**RobotX Maritime Competition**

**Buoy Detector And Navigation System**

**Vision System**

The vision system is tasked with identifying the buoys on the water, adjusting position to align with a gate and seeking out and aligning with the black buoy. The vision system detects specific objects based on colour thresholds (HUE, SATURATION and VALUE) and area thresholds (MINIMUM and MAXIMUM area) using opencv - a computer vision framework for python.

**Robotx\_buoy\_detector.py**

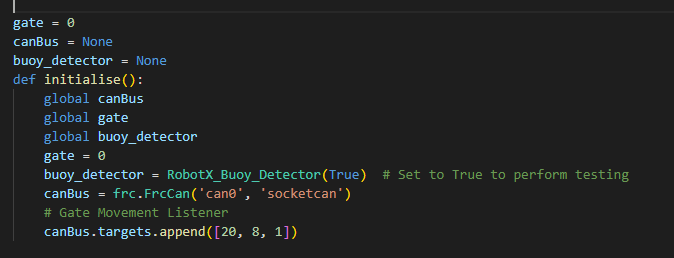
This class is a prebuilt profile that already has specific thresholds set ready for the entry and exit gates task of the robotx maritime competition. The thresholds are based on configurations set during testing on images and videos of buoys. However, values may require adjustment depending on different circumstances.

***Testing And Making Adjustments For Object Detection***

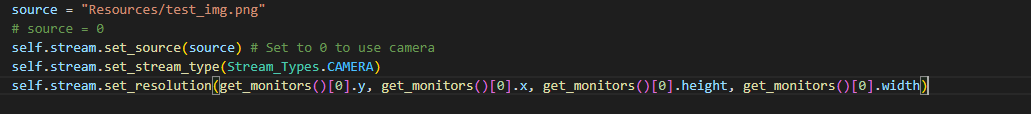
Considering that opencv is not a machine learning solution, further testing may be required to adjust values where necessary until the desired objects can be detected.

This can be done by doing the following:

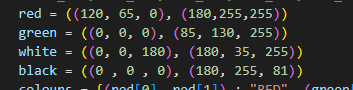
Set the **enable\_testing** value to true - the object should already be set in **Task2.py** in the initialise() function,



Change the **source** in **robotx\_buoy\_detecto**r to a file path (or 0 to test on the first available camera) and set the stream type to the appropriate stream type (for example, either Stream\_Types.CAMERA, Stream\_Types.IMAGE, Stream\_Types.VIDEO)**.**



Once these parameters are set and the stream starts. Trackbars will load allowing area and colour values to be adjusted where necessary until the correct object is detected.

The HUE, SATURATION and VALUE values can be changed in the colour variables in **robotx\_buoy\_detector.py** like so:

Each colour structure can be represented as follows:

(MINIMUM HUE, MINIMUM SATURATION, MINIMUM VALUE), (MAXIMUM HUE, MAXIMUM SATURATION, MAXIMUM VALUE))

In the example, red has a minimum hue value of 120, a minimum saturation value of 65 and a minimum *value* value of 0

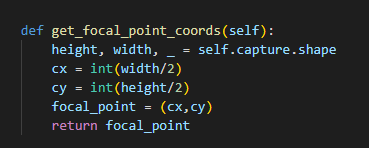
Red also has a maximum hue value of 180, a maximum saturation value of 255 and a maximum *value* value of 255.

This means that all red objects that are desired to be detected must fall in between these minimum and maximum colour thresholds.

**Stream\_settings.py**

This class is just a settings class for Stream, it provides functionality to set the resolution of the stream as well as the capture and source, if for example, the user required to use a capture from another stream, this class could be useful. This class does not need to be initialized, initialize Stream instead and use Stream Settings functions where necessary.

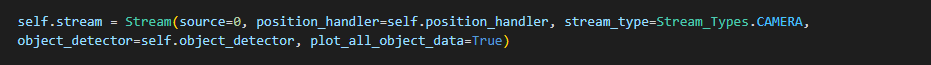
This class also has a function to get the coordinates of the focal point (**get\_focal\_point\_coords()**). This allows the coordinates of the focal point (that is - the center point of the stream capture) to be known. When using a camera, this is especially useful as it allows the program to “know” where exactly the camera is “looking”.



**Stream.py**

This class performs the actual stream which can work for video, image and live camera. It also includes multiple options to customize what the stream displays to the screen, these can be triggered on initialization of the Stream object in **robotx\_buoy\_detector.py,** simply set whatever is desired to **true**.

The default in **robotx\_buoy\_detector.py** is set to show all object data. **Note:** Turning on and off these values does not affect what is actually detected.



The stream can take in an optional object detector and or position handler, however, if an object detector is not provided, no objects will be detected, likewise if no position handler is provided, no position holding or object avoidance will be provided.

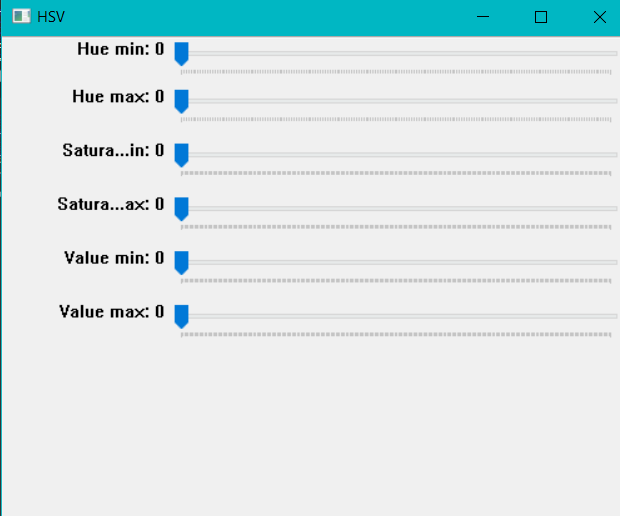
**Object Handler.py**

Due to opencv not being a machine learning solution, the built in object tracker was removed from the system because of uncontrollable CPU intensity with autonomous object tracking, therefore an object handler was implemented instead which is tasked with handling a list of objects detected per stream capture. When an object is detected, it adds the Object to the list of objects detected during the current period in time, it then immediately initializes a new Object, ready for a new Object to be detected.

Objects carry data such as coordinates, colour labels, corner data, distance from a focal point, area and boundary area.

**Object\_Detector.py**

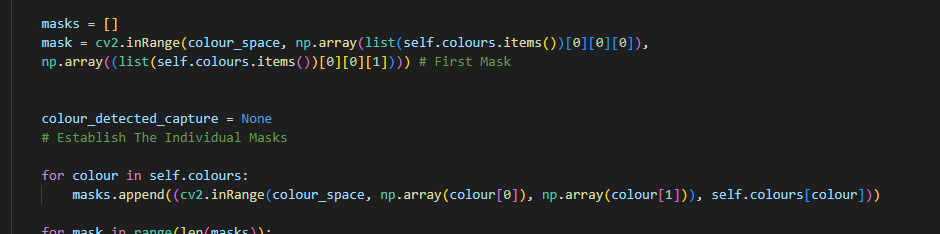
This class does the actual object detection, it has trackbars to be able to perform manual testing of thresholds (such as colour), trackbars will only be active if **use\_trackbars** is active when the Object Detector is first initialized in RobotX\_buoy\_detector.



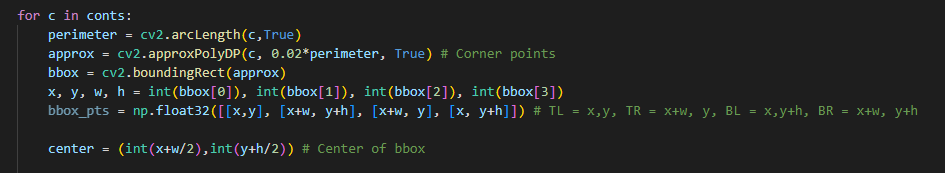
*Example of trackbars*

Object detection is as follows:

Masks are established in HSV colour format (Hue, saturation and value) for each colour threshold given, a mask represents a modified version of the stream where only objects in the specific colour ranges are identified.



The detected objects’ perimeter and approximate corner points are calculated to then allow an appropriate box to be drawn over the object.

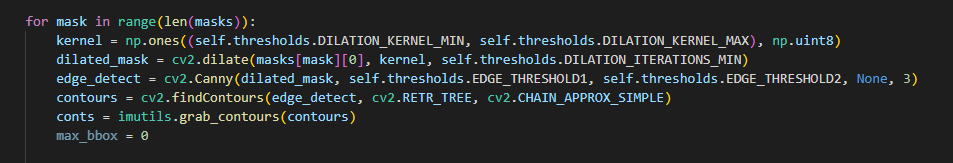


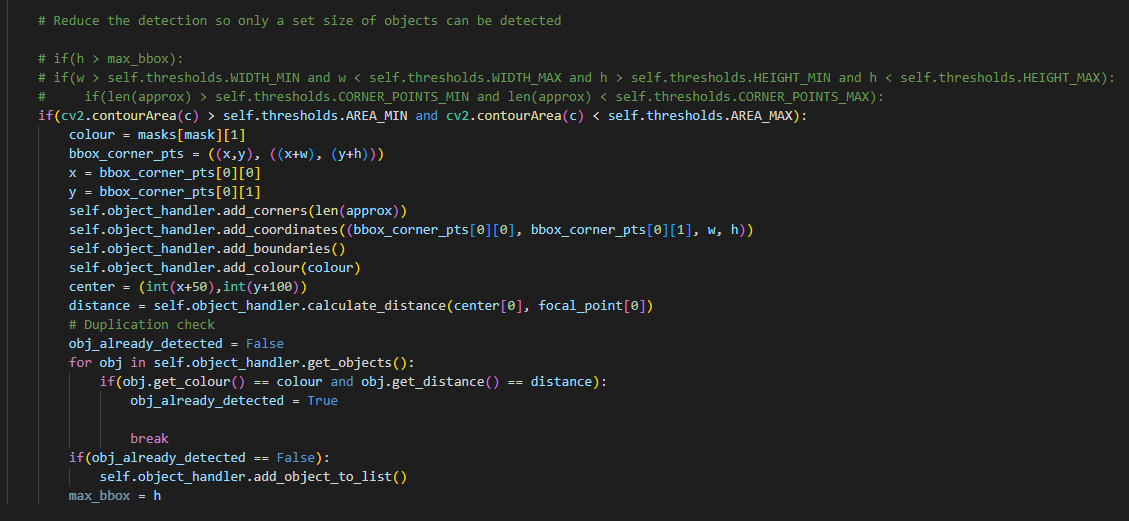
If required, the bbox\_pts are also provided, this represents in order:

The top left of the box, top right of the box, bottom left of the box and bottom right of the box - this is useful, if the user desires to know the exact coordinates of each corner of the bounding box of the object.

To reduce the false positive rate, the detected objects must then meet specific minimum and maximum area thresholds to be assumed to be buoys. Further criteria could be set to reduce the false positive rate (such as reducing detected objects to only have a certain corner point range), however, this may enforce harsher restrictions that could prevent correct buoys from being detected and therefore, these have been deliberately commented out.

The masks are then dilated and sent through a Canny edge detector where associated contours (or edges) of each object in the mask are detected.



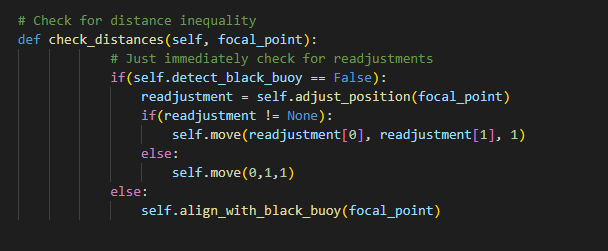


**Position\_handler.py**

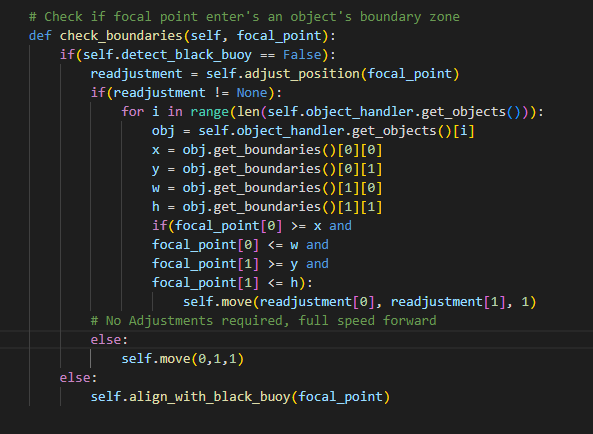
This is the main position holding class as well as the class to align with the black buoy

This is also critical for the boat to navigate through the course.

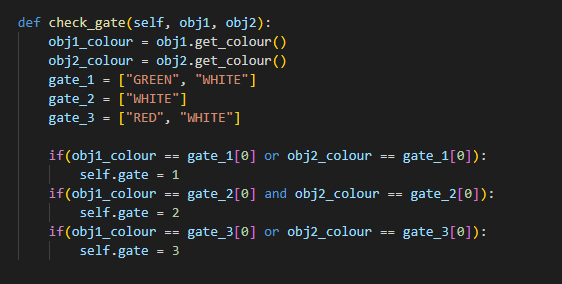
It first determines whether the boat should adjust position using distance inequality calculations - that is, it checks that both sides of the gate (representing the two closest objects) have the same distance from the focal point of the camera, if they don’t, the boat moves until centered.



It also checks that the focal point is not looking directly at a buoy, this would assume that the boat is too far to one side of the gate and the system adjusts the position accordingly. This is done in the following function:

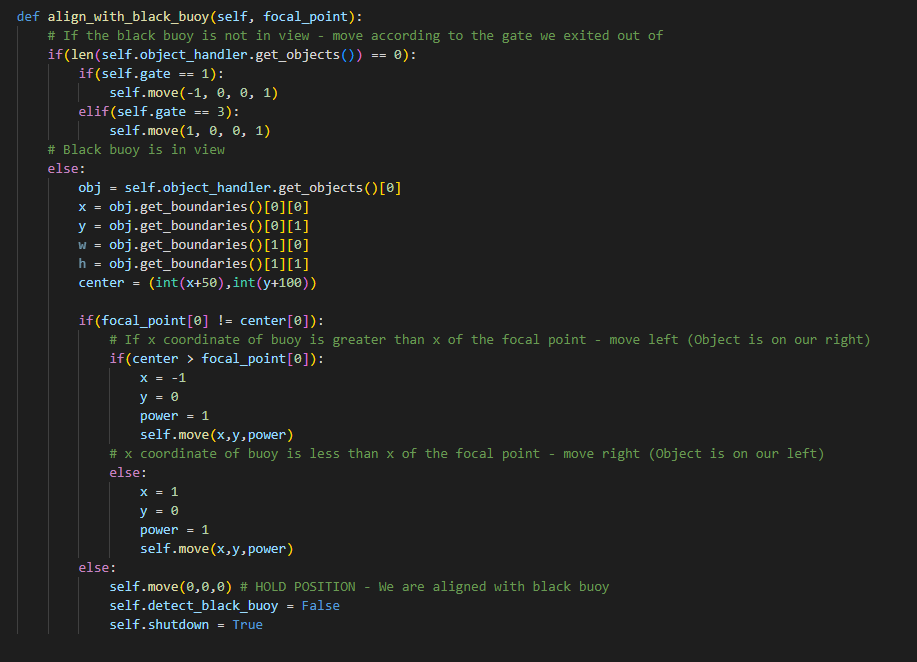


Next the class checks what gate has been entered, according to the colour sequence of the buoys,



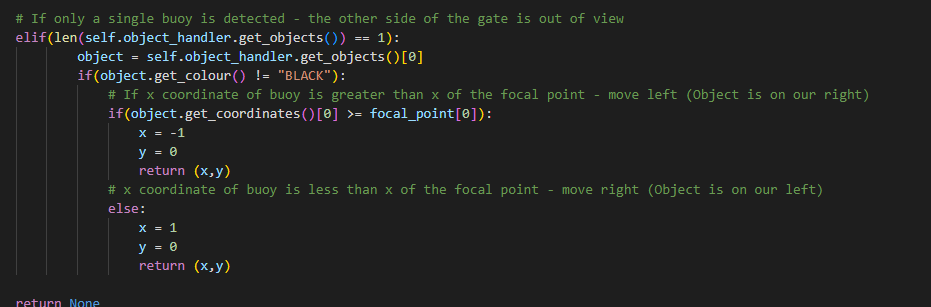
This is useful information to know for a variety of reasons, for one it allows the boat to know which gate to exit through and which way to move when coming back through the course. And additionally, should the black buoy be out of view, using the gate sequence, the boat can adjust its position accordingly until the black buoy is in view, without going off course.

As stated, the class also handles aligning with the black buoy,, if the black buoy is not in view, it determines which way to move according to the gate that was entered. If it is in view however, it moves until the focal point is in line with the center of the black buoy, where it then shuts down and hands over navigation to the Navigation system to then loop around it.



This class also handles the circumstance where only 1 side of the gate is in view, in this case, the system will determine where to move based on where the buoy is relative to the focal point. It assumes that if the buoy is to the left of the focal point, then the other side of the gate must be on the right and vice versa.

Note: This is at the end of the **adjust\_position** function.



**Navigation System**

# GPS\_Module.py

This script gets the serial output from the USB GPS module ('/dev/ttyACM0') and converts it into decimal degrees and then transmits it across the CAN bus. During initialisation, the possibility of corrupted text data in the first few messages exists, but exceptions are there to disregard them.

### Usage

Relatively straightforward, should simply be constantly running on the pi. Log location and captured NMEA sentence type can be adjusted (mostly for capture/debugging purposes).

# Navigation.py

This module contains the necessary movement commands to be used in the execution of Task 2. It listens to the GPS module, the Nav Module (for heading), and itself on the CAN Bus. Tolerance values (how close the program needs to run to its parameters) can be adjusted at the top.

### Noteable variables

**LAT/LON METER**: The GPS decimal equivalent of a meter at this part of the world. True calculation of this value is actually more complicated due to Longitude-to-meter conversion changing based on latitude, but this works in a pinch. A more advanced method using the Haversine formula is present in the get\_distance\_between() function. This has been calibrated to a fairly high degree of accuracy to the competition gps location.

**Rotational/Distance Tolerance**: Since there is little chance for a boat to accurately stop on a dime, these values are used to give the formulas some tolernace +/- in calculations. It's hard to know exactly what's an acceptable level of tolerance for the boat to not be infinitely adjusting its position, while still remaining fairly accurate. If increased, the boat movement calculations will be less precise, but might more realistically account for the boat's dimensions/movement. However, if increased too far, will cause the calculations to be wildly inaccurate.

**NORTHBOUND**: This value is the 'relative' north alignment for this task. It should be pointed towards the "end" of the course, in the direction perpendicular to the gates. In task2.py this is set at the start of the enter\_course() function to automatically be the direction the boat is facing when the command to start the task is given. This means that the boat must be properly aligned before commencing the task. This can also be set manually, in task2.py.

This is the heading NORTHBOUND should be set as. Important to note that it may not be heading 0 depending if the course is rotated. In this example, NORTHBOUND should be 90.

### 

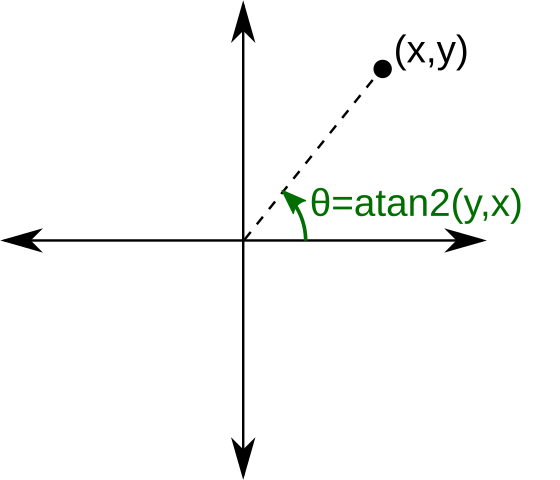
### Notes on functions

**heading\_calculator()**: Used to easily calculation addition and subtraction to heading since heading values are only from 0-360.

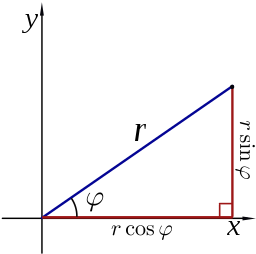
**get\_distance\_between()**: Uses the Haversine formula to calculate the distance between two given GPS coordinates.

**move\_boat()**: This is primarily used by the other movement functions to send a command on the CAN bus to move. Should generally not be used on its own because there is no check for when the movement is complete. So multiple chained move\_boat() commands might end up all executing in rapid succession.

**move\_boat\_to\_coordinate()**: Loops until the boat is at the given coordinates (+/- tolerance values). Uses atan2() to determine the vector angle from the current coordinates to the target coordinates, and then aligns the boat in that direction. The distance between the current point and the target point is then calculated, and a move\_boat() command is sent to the CAN bus with a power\_scaling based on how far away the target point is. This may have to be tweaked depending on how powerful the engine actually is.

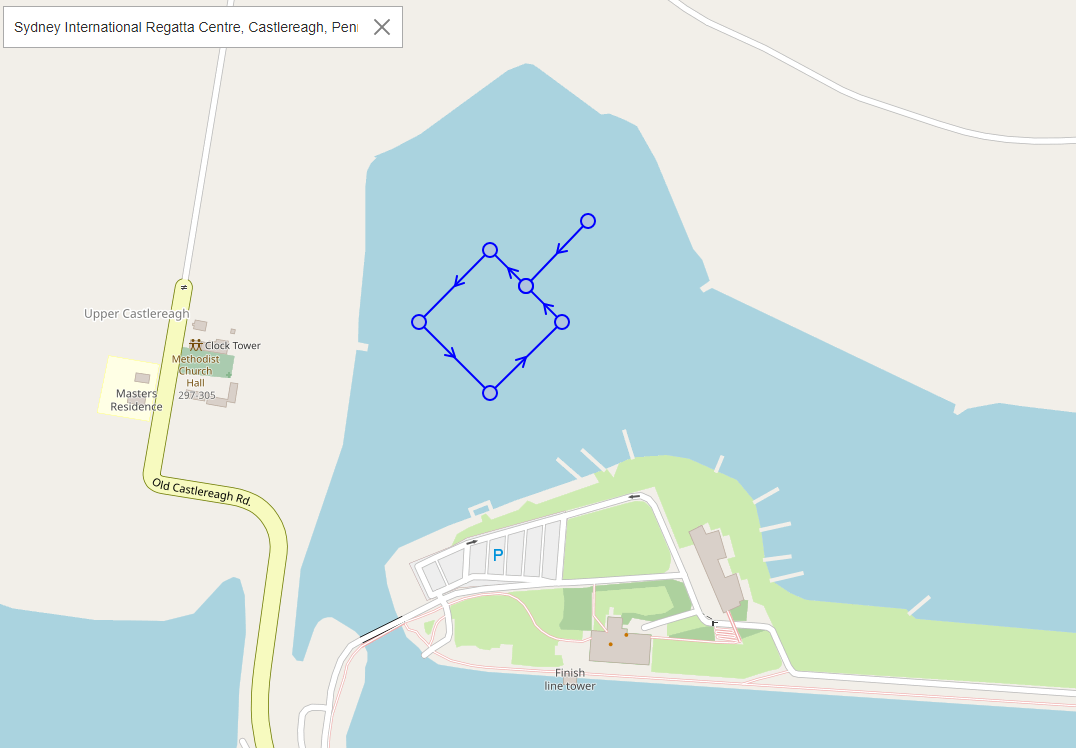


**move\_boat\_by\_distance()**: This function is primarily used to get the boat to turn and move a specific distance left or right. It uses trigonometry to convert between polar and Cartesian coordinates. This allows it to calculate the coordinates of the target point (n) meters to its left or right regardless of its heading.



It then calls the move\_boat\_to\_coordinate() function to get there. This function allows the boat to theoretically navigate the course if all the distances between objects are known, without having to obtain the coordinates manually and moving it from coordinate to coordinate. This allows the boat to calculate the coordinates to do this while only having the coordinates of the start location.

Note: this relies on the NORTHBOUND attribute being relatively accurate.



**align\_heading()**: Used to align the boat to a specific heading before carrying out movement. Turns boat clockwise at power (0.1) until the target heading (within tolerance) is reached. Power may have to be manually adjusted.

**hold\_position()**: Mostly a legacy function, not really used since the concept was largely abandoned during development. It should technically work, but has to be put in an external loop and given the coordinates of the position to hold. You can use Navigation.get\_location() to get the coordinates to hold at before you start the loop.

# Task2.py

Thie script is used to carry out the logic for Task 2. Listens for a signal to begin, and then goes into enter\_course(), since the discussion with client was for the boat to exit the gate immediately after, exit\_course() is called immediately after. The internal logic from the flow chart is preserved in the comments.

Flow Diagram:

### Usage

Relatively straightforward, should simply be constantly running on the pi. NORTHBOUND variable would likely be set in Navigation.NORTHBOUND = here if set manually