Navigation robot documentation

Analysis

1.1 Defining the solution

Problem to solve

Often in factories, warehouses, construction sites, and anywhere where things are made, parts/products/materials always need moving from one place to another. Usually items are heavy, large, or need moving often – all laborious, dangerous, or boring tasks for a human to complete, but they are necessary, so an automated process makes sense because machines do not get tired or bored and are not as liable to 'injury' as a human.

Detailed analysis

The organisations/companies that will use this product will mainly be warehouses, construction sites, and Factories, (however, the main idea of the solution could be applied to many disciplines). Ideally a warehouse or factory as this is a more controlled environment. From these companies, users would be mainly warehouse pickers, builders, and factory workers, but could extend to managers as well. These users all do physical work on a daily basis, lifting heavy objects, moving bricks/materials, and potentially heavy parts. Managers need to ensure all operations are running smoothly, coordinate workflow, and give instructions.

Currently, organisations like these either use human workers to carry parts/materials/objects, or use machinery like forklifts, wheelbarrows and trolleys. These solutions all require a human operator, reducing productivity as a human is now engaged in a labourious task (I.e. they could be doing something more skilled, which would increase work productivity).

This is a good computational problem as these environments are (typically) controlled and well organised. This means virtual coordinate systems and mapping can easily be implemented and set locations saved. This is more reliable than a human, because humans forget, get lost, get tired etc. Computers are good at making accurate measurements from certain data (e.g. the distance sensors), meaning when navigating through a crowded environment, it is fully aware of where the objects are and does not get distracted/confused like a human operator. Computers are also essential for a smooth working environment and computer integration on the factory floor gives more control and data to the managers (though the managers would not be able to control the machines directly (safety hazard if not in your line of sight), only alter final coordinates etc), which is beneficial from an efficiency and productivity point of view. Computers also improve efficiency, and productivity through efficient route planning and optimisation

1.2 User analysis

Stakeholders

There will be many stakeholders for this project as it impacts many people. The workers would be affected as the way they work would change slightly, but for some things, it would be a benefit to have a payload-carrying robot around the site, to carry the heavier objects / perform repeated tasks. The manager's workflow would change as now robots have to be directed as well as humans or would have to be automated and *maintained* (could involve hiring a technician – only feasible for large scale deployment across the environment). This increase in productivity affects many people, as the profits increase due to raised productivity. This could lead to pay rises, more employees (as the company expands), share prices increasing (good for shareholders), and a general increase in the company's success. However, these robots could raise concerns as increased automation can lead to loss of jobs if these people are not redirected to perform other, more complex tasks.

User persona of stakeholders.

Stakeholder	Description	Stakeholder's objective	Barrier to achieving objective	How my solution will help
Factory / warehouse / construction site worker	General labourer in a factory or construction site, or a warehouse picker in a distribution centre	Move objects and materials around site and perform construction /maintenance on machines, buildings etc.	Getting bored, fatigued, and tired etc, being slow, getting injured from carrying heavy loads, etc.	Laborious tasks such as heavy lifting, sensing and surveillance, can be delegated to the robot.
Manager	The manager of the workers	Direct and control people and machines	Workers having to lift things manually which has a	The solution will allow the manager's workers to

	and machines on a site.	to ensure the site performs effectively.	negative Impact on their health – less performance.	transport heavy things across the environment without manual strain.
Owner of company	Owns the company.	Ensure all stakeholders are happy by maximizing profits.	Inefficient workflow and costs are too high.	Increased efficiency in workflow means greater profits.

User stories

Factory Worker	As a Factory worker, I want to be able to carry parts, tools and product to my workspace, so that I can do my job as effectively as possible with minimal time/effort spent.
Warehouse picker	As a Warehouse worker, I want to be able to have products delivered to where they should be quickly, and be able to see what orders need picked as I go around picking them, so that I can do my job efficiently and keep my managers happy.
Construction site worker	As a Construction site worker, I want to be able to move large quantities of materials such as bricks, sand to where they are needed, so that I can start building.
Manager	As a manager, I want the working environment that I manage, to run smoothly and efficiently, so that the service/product we provide is delivered to our customers as quickly as possible.
Owner of the company	As the owner of the company, I want to achieve maximum profits, so that I make the most money.

User requirements:

1) Reliable transportation of goods / products / materials from A to B.
2) Fast and efficient transportation of goods / products / materials from A to B to maximise productivity and therefore product / service.
3) Must be safe both for the worker's safety and the payload's safety
4) Ability to remember set locations in the environment, so that workers can specify, and transport goods / products / materials to where they need to be.
5) Must be cost effective for the company to have these robots.

1.3 Researching other solutions

Alternative solution 1: Human workers / Human operators of machines

Manual operation of machinery is usually used.

Good features:

- Audio warning for when it is reversing “beep beep”.
- Easily manoeuvrable
- Human vision detecting obstacles and people.
- Human remembers best path around the factory

Alternative solution 2: Husky UGV from clearpath robotics:

Particularly used for autonomously performing sensing in outdoor environments, and larger versions are designed for carrying payloads, autonomously.

Good features

- Remote access to robot's sensor data (including internal sensor data for motors, etc), as well as camera and path planning data.
- Modular – different attachments for different tasks such as sensing, carrying payloads, or manipulating objects with robotic arms.
- GPS location for accurate positioning, to allow for better positioning, and map overlays etc, and to account for any 'slack' in real world conditions (e.g. wheels spinning due to uneven ground etc)
- Integrated with ROS (Robot operating system), to allow users to interface easily and on a deep level.

Alternative solution 3: AMRs (Autonomous mobile robots) for logistics.

Used in factory and warehouse environments for logistics (carrying payloads / parts / products)

Good features

- Autonomous navigation with sensors etc.
- Obstacle and people avoidance – Safe.
- Path planning around factory environment – Fastest route from A to B
- Some have managerial control – Feedback to main system / managers / warehouse computer.

- Flexible – Can be integrated into various automated systems, and easily changed to be integrated into a different system.

Research summary:

Features I might use:

- Autonomous navigation with sensors
- Optimal path planning
- Flexibility

Features I will not use:

- Integration with ROS
- Cameras
- Managerial control
- Remote access

1.4 Identify essential features of the solution

Feature	Explanation / justification of including feature
1) Easy interaction between human worker and robot	Productivity in the workspace is important, and for the integration of robots into the environment, the workers need to be able to interact with, and utilise the robots effectively. (Customer requirements 1, 2, and 4)
2) Ability for workers to input coordinates / select set positions in the environment	This will be in the form of a touchscreen on the back of the robot (Touchscreen capability may not be included in prototype) and the worker will select where the robot should go on a map, or input set coordinates. (Customer requirement 4)
3) Obstacle avoidance.	This will be achieved using a lidar sensor, and this creates a depth map of the world in front of the robot, this will ensure it will not crash. (User requirement 3)

4) People / dynamic avoidance	The robot will also be able to react in an appropriate manner to moving objects such as humans, trolleys, and other robots. (User requirement 3)
5) Audio or visual warning of robot's presence	A light or audio "beep" to warn humans of the autonomous robot approaching. (Customer requirement 3)
6) Navigation around the environment	In tandem with the obstacle avoidance, the robot needs to be able to navigate in a safe, efficient reliable manner, for there to be a significant increase in productivity. (User requirements 1 and 2)
7) Path planning using shortest path algorithms.	Use of shortest path algorithms such as Dijkstra's or possibly A*, to minimize time spent delivering payload. (Customer requirements 1, 2, and 4).
8) Current job status and job queue functionality.	The ability for jobs to be added onto a job queue, meaning new commands will only be completed when they reach the top of the queue, as well as displaying which job is currently being executed.
9) Option to return to a home position that can be calibrated.	Return to a set 'home' position which should be ideally located, so that at any time, it can easily be called. This should be able to be calibrated to allow more accurate results.
10) Ability to take manual control of the robot	The robot should be able to be controlled manually e.g. through a Bluetooth controller, so that new set points can be created, home points can be calibrated, and the user can carry heavy loads around the environment arbitrarily. When in manual control mode, the robot should not actively try to avoid obstacles, as the user should always be watching, and any active obstacle avoidance might be a hindrance (robot might be trying to fight user commands otherwise, which would contrast with **essential feature 1**).

1.5 Identify limitations of the solution

Feature	Justification / explanation of not including feature
1) Cameras	The inclusion of a camera feed is not essential for this project because it is not used in navigation. If cameras were used for navigation, this would require complicated computer vision software and is not always reliable –Conflicts with Customer requirement 1. Therefore, navigation will be limited to LIDAR (distance sensing).
2) Managerial control	Control of robots will not be directly given to a manager / boss because this could conflict with a worker's commands to the robot and which leads to confusion, inefficiency and workers not being able to work well (e.g. they expect the robot to do a job, and then a manager, unaware of intentions, commands the robot on a different job, and the current job doesn't get done, therefore, confusion etc) – Therefore it conflicts with Customer requirements 1 and 2 – Therefore control of robots will be limited to workers in the environment.
3) Ability for warehouse pickers to see current orders on screen	Warehouse pickers may find it useful to see current orders needing to be picked on-screen, However, this would include integration into another computer system (the warehouses), and this could cause complications. Plus, it would be easier for the warehouse picker to use a handheld device. Therefore, there will not be order information included on-screen.
4) Self-charging capabilities	This is outside the scope of the project, as this would mean even more hardware and battery indicators etc. Therefore, it will be limited to a 'return to home' function, mention above, (no. 9).

1.6 Specify requirements for the solution

Hardware:

- A *raspberry pi* will be used as the Main control system for the robot – This will be used for the prototype as it's relatively cheap and powerful for its size, and interfacing between other hardware is very easy. (Version used will be Raspberry pi 3b+ as this is easiest to interface with (Raspberry pi 4 has different HDMI ports and power ports), and the power consumption to power ratio is low.)
- *Stepper motors* will be used as the actuation for the wheels of the robot, so that the robot can move around its environment. – Stepper Motors will be used as they can be controlled accurately and to a high precision, which is essential for robot navigation. (Stepper motors used will be NEMA-17 stepper motors (as I have experience with these))
- *Stepper motor drivers* are needed to drive the stepper motors and interface with the raspberry pi. – These are needed as a custom hardware solution using transistors would take up more space, and be messier, and liable to mistakes. (DRV8825 will be used because these are easy to interact with and relatively cheap).
- *Batteries* – 5V power will be needed to power the raspberry pi (power bank will be used for its ease of recharging and high energy density) – 12V power will be needed for the stepper motors (3 cell lithium packs will be used for their easy of charging and high energy density).
- *LIDAR* – A lidar is needed for the distance sensing and ranging to navigate the environment properly (TF mini plus will be used as for its high accuracy and range).
- *A hobby servo motor* will be used to move the LIDAR through a set angle.
- A *small, low-power touchscreen* is needed for users to interact with the robot.

Software requirements

- Latest Raspbian image for the pi.
- Python and python libraries such as math, sys, time etc

1.7 Specify measurable success criteria for the solution

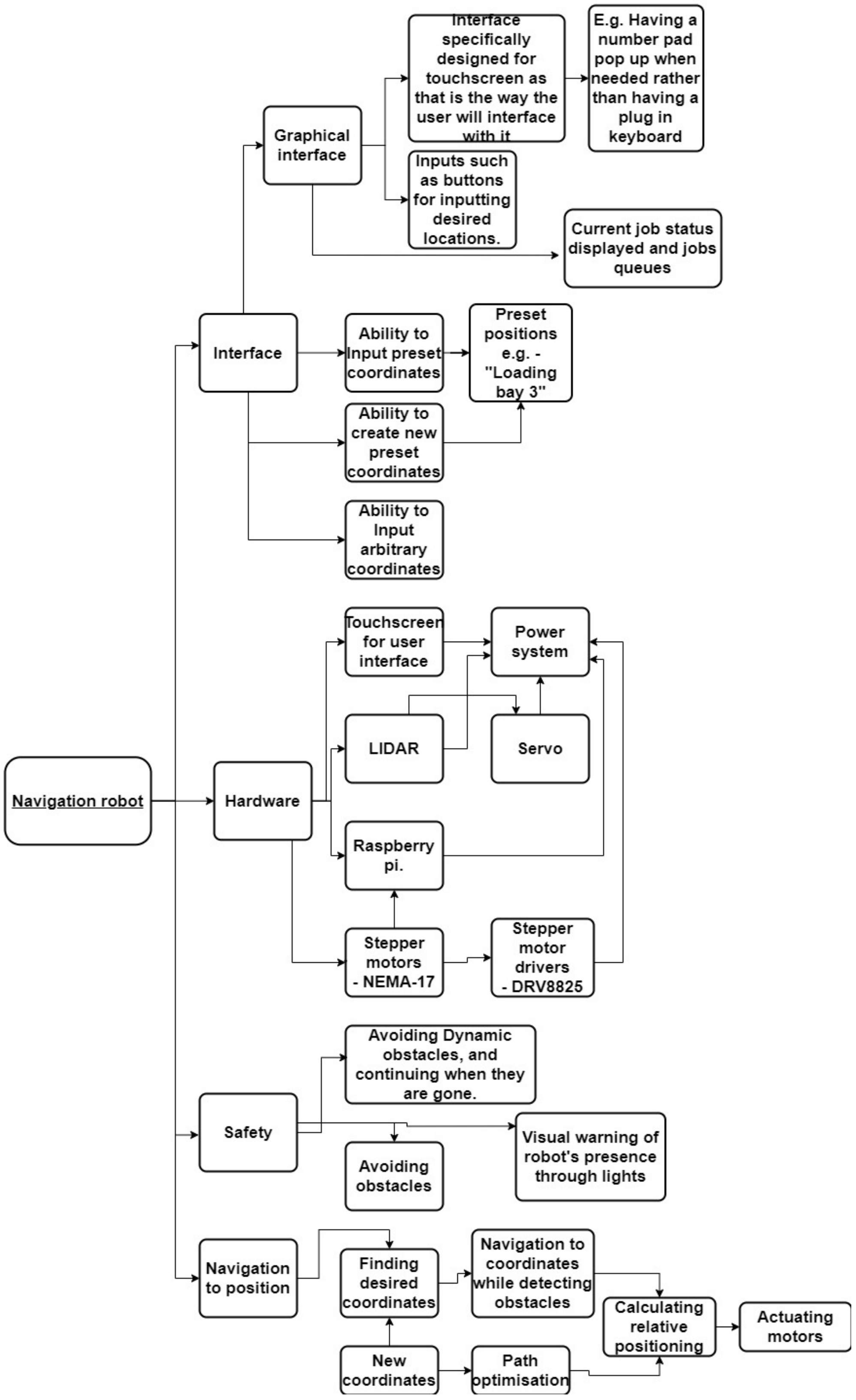
1. Human's should be able to interact with the robot by giving it commands to navigate to a set coordinate. This is specified in user requirements.
    a. Measurable success – All main functions of the UI should be on a single page, and a pop-up system should be used for any other functionality only when needed.
2. – At a bare minimum, the robot must be able to navigate to a set coordinate while avoiding *static* obstacles such as walls.
    a. Measurable success – Should be able to do basic navigation through various courses.

    - However, ideally it would be able to also avoid dynamic obstacles such as people to increase workplace safety etc.

    b. Measurable success – Should stop when presented with a dynamic obstacle, such as a human or moving trolley, and continue its path when the obstacle is gone.
3. The robot should use some form of path optimization, to ensure the shortest route while still not crashing into obstacles.
    a. Measurable success – Should be able to navigate in simple environments to the goal position, while following a shorter path than orthogonal and not crashing into obstacles.

Design

2.1 Decomposition of the problem


Navigation robot
Interface
Graphical interface
Interface specifically designed for touchscreen as that is the way the user will interface with it
E.g. Having a number pad pop up when needed rather than having a plug in keyboard
Inputs such as buttons for inputting desired locations.
Current job status displayed and jobs queues
Ability to Input preset coordinates
Preset positions e.g. - "Loading bay 3"
Ability to create new preset coordinates
Ability to Input arbitrary coordinates
Hardware
Touchscreen for user interface
Power system
LIDAR
Servo
Raspberry pi.
Stepper motors - NEMA-17
Stepper motor drivers - DRV8825
Safety
Avoiding Dynamic obstacles, and continuing when they are gone.
Visual warning of robot's presence through lights
Avoiding obstacles
Navigation to position
Finding desired coordinates
Navigation to coordinates while detecting obstacles
Calculating relative positioning
Actuating motors
New coordinates
Path optimisation

2.2 Define in detail the structure of the solution

Item being included	Justification for including item in the solution
Graphical interface	A graphical interface will be included because to enable easy interaction between human workers and the robot **(Essential feature 1)** and Graphical interfaces are a lot simpler to interact with than command-line based interfaces and other interfaces. • A graphical interface specifically designed for touchscreen will be the main design focus for the interface – this is because other input methods such as keyboard and mouse are not as quick and simple as a touchscreen, for this reason a touchscreen will be used in the prototype and hence, the graphical interface should be designed for touchscreen. - This will include things such as pop-up number pads, and button-style selection.
Various coordinate input methods	It’s not practical for workers to remember the exact coordinates of locations in the environment, so to make interaction between human workers and the robot *easy,* there will be the ability to a) Create new pre-set coordinates e.g. “Loading bay 3”, b) The ability to *select* certain pre-set coordinates at go-to positions, and c) The ability to Manually input certain coordinates to go-to (for calibration / calculating pre-set coordinates). **(Essential features 1 and 2).**
Job status display	It’s important to display the current job status on the screen, so that other workers are aware what job is currently being done, (plus any other jobs in the queue) so that the worker can establish whether he/she should wait to use that robot or decide to find another one. **(Essential feature 8).**
Avoidance of static and dynamic obstacles	For the robot to function in any environment, it needs to be able to *detect* and *avoid* **static and dynamic** obstacles so that a) it doesn’t break anything in the environment or the internals of the robot and b) so that it doesn’t harm or impede the workflow of, human workers. For static obstacles: • Obstacle should be identified as a static obstacle, so that it responds by a) stopping well before the robot would hit the obstacle, b) recalculating the best route based on sensor data, relative position, and goal position, and c) Go along new path. For dynamic obstacles: • Robot should stop, and then carry on after obstacle has gone. • If the obstacle does not move after a certain time frame, the robot should navigate around.

	This will be achieved using a single-point LIDAR sensor, and a servo motor. A mapping of the environment will be achieved based on LIDAR output and angle values (relative to the robot’s direction). (**Essential features 3 and 4)**
Navigation to position	An essential feature for an autonomous robot. A clear, well-defined coordinate system should be set up, and the robot should then be able to navigate around by constantly calculating it’s position relative to a home position (0,0) which can be manually set for calibration. This should be done using path-planning / shortest path algorithms, to maximize efficiency. (**Essential features 6 and 7**)

2.3 Algorithms

First, a word of explanation: once I began researching popular algorithms for this kind of application, the most suited was a technique called SLAM (Simultaneous localisation and mapping), where localisation (Knowing where you are in the environment) and mapping (mapping the environment) are done simultaneously. This is a very complicated approach, with very complex algorithms.

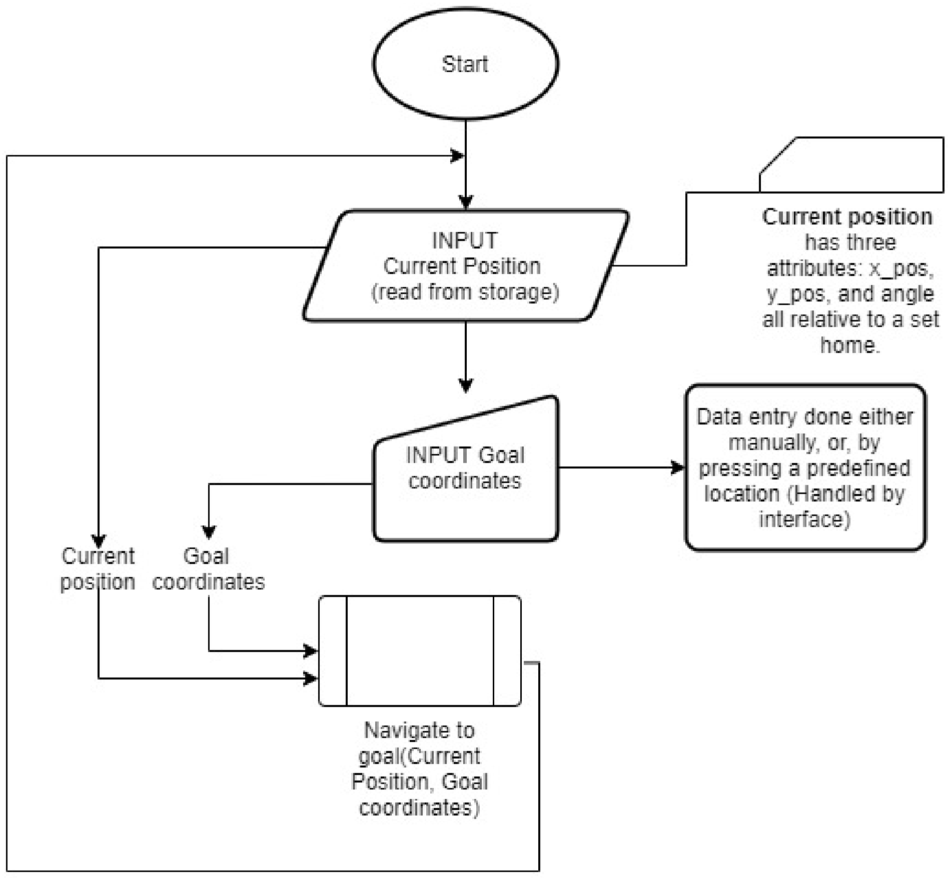
These techniques are degree level techniques and are not in the scope of this project. As an alternative, I will be using a custom algorithm which gathers concepts from both SLAM and A* pathfinding algorithms, to achieve a good **alternative**. Note: this will mean that the kind of environments the robot will navigate can't be too complex (e.g. a maze), which is okay, because the environments that the robot will be navigating are usually well set out, with simple layouts.

The final algorithm will be a mix of functional and object-oriented programming, with things like 'position' being an object with attributes and methods etc, and other functions being performed by subroutines.

The logic of the program is planned to be as follows:

Root level logic

This is the logic at the root level, or what would be in the main() function.

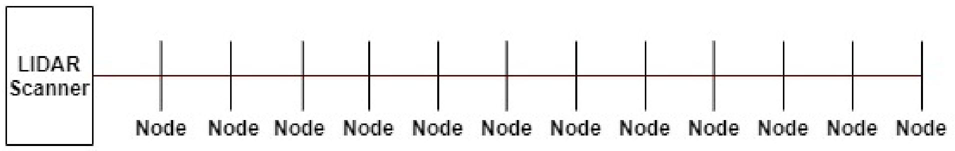


Start
INPUT
Current Position
(read from storage)
Current position has three attributes: x_pos, y_pos, and angle all relative to a set home.
INPUT Goal coordinates
Data entry done either manually, or, by pressing a predefined location (Handled by interface)
Current position
Goal coordinates
Navigate to goal(Current Position, Goal coordinates)

SUB Navigate to goal (Current Position, Goal coordinates)

Note: 'Current Position' is a 3-attribute object, with x coordinate, y coordinate and angle (relative to home) as attributes.

This is the main function for navigating to a Goal by picking sub-goals that are have the lowest **cost** value (using A* path planning techniques (f(x) = h(x) + g(x))). The 'Nodes' are generated from incrementing by a set "resolution" along the line of the LIDAR scan, as shown below:

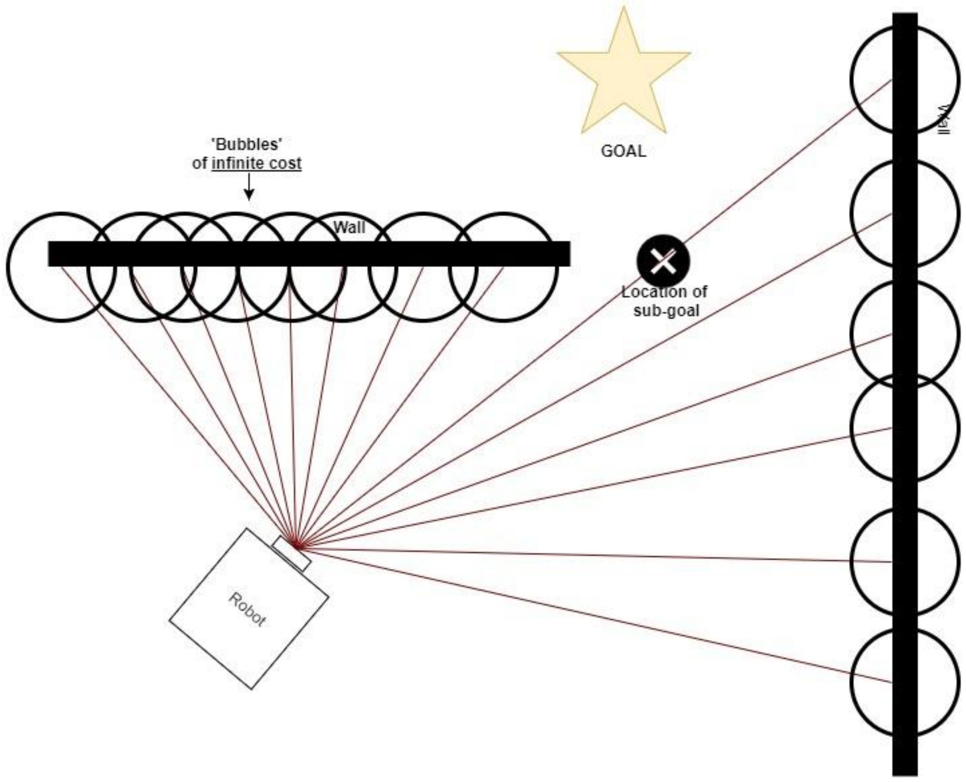


The subroutine (navigate to goal) goes as follows:

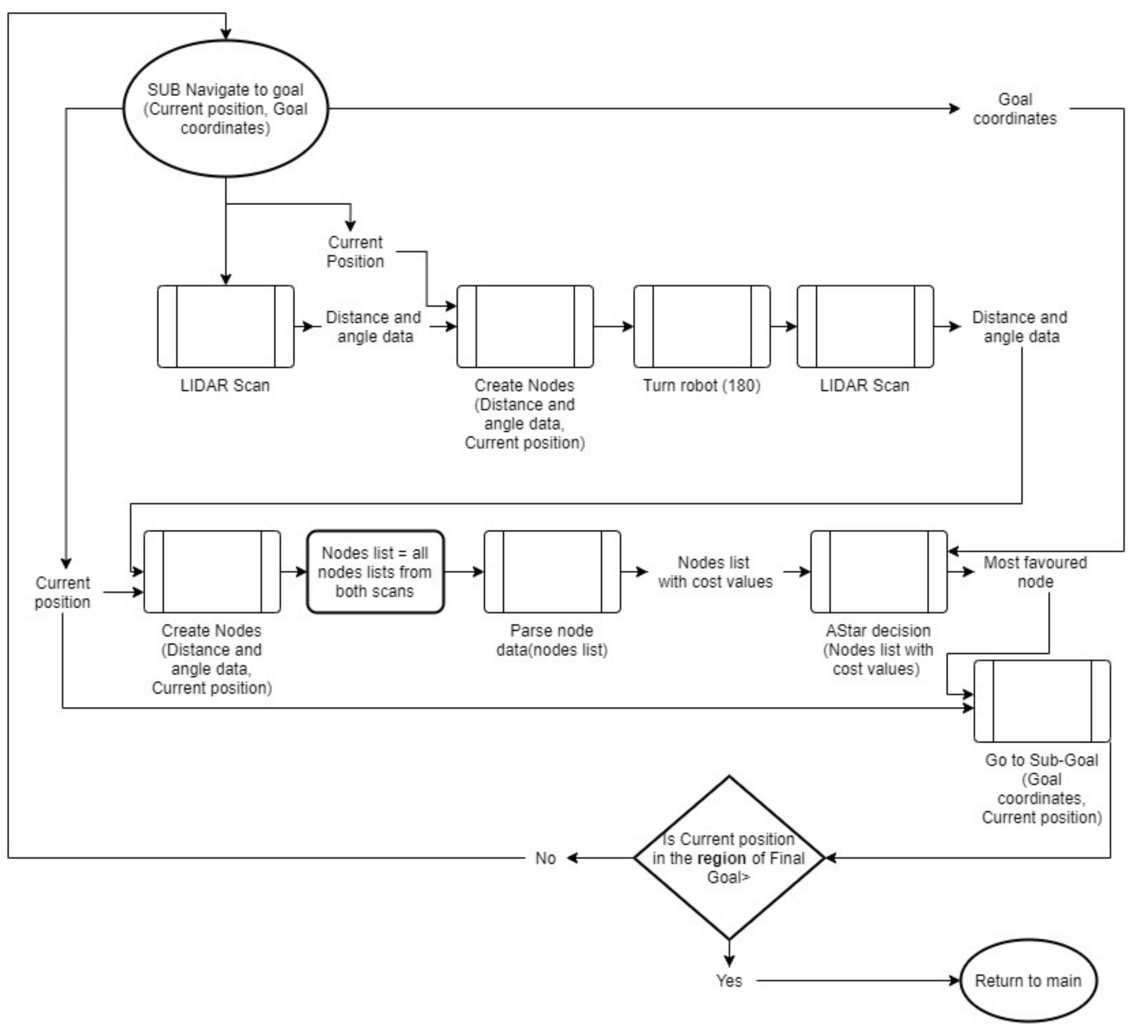
- Current position and goal coordinates are passed into the subroutine from main()
- The robot will do a full scan (0° to 180°)
- The robot body will turn 180° so that there is 360° of scanning data.

- For each scan, Nodes are created along the line (as above) and converted from 'distance and angle' data into absolute coordinates (relative to a 'home' position). All the 'End' Nodes are marked with a bubble of 'Infinite cost' (all other nodes inside the bubble immediately have their cost set to 'infinity' so that the robot cannot crash into a wall or obstacle.
- A cost is then calculated for the remaining nodes: cost = f(x) + g(x) = (distance to robot) + (distance to goal).
- A subroutine (go to sub-goal) is called.

This picture shows what the robot will do during (navigate to goal):

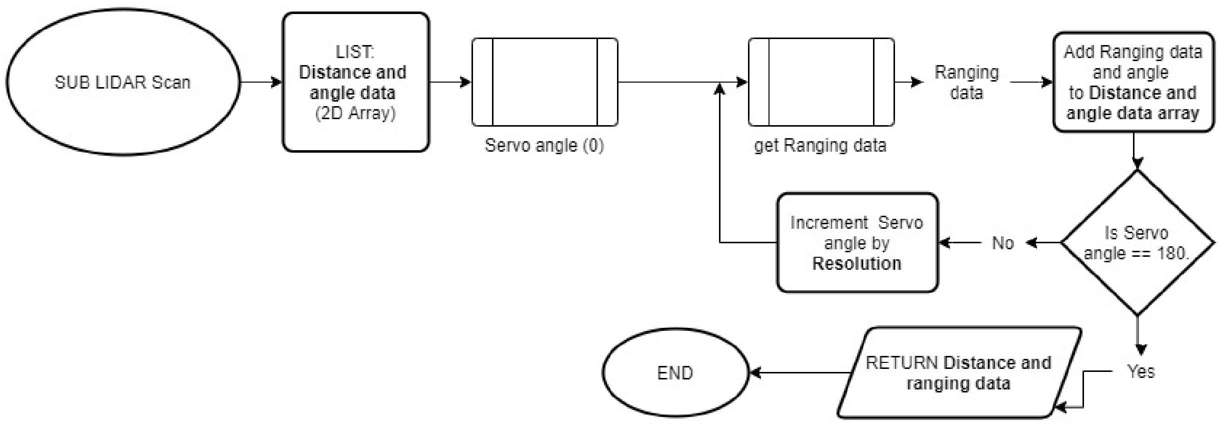


The full subroutine (navigate to goal) flow chart:


SUB Navigate to goal (Current position, Goal coordinates)
Goal coordinates
Current Position
Distance and angle data
LIDAR Scan
Create Nodes (Distance and angle data, Current position)
Turn robot (180)
LIDAR Scan
Distance and angle data
Current position
Create Nodes (Distance and angle data, Current position)
Nodes list = all nodes lists from both scans
Parse node data(nodes list)
Nodes list with cost values
AStar decision (Nodes list with cost values)
Most favoured node
Go to Sub-Goal (Goal coordinates, Current position)
Is Current position in the region of Final Goal>
No
Yes
Return to main

SUB LIDAR scan

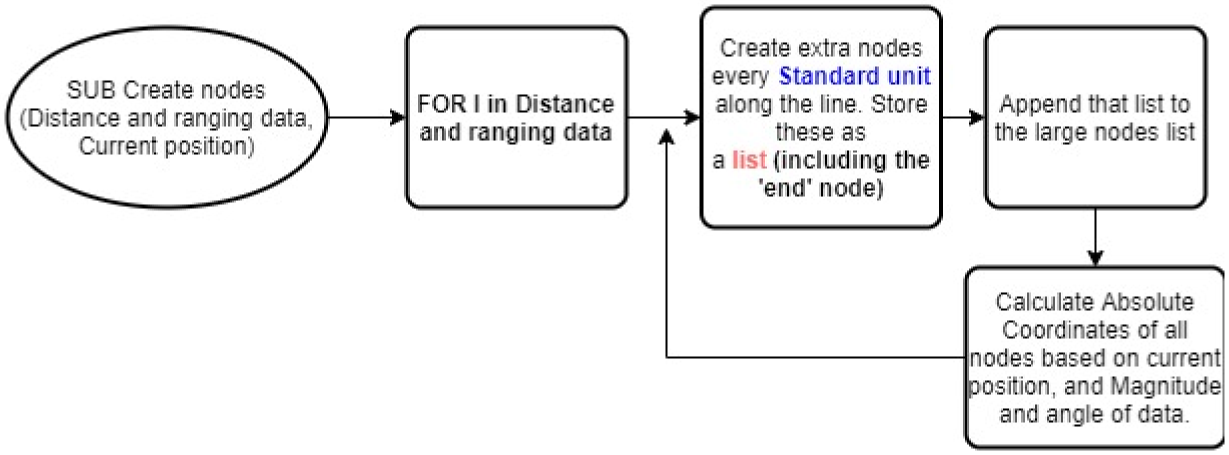
Because I am using a single point LIDAR to scan the environment, I am using a servo to change the angle of the LIDAR relative to the robot. The logic for LIDAR scan:


SUB LIDAR Scan
LIST: Distance and angle data (2D Array)
Servo angle (0)
get Ranging data
Ranging data
Add Ranging data and angle to Distance and angle data array
Is Servo angle == 180.
No
Increment Servo angle by Resolution
Yes
RETURN Distance and ranging data
END

NOTE: the subroutines (servo angle(angle)) and (get Ranging Data()) are ones that will be built using the python documentation for the raspberry pi (for controlling servos) and for the LIDAR.

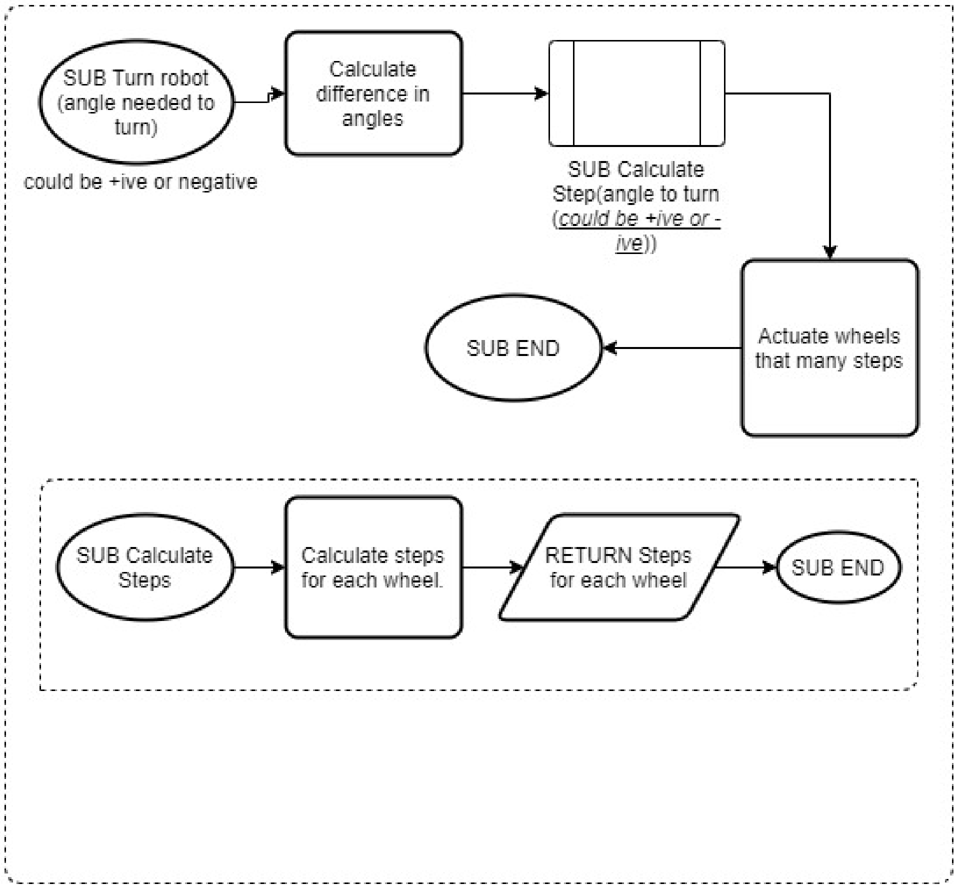
SUB Create Nodes

The logic for the Create nodes subroutine is as follows:


SUB Create nodes (Distance and ranging data, Current position)
FOR I in Distance and ranging data
Create extra nodes every Standard unit along the line. Store these as a list (including the 'end' node)
Append that list to the large nodes list
Calculate Absolute Coordinates of all nodes based on current position, and Magnitude and angle of data.

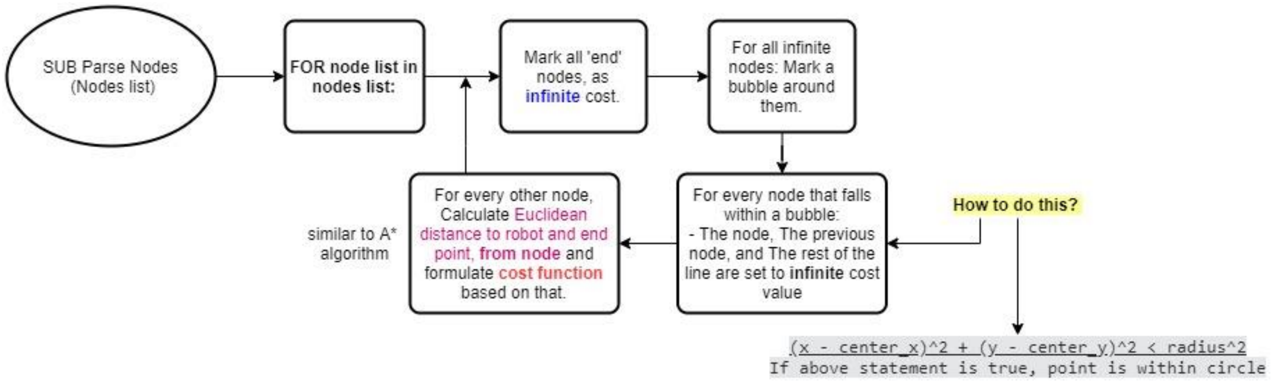
SUB Turn robot

The logic for the turn robot function is as follows:


SUB Turn robot (angle needed to turn)
could be +ive or negative
Calculate difference in angles
SUB Calculate Step(angle to turn (could be +ive or -ive))
Actuate wheels that many steps
SUB END
SUB Calculate Steps
Calculate steps for each wheel.
RETURN Steps for each wheel
SUB END

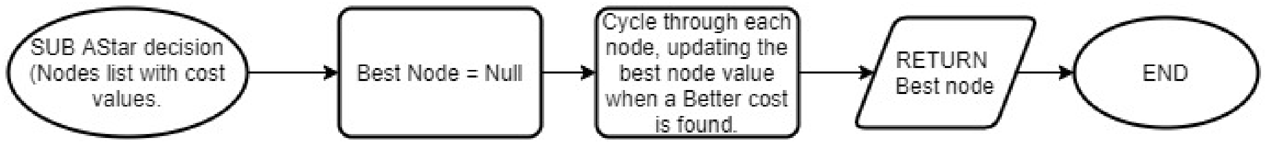
SUB Parse nodes

The logic for the parse nodes subroutine is as follows:


SUB Parse Nodes (Nodes list)
FOR node list in nodes list:
Mark all 'end' nodes, as infinite cost.
For all infinite nodes: Mark a bubble around them.
For every node that falls within a bubble: - The node, The previous node, and The rest of the line are set to infinite cost value
How to do this?
(x - center_x)^2 + (y - center_y)^2 < radius^2
If above statement is true, point is within circle
For every other node, Calculate Euclidean distance to robot and end point, from node and formulate cost function based on that.
similar to A* algorithm

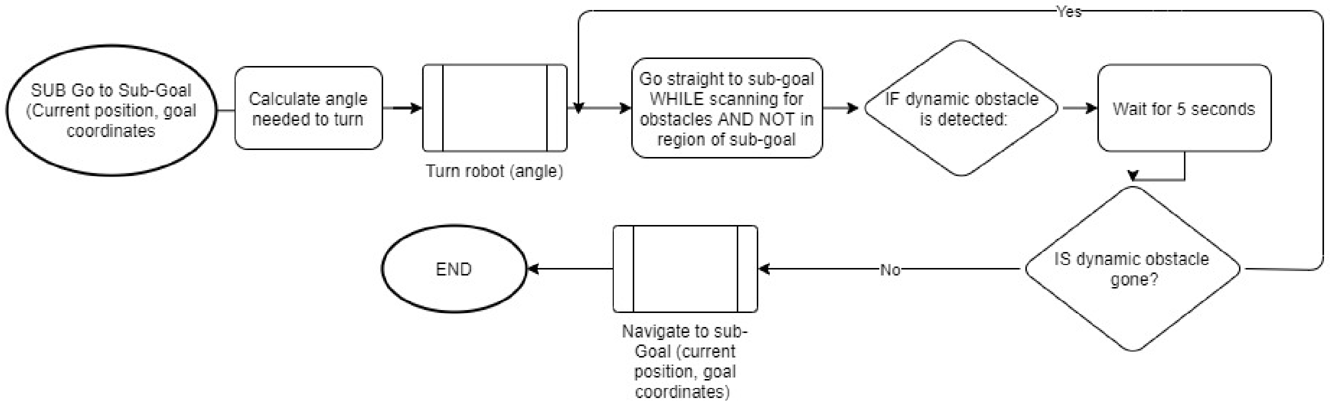
SUB AStar Decision

The logic for the AStar Decision subroutine is as follows:


SUB AStar decision (Nodes list with cost values.
Best Node = Null
Cycle through each node, updating the best node value when a Better cost is found.
RETURN Best node
END

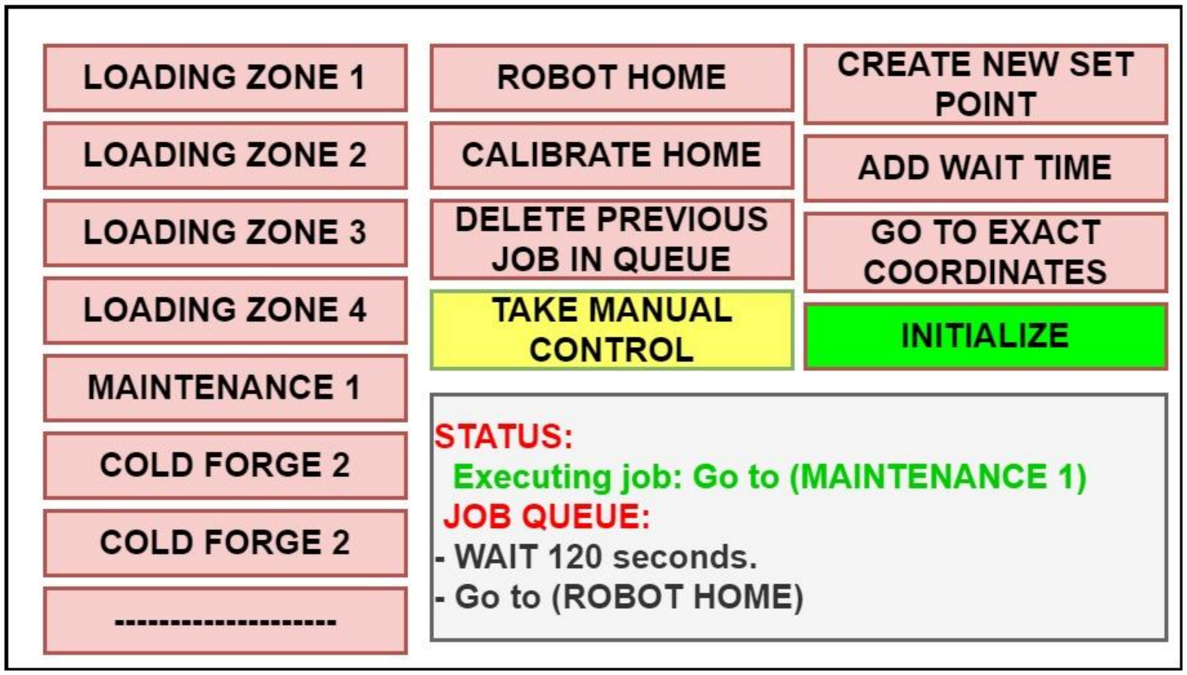
SUB Go to Sub-Goal

The logic for the Go to Sub-Goal subroutine is as follows:


SUB Go to Sub-Goal (Current position, goal coordinates
Calculate angle needed to turn
Turn robot (angle)
Go straight to sub-goal WHILE scanning for obstacles AND NOT in region of sub-goal
IF dynamic obstacle is detected:
Wait for 5 seconds
IS dynamic obstacle gone?
Yes
No
Navigate to sub-Goal (current position, goal coordinates)
END

2.4 Usability features

The plan for the graphical interface that the users will interact with (via a touchscreen on the back of the robot) is as follows:



STATUS:
Executing job: Go to (MAINTENANCE 1)
JOB QUEUE:
- WAIT 120 seconds.
- Go to (ROBOT HOME)

Part of graphical interface being included	Justification for including part
Ability to add pre-set points (e.g. LOADING ZONE 1) to a job queue.	This is part of **user requirement 4** and **essential feature 2.** The reason for these pre-set points being that if a certain process is repeated often, it is useful to be able to store the coordinates of set locations in an environment, and assign a name to them, so that when a location is added (e.g. COLD FORGE 1), and the job queue is **INIITALISED** (see below), that the robot will go the location (e.g. COLD FORGE 1). As soon as the user presses a pre-set location, it will be added to the job queue.
Ability to Delete last job in queue	This button allows the user to delete the last job in the queue. This is useful because, mistakes happen, and if the user accidentally adds a job to the queue they did not mean, they can remove it. (NOTE: a **job** includes things like waiting for a few seconds and going to locations).
ROBOT HOME	The robot should have a set home location that can be added to the job queue. This button is for when the robot needs to just 'get out of the way' when it is not needed, to increase productivity.
CALIBRATE HOME	Because the system is not perfect, assumptions like assuming a perfect friction coefficient, or if the robot gets bumped, the model of the real world can become skewed from reality, causing the robot's goal points and the actual goal point to have discrepancy. This means that the home needs to be calibrated every day or so.
TAKE MANUAL CONTROL	In order to create new set points (see below), and if the worker would like to travel with the robot, (e.g. when the robot is carrying something heavy for the worker), the ability to manually control the robot is incredibly useful. The TAKE MANUAL CONTROL button will switch the robot from autonomous mode to a manual mode and allows the robot to be control via a wireless PlayStation controller.

Ability to create and new set points in the environment (CREATE NEW SET POINT)	The ability to create set points is essential if you would like set points to be remembered so that they can be returned to. This would be done by taking manual control of the robot and driving towards the position you would like to set. Once the button is pressed, a pop-up keyboard will pop up, and a name can be set. The position will then be saved as the name given and can then be selected.
GO TO EXACT COORDINATES	**Essential feature 2** states that exact coordinates must be able to be input. This is because if any of the robots need repair / calibration by a technician, they can be tested with exact coordinates to improve accuracy. This is also useful in the development phase for iterative design.
ADD WAIT TIME	When this button is pressed, a pop-up keypad will appear, and the user will enter a time period (in seconds), and a WAIT job will be added to the job queue. This is useful because, when trying to automate a process or have the robot 'fetch' something from a location (see below).
INITIALIZE	This initialises the job queue, meaning whatever is currently in the job queue will be executed in that order. This means that jobs can be executed in a sequence, e.g. 1) Go to loading bay 1 (where the robot is loaded with materials), 2) wait 120 seconds, (for the material to be loaded), 3) Go to Cold forge 1. In this way, a worker can 'fetch' materials without having to move from his/her location, this is especially useful in an environment when many processes are repeated.
STATUS	This shows the current status of the robot (I.e. what job it is currently executing, if any) and the job queue. This is useful because it helps the users to see if they have entered the correct sequence of jobs, and for other users to check if the robot is available for use.

2.5 Data structures

Structure name	Data type	Where and how the structure will be used
currentPosition	Custom object that has three attributes, currentXPos, currentYPos, relativeAngle.	currentPosition will be used whenever the position of the robot is changed, either when the user is taking manual control, or when navigating automatically. - For this reason, the changing of the position variables will be directly linked to the actuation of the wheels - I.e. when the robot's wheels move, the position will move accordingly. The current position will be read from memory on start up.
distanceAndAngleData	This is an array of distances and angles relative to that	distanceAndAngleData will be output from the LIDAR scan subroutine and will be input into the create nodes subroutine which will

		create the additional nodes a certain standard unit along the line between the robot and the end point. It will also convert all the distance and angle data to absolute cartesian coordinates.
allNodesList	This is an 2d array of all the 'lines' of the scan. In each 'line' is the absolute coordinates of the nodes after a 360^o LIDAR scan.	This is the concatenation of the two 2d arrays from the two create nodes subroutines. It will be input into the parse nodes subroutine.
nodesListWithCostValues	This is a 2d array with each node coordinates and its corresponding cost value being in an array.	This will be passed into the AStar decision subroutine, where the best node will be picked from the list
goalNode	This is an array with 2 items, the x and y position of the most favourable node.	goalNode will be passed into the go to sub-goal subroutine, which will then navigate to that point.
File for saved points and current position	This is a .txt file with the names of the saved points and their corresponding coordinates.	This would be read when displaying the points on screen and written to when a new set point is created. It is a .txt file instead of just an array because these values need to be stored permanently, even when the system has been rebooted.
jobQueue	This is a 2D array with jobs in them. If the job has data associated with it, e.g. a wait job, the second space in the array will be used, otherwise, it is empty.	This would be written to when a new job is added and read when the jobs must be displayed.

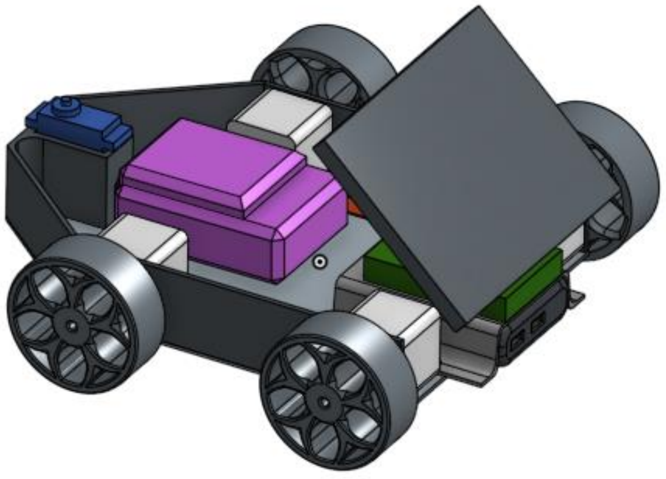
2.6 Physical design of the prototype robot

I designed the robot in CAD, first, and then used a 3d printer to print the more intricate parts. The base is an aluminium-polycarbonate composite like what is used in road signs.

The electrical system is powered by two batteries: one 4-cell lithium-ion pack provides 16V to the motor drivers and motors. A 5V power bank is used to power the Raspberry pi, as well as the top mounted touchscreen, and servo motor. The LIDAR unit gets its power from the raspberry pi's pins.

I am using stepper motors for the drive, as these can be precisely controlled (up to 0.05 of a degree in 1/32 step mode). Precise control is needed to turn the robot by exact amounts, so that the algorithm is effective.

The screen on the back is a 7” touchscreen designed to interface with the raspberry pi. This means the UI can be developed and tested with an actual touchscreen, and the prototype will be truer to the real intended design.



2.7 Test data for iterative development

Time input		
120		valid
1		valid - boundary
-1234		Invalid - erroneous
Cost values		
1002	valid	
-123	Invalid – cannot have negative cost value	
Distance values from the LIDAR		
7	valid	
0 (when no value is returned from LIDAR)	invalid – try again.	

2.8 Post-development testing plan

Testing plan	Test / testing data
Testing whether the robot will stop when presented with a dynamic obstacle while en route to a goal position. This is because of **essential feature 4**, where the safety of the workers in relation to the robot is deemed essential.	Test 1) Stand in front while en route to specified exact coordinates. Test 2) Stand in front while en route to a saved location Test 3) Stand in front during the beginning of a job. **Success: If the robot does not collide with the dynamic obstacle after each test, then the overall safety systems of the robot is successful.**
Testing whether the UI of the robot is intuitive enough. This is because of **essential feature 1**, where the intuition of the UI is deemed essential.	**Success: If 3 separate users can program a 3-step command successfully after having the UI explained once, the UI system is a success.**
Testing whether the input from the UI to the main program works and whether the robot can navigate to a position. This is because of **essential features 2 and 6**, where the ability to input specific coordinates, select pre-set ones and navigate to those coordinates is deemed essential.	Test 1) Enter exact coordinates Test 2) Select pre-set coordinates **Success: For both tests: If the robot goes to the same position (+ - 10cm) 3 times in a row when the same coordinates/set point are input from the home position, the input and navigation systems are successful.**
Testing whether the robot can avoid static obstacles to get to a location. This is because of **essential feature 3**, where the ability to avoid obstacles was deemed essential.	**Success: If the robot can avoid at least 2 obstacles on 5 trials (each trial having the obstacles in a different configuration), then the obstacle avoidance system for the robot is a success**
Testing whether the robot is using some form of path optimisation. This is because of **essential feature 7**, where a more intelligent path planning algorithm should be used (compared to a simply orthogonal path).	**Success: If the robot travels a shorter path to its goal location than an orthogonal path on 5 separate, different trials, then the path planning system is a success.**
Testing the job queue functionality of the robot. This is because of **essential feature 8**, where the ability to add jobs to, and display, a job queue, was deemed essential.	**Success: If a user can enter 5 jobs in a row, and the robot can execute them in order, removing jobs from the list as it goes (and displaying as such), then the job queue system is a success.**
Testing the robot's home return and calibration functionality. This is because of **essential feature 9**, where the ability to have a set home position, that can be calibrated (and is the home position for all the robot's future algorithms) was deemed essential.	**Success:** • **Mark the home position that the robot was initially calibrated on.** • **Send the robot to a set position, and mark where it stops.** • **Have the user drive the robot around for a long time, creating error in the system.** • **Have the user drive the robot to the marked home position and calibrate the home.**

	• **If the robot can return to the same set position as earlier (+- 10cm) then the home calibration system is a success.**
Testing the manual control capabilities. This is because of **essential feature 10**, where the ability to take full manual control (no control input from the robot) was deemed essential.	**Success: If the robot will follow the user's every command in manual control mode, (I.e., won't try to actively avoid obstacles) and is able to be recording its position constantly so that it can return to home with an accuracy of +- 30cm, the manual control system is a success.**

<u>Product backlog.</u>

Note: 'Worker' refers to warehouse, construction and factory workers.

	As a:	I want:	So that:	Priority	Sprint	Notes	Status
1	Worker	To be able to manually control the robot	Jobs that are not automated can be performed, with the robot doing the heavy lifting	Must	**1**: 21/09/20 - 07/11/20	Assembly and manual control of the robot will include getting hardware and control software working smoothly.	COMPLETE
2	Manager	The robot to have manual control functionality.	Set points can be easily set – Increased automation – increased productivity.	Must	**1**: 21/09/20 - 07/11/20		COMPLETE
3	Worker	The robot to navigate accurately.	I can rely on it not to crash into obstacles and deliver its payload to the correct location.	Must	**2**: 09/11/20 - 27/12/20	Getting mapping data using LIDAR, and using it to navigate, according to the algorithm outlined in design.	COMPLETE
4	Worker	To interact with the robot's	My job is made easier	Must	**3:** 29/12/20 -	Development of the UI Functionality.	COMPLETE

		functionality as intuitively as possible.	and not harder.		28/01/21		
5	Worker	To be able to add jobs to a job queue (including wait time)	Loading / unloading of materials can be made easier through sequences of steps	Must	**3:** 29/12/20 - 28/01/21	Part of the UI development	COMPLETE
6	Manager	The robot to be safe around humans	My workers are not injured (which would decrease productivity), and the company is not sued.	Must	**4:** 02/02/21 - 06/02/21	Addition to the *go to sub-goal* subroutine to make it safe (avoiding dynamic obstacles).	COMPLETE
7	Manager and owner	The robot to be reliable	Workflow is not interrupted because of repairs, and less money is spent on repairs.	Low priority for the prototype - more of a hardware issue for the final product.			

Development

3.1 Evidence at each stage of the iterative process

Sprint 1: 28/09/20 - 07/11/20 - Completed

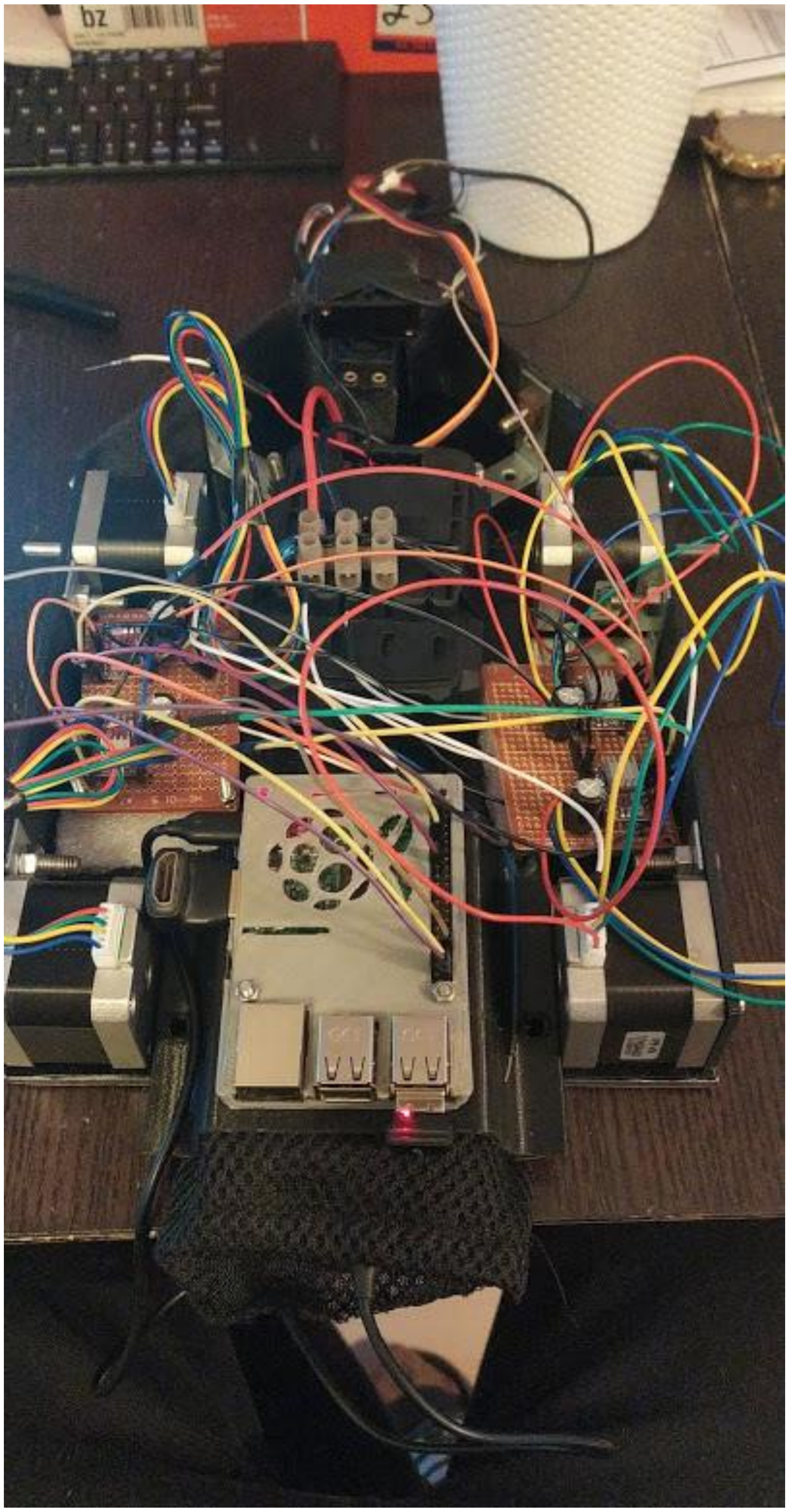
Work to be developed: Product backlog No. 1 and No.2

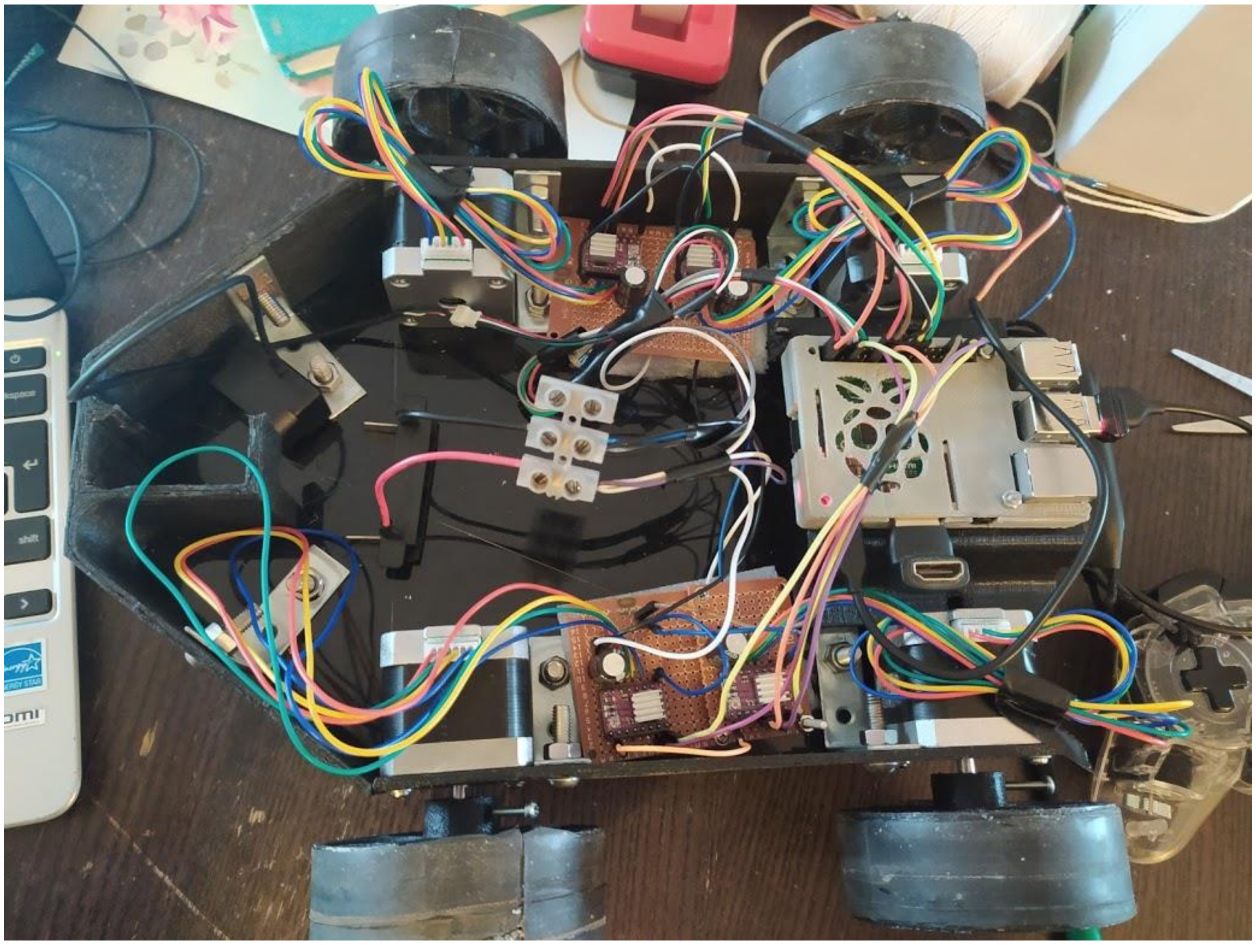
- Manual control functionality of the robot system. This includes:
    - Manufacture of the robot body (3d printed) - Completed
    - Assembly of the robot body - Completed
    - Assembly of electrical components - Completed
    - Development of control software for components - Completed
    - Integration of wireless controller to manually control robot - Completed

Evidence:

Manufacture of the robot body:

Making the robot was lots of fun, as it's the kind of project I do regularly. It improved my soldering skills substantially.





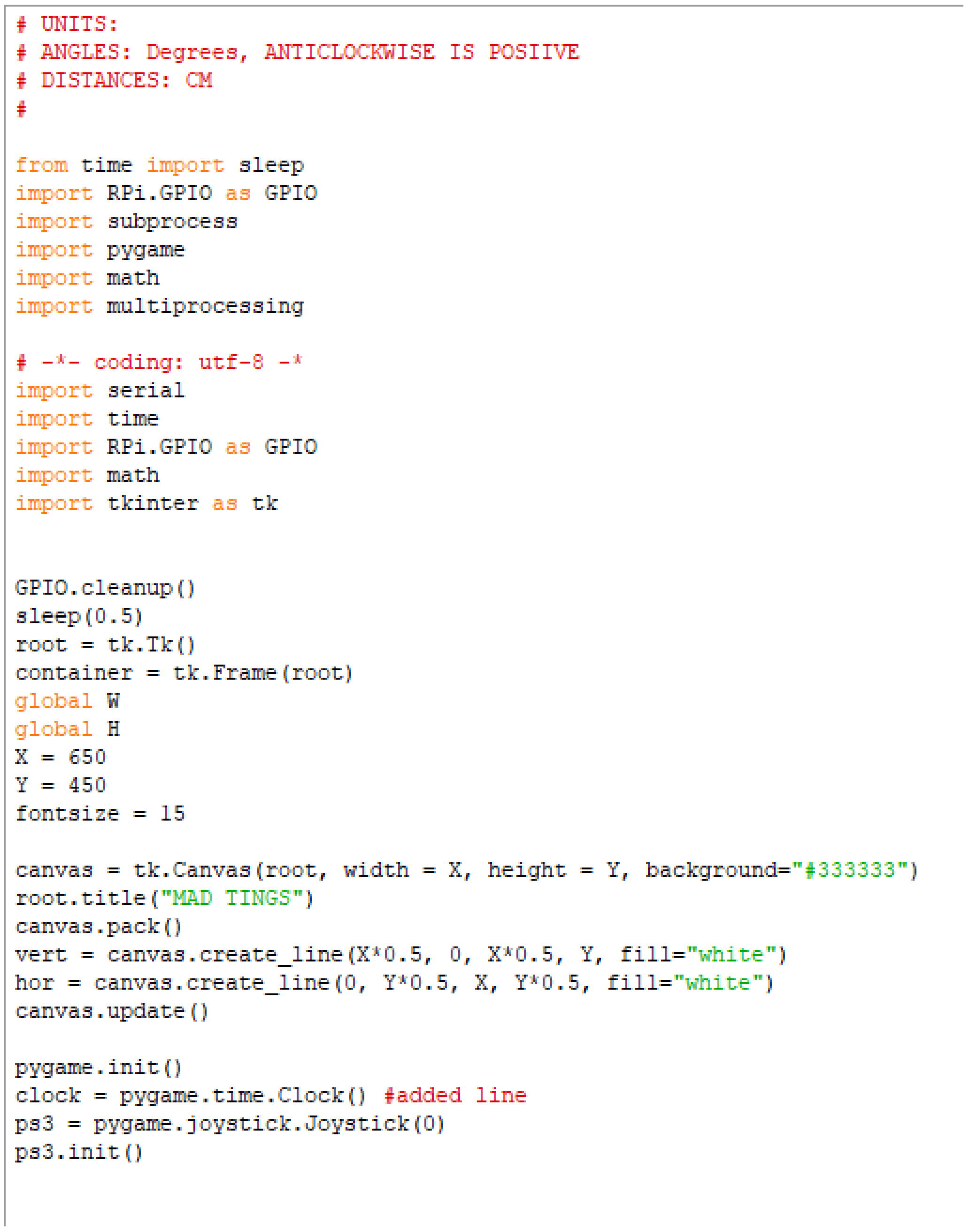
Control software for manual control:

For the control code, I followed a tutorial to ensure the correct wiring for my control board, and to understand the software inputs and outputs. Source: https://www.rototron.info/raspberry-pi-stepper-motor-tutorial/

For the wireless controller integration, I used a piece of software I had used previously, and the **ps3** library (I am using a third party ps3 controller).

Initialise code: (setting up variables etc):

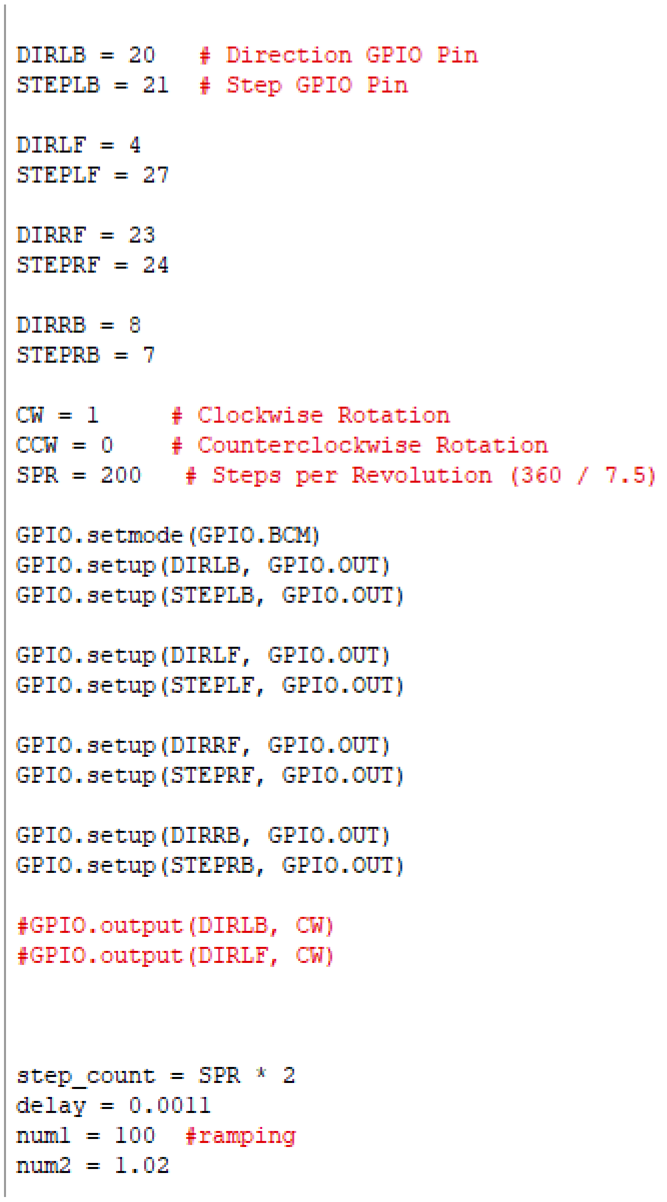
Libraries are imported and the tkinter canvas and ps3 controller is set up:



```python
# UNITS:
# ANGLES: Degrees, ANTICLOCKWISE IS POSIIVE
# DISTANCES: CM
#

from time import sleep
import RPi.GPIO as GPIO
import subprocess
import pygame
import math
import multiprocessing

# -*- coding: utf-8 -*
import serial
import time
import RPi.GPIO as GPIO
import math
import tkinter as tk


GPIO.cleanup()
sleep(0.5)
root = tk.Tk()
container = tk.Frame(root)
global W
global H
X = 650
Y = 450
fontsize = 15

canvas = tk.Canvas(root, width = X, height = Y, background="#333333")
root.title("MAD TINGS")
canvas.pack()
vert = canvas.create_line(X*0.5, 0, X*0.5, Y, fill="white")
hor = canvas.create_line(0, Y*0.5, X, Y*0.5, fill="white")
canvas.update()

pygame.init()
clock = pygame.time.Clock() #added line
ps3 = pygame.joystick.Joystick(0)
ps3.init()
```

Pins are set up for the control of the stepper motors and **step_count** is declared as the steps per revolution of the nema-17 stepper motor (200) multiplied by 2 (as I am running the steppers in half step mode). **delay** is the highest possible delay for the steps, this value is increased and decreased during ramping to speed up and slow down the motors. **num1** and **num2** are variables that handle the ‘ramping’ of the stepper motors. Ramping is used to ensure the acceleration of the wheels are not too great, meaning the wheels do not slip.

NOTE: These variables are global variables as they are used all over the program.

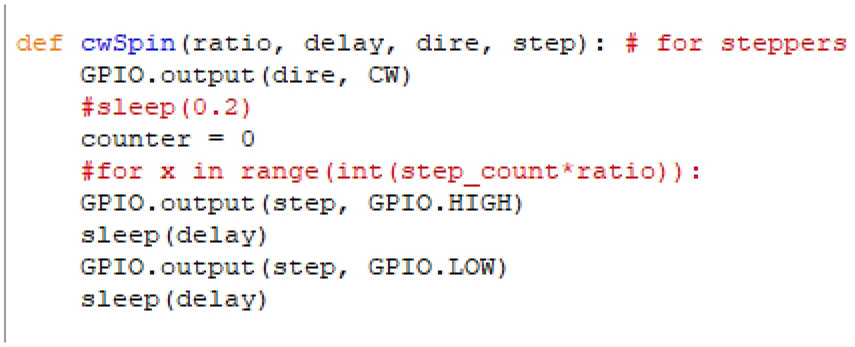


```
DIRLB = 20   # Direction GPIO Pin
STEPLB = 21  # Step GPIO Pin

DIRLF = 4
STEPLF = 27

DIRRF = 23
STEPRF = 24

DIRRB = 8
STEPRB = 7

CW = 1     # Clockwise Rotation
CCW = 0    # Counterclockwise Rotation
SPR = 200   # Steps per Revolution (360 / 7.5)

GPIO.setmode(GPIO.BCM)
GPIO.setup(DIRLB, GPIO.OUT)
GPIO.setup(STEPLB, GPIO.OUT)

GPIO.setup(DIRLF, GPIO.OUT)
GPIO.setup(STEPLF, GPIO.OUT)

GPIO.setup(DIRRF, GPIO.OUT)
GPIO.setup(STEPRF, GPIO.OUT)

GPIO.setup(DIRRB, GPIO.OUT)
GPIO.setup(STEPRB, GPIO.OUT)

#GPIO.output(DIRLB, CW)
#GPIO.output(DIRLF, CW)



step_count = SPR * 2
delay = 0.0011
num1 = 100  #ramping
num2 = 1.02
```

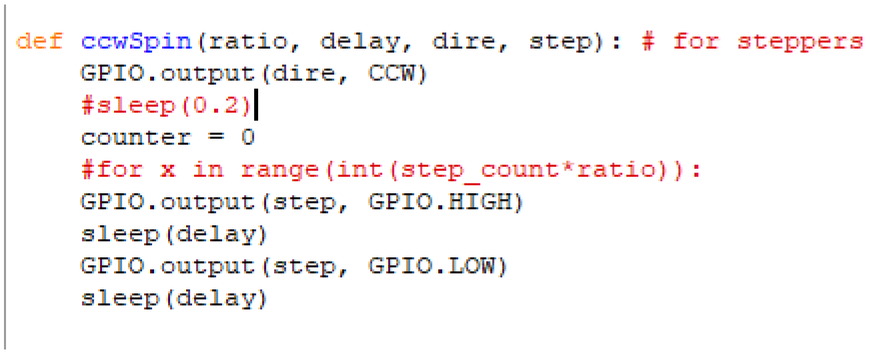
Manual control and Precision control subroutines (combined):

Redundant code will be cleaned up later when I have confirmed that it is redundant and that I don't need it anymore.

The cwSpin and ccwSpin controls individual stepper motors to turn clockwise or counter-clockwise:



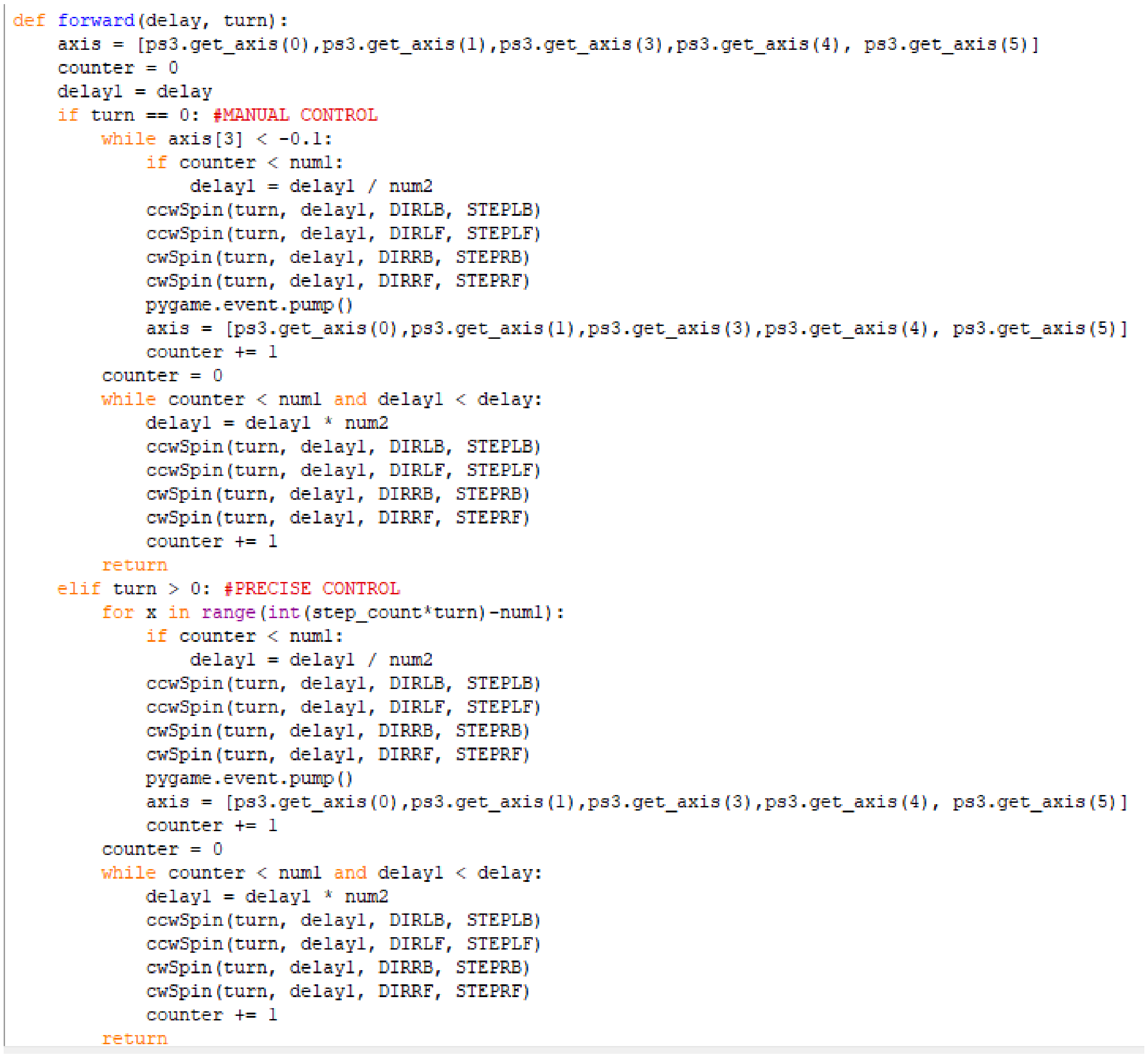
```
def cwSpin(ratio, delay, dire, step): # for steppers
    GPIO.output(dire, CW)
    #sleep(0.2)
    counter = 0
    #for x in range(int(step_count*ratio)):
    GPIO.output(step, GPIO.HIGH)
    sleep(delay)
    GPIO.output(step, GPIO.LOW)
    sleep(delay)
```



```
def ccwSpin(ratio, delay, dire, step): # for steppers
    GPIO.output(dire, CCW)
    #sleep(0.2)
    counter = 0
    #for x in range(int(step_count*ratio)):
    GPIO.output(step, GPIO.HIGH)
    sleep(delay)
    GPIO.output(step, GPIO.LOW)
    sleep(delay)
```

The forward subroutine has two purposes: During manual control (when turn is equal to 0) the subroutine will handle travelling forward (including ramping up and down) by reading values from the ps3 controller.

During precision control (when turn is more than 0) the subroutine will advance the all stepper motors a set number of turns (e.g. 1.2582 turns) also considering ramping up and down during that cycle



```python
def forward(delay, turn):
    axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
    counter = 0
    delay1 = delay
    if turn == 0: #MANUAL CONTROL
        while axis[3] < -0.1:
            if counter < num1:
                delay1 = delay1 / num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            counter += 1
        return
    elif turn > 0: #PRECISE CONTROL
        for x in range(int(step_count*turn)-num1):
            if counter < num1:
                delay1 = delay1 / num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            counter += 1
        return
```

The backward subroutine is similar to the forward subroutine, but with the controls reversed.

```
def backward(delay, turn):
    turn = 1
    axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
    counter = 0
    delay1 = delay
    if turn == 0:
        while axis[3] > 0.1:
            if counter < num1:
                delay1 = delay1 / num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            counter += 1
        return
    elif turn > 0: #PRECISE CONTROL
        for x in range(int(step_count*turn)-num1):
            if counter < num1:
                delay1 = delay1 / num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            counter += 1
        return
```

spinLeft and spinRight is similar to the forward and backward subroutines but turns the steppers in a way that they spin the whole robot anticlockwise and clockwise respectively.

```
def spinLeft(delay, turn):
    axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
    counter = 0
    delay1 = delay
    if turn == 0:
        while axis[2] < -0.1:
            if counter < num1:
                delay1 = delay1 / num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            counter += 1
        return
    elif turn > 0: #PRECISE CONTROL
        delay1 = delay1 / 3
        for x in range(int(step_count*turn)):
            if counter < 30:
                delay1 = delay1 / num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
```

```python
def spinRight(delay, turn):
    axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
    counter = 0
    delay1 = delay
    if turn == 0:
        while axis[2] > 0.1:
            if counter < num1:
                delay1 = delay1 / num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            counter += 1
        return
    elif turn > 0:
        delay1 = delay1 / 3
        for x in range(int(step_count*turn)):
            if counter < 30:
                delay1 = delay1 / num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            ccwSpin(turn, delay1, DIRRB, STEPRB)
            ccwSpin(turn, delay1, DIRRF, STEPRF)
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
```

The spin subroutine determines which subroutine to use based on the value of turn that is calculated in a later subroutine.

The manualControl subroutine determines which subroutine to use based on the values of the ps3 controller outputs. This will be called when the robot switched to manual control mode.

```
def spin(turn):
    if turn > 0:
        spinLeft(delay, turn) # anticlockwise
    elif turn < 0:
        spinRight(delay, -turn) # clockwise
    return

def manualControl():
    while True:
        counter = 0
        #delay1 = delay
        turn = 0
        pygame.event.pump()
        axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
        print(axis)

        if axis[3] < -0.1:
            forward(delay, turn)


        if axis [3] > 0.1:
            backward(delay, turn)

        if axis[2] < -0.1:
            spinLeft(delay, turn)


        if axis[2] > 0.1:
            spinRight(delay, turn)
```

Testing to inform development:

After getting the control software fully operational and testing it on a wooden floor, I realised that due to the smooth plastic that makes up the wheels, it was slipping when starting and stopping – this is a problem because the algorithm works based on calculations to do with the rotation of the wheels and assumes no slip when the wheels turn. - To fix this I added rubber strips to the outside of the wheels. The wheels were still slipping after this, so I added **ramping** code to slowly increase the speed of the motors (in this way the force due to acceleration is less than the force needed to overcome static friction). After re-testing, the slipping problem was solved, and the wheels now translate rotary motion to linear motion perfectly.

Sprint 2: 09/11/20 - 27/12/20 - Completed

Work to be developed: Product backlog No. 3

- Control software for servo and LIDAR – Completed
- Control software for turning set angles and travelling set distances. - Completed
- Translation between different coordinate systems:
    - (x, y) to magnitude and angle from position - Completed
    - Magnitude and angle from position to (x, y) - Completed

- *Create nodes* and *parse nodes* subroutines (outlined above) - Completed
- *AStar decision* subroutine (outlined above) - Completed
- *Navigate to goal* subroutine and *navigate to sub-goal* subroutine to tie it all together (outlined above). - Completed

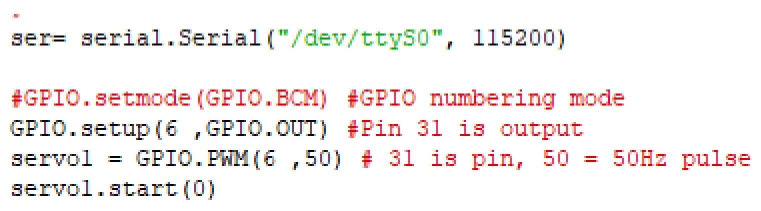
Evidence:

Control software for servo and LIDAR:

For the servo motor control software, I used my knowledge of servo motor systems, and the raspberry pi to create the code from scratch.

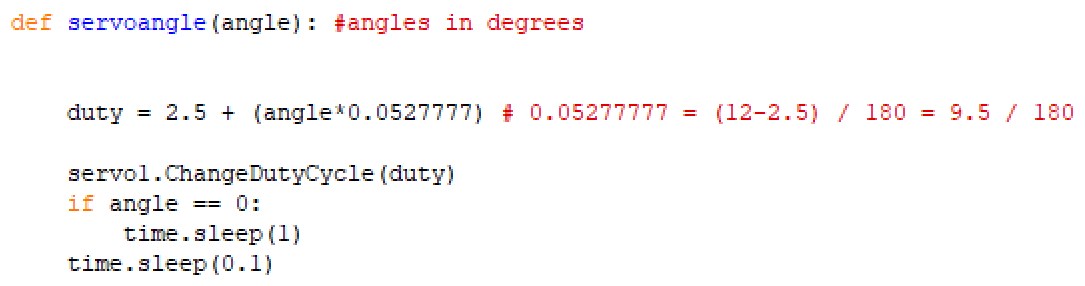
For the LIDAR control software, I used the TFmini library that was provided by the LIDAR company, and followed a tutorial online to set it up. https://github.com/TFmini/TFmini-Plus

First, more global variables are declared:



```
ser= serial.Serial("/dev/ttyS0", 115200)

#GPIO.setmode(GPIO.BCM) #GPIO numbering mode
GPIO.setup(6 ,GPIO.OUT) #Pin 31 is output
servo1 = GPIO.PWM(6 ,50) # 31 is pin, 50 = 50Hz pulse
servo1.start(0)
```

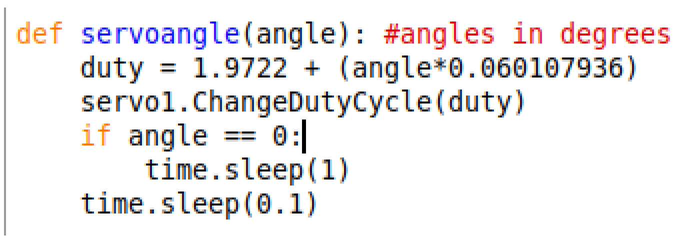
The servoangle subroutine translates an angle (from 0 – 180 degrees) into a **duty cycle** signal which is then sent to the servo motor to turn it to that specific angle.



```
def servoangle(angle): #angles in degrees


    duty = 2.5 + (angle*0.0527777) # 0.05277777 = (12-2.5) / 180 = 9.5 / 180

    servo1.ChangeDutyCycle(duty)
    if angle == 0:
        time.sleep(1)
    time.sleep(0.1)
```

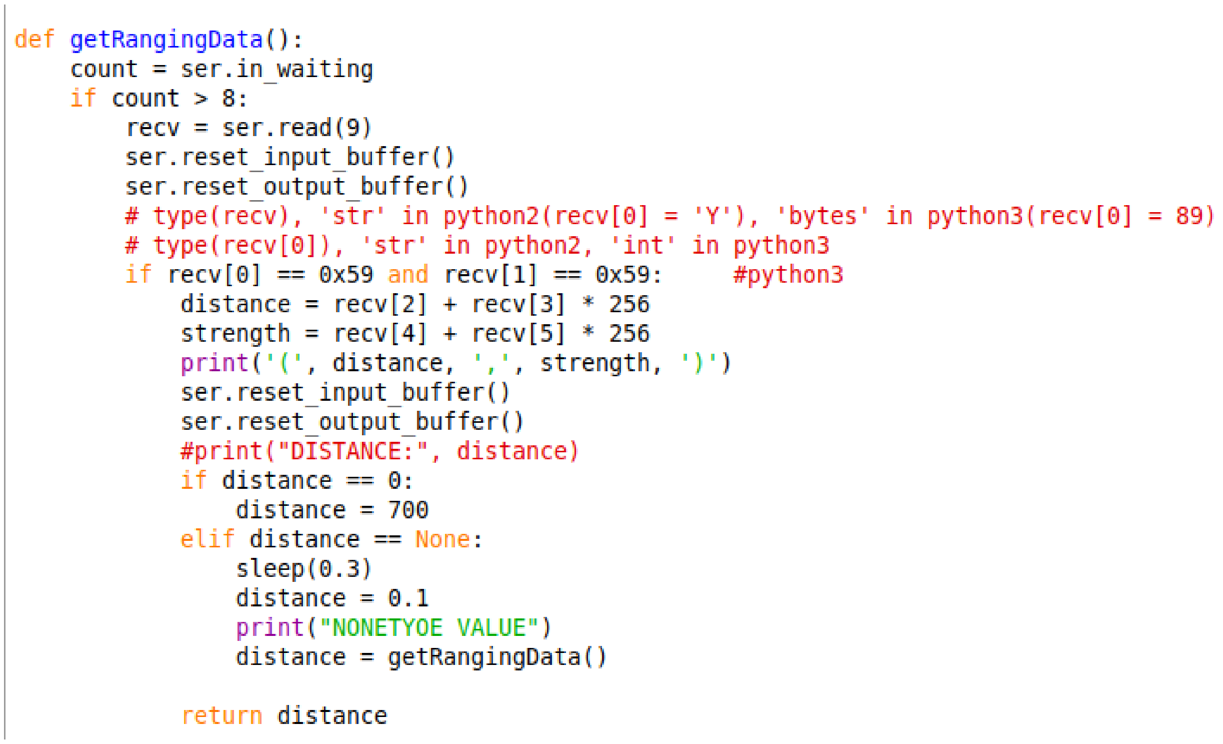
UPDATE: 02/01/21:

After some testing, I found the servo wasn't extending to 180 degrees, so I calibrated it, the new subroutine looks like this:



```
def servoangle(angle): #angles in degrees
    duty = 1.9722 + (angle*0.060107936)
    servo1.ChangeDutyCycle(duty)
    if angle == 0:
        time.sleep(1)
    time.sleep(0.1)
```

20/11/20: To obtain a ranging from the LIDAR, the getRangingData subroutine is used. This uses code from the LIDAR's library (Tfmini) to obtain a distance value and the strength of the signal. NOTE: if the lidar returns a distance value of zero, the object it was pointing at was more than its max reach (of about 7m in my testing), so it can be assumed to be at least 7m away. If it returns a None value, the lidar didn't operate properly, and just needs to be tried again.



```
def getRangingData():
    count = ser.in_waiting
    if count > 8:
        recv = ser.read(9)
        ser.reset_input_buffer()
        ser.reset_output_buffer()
        # type(recv), 'str' in python2(recv[0] = 'Y'), 'bytes' in python3(recv[0] = 89)
        # type(recv[0]), 'str' in python2, 'int' in python3
        if recv[0] == 0x59 and recv[1] == 0x59:     #python3
            distance = recv[2] + recv[3] * 256
            strength = recv[4] + recv[5] * 256
            print('(', distance, ',', strength, ')')
            ser.reset_input_buffer()
            ser.reset_output_buffer()
            #print("DISTANCE:", distance)
            if distance == 0:
                distance = 700
            elif distance == None:
                sleep(0.3)
                distance = 0.1
                print("NONETYOE VALUE")
                distance = getRangingData()

            return distance
```

The scan subroutine performs a 180-degree scan using the servo and LIDAR. It returns a list of ranging data and their respective angles.

```python
def scan(res):
    time.sleep(0.2)
    if ser.is_open == False:
        ser.open()
    l = []
    currentangle = 0
    while currentangle <= 180:
        servoangle(currentangle)
        distance = getRangingData()
        if distance == 0:
            distance = 700
        elif distance == None:# Sometimes if the signal is weak,
            sleep(0.3)        # the sensor can return a None value and needs to be redone
            continue
        l.append([])
        l[-1].append(currentangle) # angle THEN
        l[-1].append(distance) #distance
        currentangle += res
    return l # l is the list of ranging data and their respective angles
```

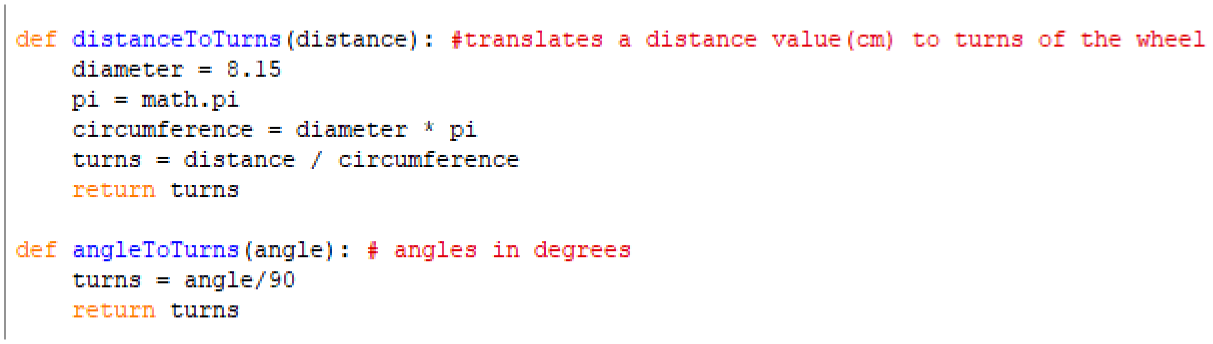
Control software for turning set angles and travelling set distances:

20/11/20:

For controlling the robot to go a certain distance forwards, I assumed no slippage, and set created a function to calculate the number of turns of the wheel for travelling a set distance forwards. With proper acceleration and deceleration of the wheel, I can obtain zero slipping for the wheels and so my assumption is accurate reducing error from wheels.

The control software for turning a set angle assumed that (using differential steering) 1 rotation of the wheels each side (going opposite directions each side) resulted in a 90 rotation for the robot body – From testing this was found to be fairly accurate, but there can be some error depending on the surface used. On hard laminate flooring, the robot can consistently turn set angles very accurately, however, not all surfaces have the same frictional properties. This is the main uncertainty in my code logic, so it is something I will have to keep in mind during testing, to make sure that systematic errors do not distort results over time.

The distanceToTurns and angleToTurns subroutines translate distance and angle values to the number of turns of the wheel needed respectively.



```
def distanceToTurns(distance): #translates a distance value(cm) to turns of the wheel
    diameter = 8.15
    pi = math.pi
    circumference = diameter * pi
    turns = distance / circumference
    return turns

def angleToTurns(angle): # angles in degrees
    turns = angle/90
    return turns
```

Translation between different coordinate systems:

MagnitudeAndAngleToxy converts from magnitude and angle form, to x and y coordinates. This also factors in the current position (parameter: a).

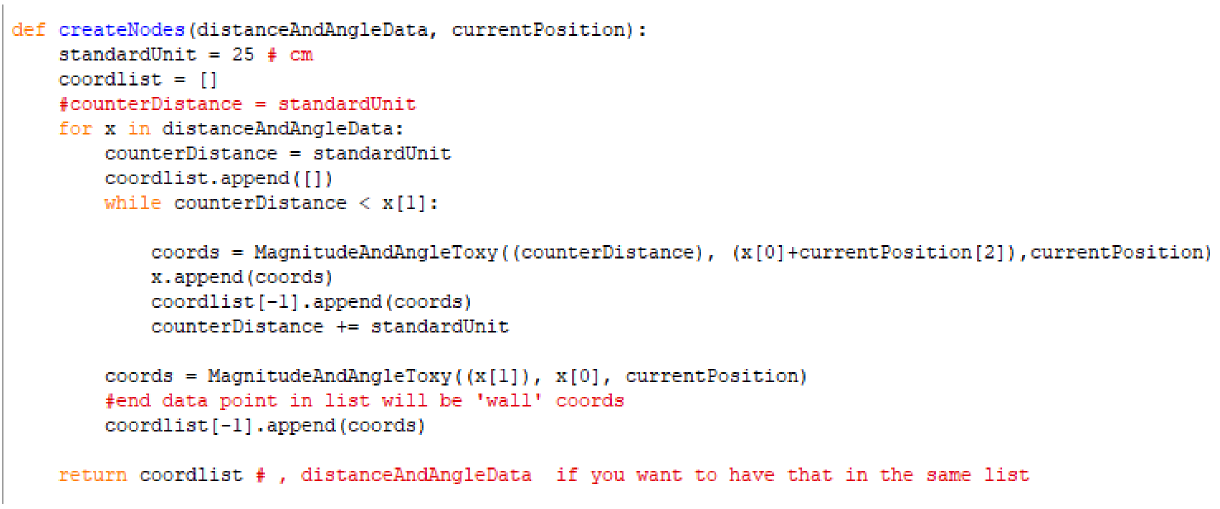
```python
def MagnitudeAndAngleToxy(mag, angle, a):
    angle = angle-90# at a, plus mag and angle, what pos would i be at?
    xplus = mag*(math.cos(math.radians(angle))) # x
    yplus = mag*(math.sin(math.radians(angle)))# y
    b = [(a[0] + xplus), (a[1] + yplus)]
    b[0], b[1] = round(b[0], 4), round(b[1], 4)
    return b
```

xyToMagnitudeAndAngle converts x and y coordinates to the angle and magnitude the from position a. It answers the question: “at a, what angle do I need to turn, and how far do need to go forward to get to b?”

```python
def xyToMagnitudeAndAngle(a, b): # Note: a contains 'angle position' data
    mag = distance(a, b)
    try:
        grad = (b[1] - a[1]) / (b[0] - a[0])
    except:
        grad = 99999
    angle = (math.degrees(math.atan(grad)) - a[2])
    print("MAG, ANGLE:",mag, angle)
    return mag, angle
```

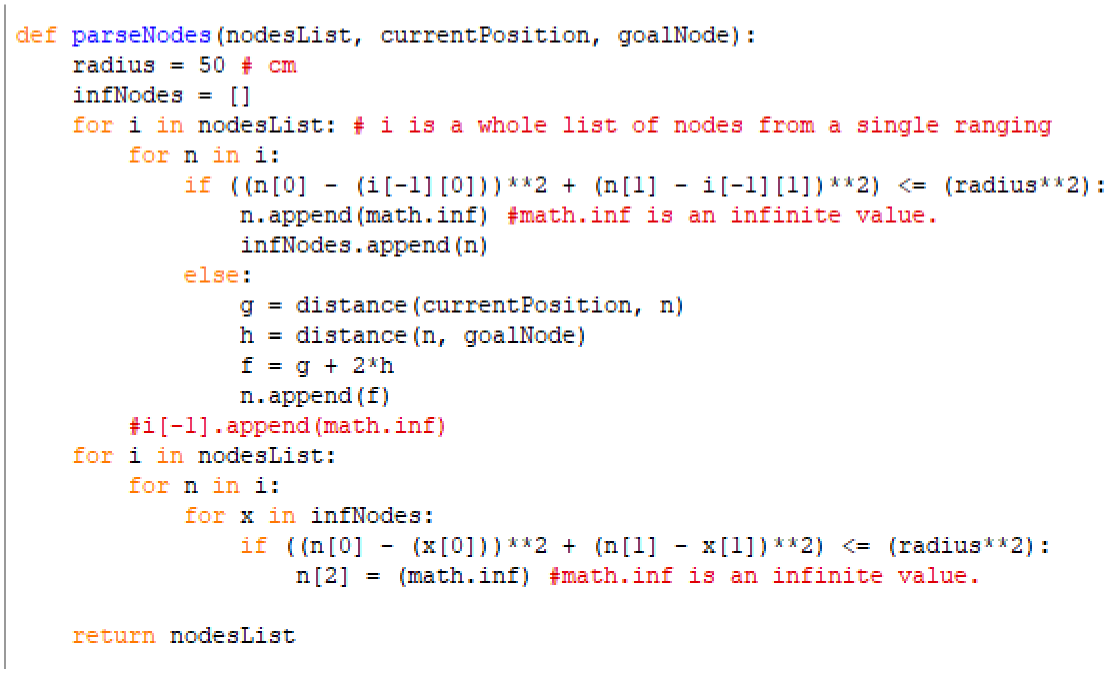
Create nodes and parse nodes subroutine:

The createNodes subroutine takes distance and angle data from a scan, converts them to x and y coordinates, and then performs the logic explained in **design**, creating additional nodes every standardUnit along each line.



```python
def createNodes(distanceAndAngleData, currentPosition):
    standardUnit = 25 # cm
    coordlist = []
    #counterDistance = standardUnit
    for x in distanceAndAngleData:
        counterDistance = standardUnit
        coordlist.append([])
        while counterDistance < x[1]:

            coords = MagnitudeAndAngleToxy((counterDistance), (x[0]+currentPosition[2]),currentPosition)
            x.append(coords)
            coordlist[-1].append(coords)
            counterDistance += standardUnit

        coords = MagnitudeAndAngleToxy((x[1]), x[0], currentPosition)
        #end data point in list will be 'wall' coords
        coordlist[-1].append(coords)

    return coordlist # , distanceAndAngleData  if you want to have that in the same list
```

The parseNodes subroutine parses the nodes such that, each node is given a cost value, according to the logic in **design.**



```python
def parseNodes(nodesList, currentPosition, goalNode):
    radius = 50 # cm
    infNodes = []
    for i in nodesList: # i is a whole list of nodes from a single ranging
        for n in i:
            if ((n[0] - (i[-1][0]))**2 + (n[1] - i[-1][1])**2) <= (radius**2):
                n.append(math.inf) #math.inf is an infinite value.
                infNodes.append(n)
            else:
                g = distance(currentPosition, n)
                h = distance(n, goalNode)
                f = g + 2*h
                n.append(f)
        #i[-1].append(math.inf)
    for i in nodesList:
        for n in i:
            for x in infNodes:
                if ((n[0] - (x[0]))**2 + (n[1] - x[1])**2) <= (radius**2):
                    n[2] = (math.inf) #math.inf is an infinite value.

    return nodesList
```

UPDATE: 04/12/20

This new revision of the parseNodes subroutine incorporates the idea of “if a node crosses into a bubble of infinite cost, the rest of the line of nodes from that scan are also of infinite cost”. This reduces errors and from my testing, means the robot doesn’t crash nearly as much into the edges of obstacles. NOTE:

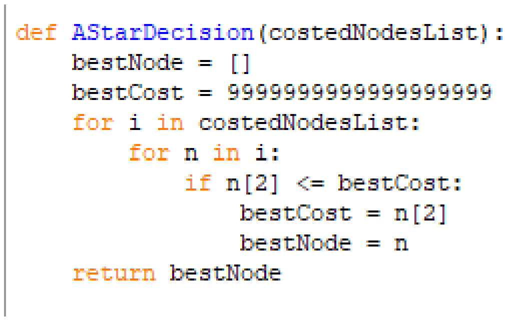
The nodes that are within a certain radius of an end node are given a cost of 99999 instead of “infinite”. This is simply for the display of the scanning data which helps me with debugging and testing, but this has the same effect as setting the cost to “infinite”.

```python
def parseNodes(nodesList, currentPosition, goalNode):
    radius = 25 # cm
    infNodes = []
    endnodes = []
    for x in nodesList:
        endnodes.append(x[-1])
    for i in nodesList: # i is a whole list of nodes from a single ranging
        for n in i:
            if ((n[0] - (i[-1][0]))**2 + ((n[1] - i[-1][1])**2)) <= (radius**2):
                n.append(99999) #math.inf is an infinite value.
                infNodes.append(n)
            else:
                g = distance(currentPosition, n)
                h = distance(n, goalNode)
                f = g + 2*h
                n.append(f)
        #i[-1].append(math.inf)
    for i in nodesList:
        for n in i:
            for x in endnodes:
                if ((n[0] - (x[0]))**2 + (n[1] - x[1])**2) <= (radius**2) and n[2] != 99999:
                    n[2] = (math.inf)
                    for a in i[(i.index(n)+1):-1]:
                        a[2] = (math.inf)



    return nodesList
```

AStar decision subroutine:

The code kept on choosing nodes that were too close to the robot's current position, resulting in many shorter journeys, rather than a single longer journey (this is due to rounding errors in my calculations of distance). To fix this, I am using a **weighted heuristic** (f(n) = g(n) + **2**h(n)) to bias the chosen node towards the goal side, cancelling out the rounding error. The weighted heuristic can be seen in the parseNodes subroutine above.

AStarDecision decides on the best possible node to go to, based on the nodes list given by the parseNodes subroutine.



```
def AStarDecision(costedNodesList):
    bestNode = []
    bestCost = 9999999999999999999
    for i in costedNodesList:
        for n in i:
            if n[2] <= bestCost:
                bestCost = n[2]
                bestNode = n
    return bestNode
```

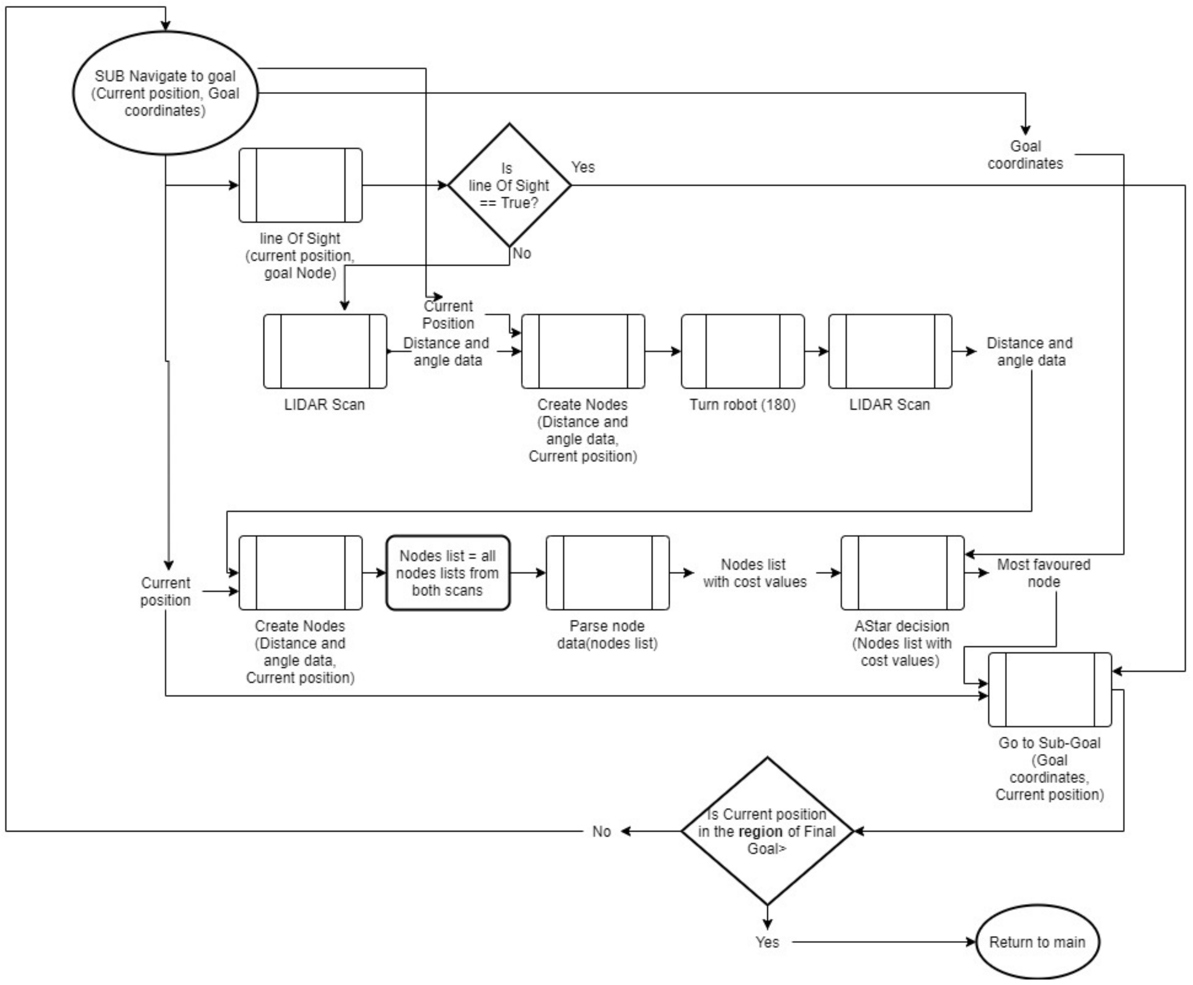
Navigate to goal subroutine and navigate to sub-goal subroutine:

20/11/20:

During my testing of the Navigate to goal subroutine, I found that the control software worked flawlessly, but the decision as to which sub-node to go to was problematic. I was able to fix this eventually, the bug was a systematic error meaning the angles were not offset by the current angle of the robot (relative to home), so it was choosing nodes that were behind walls etc).

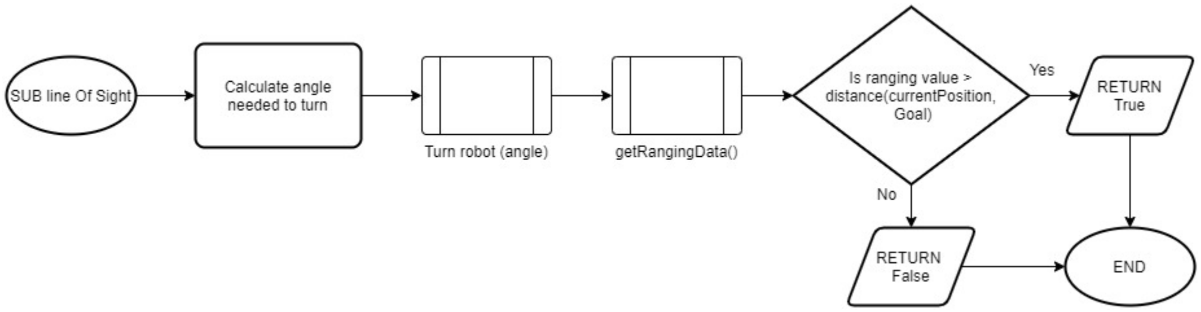
I am also changing the design of the code to include aspects from another path finding algorithm, called **Theta*** or **Basic Theta.** The concept is this: "If there is a line of sight between the goal node and current position, then go straight to the goal node". This is improving

The new Navigate to goal subroutine design now looks like this:


SUB Navigate to goal (Current position, Goal coordinates)
line Of Sight (current position, goal Node)
Is line Of Sight == True?
Yes
No
Goal coordinates
Current Position
Distance and angle data
LIDAR Scan
Create Nodes (Distance and angle data, Current position)
Turn robot (180)
LIDAR Scan
Distance and angle data
Current position
Create Nodes (Distance and angle data, Current position)
Nodes list = all nodes lists from both scans
Parse node data(nodes list)
Nodes list with cost values
AStar decision (Nodes list with cost values)
Most favoured node
Go to Sub-Goal (Goal coordinates, Current position)
Is Current position in the region of Final Goal>
No
Yes
Return to main

Now, in the case of there being a direct line of sight between the robot and the final goal, the sub-goal coordinates are set to the final goal coordinates, and the Go to Sub-Goal subroutine is then executed.

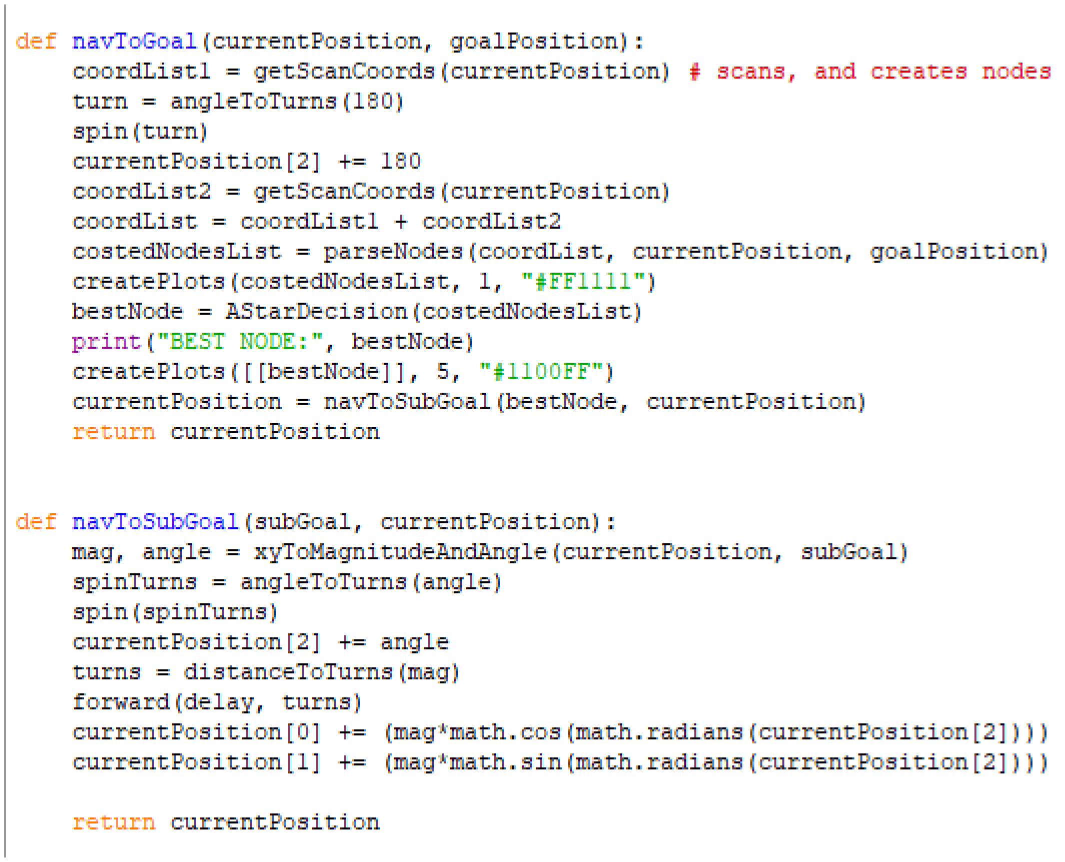
The lineOfSight subroutine design is:



lineOfSight subroutine code, following the logic outlined above:

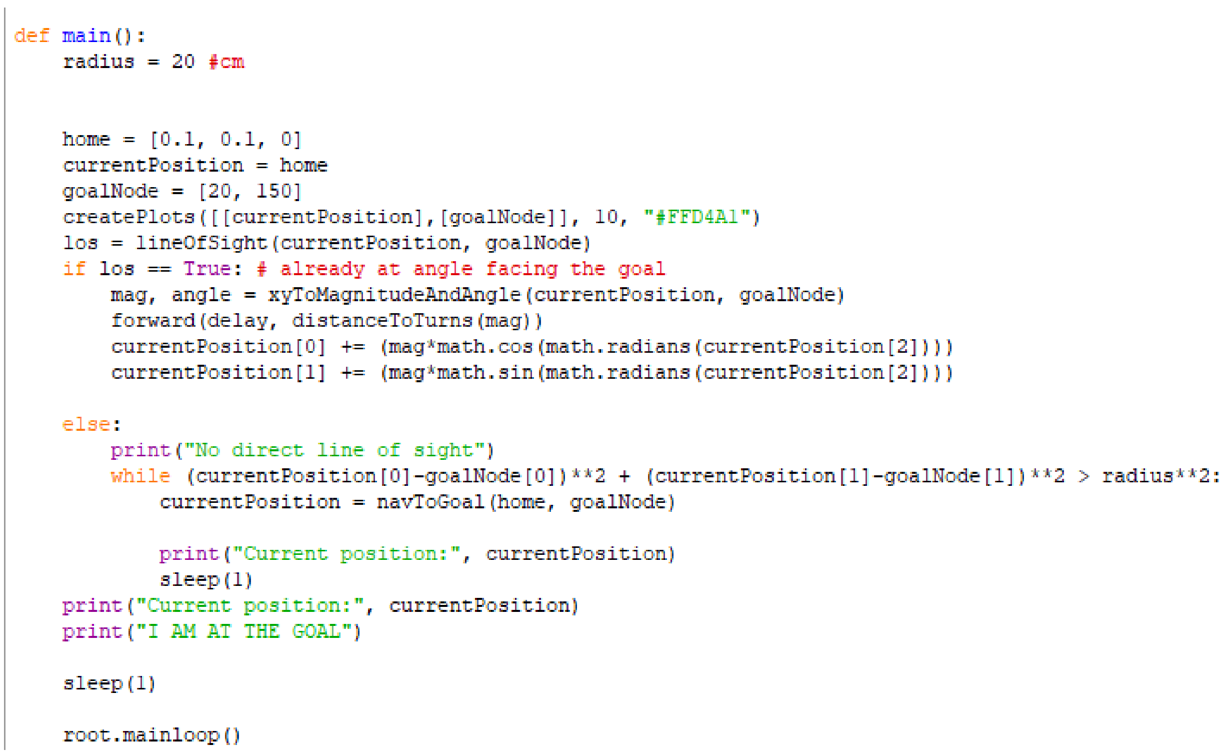
```python
def lineOfSight(currentPosition, goalNode):
    mag, angle = xyToMagnitudeAndAngle(currentPosition, goalNode)
    spin(angleToTurns(angle))
    currentPosition[2] += angle
    servoangle(90)
    sleep(1)
    ranging = 0
    for _ in range(3):
        ranging1 = getRangingData()
        ranging += ranging1
        sleep(0.5)
    ranging = ranging / 3
    print("RANGING:", ranging)
    sleep(1)
    if ranging > mag:
        return True
    else:
        spin(angleToTurns(-angle))
        currentPosition[2] += -angle
        return False
```

20/11/20: navToGoal and navToSubGoal subroutines code, following the logic mentioned above.



```
def navToGoal(currentPosition, goalPosition):
    coordList1 = getScanCoords(currentPosition) # scans, and creates nodes
    turn = angleToTurns(180)
    spin(turn)
    currentPosition[2] += 180
    coordList2 = getScanCoords(currentPosition)
    coordList = coordList1 + coordList2
    costedNodesList = parseNodes(coordList, currentPosition, goalPosition)
    createPlots(costedNodesList, 1, "#FF1111")
    bestNode = AStarDecision(costedNodesList)
    print("BEST NODE:", bestNode)
    createPlots([[bestNode]], 5, "#1100FF")
    currentPosition = navToSubGoal(bestNode, currentPosition)
    return currentPosition


def navToSubGoal(subGoal, currentPosition):
    mag, angle = xyToMagnitudeAndAngle(currentPosition, subGoal)
    spinTurns = angleToTurns(angle)
    spin(spinTurns)
    currentPosition[2] += angle
    turns = distanceToTurns(mag)
    forward(delay, turns)
    currentPosition[0] += (mag*math.cos(math.radians(currentPosition[2])))
    currentPosition[1] += (mag*math.sin(math.radians(currentPosition[2])))

    return currentPosition
```

Currently, the main subroutine looks like this. However, this is just for testing purposes, and the code and data may change (as with the other subroutines, as more and more bugs are slowly discovered and sorted):



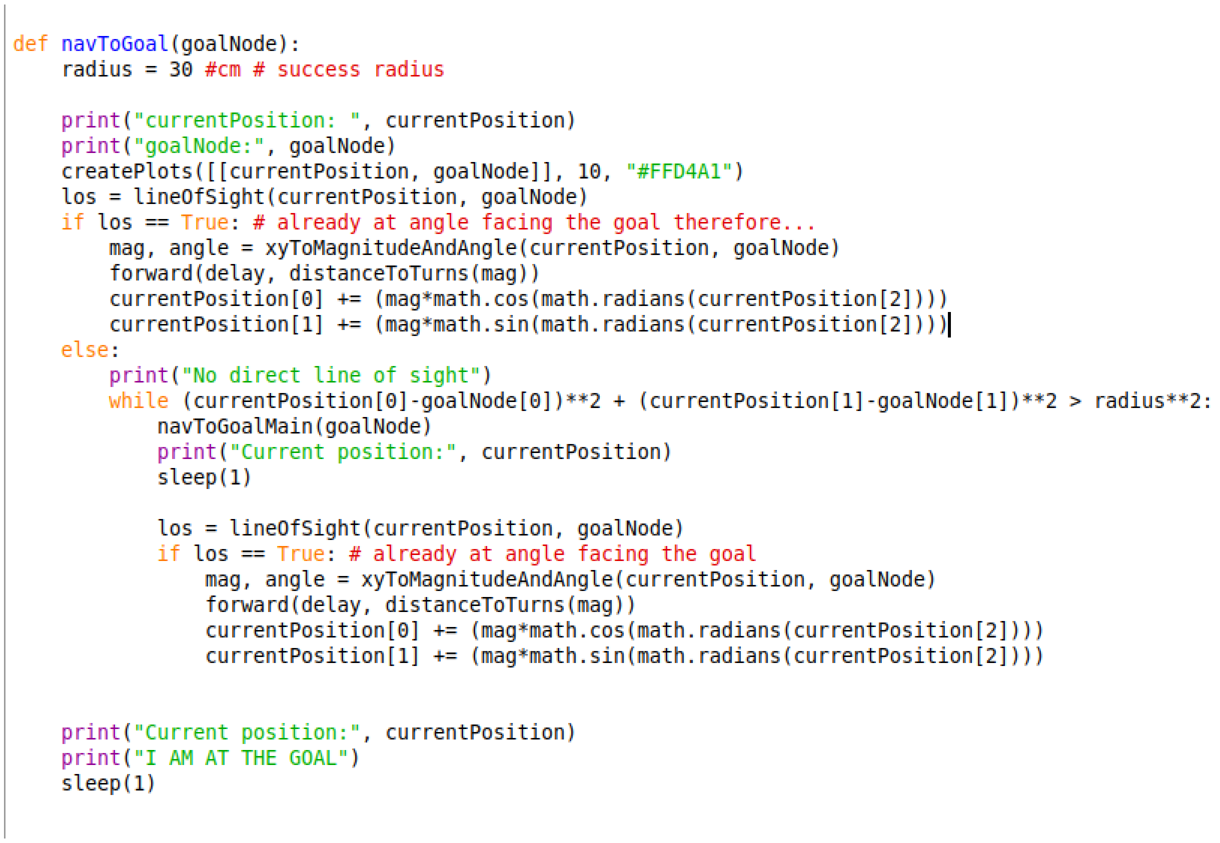
```python
def main():
    radius = 20 #cm


    home = [0.1, 0.1, 0]
    currentPosition = home
    goalNode = [20, 150]
    createPlots([[currentPosition],[goalNode]], 10, "#FFD4A1")
    los = lineOfSight(currentPosition, goalNode)
    if los == True: # already at angle facing the goal
        mag, angle = xyToMagnitudeAndAngle(currentPosition, goalNode)
        forward(delay, distanceToTurns(mag))
        currentPosition[0] += (mag*math.cos(math.radians(currentPosition[2])))
        currentPosition[1] += (mag*math.sin(math.radians(currentPosition[2])))

    else:
        print("No direct line of sight")
        while (currentPosition[0]-goalNode[0])**2 + (currentPosition[1]-goalNode[1])**2 > radius**2:
            currentPosition = navToGoal(home, goalNode)

            print("Current position:", currentPosition)
            sleep(1)
    print("Current position:", currentPosition)
    print("I AM AT THE GOAL")

    sleep(1)

    root.mainloop()
```

20/11/20: NOTE: In some of the subroutines, the createPlots subroutine was called. This subroutine is just a simple piece of plotting code for plotting the robot movements and scanning data using the tkinter library. It is primarily used for testing and development purposes.

UPDATE: 12/12/20:

The overall structure of the main, navToGoal, and navToSubGoal has changed somewhat, so that now, in the interest of cleaner code, currentPosition has been changed into SOLELY a global variable, instead of passing it around as a local variable. This makes for cleaner code, as well as making it easier for the UI logic to access the variable when it needs to:

navToGoal is now the main logic of what was previously the main() subroutine. This now calls another function, called navToGoalMain (see below).



```python
def navToGoal(goalNode):
    radius = 30 #cm # success radius

    print("currentPosition: ", currentPosition)
    print("goalNode:", goalNode)
    createPlots([[currentPosition, goalNode]], 10, "#FFD4A1")
    los = lineOfSight(currentPosition, goalNode)
    if los == True: # already at angle facing the goal therefore...
        mag, angle = xyToMagnitudeAndAngle(currentPosition, goalNode)
        forward(delay, distanceToTurns(mag))
        currentPosition[0] += (mag*math.cos(math.radians(currentPosition[2])))
        currentPosition[1] += (mag*math.sin(math.radians(currentPosition[2])))
    else:
        print("No direct line of sight")
        while (currentPosition[0]-goalNode[0])**2 + (currentPosition[1]-goalNode[1])**2 > radius**2:
            navToGoalMain(goalNode)
            print("Current position:", currentPosition)
            sleep(1)

            los = lineOfSight(currentPosition, goalNode)
            if los == True: # already at angle facing the goal
                mag, angle = xyToMagnitudeAndAngle(currentPosition, goalNode)
                forward(delay, distanceToTurns(mag))
                currentPosition[0] += (mag*math.cos(math.radians(currentPosition[2])))
                currentPosition[1] += (mag*math.sin(math.radians(currentPosition[2])))


    print("Current position:", currentPosition)
    print("I AM AT THE GOAL")
    sleep(1)
```

navToGoalMain is mostly the same code as what was previously named (before 07/12/20) navToGoal, but now only takes a single argument goalNode, and also deals with displaying the scan data (using createPlots).

```
def navToGoalMain(goalNode):
    coordList1 = getScanCoords(currentPosition) # scans, and creates nodes
    turn = angleToTurns(180)
    spin(turn)
    currentPosition[2] += 180
    coordList2 = getScanCoords(currentPosition)
    coordList = coordList1 + coordList2
    costedNodesList = parseNodes(coordList, currentPosition, goalNode)
    infNodes = []
    nNodes = []
    redNodes = []
    for i in costedNodesList:
        for n in i:
            if n[2] == 99999:
                redNodes.append(n)
            elif n[2] == math.inf:
                infNodes.append(n)
            else:
                nNodes.append(n)
    print(nNodes)
    createPlots([infNodes], 1.5, "#FFFFFF", )
    createPlots([nNodes], 1.5, "#08FF08")
    createPlots([redNodes], 3, "#FF0000")

    bestNode = AStarDecision(costedNodesList)
    print("BEST NODE:", bestNode)
    createPlots([[bestNode]], 3, "#1100FF")
    navToSubGoal(bestNode)
```

The navToSubGoal subroutine is the same as before, but with no local currentPosition handling:

```
def navToSubGoal(subGoal):
    print(currentPosition, "GOING INTO xytoMandA")
    mag, angle = xyToMagnitudeAndAngle(currentPosition, subGoal)
    spinTurns = angleToTurns(angle)
    spin(spinTurns)
    currentPosition[2] += angle
    turns = distanceToTurns(mag)
    forward(delay, turns)
    currentPosition[0] += (mag*math.cos(math.radians(currentPosition[2])))
    currentPosition[1] += (mag*math.sin(math.radians(currentPosition[2])))
    print(currentPosition, "FROM NAVTOSUBGOAL  ")
```

Sprint 3: 29/12/20 - 29/10/21 - Completed

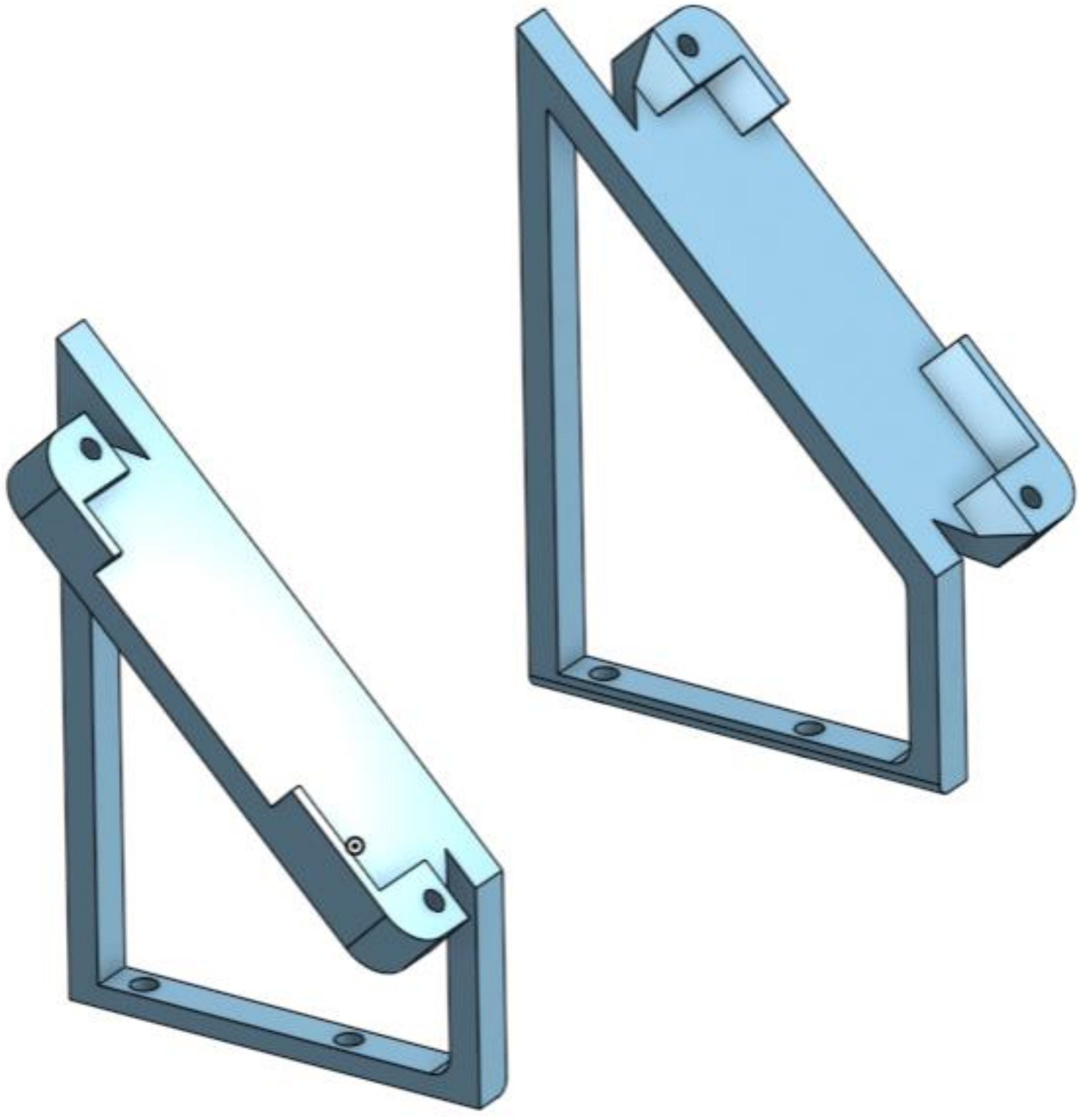
Work to be developed: Product backlog No. 4 and No. 5:

- Design of stand for touchscreen and mounting of touchscreen onto robot - Completed
- Development of UI buttons and layout. - Completed.
- Development of logic behind UI. - Completed.

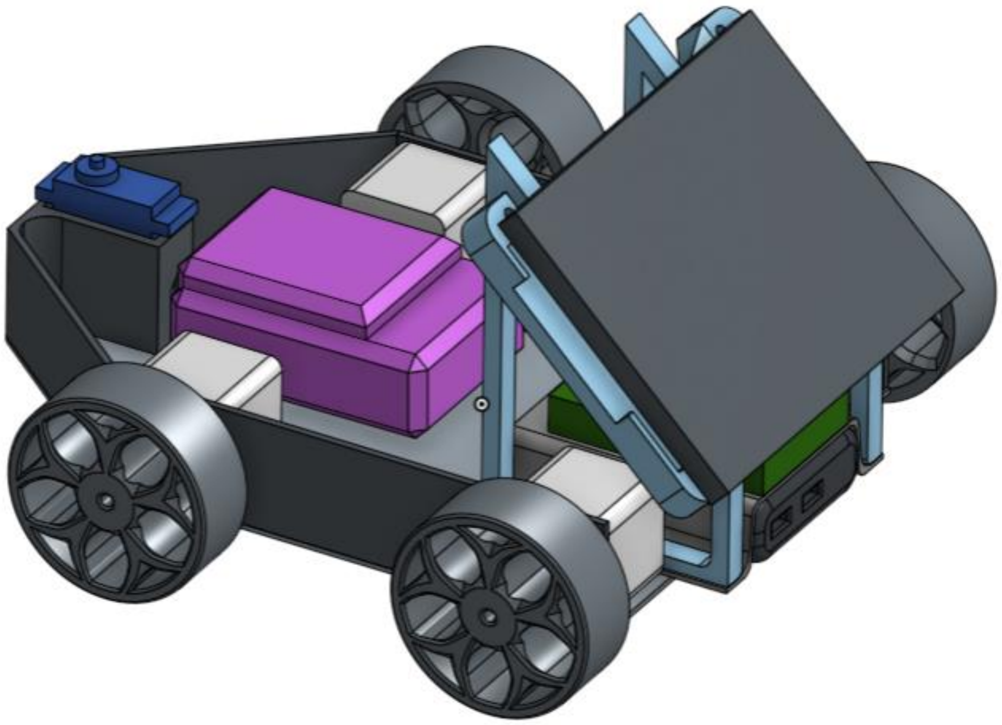
Evidence:

Design of stand for touchscreen and mounting of touchscreen onto robot

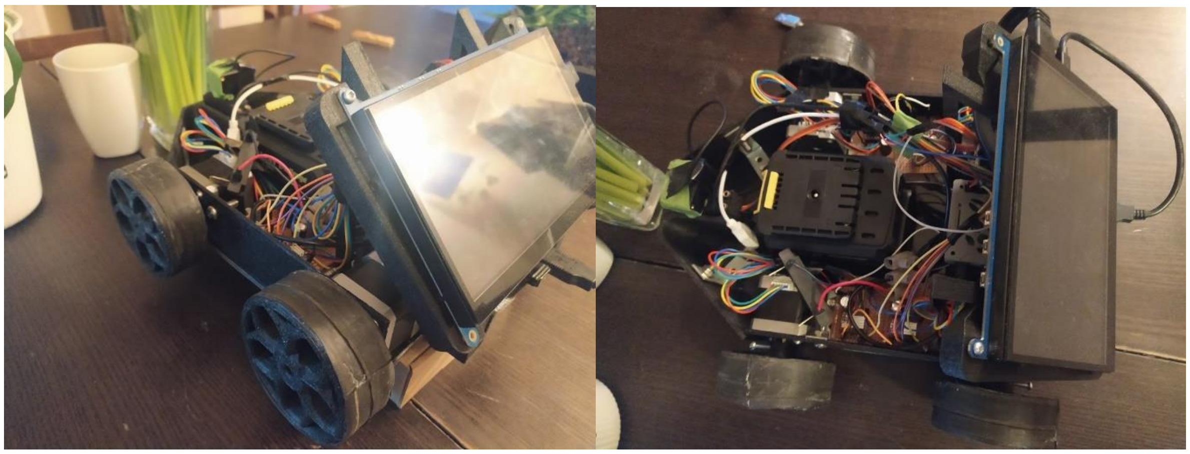
02/01/21: For the screen to be mounted properly, it needs to be supported by a custom-designed stand. I designed the stand using CAD, and then 3d printed it.



The final design now looks like this:



And this is how the robot looks in reality:



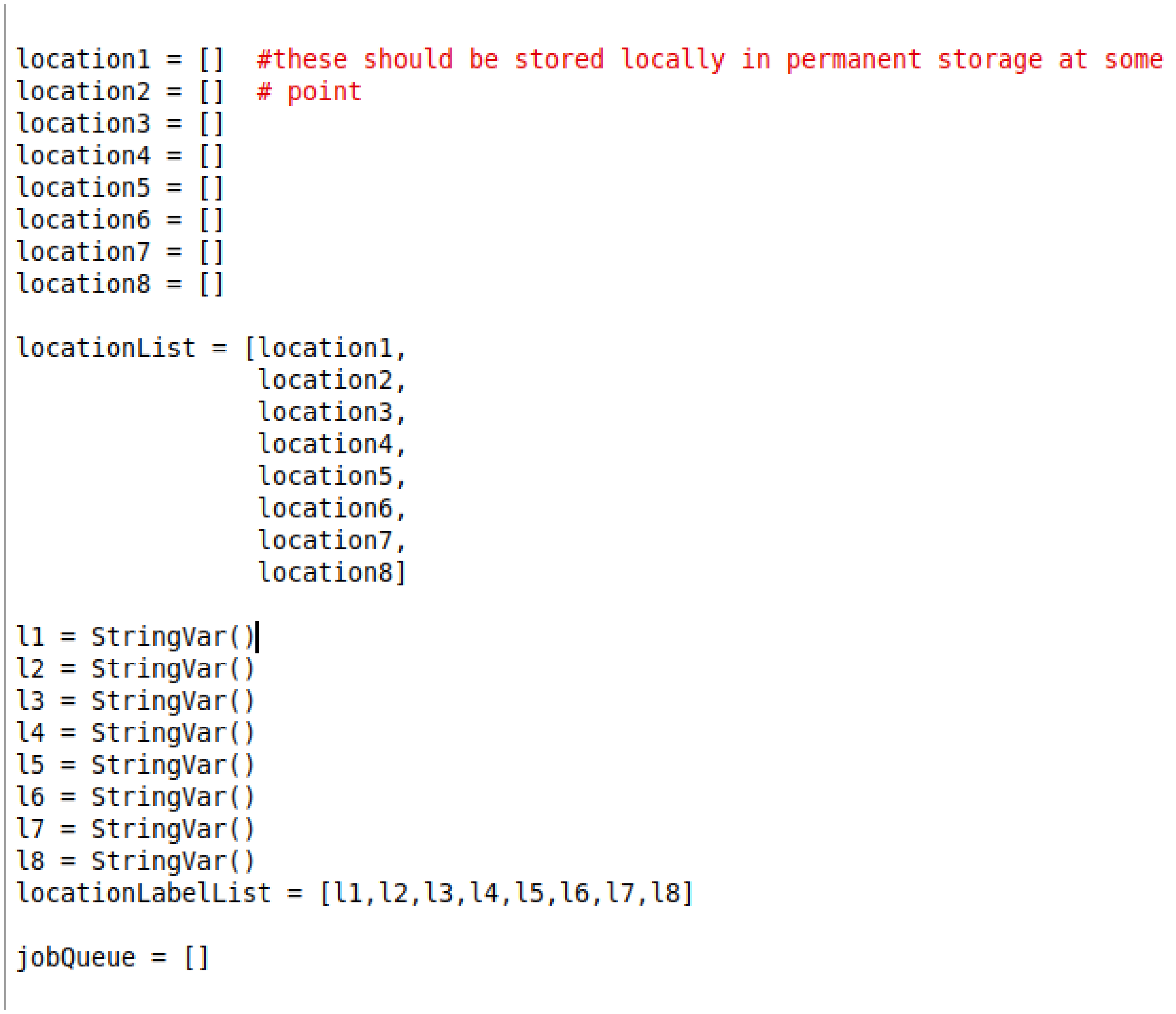
Development of UI buttons and layout

05/01/21: For the development of the UI, I used a python library called Tkinter. I don't use Tkinter often, so online resources such as https://stackoverflow.com/, and https://docs.python.org/3/library/tkinter.html were very useful in figuring out the Tkinter's syntax, and any bugs that came up.

Because my current code is largely based on procedural and functional paradigms, it made sense to develop the UI in a similar manner. After the basic UI space is created, buttons are then positioned onto the space, and then the buttons, when pressed, can call predefined subroutines that deal with variable assignment, gathering of data, window ‘termination’ (I.e. closing a window automatically) and sometimes calling of other subroutines.

First, some variables are set up to facilitate the saving of locations to specific buttons, and the job queue is set up. NOTE: The variables l1, l2, etc, are defined as StringVar(), which is a datatype used in tkinter that can be ‘set()’ and ‘get()’ at any point in the program. L1, l2, etc are the labels written on each of the set-location buttons.

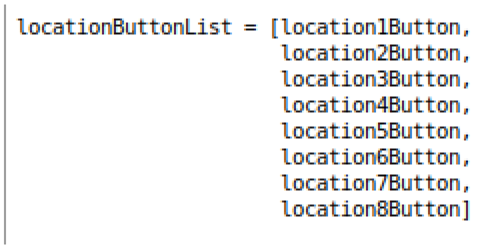
The locations in locationList will be fetched from permanent storage at some point, but for developing the UI, they are just reset after each run of the program.



```
location1 = []  #these should be stored locally in permanent storage at some
location2 = []  # point
location3 = []
location4 = []
location5 = []
location6 = []
location7 = []
location8 = []

locationList = [location1,
                location2,
                location3,
                location4,
                location5,
                location6,
                location7,
                location8]

l1 = StringVar()
l2 = StringVar()
l3 = StringVar()
l4 = StringVar()
l5 = StringVar()
l6 = StringVar()
l7 = StringVar()
l8 = StringVar()
locationLabelList = [l1,l2,l3,l4,l5,l6,l7,l8]

jobQueue = []
```

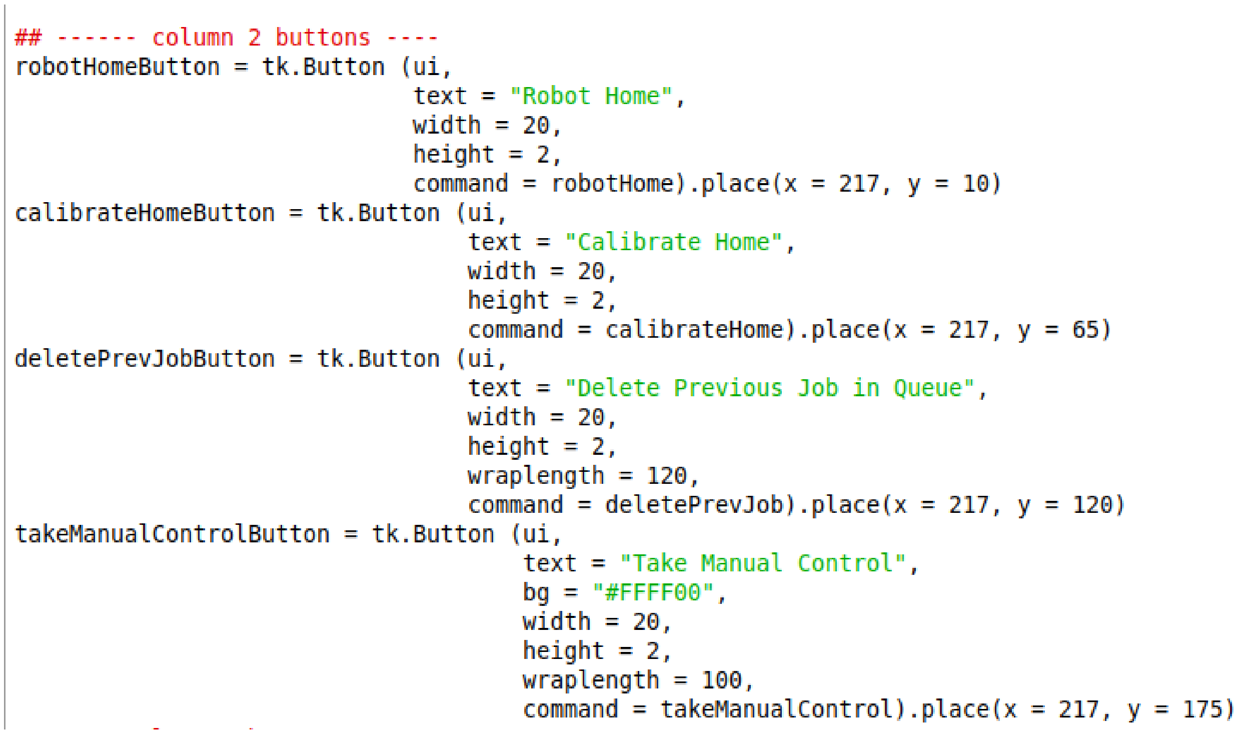
The 8 location buttons are set up in the following manner, they are then ‘placed’ on the UI space according to their coordinates.

```
## ------ column 1 buttons ----
location1Button = Button (ui,
                             text = l1.get(),
                             width = 20,
                             height = 2,              # These are INDEXES (down)
                             command = lambda: locationAddToJobQueue(0))
location1Button.place(x = 10, y = 10)

location2Button = Button (ui,
                             text = l2.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(1))
location2Button.place(x = 10, y = 65)

location3Button = Button (ui,
                             text = l3.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(2))
location3Button.place(x = 10, y = 120)

location4Button = Button (ui,
                             text = l4.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(3))
location4Button.place(x = 10, y = 175)

location5Button = Button (ui,
                             text = l5.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(4))
location5Button.place(x = 10, y = 230)

location6Button = Button (ui,
                             text = l6.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(5))
location6Button.place(x = 10, y = 285)

location7Button = Button (ui,
                             text = l7.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(6))
location7Button.place(x = 10, y = 340)

location8Button = Button (ui,
                             text = l8.get(),
                             width = 20,
                             height = 2,
                             command = lambda: locationAddToJobQueue(7))
location8Button.place(x = 10, y = 395)
```



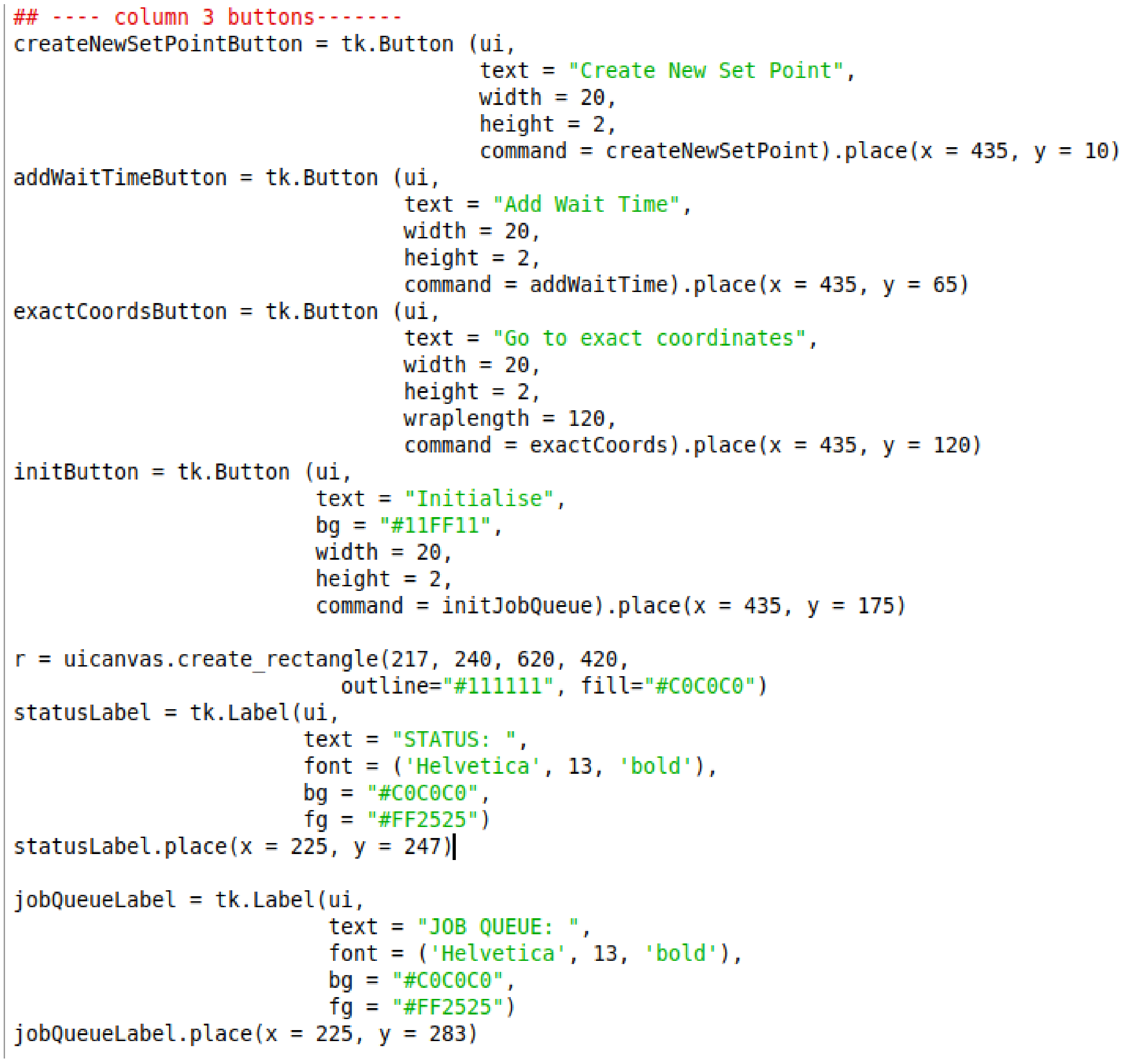
```
locationButtonList = [location1Button,
                      location2Button,
                      location3Button,
                      location4Button,
                      location5Button,
                      location6Button,
                      location7Button,
                      location8Button]
```

The functions for the second column of buttons are set up in a similar manner, with the command argument being set to a subroutine **reference** (without brackets) so that when the button is pressed, the subroutine is called.



```
## ------ column 2 buttons ----
robotHomeButton = tk.Button (ui,
                             text = "Robot Home",
                             width = 20,
                             height = 2,
                             command = robotHome).place(x = 217, y = 10)
calibrateHomeButton = tk.Button (ui,
                                 text = "Calibrate Home",
                                 width = 20,
                                 height = 2,
                                 command = calibrateHome).place(x = 217, y = 65)
deletePrevJobButton = tk.Button (ui,
                                 text = "Delete Previous Job in Queue",
                                 width = 20,
                                 height = 2,
                                 wraplength = 120,
                                 command = deletePrevJob).place(x = 217, y = 120)
takeManualControlButton = tk.Button (ui,
                                     text = "Take Manual Control",
                                     bg = "#FFFF00",
                                     width = 20,
                                     height = 2,
                                     wraplength = 100,
                                     command = takeManualControl).place(x = 217, y = 175)
```

The column 3 buttons are set up similarly, and the status bar rectangle is created along two with permanent labels, ‘STATUS’ and ‘JOB QUEUE’:

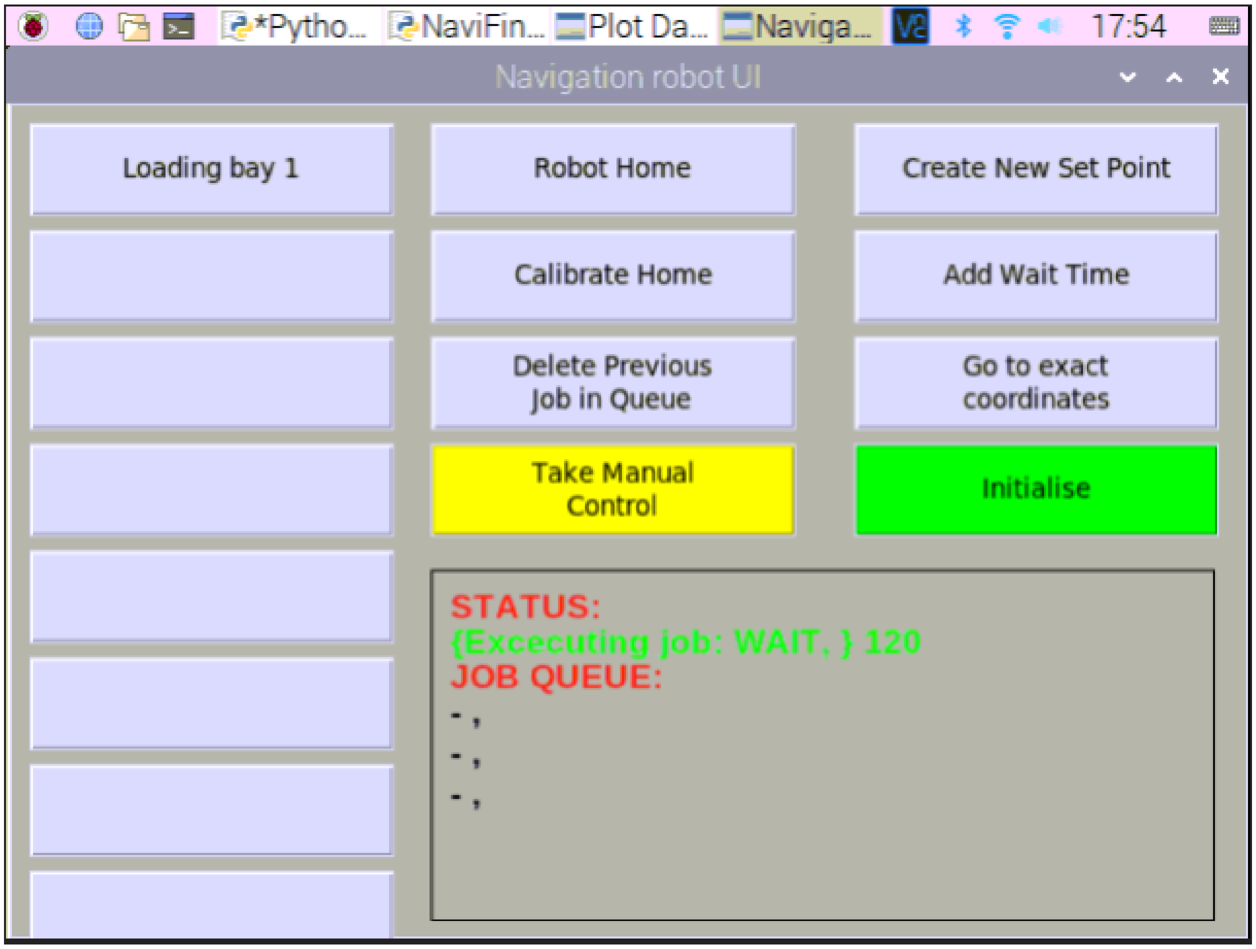


```
## ---- column 3 buttons-------
createNewSetPointButton = tk.Button (ui,
                                     text = "Create New Set Point",
                                     width = 20,
                                     height = 2,
                                     command = createNewSetPoint).place(x = 435, y = 10)
addWaitTimeButton = tk.Button (ui,
                               text = "Add Wait Time",
                               width = 20,
                               height = 2,
                               command = addWaitTime).place(x = 435, y = 65)
exactCoordsButton = tk.Button (ui,
                               text = "Go to exact coordinates",
                               width = 20,
                               height = 2,
                               wraplength = 120,
                               command = exactCoords).place(x = 435, y = 120)
initButton = tk.Button (ui,
                        text = "Initialise",
                        bg = "#11FF11",
                        width = 20,
                        height = 2,
                        command = initJobQueue).place(x = 435, y = 175)

r = uicanvas.create_rectangle(217, 240, 620, 420,
                          outline="#111111", fill="#C0C0C0")
statusLabel = tk.Label(ui,
                       text = "STATUS: ",
                       font = ('Helvetica', 13, 'bold'),
                       bg = "#C0C0C0",
                       fg = "#FF2525")
statusLabel.place(x = 225, y = 247)

jobQueueLabel = tk.Label(ui,
                         text = "JOB QUEUE: ",
                         font = ('Helvetica', 13, 'bold'),
                         bg = "#C0C0C0",
                         fg = "#FF2525")
jobQueueLabel.place(x = 225, y = 283)
```

The subroutine jobQueueStatusUpdate() is for updating the information displayed on the status bar, which is the current job, (I.e. the first job on the job queue) and the subsequent jobs. Try and except catches are used because sometimes, the queue is not long enough for all display spaces to be filled. In the case that any display spaces aren't filled, they display an empty string ("" : The default value for StringVar()). This subroutine is called any time the job queue is updated.

```
def jobQueueStatusUpdate(): # this is for updating the status bar information
    a1 = StringVar()
    a2 = StringVar()
    a3 = StringVar()
    a4 = StringVar()
    a5 = StringVar()
    a6 = StringVar()
    a7 = StringVar()
    a8 = StringVar()
    l = [a1, a2, a3, a4, a5, a6, a7, a8]
    if len(jobQueue) == 0:
        pass
    else:
        counter = 0
        for u in jobQueue[0:4]:
            try: # i have to do try, exécept because if there is no e.g. jobqueue[1] then it will throw an error.
                l[counter].set(u[0])
                l[counter+1].set(u[1])
                counter += 2
            except:
                continue
    excecJobLabel = tk.Label(ui,
                             text = ("Excecuting job: " + str(a1.get()) + ", ", str(a2.get())),
                             font = ('Helvetica', 13, 'bold'),
                             bg = "#C0C0C0",
                             fg = "#11FF11")
    excecJobLabel.place(x = 225, y = 265)
    queueLabel1 = tk.Label(ui,
                           text = ("- " + str(a3.get()) + ", " + str(a4.get())),
                           font = ('Helvetica', 13, 'bold'),
                           bg = "#C0C0C0",
                           fg = "#101010")
    queueLabel1.place(x = 225, y = 301)
    queueLabel2 = tk.Label(ui,
                           text = ("- " + str(a5.get()) + ", " + str(a6.get())),
                           font = ('Helvetica', 13, 'bold'),
                           bg = "#C0C0C0",
                           fg = "#101010")
    queueLabel2.place(x = 225, y = 322)
    queueLabel3 = tk.Label(ui,
                           text = ("- " + str(a7.get()) + ", " + str(a8.get())),
                           font = ('Helvetica', 13, 'bold'),
                           bg = "#C0C0C0",
                           fg = "#101010")
    queueLabel3.place(x = 225, y = 343)
    ui.update()
```

The UI now looks like this: (with a single WAIT job added).



```
STATUS:
{Excecuting job: WAIT, } 120
JOB QUEUE:
- ,
- ,
- ,
```

Development of logic behind UI.

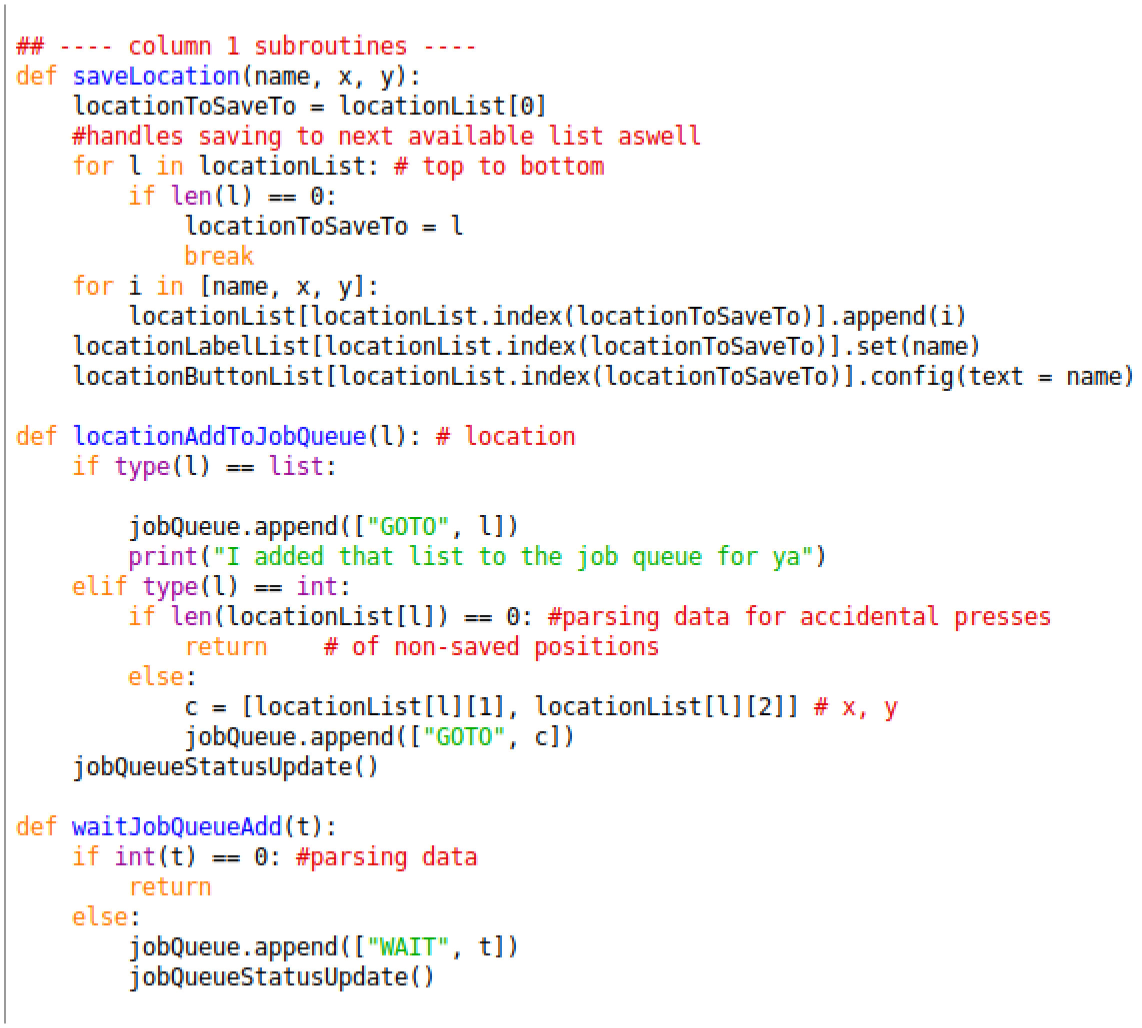
05/01/21: The logic behind the UI is mostly the subroutines called when a button is pressed, but some deal with things like adding to the job queue, initialising the job queue etc.

NOTE: not all of these are in 'column 1' per se, but they are used in a variety of places so are placed at the top for ease of access during development.

saveLocation is for saving new locations to the 8 location hotkeys. This also handles label assignment, so that the screen updates in real time.

locationAddToJobQueue saves a location to the job queue either coming from the hotkey buttons, go to exact coordinates, or go to robot home commands. The job queue status bar is updated at the end of this, so that the user can see the jobs being added in real time.

waitJobQueueAdd adds a WAIT job to the queue and omits it if the time to wait is 0 seconds.



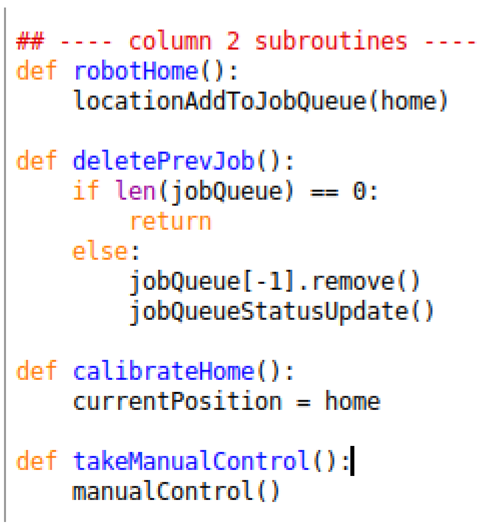
```python
## ---- column 1 subroutines ----
def saveLocation(name, x, y):
    locationToSaveTo = locationList[0]
    #handles saving to next available list aswell
    for l in locationList: # top to bottom
        if len(l) == 0:
            locationToSaveTo = l
            break
    for i in [name, x, y]:
        locationList[locationList.index(locationToSaveTo)].append(i)
    locationLabelList[locationList.index(locationToSaveTo)].set(name)
    locationButtonList[locationList.index(locationToSaveTo)].config(text = name)

def locationAddToJobQueue(l): # location
    if type(l) == list:

        jobQueue.append(["GOTO", l])
        print("I added that list to the job queue for ya")
    elif type(l) == int:
        if len(locationList[l]) == 0: #parsing data for accidental presses
            return    # of non-saved positions
        else:
            c = [locationList[l][1], locationList[l][2]] # x, y
            jobQueue.append(["GOTO", c])
    jobQueueStatusUpdate()

def waitJobQueueAdd(t):
    if int(t) == 0: #parsing data
        return
    else:
        jobQueue.append(["WAIT", t])
        jobQueueStatusUpdate()
```

robotHome adds the home location to the job queue.

deleteJobQueue deletes the most recently added job from the job queue.

calibrateHome sets the current position to the home position, for calibration purposes.

takeManualControl allows the user to manually control the robot. NOTE: this needs fixing in a later update to allow the user to exit manual control. **UPDATE: 12/02/21** This was fixed later in sprint 5:

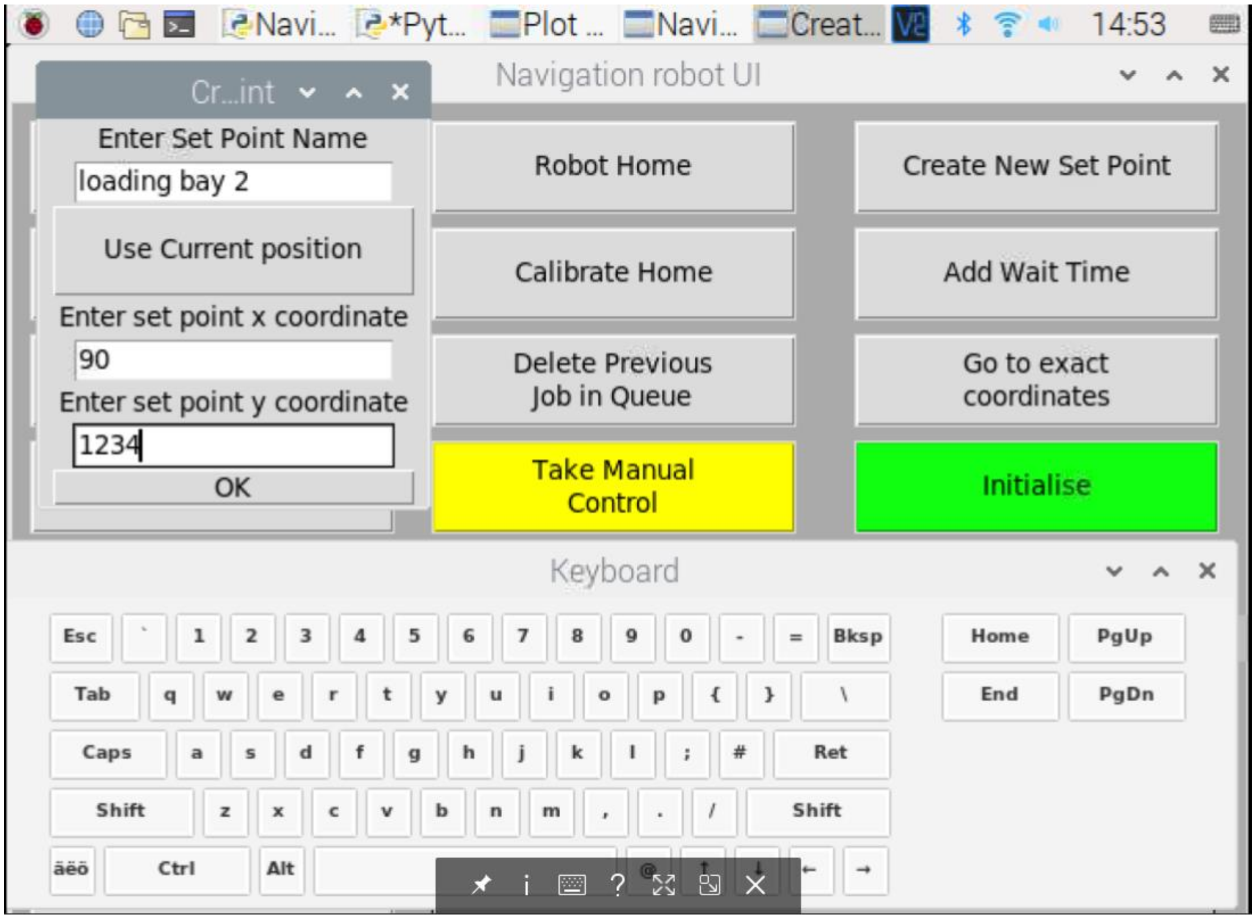


```
## ---- column 2 subroutines ----
def robotHome():
    locationAddToJobQueue(home)

def deletePrevJob():
    if len(jobQueue) == 0:
        return
    else:
        jobQueue[-1].remove()
        jobQueueStatusUpdate()

def calibrateHome():
    currentPosition = home

def takeManualControl():
    manualControl()
```

createNewSetPoint is the subroutine that is called when the “Create new set point” button is pressed. It opens a new window and allows the user to a name and enter specific coordinates as well as allowing the current position to be used as set point coordinates. When the “OK” button is pressed, the terminate subroutine calls saveLocation and closes the window. The coordinates can be entered via a touchscreen keyboard I have installed on the raspberry pi. This is hidden in the corner until pressed. When it opens allows full keyboard functionality to allow for both the set point name and coordinates to be entered.

```
## ---- column 3 subroutines ---
def createNewSetPoint():

    xVal = DoubleVar()
    yVal = DoubleVar()
    nameVal = StringVar()
    cnsp = tk.Toplevel(ui)
    cnspFrame = tk.Frame(cnsp)
    cnsp.title("Create new set point")
    cnsp.geometry("200x200")
    lname = tk.Label (cnsp,
                    text = "Enter Set Point Name").pack()
    nameEntry = tk.Entry(cnsp,
                       textvariable = nameVal).pack()
    def plugCurrentCoords():
        xVal.set(currentPosition[0])
        yVal.set(currentPosition[1])
    ccb = tk.Button(cnsp,   # current coordinates button
                    text = "Use Current position",
                    width = 20,
                    height = 2,
                    command = plugCurrentCoords).pack()
    lx = tk.Label (cnsp,
                 text = "Enter set point x coordinate").pack()
    xEntry = tk.Entry(cnsp,
                    textvariable = xVal).pack()
    ly = tk.Label (cnsp,
                 text = "Enter set point y coordinate").pack()
    yEntry = tk.Entry(cnsp,
                    textvariable = yVal).pack()
    def terminate():
        name = nameVal.get()
        x = xVal.get()
        y = yVal.get()
        print(name, x, y)
        saveLocation(name, float(x), float(y))
        cnsp.destroy()
    B = tk.Button (cnsp,
                 text = "OK",
                 width = 20,
                 height = 2,
                 command = terminate).pack()
    return
```

This is the window that pops up when the create new set point button is pressed, with the keyboard shown as well.



Enter Set Point Name: loading bay 2

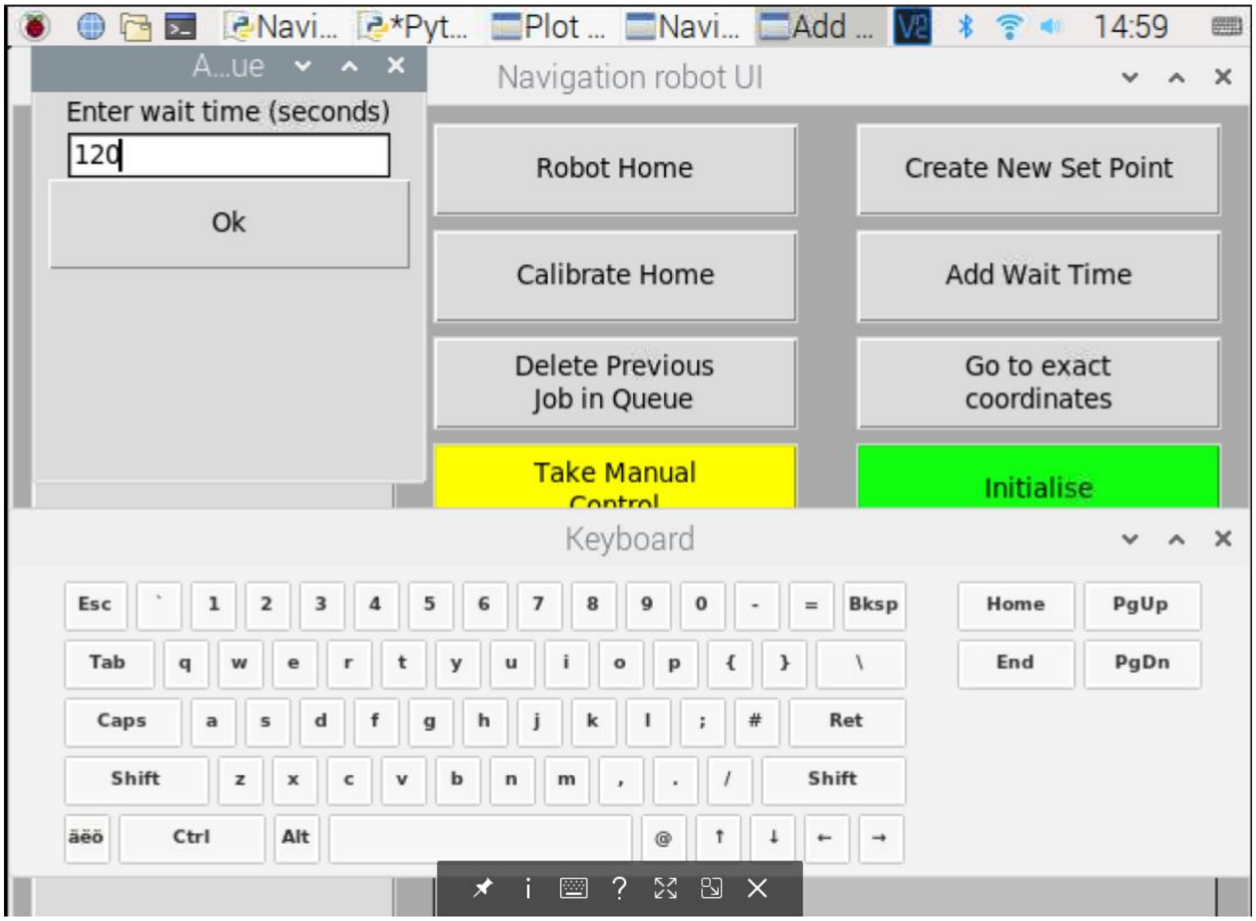
Enter set point x coordinate: 90

Enter set point y coordinate: 1234

addWaitTime is like createNewSetPoint in the fact that it opens a new window (later to be closed by the terminate subroutine). It allows the user to enter a wait time and have it immediately to the job queue. In terminate, it calls waitJobQueueAdd, described above.

```
def addWaitTime():
    tVal = StringVar()
    awt = tk.Toplevel(ui)
    awtFrame = tk.Frame(ui)
    awt.title("Add Wait Time To Job Queue")
    awt.geometry("200x200")
    l = Label (awt,
                  text = "Enter wait time (seconds)").pack()
    e = Entry (awt,
                  textvariable = tVal).pack()
    def terminate1():
        t = tVal.get()
        t = int(t)
        print(t)
        waitJobQueueAdd(t)
        awt.destroy()
    b = tk.Button (awt,
                   text = "Ok",
                   width = 20,
                   height = 2,
                   command = terminate1).pack()
```

This is the window that shows up when the “Add Wait Time” button was pressed, with the pop-up keyboard shown as well.

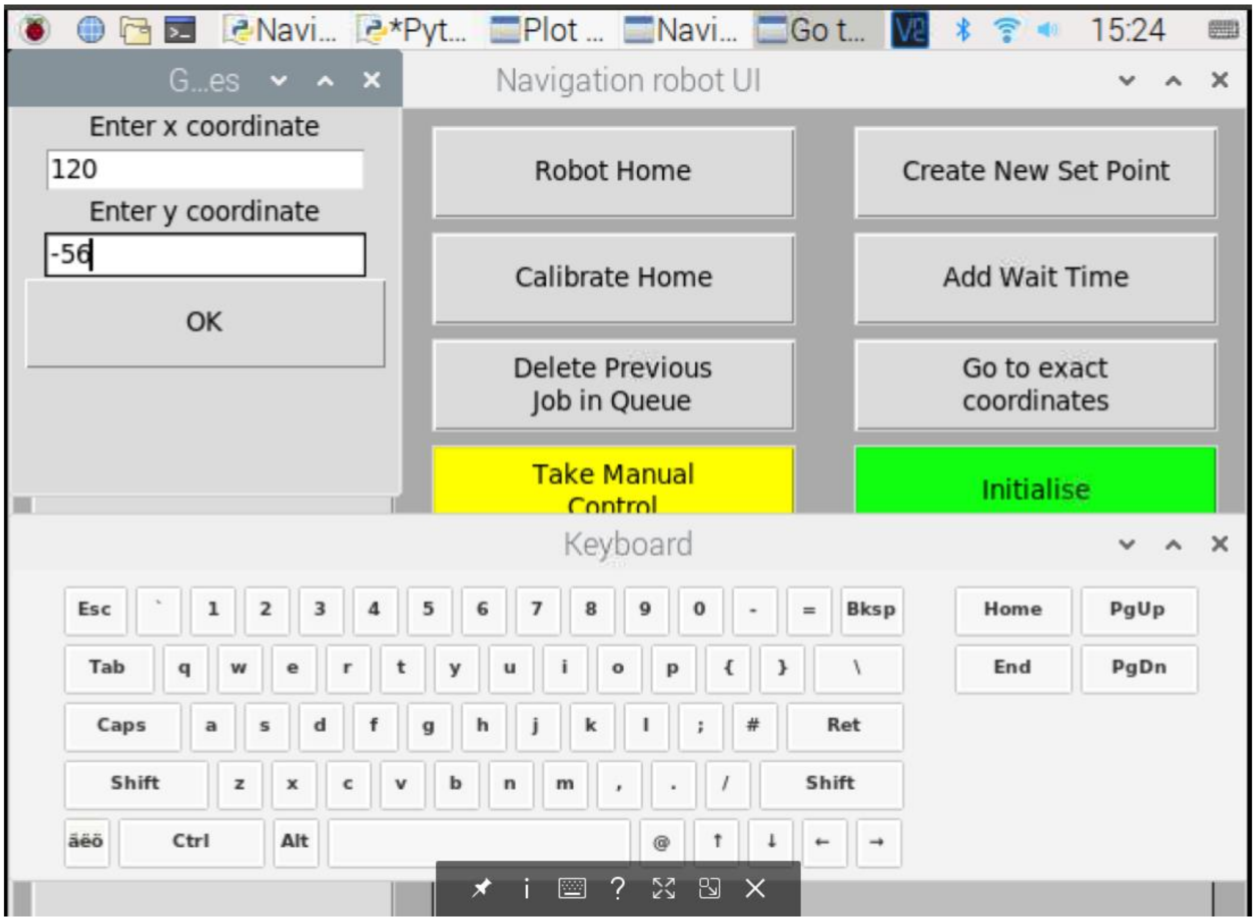


Enter wait time (seconds)
120

exactCoords is similar in function to createNewSetPoint, but instead of creating and saving a new set point, it adds the coordinates to the job queue straightaway.

```python
def exactCoords():
    xVal = StringVar()
    yVal = StringVar()
    ec = tk.Toplevel(ui)
    ecFrame = tk.Frame(ec)
    ec.title("Go to exact coordinates")
    ec.geometry("200x200")
    lx = Label (ec,
                text = "Enter x coordinate").pack()
    xEntry = Entry(ec,
                   textvariable = xVal).pack()
    ly = Label (ec,
                text = "Enter y coordinate").pack()
    yEntry = Entry(ec,
                   textvariable = yVal).pack()
    def terminate():
        x = int(xVal.get())
        y = int(yVal.get())
        print(x, y)
        locationAddToJobQueue([x, y])
        ec.destroy()
    B = Button (ec,
                text = "OK",
                width = 20,
                height = 2,
                command = terminate).pack()
```

This is the window that pops up when the “Create New Set Point” button is pressed, with the keyboard shown.



Enter x coordinate

120

Enter y coordinate

-56

OK

initJobQueue initiates the job queue, it cycles through each job, determines whether it’s a “GOTO” job, or “WAIT” job, and then executes each one accordingly.

```
def initJobQueue():
    if len(jobQueue) == 0:
        return
    else:
        for job in jobQueue:
            if job[0] == "GOTO":
                print(jobQueue)
                navToGoal(job[1])
                sleep(1)
                jobQueue.pop(0)

            elif job[0] == "WAIT":
                print(jobQueue)
                print("Sleeping for %s seconds..." % (job[1]))
                sleep(job[1])
                jobQueue.pop(0)
            jobQueueStatusUpdate()
```

Sprint 4: 02/02/21 - 06/02/21 - Complete

Work to be developed: Product backlog No. 6:

- Definition of currentPosition to be determined at the wheel control level. Dynamic obstacle handling should be integrated directly into the control functions. – Complete.

Evidence:

Definition of currentPosition to be determined at the wheel control level. Dynamic obstacle handling should be integrated directly into the control functions.

02/02/21: First, for the robot to be able to stop at any time, and then restart on its trajectory after a dynamic response, the current position must be determined by how far the wheels have travelled, not a pre-set amount. This is also necessary for the robot to keep track of its position when in manual control mode.

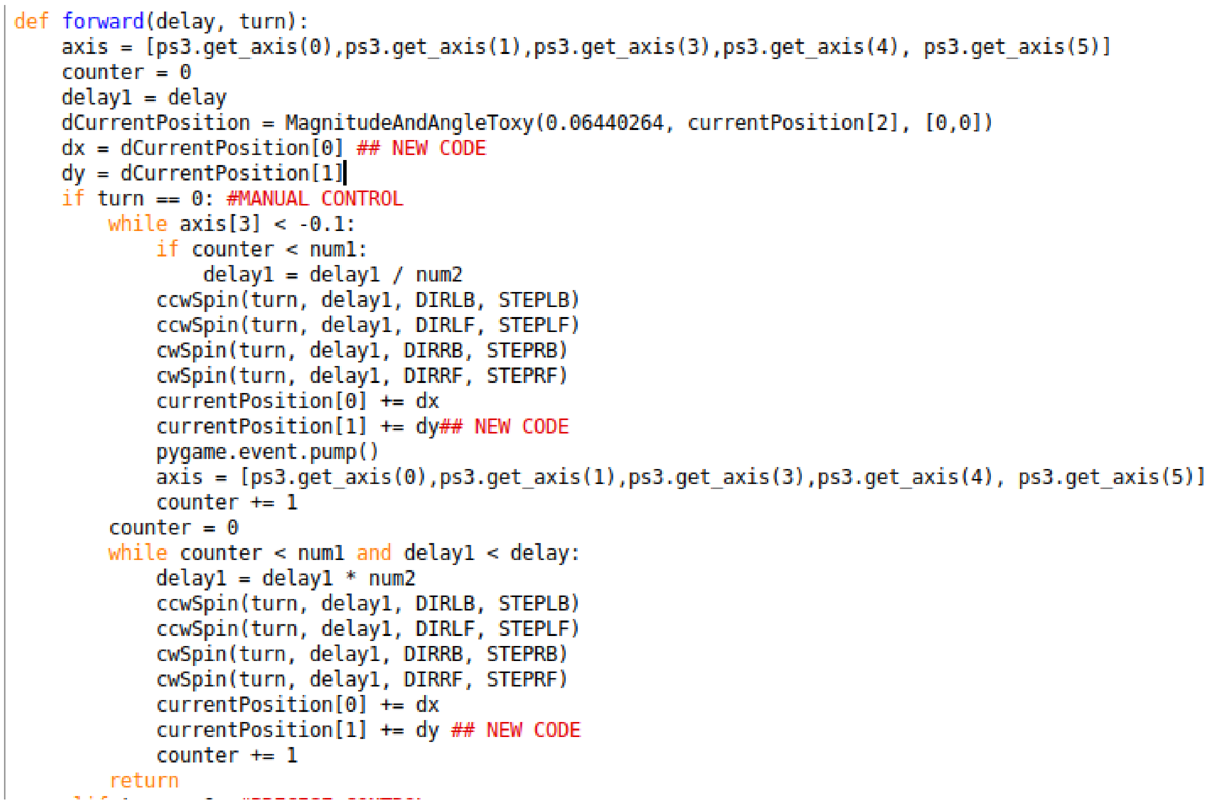
The motors are running in half-step mode therefore, because there are 200 full steps per revolution, the program is running 400 half steps to do a full revolution. The diameter of the wheel is 8.2 cm. 360/400 = 0.9 degrees, (0.9/360) * π * 8.2 = 0.06440264 cm. Therefore, the robot is advancing about 0.064cm per half step.

We can feed this value (0.06440264) into our MagnitudeAndAngleToxy function to calculate the new positions of x and y for every half step, or (to improve efficiency and not introduce any time delay (which would quite literally slow the robot down if placed into the stepping loop)) we can calculate the change in position per half for the current situation, just before we enter the loop, and then just *change* the currentPosition values inside the loop. This is far more efficient. NOTE: this will also apply to the spinLeft and spinRight subroutines, but with angles instead of x and y translation.

The necessary changes to the control functions are as follows:

- currentPosition to be changed when in manual control mode for all control functions (forward, backward, spinLeft and spinRight)
- currentPosition to be changed when in precise control mode too, but for the **forward** subroutine, the program must follow the dynamic obstacle protocol designed before.
- All other subroutines should have any additions to the current position removed.

The control functions are changed to include a calculation of the *change in current position per half step* for each time the function is called, split into *change in x* (dx) and *change in y* (dy). This needs to be recalculated every time the robot goes forward because the angle could have changed. Now for each half step, the current position is changed by the amount it needs to. Below shows the modified code for the manual control part of the forward function (for ramping up, constant speed and ramping down), but the *updating the current position* is repeated for the precise control part and the two parts of the backwards function as well.

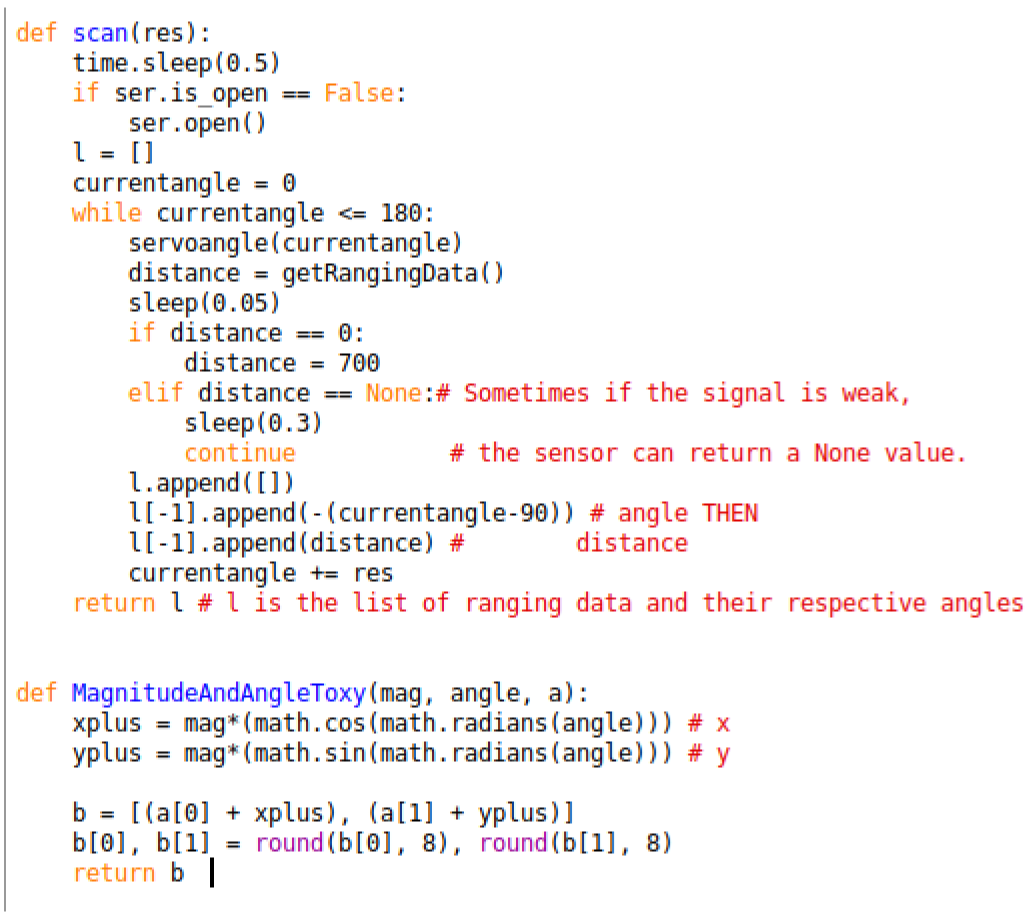


```
def forward(delay, turn):
    axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
    counter = 0
    delay1 = delay
    dCurrentPosition = MagnitudeAndAngleToxy(0.06440264, currentPosition[2], [0,0])
    dx = dCurrentPosition[0] ## NEW CODE
    dy = dCurrentPosition[1]
    if turn == 0: #MANUAL CONTROL
        while axis[3] < -0.1:
            if counter < num1:
                delay1 = delay1 / num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            currentPosition[0] += dx
            currentPosition[1] += dy## NEW CODE
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            currentPosition[0] += dx
            currentPosition[1] += dy ## NEW CODE
            counter += 1
        return
```

Similarly, the spinLeft (and spinRight) functions update the respective angle of the robot every half step. The angle is updated every time the robot spins, even if it's just a single step. (da is the *change in angle per half step*, but does not need to be calculated each time because it is constant no matter the situation (0.2125 degrees))

```
def spinLeft(delay, turn):
    axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
    counter = 0
    delay1 = delay
    da = 0.2125 #degrees
    if turn == 0:
        while axis[2] < -0.1:
            if counter < num1:
                delay1 = delay1 / num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            currentPosition[2] += da
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay:
            delay1 = delay1 * num2
            cwSpin(turn, delay1, DIRLB, STEPLB)
            cwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            currentPosition[2] += da
            counter += 1
        return
```

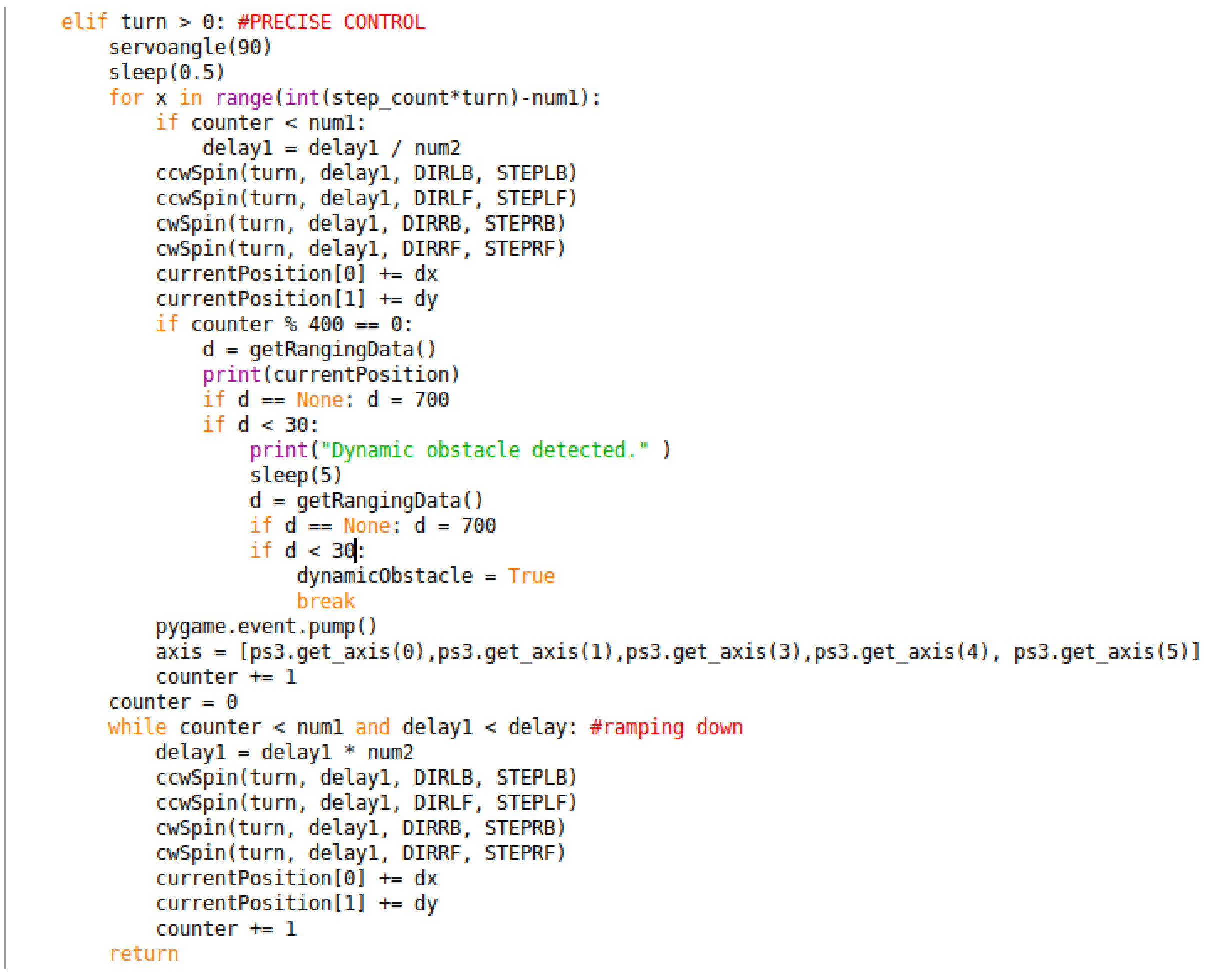
Now that I am using the MagnitudeAndAngleToxy function for more than just calculating node positions relative to the robot, I had to change it so that it is accurate all the time, and does not include any angle or position offsets (which was previously needed to calculate the node data e.g. angle += -90). Now, the scan function handles the angle and position offsets, and the MagnitudeAndAngleToxy function is accurate no matter what. This makes for cleaner and more reusable code.



```
def scan(res):
    time.sleep(0.5)
    if ser.is_open == False:
        ser.open()
    l = []
    currentangle = 0
    while currentangle <= 180:
        servoangle(currentangle)
        distance = getRangingData()
        sleep(0.05)
        if distance == 0:
            distance = 700
        elif distance == None:# Sometimes if the signal is weak,
            sleep(0.3)
            continue           # the sensor can return a None value.
        l.append([])
        l[-1].append(-(currentangle-90)) # angle THEN
        l[-1].append(distance) #        distance
        currentangle += res
    return l # l is the list of ranging data and their respective angles


def MagnitudeAndAngleToxy(mag, angle, a):
    xplus = mag*(math.cos(math.radians(angle))) # x
    yplus = mag*(math.sin(math.radians(angle))) # y

    b = [(a[0] + xplus), (a[1] + yplus)]
    b[0], b[1] = round(b[0], 8), round(b[1], 8)
    return b
```

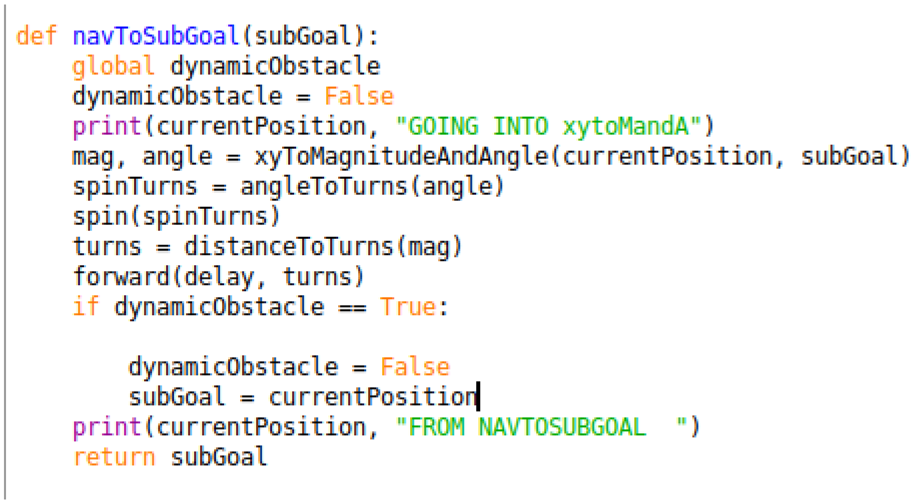
In the subroutine *forward*, in the precise control section, the robot now scans every 400 half steps (which Is a full rotation of the wheel) and checks for an obstacle closer than 15cm. If there is one, it sleeps for 5 seconds, before determining if the obstacle has moved or not. If it hasn't, it sets a global variable, (declared as global at the top of the program) called dynamicObstacle to True and breaks from the loop, and returns to the function navToSubGoal.

UPDATE 12/02/21: The distance that the dynamic obstacle should be in is now 30cm, because the circumference of the wheel is about 26cm, and so the scan range needs to be larger than that (because it checks for a dynamic obstacle every 400 half steps (I.e. when the robot has travelled a circumference length). This way the robot won't crash into the dynamic object. The safety systems now work well, if I stand in the path of the robot, it doesn't crash into me.



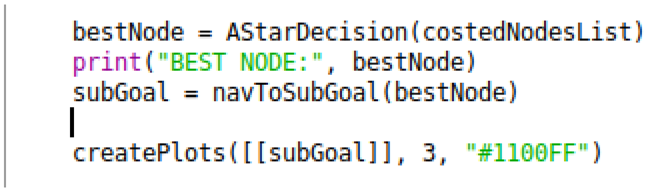
```
    elif turn > 0: #PRECISE CONTROL
        servoangle(90)
        sleep(0.5)
        for x in range(int(step_count*turn)-num1):
            if counter < num1:
                delay1 = delay1 / num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            currentPosition[0] += dx
            currentPosition[1] += dy
            if counter % 400 == 0:
                d = getRangingData()
                print(currentPosition)
                if d == None: d = 700
                if d < 30:
                    print("Dynamic obstacle detected." )
                    sleep(5)
                    d = getRangingData()
                    if d == None: d = 700
                    if d < 30:
                        dynamicObstacle = True
                        break
            pygame.event.pump()
            axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
            counter += 1
        counter = 0
        while counter < num1 and delay1 < delay: #ramping down
            delay1 = delay1 * num2
            ccwSpin(turn, delay1, DIRLB, STEPLB)
            ccwSpin(turn, delay1, DIRLF, STEPLF)
            cwSpin(turn, delay1, DIRRB, STEPRB)
            cwSpin(turn, delay1, DIRRF, STEPRF)
            currentPosition[0] += dx
            currentPosition[1] += dy
            counter += 1
        return
```

The function navToSubGoal now looks like this. It first declares that the variable dynamicObstacle it is referring to is the global one (if this was not declared, the program treats the variable assignment in the if statement as a local variable, and then throws a '*use of variable before assignment*' error). After the subroutine forward is called, an if statement is run immediately to a) reset the global variable dynamicObstacle to False, and set the subGoal to the current position, as the place it has stopped has now become a new subgoal and is subsequently treated like one. The parameter subGoal is now returned, which is explained below.



```
def navToSubGoal(subGoal):
    global dynamicObstacle
    dynamicObstacle = False
    print(currentPosition, "GOING INTO xytoMandA")
    mag, angle = xyToMagnitudeAndAngle(currentPosition, subGoal)
    spinTurns = angleToTurns(angle)
    spin(spinTurns)
    turns = distanceToTurns(mag)
    forward(delay, turns)
    if dynamicObstacle == True:

        dynamicObstacle = False
        subGoal = currentPosition
    print(currentPosition, "FROM NAVTOSUBGOAL  ")
    return subGoal
```

The reason that subgoal is now returned, is so that in the event of a dynamic obstacle, the subGoal can be a) plotted on the graph of nodes (again, this is mainly for development/testing/debugging purposes) and b) in the event of any future debugging that may be necessary, it is useful to have an accurate record of when the robot stopped (I.e., a subGoal).



```
    bestNode = AStarDecision(costedNodesList)
    print("BEST NODE:", bestNode)
    subGoal = navToSubGoal(bestNode)

    createPlots([[subGoal]], 3, "#1100FF")
```

NOTE: all changes to the currentPosition variable are now done *solely* in the control functions, (with the exception of the 'calibrateHome' function which redefines the currentPosition as the home position) Now, the currentPosition should only change when the wheels move, and the currentPosition is changed in real time (and so is more accurate).

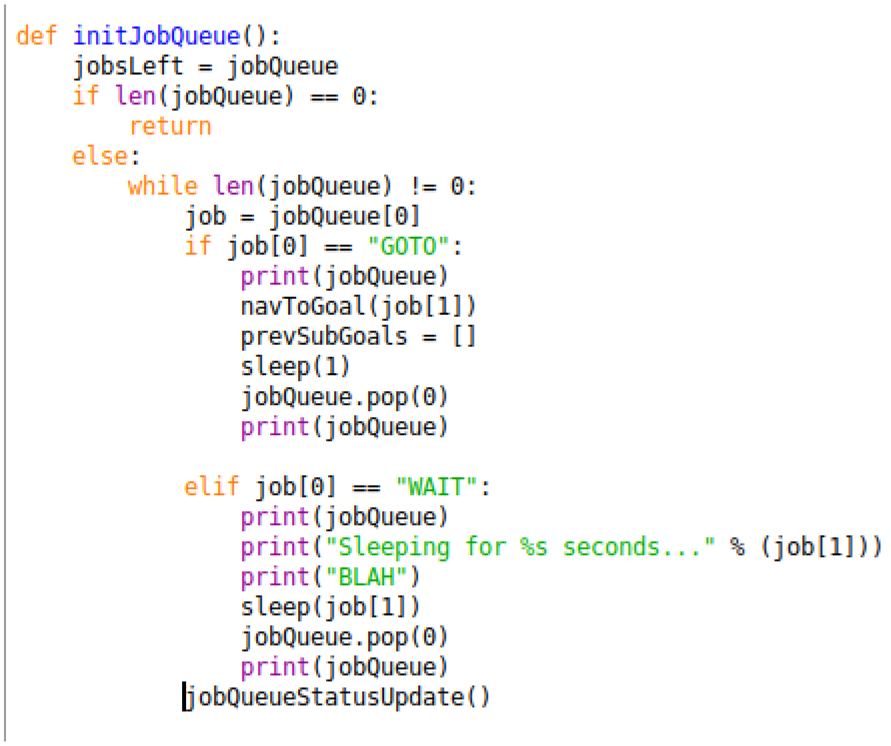
Sprint 5: 07/02/21 - 13/02/21 - Complete

Work to be developed:

- Bugs and performance issues should be fixed, as a result of informative testing. - Complete.

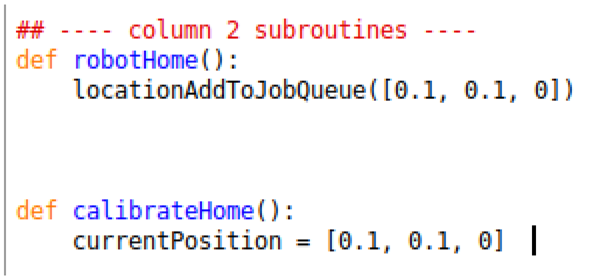
07/02/21: When I began testing the robot properly (I had tested while developing as well, but not extensively), I began to find several bugs that needed to be fixed for the robot to operate with any degree of usability.

First, the initJobQueue function, which I thought was working, turned out to have a bug in its logic. The big was this: a for loop was used to advance through the list, but during the for loop, items get pop()-ed from the list, and so now the list is different. This caused the queue to a) not update properly (in fact when it tried to update, it produced a complete mess of letters and numbers, which I think is down to Tkinter's libraries not updating entire rows text, but rather the exact positions of the text being changed to (**UPDATE: 08/02/21** this is now fixed in jobQueueStatusUpdate, below.)) and b) the list did not advance properly. This was fixed by using a while loop instead of a for loop, and just always taking the first value in the queue as the job (and the subsequently popping it). The list now advances properly.



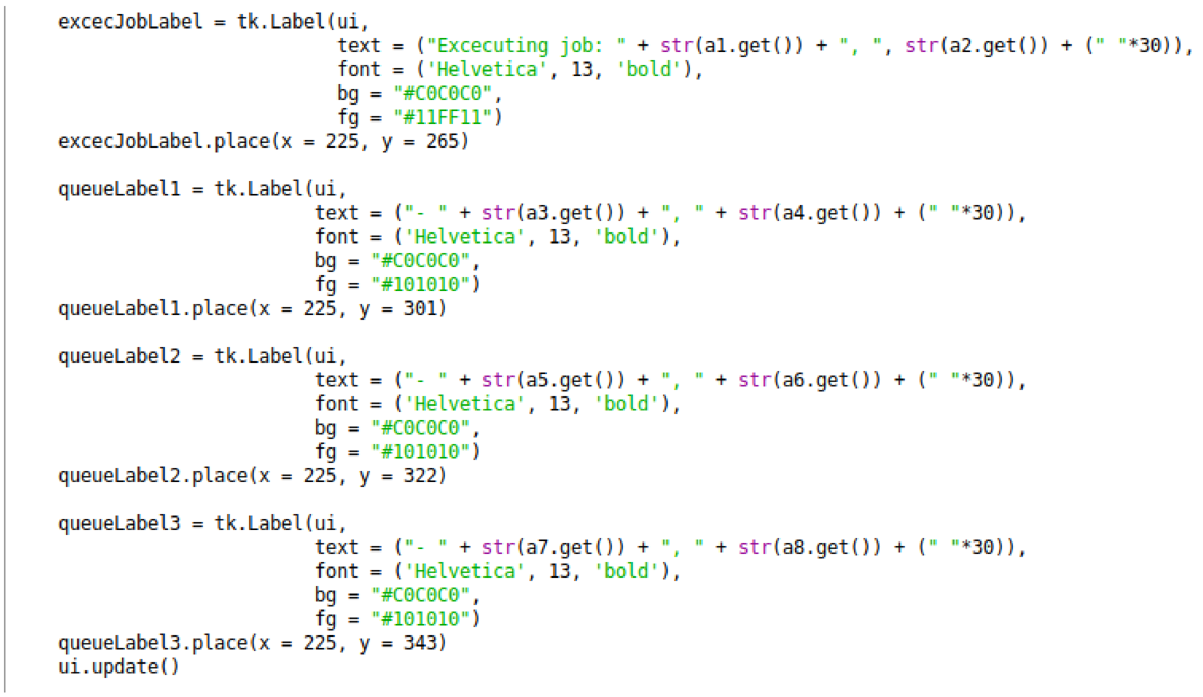
```
def initJobQueue():
    jobsLeft = jobQueue
    if len(jobQueue) == 0:
        return
    else:
        while len(jobQueue) != 0:
            job = jobQueue[0]
            if job[0] == "GOTO":
                print(jobQueue)
                navToGoal(job[1])
                prevSubGoals = []
                sleep(1)
                jobQueue.pop(0)
                print(jobQueue)

            elif job[0] == "WAIT":
                print(jobQueue)
                print("Sleeping for %s seconds..." % (job[1]))
                print("BLAH")
                sleep(job[1])
                jobQueue.pop(0)
                print(jobQueue)
            jobQueueStatusUpdate()
```

07/02/21 The functions robotHome and calibrateHome had to be fixed as well. For some reason (still unknown why, I may find out later), instead of passing the global variable *home* into the locationAddToJobQueue function, or setting the currentPosition to the global variable *home*, it would change the location data to the currentPosition every time the first job in the job queue was finished, not only on the job queue status bar, but also in the logic. Changing the *home* variable to [0.1, 0.1, 0] seems to have solved the problem, so its not an issue, but I am hoping I manage to figure out what caused this as it could be a mistake that I haven’t noticed yet.



```
## ---- column 2 subroutines ----
def robotHome():
    locationAddToJobQueue([0.1, 0.1, 0])



def calibrateHome():
    currentPosition = [0.1, 0.1, 0]
```

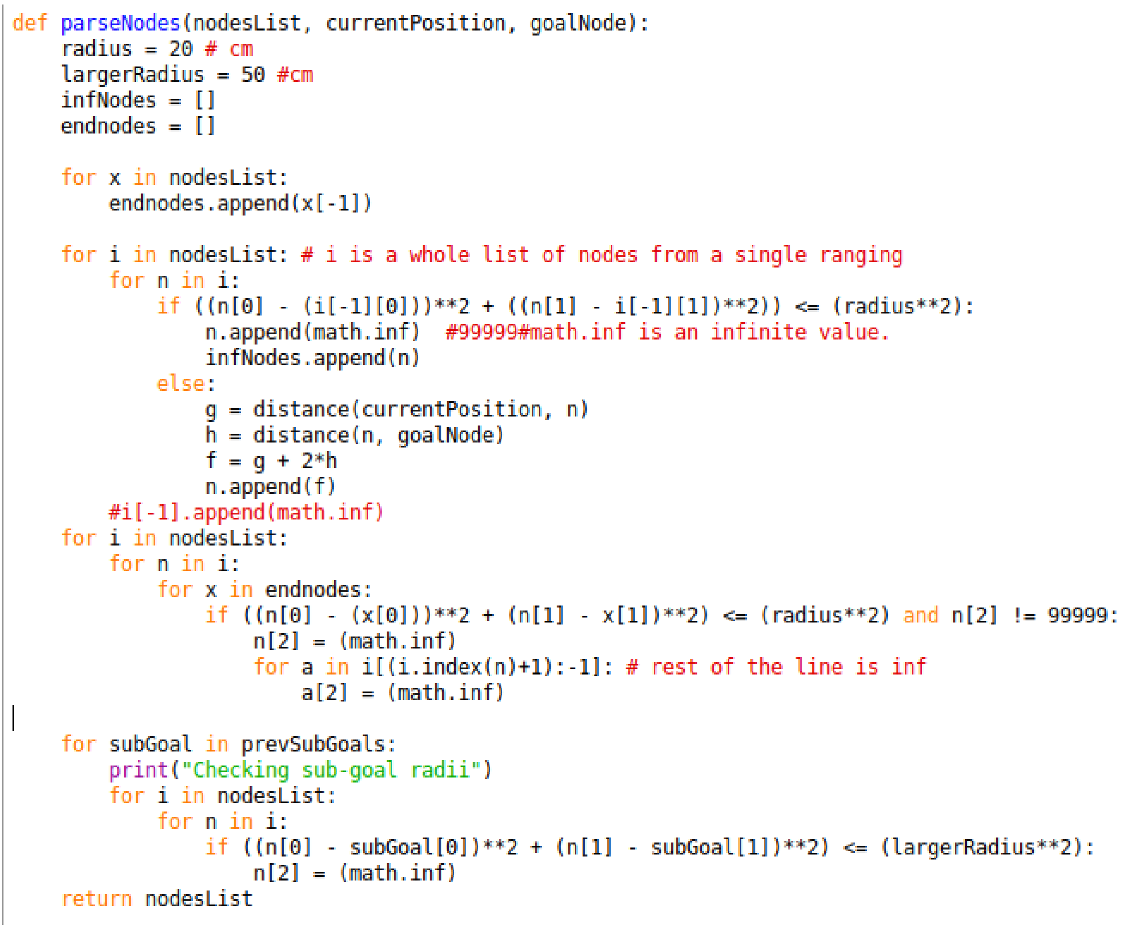
08/02/21: The issue described above with the function jobQueuestatusUpdate, when the jobs that were displayed were a mix of numbers and letters was easily fixed by adding (“ “*30) to the text parameter of the labels involved in displaying the status bar. (this also fixed issue I hadn’t noticed before, that when a job was deleted, it didn’t actually register on the status bar even though it had been removed from the jobQueue in logic) The status bar now works as excepted.



```
    excecJobLabel = tk.Label(ui,
                             text = ("Excecuting job: " + str(a1.get()) + ", ", str(a2.get()) + (" "*30)),
                             font = ('Helvetica', 13, 'bold'),
                             bg = "#C0C0C0",
                             fg = "#11FF11")
    excecJobLabel.place(x = 225, y = 265)

    queueLabel1 = tk.Label(ui,
                           text = ("- " + str(a3.get()) + ", " + str(a4.get()) + (" "*30)),
                           font = ('Helvetica', 13, 'bold'),
                           bg = "#C0C0C0",
                           fg = "#101010")
    queueLabel1.place(x = 225, y = 301)

    queueLabel2 = tk.Label(ui,
                           text = ("- " + str(a5.get()) + ", " + str(a6.get()) + (" "*30)),
                           font = ('Helvetica', 13, 'bold'),
                           bg = "#C0C0C0",
                           fg = "#101010")
    queueLabel2.place(x = 225, y = 322)

    queueLabel3 = tk.Label(ui,
                           text = ("- " + str(a7.get()) + ", " + str(a8.get()) + (" "*30)),
                           font = ('Helvetica', 13, 'bold'),
                           bg = "#C0C0C0",
                           fg = "#101010")
    queueLabel3.place(x = 225, y = 343)
    ui.update()
```

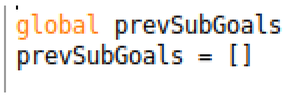
08/02/21: After some test runs, I found that if the goal was directly behind an obstacle. The robot would tend ‘hug’ the wall, meaning that it tended to go to sub-goals that were close the wall, and then find another sub-goal that was close to the wall and then reciprocate between these two, never going around the box. This was a problem because while the robot was executing the A* algorithm perfectly (those nodes near the wall were the best cost nodes as they were closest to both the robot’s position and the goal, but not so close as to be inside the radius of infinite cost), it was not actually making any progress. For this reason I had to add a clause in the parseNodes function and elsewhere, that keeps track of the *previous sub-goals for a journey*, and forms a radius of infinite cost around them, meaning that they cannot pick a sub-goal that was too close to the previous positions which **forces the robot to branch out** into space that it would not normally go to (due to the cost being higher) for example **around an object** that is in the direct path of the robot and the goal.

The parseNodes function now has an extra variable called largerRadius which is used in the checking of whether a node is close to a previous sub-goal. After a fair amount of trial and error, I found that 50cm seemed like an optimal value for largerRadius.

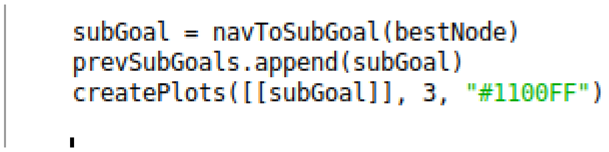


```
def parseNodes(nodesList, currentPosition, goalNode):
    radius = 20 # cm
    largerRadius = 50 #cm
    infNodes = []
    endnodes = []

    for x in nodesList:
        endnodes.append(x[-1])

    for i in nodesList: # i is a whole list of nodes from a single ranging
        for n in i:
            if ((n[0] - (i[-1][0]))**2 + ((n[1] - i[-1][1])**2)) <= (radius**2):
                n.append(math.inf)  #99999#math.inf is an infinite value.
                infNodes.append(n)
            else:
                g = distance(currentPosition, n)
                h = distance(n, goalNode)
                f = g + 2*h
                n.append(f)
        #i[-1].append(math.inf)
    for i in nodesList:
        for n in i:
            for x in endnodes:
                if ((n[0] - (x[0]))**2 + (n[1] - x[1])**2) <= (radius**2) and n[2] != 99999:
                    n[2] = (math.inf)
                    for a in i[(i.index(n)+1):-1]: # rest of the line is inf
                        a[2] = (math.inf)

    for subGoal in prevSubGoals:
        print("Checking sub-goal radii")
        for i in nodesList:
            for n in i:
                if ((n[0] - subGoal[0])**2 + (n[1] - subGoal[1])**2) <= (largerRadius**2):
                    n[2] = (math.inf)
    return nodesList
```

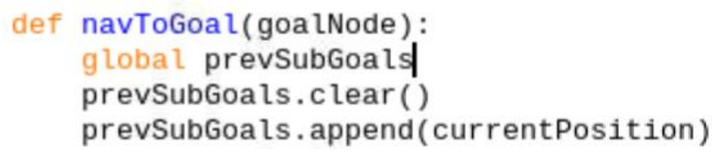
prevSubGoals is a global variable that is added to every time a sub-goal **reached.** It is declared as global at the top of the program. It is reset after each full journey, I.e., at the start of navToGoal.



```
global prevSubGoals
prevSubGoals = []
```



```
    subGoal = navToSubGoal(bestNode)
    prevSubGoals.append(subGoal)
    createPlots([[subGoal]], 3, "#1100FF")
```

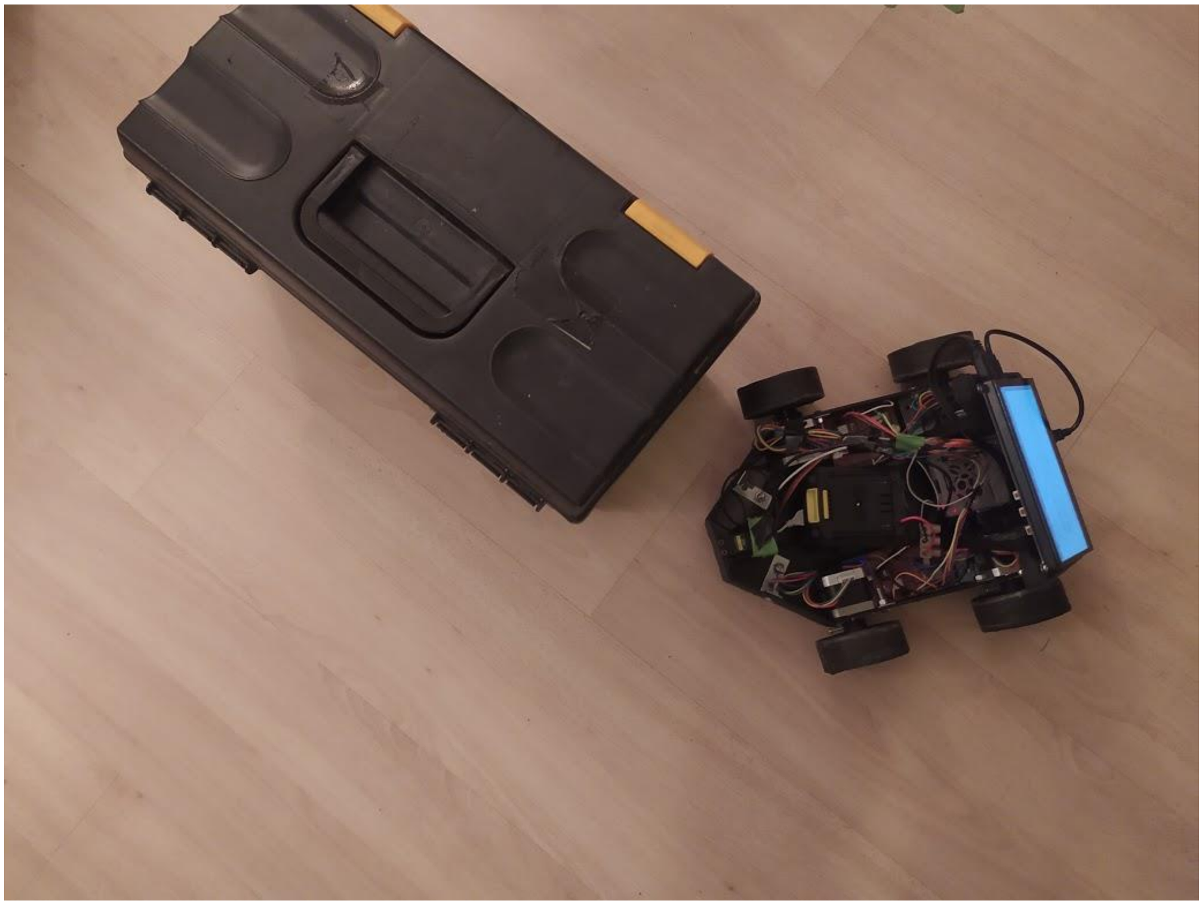


```
def navToGoal(goalNode):
    global prevSubGoals
    prevSubGoals.clear()
    prevSubGoals.append(currentPosition)
```

The robot now navigates around obstacles brilliantly (in conjunction with the fixes described below) and manages to navigate to the goal accurately.

08/02/21: After a few test runs, a problem became apparent. The lineOfSight function I programmed was causing the robot to crash. This is because if an object has any *thickness,* the sub-goal could be right next to it, but then when the lineOfSight function runs, there could well be a line of sight between the **middle** of the robot and the goal, but the side of the robot would still crash (see below). After much thought, I decided to omit the lineOfSight function from all the logic. This is because by the time the robot has scanned to try and calculate if the robot would crash, it may well have performed an entire scan (I.e., the time taken to do so would be almost the same as performing an entire scan). So, for simplicity the lineOfSight function has been removed, and the robot now continues finding subgoals until it reaches its final goal. It now does not crash into walls and performs the autonomous navigation extremely well.

14.4V



12/02/21: The manual control mode on the robot needed fixing. The UI had a button called "Take Manual Control", but that would just run the manualControl subroutine, and so the robot would be stuck in manual control, until the program was re-run. The fix below makes the button into a toggle button, turning on and off the manual control mode. NOTE, the robot cannot operate in manual control mode and autonomous mode simultaneously. First, a global variable called manualControlMode is declared at the start. The takeManualControl button subroutine now switches the state (True/False) of that variable as shown below.

The manual control subroutine works the same as before but is now continuously checking for the state of manualControlMode to change, and when it is set to True, it enters manual control mode 'until' it isn't set to True anymore. This is running simultaneously to the main program using a technique called threading (or multithreading) (the in-built python library *threading* is imported at the start). This makes It easy for the program to be checking for the change of state and run the main program at the same time.

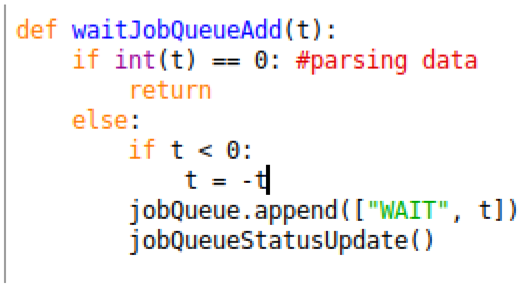
The manual control system now works as expected, and the currentPosition is changed in real time as the robot moves.

```python
global manualControlMode
manualControlMode = False
```

```python
def takeManualControl():
    global manualControlMode
    if manualControlMode == False:
        manualControlMode = True
    else:
        manualControlMode = False
```

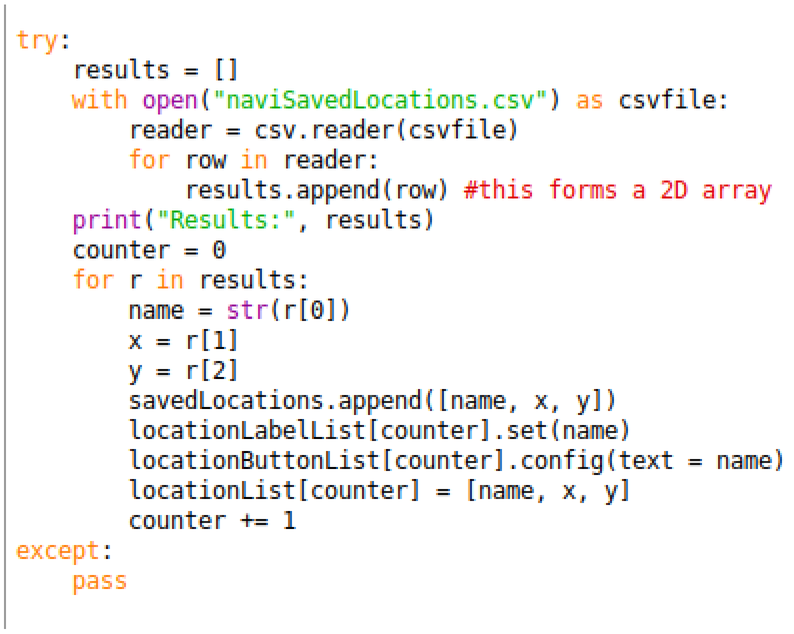
```python
def manualControl():
    global manualControlMode
    print("Manual control thread is running ")
    while manualControlMode == False:
        sleep(0.5)

    while manualControlMode == True:
        counter = 0
        #delay1 = delay
        turn = 0
        pygame.event.pump()
        axis = [ps3.get_axis(0),ps3.get_axis(1),ps3.get_axis(3),ps3.get_axis(4), ps3.get_axis(5)]
        print(axis)

        if axis[3] < -0.1:
            forward(delay, turn)


        if axis [3] > 0.1:
            backward(delay, turn)

        if axis[2] < -0.1:
            spinLeft(delay, turn)


        if axis[2] > 0.1:
            spinRight(delay, turn)
    manualControl()

#----
thread = threading.Thread(target = manualControl)
thread.start()
```

12/02/21: A smaller fix was to ensure validation on the “add wait time” function. If a user enters a negative number, it will be treated as a positive one. All other entries do not allow the user to progress until the correct syntax is entered.



```python
def waitJobQueueAdd(t):
    if int(t) == 0: #parsing data
        return
    else:
        if t < 0:
            t = -t
        jobQueue.append(["WAIT", t])
        jobQueueStatusUpdate()
```

12/02/21: The final fix was to change the system for storing saved locations. Previously, the new locations would just be created each time the program was run, and they would be stored (in RAM) until the program was terminated. This is a problem for the users of the machine, as it takes time to re-enter all the coordinates and names every time the program re-runs / the computer reboots. The new system reads and writes data from a file (in permanent storage) so that when the program is re-run or the computer reboots, all the coordinate and name data is still there and ready to be used. It even updates the labels on the buttons as well, so that the UI is exactly how it was when the program was re-run / when the computer rebooted. See below (this is placed after the Tkinter button objects have been declared):

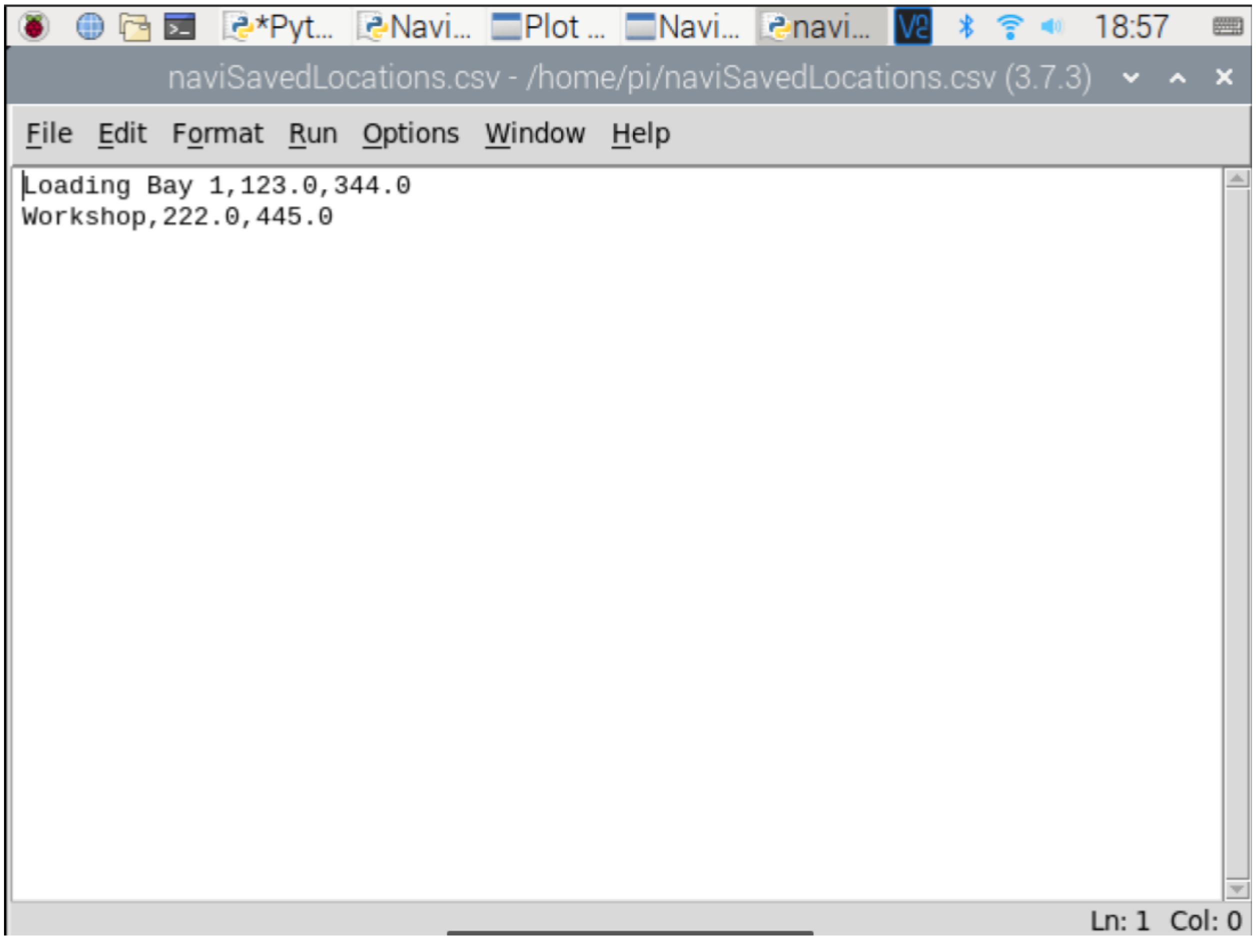


```python
try:
    results = []
    with open("naviSavedLocations.csv") as csvfile:
        reader = csv.reader(csvfile)
        for row in reader:
            results.append(row) #this forms a 2D array
    print("Results:", results)
    counter = 0
    for r in results:
        name = str(r[0])
        x = r[1]
        y = r[2]
        savedLocations.append([name, x, y])
        locationLabelList[counter].set(name)
        locationButtonList[counter].config(text = name)
        locationList[counter] = [name, x, y]
        counter += 1
except:
    pass
```

saveLocation now also saves to the location and it's name to the permanent file "naviSavedLocations.csv". I used **stackoverflow.com** to debug this, as I haven't used the csv library in a while.

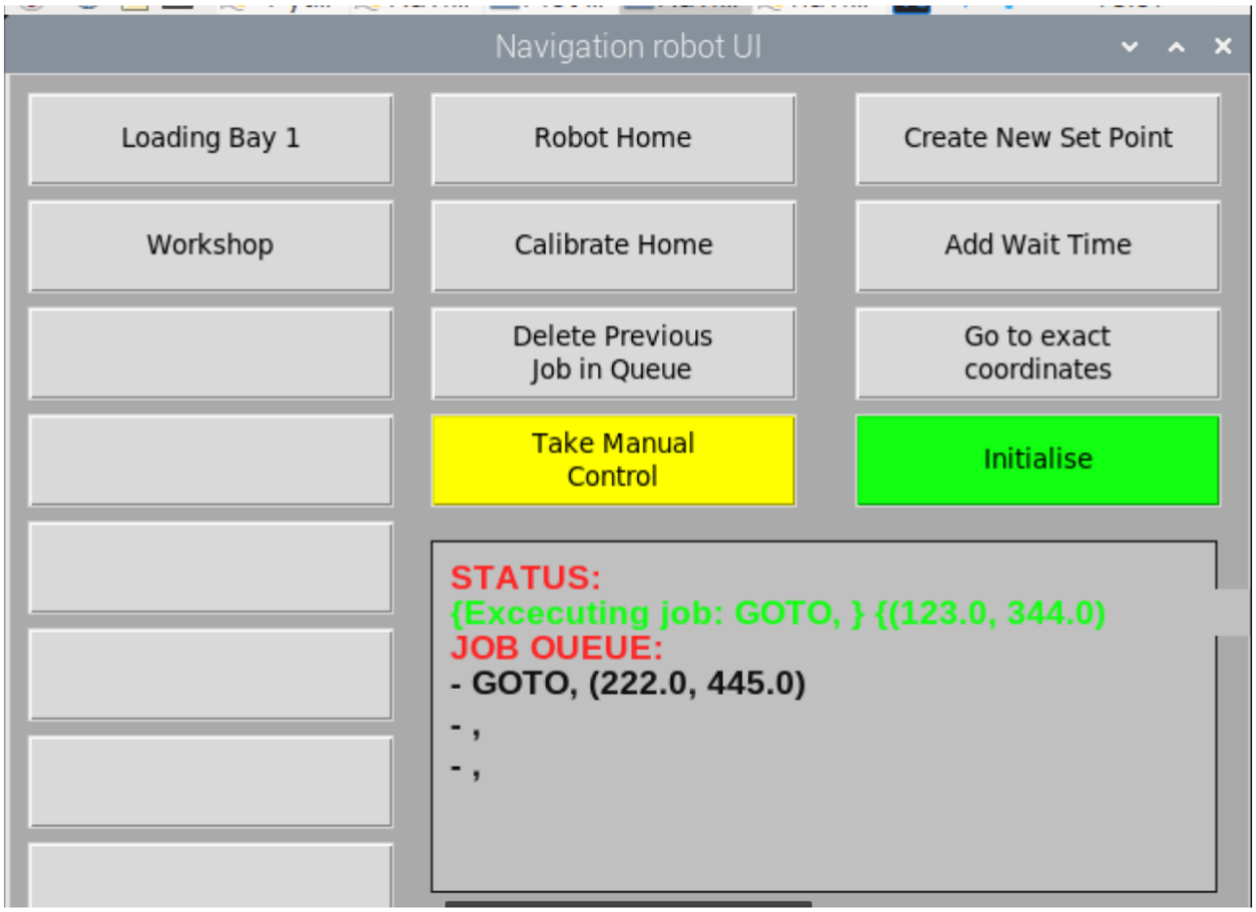
```
def saveLocation(name, x, y):
    locationToSaveTo = locationList[0]
    #handles saving to next available list aswell
    for l in locationList: # top to bottom
        if len(l) == 0:
            locationToSaveTo = l
            break
    for i in [name, x, y]:
        locationList[locationList.index(locationToSaveTo)].append(i)
    locationLabelList[locationList.index(locationToSaveTo)].set(name)
    locationButtonList[locationList.index(locationToSaveTo)].config(text = name)
    with open("naviSavedLocations.csv", "a", newline = '') as file:
        writer = csv.writer(file)
        writer.writerow([name, x, y]) # writes to file.
```

The csv file looks like this once two locations have been added.



```
Loading Bay 1,123.0,344.0
Workshop,222.0,445.0
```

After a reboot, the UI still reads the data and assigns the labels correctly.



```
STATUS:
{Excecuting job: GOTO, } {(123.0, 344.0)
JOB OUEUE:
- GOTO, (222.0, 445.0)
- ,
- ,
```

Testing.

The testing here follows the post-development test plan described in the **design** section above. Changes may be made in accordance with test results.

Safety

Running the safety tests, I found that most of the time, it performed correctly, once or twice It just bumped my foot, but was not painful. These errors are due to the single point lidar possibly missing my leg as it scans further back, but then recognizing it as it gets closer. This would be fixed with a hardware upgrade and is something I should refer to in my evaluation.

UI Intuitiveness

Due to the coronavirus pandemic, I was not able to let people test whether the UI was intuitive enough, so I have skipped this for now.

UI inputs to determine goal.

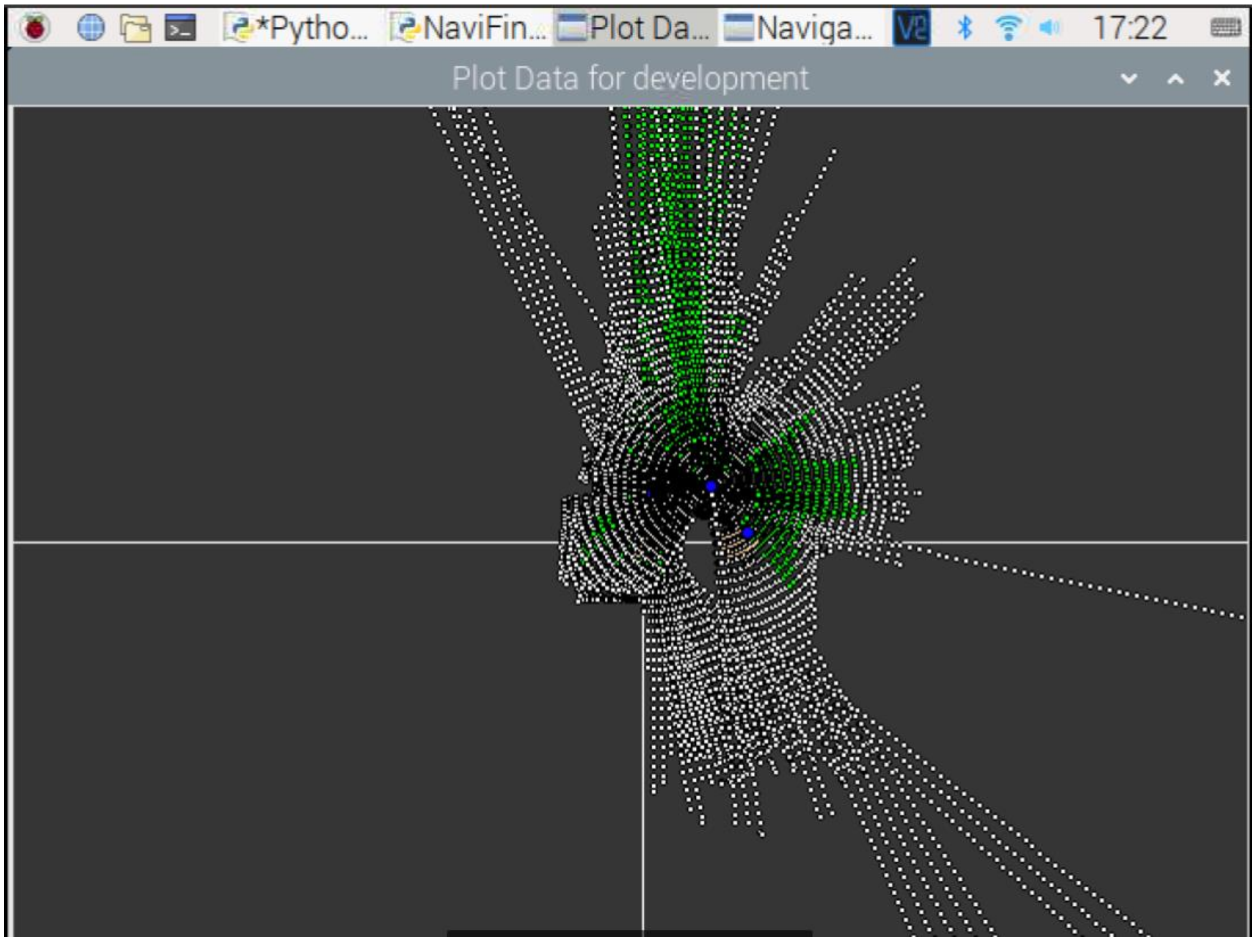
This seemed to work quite well, with the robot successfully navigating to the goal that was specified in both the ‘go to exact coordinates’ system and the saved location system.

Static obstacle avoidance.

When avoiding static obstacles, the robot was okay, but not always perfect. Depending on the configuration of the obstacles, the robot would either navigate successfully, get trapped in a ‘zone of infinite cost (due to the obstacles being in too complex a configuration), or crash into the side of one of the obstacles (this was due to the robot drifting from true position over time, something that could be fixed with other hardware, I will address this while evaluating.). In simple environments, with walls and basic obstacles, it usually performed as expected.

Path optimization.

The robot was able to perform path optimization on all 5 of the test runs and navigated to the goal successfully. Below is a screenshot of the plot data from one of the runs, with the blue dots representing the sub-goal locations that were chosen. (the home position is obscured, but home is on the intersection of the two white lines):



Job queue functionality.

After testing the job queue, I found that it worked as expected. Although, the robot was not always able to complete every job on the 5-job job queue, due to it crashing into the side of obstacles. This is because of the robot drifting position over time when navigating over a long distance (I.e., going to multiple locations simultaneously). This is an issue that will be addressed in the evaluation.

Testing the robot's return home and calibrate home functionality.

The return to home functionality works and the calibrate home functionality all work excellently. The calibrate home functionality is much needed due to the robot's tendency to drift from its true position.

Testing the manual control capabilities.

The manual control capabilities of the robot work well from a control perspective (the ps3 controller connects to the robot and controls the forward, backward, spinLeft and spinRight functions with no problems), but the gradual drift from its position is a real issue. After my test, it returned to the same spot with a currentPosition value around 28cm from where it actually was. So, while it still passed the test it did not do so 'with flying colours'. The drift issue is something I will address in the evaluation.