EG2310
Group 5
Final Report

Student Team Members:

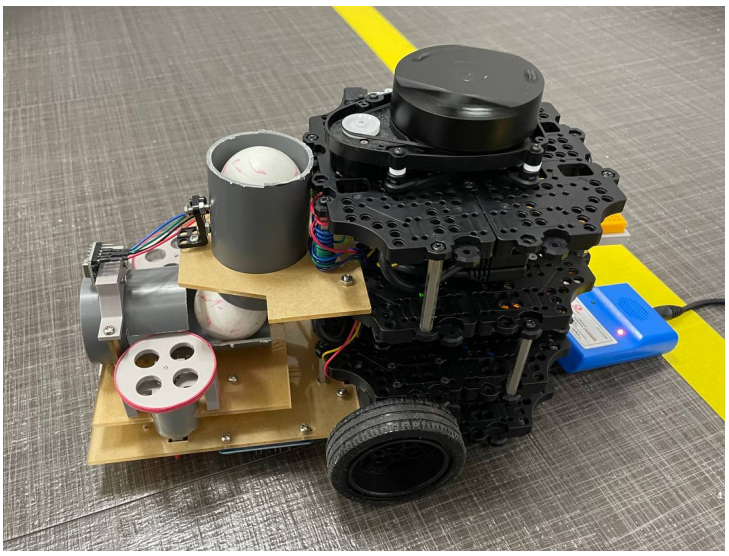
Chen Yuhan	A0211250A
Joanne Chong E Qin	A0245799E
Oong Jin Rong Jared	A0180023X
Tan Yan Ze	A0233570L
Wu Yongxin	A0233666Y

Table of Contents

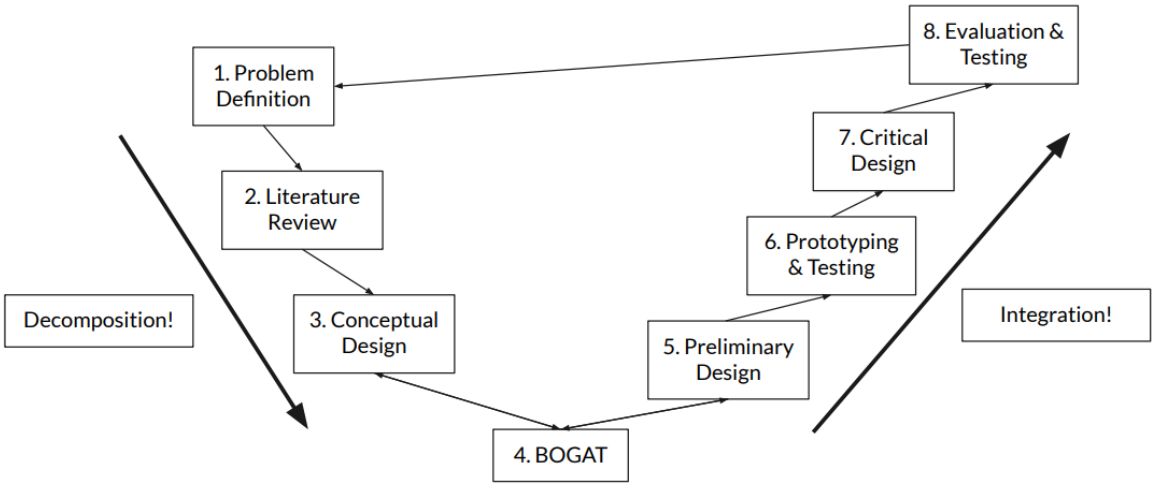
1. Introduction

This document describes the system design process of our EG2310 robotic system. The system was designed to achieve the following purpose:

> *To navigate and map a maze without colliding with obstacles, detect and stop at an NFC tag for ping pong ball loading. Subsequently, to find and detect a hot object and fire a ping pong ball at the object.*



The system design process utilised follows the modified EG2310 V-Model as shown below:



NUS EG2310 System Design V-Model

The following Sections detail each part of the system design process:

1. V-Model parts 1 - 3, 5 and 6 are described in their respective Sections from 2 - 6
2. BOGAT is subsumed under Section 4.6 Integration
3. Critical Design is described in detail over Sections 7 - 10
4. Evaluation & Testing is presented in Section 11 with a list of possible future works

2. Problem Definition

The ultimate goal of this project is to design and build a payload module for a Turtlebot3 Burger that can accomplish a set of missions in a maze of size less than 5 m by 5 m. The main problems faced are broken down into the following sections:

Autonomous navigation

a. The Turtlebot3 should detect obstacles in its surroundings, maintain a certain distance from the obstacles, and make turns when necessary for course corrections.
b. The Turtlebot3 should collect information about its surroundings and record them as a map to keep track of areas that have been explored.
c. The Turtlebot3 should identify areas that have not been explored and navigate autonomously to these areas to explore the map completely.
d. The Turtlebot3 should have sufficient power and fidelity to navigate the maze.

Detection of NFC tags

a. The Turtlebot3 should scan its surroundings for NFC tags.
b. The Turtlebot3 should stop in place when an NFC tag is detected.

Loading of ping pong balls

a. A loading bay should be attached to the Turtlebot3 to store the ping pong balls loaded by the TA. The loading bay mechanism needs to be intuitive enough for the TA to load it with ping pong balls without having been instructed on how to do so.
b. The TA should be able to perform a simple intuitive task to let the Turtlebot3 know that the loading process is completed.

Detection of “hot target”

a. The Turtlebot3 should scan its surroundings for a target with infrared heat signature.
b. Once identified, the Turtlebot3 should stop at a suitable distance from the target.

Aiming and firing of ping pong balls

a. The Turtlebot3 firing mechanism should not be triggered until it is ready to fire
b. The ping pong balls should be transferred from the loading bay to the launcher.
c. The Turtlebot3 should aim the launcher accurately at the “hot target” accurately, based on various factors such as distance and direction of the “hot target” relative to the Turtlebot3.
d. The Turtlebot3 should trigger the launcher to fire the ping pong balls towards the “hot target”. Depending on the amount of ping pong balls loaded, the Turtlebot3 then needs to repeat steps 5a to 5d to fire again at the target to improve hit probability.

3. Literature Review

3.1 Autonomous navigation

The Turtlebot3 has a 360 Laser Distance Sensor LDS-01, a 2D laser scanner that is capable of sensing 360 degrees with a detection distance of approximately 120 mm to 3,500 mm [1]. The data around the robot collected by the LDS-01 is used for SLAM (Simultaneous Localization and Mapping) and Navigation.

Two possible algorithms allow the Turtlebot3 to explore the maze autonomously: the wall follower algorithm [2] and the Rapidly Exploring Random Tree (RRT) algorithm [3]. For the wall follower algorithm, either the right-hand rule or the left-hand rule is applied to solve the maze. The rules are based on the simple logic of movements, in which the Turtlebot3 follows one side of the enclosure wall and will eventually reach the exit of the unknown maze. For the RRT algorithm, it is constructed incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree. The robot will plan its path to move to all the reachable distant endpoints detected by the LDS and map the new regions continuously using SLAM. This algorithm is very effective for environmental exploring tasks, as its nature tends to be skewed towards unexplored areas.

3.2 Detection of NFC tags

For NFC readers, there are two choices, one is PN532 and another one is the Adafruit NFC reader.

The PN532 NFC RFID Module is a possible solution for detecting NFC tags. It supports RFID reading and writing, NFC function with Android phones and data exchange with other NFC devices. Its typical operating distance is around 5 cm to 7 cm [4]. One method to connect the PN532 NFC RFID Module to the Raspberry Pi is via I2C [5].

Adafruit NFC shield uses the PN532 chip-set and can detect any 13.56MHz RFID or NFC application.

3.3 Loading of ping pong balls

The simplest solution to load ping pong balls is a tube that stores ping pong balls stacked vertically. The balls can be loaded one by one from the top, and the top can either be completely open or have a hatch that must be opened to allow the loading of balls. Should a hatch be present, it should either be controlled by the Turtlebot3 and open automatically when detecting the NFC tag and close once it detects that it is loaded. If not, the hatch should be sufficiently obvious and intuitive to open to allow the TA to open it with simple instructions or without prior knowledge of its operation. The tube to store the ping pong ball should not be taller than 16.0cm off the ground, as that is the measured height of the LIDAR sensor and any object taller than it would block the LIDAR and may be registered as a wall or otherwise affect the LIDAR performance.

The number of balls loaded can be anywhere from 1 to 5 balls, based on the group's request. As such, the number of balls to be fired can be stored as a preset variable within

the Turtlebot3 code instead of having to use a sensor to detect the number of balls, which will add cost, complexity and power draw to the system.

The Turtlebot3 needs to be able to determine when the TA is done with the loading process so that it can proceed on with attempting to find the hot target. This can be done with a simple button circuit utilising the Raspberry Pi's GPIO pins. A GPIO pin set to input mode can be connected to the 3.3V output with a button and resistor, such that pushing the button will allow the input pin to detect a non-zero voltage. A corresponding script running on the rPI will need to be created to configure the pin and allow it to publish to another subscriber script once it detects the button is pushed, which will move the Turtlebot3 into the hot target seeking part of its script [6].

To load the firing mechanism from the storage compartments, gravity can be used to pull the ping pong balls into the firing mechanism.

If a flywheel firing mechanism is used, to improve accuracy and consistency of the fired balls, the balls should be fired one at a time, so the flywheels have time to regain their rotational speed after firing a ball. A gate mechanism would need to be installed to prevent the ping pong balls from falling into the firing mechanism before the Turtlebot3 is ready to fire.

Alternatively, a piston can be used to push balls one at a time into the flywheels, similar to the operation of a firearm bolt. This will allow the balls to enter the flywheels at a controlled speed, and only allow one ball at a time to enter the launcher mechanism. To achieve this, a linear actuator can be used to provide translational movement in a single axis by converting the motion of a standard DC rotational motor. A cheap linear actuator such as those provided with the Lego EV3 Technic [7] will likely suffice in this capacity, as the piston does not need to be very fast nor does it need to support a large weight, given that ping pong balls are light (2.7g) [8]. Other possible options for the piston motor include linear solenoid motors and DC linear motors. These are more expensive than a DC motor + linear actuator combination, and solenoid motors have a limited range of motion compared as well.

3.4 Detection of "hot target"

A cheap thermal camera can be built using the AMG8833 IR Thermal Camera to detect the hot target [9]. It is an 8x8 array of IR thermal sensors by Panasonic. When connected to the Raspberry Pi, it will return an array of 64 individual infrared temperature readings over I2C. This sensor can measure temperatures ranging from 0 °C to 80 °C with an accuracy of ± 2.5 °C and a maximum frame rate of 10Hz. An 8x8 grid could be interpolated using the SciPy python library to assist image processing. The main idea is to capture the temperature readings in each grid, convert them into RGBA values and generate the thermal image based on the results.

3.5 Aiming of ping pong balls

Aiming consists of a few steps. Firstly, having detected the target using the thermal camera, the Turtlebot3 then needs to determine accurately the distance of the target relative to the bot itself. The height of the target can be assumed to be from 0 to 45cm, based on the TA's description of the target as a Khong Ghuan biscuit tin [10]. As the thermal sensor

chosen does not have distance sensing capabilities, the LIDAR can be used to determine the distance of the hot object from the Turtlebot3, by using the thermal sensor to position the hot object in front of the Turtlebot3 and then using the LIDAR to give the reading of the distance to the nearest object directly in front of the Turtlebot3.

The required distance, height and angle telemetry from the sensors should then be published for either the Raspberry Pi or connected laptop to receive and make the required calculations. The Turtlebot3 script should be able to calculate which direction to aim the launcher at, as well as make the required adjustments through the actuation of the motors. If the distance to the hot object is too far for the launcher, the Turtlebot3 should also move itself to a suitable distance from the hot object. Since there is no minimum distance required for the project between the Turtlebot3 and the hot object, the distance for firing parameters is based on the group's discretion. The calculations for the ping pong ball are in Annex G.

To aim, the launcher will require adjustments in 2 dimensions. Adjustments in the horizontal plane (x and z dimensions) can be done by rotating the Turtlebot3 itself using the r2moverotate.py script, while adjustments in the vertical plane (x and y dimensions) is not required as long as the parabolic path of the fired ping pong ball remains between 0 to 45cm, based on the height of the target and calculations in Annex H.

3.6 Firing of ping pong balls

A flywheel system could be used to launch the ping pong balls, with guides being available online to make ping pong ball flywheel launchers [12]. The flywheel launcher works by having the ball contact and be launched by two spinning wheels on opposite ends of the ball. The angular momentum of the flywheels will be transferred and converted into linear motion of the ball strictly perpendicular to the line connecting the two flywheel centres. An estimate of the rpm required for the flywheel based on other projects on the internet is 10000rpm, such as a Micro 130 DC 6V 10000rpm motor [13] [14].

Another method is to use an elastic slingshot [15] to launch the ball, or similar mechanisms involving the use of elastic potential energy, such as crossbows. These work by storing energy in the form of elastic potential energy by deforming an elastic part, then allowing the elastic part to return to its original shape and converting the stored elastic potential energy to kinetic energy, which is transferred to the ping pong ball projectile.

Lastly, a pneumatic system [16] is also possible to fire ping pong balls. These require either a reservoir of pressured air to be stored beforehand, or an onboard motorised pump to pressurise air. The pressurised air can then be released in controlled bursts through an electronically controlled valve to propel the ping pong ball out of the launcher.

The flywheel mechanism should be used as it is the simplest mechanism, requires the least sturdy materials, and, most importantly, is the easiest to reload for multiple shots. In comparison to the other systems, a flywheel which only requires the flywheels to continue spinning to launch another ball. Elastic systems will require stronger and subsequently heavier/more expensive materials to withstand the compression forces, and recocking the elastic system will require a more complex mechanism involving linear actuators with sufficient force. Pneumatic systems on the other hand will require a bulky and sturdy

pressure tank to contain the pressurised gas, and possibly a motorised pump to fill the pressure tank, which all adds to the cost, complexity and weight of the design.

However, it should be noted that the flywheel system will be very inaccurate [17]. The first point of inaccuracy is that the motors of the flywheel may spin at different speeds, the friction of the flywheels are different or the ping pong ball does not contact the motors at the same time. All of these will result in the flywheels imparting a different force on the ball, which will affect the direction the ball is fired as well as its spin. Ping pong balls are also light and have a low ballistic coefficient and hence experience a large amount of air drag, which will affect the motion path and hence accuracy of the fired ball. Hence to reduce the inaccuracy of the system and simplify the issue, the Turtlebot3 should fire the launcher when it is as close to the hot target as possible. There is no minimum range requirement for this project, however the LIDAR has a minimum range of 12cm [1], and as such the Turtlebot3 may run into issues with its algorithm if the firing distance is less than 12cm. To avoid such issues, a firing distance of 20cm will be set to give a buffer for the LIDAR minimum distance while minimising the firing distance. This may be adjusted after actual testing of the launcher.

4. Concept(s) Design

4.1 Autonomous navigation

For the autonomous navigation of this mission, two separate algorithms would be used for different forms of navigation, one for finding the NFC and the other for finding the thermal object. For the first part of finding the NFC, since the NFC tag is placed on the floor and the Turtlebot3 needs to go over the NFC tag for the tag to be detected, a complete coverage path algorithm should be used to ensure that the Turtlebot3 does not miss the NFC tag. However, as a complete map needs to be provided to correctly implement the algorithm, a simpler modified version shall be used instead. The initial algorithm to find the NFC tag would be adopted from how a Roomba operates [18], with the Turtlebot3 making a U-turn in the direction of the unexplored region every time it hits the maze edge. On top of this algorithm, in the event that the Turtlebot3 is blocked by an obstacle, it will switch over to another algorithm adopted from the Pledge algorithm [19] in order to bypass the obstacle and continue on its initial path. Hence by adopting and tweaking the complete coverage path algorithm and the pledge algorithm, the Turtlebot3 will be able to navigate the maze while covering as much ground as possible efficiently.

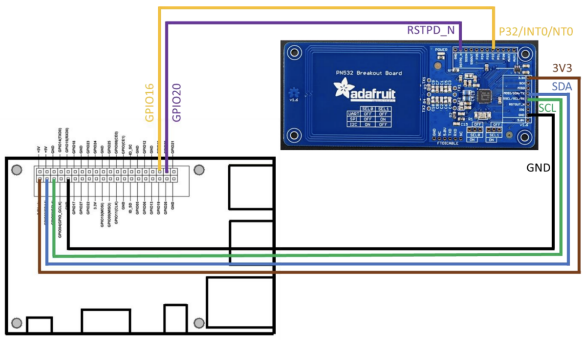
For finding the thermal object, a different concept will be implemented. Since the range of the thermal camera is approximately 7 m [20], the Turtlebot3 would be able to detect the thermal object even when the Turtlebot3 and the object are at different edges of the maze, assuming an uninterrupted line of sight. Hence in order to find the object, the Turtlebot3 will simply travel to the edge of the maze first based on where its position is after the ping pong balls are loaded. After which, the Turtlebot3 will start moving forward at fixed distance intervals and turning 90-degrees to face inside the maze. The exact distance for the fixed intervals would be trialed and tested to find the correct distance such that the Turtlebot3 checks a new area that was not within the thermal camera's line of sight from the previous position. Using this concept, the Turtlebot3 would be able to visually scan every corner of the maze systematically, which thus ensures that there would not be a situation in which the object is in a blind spot that the Turtlebot3 is unable to scan. Detailed explanation of how both algorithms work can be found in Annex E.

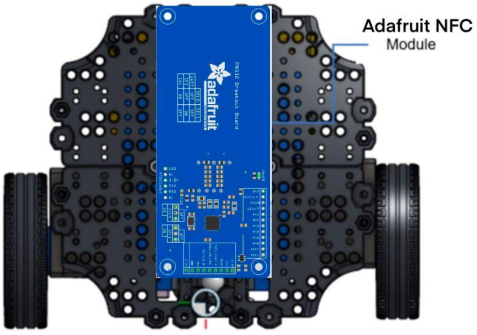
4.2 NFC reader

Although PN532 can also reach the goal (more details given in Annex H), this project will use the Adafruit NFC.

The NFC reader to be used in the design is the Adafruit NFC. The interface to be connected consists of the power supply (Vcc), the ground line (GND), the clock line (SCL), the data line (SDA), the RSTPD_N and P32. This module requires 3.3V to run and would Vcc hence be connected to the RPI on pin 1. The SCL and SDA on the module would be connected to the GPIO 3 [pin 5] and GPIO 2 [pin 3] respectively on the RPI, which are responsible for such functionality respectively while the RSTPD_N and P32 will be connected to GPIO 20 and GPIO 16 respectively. The ground would be connected to pin 9, which acts as the ground on the RPI. Since the NFC tag would be placed on the floor, the NFC module would be placed on the underside of the Turtlebot3 in order to minimise the distance between the NFC reader and the tag. The module can then be activated, which

would cause it to sense for data constantly. Once it detects an NFC tag, data regarding the NFC tag would be returned to the RPI. The small size of the module allows for easy incorporation into the model of the Turtlebot3.

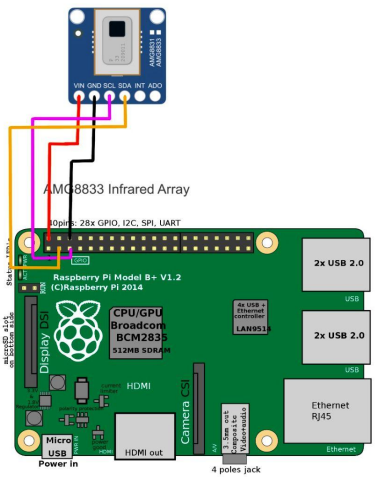


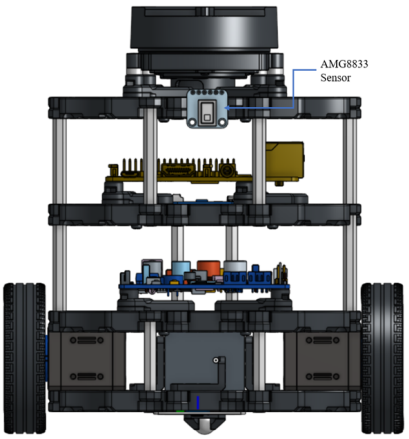


Bottom View of TurtleBot3 Burger

4.3 Thermal camera

The thermal camera to be used for the detection of the heated target is the AMG8833. It uses I2C to communicate with the RPI and transfer data. Although the module has six pins, only four will be utilised, mainly the Vcc, GND, SCL and SDA. Since the module is powered up by 3.3V, it will be connected to pin 1 of the RPI. SDA and SCL would be connected to pin 3 and 5 respectively, and GND would be connected to pin 6. The module would then be mounted at the front of the Turtlebot3 in order to sense the presence of the heated object in front of the Turtlebot3. Once the heated target shows up within the 8x8 grid of the thermal camera, the Turtlebot3 can stop navigating and instead rotate about its position such that the heated target would appear within the middle four columns of the grid, ensuring that the target and the Turtlebot3 are aligned.



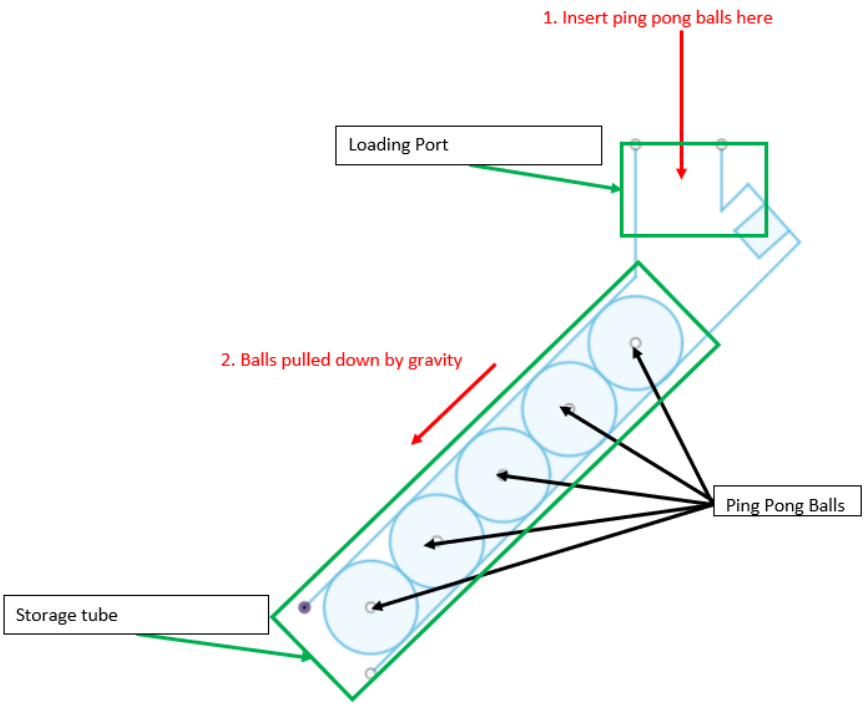


Front view of TurtleBot

4.4 Launcher loading mechanism

A preliminary design for the loading mechanism is a diagonally angled storage tube with a vertical loading port, as in the diagram below. Ping pong balls are loaded via the loading port, and fall via gravity into the storage tube and then further into the launcher mechanism. As the loading port and storage tube need only support up to the weight of 5 ping pong balls (totalling 13.5g [8]), they can be made of cheap and lightweight cardboard or PVC. Diagonally angling the tube allows for more balls to be stored.

For the piston, a rotary DC motor can be used to drive a linear actuator to perform the role of the piston. This option is more affordable than using the alternatives as stated above.



4.5 Launcher shooting mechanism

For the flywheel, 2 flywheels mounted in a vertical configuration relative to the fire ping pong ball will be used, as in Annex C. Each flywheel will be a DC rotary motor with rubberised wheels attached to grip and launch the ping pong ball. The motors should be capable of high rotation speeds to increase the speed of the launched ball. Rough calculations of the firing mechanism are available in Annex G.

The housing for the firing mechanism can also be made of cheap and lightweight cardboard, PVC tubing or 3D printed with PLA.

4.6 Integration

Firstly, due to the multiple sensors connected to the RPI, each Vcc pin of the sensor would not be connected directly to the 5V or 3.3V pin of the RPI. Instead, wires would be used to connect the 5V and 3.3V pin of the RPI to separate rows on the breadboard. Similarly, the ground pin from the RPI would also be connected to another row on the breadboard to allow all the components to be connected to a common ground.

Secondly, since two sensors are using I2C communication to the RPI, there is a need to ensure that the addresses of these two components are different. Fortunately, the default address of the NFC reader is 0x48, while the default address of the thermal camera is 0x69.

Additionally, to allow both sensors to be connected to the SDA1 and SCL1 of the RPI, the SDA and SCL pin would be connected to individual columns on the breadboard.

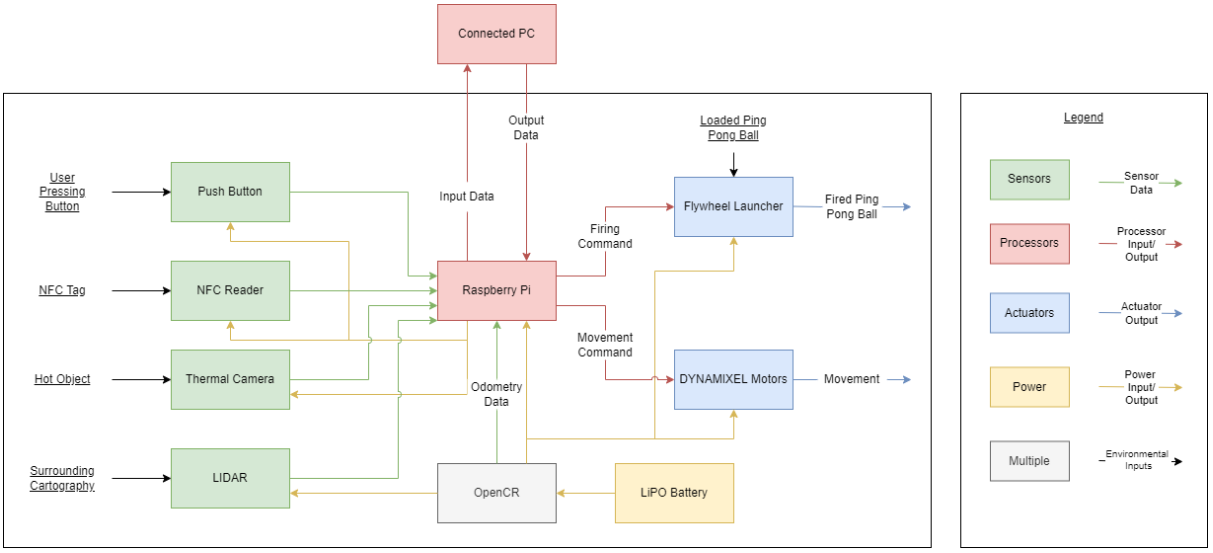
An estimated size of the loading and launcher mechanisms would be at least 12cm (Height) x 6cm (Width) x 20cm (Length) based on 5 ping pong balls (4cm ea) and allowances for the firing mechanism. This sizing ensures the system does not block the LIDAR. Due to this size, it will have to be mounted as an attachment in front/behind the Turtlebot3. Mounting it at the back is not ideal as the launcher would be blocked by the Turtlebot3 and the Turtlebot3 would have to rotate 180° to fire the launcher, introducing unnecessary complexity to the bot script and operation. Mounting to the front would require the forward facing thermal sensor to be placed in a way that they are not blocked by the launcher mechanism.

To support the weight of the loading and launcher mechanisms and prevent the Turtlebot3 from toppling to a side, the mechanism itself should be supported by wheels that allow it to travel and rotate with the Turtlebot3. This separate platform can be 3D printed with built in screw holes and then be connected to the Turtlebot3 via the M2.5x8mm sized screws used on the Turtlebot3 burger waffle. The wheels on the separate platform can also be 3D printed, and attached to pre-modelled holes in the 3D printed platform. The shaft for the wheels can be 3D printed and attached to the platform via either screws or bonding (eg. with superglue or acetone). The material, thickness and infill of the 3D printed parts should be determined after the other parts and their weights have been confirmed, to allow the 3D printed parts to be able to withstand the associated stresses (eg. shear, compression & tension, etc.) due to the launcher's weight and operation.

Lastly, to prevent the drawing of too much power out of the RPI, the launcher component which consists of the flywheel and the piston motor would be powered up by an external battery.

5. Preliminary Design

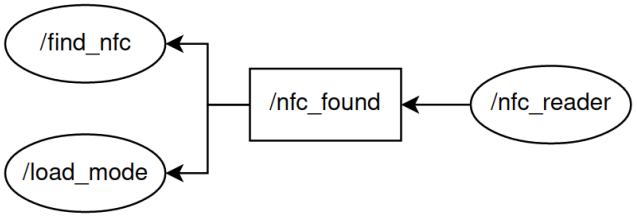
The diagrams depicting the flowchart of events for the Turtlebot3 to clear the requirements and how the nodes communicate with each other can be found in Annex B. The functional block diagram for the overall system is as seen below:



System Functional Block Diagram

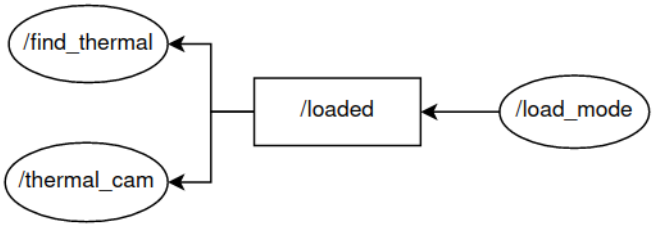
5.1 Phase 1: Finding the NFC tags

When the Turtlebot3 is started up, the nodes for both phase 1 (/find_nfc, /nfc_reader) and 2 (load_mode) would be activated. The NFC reader node (/nfc_reader) would publish over a topic (/nfc_found) which the auto_navigation node (/find_nfc) for phase 1 would subscribe to. As long as the NFC tag has not been found, the NFC reader node (/nfc_reader) would constantly publish a boolean value of 'False', and the Turtlebot3 would continue navigating the maze autonomously to find the NFC tag. The auto_navigation node consists of the functions required for the Turtlebot3 to maneuver around the maze, and the NFC reader node (/nfc_reader) consists of a function for reading in data from the NFC reader and publishing to the topic (/nfc_found). In order to sense the NFC tags on the floor of the maze, a PNC532 connected to the RPI would be placed on the underside of the Turtlebot3. I2C is used as the method of transferring data between the PNC532 and the RPI, with the PN532 returning the data of the NFC tag when found. The node in charge of the loading (/load_mode) for the next phase would be subscribed to the same topic (/nfc_found) in order to know when to activate the sensors. The nodes used in phase 1 (/find_nfc, /nfc_reader) will be destroyed once the next phase begins, and the nodes for phase 3 will be activated (/find_thermal, /thermal_cam).



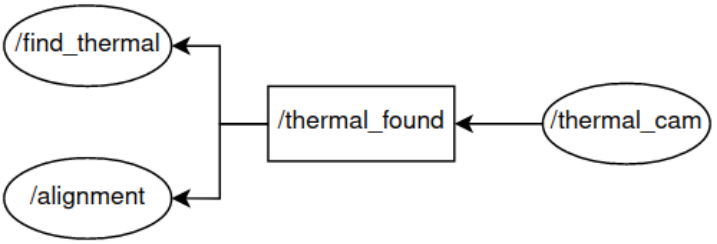
5.2 Phase 2: Loading the Turtlebot3 with ping pong balls

For the loading phase, a simple button circuit can be used to inform the Turtlebot3 when the balls have been loaded. The TA will help to press a button mounted on the Turtlebot3 once he has loaded the balls. The RPi will be able to detect when the button is pressed as detailed in the literature review, and publish once the balls have been loaded. The node in charge of the loading (/load_mode) would be publishing once the button is pressed, over a topic (/loaded) and the nodes for the next phase, finding the thermal object, (/find_thermal, /thermal_cam) would be subscribed to this topic to know when to start searching for the hot object. When the Turtlebot3 detects the button press, the Turtlebot3 would then wait for a period of time before moving off, to ensure that it would not collide with the TA's hand right after the button is pressed. Once the next phase is initiated, the node in charge of the loading phase (/load_mode) would be destroyed, and the node for phase 4 will be activated (/alignment).



5.3 Phase 3: Finding the heated object

For the third phase, in order to find the heated object, a thermal camera is attached to the front of the Turtlebot3. Using a similar concept as phase 1, the thermal camera node (/thermal_cam) would publish over a topic (/thermal_found) on whether the thermal object has been detected. The thermal camera node (/thermal_cam) would contain functions to determine whether the thermal object has been detected from the data read back from the camera. The navigation node (/find_thermal) would be subscribed to this topic (/thermal_found) to know when to stop navigating. Similar to the NFC module, I2C is used to transfer data to the RPI. The node for the next phase, which is the alignment of the heated object and the Turtlebot3, (/alignment) would be subscribed to the same topic to know when the heated object has been found so that it can begin the alignment process. Once the alignment process begins, the node for finding the thermal object (/find_thermal) would be destroyed, while the thermal camera node will be kept for use in the next phase. Additionally, the node for phase 5 would be activated (/launcher).



5.4 Phase 4: Aligning the object and turtlebot

The alignment node (/alignment) would subscribe to the topic (/thermal_found) to obtain the relative position of the object from the Turtlebot3. The alignment node (/alignment) would also publish over a topic (/centered) on whether the Turtlebot3 is centered to the object, and the node in charge of activating the launcher (/launcher) would subscribe to this topic. For the alignment process, using the data received over the topic (/thermal_found), the Turtlebot3 would rotate either left or right such that the launcher would be in line with the heated object. Ideally, this will be when the heated object is in the centre columns of the thermal camera. However, depending on the angle and distance of the launcher to the thermal camera, the heated object may have to be offset from the centre of the thermal camera to compensate. Further testing will need to be done once the launcher is attached to the Turtlebot3 to determine the offset required. Once the heated object is within the centre columns of the thermal imager, the distance between the object and the Turtlebot3 is then obtained from the LIDAR. Based on what is received from the LIDAR, the Turtlebot3 would either move closer or further away from the heated object till a desired, pre-programmed distance is reached. The movements required for this phase would be implemented as functions, similar to the navigation phases in front. Once the object is aligned to the Turtlebot3, the next phase would begin and the nodes for alignment (/alignment, /thermal_cam) would be destroyed.

5.5 Phase 5: Launching the ping pong balls

The node to launch the ping pong balls (/launcher) keeps track of the number of ping pong balls left. Once the launcher node (/launcher) is started, the flywheel mechanism will be started up. The piston motor would then be activated, where it will push a ball towards the flywheel mechanism, allowing the ball to accelerate towards the heated target. Simultaneously, the extension of the piston prevents the other balls in the loading tube from falling into the firing mechanism and being fired by the flywheel. The Turtlebot3 should be able to tell when a ping pong ball is fired by the length of extension of the piston, which is determined by the amount of time the linear actuator motor spins. This length of time is to be determined by testing once a specific linear actuator is purchased. Once the Turtlebot3 determines that the ping pong ball is fired, it will then reverse the polarity of the current through the linear actuator motor, hence reversing the spin of the motor and retracting the piston instead. Once the piston is fully retracted, again determined by the time the motor is active, the next ping pong ball should fall into the firing mechanism and be ready to be pushed by the piston and fired. The mechanism will continuously repeat until all the ping pong balls are shot out, which can be a pre-set variable in the Turtlebot3 script based on how many balls the group requests to load.

6. Prototyping & Testing

6.1 Navigation Algorithm

When testing out the code which allowed the Turtlebot3 to perform a U-turn whenever it reaches the wall, the navigation worked and the robot was able to map out the entire maze. However, it took an extremely long time. Given the new criteria that the timing was solely based on the time taken to map out the maze, our team decided to opt for a simpler yet more effective algorithm which is to use the left wall following algorithm. Using a left wall following algorithm reduced the timing required to map the maze significantly, while still allowing the Turtlebot3 to detect the NFC on the floor near the walls.

For our first iteration of the wall following algorithm, the Turtlebot3 stopped whenever it had to rotate. This resulted in the movement of the Turtlebot3 being jerky and it slowed down the Turtlebot3. Given that time was an important factor of consideration, we decided to make adjustments to the algorithm to allow the Turtlebot3 to rotate and move forward at the same time which allowed it to follow along the corners of the wall more smoothly. The rotation and forward movement speed of the Turtlebot3 was then adjusted through trial and error before arriving at the final parameters that allowed the Turtlebot3 to continuously move forward, unless it was about to collide with a wall.

Another point of consideration was the distance between the wall and the Turtlebot3. Given that the NFC tags were placed near the wall, we had to ensure that the Turtlebot3 did not veer off too far from the wall while carrying out the wall following algorithm. Additionally, we also had to ensure that there was sufficient space for the Turtlebot3 to turn without colliding into the wall which would either cause it to be stuck, or would mess up the map. This was a significant problem given the design of our Turtlebot3, as the protruding launcher resulted in a large turning radius. As a result, the distance had to be calibrated and adjusted through trial and error to give the Turtlebot3 ample clearance to ensure that its front will not hit any obstacle or wall while it is rotating.

6.2 Electrical Wiring

While testing out the code, the different sensors were attached to the Turtlebot3 using a breadboard and jumper wire headers. This was sufficient for the prototyping phase as the Turtlebot3 travelled small distances and the sensors were tested individually. The use of a breadboard also allowed for easy swapping of the sensors which was important as it allowed us to test the different functionality of the Turtlebot3 as quickly as possible.

Once the different sensors were tested to be working individually, a PCB was designed to allow for simple and neater wiring of the circuitry. Two PCBs were designed, one for the RPi and the other for the Adafruit PN532 NFC Breakout Board. Both PCBs were designed to be attached to the RPi and the Breakout Board as hats, which allows for easy attachment. The design of the PCB also reduces the chances of connecting the sensors to the wrong pins on the RPi as the breakout pins are separated and labelled clearly on the PCB, as shown in Annex I.

However, the manufacturing process was disrupted as a result of COVID-19 which resulted in uncertainty on the arrival dates of the PCB. Hence, our group switched over to using a protoboard which was designed to serve the same purpose as the PCB and also be placed on the RPi as a hat. Due to the proximity of the NFC Breakout board to the floor, no additional protoboard was designed for the NFC Breakout board to prevent a situation where the wires were being dragged along the floor as the Turtlebot3 is moving around.

6.3 Launcher Mechanism

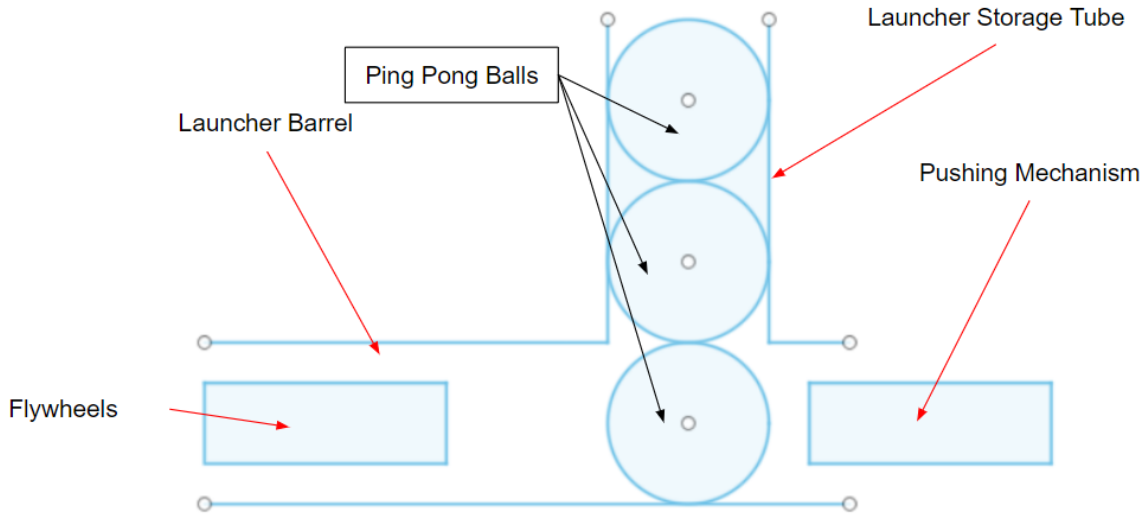
Upon clarification of mission parameters, the launcher design was updated so that the launcher barrel is no longer vertically angled with respect to the ground, and is instead parallel with the ground. This is because the target will be of a fixed height and low to the ground, and hence angling of the launcher is not required. This causes the distance of the launcher to be roughly fixed, but this can be adjusted via moving the Turtlebot3 towards or away from the target. Due to the lack of offset, the flywheels were swapped to be in the horizontal plane rather than the vertical to reduce the height of the assembly.

In addition, the storage tube for balls concept was also refined. The height of the storage tube has to be lower than the LIDAR to avoid interference of LIDAR readings, and there is no requirement on the number of balls to be loaded. The storage tube was hence changed to a vertical feed to improve reliability of feed and reduce the size of the assembly. The ultrasonic sensor concept was removed in favour of a button for the loader to press once done loading, and the number of balls will be stored as a value in the RPi rather than physically measured.

The pushing mechanism concept was also tested with a servo motor attached with an arm, to push balls into the flywheel via rotation. This design was deemed effective and feasible, and was chosen over the original concept to use a linear actuator, due to the added cost, complexity and weight of the motor and linear actuator compared to a servo motor.

The final design was refined to 4 main components, as in the picture below:

1. Launcher Storage Tube - Simple construction to hold balls in a vertical configuration and guide them to feed into the barrel
2. Launcher Barrel - Ensures balls are shot out straight
3. Pushing Mechanism - Pushes balls from storage tube into barrel
4. Flywheel Mechanism - Motor and attached flywheel to propel ball



6.4 Flywheel Test

Initially, the flywheel was designed in CAD such that the distance between the flywheels at their closest point would be exactly equal to the diameter of a ping pong ball (38mm []). Upon testing, however, balls pushed through the flywheel were not shot out consistently or straight. A new set of slightly larger flywheels were manufactured, and an anti-slip rubber layer was added to the flywheel circumference to increase the friction between the flywheel and ping pong ball. The new flywheels were able to launch the ping pong balls relatively consistently, straight ahead and to the maximum distance listed in the specifications in 7.1.

6.5 Stopper Screw

When more than one ball is loaded into the storage tube, the angle of the tube and weight of the balls on top would cause the first ball to be pushed into the flywheels without the SG90 servo pusher arm moving. This is detrimental to the performance of the launcher as it does not allow the flywheels to gain sufficient rotational velocity before pushing the ball into the flywheel and shooting it, hence the first ball shot out may be inconsistent and inaccurate. This was remedied by replacing the M3x4 screw fastening the launcher barrel to middle plate with an M3x6 screw. The added protrusion of the M3x6 screw blocks the first ball from rolling into the flywheels until pushed by the servo arm.

7. Final Design

7.1 Key Specifications

List	Specifications	Note
Size (mm)	255 x 176 x 187 (L x W x H)	Full assembly (Turtlebot3 + Launcher Assembly)
Weight (kg)	1.43	
Wheel Base (mm)	90.16	From Rear Driving Axle (DYNAMIXEL) to Front Axle (Front Ball Caster)
Steer & Drive Mode	Rear wheel drive and steering	
Drive Actuator	2x XL430-W250 DYNAMIXEL	
Maximum Translational Speed	0.22 m/s	
Maximum Rotational Speed	2.84 rad/s (162.72 deg/s)	
Sensors	Thermal Camera 1x AMG8833 Thermal Camera LIDAR 1x 360° LDS-01 NFC Reader 1x PN532 NFC Reader Odometry 1x ICM-20648 (On OpenCR 1.0)	
Buttons & Switches	1x Push button	In addition to Turtlebot3 pre-existing buttons
Battery Capacity	LiPo Battery 11.1V 1,800mAh	
Expected Operating Time	2hr 21min	Refer to Annex F
Communication interface	I^2C, USB2LDS, GPIO	I^2C AMG8833 and PN532 to RPi USB2LDS LDS to RPi GPIO Push Button
Launcher		
Flywheel Rotational Speed (rpm)	9600	Bot on flat ground
Maximum Launch Distance (m)	1.2	
Maximum Launcher Payload	3 Ping Pong Balls	Refer to Loading Instructions
Actuators	Feeder 1x SG90 Servo Motor Flywheel 2x MM28 DC Motor	

For more information on Turtlebot3 Specifications, refer to [22]. The Product Compliances of the system are listed in Annex K.

7.2 System Finances

A Turtlebot3 set was provided to create the system, as well as miscellaneous fasteners and electronic components. A budget of up to $100 was provided for supplementary parts. The breakdown of parts and their costs is as indicated below in the bill of materials.

Bill of Materials

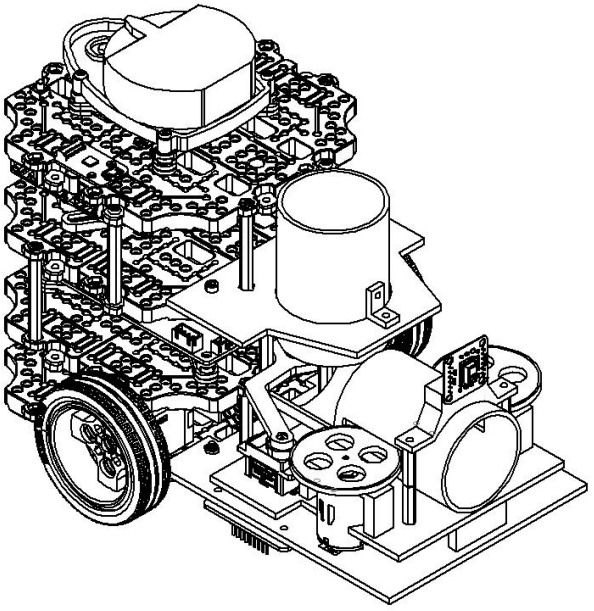
No.	Part	QTY	Unit Cost	Total Cost
Turtlebot3 (Mechanical)				
1	Waffle Plate	8	Provided	Provided
2	Wheel	2		
3	Tire	2		
4	M2.5 Nuts (0.45P)	20		
5	M3 Nuts	16		
6	Spacer	4		
7	Rivet (Short)	14		
8	Rivet (Long)	2		
9	Plate Support M3x35	4		
10	Plate Support M3x45	10		
11	Adaptor Plate	1		
12	Adaptor Bracket	5		
13	PCB Support	12		
14	PH_M2x4mm K	8		
15	PH_M2x6mm K	4		
16	PH_M2.5x8mm K	16		
17	PH_T 2.6x12mm K	16		
18	PH_M 2.5x16mm K	4		
19	PH_M 3x8mm K	44		
20	Rear Ball Caster (w/ Ball)	1		
Turtlebot3 (Electrical)				
21	Li-Po battery	1	Provided	Provided
22	Li-Po battery charger	1		
23	USB Cable	2		
24	Dynamixel to OpenCR Cable	2		
25	Raspberry Pi 3 Power Cable	1		
26	Li-Po Battery Extension Cable	1		
27	SMPS	1		
28	AC-Cord	1		
29	Prototyping Board	1		

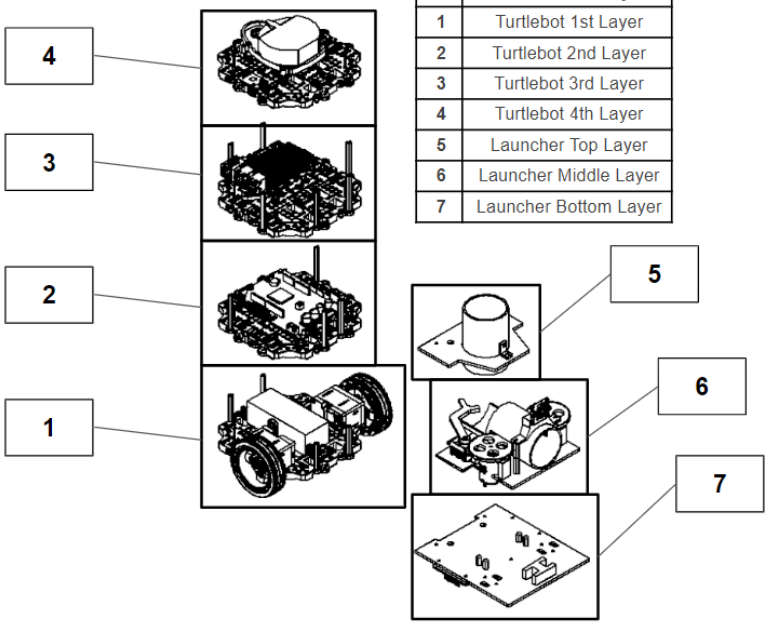
30	Dynamixel XL 430	2		
Turtlebot3 (for Software)				
31	OpenCR 1.0	1	Provided	Provided
32	Raspberry Pi 3	1		
33	360 Laser Distance Sensor LDS-01	1		
34	USB2LDS	1		
35	Push Button	1		
Launcher (Mechanical)				
	Laser Cut Acrylic Plate			
1	Bottom Plate	1	8.1	8.1
2	Middle Plate	1		
3	Top Plate	1		
	3D Printed Parts			
4	Flywheel	2	1.8	3.6
5	Motor Bridge	2	2.5	5
6	Thermal Header	1	2.2	2.2
7	Pusher Arm	1	1.6	1.6
	PVC Pipe			
8	Launcher Storage Tube	1	4.9	4.9
9	Launcher Barrel	1		
	Bolts/Nuts/Spacers			
10	M2x6 Bolt	2	Provided	Provided
11	M3x8 Bolt	22		
12	M3x16 Bolt	4		
13	M3 Nut	24		
14	M2x10 Spacer	1		
15	M3x10 Spacer	4		
16	M3x40 Spacer	2		
	Others			
17	Front Ball Caster	1	1	1
18	Anti-slip Racket Tape	1	2.15	2.15
Launcher (Electrical)				
19	MM28 DC Motor	2	6.2	12.4
20	SG90 Servo Motor	1	Provided	Provided
21	PN532 NFC Reader	1	Provided	Provided
22	AMG8833 Thermal Camera	1	44.95	44.95
Grand Total				85.9

8. Assembly Instructions

8.1 Mechanical Assembly

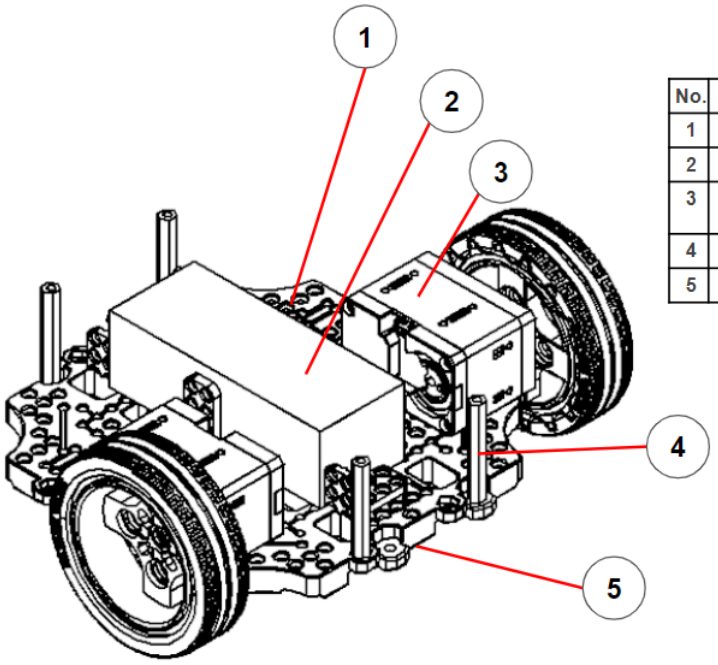
8.1.1 Overview of Assembly





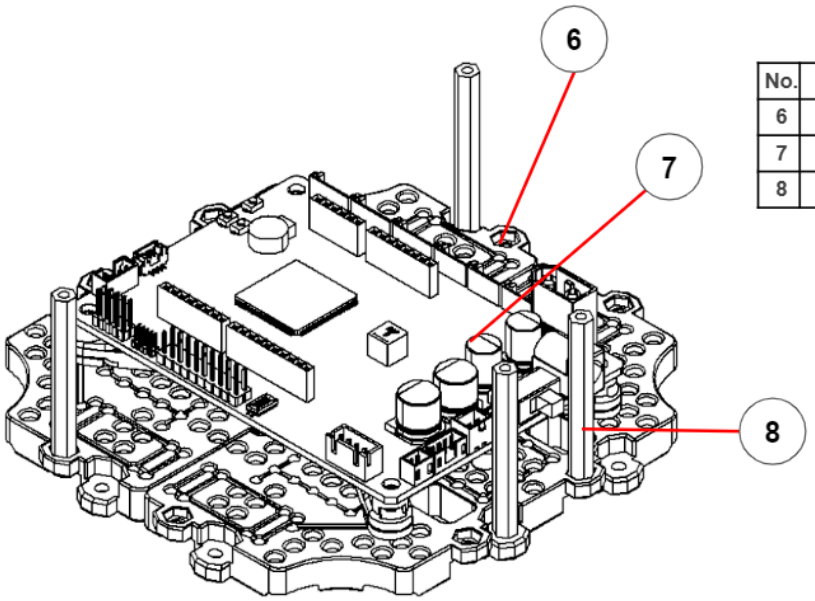
No.	Sub-Assembly
1	Turtlebot 1st Layer
2	Turtlebot 2nd Layer
3	Turtlebot 3rd Layer
4	Turtlebot 4th Layer
5	Launcher Top Layer
6	Launcher Middle Layer
7	Launcher Bottom Layer

8.1.2 Turtlebot3 1st Layer



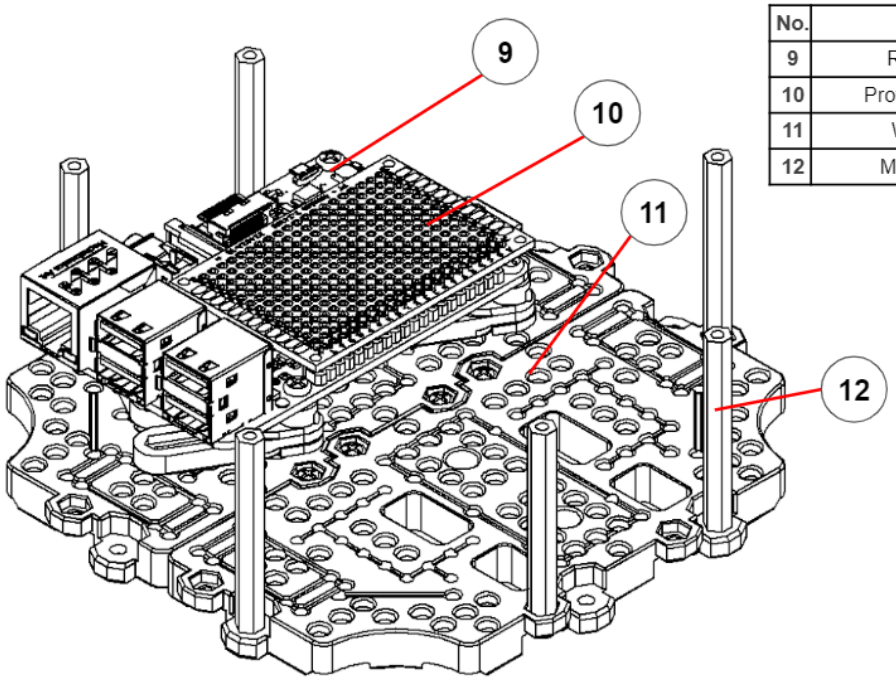
No.	Part	QTY
1	Waffle Plate	1
2	Li-Po Battery	1
3	XL430-W250 DYNAMIXEL + Rubber Wheel	2
4	M3x35 Spacer	4
5	Rear Ball Caster	1

8.1.3 Turtlebot3 2nd Layer



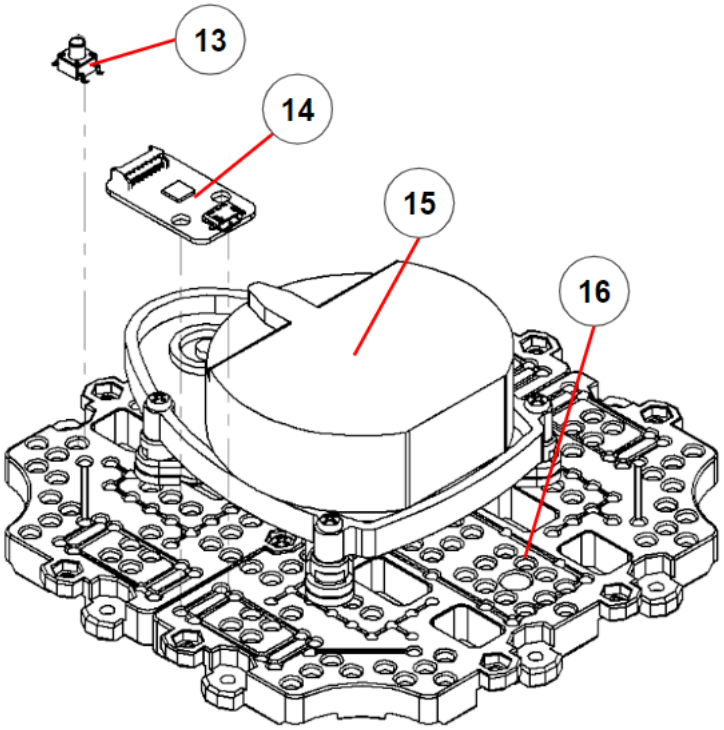
No.	Part	QTY
6	Waffle Plate	1
7	OpenCR1.0	1
8	M3x45 Spacer	4

8.1.4 Turtlebot3 3rd Layer



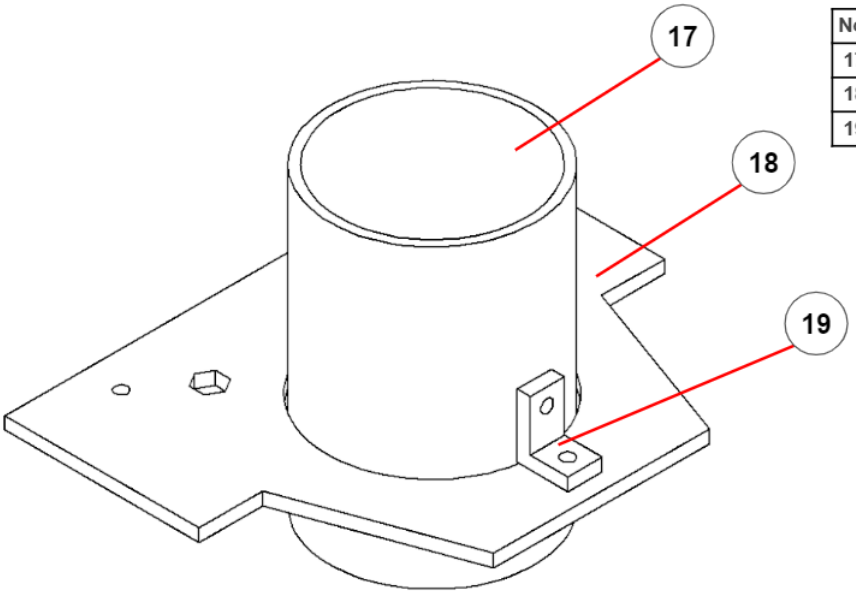
No.	Part	QTY
9	Raspberry Pi	1
10	Prototyping Board	1
11	Waffle Plate	1
12	M3x45 Spacer	6

8.1.5 Turtlebot3 4th Layer



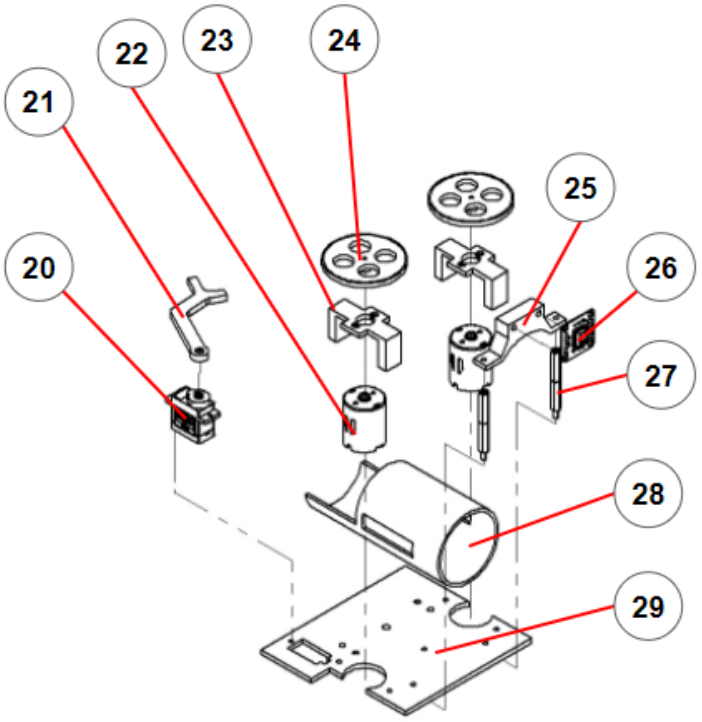
No.	Part	QTY
13	Push Button	1
14	USB2LDS	1
15	LDS-01 LIDAR	1
16	Waffle Plate	1

8.1.6 Launcher Top Layer



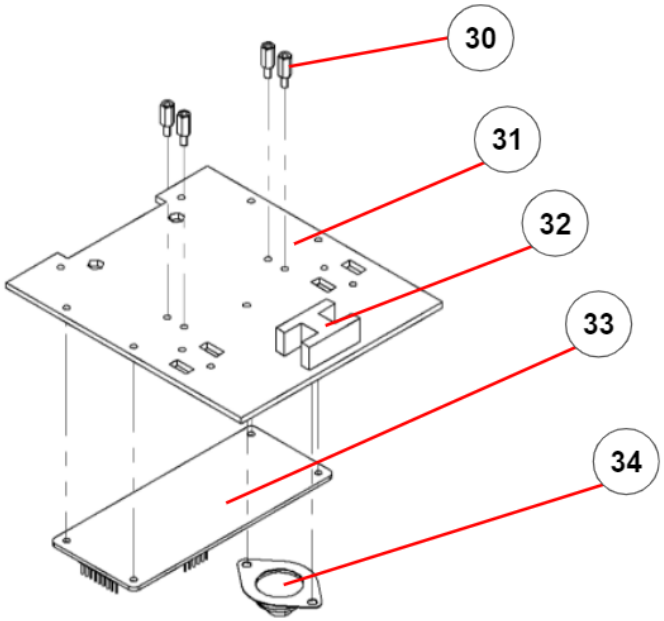
No.	Part	QTY
17	Launcher Storage Tube	1
18	Top Acrylic Plate	1
19	Storage Tube Bracket	1

8.1.7 Launcher Middle Layer



No.	Part	QTY
20	SG90 Servomotor	1
21	Servomotor Pusher Arm	1
22	MM28 DC Motor	2
23	Flywheel Motor Bridge	2
24	Flywheel	2
25	Thermal Header	1
26	AMG8833 Thermal Camera	1
27	M3x40 Spacer	2
28	Launcher Barrel	1
29	Middle Acrylic Plate	1

8.1.8 Launcher Bottom Layer



No.	Part	QTY
30	M3x10 Spacer	4
31	Bottom Acrylic Plate	1
32	Front Support	1
33	PN532 NFC Reader	1
34	Front Ball Caster	1

8.2 Electronics Assembly

8.2.1 The Schematic diagram of the prototyping board

The diagram below shows the schematic of the protoboard that was attached to the RPi as a hat to connect the RPi to the sensors. A picture of the actual protoboard used is attached in Annex J.

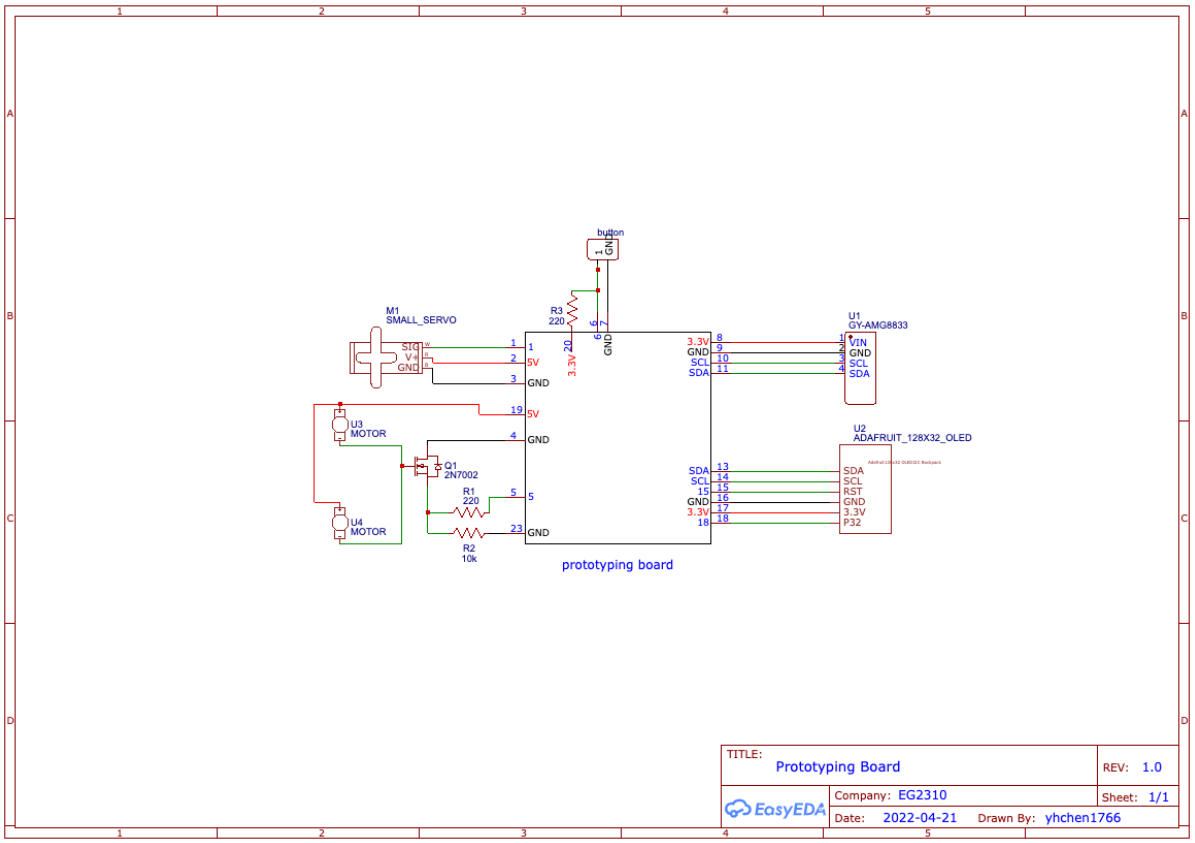


Figure 8.2.1 The schematic diagram of protoboard

A protoboard was utilised in the final design of the Turtlebot3 to allow for neater cable management and to reduce the chances of connecting the sensors to the wrong GPIO pins. The exact configuration of the sensors to the GPIO pins is elaborated below, with the pin numbering convention used being the Broadcom SOC Channel (BCM).

The push button is connected to GPIO 10, with a pull-up resistor of 220Ω connected between the 3.3V and the button.

The SDA and SCL Adafruit PN532 NFC Breakout Board are connected to the GPIO 2 and 3 of the RPi respectively, which allows the NFC Breakout board to connect to the RPi using I2C. Additionally, the RSTPD_N and P32 pins on the breakout board are connected to GPIO 20 and GPIO 16 on the RPi respectively. It is powered by the 3.3V pin of the RPi

The SDA and SCL of the AMG8833 Thermal Camera are also connected to GPIO 2 and 3 of the RPi since it also communicates with the RPi using I2C. It is powered by the 3.3V pin of the RPi.

The servo motor used to push the ball into the flywheels is connected to GPIO 23 on the RPi, and it is powered up by the 5V pin on the RPi.

The two flywheels used to shoot the ball out are powered by the 5V pin of the RPi, and both the motors are connected to a mosfet. A pullup resistor of 220Ω is connected between the mosfet and GPIO 21, which is used to switch on the flywheels, and a bleeding resistor of 10KΩ is connected between the mosfet and the GND pin of the RPi.

All the components are connected to the common ground on the RPi.

8.2.2 The connection between the protoboard and RPi

The protoboard is placed on the RPi as a hat which allows the system to be compact and eliminates the need to mount the protoboard. Placing the protoboard as a hat also ensures that the components will be connected to the correct pins of the RPi.

The OpenCR which is connected to the RPi provides the 5V and 3.3V that is supplied to the protoboard, and its ground is connected to the GND pin on the RPi.

The connection between these pins are shown in Figure 8.2.2.

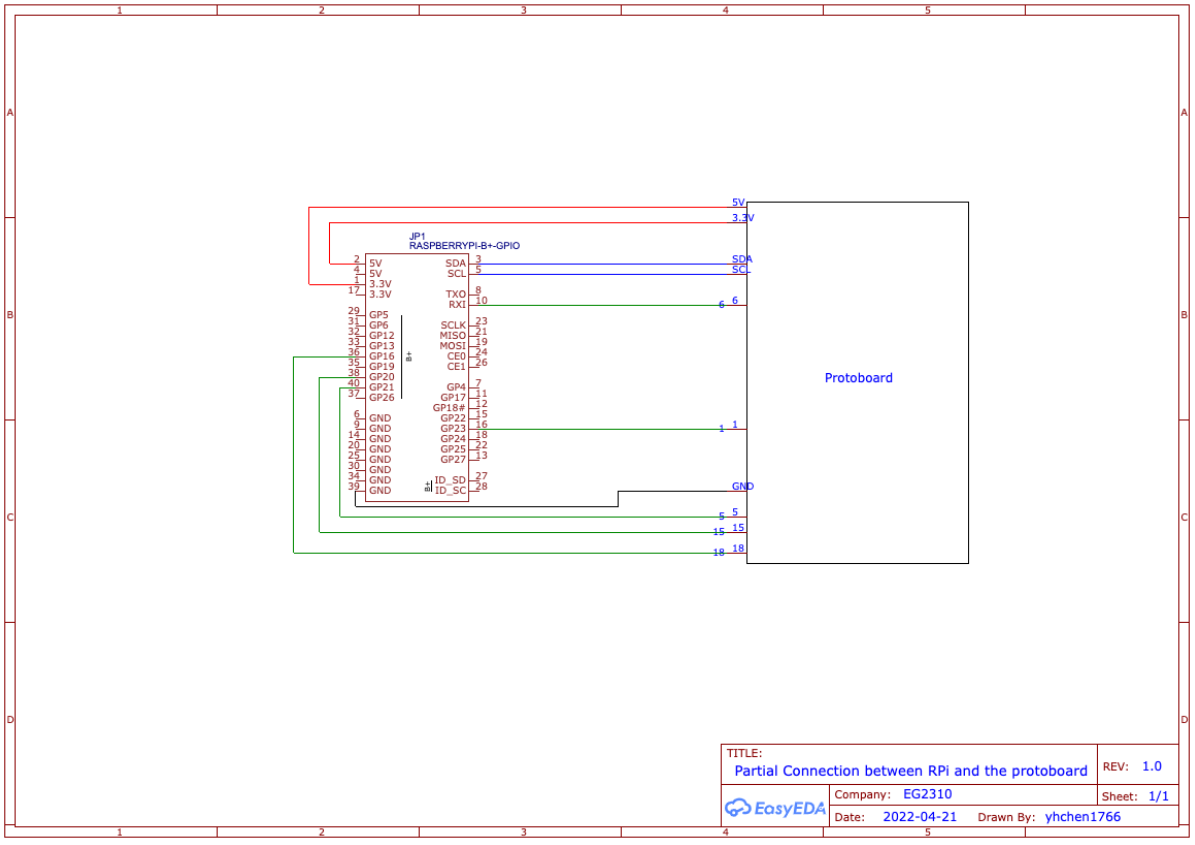


Figure 8.2.2 Connection between the RPi and the protoboard

8.3 Software Assembly

8.3.1 Setting up the devices

1. On the laptop, ensure that you have Ubuntu 20.04 and ROS 2 Foxy installed. Refer here on how to install the required software. Ensure that you are following the instructions under the "Foxy" tab.

2. Test that the ROS development environment is working by ensuring that a simple working publisher and subscriber can be created using the instructions here.

3. Using Ubuntu, follow the instructions here, burn the ROS 2 Foxy Image to the SD card onto the RPi on the Turtlebot3. Follow through the "Quick Start Guide" to get a working Turtlebot3.

4. Test that the ROS development environment is working by ensuring that a simple working publisher and subscriber can be created using the instructions here.

5. Once the ROS development environment is working on both the remote laptop and the RPi, run the publisher from the RPi and the subscriber on the laptop, and ensure that the subscriber on the laptop replicates what is being produced by the publisher. Swap the device that the publisher and subscriber are publishing from to ensure that two-way communication between the RPi and the remote laptop can be established.

6. Add the following lines to *.bashrc* of the remote laptop.

```
export TURTLEBOT3_MODEL=burger
alias rteleop='ros2 run turtlebot3_teleop teleop_keyboard'
alias rslam='ros2 launch turtlebot3_cartographer cartographer.launch.py’
```

7. Add the following lines to *.bashrc* of the RPi.

```
export TURTLEBOT3_MODEL=burger
alias rosbu='ros2 launch turtlebot3_bringup robot.launch.py'
```

8.3.2 Installing the program on the remote laptop

1. Create a ROS 2 package on the remote laptop.

```
cd ~/colcon_ws/src
ros2 pkg create --build-type ament_python auto_nav
cd auto_nav/auto_nav
```

2. Move the file in the directory temporarily to the parent directory.

```
mv __init__.py ..
```

3. Clone the GitHub repository to the remote laptop. Make sure the period at the end is included.

```
git clone git@github.com:jaredoong/r2auto_nav.git .
```

4. Move the file __init__.py back.

```
mv ../__init__.py .
```

5. Remove the unnecessary files in the workspace.

```
rm -rf amg8833 Archive RPi_files test_sensors
```

6. Move the directory ‘custom_msgs’ under ‘/colcon_ws/src’.

```
mv custom_msgs ~/colcon_ws/src
```

7. Build the ‘auto_nav’ and ‘custom_msgs' package on the laptop.

```
cd ~/colcon_ws && colcon build
```

8. Add the following line in the .bashrc file on the laptop.

```
alias start_auto='ros2 run auto_nav r2wall_follower'
```

8.3.3 Installing the program on the RPi

1. Create a ROS 2 package on the RPi.

```
cd ~/turtlebot3_ws/src
ros2 pkg create --build-type ament_python sensors
cd sensors/sensors
```

2. Remove the file in the directory temporarily to the parent directory.

```
rm __init__.py
```

3. Clone the GitHub repository to the remote laptop. Make sure the period at the end is included.

```
git clone git@github.com:jaredoong/r2auto_nav.git .
```

4. Remove the unnecessary files in the workspace.

```
rm -rf Archive .git .gitattributes __init__.py package.xml __pycache__
r2wall_follower.py README.md setup.py
```

5. Move the directory ‘custom_msgs’ under ‘/turtlebot3_ws/src’.

```
mv custom_msgs ~/turtlebot3_ws/src
```

6. Move the directory ‘RPi_files/sensors’ out to the home directory.

```
mv test_sensors ~
```

7. Move into the directory ‘RPi_files/sensors/sensors’ and move the files into the sensors package.

```
cd RPi_files/sensors/sensors && mv * ~/turtlebot3_ws/src/sensors/sensors
```

8. Exit to the parent directory and delete the empty ‘sensors’ directory.

```
cd .. && rm -rf sensors
```

9. Move the remaining files into the sensors package.

```
mv * ~/turtlebot3_ws/src/sensors
```

10. Delete the empty directory ‘RPi_files’.

```
cd ~/turtlebot3_ws/src/sensors/sensors && rm -rf RPi_files
```

11. Build the ‘sensors’ and ‘custom_msgs’ package on the RPi.

```
cd ~/turtlebot3_ws && colcon build
```

12. Add the following line the .bashrc file on the RPi.

```
alias start_sensors='ros2 run sensors run_sensors'
```

8.3.4 Calibration of Parameters

The start of ‘r2wall_follower.py’ on the laptop consists of constants that can be changed to fit the needs of the mission.

```
# calibration parameters
slow_rotate = 0.7 # for rotating the bot slowly
fast_rotate = 0.9 # for rotating the bot quickly
speed_change = 0.20 # forward speed of the bot
stop_distance = 0.35 # stopping distance of the bot
threshold_temp = 35 # calibrated to temp of thermal object
total_nfc = 3 # number of detectable NFC in 1 round
```

Similarly, ‘sensors.py’ on the RPi also consist of constants that can be changed to fit the configuration of the sensors to the Turtlebot3.

```
num_balls = 3 # number of balls loaded
total_nfc = 3 # number of detectable NFC in maze
```

The breakdown of the parameters are shown in the table below.

On the laptop	
Parameter	Description
`slow_rotate`	Speed of the bot while it is rotating slowly. Positive value indicates rotation to the left, negative value indicates rotation to the right
`fast_rotate`	Speed of the bot while it is rotating quickly. Positive value indicates rotation to the left, negative value indicates rotation to the right
`speed_change`	Forward speed of the bot
`stop_distance`	Minimum allowable distance between the LDS and the wall
`threshold_temp`	Temperature of the heated object
`total_nfc`	Total number of detectable NFC in the maze
On the RPi	
Parameter	Description
`num_balls`	Number of balls loaded into the turtlebot
`total_nfc`	Total number of detectable NFC in the maze

8.3.5 Changing the parameters on the laptop

`slow_rotate` / `fast_rotate` - If the rotation speed of the Turtlebot3 is too fast, decrease the values till the desired rotating speed is achieved.

`speed_change` - If the Turtlebot3 is moving too fast, decrease the value till the desired forward speed is reached. Take note that the default value of 0.20 is the maximum speed of the Turtlebot3 and increasing this value any further slows the Turtlebot3 down.

`stop_distance` - The value can be increased or decreased to allow the Turtlebot3 to follow the wall from a larger or shorter distance respectively.

`threshold_temp` - In the event that the Turtlebot3 is unable to detect the heated object, decrease the value. However, ensure that the value is not too close to the room temperature as doing so can result in the Turtlebot3 triggering wrongly.

`total_nfc` - Due to the speed of the Turtlebot3, there is a possibility that it is unable to detect the NFC tag when travelling over it. Hence, it is important to do a test run beforehand by allowing the Turtlebot3 to run over each of the loading zones and see whether it is able to detect the NFC at that location. The value can then be changed to the total number of detectable NFC in the maze. This value needs to be changed in both the r2wall_follower.py on the laptop and the sensors.py on the RPi.

After changing the parameters, it is important to rebuild the package and source again before running the program, using the following command.

```
cd ~/colcon_ws && colcon build --symlink-install && source ~/.bashrc
source install/setup.bash
```

8.3.6 Changing the parameters on the RPi

`num_balls` - Change this based on the desired number of balls to be loaded into the Turtlebot3. This determines that number of times that the servo motor pushes the ball forward

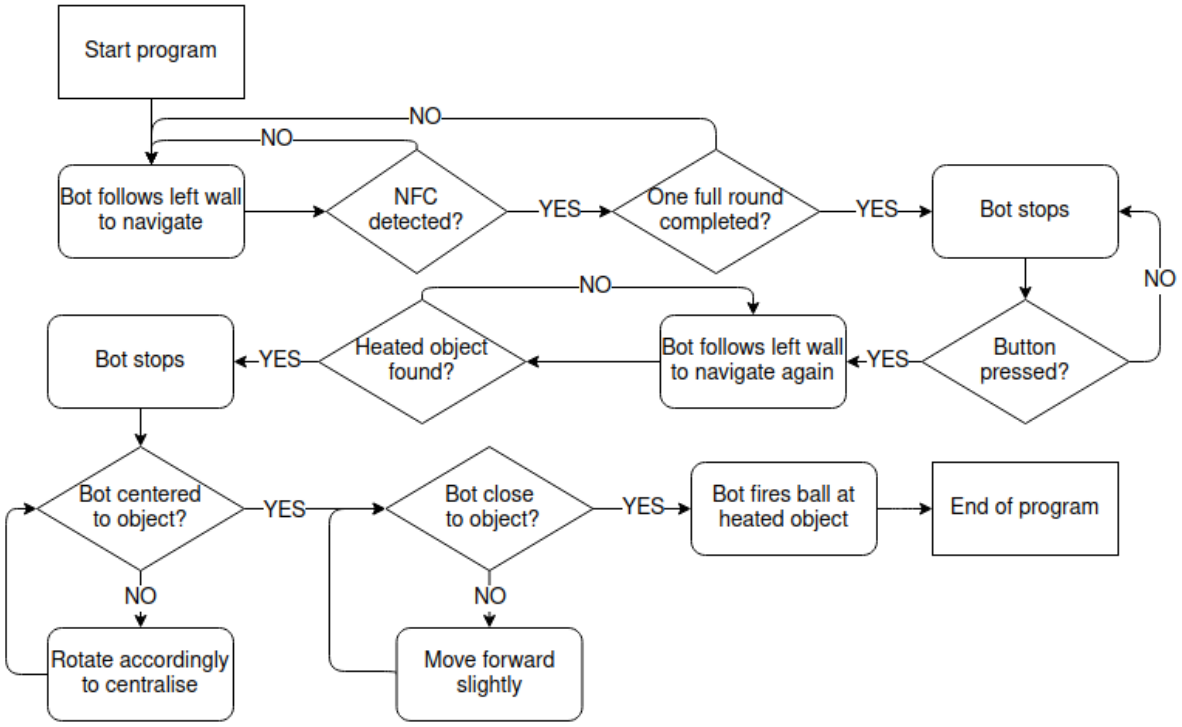
`total_nfc` - Same as that on the laptop

After changing the parameters, it is important to rebuild the package and source again before running the program, using the following command

```
cd ~/turtlebot3_ws && colcon build --symlink-install && source ~/.bashrc
source install/setup.bash
```

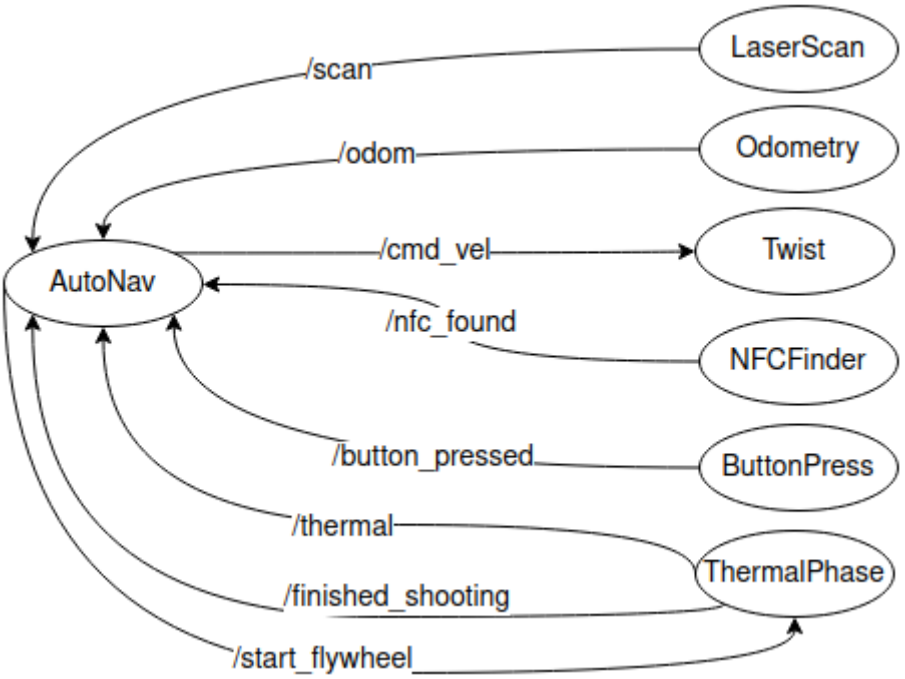
8.4 Overview of Algorithm

The diagram below shows the high level overview of the algorithm used. For the mission, the bot will complete one around the maze first in order to map the entire maze out. After it has completed one full round, the bot would then stop at the loading zone once it detects the NFC tag on the floor. The bot will then remain stationary till the button is pressed, which indicates that all the balls have been loaded. The bot then resumes its navigation to find the heated object. Once the heated object is detected, the bot would first centralise itself to the object, before moving forward. When the bot is close enough to the heated object, the flywheel motors and the servo would then be started up to fire the balls at the heated object. Upon the completion of the mission, the program on the RPi and the laptop would end.



8.4.1 Communication between laptop and RPi

The diagram below shows the nodes and topics used by the programs on the laptop and RPi in order to transmit data.



The `AutoNav` node on the left runs from the laptop while all the other nodes on the right run on the RPi. The arrows represent the topics and they show the direction of the data transfer, from the publisher to the subscriber. The data generated by the LDS is transmitted over the topic `/scan` while the odometry data from the OpenCR is sent back to the laptop over `/odom`. Rviz running on the laptop uses the data incoming from these two topics in order to map out the maze. The `AutoNav` node publishes data over the `/cmd_vel` topic in order to command the Turtlebot3 to navigate the maze. For the first part of the mission of finding the NFC, the `NFCFinder` node publishes over the topic `/nfc_found` on whether an NFC has been detected, with `True` representing an NFC being detected and `False` representing no NFC detected. For the loading phase of the mission, the `ButtonPress` node publishes over the topic `/button_pressed` a value of `False` by default, switching over to `True` only once the button has been pressed. For the last part of the mission which includes finding the heated object and shooting at it, the `ThermalPhase` node publishes an 8x8 array of data from the thermal camera over the topic `/thermal`. When the heated object is in alignment and close enough to the Turtlebot3, the `AutoNav` node would then publish a value of `True` over the topic `/start_flywheel` that the `ThermalPhase` node is subscribed to. Upon receiving the `True` value over `/start_flywheel`, the Turtlebot3 will begin the shooting of the balls at the target. Once all the balls have been fired, the ThermalPhase node would publish `True` over the topic `/finished_shooting` to signal to the `AutoNav` node that the mission is completed, and the program on both the laptop and RPi would destroy the remaining nodes and exit.

8.4.2 Detailed breakdown of program on the laptop

```python
def main(args=None):
    rclpy.init(args=args)

    auto_nav = AutoNav()
    auto_nav.find_nfc()
    auto_nav.load_balls()
    auto_nav.find_thermal()
    auto_nav.launcher()

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    auto_nav.destroy_node()
    rclpy.shutdown()
```

Once the program is started, an `AutoNav` node is created, following which the function `find_nfc()` is called.

```python
def find_nfc(self):
    try:
        # keep track of number of NFC is has detected
        num_nfc_found = 0

        # ensure data being received from LIDAR before starting
        while (len(self.laser_range) == 0):
            self.get_logger().info("Fetching LIDAR data")
            rclpy.spin_once(self)

        # Start wall following algorithm once ready
        while rclpy.ok():
            self.left_follow_wall(stop_distance, speed_change, slow_rotate, fast_rotate)
            if self.nfcfound == True:
                self.stopbot()
                self.get_logger().info("NFC found")
                num_nfc_found += 1
                self.get_logger().info("Num of NFC found: %i" % num_nfc_found)
                if num_nfc_found > total_nfc:
                    # move into loading phase only when bot has travelled one full round
                    break

            # allow the callback functions to run
            rclpy.spin_once(self)

    except Exception as e:
        print(e)

    # Ctrl-c detected
    finally:
        # stop moving
        self.stopbot()

```

The function `find_nfc()` takes zero arguments. Once it is called, it would first wait for the data from the LDS. Once it is able to receive the data, the bot would follow the left wall to navigate around the maze. The function `left_follow_wall()` takes four arguments, which are the stopping distance, the forward speed of the bot, and the slow and fast rotation speed of the bot. In the function `left_follow_wall()`, the bot will check the distance from the wall in front, 45 degree to the left, and 45 degree to the right using the data that is returned from the LDS. The bot will then move accordingly to keep the wall on its left while ensuring that it does not collide into any wall. The bot then checks whether an NFC tag has been detected by the NFC reader. If no NFC tag has been detected, the bot will then loop back into the function `left_follow_wall()` and this process repeats itself till an NFC tag is detected. Once an NFC tag is detected, the bot would then proceed to check if it has completed one full round of the maze. This is done by checking whether the number of detected NFC has exceeded the total number of detectable NFC in the maze. If the bot has not completed a full round, it would then continue navigating around the maze. Once the bot checks that it has completed a full round, it would stop moving and break out of the function `find_nfc()` before moving into the next function, `load_balls()`.

```
def load_balls(self):
    self.get_logger().info("Loading balls phase started")
    try:
        while rclpy.ok():
            # allow the callback functions to run
            rclpy.spin_once(self)

            # bot remains stationary till balls are loaded
            if self.buttonpressed == False:
                self.stopbot()
                continue
            self.get_logger().info("Balls loaded, moving off in 2 seconds")
            # 2 sec delay added to allow TA to move out of the way
            time.sleep(2.0)
            self.get_logger().info("Moving off now")
            break

    except Exception as e:
        print(e)

    # Ctrl-c detected
    finally:
        # stop moving
        self.stopbot()
```

The function `load_balls()` takes zero arguments. Once it is called, the bot will constantly check whether the button has been pressed. As long as the button is not pressed, the bot will remain stationary. Once the button has been pressed, the bot will then wait for two seconds before breaking out of this function and move on to the next function `find_thermal()` in order to find the heated object.

```python
def find_thermal(self):
    try:
        # Start and Format Figure
        plt.rcParams.update({'font.size':16})
        fig_dims = (12,9) # figure size
        fig,ax = plt.subplots(figsize=fig_dims) # start figure
        pix_res = (8,8) # pixel resolution
        im1 = ax.imshow(self.thermalimg,vmin=15,vmax=35) # plot image, with temperature bounds
        cbar = fig.colorbar(im1,fraction=0.0475,pad=0.03) # colorbar
        cbar.set_label('Temperature [C]',labelpad=10) # temp. label
        fig.canvas.draw() # draw figure
        ax_bgnd = fig.canvas.copy_from_bbox(ax.bbox) # background for speeding up runs
        fig.show() # show figure

        while rclpy.ok():
            # allow the callback functions to run
            rclpy.spin_once(self)

            # prevent flywheel from starting before target is found
            flywheel = Flywheel()
            flywheel.start_flywheel = False
            self.publisher_flywheel.publish(flywheel)

            # waits for thermal data to be updated
            if self.thermal_updated == False:
                # self.get_logger().info("Waiting for new data")
                continue

            # Plotting in real time
            # self.get_logger().info('Redrawing image')
            fig.canvas.restore_region(ax_bgnd) # restore background (speeds up run)
            im1.set_data(np.reshape(self.thermalimg,pix_res)) # update plot with new temps
            ax.draw_artist(im1) # draw image again
            fig.canvas.blit(ax.bbox) # blitting - for speeding up run
            fig.canvas.flush_events() # for real-time plot
            #self.get_logger().info('Done redrawing image')
```

The function `find_thermal()` takes zero arguments. Once it is called, it would first draw up a figure which would display the data that is being received from the thermal camera. Whenever new data is received, the figure would update itself accordingly to reflect the most updated data.

```
# transpose image so that can check by columns
thermal_data = np.transpose(self.thermalimg)
# reset updated_variable once data has been used
self.thermal_updated = False

cols_found = []
row_num = 0
for row in thermal_data:
    #self.get_logger().info("Checking col %i" % row_num)
    col_num = 0
    for temp in row:
        #self.get_logger().info("Checking row %i, Temp is %.2f" % (col_num, temp))
        if temp >= threshold_temp:
            cols_found.append(row_num)
            break
        col_num += 1
    row_num += 1
# for checking with columns the object detected in
# print(("Columns found: {}").format(cols_found))

if len(cols_found) != 0:
    self.thermalfound = True
else:
    self.thermalfound = False

if self.thermalfound == False:
    self.get_logger().info("Thermal object not yet found")
    # do wall following algo to find thermal object
    self.left_follow_wall(stop_distance, speed_change, slow_rotate, fast_rotate)

else:
    # if thermal object is found, stop the bot
    self.stopbot()
```

The new data received would then be processed and checked. If the heated object has not been detected, the bot will continue navigating around the maze by following the left wall. In the event that the heated object is found, the bot will stop before moving on to align itself to the heated object.

```python
# adjusting of position of bot relative to thermal object
if len(cols_found) != 0:
    # use the middle column to align
    reference_col = cols_found[int(len(cols_found) / 2)]
    if reference_col  < 3:
        # turn left to centralise the object
        twist = Twist()
        twist.linear.x = 0.0
        twist.angular.z = 0.2*slow_rotate
        time.sleep(1)
        # self.get_logger().info("Turning left to centralise bot")
        self.publisher_.publish(twist)
        self.centered = False
        time.sleep(0.5)
    elif reference_col > 4:
        # turn right to centralise the object
        twist = Twist()
        twist.linear.x = 0.0
        twist.angular.z = -0.2*slow_rotate
        time.sleep(1)
        # self.get_logger().info("Turning right to centralise bot")
        self.publisher_.publish(twist)
        time.sleep(0.5)
        self.centered = False
    else:
        self.centered = True
        self.stopbot()

if self.centered:
    self.get_logger().info("Centralised")
    # if too far away, move closer to object
    if (len(cols_found) < 3):
        twist = Twist()
        twist.linear.x = speed_change*0.5
        twist.angular.z = 0.0
        time.sleep(1)
        self.publisher_.publish(twist)
        time.sleep(1)
    else:
        self.stopbot()
        self.get_logger().info("Correct distance away from object")
        # move on to shooting phase once ready
        break
```

The bot would check whether the object detected lies within the centre columns of the thermal camera, and it would rotate itself accordingly till the object lies in the centre. Once centred, the bot would then move forward in increments till it detects that the heated object is close enough. The object is determined to be close enough when it fills up more than three columns of the thermal camera. Once close enough, this function would then be exited and the program would move on to the last function `launcher()`.

```python
# function to start up the launcher and check if shooting is done
def launcher(self):
    try:
        # Start up the flywheels
        flywheel = Flywheel()
        flywheel.start_flywheel = True
        self.publisher_flywheel.publish(flywheel)

        while rclpy.ok():
            rclpy.spin_once(self)
            # self.get_logger().info("Waiting for shooting to be finished")
            if self.done_shooting == True:
                self.get_logger().info("Done shooting")
                break

    except Exception as e:
        print(e)

    # Ctrl-c detected
    finally:
        # stop moving
        self.stopbot()
```

The function `find_thermal()` takes zero arguments. Once it is called, it would first start up the flywheel motors. It would then wait for the servo to fire all the balls out. Once all the balls have been fired, the program would then exit the function and come to an end.

8.4.3 Detailed breakdown of the program on the RPi

```
button_pin = 15
left_flywheel_pin = 21
right_flywheel_pin = 26
servo_pin = 23
num_balls = 3
total_nfc = 3 # number of detectable NFC in maze

GPIO.setmode(GPIO.BCM)
# Set pin 10 to be an input pin and
# set initial value to be pulled low (off)
GPIO.setup(button_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
# Set pin 37 to be an output pin,
# switches to high to start flywheels only when ready
GPIO.setup(left_flywheel_pin, GPIO.OUT, initial=0)
# Set pin 40 to be an output pin,
# switches to high to start flywheels only when ready
GPIO.setup(right_flywheel_pin, GPIO.OUT, initial=0)

# pigpio used to prevent servo jitter
p = pigpio.pi()
p.set_mode(servo_pin, pigpio.OUTPUT)
p.set_PWM_frequency(servo_pin, 50)

# Ranges from 500 (0) to 2500 (180)
# Setting the initial angle at 0
p.set_servo_pulsewidth(servo_pin, 1400)
```

At the start of the program, the GPIO pins used for the additional hardwares such as the sensors, motors and button are defined. The number of balls to be loaded and NFC in the maze are also predetermined. The numbering is set to BCM mode, and the button and flywheel GPIO pins are set to input and output pins respectively. The servo is moved back to the starting position in preparation for the launching of the balls at the end.

```python
def main(args=None):
    rclpy.init(args=args)

    # start with finding NFC
    nfc_pub = NFCFinder()
    nfc_pub.nfc_callback()
    nfc_pub.destroy_node()

    # once NFC found, wait for button press
    button_pub = ButtonPress()
    button_pub.button_callback()
    button_pub.destroy_node()

    # start the thermal function once button pressed
    thermal_pub = ThermalPhase()
    thermal_pub.start_thermal()
    thermal_pub.destroy_node()

    rclpy.shutdown()
```

Once the program on the RPi is started, an `NFCFinder` node is created and the function `nfc_callback()` is called.

```python
def nfc_callback(self):
    try:
        #pn532 = PN532_SPI(debug=False, reset=20, cs=4)
        pn532 = PN532_I2C(debug=False, reset=20, req=16)
        #pn532 = PN532_UART(debug=False, reset=20)

        #ic, ver, rev, support = pn532.get_firmware_version()
        #print('Found PN532 with firmware version: {0}.{1}'.format(ver, rev))

        # Configure PN532 to communicate with MiFare cards
        pn532.SAM_configuration()

        msg = Nfc()
        self.nfc_found = False
        num_nfc_found = 0

        #self.get_logger().info('Beginning while loop')
        while True:
            # Check if a card is available to read
            uid = pn532.read_passive_target(timeout=0.5)

            self.publisher_.publish(msg)
            self.get_logger().info("Num of NFC found: %i" % num_nfc_found)
            # Try again if no card is available.
            if uid is None:
                self.get_logger().info('NFC not found')
                continue
            self.get_logger().info('NFC found')
            msg.nfc_found = True
            num_nfc_found += 1
            self.publisher_.publish(msg)
            msg.nfc_found = False
            self.publisher_.publish(msg)
            time.sleep(2.0)

            if num_nfc_found > total_nfc:
                break
```

In `nfc_callback()`, the NFC reader is first set up. Once the configuration is done, the NFC reader would then check for the presence of an NFC tag every 0.5 seconds. This process would repeat itself till the number of detected NFC exceeds the total number NFC tags in the maze. Upon breaking out of this function, the program will then destroy the `NFCFinder` node before creating a `ButtonPress` node and calling the function `button_callback()`.

```python
def button_callback(self):
    try:

        msg = Button()
        self.button_pressed = False

        while True:
            if GPIO.input(button_pin) =/= GPIO.LOW:
                self.get_logger().info("Button not pressed")
                continue

            self.get_logger().info('Button pressed')
            msg.button_pressed = True
            self.publisher_.publish(msg)
            break
```

In `button_callback()`, the program will continuously check whether the button has been pressed, which will be indicated by the change in state of the GPIO pin from HIGH to LOW. Once the button has been pressed, the program would then exit the function and destroy the `ButtonPress` node before creating a `ThermalPhase` node and calling the function `start_thermal()`.

```python
def start_thermal(self):
    self.get_logger().info("In start thermal function")
    try:
        #finds and align itself to the thermal object first
        #self.get_logger().info("In start thermal function, moving to callback")
        while self.aligned == False:
            self.thermal_callback()
            rclpy.spin_once(self)

        # start the flywheels up once ready
        #self.get_logger().info("Starting up the flywheels")
        GPIO.output(left_flywheel_pin, 1)
        GPIO.output(right_flywheel_pin, 1)
        # let the flywheel ramp up before starting servo
        time.sleep(0.2)

        # launch the ball with interval of 1 second
        while self.num_balls > 0:
            self.get_logger().info("Entering code to fire balls")
            # Setting the angle to 165
            self.get_logger().info("Firing ball")
            p.set_servo_pulsewidth(servo_pin, 800)
            time.sleep(0.5)
            # Setting the angle back to 15
            p.set_servo_pulsewidth(servo_pin, 1400)
            time.sleep(0.5)
            self.num_balls -= 1
            # updates whether done launching
            self.launcher_callback()

        # Off the flywheel
        GPIO.output(left_flywheel_pin, 0)
        GPIO.output(right_flywheel_pin, 0)
        self.get_logger().info("Done firing all balls")
```

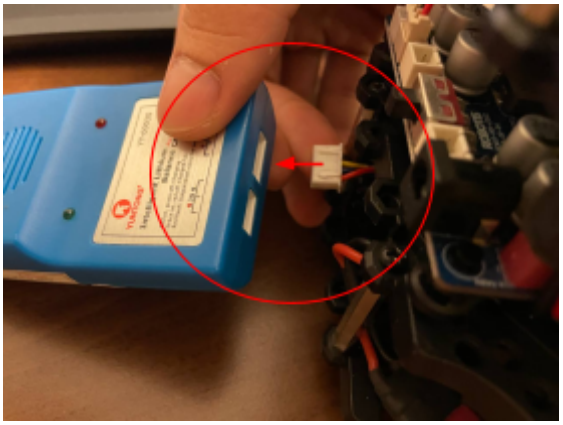
While the bot is not aligned to the object, the program would continuously transmit the data from the thermal camera to the laptop. Once aligned and close enough to the heated object, the flywheel motors would then both be started up by changing the state of the GPIO pins connected to the motors from LOW to HIGH. The servo motor would then move back and forth till all the balls have been fired, after which the flywheel motors are then turned off by setting the pins back to LOW state. The program then exits the function and destroys the node before coming to an end.

All the code used can be found on the Github repository here.

9. System Operation Manual

9.1 Charging Battery

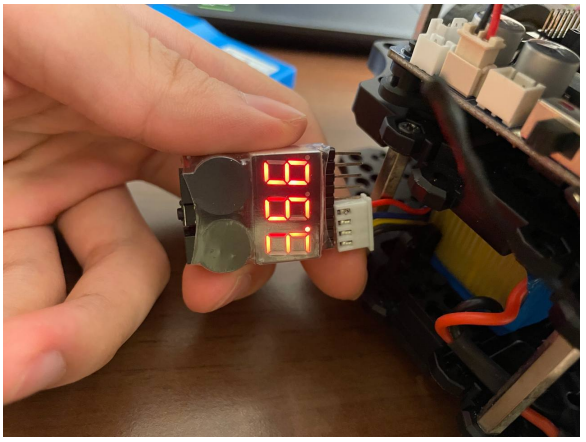
1. Connect the Li-Po battery's charging head into the charger's port as in the picture below.



2. Connect the charger to a wall outlet using the included black cable as in the picture below. If the red LED light turns on, the Li-Po battery is being charged. When the battery is fully charged, the green LED light will turn on.

9.2 Checking Battery Level

The battery voltage level can be checked using the Li-Po Battery Voltage Tester. Connect the battery to the tester starting from the negative pin as shown on the right. The tester will indicate the voltage level of the battery.

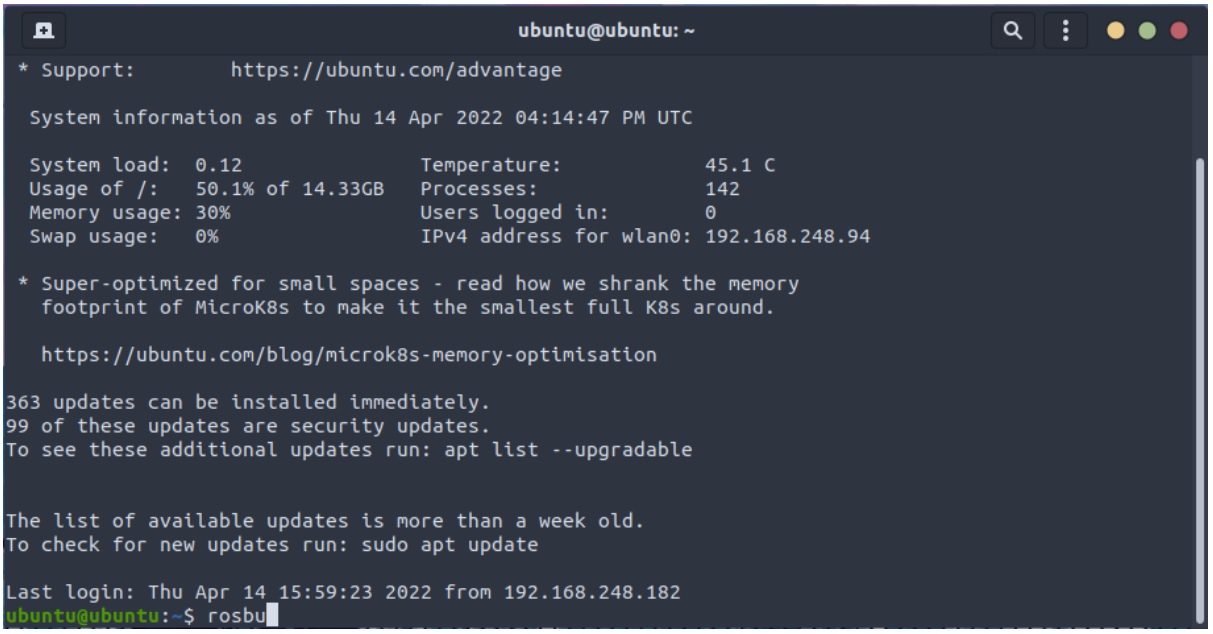


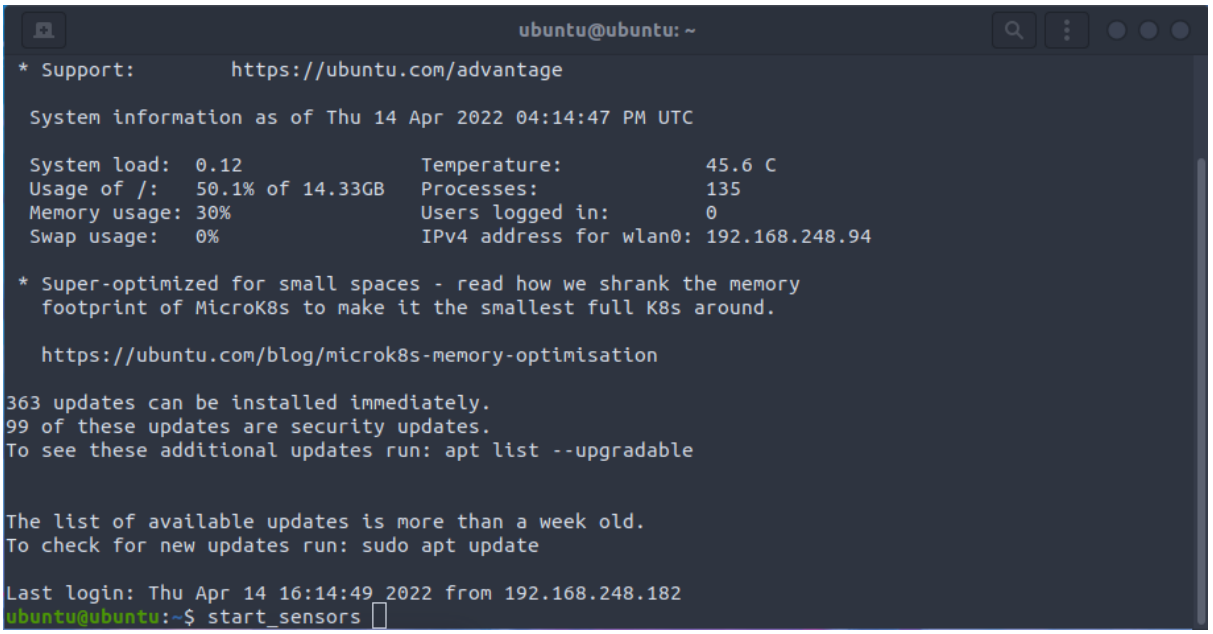
9.3 Software Boot-up Protocol

1. ssh into the RPi on 2 separate terminals

```
ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
```

2. Once logged into the RPi, run rosbu and start_sensors on the separate terminals to start up the Turtlebot3 and the sensors respectively.





3. On another terminal, run rslam to start up Rviz once rosbu has finished setting up. Rosbu indicates that it completes the setup when it returns the statement ‘Run’.

```
ubuntu@ubuntu: ~
[robot_state_publisher-1] [INFO] [1649762954.661153836] [robot_state_publisher]: got segment wheel_r
ight_link
[turtlebot3_ros-3] [INFO] [1649762954.667630216] [turtlebot3_node]: Start Calibration of Gyro
[turtlebot3_ros-3] [INFO] [1649762959.668087165] [turtlebot3_node]: Calibration End
[turtlebot3_ros-3] [INFO] [1649762959.668426737] [turtlebot3_node]: Add Motors
[turtlebot3_ros-3] [INFO] [1649762959.669739453] [turtlebot3_node]: Add Wheels
[turtlebot3_ros-3] [INFO] [1649762959.670429846] [turtlebot3_node]: Add Sensors
[turtlebot3_ros-3] [INFO] [1649762959.679471417] [turtlebot3_node]: Succeeded to create battery stat
e publisher
[turtlebot3_ros-3] [INFO] [1649762959.691607519] [turtlebot3_node]: Succeeded to create imu publishe
r
[turtlebot3_ros-3] [INFO] [1649762959.700305352] [turtlebot3_node]: Succeeded to create sensor state
 publisher
[turtlebot3_ros-3] [INFO] [1649762959.703682113] [turtlebot3_node]: Succeeded to create joint state
publisher
[turtlebot3_ros-3] [INFO] [1649762959.703924084] [turtlebot3_node]: Add Devices
[turtlebot3_ros-3] [INFO] [1649762959.704033404] [turtlebot3_node]: Succeeded to create motor power
server
[turtlebot3_ros-3] [INFO] [1649762959.711159310] [turtlebot3_node]: Succeeded to create reset server
[turtlebot3_ros-3] [INFO] [1649762959.715038138] [turtlebot3_node]: Succeeded to create sound server
[turtlebot3_ros-3] [INFO] [1649762959.718429482] [turtlebot3_node]: Run!
[turtlebot3_ros-3] [INFO] [1649762959.762739814] [diff_drive_controller]: Init Odometry
[turtlebot3_ros-3] [INFO] [1649762959.791546293] [diff_drive_controller]: Run!
```

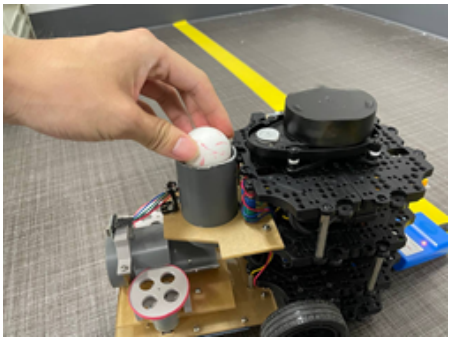
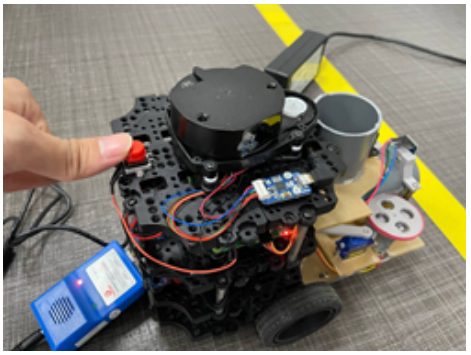
```
jaredoong@jaredoong-ubuntu: ~
(base) jaredoong@jaredoong-ubuntu:~$ rslam
```

4. Once all the above set up is done, run start_auto on another terminal on the laptop for the Turtlebot3 to start moving. Once the Turtlebot3 has completed the entire mission, it will come to a stop automatically.

```
jaredoong@jaredoong-ubuntu: ~
(base) jaredoong@jaredoong-ubuntu:~$ start_auto
```

9.4 Loading the Launcher

The system will automatically stop for loading once the System NFC Reader detects n + 1 NFC cards, where n is the total number of detectable NFCs, `total_nfc` as configured in Section 8.3.6. Once the system stops, load the system as in the instructions below.

	
1. Load **3 ping pong balls** into the Launcher Storage Tube as above, **one ball at a time**.	2. After loading 3 balls, **press** the **red button once**. After **2s**, the bot will move off. If the bot does not move off, press the button again.

10. Troubleshooting

10.1 Software

1. Laptop unable to connect to the turtlebot
    a. Ensure that the laptop and the RPi are connected to the same network.
    b. Ensure that the `ROS_DOMAIN_ID` in the `~/.bashrc` file of the laptop and the RPi is the same.
    c. Change the `ROS_DOMAIN_ID` on both the laptop and the RPi to prevent conflict of communication with those on the same network.

2. RPi unable to connect the WiFi
    a. Ensure that the correct SSID and password is in the netplan of the RPi

    ```
    cd /media/$USER/writable/etc/netplan
    $ sudo nano 50-cloud-init.yaml
    ```

    b. Replace `WIFI_SSID` and `WIFI_PASSWORD` with your wifi SSID and password

3. ImportError, no module named “***”.
    a. Simply install the module using the following command:
    `pip install <module_name>`

4. colcon build returns error/warning
    a. Ensure that you are in the correct directory `~/colcon_ws` or `~/turtlebot3_ws`
    b. Run colcon build again

5. No change observed in program even after changes in code
    a. Ensure that colcon build was run in the correct directory
    b. Ensure `source install/setup.bash` was run
    c. Ensure that the correct program is being run
    d. Rebuild the package, source it, and then run the program again

6. Unable to run ‘start_sensors’
    a. Ensure that `sudo pigpiod` has been run
    b. Rebuild the package, source and run the program again

7. ‘start_sensors’ skips the NFC / Thermal camera portion
    a. Run `i2cdetect -y 1` and ensure that both devices are connected
    b. Rebuild the package, source it, and run the program again

8. Unable to run the program on either the laptop or RPi
    a. Rebuild the package, source it, and run the program again

10.2 Hardware

1. NFC reader not returning data
    a. Ensure that the red power LED is lit up
    b. Ensure that I2C mode is enabled
    c. Run `i2cdetect -y 1` and ensure that the NFC reader is connected
    d. Check the wiring and ensure that all the wires are connected and not loose

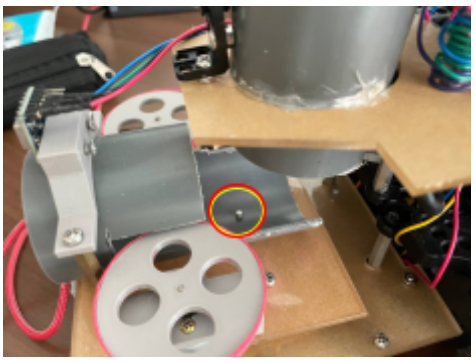
2. Thermal Camera not returning data
    a. Run `i2cdetect -y 1` and ensure that the thermal camera is connected
    b. Check the wiring and ensure that all the wires are connected and not loose

3. Flywheels not spinning
    a. Ensure that the spades at the bottom of the motors have not come loose
    b. Ensure that the wires from the RPi to the motors are not broken

4. Servo not moving
    a. Ensure that the wires have not come loose
    b. Swap out with a different servo and run the program again

5. Balls not loading from storage tube into barrel correctly
    a. Check that storage tube is aligned with barrel
        i. Check that storage tube mounting bracket is installed and secure
        ii. Physically align storage tube with barrel
    b. Check that pusher arm is able to rotate fully to receive and load balls

6. Balls are firing before the pusher arm pushes them into the flywheel
    a. Check that storage tube is correctly aligned with barrel
    b. Ensure Stopper Screw is installed into the launcher barrel, as circled below



7. Launcher balls not shooting out correctly
    a. Check launcher barrel
        i. Is aligned with Turtlebot3. If not aligned, check that Stopper Screw is installed and physically align the launcher barrel
        ii. Does not have obstructions
    b. Check that both motors are working
        i. Both motors are spinning and in the correct direction
        ii. Balls contact both motors when pushed into the flywheel

8. Turtlebot3 not moving smoothly
    a. Check rotation of front ball caster is smooth
        i. If not smooth, either oil ball caster or replace it
    b. Check rotation of rear ball caster is smooth
        i. If not smooth, either oil ball caster or replace it
    c. Check that DYNAMIXEL motors are spinning consistently via `rteleop`

11. Future Work

Although the Turtlebot3 was able to complete the mission, there were flaws in the design that complicated the process of creating a working system. Additionally, certain design choices were made due to the project scope and timeframe that was not optimal overall. This section breaks down the flaws in terms of mechanical, electrical and software, and the possible ways to improve the system in the future iterations.

11.1 Mechanical

11.1.1 Manufacturing process

The 3D printed and acrylic parts represent a significant portion of the total system cost (~25%), which can be reduced by manufacturing them using injection moulding. Manufacturing them through injection moulding, would require significant scale to be more cost effective than 3D printing due to cost of equipment, hence would only be suitable for future works when the requisite scale of production is achieved.

The table below provides proposed changes to manufacturing methods of the system:

Part(s)	**Proposed Manufacturing Method**	**Benefits**
● Top Acrylic Plate ● Launcher Storage Tube ● Storage Tube Bracket	● Plastic Injection Mould as 1 part	● Reduces cost and part count ● Eliminates need for fasteners ● Single part reduces misalignment issues between Storage Tube and Barrel
● Launcher Barrel ● Middle Acrylic Plate ● Bottom Acrylic Plate ● Front Support	● Plastic Injection Mould as 1 part	● Reduces cost and part count ● Reduces weight of component and hence improves system turning speed ● Elimination of fasteners reduces potential misalignment of barrel
● Flywheels ● Anti-Slip Racket Tape (on flywheel)	● Plastic Injection Mould for Flywheel ● Styrene Butadiene rubber used instead of Racket Tape, bonded to flywheel via cyanoacrylate	● Reduces cost of flywheels and rubber ● Improves durability of rubberised flywheel ● Reduces discolouration of Ping Pong Balls due to flywheel abrasion

11.1.2 Integration of Launcher Assembly

One major mechanical change that may be considered for future iterations of the system would be to integrate the launcher assembly into the Turtlebot3 itself by installing it into one of the layers. This was not done for this project as the external launcher system was deemed to be suitable for the scope of the project, due to being simpler to design, more reliable, and easier to troubleshoot and iterate on during the prototyping/testing processes.

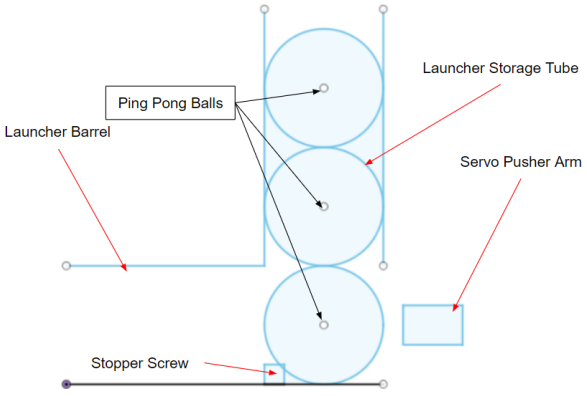
If given additional time for future works, integrating the launcher assembly into the Turtlebot3 could have the following benefits:

1. Reduce the size and turning radius of the system
2. Reduce the number of parts needed (eg. Front Ball Caster, Acrylic Plates) and hence cost of system
3. Improve turning speed by centralising the parts nearer to the Centre of Gravity of the System and hence reducing the Mass Moment of Inertia of the system

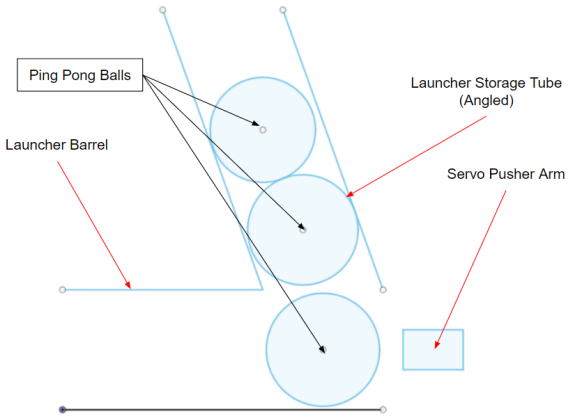
11.1.3 Alteration of Storage Tube

Due to the vertical alignment of the storage tube, a stopper screw was required to prevent the balls from prematurely loading into the barrel. This increases the parts required, and also has occasional reliability issues where two balls are shot out at once when the servo moves. Changing the design to angle the storage tube away from the launcher would eliminate the need for the stopper screw and improve the reliability of the firing (see pictures below).

Original Vertical Tube



Proposed Angled Tube



11.2 Electrical

11.2.1 Connection between components and RPi

Due to a lack of time and ease of accessibility, a protoboard was used in the final design. Although it served its purpose in the short run, the wires on the protoboard may come apart after some time due to wear and tear. For a more reliable connection in the long run, the

protoboard should be replaced by a PCB. Using a PCB will also resolve the problem of a messy circuit connection as seen from the protoboard.

11.2.2 Wire management

In the final design of the Turtlebot3, due to wires being made to the wrong length, we faced the issue of the wires being either too long or too short. The wires that were too short resulted in taut wires, and these wires sometimes broke while the Turtlebot3 was in operation. For the wires that were too long, to avoid the wires colliding or hooking onto the walls, the wires were wrapped around the plate supporters. The excess wires result in extra mess which can be troublesome to deal with whenever there is a need to change the wiring between the components and the RPi.

To prevent such problems, the length of the wires required for each connection should be precisely measured in the future. Better wire management would also allow for much simpler reconnection of wires in the event that a wire comes loose.

11.3 Software

11.3.1 Navigation algorithm

Although the left wall following algorithm worked for the mission, it is important to note that the wall following algorithm is one of the simplest navigation algorithms, which results in it being less robust. Additionally, the current algorithm is heavily dependent on all the walls of the maze being connected. In the event in which an isolated obstacle is present, or if the dimensions of the maze exceed the capabilities of the LDS, there could be areas which are impossible to map using the wall following algorithm. For mapping out the maze, the wall following algorithm is also not the most effective as it is unable to determine if a room has been fully mapped, and blindly follows the walls of the maze. This results in redundancy as the Turtlebot3 often travels a full round around the room even though it could map the entire room once it has entered.

For future iterations of the Turtlebot3, a more efficient algorithm could be used for the mapping portion of the mission. Search algorithms that are commonly used in the field of robotics include the A* search algorithm and frontier algorithm. These algorithms prioritise exploration of a new unexplored area, and would hence allow the Turtlebot3 to map out the entire maze much more efficiently and in a shorter time. These algorithms would overcome the problem of the middle portion of the maze being unmapped in the event that the parameters of the maze are much larger than the maximum range of the LDS.

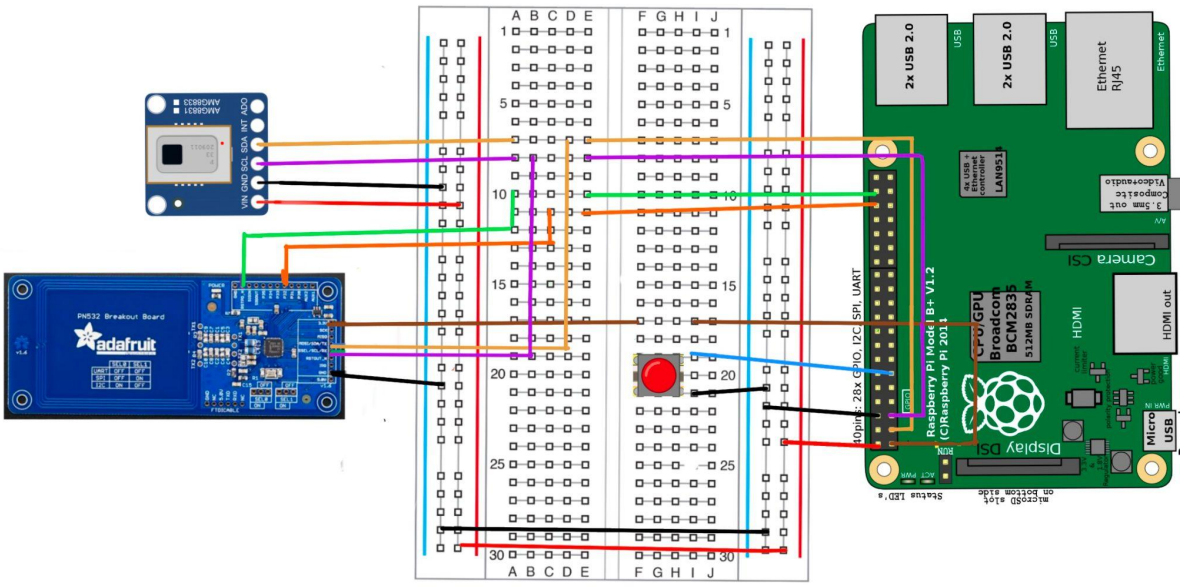
11.3.2 Communication Protocol

Both the Adafruit PN532 NFC Breakout board and the AMG8833 thermal camera are connected to the RPi using I2C. I2C allows for the neatest wiring and pin management since only two bidirectional wires are used to transmit data between the sensor and the RPi. However, I2C is not recommended on the Adafruit PN532 NFC Breakout board tutorial and should be avoided if possible.

SPI could be used instead for both the sensors. Although the number of wires needed for SPI is twice the number of wires needed for I2C, given that only two sensors are connected to the Turtlebot3, the slight increase in the number of wires would not be much of a concern. Additionally, the transmission speed of data over SPI is faster compared to I2C, and it is a good fit for the system since SPI is mainly for systems with one master and multiple slaves.

Annex

Annex A: Electrical diagram of preliminary design



Annex B: Preliminary Software block diagram

Flowchart of general algorithm in prelim design

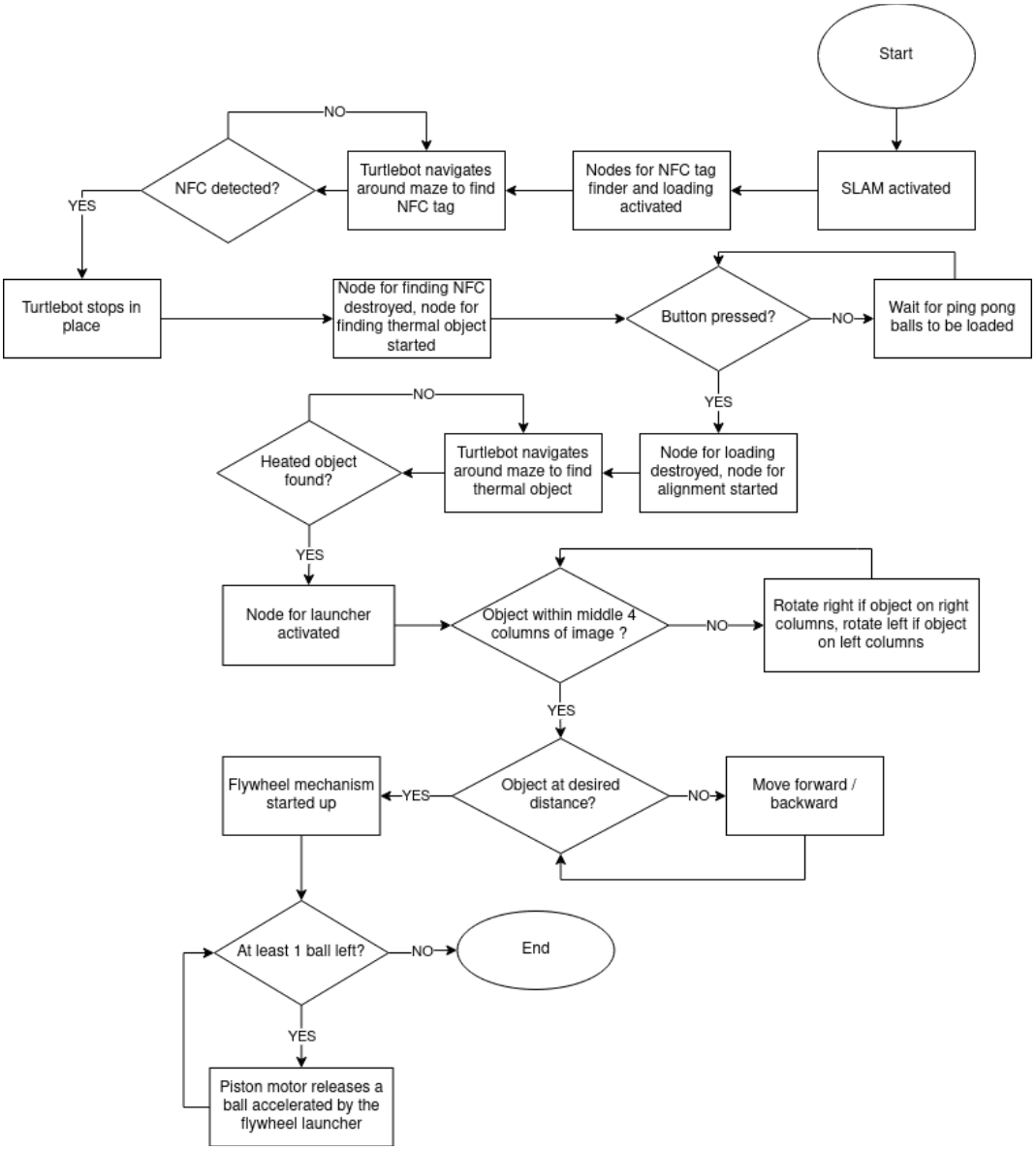
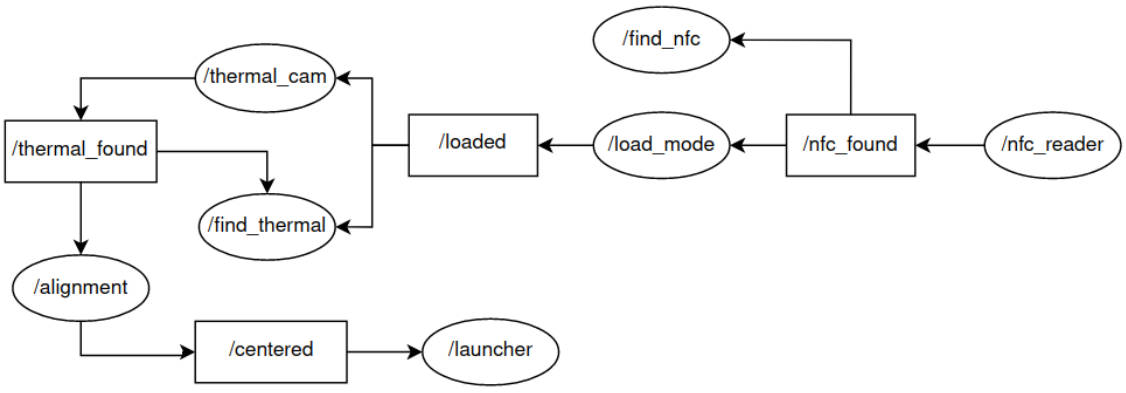
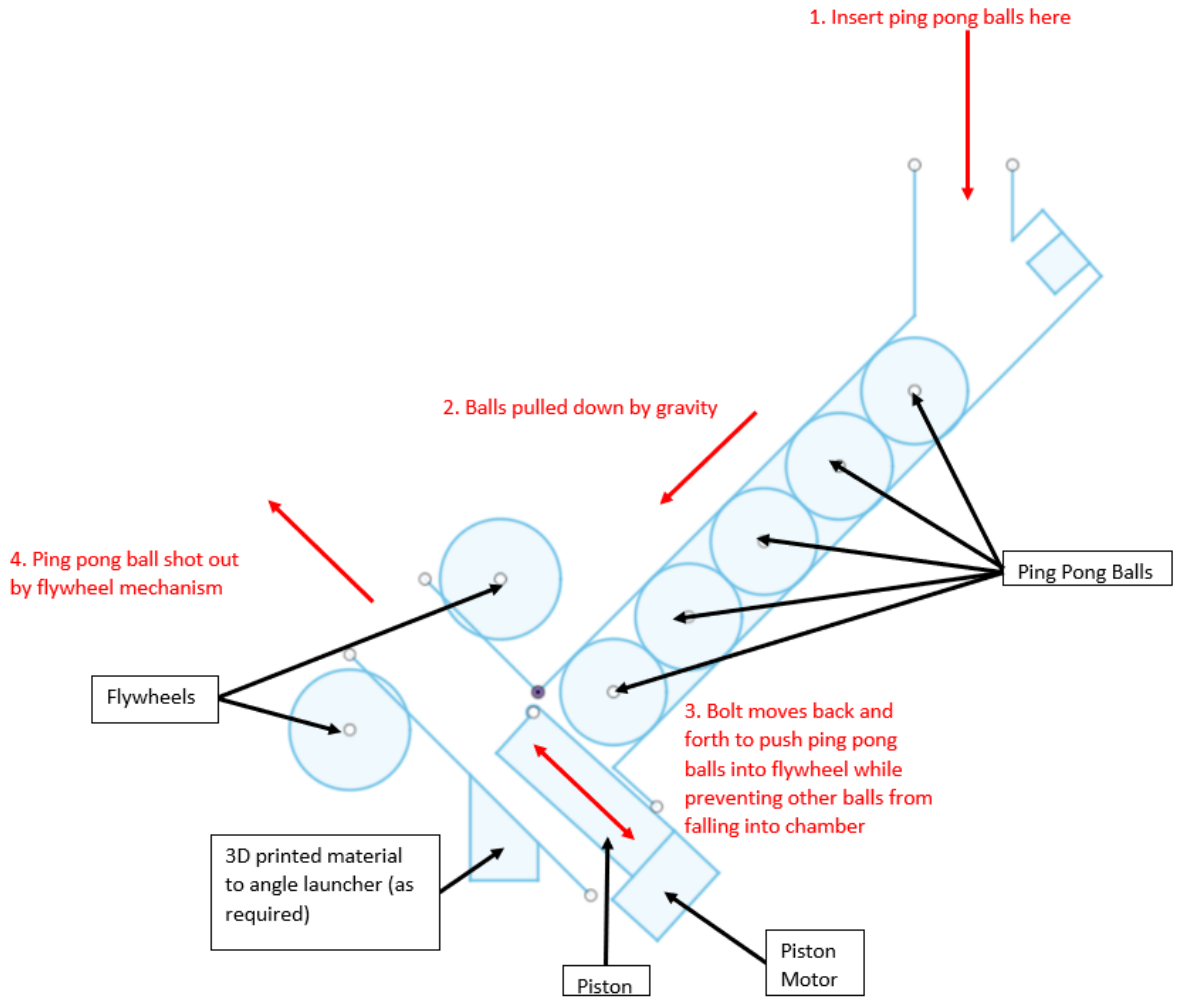


Diagram showing nodes and topics used by the Turtlebot3 in prelim design



Annex C: Preliminary launcher design

Diagram of launcher mechanism in prelim design



Annex D: Preliminary monetary budget

Component	Cost
Adafruit PN532 NFC RFID Module	-
AMG8833 IR Thermal Camera	$44.95
Lego EV3 Technic Linear Actuator	$5
DC Motor x 3 (Flywheel + Linear Actuator)	$6.20 x 3 = $18.60
Raw Materials to Make Launcher (eg. PVC)	$10
AA Battery Holder	$1
Total	$79.55

Annex E: Preliminary algorithms

Initial algorithm used to find the NFC

Upon starting up, the Turtlebot3 will move forward, and its distance travelled in the initial direction would be saved in a variable (distance_travelled). This distance can be obtained by measuring the change in the x position of the Turtlebot3 data based on the data return by the Odometry node over the topic ‘odom’. When it hits a wall, it will first check if the distance travelled in the initial direction is at least 4.5 m. If the distance travelled is at least 4.5 m, it would mean that the Turtlebot3 has reached the wall. For its first turn at the edge of the maze, the Turtlebot3 would check whether the LIDAR reading from its left or right side returns a larger distance. The Turtlebot3 would make a 90-degree turn in the direction of the larger distance, and it would keep track of this turn by appending the direction turned to an array (turns). After the initial turn, it will move forward a short distance before coming to a stop and making another 90-degree turn in the same direction as the first turn. The short distance travelled would be determined through trial and error to find a value that would allow the Turtlebot3 to cover more ground efficiently, by not going over areas that have already been checked. Once the Turtlebot3 completes its U-turn, it will continue moving in a straight line till it meets the wall again. Upon stopping at the wall, the Turtlebot3 would then check the direction of its previous turn from the array (turns). If the previous turn was a U-turn towards the left, it would then proceed to make a U-turn towards the right this time, vice versa. This thus allows the Turtlebot3 to move towards the unexplored region. In the event that the Turtlebot3 reaches a wall before the distance travelled is at least 4.2 m, it would perform a different manoeuvre to go around the obstacle and continue on its path. The Turtlebot3 would turn to its left and perform a different algorithm that is adopted from the wall following algorithm, so that it can go around the perimeter of the obstacle and reach the other side. When the algorithm is started, the Turtlebot3 would keep track of the distance travelled along the y axis by reading the data received from the Odometry node over the topic ‘odom’. For this algorithm, the Turtlebot3 would keep the obstacle constantly on its right. This is done by using the LIDAR data and ensuring the distance read in from its right is always less than the threshold value. Whenever the distance from its right surpasses the threshold value, the Turtlebot3 would turn right 90-degree to keep the obstacle on its right and continue moving forward. In the event that both the front and right sides are blocked while attempting to bypass the obstacle, the Turtlebot3 would turn left 90-degrees. The Turtlebot3 will then continuously repeat the above steps and break out of the edited wall follower algorithm when the values in the array (bypass) are equal. Once the values are equal, it would mean that the Turtlebot3 has reached the other side of the obstacle and can continue on its initial path. This ensures that the Turtlebot3 will not continuously loop around the obstacle and be stuck. The Turtlebot3 then turns to face the direction of its travel before it was stopped by the obstacle and continues on its path. This would hence increase the coverage of the Turtlebot3 and reduce the time wasted on travelling areas that it has already gone over.

Initial algorithm for finding the thermal object

Once the Turtlebot3 has detected that all the balls are loaded, it will first check whether it is at the edge of the maze. This is done by checking the horizontal distance travelled so far, similar to the NFC finding algorithm. If the horizontal distance is not at least 4.5 m, the Turtlebot3 will then continue to move in the forward direction till it reaches the edge of the

maze. Once the Turtlebot3 reaches the edge of the maze, the Turtlebot3 then turns left. The Turtlebot3 then starts to move forward along the edge of the maze and after a predetermined distance, the Turtlebot3 would rotate 90-degree left to check whether the thermal object is within the thermal camera's line of sight. The predetermined distance will be the distance such that the area within the thermal camera's line of sight from its new position would be different from its old position. This distance will be determined through trial and error. In the event that the Turtlebot3 is blocked by an obstacle while travelling along the edge of the wall, it would then adopt the same algorithm utilized by the NFC finding algorithm to bypass the obstacle. Using this method, the Turtlebot3 would be able to visually scan every corner of the maze systematically and would hence be able to detect the thermal object.

Annex F: Power Budgeting Table

Running off LiPo battery [11.1 V 1800mAh / 19.98 Wh]				
Component	**Voltage**	**Current**	**QTY**	**Power Consumption**
Turtlebot3 (During Operation)	11.1	720mA	1	7.992W
Adafruit PN532 NFC RFID Module	3.3V	150mA	1	0.495W
AMG8833 IR Thermal Camera	3.3V	4.5mA	1	0.015W
SG90 Servomotor	5V	250mA	1	1.25W
DC motor Flywheel	6V	180mA	2	2.16W
Total Power Consumption from LiPo battery				19.98Wh
Firing process takes approximately 10 seconds*				0.082 Wh
Total running time				2.35h

*In the final run, the firing process works only for less than 10 seconds. Thus, we consider the power it uses in the actual running insignificant when we calculate the total running time.

Although the theoretical running time should be 2.35h, in reality the operating time of the Turtlebot3 will be shorter. In reality, energy loss exists in the form of heat in the energy conversion process. Additionally, when the voltage of the LiPo battery falls below 11V, it would start beeping, and the power supplied is no longer sufficient for the motors to run, which prevents the Turtlebot3 from moving.

While testing the Turtlebot3, it often lasts between 1.5 - 1.75 hours, depending on how often it is moving around.

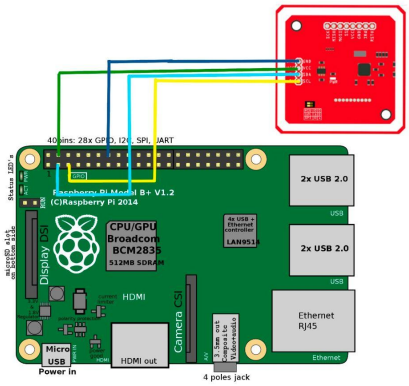
Annex G: Launcher System Calculations

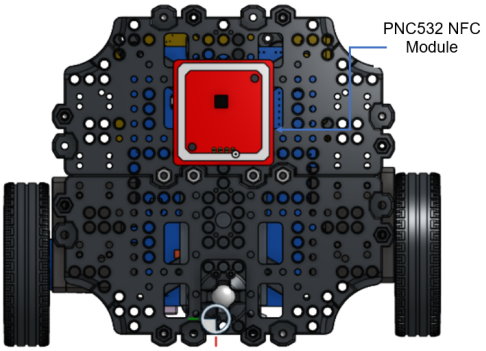
1. Estimated distance ball needs to be fired: 20cm, see Pg 4 Literature Review under "Firing of Ping Pong Balls" for rationale of distance chosen
2. Flywheel Motor calculations
    a. Assume the DC motor used is 10000rpm.
    b. $10000rpm * 2\pi/60 = 1047rad/s$
    c. Assuming a wheel radius of 25mm
    d. $v = r\omega = 1047rad/s * 25mm = 26175mm/s = 26.175m/s$
    e. Velocity calculated above is the maximum possible velocity of the ping pong ball, assuming ideal conditions where flywheels do not slow down
3. Kinematic calculations of ping pong ball
    a. Neglecting the effects of air resistance due to short distance of travel (20cm)
    b. Use velocity of v = 26.175m/s as calculated above
    c. Initial vertical component of velocity is $v_y = vsin\theta$ where v=26.175m/s and θ is the angle of launch
    d. Change of vertical component of velocity per unit time is 9.81m/s in the opposite direction of initial component due to gravitational acceleration
    e. Distance travelled in horizontal component is $s_x = v_x t$ where $v_x = vcos\theta$ and t is time elapsed from firing
    f. Distance travelled in vertical component can hence be calculated from $s_y = v_y t + 0.5(-9.81)t^2 = vsin\theta(s_x/vcos\theta) + 0.5(-9.81)(s_x/vcos\theta)^2$ $= s_x tan\theta + 0.5(-9.81)(s_x/vcos\theta)^2$ by combining the equations above. Sub in the value of v to plot a table of vertical distance travelled compared to the angle of launch and horizontal distance travelled

Annex H: Additional preliminary ideas

Another choice for NFC detection

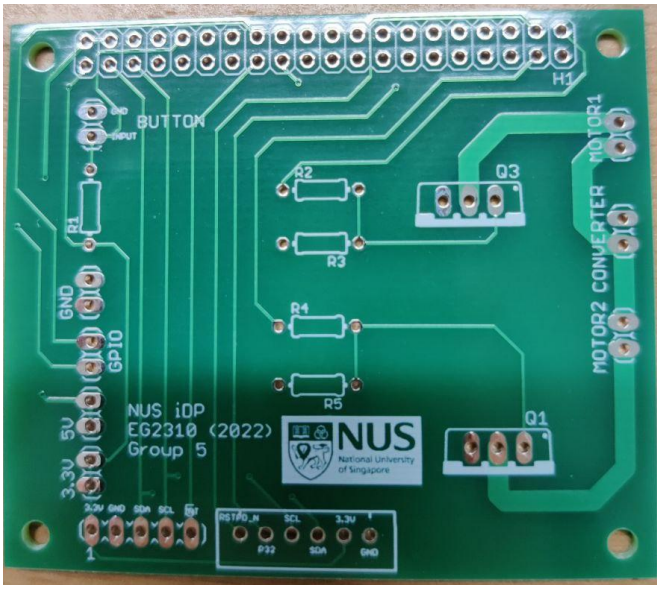
The NFC reader to be used in the design is the PNC532, an NFC module that is able to communicate with the RPI using I2C. The 4-pin interface to be connected consists of the power supply (Vcc), the ground line (GND), the clock line (SCL) and the data line (SDA). This module requires 5V to run and would Vcc hence be connected to the RPI on pin 2. The SCL and SDA on the module would be connected to the GPIO 3 [pin 5] and GPIO 2 [pin 3] respectively on the RPI, which are responsible for such functionality respectively. The ground would be connected to pin 6, which acts as the ground on the RPI. Since the NFC tag would be placed on the floor, the NFC module would be placed on the underside of the Turtlebot3 in order to minimise the distance between the NFC reader and the tag. The module can then be activated, which would cause it to sense for data constantly. Once it detects an NFC tag, data regarding the NFC tag would be returned to the RPI. The small size of the module allows for easy incorporation into the model of the Turtlebot3.



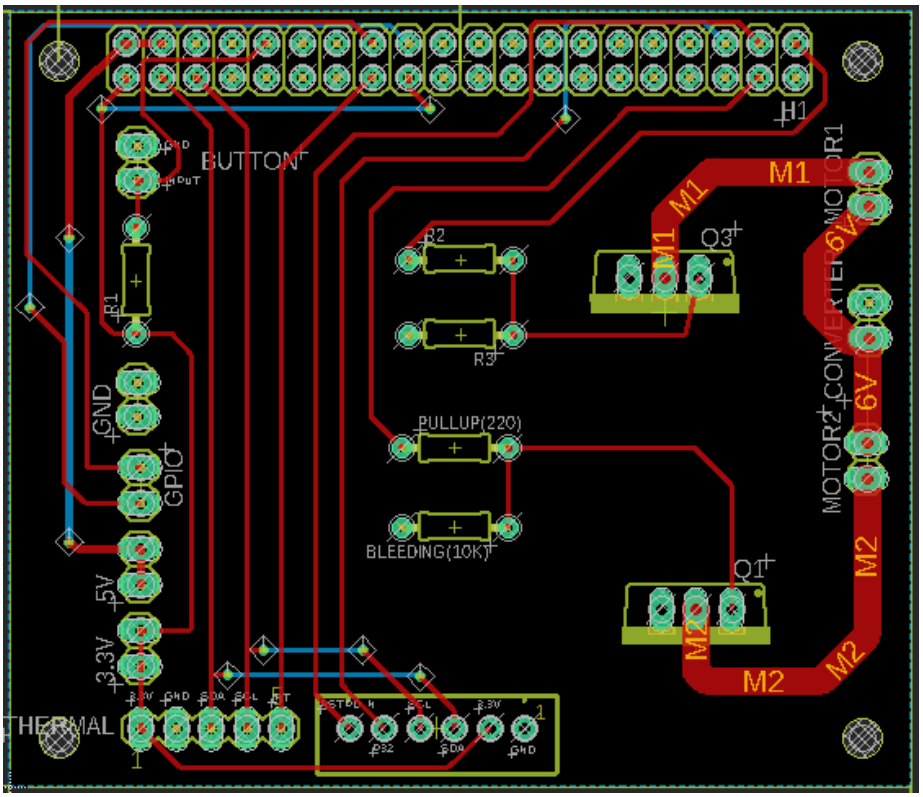


Bottom View of TurtleBot3 Burger

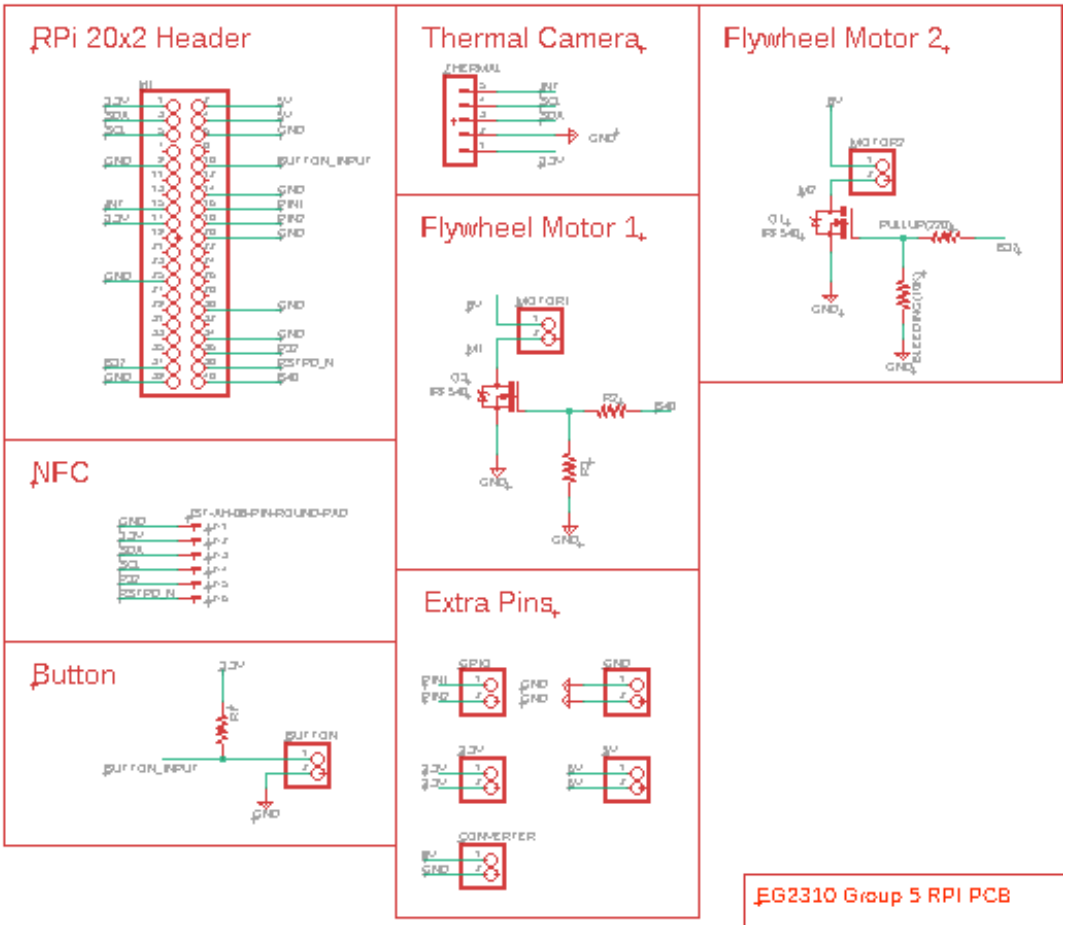
Annex I: PCB Design



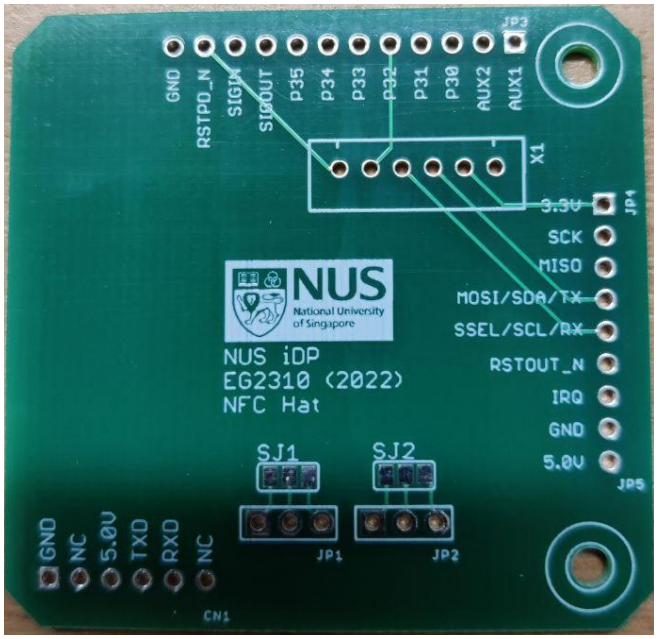
PCB for RPi



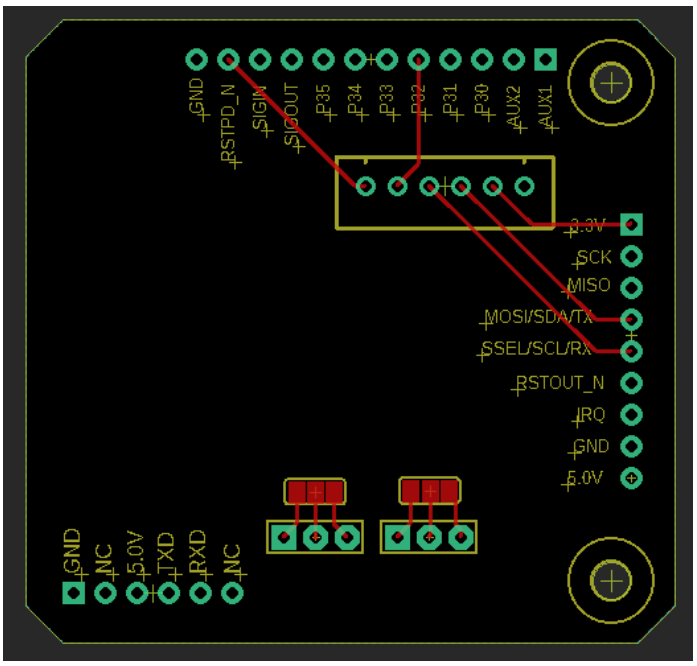
EAGLE Board for RPi PCB



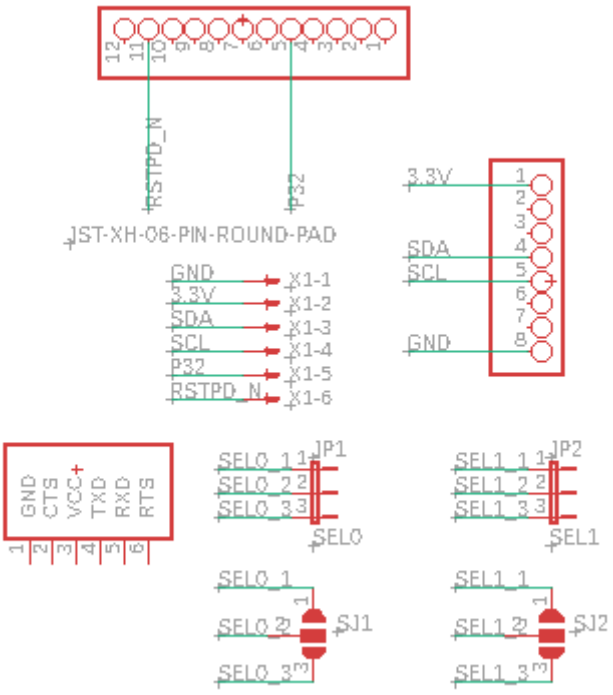
EAGLE Schematic for RPi PCB



PCB for NFC Breakout Board



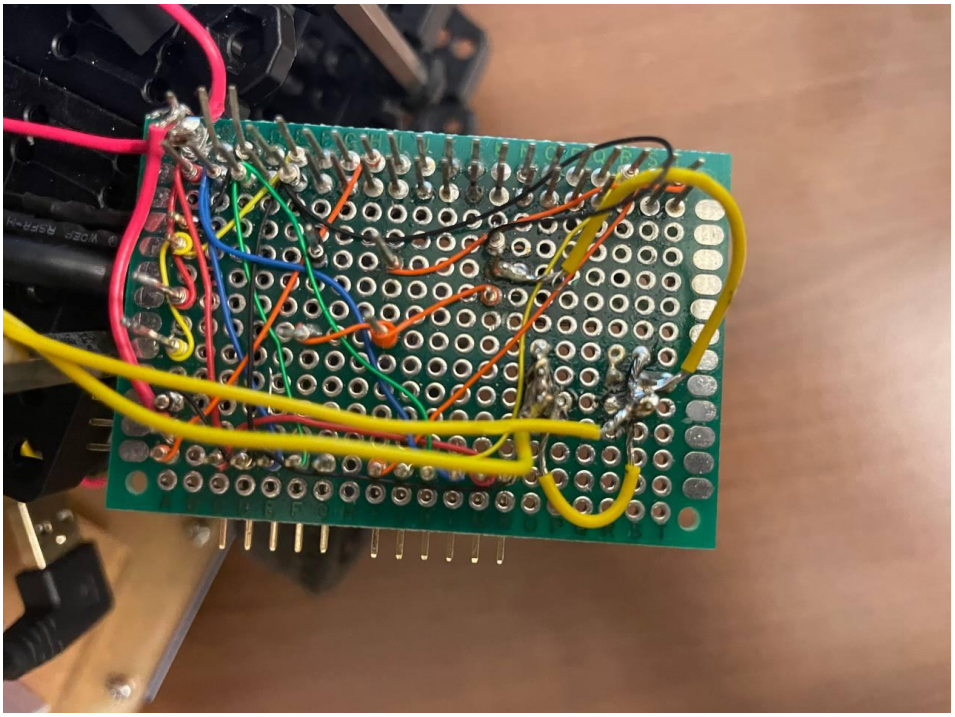
EAGLE Board for NFC PCB



EAGLE Schematic for NFC PCB

Annex J: Protoboard Design

Back side of protoboard



Front side of protoboard

Annex K: Product Industry Standard Compliances

Part	**Compliance Standard**
MM28 Flywheel Motor	ROHS compliant
Raspberry Pi	EU Council Directive 2004/108/EC
AMG8833 Thermal Camera	ROHS compliant EU Reach SVHC
OpenCR1.0	ROHS Compliant
360° LDS-01 LIDAR	FCC Class B Device

Citations

[1] Robotis e-Manual. *LDS for TurtleBot3*.
https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/
[2] Andrew, Y. (2021). *Maze Escape with Wall-Following Algorithm.* Medium.
https://andrewyong7338.medium.com/maze-escape-with-wall-following-algorithm-
[3] Mohamed, F. (2021). *ROS Autonomous SLAM using Rapidly Exploring Random Tree (RRT).* towards data science.
https://towardsdatascience.com/ros-autonomous-slam-using-randomly-exploring-random-tree-rrt-
[4] Shopee. *PN532 NFC RFID Wireless Module V3 User Kits Reader Writer Mode IC S50 Card PCB Attenna I2C IIC SPI HSU.*
https://shopee.sg/PN532-NFC-RFID-Wireless-Module-V3-User-Kits-Reader-Writer-Mode-IC-S50-Card-PCB-Attenna-I2C-IIC-SPI-HSU-
[5] Cherie, T. *NFC Module with Raspberry Pi.* Little Bird Electronics.
https://littlebirdelectronics.com.au/guides/181/nfc-module-with-raspberry-pi
[6] Soren. (2018). *Using a push button with Raspberry Pi GPIO*. Raspberry Pi HQ.
https://raspberrypihq.com/use-a-push-button-with-raspberry-pi-gpio/
[7] Shopee. *EV3 Techninc Parts Linear Actuator Pusher Connectors Compatible logoe 61904 61927 MOC Educational Building Blocks Set.*
https://shopee.sg/EV3-Techninc-Parts-Linear-Actuator-Pusher-Connectors-Compatible-logoe-61904-61927-MOC-Educational-Building-Blocks-Set-
[8] *Ping-Pong Ball | Table Tennis Ball.* Dimensions.com.
https://www.dimensions.com/element/ping-pong-ball-table-tennis-ball
[9] Khong Guan Malaysia. *Carton Size of Assorted Biscuit (Castle Brand).*
https://www.khongguan.com.my/products/gift-packs-tins/
[10] Sau, S. (2020). *Build a thermal camera with Raspberry Pi and Go.* gitconnected.
https://levelup.gitconnected.com/build-a-thermal-camera-with-raspberry-pi-and-go-8f70451ad6a0
[11] Shopee. *SG90 Servo Micro Servo Motor Mini 9G Motor arduino 180 deg RC helicptr airplane iot project 9g servo sg90.*
https://shopee.sg/(Local-Stock)-SG90-Servo-Micro-Servo-Motor-Mini-9G-Motor-arduino-180-deg-RC-helicptr-airplane-iot-project-9g-servo-sg90-
[12] Yuri Ostr. (2017). *How to Make Ping Pong Ball Launcher at Home - Full Auto Electric Machine Gun* [Video]. Youtube. https://www.youtube.com/watch?v=87dd0elBw_E
[13] Payne (2019). *Arduino Ping Pong Ball Cannon*. Hackster.io.
https://www.hackster.io/GordPayne/arduino-ping-pong-ball-cannon-3c1fdd
[14] Shopee. *Micro 130 DC 3V-6V 10000 RPM Cars Toys Electric Motor, High Speed Torque DIY Remote Control Toy Car Hobby.*
https://shopee.sg/(Ready-Stock)-Micro-130-DC-3V-6V-10000-RPM-Cars-Toys-Electric-Motor-High-Speed-Torque-DIY-Remote-Control-Toy-Car-Hobby-i.223700280.11955886941?sp_atk=810ed6f1-f819-457b-b905-6775bf3ccf82
[15] (2016). *Make a Ping Pong Ball Launcher!* Frugal Fun For Boys and Girls!
https://frugalfun4boys.com/make-ping-pong-ball-launcher/
[16] Dialed In DIY. (2018). *Air Cannon Shoots Ping Pong Balls - Complete Tutorial* [Video]. Youtube.
https://www.youtube.com/watch?v=-eDP2fVFgn8
[17] *Physics of the Flywheel Launcher. VEX Forum.*
https://www.vexforum.com/t/physics-of-the-flywheel-launcher/29357
[18] Ma, Z. (2019, December 8). *Robot vacuum cleaner part 3/3*. Medium.
https://medium.com/cse-468-568-robotic-algorithms/robot-vacuum-cleaner-part-3-3-2bc317cf17db
[19] Wikimedia Foundation. (2022, January 10). *Maze-solving algorithm*. Wikipedia.
https://en.wikipedia.org/wiki/Maze-solving_algorithm#Pledge_algorithm
[20] *Adafruit AMG8833 IR thermal camera breakout*. Adafruit AMG8833 IR Thermal Camera Breakout - Adafruit Industries LLC - Evaluation Boards - Sensors | Online Catalog | DigiKey Electronics. (n.d.).
https://www.digikey.com/catalog/en/partgroup/adafruit-amg8833-ir-thermal-camera-breakout/77282
[21] Open Source Robotics Foundation, Inc. *TurtleBot3 Burger Model.*
https://www.Turtlebot3.com/opensource/
[22] Turtlebot3 Manual. Manual. Retrieved from https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

Archive

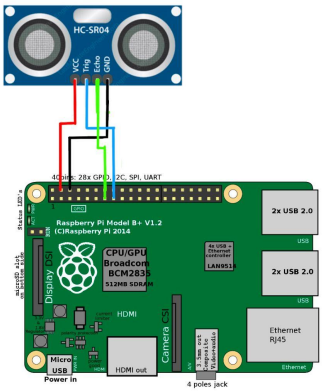
Using RRT for auto-navigation to find the NFC tag [Changed the algorithm used to find the NFC Tag]

For the autonomous navigation of this mission, the RRT algorithm can be implemented using the package from rrt_exploration. Before deploying the Turtlebot3, five points need to be clicked in the occupancy grid map, in which the first four points define the square region to be explored, and the last point indicates the tree starting point. The frontier detector node then generates target exploration goals by detecting frontier points, the boundaries separating known space from unknown space. The results are then filtered by the filter node to remove old or invalid points that are obstacles as means for obstacles avoidance. The remaining points are published by the assigner node, which commands the robot to move accordingly with the help of the navigation stack. A map of the explored environment will be constructed and constantly updated using gmapping while the Turtlebot3 explores the map. However, this idea was scrapped as while it is good for exploration purposes, it is not the best fit for the purpose of finding a specific grid within an enclosed area. Instead, an algorithm which ensures that the Turtlebot3 travels every inch of the area would be more suitable.

Ultrasonic sensors connected for loading [Switch to button instead]

An HC-SR04 ultrasonic sensor can be utilised to detect the number of ping pong balls loaded into the loading mechanism, as well as the distance between the heated object and the Turtlebot3. The ultrasonic sensor has four pins, namely Vcc, Trig, Echo, and GND. For each ultrasonic sensor, its Vcc would be connected to pin 2 to supply the 5V to power up the sensor, and the GND pin would be connected to pin 6 of the RPI. The other two pins should be connected to different GPIO pins in RPI to get the input values.

For the loading mechanism sensor, the Trig and Echo pin is connected to pin 16 and 18 of the RPI respectively. The HC-SR04 has a short minimum range of 2 cm, which is important as it helps to reduce the size of the loading mechanism, as the distance between the sensor and 5th ping pong ball when loaded only needs to be 2 cm. The accuracy of the HC-SR04 is 0.3 cm, which is well under the diameter of a ping pong ball at about 3.8 cm, which should allow the Turtlebot3 to distinguish between the number of balls loaded based on the distance sensor readings.



Launcher loading mechanism [Removed sensor to detect loaded balls in favour of simpler solution]

To determine the number of ping pong balls in the system, one can use a weight sensor [6] to measure the added weight of the ping pong balls as they are loaded. As ping pong balls are light, however, the weight sensor would have to be sufficiently precise to measure the weight of each ping pong ball.

[6] Sparkfun. Load Sensor - 50kg (Generic) https://www.sparkfun.com/products/10245

Alternatively, a distance sensor can be used to measure the distance between a fixed point and the most recently loaded ping pong ball. The sensor can either be an Infrared (IR) or Ultrasonic distance sensor. Ultrasonic sensors are generally more reliable and less affected by environmental factors, they will be preferred over IR sensors even with the added cost [7].

[7] Roderick, B. (2017). Ultrasonic vs Infrared (IR) Sensors – Which is better? https://www.maxbotix.com/articles/ultrasonic-or-infrared-sensors.htm

An ultrasonic distance sensor should be chosen instead of an IR one as it is more reliable and less affected by external conditions, and the HR-SR04 chosen is sufficiently affordable that the increased cost is not a big issue. This ultrasonic distance sensor will detect the distance to the next ball and publish the data for the Turtlebot3 script to determine the number of balls left.

Aiming of ping pong balls [Removed servo to adjust ball height in favour of simpler solution]

The motor must be able to both adjust its angle at regular intervals, and tell which angle it is currently at. As such, a servo motor (e.g. SG90 [11]) can be used to adjust the angle of the launcher. The servo motor should have sufficient torque to move the required launcher components, as sufficient rotation angle (90 degrees is sufficient). Alternatively, a combination of stepper/rotary DC motor and rack and pinion mechanism can help to adjust the vertical angle of the launcher mechanism. This option will likely increase the cost, weight and complexity of the mechanism, but may be required if the servo motor has insufficient torque to support and rotate the launcher mechanism. Mounting on the launcher mechanism itself is ill advised as the launcher mechanism will be adjusted elevation wise by the servo motor which would misalign the sensors. As such, mounting on either side is the most ideal, and the aiming script would need to be adjusted accordingly to account for the offset from the sensors

Using of ultrasonic sensor when loading the Turtlebot3 with ping pong balls [Changed to button system]

For this loading phase, an ultrasonic sensor mounted on the roof of the loading tube is used to determine when all the balls are loaded. The loading mechanism should be configured in such a way that loaded balls stack on top of each other, and the ultrasonic sensor is mounted in line with the stacked ping pong balls (See Annex C). This allows the ultrasonic sensor's distance readings to determine the number of balls loaded, as each ball will reduce the distance read by the sensor by a fixed amount to be determined through testing. The node in charge of the loading (/load_mode) would be publishing whether all balls have loaded over a topic (/loaded) and the nodes for the next phase, finding the thermal object, (/find_thermal, /thermal_cam) would be subscribed to this topic to know when to start searching for the hot object. When the Turtlebot3 detects that all the ping pong balls are loaded, the Turtlebot3 would then wait for a period of time before moving off, to ensure

that it would not collide with the TA's hand right after the loading is completed. Once the next phase is initiated, the node in charge of the loading phase (/load_mode) would be destroyed, and the node for phase 4 will be activated (/alignment).

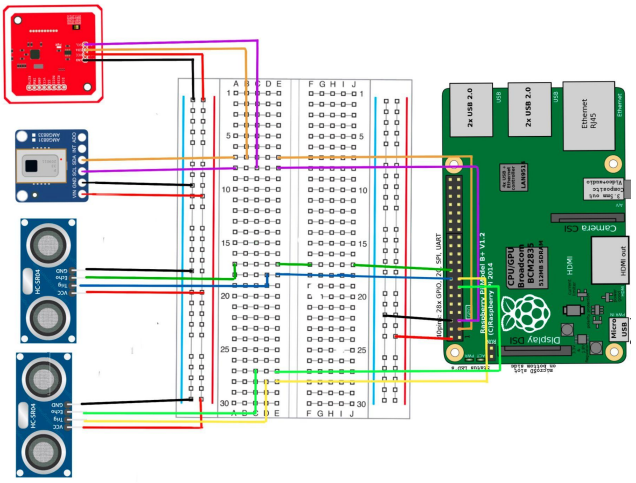
Using ultrasonic sensor to measure distance between thermal object and Turtlebot3 [Changed to using LIDAR data]

For the heated object ultrasonic sensor, the Trig and Echo pin is connected to pin 15 and 13 of the RPI respectively. When the emitter emits a burst of 8-pulses of ultrasound at 40kHz, the Echo pin will be set to state HIGH. This pin will remain high till it receives the reflected pulse. By calculating the time taken for the state to change, the distance between the heated object and the Turtlebot3 3 can be calculated. The Turtlebot3 will then use the information to either move forward or backwards to reach the desired distance. However, upon consideration we realized that we could just use the LIDAR to determine the distance of the thermal object, making this ultrasonic sensor redundant.

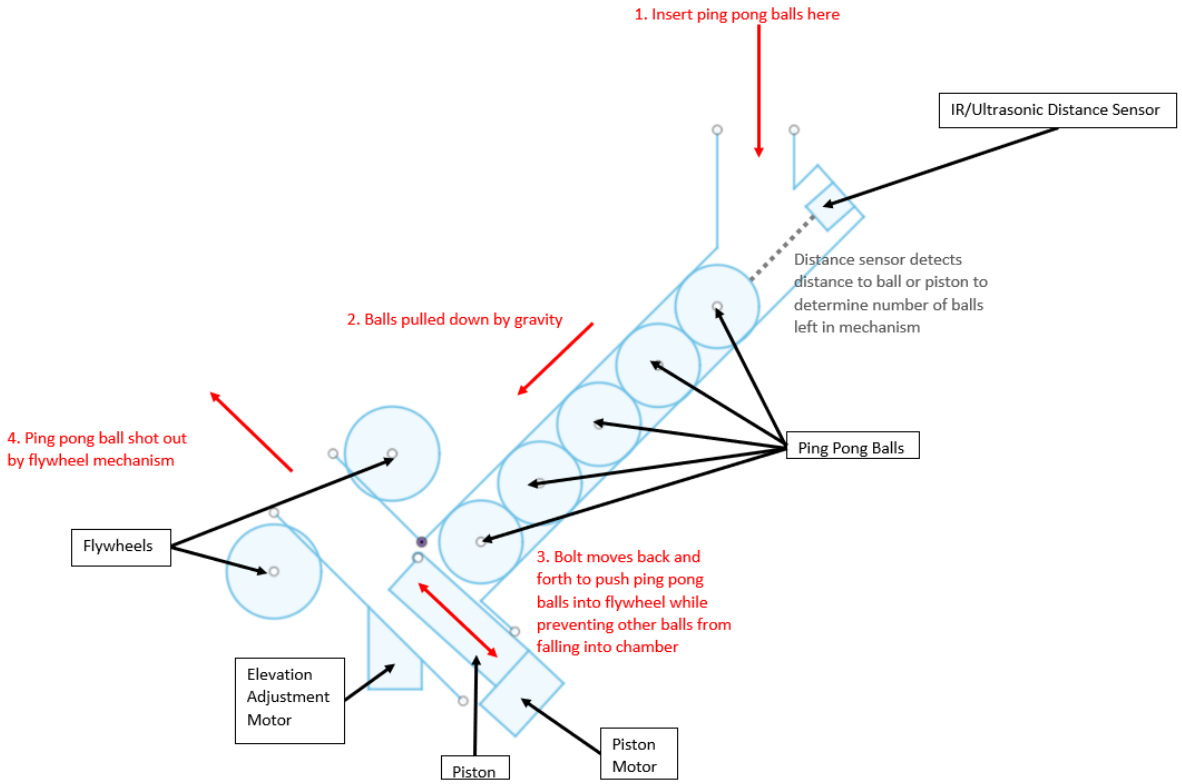
Deconflicting of ultrasonic sensors [Unnecessary when all the ultrasonic sensors removed]

Initially, when more than one ultrasonic sensor was used, the pins required by the two ultrasonic sensors would have to be deconflicted since they all require the GPIO pins to send data back to the RPI. There is also the added constraint in which the physical pins 3 and 5 cannot be used here since it would be used for the I2C required by the NFC and thermal camera. Hence, for the one for checking the loading progress, its Trig pin would be connected to the physical pins 16 and Echo pin would connect to pin 18 on the RPI. For another ultrasonic sensor, the Trig pin is connected to GPIO 22 [pin 15] which would be configured as an output pin, and the Echo pin would be connected to GPIO 27 [pin 13] on the RPI which would be configured as an input pin. However, this was no longer needed once we removed the need for the ultrasonic sensor to measure the distance between the Turtlebot3 and the thermal object.

Initial Circuit Diagram with two ultrasonic sensors



Initial Launcher diagram



Power Budgeting Table

Component	Voltage	Current	QTY
Running off LiPo battery			
R. Pi Board	5V	2500mAh	1
OpenCR Board	11.1V	1800mAh	1
Dynamixel XL430-W250 Motor	11.1V	1300mAh	2
LDS-01	5V	400mAh	1
Adafruit PN532 NFC RFID Module	3.3V	100mAh	1
AMG8833 IR Thermal Camera	3.3V	100mAh	1
Total Power consumption from LiPo battery			

Running off 4x Parallel AA Battery			
DC Motor (Flywheel + Linear Actuator)	6V	60mAh	3

Preliminary Power Budgeting Table

Component	Voltage	Current	QTY	Power Consumption
Running off LiPo battery [11.1 V 1800 mAh / 19.98 Wh]				
Turtlebot3 (During Operation)	Data from E2 Assignment		1	8.0W
Adafruit PN532 NFC RFID Module	3.3V	150mA	1	0.495W
AMG8833 IR Thermal Camera	3.3V	4.5mA	1	0.015W
Total Power consumption from LiPo battery				8.51W
Total Operation Time from LiPo battery				19.98Wh / 8.51W = 2.35h
Running off 4x Parallel AA Battery [4 x 1.5V 2.5Ah / 3.75Wh]				
Component	Voltage	Current	QTY	Power Consumption
DC Motor (Flywheel + Linear Actuator)	6V	180mA	3	3.24W
Total Power consumption from 4x Parallel AA Battery				3.24W
Total Operation Time from 4x Parallel AA Battery				3.75Wh / 3.24W = 1.16h