# Memo

**To:** Dr.Berry

**From:**  Carson Stone | Peter Garnache

**Date:**  1/12/20

**Subject:**  Mobile Robotics Lab 2

|  |
| --- |
|  |

## Introduction

The purpose of this lab is to develop the code needed to move the robot based on input from its sensors.

We created several different functions in order to achieve a range of movement based on sensors. We first created calibration equations that correlated the measured analog inputs from each sensor to the distance reading in inches. We also created functions that included the following functionality;

* Move in a direction until an obstruction is sensed, then stop.
* Move away from all sensed obstacles.
* Move in random directions at random speeds.
* Move around a room while avoiding any obstacles that are encountered.
* Move to a goal while avoiding any obstacles that are encountered

These functions give us experience working with the sensors which will help us in the completion of our wall following program for lab 3.

## Method

The basic movement functions that we used in this lab were drive (), spin (), and goToAngle(). We also kept the function runToStop() which drives both steppers at a constant speed until they reach their desired position. We also kept our LED handling functions setLED() and resetLED().

The first functions we created were functions to read values from each of our sensors and convert them to a readable value (inches). This includes readIRLeft(), readIRRight(), readIRFront(), and readIRBack(). These functions read the analog port corresponding to the IR sensor, and then used the calibration equation to return the measured distance. We also created functions to deal with reading the sonar sensors (readSonR() and readSonL()) that read the sonar senor values with the ping library. These functions all return a double with value equal to the equivalent distance sensed in inches.

The first new movement function we created is randomWander(). This function creates a random movement vector that the robot tries to follow. Using the random() function, we can create the x and y coordinate of the vector. We gave the y coordinate a better chance to be forward facing because we want the robot to prefer to move forward while still moving randomly. The equation we used to calculate the speed that we should move each wheel at is shown below.

We then called stepper.runSpeed() in a for loop and increased the target position whenever the steppers reached their pervious target position. This gave us movement at a constant velocity for the entire length of the movement for loop.

The next function that we created was angryKid(). This function moves the robot either forward or backward and stops when the robot senses an obstacle in its path. We used a simple if else structure to set up the logic for this step. The robot checked whether there was an obstacle in the path by reading the front and back sensors, then comparing their measured value in inches to a threshold value. If either of the measurements were smaller than the threshold value, the robot stopped. For every other sensor case, the robot moved forward using the same method of the randomWander(), but with both the left and right wheels moving at the same speed.

The next function created was shyKid(). This function moves the robot away from any obstacles that the IR sensor sees until the robot is safely free of obstacles. The first thing the program does is to read all four IR sensors around the robot and create Boolean variables that store the logical of whether the read value is lower than the threshold (i.e. whether an object is sensed). What then follows is a string of if and else if statements that cover all possible logical combinations of sensors readings. The first case is when no obstacles are seen, which tells the robot not to move. Another case where the robot should not move is when it sees obstacles on all sides. If the robot sees an obstacle on just the right or on the right, back, and front, the robot chooses to spin left. In the same way, if the robot sees an obstacle on the left, or on every side but the right, the robot should choose to spin right. The following case allows the robot to drive forward when it sees an obstacle behind it but not in front, or when it sees something on both sides. The last case that we check is when the robot senses an obstacle with the front sensor and the back is clear. In this case, the fastest way away from the obstacle is to just drive backwards, which is what our robot does. For all of these cases, they either spin the robot 5 degrees or drive the robot 1 inch before it re-measures the sensors and decides on the next movement to implement.

The next function was goToGoal(). The first time this function is called, it requires an x and y coordinate as parameter inputs to the function. This first call sets the global variable xGoal and yGoal to be equal to the parameters into the function. This function then calls a different goToGoal() function that doesn’t take any local parameters but relies only on the global variables. We designed our program this way so that we are able to use recursion to effectively loop our functions which removes their multiple calls from the for loop. This also allows us to keep our code modular by keeping each robot behavior controlled by one method.

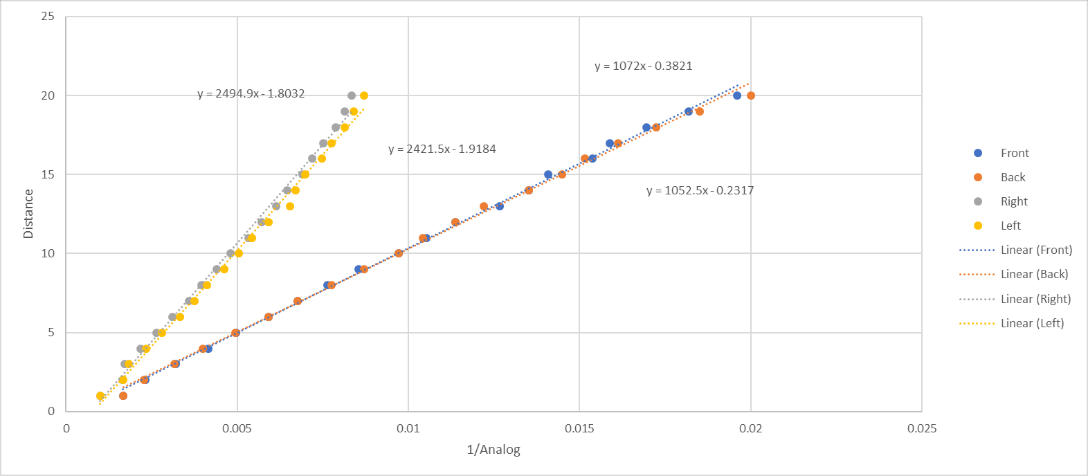
goToGoal(void) is the next function that we developed. This function first checks for the case that the robot has reached its goal and does not move if that is the case. If the robot has not reached its goal, then it checks each sensor to determine whether there is an obstacle in its path. If none of the sensors detect an obstacle the robot knows that it is free to drive towards its goal. We decided to attack our movement by driving along a 4-inch grid. We have a global variable currentCoord which keeps track of the current coordinate that our robot is trying to achieve. We also have global variables xPos and yPos that are updated in the drive() function and keep track of the current position of the robot. There is also a variable “angle” that keeps track of the current heading of the robot. If the robot has achieved its position in the currentCoord direction, then it sets its value to the other coordinate. If the robot has not achieved the desired position in the currentCoord direction, then a temporary variable “theta” is set to the desired heading for the robot to move in the direction of the currentCoord closer to the goal. If the robot is pointed in a direction other than theta, the goToAngle function spins the robot from its current heading to the desired heading of theta. The robot is then told to drive forward 4 inches and restart its calculations. goToGoal() is called recursively at the end of the movement if loop. If the robot detects an obstacle, then it immediately calls the avoid Obstacle subroutine;

avoidObstacle() is the final function that we worked on developing. This function assists the goToGoal() function by maneuvering the robot around obstacles. The first thing that avoidObstacle() does is reads all sensors and creates Boolean variables for which sensors read values that are below the sensing threshold. If the robot senses an obstacle, but not one in front of the robot, then it will continue to drive forward until the obstacle is passed. If the robot does not sense any obstacles, then it calls goToGoal() in order to return to moving towards the goal. If an obstacle is sensed in front of the robot, then the robot checks several other conditions to determine the next best movement. If there is also an obstacle to the right, then the robot turns left. If there is also an obstacle to the left, then the robot turns right. If neither of these conditions are true, then the robot chooses to turn in the direction that points it toward the goal. If the robot has a heading of either 180 degrees of 0 degrees, then it turns to move in either the positive or negative y direction so that it moves towards the goal. If it has a heading of 90 or 270 degrees, then it chooses the direction based on the x distance to the goal. At the end of these movement functions, it recursively calls avoidObstacle().

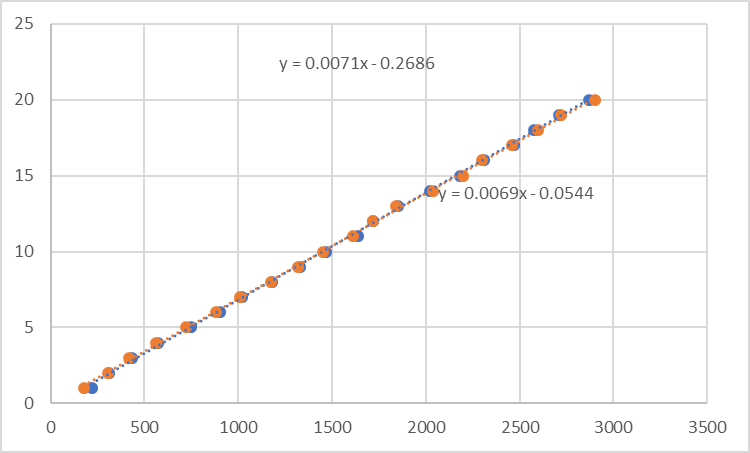
## Results

In the beginning of this lab, we had to calibrate each of the six sensors that our robot is equipped with. This was done according to the procedure below:

The robot was placed at a known distance from a non-reflective wall. Care was taken to make sure that the robot’s IR sensor remained perpendicular from the wall. The sensor analog value was measured for the known distance by averaging 20 readings from the IR sensor. This was done in one-inch increments starting 1 inch from the wall and ending with a reading 20 inches away from the wall. These measurements were printed out in the serial monitor and recorded in an excel spreadsheet. We linearized the sensor readings using the equation below to form a line of best fit. We computed errors for each of our measurements that compared the actual measurement in inches, and the sensor’s value converted into inches using our linearization and the line of best fit. We then changed the k value for the linearization of each sensor to eliminate as much error in the sensor’s readings. These lines of best fit were used in our code to convert the analog reading of each sensor into a physical distance reading in inches. The results of this test are summarized in the following plot.



We followed a similar procedure for calibrating the sonar sensors, but their readings had a linear relationship with the measured distance, so we did not have to try linearizing the data. A plot summarizing our results is attached below.



The full data set from our testing will be included in the appendix.

## Questions:

1. What was the general plan you used to implement the random wander and obstacle avoidance behaviors?

For randomWander, we created a vector that had a random magnitude and direction. That vector defined the speed of the robot, with the y coordinate mainly defining the speed, and the x coordinate defining the differential between the right and left wheel speeds. This process was repeated until random wander was told to stop.

For obstacleAvoidance we created calibration equations to change the sensor readouts in inches. We could then determine if any of the values fall below our chosen threshold value. Each sensor that had a readout below the threshold value meant that the robot sensed an obstacle in that direction. Depending on what sensors detect obstacles, we either tell the robot to spin left, spin right, drive forward, or drive backwards to avoid the obstacles.

1. How did you create a modular program and integrate the two layers into the overall program?

To make our program modular we had each of our robot behaviors be controlled by one method. That way we could just call a singular method to have our robot do something. This became very helpful for when we had to write the smartWander program which has both the randomWander and obstacleAvoidance behavior. To do this, all we had to do was have the robot do obstacleAvoidance when it detected an obstacle and do randomWander when it didn’t.

1. Did you use the contact and IR sensors to create redundant sensing on the robot’s front half?

We did not use any redundant sensing because the IR sensors performed well enough to allow the robot behaviors we needed.

1. How could you create a smart wander routine to entirely cover a room?

In order to create a smart wander routine that covers the floor of a room, I would first have the robot go forward until it hits a wall. This wall will now be considered the top wall. I will then have the robot turn left and move along the wall until it comes to a corner. This new wall will now be called the left wall. Next the robot will turn around and follow the top wall again until it comes to another corner. This wall will be considered the right wall. The robot will now go back and forth between the left and right wall in a line parallel to the line it followed while tracing the top wall. After each time the robot moves from one of the side walls to the other, I will have the robot move down one robot thickness before heading back to the other side wall. If the robot encounters a wall while trying to move down as just described, I will have that robot move along that bottom wall. If that wall meets the opposite side wall, the program terminates, and the robot has covered the entire floor of a room. If the bottom wall does not go all the way to the opposite side, the robot finds where the bottom wall ends and establishes a new side wall based off that. The robot will then go back to bouncing between the side walls until it finds a bottom wall that connects the two side walls. During this entire program the robot looks for obstacles and goes around them to continue its path.

1. What kind of errors did you encounter with the obstacle avoidance behavior?

The biggest error we encountered during the obstacle avoidance happened after the robot cleared an obstacle which typically meant the robot followed a wall until it ended. The problem was that the robot would then move in a direction that caused it to approach the next side of the object it just cleared at an angle where its sensors could not detect the obstacle. We ended up fixing this problem by redesigning our obstacleAvoidance routine to work completely in a square grid where it would only execute 90 degree turns.

1. How could you improve the obstacle avoidance behavior?

Implementing better wall fallowing would drastically improve the robot’s obstacle avoidance behavior. Having the robot more accurately move around corners of an obstacle would also improve the obstacle avoidance behavior.

1. Were there any obstacles that the robot could not detect?

The robot could detect any obstacle that was straight off one of the sides it. The robot cannot detect any obstacles located diagonal to the robot. The robot also had trouble detecting the clear plastic box that it is stored in with the IR sensors.

1. Were there any situations when the range sensors did not give you reliable data?

The sensors had a lot of noise, but the calibration curve and averaging several readings together helped smooth out the signal. The sensors lost accuracy at distances greater than 19 inches and less than 3 inches.

1. How did you keep track of the robot’s states in the program?

We kept track of the robot’s state by knowing what method the robot was using at all times. These methods dictated how the robot acted and each state had its own LED signal allowing us to determine the behavior the robot was exercising and when. This allowed us to better understand what methods and behaviors were causing problems and see if there were problems switching between different states.

1. Did the robot encounter any “stuck” situations? How did you account for those?

We had catch-all statements in our program in case something happened to the robot that did we did not account for in our in our program’s logic. When a catch-all was reached, all the robot’s LEDs were turned on the robot stopped. This allowed us to clearly see what was causing us to go into fall out of the logic of the program and into the catch-all. We also had one time where our obstacleAvoidance got in a loop

1. How did you keep track of the goal position and robot states as it integrated avoid-obstacle and go-to-goal behaviors?

The goal position was just implemented two global variables yGoal and xGoal. The robot’s current position was also implemented as global variables which were updated whenever the robot was asked to move. The robot’s current angle was also implemented as a global variable in order to update the robot’s position correctly. Because all of these variables where global, we could call on them and update them from any state.

1. What should the subsumption architecture look like for the addition of the go-to-goal and avoidObstacle behaviors?

The subsumption architecture should have the avoidObstacle as layer 0. This will ensure that the robot is constantly trying to maneuver around obstacles until it all of its sensors no longer detect an obstacle. When all of the robot’s sensors do not detect anything the robot implements the go-to-goal behavior in layer 1 of the subsumption architecture. The robot can be brought back down to avoidObstacle whenever a sensor detects an obstacle.

## Conclusion

This lab, we learned how to use logic to control a mobile robot based on its view of the surrounding through sensors. We had to create calibration equations for each of our 6 sensors in order to effectively use then together in our programs. Although each sensor looked the same externally, we determined that the electronics controlling the left and right IR sensors were different from the front and back sensors because they had very different calibration curves. Both the front and back IR sensors had similar calibration equations, and the left and right ones were similar to each other, but there was a very visible difference between the two groups of sensors.

One of the most interesting things that we did was the design of the goToGoal() group of functions. It was very interesting to program in a logic sequence and state machine and have the robot choose which direction to go and determine the best way to navigate around the obstacle. This was the first time that we had a chance to program robots with that much freedom of movement to the point where we were not even sure how the robot would overcome some of the test conditions we subjected it to, but it was able to overcome them all. We had to tweak the logic a little whenever we found a configuration that would get it stuck, but overall it was a very exciting and interesting.

We successfully completed the target of the lab and created several different behaviors for our robot that will help us be successful in future labs and tasks.

## Appendix:

