a level of swarm intelligence to be displayed among the agents as they worked together to achieve the task. There are improvements that can be made to the software in order to create a more effective system.

6.5.1 Modifications

Although the agents each have a different starting heading, they only continue to travel straight until they encounter an obstacle, in which case they only turn right. A change would be to have the agents turn randomly left or right depending on the ToFs' sensor information.

In larger spaces that have no obstacles, the food sources may never be found by the agents if they only turn when encountering walls. By including a random change in direction every few seconds, the agents are more likely to locate food sources even in lightly obstacle environments.

Another problem is that only one agent determines when messages are sent. This means that if this agent ceases to function, the remaining agents can longer communicate with the rest of the swarm. Having a failsafe that triggers in each agent after no messages have been received for a certain amount of time, an agent takes over the initial sending message role.

The return pathing only accounts for navigating in structured environments. This means that if a new obstacle is added to the environment as it returns home, the agent is not able to return properly to its nest. Using a wall following algorithm that utilises the mounted ToF sensors, the agent can travel around the edge of an obstacle while the active return home instruction remains suspended until the newly encountered object has been manoeuvred around.

7 Testing

After the hardware had been implemented for the agents and software had been developed for the swarm, a structured test environment was setup to test the functionality of the swarm. Tests are carried out to determine if the swarm behaves as laid out in Section 3.3.

7.1 Test Setups

The initial setup is placed on a table of dimensions 200x100 cm. The nest is placed on the edge of the table, with the "food sources" placed at locations 50 to 150 cm away from the nest. The food sources are 3V coloured LEDs – one red and one blue.

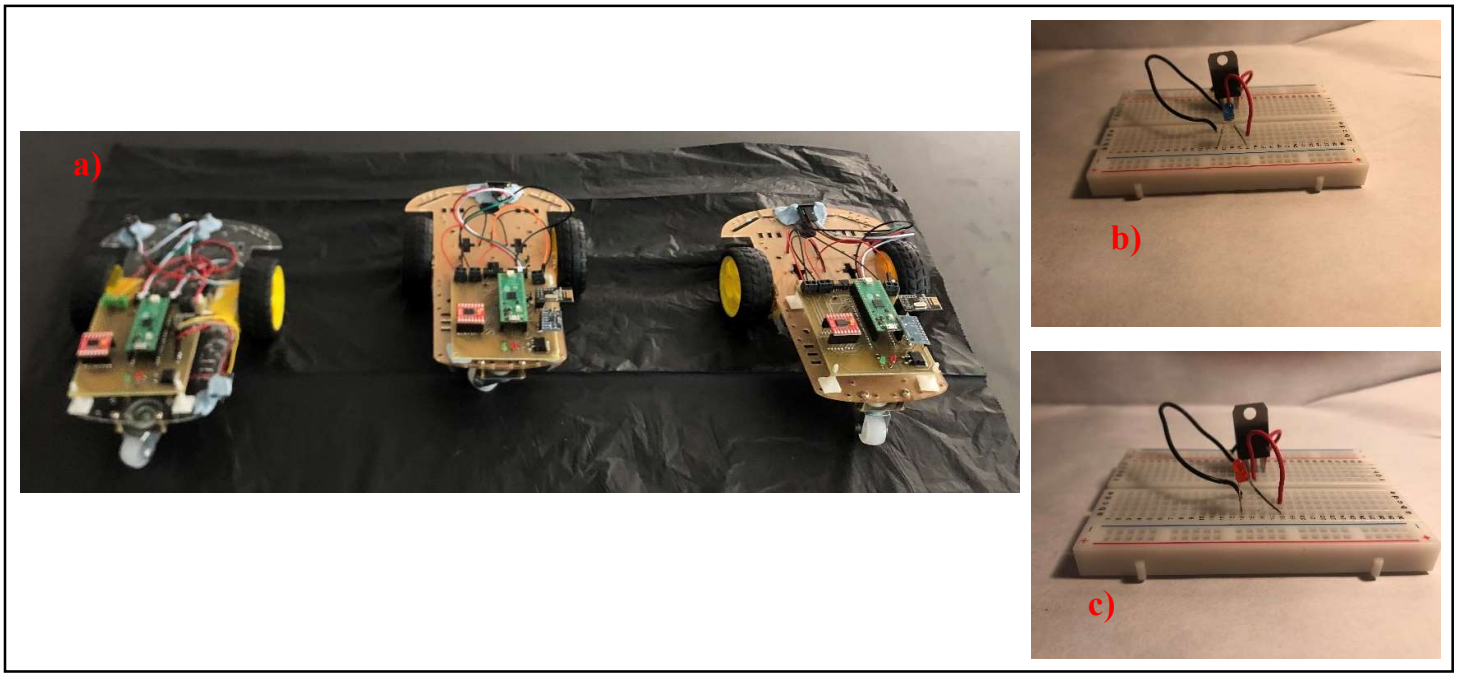


Figure 7.1 – a) the three robots/agents at nest, b) the blue LED "food source" and c) the red LED "food source"

7.2 Results

Because of the test setup, the robots did not execute their startup code in which they rotate to select a random heading. Instead, they continue to drive straight, turning right when an obstacle is detected in front of them.

In the first test, agents in the swarm encounter no obstacles with the targets placed directly in front of the agents' paths. Once the two sources are located, the swarm is required to return to the nest.

Figure 7.2 – The swarm advancing forwards, with the right-most agent returning to the nest after finding the red food source

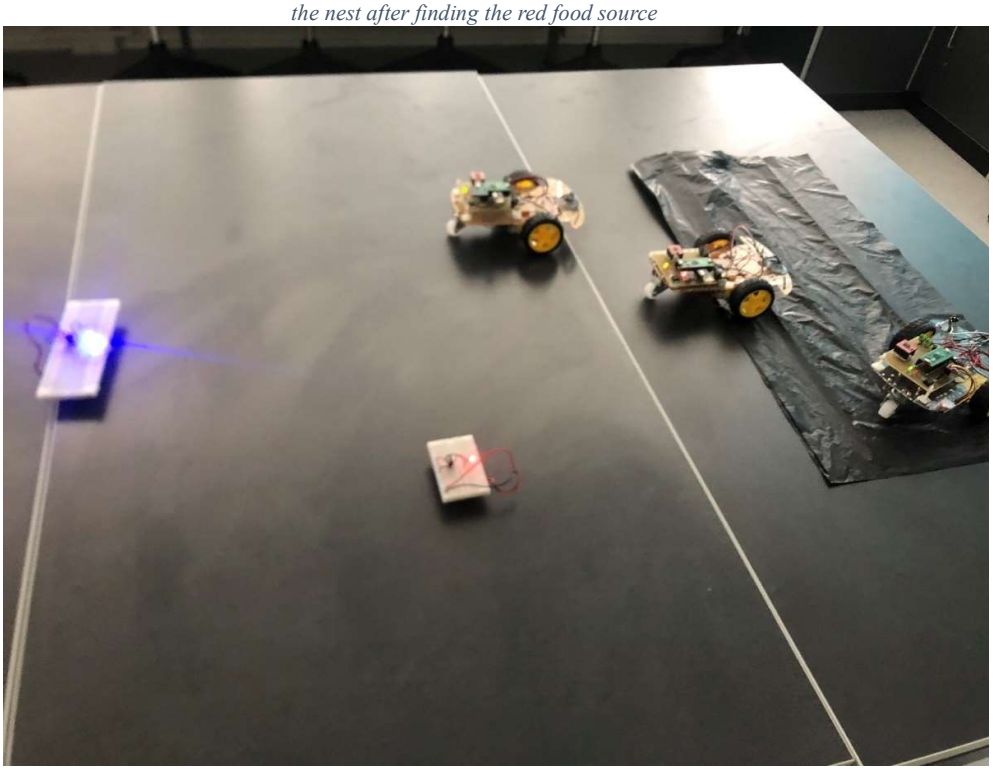


Figure 7.3 – all agents returning to nest after all sources have been found

In the second test, agents in swarm encounters obstacles as they travel in a straight line. By turning right, they are able to avoid obstacles and are able to path their way back to the nest once targets have been found while avoiding obstacles on their return path.

Figure 7.4 - Structured environment where the agents avoid the obstacles to locate the food sources before returning to the nest

7.3 Review

After testing, the swarm behaved exactly as expected. The tests carried out were limited however, as they did not account for operating in a larger environment, including more agents in the swarm or encountering more obstacles on their outgoing or returning path. Because of this, the third aim of the project specification was not fully achieved.

In Section 10, improvements are suggested to create a more refined laboratory-based platform to demonstrate the full capabilities of the swarm.

8 Limitations

Although the project was able to display the main requirements of a robot swarm, there are many limitations to the system that inhibit the full potential of the system. These limitations are discussed in this section.

8.1 Colour sensor

The colour sensor is a basic module that can only detect illuminated objects. This APDS-9960 does not include an inbuilt white colour LED like similar sensors such as the TCS34725, which illuminates objects so the colour of any reflective object can be identified.

Due to illuminating objects having to be used, the test environment must not allow in an natural light, as it can change the set thresholds for detecting colours that need to be

identified which greatly affects the performance of the swarm as they may never locate the targets.

As mentioned in the hardware section, a camera would provide a better solution for object detection as they are able to detect objects regardless of their reflection. However, this does raise the cost of each robot, as even the cheapest monochrome, QVGA resolution camera for the Raspberry Pi Pico costs £16 [20] – around 2.5 times more than the APDS-9960. As the flash memory storage on the Pico is only 2MB, not many images can be stored and images will have to be stored and processed every time an object is detected. As a machine learning library is required to process these images, a significant amount of storage space will be taken up, meaning an SD card will have to be added to increase the flash storage of the microcontroller. This also increases the cost of each robot, so selecting an appropriate colour sensing module or defining a new detection process can be considered.

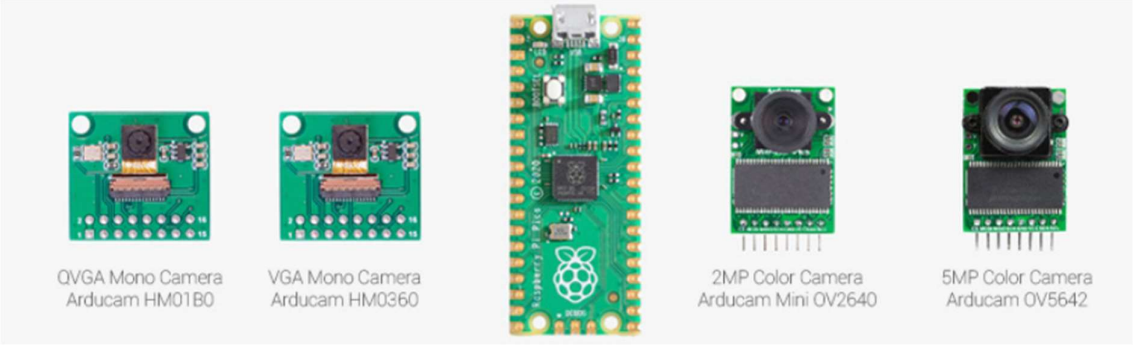


Figure 8.1 - some cameras compatible with the Raspberry Pi Pico

8.2 Pathing

A flaw in the return pathing method is that it uses time travelled forward at before reaching an object. Due to python being an interpreted language, there can recorded time is usually slightly longer than the actual travel time due to delays in the software caused by compilation into bytecode before interpreting and execution. This is circumvented by using wall detection methods in the return path to avoid agents running into obstacles if they receive mistimed statements, but agents can overshoot the nest. Using optical source encoders with the provided encoder disk in the 2WD robot kit, the total rotations of the wheel can also be processed by the microcontroller. The values can be used as a complimentary variable when pathing back to the nest and also controlling the speed of the motors to provide better control over the agents' movements. There are LM393 modules that provide cheap speed measuring methods for the agents, with units costing £1.50 each [21].

As mentioned earlier, a random pathing algorithm would need to be implemented in order to provide better probability in the food sources being found. This means changes to the code needs to be improved in order to create a more sophisticated return pathing algorithm, that takes the shortest path back to the nest after locating a source.

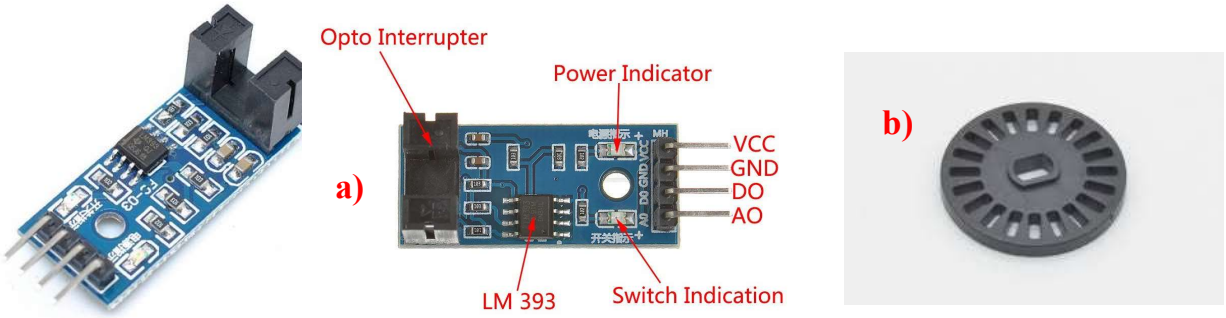


Figure 8.2 – a) an LM393 module that utilises an opto- interrupter that detects light that passes through encoder disk in b)

Also, for unstructured environments, where changes can occur in the expected return path a wall following method needs to be added. Utilising the right and left mounted ToF sensors, the agent can travel around the edge of an obstacle while the active return home instruction remains suspended until the newly encountered object has been manoeuvred around. With encoders, the length of the evaded objects can also be recorded and then subtracted from the current return path instruction.

8.3 Partially centralised system

A main requirement that was not entirely fulfilled was achieving decentralised communication among the swarm. This is because only one agent receives the messages sent by the other agents and updates the entire swarm on the targets located by itself and the rest of the swarm. If this agent stops operating, the other agents are not able to communicate with each other. By implementing a failsafe, this limitation can be circumvented. If the senders have not received reply from the "master" agent after a certain amount of time, one of the agents can take over the role of the "master" by changing its address pipelines. The next master can be decided by an internal priority system in the agents. The flowchart below shows how this could be implemented:

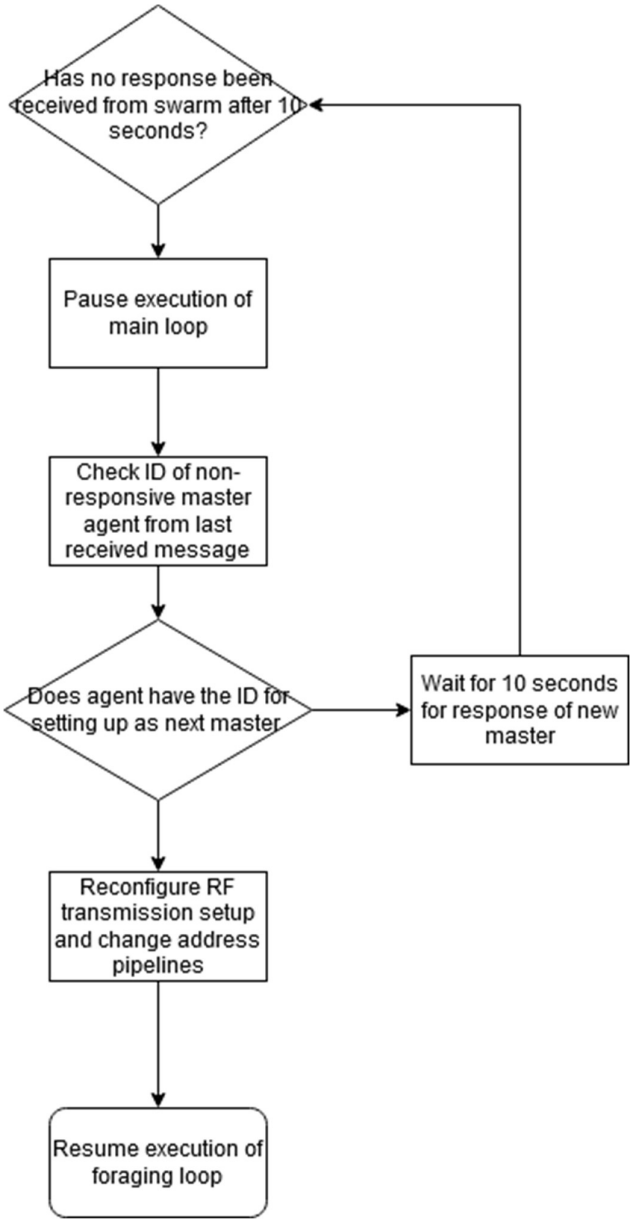


Figure 8.3

9 Development Challenges

During the course of the project, obstacles arose with both the hardware and software that hindered or halted the progression of idea. This meant that the implementation of the project idea had to be changed twice. This section covers the issues encountered, and why the ideas had discarded for new ones.

9.1 First idea prototype

The first idea outlined in section 3.1 incorporated drones to be used as agents. The first few weeks of the project was dedicated to building these drones. To do this, the drone was to be first prototyped on a stripboard before a manufacturing a PCB. This presented problems surrounding power management to the components. The initial problem was that the microcontroller unable to read the values from the IMU due to the current draw of the motors being fairly large (3.2A at 3.7V), and both the IMU and motors were being powered by the Pico's output voltage pin (up to 300mA at 5.5V). This was solved by adding an external battery to power the motors separately. However, when connecting all 4 motors, the noise from the motors meant that the IMU would stop outputting data for long periods of time or stop working all together. This was solved by adding filtering capacitors to the IMU input voltage and the input voltage of the motors. The values of the capacitor were based on the Arduino nano quadcopter circuit schematic design [17].

This allowed for constant readings from the IMU while all 4 motors were being powered. Next, the PID values used to stabilise the drone had to be calculated, so a linear system was simulated to obtain PID values using MATLAB's parrot minidrone toolbox [22] and a tutorial [23] but after a time the IMU could no longer read values when the motors were in use, even after adding a filtering capacitor for the power supply. It was decided that the best way to solve this would be to print a PCB and order extra components required to manage the power and current switching that the drone needed to consistently and reliably work. Because of uncertainty in how long this would take to achieve, the choice to switch over to a simpler robot design was made, as too much of the project time would have been dedicated to creating an autonomous drone rather than creating a swarm.

The simulation implementation and results, as well as the drone code and created stripboard prototype can be found in Appendix D.

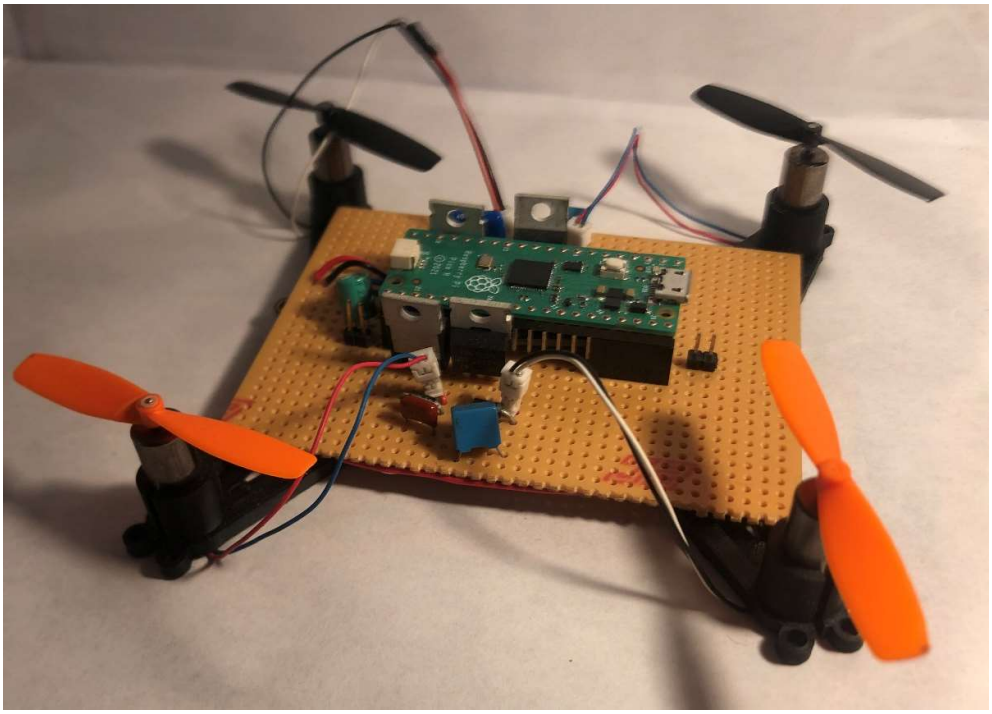


Figure 9.1 – Unfinished drone developed for idea 1

9.2 Second idea software

The second idea outlined in Section 3.2 proposed to use dead reckoning odometery to get the relative position of the robots using their IMUs. This was inspired by a paper that uses Convoluted Neural Networks (CNN) to calculate the noise parameters of an Invariant Extended Kalman Filters (IEKF) when predicting the position of a sensor [24].

The major discovery of this paper was that, by using their implementation of dead-reckoning, the authors could achieve accurate position tracking of an vehicle using solely an IMU. This is an achievement as IMUs are noisy, so double integrating their accelerometer readings to calculate their position causes drifts greatly over time even while the sensor is stationary. In order to get the dead-reckoning of a vehicle, sensor fusion is typically used to get the position of an object, e.g. using GPS or a camera in conjunction with an IMU to get the position of vehicles.

The initial approach to implement this was to use TinyML or TensorFlow lite libraries to create the convoluted neural network model, as the weights used had been made publicly available by the authors. However, implementation of the libraries requires using a Linux-based platform and using C/C++ as the programming language. As the project up to that point had been developed entirely in MicroPython on a Windows platform, the change would have been time consuming as all robot functions would have to be recoded. In addition to this, the Pico's sensor libraries were limited and less supported in C++ than its MicroPython equivalents.

Focus was then dedicated to using Extended Kalman Filters (EKFs) to calculate the position of the robot to decide if it was a suitable solution by itself.

First the position was calculated using the raw sensor values, and as expected large drift is observed when stationary. Then using an implementation of Kalman filters designed for the MPU6050[1], the position was plotted again. Although the drift was a lot less erratic when using the Kalman Filter, the drift was still large.

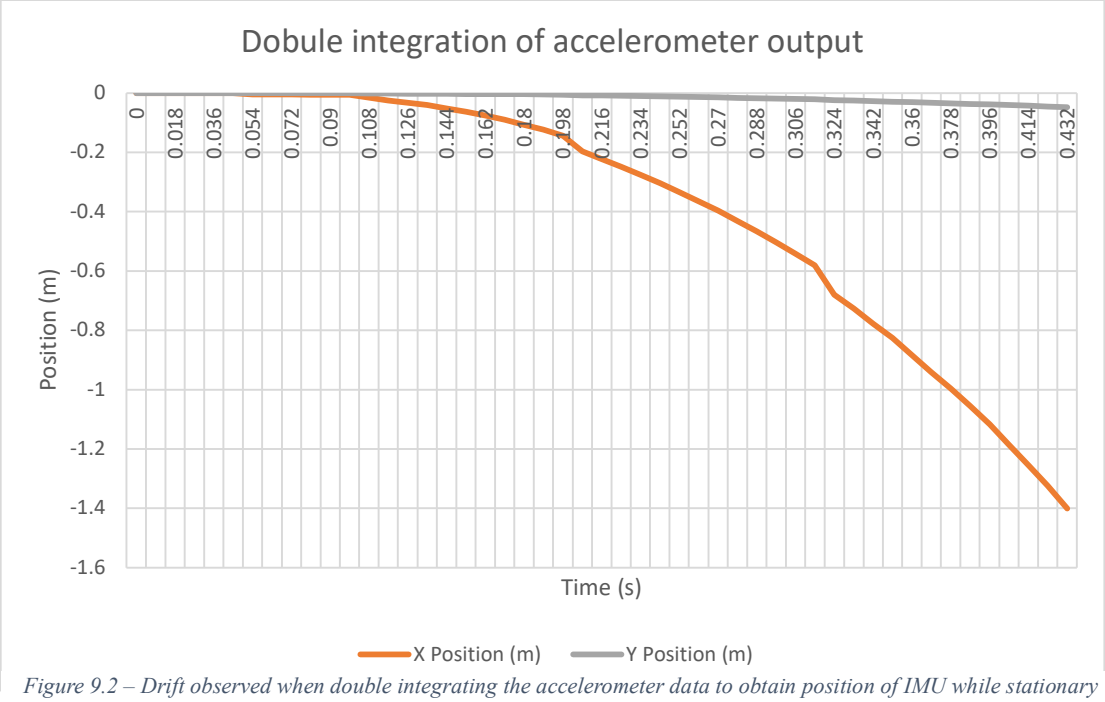


Figure 9.2 – Drift observed when double integrating the accelerometer data to obtain position of IMU while stationary

[1] https://github.com/rocheparadox/Kalman-Filter-Python-for-mpu6050

Because of the large drift still observed when using a Kalman Filter, the idea to get the position of the agents was discarded and the third idea was developed instead. Code used can be found in the Appendix D.3.

9.3 Radio inconsistency

A problem encountered earlier in development was using the NRF24L01 for radio communication. The packet loss encountered when trying to send strings between two modules would be 100% at the default settings. However, after adding decoupling capacitors to the power rails and changing the baud rate to 2M, packets were able to be transferred at a much better rate of around 50% packet loss. However, when using an external power source rather than using the Pico's power supply, the packet loss dropped to 0%. This is likely due to the shared components on the Pico's power supply line and only a problem encountered during breadboard prototyping, as the PCB design uses a shared power line for all components and microcontroller.

10 Future Improvements

10.1 Transporting sources

As mentioned in Section 5.5.1, the robots also do not have any way to transport the targets back to the "nest". A non-moving scooping mechanism would provide a simple way of loading the "food sources" onto the robot which they can then drop off at the nest. Depending on the shape of the load a more advance mechanism such as a mechanical arm or shovel can be used to by the robots to carry loads in a more sophisticated manner, however, a more advanced target sensor would have to be used to utilise the mechanical part well. A suitable type of sensor would be a camera, but an alternate microcontroller would have to be used instead of a Raspberry Pi Pico as the memory and storage size are too small. A Raspberry Pi Zero or similar microcontroller would be suitable.

However, the cost of including a mechanical collector method, a camera sensor and a more advanced microcontroller would cause the cost of each agent to increase by a large amount. Keeping it simple with a non-moving shovel should be suitable for modification.

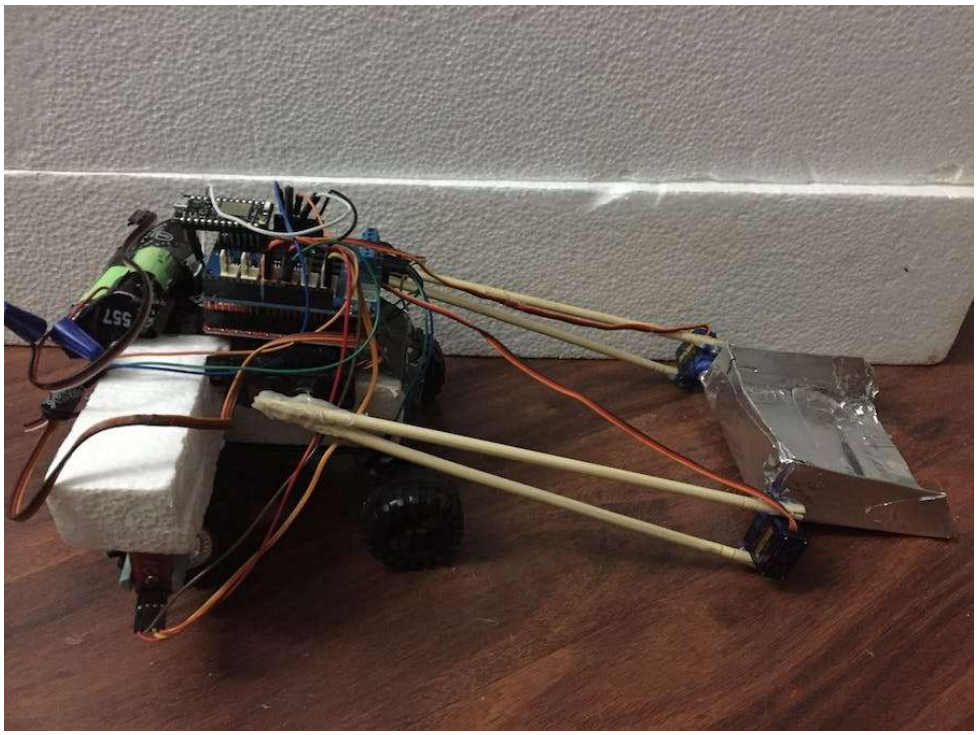


Figure 10.1 – Possible shovelling mechanism that can be used to scoop targets once located. RGB sensor to be counted on shovel

10.2 Sophisticated return algorithm

As mentioned in the Section 8.2, a more sophisticated return algorithm could be implemented to find the shortest path back to the nest. There are many ways to implement this, as detailed on an MIT website [25].

In summary, to be able to plan a shortest path between two points, the agents need to be able to build a map implicitly to create occupancy grid maps that discretise a space into squares and assigns each square a probability of it being occupied or empty. This map can be used as a graph for tree search algorithms, such as Dijkstra's algorithm or the A* algorithm that can find the shortest path to a location given unlimited time and resources, or sample based planning that use rapidly-exploring random trees to search spaces by expanding until the target point is found, and even machine learning that use reinforcement algorithms like Q-learning to make the best decisions on optimal routes to take during an agents return pathing.

10.3 Laboratory-based platform

Creating an appropriate laboratory-based platform is important to assess the full capability of the swarm system. Below in figure 10.2 suggests how a better system can be made.

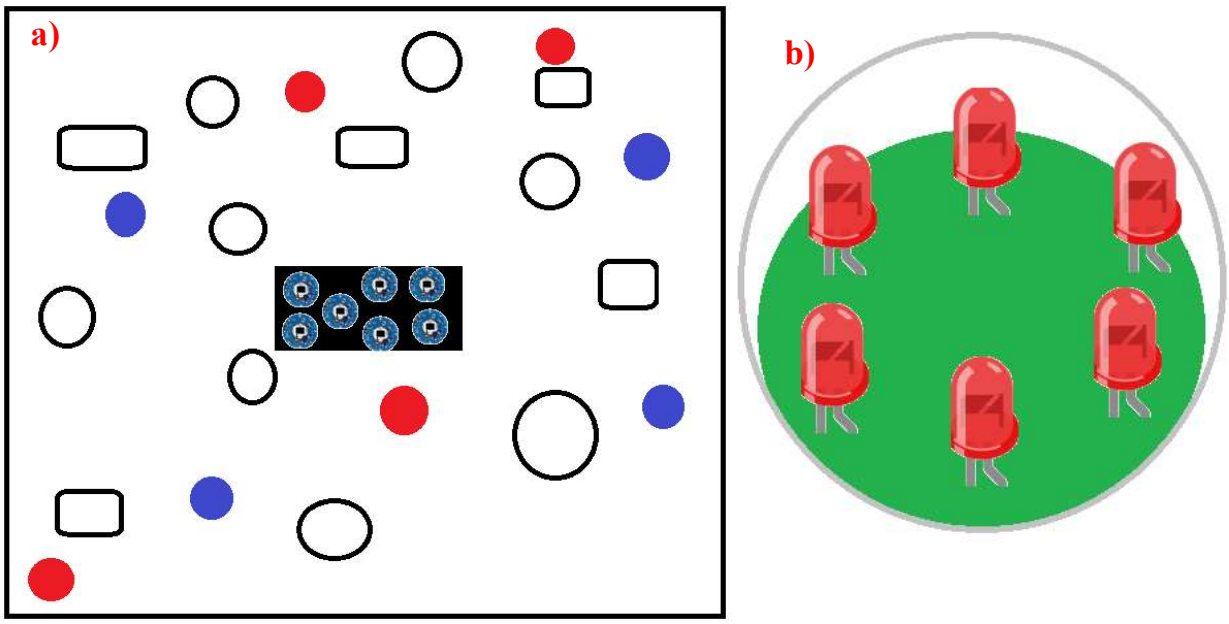


Figure 10.2 – a) the laboratory based platform features the nest placed in the middle of the environment, more obstacles of various shapes and sizes placed in the room and four "food sources" of each colour scattered around the room behind and between obstacles; b) shows the modified source that features more LEDs housed in a case

This platform incorporates:

- **More obstacles** – more obstacles are placed within the area so random walk does not have to be implemented, only an initial heading direction. These are arranged to indirectly guide the agents to the targets
- **More agents** – more agents reduce the amount of time searching for targets and show the scaling capabilities of the swarm
- **More food sources** – using multiple of the sources will display the capability of agents being able to take multiple journeys to and from the nest
- **Easier to detect food sources** – this achieved a custom set up that uses a ring of 6 LEDs placed on a PCB and placed inside translucent housing. This is increases the luminosity and size of the beacons so that the food source is easier to detect.

- **Wider area** – this is used to test the capability of the return pathing algorithm and how effective it can be even in larger areas. Also decreases the possibility of agents taking the same path as the traverse the environment.

11 Conclusion

The aim of the project was to develop a swarm robotics system that displays the typical characteristics of a swarm that is capable of operating in structured and unstructured environments.

A literature review was carried out to research implementations of a swarm system, with findings used to inspire ideas that could be implemented into a physical system. Although the project idea had to be changed twice, implementing a working system was achieved within the given time frame.

A swarm of three robots was created, capable of traversing an environment, detecting food sources and returning to the nest once an individual agent found a target or when all targets were located. Communication was achieved by using RF transceiver modules with object detection achieved using RGB and ToF sensors.

Challenges with developing a suitable idea proved to be a time consuming part of the project, but lessons learned at each at each stage proved to be useful for developing the final system. These challenges were the power distribution issues faced when creating a drone and RF communication when first implementing a swarm of UGV robots. The return pathing algorithm, all though mostly consistent, can fail at times due to using timing for determining how long each robot travels. This can be circumvented by adding additional hardware such as using encoders to improve the accuracy of travel measurements.

Limitations of the system do exist as target detection is inconsistent while using colour sensors, but a more sophisticated setup was suggested as well as changes to the hardware which would improve consistency but also increase individual robot cost. Decentralised communication was not fully realised in the system due to the operation of the NRF24L01 transceivers, however, this is a known limitation within the field of swarm robotics due to limitations of current communication architectures [26]. This can be circumvented for by using additional software that accounts for communication failure among the swarm as suggested. The swarm traversing unstructured environments was not fully explored, but using a wall following algorithm with the implemented hardware is an feasible task.

Overall the project was a success, but more testing and software modifications are required to explore the full capabilities of the system. Suggestions are made for the tests that can be implemented to test the potential of the system and software that can be written to improve the performance of the swarm.

References

Bibliography

[1] “Swarm robotics,” Wikipedia, Aug. 13, 2022. https://en.wikipedia.org/wiki/Swarm_robotics

[2] S. Edgar, “Navigation in Swarm Robots,” AZoRobotics.com, Sep. 20, 2012. https://www.azorobotics.com/Article.aspx?ArticleID=36

[3] H. Wang and M. Rubenstein, “Shape Formation in Homogeneous Swarms Using Local Task Swapping,” IEEE Transactions on Robotics, vol. 36, no. 3, pp. 597–612, doi: https://doi.org/10.1109/tro.2020.2967656.

[4] A. Pradhan, M. Boavida, and D. Fontanelli, “A Comparative Analysis of Foraging Strategies for Swarm Robotics using ARGoS Simulator,” IEEE Xplore, Jul. 01, 2020. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9202671

[5] V. Kumar and F. Sahin, “Cognitive maps in swarm robots for the mine detection application,” IEEE Xplore, Oct. 01, 2003. https://ieeexplore.ieee.org/document/1244409

[6] L. Rudd, “OFFensive Swarm-Enabled Tactics,” www.darpa.mil. https://www.darpa.mil/program/offensive-swarm-enabled-tactics

[7] P.-Y. Lajoie and G. Beltrame, “Swarm-SLAM : Sparse Decentralized Collaborative Simultaneous Localization and Mapping Framework for Multi-Robot Systems,” arXiv:2301.06230 [cs], Jan. 2023, Available: https://arxiv.org/abs/2301.06230

[8] A. Rahmani, S. Bandyopadhyay, F. Rossi, J.-P. De La Croix, J. Hook, and M. Wolf, “Space Vehicle Swarm Exploration Missions: A Study of Key Enabling Technologies and Gaps,” 2019. Available: http://publish.illinois.edu/saptarshibandyopadhyay/files/2020/07/IAC_Blue_Sky_v1.pdf

[9] H. Sabine, L. Severin, Z. Jean-Christophe, and F. Dario, “Laboratory of Intelligent Systems - SMAVNET Project,” lis2.epfl.ch. http://lis2.epfl.ch/CompletedResearchProjects/SwarmingMAVs/

[10] Intel Technology Innovation, “Queen of Drones: Revolutionizing Night Sky Light Shows,” Intel. https://www.intel.co.uk/content/www/uk/en/technology-innovation/article/queen-of-drones-revolutionizing-night-sky-light-shows.html

[11] J. Saez-Pons, L. Alboul, J. Penders, and L. Nomdedeu, “Multi-robot team formation control in the GUARDIANS project,” Industrial Robot: An International Journal, vol. 37, no. 4, pp. 372–383, Jun. 2010, doi: https://doi.org/10.1108/01439911011044831.

[12] dpreview staff, “EHang uses 1000 GhostDrone 2.0 drones for massive light show in China,” DPReview, Feb. 25, 2017. https://www.dpreview.com/articles/5881540638/ehang-uses-1000-ghostdrone-2-0-drones-for-massive-light-show-in-china

[13] J. Pugh and A. Martinoli, “Multi-robot learning with particle swarm optimization,” Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems, May 2006, doi: https://doi.org/10.1145/1160633.1160715.

[14] “A comparison analysis of swarm intelligence algorithms for robot swarm learning,” ieeexplore.ieee.org, Jan. 08, 2018. https://ieeexplore.ieee.org/document/8248025

[15] M. M. Shahzad et al., “A Review of Swarm Robotics in a NutShell,” Drones, vol. 7, no. 4, p. 269, Apr. 2023, doi: https://doi.org/10.3390/drones7040269.

[16] G. L. Herranz, S. Hauert, and S. Jones, “Decentralised Negotiation for Multi-Object Collective Transport with Robot Swarms,” IEEE Xplore, Apr. 01, 2022. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9784801

[17] Montvydas, “Arduino Nano Quadcopter,” Instructables. https://www.instructables.com/Arduino-micro-Quadcopter/

[18] S. Adams, D. J. Ornia, and M. Mazo Jr, “A Self-Guided Approach for Navigation in a Minimalistic Foraging Robotic Swarm,” arXiv:2105.10331 [cs], Sep. 2021, Available: https://arxiv.org/abs/2105.10331

[19] K. N. McGuire, C. De Wagter, K. Tuyls, H. J. Kappen, and G. C. H. E. de Croon, “Minimal navigation solution for a swarm of tiny flying robots to explore an unknown environment,” Science Robotics, vol. 4, no. 35, Oct. 2019, doi: https://doi.org/10.1126/scirobotics.aaw9710.

[20] “QVGA SPI Camera Module for Raspberry Pi Pico (HM01B0),” The Pi Hut. https://thepihut.com/products/qvga-spi-camera-module-for-raspberry-pi-pico-hm01b0

[21] “Youmile 5Pcs Speed Measuring Sensor LM393 Speed Measuring Module Tacho Sensor Slot Type IR Optocoupler for MCU RPI Arduino DIY Kit with Encoders : Amazon.co.uk: Business, Industry & Science,” www.amazon.co.uk. https://www.amazon.co.uk/Youmile-Measuring-Optocoupler-Arduino-Encoders/dp/B0817H9436/

[22] “Parrot Minidrones Support from Simulink,” uk.mathworks.com. https://uk.mathworks.com/hardware-support/parrot-minidrones.html (accessed Jan. 07, 2023).

[23] “Drone Simulation and Control - Video Series,” uk.mathworks.com. https://uk.mathworks.com/videos/series/drone-simulation-and-control.html (accessed Jan. 07, 2023).

[24] M. Brossard, A. Barrau, and S. Bonnabel, “AI-IMU Dead-Reckoning,” IEEE Transactions on Intelligent Vehicles, pp. 1–1, 2020, doi: https://doi.org/10.1109/tiv.2020.2980758.

[25] “Robotic Path Planning,” Path Planning. https://fab.cba.mit.edu/classes/865.21/topics/path_planning/robotic.html

[26] M. Schranz, M. Umlauft, M. Sende, and W. Elmenreich, “Swarm Robotic Behaviors and Current Applications,” Frontiers in Robotics and AI, vol. 7, Apr. 2020, doi: https://doi.org/10.3389/frobt.2020.00036.

Appendices

Appendix A: Additional Costs

Part	Component Name	Vendor	Cost per unit	Quantity	Total
		Unused			
Propellers pack of four	Walkera ladybird	Drone Junkie	£2.99	1	£2.99
Motors pack of four	CL-8020-17 MMW	Drone Junkie	£21.99	1	£21.99
Battery	Mylipo 480mAh 3.7V 40C / 80C JST	Drone Junkie	£5.99	2	£11.98
IMU sensor	BMI160	Farnell	£8.69	1	£8.69
Colour sensor	TCS34725	Farnell	£6.93	1	£6.93
Through-hole 130A NMOS	IRL1004PBF	Farnell	3.14	2	£6.28
Through-hole 80A NMOS	IPP034N03LGXKSA1	Farnell	1.6	2	£3.20
Surface mount 8A NMOS	DMN2024UQ-7	Farnell	0.454	5	£2.27
Surface mount 7A NMOS	PMV15UNEAR	Farnell	0.405	5	£2.03
Battery charger	JST PH charger	Farnell	6.75	2	£13.50
		Unpaid for			
Batteries pack of twelve	AA batteries	N/A	£9.00	1	£9.00
IMU sensor	MPU-9250	N/A	£12.99	1	£12.99
3.3V Voltage regulator	LD33CV	N/A	£1.45	3	£4.35
				Total	**£101.85**

Table A-1

Appendix B: Schematics

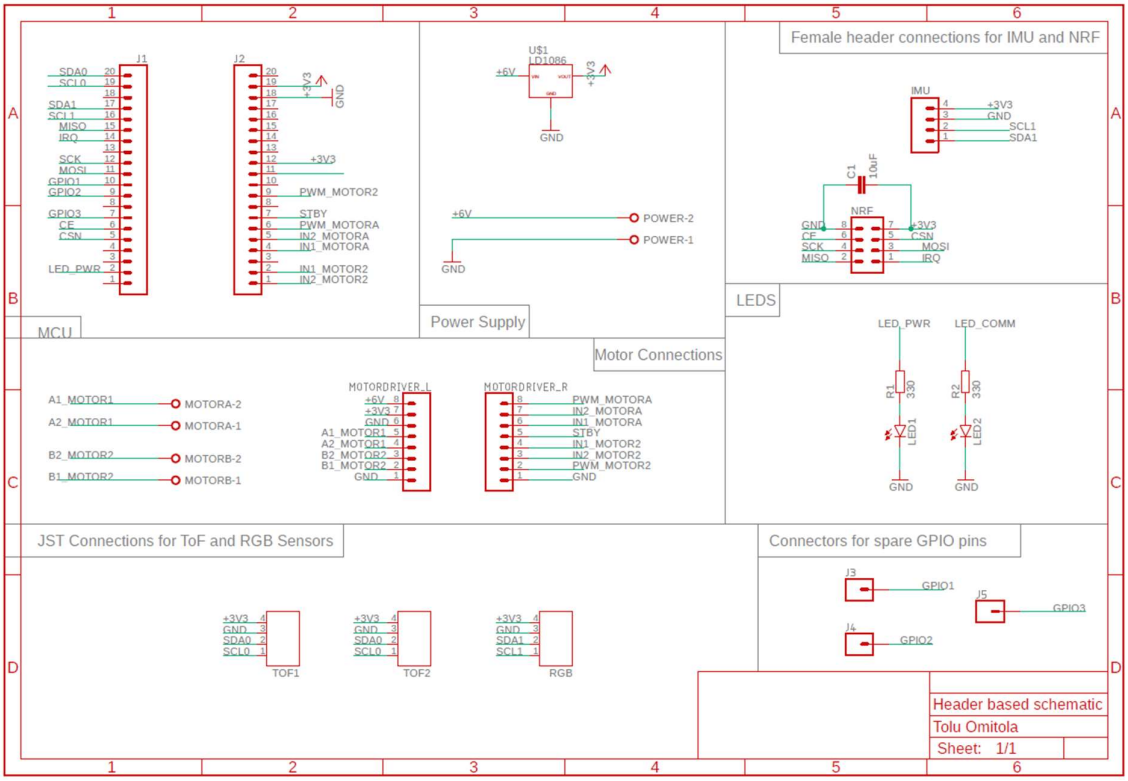


Figure B-1 – Eagle Schematic used for PCB that used mounted headers

Appendix C: Swarm System Code

```
from mpu6050 import MPU6050
from apds9960 import *
from apds9960 import uAPDS9960 as APDS9960
from nrf24l01 import NRF24L01
from time import time, sleep, sleep_us, ticks_ms, ticks_diff, ticks_add
from pid import PID
from machine import Pin, I2C, PWM, SPI
from vl53l0x2 import setup_tofl_device, TBOOT
import sys
import struct
from array import array
from random import randint

green_led = Pin(26, Pin.OUT)
red_led = Pin(14, Pin.OUT)

csn = Pin(12, mode=Pin.OUT, value=1) # Chip Select Not
ce = Pin(11, mode=Pin.OUT, value=0)  # Chip Enable
led = Pin(25, Pin.OUT)               # Onboard LED
payload_size = 20

i2c0 = I2C(id=0, sda=Pin(0), scl=Pin(1))
i2c = I2C(1, sda=Pin(2), scl=Pin(3), freq=400000)
imu = MPU6050(i2c)

apds = APDS9960(i2c)
apds.enableProximitySensor()
apds.enableLightSensor()

device_1_xshut.value(0)
tof0 = setup_tofl_device(i2c0, 40000, 12, 8)
tof0.set_address(0x31)

pwm_B = PWM(Pin(22))
pwm_A = PWM(Pin(20))

b_bk = Pin(17, Pin.OUT)
b_fd = Pin(16, Pin.OUT)

a_fd = Pin(19, Pin.OUT)
a_bk = Pin(18, Pin.OUT)

stby = Pin(21, Pin.OUT)

stby.on()
pwm_B.duty_u16(0)
pwm_A.duty_u16(0)

start_speed = 30000

Kp = 2
Ki = 0.01
Kd = 0.2
max_inc = 2
pid = PID(p=Kp, i = Ki, d = Kd, imax=max_inc)
target=0

out_L, out_R = start_speed, start_speed

left_turn = 90
right_turn = -90
half_left = 45
half_right = -45
right_adj = -1
left_adj = -1
```

```
half_turn = 180
turnCounter = 0
travel_time = array('i')

lastInstruct = 0

red = "N"
blue = "N"
redFound = False
blueFound = False

send_pipe = b"\xe1\xf0\xf0\xf0\xf0"
receive_pipe = b"\xd2\xf0\xf0\xf0\xf0"

def setup():
    nrf = NRF24L01(SPI(0), csn, ce, payload_size=payload_size)
    nrf.open_tx_pipe(send_pipe)
    nrf.open_rx_pipe(1, receive_pipe)
    nrf.flush_tx()
    nrf.flush_rx()
    nrf.start_listening()
    return nrf

nrf = setup()

#RF transceiver methods for sending robot
def sendAngle(angle):
    rfid = 1
    nrf.stop_listening()
    print("sending colours: ", angle)

    try:
        nrf.send(struct.pack("ii", angle,rfid))
    except OSError:
        pass

    nrf.start_listening()
    returned = nrf.recv()
    (maina,rfid1,rf1a,rfid2,rf2a ) = struct.unpack("iiiii",returned)
    if rfid == 1:
        r_angle1 = maina
        r_angle2 = rf2a
    else:
        r_angle1 = maina
        r_angle2 = rf1a
    return r_angle1,r_angle2

def sendColour(r,b):
    nrf.stop_listening()
    print("sending colours: ", r,b)
    try:
        encoded_r = r.encode()
        byte_r = bytearray(encoded_r)
        encoded_b = b.encode()
        byte_b = bytearray(encoded_b)
        nrf.send(struct.pack("ss", byte_r, byte_b))
    except OSError:
        pass
    nrf.start_listening()
    returned = nrf.recv()
    (reply_r,reply_b, ) = struct.unpack("ss",returned)
    rec_r = reply_r.decode()
    rec_b = reply_b.decode()
    return rec_r,rec_b

#RF transceiver methods for receiving robot
```

```
def recAngle(angle):
    rfid1 = 0
    rfid2 = 0
    rf1a = 999
    rf2a = 999
    if nrf.any():
        while nrf.any():
            package = nrf.recv()
            (r_angle,rfid, ) = struct.unpack("ii",package)


        nrf.stop_listening()
#         print("received: ",message, " from: ", rfid,"; sending: ", num2)
        if rfid == 1:
            rfid1 = rfid
            rf1a = r_angle
        elif rfid == 2:
            rfid2 = rfid
            rf2a = r_angle
        try:
            nrf.send(struct.pack("iiiii", angle, rfid1, rf1a, rfid2, rf2a))
        except OSError:
            pass
        nrf.start_listening()
    r_angle1 = rf1a
    r_angle2 = rf2a
    return r_angle1, r_anlge2

def recCol(r,b):
    msg_r = ""
    msg_b = ""
    if nrf.any():
        while nrf.any():
            package = nrf.recv()
            (message_r,message_b, ) = struct.unpack("ss",package)
        utime.sleep_ms(25)
        nrf.stop_listening()
        msg_r = message_r.decode()
        msg_b = message_b.decode()

        try:
            encoded_r = r.encode()
            encoded_b = b.encode()
            byte_r = bytearray(encoded_r)
            byte_b = bytearray(encoded_b)
            print("sending colours: ", r,b)
            nrf.send(struct.pack("ss", byte_r, byte_b))
        except OSError:
            pass
        nrf.start_listening()
    return msg_r,msg_b


def searchLED():
    red_led.value(1)
    green_led.value(0)

def homingLED():
    red_led.value(0)
    green_led.value(1)

def resetPathing():
    global travel_time
    global turnCounter
    turnCounter = 0
    travel_time = array('i')
```

```
def forwardControl():
    global out_L, out_R
    lout = max(0, out_L)
    rout = max(0, out_R)

    gz=round(imu.gyro.z)+1
    adj = pid.get_pid(gz, target)

    if adj < 0:
        if adj/2 >= -1:
            out_L = out_L
            out_R = out_R
        else:
            adj = -1*adj
            out_L = out_L+ adj
            out_R = out_R-adj
    elif adj > 0:
        out_R = out_R+adj
        out_L = out_L-adj
    else:
        out_L = out_L
        out_R = out_R

    lout = int(max(0, out_L))
    rout = int(max(0, out_R))

    return lout, rout

def moveForward(speedL, speedR):
    a_fd.value(1)
    a_bk.value(0)
    b_fd.value(1)
    b_bk.value(0)
    pwm_A.duty_u16(speedR)
    pwm_B.duty_u16(speedL)

def moveBack(speedL, speedR, t_time):
    a_fd.value(0)
    a_bk.value(1)
    b_fd.value(0)
    b_bk.value(1)
    pwm_A.duty_u16(speedR)
    pwm_B.duty_u16(speedL)
    sleep(t_time)

def turnR(out_L, out_R):
    a_fd.value(0)
    a_bk.value(1)
    b_fd.value(1)
    b_bk.value(0)
    pwm_A.duty_u16(out_R)
    pwm_B.duty_u16(out_L)

def turnL(out_L, out_R):
    a_fd.value(1)
    a_bk.value(0)
    b_fd.value(0)
    b_bk.value(1)
    pwm_A.duty_u16(out_R)
    pwm_B.duty_u16(out_L)

def turnAngle(deg,out_L,out_R):
    stop()
    moveBack(out_L, out_R, 0.1)
    angle = 0
```

```
    gz = 0
    deltaT_s = 0
    if deg < 0:
        turn = deg*0.944444444
        lastSample = ticks_ms()
        while angle>turn:
            if(ticks_diff(ticks_ms(),lastSample) > 9):
                gz=round(imu.gyro.z)+0.6785
                deltaT_s = (ticks_ms() - lastSample)/1000
                angle = angle + gz*deltaT_s
                lastSample = ticks_ms()
            turnR(out_L,out_R)
    else:
        turn = deg*0.944444444
        lastSample = ticks_ms()
        while angle<turn:
            if(ticks_diff(ticks_ms(),lastSample) > 9):
                gz=round(imu.gyro.z)+0.6785
                deltaT_s = (ticks_ms() - lastSample)/1000
                angle = angle + gz*deltaT_s
                lastSample = ticks_ms()
            turnL(out_L,out_R)
    stop()

def stop():
    a_fd.value(0)
    a_bk.value(0)
    b_fd.value(0)
    b_bk.value(0)
    pwm_A.duty_u16(0)
    pwm_B.duty_u16(0)

def wallDetectLeft():
    distL = tof.ping()-50
    if distL <= 100:
        return True
    return False

def wallDetectFront():
    dist = apds.readProximity()
    if dist >= 230:
        return True
    return False

def detectColours():
    redFound="N"
    blueFound = "N"
    bDetect = 100
    rDetect = 50
    r = apds.readRedLight()
    g = apds.readGreenLight()
    b = apds.readBlueLight()
    if(r>=rDetect and b < rDetect):
        redFound = "Y"
    if(b>=bDetect and r < bDetect):
        blueFound= "Y"
    return redFound, blueFound

def updateTargets(red,blue):
    global blueFound, redFound
    return False
    n_red,n_blue = sendDist(red,blue)
    while n_red == "" and n_blue == "":
        n_red,n_blue = sendDist(red,blue)
        if n_blue == "Y":
            blueFound = True
```

```
            blue = n_blue
        if n_red == "Y":
            redFound = True
            red = n_red
    return True

def initialiseHeading():
    global initialHeadingComplete
    startHeading = randint(-250,250)
    (angle1,angle2)=sendAngle(startHeading)
    while angle1 == 999 or angle2 == 999:
        (angle1,angle2)=sendAngle(startHeading)
        while startHeading == angle1 or startHeading == angle2:
            (angle1,angle2)=sendAngle(startHeading)
            startHeading = randint(-250,250)
    turnAngle(startHeading)

def newHeading():
    startHeading = randint(-250,250)
    turnAngle(startHeading)

initaliseHeading()

try:
    device_1_xshut.value(1)
    sleep_us(TBOOT)

    tof1 = setup_tofl_device(i2c0, 40000, 12, 8)

    def wallDetectRight():
        distR = tof1.ping()
        if distR <= 100:
            return True
        return False

    def returnHome(counter):
        homingLED()
        global red, blue
        sendColour(red,blue)
        while counter > 0:
            duration = travel_time[counter-2]
            instruct = travel_time[counter-1]
            if instruct == 3:

turnAngle(half_turn,forwardControl()[0],forwardControl()[1])
            elif instruct == 1:

turnAngle(left_turn,forwardControl()[0],forwardControl()[1])
            elif instruct == 2:

turnAngle(half_left,forwardControl()[0],forwardControl()[1])

            start = ticks_ms()
            while ticks_ms() < ticks_add(duration,start):
                sendDist(red,blue)
                if wallDetectFront():

                elif wallDetectLeft():

turnAngle(right_adj,forwardControl()[0],forwardControl()[1])
                elif wallDetectRight():

turnAngle(left_adj,forwardControl()[0],forwardControl()[1])
                else:
                    moveForward(forwardControl()[0],forwardControl()[1])
            counter=counter-2
```

```
        stop()
        sleep(2)
        newHeading()

    last_time = ticks_ms()
    while not blueFound and not redFound:
        searchLED()
        sendColour()
        if wallDetectFront():
            newTravel = ticks_diff(ticks_ms(), last_time)
            travel_time.append(newTravel)
            if detectColours()[0] == "Y":
                red = "Y"
                redFound = True
                stop()
                sleep(2)
                updateTargets()
                moveBack(forwardControl()[0],forwardControl()[1], 0.1)
                returnHome(turnCounter)
                resetPathing()

turnAngle(half_turn,forwardControl()[0],forwardControl()[1])
                newHeading()
                last_time = ticks_ms()
            elif detectColours()[1] == "Y":
                blue = "Y"
                blueFound = True
                stop()
                sleep(2)
                updateTargets()
                travel_time.append(3)
                turnCounter = turnCounter + 2
                moveBack(forwardControl()[0],forwardControl()[1], 0.1)
                returnHome(turnCounter)
                resetPathing()

turnAngle(half_turn,forwardControl()[0],forwardControl()[1])
                last_time = ticks_ms()
            else:
                travel_time.append(1)

turnAngle(right_turn,forwardControl()[0],forwardControl()[1])
                turnCounter = turnCounter + 2
                last_time = ticks_ms()
        elif wallDetectLeft():
             turnAngle(right_adj,forwardControl()[0],forwardControl()[1])
        elif wallDetectRight():
            turnAngle(left_adj,forwardControl()[0],forwardControl()[1])
        else:
            moveForward(forwardControl()[0],forwardControl()[1])
        sleep(3)
    stop()
    sleep(1)
    returnHome(turnCounter)
    stop()
finally:
    tof0.set_address(0x29)
```

Appendix D: Challenges

D.1 – Drone

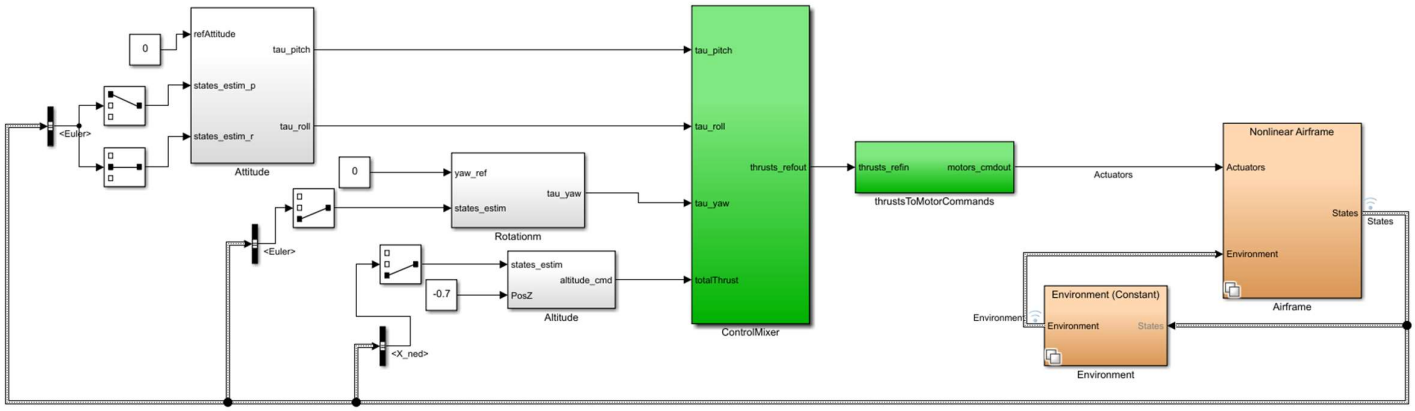


Figure D-1 – linearised parrot drone model in Simulink that used parameters of prototype drone (e.g., weight, motor thrust, frame diameter, etc)

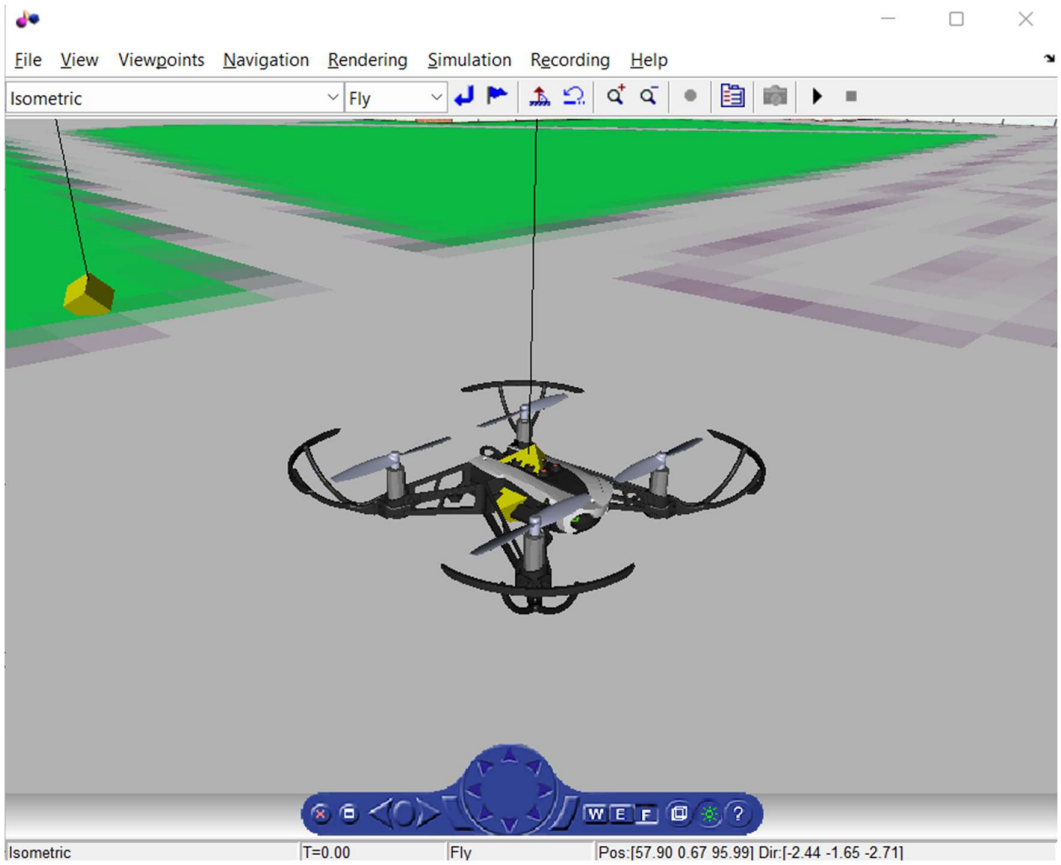


Figure D-2 – 3D visual simulation of parrot drone to observe changes in PID values on stability of drone

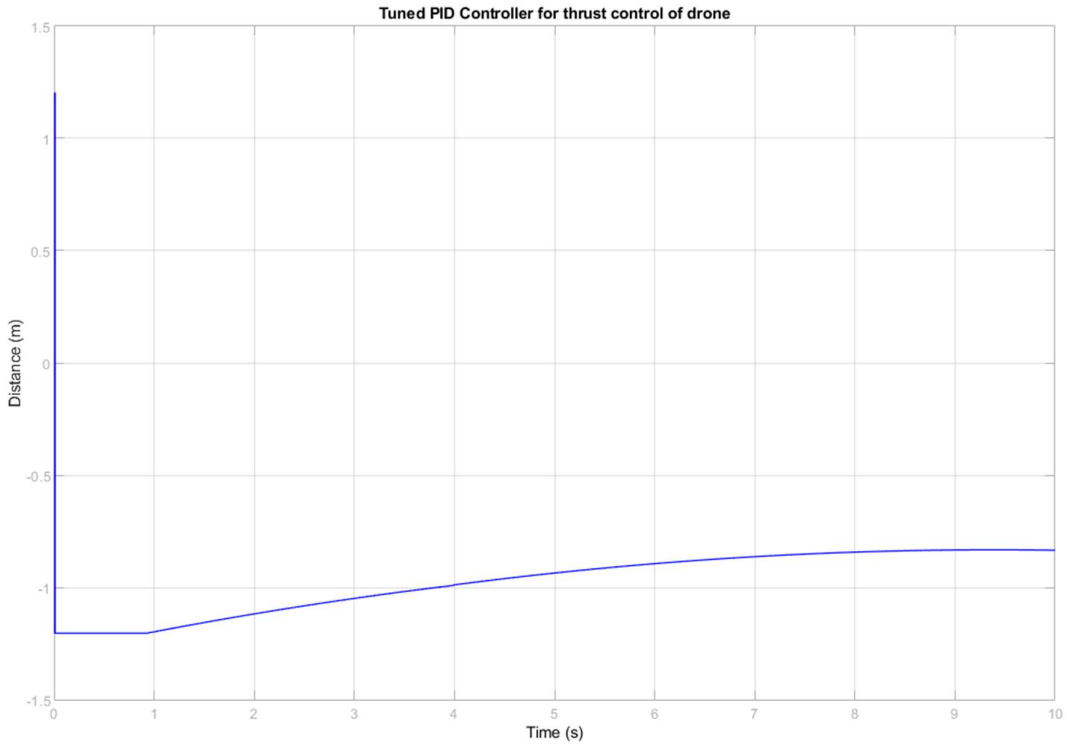


Figure D-3 – tuned controller shows effect of PID on stabilisation of drone over 10s

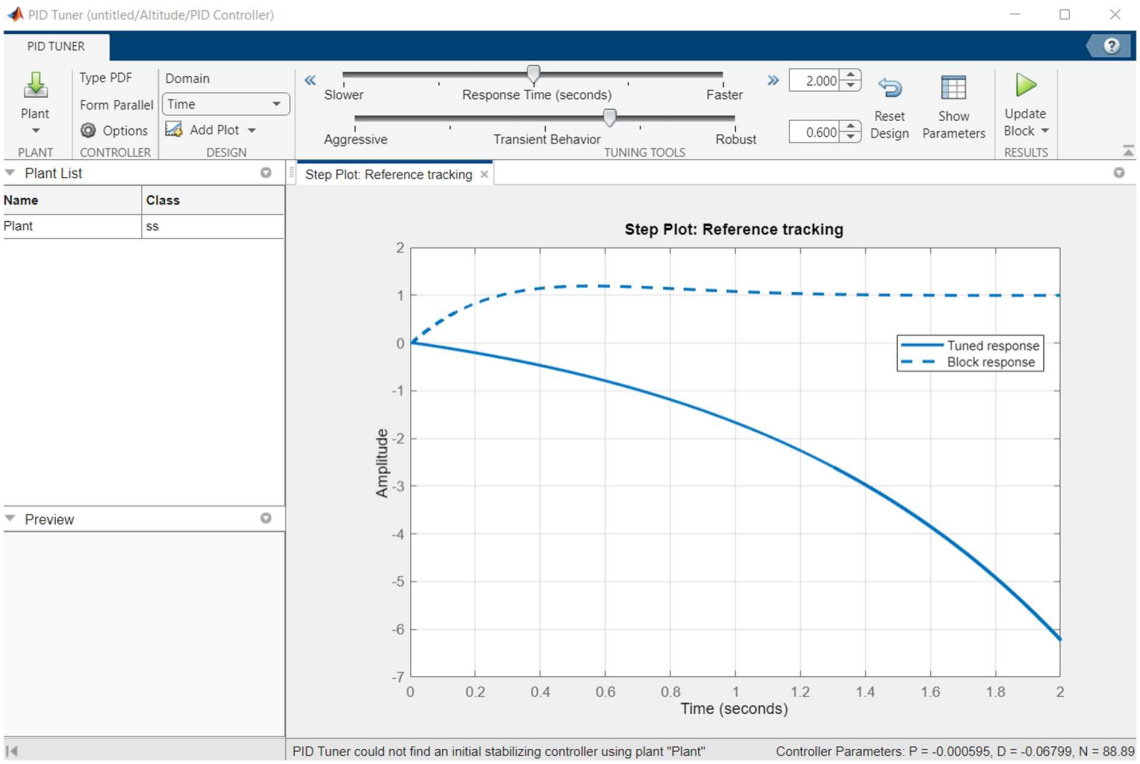


Figure D-4 – PID tuner tool used to get values to be used in physical system

PID Constants	Altitude	Attitude – roll	Attitude - pitch	Orientation - yaw
K_P	1	0.001	0.013	0.000788
K_I	0	0.001	0.01	0
K_D	0.4	0.0028	0.002	0.00073562

Table D-1 – PID values derived for thrust, roll, pitch and yaw of prototype drone

Figure D-5 – a) top of prototype stripboard; b) underside of stripboard;
c) 3D printed airframe with motors and propellers mounted

D.2 – Drone stabilisation code

```
from mpu6050 import MPU6050
from vl53l0x import VL53L0X
import utime
from pid import PID
from time import sleep, time
from machine import PWM, Pin, I2C


sda = Pin(0)
scl = Pin(1)
id = 0
freq = 400000

#ToF
bus = I2C(id=id, sda=sda, scl=scl)
tof = VL53L0X(bus)
tof.set_measurement_timing_budget(40000)
tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 12)
tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 8)

# MPU
i2c = I2C(id=id, sda=sda, scl=scl, freq=freq)
imu = MPU6050(i2c)

#Motors
pwm_B_CW = PWM(Pin(14))
pwm_F_CCW = PWM(Pin(15))
pwm_F_CW = PWM(Pin(16))
pwm_B_CCW = PWM(Pin(17))

#PID Constants
r_Kp = 0.01
r_Kd = 0.0028
pr_Ki = 0.01
p_Kp = 0.013
p_Kd = 0.002
y_Kp = 0.0007875
y_Kd = 0.0007356
t_Kp = 1
t_Kd = 0.4
_max_increment = 2
_max_increment_high = 10000

pid_roll = PID(p=r_Kp, i=pr_Ki, d=r_Kd, imax=_max_increment)
pid_pitch = PID(p=p_Kp, i=pr_Ki, d=p_Kd, imax=_max_increment)
pid_yaw = PID(p=y_Kp, d=y_Kd)
pid_thrust = PID(p=t_Kp, d=t_Kd)


r_target = 0
p_target = 0
y_target = 0
thrust_target = 100
land_target = 5
takeoff_gain = 1
takeoff = -9.81*0.015*takeoff_gain


start_speed = 400
end_speed = 15000
stop_speed = 100
stop_time = 10
ramp_const = 5
thrust = 0
```

```
out_B_CW = start_speed
out_B_CCW = start_speed
out_F_CW = start_speed
out_F_CCW = start_speed

start_time = time()
while True:
    fout = max(0, out_F_CW)
    fcout = max(0, out_F_CCW)
    bout = max(0, out_B_CW)
    bcout = max(0, out_B_CCW)
    pwm_F_CW.duty_u16(int(fout))
    pwm_F_CCW.duty_u16(int(fcout))
    pwm_B_CW.duty_u16(int(bout))
    pwm_B_CCW.duty_u16(int(bcout))

    gx=round(imu.gyro.x)+2
    gy=round(imu.gyro.y)+1
    gz=round(imu.gyro.z)+2
    dist=tof.ping()-30

    roll = pid_roll.get_pid(gx, r_target)
    pitch = pid_pitch.get_pid(gy, p_target)
    yaw = pid_yaw.get_pid(gz, y_target)
    thrust = pid_thrust.get_pid(dist, thrust_target) + takeoff

    out_F_CW = out_F_CW + roll - pitch
    out_F_CCW = out_F_CCW + roll + pitch
    out_B_CW = out_B_CW - roll + pitch
    out_B_CCW = out_B_CCW - roll - pitch

#
print("thrust",thrust,"\t","dist",dist,"F_CW",fout,"\t","F_CCW",fcout,"\t",
"B_CW",bout,"\t","B_CCW",bcout,end="\r")
#
print("F_CW",fout,"\t","F_CCW",fcout,"\t","B_CW",bout,"\t","B_CCW",bcout,"\
t","gx",gx,"\t","gy",gy,end="\r")
#
print("thrust",thrust,"\t","dist",dist,"F_CW",fout,"\t","F_CCW",fcout,"\t",
"B_CW",bout,"\t","B_CCW",bcout,"\t","gx",gx,"\t","gy",gy,end="\r")

while out_F_CW > 0:
    fout = max(0, out_F_CW)
    fcout = max(0, out_F_CCW)
    bout = max(0, out_B_CW)
    bcout = max(0, out_B_CCW)
    pwm_F_CW.duty_u16(int(fout))
    pwm_F_CCW.duty_u16(int(fcout))
    pwm_B_CW.duty_u16(int(bout))
    pwm_B_CCW.duty_u16(int(bcout))

    gx=round(imu.gyro.x)+2
    gy=round(imu.gyro.y)+1
    gz=round(imu.gyro.z)+2
    dist=tof.ping()-30

    roll = pid_roll.get_pid(gx, r_target)
    pitch = pid_pitch.get_pid(gy, p_target)
    yaw = pid_yaw.get_pid(gz, y_target)
    thrust = pid_thrust.get_pid(dist, land_target) + takeoff

    out_F_CW = out_F_CW + thrust - yaw + roll - pitch
    out_F_CCW = out_F_CCW + thrust + yaw + roll + pitch
    out_B_CW = out_B_CW + thrust + yaw - roll + pitch
    out_B_CCW = out_B_CCW + thrust - yaw - roll - pitch
```

```
#
print("thrust",thrust,"\t","dist",dist,"F_CW",fout,"\t","F_CCW",fcout,"\t",
"B_CW",bout,"\t","B_CCW",bcout,end="\r")

no_flight = 0
pwm_B_CW.duty_u16(no_flight)
pwm_B_CCW.duty_u16(no_flight)
pwm_F_CW.duty_u16(no_flight)
pwm_F_CCW.duty_u16(no_flight)
```

D.3 – Position estimation calculation from IMU sensor

```
from mpu6050 import MPU6050
from time import sleep,ticks_ms
from machine import Pin, I2C


i2c = I2C(1, sda=Pin(2), scl=Pin(3), freq=400000)
imu = MPU6050(i2c)

file = open("doubleInt.txt", "w")

currentSpeed_x = 0
currentPos_x = 0
currentSpeed_y = 0
currentPos_y = 0
currentSpeed_z = 0
currentPos_z = 0
lastSample = ticks_ms()
ax = 0
az = 0
ay = 0
time_taken = 0
deltaT_s = 0
while True:
    if(ticks_ms() - lastSample > 9):
        ax=round(imu.gyro.z,2)
        ay=round(imu.accel.y,2)
        az=round(imu.accel.z,2)
        deltaT_s = (ticks_ms() - lastSample)/1000
        currentSpeed_x = currentSpeed_x + ax*deltaT_s
        currentPos_x = currentPos_x + currentSpeed_x*deltaT_s
        currentSpeed_y = currentSpeed_y + ay*deltaT_s
        currentPos_y = currentPos_y + currentSpeed_y*deltaT_s
        currentSpeed_z = currentSpeed_z + az*deltaT_s
        currentPos_z = currentPos_z + currentSpeed_z*deltaT_s
        lastSample = ticks_ms()
    file.write("X position: " + str(currentPos_x) + "  " + "Y position: " +
str(currentPos_y) + "\n")
    file.flush()
```