Assignment 4
Maze Navigation

Luke Doman
Team 10

Introduction

The goal of this assignment was to take all the skills we have learned in the previous assignments and put them to use in solving a more interesting problem, solving a maze. The assignments have been increasing in level of difficulty to prepare us for this, and is the first time that we actually can revisit our hardware design. We know some information about the maze that assists us with this task, such as all the turns are 90 degrees and there is a light source at the end destination. This first known fact is very useful for our design which will be discussed in the sections below.

Mechanical Design

Our initial approach involved experimentation with our baseline mechanical architecture. We thought it may be advantageous to reduce the size of our robots frame so there would never be a scenario in which our robot gets stuck in the maze due to its size. After experimenting with several ideas, we concluded that we would stick with our squarebot frame, but make several modifications to it. Our reasoning here was that since it appeared that there was no instance where squarebot's size would be the reason it is stuck, we could save time on our hardware design and get a head start on our software approach.

To help clarify our hardware approach, I first must mention our idea for solving the maze. Our strategy is to follow the right wall for the length of the maze with just touch sensors first, and then integrate an ultrasonic sensor later. We decided this would be a better approach than just throwing all the sensors on at once, and then tweaking them all together. Please see figure 1 below to see the logic for our touch sensor placement. With this placement we can always determine the next step to take to reach the optimal state of being perfectly parallel to the wall.

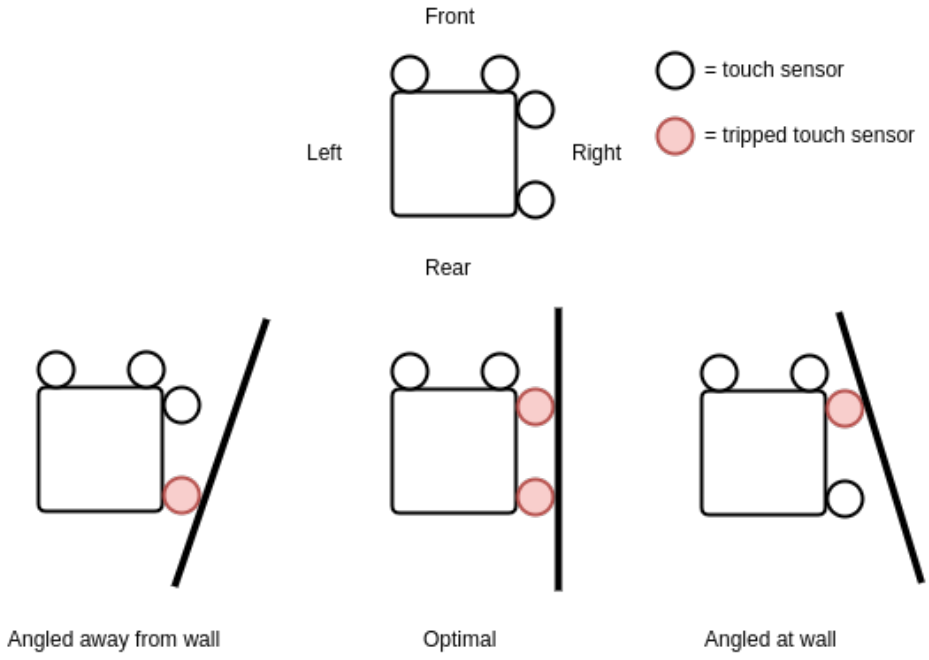

Figure 1: Touch sensor placement

The modifications we made to squarebot started with the gear ratio. Our idea was that when correcting to try and reach our optimal state we move slowly, but once parallel to the wall we should go as fast as possible. We switched out the medium size gears that the wheels were initially on the same shaft as with the smallest gears in the set. This required minimal modification to the frame which was central to our approach. Once modified we began testing this new gear ratio both on a test stand and the carpet. While this new ratio gave us a huge boost in speed, we also lost a significant amount of torque. When running just on the floor we could see that it moved with ease, but on the test stand we applied pressure to the wheels to see how much force would be required to stop it and observed disappointing results.

We were able to bring our wheels to a halt with a minimal amount of force, even with just our hands. This posed a problem since our solution to the maze required us to constantly be riding against the wall. We concluded to return to the previous gear ratio since our robot needed the additional torque. The next modification we made was to remove the battery stand to save some weight and free up space for our side touch sensors. We relocated the battery to the rear of the frame, housed within metal supports to ensure it doesn't fall off while navigating the maze. With these changes and our sensor placement we had a good platform to start our software development process.

We noted that for future work we could possibly replace our side sensors with an ultrasonic one to reduce our touches penalty, but we would lose some of the precision of knowing our exact angle in relation to the wall. We also had the idea of replacing our front touch sensors with an ultrasonic to again save on the touches penalty, but we never came back to hardware revisions after we made significant progress on the software side of the project and observed consistent success.

Software Design

Our approach to our robot's software architecture was based around several states that our robot could be in at any given time. This approach allowed for us to break functionality out into several functions and make our code significantly more readable while simultaneously making our debugging process simpler. The central part of our code then simply determined which state it was in and updated a state variable to reflect that. Once we know the state, we simply call a router function that then executes the correct function for that state. Our main control flow for determining state can be seen in figure 2 below.

Something to note in that flowchart is the state counter. During testing in the maze we noticed at several points the robot has a tendency to get stuck. While we made appropriate hardware and software modifications to reduce the probability of this, we still observed it when running in the maze. To counteract this we keep track of how many iterations of our control loop have executed in a row for the same state. Once the state counter reaches a threshold, which we defined as 30 iterations, it calls an escape function.

Escape behaves differently for each state of the robot. For example if we're stuck in state 2, which means both side sensors our touching the wall, we know there is just too much friction on the wall and not enough torque for the robot to move. In this instance the robot reverses a short distance and then charges ahead at a faster speed than before. This proved to be a highly effective solution to getting stuck in the maze.

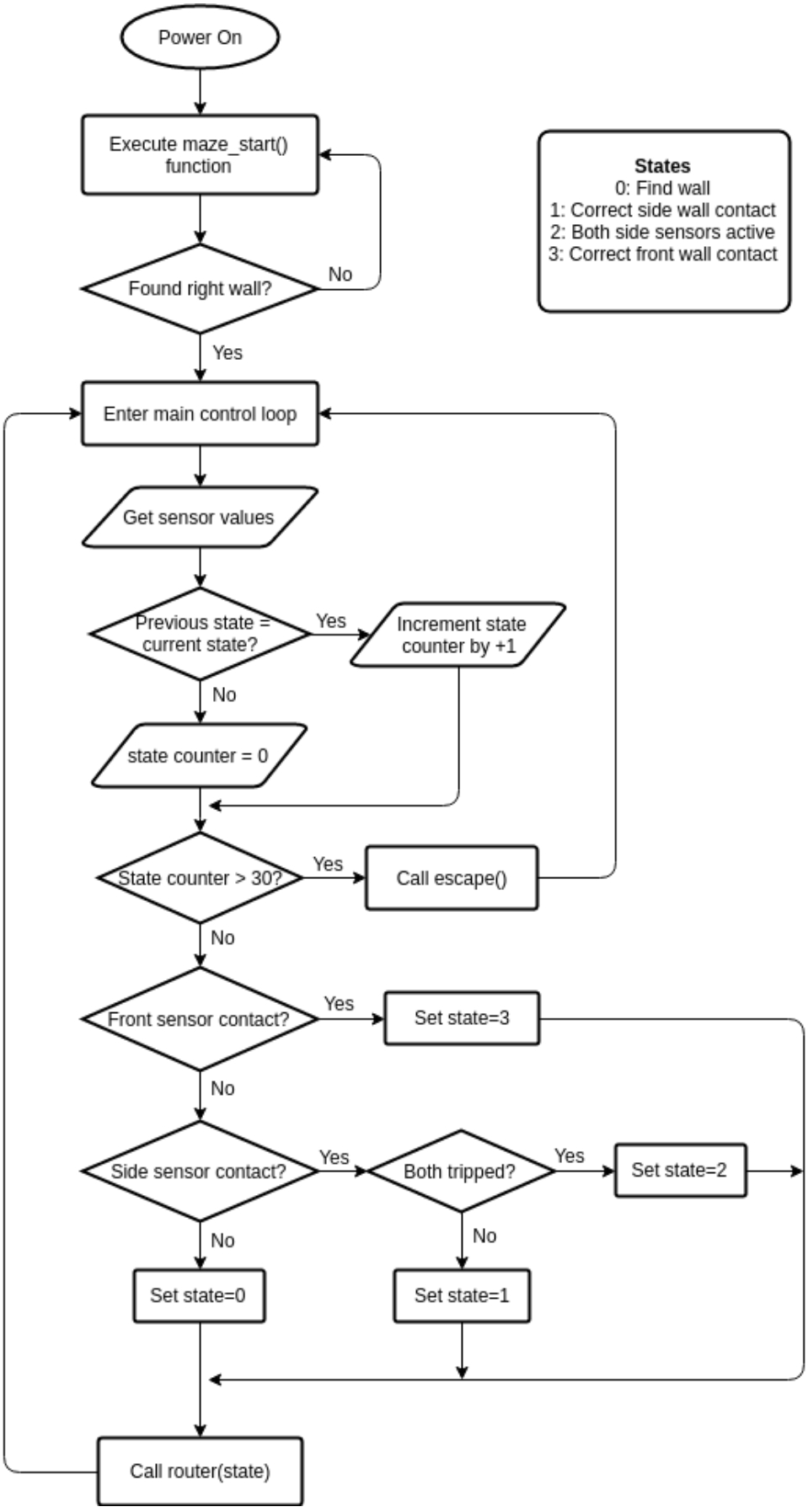

Figure 2: Main control flow

Each state has an affiliated function that handles it's respective behavior. The behavior for state 1 is modeled in figure 3 below. To be put in state 1 the robot has to not have any front wall contact and at least 1 side wall contact. Knowing this, we can reorient the robot in an optimal way so that it reaches state 2. If the right front sensor is touching we're angled at the wall and need to soft turn left. Soft turning is when the robot only uses one motor to execute a turn, instead of two in opposite directions. The logic governing the right rear sensor is a little more complicated to add some nice functionality to our robot. After the right rear is tripped the robot will execute a soft right turn until either any sensor but right rear is tripped or it's local state counter exceeds a threshold. The reasoning here is that this pushes the orientation back against the wall until in state 2, or if it needs to, execute a 180 degree turn. The local state counter here acts as a failsafe in case we end up going in circles and need to reacquire the wall. Figure 1 helps illustrate the robot's orientation in relation to the wall.

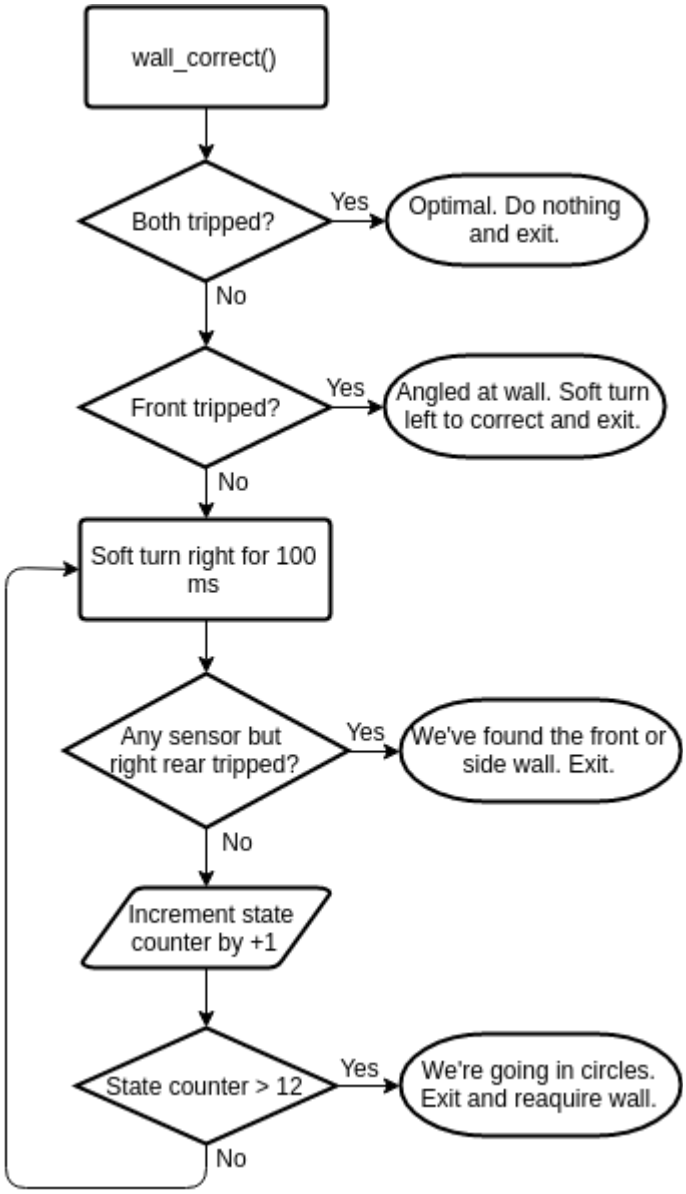

Figure 3: State 1 function (side wall correction)

Once we had this architecture in place a great deal of our time was spent optimizing turn times and only adding minor amounts of functionality as we saw fit. Initially the robot would have a difficult time transitioning from state 1 to 2. Side wall correct would overcompensate for the angle to the wall and would cause the robot to be perpetually angled at or away from the wall, but never parallel. We were able to address this by shortening our turn times drastically. A potential concern with this is under correcting, but since we would still be in the same state the robot would just finish the turn in the next iteration.

Optimizing these corrections is crucial for our approach since we are knowingly accepting a penalty by touching the wall. The idea here is that if we can get to state 2 as fast as possible and with the minimum number of corrections we can make up for our loss with our speed in state 2. While we abandoned our high gear ratio approach, we still kept our idea to heart that if we know we are perfectly parallel to the wall, we want to move forward as fast as reasonably possible.

Performance Evaluation

In our many testing trials we started with minor successes, but eventually reached a point where we could consistently solve the mazes in less than a minute and a half, not counting penalties. Seeing our capability and limited time remaining before competition day we opted out of adding the ultrasonic sensor. While it would help decrease our number of penalties overall, we opted instead to further optimize our proven configuration.

On competition day our robot completed the maze in one minute and eighteen seconds, tying for fourth place. We experienced an anomaly on our first launch which required a restart, so ideally we may have performed even better. Comparing our robot to the other teams, ours may have been the most differing from the common strategies seen. Every other team incorporated ultrasonic sensors to reduce number of contacts. We witnessed several occasions where they exhibited the exact behavior that was the precise reason we didn't incorporate it into our design – lose of precision. The robot would be attempting to follow a wall, but would not be perfectly parallel and that would lead it to non optimal paths or getting stuck.

The competition day maze definitely favored right wall following robots vs left wall. There appeared to be an even distribution between the two, and the left wall following teams appear to have suffered for that reason. Our team had an idea to alleviate this issue, but we didn't have the amount of sensors required for it. Our idea was to have a mirroring set of sensors on the left side, and use a jumper to set it in left or right wall following mode on competition day. We also thought of the ability to have a jumper increase our max speed if it was present, and then if we messed up the first run we could just unplug it, restart the robot, and run it in "safe" mode. Ultimately we did not execute these ideas, but they would certainly increase the robustness of the robot's performance.

Conclusion

Considering our approach was centrally based around accruing penalties in favor of speed and determinism, I'm happy with the end result of our robot. The difficulty of this task was a significant step up from previous assignments which made this a really interesting problem to tackle. This assignment also really stressed the amount of tweaking required to achieve optimal performance as expected. The difference in performance between our first run in the maze compared to our last was dramatic, even though the code base stayed mostly the same. I look forward to applying the various skills learned from this assignment to our next.

Appendix

```
#pragma config(Sensor, dgtl1,  front_left,     sensorTouch)
#pragma config(Sensor, dgtl2,  front_right,    sensorTouch)
#pragma config(Sensor, dgtl3,  right_front,    sensorTouch)
#pragma config(Sensor, dgtl4,  right_rear,     sensorTouch)
#pragma config(Motor,  port2,           leftMotor,     tmotorVex393, openLoop, reversed)
#pragma config(Motor,  port3,           rightMotor,    tmotorVex393, openLoop)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

//// Constants
int turn_speed = 50;
int soft_turn_speed = 30;
int forward_speed = 57;
int hyper_speed = 85;
float state_threshold = 30;

//// Variables
int prev_state;
int state;
int front_left_bump;
int front_right_bump;
int right_front_bump;
int right_rear_bump;
float state_counter;


bool sensor_contact_turn()
{
  front_left_bump = SensorValue(front_left);
  front_right_bump = SensorValue(front_right);
  right_front_bump = SensorValue(right_front);
  right_rear_bump = SensorValue(right_rear);

  if (front_left_bump == 1 || front_right_bump == 1 || right_front_bump == 1)
  {
                  return true;
  }
  else
  {
                  return false;
  }
}

bool sensor_contact()
{
  front_left_bump = SensorValue(front_left);
  front_right_bump = SensorValue(front_right);
  right_front_bump = SensorValue(right_front);
  right_rear_bump = SensorValue(right_rear);
```

```
  if (front_left_bump == 1 || front_right_bump == 1 || right_front_bump == 1 || right_rear_bump == 1)
  {
                  return true;
  }
  else
  {
                  return false;
  }
}



// Move for a given amount of time in direction left (-1), forward (0), right (1), backwards(2), soft
left(3), or soft right(4)
void move(int dir, int time, int speed)
{
  switch(dir)
    {
       case -1: // Left
          motor[leftMotor] = -speed;
          motor[rightMotor] = speed;
          break;
       case 0: // Forwards
          motor[leftMotor] = speed;
          motor[rightMotor] = speed;
          break;
       case 1: // Right
          motor[leftMotor] = speed;
          motor[rightMotor] = -speed;
          break;
       case 2: // Backwards
          motor[leftMotor] = -speed;
          motor[rightMotor] = -speed;
          break;
       case 3: // Soft Left
          motor[leftMotor] = 0;
          motor[rightMotor] = speed;
          break;
       case 4: // Soft Right
          motor[leftMotor] = speed;
          motor[rightMotor] = 0;
          break;
       default:
          // Do nothing
          break;
    }
```

```
    // Wait for passed time then stop
    wait1Msec(time);
    motor[leftMotor] = 0;
    motor[rightMotor] = 0;
}









// Turn and move forward in search for wall
void start_maze(int f_speed)
{
                    if(prev_state == 1 || prev_state == 2)
                    {
                       // Turn right
                       move(1,400,turn_speed);
                       wait1Msec(100);
                       // Move forward
                       move(0,100,f_speed);
                       wait1Msec(100);
                    }
                    else
                    {
                       // Turn right
                       move(1,100,turn_speed);
                       wait1Msec(200);
                       // Move forward
                       move(0,1000,f_speed);
                       wait1Msec(200);
                    }
}

// State 1. To be put in this state one right touch sensor must be activated
void side_wall_correct()
{
    // Right side contact
    if(right_front_bump == 1 || right_rear_bump == 1)
    {
       // Optimal - do nothing
       if (right_front_bump == 1 && right_rear_bump == 1)
       {
          return;
       }
       // Angled at wall - soft turn left
```

```
    if(right_front_bump == 1)
    {
      move(3,300,soft_turn_speed);
    }
    // Angled away from wall - soft turn right
    if(right_rear_bump == 1)
    {
            int counter = 0;
            while(!sensor_contact_turn() && counter < state_threshold)
            {
                    move(4,200,turn_speed);
                    counter++;
                    if(counter >= state_threshold)
                    {
                            move(2,400,soft_turn_speed);
                    }
            }
    }
  }
}

// State 3. To be put in this state one front touch sensor must be activated
void front_wall_correct()
{
  // Front sensor
  if (front_left_bump == 1 && front_right_bump == 1)
  {
      // Move backwards then turn left
      move(2,300,turn_speed);
      wait1Msec(100);
      move(-1,650,turn_speed);
      wait1Msec(100);
  }
  else if (front_right_bump == 1)
  {
      move(-1,650,soft_turn_speed);
   }
  else // Left front is tripped
  {
      if (right_front_bump == 1 || right_rear_bump == 1)
      {
            move(2,300,turn_speed);
            wait1Msec(100);
             move(-1,550,turn_speed); // CHANGED
             wait1Msec(100);
      }
      else
      {
            move(1,300,soft_turn_speed);
```

```
        }
    }
}

// We're stuck in some state. Back up and turn left
void escape()
{
                move(2,600,turn_speed);
                move(-1,600,turn_speed);
}




// State 0. Turn and move forward in search for wall
void find_wall()
{
                int counter = 0;
                while(!sensor_contact() && counter < 12)
                {
                                move(4,70,turn_speed);
                                counter++;
                }
                move(0,500,forward_speed);
}

// Call the function affiliated with the passed state
void router(int state)
{
   switch(state)
   {
      case 0: // Find wall
         find_wall();
         break;
      case 1: // Postion self on wall
         side_wall_correct();
         break;
      case 2: // Full speed ahead
         move(0,70,hyper_speed);
         break;
      case 3:
         front_wall_correct();
         break;
       default:
         // TODO
         break;
   }
}
```

```
/* Assignment 4
States:
0: Find wall
1: Wall correct
2: Full speed ahead
3: Hit forward wall
4:
*/




task main()
{
   wait1Msec(2000); // give stuff time to turn on

   state = 0;
   state_counter = 0;

   while(state == 0)
   {
      start_maze(forward_speed);

      // Get current state
         front_left_bump = SensorValue(front_left);
         front_right_bump = SensorValue(front_right);
          right_front_bump = SensorValue(right_front);
          right_rear_bump = SensorValue(right_rear);

      if (front_left_bump == 1 || front_right_bump == 1 || right_front_bump == 1 || right_rear_bump ==
1)
                        {
                           state = 1;
                        }
   }

   // Main control loop
   while(true)
   {
      writeDebugStreamLine("State: %d", state);
      writeDebugStreamLine("Prev state: %.6f", prev_state);
      writeDebugStreamLine("State counter: %.6f", state_counter);

      // Check if stuck in current state
     if(prev_state == state)
     {
         writeDebugStreamLine("should increment");
```

```
        state_counter++;
      }
     else
     {
        state_counter = 0;
     }

     if(state_counter > state_threshold)
    {
        if(state == 1)
        {
                if (right_front_bump == 1)
                {
                        move(-1,200,turn_speed);
                }
                else
                {
                        move(1,200,turn_speed);
                }
        }
        else if(state == 2)
        {
                move(-2,250,forward_speed);
                move(0,300,hyper_speed);
        }
        else
        {
                escape();
        }
        state_counter = 0;
     }

      // Get current state
      front_left_bump = SensorValue(front_left);
      front_right_bump = SensorValue(front_right);
      right_front_bump = SensorValue(right_front);
      right_rear_bump = SensorValue(right_rear);

      // Any touch sensor activated
     if (front_left_bump == 1 || front_right_bump == 1 || right_front_bump == 1 || right_rear_bump ==
1)
   {
          // First check front sensors
          if (front_left_bump == 1 || front_right_bump == 1)
          {
              prev_state = state;
              state = 3;
              state_counter += 5;
           }
```

```
            // Then both side sensors
            else if (right_front_bump == 1 && right_rear_bump == 1)
            {
                prev_state = state;
                state = 2;
                state_counter -= .85;
             }
              // Else wall correct
              else
              {
                  prev_state = state;
                  state = 1;
              }
        }
        // Find wall if nothing is touching
        else
        {
            prev_state = state;
            state = 0;
            state_counter += 3;
         }

          router(state);
    }
}
```