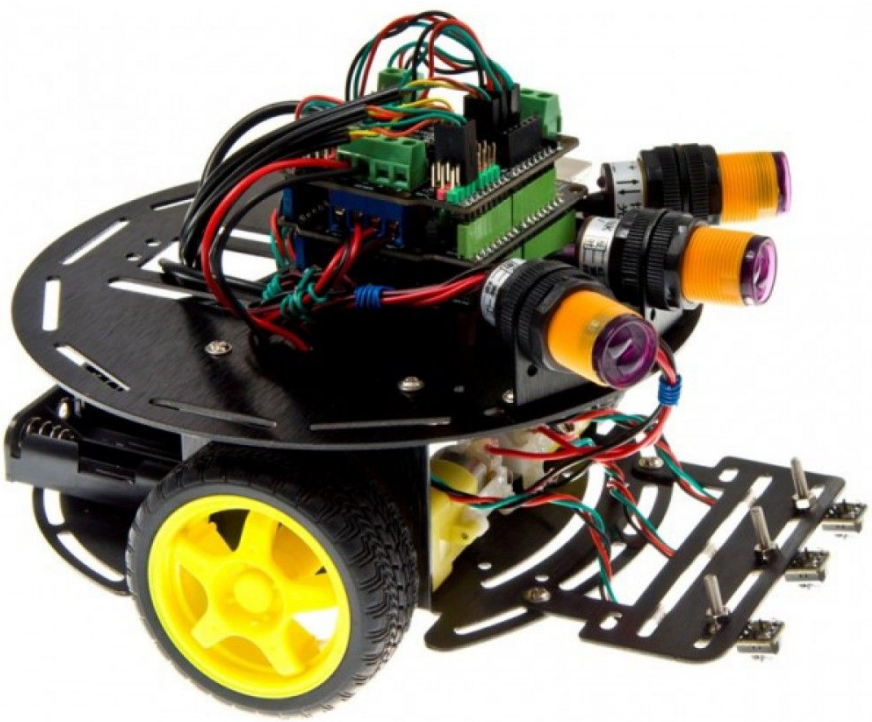
EEE3099S 2021

Milestone 3:

Done By:
Luke Baatjes (BTJLUK001)
Jonathan Campbell (CMPJON005)



Date of Submission:
18 October 2021

Project Description

The aim of this project was to design a line-following robot in the Matlab and Simulink environment. The first milestone was a guide in that it helps to familiarise the student in controlling a robot using simulation software. To demonstrate this, the robot needed to move in a straight line and stop after travelling one meter within 10% error. The robot also needed to be capable of rotating 90, 180, and 270 degrees within a 10% error of those values.

To understand Matlab and Simulink, onramp courses were compulsory before beginning the project.. The robots are then simulated in simulink using software compatible with an arduino. This was done using the Robotics simulator blocks that have encoder simulators and robot simulators, to simulate control of the robot.

Once the robot showed acceptable control in simulation, digital output and input blocks were used to verify the control on the Romeo V2 board using simulink software, compatible with arduino. The robotic platform makes use of the Romeo V2 board, which is compatible with the Arduino Leonardo board. Therefore, the Simulink Support Package for Arduino Hardware Library was used to communicate with the robotic platform.

The second milestone introduced the use of stateflow to control the entire motion of the robot as well as interfacing with sensors. In this milestone the robot simply had the task of detecting what kind of junction it came across. In this milestone however it was an integration of the previous two milestones in that the robot had to move forward while tracking a line and then detecting a junction (milestone 2) and then rotating (milestone 1).

Maze Learning Algorithm

For the maze learning algorithm, the Left Hand wall following method was used. This means that when the robot comes to a junction the priority is to turn left, if it can't turn left it will choose to go straight, if it can't go straight or left it will either turn around 180 degrees or go right (provided the option is available). This only happens at a junction, if the junction is strictly a right turn or left turn instead of a right T junction or left T junction, the robot simply moves in the direction of the turn. When it reaches a dead-end it will then just turn around and continue tracking the line until it detects a new junction.

Rotation Module Change

During the implementation of our rotation simulink model from milestone 1 along with the junction detection model from milestone 2, it was evident that the way the rotation module was designed, would cause unwanted complications during milestone 3. The rotation module from milestone 1 was originally designed using a proportional controller to control the distance that the robot travels and then outputting a velocity value with each wheel rotating in opposite directions. At the time, we had overlooked the possibility of using angular velocity values to drive the wheel motors. Therefore, the rotation module used in milestone 3 was implemented using angular velocity values, w, instead of velocity, v. This allowed for a more versatile and simple control of the robot's wheel velocities while rotating.

Maze Learning module design

Using the new idea of the rotation module described above and the junction detection from milestone 2, superstates were used to transition between a line tracking state, and a junction detection state. The use of superstates allowed for a fluid and sequential transition between states i.e. The robot would move off starting by tracking the line (line tracking state) and once the sensors determined that a junction was detected the next superstate would be entered where the type of junction was determined. From this superstate the robot moves on to rotation and then transitions back to line tracking. Using this method, the robot traverses through the entire maze eventually reaching the end of the maze. Next the method of storing all the turns taken will be described.

Once the robot has detected a junction and then rotated accordingly, an ASSCI value corresponding to the direction the robot has turned is output from the state block and sent via a "to workspace" library block in simulink, to the matlab workspace. For example, if the robot were to turn left, an ASSCI value for the string "L", 76, would be output, if the robot were to turn right, an ASSCI value for the string "R", 82, would be output, if the robot were to go straight, an ASSCI value for the letter "S", 83, would be output, and if the robot were to turn back 180 degrees, an ASSCI value for the letter "B", 66, would be output. This is done for every junction detected until the robot reaches the end of the maze. Once the end of the maze is reached, all the robot's rotational decisions are available in the matlab workspace in a variable called "out". This variable is structured as a column vector holding all the values at the point the robot has rotated, along with many unwanted zeros.

To remove these unwanted zeros to obtain a vector strictly containing the robot's turn decisions, we use another matlab script to remove the zeros using for loops. The first for loop is responsible for counting the number of nonzero numbers in order to specify the size of an array created later. The second for loop is used to append the nonzero numbers to an array using the size parameter determined from the previous for loop. We now have an array strictly filled with numbers corresponding to the decisions or turns the robot has made. This array is available through the matlab workspace. The robot has now learned the maze.

Shortest Path Algorithm

Shortest path module design

Now that we have an array consisting of the route travelled by the robot, we can access this array using a constant block in simulink and setting its parameter to the name of the array.

The shortest_path matlab function works by making a new zero array of length(input_array). It then iterates through the input_array value by value and checks if the element is a deadend represented by ASSCI value, 66. If it is not a dead-end the function then just saves the value in the new array at the position of the counter, which counts how many elements are in the new array.

If an element is a dead-end, the function looks at the previous value as well as the next value before and after the dead end. The shortest_path function then uses that information and can see what condition it falls under.

An example is when a robot has come to a junction with a dead-end on the left (Figure 1). It will first turn left because of left hand following, then come to the dead-end which it will turn around and come to the junction which we will turn left back on to the same line that the robot was on earlier.

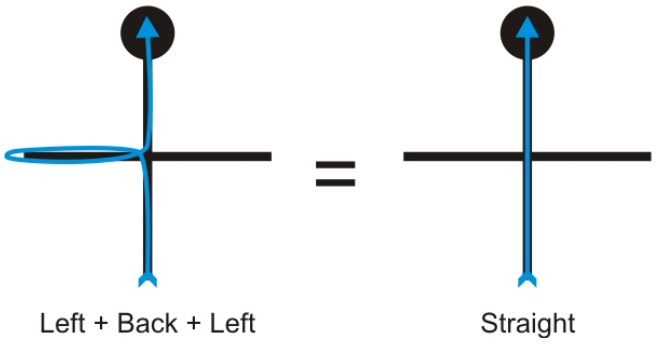


Figure 1

In the array the shortest_path function will see the ASSCI values, 76 66 76. Looking at the maze the robot does not need to go left all it needs to do is go straight this will simplify the maze array by just going straight. The function recognizes this configuration and adds 83 to a new array. And does not add the previous left turn, 76, and the next left turn, 76.

Here is a list of simplifications that the shortest_path uses to get the shortest path.

- LBR = B
- LBS = R
- RBL = B
- SBL = R
- SBS = B
- LBL = S

It can be seen that there are more than one order of turns that output Back. This is solved by running the shortest_path function again with a while loop.

When all the dead-ends are removed and there is no new ‘Back’ the function then outputs the optimized function matrix.

Optimised module design

Allowed the opportunity for more time, we would have submitted the final optimised file where the robot follows and completes the maze, following the optimised path determined. For the sake of completeness we will describe the method used to achieve this. Similar to the way in which the robot has learned the maze, we will control the motion of the robot every time it detects a junction. Once a junction is detected the robot would look to the array for its instruction on which direction to move in. After the first iteration is complete, a counter is incremented and the next instruction can be accessed. Once the robot adheres to the instruction control passes back to the line tracking superstate and a junction is once again detected. This repeats until the robot reaches the end of the maze.