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

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, and 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. The search and retrieval of a bomb that is to be detected by an implementation we decided to use, and to navigate several complicated mazes of walls which contain the bomb(s) such that (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.

*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 140 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.

*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 for example (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.

***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 mapping left or 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 millimeters.

Right wheel diameter: 5.22 millimeters.

Track width: 3.15 millimeter.

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 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 second2.

DifferentialPilot travel speed: 20 wheel diameter units per second.

DifferentialPilot rotate speed: 100 degrees per second.

***Milestone 2 Analysis***

*Experimental Data and Analysis*

- We coded a simple tester where our robot locates the bomb 60cm 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 retrevial plan. For example, collectively, we noticed that accurately backing up into the bomb was one of the most difficult aspects of the 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 11 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.

2.2. 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 2 lights and return their bearings to be used in Locator classsd

-          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.

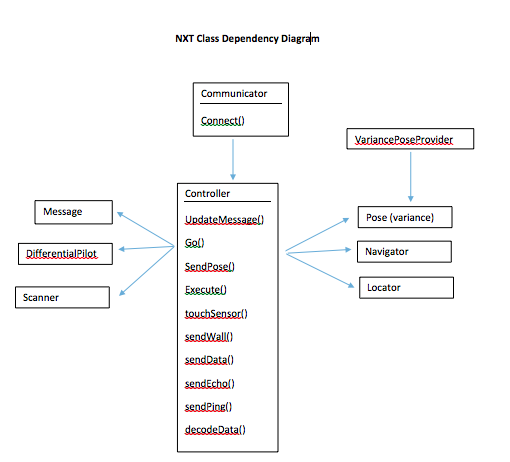
2.3 Software design:

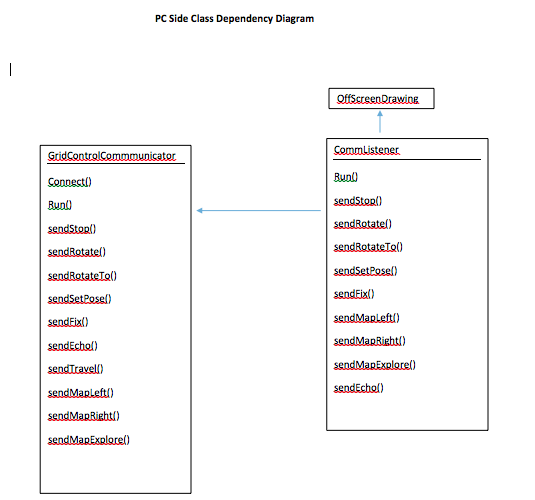
In order to scan for the beacons, we first decide which wall is nearer to the robot based on the current Y location, then that wall will be chosen to calculate other things. When the scanner is called, it will first scan and decide which angle to scan. No matter which heading it is, it will always scan 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.

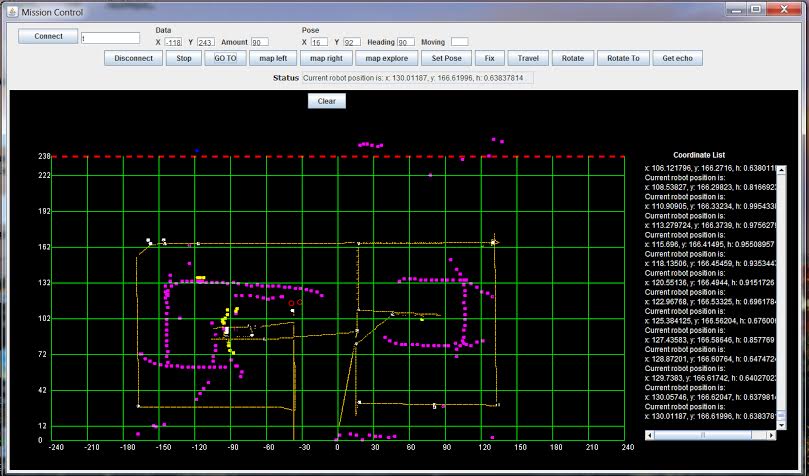
We used the data and algorithm from the second milestone for this project. It helps us to get the right values for both beacons and the distance to the wall that we want to have.

**3. Class Dependency Diagram:**



****

**Milestone 6 Mapping**



**4. Problem Analysis and Algorithms**

4.1.1. Algorithms Implemented:

- The general algorithm of this Milestone is to implement many methods that help the PC communicate with the NXT and command it’s 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. Depending on that value, we have a 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.

5.   **Software design** – Below are the newly implemented method descriptions added for the PC and NXT sides.

***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() accrodingly. 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.

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

There are a lot of difficult parts in this milestone.

To decide how to make the scanner always scan in the heading that have 2 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 rotate to another heading, the location of the axis is changed that makes the angles to 2 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.

The most challenging part of this milestone is the hardware. At first the threshold for light values keep changing for us, both of the lights don’t 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. (let’s say if we want to turn 90 degree 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 doesn’t get the right value either. Our task was to find a right speed, acceleration for it in order to make it work accurately.

**Links to source code and JavaDocs:**

Steam/Team5/FinalPoject-NXT/Milestone 6/