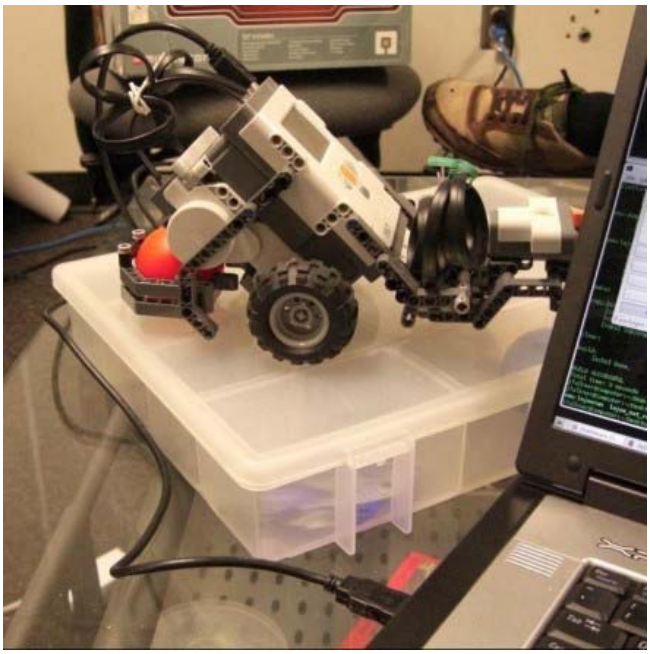


FINAL PROJECT

Final report for the Final Project including all of the Milestone from 1 to 6.

TEAM 5 MEMBERS:

Phuoc Nguyen, Khoa Tran, Corey Short, Trevor Davenport

Professor:

Roger Glassy – Fall 2013

Table of Contents

Corey Short, Trevor Davenport, Khoa Tran, Phuoc Nguyen

Team 5

Professor Glassey

IEOR 140

6 December 2013

Bomb Capture Team 5 Final Report and Analysis

1. Introduction: Prototype of Bomb Disposal Robot

Plainly, the goal of this final project and the planning and implementation of the previous milestones that lead up to this demonstration is to place our NXT robot onto a location in a coordinate grid, the hallway outside 1173 Etcheverry Hall, have our robot locate its position in the grid based on its relative location to two light beacons located on the north and south ends of the hall, to have our robot search for and retrieve a bomb that is to be detected by an implementation we decided to use, and to navigate several complicated mazes of walls, one of them contains the bomb, s.t. we map the walls the robot detects onto a general user interface (GUI) that is communicating with and sending commands, in our case, messages, back and forth between the personal computer (PC) and the NXT robot, t, which is short for Terminator, as we have named him.

1.1. Name of the assignment, due date:

Final Project, December 6th 2013

1.2. Lab team number:

Phuoc Nguyen, Khoa Tran: NXT Side

Corey Short, Trevor Davenport: PC Side

1.3. Approximate number of person hours spent on the project

150 Hours

1.4. Source Code/JavaDoc Location

NXT: Steam/Team5/ Final Project (All Milestones)/Final/Codes/FinalProject-NXT/

PC: Steam/Team5/ Final Project (All Milestones)/Final/Codes/FinalProject-PC/

JavaDoc: Steam/Team5/ Final Project (All Milestones)/Final/JavaDoc/

1.5. Distribution of work

Every person on our team put forth much effort to see this project to its completion. The combined number of hours we spent collectively is around 150 for this project. Phuoc and Khoa were responsible for implementation of much of the NXT code. Corey and Trevor for the PC. Although there was much collaboration and work done by all individuals in regards to design and testing both halves of the code.

2. Which of the performance specifications you robot meets

Our robot meets all of the requirement for the Final Project, it will traverse through the maze, locate the bomb and bring the bomb back to the starting point.

3. Hardware Design

Our robot design is very similar to other teams and spec that was provided for the milestone 6 bomb retrieval strategy with a few modifications.

- Wires - In order to prevent our wires from dragging on the ground or from sticking out and disrupting bomb retrieval, we zip-tied them underneath the NXT and above the wheels. We noticed in previous projects that the wires were too long and this was slowing our robot down during travel. We also noticed during bomb retrieval testing that the

wires actually hit the bomb and prevented us from sufficiently retrieving it. Zip-ties were the perfect solution. See picture in Appendix 9.1.

- Added Stability - Our robot is a beneficiary to numerous stability additions.
    - Scanner head - We added additional Legos and connectors to the opposite side of the axle connecting the motor to the scanner heads. In final project testing, this single modification greatly reduced the "pivot sway" during fix and mapping commands which resulted in more accurate pings. We noticed the difference in the mappings provided back to our GUI. i.e. map left was not as "noisy" and get echo was more accurate. See pictures in Appendix 9.2.
    - Frame support - Additionally, in order to achieve less "pivot sway", less noisy mapping on the GUI, and increased stability, we added a second axle, more of a sway-bar really, to right of the axle connecting the scanner head to its motor. See picture in Appendix 9.3. We also did this to reduce the weight-bearing on the bomb retrieval hardware and on the front of the robot, so we added some support underneath the actual NXT hardware. See picture in Appendix 9.4.
    - Miscellaneous additions - For further robot reliability and stability, we added numerous connectors on different components; and more specifically, we added connectors from the outside of the robot's tires to the end of the axle underneath the robot. We did this so our wheel placement would not move; thus, affecting alignment and overall drivetrain capabilities. See pictures in Appendix 9.4.
- Plastic cover - After talking to the professor and other students, we determined we needed to cover the glass on the NXT hardware s.t. the reflection of the glass was not

detected by the ultrasonic sensor during scans. As a result of doing this, we noticed more accurate fix positions. See picture in Appendix 9.5.

- Bumper design - We decided to use one of the smaller bumper designs s.t. if there was a chance we were going to hit a wall or miss a wall, we would rather miss it and rely on our more efficient mapping skills. See picture in Appendix 9.6.
- Broken Pieces - A key Lego piece for our bomb retrieval hardware was broken. This was causing the bomb retrieval hardware to have an improper alignment when attached to the robot, so we used a pair of needle node pliers to straighten it. Then we actually zip-tied the broken Lego piece to its non-broken counterpart, and we applied a generous amount of JB Weld epoxy adhesive to strengthen and stiffen the broken Lego piece. This fixed our bomb retrieval hardware alignment. See picture in Appendix 9.7.
- Magnet placement - Originally, we used some duct tape to secure our magnet. It was probably the poor bomb retrieval hardware alignment prior to our fix, but we decided to use some of the same JB Weld epoxy to secure our magnet anyway. Bomb retrieval has worked flawlessly ever since. See picture in Appendix 9.7.
- Batteries - Our robot has always seemed to be a bit buggy. The noise decides to work one moment, and it does not work for a week the next. However, we noticed in previous projects that our robot always seemed to work better on batteries, so we used our own batteries during final testing and for the final demonstration.

4. Experimenting Works

4.1. Experiment description and purpose with Data

Milestone 1 Analysis

As trivial as the first milestone might have seemed, we actually ran into problems on several occasions where our robot would not travel in as straight of a line as we needed it to, and our robot would over-rotate; thus, causing inaccurate travel during map left or map right and inaccurate robot heading location reporting from the NXT to the PC GUI. These are both problems that led to us having problems optimizing our fix position for later milestones as well. Initially, we were unaware that the wheel diameter was different for each wheel. After realizing this and through measurement and much trial and error testing, we were able to fix both the travel and rotate functionality of our robot, Terminator.

Left wheel diameter: 5.23 centimeters.

Right wheel diameter: 5.22 centimeters.

Track width: 13.15 centimeters.

When we first completed this milestone, we noticed that our travel and rotate was not perfect. But it was however within the given stated bounds for error. In any event, this error ended up being too high and during testing phases for the final project, if we did not fix Terminator's position before our standard deviation became too high, our robot could not properly locate itself. During testing, if this happened to us, we were required to pick up the robot and to reset the current robot position from the GUI. This would have been extremely problematic for completing the bomb location and removal because errors continually accumulate, and it would have been very costly during the final, as in costing twenty-five points for a rescue mission to move the robot.

One last set of problems here, we also had a problem where our acceleration, travel, and rotate speeds were all set too high. The over-acceleration caused Terminator to "burn-out", applying too much torque to the wheels before an adequate travel speed had been reached. Through discussion with Glassey and trial and error, we deemed the following speeds to be correct and work for all moving or stopping instances performed for the bomb location and retrieval:

DifferentialPilot acceleration: 40 wheel diameter units per second^2.

DifferentialPilot travel speed: 20 wheel diameter units per second.

DifferentialPilot rotate speed: 100 degrees per second.

Milestone 2 Analysis

In Milestone 2, we were able to determine our scanner accuracy through experimental data. The purpose of this milestone was to successfully modify and test the Scanner class to return an array of Beacon bearings. The purpose of this project is to create a basis for our final project which includes being able to detect a bomb.

Part 1: We place the robot at two different location (20, 30) and (210, 240). The scanner will scan through a range of 200 degrees, and it will find a bearing for the light beacons. Because the light scanner sweeps twice, we will actually have two values for each of the beacon bearings. Then, we take the average of those two values. The purpose of this part is to measure the bearing to each beacon. See data in Appendix 9.8.

Part 2: We place the robot at three different locations, 30 cm, 90 cm, and 180 cm, facing the Ox direction. At each location, the robot records two values from the ultrasonic sensor from the south end of the hall to the north end. This calculation adds up and gives the robot the total width of the hall. The purpose of this was to make sure the robot records the right value for the

distance between the north and south ends of the hallway. Also, it was to make sure the distance calculation is correct and accurate. See data in Appendix 9.9.

Part 3: We place the robot at two difference locations (20, 30) and (210, 240). For each of the locations, we record the bearings for each beacon eight times. The purpose of this experiment was to make sure we are achieved an accurate and precise value for the same location after numerous scans. See data in Appendix 9.10.

When using the coordinates (20, 30) we found that there was a difference between three to four degrees. When using the coordinates (240, 210) the difference was between one to four degrees.

Milestone 3 Analysis

Milestone 3 was a critical milestone in which we were able to obtain a small locator standard deviation in our x, y, heading, and bearing difference. This was instrumental in obtaining an accurate fix position in later tests. From this test, we are able to calculate the standard deviation to use for our VariancePose class in later milestones. See data in Appendix 9.11.

Milestone 4 Analysis

Milestone 4 was the development of our MissionControlGUI for the PC control and all our NXT code for robot communications. On the NXT side, we decided to create a data type called Message that would hold an enum list of commands between the NXT and PC code called MessageType and an ArrayList of floats of MessageType's corresponding values. In order to easily add subsequent commands to follow this milestone, we used a switch statement and a method to get the message type. Messages received by the robot's communicator are sent to its controller where the message is decoded and executed. The result of the executed message is sent

to the PC's communicator called GridControlCommunicator. On the PC side, the message received is decoded and the data is read in and sent to MissionControl for handling. For instance, if Terminator's pose is sent back to GridControlCommunicator then a MessageType of POS_UPDATE is received and methods to update the GUI's x and y data fields, its coordinate list, and a method to draw the robot path are called. It was the development of this exchange of commands between the NXT and PC code that allowed us to efficiently test bomb retrieval during milestone 5 and milestone 6.

Milestone 5 Analysis

Milestone 5 was the first milestone we tested mapping walls on the GUI. Many of the problems that challenged us throughout this milestone have already been address above. It should be noted that in order to send back the standard deviation from the robot to the PC, we changed all of our Pose's in Controller and Locator to VariancePose's s.t. we wrote a method in Controller that gets the current variance of x, y, and the heading, takes the square root of each of them, and sends the value to the PC. We reset the variances anytime a fix position is calculated. If we did not do this then the respective standard deviation would increase and never reset.

Part 1: The standard deviation is big for X because the location (0, 0) and (241, 0) cannot be reached. We have to estimate our first starting point and calculate from that point. In order to prevent the error from building up, we try to fix position along the way so make the calculation more accurate. See data in Appendix 9.12.

Part 2: We added more enum types to our MessageType. See map in Appendix 9.13.

Milestone 6 Analysis and Dead Reckoning

Milestone 6 is the milestone that merges all the aforementioned milestones together. We spent exorbitant hours inside and outside lab testing and making small optimizations for Dead

Reckoning. However, all our effort would not bear fruit without the implementation of the Grab Bomb button in MissionControlGUI and the locateTheBomb() method in Scanner. In testing, our map of the grid and bomb detection was so accurate that we were able to locate and retrieve the bomb without the use of an auto-detector. However, as a team, we felt this would be insufficient and worrisome if we did not implement something more concrete because there were times when we thought we were very close to the bomb, and we were, yet we were unsuccessful in retrieval. Additionally, we were victim of several instances of the bomb not being attached to our magnet well enough, by manual retrieval, such that we dropped the bomb while exiting the maze toward the drop point. To ease our minds and solve our problems, we implemented a method in the NXT's Controller called grabTheBomb(). In order to test this, we placed a bomb on the desk in the hallway without any walls around it. We verified our optimal range for scanning by using Get echo, and played around with the limitAngle. When we had this working, we added the walls around the bomb and realized that our scanner was having some problems with retrieval. Our tactic was to have the user on the PC end get within twenty or thirty centimeters of the bomb and press the Grab Bomb button. Originally, the robot would do a scan for the bomb rotate about 180 degrees and do another scan. This was the problem we faced. During our second scan, the robot was facing away from the bomb; and as a result, we noticed a large drop in scanner accuracy. We realized this when we performed a get echo of 180 while the robot was facing away from the bomb. When the scanner head is completely turned around, it is not facing a "true" 180 degrees in comparison to when its heading is zero, which is very accurate. See picture in Appendix 9.16. Therefore, we opted to ditch the second scan entirely since the user is able to detect and get close enough to the bomb. After a bit more testing, using one scan in front of the bomb and reversing toward it worked flawlessly even when the robot is detecting the bomb from a very wide angle.

Additionally, using the scanner to detect and to retrieve the bomb eliminated the bomb-drop problem we were having with manual retrieval.

- We coded a simple tester where our robot locates the bomb x centimeter away, travels over to pick it up, and comes back to its original location. This is to demonstrate that the accuracy of our robot will be reliable in performing the end goal. This test proved to be quite useful in the preliminary stages of our retrieval plan. For example, collectively, we noticed that accurately backing up into the bomb was one of the most difficult aspects of this milestone.

- After changing some code around, our robot was not as accurate as before. In order to compensate for the lack of accuracy, we chose to recalculate the variance and standard deviations by performing around eleven different pose checks. After collecting the information, we were reassured that our original standard deviation and variance calculations were, in fact, correct. After reverting our code back, we found that our problem was something much more trivial. We now implement the VariancePose and VariancePoseProvider in order to get a better, more accurate result. We tested the robot at different locations facing various headings to get the standard deviation based on that. Then we take the average of all the standard deviation and come up with our final standard deviation for x, y and heading. Based on those calculations, we parse them in to VariancePose to use. The Controller also uses the values varX, varY, varH and send them back to the PC side. PC will take those value and display it in the GUI. See simple maze in Appendix 9.14 and the more complicated maze in Appendix 9.15.

- Dead Reckoning

Dead Reckoning proved to be quite the challenge for us. Our fix position was not working optimally as it had in all of our previous bomb retrieval runs and testing so we had to rely a bit more on pinging our surroundings since our heading was slightly off. This worked because we

were able to identify and capture the bomb successfully; however, our time was adversely affected because of the need to send additional pings. Also, during a critical scanning point on the map, near a wall towards the bomb, before we were able to fix our position, some people walked by, that we mapped to the GUI, so this also gave us a degree of uncertainty when moving between the walls towards the bomb and back to the safe drop point. See map in Appendix 9.20.

4.3. Calculations and analysis

The hardest calculation of this project is to calculate the fixPosition function, in which based on the two beacon bearings and current known position, we update our current location to the most accurate location based on the all the information that we have. Our method was:

Part 1: The angle is calculated based on the basic trigonometry from the illustration below: See picture in Appendix 9.16.

Point (20, 30):

1. tan(a1) = 30/20 ===> arctan(a1) = 56 degrees
2. tan(a2) = (241-30)/20 ===> arctan(a2) = 84 degrees ===> -84 degrees

Point (240, 210):

1. tan(a1) = 210/240 ===> arctan(a1) = 41 degrees
2. tan(a2) = (241-210)/240 ===> arctan(a2) = 7.3 degrees

Unfortunately, the values that we got never were accurate. We have tried with different motor speeds range from 10-100. The reason is because at first location (20, 30), we can get the right value for the left beacon bearing because it is right next to the robot (a slightly different value). However, for the right beacon, the light sensor takes more time to get the value due to the distance from the robot to the right beacon. The same reason can be applied to the scan at

location (210, 240) too. Because that point is far away from the robot so the reading needs a lagging time in order to travel back to the robot, but at the time that light sensor is at another angle. That explains why the data we got never be accurate.

Part 2: After getting the distance from both side by using Ultrasonic detector, we add the values up to get the width of the hall. The values that we got are slightly different from the actually length (1 centimeter off). The reason is because we are not sure where the exact point that the ultrasonic sends out the signal is. Also when it turns to another side, the point that it starts sending signal again maybe a little off from the one that it sent signal previously. That is why we did not get the accurate values for the distance when adding the two values up.

Part 3: We do not have the accurate values for the bearings. The reason is as same as Part 1, when the light sensor sends and receives a signal, there is a lag time between the two actions so it may be a slightly different result.

4.4. How results were used in your code

With the value that we get back from the fixPosition, along with the VariancePose class that takes care of the Variance, we can update our current position. Moreover, with the Standard Deviation that we get back from the NXT, we will determine whether we should fix location or not, and we can also see how further we are form the actual point.

5. Class Collaboration Analysis and Bomb Retrieval Strategy

The following is an explanation of class collaboration between the PC code and the NXT code starting with methods that are called when a button is pressed on the PC's MissionControlGUI and after the NXT code has been loaded onto Terminator and it has already established a valid Bluetooth connection:

To begin, our strategy is to first send out a ping and get the distance back via an echo in front of the robots current heading which is mapped as a magenta-filled oval on the PC's GUI. However, for the sake of explaining class collaboration, we start with what happens when setting the initial pose. In testing, we noticed that the scanner head and the rear turn wheel, if not set as straight as possible in the robot's current heading, would cause inaccurate mapping on the GUI, and it would cause an inaccurate fix position. In particular, if the rear turn wheel is e.g. ninety degrees from the robots current heading when setting the position of the robot, when performing a map left, map right, or travel, immediately after, Terminator would not travel in a straight line and this made determining the robot's location accuracy via mapping its current pose on the GUI and by a fix position much less accurate.

5.1. Mission Control GUI Design

The MissionControlGUI is designed to provide us with an accurate location and heading of our robot Terminator while allowing the user to assist and guide our robot to successfully map the coordinate plane consisting of walls and bombs, and to do perform this in swift execution. There was lots of design planning and hours spent in implementing a GUI that we deemed optimal for the successful completion of the final test. The following are implementations we designed to make our GUI as effective and accurate as possible:

- Grid drawing - During testing of milestone 4, we noticed that after doing a fix position that our robot's actual location was not represented on the GUI like we observed its physical placement on the tiles on the hallway. Initially, we thought this error was in our fix position which we thoroughly tested to check our assertion. However, we noticed that our GUI did not account for the twelve centimeter tile on the south end of the hall. After taking some measurements and modifying the code in the PC's OffScreenDrawing, we

accounted for this tile and had ourselves a much more representative map of the hallway. So when further rigorous testing commenced, any errors would be more easily visually represented in our GUI and more representative of the data our robot was collecting.

- Coordinate list - Any time our robot is moving, a pose is sent from Terminator to the PC's GridControlCommunicator. Initially, we sent the coordinates of the pose, the mapping, and the standard deviation to our status field on the GUI. However, during testing of the bomb retrieval, it became very important when performing a get echo, which sends a ping and returns a mapped dot of the distance found, to know the current robot position. However, we had just overwritten it with our map coordinates that had been updated on to the status field and we no longer were able to perform pinpoint accurate map left or map right. So we implemented a JTextArea with a JScrollPane with an adjustment listener that automatically scrolled to the bottom of our list s.t. we would have quick and easy access to our current position, standard deviation, and any mapping coordinates sent back from the robot. This improved overall mapping effectiveness and speed.
- Auto update Data X and Data Y fields - We opted for another optimization in our GUI to enhance reliability and speed performance. When performing a map left or map right, one could simply click on the grid, which updates the GO TO coordinates of our mapping buttons. However, during testing, we noticed that using go to would cause the robot to rotate and map in that direction. Or, the robot would not rotate; but instead, it would travel to the coordinate selected, just not in a straight line. We found this produces the most inaccurate maps, as dots represented as walls on the GUI tend to drift in the direction the robot is traveling. e.g. say our robot was at location (210, 132) with heading

180 and we wanted to map left to (25, 132). After doing a fix position, our location would more realistically be something like (212.02765, 130.93036) with heading 180.034521. If we entered map left to location (25, 130), “drifting” during travel of the robot still occurred because the robot was trying to compensate for not including the trailing decimals. Thus, we were mapping really close to a straight line, but it was not as straight as it could be. And sometimes but mostly rarely, we would hit a wall because of this. Our solution that worked: typing in the full location including the float. This was exiguously slow for the user. Therefore, our optimal solution, so our heads would not explode, was to have our Data X and Data Y fields auto-update with the robot’s current position. This worked like a charm. The overall quality of our maps and testing efficiency increased greatly. Additionally, this made travel much more efficient and desirable to use and it did not affect the use of clicking directly on the map for GO TO or entering a destination manually.

- Mapping color - We use three different colors to map. Initially, we used a magenta filled oval dot all of our mapping methods. During testing, oftentimes this proved confusing when trying to remember if we had detected a wall or a bomb inside a maze. Mapping the grid was also a bit confusing sometimes because sending a ping to get back the echo and performing a map explore looked the same on the GUI. We wanted to the ability to differentiate our robot’s actions more efficiently. So we implemented the different colors as follows:
    - Map left and map right - We use a magenta filled oval dot.
    - Map explore - We use a yellow filled oval dot. More about this implementation below.

    - Get echo - We use a cyan filled oval dot.
- Crash detection - When the CRASH message type is received by GridControlCommunicator, Mission control calls OffScreenDrawing to draw the crash detected via our bumpers as a larger red outline of an oval on the GUI.
- Enter echo - During testing, it became annoying to repeatedly go back and forth from rotating to a heading and getting the echo at scanner heading zero relative to the robot. So we added an additional JTextField for the echo s.t. 0 is oftentimes left in the field, while the amount to rotate to or travel to, in the amount field, is constantly being changed.
- Rotate and Rotate to - We wanted the functionality to be able to rotate by an amount and to be able to rotate to a specified amount. So we implemented both.
- Functional difference in map buttons - Again, we wanted the most functionality to make wall and bomb detection as efficient and simple as possible. So we implemented three mapping techniques and get echo as follows:
    - Map left - Map left turns the scanner head to ninety degrees regardless of robot heading and maps from the robot's current position to the specified location either by clicking on the map to obtain the coordinates or by entering them manually via the Data X and Data Y fields.
    - Map right - Map right is identical to map left except that the scanner head rotates to negative ninety degrees before the map operation.
    - Map explore - Map explore differs from map left and map right by taking an angle as an input amount. The scanner head rotates from the positive input amount to its negative amount and sends back all the coordinates to map on the GUI. We avoid doing large scans with this mostly because it produces inaccurate

and sloppy map results. However, when used in +/- 5 or +/- 15 amounts, map explore is key in bomb detection. Additionally, we use this technique of scanning very small windows when sending a ping with get echo to a single point generates uncertainty.

- Get echo - Get echo rotates the scanner head by a specified input amount relative to the robot's current position. Functionally, this is very similar to map explore but with increased accuracy and precision. Moreover, we generally use this to do scans at heading zero, in front of our robot, to determine the near-maximum distance we can map left or map right without hitting a wall.

5.2.Robot Design

See 3/ Hardware Design for robot.

6. Software design:

- The general algorithm of this Milestone is to implement many methods that help the PC communicate with the NXT and command it to do the work. Accuracy needs to be taken into account. We have implemented some buttons that each of them is responsible for each particular movement of the report.

- To make thing simple when transmitting the data and message between Robot and PC, we use enum type for the Message. The DataStream consists of a type of enum for message such as: MOVE, TRAVEL, ROTATE, ROTATE_TO, MAP etc… and an array that contains information regarding to the message.

- In order to make the Detector work while the robot is moving and transmitting the message back and forth to the PC, we use the class SensorPortListener provided by the Lejos libraries. Our method is to implements the Detector and attach it to the main Controller which

has full ability to control everything. Each TouchSensor will be assigned to a different SensorPortListener. However, since we need to use the instances inside the Detector class and send back the message/command to the Controller, we make another Inner class DetectorListener inside the Detector which implement the SensorPortListener class. With that, whenever a left/right TouchSensor is pressed the new state will be sent back to Controller and ask the controller to stop the Navigator immediately and send back the message to the PC. Lejos provides a method to check whether the TouchSensor is touched or not, which is stateChanged. For that method, we will be able to get back the new values of the state of a specific TouchSensor (left or right). By description, when the new value is more than 1000, it's when the TouchSensor is released (not pressed). Otherwise, when the value is below 1000 its state changes either deeply pressed or lightly pressed depends on the value that we get. Therefore, depending on that value we have an if statement to send back whenever a left/right sensor is pressed.

- One algorithm we chose to implement is to input to the rover a specified amount of degrees to scan (rotateTo()). This method uses an algorithm that will take in a degree and scan positive and negative directions of that degree. By doing so, this gives us modularity and the ability to perform a wider range scan instead of simply scanning one direction or simply scanning every direction. It was our idea that by doing so, we could save time and unnecessary rotations.

- In order to scan for the beacons, we first decide which wall is nearer to the robot based on the current Y location, then that wall will be chosen to calculate other things. When the scanner is called, it will first scan and decide which angle to scan. No matter which heading it is, it will always scan from -90 to 90 in the heading that the 2 lights are. Then the bearings to the angle will be returned to the locator and those data will be calculated for the current location.

Based on the calculated value of X, Y coordinate, we can compare it to the right value to determine how accurate our robot is.

6.1. Class Dependency Diagram:

All the pictures of classes can be found in the Final Projects folder on steam. See Appendix 9.17. for the NXT class diagram and Appendix 9.18. for the PC class diagram.

6.2 Classes and Responsibilities

The Milestone 6 class is the main class that gathers up other classes that are used to finish this Milestone. Including these main classes:

- Communicator: This is the main communication center between the robot and the PC. It initializes the connection between the PC and NXT brick, also creates DataStreamIn and DataStreamOut in order to communicate between PC and NXT. In order to make the communication simple, we make other classes called Message and MessageType which implements the enum variables to transmit the message.
- Controller: this is the heart of the NXT. It controls all the message transmitted between robot and PC. When the communicator received a message, it is sent to the Controller to decode and execute it.
- Locator: This class will use a Pose object to save the current location and heading of the robot. Then the method locate() is called, it will calculate the angle to both of the walls (North and South). Based on the current Y-coordinate, if it is near the North Wall then we are going to use the calculation based on the distance to the North wall. Otherwise, we will use the value based on the South Wall. The method will then save the bearings to both beacons, along with current X and Y coordinates. All the information will be passed into FixPosition() method, it is

when the current location is calculated and saved back to the Pose object. The FixPosition() method plays a role as our calculator of the current location.

- Scanner: this class also plays an important role in this milestone. The role of it is to calculate the distance to the wall (distanceToWall() method). The scanLights() method will scan for the two lights and return their bearings to be used in Locator class.
- Detector: This class takes care of the two bumpers. By using SensorPortListener class given by LeJOS, we make a DetectorListener class that implements the SensorPortListener to work as a thread. With that being said, whenever a left or right bumper is pressed (by using the given stateChanged() method), it will stop the NXT immediately and response back to the PC with the value of where the crash was.

6.3 Important Methods in both PC/NXT side

NXT →

- touchSensorTouch() -- This method uses two Booleans, isRightTouched and isLeftTouched to determine if either sensor has been activated. This is our fail-safe implementation of reassuring we are not simply running into a wall if all else fails. Once a sensor is touched, we send the current distance and angle to our sendWall method.
- sendWall() -- The sendWall method is our bridge between the NXT side and the PC side to help our robot determine where the exact location of each wall we are drawing is located.
- sendData() -- SendData is also another one of our bridges that connects NXT data to the PC side, however, this method instead uses Booleans to differentiate between sending a Pose or a wall location.

- sendEcho() -- SendEcho is our implementation of sending and pinging the current given angle.
- sendPingAll() -- SendPingAll is our way of instead of pinging the current given angle, we simply ping the entire surrounding area. This method has proven quite useful when we get ourselves into sticky situations.
- decodeData() -- This method is one of our more important methods in terms of data abstraction. Within this method, we decode the current message, send it out, and execute accordingly. decodeData uses our ENUM class to determine the current MessageType. After determining the MessageType with a simple switch statement, we updateMessage() accordingly. updateMessage either clears the Queue if the message == STOP, or adds the current message into the Queue.

PC → Our PC Side has not changed much. However, we have changed some graphical looks of our GUI such as adding a list of previous moves on the right hand side of the graphical interface. By adding this list of moves, we now have the ability to post-navigate through correct and open locations after locating the bomb.

7. The most interesting/challenging/difficult parts of the project:

There are a lot of difficult parts in this milestone and many of our challenges have already been discussed above in previous sections.

To decide how to make the scanner always scan in the heading that has two lights, no matter what its current heading is.

To calculate the exact location of the robot. We can never get the right value for the location, mostly because when the robot rotates to another heading, the location of the axis is

changed and that makes the angles to the two beacons change. We first put the axis on the top of the point that explains why the standard deviant for all of the values are really big.

One difficult part of this milestone is to come up with the right formula to calculate the current location. Because the value is rounded, and we have to cast (float) to (int) or vice versa many times, it results in the loss of value.

Another challenging part of this milestone is the hardware. At first the threshold for light values keep changing for us, both of the lights did not have the same values so it was really difficult to code the scanner, especially when it is at a far location. Moreover, when we call rotate for both the scanner and the brick, sometimes it doesn't rotate the exact degree that we want. (e.g. say if we want to turn ninety degrees we now need to have a tuning factor in order to get to the right angle, same thing for the scanner). If we make the scanner go fast, then the value is not accurate. On the other hand, if we make it go slower, it will shake and it does not get the right value either. Our task was to find a right speed, acceleration for it in order to make it work accurately.

8. Links to source code and JavaDocs

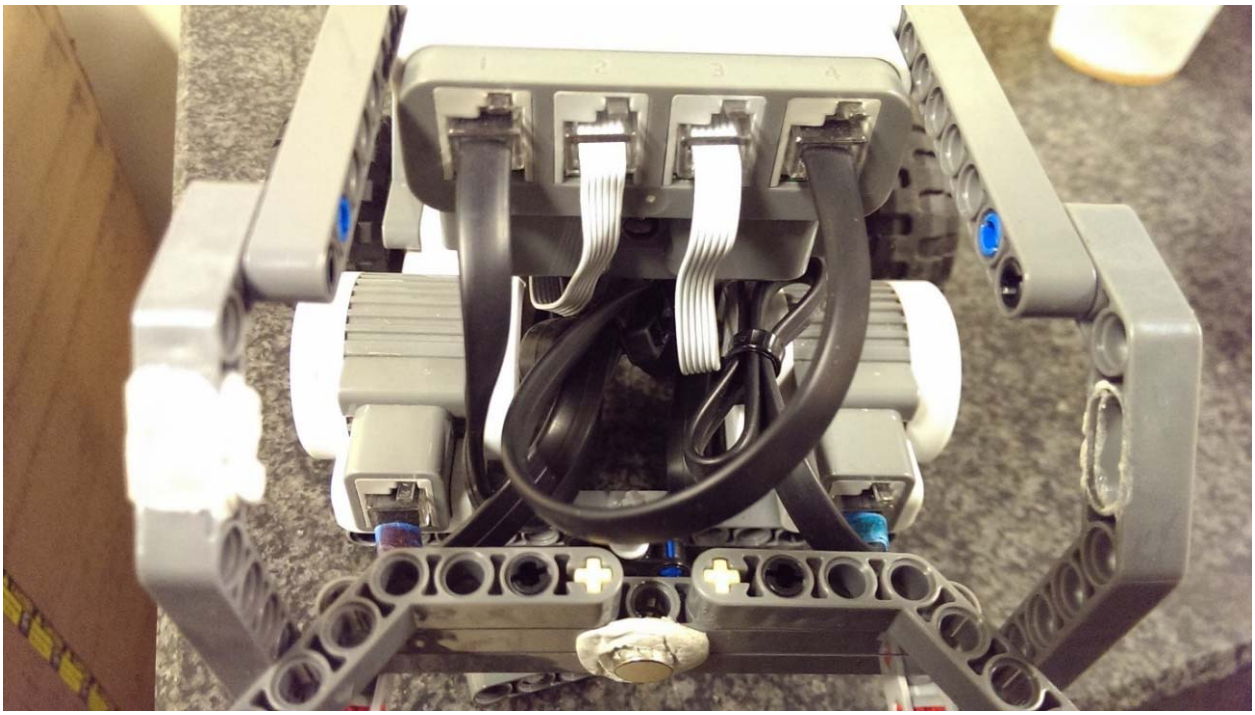
NXT: Steam/Team5/ Final Project (All Milestones)/Final/Codes/FinalProject-NXT/

PC: Steam/Team5/ Final Project (All Milestones)/Final/Codes/FinalProject-PC/

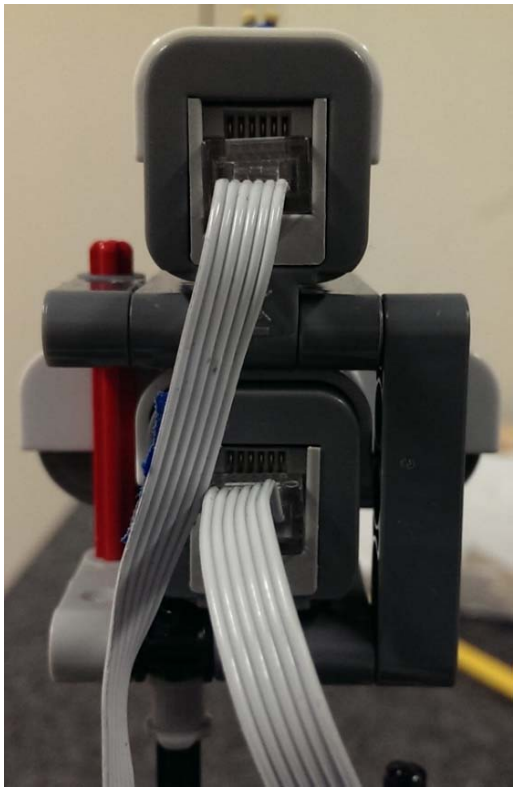
JavaDoc: Steam/Team5/ Final Project (All Milestones)/Final/JavaDoc/

9. Appendix

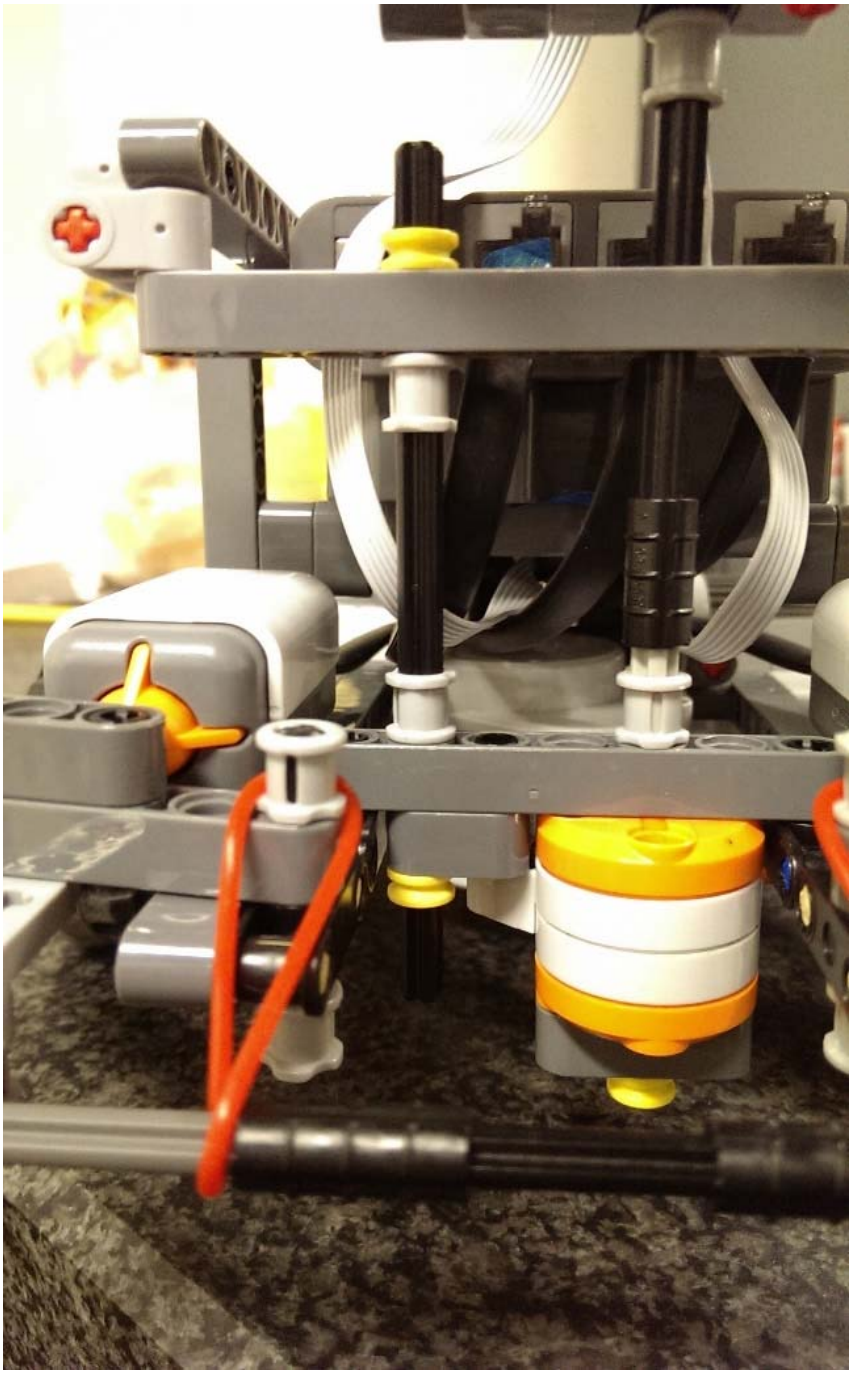
Appendix 9.1. Zip-ties



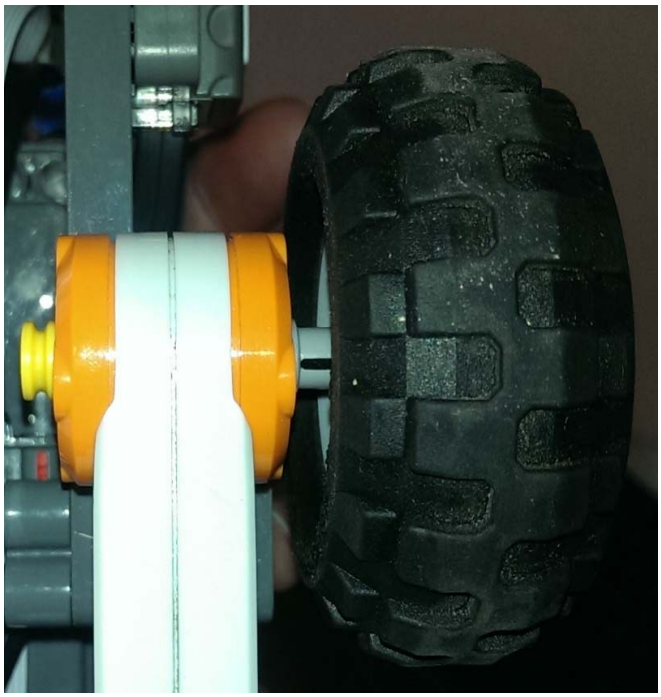
Appendix 9.2. Scanner head



Appendix 9.3. Frame support

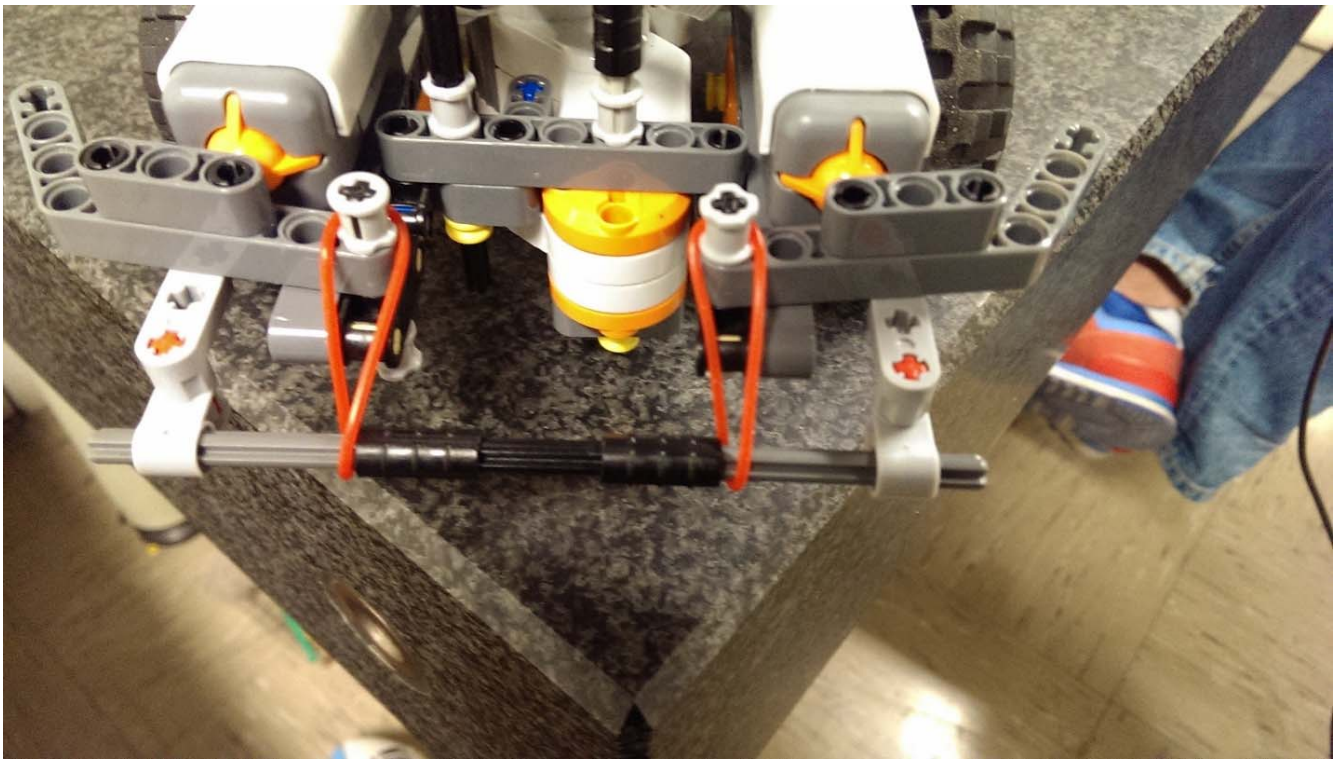


Appendix 9.4. Additional hardware

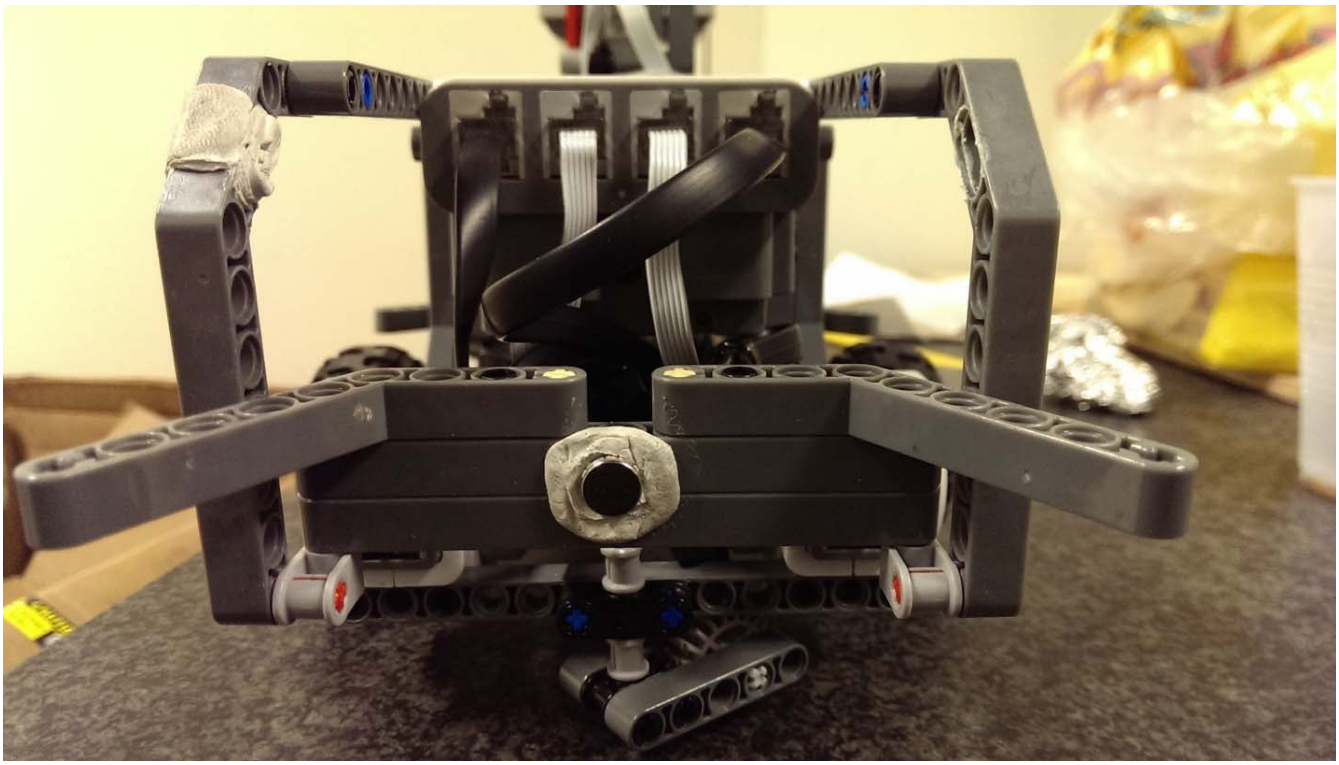


Appendix 9.5. Plastic cover

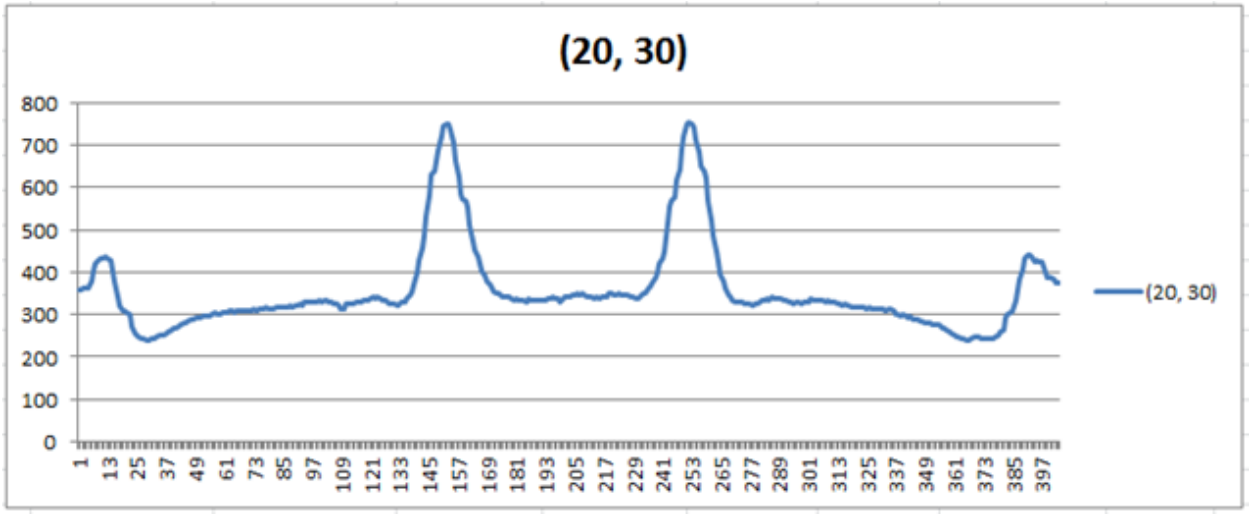
Appendix 9.6. Bumper

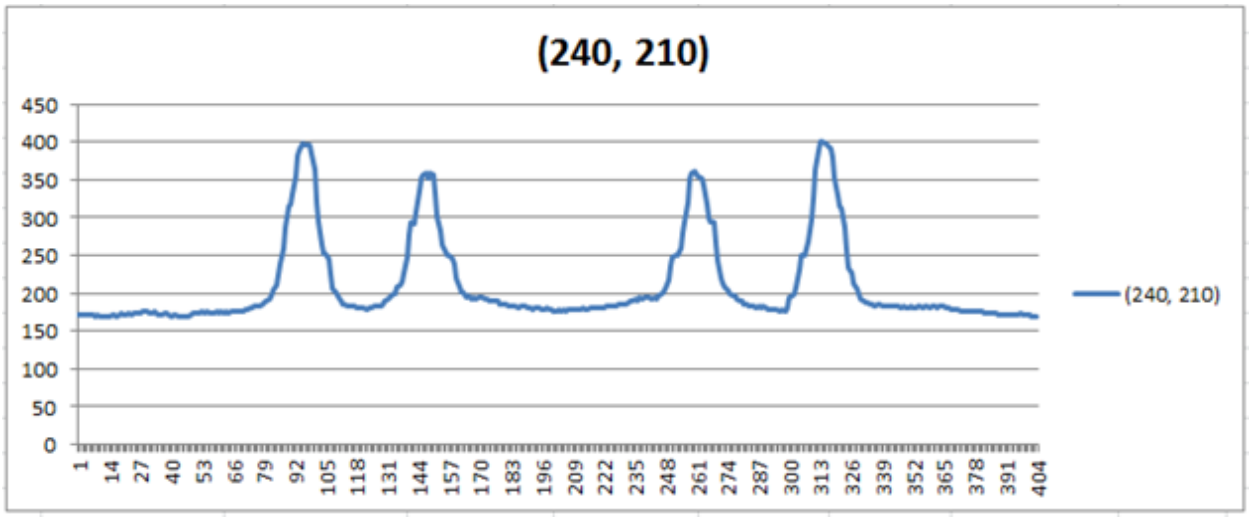


Appendix 9.7. Broken pieces



Appendix 9.8. Beacon Bearings Data from Scanner/LightSensor





Coordinate	Beacon	First Scan	Second Scan	Average	Theory	Off Degree
30, 20	Beacon 1	-89 °	-88 °	-89 °	-84 °	3.50 °
	Beacon 2 (0,0)	52 °	51 °	52 °	56 °	4.50 °
240, 210	Beacon 1	-6 °	-10 °	-8 °	-7 °	1.00 °
	Beacon 2 (0,0)	46 °	43 °	45 °	41 °	3.50 °

Appendix 9.9. Wall Distance Data from Scanner/Ultrasonic Sensor

Distance	Left wall	Right wall	Total	Theory	% Difference
30 cm	41 cm	197 cm	238 cm	241 cm	1.24 %
90 cm	119 cm	119 cm	238 cm	241 cm	1.24 %
180 cm	193 cm	45 cm	238 cm	241 cm	1.24 %

Appendix 9.10. Milestone 2 data

	Coordinate	1st scan	2nd scan	3rd	4th	5th	6th	7th	8th	Median	Theory	Off degree
Beacon 1	**20-30**	-90 °	-90 °	-90 °	-90 °	-91 °	-90 °	-90 °	-90 °	-90 °	-84 °	4 °
Beacon 2		53 °	53 °	52 °	53 °	52 °	52 °	53 °	53 °	53 °	54 °	1 °
Beacon 1	**210-240**	46 °	46 °	47 °	47 °	47 °	46 °	46 °	46 °	46 °	41 °	5 °
Beacin 2		-7 °	-7 °	-8 °	-8 °	-7 °	-7 °	-8 °	-8 °	-8 °	-7 °	1 °

Appendix 9.11. Milestone 3 data

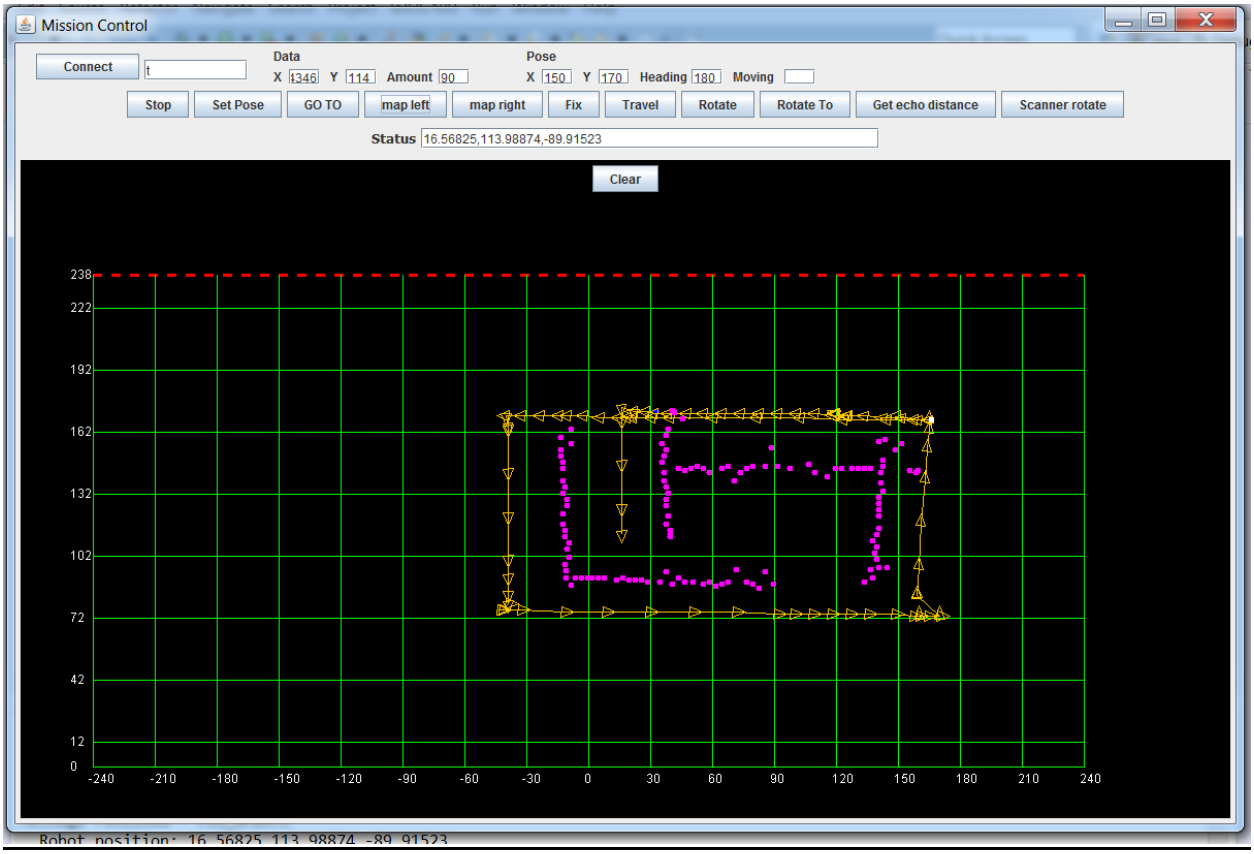
	Point (-30, 35)						Point (240, 185)					
	X	Y	Heading	Bearing1	Bearing2	Bearing Difference	X	Y	Heading	Bearing1	Bearing2	Bearing Difference
0	-24.684	36.874	0.658	-42	92	-134	234.265	178.154	3.254	177	228	-51
	-24.687	35.268	2.985	-43	93	-136	235.254	178.568	4.235	176	227	-51
	-25.425	36.458	2.358	-43	92	-135	233.325	179.542	3.254	177	227	-50
	-25.687	36.547	2.985	-42	93	-135	230.125	178.254	5.017	176	227	-51
	-25.487	36.598	1.325	-43	92	-135	230.254	178.658	4.892	177	227	-50
	-24.874	34.125	1.357	-43	93	-136	234.254	179.542	4.658	176	226	-50
	-24.125	34.158	1.125	-43	92	-135	233.254	178.254	3.387	177	227	-50
	-25.217	34.874	1.125	-42	93	-135	234.254	178.547	4.985	177	227	-50
90	-28.657	36.154	88.125	-142	-8	-134	242.254	176.584	98.548	82	133	-51
	-29.367	36.254	88.458	-142	-8	-134	242.658	176.589	98.547	81	132	-51
	-29.365	37.154	88.598	-142	-8	-134	241.257	176.658	98.365	82	132	-50
	-28.325	36.155	88.575	-143	-8	-135	243.356	177.125	100.658	81	133	-52
	-29.574	37.587	87.587	-143	-8	-135	242.658	176.657	98.369	81	132	-51
	-28.387	37.854	87.547	-142	-7	-135	242.643	176.598	100.574	81	132	-51
	-28.654	37.157	87.574	-143	-7	-136	242.368	176.654	100.258	81	132	-51
	-28.674	38.587	88.547	-142	-7	-135	243.661	177.654	100.268	81	133	-52
180	-27.254	34.124	179.872	-276	-140	-136	229.658	175.658	-175.325	-7	45	-52
	-27.584	33.541	180.541	-275	-141	-134	228.547	175.548	-175.128	-7	44	-51
	-28.125	33.254	179.654	-275	-140	-135	227.658	174.268	-174.687	-6	45	-51
	-27.597	33.578	179.584	-276	-141	-135	229.658	174.689	-175.687	-6	44	-50
	-28.658	32.598	180.598	-275	-140	-135	225.135	174.012	-169.657	-6	45	-51
	-27.124	34.157	180.245	-276	-141	-135	225.684	174.697	-171.658	-7	44	-51
	-27.687	33.254	179.568	-276	-141	-135	229.658	174.214	-174.687	-7	44	-51
	-28.125	33.322	179.265	-275	-141	-134	228.365	174.235	-174.748	-7	44	-51
-90	-24.125	33.215	-88.125	46	181	-135	234.325	174.485	-84.658	-97	-46	-51
	-25.114	32.254	-87.265	45	181	-136	231.148	172.125	-77.365	-97	-46	-51
	-24.367	32.125	-87.365	46	181	-135	231.578	172.670	-76.456	-98	-47	-51
	-25.256	33.125	-88.214	46	182	-136	234.256	175.215	-84.325	-98	-47	-51
	-24.368	32.257	-86.012	45	181	-136	235.665	174.264	-84.874	-98	-46	-52
	-24.125	32.658	-86.325	46	181	-135	235.335	174.568	-84.365	-98	-47	-51
	-24.259	32.657	-87.287	46	182	-136	234.328	174.586	-84.325	-98	-47	-51
	-24.325	32.598	-88.365	45	180	-135	234.328	174.658	-84.257	-98	-47	-51
Mean	-26.540	34.704				-135.06	234.411	176.060				-50.906
SD	1.920826	1.948211				0.67	5.529035	1.976779				0.59

Appendix 9.12. Milestone 5A

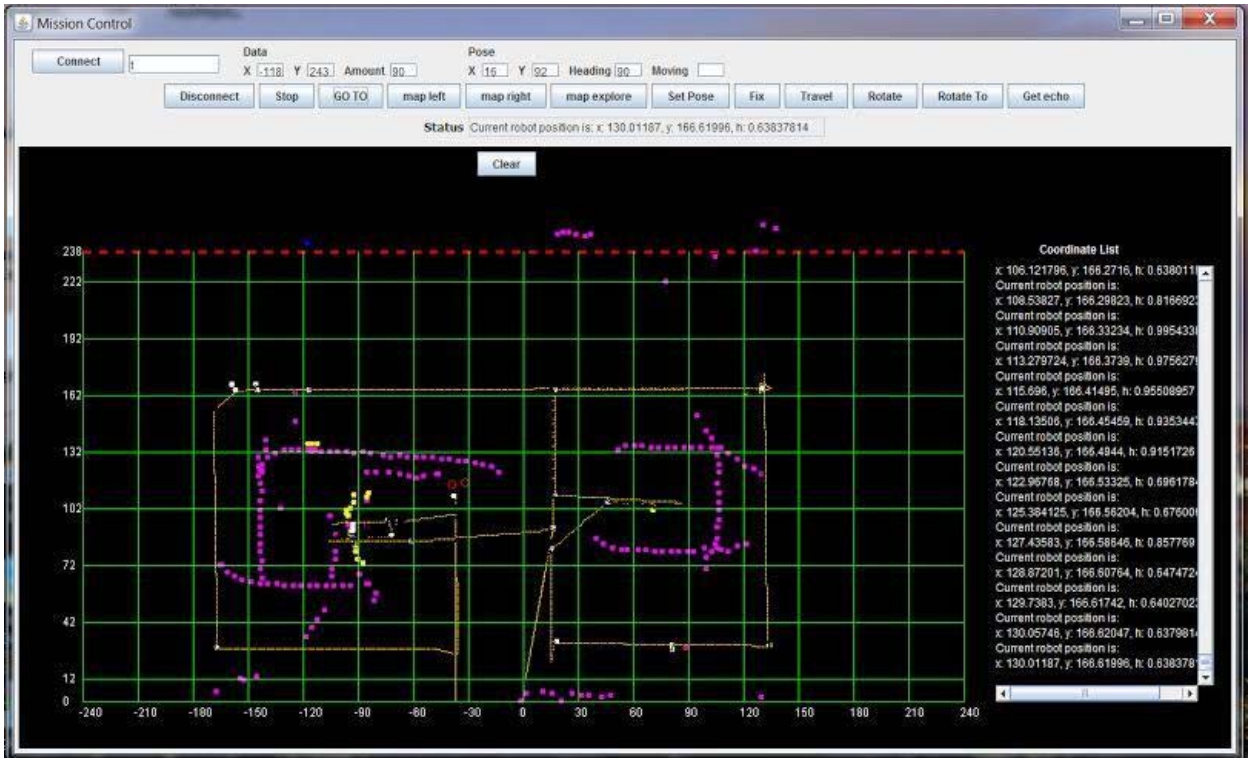
	X	Y	H
Original point	240	180	0
First Round	232	180	1
Second Round	231	182	-1
Std Dev	4.932883	1.154701	1

	X	Y	H
Original point	-240	60	180
First Round	-247	61	184
Second Round	-249	62	183
Std Dev	4.725816	1	2.081666

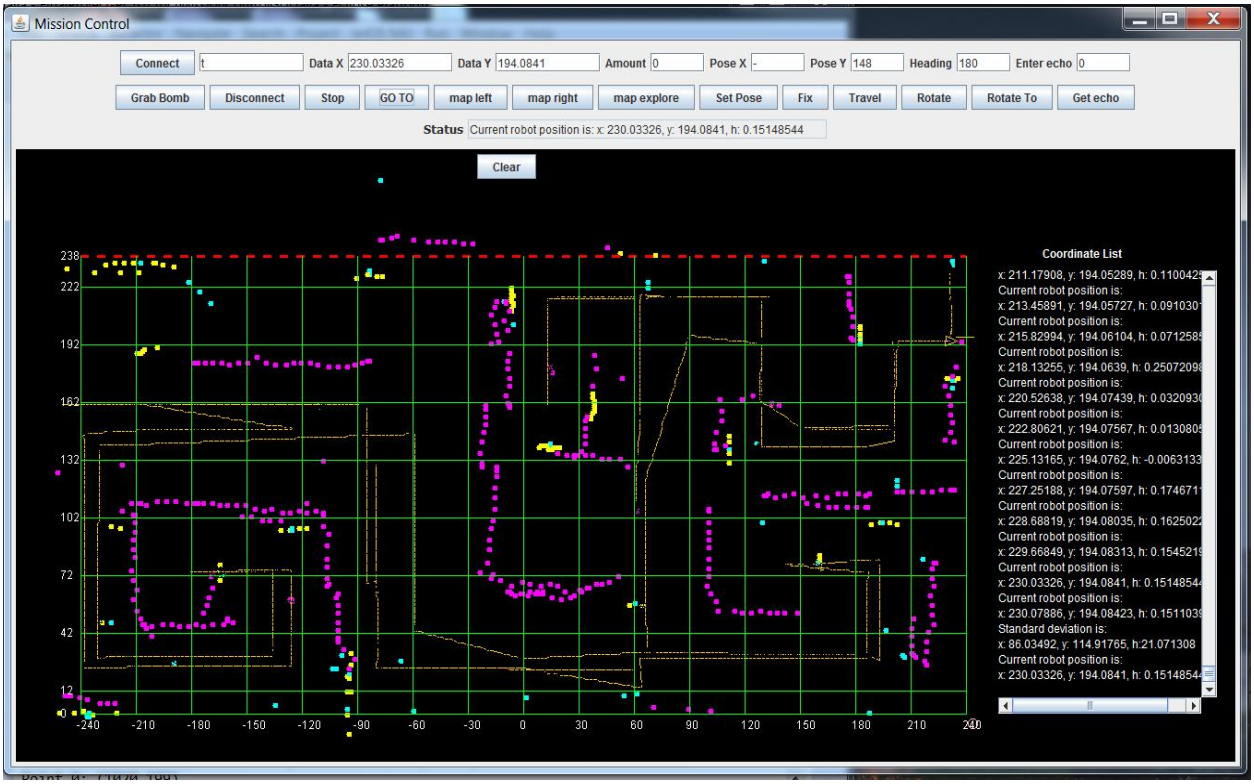
Appendix 9.13. Milestone 5B



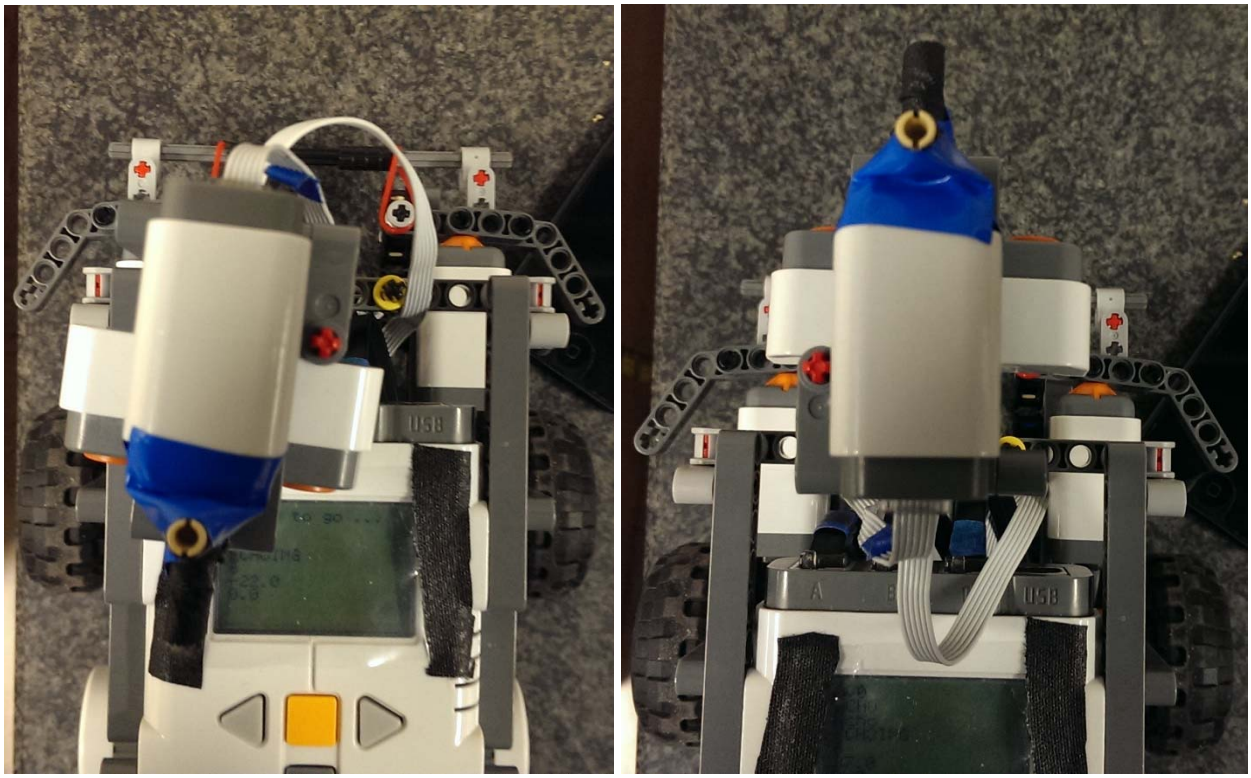
Appendix 9.14. Milestone 6 Simple Maze



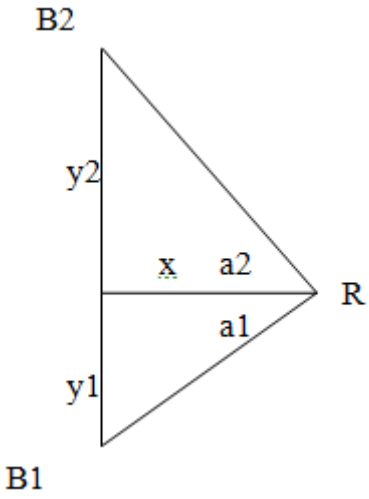
Appendix 9.15. Milestone 6 Complicated Maze



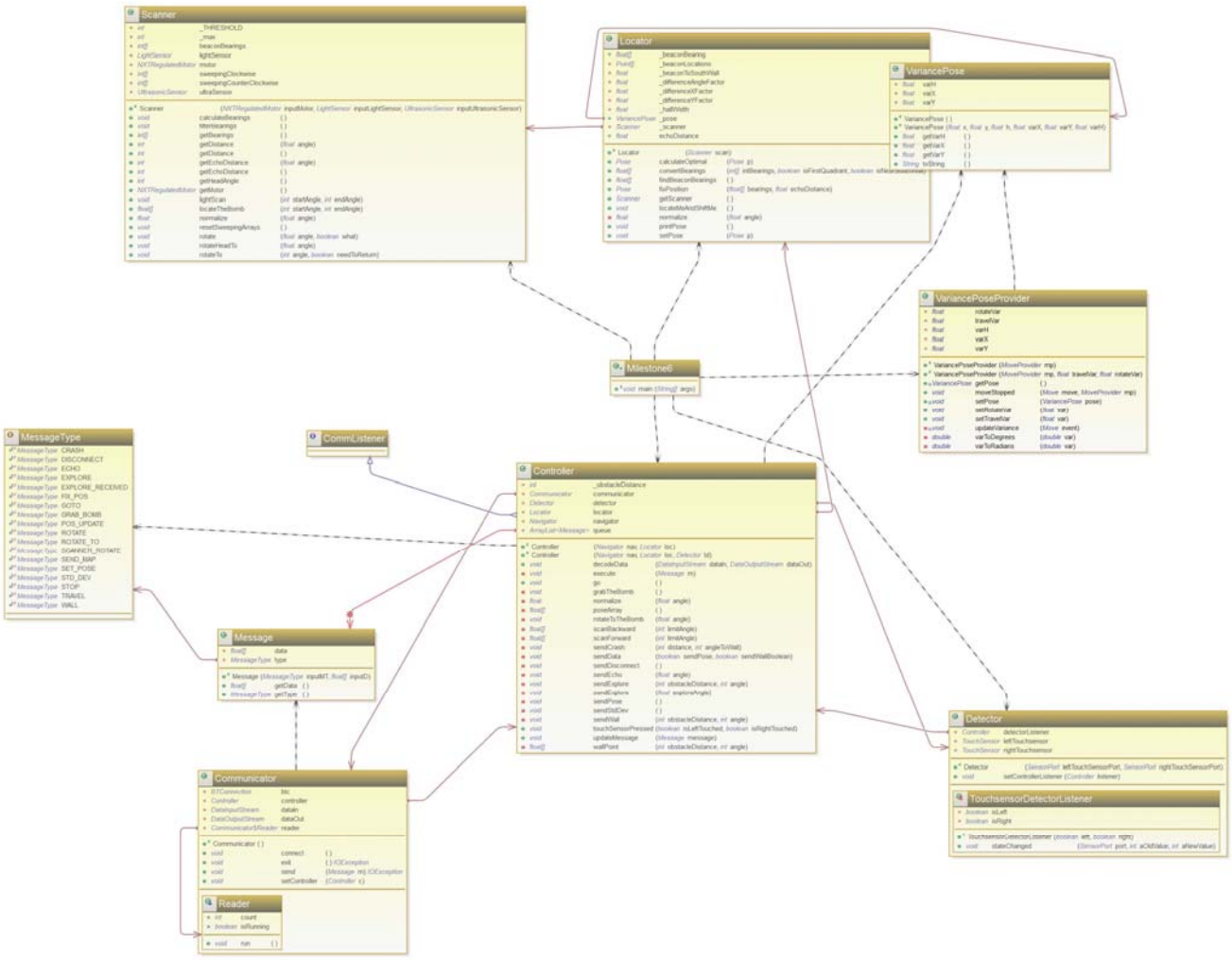
Appendix 9.16. Scanner head Ping 180 versus Ping 0



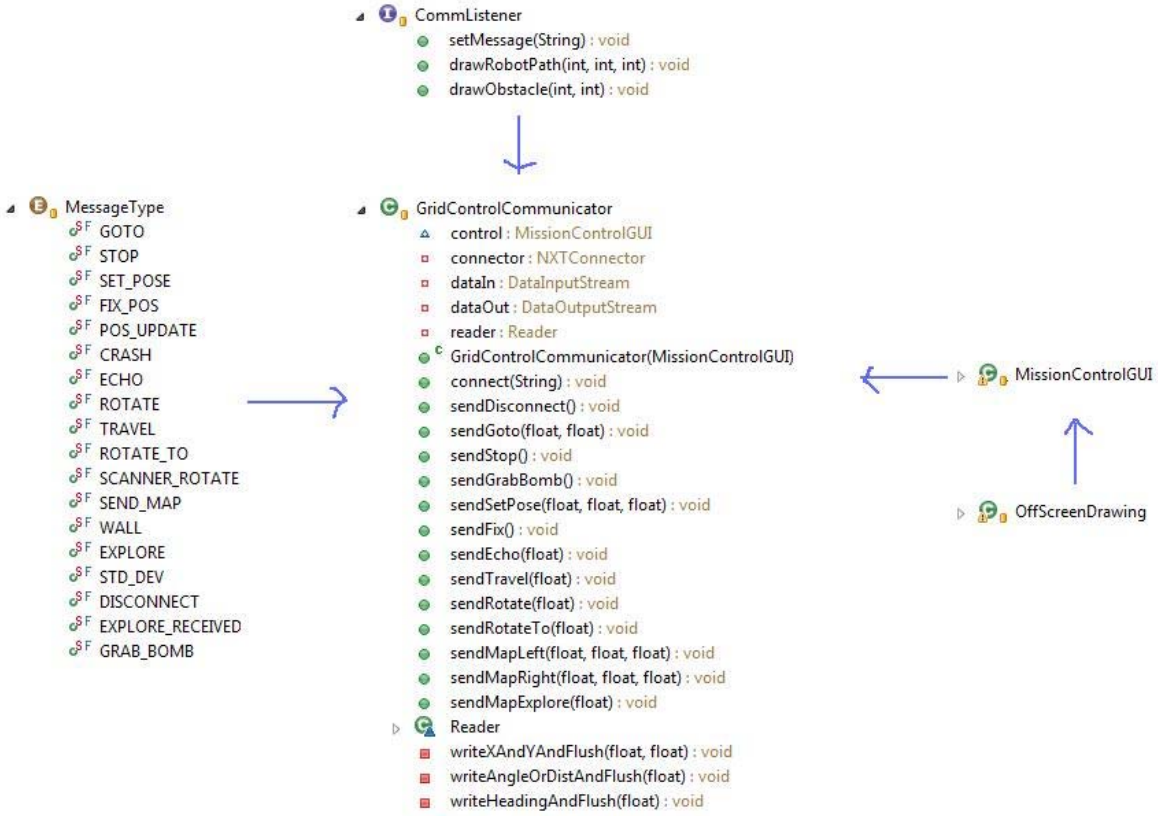
Appendix 9.17. Fix position geometric picture



Appendix 9.18. NXT Class Diagram



Appendix 9.19. PC Class Diagram



Appendix 9.20 Final Demonstration Map