iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 9 **Issue:** IV **Month of publication:** April 2021

DOI: https://doi.org/10.22214/ijraset.2021.33883

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

A Stateflow based Approach for Simulation of Line Following Maze Solver Robot

Tejas Phutane[1], Sahil Pillikandlu[2], Niyati Vaidya[3], Avadhoot Khedekar[4]
[1]*Department of Electronics and Telecommunication Engineering,* [1, 2]*Department of Mechanical,* [4]*Department of Computer Science Engineering, MCT's Rajiv Gandhi Institute of Technology, Mumbai, Maharashtra, India*

Abstract: ***This paper presents the simulation of the classical PD control technique for implementation of Line Follower Robot followed by integration of the maze-solving algorithm with insights of differential drive mechanism performed on MATLAB software. The system incorporates parameters of real components such as motors and sensors to replicate the fundamental characteristics of the robot. To carry out experimentation at low cost as well as at low risk, it is possible to develop a virtual system along with the exact environment surrounding it. Simulation is the key to determine the underlying mechanisms that control the behaviour of the system without actually implementing it in the real world.***
Keywords: Motion and Path Planning, Sensor Based control, Differential Drive, Simulation Systems, Wheeled Robots.

I. INTRODUCTION

A Line Following Maze-solver Robot is an autonomous system that follows either black or white line with a highly contrasting background drawn on the plane surface. Simulation is carried out to avoid manufacturing cost of the proposed system and construction of the environment required for testing phase of the system and also to eliminate the risk involving damage to the actual resources. The entire process is performed on MATLAB platform as it offers the visualization and methodical analysis of the respective system under the various practical scenarios and thus provides modulation of the parameters affecting the performance of the same. The Line Following Maze-solver robot incorporated with reflectance sensors must be capable of following the path (white line in this case) and along with that, it must keep the track of the intersections or immediate variation in angle along its path. The motion of the robot is implemented using PD (Proportional-Derivative) control technique. This paper is divided into sections elaborating below three phases of simulation:

1) *Creating the Environment*: In order to receive accurate results from the simulation, it is necessary to create a real-world scenario virtually which includes the physical properties of both the system and its surrounding environment to test the behaviour of the system.
2) *Dry Run*: It refers to the traversal of the robot throughout the maze until the end point is encountered. This phase stores the path-tracking information in its volatile memory to fulfil the requirements for the next phase.
3) *Actual Run*: The final phase evaluates the path-tracking information in order to determine the possible shortest path.

The parameters are received from the simulation which are based on the valid sources of information, relevant selections of characteristics and assumptions within the simulation.

II. PRELIMINARIES

A. Creating the Environment

Prerequisite for performing any kind of simulation is to create a replica of the environment surrounding the system. For creating a line maze, a binary occupancy map is generated through the *Simulation Map Generator*, which is preinstalled in MATLAB. Initially, a 2D map is designed in any CAD software such as Solidworks, AutoCAD etc. It is necessary to consider desired dimensions viz. width of line, angle of intersection since these factors influence the performance of the system/simulation. The resulting map is then converted into an image with resolution 1000 pixels per meter.

The image is imported into the Simulation Map Generator and dimensions (in meters) of the maze must be specified manually. Map option is selected as *Line Following*. It provides an image threshold option to increase accuracy in binary occupancy map by minimizing the noise in the image. The generated map is exported with *'.mat'* extension and the same file is imported in MATLAB workspace for further simulation. After importing, MATLAB automatically creates a variable named *MapForSim*. This variable serves as the entire maze map.

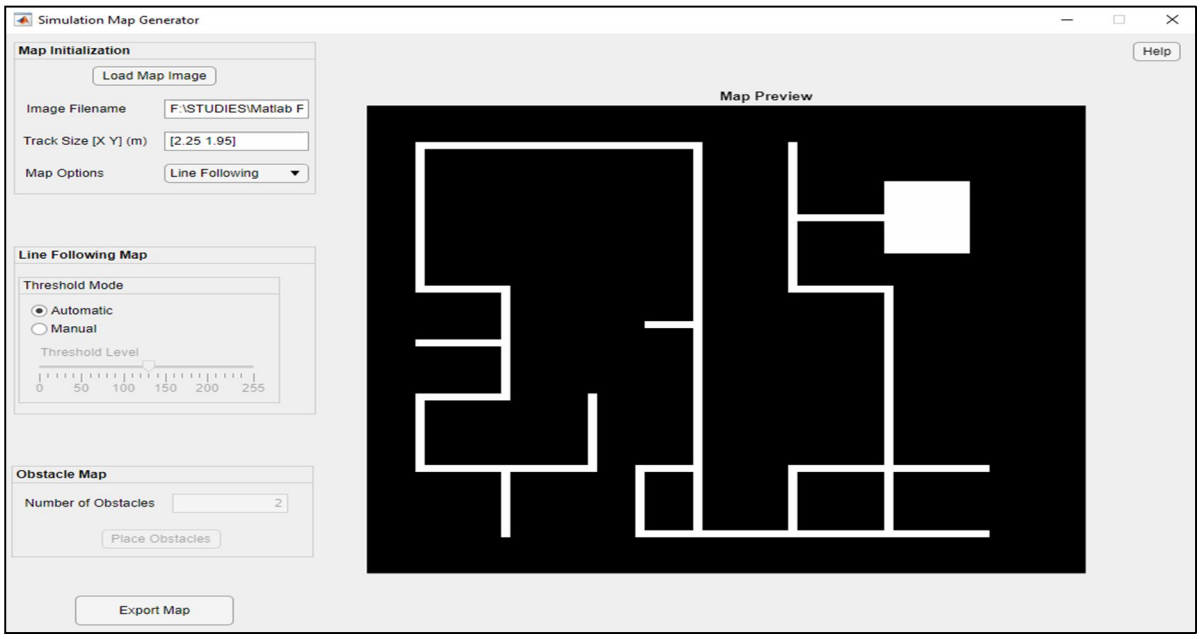


Fig. 1 Binary Occupancy Maze Map generation in Simulation Map Generator.

B. Configuration of Line Sensor Block

The proposed robot system is incorporated with a Reflectance Line Sensor Array to distinguish between Line and background Environment. To simulate the line following task, a Line Sensor Block that serves as a reflectance sensor which is available in Mobile Robotics Training Library within MATLAB is used. The block outputs viz. Environment values as well as Line values of the binary map are based on whether the sensor is on top of the Environment or the Line. It is necessary to customize the characteristics of the available Line Sensor Block according to the sensor, which will be used in real world implementation.

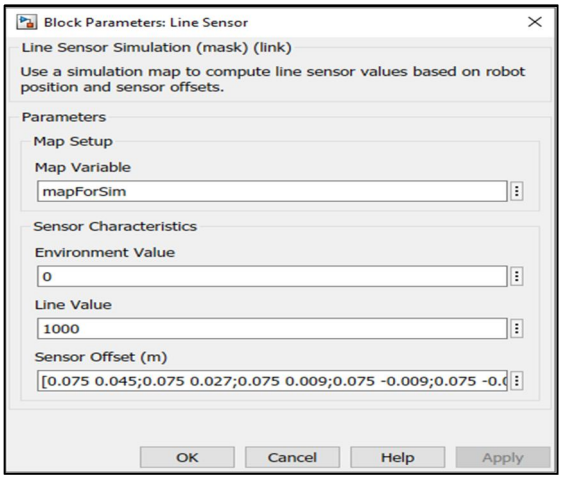


Fig. 2 Specification of sensor array characteristics to Replicate the behaviour in the virtual environment.

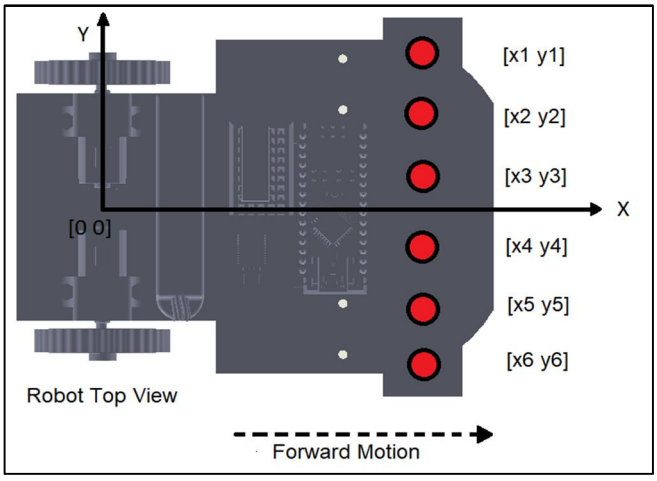


Fig. 3. Position of individual sensors in sensor array with respect to center of wheel rotation axis.

In Block Parameters window, the sensor characteristics that are needed to be specified are as follows:

1) *Map Variable*: The variable which is automatically created after importing the '.mat' file of the line maze. This variable reflects the maze characteristics through the map created in Simulation Map Generator.
2) *Environment Value*: The minimum value which we have to specify such that individual sensors give the same value when they occur over the background surface.
3) *Line Value*: The maximum value which we have to specify such that individual sensors give the same value when they occur over the line surface.

4) Sensor Offset in meters: Array of [x1 y1; x2 y2;….;xn yn] values that represents the offset position of the line sensors from the center of wheel axis. Each coordinate in the array corresponds to an individual line sensor in the array. Fig. 3 explains the coordinates of individual sensors with respect to center of the axis of the wheel. The red circles represent each individual sensor. Since, each sensor is collinear and parallel to the wheel axis, the x-coordinate is identical and y-coordinate differs by the corresponding offset.

Due to difference in contrast at various spots in environment as well as variation in ground clearance of the sensors, they may give intermediate values between 0 and 1000. Hence, we apply average threshold value as a filter to distinguish the readings.

C. Weighted Average Error

Line sensor provides six readings corresponding to each individual sensor. These readings have to be evaluated into one error value. Hence, each sensor is assigned with a weight according to its position in sensor array to get the weighted sum.

We have assigned the left position to the sensor located at coordinates [x1, y1] as shown in Fig. 3. This weighted sum is averaged by the summation of readings of the active sensors. In the case where line sensor is entirely over environment, then 2.5 value is considered as default weighted average irrespective of the sensor readings to avoid *divide-by-zero exception*. This causes the range of the weighted average values to reside between 0 and 5.

$$\text{Weighted Sum} = \frac{\sum_{i=0}^{5} Sensor[i]*i}{\sum_{i=0}^{5} Sensor[i]} \qquad (1)$$

$$\text{Error} = 2.5 - \text{Weighted Sum} \qquad (2)$$

The final error value is evaluated by subtracting the weighted average from 2.5 as the error must be zero when only centric sensors are activated i.e. above the black line. The negative error indicates that the motion of the robot tends to greater on the right direction and vice versa.

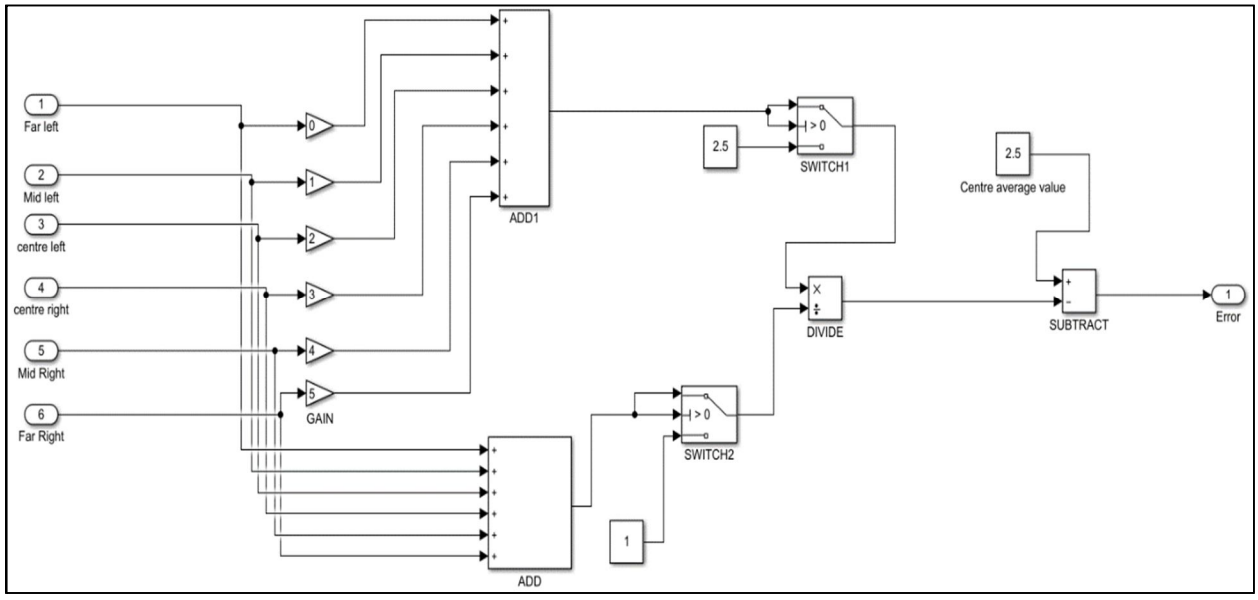


Fig. 4 Implementation of weighted average error in Simulink MATLAB.

The above figure shows the Simulink implementation of aforementioned formulae. The triangular GAIN blocks represent the weights assigned to individual sensors followed by the Summation block. This is equivalent to numerator of (1). Sensor readings are bypassed and then added which is equivalent to the denominator of (2). Switch blocks are used to avoid exceptions and then error is calculated from the SUBTRACT block. This error is given to PD control block. Here, each mentioned primary block is inbuilt in Simulink

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

III.PD CONTROL SYSTEM

A. Configuration of PID block

In the line following application, a closed loop controller is used to achieve steady motion. A closed loop controller is a system which constantly provides the feedback about its response to the readings provided by the sensors. This feedback helps the system to keep the error value as close as possible to zero thus improving the stability of the robot. PD controller is the most commonly used controller for such purposes. The reason behind choosing a PD controller is that the proportional gain causes the robot system to respond quickly to minimize the error while derivative gain prevents overshooting of the robot system resulting from the quick response due to proportional gain, thus maintaining the transient stability throughout its traversal.
Given the weighted average error e(t) at instance t, the mathematical formula for calculating PD output is given by:

$$PD\ output = P.e(t) + D.\frac{d}{dt}e(t) \quad (3)$$

In MATLAB, there exists a PID block inside Simulink. The only input required to the block is the weighted error value evaluated from the method mentioned in the one of the previous sections and its output is angular velocity (**ω**) which is then supplied to the *Robot Simulator* block. As with every other block, the PID block also needs to be configured. Here, we are using PD control in a discrete time domain since the respective *Robot Simulator* block is compatible with discrete time domain

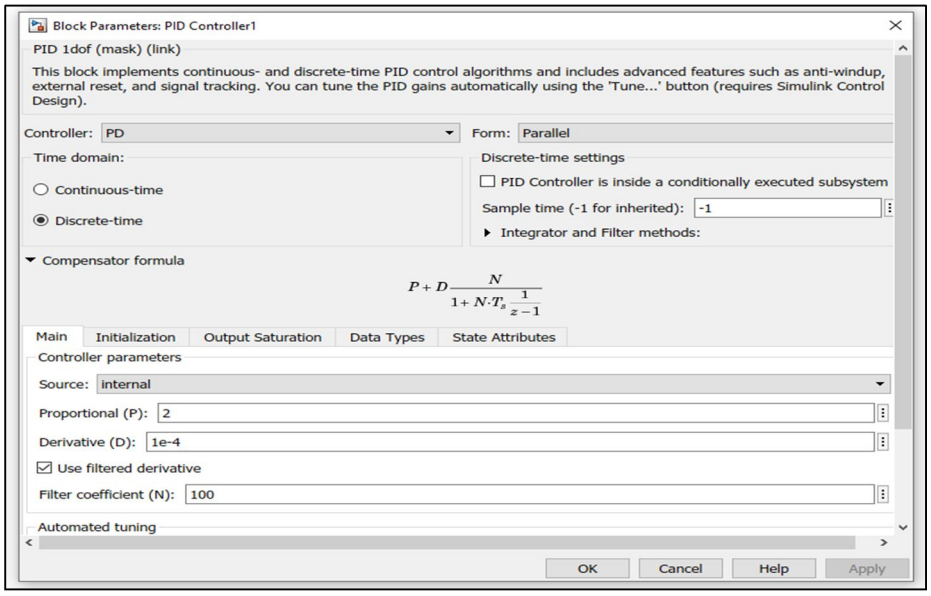


Fig. 5 Configuration of PID block as it can be used as PD control block by specifying required parameters.

The few parameters, which are necessary to be specified as mentioned in Fig. 5, are as follows:

1) *Controller Type:* Among the various types of control systems, PD controller is implemented for this application. We cannot use PID controller since it slows down the system response
2) *Proportional Gain (P):* This term, after multiplying with error adjusts the proportional response. It is primary factor of PD control system. Generally, P term is set before the other gain constants.
3) *Derivative Gain (D):* This term slows down the controller output resulting in the minimization of overshooting of the robot. The D term is evaluated according to the resulting response due to P term.

The MATLAB also provides the option for automatic tuning where it takes input as required response curve and then adjusts the gain constants such that it fits the input curve. In some cases, it may not provide the required output, hence we need to further tune it.

B. Calibration of Gain Constants

Response time of the robot depends on the architecture of the same. Hence, to achieve transient motion, the gain constants need to be tuned by observing the robot's behavior for various parameters. To reach utmost stability, multiple values need to be tried out. Initially, the P gain constant is set followed by the D gain constant.

P-gain: The proportional gain must be set such that the robot starts to recognize the existence of line. If P-gain has small value, then robot will not follow the line. However, high value of P-gain will result in undamped oscillations with high amplitude. In practice, the P-gain value should be initialized with 0 and updated with small increments till the robot starts to follow the line even though it shows a minute oscillatory motion.

D-gain: After adjusting the P-gain, the resulting oscillations can be dampened by the appropriate value of D-gain. Similar to P-gain, this value should be initialized with 0 and updated with even small increments till the oscillations completely disappear. The gain parameters must be positive real values and can be obtained by trial-and-error method. There is no need to change the remaining default parameters in the block parameter window as in Fig. 5.

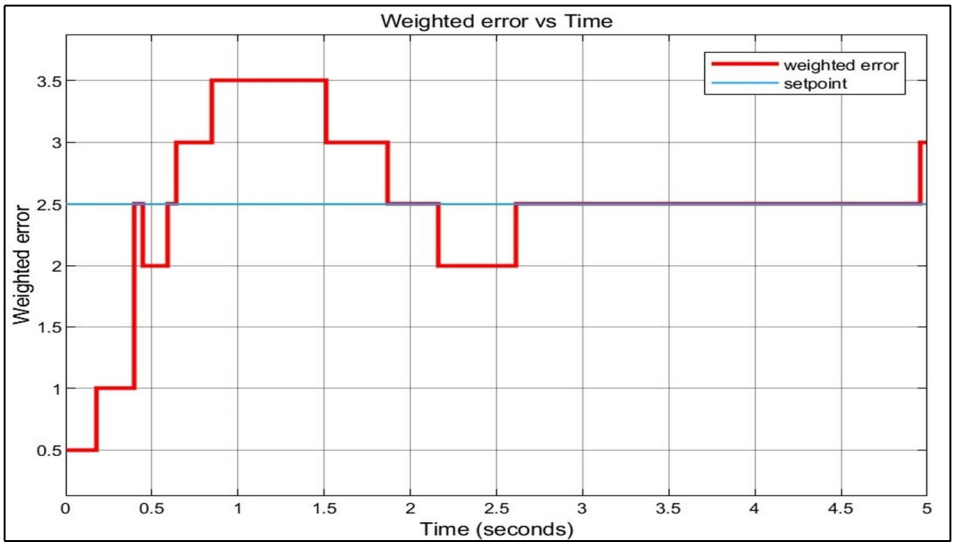


Fig. 6 The discrete signal indicating the response of PD control in terms of average weighted error with respect to time (P=2, D=1e-4).

IV. MAZE TRAVERSAL ALGORITHM

A line maze is made of a black or white line with a highly contrasting background. The line stretches across the given maze dimensions forming the maze. It forms various turns, intersections and dead ends between the start and the end. The types of intersections or turns are as shown in the Fig. 7. The Line Following Robot will first scan the maze in the dry run, and apply the shortest path algorithm to eliminate all redundant traversals in order to achieve the shortest path. The robot may encounter many dead ends in its first run as the robot typicallycannot traverse the maze without taking a number of wrong turns. This in turn helps it to solve for the shortest route in the next run.

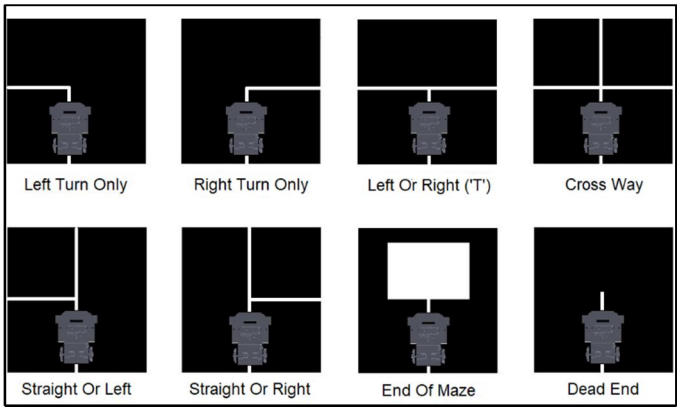


Fig. 7 Possible conditions a robot can encounter while traversing the maze.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.429
Volume 9 Issue IV Apr 2021- Available at www.ijraset.com

The eight cases when encountered need to be sorted into categories- turns and intersections. The actions taken by the robot with respect to the corresponding situation are as follows:

1) In the first two cases of *Left Turn Only* and *Right Turn Only*, the line path is extended in that direction. The robot has no choice except to follow the marked path. These turns need not be stored in memory for the shortest path calculation.

2) In the case of a *Dead End*, the robot has to make a 180-degree rotation. The dead-end signifies that the robot has made a wrong move. Hence, this case should be stored in memory for the shortest path calculation.

3) In the case of *Straight or Right Turn and Straight or Left* Turn, the robot needs to check if it is a dead-end. To do this the robot moves a short distance forward to check if the path continues or not. If it is a dead-end then it is stored in memory for shortest path calculation else, the robot continues to trace the path.

4) In the case of T junction and Cross Way, the robot should move a small distance forward after making a turn to check for dead-end. This turn should be stored in memory irrespective of its result.

In order to determine the shortest path to solve the maze, robot needs to traverse the maze twice.

a) *Dry Run:* The robot finds a way to the end of the maze via an imperfect route having multiple dead-ends and unnecessary turns. Decisions at some intersections are recorded in memory. This data is then used to determine the shortest path.

b) *Actual Run:* Once the robot traverses the dry run, the shortest path is determined. The robot then follows this path, with no dead-ends, from start to the end. The robot needs to be manually placed at the starting position in shortest path run.

The approach implemented to solve the maze is *Left Hand Rule* which prioritizes left turn over right turn or going straight. There is a possibility that robot will traverse the shortest path in the dry run. In that case, there will be no alteration made in the path during shortest path run. The line sensor can detect five cases in a maze. Table I indicates the value assigned to the five possible combinations of the sensor. A MATLAB function is developed that assigns value to these five cases.

Table I
Assigned values to the states of the six sensor array

Line	Sensor Value	Assigned value (Integer)
On Centre	0-0-1-1-0-0	1
On Left	1-1-1-1-0-0	2
On Right	0-0-1-1-1-1	3
Intersection	1-1-1-1-1-1	4
Dead end	0-0-0-0-0-0	5

Table II
Transition in the assigned value according to the change in the states of the six sensor array.

Case	Sensor at time t	Sensor at time **t + Δt**	Value Transition
Left Only	1-1-1-1-0-0	0-0-0-0-0-0	2 → 5
Right Only	0-0-1-1-1-1	0-0-0-0-0-0	3 → 5
T intersection	1-1-1-1-1-1	0-0-0-0-0-0	4 → 5
Cross Intersection	1-1-1-1-1-1	0-0-1-1-0-0	4 → 1
Straight / Left	1-1-1-1-0-0	0-0-1-1-0-0	2 → 1
Straight / Right	0-0-1-1-1-1	0-0-1-1-0-0	3 → 1
Dead End	0-0-0-0-0-0	0-0-0-0-0-0	5 → 5
End of Maze	1-1-1-1-1-1	1-1-1-1-1-1	4 → 4

Here, for the simplicity the sensor values are denoted as either 0 or 1 instead of 0 and 1000 respectively.

$$Sensor\ Value[i] = \begin{cases} 1, & sensor[i]\ is\ on\ the\ white\ line. \\ 0, & sensor[i] is\ on\ the\ black\ environment. \end{cases} \quad (4)$$

Where $1 \leq i \leq 6$ since array includes six individual sensors. At each intersection or turn, the robot intends to move further by a small distance to check the availability of additional path. If it exists, the robot continues to follow that particular additional path and make transitions accordingly. The transitions of the robot are explained in Table II below where the Line Value is taken from the Table I.

V. SHORTEST PATH ALGORITHM

The path array obtained from the dry run is used to determine the shortest path. This path array contains the path chosen at each intersection. To obtain the shortest path from the start of the maze to the end, all the dead ends must be replaced by an alternate path. To achieve this, whenever an x4x path sequence is encountered, the equivalent shortest path replaces it. The values of x can be 1, 2, 3 or 4. The x4x sequences are replaced by x00 sequence obtained by the following patterns:

Table III
The redundant sequences in path array with respective alternate values

Sub-sequences	Alternate value
[2,4,3] or [3,4,2]	[1,0,0]
[3,4,1] or [1,4,3]	[2,0,0]
[1,4,1]	[3,0,0]
[2,4,1] or [1,4,2] or [3,4,3]	[4,0,0]

1- Left turn 2- Right turn 3- Straight 4- Dead-end

After the sequences are replaced according to the respective pattern, the zeroes are shifted to the tail of the array. This loop continues until all the 4s in the path array are replaced if any. Considering the successful implementation of dry run on the mentioned maze in earlier sections, it gives the following path.

PathArray =

[1 1 4 1 3 3 1 1 1 3 4 1] →Path array received from dry run

[1 3 0 0 3 3 1 1 1 3 4 1]

[1 3 0 0 3 3 1 1 1 2 0 0]

[1 3 3 3 1 1 1 2 0 0 0 0] →Final Shortest path

The number of non-zero elements in the shortest path array indicates the exact number of intersections that the robot will encounter where it has to make a decision.

VI. STATEFLOW IMPLEMENTATION

Stateflow is a graphical language which is used to describe the way MATLAB algorithms and Simulink models react to input signals, events, and time-based conditions. With Stateflow, users can model combinatorial and sequential decision logic that can be simulated as a block within a Simulink model or executed as an object in MATLAB. Graphical animation enables to analyze and debug logic while it is executing. The Stateflow involves the entire algorithm from receiving the sensor inputs to determining the shortest path within a block which acts as a centralized system. It includes two fundamental blocks for dry run and shortest path run where only one can be activated at a time. Depending on the mode of operation, the inputs are given to the block.

A. Inputs to the Stateflow

1) LF, LM, LC, RC, RM, RF: As line sensor may give intermediate values between 0 and 1000. The readings from six individual sensors are applied to a threshold using threshold block and then supplied to the Stateflow.

2) velocity: For controlling the linear motion, it is required to provide a linear velocity threshold to the motor model block. Robot will be unable to exceed this velocity.

3) line_pos: A MATLAB function which evaluates line_pos value according to the line sensor readings. This value is used by the robot to know the path ahead.

4) delay distance: The time period in seconds in which the robot intends to move further in order to check the possibility of additional path.

5) delay_turn: In this time period, the robot ignores the line sensor reading to ignore the straight line provided the existence of the path to the left.

6) mode: This switches the functionality of the robot from dry run to shortest path run. Dry run can be used to find new paths in the same maze while shortest path eliminates the dead-end along that path

7) *Solved path:* After completing the dry run, shortest path array needs to be provided to the robot which is received from *solvedpath()* function.

B. Outputs from the Stateflow

Depending on the position of the robot, Stateflow gives following outputs which control the traversal of the robot.

1) *Velocity*: It is an intermediate value between 0 and the linear velocity threshold which is provided as input.
2) *Angular velocity:* It is the angular velocity corresponding to each motor of the robot. This velocity will be further converted into corresponding voltages in the motor model. This velocity also plays a role in ensuring steady movement of the robot.
3) *Path*: It provides simultaneous state of the path array to observe at the time of dry run. The array is updated as the robot passes each junction in the maze.
4) *Maze*: It is in the format of array which is the final state of path output. The same array is then passed to *solvedpath()* function.

There are two parameters which need to be regulated according to the desired linear velocity:

a) *delay_distance:* Distance delay to travel a short distance after intercepting a line value.
b) *delay_turn:* Sensor Delay while turning from cross, straight or left, straight or right cases to avoid transition to straight line directly.

C. Dry Run Stateflow

The stateflow includes *Action states* such as *Follow line*, *Go left*, *Go right*, *U turn* and *Stop*. During each Action state, linear velocity (v) and angular velocity (ω) are given as output. The intermediate states are *check left or straight, check right or straight, check intersection or maze end.* The intersections are again checked for cross or T intersection. Whenever the robot encounters a junction, the counter is incremented and the path is stored in an array. The *follow line* state has a Simulink function which calculates required angular velocity of robot using PD controller to follow straight line. Transition to intermediate state occurs when condition for particular state is true. Each time the transition occurs the linear velocity (v) and angular velocity (ω) are set to zero.

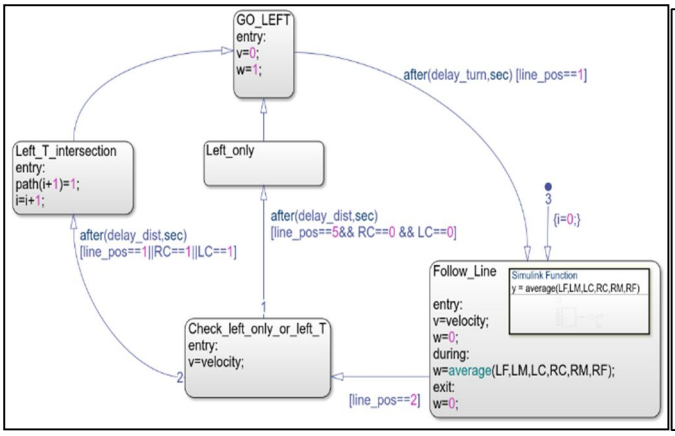


Fig. 8. Visual representation of Maze Traversal Algorithm in the form of the time of encountering a left path possibility.

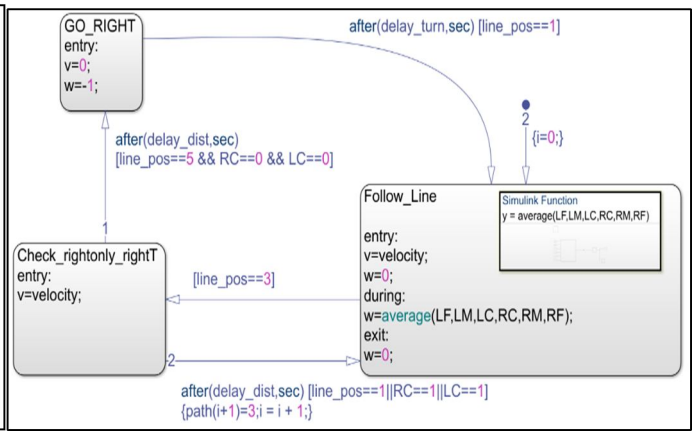


Fig. 9. Stateflow implementation representing algorithmic approach at the time of encountering a right path possibility.

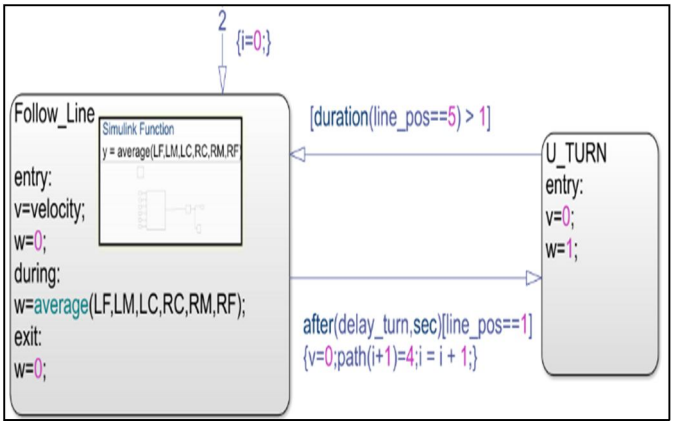


Fig. 10 Stateflow implementation representing algorithmic approach at the time of encountering a dead-end possibility.

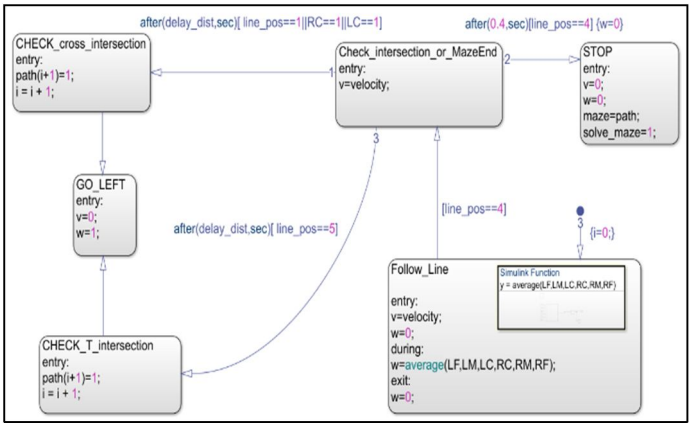


Fig. 11 Stateflow implementation representing algorithmic approach at the time of encountering an intersection path possibility.

When a state transition occurs from *Follow Line* to *check left or straight*, the robot again checks the line sensor value after distance delay. If the robot encounters a straight line, then it stores the left path in path array and turns left after turn delay to ignore straight line. The robot turns till it encounters the straight line and state transition occurs from *Go left* to *Follow line*. When the robot encounters only left turn, the path taken is not stored in path array. When a state transition occurs from *Follow Line* to *check right or straight*, the robot again checks line sensor value after distance delay. If the robot encounters a straight line, then it stores the straight path in path array and continues the straight path. When the robot encounters only right turn, a state transition occurs from *Follow Line* to *Go right* and the path taken is not stored in path array. While following a straight line, if the robot detects no line for more than a certain period, a state transition occurs from *Follow Line* to *U turn*. The robot rotates by an angle of 180 degrees around itself till it encounters white line. Being an incorrect decision, the path is stored in path array. When the robot intercepts an intersection, a state transition occurs from *Follow Line* to *Check intersection or maze end*. The intersection is checked whether it is a cross or a T by detecting a straight line after a distance delay. The robot turns left till it encounters a straight line and the path taken is stored in path array. To detect the end of maze after encountering an intersection, the line sensors are again checked for same intersection after a delay. When the robot reaches the end of the maze, a state transition occurs and the robot stops. The stored path is taken as output in the form of an array to calculate the shortest path.

D. Shortest Path Stateflow

To traverse the maze in shortest time avoiding dead ends, the mode of the stateflow is set to the shortest path run. In this mode, some inputs are disabled while other required inputs are enabled. The path array received from the dry run is processed by shortest path MATLAB function and given as input for shortest path run. Initially, the counter is set to zero. Whenever the robot encounters an intersection, the counter is incremented and the respective path from the shortest path array is selected. The function of the robot terminates when it reaches the end of the maze. It also involves the weighted average error function and PD control to follow the line. The shortest path stateflow does not store the path at the intersection as it already knows the decision at every intersection.

VII. KINEMATICS OF DIFFERENTIAL DRIVE ROBOT

The mechanical parameters are critical for designing an accurate control system of Line Following Robot. To understand the movement of a robot, it is important to consider the contribution of its wheels. The direction of robot can be changed by varying the relative rate of rotation of its wheels and hence does not require an additional steering force. To balance the robot, additional wheels or caster wheels may be added at the front part of the robot. For Line Following Robot application, Differential Drive mechanism is used. Differential Drive robot has two wheels that can turn at different rates and by turning the wheels at different rates you can make the robot move around. Typically, it has two wheels at the back and a caster wheel at the front. The velocity of each wheel can be controlled independently. For instance, if they are turning at the same rate, the robot is moving straight ahead. If one wheel is turning slower than another, then robot turns towards the direction in which the slower wheel is located.

A. Forward Kinematics

$$v = \frac{R}{2}(\omega_R + \omega_L) \qquad (5)$$

$$\omega = \frac{R}{L}(\omega_R - \omega_L) \qquad (6)$$

B. Inverse Kinematics

$$\omega_L = \frac{1}{R}(v - \frac{\omega_R}{2}) \qquad (7)$$

$$\omega_R = \frac{1}{R}(v + \frac{\omega_L}{2}) \qquad (8)$$

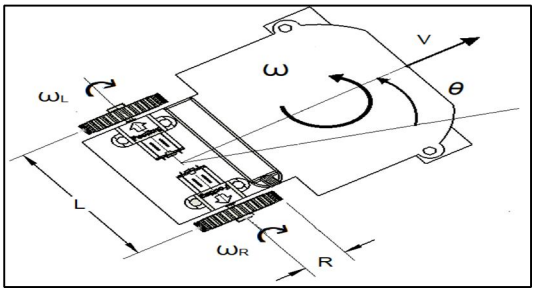


Fig. 12 Mechanism of differential drive considering the architecture of real-world robot model implementation.

As shown in Fig. 12, L is the perpendicular distance between the centre of the two wheels, ω_L and ω_R are the left and right wheel angular velocities respectively and R denotes the radius of the wheel. These expressions for linear velocity v and angular velocity ω contain most necessary information to plan the position of the robot. These parameters may vary depending on the architectural model of the robot. The Robot is very sensitive to the slight changes in velocity of each of its wheels. Small errors in the relative velocities between the wheels can affect the robot trajectory.

VIII. ROBOT SIMULATOR

The Robot Position and orientation can be placed using Robot Simulator Block. The differential robot pose [x y θ] is updated using left wheel angular velocity ωL and right wheel angular velocity ωR.

The dynamics of differential robot are given by,

$$\dot{X} = \frac{R\cos\theta*(\omega R + \omega L)}{2} \quad (9)$$

$$\dot{Y} = \frac{R\sin\theta*(\omega R + \omega L)}{2} \quad (10)$$

$$\dot{\theta} = \frac{R*(\omega R - \omega L)}{L} \quad (11)$$

The pose of the robot is updated with time interval Δt as,

$$X_{k+1} = X_k + \Delta t * X \quad (12)$$

$$Y_{k+1} = Y_k + \Delta t * Y \quad (13)$$

$$\theta_{k+1} = \theta_k + \Delta t * \theta \quad (14)$$

Where X is position along x-axis, Y is position along y-axis, θ is the angle of the robot with respect to x axis on binary map, Δt is the sampling time, XK, YK, θK are the position at instance of time t. For simulation, Simulink blocks are used to get robot's forward and inverse kinematics. Although, it is a completely autonomous robot, the starting position is given to Robot Simulator block which is considered as (X0, Y0, Θ0). It determines the next coordinates at interval Δt according to the (12), (13) & (14) and maps them on the maze. The derived position is given to the Robot Simulator block which updates the robot position over the binary occupancy map of the maze. The sampling time is set to 0.01. The Inverse kinematic block takes robot's linear velocity and angular velocity as inputs and converts it into left wheel angular velocity and right wheel angular velocity as outputs using robot parameters as in (7) & (8). Similarly, as per (5) & (6), forward kinematic block takes Left wheel angular velocity and right wheel angular velocity as input and converts it into linear velocity and angular velocity as output which makes the robot to move around. The robot uses Axle Length and wheel radius as its parameters for simulation blocks.

IX. LINEAR TIME INVARIANT MOTOR MODEL

A Motor model is created to get the wheel angular velocity response similar to an actual DC motor. A simple model of a DC motor driving an inertial load shows the angular rate of the load, ω(t), as the output and applied voltage υapp (t) as the input. The goal is to control the angular velocity rate by varying the applied voltage. In this model, the dynamics of the motor itself is idealized. For instance, the magnetic field is assumed to be constant. With this model and laws of physics, it is possible to develop differential equations that describe the behavior of this electromechanical system. The relationships between electric potential and mechanical force are given by Faraday's law of induction and Ampère's law for the force on a conductor moving through a magnetic field.

A. Mathematical Derivation

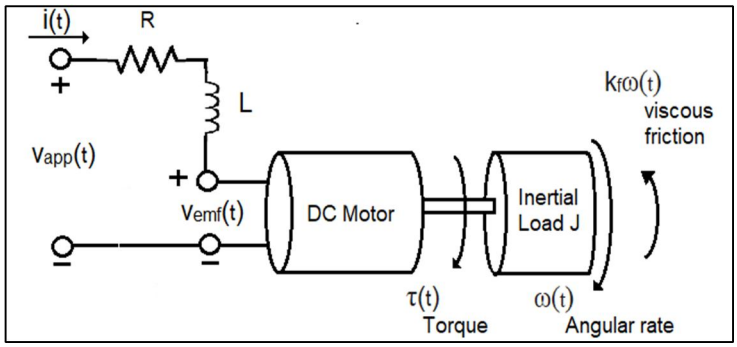


Fig. 13 Underlying mechanism of Linear Time Invariant (LTI) Motor Model.

The mechanical part of the motor equations is derived using Newton's law, which states that the inertial load is J times the derivative of angular rate equals the sum of all the torques about the motor shaft. Hence, the equation is,

$$J\frac{d\omega}{dt} = \sum \tau_i = -K_f\omega(t) + K_m i(t) \quad (15)$$

where Kfω(t) is a linear approximation for viscous friction and the term Kmi(t) represents the torque τ at the motor shaft, Km, the armature constant.

The back (induced) electromotive force, υemf, is a voltage, proportional to the angular rate ω at the shaft.

$$v_{emf}(t) = K_b.\omega(t) \quad (16)$$

where K_b, the emf constant, also depends on certain physical properties of the motor. Finally, the electrical part of the motor equations can be described by

$$v_{app}(t) - v_{emf}(t) = L\frac{di}{dt} + Ri(t) \quad (17)$$

Equation (17) gives two differential equations that describe the behaviour of the motor.

$$\frac{dy}{dx} = -\frac{R}{L}i(t) - \frac{K_b}{L}\omega(t) + \frac{1}{L}v_{app}(t) \quad (18)$$

$$\frac{d\omega}{dt} = -\frac{1}{J}K_f\omega(t) + \frac{1}{J}K_m i(t) \quad (19)$$

B. State-Space Equations for the DC Motor

Considering the (18) & (19), one is able to develop a state-space representation of the DC motor as a dynamic system. The current *i* and the angular rate ω are the two states of the system. The applied voltage, υapp, is the input to the system, and the angular velocity ω is the output.

$$\frac{d}{dt}\begin{bmatrix} i \\ \omega \end{bmatrix} = \begin{bmatrix} -R/L & -K_b/L \\ K_m/J & -K_f/J \end{bmatrix}.\begin{bmatrix} i \\ \omega \end{bmatrix} + \begin{bmatrix} 1/L \\ 0 \end{bmatrix}.v_{app}(t) \quad (20)$$

(A) (B)

$$y(t) = [0\ 1].\begin{bmatrix} i \\ \omega \end{bmatrix} + [0].v_{app(t)} \quad (21)$$

(C) (D)

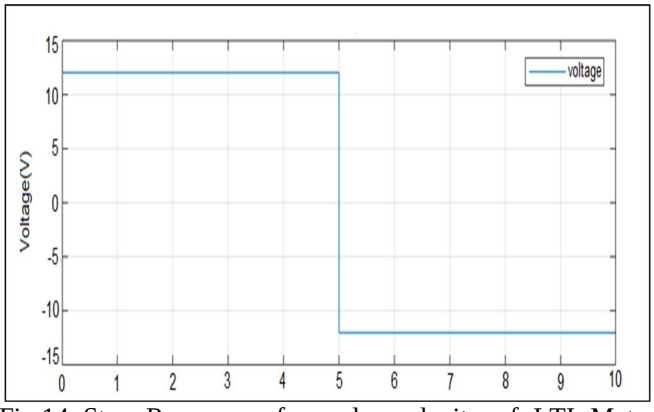


Fig.14 Step Response of angular velocity of LTI Motor according to voltage variation with respect to time

The state-space representation is constructed using the Simulink State Space block. In state-space motor model block, (20) & (21) are needed to be specified in the form,

$$\frac{dx}{dt} = Ax + Bu \quad (22)$$

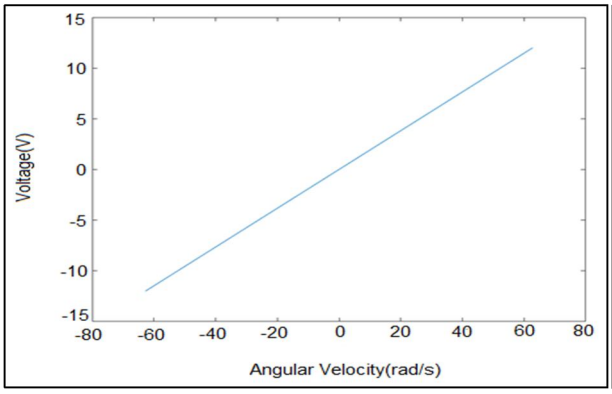
$$y = Cx + Du \quad (23)$$

These parameters can be derived by estimating parameters of a DC motor from measured input and output data using Simulink Design Optimization tool. In this case, the parameters of 12V-600rpm Micro Metal Gear Motor is taken into consideration for motor modelling.

Table IV
Parameters of 12V-600 rpm Micro Metal Gear Motor.

Parameter	Description	Unit	Value
R	Armature resistance	Ohm	16
L	Armature inductance	H	0.5
Km	Torque constant	Nm/A	0.084
Kf	Rotor damping coefficient	N*m/(rad/s)	0.000025
Kb	EMF constant	V/(rad/s)	0.19
J	Rotor moment of inertia	kg.m^2	0.0001

As shown in Fig. 14, although there is immediate voltage drop, due to Rotor's inertia, the resulting angular velocity falls gradually. Before adding the LTI motor model in simulation, the relation between Angular velocity of motor and Input Voltage is derived by varying the voltage from -12V to 12V. The obtained values are specified in a 1-D Lookup Table block.



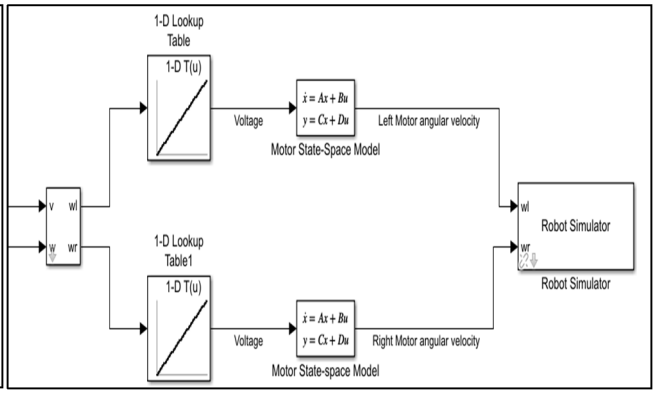


Fig. 15. Relationship between angular velocity and voltage required Space Model. Fig. 16. Robot Simulator Block with Motor State for 1-D Lookup Table Block

The output (v, ω) from Stateflow is given to 1-D lookup table. The lookup table output is the voltage corresponding to the angular velocity of the left and right motor. This voltage is supplied as input to the Motor State-Space block. The output of the motor is given to the Robot Simulator block for mapping the motion on two-dimensional output screen on computer.

X. OUTPUT

The robot simulator block is used to visualize the Line Following Maze Solver Robot on a binary occupancy map. The parameters such as linear velocity, PD gain constants, distance delay and turn delay are calibrated until the robot achieves steady transient motion. Stateflow offers visualization of instantaneous state transition, thus providing ease to the developers while debugging the algorithm. The MATLAB simulation model can be used for multiple robots having similar architecture by modifying some parameters. Each block in Fig. 17 shows the implementation that is explained in earlier sections. During the testing phase, the mode is set to 0 for dry run simulation initially to obtain a path traversed through the maze. The obtained path is passed to a MATLAB function block as parameter to determine the shortest path if any. For the actual run, shortest path information is used while setting the robot in mode 1. A state in the stateflow is highlighted corresponding to the activated condition along with the interconnections. Hence, real time visualization can be done to debug the algorithm in case of malfunctioning of the robot.

The goal is achieved as the robot takes the shorter path in the actual run than dry run. We can observe in Fig. 18, the dark shaded path indicates dry run traversal while the light shaded path is resulting from the shortest path run. Two wrong decisions taken by the robot at two different intersections in dry run are eliminated at the time of shortest path run, thus solving the maze in minimum possible time.

A set of experiments have been carried out to validate the simulation. The results in the Table IV show the variation in time duration to complete the Dry run and Shortest path run. The robot parameters such as PD values, distance delay, turn delay, etc. have to be adjusted to get the desired results. It has been found that the variations in turn velocity significantly improves the time taken by the robot during dry and shortest path run.

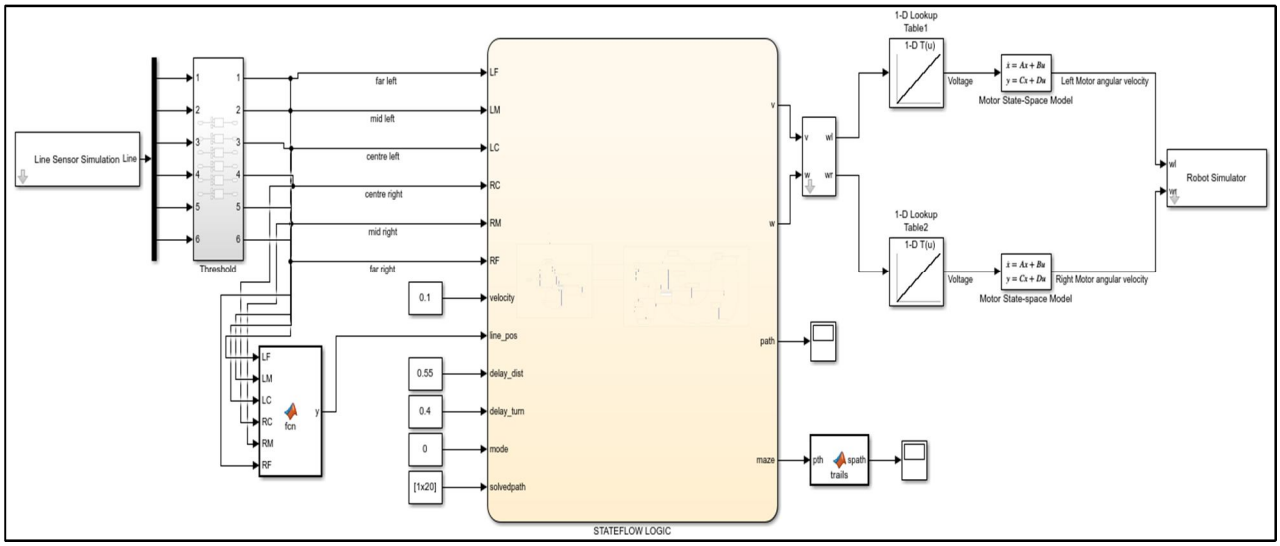


Fig. 17 Complete Simulation using Simulink blocks.

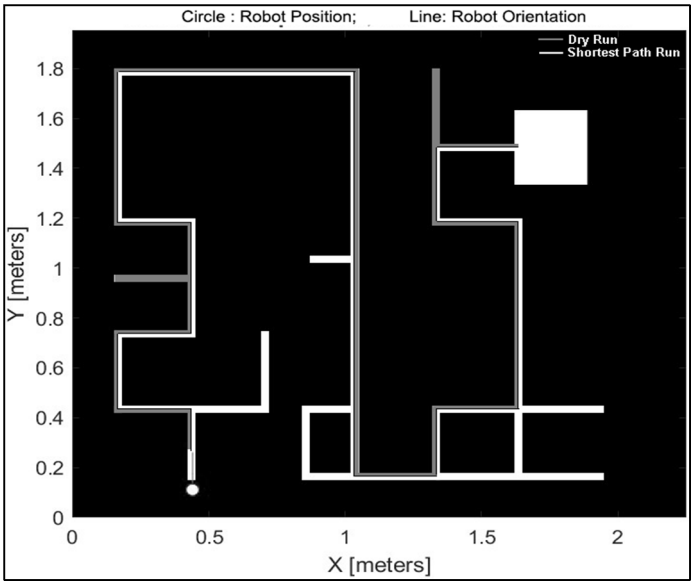


Fig. 18 Output Path Resulting from the Dry Run Traversal and Shortest Path Run Traversal

Table v
Effect on runtime due to variation in velocities.

Linear Velocity m/s	**Turn velocity Rad/s**	**Dry run time (s)**	**Short run time (s)**
0.1	1	121.29	98.18
0.1	2	104.6	85.25
0.2	1	80.44	63.45
0.2	2	62.63	51.74
0.3	1	63.76	49.85
0.3	2	48.40	38.35
0.4	1	55.75	42.64
0.4	2	40.87	31.38
0.5	1	51.55	39.15
0.5	2	36.46	28.05

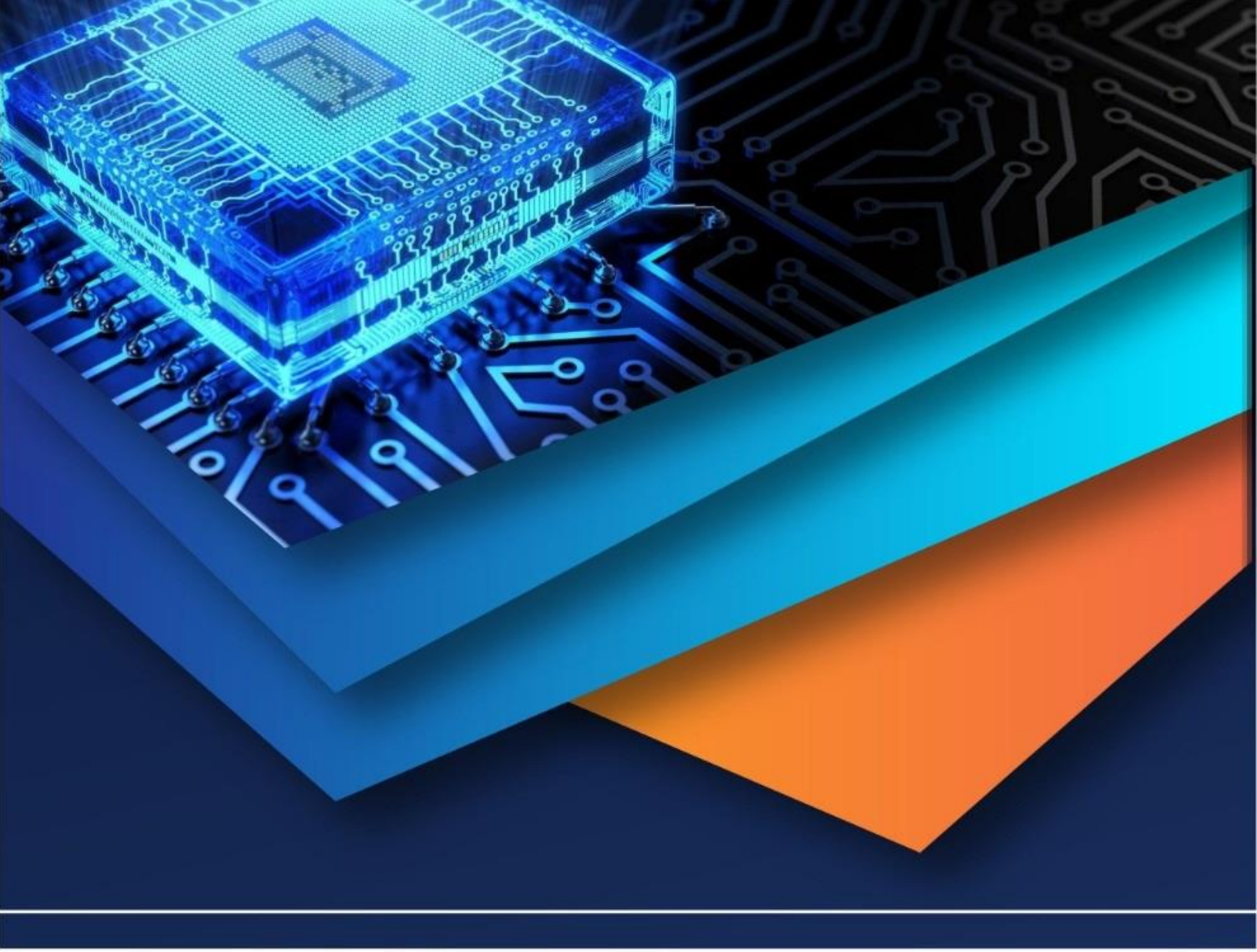
XI. CONCLUSION

This paper explains the simulation workflow of Line Following Maze-solver robot in MATLAB. This software is selected as it includes the mobile robotics library which offers multiple features required for the proposed simulation having compatibility on various platforms along with detailed online documentation. In comparison with real world testing, simulation offers the absolute approach by enabling an individual to operate from anywhere irrespective of the location. It is possible to achieve the precision of the output model with least error despite the unavailability of the resources and reduces the development time for entire hardware model.

Any changes in the characteristic components such as DC motor and line sensor array can significantly affect the simulation and hence, the corresponding parameters must be adjusted accordingly. The received simulation parameters can be further used in hardware such as Arduino microcontroller for real time testing and debugging. However, in some cases, it may be require to calibrate the parameters as the actual behavior of the components differs from the expected behavior. The robot must be traversed in both the modes whenever it resets as the path information is stored in volatile memory thus it gets wiped on shutting down. In order to retain the information, we can store it in non-volatile memory. Line Following Maze-solver robot has countless applications in healthcare services and industries such as automated carriers replacing the conveyor belts and in robot based food serving techniques in restaurants.

REFERENCES

[1] K. Norbert-Brendan and T. Cristian Marius, "Autonomous Line Maze Solver Using Artificial Intelligence," 2019 15th International Conference on Engineering of Modern Electric Systems (EMES), Oradea, Romania, 2019, pp. 133-136, doi: 10.1109/EMES.2019.8795101.
[2] Bienias, Lukasz & Szczepański, Krzysztof & Duch, Piotr. (2016). "Maze Exploration Algorithm for Small Mobile Platforms." Image Processing & Communications. 21. 10.1515/ipc-2016-00113.
[3] "PID Control for Electric Vehicles Subject to Control and Speed Signal Constraints." Amanda Danielle O. da S. Dantas,1 André Felipe O. de A. Dantas,2 João Tiago L. S. Campos,2 Domingos L. de Almeida Neto,2 and Carlos Eduardo T. Dórea. Journal of Control Science and Engineering (2018)
[4] Reddy, H.K. & Immanuel, J. & Parvathi, C.S. & Bhaskar, P. & Sudheer, L.S.. (2011). "Implementation of PID controller in MATLAB for real time DC motor speed control system. Sensors & Transducers." 126. 110-118.
[5] Nurmaini, Siti & Dewi, Kemala & Tutuko, Bambang. (2017). "Differential-Drive Mobile Robot Control Design based-on Linear Feedback Control Law." IOP Conference Series: Materials Science and Engineering. 190. 012001. 10.1088/1757-899X/190/1/012001.
[6] Silva, Pedro & Trigo, Antonio & Varajão, João & Pinto, Tiago. (2010). "Simulation - Concepts and Applications." 429-434. 10.1007/978-3-642-16324-1_51.
[7] Kevin M. Lynch and Frank C. Park (May 3 2017). "Modern Robotics Mechanics, Planning and Control"
[8] C. Engineering, S. P. Yadav, V. K. Tripathi, and M. T. Student, "A Case Study of DC Motor Speed Control with PID Controller through MAT LAB," International Journal of Advanced Research in Computer and Communication Engineering, vol. 5, no.5, pp.1008–1011,2016.
[9] K. W. G. Michael, "Auto-Tuning: From Ziegler-Nichols to Model Based Rules," J. Chem. Inf. Model., vol. 53, p. 160, 1989
[10] Osorio, Román, et al. "Inteligent line follower mini-robot system." International Journal of Computers, Communications & Control 1.2 (2006): 73-83.
[11] G. Shahgholian and P. Shafaghi, "State space modeling and eigenvalue analysis of the permanent magnet DC motor drive system," 2010 2nd International Conference on Electronic Computer Technology, Kuala Lumpur, 2010,
[12] pp. 63-67, doi: 10.1109/ICECTECH.2010.5479987.Ruderman, Michael & Krettek, Johannes & Hoffmann, Frank & Bertram, T.. (2008). Optimal State Space Control of DC Motor.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY