Project3 - Obstacle Race

Team Members: Trevor Davenport, Phuoc Nguyen, Khoa Tran, Corey Short

Brief Description: Two beacon lights will be set up about 20 feet apart, in middle of the hall outside 1174 Etcheverry. The robot starts within 2 feet of one light facing the other and makes 2 round trips. In future milestones, we will add objects detector and avoider.

Code: in our Project3 directory is a `src` directory which contains all the code. We break the code down to different packages. `experimental` indicates that the code there is used for data-collecting purposes. `robot` has all the robot functionalities code needed for all milestones. Finally, we have a separate package for every different milestone to keep our test code clean

Javadoc: can be found in the `doc` directory.

Milestone 1 Report

Control Scheme to Keep the Robot Moving toward the Light

In order to keep our robot efficiently and effectively moving towards the light, we implemented a toLight() method in our Racer.java s.t. our scanner object compares its current light sensor reading with two different constant light THRESHOLD's that we deemed optimal based on the experimental work via ScanRecorder.java. (graph provided below) The scanTo() method in Scanner.java rotates the light sensor to the input degree, (in our case, its between -60 and 60 degrees) calculates the tacho count of the respective motor, and while it is doing this, the LightSensor's getLightValue() method is being called to determine the the maximum light value at the robots current position. This value is defined in Scanner.java as xLight.

When we first tested the light intensity, we got different values for both light as one is brighter than the other. On our demonstration day, proffesor changes the light bulbs that makes the light intensity of both lights are the same. At first, we used two different THRESHOLD for 2 light bulbs but now we use only one because the light intensity is the same.

Another problem that we ran into when doing this Milestone is that the light bulb is too high for the light sensor that makes it unable to detect the right value of the light intensity. We all agreed to raise up the Light Sensor a little bit so it can detect the light value accurately.

For some cases, the light sensor detect the reflection of the light on the wall. We fixed it by raising up the THRESHOLD value, because the light intensity from the light bulb is bigger than the one reflected on the wall.

Racer.java's toAngle() method is called based on the the Scanner's tacho count in scanTo() which calls the differential pilot to steer the robot the direction of xLight. The trick we used to make this effiecnt was to add the case if our xLight is greater than our pre-set THRESHOLD we stop the robot and put it to sleep for 500 ms. This allows us to effectively call the scanTo() method (which was originally created by professor Glassey in ScanRecorder.java) again and repeat this process in a loop or break out of the loop and turnaround to start the next lap.

Relationship between Classes and Methods for the Tasks and Sub-tasks

Milestone1.java has the main method for testing and executing our code. It constructs the differential pilot and scanner instances to their respective motor and sensor ports on the NXT. Then the racer calls the toLight() method as it is described above.

The Racer itself encapsulates a DifferentialPilot object and a Scanner object, so that it can both trace the distance, while search for the lights at the same time

Finally, we package our code to divide up the functionality. The experimental code used to collect the data is separated into one package, while all the robot's essentials are in another. The test code for each Milestone is in their own corresponding package.

Graph Output of ScanRecorder.java, and How We Use this Data

Based on the graph we are able to determine the light intensity coresponding to each angle from the TachoCount() from Scanner, then the pilot's job is just to steer to that value of TachoCount(). Also from the data and graph, we can learn that whether the light sensor scanner is accurate or not. Because at first, there was a gap between 2 sides of the light sensor that make the value is up/down unexpectedly. Secondly, we did have a flat peak at first because the eye's angle we scanned was to large that make a lot of same peak values. The way we fixed it was to use another piece of lego (a pipe) so limit the eye angle that makes the eye just looks straigth.

Milestone 2 Report

Responsibilities of the classes designed for this milestone

- Detector: Inner class within Racer, used to detect objects using two seperate sensors (Ultrasonic and touch).
- Scanner: The 'main' class for our Robot, this is the skeleton that aides in getting and setting values for light objects and other higher level implementations.
- Racer: The racer class implements the algorithms and logic behind our robot being able to locate and determine light sources.

Task analysis and Brief Description for Object Detection

Since we want the Detector to have full access to the Racer's variables, we decided to implement the detector class as an inner class of Racer. This seemed like the most straight-forward way to implement the multiple threads needed to execute detection as required by the Milestone2 outline. We decided to give the same angle from our LightSensor to the UltraSonicSensor in Scanner.java.

However, we had an issue with objects being detected too far away or not close enough. So to fix this, we set the motor speed and acceleration slower for an overloaded Scanner constructor. This helped our robot scanobjects more accurately in our findLight() method in Racer.java. This method starts the detector thread while the robot is scanning.

In Detector, we used a boolean isDetected, initialized to false, for our run() method. While findLight() is being executed by main(), run() uses the ultraSensor's getDistance() to compare objects detected against a a threshold we set called distanceLimit. We were not really sure how to set this limit at first because the values returned by the ultraSonicSensor did not always fall within the 0-250 range as specified by the leJOS API. Trial and error allowed us to determine that a proper value to set for the distanceLimit, which is 25. If either getDistance() is greater than distanceLimit or if the left or right bumpers (as a last result or if the UltraSonicSensor is unable to detect an object because it is below its viewing limitations), isLeftTouched() and isRIghtTouched() respectiveley, return true for isDetected, the robot is stopped, and the detector thread yields.

Finally, just like in the first milestone, when the robot encounters the light, it stops, turns around, and finishes up the lap.

Milestone 3 Report

Responsibilities of the classes in this milestone

Milestone 3 implements all of the classes from the previous milestones with a few crucial changes and additions. Instead of putting the avoid logic in the Racer class, we chose to implement a separate Avoider class itself. As recommending by Glassey, our avoid() function uses a Random object to determine angles to choose from. Prior to this recommendation, we were simply reversing the robot and turning 45 degrees. This proved to be somewhat troublesome due to increased unecessary manuevers. For example, we have a better chance of making a correct turn angle if the choices are randomized. Our avoid() method takes in an integer argument that allows us to determine which sensors is relaying information to ourselves. Next, we also chose to implement the rovers built-in Sound.beep() in order to notion the specific location of an object. I.e. if our rover beeps once, we know the object is on the right most side of the sensors, twice relays that object is on the left.

Sub-tasks that the robot should execute after an obstacle is detected

As the robot is traversing through the obstacle course, when an object is detected we do the following:

```
1. Determine which sensor detected the object.
2. Pause the detections until we are out of range. (The object detected notifies the Detector to stop).
3. Determine where the relative location of said object is.
4. Reverse.
5. Rotate Using Random Number (Angle) Generator.
6. Tell Detector to continue detecting.
7. Continue to proceed on path.
```

Next steps after handling objects detection

Throughout the obstacle course, our rover is continously searching and seeking for the final destination light. We chose to implement a constant variable named THRESHOLD and chose to set the value to 55 after some careful testing. This constant value is what allows our rover to know that it has rearched the final destination light. This light determines that we have completed the course and notifies our rover to:

```
1. Stop Motion
2. Relay Stop to subprocesses
3. Rotate 180 degrees
4. Complete Obstacle course
```

Flow of information and control among Racer, Detector, and Avoider

```
                            >  Racer
                          /           \
                         /             \
                       <                 >
        Detector(inner class)         Avoider(communicates with Racer)
                |                                  |
                |                                  |
                |                                  |
Scanner(Detector relays information to Scanner)          Warns Detector of Objects, pauses detetction.
```