MAZE SOLVER ROBOT

Prepared by:

STLTSE004 & MVMAMA001
GROUP 36

REPORT

Milestone 4

1. INTRODUCTION

The concept of autonomous navigation is a very pivotal feature in mobile robotics that allows independent movement from a certain point to another without a tele-operator. This feature requires the mobile robot to first localize, explore, and map the surroundings on which it will be operated. A differential drive which is a drive system with two-wheels and independent actuators on each wheel, is employed in this project. The term "differential" refers to the postulation that the motion direction is the average sum of the independent wheel turning speeds.
This report describes the implementation of a small-sized Arduino Leonardo mobile robot designed to solve a maze using the "left hand on wall"-following algorithm. The Arduino Leonardo is a microcontroller board that is based on the ATmega32u4. The robot was programmed to move on the black lines of the maze as shown in Fig 2, and uses four optical sensors-as shown in fig 1 below, placed on the bottom of the robot- that are used to track the lines.

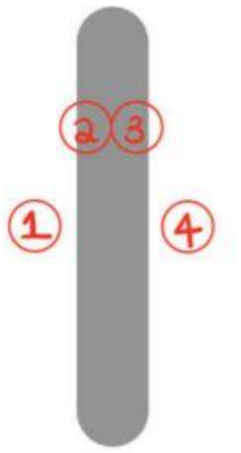


Fig1. Sketch of the four sensors on the mobile robot while the robot moves along a path (i.e. the grey line)

High level description of the "left hand on wall":

- Always turn to the left if it can.
- If it cannot left, go straight.
- If it cannot either turn left or go straight, it must turn right.
- If it cannot do any of the above, it must turn around because it has reached the dead end.

1.1 Objective I:

Learning the maze:-the robot needed to follow a black track(line), learn this track, and find the shortest path.

In this phase, the robot needed to start moving by pushing a configured button; it needed to further indicate, using an LED, it's mode of operation. Once the robot was done learning the maze, it then had to stop at the solid black rectangular block shown in fig2. Following this, it had to then indicate, with an LED, that it has learnt the track.

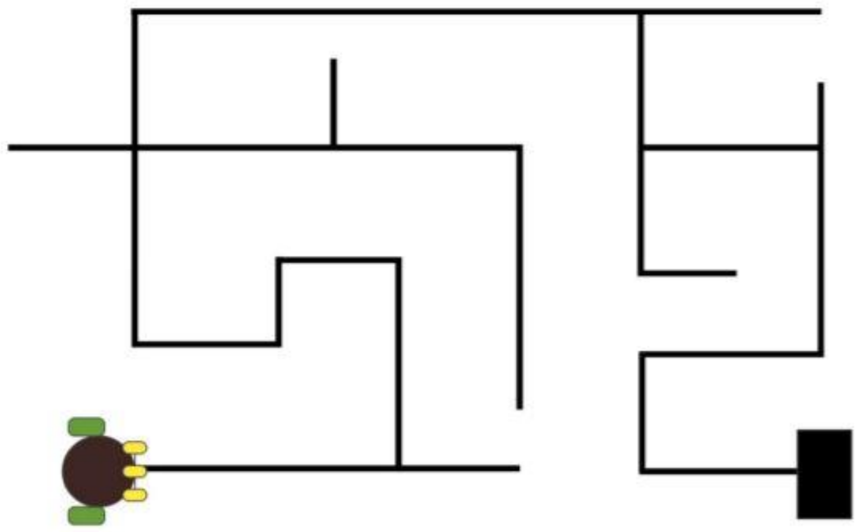


Fig 2. A sample maze

Technical specifications:

- The robot starts with a push of a button and indicates with an LED that it is learning the maze
- The robot needs to sense a line and follow it.
- The robot should implement a maze learning algorithm
- The robot should cease movement and indicate with an LED once it has learned the maze successfully
- The robot should be able to sense:
    - a cross
    - a dead-end
    - the end of the maze
    - a left/right T junction
    - a left/right turn

1.2 Objective II:

Optimizing and solving the maze:-The robot had to find the end of the maze using the shortest path and in the fastest time.

In this second phase, given any kind of map, the robot needs to optimize this path and figure out the shortest route to the end of the maze as shown in fig 3. The robot should indicate with an LED that the shortest path has been found.

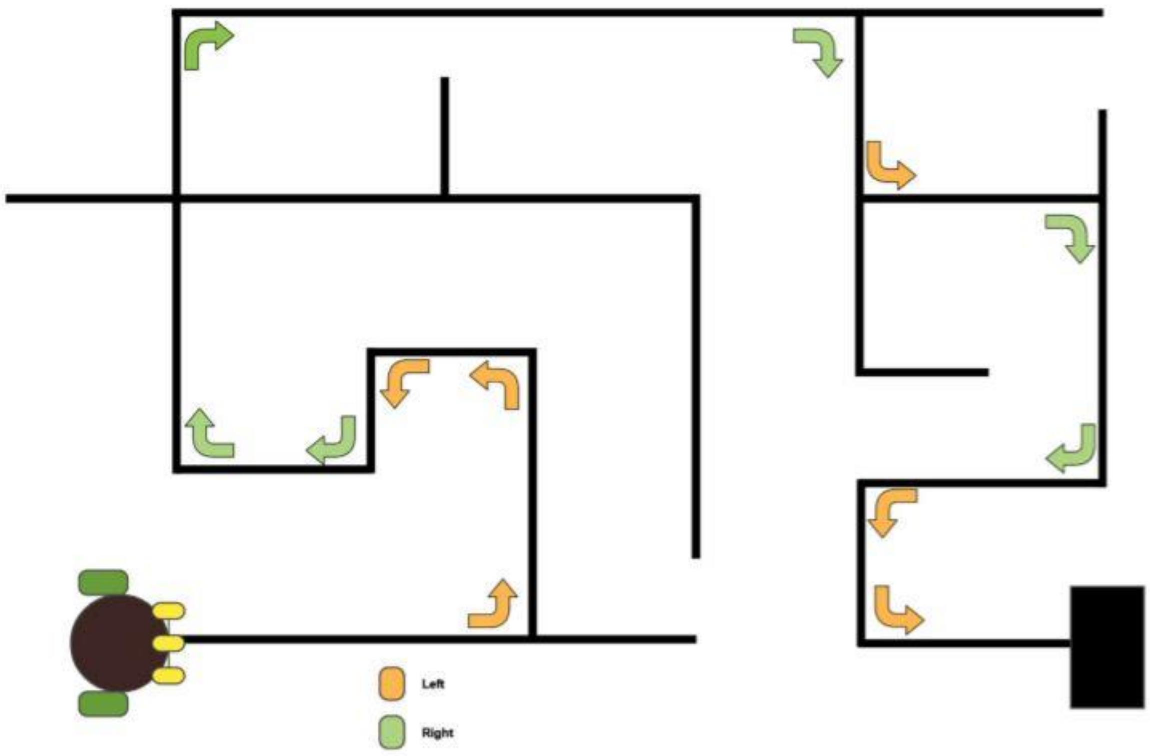


Fig 3. Sample optimized path

2. MATERIALS AND METHOD

2.1 Hardware description

The most cardinal hardware components of the robot are shown in fig 4 below.

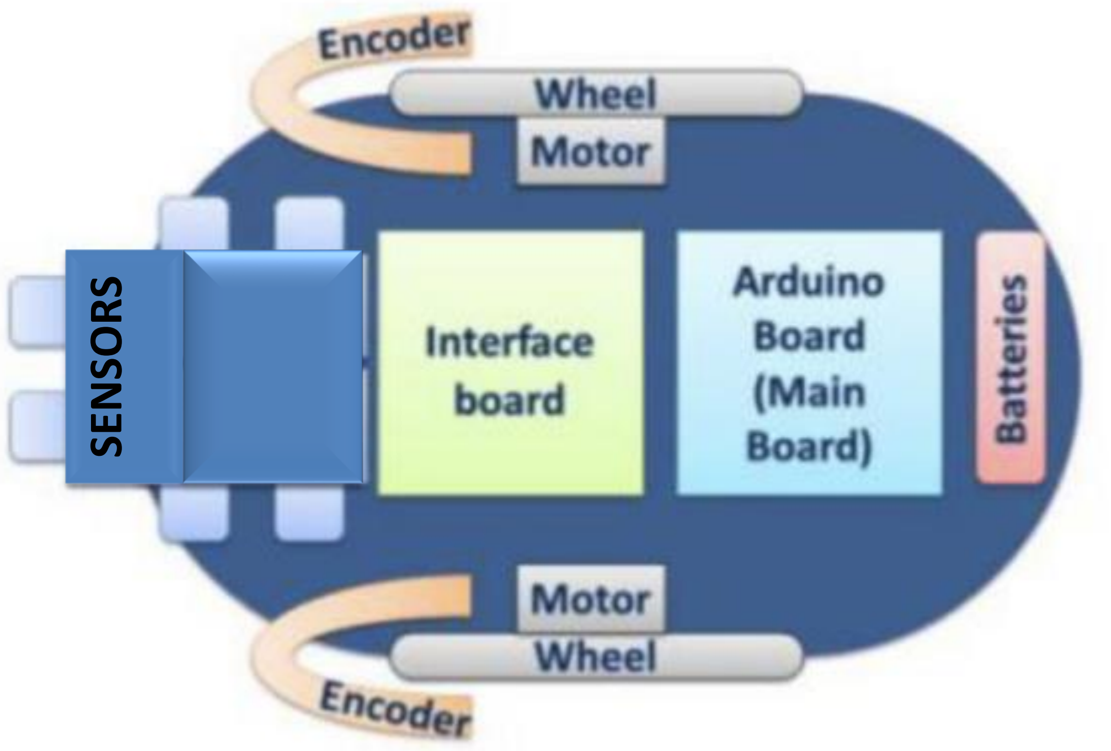


Fig 4. Hardware design

2.1.1 Control board

The Arduino board supplies the processing power for the robot. This board is powered by the ATmega32u4, which has a 32KB flash memory to store the code used to operate the robot. Each of the digital pins on the Arduino can be used as an input or output; operating at 5 volts. Some pins have specialized functions including the Serial, PWM, SPI, LED, and analog inputs. (Arduino, 2018)

2.1.2 Sensors

Line sensors operate in such a way that they detect the existence of a black line. This is achieved by emitting infrared light and subsequently capturing the level of light that returns to the sensors. Four sensors were placed on the front of the robot (as shown in fig 5 below), facing downwards to detect light. The logic in which this robot follows the line is focused mainly on the operation of the sensors. Sensors 1,2,3, and 4 have two different states- the high and low states(or 0 and 1). The sensors go low (0 state) whenever they detect (or "see") a black line, otherwise when they detect white space(the environment).

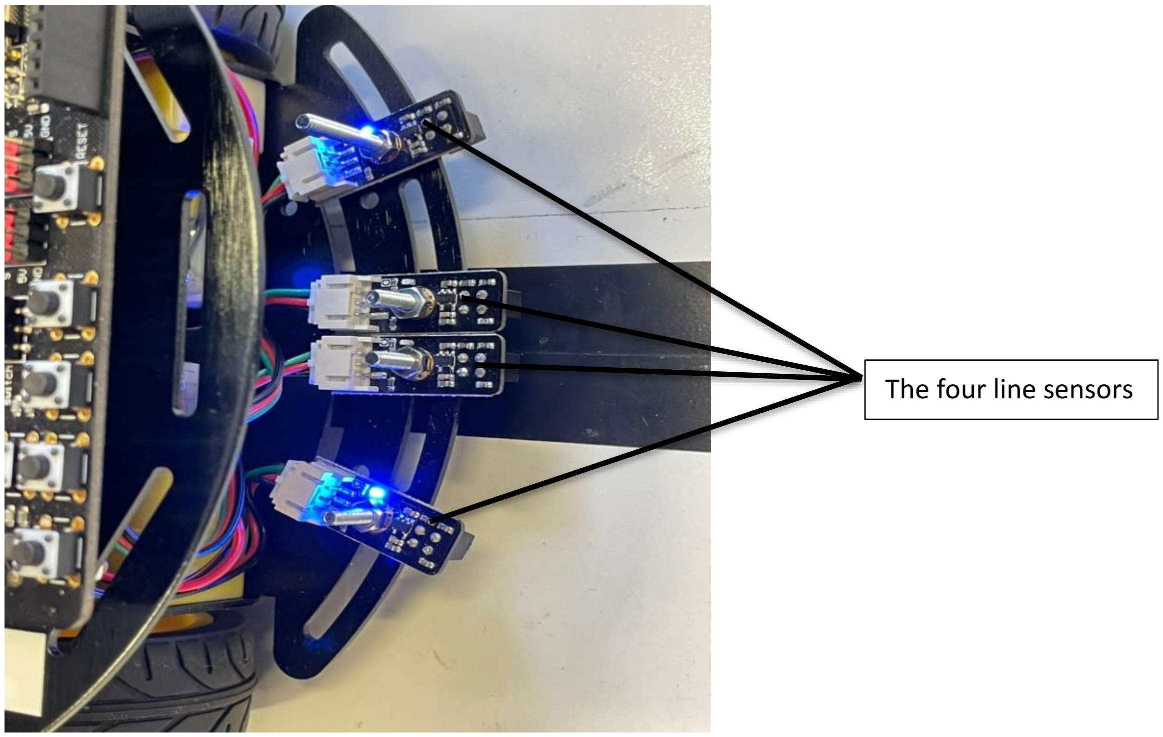
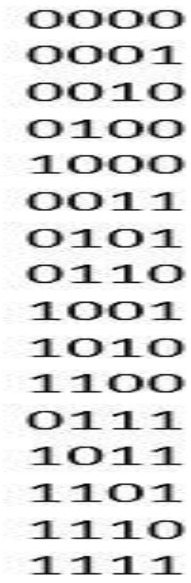


Fig 5. Robot line sensors

There are 16 (2^4)combinations of sensor states that the robot can occupy and they are shown in fig 6 below



0000
0001
0010
0100
1000
0011
0101
0110
1001
1010
1100
0111
1011
1101
1110
1111

Fig 6. Different states of the sensors.

The corresponding actions taken during their respective sensor states are outlined in the state chart shown in the appendix Fig 20.

2.1.3 The Chassis and Robot encoder:

Chassis parameters:

- wheel radius :- 0.031m : : Helpful in determining the distance moved by the wheel in one rotation(circumference).
- axle length :- 0.136m : : Essential parameter for determining the direction angle of the drive and how far it should rotate.

The chassis is chosen such that it can be able to navigate through the narrow maze, as it is very compact in design. This compact design also minimizes the costs of the robot hardware design. (Elshamarka, 2012)

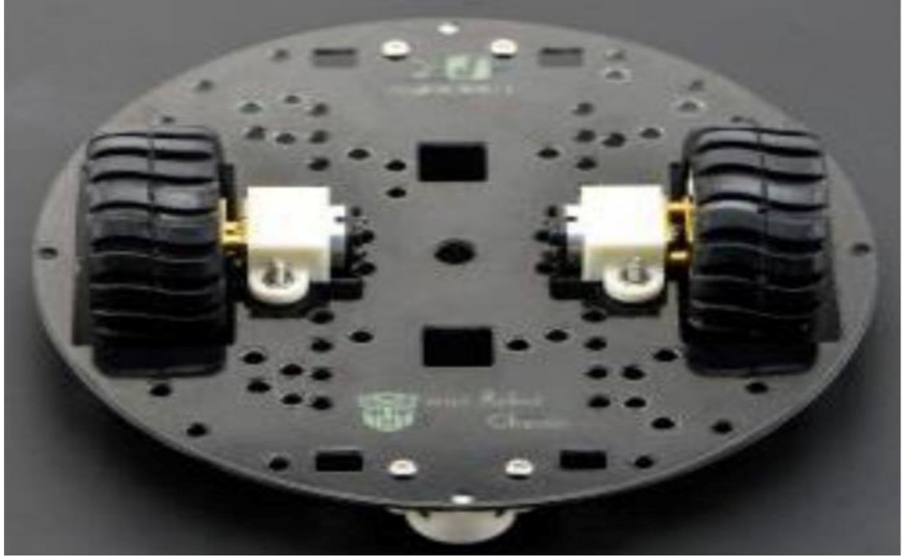


Fig 7. The robot chassis

The wheel rotation encoder is located near each of the wheels so that the extent to which the wheel rotates can be detected. The encoder is a pair of an infrared transmitter and a receiver. The encoder used is an Arduino binary incremental rotary optical encoder which uses laser lights shot through slightly misaligned slots to count ticks as the wheel rotates. The manner in which it works is that it models its output signal as a function of the motor(wheels) rotation. The encoder works in such a way that it provides electrical pulses into the drive model. It has two sensors and also produces an output of two pulses when the shaft is rotated. The pulses can then be counted to find out about the extent of the shaft rotation and the direction. The encoder outputs a square wave from which position and angle can be determined. Knowing information such as wheel radius and encoder resolution, it is then possible to determine the distance and speed of the robot. This is very useful in controlling the motion of the robot as described in the next section.

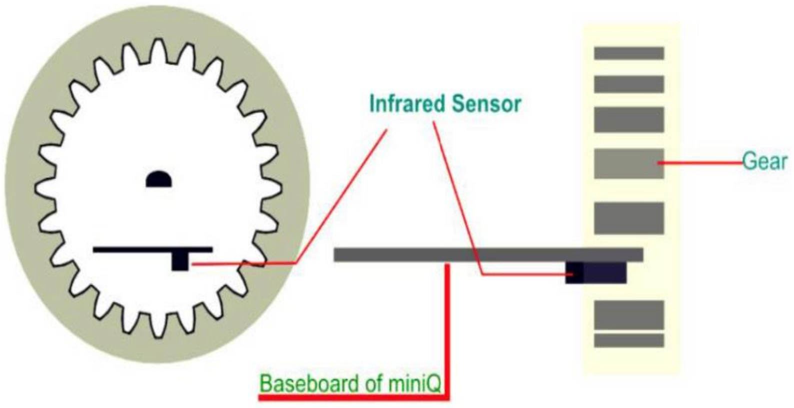


Fig 8. Encoder working (KIm, 2018)

2.2 Motion control design and implementation

Robot kinematic model and description

As mentioned above, the drive has two main wheels, each with its own motor attached to it. Important parameters are to be taken into consideration when implementing this project; these include both spatial and time quantities. It is also of crucial to note that in the determination of this mathematical model, physical constraints such as inconsistency in mechanical structure, friction, and inertia were neglected.

$$Distance\ Traveled = \frac{Total\ TIcks\ in\ the\ Encoder}{TIcks\ per\ rotation} * 2\pi * R, \qquad where\ R\ is\ the\ wheel\ radius$$

Kinematic equations:

Using v_r and v_l to represent the right and left wheels action vectors respectively, and, using V_ω and V_φ to symbolize the action variables for translation and rotation respectively, it can be shown that:

$$V_\omega = \frac{v_r + v_l}{2} \quad \text{,and}$$

$$V_\varphi = v_r - v_l,$$

giving rise to the following kinematics equations:

$\dot{d}$ = rV_ω cosθ, where $\dot{d}$ is the translational velocity of the drive, and

$\dot{\theta} = \frac{r}{axleLength} * V_\varphi$, where $\dot{\theta}$ represents the angular velocity of the drive

Distance Control Algorithm

The following distance control algorithm was implemented in order to achieve a specific driving distance for the robot. Every time the encoder is sampled, the difference between the current number of ticks and the previous is calculated.

$\Delta tick$ = current tick count – old tick count

The distance traveled by each wheel was calculated by the following formula:

Distance = $\frac{2\pi * wheel\ Radius * \Delta tick}{no.\ of\ ticksPerRotation}$

The average distance is taken to represent the distance travelled by the robot chassis, A PID controller is then tuned in such a way that it outputs velocity signal to ensure a stable velocity and deceleration; this overcomes the effects that may be caused by the drive's inertia, ensuring that the drive does not overshoot the distance

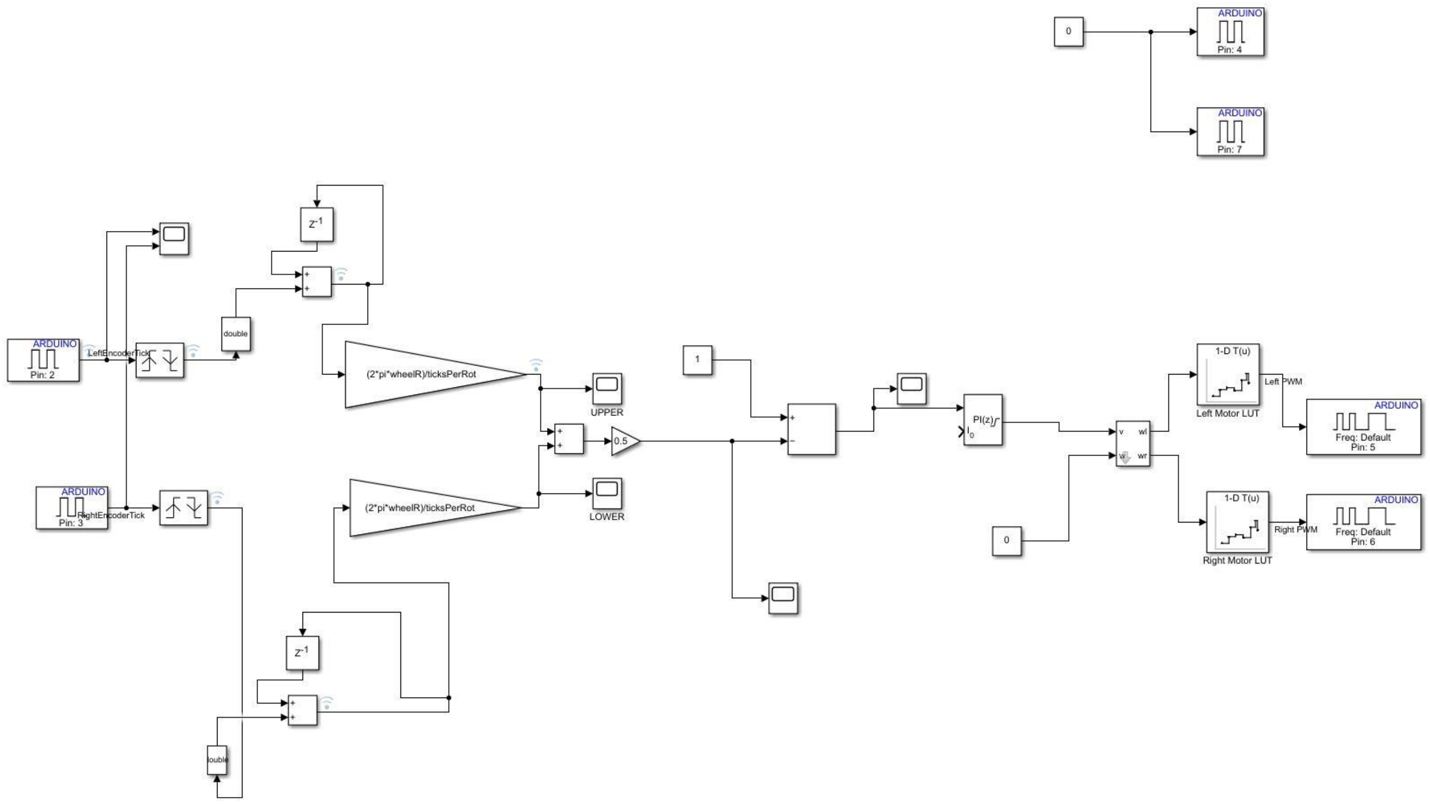


Fig 9: Distance algorithm block diagram.

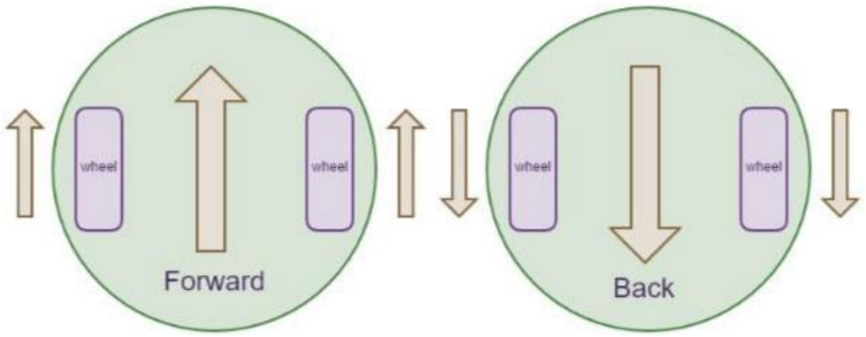


Fig 10. Directions of the wheels for translational motion

Angle Control Algorithm

The requirements here are to ensure that the differential drive can rotate at angles required. The algorithm used is essentially configuring either of the wheels to a have a negative velocity while the other travels the required distance and sweeps the angle specified. The configuration largely the same as the distance control algorithm. However, the constant block added to the 'subtract' block is configured to change the angle swept by the drive. By altering the values in this constant block, the angle swept will change.

The axle length, being the diameter, can be used to sweep an arc length by calculating a fraction of the circumference of the circle found by using the pi*diameter formula.

The formula used in the constant block is (axleLength*pi)/K, where K is may be 2, 1, or 0.75 to represent the angles 90°,180°, and 270° respectively.

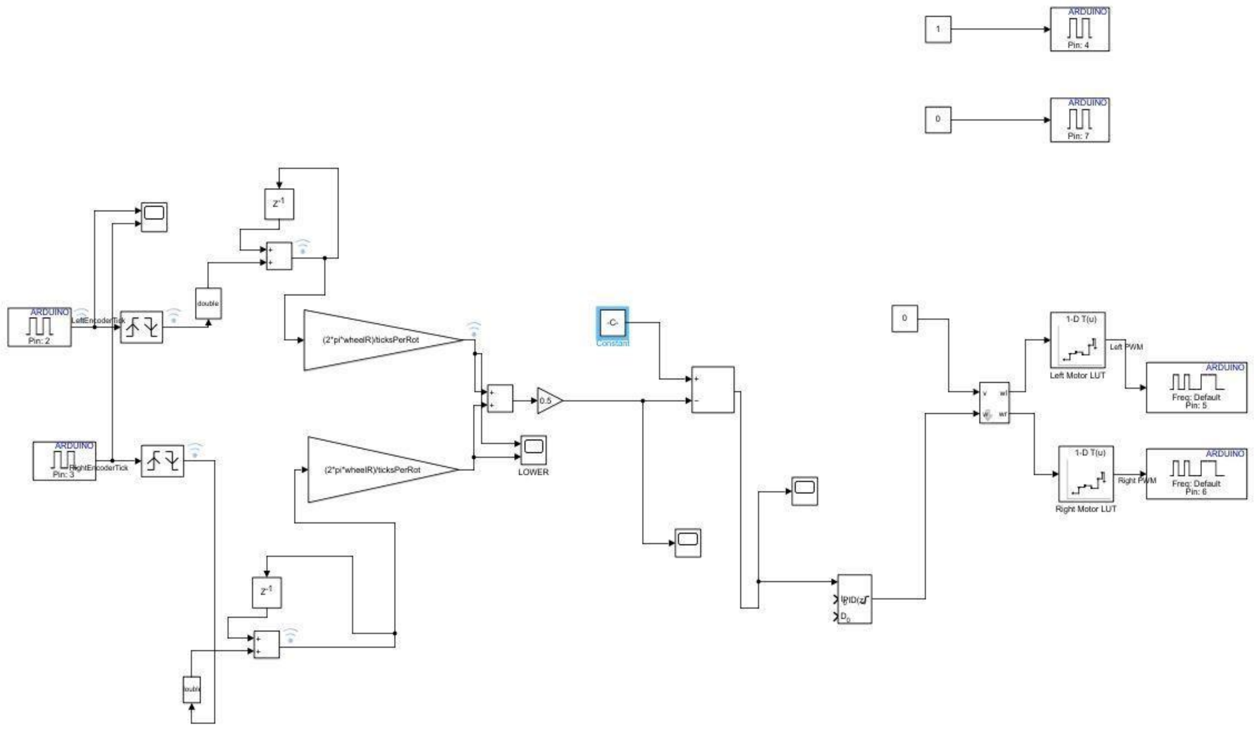


Fig 11: Angle algorithm block diagram.

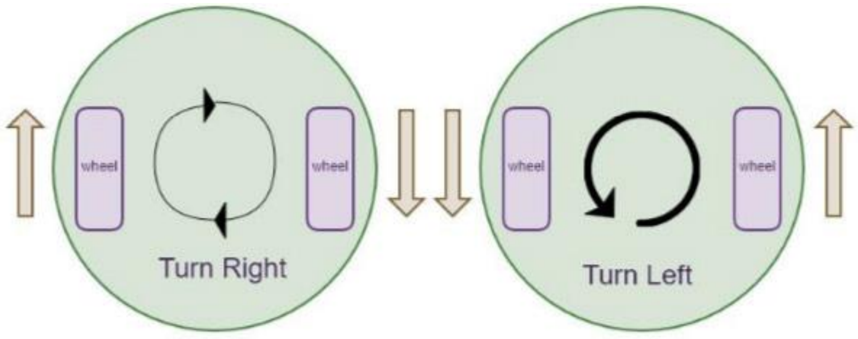


Fig 12. Directions of the wheels for rotational motion

2.3 Line sensing design and implementation

2.3.1 Line-Following Algorithm

The maze consists of straight lines with sharp corners. This allowed for a simple line-tracking algorithm that does not need an on-off algorithm, which is typically used for curved lines. As a result, while only sensors 2 and 3 are high, the robot will move linearly at a constant speed until this sensor feedback changes. This algorithm is much more simplified because the simulation assumes ideal conditions e.g, no axle anomalies. The above-mentioned selection algorithm is seen in the default of the Stateflow chart attached in the appendix; Fig 20. A PID module is used to ensure gradual deceleration and to minimize overshoot. An enhancement that is made for hardware is to mitigate the effects of friction, misalignments and slip is for the robot to adapt the on-off algorithm where it continually slightly turn to the left and right and using the feedback from the sensors to stay on the line.

2.3.2 Line Configuration Algorithm

Once the leading sensors 2 and 3 go low, the robot stops and goes into state detection state, if sensor 4 is high while 1 is low, then it is a right turn, on the other hand, the opposite would occur it the line configuration was that of a left turn. The robot will also stop and assess the state if both sensors 1 and 4 go high junction. If sensors 2 and 3 go low and the robot moves forward slightly, and the state remains, then a dead-end is registered. At the opposite end of the spectrum is a case where sensors 2 and 3 are both high, and when the robot moves forward slightly as part of state detection sensors 1 and 4 go high, in this case, this is registered as an end of the maze. A cross is detected if all sensors are high, followed by 1 and 4 going low when it moves forward slightly, while in a T-junction 2 and 3 go low while 1 and 4 go high. Finally, the differentiating factor between right/left Turn and right/left T is that when the robot advances slightly, sensors 2 and 3 go low while 1 or 4 goes high respectively. In the case of T 2 and 3 don't go low.

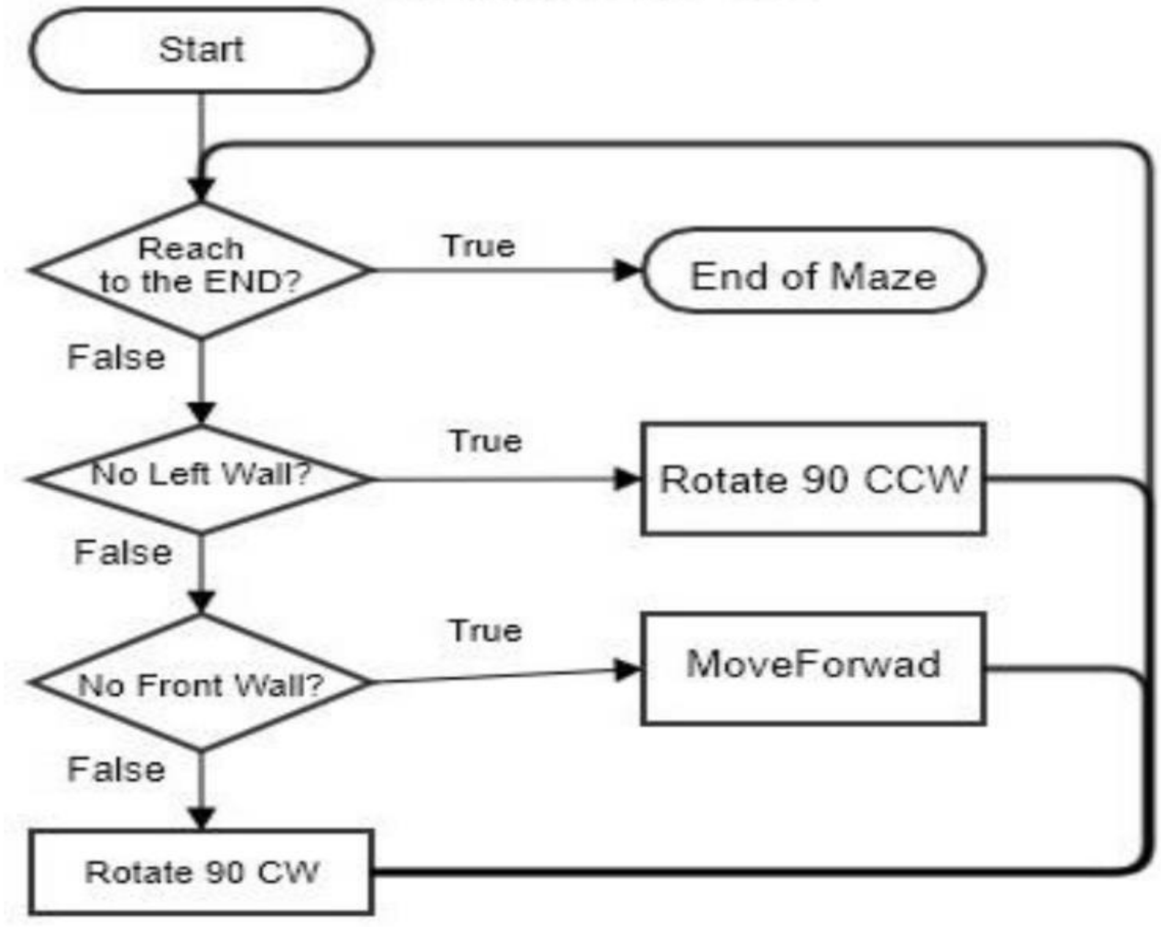


Fig 11. Left wall follower algorithm

Simulation Submodules;

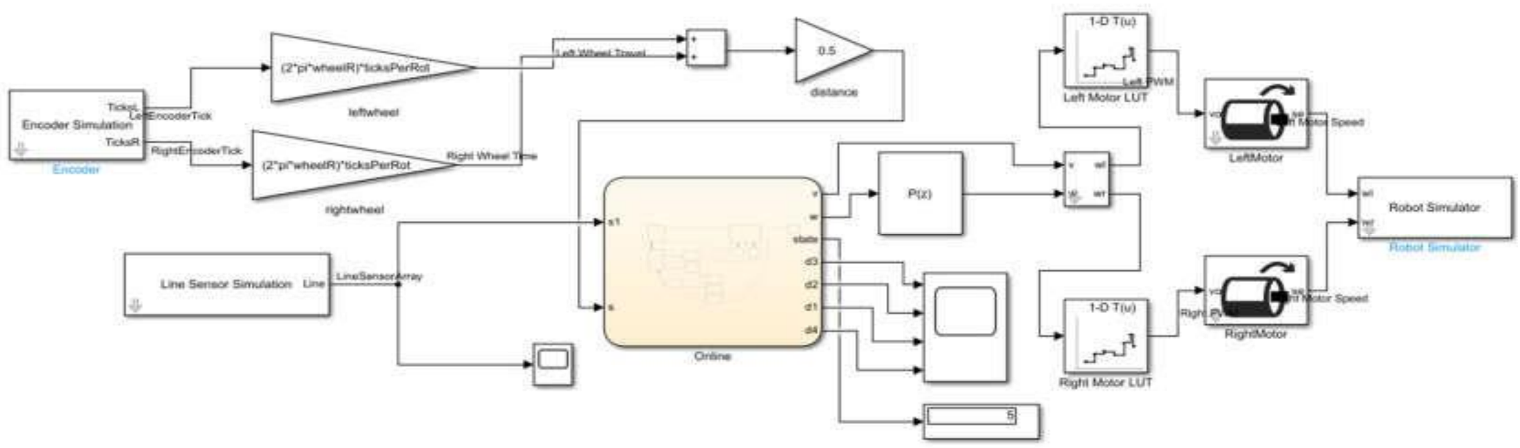


Fig 12: Simulation Modules

The model consists of an encoder that resembles the wheel encoders of the Arduino, encoder works in such a way that it provides electrical pulses into the drive model, the pulses can then be counted to find out the extent of the shaft rotation and the direction. Knowing information such as wheel radius and encoder resolution, it is then possible to determine the distance and speed of the robot. Its output is a digital 5V-0V square wave. The gain after the encoder multiplies the output voltage so that its numerical value equals that of the distance travelled. The online Stateflow is a module responsible for the “decision making” of the robot, the logic flow under the mask is attached in the appendix, Fig 20; it shows the algorithm discussed in the line following and configuration sections above. The outputs of this include the state value which is connected to the displays, its values range from 0(from the top left corner)-8(bottom right) seen in the key below.

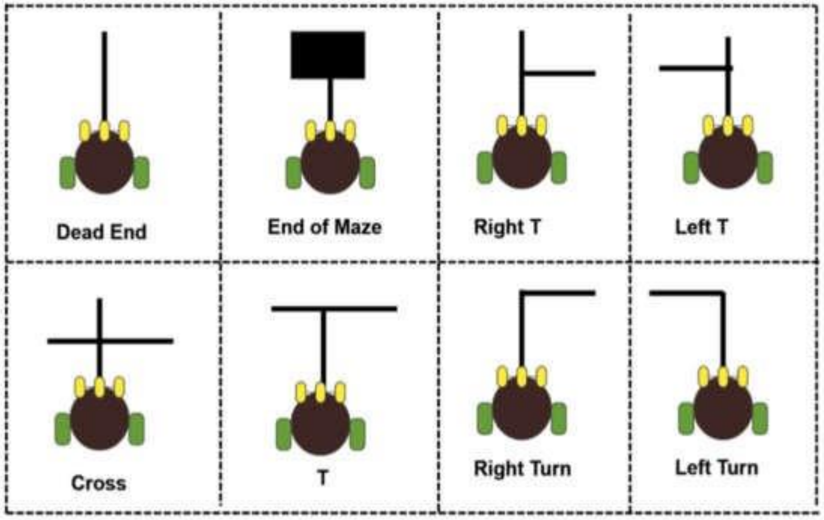


Figure 13: Maze Junctions

Additionally, other outputs are the average linear velocity v and the angular velocity w which are fed into wlwr controller, a module that interfaces with and controls the motors. The linear velocity output is connected to the wlwr through a PID to get an optimal response that accounts for the inertia, friction and other physical parameters that affect the response of the robot, this minimizes overshoots as well.

Hardware:

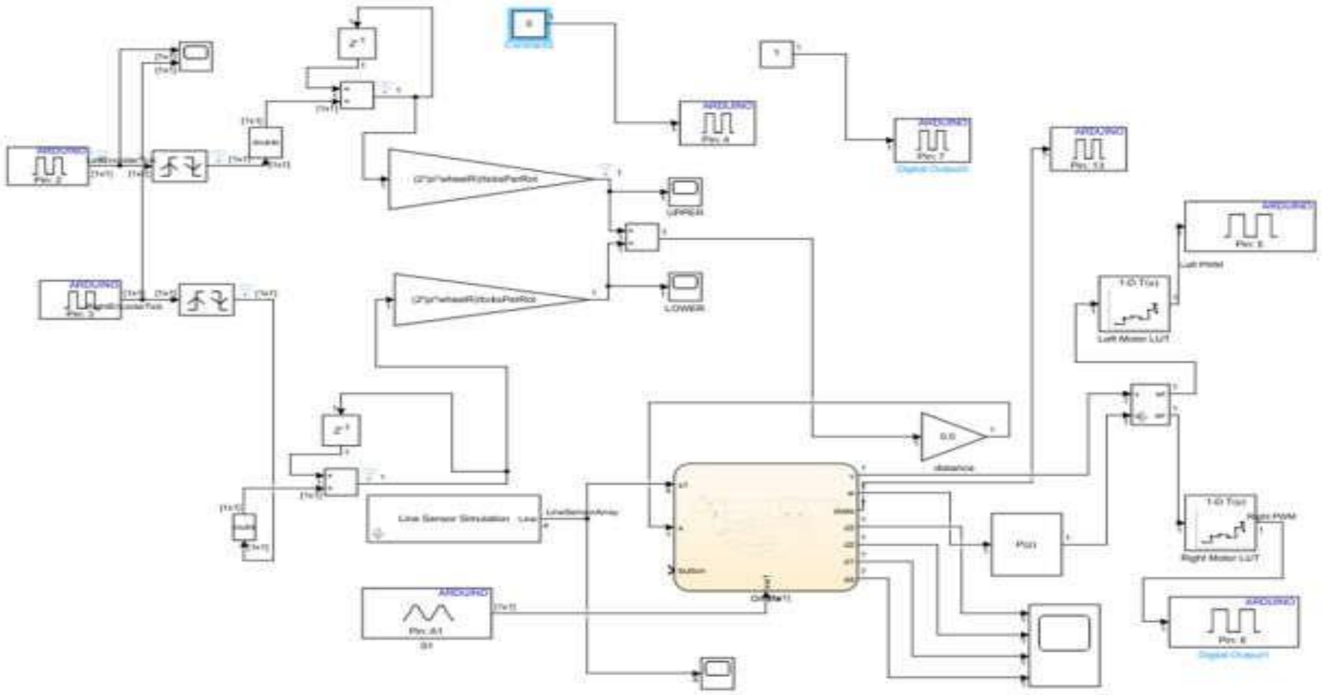


Figure 14: Hardware modules

The configuration for the hardware is largely similar except for the following modifications. The encoder is swapped for the Arduino optical encoder pins 3 and three, furthermore, the output from the Stateflow is connected to the input pins 5 and 6 which actuate the motors. The button is connected to the Stateflow so that it initiates the whole process, this is seen clearly beneath the mask in the appendix. Finally, the state output from the Stateflow chart, which indicates the state, is connected to the LED in order to enable the functionality that the LED may serve as an indicator for the state.

2.4 Maze solver and shortest pathfinder algorithm design and implementation

The maze learning algorithm involves the robot tracking the maze using the left-hand rule, this means the robot is continually checking for veering and auto correcting it by comparison to a reference. Veering senses if one of the middle sensors (2&4) go high which implies they are on the environment, and depending on one that goes high, the robot to the opposite side to correct the error. A PID is incorporated to reduce overshoot and settling time of the correction. The left hand implies that during learning, the robot turns left at each junction and goes through the maze exhaustively.

The process of solving the maze for the shortest path was carried out by keeping track of the turns made, thus reducing the list of the turns made as the latest turn is added to the list of the known substitutions. The used substitutions are three combinations either (R)ight, (L)eft, or (S)traight would be on either end of the (B)ack as shown below:

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

An illustration of the above logic is shown below:

If a robot were to move towards a left T-junction, then turn to the left to a path that leads to a dead end, it would turn back and then turn to the left (Left-hand algorithm) and proceed straight. This movement, as seen in figure 14 below, simplifies to the robot just moving straight past that junction. The simplification below would also result from

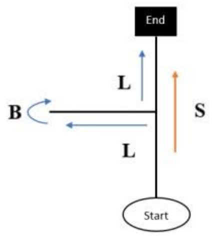


Figure 15: LBL = S

One other scenario is where the robot moves towards a left turn which leads to a dead end, then turns back out of the turn past its starting point. This simplifies to the robot turning 180 and moving straight (i.e going back). This is illustrated below. The case also holds for RBL.

Group 36

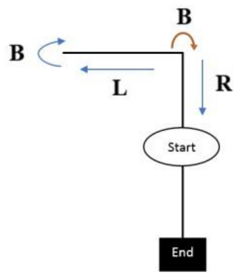


Figure 16: LBR =B

The robot could also head towards a T-junction and then turn left towards a dead end, turn back and head straight to the end simplifies to it turning to the right and proceeding straight.

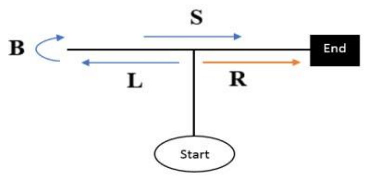


Figure 17: LBS =R

The penultimate case is one where the robot moves past a left turn to a dead end, then turns back, and then left towards the turn. This simplifies to turning right.

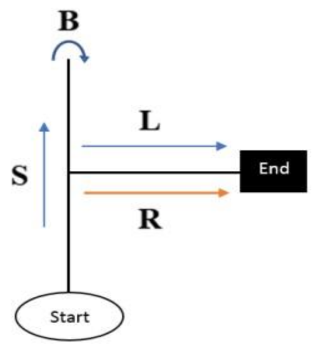


Figure 18: SBL = R

The final case is of the robot moving straight towards a dead end, turning back and proceeding straight. This can be simplified to just moving back (turning 180 and moving back). The above rules can now be used on the resultant path for simplification.

The implementation of the above combinations can be seen in the appendix

High level description of the above logic:

- Move forward
- Make a (L)eft turn at a found intersection
- Add "L" to the list
- Check whether the last three items on the list can be shortened: NO in this case
- (B)ack if a dead end is reached
- Add "B" to the list
- Check whether the last three items on the list can be shortened: NO in this case
- Make a (L)eft turn if the initial intersection at which a left turn was made is found again
- Add "L" to the list
- Check whether the last three items on the list can be shortened: YES in this case
- Replace the "LBL" with (S)traight
- Keep moving until the end point is reached.

Implementing the above idea, the path has now been shortened; the robot will now move straight through intersections and avoid dead ends.

2.5 RESULTS

Motion control

The robot managed to move a prescribed distance(e.g 1 metre) within a 10% error margin. The hardware test run was affected by environmental factors such as friction, slipping, inertia and more. The angle control was also successfully achieved and managed to lie within 10% of the error margin. However, the constants used to change the angles had to be modified to fit the prescribed angles because of the instabilities caused by the PID controller.

Line sensing

The line sensing algorithm was successfully implemented. However there were certain eccentricities caused by the inconsistency in the thickness of the maze lines. There were points at which delays had to be introduced in order to counter the effects of the sensors mistaking, for example, a cross junction for the end of the maze. Generally, the robot was able to follow the lines, make left turns where necessary, and stop at the end of the maze. The biggest challenge was tuning the PID controller to get the robot to move straight

Maze solver and shortest pathfinder

The robot successfully learned a big part of the maze. However, it was realized that when different iterations were taken, there were anomalies detected. The robot would, for instance, detect a false dead end, or the end of the maze. 10 iterations were made and out the 10 the robot managed to find the shortest path 8 times and failed 2 times. The data is presented in fig 18 below.

No. of trial	Time taken to solve maze
1	52 seconds
2	88 seconds
3	41 seconds
4	79 seconds
5	43 seconds
6	55 seconds
7	40 seconds
8	41 seconds

3.CONCLUSION

Esssentially, the robot was able to find the shortest path in most of the iterations made. The simulation was able to work almost perfectly because the robot was not exposed to environmental variables such as friction. The biggest challenge pertaining to the simulation was the ability to tune the PID controller in such a way that the robot is fast enough to complete the maze quickly but also does not have overshoots at junctions and dead-ends. The hardware was also successful with a few exceptions such as detecting false junctions during some test runs. This may be attributed to the delays that were caused by (1) the friction between the wheels and the maze environment, or (2) the reflections from the environment that cause the robot sensors to detect false signals.

3.1 Future recommendations:

- Minimize the errors that affect line following such as an unstable axle that causes the wheels to not move straight.
- Demoing the robot in a room with even lighting to avoid false reflections
- Minimizing the friction between the wheels and the maze environment by using a material with a lower coefficient of friction.

References

Arduino, 2018. *Arduino Leonardo.* [Online]
Available at: https://www.arduino.cc/en/Main/Arduino_BoardLeonardo
[Accessed 21 October 2021].

Elshamarka, I., 2012. Design and Implementation of a Robot for Maze-Solving using Flood-Fill Algorithm. *International Journal of Computer Applications(0975-8887),* 56(5), pp. 8-9.

KIm, H., 2018. *Smart Maze robot,* Evansville, Indiana: s.n.

APPENDIX

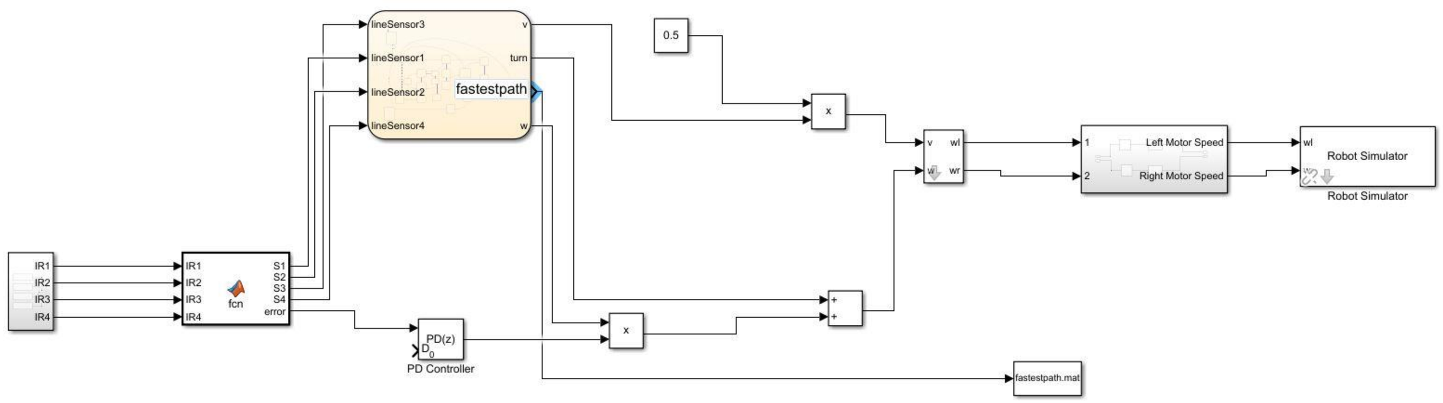


Fig 19. Simulation model

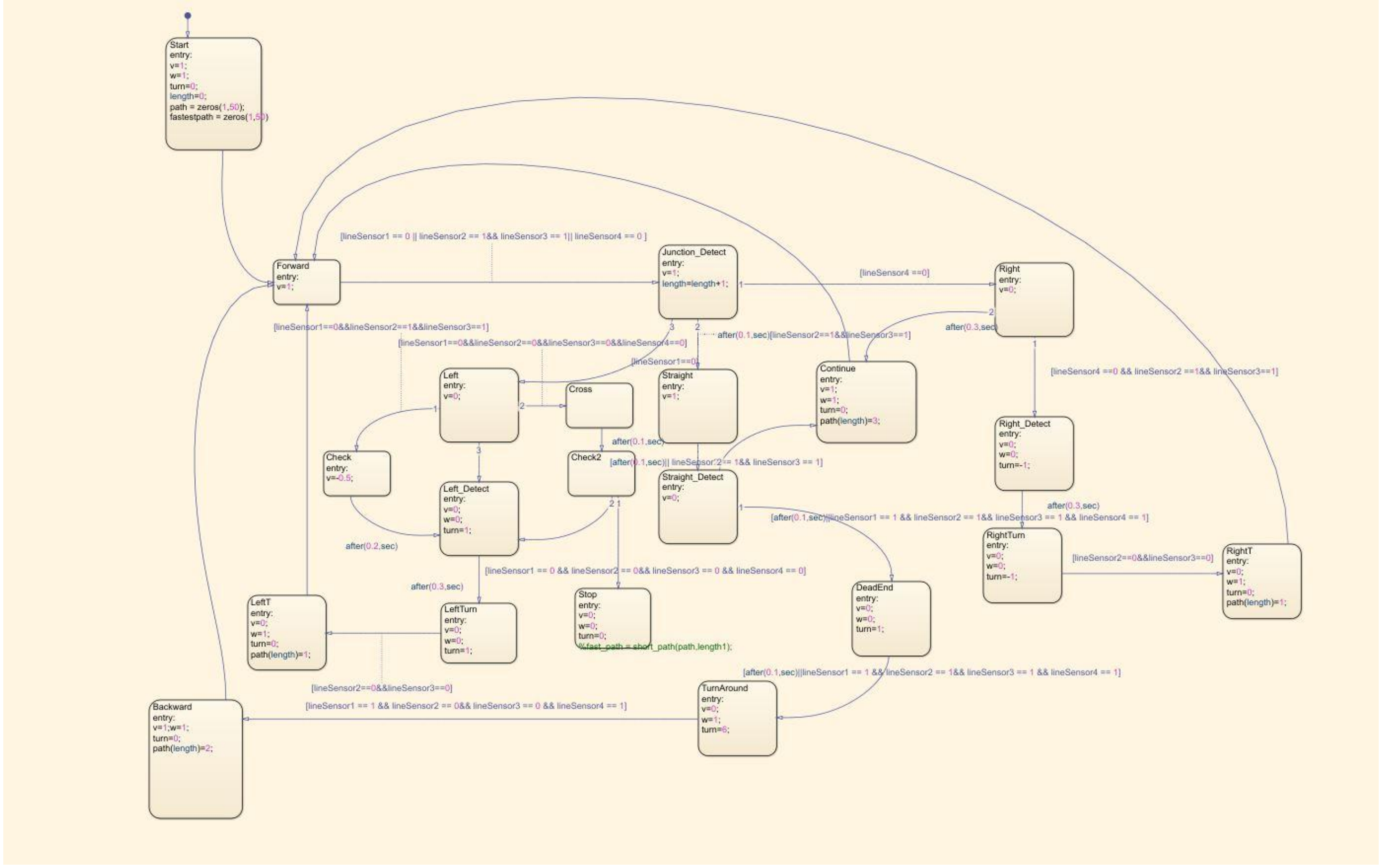

Start entry: v=1; w=1; turn=0; length=0; path = zeros(1,50); fastestpath = zeros(1,50)
Forward entry: v=1;
Junction_Detect entry: v=1; length=length+1;
Right entry: v=0;
[lineSensor1 == 0 || lineSensor2 == 1&& lineSensor3 == 1]|| lineSensor4 == 0]
[lineSensor4 ==0]
after(0.3,sec)
[lineSensor1==0&&lineSensor2==1&&lineSensor3==1]
[lineSensor1==0&&lineSensor2==0&&lineSensor3==0&&lineSensor4==0]
after(0.1,sec)[lineSensor2==1&&lineSensor3==1]
[lineSensor1==0]
Left entry: v=0;
Cross
Straight entry: v=1;
Continue entry: v=1; w=1; turn=0; path(length)=3;
[lineSensor4 ==0 && lineSensor2 ==1&& lineSensor3==1]
Check entry: v=-0.5;
Check2
after(0.1,sec)
[after(0.1,sec)|| lineSensor2 == 1&& lineSensor3 == 1]
Left_Detect entry: v=0; w=0; turn=1;
Straight_Detect entry: v=0;
Right_Detect entry: v=0; w=0; turn=-1;
after(0.2,sec)
after(0.3,sec)
[after(0.1,sec)||lineSensor1 == 1 && lineSensor2 == 1&& lineSensor3 == 1 && lineSensor4 == 1]
RightTurn entry: v=0; w=0; turn=-1;
[lineSensor2==0&&lineSensor3==0]
RightT entry: v=0; w=1; turn=0; path(length)=1;
[lineSensor1 == 0 && lineSensor2 == 0&& lineSensor3 == 0 && lineSensor4 == 0]
LeftT entry: v=0; w=1; turn=0; path(length)=1;
LeftTurn entry: v=0; w=0; turn=1;
Stop entry: v=0; w=0; turn=0; %fast_path = short_path(path,length1);
DeadEnd entry: v=0; w=0; turn=1;
[after(0.1,sec)||lineSensor1 == 1 && lineSensor2 == 1&& lineSensor3 == 1 && lineSensor4 == 1]
[lineSensor2==0&&lineSensor3==0]
[lineSensor1 == 1 && lineSensor2 == 0&& lineSensor3 == 0 && lineSensor4 == 1]
TurnAround entry: v=0; w=1; turn=6;
Backward entry: v=1;w=1; turn=0; path(length)=2;

Fig 20. Simulation Stateflow chart

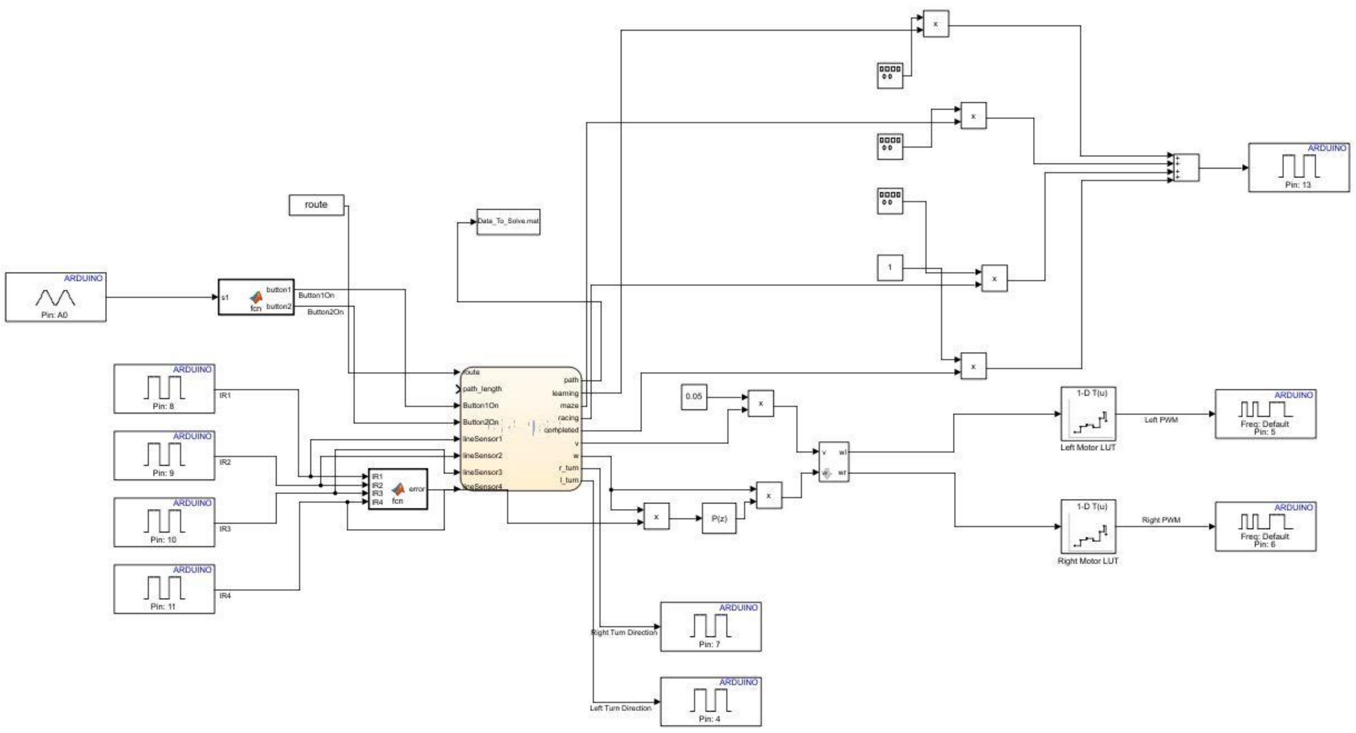


Fig 21. Hardware model

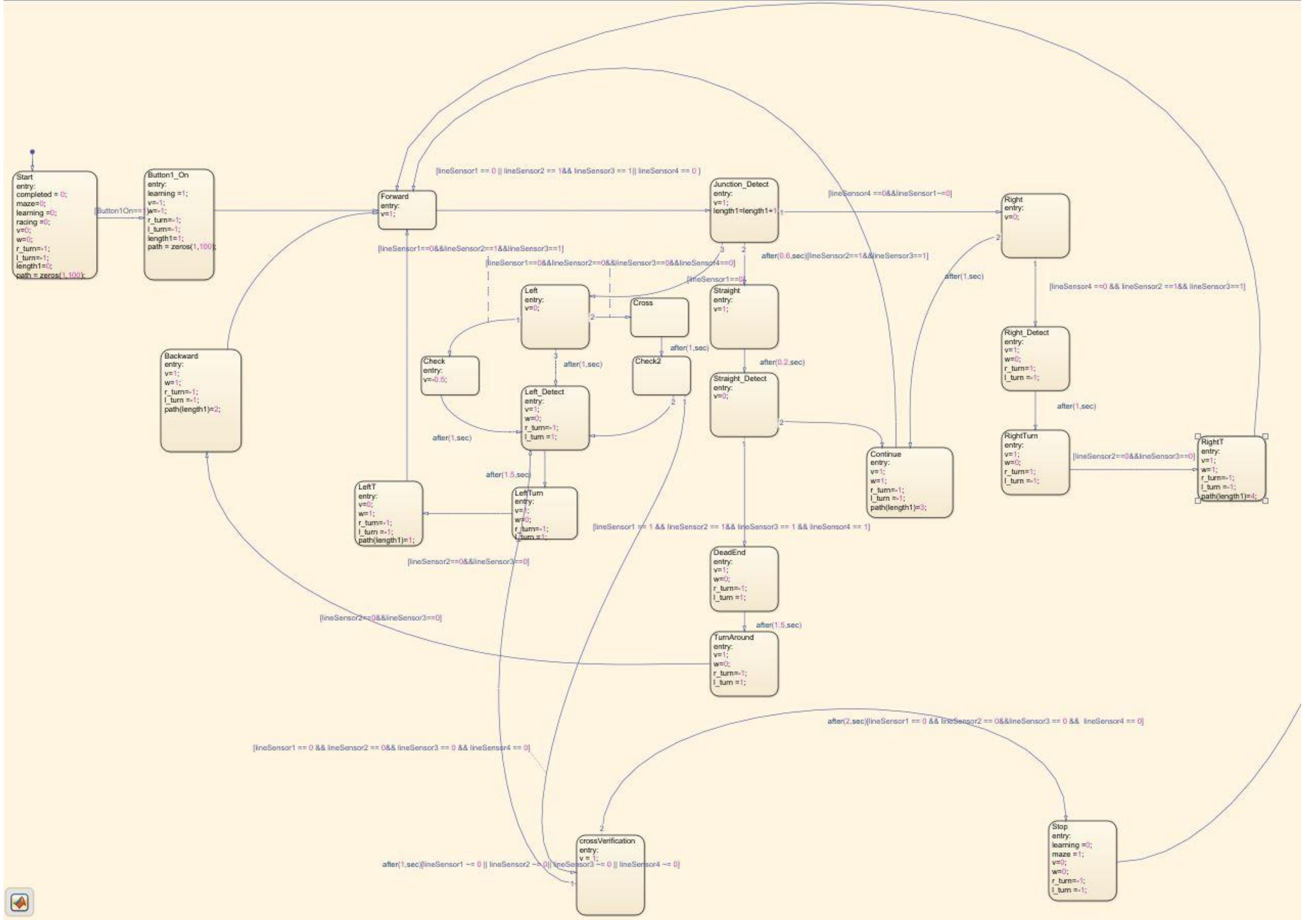


Fig 22. Hardware maze learning Stateflow chart

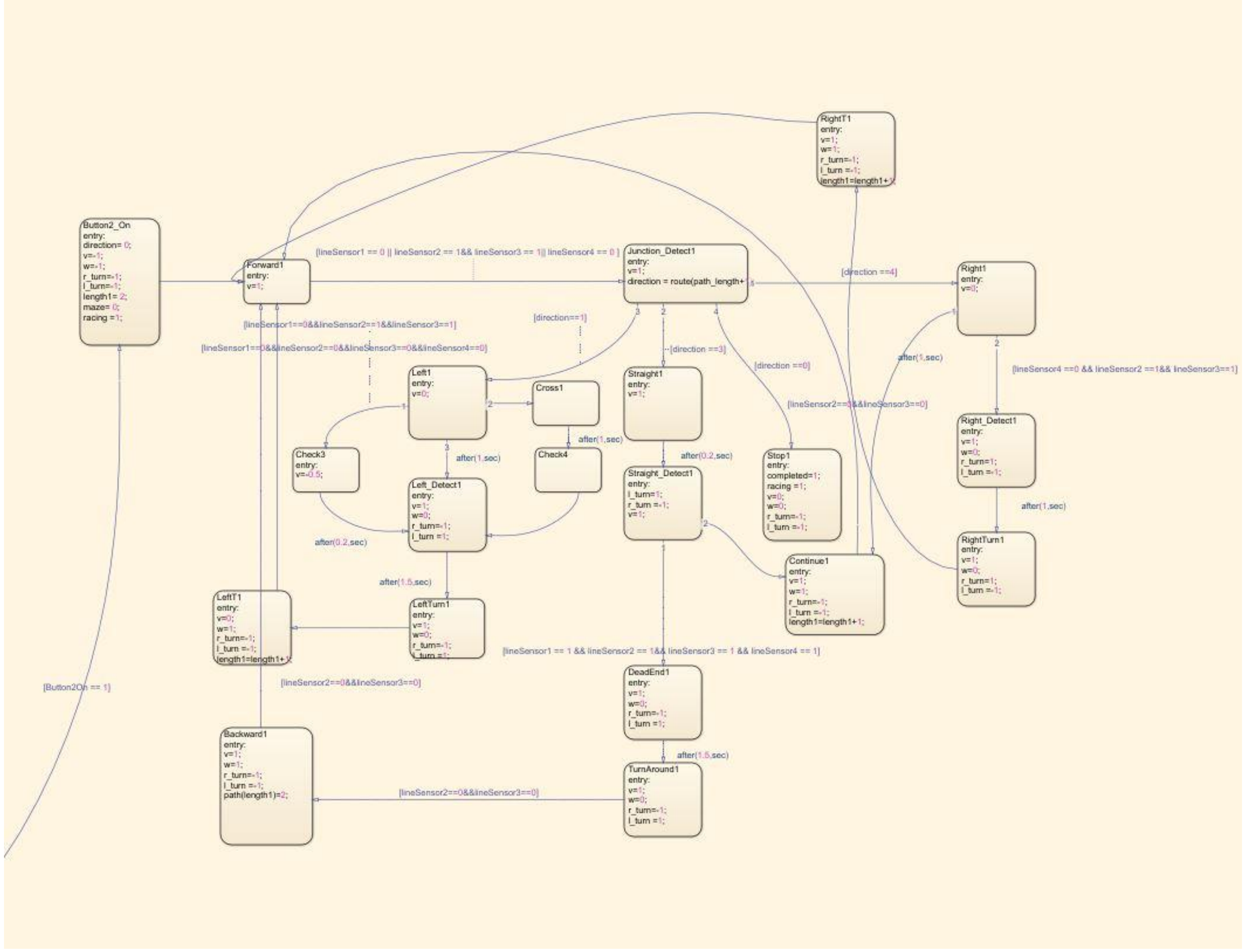

Button2_On entry: direction= 0; v=-1; w=-1; r_turn=-1; l_turn=-1; length1= 2; maze= 0; racing =1;
Forward1 entry: v=1;
[lineSensor1 == 0 || lineSensor2 == 1&& lineSensor3 == 1|| lineSensor4 == 0]
Junction_Detect1 entry: v=1; direction = route(path_length+1)
RightT1 entry: v=1; w=1; r_turn=-1; l_turn =-1; length1=length1+1;
[direction ==4]
Right1 entry: v=0;
[lineSensor1==0&&lineSensor2==1&&lineSensor3==1]
[lineSensor1==0&&lineSensor2==0&&lineSensor3==0&&lineSensor4==0]
[direction==1]
[direction ==3]
[direction ==0]
after(1,sec)
[lineSensor4 ==0 && lineSensor2 ==1&& lineSensor3==1]
Left1 entry: v=0;
Cross1
Straight1 entry: v=1;
[lineSensor2==0&&lineSensor3==0]
Right_Detect1 entry: v=1; w=0; r_turn=1; l_turn =-1;
Check3 entry: v=-0.5;
after(1,sec)
Check4
after(1,sec)
after(0.2,sec)
Stop1 entry: completed=1; racing =1; v=0; w=0; r_turn=-1; l_turn =-1;
Left_Detect1 entry: v=1; w=0; r_turn=-1; l_turn =1;
Straight_Detect1 entry: l_turn=1; r_turn =-1; v=1;
after(1,sec)
after(0.2,sec)
Continue1 entry: v=1; w=1; r_turn=-1; l_turn =-1; length1=length1+1;
RightTurn1 entry: v=1; w=0; r_turn=1; l_turn =-1;
after(1.5,sec)
LeftT1 entry: v=0; w=1; r_turn=-1; l_turn =-1; length1=length1+1;
LeftTurn1 entry: v=1; w=0; r_turn=-1; l_turn =1;
[lineSensor1 == 1 && lineSensor2 == 1&& lineSensor3 == 1 && lineSensor4 == 1]
[lineSensor2==0&&lineSensor3==0]
[Button2On == 1]
DeadEnd1 entry: v=1; w=0; r_turn=-1; l_turn =1;
after(1.5,sec)
Backward1 entry: v=1; w=1; r_turn=-1; l_turn =-1; path(length1)=2;
[lineSensor2==0&&lineSensor3==0]
TurnAround1 entry: v=1; w=0; r_turn=-1; l_turn =1;

Fig 23. Hardware maze solving(racing) Stateflow chart