# Memo

**To:** Dr.Berry

**From:**  Carson Stone | Peter Garnache

**Date:**  1/26/20

**Subject:**  Mobile Robotics Lab 3

|  |
| --- |
|  |

## Introduction

## The purpose of this lab is to develop the code needed to transition between different states based on sensor input. These different states each controlled the motion of the robot based on IR sensor values differently.

## randomWander looks for walls to follow and obstacles to avoid

## followLeft makes the robot follows a wall on the left side of the robot

## followCenter makes the robot stay in between two walls

## followRight makes the robot follow a wall on the right side of the robot

## collide makes the robot go around an obstacle and return to the previous state

## avoidObstacle makes the robot avoid running into things once it comes to close to something

## Method

The basic movement function we used for this lab was drive(). All this method does is run both the right and left stepper at a previously inputted speed. The main loop of the program just calls drive() to ensure we have continuous motion.

We also initialized an interrupt that occurs at a set timer\_rate. We had our timer interrupt occur at 20 times per second. This interrupt calls updateSensors(). This method updates all of the IR sensors and then updates the current state of the robot based on each of the values of the IR sensors. We decided to not use the sonar sensors due to the unreliability in their readings.

There are 6 states the robot could switch between. See the attached state diagram in the appendix to see how each of the states is triggered to transition from one to another. updateState() which is called at the end of updateSensors() puts the robot in either randomWander, followLeft, followCenter, followRight, collide, or avoidObstacle. We gave each of these states its own unique LED signature so we could tell what state the robot was in just by looking at what LEDs were lit. this helped us debug the robot be letting us see when the robot was in each state and what was causing it to transition between states.

The default state, randomWander, causes the robot to randomly wander while mostly going forward. This allows the robot to continue moving straight while keeping an eye open for obstacles to avoid and walls to follow. If the robot stays within another state for too long without any of the sensors detecting anything, we have the robot re-enter randomWander to search for another wall or obstacle.

The robot enters the avoidObstacleState if any sensor detects an obstacle that is too close to the robot. Once in the avoidObstalcleState, the program will set the speed of the motors to move the robot in the opposite direction of the obstacle.

The robot enters followLeft if the IR sensor on the left side of the robot detects an obstacle that is not too close. While in this state the program will move the robot towards the wall if the sensor tells the robot that it is more than 5 inches away. The program will also move the robot away from the wall if the IR sensor detects that the wall is less than 5 inches away. This method does this by calling updateSpeed() which sets the speed of the motors based on the calibrated Kp and Kd coefficients. These coefficients look at the size and rate or the error between where the robot is and 5 inches from the wall to calculate how to adjust the motor speeds.

FollowCenter is entered when the robot sensors detect obstacles off of the left and right side of the robot. This state looks at the difference in the distances between the robot and the two walls and inputs how offset the robot is from the center as the error to the Kp and Kd coefficients. This error is used by the updateSpeed method which changes how the drive method moves the robot in the continuous loop.

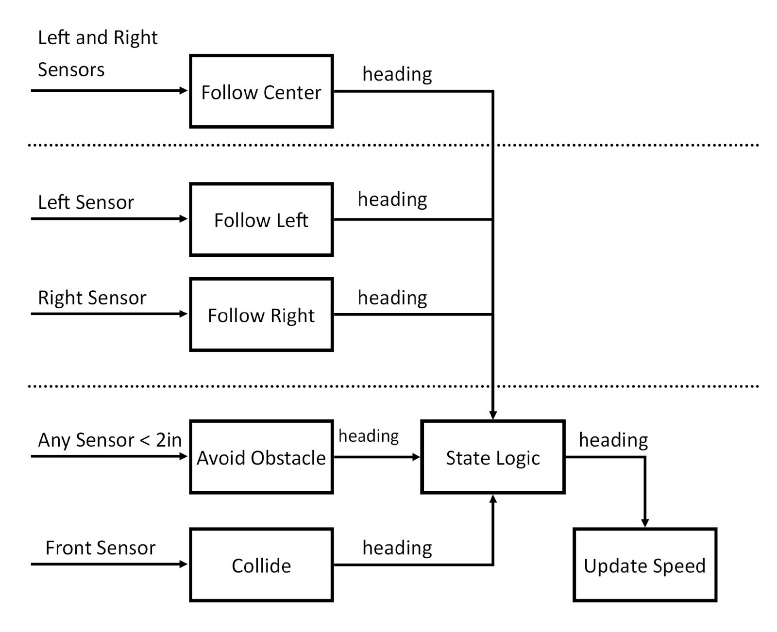
The robot enters followRight if the IR sensor on the right side of the robot detects an obstacle. This state calculates error and controls the motion of the robot just like the followLeft state except following a wall on the right side.

Collide is our most complicated state because it behaves differently based on what state the robot was in prior before entering it. The robot enters collide once the front IR sensor of the robot detects something is within a set threshold value. If the robot was previously in followRight, the robot will attempt to turn left to get around the obstacle. If the robot was previously in a followCenter or followLeft, the robot will attempt to turn right to get around the obstacle.

## Results

Lab Questions:

1. What does a diagram for the 3 layer subsumption architecture look like?



1. What did the robot do when it encountered a corner while wall following?

When the robot encountered a corner, the state machine set the state to collide, and the robot started turning away from the wall. The robot turned sharper as it got closer to the wall, and as soon as the front IR sensor was not triggered, it resumed wall following.

1. What did the robot do when it encountered doorways and/or corners?

When the robot encountered doorways, it would continue acting in the followWall state for two seconds. During that time it would still be turning based on a PD feedback loop and the IR sensor values. Because the sensor was reading above the threshold, then the robot would be turning at the maximum turn rate possible. If the robot could not find the wall within two seconds, then the state machine switched to the random wander state to go look for a wall.

1. When tuning the proportional controller and/or derivative controller, did the robot exhibit any oscillating, damping, overshoot or offset error? If so, how much?

The robot started out with a small amount of oscillation and overshoot when we started implementing just P control, but when adding the D control and tuning both gains, we practically eliminated any oscillation.

1. What were the results of the different P and D controller gains? How did you decide which one to use?

Increasing the P gains increased the magnitude and speed of the oscillations of the robot’s position relative to the wall. A smaller P gain decreases the speed of the oscillations, and improves the accuracy of the response, at the expense of a larger turning radius when turning a corner. Increasing the D control slows the oscillations, which removes overshoot and reduces oscillations. Decreasing the D control brings the system closer to only P control.

1. How accurate was the robot at maintaining a distance between 4 and 6 inches?

Our robot was able to very accurately maintain a distance between 4 and 6 inches. After tuning the PD controller, we did not see oscillations of more than 1.5 inches away from our goal of 5 inches away from the wall.

1. Did the robot ever lose the wall?

Our robot never lost the wall under normal running circumstances. If we pushed the robot away from the wall, or removed the wall, then the robot would look for the wall for about 2 seconds, then switch into randomWander and start looking for another wall.

1. Compare and contrast the performance of the Wander and Avoid behaviors compared to last week’s lab.

This week, we were forced to re-work the random wander and avoid behaviors to work with our new drive() function. The result of this re-work is that the random wander behavior moves forward more often, so it is easier for it to find a wall. The avoid behavior changed form a little, but its response to outside stimuli is not greatly changed.

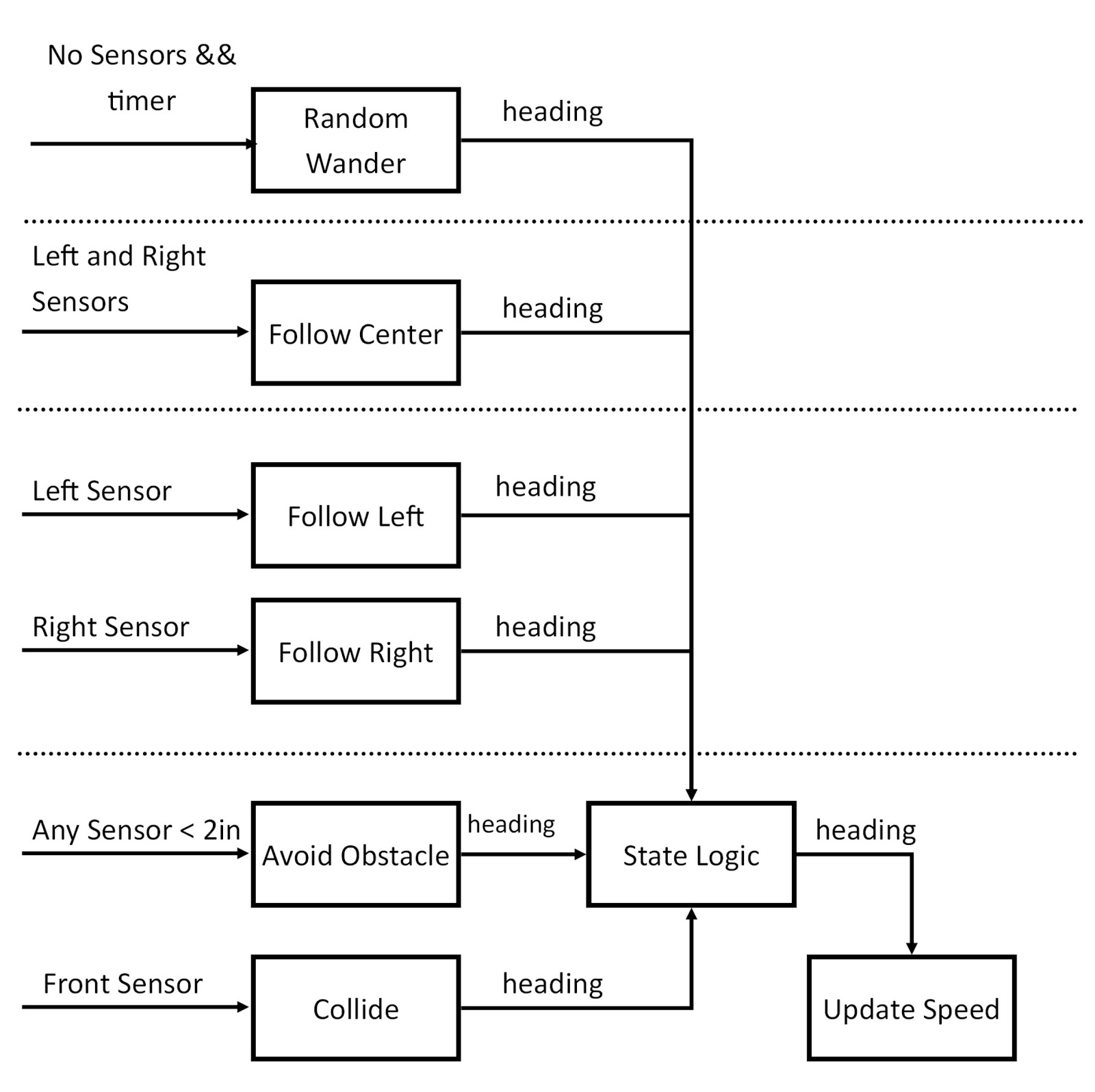
1. What was the general plan to implement the feedback control and subsumption architecture on the robot?

The general plan to implement the subsumption architecture was to use timer interrupts to update sensor values, and then run through a stacked if statement to check conditions. The first conditions that were checked were the bottom layers of the subsumption architecture. We implemented the feedback control into the followRight, followLeft, and followCenter functions that determined the direction that the robot should drive in each logical cycle.

1. How could you improve the control architecture and/or wall following/follow center behaviors?

We could improve the control architecture by implementing a switching drive control so that we could enact movements that lasted longer than one logical cycle. This would allow our robot to turn 90 degrees when it sensed itself running into a corner, or spinning 180 degrees when it is in a dead end hallway.

1. What does the overall subsumption architecture diagram with all 4 layers look like?



1. What was the pseudocode and flowchart for the program design?

The flowchart for the program design is attached in the appendix.

1. Did you use any suppression and inhibition with the integration of layers 2 and 3?

We used suppression between our different states to make sure that only one behavior output was present at a time. There was no circumstances where it would be beneficial to mix an input between two different behaviors so we replaced the behaviors we did not want present for a given input condition.

1. How did you implement the finite state machine to integrate the various behaviors. Did you use any inhibition and suppression to create layers in this behavior?

We created a state function that was called at the start of one logical cycle (a timer interrupt that refreshed our sensors). The state function uses stacked else if statements to give each possible function their respective priorities. Because avoidObstacle is the most important behavior, it is the condition that is checked first. We then check the other conditions in the order of their priority to the system.

1. How did you keep track of the robot’s state as it switched between behaviors.

We kept track of the robot’s state and past state as it switched between behaviors with global variables. Each state’s logical function updated the global variables the first time it was called.

## Conclusion

This lab, we learned how to implement a subsumption architecture with several layers through a finite state machine. We created a state update function to implement switching through states based on inputs to the robot, then fine-tuned each input to do exactly what we wanted them to do. We implemented a position-derivative controller to accurately follow walls on either or both sides of the robot.

One of the interesting things that we tried for this lab was to make our movement function more robust by using only 1 drive function and constantly updating the speed of each motor based on the most current sensor readings. Because the drive function was just a continuous loop, we were able to move all of our logic into the timer interrupt, which lowered our computational power during runtime and made our code leaner.

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:

