AutoCart

Carlos Gonzalez, Zehong Lin, Zeyi Huang: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— In this paper we propose AutoCart: a mobile robot based basket system that guides shoppers through a grocery store by leading them through the location of each item on their shopping list while also carrying all of the shoppers items in its basket. AutoCart results in a higher rate of customer throughput which increases revenue for the store while also increasing customer satisfaction by eliminating the hassle of carrying a basket or pushing a cart.**

***Index Terms*—Autonomous, iRobot, Line Following, Mobile, Path Planning, Rangefinder, Robot, Web**

1 INTRODUCTION

There are 3 common problems that grocery stores face. First, shopping carts and baskets are heavy and inconvenient when more than a few items need to be carried which results in poor customer satisfaction. Next, grocery stores are full of people who all would like to purchase similar items and so aisles containing popular products have a slow rate of traffic through them which slows customer traffic and thus causes a bottleneck for increasing a store's revenue per hour. Lastly, grocery stores often rearrange items which leads to items changing location in the store.

In order to solve these problems, we propose AutoCart. AutoCart is a mobile robot basket platform that guides customers through the grocery store by leading them to efficiently retrieve every item on their shopping list. AutoCart eliminates the need for customers to carry a basket, while simultaneously increasing the flow of traffic through a store which leads to an increase in customer satisfaction and an increase in revenue per minute for a grocery store. Autocart is able to guarantee that it takes the optimal path through the store while also guiding the customer at a walking speed of 5 Km/Hr. Furthermore, Autocart is able to stop within +-0.25m of any object in the store, carry loads of up to 25Kg, and sense items and continue guiding customers through the store with a lag time of less than 2 seconds.

2 DESIGN REQUIREMENTS

We have chosen six design requirements and a test to verify each metric. If all 6 tests are passed then we can be sure that AutoCart is a success.

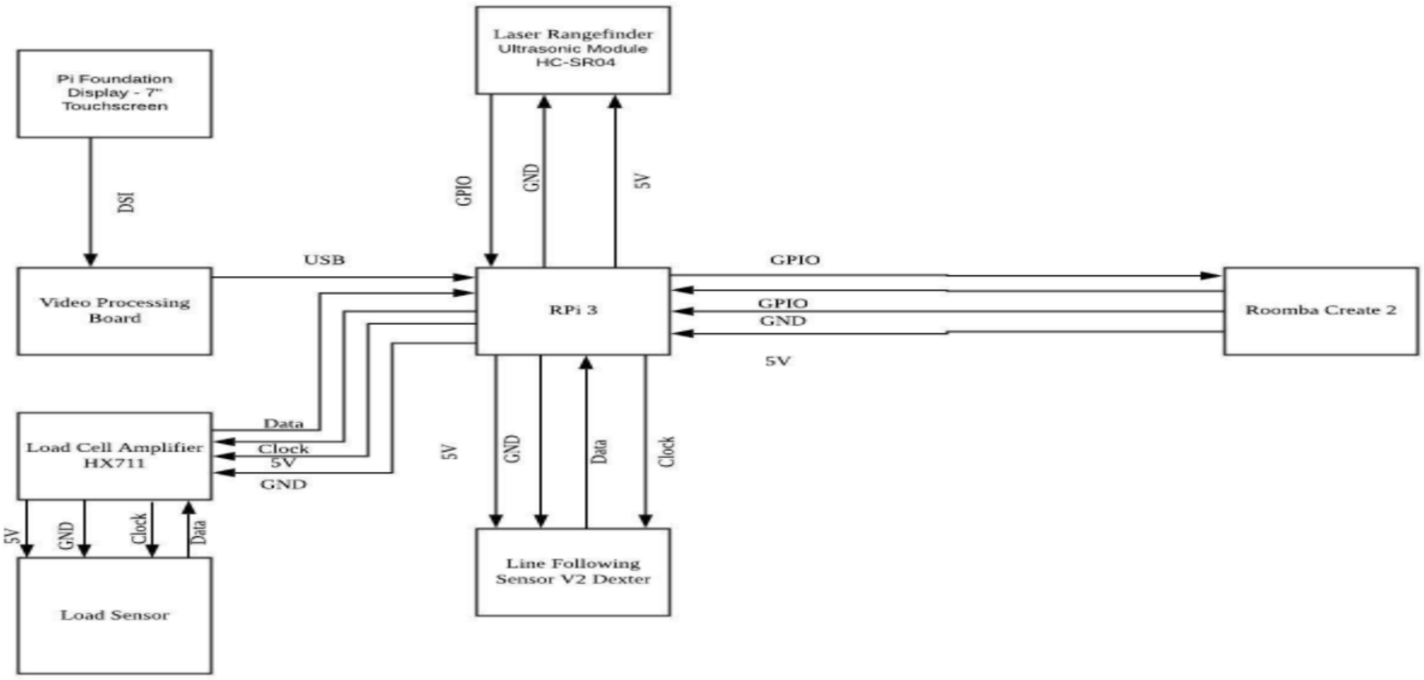
The first requirement is that AutoCart must be able to maintain a minimum traveling speed of 5 kilometers per hour. This metric was chosen because 5 kilometers per hour is the average human walking speed and so will make a perfect cruising speed for someone following AutoCart through the store. This metric will be tested by measuring the average speed of AutoCart when traveling between the 10 most common items. To be more specific, we will pick one of the ten most popular items as the starting location. Then we will measure AutoCart's average speed to each of the 9 other items. If the average speed is greater than 5 Km/hour then the first part of this test is considered passed. Next we will change the starting location to another one of the 10 most popular items and repeat the process. This process will continue until we have measured the average speed using all 10 items as the starting point. If all averages are greater than 5 Km/hr, this test was passed.

The second requirement is that AutoCart must stop within +-0.25m of all requested items. This metric was chosen because if AutoCart is to stop any farther than this from the desired item it would lead to the shopper having to inconveniently walk over to the item and then walk an uncomfortable distance back to AutoCart while carrying the potentially heavy item which would lead to a decrease in customer satisfaction. This metric will be tested by placing AutoCart at a random location in the store and having it navigate to the 10 most common items in the store. The test will be passed if AutoCart always stops within +-0.25m of each of the 10 items.

The third requirement is that AutoCart must always go to the correct location of the desired item. This metric was chosen because if AutoCart is ever to go to the incorrect location, then we are not increasing efficiency and thus revenue per hour for the store will decrease. We will be testing this metric by placing AutoCart in a random location in the store. Next, we will then ask it to go to the location of a random item. We will repeat this process 10 times and the test will be passed if and only if AutoCart travels to the correct location of all 10 items.

The fourth requirement is that AutoCart always takes the optimal path to pick up the customer's desired items. This metric was chosen due to the fact that grocery stores's key interest in AutoCart will stem from the increased rate of customers through the store which leads to a higher revenue for the store. This metric will be tested by placing AutoCart in a random location. Then, one by one, we will input an item for AutoCart to drive to. Once AutoCart reaches its destination, we will manually verify that the route it took to get there was optimal. We will repeat this process for 10 items and the test will be passed if and only if AutoCart takes the optimal route for all 10 items.

The fifth requirement is that AutoCart must sense an item placed into its basket in 90% of cases. 90% was chosen because we want to balance the inevitable inconsistencies that will occur from sensing an object with making a platform that is robust enough that it will not cause an inconvenience to shoppers. Furthermore, 90% was chosen



Pi Foundation Display - 7" Touchscreen
DSI
Laser Rangefinder Ultrasonic Module HC-SR04
GPIO
GND
5V
Video Processing Board
USB
RPi 3
GPIO
GPIO
GND
5V
Roomba Create 2
Load Cell Amplifier HX711
Data
Clock
5V
GND
5V
GND
Data
Clock
5V
GND
Clock
Data
Line Following Sensor V2 Dexter
Load Sensor

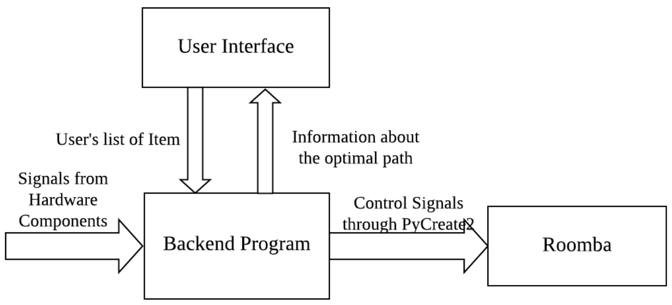
Figure 3.1: hardware block diagram

because we will be including a button on the GUI that the user interfaces with in real time so that if an item is not detected, the user can tap on the button and AutoCart will proceed to the next time. In this way, the 10% of cases in which an item is not detected will not cause a great inconvenience to the customer. We plan on testing this metric by placing 20 items of weights ranging from 0.5kg to 10kg and noting whether the item was detected. If 18 out of the 20 items are detected, this test was passed.

The sixth requirement is that in 90% of cases that AutoCart senses an object, it must begin to travel to the next object in less than two seconds. Two seconds was chosen because it offers enough time for the customer to place an item and be ready to head to the next one without taking so long as to inconvenience them. In a similar manner to our fifth requirement, we will be testing this by placing items one at a time into AutoCart's basket. If the item is detected, AutoCart will have 2 seconds to begin going to the next object. In this test, if an item is not detected, we simply discard the item and the sample. Since we are measuring the response time after detection, we will only consider a sample to be valid if it is detected and thus will count the sample as a success, once detected, AutoCart moves to the next item within 2 seconds. We will run this process for 20 detected items and the test is considered passed if and only if AutoCart moves to the next item in 18 out of the 20 samples.

3 ARCHITECTURE OVERVIEW

The hardware architecture of our system is described in the Figure 3.1. The RPi receives signals from other hardware parts and do computations that will be elaborated in the software section. Then, RPi sends signals to the iRobot Roomba Create 2, connected through USB, in order to control the robot's motor as well as direction. A line-following sensor is attached at the bottom of the Roomba, and connected to the RPi through I2C. Working with the lines that we place on the map of the market, the line-following sensor will help the robot to keep itself at the center of aisles and to locate itself on the map. As a result, the robot can accurately travel to items on the user's list. To allow the robot to start moving automatically after items at the current position is put on it, load sensor is installed onto our system; the read of the load sensor helps the program running on RPi to decide whether an item has been placed. An amplifier is needed for the load sensor because of its weak signal, and they will be hooked up through I2C. The amplifier will also be connected to the RPi with I2C. To prevent the robot from crashing into customers in the store, we decide to use an ultrasonic rangefinder to detect obstacles on the path and the rangefinder is connected to the RPi through I2C. Lastly, a touchscreen display is needed to allow the interaction between the user interface and the user. It is first connected to a video processing board through DPI and the board is connected to the RPi using USB.



User Interface
User's list of Item
Information about the optimal path
Signals from Hardware Components
Backend Program
Control Signals through PyCreate2
Roomba

Figure 3.2: software block diagram

The software of the system (Figure 3.2) will be on the RPi and is in charge of interacting with users, calculating optimal path, processing the signals from other hardware components, and controlling the Roomba. It asks the user for list of items as input and stores an optimal path would be calculated with DP. DP is used here for path planning, because it can makes it easier to plan an alternative path if an obstacle stays on the current path for a long time. Information about the optimal path is then displayed to the user through the user interface. In the process of following the path, the program would use signals from the line-following sensor and the map inputted beforehand to determine the robot's location on the map, and send signals of motor and direction control to the Roomba, using Python interface PyCreate2. The signals from the rangefinder can be used to prevent crash, and determine if re-planning is necessary.

4 SYSTEM DESCRIPTION

4.1 iRobot Create 2

This serves as the mobile base of the AutoCart. We will be commanding the motor speeds of iRobot Create 2 using Python API pycreate2 from Raspberry Pi 3. We will use the `drive_direct(self, right_velocity, left_velocity)` to control the velocity of right and left motors separately. Specific speed for each motor will be calculated using sensor value returned from Line Follower Sensor which will be discussed below.

4.2 Raspberry Pi 3

The Raspberry Pi 3 will be connected to iRobot Create 2 using Create 2 USB to Serial Cable which is a 7-pin mini-DIN cable on the Roomba side and a USB cable on the Raspberry Pi 3 side. It receives signals from the load sensor, and line-following sensor and a back-end program will be running on the RPi to process these signals.

4.3 User Interface

From the user interface, the user should be able to add items that they want to purchase, select the quantity of the items, see the item that the cart is currently heading to and the next item that the cart to heading to, and command the cart to skip the next item on the list.

Figure 4.1 describes the interaction between the user and the user interface, and the interaction between the user interface. First, the user click start button on the starting page (Figure 4.2), to confirm that they will start shopping. Then, the user interface would be activated and present to a list of items and through which the user can select the items they are going to purchase (Figure 4.3). After pressing the "submit" button on the user interface, the items selected would be sent to the back-end and call the path planning algorithm. The path planning algorithm generates an optimal path and sends it back to the user interface.

Then, the user interface switches to the guiding page (Figure 4.4) and the cart would start guiding the user according to the planned path. A map of the store, the current and next node that the cart are displayed to the user. Also, the user interface the list of items selected by the users, in the sequence in which they will be visited, and a list of nodes the cart is going to visit. When the cart arrives at an item on the list, it will guide the user to pick up the item on the left or right shelf. In case that the cart fails to sense that items are placed on it, or the user no longer wants to purchase the current item, the user can press the "skip this item" button to instruct the cart to move on. The "Done Placing Item" and "Place an Obstacle buttons" are for simulation purpose and will be elaborated in section 6.2.

After all the items on the list are visited, the user interface will control the cart to return to the place of carts return, and reset itself to the starting page, indicating that it's ready for use.

4.4 Tablet

We will be using the Pi Foundation Display – 7" Touchscreen Display for Raspberry Pi as a display of the user interface to the user. The touchscreen can be easily connected to the Raspberry Pi, with the help of an adapter board; only two connections to the RPi are required: power from RPi's GPIO port and a ribbon cable connection to RPi's DSI port. A detailed build instruction can be found on the product's official page. The product well fits as our display of the user interface. It is highly interactive, with drivers that support a virtual 'on screen' keyboard, and would be comfortable and easy to user for users in a grocery store.

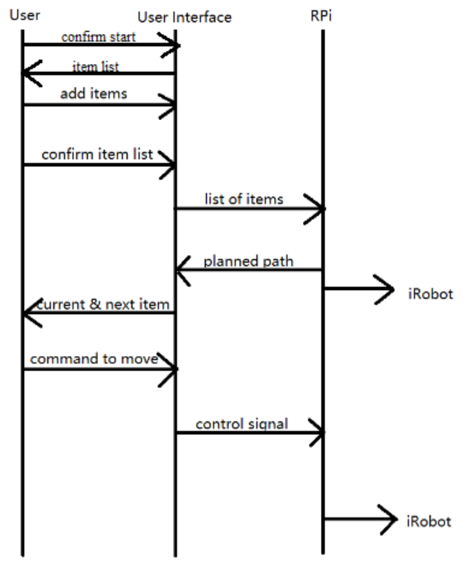


Figure 4.1: User Interaction Diagram

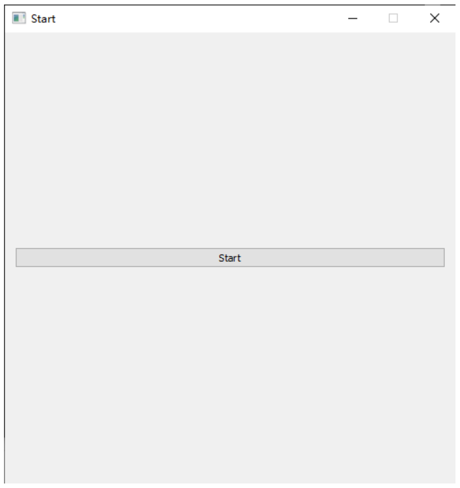


Figure 4.2: Starting Page

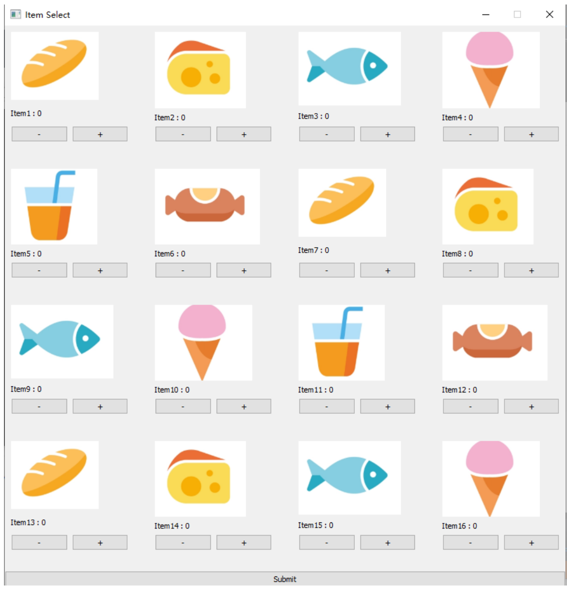
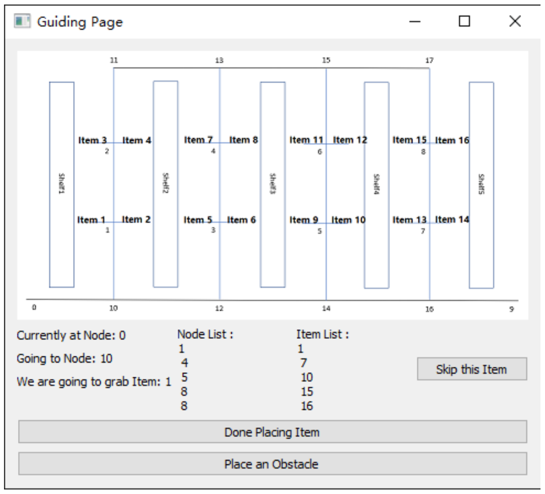


Figure 4.3: Item Selecting Page



Currently at Node: 0
Going to Node: 10
We are going to grab Item: 1

Node List :
1
4
5
8
8

Item List :
1
7
10
15
16

Figure 4.4: Guiding Page

4.5 Load Sensor and Amplifier

We will be using a Load Sensor with its corresponding amplifier in order to measure the weight of items placed into the basket. The load sensor is able to measure to within 1 kg of the weight in the basket. We will mount this at the bottom of the basket and we will use it to detect when an item is placed into the basket. The connection between the amplifier and the Raspberry Pi is done via i2c.

4.6 Line Following Sensor

We will be using Dexter Line Follower Board (black) to perform line following. Conveniently, DI-sensors have well-documented Python API for interacting with the line follower sensor. This board is consist of 6 IR sensors and integrates the sensor values from these 6 IR sensors to compute one sensor value from 0.0 to 1.0. If the black line is under the left half of the line follower board, the returned value will be in the range of 0.0-0.5; if the black line is under the right half of the line follower board, the returned value will be in the range of 0.5-1.0, thus making 0.5 the center point of the black line. With this value, we can write a PID-based algorithm for line following. Learning from the starter code given in the API, the tentative code for line following is included in the Appendix.

4.7 Laser Rangefinder

We will be mounting one Laser Rangefinder at the front of the iRobot Create. This will allow us to incorporate obstacle avoidance into our system by allowing us to constantly poll the sensor for an obstacle within its line of sight. If an obstacle is detected within 0.5m, then the iRobot Create will stop. It will then wait 20 seconds and if after 20 seconds the obstacle has not moved, it will then replan and turn around and follow another path to its destination. In order to accomplish this, we will be using an HC-SR04 Distance Sensor which has a detection range of 2cm - 500cm with an effectual angle of 15 degrees and precision of +- 0.3cm.

4.8 Path Planner

Our group came up with our original path finding algorithm to best address the problem at hand. We will go over why we did not go with some of the off the shelf algorithms that solve Traveling Salesman problem in the Design Trade Off Studies section. The path finding algorithm takes in three inputs: the starting point, the ending point, and `requiredNodes`: a list of nodes that must be visited. It returns the optimal route as a list of node in the order they should be visited. The path finding algorithm has two big stages two it. Stage one is a preparation step that involves the use to A* algorithm. Stage two applies an iterative greedy approach to generate the optimal route bit by bit. This stage is consist of multiple steps that I will explain below.

4.8.1 Stage 1

Stage 1 uses A* to find the shortest path between each and every pair of nodes in the map. I chose to use Euclidean distance as the heuristic function because it takes constant time to compute as long as the x and y-coodinates of the nodes are given. This step serves as a preparation for the second stage. Although each A* runs really fast, usually less than 0.1 seconds, the number of A* that needs to be computed grows fast, $O(N^2)$, with respect to number of nodes in the map. As a result, with a map of 1000 nodes, stage 1 can take up to 30 minutes to compute. The saving grace here is that stage one can be pre-computed for each map ahead of time, and it does not need to be recomputed unless the map changes permanently. As a result, the algorithm save the result of stage 1 alongside with all the map information in pickled file locally. At run time when user enters a shopping list, the algorithm loads in these information and goes directly into stage two.

4.8.2 Stage 2

Stage 2 is where the majority of the design work went in. It has multiple stages. Stage 2 always starts with a current_optimal_route which is the shortest path from origin to destination. It then iteratively builds or alters this route with a greedy approach to eventually include all required nodes. If any required nodes are already on the shortest route from origin to destination, they are removed from `requiredNodes` list before any iterations of stage 2 is ran. For ease of understanding, I will use a contrived example to demonstrate one iteration of stage two.

Assume we have a map in which all edges have the same length. On this map, we are planning a route that starts at 0, ends at 9, and must visit [2,4,6]. After some iterations, the current optimal route has become $0- > 1- > 2- > 3- > 2- > 1- > 9$.

1. Step 1: the first step of every iteration of stage 2 is figuring out where can the next required node be inserted into the `current_optimal_route`. This is equivalent to finding all the pairs of nodes in the current optimal route that either: 1.does not have any required nodes in between them, **OR** 2.only has required nodes that are also visited somewhere else in the route between them. A node can be a pair with itself. Once we finish this step for the above example, we should arrive at this list. `possible_insertion_pairs` = [(0, 1), (0, 3), (0, 2), (1, 2), (1, 3), (2, 3), (3, 9), (1, 9), (0, 0), (1, 1), (2, 2), (3, 3)]. If duplicate pairs arise in this process, we keep the pair that is furthest apart. For example, pair (0, 2) occurs once at index 0 & 2 and occurs again at index 0 & 4. Because of the assumption that all edges have the same length, the second (0, 2) is further apart; so, the second pair is kept.

2. Step 2: The second step of every iteration of stage 2 is figuring out which required node that has not been visited yet can be inserted into the current optimal route at least cost. This step is when the shortest paths computed in stage 1 come in handy. For required node x, and possible insertion pair (i, j), the cost function is: length of shortest path from i to x, plus length of shortest path from x to j, minus the length of the current path from i to j or in pseudo code: `len(shortest_path(i, x)) + len(shortest_path(x, j)) - len(current_path(i, j))`.

3. Step 3: After finding the least cost way to insert the next unvisited required node, last step of every iteration of stage 2 is updating the current optimal route accordingly. If we go back to our example, say the next unvisited required node to insert is 4, best insertion place is (0, 2), and the best way to visit 4 is going from $0- > 5- > 4$ and going back directly from 4 to 2. Then the current route is updated to $0- > 5- > 4- > 2- > 1- > 9$. Notice how when updating, node 1, 2 and 3 (originally at index 1, index 2, and index 3) are deleted. We can do this because node 2 is visited again somewhere else in the route, and node 1 and node 3 are not requited nodes.

Stage two continues until all required nodes are visited. At this point, the current optimal route is the overall optimal route.

4.8.3 Final augmentation

Now the hard work of planning is done, the optimal route sometimes need to be polished a bit more in order to better guide Auto Cart is a realistic setting. If more than one item is needed at a particular node, that node is repeated multiple times in the overall optimal route to match the number of items needed on that node. This is to allow the functionality of skipping items. With the multiplicity of nodes, user can choose to skip one item at a node without skipping all other nodes at the same node.

4.8.4 Re-planning

Auto Cart has the ability to detect obstacle and avoid them while navigating the shopping mall. Obstacle detection is covered in the sensor section, we will go over how the path planner re-plans a route to circumvent obstacle in this section. Put simply, when an obstacle is encountered, the path planner alters the map temporarily to reflect the position of the obstacle and performs A* to circumvent the obstacle in a locally optimal way. Once the obstacle is circumvented, Auto Cart gets back onto the planned optimal route and continues execution from there.

First let us notice a fact about the optimal route generated by the path planner. It is always a neighbor-to-neighbor route, which means that adjacent nodes in the route are neighbors. This is both a simplification and a complication at the same time. It is a simplification because if Auto Cart were to detect an obstacle while following the planned route, the path planner knows exactly where the obstacle is at. This is a complication because all the shortest paths computed in Stage 1 of path planner are not useful any more. This is because if two nodes are neighbors, then it is very likely that the shortest path between these two nodes is just going directly from one to the other. In other words, the way path planner is written makes that if an obstacle were to appear between two nodes in the optimal route, the obstacle is almost always guaranteed to be on the shortest path between those two nodes. This means that re-planning is almost always needed. However, the good thing is that one iteration of A* is very cheap.

Before re-planning, the path planner first removes the edge that has the obstacle on it. It then performs A* to find the new shortest path between the two neighboring nodes, Node1 and Node2, that have an obstacle between them. Auto Cart pauses its execution of the overall optimal path and executes this newly re-planned alternative path from Node1 to Node2 first. After executing this newly re-planned alternative path, Auto Cart should have circumvented the obstacle and should be back on to the overall optimal path. At this point, Auto Cart continues execution of the overall optimal route until another obstacle is met.

After re-planning, the removed edge is put back into the map. Running one A* is very cheap, so if the user encounters the same obstacle again, it does not hurt to run A* again on the same Node1 and Node2. We put back the removed edge also because we think it is reasonable to assume all obstacles are temporary; removing an edge permanently might reduce the optimality of potential later re-planning.

4.8.5 Testing

To perform testing, we first need to generate maps of our own. We first randomly generate the x and y coordinates of n nodes. The weight(length) of edge between each pair of nodes is always the euclidean distance. We chose this to be the weight to avoid potential problems with the heuristic function in A* (Heuristic function must be admissible in order for A* to return the least-cost path. This means that the heuristic function should never overestimates the actual cost of going from node A to node B). After generating the nodes, we generate the edges. In order to test with maps of different sparseness, we only choose to include a certain percentage of all edges. When we include 100% of all edges, then the map is fully connected and these maps are usually the most challenging ones within the same node count.

While testing, we also alter the number of nodes that need to be visited. We do this by randomly marking y% of all nodes to be required nodes. For each map with n nodes and x% sparseness and y% required nodes, we perform 50 different tests. Each test has a randomly selected origin, a randomly selected destination, and a randomly selected list of nodes that need to be visited.

We tested with map up to 1000 nodes and up to 50% sparseness. We stopped at 50% sparseness because we think it is unfeasible for supermarkets to have a map that is denser than that. There does not exist a direct path from the middle of an aisle to the middle of another aisle. For each test, we record and calculate the run time and check for correctness.

1. Correctness testing: After the algorithm runs and finds a optimal route, we check for correctness by checking whether `optimal_route[0] == origin`, `route[-1] == destination`, and whether all required nodes are in `optimal_route`.

   Our algorithm's result is correct for all tests. The most computationally intensive testing we did was map with 1000 nodes, 50% sparseness, and 200 of those nodes need to be visited. The planning can be done within 2.5 minutes. One caveat to this is that stage 1 of path planner this map takes around 30 minutes. However, this is not an issue because our algorithm pre-computes stage 1 and saves the information. When each customer submits his or her shopping list, no stage 1 needs to be run.

   We did not check whether planned route is actually optimal for most of the tests. This is because we did not find a good way to verify optimality. However, we did check for optimality on small maps. The algorithm does always return the optimal route.

2. Stage 1 run time bench-marking: We tested and recorded time of running stage one on maps of 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 nodes with 10% edge density. The result is in the following graph:

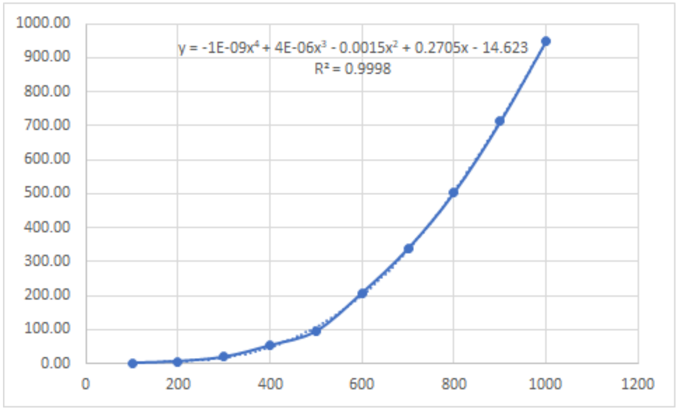


Figure 4.8.5.1

   We can conclude that time required to run stage one increases polynomially with number of nodes in map.

3. Stage 2 run time bench-marking: after performing the 50 tests on each setting, we record the average run time and graph them in excel. We mainly looked at two variables' impact on run time: number of required nodes and number of total nodes in map. We draw a trend line in excel to evaluate the complexity.

The first graph is running stage 2 on maps with 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 nodes with 10% edge density and 100 required nodes.

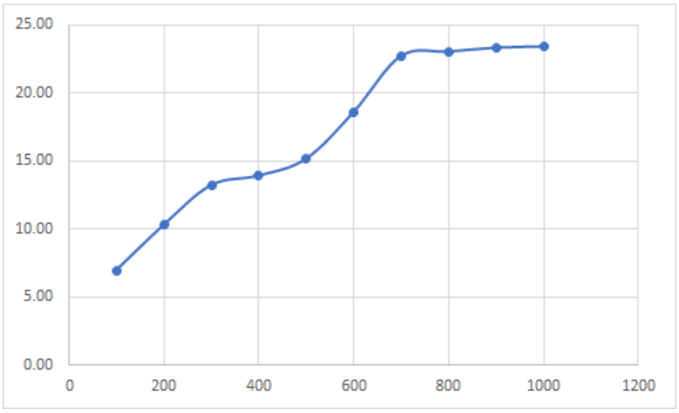


Figure 4.8.5.2

We see a unusual trend. It looks like the run time remains the same for a range of number of nodes. The run time then increases linearly until it hits the next range.

The second graph is running stage 2 on maps with 1000 nodes, 10% edge density, and varying number of required nodes.

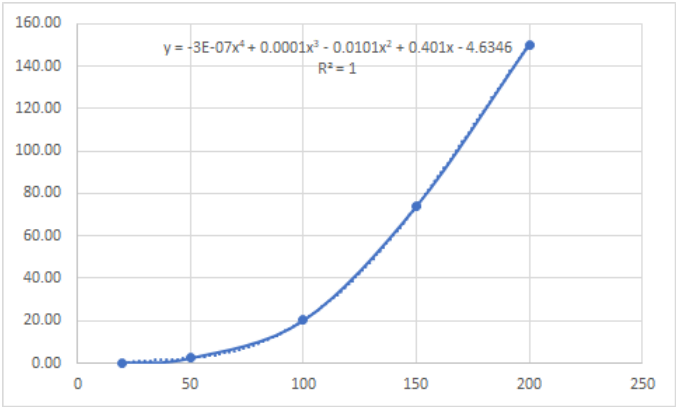


Figure 4.8.5.3

Here we can see clearly that run time increases polynomially with number of required nodes. This turns out to be a better theoretical run time than the best off-the-shelf TSP solver.

4.8.6 Interesting additional findings

We stumbled upon the following findings unexpectedly. We did not set out to thoroughly investigate these findings as they do not pertain to the core of our project. However, we mention them here as they can potentially be interesting projects for future ECE capstone or research.

1. It is possible that a planning on a denser map is actually faster. Most of our run time testings were done on maps with 10% edge density, which means only 10% of all possible edges are included. We feel like this is a reasonable assumption because most supermarket maps should be very sparse.

5 PROJECT MANAGEMENT

5.1 Schedule

Our GAANT Chart has been inserted at the end of the end of the document in Appendix A.We have revised the GAANT Chart after the spring break, with some changes to the focus of work for each member. In the second half of the semester, Carlos focused on the testing of sensors, Zehong and Zeyi continued to implement their own software portions and worked together on software integration.

5.2 Team Member Responsibilities

The 6 primary responsibilities of the project are: Path Planning, Control, Interfacing with sensors, User Interface, Tablet GUI, and Interfacing with the Roomba. They are split up as follows:

Carlos	Zeyi	Zehong
Control	Tablet GUI	Roomba Interface
Sensor Interface	User Interface	Path Planning

Zeyi's primary responsibility will be the Tablet GUI and User Interface. Zehong's primary responsibility will be Path Planning and interfacing with sensors. Carlos's primary responsibility will be interfacing with the Roomba and Control.

The secondary responsibilities are split up as follows: Carlos will aid with Path Planning and the Tablet GUI, Zehong will aid with Control and the User Interface, and Zeyi will aid with the sensor interface and Roomba interface.

5.3 Budget

The bill of materials is attached at the end of the project. The proposed cost is 380$.

5.4 Risk Management

The main risk of our project is in the testing phase. The sensors that are installed outside the Roomba could be broken if the robot crashes into objects. Considering the risk of breaking the sensors, we have ordered more sensors than we needed. Also, in the case that the Roomba is not functioning, we have left enough budget space to purchase another one. Furthermore, there may be problems with keeping track of the vehicles location throughout the store. If this happens, we will have to pivot to a more robust map implementation that will use multiple landmarks in order to develop a more robust localization.

6 Design Trade Off Studies

Our design trade off focused on 3 key areas: varying the number of sensors, varying the path planning algorithm, and varying the user interface.

6.1 1 sensor vs. 2 sensors

For the sensors, we wanted to know whether 1 sensor was sufficient to detect obstacles position in front of the iRobot Create or whether 2 sensors at a varied distance were needed. In order to test this, 2 tests were created.

The first test consisted of detecting 3 objects of widths 0.1m, 0.3m, and 5m from a distance of 0.5m, 1m and 1.8m. The widths of the objects were chosen specifically because they were the widths of objects that we are likely to encounter in our project. To be precise, the 0.1m width simulates detecting a human ankle positioned in front of the iRobot Create whereas the 0.3m object simulates detecting another iRobot Create that is positioned in front of our current robot. Lastly, the 5m width simulates a wall that is in front of our robot. This test was created in order to find the maximum distance at which a range sensor could reliably detect common objects.

The results of this test were as follows: all of the objects were detected from a distance of 0.5m and 1m. However, only the wall was detected from a distance of 1.8m. After these results, we ran a binary search (manually) to find the farthest distance at which the objects of width 0.1m, and 0.3m were detected. Both of these objects had a maximum detection distance of 1.3m. Due to these results, we can guarantee that the range sensor will detect other robots and humans from 1.3m away while also guaranteeing that walls can be detected from 1.8m away.

The second test consisted of setting the same 3 objects up at their maximum detectable distance found in test 1 and varying the angle at which they pointed at the object from 0 degrees to 15 degrees in intervals of 3 degrees.

The results are as follows, all objects were detected when the object was within 9 degrees of the field of view, but none of the objects were able to be detected when the range sensor field of view varied by more than 9 degrees. The image below better conveys the definition of the field of view.

In order to better understand the sensors, we went about deriving a theoretical model for the maximum

Figure 6.1, shown below, illustrates the model used to derive the maximum theoretical field of view of the rangefinder sensors. The derivation is straightforward:

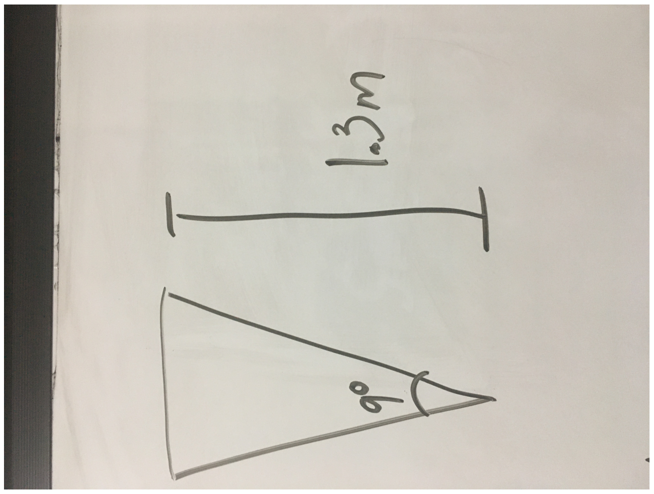


Figure 6.1: Sensor Field of View

6.2 User Interface Changes and Trade off

For the user interface, we decided to change our design according to the facts that the project's hardware portion of the project can no longer be integrated.

Firstly, different from what we proposed in the design document, a starting page is added to the user interface. The starting page let the users confirm that they are going to start shopping, before actually using the interface. This would help to save the resources for the grocery stores: consider the case that a user submits an item list but ends up skipping items on list; this is waste of the shopping cart resource and makes the waiting time of other users longer. The starting page prompts the user to think more carefully before using the cart. It would also be a clearer indication of the cart's idle state that it is available to be used.

The item selecting page is not changed much from the original design. Because the starting page added let the users confirm that they want to start shopping, the item selecting page now requires users to select some item before clicking the submit button and proceed.

Lastly, because the integration between hardware and software is no longer feasible, the simulation of a real-world situation cannot be done without the line-following sensor and Roomba. We decided to perform a simple simulation of the process of travelling the optimal path on the guiding page, with the assumption that line-following algorithm is always correct and the cart can visit the next node on its planned path as expected. The purpose of the simulation is to test the robustness of the software integration. The "Done Placing Item" button simulates that the weight sensors detects that the user has placed the current item onto the cart. It will then start travelling to the next item, or ask the user to grab the next item if the next item is on the other side of the current node. The "Place an Obstacle" button simulates the situation that the rangefinder sensors detect an obstacle on the current route. In this case, the user interface asks the path planning algorithm for a re-planned path around the obstacle. Also a map is added onto the top of the guiding page for a better user experience: with the map, users would have a better idea of their position in the store.

6.3 Path Planning Algorithm Trade Offs

6.3.1 Off-the-Shelf Traveling Salesman algorithm v.s. design our own algorithm

Initially we thought we could use an off-the-shelf algorithm that solves Traveling Salesman Problem (TSP) for our path planner. Since TSP has been studied substantially, using an off-the-shelf algorithm can guarantee correctness. We were initially a bit worried about run time, but we wanted to implement and test it first. If run time turns out to not meet our requirement, we will implement our own algorithm.

Then we researched more about traveling salesman problem. After research, we feel like it might not be the most accurate representation of the problem at hand. One thing that TSP does not allow is visiting a node more than once, but Auto Cart can visit any node any number of times as long as the total distance remains the shortest. Another thing TSP makes sure is that all nodes are visited, which is highly likely not the case for the usage of Auto Cart in a shopping mall. A user is very unlikely to have to visit every stop of every aisle. To address this problem, we first thought about building a sub-graph consist of only the nodes we need to visit. It is easy to figure out which nodes to keep, but when it comes to figuring out which edges to keep, it is unclear how that can be done easily. Also, more nodes than those that are required might need to be included in order to keep the graph connected. After discussion, we feel like constructing a sub-graph than guarantees to still contain the optimal route is just as computationally complex as finding out the optimal route directly.

As a result, we decided to go directly to designing our own algorithm.

6.3.2 Efficiency v.s. Optimally for re-planning

As we recall from the re-planning section of path planner, re-planning only considers what is the locally optimal way to circumvent the obstacle. It is possible that while circumventing the obstacle, some of the required nodes that are scheduled to be visited later are visited. The creates redundancy in the overall route. In order to truly re-plan an optimal path to circumvent the obstacle, the entirety of stage 2 potentially need to be re-run. This is very costly and we do not think it is worth the wait for the user.

7 Related Work

Our project is inspired by Cart-i B (`http://course.ece.cmu.edu/~ece500/projects/f18-team5/`), which builds a cart that follows the user in the store.

8 Summary

The goal of the project was to overall help improve the shopping process and ultimately create more convenience in a system where we see inefficiencies. Inspired like projects such as Amazon Go, we wanted to simplify the shopping process. Through the optimization path finding algorithm and user interface, we feel we have added to the realm of convenience in the shopping experience.

While we were not able to integrate the hardware portion of the project, we were able to make substantial process on the integration of all parts that were available to us. Moreover, our code is ready to simply make the hardware connections and automatically begin to poll the sensors that we were going to use prior to the impact of COVID-19. It should be noted that the parts of the project regarding the control of the iRobot Create with a Raspberry Pi were not implemented because of the unavailability of the platform. Lastly, we were not able to test the weight sensors since that would require the integration of the basket and the iRobot Create which we did not have access to. Despite this, we were able to change the focus of the project to one of a study of design trade offs instead of one with a focus on hardware integration.

9 References

1. DECHTER, RINA. Generalized Best-First Search Strategies and the Optimality of A* . UIUC, July 1985, www.ics.uci.edu/ dechter/publications/r0.pdf.

2. Delacroix, Malloy. A PyQt5 Example of How to Switch between Multiple Windows.

3. Goldberg, Andrew V., and Chris Harrelson . Efficient Point-to-Point Shortest Path Algorithms.

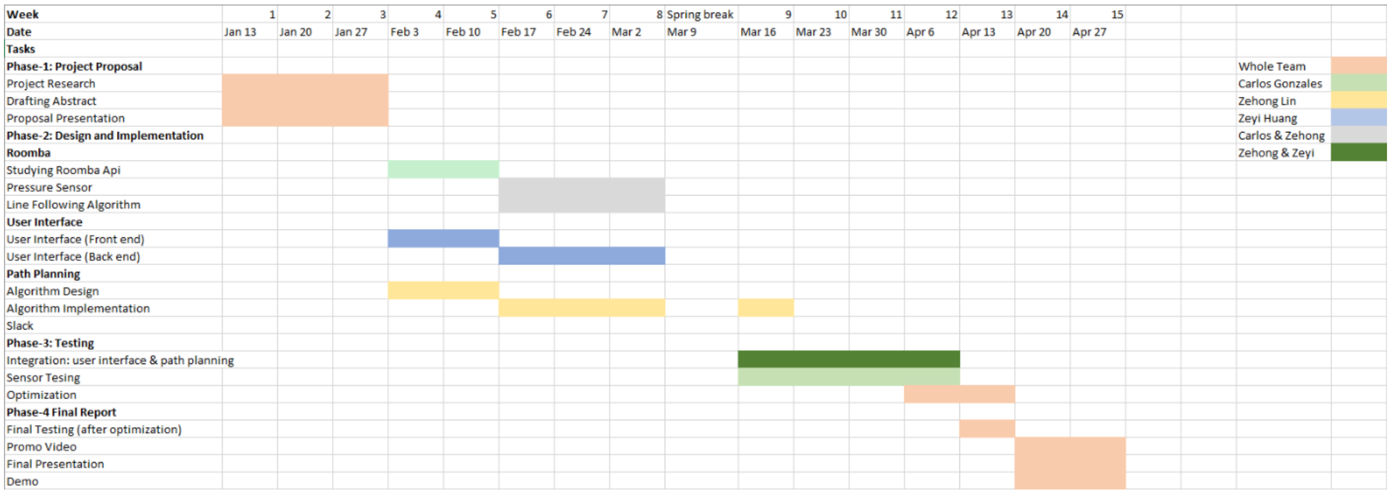
4. Kagan, Eugene. A Group Testing Algorithm with Online Informational Learning. Tel-Aviv University, Nov. 2011.

5. Okuntseva , Ana. "Icons for Groceries App." Dribbble, 2019, dribbble.com/shots/7144128-Icons-for-Groceries-App.

6. Sosic, Martin. "Martinsos/Arduino-Lib-Hc-sr04." GitHub, 18 Mar. 2020, github.com/Martinsos/arduino-lib-hc-sr04.

7. Using the Line Follower. Dexter Industries Revision, 2017.

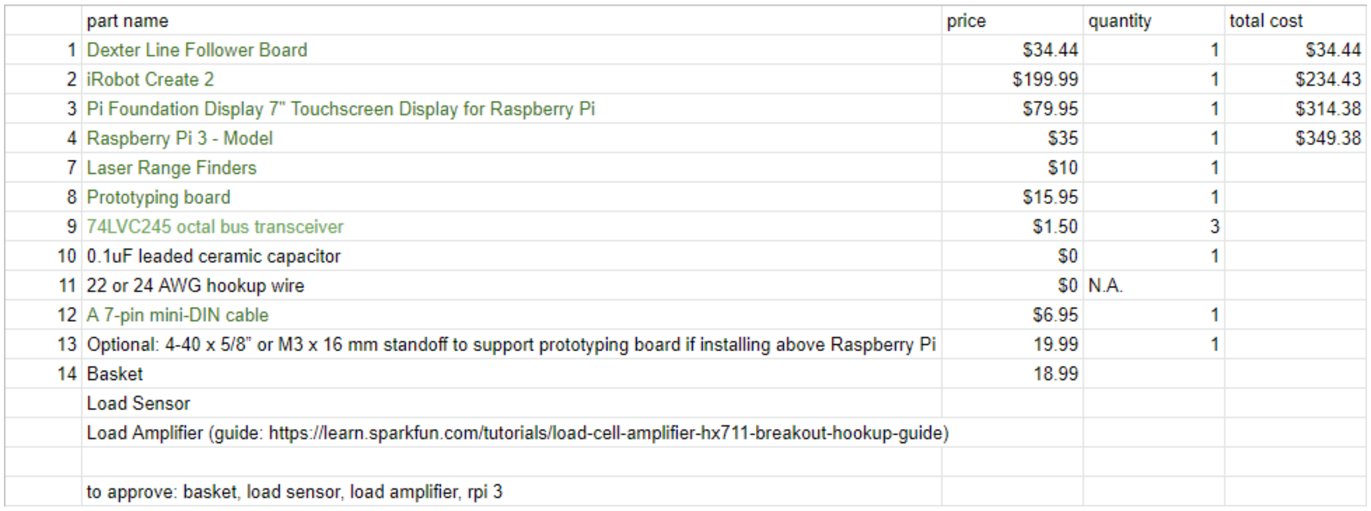
Appendix A



Week	1	2	3	4	5	6	7	8	Spring break	9	10	11	12	13	14	15
Date	Jan 13	Jan 20	Jan 27	Feb 3	Feb 10	Feb 17	Feb 24	Mar 2	Mar 9	Mar 16	Mar 23	Mar 30	Apr 6	Apr 13	Apr 20	Apr 27
Tasks																
Phase-1: Project Proposal																
Project Research	🟧	🟧	🟧													
Drafting Abstract	🟧	🟧	🟧													
Proposal Presentation	🟧	🟧	🟧													
Phase-2: Design and Implementation																
Roomba																
Studying Roomba Api				🟩	🟩											
Pressure Sensor						⬜	⬜	⬜								
Line Following Algorithm						⬜	⬜	⬜								
User Interface																
User Interface (Front end)				🟦	🟦											
User Interface (Back end)						🟦	🟦	🟦								
Path Planning																
Algorithm Design				🟨	🟨											
Algorithm Implementation						🟨	🟨	🟨		🟨						
Slack																
Phase-3: Testing																
Integration: user interface & path planning										■	■	■	■			
Sensor Tesing										🟩	🟩	🟩	🟩			
Optimization													🟧	🟧		
Phase-4 Final Report																
Final Testing (after optimization)														🟧		
Promo Video															🟧	🟧
Final Presentation															🟧	🟧
Demo															🟧	🟧

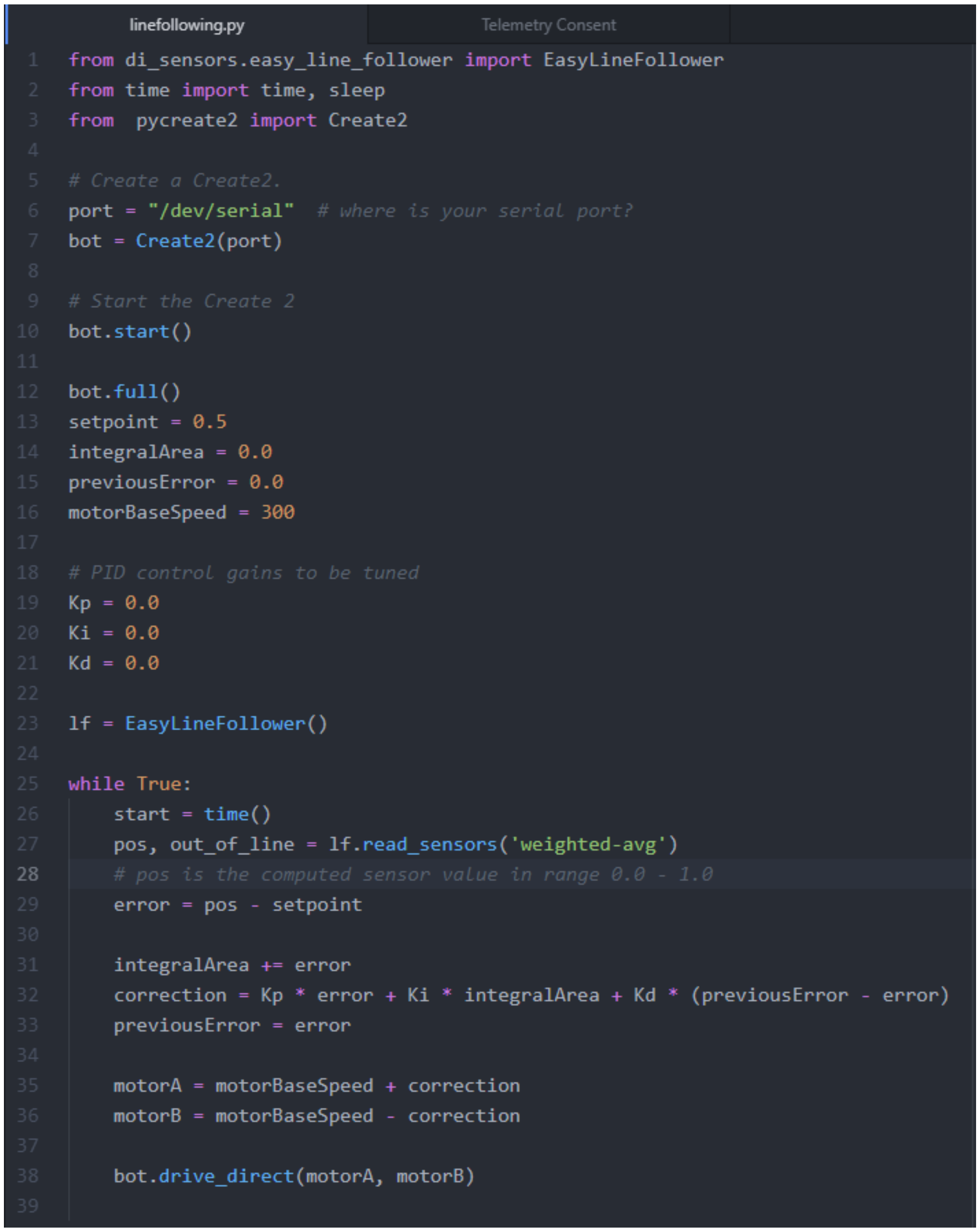
Whole Team	🟧
Carlos Gonzales	🟩
Zehong Lin	🟨
Zeyi Huang	🟦
Carlos & Zehong	⬜
Zehong & Zeyi	■

Gantt Chart



	part name	price	quantity	total cost
1	Dexter Line Follower Board	$34.44	1	$34.44
2	iRobot Create 2	$199.99	1	$234.43
3	Pi Foundation Display 7" Touchscreen Display for Raspberry Pi	$79.95	1	$314.38
4	Raspberry Pi 3 - Model	$35	1	$349.38
7	Laser Range Finders	$10	1	
8	Prototyping board	$15.95	1	
9	74LVC245 octal bus transceiver	$1.50	3	
10	0.1uF leaded ceramic capacitor	$0	1	
11	22 or 24 AWG hookup wire	$0	N.A.	
12	A 7-pin mini-DIN cable	$6.95	1	
13	Optional: 4-40 x 5/8" or M3 x 16 mm standoff to support prototyping board if installing above Raspberry Pi	19.99	1	
14	Basket	18.99		
	Load Sensor			
	Load Amplifier (guide: https://learn.sparkfun.com/tutorials/load-cell-amplifier-hx711-breakout-hookup-guide)			
	to approve: basket, load sensor, load amplifier, rpi 3			

Bill of Materials



```python
from di_sensors.easy_line_follower import EasyLineFollower
from time import time, sleep
from  pycreate2 import Create2

# Create a Create2.
port = "/dev/serial"  # where is your serial port?
bot = Create2(port)

# Start the Create 2
bot.start()

bot.full()
setpoint = 0.5
integralArea = 0.0
previousError = 0.0
motorBaseSpeed = 300

# PID control gains to be tuned
Kp = 0.0
Ki = 0.0
Kd = 0.0

lf = EasyLineFollower()

while True:
    start = time()
    pos, out_of_line = lf.read_sensors('weighted-avg')
    # pos is the computed sensor value in range 0.0 - 1.0
    error = pos - setpoint

    integralArea += error
    correction = Kp * error + Ki * integralArea + Kd * (previousError - error)
    previousError = error

    motorA = motorBaseSpeed + correction
    motorB = motorBaseSpeed - correction

    bot.drive_direct(motorA, motorB)

```

Tentative Line Following Code