EEE3099S Design Project 2021:

Maze Solver

MILESTONE 4 – The Report

GROUP 39:

TLNNTH001

NLNCOC001

Submission date: 24 October 2021

INTRODUCTION

This project aims to develop an autonomous mobile robot. The robot needs to navigate through a maze while learning the maze and perform maze solving operations. The project will be divided into two phases. Phase one involves the robot learning the maze and calculating the shortest path through the maze. This will be accomplished using simulations on MatLab and Simulink. Simulink will also be used for implementation using a Romeo V2.2 (compatible with Arduino Leonardo) robot. The second phase concerns the optimisation of the path the robot takes.

In the first phase whereby the robot learns the maze it must start to operate (move) at the push of a button. The robot must indicate using a LED the mode in which it is operating. The robot must learn the maze and show that it has completed the process while stopping at the end of the maze. The end of the maze is designed as a solid black box.

In the second phase the robot must optimise the path through the maze so as to find the shortest route to the end of the maze. Then the robot must indicate that it has found the shortest path through the maze using a blinking light emitting diode. The robot should be able to indicate that it can travel through the maze via the optimised path.

HARDWARE DESCRIPTION:

We used a self-powered differential drive robot. The robot uses a ROMEO V2 – a robot control board with Motor Driver. The board is compatible with Arduino. The DC motor driver allows us to start the project instantly without the need of an extra motor driver. It behaves like an Arduino Leonardo based on the ATmega32u4 chip.

The robot uses a 2WD mobile platform for Arduino which is a small and inexpensive mobile platform. It is used with a standard Arduino microcontroller. The robot is supplied as a kit that includes two drive motors, wheels (and rear roller), frame and all mounting accessories.

The robot uses four Gravity: Digital Line Tracking (Following) Sensors. The sensors can detect the white lines on a black background and the black lines on a white background. The maze used in this project is made up of a single black line thus the environment is considered as the white background. The robot has two Gravity: TT Motor Encoders Kits.

The hardware is connected via USB and used along with Simulink models to implement the maze solving operations. The Arduino support package on Simulink is used for implementation. The individual hardware components are further explained in the subsystems they are used.

MOTION CONTROL DESIGN, IMPLEMENTATION AND RESULTS

MOTION CONTROL MODEL:

REQUIREMENTS

- The model should move the robot a specified distance of one meter and stop
- The model should rotate the robot for specified angles of: 90, 180, 270 degrees.

SPECIFICATIONS

- Use Mobile Robotic Training Library on Simulink for simulations
- Use Simulink Support Package for Arduino Hardware Library for implementation

MOTION CONTROL SUBSYSTEMS:

Robot kinematic model and description

The robot uses a differential drive. The differential wheel robot is a mobile robot whose motion is based on two individually driven wheels placed on both sides of the robot's body. Therefore, it can change the direction by changing the relative rotation speed of the wheels, so no additional steering movement is required. Pure translation occurs when the two wheels move at the same angular velocity. When the wheels move at the opposite speeds, pure rotation occurs. In order to build a simple differential drive constraint model, only the distance between the two wheels (axle length) and the wheel radius are needed.

For our model:

- Axle length: 13.6 cm
- Wheel Diameter: 6.2 cm

The velocity vector ω = (ωr , ωl) , where ωr is the right wheel angular velocity and ωl is the left wheel angular velocity [in radians per second]. The distance travelled by the robot in translation (ωr = ωl) is equal to the time taken multiplied by the wheel radius*ωl . IE d = t × ωl × r [meters]. Where v = ω × r.

Let the total angular velocity of the robot, ω = $\frac{1}{2}$ (ωr + ωl). Therefore, the distance travelled is equal to the total angular velocity multiplied by the wheel radius*time. For rotation to occur we need to find the difference between the angular velocities. Let $\omega\forall$ = (ωr − ωl). The rate of change of rotation, θ = $\frac{\text{wheel radius}}{\text{axle length}}$ × ($\omega\forall$). The rate of change of rotation is dependent on the translation velocity as shown.

Encoder Model and description

A rotary encoder is an electro-mechanical device that converts the angular position, or movement of a shaft into an analogue or digital output signal.

It operates using two lasers that project light onto a ring around each drive wheel axis. The ring is cut perpendicular to the rotation. Therefore, when the shaft and ring rotate, the light receiver detects the light pulse. The encoder is able to determine whether the wheel is turning forwards or backwards by offsetting the two lasers relative to each other. The encoder counts pulses or ticks to determine the distance the wheel travels.

$$D = 2 \times \pi \times wheel\ radius \times \frac{\Delta \text{tick}}{N}$$

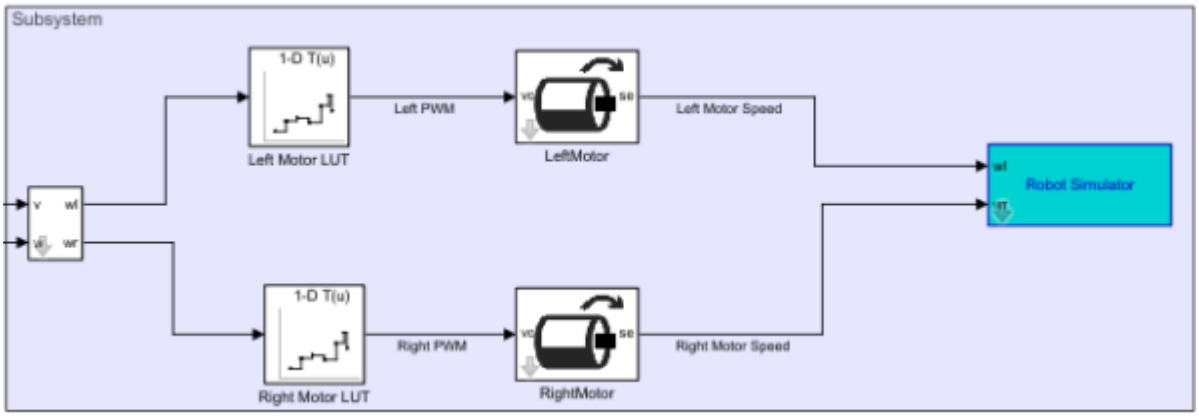
Where: N is the number of ticks per revolution

$$\Delta tick = tick - old\ tick$$

We used the encoder from the mobile robotics training Simulink toolbox for our simulations. N = 20 ticks.

PWM explanation

Pulse Width Modulation is a method of decreasing the average power delivered by an electrical signal. The signal is divided into discrete parts to achieve this. The voltage (average) supplied to the load is controlled by quickly opening and closing the switch between the power supply and the load.



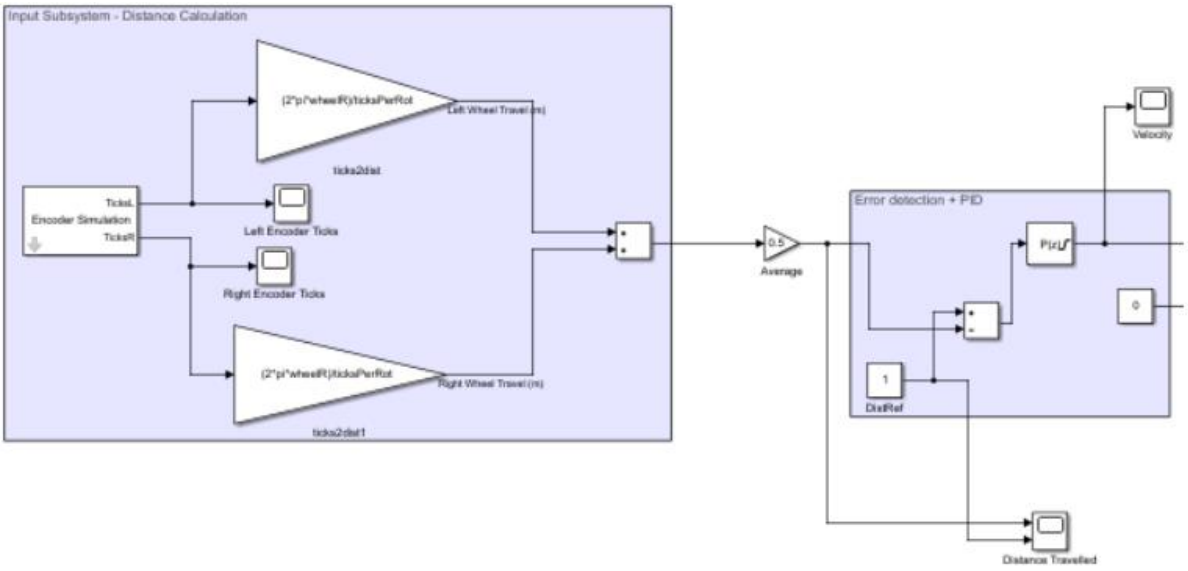
The figure above shows the subsystem that includes PWM for the robot simulations.

How the subsystem works:

The “wlwr” block (left most block) converts robot linear velocity (v) and angular velocity (w) into right wheel angular velocity (wr) and left wheel angular velocity (wl) using our robot parameters. Using a look up table (LUT) for each wheel with the table data being the “inputPWM”. Whereby “inputPWM” is an array [1 173] of doubles. Therefore, the output of each LUT is a constant value (before experiencing breakpoints). The output is then supplied to the motors in Volts for the operation of the simulator.

Distance Control Algorithm

The distance travelled by the robot is achieved using the following subsystem:

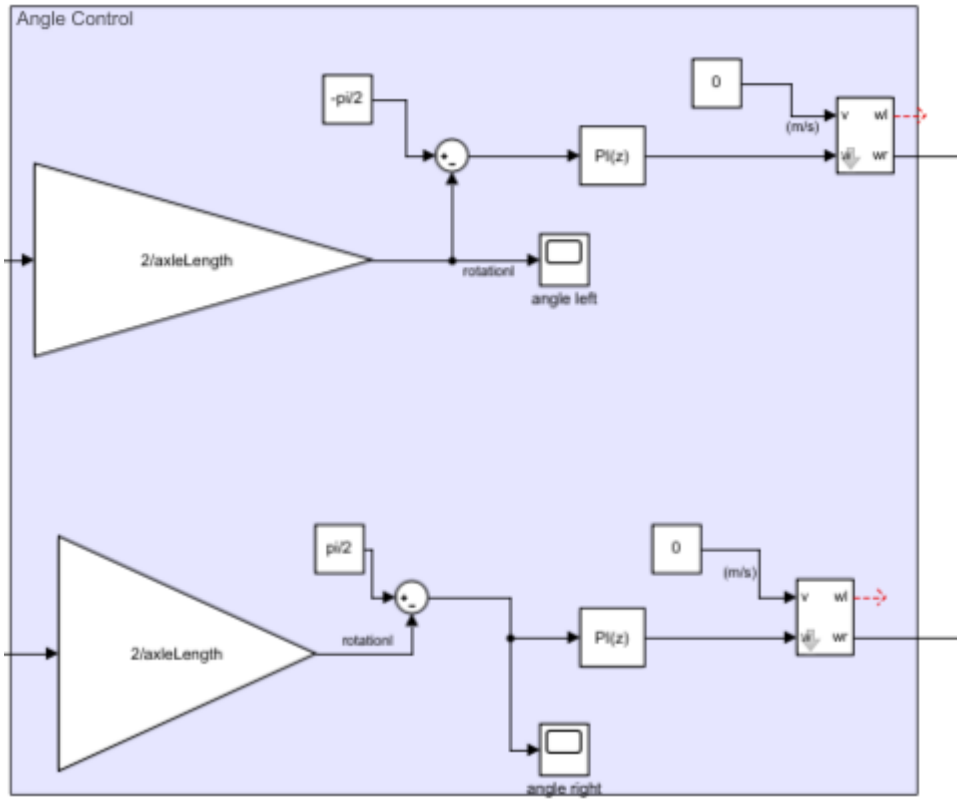


In the Distance Calculation subsystem the encoder measures the rotation of the right and left motors in ticks. The gain blocks for each wheel calculates the distance travelled by each wheel using: $d = \frac{2\times\pi\times\text{wheel radius}}{\text{ticks per rotation}}$. The individual distances travelled by each wheel are the added and divided by two $D = \frac{1}{2}(dl + dr)$ to find the average which is the actual distance travelled by the robot.

Our robot is supposed to travel for 0ne meter and then stop. A constant block named “DistRef” is added to the error detection + PID subsystem to represent this. We subtracted the calculated distance, D from the reference distance, DistRef to get an ouput that was fed into a PID block. We used a P controller. We used this method to prevent our robot from overshooting the reference distance before settling down to one metre. This is achieved by gradually decreasing the voltage before reaching the one metre mark instead of only reducing the voltage after reaching the desired distance. The output to the PID block is a velocity.

Angle Control Algorithm

The angle was controlled using the following subsystem:



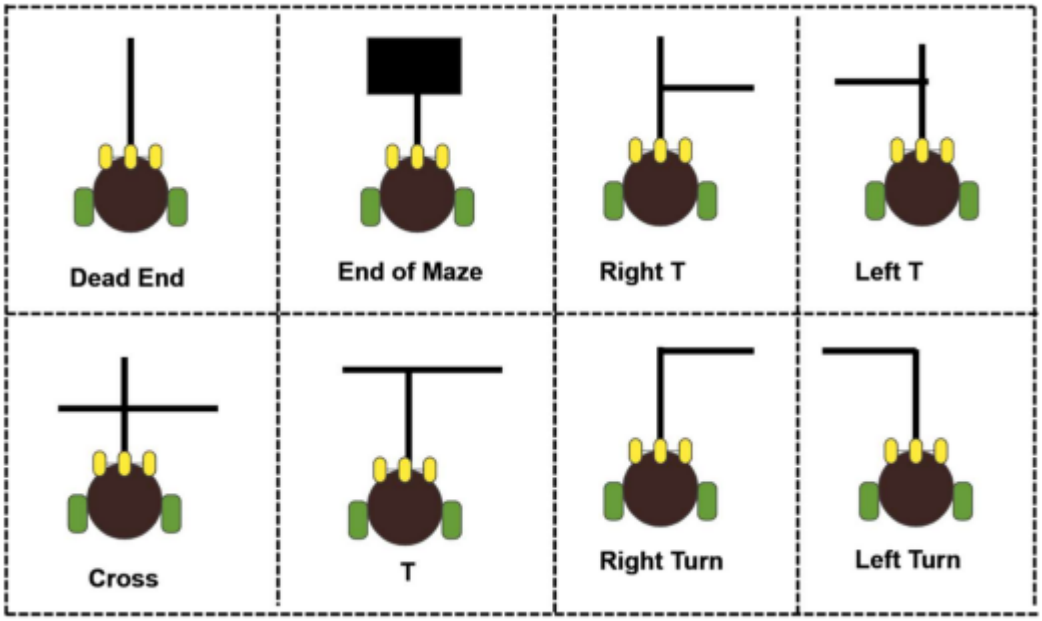
This subsystem uses an algorithm similar to that of the distance control where the only difference here is that the angle of rotation of the robot is what is being controlled. The counted ticks are used as an input for the subsystem for the left wheel and the right wheel individually. This is done for error correction purposes with regard to the distance travelled by each wheel. The output to the gain block, (2/axle length) gives the rotation in radians per second for each wheel. The angle is therefore subtracted from a reference angle (pi/2). The output of this is fed into a PI controller for error detection and to linearize the output. The output is an angular velocity which is fed into the “to wlwr” block.

LINE SENSING DESIGN, IMPLEMENTATION, AND RESULTS

LINE SENSING:

REQUIREMENTS:

- Robot should be able to detect a black line
- Robot is required to sense the following maze junctions (line configurations):



SPECIFICATIONS:

- Use Mobile Robotics Training Library on Simulink for simulations
- Use Simulink Support Package for Arduino Hardware Library for implementation
- Use 4 x Gravity: Digital Line Tracking (Following) Sensors
- 18 mm track size

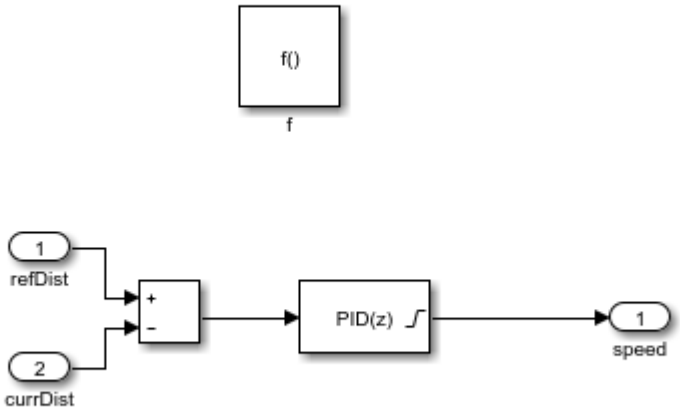
LINE SENSING SUBSYSTEMS

Line sensing:

The robot uses Digital Line Tracking (Following) Sensors. These sensors detect whether the robot is on a black line (maze) or not (which would be the environment). The sensors outputs binary digits whereby 0 indicates the line and 1 indicates the environment. We use a threshold of $\frac{environment+line}{2} = 0.5$ to position the sensor such half of it is on the line and half of it is on the environment.

We use the line sensor in the mobile robotics library to represent the sensors on the robot. The block parameters are environment value= 1 and line value= 0 and the offset is an array of the relative positions of the sensors.

We used a Simulink function to control the velocity of the robot. This function ensures that the robot stops after the distance given. Speed = $comp\_Dist(refDist, currDist)$.



Shown above is a screenshot of the stateflow function. We used a PID for error detection and correction.

We used a stateflow chart with conditional statements for our algorithms.

There are 4 sensors on the robot at the following positions:

1. (x: 64.3mm, y: 35.4mm),
2. (x: 67.5mm, y: 5mm),
3. (x: 67.5mm, y: (-)5mm) and
4. (x: 64.3mm, y: (-)35.4mm)

Sensor 1 = L_Out; Sensor 2= L_In; Sensor 3 = R_In; Sensor 4 = R_Out

Line following algorithm:

To follow the line (moving forward in a straight line) we set the velocity value to v = 0.2 and w = 0 (because we are not rotation in any direction). We set conditional statements in the case of needing to rotate left or right in order to stay on the line.

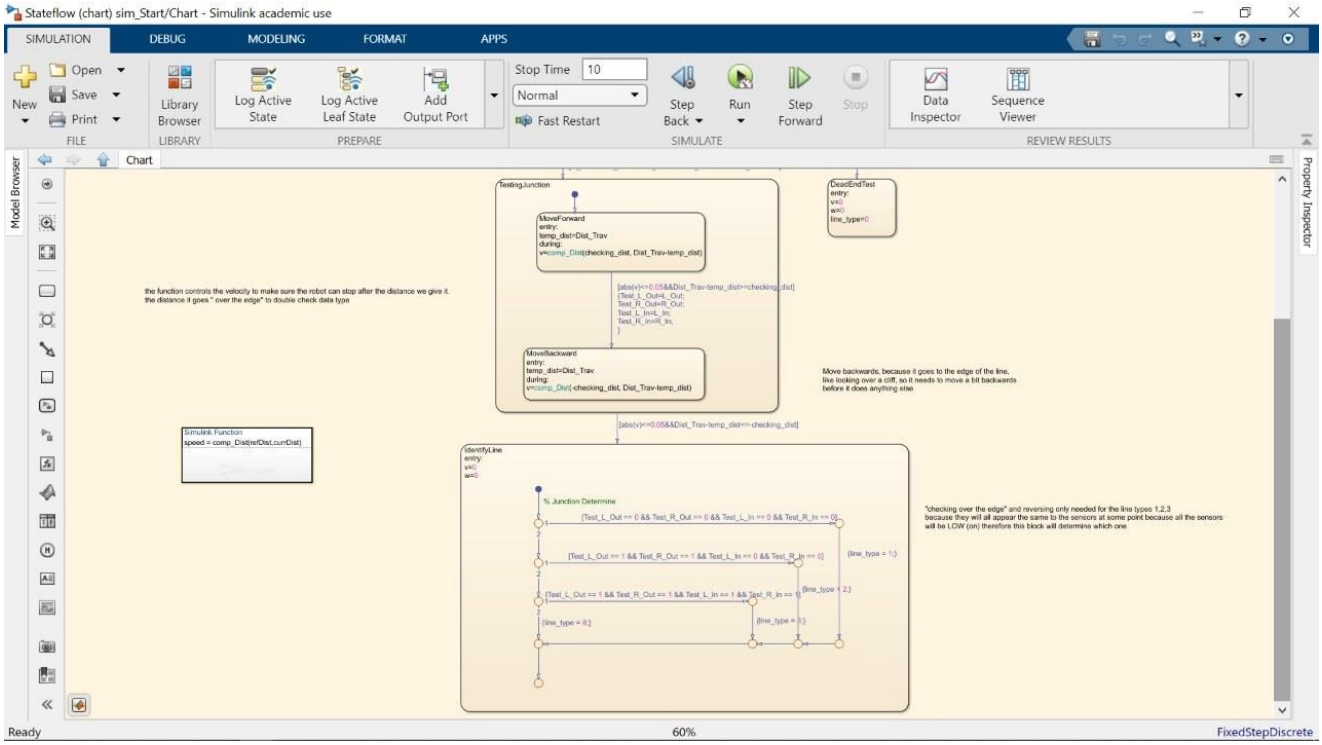
If [L_In = 0 and R_In = 1] i.e. the front sensors indicate that the robot is on the right edge of the line hence R_In is sensing the environment; the robot is supposed to rotate to the left (w = -1) . The values for L_In and R_In will therefore equal 0 signifying that the robot is on the line. Then the robot will continue moving straight forward.

If [L_In = 1 and R_In = 0] i.e. the front sensors indicate that the robot is on the left edge of the line hence L_In is sensing the environment; the robot is supposed to rotate to the right (w = 1) . The values for L_In and R_In will therefore equal 0 signifying that the robot is on the line. Then the robot will continue moving straight forward.

To follow the line the robot shuttles between these two states.

Line configuration algorithm

Using the line sensing information from the line sensor outputs a binary code indicating whether the sensor is on the line on the environment, with 1 being the environment and 0 being the line. A Simulink Stateflow is used to help indicate the different line configurations. With the help of the if-statements feeding into each state indicating the line configuration, the different states are established, that is whether the robot is approaching a T-junction, a left/right turn, and more.

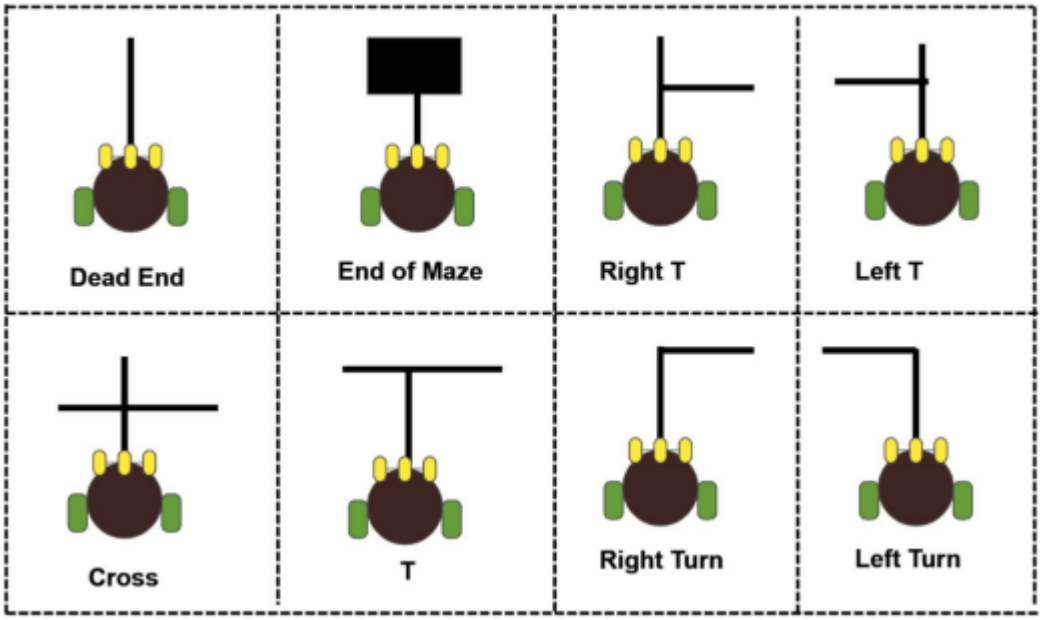


MAZE SOLVER AND SHORTEST PATHFINDER ALGORITHM DESIGN, IMPLEMENTATION, AND RESULTS

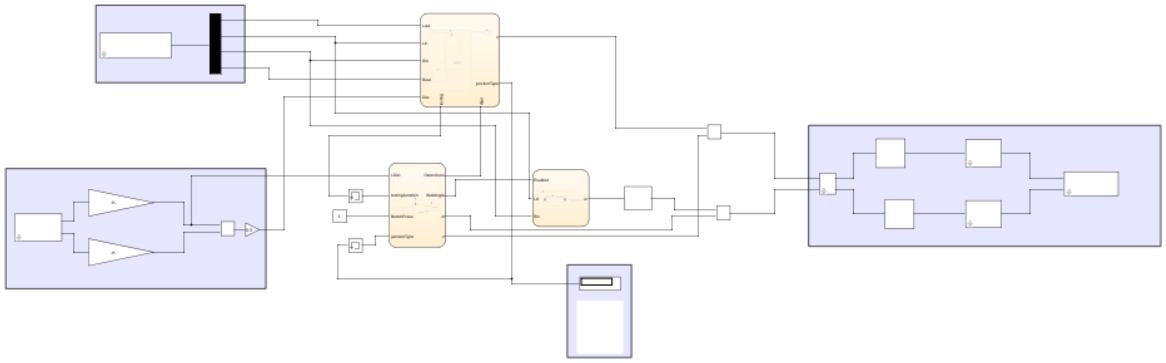
MAZE LEARNING:

REQUIREMENTS:

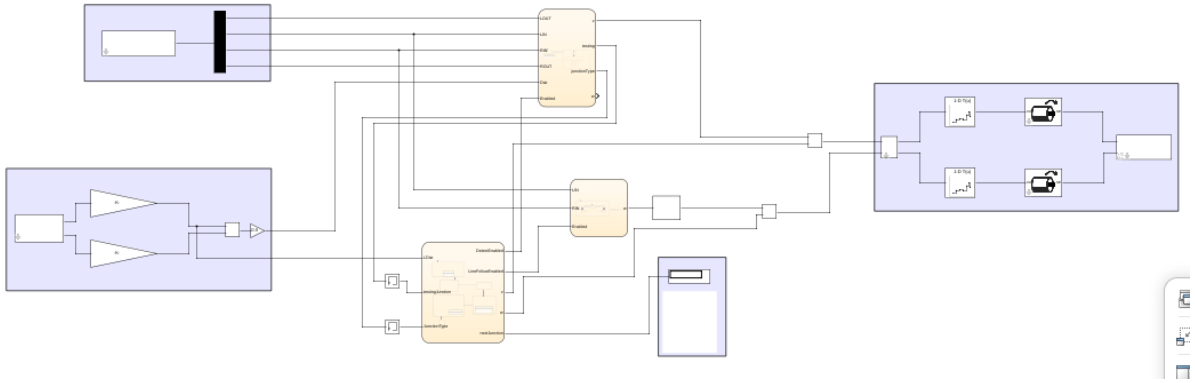
- Robot should be able to go through the maze
- Robot is required to record the following maze junctions:



Subsystem Block Diagrams:



Shown above is a screenshot of the maze learning algorithm Simulink file



Shown above is a screenshot of the shortest path algorithm Simulink file

Maze learning algorithm:

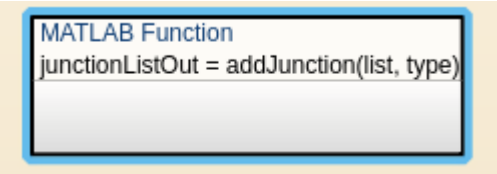
The maze learning algorithm builds up from the results from milestone 2 (line sensing). The maze learning algorithm implemented is a left wall following algorithm also known as LSRB algorithm. The algorithm works according to the following steps:

1. If there is a left turn, turn left
2. Else if continue moving straight ahead
3. Else if turn right
4. If at a dead end then the robot must turn around.

When the robot is at any intersection (junction) it follows the LSRB algorithm to decide in which direction it will. The result (decisions) of the path travelled is then stored in memory therefore learning the maze. The algorithm works so long as there are no loops in the maze.

Shown below is the MazeContoller state chart used for simulations in Simulink. From milestone 2 -line sensing the robot detects the type of junction it has reached, and the information is used for the LSRB algorithm.

```
MazeController
entry:
WaitAngle = 1
DetectJunc = 1
% if there is a left turn, turn left
if junctionType == 8 || junctionType == 6 || junctionType == 5 || junctionType == 4
   Rotate = 90;
end
% if at a dead end turn around
if junctionType == 1
   Rotate = -180;
end
% if there is a right turn, turn right
if junctionType == 7
   Rotate = -90;
end
% when at the end of the maze, stop
if junctionType == 2
   MazeComplete = 1;
end
% else just go straight aka do nothing
junctionList = addJunction(junctionList, junctionType) % add junction type to an array (list)
```

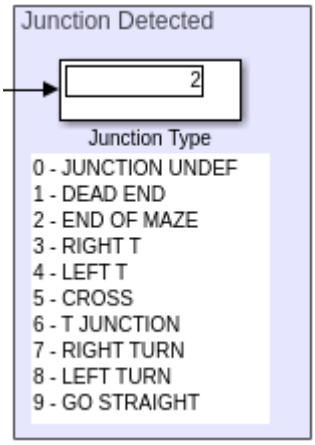



```
function junctionListOut = addJunction(list, type)
    junctionListOut = [list; type];
end
```

Shortest path algorithm

The shortest path travelled is obtained using the list in memory from the maze learning algorithm. The type of junction that the robot encountered is stored in the list (matlab function shown in the figure above).

The shortest path algorithm uses the list as an input. The type of junction encountered is stored as a number according to the figure shown below.

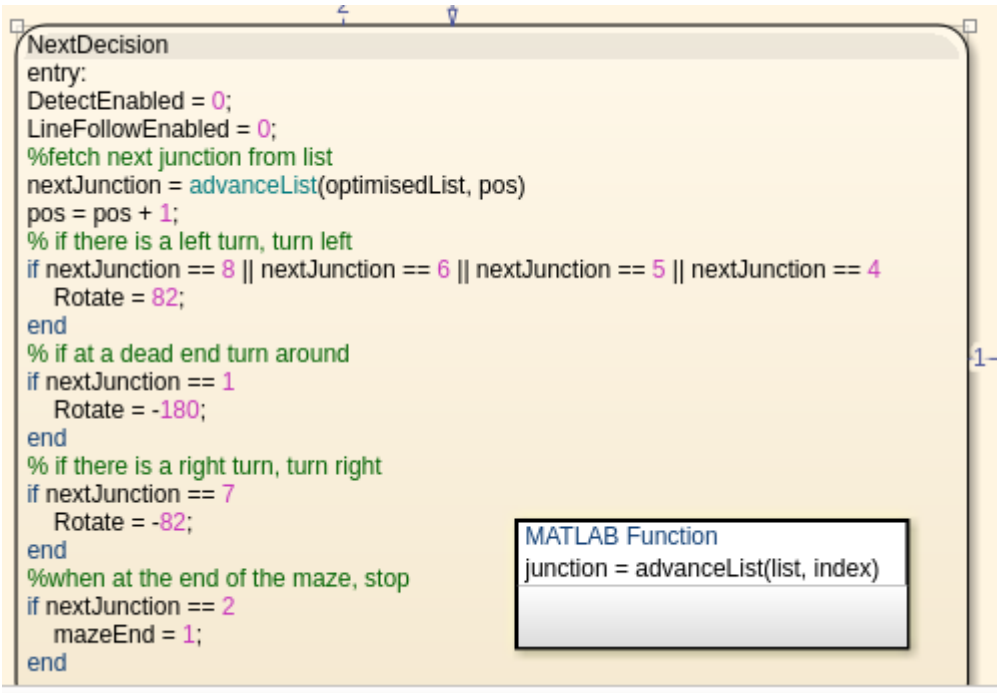


Two types of data values are flagged:

1. Paths that can lead to permutations (example: RIGHT T (3), T JUNCTION (6), etc)
2. Paths that lead to a DEAD END (1)

For every dead end that the robot came across that is route is interpreted as a bad turn (more time consuming and does not lead to the end of the maze). From the flagged data values, the algorithm backtracks to the previous choice (turn) as it is interpreted as a bad choice. The path of that choice is therefore removed and replaced with a summing point. The summation point is the alternative path (turn) that the robot could have taken instead of making the "bad choice". By removing the wrong turns the robot is essentially reducing the time it took in order to reach the end of the maze while learning the maze, therefore finding the shortest path to the end of the maze.

The matlab function "optomiseMaze" implements the knowledge of the shortest path algorithm (in the controller state chart).



CONCLUSION

The robot was able to meet the requirements stated above. The robot however did not follow the path (black line) perfectly on the Simulink map nor during the demos of the robot. There are multiple possible causes for the errors which include that the surface the robot was moving on was not perfectly flat surface during the demos, that the encoders encountered errors while detecting the maze, our code had logical discrepancies.

Overall, the project was successful. The robot was able to learn the maze and indicate that it has done so. The time taken for the robot to go through the maze after optimising the path was shorter.

References

1. Arduino Information - https://www.dfrobot.com/product-844.html
2. Mobile Robotics Training - https://www.mathworks.com/videos/series/student-competition-mobilerobotics-training.html
3. Dead Reckoning Lab - http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/16311/www/s07/labs/NXTLabs/Lab%2
4. Information on Encoders - https://www.youtube.com/watch?v=oLBYHbLO8W0&ab_channel=SparkFunElectronics
5. https://www.mathworks.com/help/supportpkg/legomindstormsev3/examples/line-tracking.html
6. https://www.mathworks.com/help/supportpkg/arduino/examples/arduino-robot-line-follower-application.html
7. https://www.mathworks.com/help/supportpkg/legomindstormsev3/examples/line-tracking.html
8. https://www.mathworks.com/help/supportpkg/arduino/examples/arduino-robot-line-follower-application.html
9. Richard T. Vannoy II, “Line Maze Algorithm”, April 2009.
10. https://en.wikipedia.org/wiki/Maze-solving_algorithm