Pluto Rover - Autonomous Navigation

Pallav Hingu

Contents

Chapter 1

Analysis

1.1 Description of the Project

Space Exploration has advanced massively in recent years. Large companies such as NASA and SpaceX are taking large steps forward to explore and colonize new planets such as Mars. We know a lot about the planets in our solar system: We have pictures, samples and critical data about their atmospheres. This has opened a gateway allowing us access to further resources from the planets in our solar system. But, there is one "planet" that we have yet to explore in detail; a "planet" where no man-made object has never landed: Pluto. Having been recently described on Google as: *"Our favourite dwarf planet since 2006"*, we have yet to discover its surface and internal composition. To continue our research of the planets in our solar system, we will need to **send a rover on a research and exploration mission to Pluto.**

As Pluto is extremely far away from the Earth, there are many obstacles to sending a rover to Pluto. Whilst the most obvious is transportation, NASA have already proved that they are capable of doing this using their "New Horizons" interplanetary space probe launched as part of the "New Frontiers" program. But, after getting to Pluto, there are many challenges that a rover can face. Firstly, due to the large distance between Earth and Pluto, there is a considerable delay in receiving signals, and there is also a lot of data loss. This means that the rover cannot easily be controlled remotely as there would be a large delay in sending instructions as well as receiving camera/sensor feed, which means that the camera may not represent the actual situation of the rover when the feed is received on Earth. As this poses a large threat to the condition of the rover, the best approach to address the safety of the robot is to install an autonomous navigation software, which avoids all obstacles in its path and uses other available sensors on the rover to get more information on the surroundings.

The information received using these sensors will be stored on the rover as well as sent to the nearest orbiting satellite, which would have been used to transport the rover as well. This satellite will then also store the data received from the rover's sensors and then will broadcast these results to Earth, allowing it to be analysed.

I will design a software with a friendly user interface (which will be on the companion app) that presents live information from the rover on a remote device, which can be on a satellite or on Earth to monitor the rover and receive crucial data. The rover is on Pluto to detect the presence and location of metals on the surface. My software will utilize IoT protocols to present the output

of the metal detector. It will also use a built-in IR camera to perform autonomous driving. A smart algorithm will also be used to generate the relative location of the rover on Pluto's surface according to its set landing location and landing orientation. This will allow the rover's position to be estimated, as GPS cannot be utilised on Pluto.

The aim of this mission is to detect if there are metals near the surface of Pluto, if yes, then to identify and record where.

1.2 Computational Methods

I think that this problem lends itself well to computational methods such as abstraction and decomposition for a variety of reasons. The solution will be a software that controls the Pluto rover on a hardware level, allowing it full autonomous control as well as manual, using the rover's cameras and sensors. This will simply need to run on a computer as that will be the medium on which the images from the camera will be processes and the next course of action for the rover will be determined. There is no alternative solution for this problem that will not require a computer, due to the context of the problem, and the environment in which the solution will need to function (the harsh climate of Pluto).

1.2.1 Methods Used

Some of the computational methods used for the solution are:

- **Problem Decomposition**
  This large problem can be decomposed into many smaller steps. These steps can be thought to be the following:

  1. Use the rover's camera system to get some RAW data about the surroundings
  2. Add threshold to data to limit the amount of data being passed through the program
  3. Plot an image representation of the RAW data
  4. Use image processing to detect the obstacles in the image
  5. Use a smart algorithm to determine the action needed to avoid obstacles
  6. Overwrite algorithm-determined movement with manual instructions from companion app if manual mode is enabled.
  7. Send instructions to a driver board (allowing hardware control to motors)
  8. Use an algorithm on the driver board to control the motors
  9. Send accurate movement of the rover to the companion app
  10. Calculate vector location of rover using its movement(companion app)
  11. Get data from the metal detector
  12. Send data to companion app
  13. Write vector location and metal detector data to local file (companion app)

  After these steps are completed, the problem can be said to be solved. The steps will allow the rover to react to obstacles in its surroundings, allowing for autonomous navigation. Also,

the steps allow the rover to utilise a metal detector and send its data to a server for analysis, meaning that we can get more information about the surroundings of the rover and the surface of Pluto. The vector location calculation will also allow for the rover to be located using no other hardware components, using only the velocity and angular velocity of the rover.

- **Problem Recognition**
  The overall and apparat problem is autonomous navigation through an extreme terrain. But, the underlying and primary problem is the detection of obstacles and their location. If the obstacles can be located, this information can be passed through a set algorithm that will allow a motor response.

- **Thinking Procedurally - Divide and Conquer**
  All the decomposed steps above allow for the problem to be divided into smaller problems, which are easier to solve than the large overarching problem. Creating functions/modules to solve the smaller problems, and them combining them all into a larger modular program is an example of divide and conquer.

- **Abstraction**
  Abstraction plays a large role in ensuring that the program responds to input data immediately. The incoming data stream from the camera sensor contains a lot of information, most of which is irrelevant for this application. So, it is vital to utilise abstraction and extract useful information only from the camera's input data. In this case, we would add a threshold so that the data only represents obstacles a certain distance from the camera. This allows us to reduce the amount of information being fed into the program, decreasing its response time and increasing its speed. The obstacles that are detected by the camera but are further away from the camera are removed as they do not need to be considered yet; these obstacles do not pose a risk of collisions yet.

- **Logic Modelling**
  Logic modelling will allow me to illustrate how the program must work to fulfil the system requirements. It will also be helpful to depict the actual program's activities compared to its intended needs, allowing me to measure how successful the program is at fulfilling it's function and solving the problem. As this will help with the application evaluation, it will be useful when improving the program. It ensures that all the requirements are fulfilled well.

1.3 Stakeholders

The ideal stakeholders for this project would include the large space research firms and their representatives. They would be the clients that would use this software and adapt it for space exploration missions. But, as this is not possible right now, the stakeholders for my project will be people who have done research around artificial intelligence, space exploration and robotics.

My first stakeholder will be a family relative who has completed a PhD in artificial intelligence and is current a professor at a university. I will attempt to add some type of artificial intelligence within the software so that it can learn more from its surroundings and adapt to the current situation. The university professor will use my software to create a demonstration for young student visitors during university open days. This will allow the young students to get inspired and learn more about the future application of artificial intelligence. My software will be one of the suitable solutions for the university as it will be free to use and will have a simple concept, GUI and set-up. While many alternative demonstrations of artificial intelligence on robots require a long set-up time, my software will be very simple to get ready and will be reliable so that there are not unexpected surprises on university open days.

My second stakeholder will be my schools Young Engineers groups who have engineered a prototype Pluto rover for their competition this year. They will use this software to allow their prototype rover to move across a modelled terrain of Pluto. My solution will benefit this group highly as it will allow them to test their prototype, while also testing my software in the most realistic situation that I have access to. The solution will be most appropriate to this group's needs as it will provide them with a free software that is specifically designed for their rover's hardware and is optimised for the computational power that is available to them. Other solutions would either cost a lot of money or time and would need much more computation power, making them very expensive and complex.

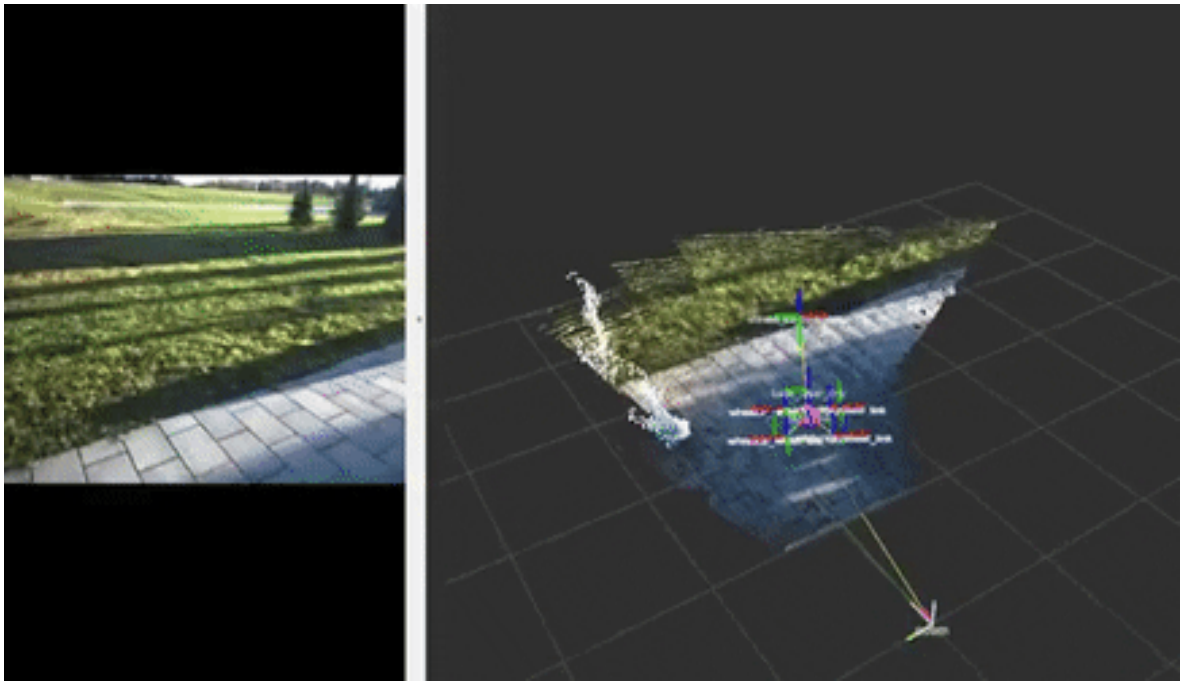
1.4 Research

1.4.1 Existing Solutions

As this is not a commonly commercially developed program, there are very few re-viewable examples available online. It is also very specific to the context, which may be a reason why the program is uncommon. But, if we widen the scope and look for software that are not specifically designed for this application but can be used in this context, we can find more programs online.

Examples:

- **RTAB-Map with ROS**



Overview

RTAB-Map (Real-Time Appearance Based Mapping) is an open-source application which uses different SLAM (Simultaneous localisation and mapping) approaches to generate a 3D dot-cloud of the surroundings of a sensor. It can be installed and used as a stand-alone application to generate 3D dot-clouds based on sensor inputs, or can be integrated into ROS (Robot Operating System) to allow the application to control hardware according to the 3D dot-cloud generated by a camera/sensor on a robot. It produces a 3d Map of the surrounds and uses localisation to detect the robot's location in relation to the 3d generated map. it then uses this to detect obstacles and find the most efficient way around the map to a given goal location.

ROS, which is essential in this application, is a robotics middle-ware and allows the RTAB-Map here to publish real-time messages to the appropriate nodes. In ROS, a node is a process that performs computation. Nodes are combined together in a graph and can communicate

with each other to perform certain tasks. RTAB-Map, when integrated well with ROS, publishes messages to many external nodes, which communicate with hardware drivers in the system to move the robot according to RTAB-Map's instructions. As RTAB-Map can get a large image of the surroundings, it uses a smart algorithm to conduct path finding, meaning that it calculates the most efficient route to reach a destination while avoiding obstacles.

As this is a complicated process, it requires a lot of advanced hardware and a high amount of processing power. ROS is also extremely difficult to use, as well as install correctly. ROS is used for many complex robots, like the Robonaut 2 aboard the International Space Station.

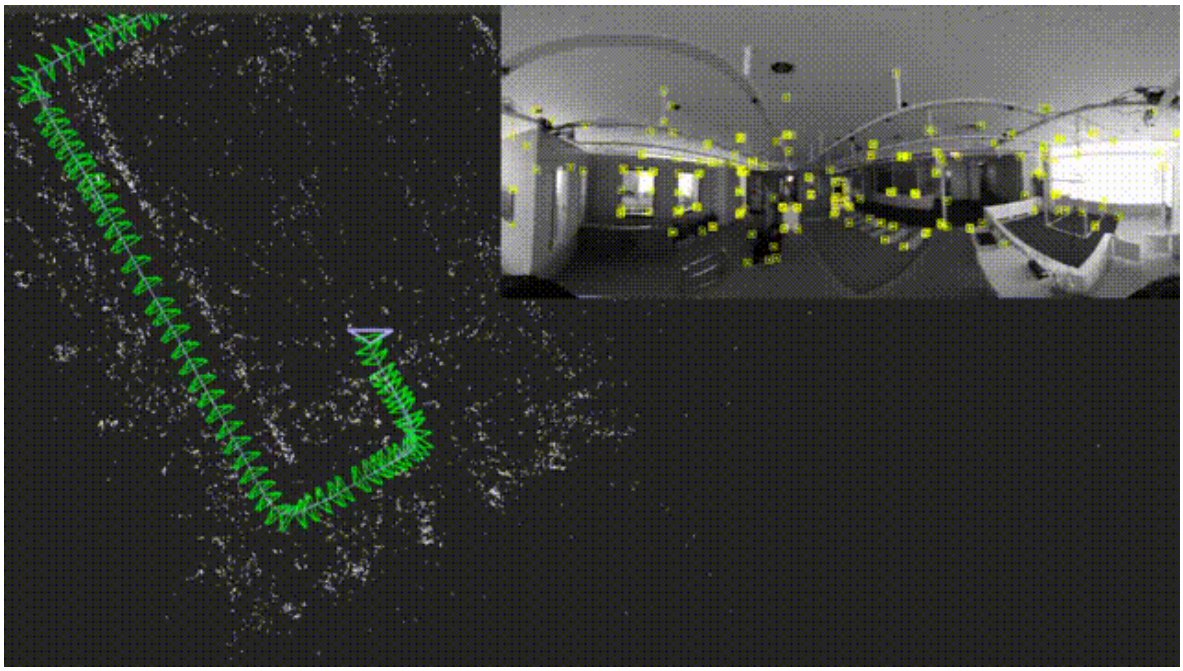
The Graphical User Interface(GUI) that it uses displays lots of information, such as the 3d Dot-map created of the surroundings in real-time, the raw images that are being captured, and the calculation of odometry (the relative location of the vehicle). This is a simple but effective and advanced user interface that provides a lot of data that can be useful for environmental analysis, but is not hard to use. As this application is not specific to a problem, the GUI can be adjusted to display less or more information, or different information by tweaking the settings. This allows the application to be adapted to many different situations, making it ideal even for an autonomous Pluto Rover.

Parts that I can apply to my solution

From this application, there are many things that I can use in my application:

- *Display Odometry* - the application displays the odometry (relative location) of the vehicle on the GUI. This will allow the relative location of the rover to be displayed on the GUI so that the Rover can be located.
- *Hardware Control* - the software can control hardware directly by converting the output into electrical signals for the components. This will allow the Rover's motors to be controlled by my application directly.
- *Navigation* - this application can determine movement to navigate around a map with a set objective. This will be useful to move the Rover around the surface of Pluto with an objective to detect metal.
- *GUI settings* - the use of GUI settings allows the displayed information on the GUI to be adjusted according to user preference or need. This can also be helpful in low resource scenarios as displaying less information means that the program runs smoother and faster. This could be helpful on the Rover as there are limited computational resources and so the adjustment of the GUI to the scenario may be helpful in improving performance.

- **OpenVSLAM**



Overview

OpenVSLAM is a monocular, stereo and RGB-D visual SLAM system similar to RTAB-Map. It is a software that uses pattern recognition to detect movement and estimate current location of the object, its odometry. But, the large advantage with this application is that it is compatible with many different cameras and so can be used for a variety of different projects. This system is also completely modular as it is designed by encapsulating several functions in separated components with APIs.

When comparing OpenVSLAM with RTAB-Map, we can see that RTAB-Map displays a lot more data. This is because it creates a dot-cloud with as much detail as possible to generate an accurate computational representation of its surroundings. While this data may be useful for many applications, it is not essential for obstacle detection and autonomous navigation. On then other hand, OpenVSLAM generates a simple but informative 2D birds eye map of the movement of the vehicle. It also creates a simpler dot-cloud by only detecting the presence of obstacles and not their shape, size or colour. This allows it to use minimal relevant data to navigate, which is optimal as it reduces the amount of data being processing, reducing processing time, latency and hardware demand.

The minimal approach on this monocular SLAM software makes it more suited for our application than RTAB-Map as we have limited hardware, and an overload of unnecessary data may product uncertainty in the reliability of the software.

Parts that I can apply to my solution

There are a few things from OpenVSLAM that I can apply to my solution:

- *Input Abstraction* - the ability to abstract the relevant information form the input device to suit the need of the software. This allows the software to be more efficient as well as reliable.
- *GUI Simplicity* - the GUI for OpenVSLAM is very simple and effective as it displays essential data making it easy to read, ensuring that there is not an overload of data being displayed to cause software lag as well as user confusion.

- **Openpilot**

Overview

Openpilot is an open source, semi-automated driving system developed by comma.ai. This program allows every-day cars to adopt semi-autonomous lane assistance and many other similar programs.

OpenVSLAM is a monocular, stereo and RGB-D visual SLAM system similar to RTAB-Map. It is a software that uses pattern recognition to detect movement and estimate current location of the object, its odometry. But, the large advantage with this application is that it is compatible with many different cameras and so can be used for a variety of different projects. This system is also completely modular as it is designed by encapsulating several functions in separated components with APIs.

When comparing OpenVSLAM with RTAB-Map, we can see that RTAB-Map displays a lot more data. This is because it creates a dot-cloud with as much detail as possible to generate an accurate computational representation of its surroundings. While this data may be useful for many applications, it is not essential for obstacle detection and autonomous navigation. On then other hand, OpenVSLAM generates a simple but informative 2D birds eye map of the movement of the vehicle. It also creates a simpler dot-cloud by only detecting the presence of obstacles and not their shape, size or colour. This allows it to use minimal relevant data to navigate, which is optimal as it reduces the amount of data being processing, reducing processing time, latency and hardware demand.

The minimal approach on this monocular SLAM software makes it more suited for our application than RTAB-Map as we have limited hardware, and an overload of unnecessary data may product uncertainty in the reliability of the software.

Parts that I can apply to my solution

There are a few things from OpenVSLAM that I can apply to my solution:

- *Input Abstraction* - the ability to abstract the relevant information form the input device to suit the need of the software. This allows the software to be more efficient as well as reliable.
- *GUI Simplicity* - the GUI for OpenVSLAM is very simple and effective as it displays essential data making it easy to read, ensuring that there is not an overload of data being displayed to cause software lag as well as user confusion.

1.4.2 Primary Research

1.4.2.1 Interview Questions

To ensure that an interview with the stakeholders is as effective as possible, the questions to be asked will require planning so that all required information related to the solution can be communicated.

I will first interview the ***Artificial Intelligence Professor*** with the following questions.

Background Questions:

1. *How important do you think artificial intelligence to be for the Pluto rover?*
2. *In this case, what can artificial intelligence do that a standard algorithm cannot?*
3. *How will adding artificial intelligence impact the hardware utilisation of the program?*

Questions for client requirements:

1. *What are your expectations for the GUI?*
2. *What hardware would you be able to use for running this software?*
3. *What should you be able to control remotely?*
4. *Would you prefer using a companion app on a computer or wiring up the embedded computer?*
5. *Is there anything else you would like to add?*

Apart from the above question, I may ask follow up questions or ask them to elaborate on certain answers.

After completing the interview with the Professor, I will interview the ***Young Engineers group*** with the following questions.

Questions for client requirements:

1. *What are your expectations for the GUI?*
2. *What hardware would you be able to use for running this software?*
3. *What should you be able to control remotely?*
4. *Would you prefer using a companion app on a computer over wiring up the embedded computer?*
5. *Is there anything else you would like to add?*

These questions are the same as the client requirement questions from the Artificial Intelligence Professor, allowing us to compare and create a collective requirements list to suit both stakeholders.

I have included some background questions for the Professor so that I can use the answers when attempting to implementing AI into my program.

1.4.2.2 Interview

Artificial Intelligence Professor:

Background Questions:

1. ***How important do you think artificial intelligence to be for the Pluto rover?***

   "I think that the use of artificial intelligence is extremely important in a rover, especially if it made for Pluto as the distance between the user and the rover will be very far. Therefore, we will need something to control the rover locally without relying on a human, and so artificial intelligence will be needed."

2. ***In this case, what can artificial intelligence do that a standard algorithm cannot?***

   "Well there are many things that an artificial intelligence model can be used for in this scenario. For example, it can be used to move the rover by observing its surroundings, it can be used to predict and recognize hazards. Unlike standard algorithms, it can learn more from the information it is observing, allowing it to perform better in unaccounted situations."

3. ***How will adding artificial intelligence impact the hardware utilisation of the program?***

   "Well, there are many different ways of implementing artificial intelligence, so it does depend on your chosen method. But, in general, AI utilises more hardware than a standard algorithm, especially if it is constantly learning from it's environment. So, I would say that adding AI to the program will increase its hardware requirements quite dramatically."

Questions for client requirements:

1. ***What are your expectations for the GUI?***

   "In this case, it would be most effective to use a simple GUI that allows control over all vital hardware features that are accessible using the software. I would also expect to see some sort of statistics on the current state of hardware or the output of mounted sensors."

2. ***What hardware would you be able to use for running this software?***

   "For demonstrations with this software, I would have access to a computer and possibly the model rover needed for the physical demonstration. It would be ideal if I could use any computer to control the rover."

3. ***What should you be able to control remotely?***

   "As the software is designed for long range control, I would expect options for manual control for as many things as possible. there are many things that can go wrong and so the user should have the choice to control all hardware possible remotely, as well as the movement of the rover obviously."

4. ***Would you prefer using a companion app on a computer or wiring up the embedded computer?***

   "It would definitely be easier to use a companion app, as well as much more professional. Wiring would increase the chances of errors and make everything much more fiddly during demonstrations, and so I would definitely prefer a companion app. But, the companion app should be easy to use and should be reliable, having quick setup times as well to make it robust."

5. ***Is there anything else you would like to add?***

   "I think I have said everything I would like to. Thank you."

Young Engineers Group:

1. ***What are your expectations for the GUI?***

   "A clean user interface with complex functionality allowing full control of the rover. We vision the GUI with a clean look presenting the location of the rover and a few buttons that allow full control of the rover."

2. ***What hardware would you be able to use for running this software?***

   "To run the embedded app, we have a Raspberry Pi 3 that is built into our model rover. Alongside, we also have an Arduino PCD that is hardwired to all rover sensors and motors. The Arduino can also communicate with the Raspberry Pi. If needed, we also have access to a remote laptop. Regarding the rover's sensors, it has a Kinect 360 RGB-D camera along with a magnetic coil for metal detection and a receiver for a RC transmitter. The Rover also has 4 DC motors that can be used for movement. This is the hardware that we would ideally run the application on."

3. ***What should you be able to control remotely?***

   "It would be nice to be able to control the rover's movement remotely to control the rover as required during testing and demonstrations. We should also be able to switch quickly between autonomous and manual control, which would helpful during presentations of the rover at events."

4. ***Would you prefer using a companion app on a computer over wiring up the embedded computer?***

   "It would be helpful to have the option use a companion app but we would also want to be able to control the rover with a RC transmitter so that it can be manually controlled without having do download specific software."

5. ***Is there anything else you would like to add?***

   "One thing we would like to add is that the Kinect sensor can see objects that are not directly in front of the rover. This may need to be considered during development."

1.4.2.3 Interview Analysis

During the interview with the University Professor, I asked a few background questions to allow me to expand my knowledge and understand the effects and impacts when attempting to add artificial intelligence to my program. From these background questions, I have learnt that adding artificial intelligence to the software would be extremely helpful for my purpose but it can have many obstacles such as higher hardware requirements.

From the other questions that I asked the University Professor and the Young Engineers group, I learnt that the GUI would need to be basic in appearance but should have the ability to control all the hardware possible. It should also present statistics about the current state of the rover along with the location of the rover. Ideally, it should also allow the rover to be switched between manual and autonomous mode quickly. The program should also be extremely efficient, using very low processing power as it needs to be run on a single board computer like the Raspberry Pi 3. There are also many hardware units hardwired to this single board computer and so the program needs to be able to control those units appropriately. Remotely, both stakeholders would prefer being able to control all hardware possible, especially the movement of the rover. The software should also include a professional and easy to use companion app, as preferred by both stakeholders. While being able to control the rover using a companion app, the Young Engineers group would also want to be able to control the rover using a remote transmitter that has a hardwired receiver. At the end of the interview with the Young Engineers group, they have also pointed out that the camera sensor has a wide perspective and so can see objects that are not directly in front of the rover, meaning that the software has to differentiate between objects that are directly in front and so are an obstacle, and objects that are not in the rover's path.

1.5 Essential Features of the Proposed Solution (System Goals)

From the research of existing software and interviews, a system can be designed as a solution to the initial problem. The solution proposed for this problem would have the following features:

Feature	Description	Purpose
Map	Display the location	The stakeholder will be able to view the relative location of the rover to the starting position.
Menu	Buttons for control	The menu will be a collection of controls that will allow the hardware of the rover to be controlled.

Data Abstraction	Control input data	The input data will immediately be abstracted to useful data so that less processing is needed in the program, decreasing hardware utilization.
Auto mode	Autonomous navigation of the rover	The user can turn on autonomous mode to control the rover's movement using the internal algorithm.
Manual mode	Manual control over rover movement	This is turned on when auto mode is turned of. Manual mode allows the user to have complete control over the rover's movement.
Stats Display	Collection of live information about hardware	This will be a small collection of graphs and numbers that allow the live information form the sensors to be monitored.

1.6 Desirable Features of the Proposed Solution

Feature	**Description**	**Purpose**
AI	Artificial Intelligence navigation	The use of a trained Artificial Intelligence model for autonomous navigation.
Data Preserve	Save Essential Data from Rover	Save information of the sensors, co-ordinates of rover and current vital variables in the program.
Negative Obstacle Detection	Detect any absence of the ground	Use a more advanced algorithm to detect potholes in front of the rover by detecting the absence of a ground plane.

1.7 Software and Hardware Requirements

1.7.1 Software Requirements

Embedded Application Requirements:

- Operating System with independent requirements:
  - **Windows 7+ (x86 or x64) or Windows IoT core** - *The drivers are only officially available for these releases.*
    - Kinect for Windows SDK - *To support the Kinect 360 RGB-D Camera*
    - Visual Studio 2010 Edition - *A dependency for Kinect Drivers.*
    - Microsoft .NET Framework 4.0 - *A dependency for Kinect Drivers.*
    - Java Runtime Environment - *Required to run my Application as it is Java based.*
    - Active Internet Connection - *To allow communication with companion app.*
  - **Ubuntu 18.04** - *The drivers and software is tested and robust on this specific OS release.*
    - Libfreenect Drivers - *To support the Kinect 360 RGB-D Camera.*
    - Java Runtime Environment - *Required to run my Application as it is Java based.*
    - A Desktop Environment - *Needed to interact with the program UI.*
      - Recommended: GNOME Desktop Env.
    - Active Internet Connection - *To allow communication with companion app.*
  - **Mac OS** - *Cannot be used as it should not be loaded onto a Raspberry Pi 3 or other single board computers.*

Companion Application Requirements:

- Operating System with independent requirements:
  - **Any Windows Release** - *The JRE can be installed on all Windows releases*
    - Java Runtime Environment (JRE) - *Required to run my Application as it is Java based.*
    - Active Internet Connection - *To allow communication with embedded app.*
  - **Any Ubuntu Release** - *The JRE can be installed on all Ubuntu releases*
    - Java Runtime Environment (JRE) - *Required to run my Application as it is Java based.*
    - A Desktop Environment - *Needed to interact with the program UI.*
      - Recommended: GNOME Desktop Env.
    - Active Internet Connection - *To allow communication with embedded app.*
  - **Any Mac OS** - *All macOS releases are supported by JRE*
    - Java Runtime Environment (JRE) - *Required to run my Application as it is Java based.*
    - Admin Privileges - for installation of JRE
    - Active Internet Connection - *To allow communication with embedded app.*

1.7.2 Hardware Requirements

Embedded Application Requirements:

- Kinect 360 RGB-D Camera - *Needed to detect obstacles and analyse surroundings*
- Computer with a dual-core, 1.2 GHz or faster processor - *Required for Kinect SDK*
- Windows compatible graphics card that supports DirectX 9.0c capabilities - *Required for Kinect SDK on Windows*
- 1GB RAM (2-GB RAM recommended) - *Would be required to run Kinect drivers as well as my application and any other libraries.*
- 4 DC motors - *To act as movement output and move the wheels.*
- Arduino connected with a I2C bus to the Computer - *To interface with the sensors and motors directly, acting as a driver board.*

Companion Application Requirements:

- An Intel-based Mac or any Windows/Ubuntu computer. - *Requirement for JRE.*
- Minimum 128MB memory available. - *Requirement for JRE.*
- Cursor Input (Mouse, Trackpad, Touch screen, etc.) - *needed to use navigate menu and press buttons.*
- Keyboard (optional for running application but required for manual movement control.) - *Keyboard keys would be used to control the rover manually and so a keyboard will be needed for such use.*

1.8 Requirements Specification

Input Requirements

Number	Criteria	Justification
1.1	The user should be able to promptly control the movement mode.	Input from the user will toggle auto and manual mode on or off.
1.2	The user should be able to send instructions to move using keyboard keys.	The user will be able to use certain keys from the keyboard to control the rover movement as well as other motors.
1.3	The user will be able to use the menu to hide control buttons and statistics.	This will allow the user to hide objects that they do not need to keep the UI clean and to prevent accidents.
1.4	There should be input from the sensors	This will be used to determine the next course of action for the rover, using either an algorithm or from manual control instructions.

Processing Requirements

Number	Criteria	Justification
2.1	The embedded program will use around 300MB of memory.	This will allow the program to run smoothly on single board computers and slower computers.
2.2	The program should only account for obstacles within 1m of the sensor.	This will allow the program to run smoother and use efficient abstraction to make appropriate decisions.
2.3	The program will ignore data of obstacles not on the rover's path.	This will allow the rover to detect only the obstacles directly in from of the rover that pose a direct hazard to the rover.
2.4	The data from the sensors will be sent to the companion app within 5 seconds.	This will ensure that the data viewed by the user is an accurate representation of the current situation, allowing the correct actions to be taken in time.

User Interface Requirements

Number	Criteria	Justification
3.1	The user interface should be clean and simple.	This will allow the program interface to be suitable for presentations and will only display a small amount of vital information when not in use.
3.2	The menu will allow full manual control.	The menu of the user interface should allow the user to toggle on full manual control of the rover where they can override the algorithm decisions manually.
3.3	The UI should present the sensor data.	This will allow the user to analyse the terrain and environment on which the rover is on and make appropriate decisions when manually controlling.
3.4	The Embedded Application will be observable.	This will be helpful for debugging as well as presenting software. The Embedded Application can be observed without the companion app.

1.9 Success Criteria

To ensure that the solution has fulfilled all system requirements, it will need to be tested. I will test the program in 3 different stages. The first stage will test the solution's ability to fulfil all general requirements. The second stage will test the input and output usability of the program. The third stage will test the program's processing functionality and robustness.

1.9.1 Stage 1: General Requirements

Num.	Criteria	Desirable	Acceptable	Fail
1.1	The user should be able to promptly control the movement mode.	The user can change the movement mode within 3 seconds.	The user can change the movement mode within 20 seconds	The user cannot change the movement mode.
1.2	The user should be able to send instructions to move using keyboard keys.	The user can send all movement instructions using Keys.	The user can only send start and stop.	The user cannot send instructions using the keyboard keys.
1.3	The user will be able to use the menu to hide control buttons and statistics.	The control buttons will be completely hidden within a menu.	Some buttons can be hidden with a menu.	The menu cannot be hidden and all control buttons are always visible.
1.4	There should be input from the sensors	All sensors send input which can be used.	Most sensors send inputs that are accessible.	No sensors on rover input data into the program.
2.1	The embedded program will use around 300MB of memory.	The program uses less than 250MB of RAM	The program uses between 250MB - 350MB of RAM	The program uses more than 350 MB of RAM.
2.2	The program should only account for obstacles within 0.5m of the sensor.	The program accounts for all obstacles only within 1m.	The program accounts for obstacles within 2m.	The program accounts for all visible obstacles.
2.3	The program will ignore data of obstacles not on the rover's path.	All obstacles not on the rover's path are ignored.	Most obstacles not on rover's path are ignored.	There is no filter for the obstacles not on the rover's path.
2.4	The data from the sensors will be sent to the companion app within 5 seconds.	Data is received within 5 seconds.	Data is received within 10 seconds	Data is received after 10 seconds.

Num.	Criteria	Desirable	Acceptable	Fail
3.1	The user interface should be clean and simple.	There is minimal information displayed and no control buttons are visible by default.	Some control buttons are visible by default but most are hidden.	All control buttons and elements are visible by default.
3.2	The menu will allow full manual control.	The rover can be switched to full manual control using one control button.	The rover can be switched to manual control using the menu.	The menu does not allow the rover to be switched to full manual control.
3.3	The UI should present the sensor data.	The UI presents graphed sensor data.	The UI presents numerical sensor data.	The UI does not display sensor data.
3.4	The Embedded Application will be observable.	Another computer can be used to view the operations of the embedded computer.	A HDMI cable can be used to view operations of the embedded computer.	The operation of the embedded software cannot be observed.

1.9.2 Stage 2: Input and Output Usability of the Program

In this section, we will test how useful the input and output of the solution is.

Input:

Criteria	Desirable	Acceptable	Fail
The user should be able to control the rover movement using keyboard keys.	The user can move the rover in all directions using computer the keyboard.	The user can move the rover forwards and backwards.	The user has no control over the movement of the rover.
The user should be able to interact with hardware control buttons.	All buttons are functional and available to user	Some buttons do not work.	No buttons can be clicked.
The embedded application should receive input from all sensors on the rover.	Required sensor information from all sensors can be read and used for algorithms.	Some sensor inputs can be read and used.	No sensor inputs can be read.

Output:

Criteria	**Desirable**	**Acceptable**	**Fail**
The program outputs the location of the rover.	The coordinates can be saved to a file with timestamps and location can be displayed	The UI displays the rough location of the rover.	The location of the rover can not be found.
The program outputs the information from sensors.	All information from all sensors can be presented well (e.g. using graphs).	The information can be viewed using the back-end.	No sensor information can be found.
The companion program can output the current movement of the rover.	The information about the current movement can be saved with timestamps.	The information about the current movement can be viewed using back-end.	The current movement of the rover cannot be found using companion app.
The embedded application can output signals to move appropriate motors on the rover.	All motors onboard can be controlled individually by the embedded application.	Most motors on the rover can be controlled, enough for movement of the rover.	Motor control is not sufficient for movement of the rover.

1.9.3 Stage 3: Processing Functionality and Robustness

To test how robust the program is, each module of the solution must be tested to ensure that they all work as required. The lack of strength of one module can be a bottleneck and weaken the whole program, making it unreliable. This test will be a pass or fail test, where each module will either be approved or rejected. Below I will describe the expected output from each module.

The modules to be tested and how they will be tested:

1. **Use the rover's camera system to get some RAW data about the surroundings**

   For this step to be rigid, the camera system should be able to give a continuous stream of RAW data to the Raspberry Pi for at least 5 hours.

2. **Add threshold to data to limit the amount of data being passed through the program**

   This is a vital step as it can drastically increase the performance and reliability of the solution. The output of this module should be reduced size image data that displays only the closer items.

3. **Plot an image representation of the RAW threshold data**

   This module will present an image representation of the RAW threshold data on the Raspberry Pi.

4. **Use image processing to detect the obstacles in the image**

   This module will highlight the obstacles in the image.

5. **Use a smart algorithm to determine the action needed to avoid obstacles**

   This module will output the either "right" or "left" or any other verbs to reflect the best course of action to avoid the obstacles when detected.

6. **Overwrite algorithm-determined movement with manual instructions from companion app if manual mode is enabled.**

   If in Manual mode, any instructions from the companion app will replace the instructions from the smart algorithm.

7. **Send instructions to a driver board (allowing hardware control to motors)**

   This module should output electrical signals over the I2C bus between the Raspberry Pi and Arduino(being used as a driver board).

8. **Use an algorithm on the driver board to control the motors**

   The output from this module should be an electrical pulse sent to the DC motors to control movement.

9. **Send the accurate movement of the rover to the companion app**

   The current movement of the rover will be sent to the companion app as a string to represent the direction of movement, but constant velocity will be assumed.

10. **Calculate vector location of rover using its movement (companion app)**

    This module will run on the companion app and will use the current state of movement of the rover to perform vector calculations on the last vector location.

11. **Get data from the metal detector**

    The output of this module will be an integer to represent the strength of the magnetic induction in the metal detector.

12. **Send metal detector data to companion app**

    The output of this module will be a transfer of data from the embedded app to the companion app. This module will output integers in the companion app.

13. **Write vector location and metal detector data to local file (companion app)**

    The output of this module will be a constantly updating file that holds the vector location, metal detector data and timestamps.

1.10 Limitations

There are quite a few limitations to this solution. The first is that the solution relies on an active internet connection to transfer data. Although the rover can be kept safe and away from obstacles without an internet connection, the rover needs it to send any data or receive instructions for manual mode. This can be an issue if used for the actual Pluto rover as it would require a satellite to orbit the planet and relay information to Earth, which can be difficult. When used for models and demonstrations, the limitations is that there needs to be an available internet connection that the rover can pair to, which can sometimes be a problem.

Another limitation with this solution, is that we do not know the exact details about the terrain and atmosphere of Pluto and we have never sent a rover there, therefore the solution is optimised to our current knowledge of the terrain, which may not be accurate. This means that the solution may not use the optimal algorithm to autonomously navigate the surface of Pluto.

An additional limitation is that if a smart algorithm is used for the autonomous navigation, then it will only respond to pre-programmed circumstances and that may not always be optimal. But, when using a AI algorithm, the training will change the way it will react to unseen circumstances, and so training it on Earth may not make it optimal for Pluto terrain, meaning that it may behave unexpectedly.

Another limitations that we have is to do with the available hardware. There isn't powerful hardware available in our context and so the program needs to be extremely efficient. The hardware that we have available cannot run existing solutions such as RTAB-Map and OpenVSLAM to complete the functions that we require (autonomous navigation). This means that a new approach to this problem is needed to make it more efficient and allow it to run on the limited resources available.

While there are many existing solutions available that solve problems similar to ours, there are no easily accessible programs that are designed specifically for rovers. This means that there is no available help online that is specific to our context, and there is no example solution from which the basis of the application can be built. This makes developing the solution much more challenging.

Chapter 2

Design

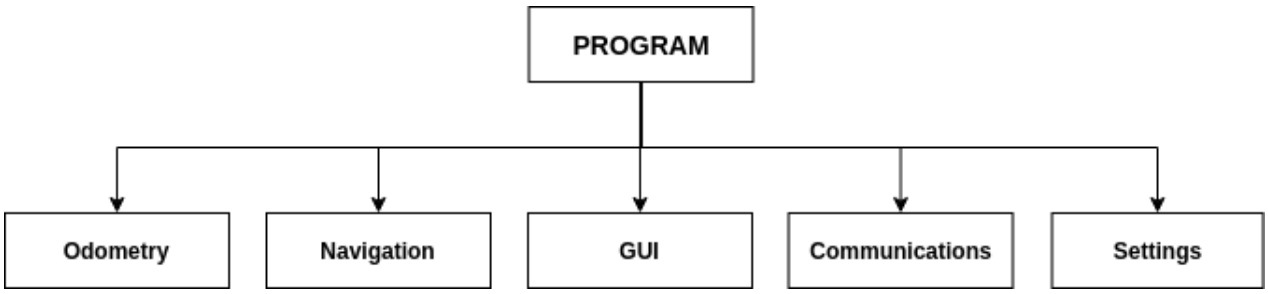
2.1 Introduction

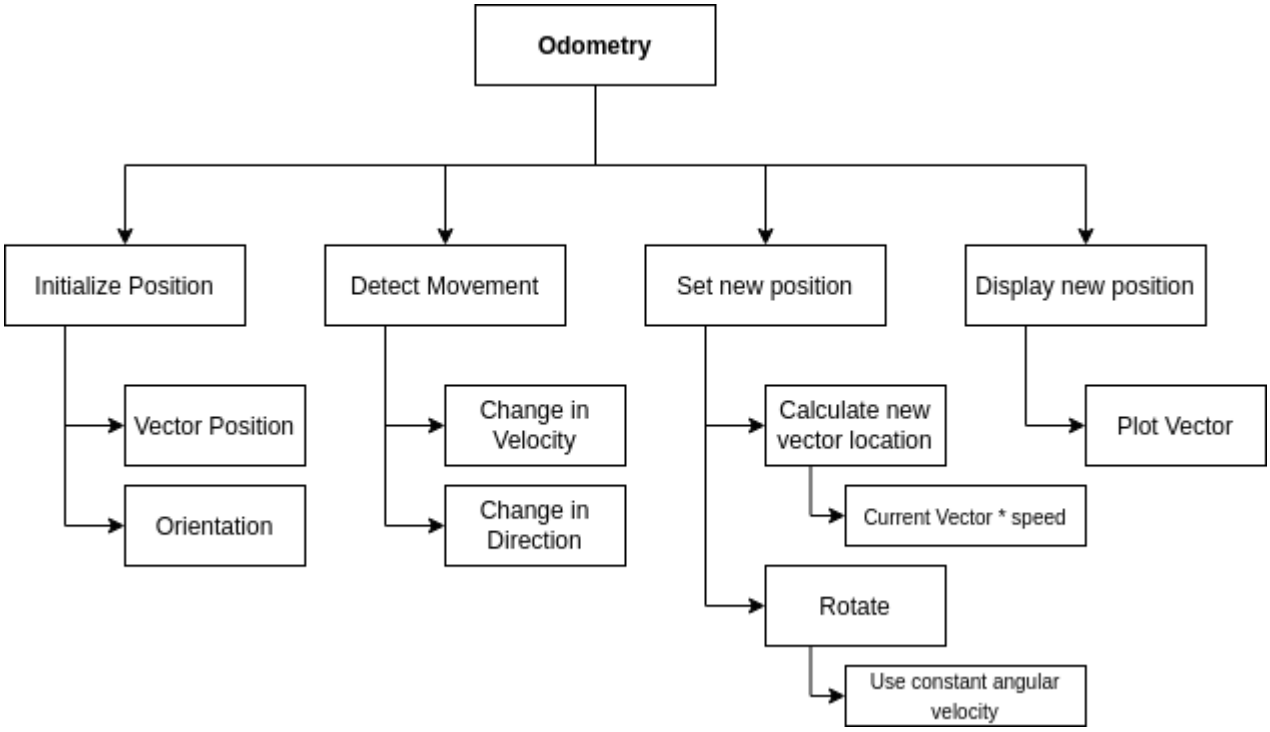
My project will be written using "processing 3" which is a language based on Java. Processing 3 is an open-source graphical library and integrated development environment. This is primarily because it is compatible with all raspberry pi processors, and it also allows for hardware control, which I will need to control the input/output devices on the rover. The libraries that I will use are:

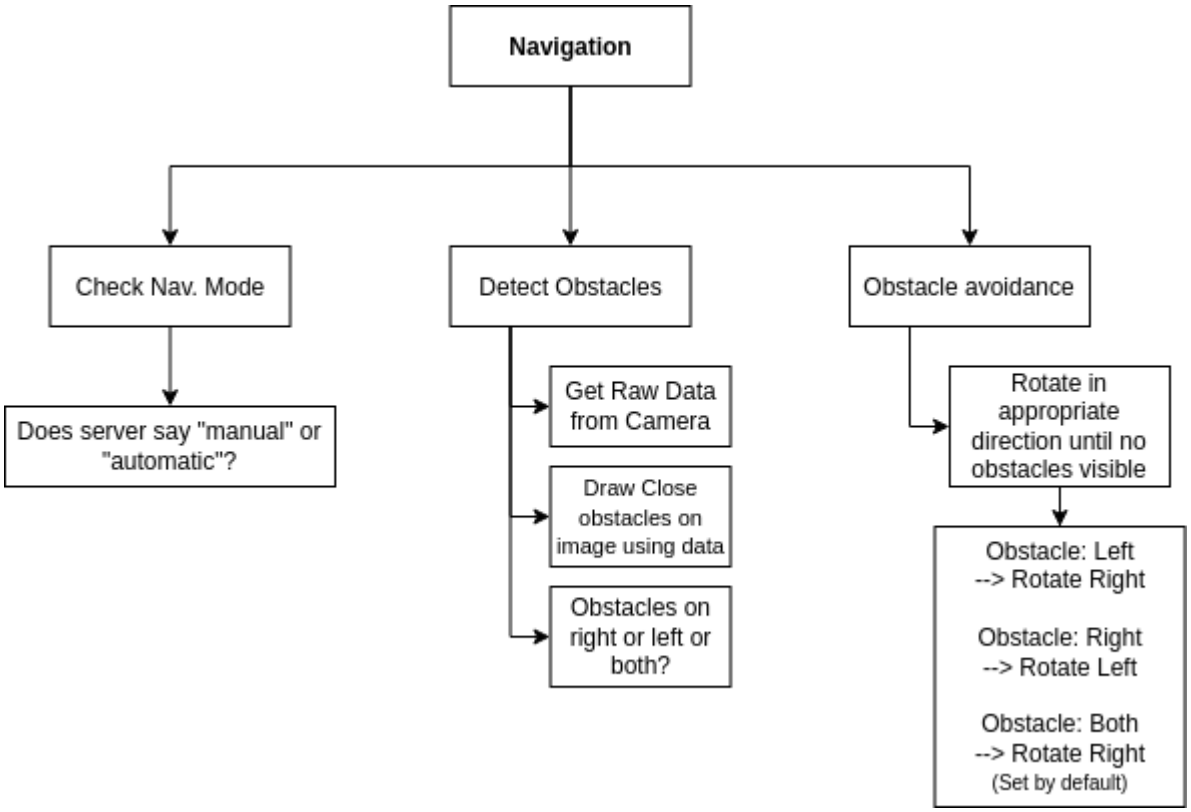
- Insert Libraries used here.

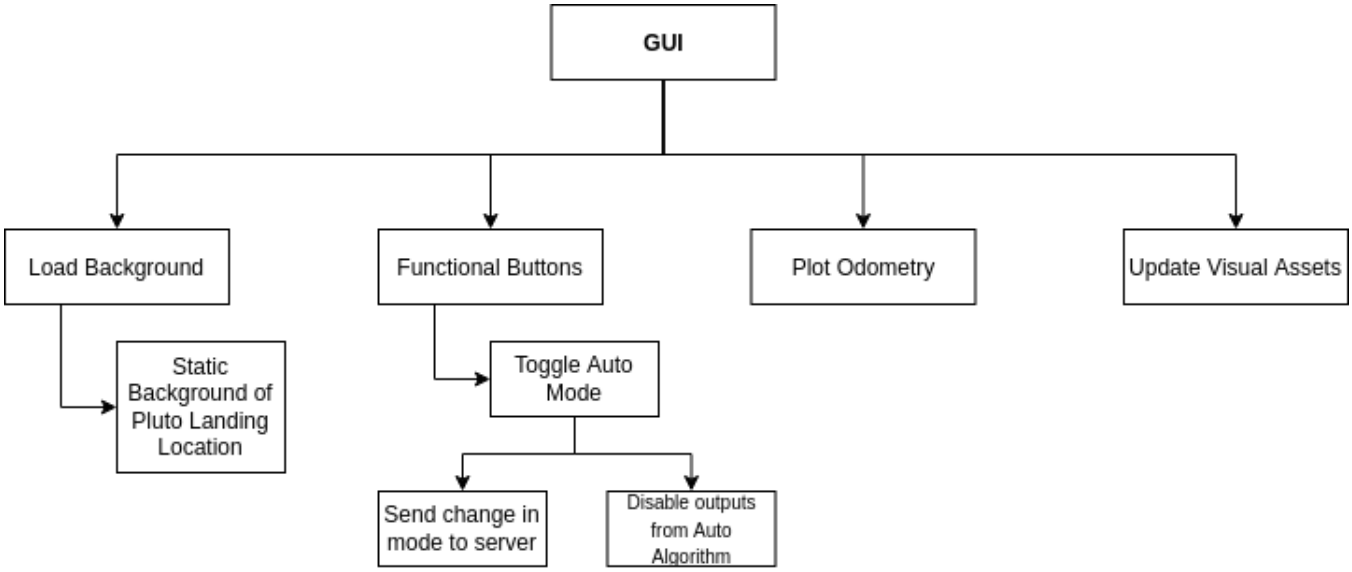
Along with this, I will also use C++ to program the Arduino to work as a driver board. It will be useful for using the digital signals from the Raspberry pi to toggle on/off the motors.

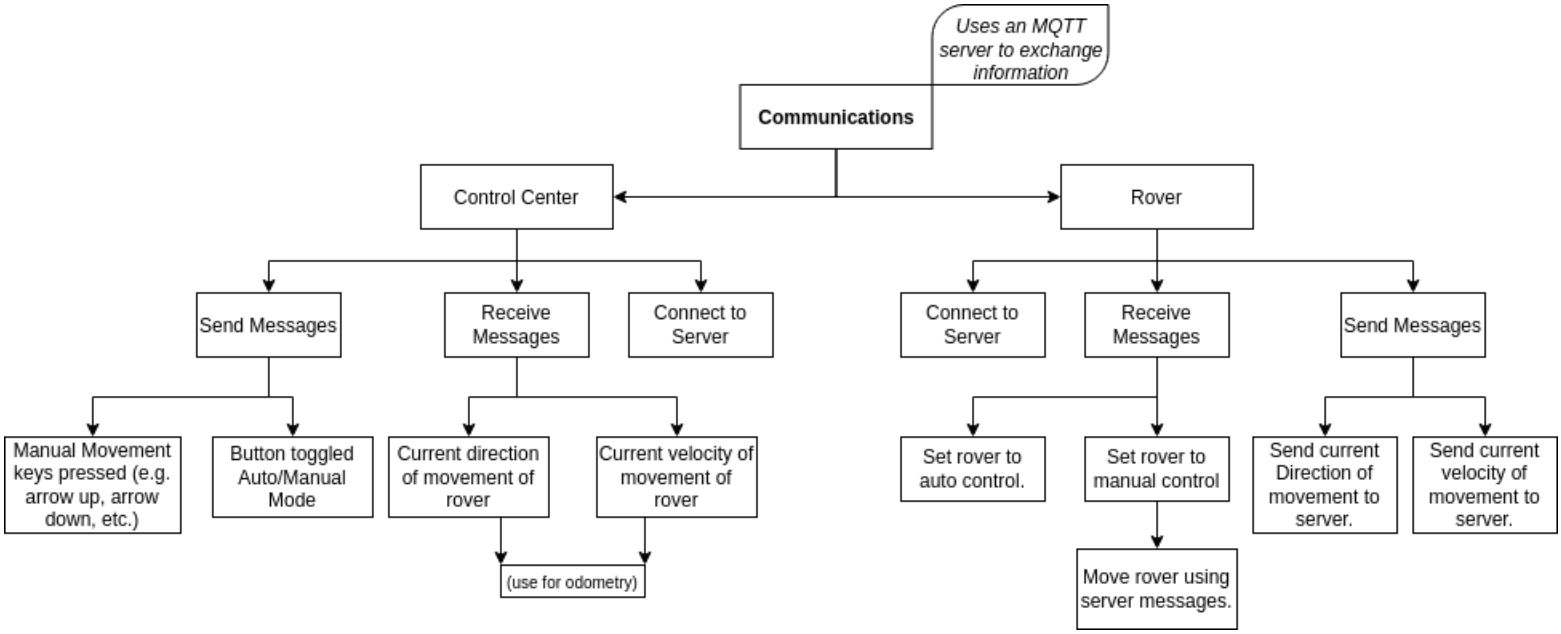
2.2 Top Down Design

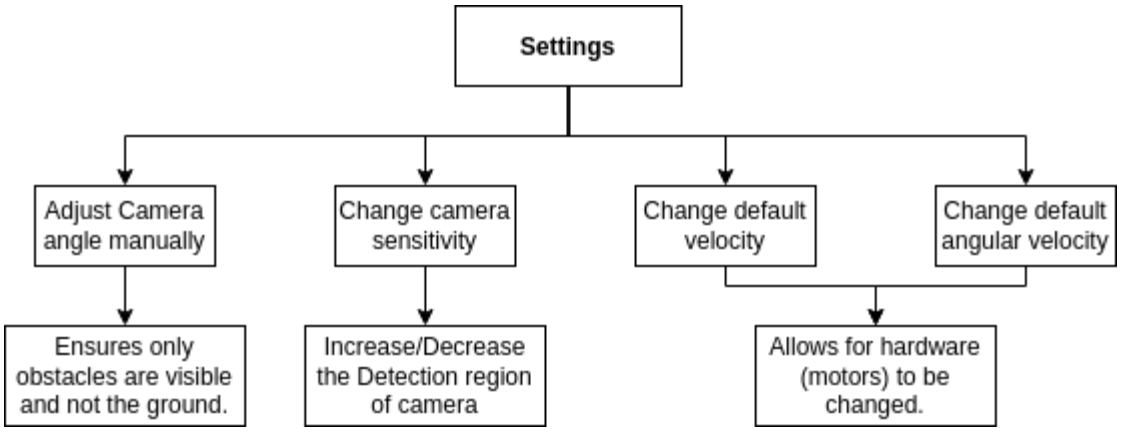



Odometry
Initialize Position
Detect Movement
Set new position
Display new position
Vector Position
Orientation
Change in Velocity
Change in Direction
Calculate new vector location
Current Vector * speed
Rotate
Use constant angular velocity
Plot Vector


Navigation
Check Nav. Mode
Does server say "manual" or "automatic"?
Detect Obstacles
Get Raw Data from Camera
Draw Close obstacles on image using data
Obstacles on right or left or both?
Obstacle avoidance
Rotate in appropriate direction until no obstacles visible
Obstacle: Left
--> Rotate Right
Obstacle: Right
--> Rotate Left
Obstacle: Both
--> Rotate Right
(Set by default)


GUI
Load Background
Functional Buttons
Plot Odometry
Update Visual Assets
Static Background of Pluto Landing Location
Toggle Auto Mode
Send change in mode to server
Disable outputs from Auto Algorithm


Uses an MQTT server to exchange information
Communications
Control Center
Rover
Send Messages
Receive Messages
Connect to Server
Connect to Server
Receive Messages
Send Messages
Manual Movement keys pressed (e.g. arrow up, arrow down, etc.)
Button toggled Auto/Manual Mode
Current direction of movement of rover
Current velocity of movement of rover
(use for odometry)
Set rover to auto control.
Set rover to manual control
Send current Direction of movement to server.
Send current velocity of movement to server.
Move rover using server messages.



2.3 Explaining the Modules

2.3.1 Odometry

2.3.1.1 User Interface

The user will be able to view the odometry of the rover on the GUI. It will be presented as a point on a map of Pluto. This will allow the user to view the current location of the rover compared to its landing spot. The relative coordinate location will also be displayed for recording purposes.

2.3.1.2 Logical

This module will receive the current direction of movement and the current speed of the rover and it will perform calculations (using the speed distance time formula) to move the digital model of the rover in the same direction. This aims to represent the displacement of the rover digitally, allowing the new location of the rover to be traced in relation to its landing position.

2.3.2 Navigation

2.3.2.1 User Interface

There will be no visual aspect for the user to view the navigation of the rover as the navigation processing will be done on the rover's inbuilt computer. This is to reduce the amount of data being sent between the rover and the control station as there is highly varying feedback lag times. But, the user will be able to use buttons to control the rover when in manual control.

2.3.2.2 Logical

This module will use the raw data from the rover camera to detect any obstacles in its path. The depth information from the camera will be used to abstract obstacles close to the rover, posing a threat, from obstacles further away. The module will then decide the best path for the rover around the obstacle and output that.

2.3.3 GUI

2.3.3.1 User Interface

This module will present the graphical user interface to the user using a library. It will allow buttons, maps, menus and many other assets to be represented, allowing input and retrieval of information about the rover.

2.3.3.2 Logical

A library built into processing 3 (the programming language) will be used to create a simple GUI. It will have a large map of Pluto with the odometry of the rover, a settings menu to control the variables of the rover, a graph to represent the metal detector information, and a few other assets for the user to use.

2.3.4 Communications

2.3.4.1 User Interface

This module will be used every time the user presses any of the buttons on the user interface. It will also constantly be used to change the location of the rover on the map. All events occurring on the user interface will either send a message to the rover (such as key pressed) or be caused due to a message received from the rover (such change in direction).

2.3.4.2 Logical

Every time there is an event that has occurred on the rover's computer or on the user's computer, a message will be sent to the other computer to ensure that any consequent actions that need to taken have been addressed. This module allows the two computers to communicate and so work together.

2.3.5 Settings

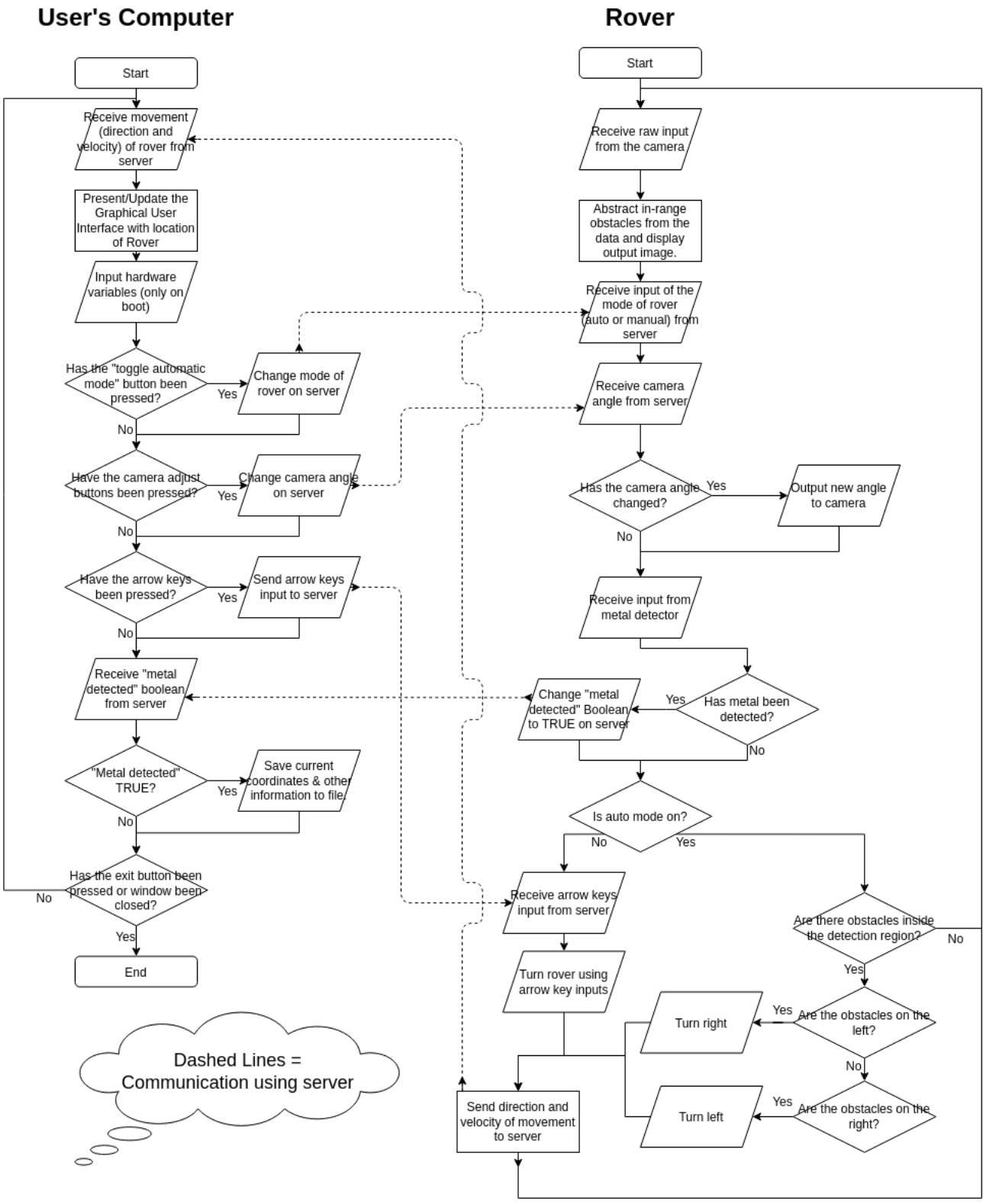
2.3.5.1 User Interface

The user will be able to control the angle of the camera to avoid recognizing the ground as an obstacle. The user will also be able to change the sensitivity of the camera (how far the obstacles detected can be and how much of the camera's width is used as the detection region) and they can also input the variables referring to the hardware of the rover.

2.3.5.2 Logical

This module will be responsible for using the user inputs to change the variables that map the program to the hardware, such as the velocity and angular velocity of the rover. It will also be used to adjust the camera, which will be helpful in optimising obstacle detection and also troubleshooting and debugging the rover in case of any problems.

2.4 Flowchart



2.5 Algorithms

2.5.1 Odometry

The first part of presenting the odometry to the user will be to connect the rover and the user's computer to the MQTT server. To do this, we will use the MQTT library, allowing us to connect and communicate using the MQTT protocol.

Rover AND User's Computer:

```
// Import the MQTT library
import mqtt.*;

PROCEDURE setup()
   client = new MQTTClient(this) //Create a client to connect to the server.
   client.connect("mqtt://bca037b2:c6847c9a415bf357@broker.shiftr.io") //Specify the
       address of the server
```

Then, as the rover moves, we will need to send a message to the server to state that it is moving, along with the direction that it is moving in. To do this, we will construct a "topic" on the server named "movement". This topic will only contain messages regarding the direction of the movement of the rover, therefore if it receives a message, than we know that the rover is moving, and that message will also hold information on which direction it is moving in. This message can be read by the user's computer and can be used to perform odometry calculations.

Rover:

```
PROCEDURE move_forward()
   //Insert code to move rover forward
   client.publish("/movement","forward") //Send message to server - Rover is moving
       forward
   movement_current = "forward" //Use local variable to keep track of movement

PROCEDURE move_right()
   //Insert code to turn rover clockwise
   client.publish("/movement","right") //Send message to server - Rover turning right
   movement_current = "right" //Use local variable to keep track of movement

PROCEDURE move_left()
   //Insert code to turn rover anticlockwise
   client.publish("/movement","left") //Send message to server - Rover turning left
   movement_current = "left" //Use local variable to keep track of movement

PROCEDURE move_back()
   //Insert code to move rover backwards
   client.publish("/movement","back") //Send message to server - Rover is moving
       backwards
   movement_current = "back" //Use local variable to keep track of movement
```

User's Computer:

```
PVector location_init = new PVector(500,750) //Sets initial position of the mover object
    on GUI

CLASS Mover {

   // Our object has two PVectors: location and velocity
   PVector location
   PVector location_relative
   Float direction

   PROCEDURE __init__()
      location = location_init //Set the initial position of the object using global
          variable
      direction = 0.0 // Set initial direction that the object will be pointing (straight
          up)

   PROCEDURE update()
      PVector velocity = new PVector(0,0) //Locations changes by velocity.

      if (inputdata.equals("forward"))
         velocity = new PVector(0,-speed)
         velocity.rotate(direction)

      if (inputdata.equals("left"))
         velocity = new PVector(0,0)
         direction += -0.001
         velocity.rotate(direction)

      if (inputdata.equals("back"))
         velocity = new PVector(0,speed)
         velocity.rotate(direction)

      if (inputdata.equals("right"))
         velocity = new PVector(0,0)
         direction += 0.001
         velocity.rotate(direction)

      if (inputdata.equals("stop"))
         velocity = new PVector(0,0)
         velocity.rotate(direction)

      //The above calculate the velocity of the rover using the fact that it is moving
          forward/backward or is rotating on the spot.

      location.add(velocity) //Calculated the new position of the rover.
      location_relative = new PVector(location.x - location_init.x, location.y -
          location_init.y) //Calculates the change in position of the rover from it's
          starting position, aka the Odometry.
```

```
PROCEDURE messageReceived(String topic, byte[] payload) //Procedure executed by library
    when a message is received
   inputdata = new String(payload)
   print(topic + "" + inputdata)

   if (inputdata == "forward")
         movement = "forward"
      else if (inputdata == "right")
         movement = "right"
      else if (inputdata == "left")
         movement = "left"
      else if (inputdata == "back")
         movement = "back"
      else
         movement = "stop"

      Mover.update() //Updates the location and direction of the "Mover" (representing
          the rover) every time a message is received.
```

2.5.2 Navigation

The navigation system can be used in 2 ways: autonomous navigation and manual navigation. The autonomous navigation method uses a computational approach to identify and avoid obstacles. The manual navigation can be used to override this computational approach, which can be useful for a few specific scenarios.

Most of the autonomous navigation processing will be done on the rover. A library will also be used to communicate with the camera sensors.

Rover:

```
import org.openkinect.freenect.*
import org.openkinect.processing.*

Kinect kinect //Declare "kinect" as a Kinect object

//initialise variables representing obstacles (red = obstacle)
Boolean isred = false
Boolean isred_right = false
Boolean isred_left = false

PImage depthImg //declare a new image named "depthImg" (blank pixels)

//Set the threshold for the depth of obstacles detected.
float minDepth = 0
float maxDepth = 500

// A variable holds the angle of the camera
float angle
```

```
PROCEDURE setup()
   kinect = new Kinect(this) //initialise camera object
   kinect.initDepth(); //initialise the depth of the camera
   angle = kinect.getTilt(); //initialise the tilt of the camera

   depthImg = new PImage(kinect.width, kinect.height); //Generate an image canvas - same
       size as camera output
   depthImg.loadPixels();

//A loop is then created with the help of processing 3 to display the image "depthImg"
    and update it.

PROCEDURE draw()
   // Threshold the depth image (abstract close obstacles only)
   int[] rawDepth = kinect.getRawDepth();
   for (int i=0; i < rawDepth.length; i++) //Loop through all pixels in image from kinect
      if (rawDepth[i] >= minDepth && rawDepth[i] <= maxDepth) //Select images in region
        depthImg.pixels[i] = color(252,3,3) //Color them red
      else
        depthImg.pixels[i] = color(0); //Color others black

   // Draw the filtered image
   depthImg.updatePixels()
   image(depthImg, 0, 0)

   checkredright() //Used to check if there is red displayed on the right, and so if
       there is an obstacle on the right.

PROCEDURE checkredright() //Check for obstacles on right side
   isred_right = false
   for(int a=(depthImg.width / 2); a < (depthImg.width)
      for(int b=0; b < depthImg.height; b++)          //Cycle through all pixels on right
         if(depthImg.get(a,b) == color(252,3,3)) //Check if red
            isred_right = true

   checkredleft() //Repeat for left side

PROCEDURE checkredleft() //Check for obstacles on left side
   isred_left = false
   for(int a=0; a < (depthImg.width / 2); a++)
      for(int b=0; b < depthImg.height; b++)          //Cycle through all pixels on left
         if(depthImg.get(a,b) == color(252,3,3)) //Check if red
            isred_left = true

   checkred() //procedure for decision making.

PROCEDURE checkred() //Decision on obstacle avoidance
   if(isred_right == true && isred_left == false)        // avoid obstacle on right
      move_left()                                          // by turning left
   else if(isred_right == false && isred_left == true)     // avoid obstacle on left
      move_right()                                         // by turning right
   else if(isred_right == true && isred_left == true)    // avoid obstacles on both sides
      move_right()                                         // by turning right
```

For manual navigation, there is a simple code on the user side, passing messages to the rover.

User's Computer:

```
PROCEDURE keyPressed() //Sends message to rover when a key is pressed.
  if (key == 'w')
    client.publish("/GPIOout", "forward -m")

  if (key == 'a')
    client.publish("/GPIOout", "left -m")

  if (key == 's')
    client.publish("/GPIOout", "back -m")

  if (key == 'd')
    client.publish("/GPIOout", "right -m")

  if (key == 'x')
    client.publish("/GPIOout", "stop -m")
```

Rover:

```
PROCEDURE messageReceived(String topic, byte[] payload) //Decode message from server
  message = new String(payload) //store message data in a variable

  //Manually move rover according to message received

  if(message.equals("forward -m"))
    movement_manual = "forward"
    move_forward()

  else if (message.equals("right -m"))
    movement_manual = "right"
    move_right()

  else if (message.equals("left -m"))
    movement_manual = "left"
    move_left()

  else if (message.equals("back -m"))
    movement_manual = "back"
    move_right()

  else if (message.equals("stop -m"))
    movement_manual = "stop"
    stop()
```

2.5.3 GUI

The coding for the graphical user interface will be pretty simple as processing 3 has a GUI library built in, making it easier to create basic visual assets. All of the code for the GUI will be on the user's computer as the program on the rover will not need to have a user interface.

Processing 3 also reserves the procedure "draw()" for presenting the GUI. It loops the contents of "draw()" allowing the GUI to be responsive.

User's Computer

```
import controlP5.* //library to add buttons
PImage img //Declare background image

cp5 = new ControlP5(this) //create a new controller for the buttons

img = loadImage("pluto-surface.jpg") //import the background image
img.resize(width,height) //resize image to fit window

PROCEDURE display_mover() //used to display mover object on the GUI
   //Create a custom shape to represent the rover on the surface of Pluto
   stroke(0);
   fill(175);
   ellipse(location.x,location.y,16,16);
   PVector line = new PVector(0, -10);
   line.rotate(direction);
   line(location.x,location.y,location.x+line.x,location.y+line.y);

PROCEDURE draw()
   size(1920, 1080) //set window width and height

   background(img) //apply background

   // Call functions on Mover object.
   mover.update()
   mover.disp() = display_mover() //add "display_mover()" method to object "mover"
   mover.disp() //Display the mover object (that represents the odometry of the rover)

   cp5.addToggle("ToggleAuto").setPosition(width-450,50) //Add toggle AUTO button on GUI
      .onpress(client.publish("/GPIOout", "setauto"),
               client.publish("/GPIOout", "setmanual")) //When toggled, it will publish
                   either "setauto" or "setmanual" to the server to change the mode of
                   the rover.

   myChart = cp5.addChart("Metal Detector") //Add a line graph to show the info from
              .setPosition(width-325, 50)   //the metal detector
              .setSize(300, 200)
              .setRange(-1, 150)
              .setView(Chart.LINE)

    draw_settings()
```

2.5.4 Communications

The communications module is responsible for ensuring that the program running on the user's computer is in sync with the program running on the rover. This module utilises a lot of the library procedures and rules built into the MQTT protocol library. For each program (on user's computer and the rover), the communications module is split into the sections:

1. **Connecting to the server**
2. **Receiving messages from the server**
3. **Sending messages to the server**

The algorithm used for each section is identical for both the user's computer and the rover.

1. ***Connecting to the server:***

```
MQTTClient client

client = new MQTTClient(this) // Create a new client to connect to the server.
client.connect("mqtt://bca037b2:c6847c9a415bf357@broker.shiftr.io"); // Provide
    server address
```

   This process uses a MQTT broker, which is a server that can hold many MQTT packets and is used for IoT protocols. The broker being used here is "Shiftr.io", and the address of this server is provided to the MQTTClient object, which uses other procedures within the MQTT library to connect to the server.

2. ***Receiving messages from the server:***

```
PROCEDURE messageReceived(String topic, byte[] payload) //This is a procedure
    that is required for the MQTT library as it is called every time there is an
    incoming message from the MQTT server. The message is split into the "topic"
    and the information that it was holding, known as its "payload".

   inputdata = new String(payload) //this variable now holds the information
      received from the server and can now be used within the program.
```

3. ***Sending messages to the server:***
   Sending a message is very easy as it only requires 1 line of code and most of the processing and encoding is done by the MQTT library. It is as simple as:

```
client.publish("/onthistopic", "thismessage")
```

2.5.5 Settings

Settings will have a similar algorithm to the GUI, as it will use the same library that us built into processing 3, but will require different inputs and will take different actions. Some inputs will change local variables that change the way that the program runs, and some may send messages to the server to change the state of the rover.

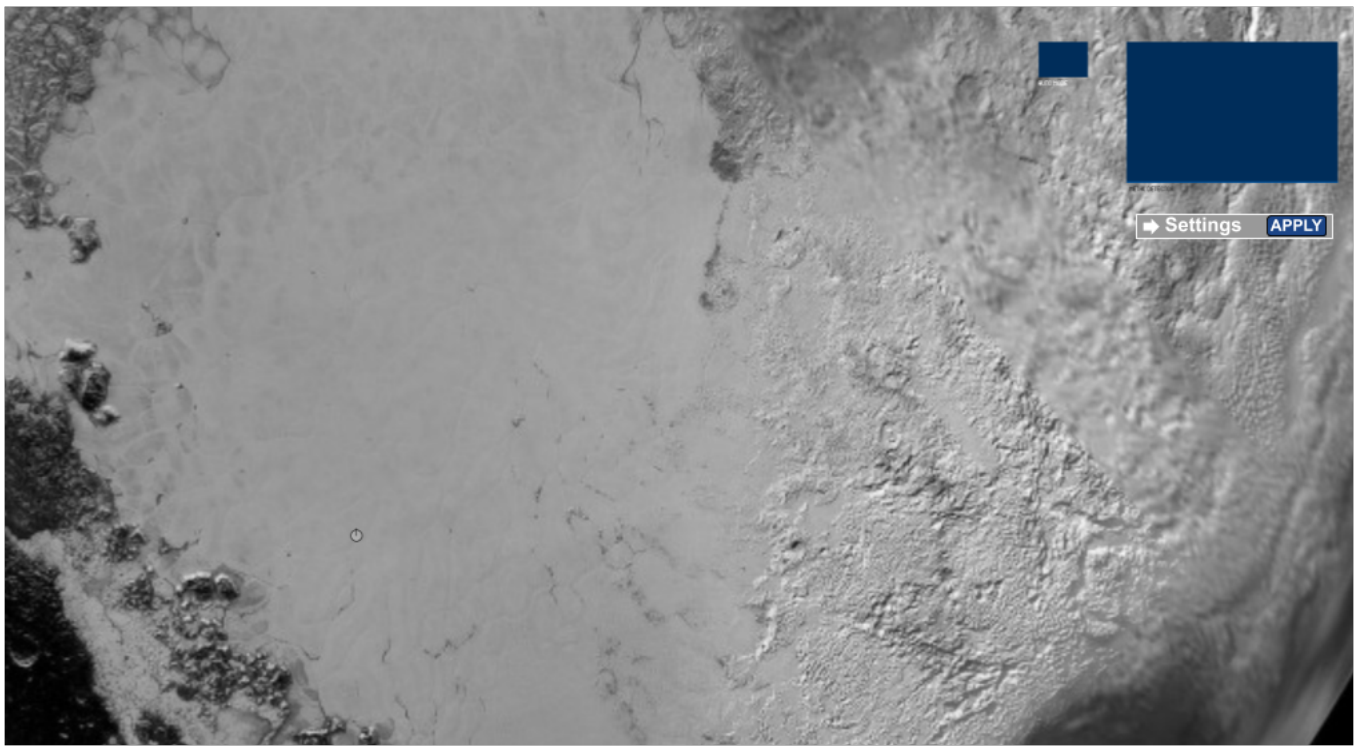
User's Computer:

```
PROCEDURE draw_settings()
   dropdown{
         camera_tilt = textbox(""),
         detection_region_padding = textbox(""),
         depth_threshold = textbox(""),
         default_speed = textbox(""),
         default_angular_speed = textbox(""),
         cp5.button("Apply")
            .onpress( apply_settings(camera_tilt, detection_region_padding,
                depth_threshold, default_speed, default_angular_speed) )}

PROCEDURE apply_settings(tilt, padding, threshold, d_speed, d_ang_speed) //Apply the new
    settings
   // Send required settings to server
   client.publish("camera_tilt", tilt)
   client.publish("det_region_padding", padding)
   client.publish("threshold", threshold)
   // Change required local variables
   speed = d_speed
   angular_speed = d_ang_speed
```

The above messages sent to the rover to change settings will be received using the algorithm described previously in the communications section. After receiving the appropriate message to change the settings, the code will perform hardware changes to match the server-side changes. For example, if the camera tilt angle is changed, that the rover will push the new angle to the camera and change it to match the server.

2.6 Usability Features UI design

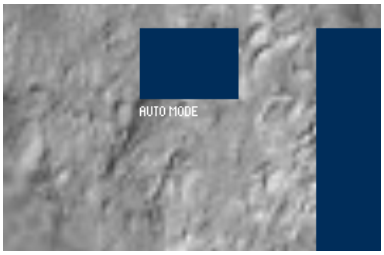
2.6.1 Interface Layout

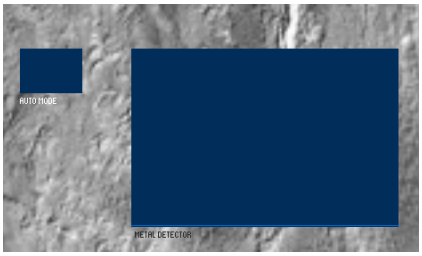


This is the graphical user interface that will be presented on the end user's computer. The background is the map of the landing and exploration region of the rover on Pluto.

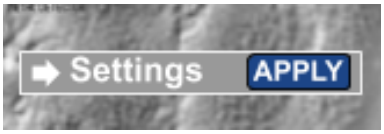
A small circle with a line represents the location current location of the rover on the surface of Pluto. The line represents the direction in which the rover is facing. Whenever the rover moves, the small circle with a line will also move to represent the displacement of the rover and any change in direction. This allows the user to know the rough location of the rover relative to its starting position without satellite images, making it cheaper and faster to locate the rover.

There is a large button on the top right labelled "Auto Mode". This buttons allows the user to toggle between the autonomous navigation built into the rover, and manual control very fast, which is useful for scenarios where the autonomous navigation is not required or does not perform as needed.

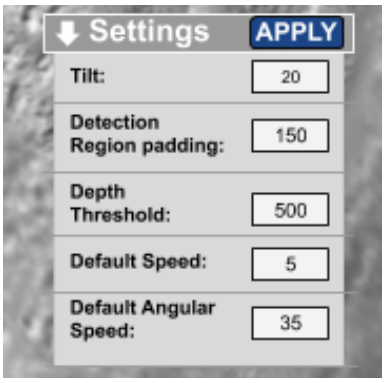




The GUI also contains a live line graph on the top right that presents the recordings from the metal detector. the image on the left shows the line at 0 as there is not metal detector input from the server. Whenever the line goes above a threshold, a metal can be said to be detected and the coordinated of the rover will be stored into a file.



The user is also presented with a dropdown menu which present the settings. This is useful as they are safely hidden out of sight when not needed, and there is a reduced risk of accidental changes.



Once the settings dropdown is opened, it present the different variables that can be adjusted to change the function of the program. All the different variables are named and there is a integer input textbox to increase precision. Once the numbers are input, the user can click the "apply" button which will send the required changes to the server and make any local adjustments using the algorithm outlines in the previous section.

2.7 Key Variables and Objects

Variable Name	Data Type	Purpose
modecheck	String	Set the mode that the rover is in (manual or auto).
message	String	stores incoming message from server.
isred_left	Boolean	True when obstacles on the left of the rover.
isred_right	Boolean	True when obstacles on the right of the rover.
isred	Boolean	True when any obstacles are within the detection region.
maxdepth	Integer	Stores the maximum distance that an obstacle can be.
rawDepth	Array	Input depth image data from the camera.
pin_fwd	Integer	GPIO pin number to move rover forward.
pin_right	Integer	GPIO pin number to rotate rover clockwise.
pin_left	Integer	GPIO pin number to rotate rover anti-clockwise.
pin_back	Integer	GPIO pin number to reverse rover.

Object name	Object function
kinect	This object represents the camera of the rover and will be used to perform hardware actions on the camera (such as adjust tilt) as well as retrieve information from it's sensors.
client	This object is used as a messenger that connects to the server. It can receive and send messages to the specified MQTT server.
Mover	This object is a physical representation of the rover on a virtual Pluto map. This object will be able to move around the map, perform identical moves to the actual rover in real time.

2.8 Testing

2.8.1 Alpha Testing

Alpha testing is a type of user acceptance testing that is performed to identify bugs before the final program is released. The goal of this testing is to perform common tasks that a normal user may use, and then check if the program completes these tasks successfully.

Function	Justification	Method of Input	Input	Expected Result
Connecting to rover	The user needs to connect to the rover using their program	Opening the application on the user's computer	Double-click app icon	Program on user's computer displays that the rover is connected
Change settings	The user should be able to adjust the settings using the GUI.	the user would change the settings by inputting new numbers into the boxes. This will then change the speed of the rover on the GUI and camera sensitivity.	Increase all values in the settings.	The rover moves faster on the GUI and the it avoids obstacles much sooner than before as they are detected from further away.

Changing rover navigation mode	This function toggles the rover from auto mode to manual control mode and vice versa	The user will need to toggle a button on and off, which will send a message to server to change the mode	Click Button	The button should change colour to indicate that the rover has changed modes.
Avoid obstacles autonomously	One of the main features of the rover is to avoid obstacles autonomously, and so it should be able to navigate around obstacles without any assistance.	The user will need to turn on the autonomous system and provide obstacles in front of the rover to test if it can avoid them.	Use the rover navigation mode button to change the rover's mode to auto	The rover will avoid all the obstacles that it has in it's path and will move in a path without interacting with the obstacles in any physical way.
Manual navigation	In certain scenarios, the rover will need to be controlled manually by the user, and so the manual navigation system should be working as required.	The user will put the rover in manual control mode and use the arrow keys on the keyboard to control the movement of the rover. Pressing the keys sends a message to the server which in turn moves the rover.	The user will change the rover navigation mode to "manual" and then use the arrow keys to control the rover movement.	When the arrow keys are pressed, the rover responds and moves in the related direction within seconds, allowing for real-time control.
Metal Detection	The main aim of the rover is to detect metals on the surface of Pluto, and so it should be able to do that successfully.	The user will need to place a metallic object near the detection system, which should detect it.	Place a metallic sheet under rover and then remove it.	There should be a spike in the metal detection graph and the current date, time and location of rover should be stored in a file.

2.8.2 Robustness Testing

This testing is used to check how robust a program is. It checks how well the program handles unexpected termination and unexpected actions.

Function	Justification	Method of Input	Input	Expected Result
Run program for 24 hours	This program will need to be able to run on both the rover and the user's computer for prolonged periods of time.	The user will use as many functions of the program as possible within 24 hours	Turn both programs on and use different functions frequently	Both programs should work in the same way as they were at the start of the test. There should be no noticeable lag as the program is not memory intensive.
User inputs words in place of numbers in settings	The program will need to regulate the inputs and handle unexpected formats.	The user will use a keyboard to input letters and unexpected characters into the settings menu of the program.	Words such as "Zero" or "Testing!" inputted into the settings menu.	In such a scenario, the inputs will be ignored and not sent to a server, and a subtle message will be displayed to the user.
Press random keys that are not specified as a useful key	This is important as it ensures that the user does not complete an unwanted action without knowing it.	Use the keyboard to press all the keys except the arrow keys and any shortcut buttons set for the program	Press all the keys on keyboard except the shortcut and arrow keys.	The program should not respond with any action.
Delete the metal detection storage file	This is vital as this file may become corrupt at some stage or may not exist when the program is setup.	The user will delete or purposely corrupt the file storing the details of places where metal was detected.	Delete the storage file and place metal sheet under rover.	The program should create a new file with a new name and store the details of the newly detected metal in that file.

2.8.3 Usability Testing

In this type of testing, program is given to the user to perform common tasks, and their ability to complete these tasks using the program is evaluated. This allows the us to know how suitable the program us for the intended user.

Task	**Expected Result**	**Pass or fail?**
Change the settings of the rover	The location of the settings on the GUI is easily accessible and the textbox + button system is very intuitive, therefore it should be easy for the user to change any of the settings.	This function will pass if the user can change the ALL settings of the rover without any assistance.
Change the mode of the rover	The user will be able to change the mode of the rover easily as it is a large labelled button on the front page of the GUI.	This function will pass if the user can toggle between the modes 5 times without any assistance.
Navigate the rover autonomously	The user should be able to use the autonomous navigation feature of the rover to avoid obstacles.	This function will pass if the user can make the rover avoid 10 obstacles without the use of manual control.

Chapter 3

Development of the coded solution

I used an iterative approach to develop the final coded solution. This allowed the solution to change gradually to suit the client's requirements using their feedback.

Features to develop:

1. Odometry
2. Navigation
3. GUI
4. Communications
5. Settings

3.1 GUI and Odometry

3.1.1 Prototype 1

The first feature I decided to develop was the GUI. This allows me to create an application layout suited for the client easily, adding features after finalisation. It also gave me a canvas to draw the application appropriately. The development of the GUI also involved adding the location of the rover onto the map, therefore I also developed the odometry feature alongside the GUI.
Using the built in libraries that come with processing 3, I wrote the following code:

I first used the "setup" procedure that is used to initialize the GUI when the program starts. It only runs the code once on start-up. Then I use "size(...)" to constrain the size of the window and "nostroke()" to remove outlines from all objects. Then I load a background image and resize it, ready for further usage later on in the program.

```
void setup() {
  size(1920, 1080);
  noStroke();

  img = loadImage("pluto-surface.jpg");
  img.resize(width,height);
```

Then I used the package "controlcp5" to add a button and a metal detector chart to the GUI. The button is a toggle button that changes the boolean "ToggleAuto" when clicked. The button

controller and boolean is initialised at the start of the code. I also established the MQTT server connection using an imported mqtt package.

```
cp5 = new ControlP5(this);

  cp5.addToggle("ToggleAuto").setPosition(width-450,50).setSize(70,50)
     .setCaptionLabel("AUTO MODE")
     .setColorForeground(color(0,0,255))
     .setColorActive(color(0,255,0));

  myChart = cp5.addChart("Metal Detector")
              .setPosition(width-325, 50)
              .setSize(300, 200)
              .setRange(-1, 150)
              .setView(Chart.LINE) // use Chart.LINE, Chart.PIE, Chart.AREA,
                  Chart.BAR_CENTERED
              .setStrokeWeight(1.5)
              .setColorCaptionLabel(color(40))
              ;

  myChart.addDataSet("incoming");
  myChart.setData("incoming", new float[100]);

  client = new MQTTClient(this);
  client.connect("mqtt://bca037b2:c6847c9a415bf357@broker.shiftr.io");

  mover = new Mover();
```

After having initialized the GUI using the "setup" procedure, I used the built-in "draw" procedure which loops throughout the execution of the whole program. This allows me add live GUI features such as the rover location. To do this, an object (initialized outside "draw()") called "mover" will be used to represent the rover, and it will be repeatedly updated and plotted on the GUI. Then, I also use a long if statement to repeatedly check for updates on the toggle button. When it is updated, the if statement sends a signal to the MQTT server to inform the rover of the mode change.

```
void draw() {
  background(img);

  // Call functions on Mover object.
  mover.update();
  mover.display();

  if(ToggleAuto && ToggleAutoToggle) {
    client.publish("/GPIOout", "setauto");
    ToggleAutoToggle = false;
  } else if (ToggleAuto == false && ToggleAutoToggle == false){
    client.publish("/GPIOout", "setmanual");
    ToggleAutoToggle = true;
  }
}
```

The object "mover" that is used to represent the location of the rover on a map needs to be

initialized as an object. It needs to have methods that allow it's location to be updated and displayed on the GUI using the above "draw()" loop.
The object has 2 attributes: it's location, and it's direction.

```
class Mover {

  PVector location;
  Float direction;

  Mover() {
    location = location_init.copy();
    direction = 0.0;

  }
```

The "location_init" variable is used to set the starting location of the rover on the map (the landing location).

The first method required is the "update" method that allows the vector location of the rover to be changed upon movement confirmation from the mqtt server. As the rover moves in a certain direction, it's constant velocity and angular velocity is used to update the digital location on the GUI.

```
  void update() {
    PVector velocity = new PVector(0,0);
    // Motion 101: Locations changes by velocity.
    //print(inputdata);
    if (inputdata.equals("forward")) {
      velocity = new PVector(0,-speed);
      velocity.rotate(direction);
    }
    if (inputdata.equals("left")) {
      velocity = new PVector(0,0);
      direction += -0.001;
      velocity.rotate(direction);
    }
    if (inputdata.equals("back")) {
      velocity = new PVector(0,speed);
      velocity.rotate(direction);
    }
    if (inputdata.equals("right")) {
      velocity = new PVector(0,0);
      direction += 0.001;
      velocity.rotate(direction);
    }
    if (inputdata.equals("stop")) {
      velocity = new PVector(0,0);
      velocity.rotate(direction);
    }
    location.add(velocity);
  }
```

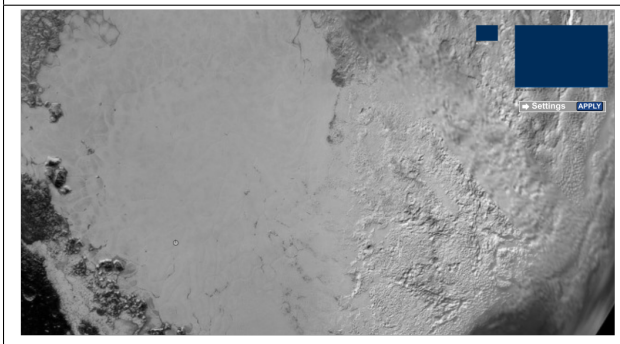
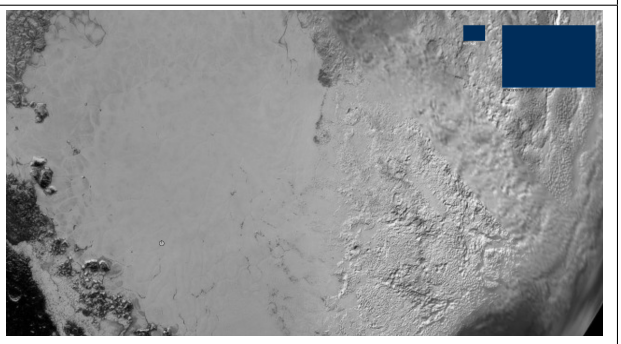
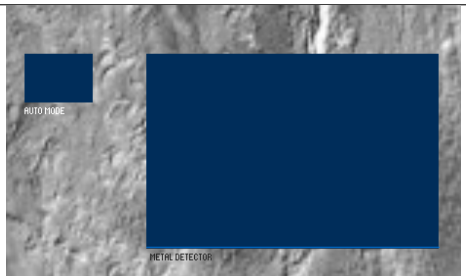
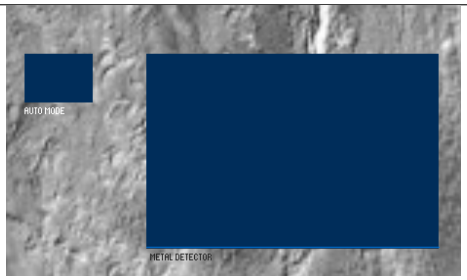
The second method required is the "display" method that is used to create a GUI visual element

and set it's x and y pixel location.

```
void display() {
  stroke(0);
  fill(175);
  ellipse(location.x,location.y,16,16);
  PVector line = new PVector(0, -10);
  line.rotate(direction);
  line(location.x,location.y,location.x+line.x,location.y+line.y);
}
```

3.1.2 Module Testing

From the testing plan in the previous chapter, there are not any relevant functions that can be tested at this stage of development. However, we can compare the current GUI to the interface layout plan in the previous chapter.

Layout Plan	Current GUI
	
	
	

The current GUI prototype matches the plans, with the current prototype only missing the settings tab. This is a feature that will be added at a later date as it is an extension to the functionality of the program.

3.1.3 User Feedback

The end users of the program reviewed the GUI at it's current state and gave feedback. They appreciated the simplicity of the GUI, and found it useful that the buttons and graphs were out of the way. They also liked the birds eye view of the landing location of the rover as it clearly and quickly shows it's rough location.

Their primary concern was the robustness of the GUI and odometry. The odometry tracks the rover's location and plots it on the GUI. But, what would happen if the rover's odometry indicator reaches the edge of the screen? The program cannot deal with this situation currently, therefore a change should be made prevents the indicator from leaving the edge of the screen. This ensures that even if the rover leaves the map, it's direction of travel is still displayed on the screen.

Using the user feedback, a second prototype was made.

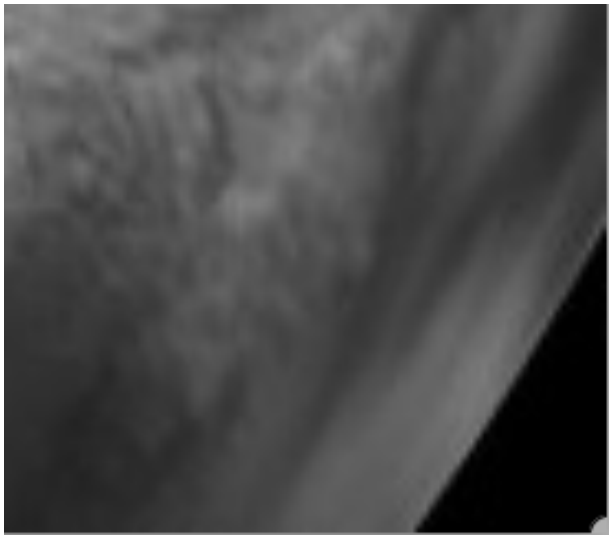
3.1.4 Prototype 2

To resolve the edge problem, I added a simple method to the "mover" object. This method, "checkEdges" compared the current digital vector location of the rover to the width and height of the screen. If the screen constraints are exceeded, the odometry tracking stops at the edge of the screen.

```
void checkEdges() {

  if (location.x <= width) {
    location.x = width;
  } else if (location.x >= 0) {
    location.x = 0;
  }

  if (location.y <= height) {
    location.y = height;
  } else if (location.y >= 0) {
    location.y = 0;
  }

}
```

But, after executing the new code, the rover odometry indicator seemed to have gone from it's original position and could just about be seen at the bottom right-hand side corner.

Original position (not visible anymore)



New position (visible in bottom right corner)

The code still runs without any errors and all other functionality is not affected, therefore this is most likely a logic error. As we know that this only happened with the new method added in this iteration, we can look at that specific snippet of code to find the error.

After using multiple "print(...)" lines in the code, I figured out that the IF statements "if (location.x <= width)" and "if (location.y <= height)" were constantly being passed/executed even when they are not supposed to. they must only be run when the rover odometry indicator is going off the screen.

After further review, I realised that this is an Arithmetic Logic Error as the wrong inequality symbols were used for the IF statements. To solve this issue, the inequality symbols were replaced with the correct ones, resulting in the following code:

```
void checkEdges() {

  if (location.x >= width) {
    location.x = width;
  } else if (location.x <= 0) {
    location.x = 0;
  }

  if (location.y >= height) {
    location.y = height;
  } else if (location.y <= 0) {
    location.y = 0;
  }
```

3.1.5 User Feedback

The end users were happy with the GUI of the program. Therefore this module is fully functional.

3.2 Navigation and Communication

3.2.1 Prototype 1

After developing the GUI for the user's computer, I decided to design the navigation algorithm for the rover. This algorithm runs on a raspberry pi and uses the sensors available to navigate around obstacles. To do this, it is first important to receive a live feed of data from the on-board sensors.

The primary sensor being used is the Xbox 360 kinect. To receive information from this camera, a library called "openkinect" is used. This library allows the Kinect to be initialized and used as an object in the code.

```
import org.openkinect.freenect.*;
import org.openkinect.processing.*;

import processing.io.*;

Kinect kinect;
```

To detect obstacles in the rovers path, we are going to get a depth image from the sensor of the objects in front. This allows us to detect the distance of anything in front of the rover. Then, we will set a threshold where if the object is closer than a set distance, it will be plotted on a canvas as red. The red now represents obstacles close to the rover. An algorithm will then be used to determine if there is any red on the left and/or on the right half of the displayed canvas. This will allow us to determine if the rover should turn right or left to best avoid the obstacles. The algorithm will continue turning until there is no red displayed on the canvas. In the case that there are obstacles on both right and left, the rover will just turn right until obstacles are no longer in its path.

Initializing required objects and variables:

```
String modecheck = "manual";
String message;
String movement_manual = "stop";
String movement_current;

// Red pixel detection
Boolean isred = false;
Boolean isred_right = false;
Boolean isred_left = false;

String movement;

// Depth image
PImage depthImg;

// Which pixels do we care about (thresholds)?
float minDepth = 0;
float maxDepth = 500;

// What is the kinect's angle
float angle;
```

```
boolean MANUAL = true;

// GPIO pin outputs
Integer pin_fwd = 5;
Integer pin_right = 6;
Integer pin_left = 13;
Integer pin_back = 19;
```

GPIO pin outputs will be used on the Raspberry pi to control the movement of the rover. Within the "setup" function of the code, the first thing to do is set the mode of the above GPIO pins to output pins.

```
void setup() {
  GPIO.pinMode(pin_fwd, GPIO.OUTPUT);
  GPIO.pinMode(pin_right, GPIO.OUTPUT);
  GPIO.pinMode(pin_left, GPIO.OUTPUT);
  GPIO.pinMode(pin_back, GPIO.OUTPUT);
```

Then, we set the display size of the canvas that will be used to plot red regions representing the obstacles.

```
  size(640, 480);
```

Then, still within the "setup" function, we create a new Kinect instance, start it's depth imaging function and import it's current tilt. We also create a blank image to display on the screen, which will be used as a canvas for displaying a red pixel representation of obstacles.

```
  kinect = new Kinect(this);
  kinect.initDepth();
  angle = kinect.getTilt();
  // Blank image
  depthImg = new PImage(kinect.width, kinect.height);
  depthImg.loadPixels();
}
```

After the "setup" function, we can use the "draw" function which loops throughout the whole execution of the program. Firstly, we can use this to display and update the canvas pixels on the screen. Then, we need to change the colours of the pixels according to the "rawDepth" data from the Kinect. We can also use the mouse location to calibrate the thresholds of the program, optimizing it for best performance, but this would only need to be done once for setup, then it is not needed (therefore it is commented out).

```
void draw() {
  // Draw the raw image
    depthImg.loadPixels();

  // Calibration
  //minDepth = map(mouseX,0,width, 0, 4500);
  //maxDepth = map(mouseY,0,height, 0, 4500);

  // Threshold the depth image
  int[] rawDepth = kinect.getRawDepth();
  for (int i=0; i < rawDepth.length; i++) {
```

```
    if (rawDepth[i] >= minDepth && rawDepth[i] <= maxDepth) {
      depthImg.pixels[i] = color(252,3,3);
    } else {
      depthImg.pixels[i] = color(0);
    }
  }

  // Draw the thresholded image
  depthImg.updatePixels();
  image(depthImg, 0, 0);

  //Comment for Calibration
  fill(0);
  text("TILT: " + angle, 10, 20);
  text("THRESHOLD: [" + minDepth + ", " + maxDepth + "]", 10, 36);

  //Calibration Text
  //fill(255);
  //textSize(32);
  //text(minDepth + " " + maxDepth, 10, 64);
  checkredright();
}
```

The "draw" loop also repeatedly calls the "checkredright" function, which checks if there are any red pixels on the right side of the displayed canvas. That function then calls another function which checks if there are any red pixels on the left side of the screen. After there functions are called, the obstacles in front would be established and action can be taken to avoid them.

But, before we continue to create the "checkredright" function, we need to be able to adjust the sensor tilt and the depth thresholds. Processing 3 allows us to use the "keyPressed()" procedure to do this. "keyPressed()" gets called anytime there is a keyboard input into the program. So using "keyPressed()" we can increase and decrease the required variables. I also used "contrain(...)" to ensure that the variables did not exceed their range.

```
// Adjust the angle and the depth threshold min and max
void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP) {
      angle++;
    } else if (keyCode == DOWN) {
      angle--;
    }
    angle = constrain(angle, -30, 30);
    kinect.setTilt(angle);
  } else if (key == 'a') {
    minDepth = constrain(minDepth+10, 0, maxDepth);
  } else if (key == 's') {
    minDepth = constrain(minDepth-10, 0, maxDepth);
  } else if (key == 'z') {
    maxDepth = constrain(maxDepth+10, minDepth, 2047);
  } else if (key =='x') {
    maxDepth = constrain(maxDepth-10, minDepth, 2047);
  }
```

```
}
```

So now we need to add the function "checkredright". All this function needs to do is to change the value of the variable "isred_right" if there are any red pixels on the right half of the screen. We can do this using two loops: one loop cycles through the horizontal axis, and the other loop checks the vertical axis for each horizontal value.

```
void checkredright() {
  isred_right = false;
  for(int a=(depthImg.width / 2); a < (depthImg.width); a++) {
    for(int b=0; b < depthImg.height; b++) {
      if(depthImg.get(a,b) == color(252,3,3)) {
        isred_right = true;
      }
    }
  }
  checkredleft();
}
```

"checkredleft()" is then called at the end of this procedure. The purpose of this is to replicate the process we just did on the right half, for the left half of the canvas.

```
void checkredleft() {
  isred_left = false;
  for(int a=0; a < (depthImg.width / 2); a++) {
    for(int b=0; b < depthImg.height; b++) {
      if(depthImg.get(a,b) == color(252,3,3)) {
        isred_left = true;
      }
    }
  }
  GPIOoutput();
}
```

After having checked both halves of the canvas for obstacles, the program will then need to output GPIO signals. To make the decisions on the movement using the right and left obstacle detection, this procedure is used:

```
void GPIOoutput() {
  //autonomous navigation = manual mode off
  if (MANUAL == false) {

    client.publish("/modecheck", "auto");

    if (movement == "forward") {
      move_forward();
    }
    if (movement == "right") {
      move_right();
    }
    if (movement == "left") {
      move_left();
    }
```

```
    if (movement == "back") {
      move_back();
    }
  }
}
```

```
void move_forward() {
  println("GPIO - FORWARD pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.HIGH);
  GPIO.digitalWrite(pin_right, GPIO.LOW);
  GPIO.digitalWrite(pin_left, GPIO.LOW);
  GPIO.digitalWrite(pin_back, GPIO.LOW);

  movement_current = "forward";
}
void move_right() {
  println("GPIO - RIGHT pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.LOW);
  GPIO.digitalWrite(pin_right, GPIO.HIGH);
  GPIO.digitalWrite(pin_left, GPIO.LOW);
  GPIO.digitalWrite(pin_back, GPIO.LOW);

  movement_current = "right";
}
void move_left() {
  println("GPIO - LEFT pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.LOW);
  GPIO.digitalWrite(pin_right, GPIO.LOW);
  GPIO.digitalWrite(pin_left, GPIO.HIGH);
  GPIO.digitalWrite(pin_back, GPIO.LOW);

  movement_current = "left";
}
void move_back() {
  println("GPIO - BACK pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.LOW);
  GPIO.digitalWrite(pin_right, GPIO.LOW);
  GPIO.digitalWrite(pin_left, GPIO.LOW);
  GPIO.digitalWrite(pin_back, GPIO.HIGH);
  movement_current = "back";
}
```

After having developed the autonomous navigation and object avoidance algorithm, the program needs to communicate with the MQTT server and send messages to the user's computer for the odometry, manual control and metal detection.

To do this, I again utilised a package "mqtt".

```
import mqtt.*;
```

After importing this package, I initialised the server object, and also connected to the required

server within the "setup" procedure. To ensure that the user end of the program and the rover both start on the same mode, I also published "manual" to a mqtt server node called "/modecheck". This can be used by the user end program to sync to the correct mode.

```
MQTTClient client;

void setup() {
   ...
  client = new MQTTClient(this);
  client.connect("mqtt://bca037b2:c6847c9a415bf357@broker.shiftr.io");
  client.publish("/modecheck", "manual");
   ...
}
```

When the program is in manual mode, the rover needs to receive signals from the user end to guide it's movement. For this, a node can be used to control the GPIO output directly from the user end. This node can be called "/GPIOout". This means that the rover needs to subscribe to this mqtt node.

```
void clientConnected() {
  println("client connected");

  client.subscribe("/GPIOout");
}

void messageReceived(String topic, byte[] payload) {
  message = new String(payload);
  //println("new message: " + topic + " - " + message);
  if(message.equals("setmanual")) {
    settomanual();
  }
  else if(message.equals("setauto")) {
    settoauto();
  }
  if(message.equals("forward -m")) {
    movement_manual = "forward";
  }
  else if (message.equals("right -m")) {
    movement_manual = "right";
  }
  else if (message.equals("left -m")) {
    movement_manual = "left";
  }
  else if (message.equals("back -m")) {
    movement_manual = "back";
  }
  else if (message.equals("stop -m")) {
    movement_manual = "stop";
  }

}
```

This node can also be used to send the rover instructions to toggle nodes. This would mean that

another node doesn't need to be created. When the rover received the toggle instructions, it calls a procedure to change the variable "MANUAL". To make the code more efficient and robust, an if statement is also used to only change the value of the variable when required.

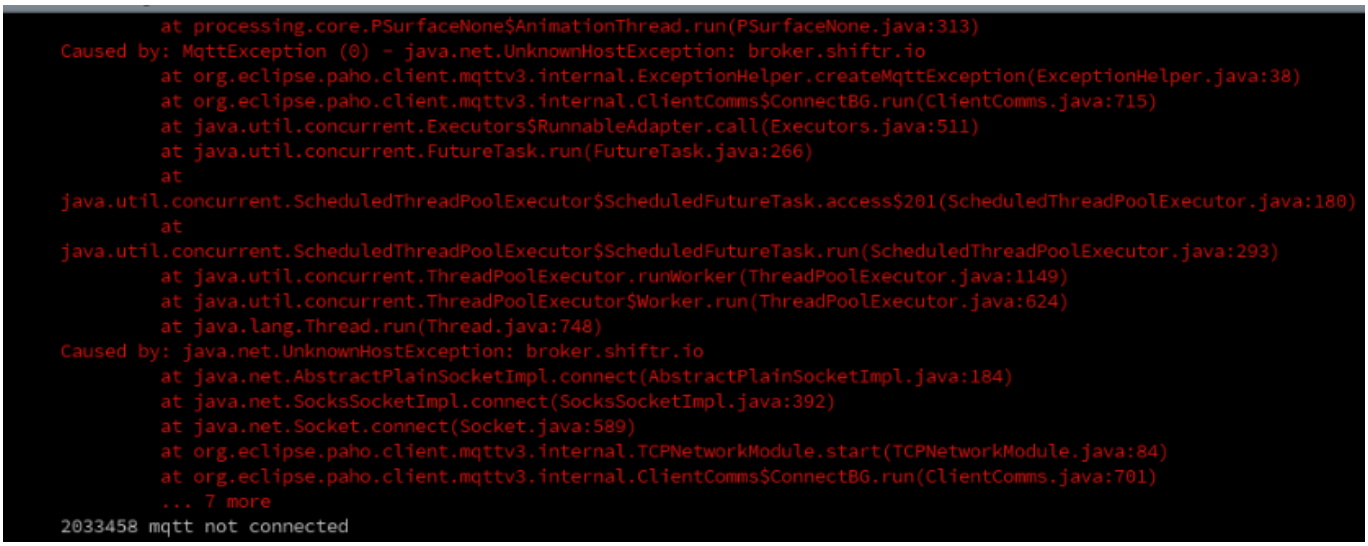
```
void settomanual() {
  if(MANUAL != true) {
    MANUAL = true;
    println("Set to MANUAL");
  }
}

void settoauto() {
  if(MANUAL) {
    MANUAL = false;
    println("Set to AUTO");
  }
}

void connectionLost() {
  println("connection lost");
}
```

So now we have made the communication network for the rover to receive manual movement information from the user end. But, this means that the "GPIOoutput()" procedure would need to be changed to output the movement instructions from the mqtt server instead of the autonomous algorithm instructions. This can be done by changing the "GPIOoutput()" procedure to this:

```
void GPIOoutput() {
  if (MANUAL == false) {
    client.publish("/modecheck", "auto");

    if (movement == "forward") {
      move_forward();
    }
    if (movement == "right") {
      move_right();
    }
    if (movement == "left") {
      move_left();
    }
    if (movement == "back") {
      move_back();
    }
  }


  if (MANUAL) {
    client.publish("/modecheck", "manual");

    if (movement_manual == "forward") {
        move_forward();
    }
    if (movement_manual == "right") {
```

```
        move_right();
    }
    if (movement_manual == "left") {
        move_left();
    }
    if (movement_manual == "back") {
        move_back();
    }
    if (movement_manual == "stop") {
        move_stop();
    }
  }
}
```

After quickly executing the code that we have so far, I ran into a runtime error.



It is clear that the problem with the program is that the MQTT server disconnects from this client after a few seconds of execution. This means that there must be an error on the server. After checking, there was a "Static Rate Limit reached" error. After reading the documentation of this MQTT borker, I found more information:

> "The static rate limit applies to all MQTT connections and will close the connection immediately if the connection goes over the threshold of 100 operations per second. For the static rate limit, we count every MQTT packet including (Ping) as operations against the rate limit."

To resolve this runtime error, we need to reduce the amount of information that we are writing to the server in the above procedure. While we are simple writing strings of the direction of movement, we have to remember that the GPIO output is called regularly in response to the input data from the Kinect. As the procedure is called so many times, we need to add a mechanism to only write to the server when there is a change in the direction.

A new variable ("movement_current") can be used to keep track of the movement that was last written to the server, or in other words, the direction that the rover is moving in according to the server. When the rover actually starts moving in a different direction to the one that is already written onto the server, a message will be sent to change the direction on the server. That will result in the following code:

```
void GPIOoutput() {
  if (MANUAL == false) {
    client.publish("/modecheck", "auto");

    if (movement == "forward") {
      if(movement_current != "forward") {
        move_forward();
      }
    }
    if (movement == "right") {
      if(movement_current != "right") {
        move_right();
      }
    }
    if (movement == "left") {
      if(movement_current != "left") {
        move_left();
      }
    }
    if (movement == "back") {
      if (movement_current != "back") {
        move_back();
      }
    }
  }
  if (MANUAL) {
    client.publish("/modecheck", "manual");

    if (movement_manual == "forward") {
      if(movement_current != "forward") {
        move_forward();
      }
    }
    if (movement_manual == "right") {
      if(movement_current != "right") {
        move_right();
      }
    }
    if (movement_manual == "left") {
      if(movement_current != "left") {
        move_left();
      }
    }
    if (movement_manual == "back") {
      if (movement_current != "back") {
        move_back();
```

```
    }
  }
  if (movement_manual == "stop") {
    if (movement_current != "stop") {
      move_stop();
    }
  }
 }
}
```

While I thought this would work, I got the same error but after a longer time. This means that the changes made a difference but there was still something in the procedure that caused the server to crash. As I couldn't observe the code and figure out the issue, I decided to observe the MQTT server for the incoming messages and work out from there which messages were causing the error. After trying a few times, I found that the message "manual" was being sent to the server very fast and very many times, causing the static rate limit to be reached within seconds. To confirm this, I also executed the code and printed onto the shell everytime a message was sent to the MQTT server. In the end, I found that the problem was the "client.publish("/modecheck", "auto");" and "client.publish("/modecheck", "manual");" line in the above code. To resolve the issue, I did the exact thing that I did previously, create another variable (modecheck) which keeps track of the mode on that was last written to the server, and only sends messages once a change has been made.

The final version of this procedure without any errors:

```
void GPIOoutput() {
 if (MANUAL == false) {
   if (modecheck != "auto") {
     client.publish("/modecheck", "auto");
     modecheck = "auto";
   }

   if (movement == "forward") {
     if(movement_current != "forward") {
       move_forward();
     }
   }
   if (movement == "right") {
     if(movement_current != "right") {
       move_right();
     }
   }
   if (movement == "left") {
     if(movement_current != "left") {
       move_left();
     }
   }
   if (movement == "back") {
     if (movement_current != "back") {
       move_back();
     }
   }
 }
```

```
  if (MANUAL) {
    if (modecheck != "manual") {
      client.publish("/modecheck", "manual");
      modecheck = "manual";
    }

    if (movement_manual == "forward") {
      if(movement_current != "forward") {
        move_forward();
      }
    }
    if (movement_manual == "right") {
      if(movement_current != "right") {
        move_right();
      }
    }
    if (movement_manual == "left") {
      if(movement_current != "left") {
        move_left();
      }
    }
    if (movement_manual == "back") {
      if (movement_current != "back") {
        move_back();
      }
    }
    if (movement_manual == "stop") {
      if (movement_current != "stop") {
        move_stop();
      }
    }
  }
}
```

To ensure that the digital odometry of the rover on the user end is accurate, we can use a individual node on the MQTT server to track the movements of the rover. To ensure the information on the server is also accurate, we can publish the movement of the rover to the server right after the GPIO output instructions. It would be efficient to use the user input when the rover is in manual mode as it would reduce the strain on the server, but this would be inaccurate as it a representation of the user input, rather than the actual movement of the rover. So, no matter the mode of the rover, I decided to use the MQTT server node called "/movement", which is written to right after the GPIO output code, for the odometry on the user end of the program. To do this, I replaced the GPIO output code with this new version that communicated with the MQTT server as well:

```
void move_forward() {
  println("GPIO - FORWARD pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.HIGH);
  GPIO.digitalWrite(pin_right, GPIO.LOW);
  GPIO.digitalWrite(pin_left, GPIO.LOW);
  GPIO.digitalWrite(pin_back, GPIO.LOW);
  client.publish("/movement","forward");

  movement_current = "forward";
}
void move_right() {
  println("GPIO - RIGHT pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.LOW);
  GPIO.digitalWrite(pin_right, GPIO.HIGH);
  GPIO.digitalWrite(pin_left, GPIO.LOW);
  GPIO.digitalWrite(pin_back, GPIO.LOW);
  client.publish("/movement","right");

  movement_current = "right";
}
void move_left() {
  println("GPIO - LEFT pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.LOW);
  GPIO.digitalWrite(pin_right, GPIO.LOW);
  GPIO.digitalWrite(pin_left, GPIO.HIGH);
  GPIO.digitalWrite(pin_back, GPIO.LOW);
  client.publish("/movement","left");

  movement_current = "left";
}
void move_back() {
  println("GPIO - BACK pin on ");
  GPIO.digitalWrite(pin_fwd, GPIO.LOW);
  GPIO.digitalWrite(pin_right, GPIO.LOW);
  GPIO.digitalWrite(pin_left, GPIO.LOW);
  GPIO.digitalWrite(pin_back, GPIO.HIGH);
  client.publish("/movement","back");

  movement_current = "back";
}
```

For this complete program to work, the user end program needs to read these "/movement" messages and calculate the odometry accordingly.
Firstly, we need to subscribe to the mqtt server "/movement" to calculate odometry.

```
void clientConnected() {
  println("client connected");

  client.subscribe("/movement");
}
```

Whenever a message a detected, a mqtt package procedure called "messageReceived()" can be used to generate a response.

From the mqtt node that we have subscribed to, we know that the only messages that we will receive are movement directions. Using this, we can store the payload of the message into a variable and call another procedure that changes the digital movement of the rover, which can be used to calculate its odometry/location.

```
void messageReceived(String topic, byte[] payload) {
  //println("new message: " + topic + " - " + new String(payload));

  inputdata = new String(payload);

  //println(topic);
  //println(inputdata);
}
```

The mover object previously created uses the input data message to change the vector location of the rover. While we have been able to track the movement of the rover, we still need to be able to control it when it is in manual mode. To do this, we first need to be able to change the mode of the rover. When the toggle switch on the GUI was created, it was made to toggle the variable "ToggleAuto". This variable can then be used within the "draw" loop: when the variable is changed, a "setauto" or "setmanual" message can be passed to the "/GPIOout" node, the only node that the rover is subscribed to.

```
void draw() {

  background(img);

  // Call functions on Mover object.
  mover.update();
  mover.display();

  if(ToggleAuto) {
    client.publish("/GPIOout", "setauto");
  } else if (ToggleAuto == false){
    client.publish("/GPIOout", "setmanual");
  }
}
```

The procedures on the rover can will respond to this message and change the mode of the rover.

After having changed the mode, we want to be able to use the keyboard keys to control the rover manually:

```
void keyPressed() {
  if (key == 'm') {
    if(toggle) {
      client.publish("/GPIOout", "setauto");
      toggle = false;
    }
    else {
      client.publish("/GPIOout", "setmanual");
      toggle = true;
```

```
    }
  }
  if (key == 'w') {
    client.publish("/GPIOout", "forward -m");
  }
  if (key == 'a') {
    client.publish("/GPIOout", "left -m");
  }
  if (key == 's') {
    client.publish("/GPIOout", "back -m");
  }
  if (key == 'd') {
    client.publish("/GPIOout", "right -m");
  }
  if (key == 'x') {
    client.publish("/GPIOout", "stop -m");
  }
}
```

The program also has to communicate with the Arduino powered metal detector. To do this, an I2C bus will be used to send and receive numbers between the Arduino and the Raspberry Pi. The processing 3 library can be used to begin the I2C connection. The numbers received from the Arduino will then be send to the MQTT server so that the user end software can access it for review and analysis.

```
import processing.io.*;
I2C dac;

...

void setup() {
   ...
  dac = new I2C(I2C.list()[0]);
   ...
}

void draw() {
   ...
  /*
  Request and Receive data from Arduino
  */
  dac.beginTransmission(0x8);
  byte[] in = dac.read(1);
  int in_int = in[0];
  client.publish("/metal", str(in_int));
   ...
}
```

This operation works but the numbers received are different to the numbers sent from the Arduino. This results in a Logic Error in the communication between the metal detector and the Raspberry Pi. After exploring the documentation of the processing 3 I2C library, I found a solution.

The Arduino powered metal detector was outputting the numbers in 2 bytes. Therefore, when reading for the signals, we need to read 2 bytes of incoming data. So, I changed this:

```
byte[] in = dac.read(1);
```

to this:

```
byte[] in = dac.read(2);
```

This solved the issue and I was now able to read the incoming numbers from the metal detector.

3.2.2 Module Testing

From the general requirements, this section of the code aims to meet the requirements 1.1, 1.2, 1.4, 2.2, 2.3, 2.4, 3.2. The testing phase will determine how well these requirements have been met.

Num.	Criteria	Outcome
1.1	The user should be able to promptly control the movement mode.	The user can change the movement mode within 3 seconds.
1.2	The user should be able to send instructions to move using keyboard keys.	The user can send all movement instructions using Keys.
1.4	There should be input from the sensors	All sensors send input which can be used.
2.2	The program should only account for obstacles within 0.5m of the sensor.	The program accounts for all obstacles only within 1m.
2.3	The program will ignore data of obstacles not on the rover's path.	Most obstacles not on rover's path are ignored.
2.4	The data from the sensors will be sent to the companion app within 5 seconds.	Data is received within 5 seconds.
3.2	The menu will allow full manual control.	The rover can be switched to full manual control using one control button.

Most of the success criteria relevant to this section has been met very well, apart from 2.3. This is because the rover doesn't successfully ignore all obstacles that are NOT on it's path. This is because it detects object slightly to the side of the sensors which do not pose a collision risk if the rover moves forward as obstacles. This means that the rover feels the need to turn unnecessarily. Next, we can use the relevant functions from the alpha testing plan in the design section to further test this part of the code.

Testing relevant functions from the alpha testing plan:

Function	Method of Input	Input	Expected Result	Pass or Fail
Connecting to rover	Opening the application on the user's computer	Double-click app icon	Program on user's computer displays that the rover is connected	Pass
Changing rover navigation mode	The user will need to toggle a button on and off, which will send a message to server to change the mode	Click Button	The button should change colour to indicate that the rover has changed modes.	Pass
Avoid obstacles autonomously	One of the main features of the rover is to avoid obstacles autonomously, and so it should be able to navigate around obstacles without any assistance.	Use the rover navigation mode button to change the rover's mode to auto	The rover will avoid all the obstacles that it has in it's path and will move in a path without interacting with the obstacles in any physical way.	Pass
Manual navigation	In certain scenarios, the rover will need to be controlled manually by the user, and so the manual navigation system should be working as required.	The user will change the rover navigation mode to "manual" and then use the arrow keys to control the rover movement.	When the arrow keys are pressed, the rover responds and moves in the related direction within seconds, allowing for real-time control.	Pass

3.2.3 User Feedback

The current program was given back to the end users for feedback.

Positive feedback:

- The communication using the MQTT server is pretty fast and reactive.
- The odometry works pretty well and it's very simple to read the current location.
- Toggling between auto and manual mode is very fast and responsive
- The manual mode works very well.

Negative Feedback:

- In manual mode, the rover doesn't stop when the keys are released. It can be stopped using the "x" key but it would be more intuitive if the rover stops once the control keys have been released.
- When in auto mode, the sensors seem to view objects not in the rover's path as obstacles and attempt to avoid objects that do not pose a threat. This is not a huge problem but it does result in the rover being too cautious for demonstrations.

Using the feedback from the end user and results from the testing, I concluded that a second prototype needs to be made with the following improvements:

1. The object detection should be adjusted to ignore objects that do not pose a threat to the rover. The best way to do this is to create a region on the edge of the Kinect 360 Image which would be ignored when determining if reacting to an obstacle is required. Not completely ignoring this is important as it allows the rover to still be aware of the object that would become an obstacle in the rovers path if the rover turns.
2. Make the manual movement of the rover more user friendly by stopping the rover's movement once the user releases the WASD keys.

This prototype of the module can be said to have a Logic Error as it executes and avoids obstacles, but not the way we intended.

3.2.4 Prototype 2

Firstly, to address improvement 1:

After the presence of obstacles on both sides of the rover has been detected by the procedures "checkredleft()" and "checkredright()", we need to create a new procedure that makes the final decision determining the response of the rover. We can name this "checkred()" and add it to the code by changing this:

```
void checkredleft() {
  isred_left = false;
  for(int a=0; a < (depthImg.width / 2); a++) {
    for(int b=0; b < depthImg.height; b++) {
      if(depthImg.get(a,b) == color(252,3,3)) {
        isred_left = true;
      }
    }
  }
  GPIOoutput();
}
```

to this:

```
void checkredleft() {
  isred_left = false;
  for(int a=0; a < (depthImg.width / 2); a++) {
    for(int b=0; b < depthImg.height; b++) {
      if(depthImg.get(a,b) == color(252,3,3)) {
        isred_left = true;
      }
    }
  }
  checkred(); //final decision
}

void checkred() {
   ...
  GPIOoutput();
}
```

Then we create the "checkred()" procedure which will only perform turns if the obstacle is in the middle half of the screen.

```
void checkred() {
  isred = false;

  //"for" statement underneath --> checks if red is middle half by removing first 1/4
      and last 1/4

  for(int a= (depthImg.width / 4); a < (3 * (depthImg.width / 4)); a++) {
    for(int b=0; b < depthImg.height; b++) {
      if(depthImg.get(a,b) == color(252,3,3)) {
         isred = true;
      }
```

```
    }
  }
  if(isred) {
    //print("/");
      if (isred_right == true && isred_left == false) {if(movement != "left") {movement =
          "left"; client.publish("/GPIOout", movement);}}
      else if (isred_right == false && isred_left == true) {if(movement != "right")
          {movement = "right"; client.publish("/GPIOout", movement);}}
      else if (isred_right == true && isred_left == true) {if(movement != "right")
          {movement = "right"; client.publish("/GPIOout", movement);}}
      else if (isred_right == false && isred_left == false) {if(movement != "forward")
          {movement = "forward"; client.publish("/GPIOout", movement);}}
      else {movement = "right"; client.publish("/GPIOout", movement+"-F");}
  }
  else {
    //print(".");
    if(movement != "forward") {
      movement = "forward";
      client.publish("/GPIOout",movement);
    }
  }
  GPIOoutput();
}
```

Using the above code, I have successfully modified the navigation module to ignore objects that are on the edge of the camera.

Now to address improvement 2:

We will add a function that is called automatically upon release of a keyboard key, which sends the stop command to the rover once the WASD keys are released. The following code will be added to the user program:

```
void keyReleased() {
  if (key == 'w') {
    client.publish("/GPIOout", "stop -m");
  }
  if (key == 'a') {
    client.publish("/GPIOout", "stop -m");
  }
  if (key == 's') {
    client.publish("/GPIOout", "stop -m");
  }
  if (key == 'd') {
    client.publish("/GPIOout", "stop -m");
  }
}
```

3.2.5 Module Testing

From the general requirements, this section of the code aims to meet the requirements 1.1, 1.2, 1.4, 2.2, 2.3, 2.4, 3.2. The testing phase will determine how well these requirements have been met.

The last prototype of this module has already been tested once, but this iteration will also need to be tested against all the requirements to make sure that while improvement have been made, the program has not been altered to not meet a general requirement.

Num.	Criteria	Outcome
1.1	The user should be able to promptly control the movement mode.	The user can change the movement mode within 3 seconds.
1.2	The user should be able to send instructions to move using keyboard keys.	The user can send all movement instructions using Keys.
1.4	There should be input from the sensors	All sensors send input which can be used.
2.2	The program should only account for obstacles within 0.5m of the sensor.	The program accounts for all obstacles only within 1m.
2.3	The program will ignore data of obstacles not on the rover's path.	All obstacles not on rover's path are ignored.
2.4	The data from the sensors will be sent to the companion app within 5 seconds.	Data is received within 5 seconds.
3.2	The menu will allow full manual control.	The rover can be switched to full manual control using one control button.

All of the general requirements have been met well, therefore this code passes the general requirements test.
Next, we can use the relevant functions from the alpha testing plan in the design section to further test this part of the code. We have done this with the last prototype but due to the changes, it is best to test it again.

Testing relevant functions from the alpha testing plan:

Function	Method of Input	Input	Expected Result	Pass or Fail
Connecting to rover	Opening the application on the user's computer	Double-click app icon	Program on user's computer displays that the rover is connected	Pass
Changing rover navigation mode	The user will need to toggle a button on and off, which will send a message to server to change the mode	Click Button	The button should change colour to indicate that the rover has changed modes.	Pass
Avoid obstacles autonomously	One of the main features of the rover is to avoid obstacles autonomously, and so it should be able to navigate around obstacles without any assistance.	Use the rover navigation mode button to change the rover's mode to auto	The rover will avoid all the obstacles that it has in it's path and will move in a path without interacting with the obstacles in any physical way.	Pass
Manual navigation	In certain scenarios, the rover will need to be controlled manually by the user, and so the manual navigation system should be working as required.	The user will change the rover navigation mode to "manual" and then use the arrow keys to control the rover movement.	When the arrow keys are pressed, the rover responds and moves in the related direction within seconds, allowing for real-time control.	Pass

3.2.6 User Feedback

The users are happy with this module as it passes all tests and user requirements.

3.3 Settings

3.3.1 Prototype 1

This purpose of this module is to allow the use to adjust some parameters to alter the program slightly, aiding in accurate movement and obstacle detection. The versatility of the program can be increased by utilising this program as that means similar hardware can be used.

To make the drop-down menu, we can utilise the function that is built into the ContolP5 library that we have already imported. To do this, we will first initialise the object type at the start of the code:

```
DropdownList p1;
```

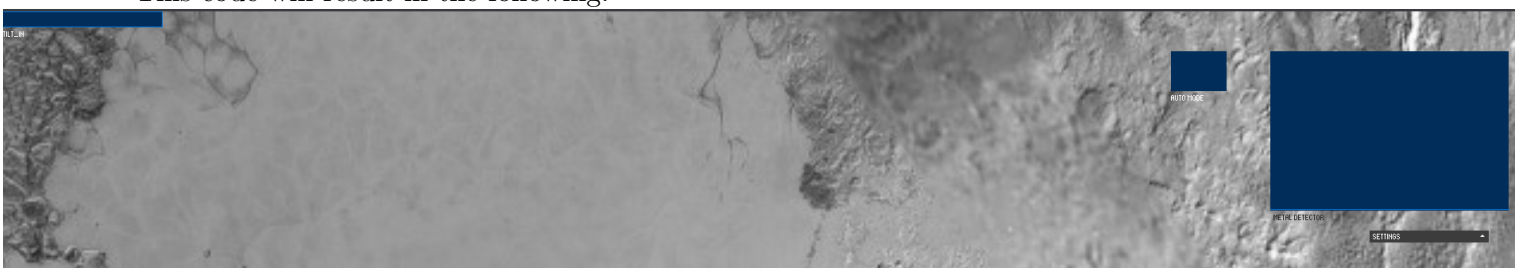
Then, we will create the drop-down menu within the "setup()" procedure, and call a procedure "customize(p1)" which we can use to change the look and function of the menu as required.

```
  p1 = cp5.addDropdownList("Settings",width-200,275,150,240);
  customize(p1);
```

We will then create the customizing procedure:

```
void customize(DropdownList ddl) {
  ddl.setBackgroundColor(color(190));
  ddl.setItemHeight(20);
  ddl.setBarHeight(15);
  ddl.addItem("Tilt", box1);
  ddl.addItem("Detection Region Padding",1);
  ddl.setColorBackground(color(60));
  ddl.setColorActive(color(255,128));
  ddl.close();
}
```

This code will result in the following:



As you can see, the textbox is created as a completely different entity to the drop-down menu. I tried to find a solution to this but it was not possible. Instinctively, I tried to create the textbox inside the "additem()".

```
void customize(DropdownList ddl) {
  ddl.setBackgroundColor(color(190));
  ddl.setItemHeight(20);
  ddl.setBarHeight(15);
  ddl.addItem("Tilt", cp5.addTextfield("tilt_in"));
  ddl.addItem("Detection Region Padding",1);
```

```
  ddl.setColorBackground(color(60));
  ddl.setColorActive(color(255,128));
  ddl.close();
}
```

Unfortunately, this didn't change anything and the textbox was still not within the drop-down menu. I also tried to add the "addtextfield(...)" in place of the string "Tilt", but that caused the runtime error:

After attempting to use the inbuilt drop-down menu function and encountering the runtime error, I decided to change the approach to this module. We can make the drop-down menu into a container with the required inputs and a close button which, when clicked, hides the inputs and displayed text, therefore creating the illusion of a drop-down menu.

To do this, we will first add the text-input and labels in the correct position. This can be done in the "setup" procedure by creating text fields using the CP5 library. By default, the labels are added below the text input box, but we can manipulate these after creating the text field object so hat we can move the labels to the left and align them as desired.

At the start of the code, we will first initialise the variable/object types:

```
Textfield t;
Textfield t2;
Textfield t3;
Textfield t4;
Textfield t5;
```

After that, we will add the following code inside the "setup()" procedure.

```
  t = cp5.addTextfield("Tilt: ")
    .setPosition(width - 90, 340)
    .setSize(40, 20)
    .setFont(createFont("arial", 20))
    .setAutoClear(false)
    .setColorValue(0xffffffff)       //White
    .setColorActive(0xff00ff00)      //Green
    .setColorBackground(0xff0a2b61) //Dark Blue
    ;

  t.getCaptionLabel().align(ControlP5.LEFT_OUTSIDE,
      CENTER).getStyle().setPaddingLeft(-17);

  t2 = cp5.addTextfield("Det. Region Padding: ")
    .setPosition(width - 90, 370)
    .setSize(40, 20)
    .setFont(createFont("arial", 16))
    .setAutoClear(false)
    .setColorValue(0xffffffff)       //White
```

```
  .setColorActive(0xff00ff00)      //Green
  .setColorBackground(0xff0a2b61) //Dark Blue
  ;

t2.getCaptionLabel().align(ControlP5.LEFT_OUTSIDE,
    CENTER).getStyle().setPaddingLeft(-17);

t3 = cp5.addTextfield("Depth Threshold: ")
  .setPosition(width - 90, 400)
  .setSize(40, 20)
  .setFont(createFont("arial", 16))
  .setAutoClear(false)
  .setColorValue(0xffffffff)       //White
  .setColorActive(0xff00ff00)      //Green
  .setColorBackground(0xff0a2b61) //Dark Blue
  ;

t3.getCaptionLabel().align(ControlP5.LEFT_OUTSIDE,
    CENTER).getStyle().setPaddingLeft(-17);

t4 = cp5.addTextfield("Default Speed: ")
  .setPosition(width - 90, 430)
  .setSize(40, 20)
  .setFont(createFont("arial", 16))
  .setAutoClear(false)
  .setColorValue(0xffffffff)       //White
  .setColorActive(0xff00ff00)      //Green
  .setColorBackground(0xff0a2b61) //Dark Blue
  ;

t4.getCaptionLabel().align(ControlP5.LEFT_OUTSIDE,
    CENTER).getStyle().setPaddingLeft(-17);

t5 = cp5.addTextfield("Angular Speed: ")
  .setPosition(width - 90, 460)
  .setSize(40, 20)
  .setFont(createFont("arial", 16))
  .setAutoClear(false)
  .setColorValue(0xffffffff)       //White
  .setColorActive(0xff00ff00)      //Green
  .setColorBackground(0xff0a2b61) //Dark Blue
  ;

t5.getCaptionLabel().align(ControlP5.LEFT_OUTSIDE,
    CENTER).getStyle().setPaddingLeft(-17);
```

This gives us the following result:

The next step is to add a box in the back, a button to hide the input fields and a button to apply the inputted numbers to the appropriate variables.

To add a box in the back, we can use a PImage object. This also allows us to hide the objects when we want to "minimize" the dropdown menu. Another box is also made to hold "close" and "apply" buttons.

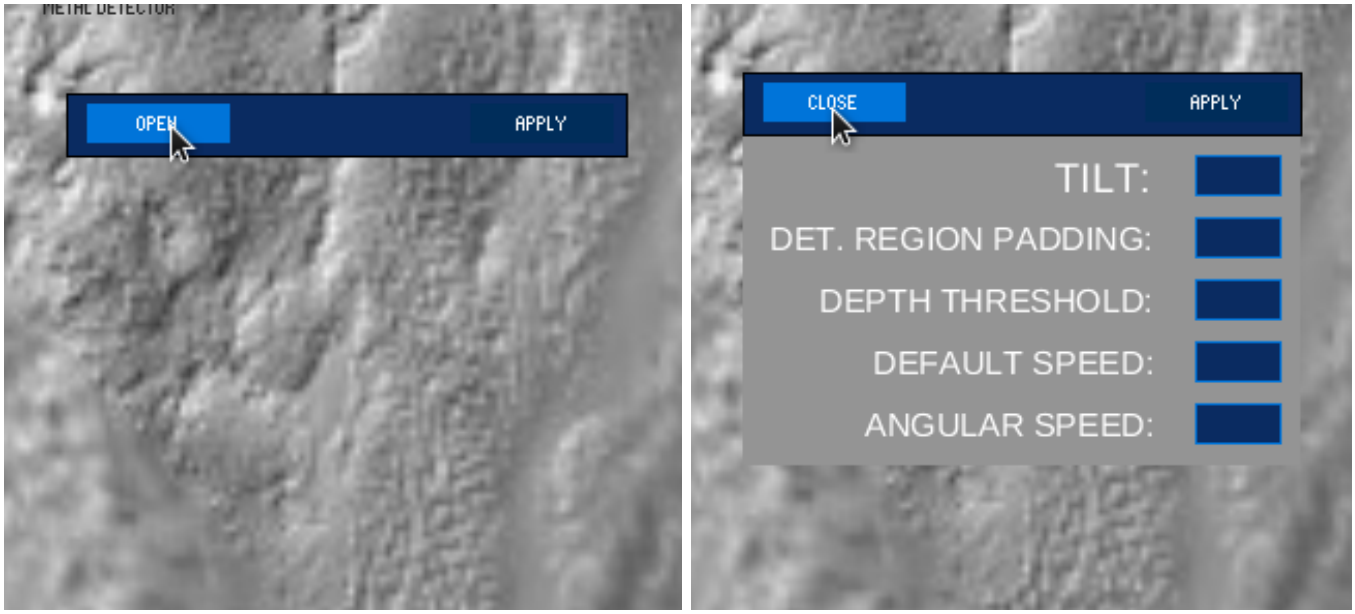
Making the boxes and buttons:

```
Button close;
Button apply;

PShape dropdown;

void setup() {
   ...

  dropdown = createShape(RECT,width - 310,330,270,160);
  dropdown.setFill(color(148, 148, 148));

  close = cp5.addButton("Close")
    .setPosition(width - 300,305);

  apply = cp5.addButton("Apply")
    .setPosition(width - 115,305);

   ...
}
```

Now to add functionality to the "close" button:

```
public void Close() {
  if (close.getLabel() == "Close") {
    close.setLabel("Open");
    hidemenu();
  }
  else if (close.getLabel() == "Open") {
    close.setLabel("Close");
    openmenu();
  }
}

void hidemenu() {
  t.hide();
  t2.hide();
  t3.hide();
  t4.hide();
  t5.hide();
  dropdown.setVisible(false);
}

void openmenu() {
  t.show();
  t2.show();
  t3.show();
  t4.show();
  t5.show();
  dropdown.setVisible(true);
}
```

This allows us to have the following:



Now we need the "Apply" button to set the values of the required variables to the input numbers. The speed and angular speed inputs are to change local variables, and therefore they can be changed on the user's computer, but the other inputs need to be sent to the server/rover to make the changes.

```
void Apply() {
  if (t.getText() != "") {
    client.publish("/GPIOout", "tilt="+t.getText());
  }
  if (t2.getText() != "") {
    client.publish("/GPIOout", "padding="+t.getText());
  }
  if (t3.getText() != "") {
    client.publish("/GPIOout", "depth_thres="+t.getText());
  }
  if (t4.getText() != "") {
    speed = float(t4.getText());
  }
  if (t5.getText() != "") {
    angular_speed = float(t5.getText());
  }
}
```

On the rover end, the following procedure (executed when a message is received) is changed to accept and take action on the new commands:

```
void messageReceived(String topic, byte[] payload) {
  message = new String(payload);
  //println("new message: " + topic + " - " + message);
  if(message.equals("setmanual")) {
    settomanual();
  }
  else if(message.equals("setauto")) {
    settoauto();
  }
  if(message.equals("forward -m")) {
    movement_manual = "forward";
  }
  else if (message.equals("right -m")) {
    movement_manual = "right";
  }
  else if (message.equals("left -m")) {
    movement_manual = "left";
  }
  else if (message.equals("back -m")) {
    movement_manual = "back";
  }
  else if (message.equals("stop -m")) {
```

```
    movement_manual = "stop";
  }
  else if (message.contains("tilt")) {
    angle = round(float(message.replace("tilt=", "")));
  }
  else if (message.contains("padding")) {
    depth_image_padding = round(float(message.replace("padding=", "")));
  }
  else if (message.contains("depth_thres")) {
    maxDepth = round(float(message.replace("depth_thres", "")));
  }
}
```

We can now use the "apply" button to make the required changes that we need to the function of the program.

3.3.2 Module Testing

The relevant requirements to this module from the general requirements are 1.3 and 3.1, therefore this prototype of the module will be tested against these requirements.

Num.	Criteria	Outcome
1.3	The user will be able to use the menu to hide control buttons and statistics.	Some buttons can be hidden with a menu.
3.1	The user interface should be clean and simple.	Some control buttons are visible by default but most are hidden.

Both the requirements are met to an acceptable standard, but to complete them to the desired standard, we need to be able to hide all the buttons and settings on the GUI. As both requirements are amber and the change required is small, I will make the change before going back to the end user to get feedback on this module.

3.3.3 Prototype 2

The goal in this iteration is to add a button on the top left corner that will allow all elements of the GUI (buttons, graphs and input fields) to be hidden. To do this, I will use the code written for the previous "close" button, but change it so that the other elements also hide and show when needed.

Declaring object at the start of the code:

```
Button hideall;
```

Creating an instance of the object within the "setup()" procedure:

```
  hideall = cp5.addButton("CLOSE")
    .setPosition(width - 115, 25);
```

Creating the procedure to hide all elements when button is pressed:

```
void CLOSE() {
  if (hideall.getLabel() == "CLOSE") {
    t.hide();
    t2.hide();
    t3.hide();
    t4.hide();
    t5.hide();
    dropdown.setVisible(false);
    myChart.hide();
    toggle_auto.hide();
    close.hide();
    apply.hide();
    hideall.setLabel("SHOW");
    dropdown2.setVisible(false);
  }
  else if (hideall.getLabel() == "SHOW") {
    t.show();
    t2.show();
    t3.show();
    t4.show();
    t5.show();
    dropdown.setVisible(true);
    myChart.show();
    toggle_auto.show();
    close.show();
    apply.show();
    hideall.setLabel("CLOSE");
    dropdown2.setVisible(true);
  }
}
```

3.3.4 Module Testing

The same requirements will be used to test this prototype as were used for the previous iteration of this module. The general requirements that were tested were: 1.3 and 3.1, both resulting in amber ("acceptable") performance.

Num.	Criteria	Outcome
1.3	The user will be able to use the menu to hide control buttons and statistics.	The control buttons will be completely hidden within a menu.
3.1	The user interface should be clean and simple.	There is minimal information displayed and no control buttons are visible by default.

This module has met all the relevant general requirements. To further test this module, we will use the relevant part of the alpha testing plan:

Function	Method of Input	Input	Expected Result	Pass or Fail
Change settings	The user would change the settings by inputting new numbers into the boxes. This will then change the speed of the rover on the GUI and camera sensitivity.	Increase all values in the settings.	The rover moves faster on the GUI and the it avoids obstacles much sooner than before as they are detected from further away.	Pass

The module has also passed the alpha testing, therefore it is ready for client feedback.

3.3.5 Client Feedback

The end users tested this module and gave feedback. As the module met all the general requirements and passed all the tests perfectly, they were very happy with the end result of this module, giving only positive feedback. They gave the following feedback:

1. The ability to hide all elements of the menu is super helpful as it cleans up the screen and prevents any accidental clicks/changes.
2. Input boxes combined with the "apply" button make it super easy and quick to adjust parameters, which is very helpful for presentations and hardware testing, especially when the audience are coding amateurs.

After developing all the above modules successfully, I conclude the development section of the code. All the modules have been coded, tested and reviewed by end users.