MTE 100 / MTE 121 Course Project: RenoBot

Group 4 – 22

Vishesh Garg – 21001699

Victor Kalenda – 21017829

Vishan Muralikaran – 20966382

Connor Switzer – 21029354

December 6th, 2022

Summary

The RenoBot was designed to aid humans in construction and renovation. Its primary function is to move around a space and calculate its area and perimeter. Additionally, the RenoBot will return information regarding the amount of material needed for that space. Finally, the RenoBot can also measure a specific angle, and then draw it out on paper. These functions can all be called by the main menu, which is a navigable interface made for the user to interact with the robot. This main menu is how the robot is started and shut down, and once a function is called the robot moves autonomously until it has completed a task, and returns to the main menu

The criteria and constraints of this project focus on both the mechanical and software side of the robot. They drove the design of the robot by setting a requirement on size, degree of accuracy, and autonomy.

The mechanical design of the robot was built entirely from Lego Mindstorms EV3 kit parts. The main sub-assemblies involved in the RenoBot's design are the chassis, the motor drive mechanism, the touch sensor/bumper, and the pencil actuator. Major considerations that were addressed while designing the robot were the EV3 brick placement, the ultrasonic sensor placement, and the IR sensor placement. The overall assembly of the RenoBot was executed with its dimensions and functionality in mind. As a result, RenoBot was built with a considerable degree of excellence in its functionality, user interface, and overall design.

The main menu contains all the functions of the RenoBot. The menu allows the user to cycle through all the function options. It uses functions such as getButtonValue and mainMenuScreen to make it possible to navigate the different areas of the menu.

The main functions of the robot are called through the user interface, they call a number of subfunctions responsible for basic functionality. For example, subfunctions such as drive_dist, and rotate were called numerous times in all main functions. Careful consideration was given to the timing of functions and the logic behind all calculations.

The Renobot was verified based on the stated criteria and constraints. Based on these, it performed excellently in the demo. Apart from one small autonomy issue, which was a result of provided hardware, everything worked to plan.

Table of Contents

Summary ii
Table of Contents iii
Table of Figures v
Introduction 1
Scope 1
Main Functionality 1
Measurements and Detection 1
Environmental Interactions 2
Task Completion 2
Changes in Scope 2
Constraints and Criteria 3
Mechanical Design and Implementation 4
Chassis Design 4
Motor Drive Design 5
Touch Sensor/Bumper Attachments 6
Pencil Actuating Design 7
EV3 Brick Placement 7
Ultrasonic Sensor Placement 8
Infrared Sensor Placement 8
Overall Assembly 9
Software Design and Implementation 10
Main Menu 10
Area and Perimeter Menu 10
Materials Menu 11
Menu Navigation Functions 11
Materials Functions 11
Main Functions 11
Navigation and Coordinate Calculation 12
Remote Control and Open Area Calculation 16
Draw_Corner Function 17
Tests 19

Menu Screen 19
Function Values 19
Tradeoffs 20
Significant challenges and Major Design Decisions 20
Selecting Menu Options 20
Gyroscopic and Coordinate values 20
Simplicity 21
Timing Issues 21
Multiplexor Troubleshooting 22
Multiplexor sensor choices 22
Verification 23
Project Plan 23
Workload Distribution 24
Programming 24
Main Menu Code 24
Main Code 25
Conclusions 25
Recommendations 25
Mechanical design recommendations 25
Pencil Mechanism 25
Weight Distribution 25
Bumpers 26
Software recommendations 26
Pencil calibration 26
Concrete 26
References 27
Appendices 28

Table of Figures

Figure 1 RenoBot Base 5
Figure 2: Main Menu flow chart 10
Figure 3: Area_Perimeter flow chart 12
Figure 4: correct_path flow chart 13
Figure 5: coordinate_process flow chart 14
Figure 6: Wall calculations 15
Figure 7: Obtuse angle calculations 15
Figure 8: Acute angle calculations 16
Figure : Draw_Corner flow chart 17
Figure : map_corner flow chart 18
Figure : trace_corner flow chart 19

Introduction

Construction as well as home renovations are filled with tedious tasks which do not need to be completed by a human. In the eyes of CV3 (The RenoBot group's title, derived from the first letters of their names) this is a problem. Humans should not have to spend their precious time on tasks not worth their while, and for that reason the RenoBot was created. The purpose of the RenoBot is to both save time and energy of someone who is involved in the process of constructing or renovating a space. It does so by working autonomously to collect data, and then returns calculated values regarding the collected data. This is meant to be convenient for a human, as they can start the RenoBot, continue with another task, and then come back when they are ready to collect the information they need.

Scope

Initially the RenoBot was meant to move around a space, detecting and marking studs in the walls it drove past. CV3 quickly pivoted from this idea and began exploring how the robot might move around a room, and what information it can gather as it does this.

Main Functionality

The completed RenoBot's main functionality is to calculate the area and perimeter of either a closed or enclosed space. After collecting this data, it can return information about construction materials. An example of this is how much paint one might need to paint the measured space. Additionally, the RenoBot can measure out an angle, and then retrace that angle in a separate location, which has a practical use in flooring. This can provide a guideline one might use to cut to obtain appropriately shaped pieces of carpet or hardwood. All the functions of the RenoBot can be called from the main menu, which allows the user to scroll through the different options and select a function.

Measurements and Detection

To gather the information and move about the space being measured, the RenoBot uses two touch sensors, motor encoder values, an ultrasonic sensor and in an unenclosed space, an IR sensor. The touch sensors work in conjunction with the ultrasonic sensor to allow the RenoBot to move though an unknown space. The touch sensors reside on the front of the robot and allow the RenoBot to detect walls in its path. Once a wall is detected the RenoBot knows it must make a turn. The type of turn it must make is dependent on which touch sensor was hit first, the right or the left. In the situation where the RenoBot must turn but there is no wall in front of it, for example a reflex angle, the RenoBot will use the ultrasonic sensor. By reading the distance between the RenoBot and the wall, the ultrasonic sensor and a coded PID controller work in part to determine the values each driving motor should be set at. This allows the RenoBot to follow along walls including curves, and reflex angles where there would be nothing for the

touch sensors to react to. The IR sensor is used to drive the RenoBot via a remote, and therefore the information above does not apply to the unenclosed space function of the robot.

As the RenoBot is moving around the space, it is plotting points around the perimeter which are used for calculating the area and perimeter. This is covered more in depth in the “navigation and coordinate calculation” section of this report.

Environmental Interactions

As previously mentioned, the RenoBot moves around a space and takes in information from the environment via the ultrasonic and touch sensors. This information can then be communicated by either outputting calculated information to the LCD screen, or in the case of the draw angle function, being drawn on paper. The robot can do these things using three different motors. The first and second motor are used in separate drive trains connected to the wheels which enable the robot to move. The third motor is used in a pencil lowering mechanism which allows the robot to recreate a measured angle, communicating to a human the specific cut they would need for a specific angle.

Task Completion

An important feature of the RenoBot is autonomous operation. For this reason, the RenoBot was designed to stop on its own when it has determined it is back at its starting location. As it moves the RenoBot is constantly plotting points. Once it realizes it has arrived back at the origin point, it knows to stop driving and display the area and wait for human interaction. Once a human has pressed a button the robot displays the main menu and from this point it can be shut down by pressing the shutdown button. This is the standard procedure to stop the program. All functions once completed will return to the main menu, allowing the user to shut down.

Changes in Scope

CV3 had many ideas when beginning the final project, including the stud finder mentioned above. Additionally, a rotating IR sensor was considered, and would have been completed if not for the constraints of the IR remote, and an option to paint the floor was also considered, but not featured in the RenoBot due to lack of conventional use. Many ideas were considered, however once the mechanical design for the RenoBot was built there were very few changes needing to be made, as it worked as intended.

Constraints and Criteria

Although the RenoBot performed exactly as it should have, and in testing, returned values approximately equal to the calculated areas. This was only due to careful consideration of specific criteria, constraints, and formatting our robot to excel in them.

The first criterion applied to the project was regarding the space which the Robot had to navigate. The space had to be smooth, continuous and a minimum height of 25 centimeters. This constraint ensures that the space being measured would be readable by the robot. Due to the nature of the PID controller and the ultrasonic sensor, even slight deviations in the continuity or flatness of the wall would cause the RenoBot to turn inwards toward the wall which would impact the location of the points being plotted and skew the area and perimeter calculations. In most cases the RenoBot would still be able to continue around the space, however, would not return values accurate to the constraint. This constraint, as determined during the testing stages was to be accurate to 0.75 square meters of area, and 0.5 meters when calculating these measurements of a space. (Note: This was a change made by CV3 during the testing stage, before testing it was projected to be accurate to 0.5 square meters)

The second constraint the robot had to follow was regarding the physical design, rather than the environment. This constraint required it to be as compact as possible in both the length and width direction. For this reason, the design of the robot is longest in the vertical direction, and all parts appear stacked on top of each other. This was a constraint that had to be followed in the attempt of making the robot as accurate as possible, which worked in two ways. The length of the robot had to be significantly shorter than the width, as when the robot makes a turn the back-end swings out. If the back end of the robot was too long, there would be interference between it and the wall it is following. Additionally, the width of the robot had to be as small as possible, as when the RenoBot is moving about a space, it can only measure what it can fit into. If there exists in a space, a subspace so small that the RenoBot cannot fit inside, points cannot be plotted, and therefore it will not be included in the overall area. In conclusion the mechanical design of the robot was also designed based off a constraint.

Finally, A piece of criterion which was not a leading factor to the physical design of the robot was the fact that the RenoBot had to be as autonomous as possible. This requirement was more impactful on the design of the code behind the movement and calculation that the robot made. The RenoBot had to know exactly what to do in any possible situation it may encounter. The code had to be made for acute, obtuse, right, and reflex angles, as well as curves. Due to there being no human interaction with the robot, this required excessive amounts of code just for navigation.

Mechanical Design and Implementation

The RenoBot used three motors in design; two large motors to drive the robot and a medium motor to retract/deploy the pencil for drawing. In addition to this, another medium motor was mounted to the robot with the IR Receiver. While this motor was not activated in the code, there was software written for a PID controller that would turn the IR receiver to face the remote. This software was not implemented due to limitations from the IR Remote's signal. However, since the ability to manually turn the IR Receiver was still desirable, the fourth motor stayed on the robot's body. Please refer to "Tradeoffs" for more information.

The robot used five sensors to navigate the room: two touch sensors, an ultrasonic sensor, a gyroscopic sensor, and an IR sensor. The touch sensors were mounted towards the front of the robot on either side to be activated by walls that the robot would encounter. The gyroscope was mounted in the chassis of the RenoBot, between the two large motors that drove its motion. The ultrasonic sensor was mounted on the top of the robot, facing the wall at an angle. Lastly, the IR receiver was mounted on the fourth motor towards the left side of the robot, and the motor mount gave the user the ability to turn the receiver in any direction they choose.

RenoBot was entirely built out of LEGO Mindstorms robotics parts, without any external 3D printing or laser cutting. When discussing their design, CV3 wanted to embrace the challenge of working with a limited set of parts to test their mechanical design skills. While many complex designs can be easily 3D printed, a multifaceted robot that is built entirely from elementary Lego robotics is much harder to pursue. Thus, the RenoBot's mechanical design was a direct testament to CV3's emphasis on mechanical creativity.

Chassis Design

The RenoBot's chassis design was the first sub-assembly of the robot to be completed. The drive train was selected to be powered by two large motors, using two standard Lego wheels and a third marble omni-wheel to maintain contact with the ground. The omni-wheel was placed at the back of the robot so that the two motors could perform a point turn (a turn with the axis of rotation being centered in and perpendicular to the robot's top plane).

To connect the two motors, Lego frame pieces were used both above and below the motor bodies. To retain consistency and stability in the design, there was complete symmetry in the chassis. This ensured that the margin of error that occurred due to chassis misalignment was minimized. Lego frames were heavily reinforced underneath the motors with other beams to ensure that the chassis did not bend under the stress of the robot's weight. To ensure that the RenoBot could drive forward in a straight line, reinforcing the chassis and ensuring it did not compress in any dimension was critical.

Along with the attention to detail in terms of stress analysis and weight distribution, the chassis was built with a macro perspective; there were components of the RenoBot that needed to interact with the chassis. For example, the gyroscope sensor was placed directly in between the large motor bodies to ensure it is in the center of the robot in a secure location. By confining the gyroscope sensor to a confined space, the margin of error caused by the gyro shaking during motion was minimized.

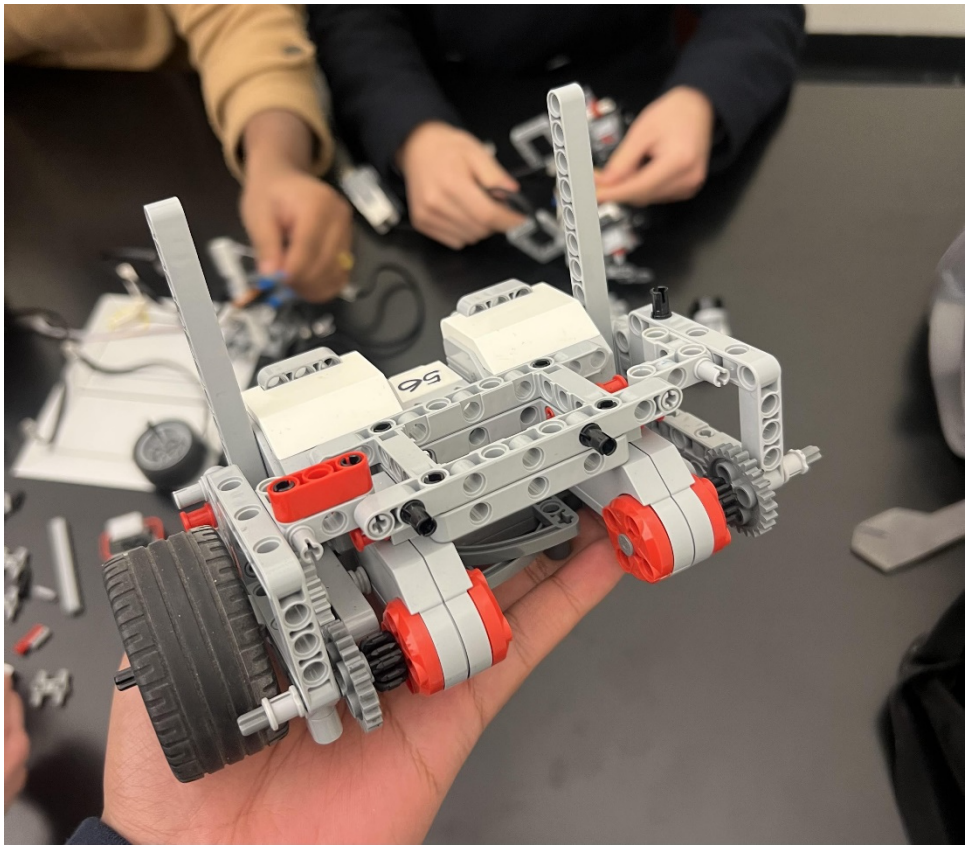

Figure 1 RenoBot Base

Furthermore, the RenoBot needed to be able to redraw curves and angles with a pencil with a high degree of accuracy. Due to the three-wheel design, the axis of rotation for the RenoBot was in the middle of the space between the two powered wheels. This was a very specific location, and it was critical for the pencil to be actuated in this precise spot. Thus, the chassis was constructed with frame pieces to ensure a space in the middle in the appropriate location. This space needed to be reinforced near the floor so that the pencil did not slip or move while dragging along the surface. As a result, the chassis was built with a hollow center to make room for subsequent sub-assemblies. These design considerations proved critical when completing the rest of RenoBot's body.

Motor Drive Design

To move the RenoBot around a room, a great deal of consideration was put into the motor drive design. While only two wheels needed to be powered, there were mechanical methods that were applied to ensure a higher level of accuracy in motion.

First, the axles for the wheels needed to be cantilevered due to the motors' design. As such, multiple Lego frames and beams were used to secure the axle before the wheel was fitted on. Due to sizing constraints in terms of the width of the robot, the shaft couldn't be supported on both sides of the wheel. Therefore, bushings and half-bushings were used between the support beams to ensure the shaft has maximum support on one end. While the cantilevered design caused a slight bend in the chassis when

the RenoBot was fully built, this was a margin of error that was unavoidable due to the size restrictions that CV3 had determined.

To accurately calculate the distance that the RenoBot travelled, the motor encoder counts were used on motor A. However, since one encoder count represents one degree of rotation of the robot, a gear ratio was implemented in the RenoBot to increase the clarity of the encoder counts. The gear ratio used was 5:3, indicating that five turns of the motor translated to three turns of the wheel. This increased the motor count clarity, as one motor encoder count translated to 3/5 of a degree of rotation of the wheel.

This gear ratio brought a host of problems; the direction of the motors was flipped. This caused more logical errors in the code than CV3 expected. To elaborate, the direction of the motors being reversed meant that every drive function that was written needed to be powered in the opposite direction. This led to a complication with the motor encoder counts, as the more distance the robot travelled, the more negative the encoder count values became. This caused mathematical errors in the distance calculations that CV3 needed to dedicate time to fix.

Touch Sensor/Bumper Attachments

To navigate the room, two bumpers were attached to the front of the RenoBot on either side. These bumpers had several iterations in terms of their mechanical designs, as they needed to perform consistently and reliably for the RenoBot to function appropriately.

The main function of the left bumper was to be activated during an acute angle turn, as the wall would hit the left bumper as the RenoBot drove forwards. The right bumper was a little more versatile, as it was used for both obtuse angle turns and concave curves. However, both bumpers needed to be consistently activated as soon as the robot contacted the wall. This led to the first major mechanical consideration; the bumper must be designed so that any movement in its surface profile causes translation in the touch sensor's button.

This was quite a challenge due to the EV3 touch sensor's button tolerance. The low quality of the fit in the touch sensor led to a lot of unnecessary peripheral movement in the orange button. This meant that if the bumper plate was activated at an angle, the orange button would turn instead of being pushed inwards. This was a problem that couldn't be addressed directly due to hardware limitations. As such, several iterations of mechanical designs for the bumpers were considered.

A simple plate that connected to the orange touch button was impractical, as the plate would simply turn in place instead of relaying any motion to the sensor. Thus, CV3 concluded after quite a few tests that a bumper that rests on an axle and rotates on a fixed axis would work best, as it would be limited to only one direction of motion. This worked well, as any movement in the bumper's front profile would only be able to relay movement in the axis of the touch sensor's actuator.

Due to the Lego parts available, the bumper would sometimes be unable to achieve the right angle of contact with the wall. This led to the second major mechanical consideration for the bumper; using a planar bearing (turntable) to have both sensors rotate in the z-axis. This led to a considerable improvement in bumper consistency. When either of the bumpers contacted the wall, they would rotate on their axis to ensure the motion is properly translated to the touch sensor.

The combination of a bumper mounted on an axis and the rotary motion of the touch sensors created a dynamic sensor arrangement on the front of the RenoBot. This was a critical aspect of the robot's ability to navigate a room autonomously. Both bumpers increased in size as these mechanical designs were implemented. Despite this, the RenoBot's width needed to be controlled and the touch sensors needed to be mounted directly above the wheels to ensure they do not get activated at the wrong times. This led to the touch sensors being mounted higher than expected; there was a tradeoff between height and width, and minimizing width was the priority. Using several double connectors and angular connectors, the bumpers were mounted in the ideal location.

Pencil Actuating Design

Since one of the functions of the RenoBot was to retrace angles that it had mapped, it needed to be able to deploy and retract a pencil from its body. This was a challenging mechanical task, as the pencil needed to maintain a certain amount of pressure against the ground to be able to draw.

The pencil's position was specified in the drive chassis description. However, mounting the pencil and actuating it in the middle of the RenoBot's width was a challenge. To actuate the pencil, a medium motor was used with a rack and pinion mechanism. The pinion was attached to the medium motor that was mounted above the chassis, and the rack was fastened to the pencil with rubber bands. Lego beams were used around the pencil to guide it and maintain its perpendicularity with the ground.

Since the medium motor was a long rectangular prism, space needed to be dedicated to it in the main body. This led to raising the bumper touch sensors even further, as the medium motor needed to be closer to the chassis to touch the pencil. This tradeoff made sure that the robot did not get any wider or longer as a result of the pencil mechanism, but it became taller.

EV3 Brick Placement

Since the EV3 brick represented a large volume and mass of the RenoBot, a considerable amount of thought was put into its placement and orientations. There were several factors to consider when deciding its location: weight distribution, robot dimensions, accessibility for the user, motor/sensor wire accessibility, charging port accessibility, and structural integrity. Due to the dimension constraints and limited Lego pieces available, it was necessary to make tradeoffs between certain factors that CV3 considered.

Since there was a large amount of weight near the front of the robot, the EV3 brick needed to be mounted near the back. Weight distribution was a primary concern, as the RenoBot could not, under any circumstances, fall over while driving autonomously. Knowing that the brick needed to be mounted at the back, having it "lay down" on the back of the chassis would have increased either the length or the width of the robot. CV3's second focus was to maintain the compact length and width that had been a priority throughout the design process. As such, the EV3 needed to be placed in an "upright" orientation so that the width and length of the robot would be maintained. As a result, the height of the robot increased.

While the motor ports were highly accessible at the top, the sensor ports were not easily accessible. While the EV3 was mounted to the RenoBot, the charging port was rendered completely inaccessible. However, the upright position provided a semi-accessible user interface with the screen. It made the robot easy to hold to navigate the main menu, but it was hard to observe the screen while it was driving during the testing process. These tradeoffs were necessary to complete the mechanical design.

One last tradeoff was made to minimize the impact of the limited access to sensor and charging ports; the EV3 brick was only mounted to the frame of the robot with two double connectors. This meant that removing and reattaching the brick to the frame of the robot was very simple. The downside to this design was the structural integrity; without the sensor wires pushing the EV3 brick up against the drive motors, the EV3 brick would hang off the frame almost as if it was going to fall off. With the support of the wires, however, the EV3 brick was firmly secured to the frame, but still easily removeable for charging. This was a necessary part of the mechanical design, and while it was unavoidable, CV3's use of the double connectors minimized its impact on the robot's function/use.

Ultrasonic Sensor Placement

The primary way for the robot to follow a wall was its ultrasonic sensor. Using a PID controller, the ultrasonic sensor of the robot constantly measured the distance to the wall and adjusted the motor speeds accordingly.

The ultrasonic sensor needed to be placed facing the right side of the RenoBot, as it was designed to navigate a room in the counterclockwise direction. The point at which the ultrasonic sensor's "line of sight" met the wall needed to be a few centimeters in front of the axle of the robot's drive wheels. This was important so that the RenoBot could "look ahead" and determine a reflex angle or convex curve, and so that the motors had enough time to respond accordingly.

However, when the ultrasonic sensor was mounted so that it was "looking" perpendicular to the wall, the robot wouldn't be able to tell if it needed to drive closer to the wall or away from it based on the angle of its motion. This created an unsolvable software problem with the current sensor configuration, and as such, the placement of the sensor needed to be adjusted.

Ultimately, the ultrasonic sensor used a double connector and a double angular beam to face the wall at an angle. This meant that the sensor needed to be placed closer to the EV3 brick so that it would "look" at the wall in the right spot in reference to the location of the wheel shaft. As a result, CV3 was forced to mount the ultrasonic sensor slightly elevated above the EV3 for it to fit in the right location. This further increased the RenoBot's height, making it quite a tall robot. However, this tradeoff was critical to the robot following walls and convex curves accurately.

Infrared Sensor Placement

As mentioned previously, the infrared sensor was mounted on a medium motor, but never rotated in the main code of the robot. As a result, the orientation of the IR sensor wasn't a concern. However, the IR sensor needed to be elevated above the body of the robot so that none of the robot's wires interfered

with the IR light. This meant that the medium motor needed to be mounted facing upwards, and the IR sensor became the tallest object on the RenoBot.

The medium motor was fastened in a location that didn't interfere with any of the other functions of the robot. It used a combination of several angular connectors to connect to the frame of the robot so that the EV3 was still easily removeable.

Overall Assembly

One of the most prominent features of the RenoBot is its integration between sub-assemblies. While the many functions of the robot led to a variety of different considerations, the RenoBot's mechanical design exhibits a strong sense of coherence. The way the touch sensors are designed to accommodate for the pencil mechanism, and the way the wires are organized so that they don't interfere with any motion is all a product of CV3's mechanical design considerations. The overall assembly of the RenoBot was executed with its dimensions and functionality in mind. As a result, the RenoBot was built with a considerable degree of excellence in its functionality, user interface, and overall design.

Software Design and Implementation

Main Menu

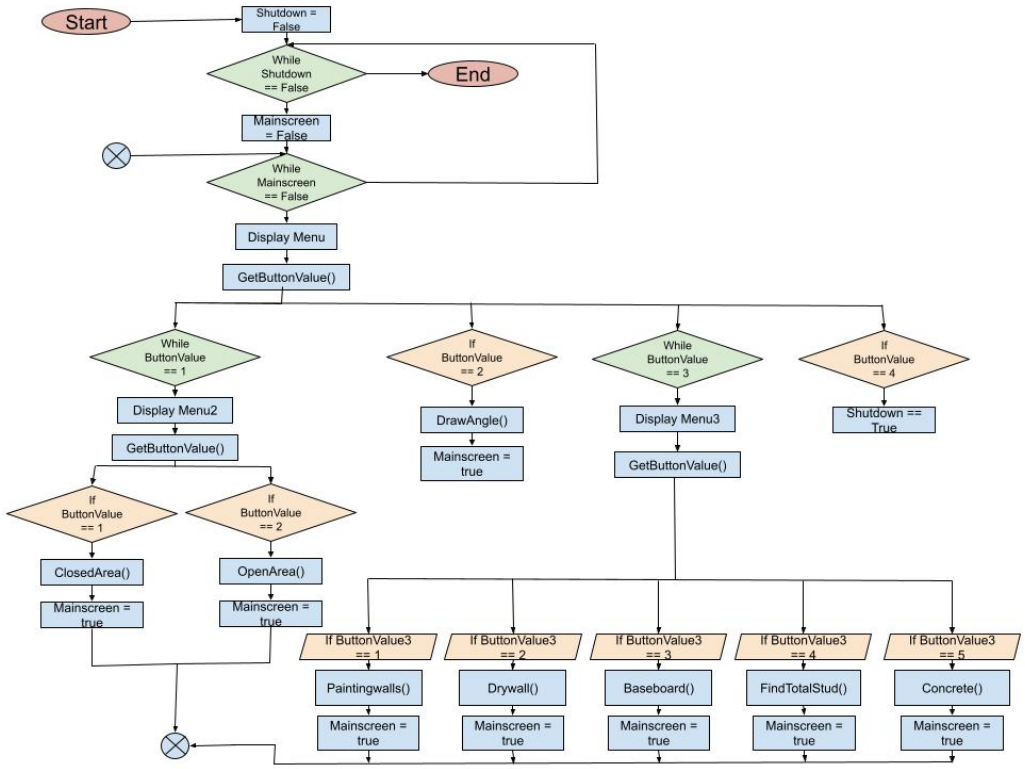

Figure 2: Main Menu flow chart

The functions for the RenoBot are accessed through the main menu. The main menu consists of four options: area and perimeter, angle calculation, materials, and shutdown. The arrow keys on the RenoBot are used to navigate through the menu screens. The user was sent back to the previous screen when the left button was pressed. The up and down buttons navigate through the options for the given menu screen. When the enter or right buttons are pressed, the menu will proceed with the selected option. Within the main menu, there were two sub-menus which allowed the user to access the area and perimeter functions along with the material functions.

Area and Perimeter Menu

The area and perimeter menu screen contained 2 options, closed area and open area as illustrated in “Figure 2”. When either of these options are selected, the corresponding code will run. Once the function has been completed, the user returns to the main menu where they can choose another function or shut down the program.

Materials Menu

The RenoBot must contain values for the area and perimeter to access the materials screen. The materials screen contained 5 options, painting, drywall, baseboard, total studs, and concrete as illustrated in “Figure 2”. Each option within the materials screen requires different area and perimeter values depending on their application. The first 4 options require the RenoBot to contain closed area and perimeter values while concrete requires open area and perimeter values as illustrated in “Appendices”.

Menu Navigation Functions

The functions required to navigate the menu screen are getHeight, getButtonValue, and mainMenuScreen. Within the main menu, there are 2 other sub-menus that use the getButtonValue function to display the numbers for the list of options for the corresponding menu. The mainMenuScreen allows the user to return to the original menu while in the sub-menus. Finally, getHeight allows the user to input the height of a room, allowing the user to navigate through some of the material functions.

Materials Functions

The functions required for the materials are surfaceAreaOfWalls, paintingWalls, drywall, baseboard, findTotalStud, and concrete. Refer to “Appendices” for information regarding these functions.

Main Functions

Every sub-function developed was designed with the goal of helping the robot accomplish simple tasks as efficiently as possible. Both trivial and non-trivial functions such as drive, drive_dist, drive_to_wall, remote_drive, rotate, rotate_gyro, turn_left, reset_variables, calibrate_gyro, wait_read, reset_IR, set_pencil, configure_sensors, and mission_possible will not be fully explained as they would make this document unreadable. Please refer to “Appendices” for more information on each specific function. Each of the functions listed are called readily throughout the basic functioning of the robot, making up the framework of the RenoBot’s main functions.

3 structures were developed to simplify the passing of data by reference into each sub-function. These structs were called coordinate, past, and present. Most of these structs are made up of floats representing relevant information for all main functions regardless of their needs.

Navigation and Coordinate Calculation

Navigation around the space was completed autonomously by the robot in the main function Area_Perimeter. Please refer to “Figure 3”, which showcases how the functions were restricted to their specified domains without conflicting with other tasks.

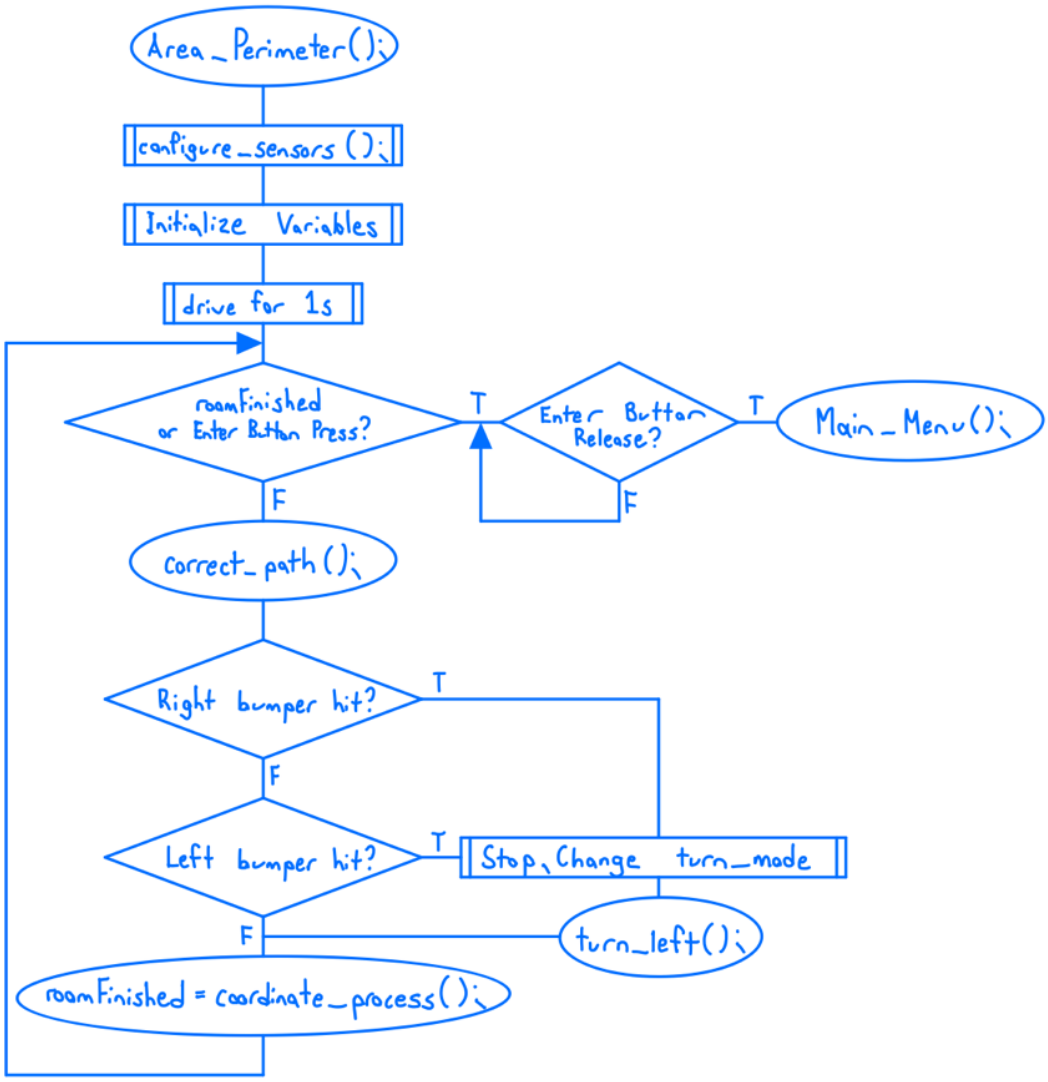

Figure 3: Area_Perimeter flow chart

The function correct_path held the robot at a specified distance from the wall. This distance was defined by the constant float PID_TARGET. The function is a PID controller which modulated the motor powers of the left and right motors depending on the ultrasonic reading. This function was called every 0.005 seconds as defined by the constant integer MUX_TOL. Please refer to "Multiplexor Troubleshooting" for more transparency on this value. The following "Figure 4" showcases the basic operation of the PID controller.

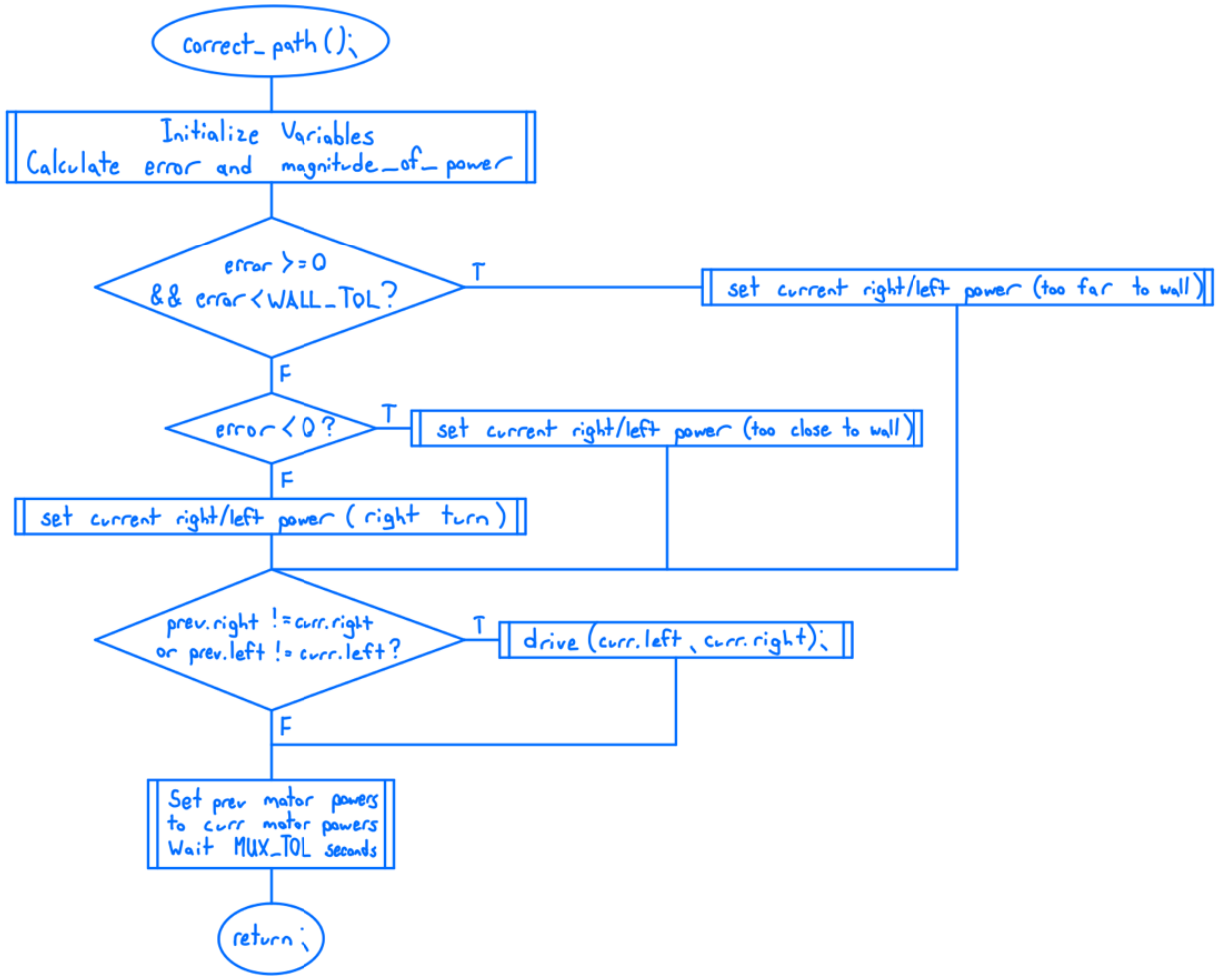

Figure 4: correct_path flow chart

To ensure consistency between timer-based intervals, an if statement was developed which would check if the index was an integer smaller than the timer. If it was, then the integer would be set equal to the timer, plus whatever time gap constant was specified. This ensured that if for whatever reason the timer happened to be much larger than the integer, the function inside the if statement would only be executed once, than again after the specified time gap. With this, functions within the if statement would not fail due to overloading. Please refer to "Timing Issues" for more clarity on this topic.

The function coordinate_process, as shown in “Figure 5”, is the only function used for calculating data points. The function was called every 0.5 seconds with the use of an if statement as stated previously.

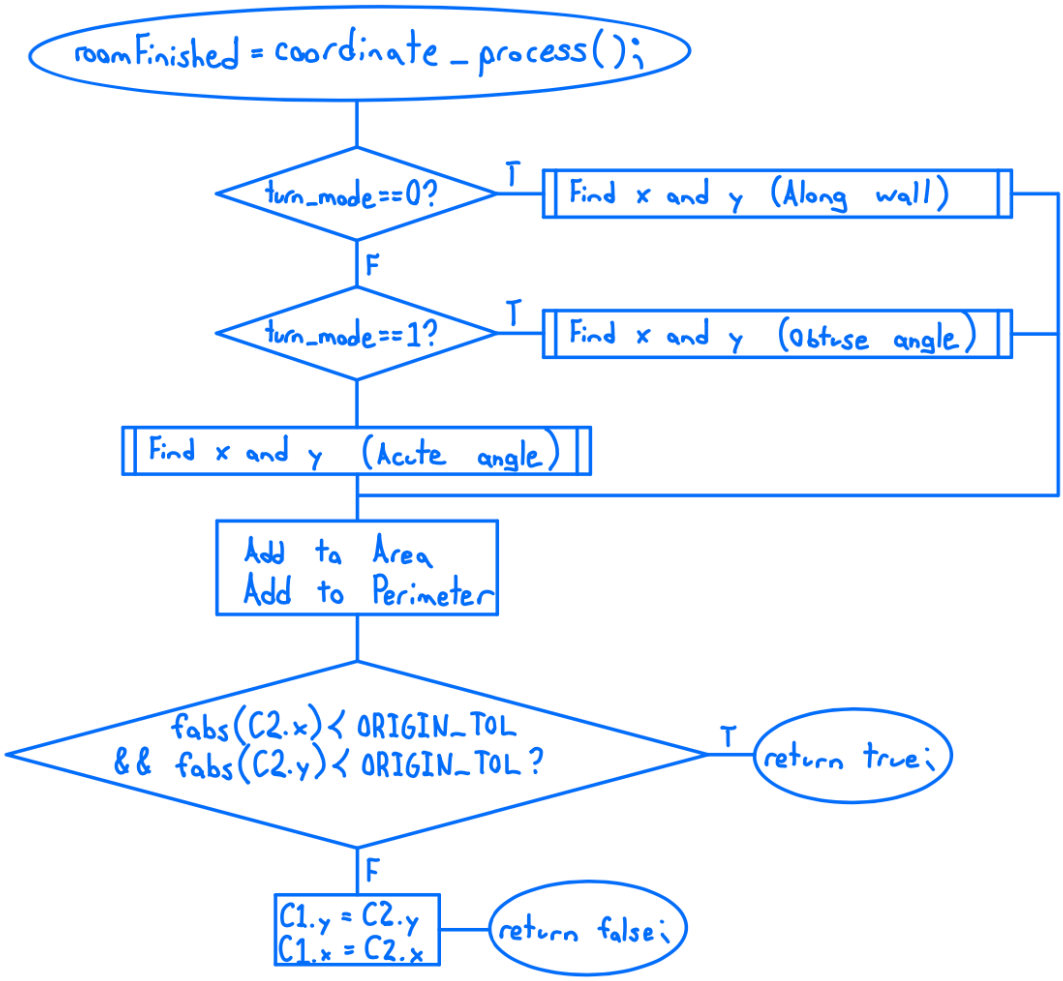

Figure 5: coordinate_process flow chart

The math behind determining the coordinates is dependent on ensuring that the gyroscope and motor encoders do not get reset at any point throughout the duration of its main function. This was done to maintain a consistent margin of error between calculations, improving the overall precision of the robot.

The 3 instances when coordinates were calculated are as follows…

- While alongside a wall

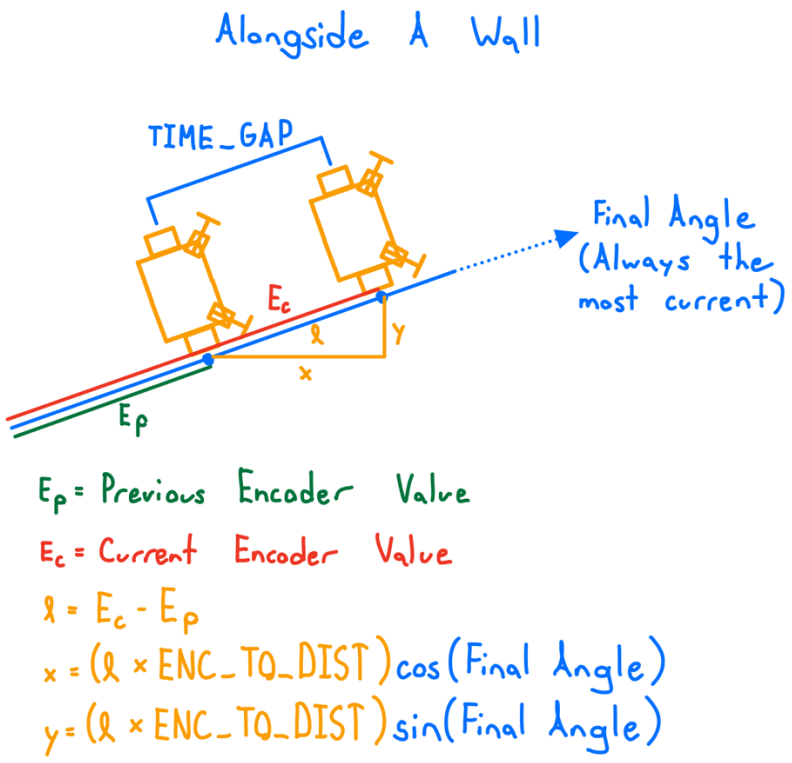

Figure 6: Wall calculations

- During an obtuse angle turn

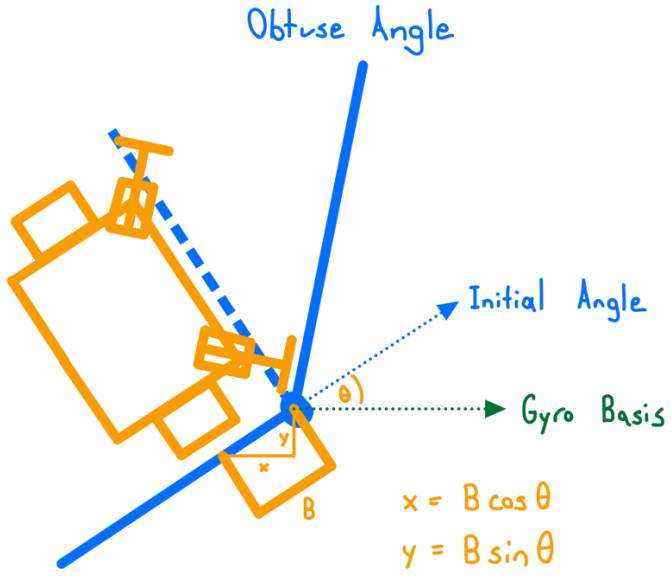

Figure 7: Obtuse angle calculations

- During an acute angle turn

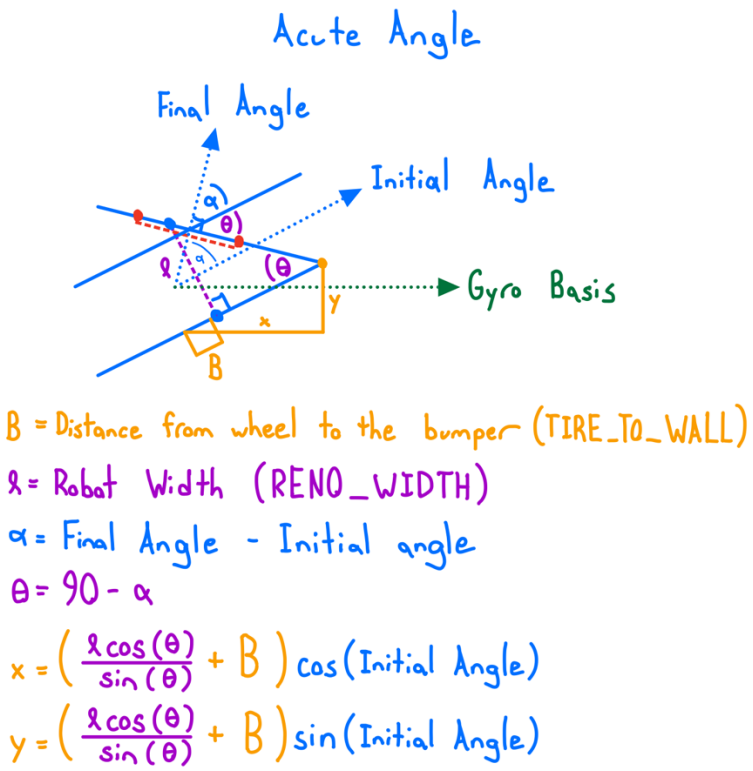

Figure 8: Acute angle calculations

Remote Control and Open Area Calculation

In the function Remote_Area_Perimeter, the user controls the robot using the IR Sensor while the robot travelled around a given space. The robot constantly corrects its current coordinate position relative to where it started while also measuring the current area and perimeter according to the shoelace algorithm [4] specified in the coordinate_process function. For more information on the coordinate_process function please refer to “Navigation and Coordinate Calculation”.

The function remote_drive is called in a loop, collecting data from the IR receiver, and translating that data into the specific commands for the RenoBot’s motors. Please refer to “Appendices” for more information regarding the specifics of remote_drive.

Like Area_Perimeter, Remote_Area_Perimeter constantly checks for whether it was within the tolerance of coordinates around the origin. The function can also be exited by simply pressing the enter button.

Draw_Corner Function

In the function Draw_Corner, the user places the robot alongside a wall pointing towards a corner the user would like to draw.

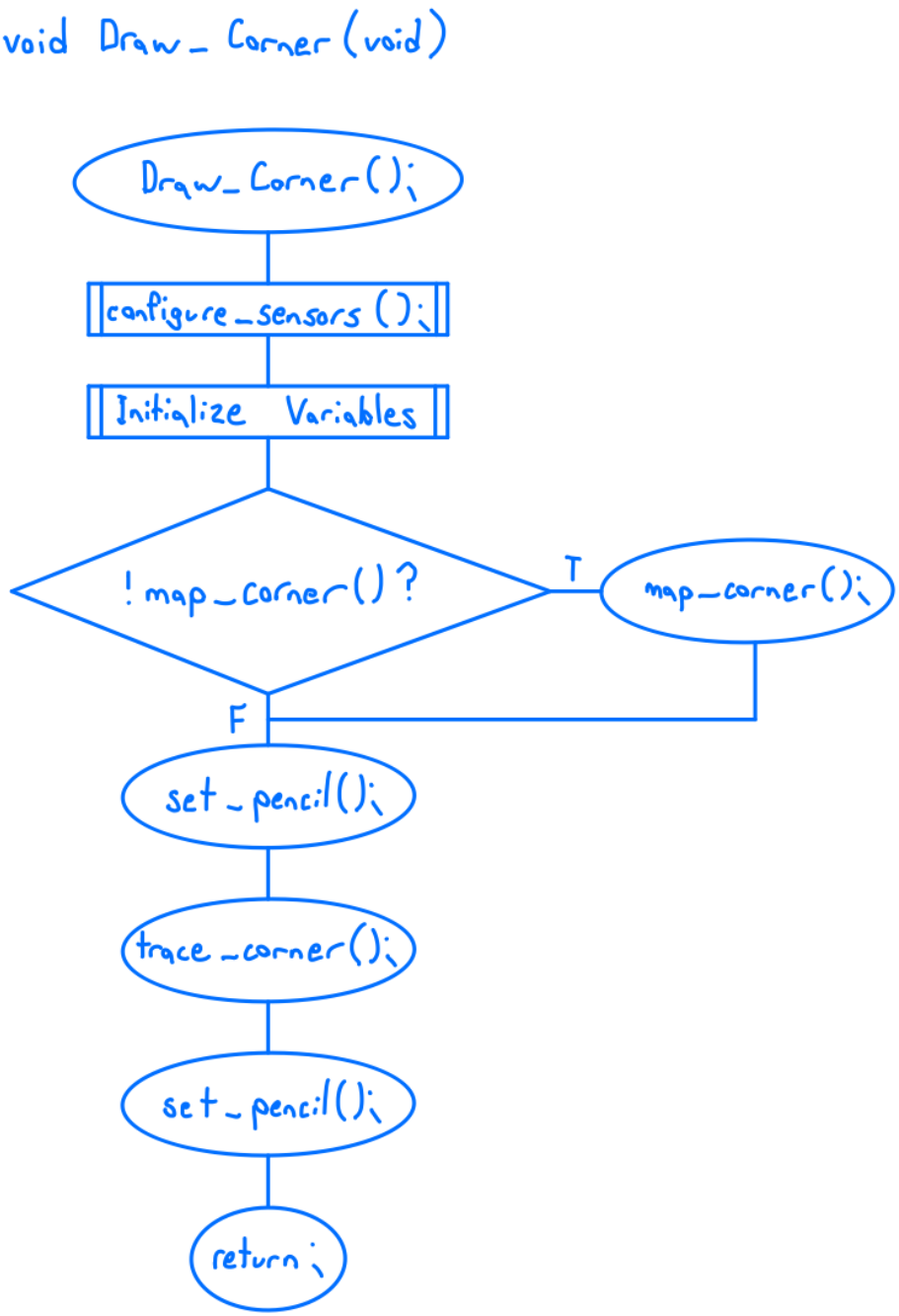

Figure 9: Draw_Corner flow chart

The robot then maps the corner using the map_corner function, which logs coordinates into a coordinate array named data. The wait_read function is then used to wait for the user's input.

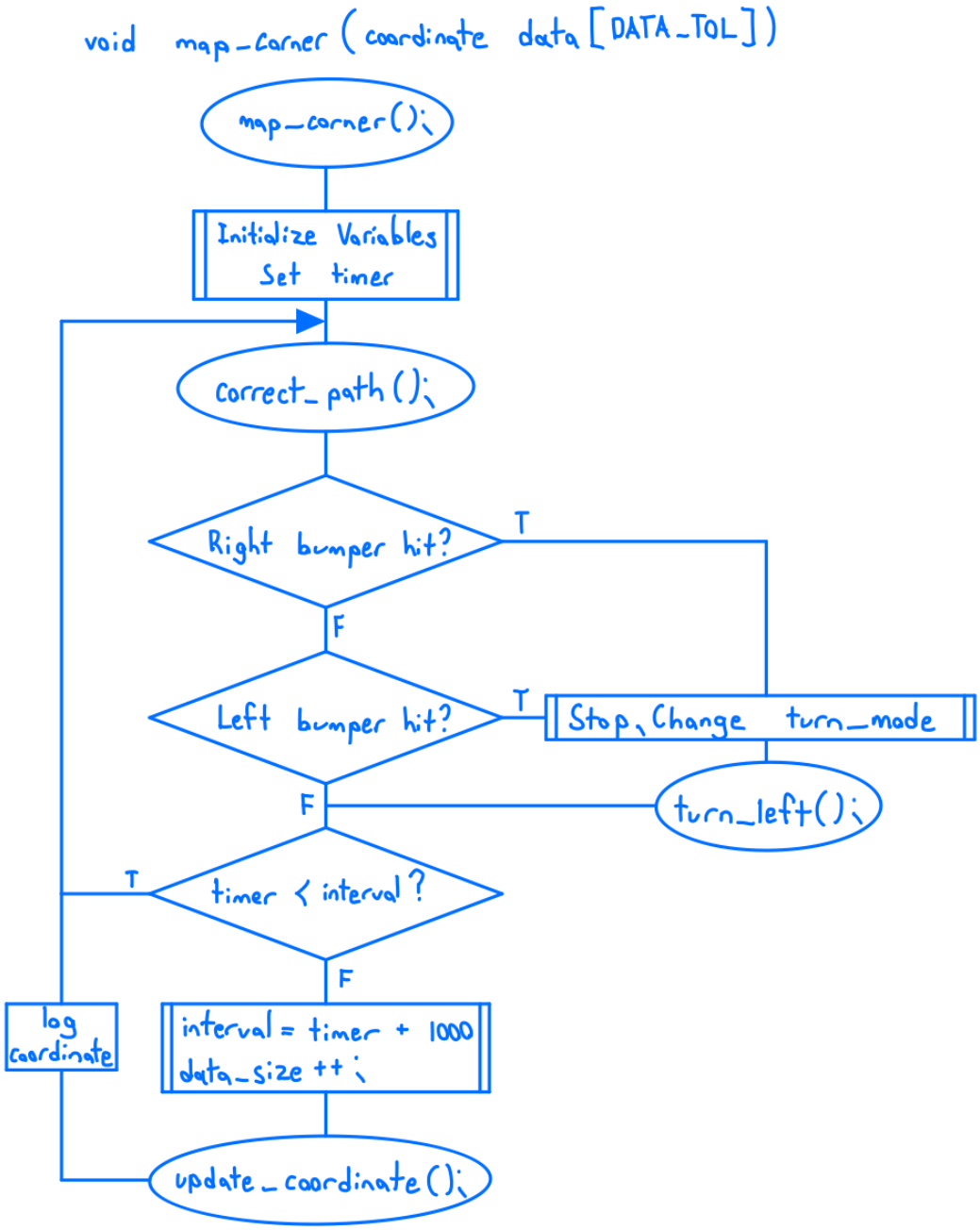

Figure 10: map_corner flow chart

Once the user presses the enter button, the robot calls set_pencil, to lower the pencil, then trace_corner, to replicate the angle based on the coordinates saved from map_corner.

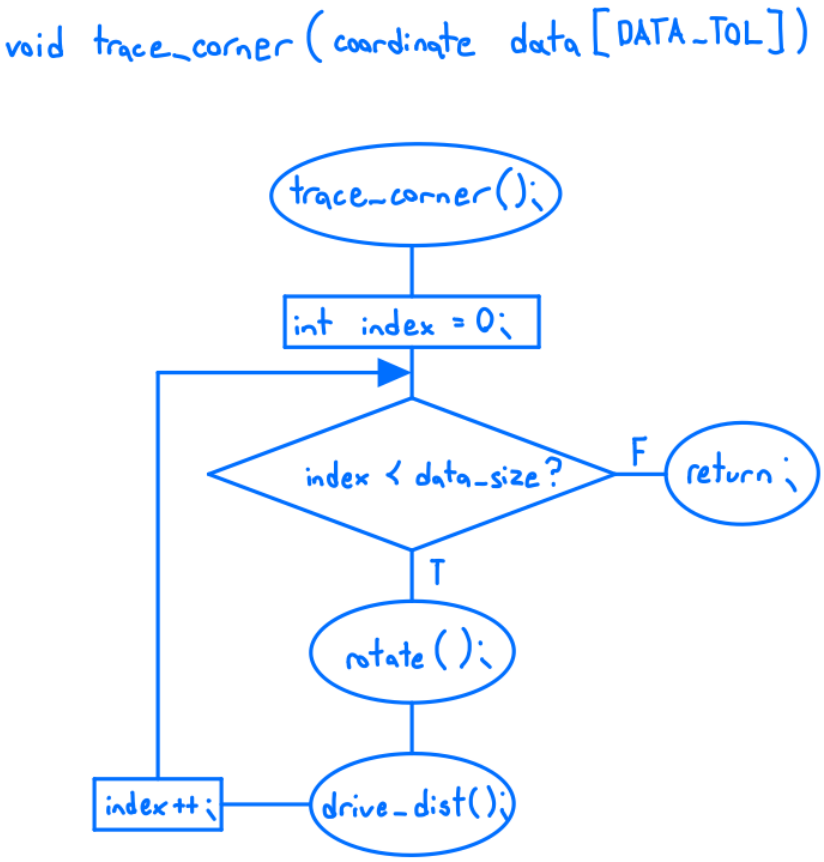

Figure 11: trace_corner flow chart

Once finished, the robot calls set_pencil to raise the pencil and return to the main menu.

Tests

Menu Screen

Each function was tested independently from another in a constructive manner. When the function worked independently, it was added to the rest of the code and this process continued throughout the entire project.

Function Values

Careful consideration was taken for confirming that values were measured correctly and accurately. For every function, values were displayed during the tests we conducted in various environments. A recording of the how the values changed over time revealed whether they were accurate according to their position.

Tradeoffs

The IR sensor has 2 modes, 1 where the sensor returns a value representing the location of the IR remote relative to the receiver, and another where the receiver collects data transmitted from the remote. The unreferenced function correct_IR is a PID controller which was implemented to make use of the first mode.

The function was designed to take positional data from the IR sensor and relate this data to an error within a PID controller. The PID controller would then adjust the power relayed to the motor the IR receiver was mounted to, turning receiver in the direction of the remote.

Simply put, a function was developed to have the IR sensor receive data from the remote, while also adjusting its position relative to the location of the remote. The robot would "look" at the user while they were controlling the robot.

Unfortunately, this could not be successfully implemented, as the IR sensor could only be in 1 mode at a time. There was even an attempt to manually change the mode while the robot was running, and the function would simply fail.

In conclusion, technological limitation of the IR sensor made it impossible to implement 2 functional programs at once. For this reason, there was a tradeoff between making the robot have a longer range or capabilities, and simply making the robot functional.

Significant challenges and Major Design Decisions

Selecting Menu Options

There were significant challenges around the getButtonValue function. The function contained 3 different arrays with the options for the different menu screens as illustrated in "Figure 2". Originally, the function passed an array parameter by reference containing the desired array for the given menu screen. Although the code compiled successfully, the RenoBot froze and had to be terminated manually by removing the batteries. After careful consideration, in the end it was best to create the array multiple times instead of passing them into the required functions.

Gyroscopic and Coordinate values

Tests were focused on improving gyroscopic and encoder values. However, the most notable of these tests did not have to do with the accuracy of either of these.

A strange occurrence during certain tests happened when x and y values suddenly jumped to numbers far greater than expected. Occurring when the robot would turn left after a left bumper hit, X and Y values would seemingly increase exponentially reaching values beyond the capacity of a float.

After weeks of thought, it had been concluded that there was a logical error when coordinates had been calculated during a left bumper hit. The error occurred due to the original method which CV3 had developed for turning left and measuring coordinates.

The process evolved measuring a coordinate, then basing the next coordinate off the last coordinate. When turning an acute angle, the method used entailed turning 90 degrees left on a left

bumper hit, driving forward a given distance and checking if the right bumper gets hit, and if it does, then changing into the obtuse angle algorithm until the angle is complete. After the angle is complete, then measure the acute angle.

The problem was that often, multiple obtuse angle measurements would be completed, changing the measured coordinates incrementally. By the time the acute angle was finished, the acute angle calculation would commence changing the value of the last coordinate in an illogical manner. This would throw off the previous coordinate value, which changed every coordinate measurement afterwards with an increasing margin of error.

To fix this, an additional component to the coordinate struct was added. An integer value which measured the encoder value of the right motor. A new algorithm for turning was also developed. The algorithm would turn incrementally until the right bumper got hit, then the acute angle calculation would be completed, and the obtuse angle algorithm would commence.

In conclusion, coordinates were measured one after the other, rather than at different times and locations. Basing them mostly on encoder measurements which were independent of position. Although these changes did slow down the robot, they made calculations more accurate and logically correct, with no possibility of such an error occurring again.

Simplicity

Many functions are overcomplicated and hard to interpret due to the mechanical design choices CV3 made, like the gear reduction on the motors, which reversed encoder values. In general, it was a challenge to make the code user and programmer friendly.

To fix this, trivial functions such as drive had been developed by taking the negative values of any parameters passed to them. Sometimes, however, this would make developing other elementary functions such as rotate, rotate_gyro, and drive_dist, more complicated for the programmer.

Some of CV3's biggest challenges were embarrassingly the implementation of drive_dist and rotate. Please refer to the code in "Appendices" for more transparency.

Most notably, however, working as a team turned out to be far more complicated than expected. Close to the demonstration date, CV3 identified that 2 separate functions that were developed by 2 different people could have been combined into 1 singular function. The 2 functions were called point_process and update_coordinate. They have since then been combined into coordinate_process, being the most crucial aspect of the robot's functionality.

In conclusion, simplifying the project tasks and working as efficiently as possible proved to be difficult, but eventually became easier as the project came to completion.

Timing Issues

The first implementation of an if statement comparing an index to a timer incremented said index by a specified time gap at the end of each if statement. This led to a logical error in the code when certain commands would be fulfilled.

The left_turn function is executed anytime the left or right touch sensors read a value of 1. This function completed incremental left turns until the corner in question has successfully been turned. The issue is that while this function had been executed, the timer specified in the main function would continue to change while the function was running. Fast forward till after the left_turn function, and the timer is at a value far greater than the index. This led to the if statement being executed at full processing

speed until the index had reached a value close to the timer. Unfortunately resulting in many errors, most notably an error involving the multiplexor.

To fix this, the iteration of index was changed from changing the index by itself, to changing the index according to the current timer value.

In conclusion, a single line of code completely changed the logic of how the robot operated, helping the robot more precisely execute the appropriate commands.

Multiplexor Troubleshooting

Throughout the development of the RenoBot, numerous errors occurred due to the multiplexor, often occurring when the robot needed to read a value from a sensor in a loop.

Upon analysis of the error messages displayed, it was concluded that they did not reveal any useful information regarding the sensor configuration. Simply put, every piece of code appeared to be logically correct according to the example sketches provided for the multiplexor. Yet still, the multiplexor would lead the robot to fail and exit out of the main function.

Further investigation revealed that the multiplexor does not have the same processing power as the EV3 block. Meaning that the multiplexor requires a time delay between each read command executed in a while loop. After some experimentation, it was concluded that the fastest processing speed of the multiplexor would require a 0.005 second time delay between subsequent commands.

Therefore, to solve this issue, CV3 ensured that every multiplexor reading of sensor values had this 0.005 second time delay to prohibit the robot from failing.

Multiplexor sensor choices

Design choices were made due to the processing speed of the multiplexor. The gyroscopic and ultrasonic sensors were prioritized over the 2 touch sensors. Therefore, they were connected directly to the robot, while the touch sensors were connected to the multiplexor.

The ultrasonic sensor was prioritized because it read a value directly responsible for maintaining the course of the robot. The gyroscopic sensor was prioritized because it was the most critical variable in making accurate coordinate calculations. The touch sensors were not prioritized because they simply had to read a singular value while the robot was running.

In conclusion, sensors with quantitative importance were prioritize over sensors with trivial outputs.

Verification

During the RenoBot demonstration, nearly all criteria and constraints were met, the robot was successful due to careful planning and design of both the code and robot itself.

The First success that the robot saw was in navigating the room autonomously. In the demo the RenoBot successfully moved around the space nearly without issue. The only problem the robot encountered was occasionally the ultrasonic sensor would read no value. This happened only once in the demo, and a wave of the hand in front of the sensor reset it and allowed the robot to continue working. This was an issue on the hardware side of the sensor and was not something that could be fixed in the software design. Apart from this the RenoBot moved on its own, and hence the criteria / constraint of moving autonomously around an enclosed space was met.

The second constraint that was met was the degree of accuracy that the RenoBot must return measurements with. When tested in the demonstration of the robot, the RenoBot displayed values well within the criteria for success. For perimeter, the RenoBot returned a value of 9.98 meters, and the true perimeter of the space measured was 10 meters. In terms of area the Renobot returned 3.68 square meters, and the true area was 3.7 square meters. In both cases the values were well within the expected tolerances.

Lastly, the final constraint that was worked under was regarding the size and shape of the Renobot, and this was done successfully. When the robot had to make a turn about a point, the back end of the robot was so small that it did not collide with the wall it was following. By following this constraint, we were able to successfully complete the full navigation of the room. The physical design of the robot is also an identifier of our success, as it is visibly taller than it is wide, or long.

Project Plan

Initially, the RenoBot was a stud finder, which would drive alongside the walls of a room and mark out the studs as it drove. This eventually morphed into a robot which could calculated the perimeter, then the area after learning about the shoelace algorithm [4]. This became the main goal of the project, during which, other subfunctions were conceptualized, such as the Draw_Corner, and Remote_Area_Perimeter functions.

CV3 expected at the very least to complete the Area_Perimeter Function. All other functions were an addition to compliment the RenoBot's versatility. The project expectations and the actual path and time of the project differed from the fact that CV3 successfully implemented so many versatile sub-functions, that the addition of Draw_Corner and Remote_Area_Perimeter became trivial.

Workload Distribution

Every CV3 member contributed to each presentation equally and professionally.

Name	Tasks
Vishesh Garg	-Mechanical Design of the robot -Navigation code -Area Calculation code Final report Sections • Mechanical Design
Victor Kalenda	-Navigation code -Point plotting code Final report Sections • Software Design (Main Functions, Significant changes, Major Design Decisions) • Project Plan • Recomendations
Vishan Muralikaran	-Menu code -Organized deliverables Final report Sections • Software Design (Main Menu)
Connor Switzer	-Menu code -Designed and built testing stadium -Organized deliverables -Final report Sections • Scope • Criteria and Constraints • Verification • Conclusions

Programming

The following CV3 members worked on the specified functions with assistance from all CV3 members, online references, and teaching staff.

Main Menu Code

Vishan Muralikaran & Connor Switzer (functions created together, using pair programming)

getHeight, getButtonValue, mainMenuScreen, surfaceAreaOfWalls, paintingWalls, drywall, baseboard, findTotalStud, concrete, task main.

Main Code

Victor Kalenda

Area_Perimeter, Remote_Area_Perimeter, Draw_Corner, drive_dist, drive_to_wall, turn_left, rotate, rotate_gyro, configure_sensors, correct_path, correct_IR, coordinate_process, trace_corner, map_corner, wait_read

Vishesh Garg

Area_Perimeter, Remote_Area_Perimeter, Draw_Corner, drive_dist, drive, turn_left, rotate, reset_variables, calibrate_gyro, coordinate_process, set_pencil, correct_IR, remote_drive, reset_IR, mission_possible

Conclusions

As mentioned in the introduction, the problem which the RenoBot was designed to solve was aiding a human in the construction or renovation of a space. As proven in the demo the robot succeeded in doing this by moving autonomously and accurately measuring the space it moved around, as declared in the criteria and constraints of the project. These successes were a result of careful planning, and key components of the design. The touch sensors and bumpers worked in unison with the ultrasonic sensor to allow the RenoBot to move around the space, the gear ratioed drive train allowed for precise motor encoder measurements, and the point plotting software allowed everything to happen correctly. Additionally, the draw angle function and functions which returned construction information also worked correctly because of the previously mentioned hardware and software items.

Recommendations

Mechanical design recommendations

A great deal of thought was put into every mechanical specification of the RenoBot. Yet still, design opportunities were missed due to lack of time and necessity.

Pencil Mechanism

Many preliminary designs involving CV3's pencil lowering mechanism incorporated a spring to load the pencil during its operation. CV3 did not implement this design into the final revision of the RenoBot due to its lack of necessity at the time. Such a mechanism would have made for a more consistent line with a wider variety of surfaces, making drawing on things such as plywood easier and more versatile.

Weight Distribution

The final design of the RenoBot was top and front heavy making it prone to tipping forward depending on the motor power constants set in the code. Most of this is due to the heavy weight of the EV3 brick and the very specific restrictions of the pencil's location along with its motor. Adding weight

to the back of the robot would have made the robot capable of faster speeds, making it more versatile for larger areas where precision to the nearest centimeter is not a necessity.

Bumpers

RenoBot was designed with the intent of have the bumpers of the robot in line with the wheels alongside the wall it would travel beside. This would allow the robot to have the closes error tolerance possible to the wall while also yielding the most accurate obtuse angle measurements possible. The final design unfortunately extends around 1 centimeter passed the wheel, forcing the RenoBot to increase its PID_TARGET constant value from the wall. This makes for less precise measurements, and a larger margin of error for the final calculated area and perimeter.

Software recommendations

Pencil calibration

Towards the finalization of the Draw_Corner function for the RenoBot, a need came up for a function to calibrate the pencil lowering function of the robot. Such a function was not completed due to the shear complexity of designing a program that sets and stores multiple variables for positions, while also being user friendly. In addition to this, the user can relatively easy hand set the pencil location after a bit of trial and error with the Draw_Corner function. The user would simply run the function on a short straight line and physically move the pencil until they place it in an appropriate position for drawing.

Concrete

The initial intention of a concrete volume measurement came from the idea of having a Remote_Area_Perimeter function. The idea was that the robot would be able to drive along the sides of a driveway and determine how much concrete would be needed for that area. For this reason, concrete was only displayed if the user had completed the Remote_Area_Perimeter function.

However, upon further analysis, CV3 had concluded that the function would still be useful in an enclosed area. This capability was not implemented due to time constraints. However, having done so would have given the robot an extra level of versatility.

References

[1] StackOverflow, “PID controller and transfer function in C++” StackOverflow. January, 2020 [Online] Available: https://stackoverflow.com/questions/50017307/pid-controller-and-transfer-function-in-c

[2] Home Hardware, “Painting calculator” Home Hardware. [Online] Available: https://www.homehardware.ca/en/painting-calculator?gclid=CjwKCAiAyfybBhBKEiwAgtB7fu9vuBW7fhs9idyGqXKTlpsMARPTZ4Yh3JZN0nagTFXfX53zI_6TNBoCfAsQAvD_BwE&gclsrc=aw.ds

[3] The Home Depot, “Types of Drywall” The Home Depot. [Online] Available: https://www.homedepot.com/c/ab/types-of-drywall/9ba683603be9fa5395fab90c24feaae

[4] AoPS Online, “Shoelace Theorem” Art of Problem Solving. [Online] Available: https://artofproblemsolving.com/wiki/index.php/Shoelace_Theorem

[5] Chambers, A. “The Shoelace Algorithm to find areas of polygons” IB Maths Resources from Intermathematics. October 7, 2019. [Online] Available: https://ibmathsresources.com/2019/10/07/the-shoelace-algorithm-to-find-areas-of-polygons/

[6] Erasmus+ Programme of the European Union *PROGRAMMING MOBILE ROBOTS IN THE ROBOTC PROGRAMMING ENVIRONMENT*. Published 2019 [E-book] Available: https://discover-project.eu/uploads/text/4/documents_en/Programming%20Robots%20with%20Robot%20C%20-%20A%20Practical%20Guide.pdf

[7] RobotC, “getIRRemoteChannelButtons” RobotC a C programming Language for Robots. January 27, 2016. [Online] Available: https://www.robotc.net/WebHelpMindstorms/index.htm#Resources/topics/LEGO_EV3/ROBOTC/Sensors/IR_Sensor/getIRRemoteButtons.htm

Appendices

The following is the completed RenoBot code demonstrated to the TAs

```
/*
Group 4-22
Authors: Vishesh Garg, Victor Kalenda, Vishan Muralikaran, Connor Switzer
Version 42.0

Description: This is the final product code of group 4-22 for the MTE 100/121
final project. This code begins with a menu screen which allows the user to
select a task for the Renobot to complete, this will then run the selected task,
and then return to the main menu.

Tasks include:
Calculate Area / Perimeter
Measure and draw an angle
Calculate expected number of various materials (Paint, drywall etc...)


Assumptions:

The Renobot drives counter clockwise around a space, you should orient the
ultrasonic sensor so that it faces an exterior wall of the room you want to
measure.

The wall must be opaque, relatively smooth and continuous.

There must not be any obstructions in the path of the Renobot.

Any space with width smaller than the width of the Renobot will not be measured.

The robot cannot be started facing a corner.

Failure to adhere to these suggestions may result in an inaccurate measurment.
*/

#include "UW_sensorMux.c"


typedef struct

{
  float right;
  float left;
```

```
  float time;
  float proportional;
  float IR;
} past;

typedef struct

{
  float initial_ang;
  float final_ang;
  float right;
  float left;
  float proportional;
  float integral;
  float derivative;
  float origin_tol;
  float IR;
} present;

typedef struct

{
  float x;
  float y;
  int encoder;
} coordinate;

// Conversion Constants
const float DEG_TO_RAD = PI / 180;
const float RAD_TO_DEG = 180 / PI;
const float ENC_TO_DIST = 9 * PI / 1000;
const float DIST_TO_ENC = 1000 / (9 * PI);
const float CM2_TO_M2 = 1 / 10000.0;
const float CM_TO_M = 1 / 100.0;

// PID Constants
const float PID_TARGET = 10;
const float CONST_PROPORTIONAL = 2;
const float CONST_INTEGERAL = 10;
const float CONST_DERIVATIVE = 10;
const float ERROR_TOL = 10;
const int MUX_TOL = 5;

// Drive Constants
const float DRIVE = 40;
const float TRACE_DRIVE = 10;
const float TURN_DRIVE = 25;
const int ENCODER_OFFSET = 5;
```

```
const float RENO_WIDTH = 22.5;
const float MARGINAL_DIST = -5;
const float MARGINAL_ANG = -15;
const float DIST_TOL = 7;
const float TIME_GAP = 500;
const float ACUTE_MARGINAL_ANG = -5;
const float ACUTE_MARGINAL_DIST = -5;
const float ANG_TOL = 0.4;

// Origin Constants
const float ORIGIN_FACTOR = 1 / 5.5;
const int ORIGIN_SHIFT = 4;

// IR Constants
const int IR_DRIVE_MOTOR_POWER = 70;
const float IR_CONST_PROPORTIONAL = 0.5;
const float IR_CONST_INTEGERAL = 1.5;
const float IR_CONST_DERIVATIVE = 0.1;
const float IR_ERROR_TOL = 0;
const int IR_TIME_TOL = 5;
const int IR_POWER_MAX = 50;
const int IR_RESET_ANGLE = 360;
const int IR_RESET_TOL = 5;
const int IR_TURN_MODE = 0;
const float IR_AREA_OFFSET_FACTOR = 1;
const float IR_PERIM_OFFSET_FACTOR = 1;

// Coordinate Process Constants
const float TIRE_TO_WALL = 3;
const float AREA_OFFSET_FACTOR = 1;
const float PERIM_OFFSET_FACTOR = 1;


// Pencil Constants
const int PENCIL_INCREMENT = 55;
const int PENCIL_POWER = 20;
const int MAX_SIZE = 120;

// Array for coordinate data collection (Used for Draw_corner function)
coordinate data[MAX_SIZE];


// Main Menu Function Prototypes
bool checkForArea(float closedAreaCalc, float closedPerimCalc,
                  float openAreaCalc, float openPerimCalc, string typeOfArea);
int mainMenuScreen(float openArea, float openPerimeter, float closedArea,
                   float closedPerimeter);
```

```
float surfaceAreaOfWalls(float heightOfRoom, float closedPerimeter);
void findTotalStud(float closedPerimeter);
void paintingWalls(float surfaceArea);
void baseboard(float closedPerimeter);
int getButtonValue(int numOfOptions);
void drywall(float surfaceArea);
void concrete(float openArea, float thicknessOfPavement);
int getHeight(int interval);

// Area Perimeter Function Prototypes
void configure_sensors(void);
void drive(float left_pow, float right_pow);
void turn_left(int &mode, float &area, float &perimeter, float &origin_tol,
                    present &curr, coordinate &C1, coordinate &C2, bool &roomFinished);
void correct_path(past &prev, present &curr);
void drive_dist(float motor_pow, float dist);
void rotate(float motor_pow, float ang);
void drive_to_wall(int &mode, float &initial_dist);
void Area_Perimeter(float &area, float &perimeter, past &prev, present &curr);
bool coordinate_process(int turn_mode, present & curr, float &area, float &perimeter, float &origin_tol,
                coordinate &C1, coordinate &C2);
void reset_variables(past & prev, present & curr, float & area, float & perimeter, coordinate & C1,
coordinate & C2);
void calibrate_gyro(void);

// Draw Corner Function Prototypes
void rotate_gyro(float motor_pow, float ang);
void map_corner(int & data_size, past & prev, present & curr);
void trace_corner(int & data_size);
void Draw_Corner(past & prev, present & curr);
void set_pencil(bool action);
void wait_read(void);

// IR Function Prototypes
void Remote_Area_Perimeter(float & area, float & perimeter, past & prev, present & curr);

void correct_IR(past & prev, present & curr);
void remote_drive();
void reset_IR();
void mission_possible();



task main()

{
  bool shutDown = false;
```

```
  float openAreaCalc = 0;
  float openPerimCalc = 0;
  float closedAreaCalc = 0;
  float closedPerimCalc = 0;
  float heightOfRoom = 1;
  float thicknessOfPavement = 1;

  present curr;
  past prev;
  coordinate C1, C2;

  reset_variables(prev, curr, closedAreaCalc, closedPerimCalc, C1, C2);

  // Runs code while shut down option isn't pressed
  while (shutDown == false)

{
    bool returnToMain = false;
    while (returnToMain == false)

    {
      // Main Menu Screen
      int buttonValue =
          mainMenuScreen(openAreaCalc, openPerimCalc, closedAreaCalc,closedPerimCalc); // change
later
      eraseDisplay();

      // Area and Perimeter Screen
      while (buttonValue == 1 && returnToMain == false)

      {
        int buttonValue = 1;
        string areaDisplayOption[2] = {" Enclosed", " Open"};
        displayBigTextLine(5, "%d %s", buttonValue,areaDisplayOption[buttonValue - 1]);

        displayBigTextLine(3, "Area/Perimeter");
        displayBigTextLine(8, "1 Enclosed");
        displayBigTextLine(10, "2 Open");

        int areaButtonValue = getButtonValue(2);

        if (areaButtonValue == 1)

        {
          eraseDisplay();
          wait_read();
          eraseDisplay();
          displayBigTextLine(3, "Calculating");
```

```
  displayBigTextLine(5, "Area and");
  displayBigTextLine(7, "Perimeter ...");
  Area_Perimeter(closedAreaCalc, closedPerimCalc, prev, curr);
  eraseDisplay();
  displayBigTextLine(3, "Area:");
  displayBigTextLine(5, "%.2fm^2", closedAreaCalc);
  displayBigTextLine(9, "Perimeter:");
  displayBigTextLine(11, "%.2fm", closedPerimCalc);
  while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

  {}
  while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
        !getButtonPress(buttonRight))

  {}
  returnToMain = true;
  eraseDisplay();
}

else if (areaButtonValue == 2)

{
  eraseDisplay();
  wait_read();
  eraseDisplay();
  displayBigTextLine(3, "Calculating");
  displayBigTextLine(5, "Area and");
  displayBigTextLine(7, "Perimeter ...");
  Remote_Area_Perimeter(openAreaCalc, openPerimCalc, prev, curr);
  eraseDisplay();
  displayBigTextLine(3, "Area:");
  displayBigTextLine(5, "%.2fm^2", openAreaCalc);
  displayBigTextLine(9, "Perimeter:");
  displayBigTextLine(11, "%.2fm", openPerimCalc);
  while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

  {}
  while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
        !getButtonPress(buttonRight))

 {}
  returnToMain = true;
  eraseDisplay();
}

if (getButtonPress(buttonLeft))

{
  returnToMain = true;
  while (getButtonPress(buttonLeft))
```

```
  {}
 }
}

// Map Angle Function
if (buttonValue == 2 && returnToMain == false)

{
 eraseDisplay();
 displayBigTextLine(3, "Calculating");
 displayBigTextLine(5, "Angle ...");
 Draw_Corner(prev, curr);
 eraseDisplay();
 displayBigTextLine(3, "Angle");
 displayBigTextLine(5, "Calculated!");
 while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

 {}
 while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
       !getButtonPress(buttonRight))

 {}
 returnToMain = true;
 eraseDisplay();
}

// Materials Screen
while (buttonValue == 3 && returnToMain == false)

{
 int buttonValue = 1;
 int materialsButtonValue = 0;
 string materialDisplayOption[5] = {" Paint", " Drywall", " Baseboard", " Studs", " Concrete"};
 displayBigTextLine(3, "%d %s", buttonValue, materialDisplayOption[buttonValue - 1]);


 displayBigTextLine(1, "   Materials");
 displayBigTextLine(6, "1 Paint");
 displayBigTextLine(8, "2 Drywall");
 displayBigTextLine(10, "3 Baseboard");
 displayBigTextLine(12, "4 Studs");
 displayBigTextLine(14, "5 Concrete");

 string typeOfArea = " ";
 bool hasAreaAndPerimeter = checkForArea(closedAreaCalc, closedPerimCalc, openAreaCalc,
            openPerimCalc, typeOfArea);
 if (hasAreaAndPerimeter)
```

```
{
 materialsButtonValue = getButtonValue(5);
}

 else

{
 returnToMain = true;
}

if (materialsButtonValue == 1) // Gets surface area, paint time, and amount of paint
{
 typeOfArea = " Closed ";
 hasAreaAndPerimeter =
    checkForArea(0, closedPerimCalc, 0, 0, typeOfArea);
 if (hasAreaAndPerimeter)

 {
  eraseDisplay();
  displayBigTextLine(1, "Enter height");
  displayBigTextLine(3, "of the room: ");
  displayBigTextLine(7, "0 feet");
  heightOfRoom = getHeight(1);
  while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

  {}
  eraseDisplay();
  if (heightOfRoom != 0 && !getButtonPress(buttonLeft))

  {
    paintingWalls(surfaceAreaOfWalls(heightOfRoom, closedPerimCalc));
    while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

    {}
    while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
          !getButtonPress(buttonRight))

    {}
    eraseDisplay();
  }
 }

 else

{
  returnToMain = true;
 }
}

else if (materialsButtonValue == 2) // Gets amount of drywall needed for surface area
{
```

```
 typeOfArea = " Closed ";
 hasAreaAndPerimeter = checkForArea(0, closedPerimCalc, 0, 0, typeOfArea);
 if (hasAreaAndPerimeter)

 {
  eraseDisplay();
  displayBigTextLine(1, "Enter height");
  displayBigTextLine(3, "of the room: ");
  displayBigTextLine(7, "0 feet");
  heightOfRoom = getHeight(1);
  while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

  {}
  eraseDisplay();
  if (heightOfRoom != 0 && !getButtonPress(buttonLeft))

  {
   drywall(surfaceAreaOfWalls(heightOfRoom, closedPerimCalc));
   while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

   {}
   while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
         !getButtonPress(buttonRight))

   {}
   eraseDisplay();
  }
 }

 else

 {
  returnToMain = true;
 }
}

 else if (materialsButtonValue == 3) // Gets amount of baseboard needed for surface area
{
 typeOfArea = " Closed ";
 hasAreaAndPerimeter = checkForArea(0, closedPerimCalc, 0, 0, typeOfArea);
 if (hasAreaAndPerimeter)

 {
  eraseDisplay();
  baseboard(closedPerimCalc);
  while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

  {}
  while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
        !getButtonPress(buttonRight))
```

```
    {}
    eraseDisplay();
   }
   else
  {
    returnToMain = true;
   }
 }
  else if (materialsButtonValue == 4) // Gets amount of studs in wall
 {
  typeOfArea = " Closed ";
  hasAreaAndPerimeter = checkForArea(0, closedPerimCalc, 0, 0, typeOfArea);
  if (hasAreaAndPerimeter)
  {
    eraseDisplay();
    findTotalStud(closedPerimCalc);
    while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))
    {}
    while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
          !getButtonPress(buttonRight))
    {}
    eraseDisplay();
   }
   else
  {
    returnToMain = true;
   }
 }
else if (materialsButtonValue == 5)
{
  typeOfArea = " Open ";
  hasAreaAndPerimeter = checkForArea(0, 0, openAreaCalc, 0, typeOfArea);
  if (hasAreaAndPerimeter)
  {
    eraseDisplay();
    displayBigTextLine(1, "Enter thickness:");
    displayBigTextLine(7, "0 feet");
    thicknessOfPavement = getHeight(1);
    while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))
```

```
          {}
          eraseDisplay();
          if (thicknessOfPavement != 0 && !getButtonPress(buttonLeft))
          {
           concrete(openAreaCalc, thicknessOfPavement);
           while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))
           {}
           while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
                 !getButtonPress(buttonRight))
           {}
           eraseDisplay();
          }
         }
         else
         {
          returnToMain = true;
         }
        }
        if (getButtonPress(buttonLeft))
        {
         returnToMain = true;
         while (getButtonPress(buttonLeft))
         {}
        }
       }
       // Shut down
       if (buttonValue == 4 && returnToMain == false)
       {
        returnToMain = true;
        shutDown = true;
       }
      }
     }
    }

/*************************************************************************
Displays main menu screen (might remove function later)
*************************************************************************/
int mainMenuScreen(float openArea, float openPerimeter, float closedArea, float closedPerimeter)
```

```
{
  eraseDisplay();
  int buttonValue = 1;
  string menuDisplayOption[4] = {" Area", " Draw Angle", " Materials", " Shut Down"};

  // Main Menu Screen
  displayBigTextLine(3, "  Select Mode");
  displayBigTextLine(8, "1 Area");
  displayBigTextLine(10, "2 Draw Angle");
  displayBigTextLine(12, "3 Materials");
  displayBigTextLine(14, "4 Shut Down");

  displayBigTextLine(5, "%d %s", buttonValue, menuDisplayOption[buttonValue - 1]);
  buttonValue = getButtonValue(4);

  return buttonValue;
}

/*********************************************************
Find surface area of walls
*********************************************************/
float surfaceAreaOfWalls(float heightOfRoom, float closedPerimeter)

{
  const float feetToMeters = 0.3048;
  float heightToMeters = heightOfRoom * feetToMeters;
  float surfaceArea = heightToMeters * closedPerimeter;
  return surfaceArea;
}

/*********************************************************
Find paint time and amount function
*********************************************************/
void paintingWalls(float surfaceArea)

{
  const int PAINT_PER_LITER = 6;
  const float LITER_TO_GAL = 3.785;
  const int AVERAGE_PAINT_PER_HOUR = 7;

  float totalPaintLiters = surfaceArea / PAINT_PER_LITER;
  float totalPaintGal = totalPaintLiters / LITER_TO_GAL;

  float paintTime = surfaceArea / AVERAGE_PAINT_PER_HOUR;

  displayBigTextLine(1, "Surface area:");
  displayBigTextLine(3, "%.2fm^2", surfaceArea);
  displayBigTextLine(6, "Paint Time:");
```

```
  displayBigTextLine(8, "%.2f hours", paintTime);
  displayBigTextLine(11, "Liters: %.2f", totalPaintLiters);
  displayBigTextLine(14, "Gallons: %.2f", totalPaintGal);
}

/*********************************************************
Drywall Function
*********************************************************/
void drywall(float surfaceArea)

{
  float drywallConversion = 2.88;
  float amountOfDrywall = surfaceArea / drywallConversion;

  displayBigTextLine(1, "Surface area:");
  displayBigTextLine(3, "%.2fm^2", surfaceArea);
  displayBigTextLine(7, "# of Boards: ");
  displayBigTextLine(9, "%.2f", amountOfDrywall);
}

/*********************************************************
Baseboard Function
*********************************************************/
void baseboard(float closedPerimeter)

{
  int amountOfBaseboards = closedPerimeter;

  displayBigTextLine(1, "Baseboards: ");
  displayBigTextLine(3, "%.2f", amountOfBaseboards);
}

// Find total amount of studs
void findTotalStud(float closedPerimeter)

{
  const float AMOUNT_OF_STUDS_DISTANCE = 0.4064;
  int totalAmountOfStuds = ceil(closedPerimeter / AMOUNT_OF_STUDS_DISTANCE);

  displayBigTextLine(3, "Estimated Studs: ");
  displayBigTextLine(6, "%d", totalAmountOfStuds);
}
/*********************************************
Finds Total Amount of Concrete Needed
*********************************************/
void concrete(float openArea, float thicknessOfPavement)

{
```

```
  float areaVolume = openArea * thicknessOfPavement;
  float concreteConversion = 5.4;
  float concreteAmount = areaVolume / concreteConversion;

  displayBigTextLine(3, "# of Concrete: ");
  displayBigTextLine(6, "%dm^3", concreteAmount);
}

/*********************************************************
Gets Height of Room with Button Presses
*********************************************************/
int getHeight(int interval)

{
  int buttonValue = 0;

  while (getButtonPress(buttonEnter) || getButtonPress(buttonRight))

  {}
  while (!getButtonPress(buttonEnter) && !getButtonPress(buttonRight))

  {
    if (getButtonPress(buttonLeft)) // Leaves and changes nothing in task main
                                    // when left button is pressed
    {
      return 0;
    }
    if (getButtonPress(buttonDown))

    {
      if (buttonValue > 0)

      {
        buttonValue -= interval;
        displayBigTextLine(7, "%d feet", buttonValue);
        while (getButtonPress(buttonDown))

        {}
      }
    }

     else if (getButtonPress(buttonUp))

     {
      if (buttonValue < 1000)

      {
        buttonValue += interval;
        displayBigTextLine(7, "%d feet", buttonValue);
        while (getButtonPress(buttonUp))
```

```
      {}
    }
   }
  }
  return buttonValue;
}

/*********************************************************************
Checks To See If There Are Values For Area And Perimeter
*********************************************************************/
bool checkForArea(float closedAreaCalc, float closedPerimCalc,
                  float openAreaCalc, float openPerimCalc, string typeOfArea)

{
  if (closedAreaCalc != 0 || closedPerimCalc != 0 || openAreaCalc != 0 || openPerimCalc != 0)

  {
    return true;
  }

  else

  {
    eraseDisplay();
    displayBigTextLine(3, "No%sArea", typeOfArea);
    displayBigTextLine(5, "and Perimeter ");
    displayBigTextLine(7, "Found.");
    displayBigTextLine(11, "Run%sArea", typeOfArea);
    displayBigTextLine(13, "Function");
    while (getButtonPress(buttonEnter) || getButtonPress(buttonLeft) || getButtonPress(buttonRight))

    {}
    while (!getButtonPress(buttonEnter) && !getButtonPress(buttonLeft) &&
           !getButtonPress(buttonRight))

    {}
    eraseDisplay();
    return false;
  }
}

/*********************************************************************************
********
Changes the button values with respected menu values and returns the value when
pressed
*********************************************************************************
*******/
int getButtonValue(int numOfOptions)
```

```
{
 int buttonValue = 1;

 // Can change to passing in an array
 string menuDisplayOption[4] = {" Area", " Draw Angle", " Materials", " Shut Down"};
 string materialDisplayOption[5] = {" Paint", " Drywall", " Baseboard", " Studs", " Concrete"};
 string areaDisplayOption[2] = {" Enclosed", " Open"};

 string displayOption = menuDisplayOption[buttonValue - 1];

 while (getButtonPress(buttonEnter) || getButtonPress(buttonRight))

 {} // When button is released, run the rest of the code (only for press left)
 while (!getButtonPress(buttonEnter) && !getButtonPress(buttonRight))

 {
  if (getButtonPress(buttonLeft)) // Leaves and changes nothing in task main
                                  // when left button is pressed
  {
   return 0;
  }
  if (getButtonPress(buttonUp))

  {
   if (buttonValue > 0)

   {
    buttonValue--;
    if (buttonValue == 0)

    {
     buttonValue = numOfOptions;
    }
    if (numOfOptions == 4)

    {
     displayOption = menuDisplayOption[buttonValue - 1];
     displayBigTextLine(5, "%d %s", buttonValue, displayOption);
    }

     else if (numOfOptions == 5)

    {
     displayOption = materialDisplayOption[buttonValue - 1];
     displayBigTextLine(3, "%d %s", buttonValue, displayOption);
    }

    else if (numOfOptions == 2)

    {
     displayOption = areaDisplayOption[buttonValue - 1];
```

```
        displayBigTextLine(5, "%d %s", buttonValue, displayOption);
       }
      }
      while (getButtonPress(buttonUp))

      {}
    }

    else if (getButtonPress(buttonDown))

    {
     if (buttonValue < numOfOptions + 1)

     {
      buttonValue++;
      if (buttonValue == numOfOptions + 1)

      {
       buttonValue = 1;
      }
      if (numOfOptions == 4)

      {
       displayOption = menuDisplayOption[buttonValue - 1];
       displayBigTextLine(5, "%d %s", buttonValue, displayOption);
      }

       else if (numOfOptions == 5)

      {
       displayOption = materialDisplayOption[buttonValue - 1];
       displayBigTextLine(3, "%d %s", buttonValue, displayOption);
      }

       else if (numOfOptions == 2)

      {
       displayOption = areaDisplayOption[buttonValue - 1];
       displayBigTextLine(5, "%d %s", buttonValue, displayOption);
      }
     }
     while (getButtonPress(buttonDown))

     {}
    }
   }
   return buttonValue;
}

// ---------------------- END OF MAIN MENU FUNCTIONS ----------------------------------------------
// ------------------------------------------------------------------------------------------------
```

```
void Area_Perimeter(float &area, float &perimeter, past &prev, present &curr)
{
 configure_sensors();
 calibrate_gyro();
 bool roomFinished = false;
 time1[T1] = 0; // PID
 time1[T2] = 0; // Area Perim
 time1[T3] = 0; // IR PID

 coordinate C1, C2;
 reset_variables(prev, curr, area, perimeter, C1, C2);

 int turn_mode = 0;
 int index = 0;
 int index1 = 0;
 float origin_tol = 0;

 // Drive out of the tolerance of the origin vector
 drive(DRIVE, DRIVE);
 wait1Msec(1000);



 // find the initial gyro reading to offset all calculations accordingly

 // Drive until you complete the room
 while (!roomFinished && !getButtonPress(buttonEnter))
 {

  if (time1[T1] > index)
 {
   turn_mode = 0;
   curr.initial_ang = getGyroDegrees(S2) * DEG_TO_RAD;
   correct_path(prev, curr);

   if (readMuxSensor(msensor_S1_3) == 1)
   {
    drive(0, 0);
    turn_mode = 1;
    turn_left(turn_mode, area, perimeter, origin_tol, curr, C1, C2, roomFinished);

   }
   else if (readMuxSensor(msensor_S1_1) == 1)
```

```
    {
      drive(0, 0);
      turn_mode = 2;
      turn_left(turn_mode, area, perimeter, origin_tol, curr, C1, C2, roomFinished);
    }
    index = time1[T1] + MUX_TOL;
  }


  // Calculate x2, y2, and process the perimeter and area
  if (time1[T2] > index1)

  {
      curr.final_ang = getGyroDegrees(S2) * DEG_TO_RAD;
    roomFinished = coordinate_process(turn_mode, curr, area, perimeter, origin_tol, C1, C2);
    index1 = time1[T2] + TIME_GAP;
  }
 }
 while (getButtonPress(buttonEnter))

 {}
 area = (pow(fabs(area), AREA_OFFSET_FACTOR)) * CM2_TO_M2;
 perimeter = (pow(fabs(perimeter + origin_tol), PERIM_OFFSET_FACTOR)) * CM_TO_M;
 drive(0, 0);
 mission_possible();
}

bool coordinate_process(int turn_mode, present & curr, float &area, float &perimeter, float &origin_tol,
              coordinate &C1, coordinate &C2)

{

 if (fabs(area) > ORIGIN_SHIFT)
      origin_tol = ORIGIN_FACTOR * sqrt(fabs(area) - ORIGIN_SHIFT);
 // Because we are driving counter-clockwise

 // Note that all the display commands that are commented out
 //  are for testing purposes.

 //curr.initial_ang = curr.initial_ang * DEG_TO_RAD;
 //curr.final_ang = curr.final_ang * DEG_TO_RAD;

 C2.encoder = nMotorEncoder[motorA];

 // displayBigTextLine(2, "init Ang = %f", curr.initial_ang);
 // displayBigTextLine(4, "fin Ang = %f", curr.final_ang );
 // displayBigTextLine(6, "Gyr Ang = %f", getGyroDegrees(S2));
```

```
  // If driving alongside a wall or in any right turn
  if (turn_mode == 0)

  {
    C2.x = C1.x + (-(C2.encoder - C1.encoder) * ENC_TO_DIST) * cos(curr.final_ang);
    C2.y = C1.y + (-(C2.encoder - C1.encoder) * ENC_TO_DIST) * sin(curr.final_ang);
  }

  // If in an obtuse left turn
  else if (turn_mode == 1)

  {
    C2.x = C1.x + TIRE_TO_WALL * cos(curr.initial_ang);
    C2.y = C1.y + TIRE_TO_WALL * sin(curr.initial_ang);
  }

  // If in an acute left turn
  else
  {
            float interior_ang = PI - curr.final_ang + curr.initial_ang;
            C2.x = C1.x + (RENO_WIDTH * cos(interior_ang) / sin(interior_ang) + TIRE_TO_WALL) *
cos(curr.initial_ang);
            C2.y = C1.y + (RENO_WIDTH * cos(interior_ang) / sin(interior_ang) + TIRE_TO_WALL) *
cos(curr.initial_ang);
  }
  displayBigTextLine(10, "X = %f", C2.x); //---------------------------------------------- FOR TESTING
  displayBigTextLine(12, "Y = %f", C2.y); //---------------------------------------------- FOR TESTING

  C1.encoder = C2.encoder;

  // Beginning of Point_process----------------------------------------------
  // Calculate area with Shoelace algorithm
  area += ((C1.x * C2.y) - (C1.y * C2.x)) / 2;

  // Calculate perimeter
  perimeter += sqrt(pow(C2.y - C1.y, 2) + pow(C2.x - C1.x, 2));

  // Check if you have returned to the origin
  if (fabs(area) > ORIGIN_SHIFT)

  {
    if (fabs(C2.x) < origin_tol && fabs(C2.y) < origin_tol)
      return true;
  }
  C1.x = C2.x;
  C1.y = C2.y;

  return false;
```

```
}

void turn_left(int &mode, float &area, float &perimeter, float &origin_tol, present &curr, coordinate
&C1, coordinate &C2, bool &roomFinished)
{
 float initial_dist = 0;
 // Right touch sensor hit, angles = A, C, I
 while (mode != 0)

 {
  initial_dist = fabs(nMotorEncoder[motorA] * ENC_TO_DIST);
  if (mode == 1)

  {
   // Obtuse Angle
   while (readMuxSensor(msensor_S1_3) == 1)

    {
     curr.initial_ang = getGyroDegrees(S2) * DEG_TO_RAD;
     roomFinished = coordinate_process(mode, curr, area, perimeter, origin_tol, C1, C2);
     drive_dist(TURN_DRIVE, MARGINAL_DIST);
     rotate(TURN_DRIVE, MARGINAL_ANG);
     drive_to_wall(mode, initial_dist);
     }
  }
  // Acute Angle
  // if (mode == 2)
  else

  {
      curr.initial_ang = getGyroDegrees(S2) * DEG_TO_RAD;
      while(readMuxSensor(msensor_S1_3) == 0 || readMuxSensor(msensor_S1_1) == 1)
      {
      drive_dist(TURN_DRIVE, ACUTE_MARGINAL_DIST);
      rotate(TURN_DRIVE, ACUTE_MARGINAL_ANG);
      drive_to_wall (mode, initial_dist);
      }
      mode = 1;
      curr.final_ang = getGyroDegrees(S2) * DEG_TO_RAD;
      roomFinished = coordinate_process(mode, curr, area, perimeter, origin_tol, C1, C2);
  }
 }
}


void drive_to_wall(int &mode, float &initial_dist)

{
 drive(TURN_DRIVE, TURN_DRIVE);
```

```
  if (mode == 2)
  {
        while (readMuxSensor(msensor_S1_1) == 0 && readMuxSensor(msensor_S1_3) == 0)
        {
              wait1Msec(MUX_TOL);
        }
  }
  else
  {
        while (readMuxSensor(msensor_S1_3) == 0 &&
        fabs(nMotorEncoder[motorA] * ENC_TO_DIST) - initial_dist < DIST_TOL)

        {
          wait1Msec(MUX_TOL);
        }
  }
  drive(0, 0);

  if (mode != 2)
  {
        if (fabs(nMotorEncoder[motorA] * ENC_TO_DIST) - initial_dist > DIST_TOL)
        {
        mode = 0;
        }
        else
        {
        mode = 1;
        }
    }
}



void correct_path(past &prev, present &curr)

{
  // PID control function adjusting to motor power ratings
  float control_signal = 0;
  float magnitude_of_power = 0;
  float error = 0;

  // Calculate the change in time (in milliseconds)
  float curr_time = time1[T1];
  float delta_time = curr_time - prev.time;
  prev.time = curr_time;

  // Calculate info on the error/time graph
```

```
  error = SensorValue[S3] - PID_TARGET;
  curr.proportional = fabs(error);
  curr.integral = (curr.integral + curr.proportional) / delta_time;
  curr.derivative = (curr.proportional - prev.proportional) / delta_time;
  prev.proportional = curr.proportional;

  control_signal = CONST_PROPORTIONAL * curr.proportional +
                   CONST_INTEGERAL * curr.integral +
                   CONST_DERIVATIVE * curr.derivative;
  magnitude_of_power = fabs(control_signal);

  if (error >= 0 && error < ERROR_TOL)

  {
    curr.right = DRIVE - magnitude_of_power;
    curr.left = DRIVE + magnitude_of_power;
  }

  else if (error < 0)

  {
    curr.right = DRIVE + 2 * magnitude_of_power;
    curr.left = DRIVE - 2 * magnitude_of_power;
  }

   else

  {
    curr.right = 1 / 3.0 * DRIVE;
    curr.left = DRIVE + DRIVE * 1 / 3.0;
  }

  if (prev.right != curr.right /*|| prev.left != curr.left*/)

  {
    drive(curr.left, curr.right);
  }
  prev.right = curr.right;
  prev.left = curr.left;
}


void drive_dist(float motor_pow, float dist)

{
  int orig_enc = nMotorEncoder[motorA];
  int target_encoder = 0;

  target_encoder = orig_enc - (dist * DIST_TO_ENC);
```

```
  if (target_encoder > orig_enc)
  {
    drive(-motor_pow, -motor_pow);

    while (target_encoder > nMotorEncoder[motorA] + ENCODER_OFFSET)
    {}
  }
  else
  {
    drive(motor_pow, motor_pow);

    while (target_encoder < nMotorEncoder[motorA] - ENCODER_OFFSET)
    {}
  }
  drive(0, 0);
}


void drive(float left_pow, float right_pow)
{
  motor[motorA] = -1 * right_pow;
  motor[motorD] = -1 * left_pow;
}


void rotate(float motor_pow, float ang)
{
  int orig_ang = SensorValue[S2];
  int target_ang = 0;

  target_ang = orig_ang + ang;

  if (target_ang > orig_ang)
  {
    drive(motor_pow, -motor_pow);

    while (target_ang > SensorValue[S2])
    {}
  }
  else
```

```
  {
    drive(-motor_pow, motor_pow);

    while (target_ang < SensorValue[S2])

    {}
  }
  drive(0, 0);
}

void calibrate_gyro(void)
{
  wait1Msec(100);
  SensorMode[S2] = modeEV3Gyro_Calibration;
  wait1Msec(100);
  SensorMode[S2] = modeEV3Gyro_RateAndAngle;
  wait1Msec(100);
  resetGyro(S2);
}

void configure_sensors(void)

{
  SensorType[S1] = sensorEV3_GenericI2C;
  wait1Msec(100);
  if (!initSensorMux(msensor_S1_1, touchStateBump))
    return;
  if (!initSensorMux(msensor_S1_3, touchStateBump))
    return;

  SensorType[S2] = sensorEV3_Gyro;
  wait1Msec(50);
  calibrate_gyro();
  SensorType[S3] = sensorEV3_Ultrasonic;
  wait1Msec(50);
  SensorType[S4] = sensorEV3_IRSensor;
  wait1Msec(50);
}

void reset_variables(past & prev, present & curr, float & area, float & perimeter, coordinate & C1,
coordinate & C2)
{
  curr.left = 0;
  curr.right = 0;
  curr.proportional = 0;
  curr.integral = 0;
  curr.derivative = 0;
  curr.initial_ang = 0;
```

```
  curr.final_ang = 0;
  curr.IR = 0;

  prev.left = 0;
  prev.right = 0;
  prev.proportional = 0;
  prev.time = 0;
  prev.IR = 0;

  area = 0;
  perimeter = 0;

  C1.x = 0;
  C1.y = 0;
  C1.encoder = 0;
  C2.x = 0;
  C2.y = 0;
  C2.encoder = 0;
  nMotorEncoder[motorA] = 0;
  nMotorEncoder[motorD] = 0;
}

// -------------------- Draw Angle Functions -------------------------------

void Draw_Corner(past & prev, present & curr)
{

      int data_size = 0;
      wait_read();
      configure_sensors();
      calibrate_gyro();
      map_corner(data_size, prev, curr);
      wait_read();
      set_pencil(true);
      calibrate_gyro();
      trace_corner(data_size);
      set_pencil(false);
}

void map_corner(int & data_size, past & prev, present & curr)
{
  time1[T1] = 0;
  time1[T2] = 0;
  int index = 0;
  int index1 = 0;
  int turn_mode = 0;
```

```
 float perimeter = 0;
 float area = 0;

 float origin_tol = 0;
 bool roomFinished = false;

 coordinate C1, C2, C3;
 C3.x = 0; C3.y = 0; C3.encoder = 0;

 reset_variables(prev, curr, area, perimeter, C1, C2);

 while (!getButtonPress(buttonEnter))
 {
   if (time1[T1] > index)

   {
    turn_mode = 0;
    curr.initial_ang = getGyroDegrees(S2) * DEG_TO_RAD;
    correct_path(prev, curr);

    if (readMuxSensor(msensor_S1_3) == 1)

    {
     drive(0, 0);
     turn_mode = 1;
     turn_left(turn_mode, area, perimeter, origin_tol, curr, C1, C2, roomFinished);
    }

    else if (readMuxSensor(msensor_S1_1) == 1)

    {
     drive(0, 0);
     turn_mode = 2;
     turn_left(turn_mode, area, perimeter, origin_tol, curr, C1, C2, roomFinished);
    }
    index = time1[T1] + MUX_TOL;
   }

   // Calculate coordinate
   if (time1[T2] > index1)
   {
       curr.final_ang = getGyroDegrees(S2) * DEG_TO_RAD;
       C3.x = C1.x;
       C3.y = C1.y;
       C3.encoder = C1.encoder;
    roomFinished = coordinate_process(turn_mode, curr, area, perimeter, origin_tol, C1, C2);

            if(fabs(atan2(C2.y, C2.x) - atan2(C3.y, C3.x)) * RAD_TO_DEG > ANG_TOL ||
getButtonPress(buttonEnter))
```

```
            {
              data[data_size].x = C2.x;
              data[data_size].y = C2.y;
              data_size++;
            }
               index1 = time1[T2] + TIME_GAP;
            }
      }
      drive(0,0);
      while(getButtonPress(buttonEnter))
      {}
}


void trace_corner(int & data_size)
{
      float angle = 0;
      float dist = 0;

  for(int index = 0; index < data_size; index++)
  {
       if((index - 1) < 0)
       {
            angle = atan2(data[index].x, -data[index].y) * RAD_TO_DEG -90;
            dist = sqrt(pow(data[index].x, 2) + pow(data[index].y, 2));
       }
       else
       {
            angle = atan2((data[index].x - data[index - 1].x), (-data[index].y + data[index - 1].y)) *
RAD_TO_DEG -90;
            dist = sqrt(pow(data[index].x - data[index - 1].x, 2) + pow(data[index].y - data[index - 1].y,
2));

       }
       rotate_gyro(TURN_DRIVE, angle);
       drive_dist(TRACE_DRIVE, dist);
  }
}



void rotate_gyro(float motor_pow, float ang)
{
      if(ang > SensorValue[S2])
      {
            drive(motor_pow, -motor_pow);
            while(SensorValue[S2] < ang)
```

```
            {}
        }
        else
        {
            drive(-motor_pow, motor_pow);
            while(SensorValue[S2] > ang)
            {}
        }

            drive(0,0);
}


void set_pencil(bool action)
{
    if (action)
    {
        int target = nMotorEncoder[motorC] + PENCIL_INCREMENT;
        motor[motorC] = PENCIL_POWER;
        while (nMotorEncoder[motorC] < target)
        {}
        motor[motorC] = 0;
    }
    else
    {
        int target = nMotorEncoder[motorC] - PENCIL_INCREMENT;
        motor[motorC] = -PENCIL_POWER;
        while (nMotorEncoder[motorC] > target)
        {}
        motor[motorC] = 0;
    }
}

void wait_read(void)

{
    while (getButtonPress(buttonEnter))

    {}
    while (!getButtonPress(buttonEnter))

    {
      displayBigTextLine(3, "Press enter ");
      displayBigTextLine(5, "when you're ");
      displayBigTextLine(7, "ready.");
    }
    while (getButtonPress(buttonEnter))
```

```
    {}
}


// ---------------------------- IR FUNCTIONS ---------------------------------------
// ---------------------------------------------------------------------------------

void Remote_Area_Perimeter(float & area, float & perimeter, past & prev, present & curr)
{
 time1[T1] = 0;
 time1[T2] = 0;
 time1[T3] = 0;
 nMotorEncoder[motorA] = 0;
 resetGyro(S2);
 int index = 0;
 int index1 = 0;
 configure_sensors();
 bool roomFinished = false;

 coordinate C1, C2;

 reset_variables(prev, curr, area, perimeter, C1, C2);

 float origin_tol = 0;

 while(!roomFinished && !getButtonPress(buttonEnter))
 {
      if (time1[T1] > index)
      {
        // can be implemented in future
        //correct_IR(prev, curr);
        index = time1[T1] + IR_TIME_TOL;
       }

  while(getIRRemoteChannelButtons(S4, 1))
  {
      remote_drive();
      curr.final_ang = getGyroDegrees(S2) * DEG_TO_RAD;
      if (time1[T2] > index1)

      {
        if (fabs(area) > ORIGIN_SHIFT)

            origin_tol = ORIGIN_FACTOR * sqrt(fabs(area) - ORIGIN_SHIFT);
        roomFinished = coordinate_process(IR_TURN_MODE, curr, area, perimeter, origin_tol, C1, C2);
        index1 = time1[T2] + TIME_GAP;
      }
  }
```

```
  drive(0, 0);
 }
 area = (pow(fabs(area), IR_AREA_OFFSET_FACTOR)) * CM2_TO_M2;
 perimeter = (pow(fabs(perimeter + origin_tol), IR_PERIM_OFFSET_FACTOR)) * CM_TO_M;
 drive(0, 0);
 mission_possible();
}



void remote_drive()
{
    int left_power = 0, right_power = 0;

   if (getIRRemoteChannelButtons(S4, 1) == 5)
   {
      left_power = IR_DRIVE_MOTOR_POWER;
      right_power = IR_DRIVE_MOTOR_POWER;
   }
   else if (getIRRemoteChannelButtons(S4, 1) == 8)
   {
      //left_power = -IR_DRIVE_MOTOR_POWER;
      //right_power = -IR_DRIVE_MOTOR_POWER;
   }
   else if (getIRRemoteChannelButtons(S4, 1) == 6)
   {
      //left_power = IR_DRIVE_MOTOR_POWER;
      //right_power = -IR_DRIVE_MOTOR_POWER;
   }
   else if (getIRRemoteChannelButtons(S4, 1) == 7)
   {
      left_power = -IR_DRIVE_MOTOR_POWER;
      right_power = IR_DRIVE_MOTOR_POWER;

   }
   else if (getIRRemoteChannelButtons(S4, 1) == 1)
   {
      left_power = IR_DRIVE_MOTOR_POWER;
      right_power = 0;
   }
   else if (getIRRemoteChannelButtons(S4, 1) == 3)
   {
      left_power = 0;
      right_power = IR_DRIVE_MOTOR_POWER;
   }
   else if (getIRRemoteChannelButtons(S4, 1) == 2)
   {
```

```
        left_power = -IR_DRIVE_MOTOR_POWER;
        right_power = 0;
    }
    else if (getIRRemoteChannelButtons(S4, 1) == 4)
    {
        mission_possible();
        //left_power = 0;
        //right_power = -IR_DRIVE_MOTOR_POWER;
    }

    if (left_power != motor[motorD] || right_power != motor[motorA])
        drive(left_power, right_power);
}


void correct_IR(past & prev, present & curr)
{

  // PID control function adjusting to motor power ratings
  float control_signal = 0;
  float magnitude_of_power = 0;
  float error = 0;

  // Calculate the change in time (in milliseconds)
  float curr_time = time1[T1];
  float delta_time = curr_time - prev.time;
  prev.time = curr_time;

  // Calculate info on the error/time graph
  error = getIRBeaconChannelDirection(S4, 2);
  curr.proportional = fabs(error);
  curr.integral = (curr.integral + curr.proportional) / delta_time;
  curr.derivative = (curr.proportional - prev.proportional) / delta_time;
  prev.proportional = curr.proportional;

  control_signal = IR_CONST_PROPORTIONAL * curr.proportional +
                   IR_CONST_INTEGERAL * curr.integral +
                   IR_CONST_DERIVATIVE * curr.derivative;
  magnitude_of_power = fabs(control_signal);

  if (nMotorEncoder[motorB] >= IR_RESET_ANGLE || nMotorEncoder[motorB] <= -
IR_RESET_ANGLE)
        reset_IR();

  if (magnitude_of_power > IR_POWER_MAX)
  {
        magnitude_of_power = IR_POWER_MAX;
```

```
  }

  if (error > 0 && error > IR_ERROR_TOL)
  {
       curr.IR = magnitude_of_power;
  }
  else if (error < 0 && error < -IR_ERROR_TOL)
  {
    curr.IR = -magnitude_of_power;
  }

  if (prev.IR != curr.IR)

  {
    motor[motorB] = curr.IR;
  }
  prev.IR = curr.IR;
}

void reset_IR()
{
      if (nMotorEncoder[motorB] >= IR_RESET_ANGLE)
            motor[motorB] = -IR_POWER_MAX;
      else if (nMotorEncoder[motorB] <= -IR_RESET_ANGLE)
            motor[motorB] = IR_POWER_MAX;

      while (abs(nMotorEncoder[motorB]) > IR_RESET_TOL)
      {}
      motor[motorB] = 0;
}


void mission_possible()
{

  //        100 = Tempo
  //          6 = Default octave
  //    Quarter = Default note length
  //        10% = Break between notes
  //
  playTone(  880,    7); wait1Msec(  75);  // Note(D, Duration(32th))
  playTone(  933,    7); wait1Msec(  75);  // Note(D#, Duration(32th))
  playTone(  880,    7); wait1Msec(  75);  // Note(D, Duration(32th))
  playTone(  933,    7); wait1Msec(  75);  // Note(D#, Duration(32th))
  playTone(  880,    7); wait1Msec(  75);  // Note(D, Duration(32th))
  playTone(  933,    7); wait1Msec(  75);  // Note(D#, Duration(32th))
  playTone(  880,    7); wait1Msec(  75);  // Note(D, Duration(32th))
```

```
playTone(  933,    7); wait1Msec(  75);  // Note(D#, Duration(32th))
playTone(  880,    7); wait1Msec(  75);  // Note(D, Duration(32th))
playTone(  880,    7); wait1Msec(  75);  // Note(D, Duration(32th))
playTone(  933,    7); wait1Msec(  75);  // Note(D#, Duration(32th))
playTone(  988,    7); wait1Msec(  75);  // Note(E, Duration(32th))
playTone( 1047,    7); wait1Msec(  75);  // Note(F, Duration(32th))
playTone( 1109,    7); wait1Msec(  75);  // Note(F#, Duration(32th))
playTone( 1175,    7); wait1Msec(  75);  // Note(G, Duration(32th))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1398,   14); wait1Msec( 150);  // Note(A#, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone(  784,   14); wait1Msec( 150);  // Note(C, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1047,   14); wait1Msec( 150);  // Note(F, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone( 1109,   14); wait1Msec( 150);  // Note(F#, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1398,   14); wait1Msec( 150);  // Note(A#, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone(  784,   14); wait1Msec( 150);  // Note(C, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(    0,   27); wait1Msec( 300);  // Note(Rest, Duration(Eighth))
playTone( 1047,   14); wait1Msec( 150);  // Note(F, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone( 1109,   14); wait1Msec( 150);  // Note(F#, Duration(16th))
playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
playTone( 1398,   14); wait1Msec( 150);  // Note(A#, Duration(16th))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(  880,  108); wait1Msec(1200);  // Note(D, Duration(Half))
playTone(    0,    7); wait1Msec(  75);  // Note(Rest, Duration(32th))
playTone( 1398,   14); wait1Msec( 150);  // Note(A#, Duration(16th))
playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
playTone(  831,  108); wait1Msec(1200);  // Note(C#, Duration(Half))
```

```
  playTone(    0,    7); wait1Msec(  75);  // Note(Rest, Duration(32th))
  playTone( 1398,   14); wait1Msec( 150);  // Note(A#, Duration(16th))
  playTone( 1175,   14); wait1Msec( 150);  // Note(G, Duration(16th))
  playTone(  784,  108); wait1Msec(1200);  // Note(C, Duration(Half))
  playTone(    0,   14); wait1Msec( 150);  // Note(Rest, Duration(16th))
  playTone(  932,   14); wait1Msec( 150);  // Note(A#5, Duration(16th))
  playTone(  784,   14); wait1Msec( 150);  // Note(C, Duration(16th))
  return;
}
```