LINE FOLLOWING ROBOT
WITH MAZE SOLVING APPLICATION

By,

Ankit Anand

B.Tech. (2nd year), EE

Meet Gandhi

B.Tech. (2nd year), ECE

Deepeshwar Kumar

B.Tech. (3rd year), EE

S V National Institute of Technology, Surat

AUTONOMOUS ROBOT

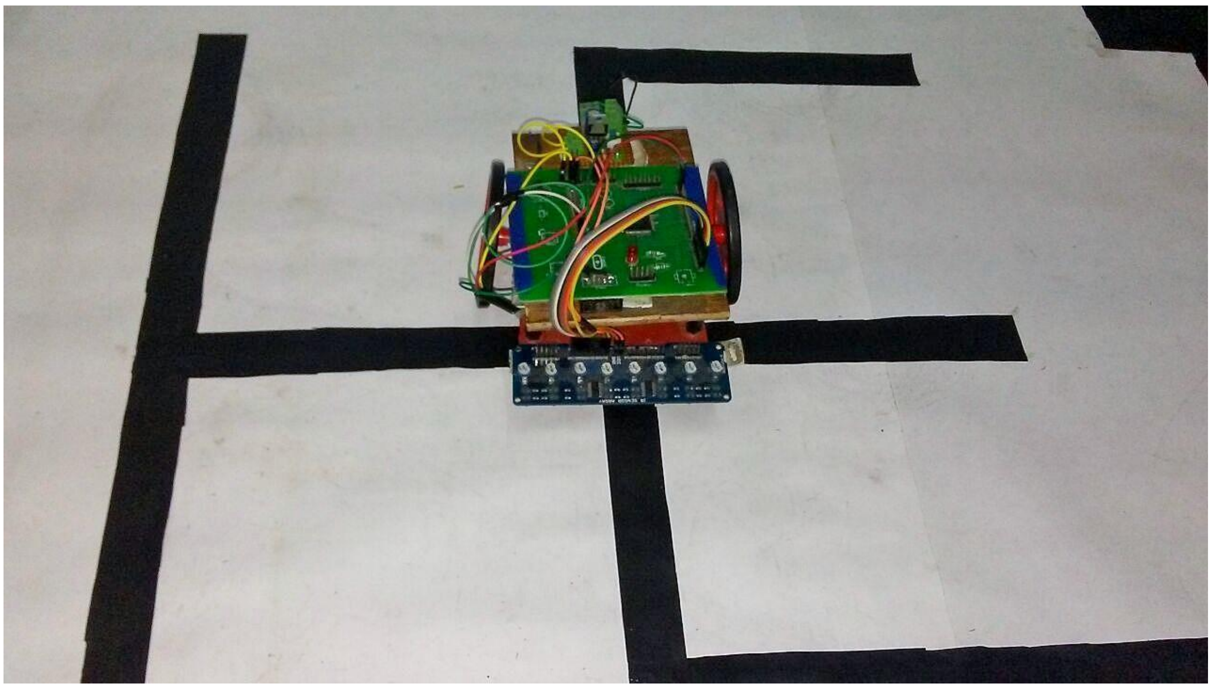
An autonomous robot is one with the ability to behave by taking decisions autonomously in accordance with its environment, with little or no human intervention.

An autonomous robot is able to gain information about its environment and work for an extended period without human assistance.

LINE FOLLOWER ROBOT

A line follower is a very basic autonomous robot which can be programmed to follow a black line on a white surface or a white line on a black surface.

It is basically a robot which follows a particular path or trajectory and decides its own course of action which interacts with obstacles.



BASIC METHOD OF LINE FOLLOWING

In order to detect a line, let's say a black line on a white surface or vice versa, we use infrared or IR sensors.

IR sensors work by using a specific light sensor to detect a selected light wavelength in the invisible IR spectrum.

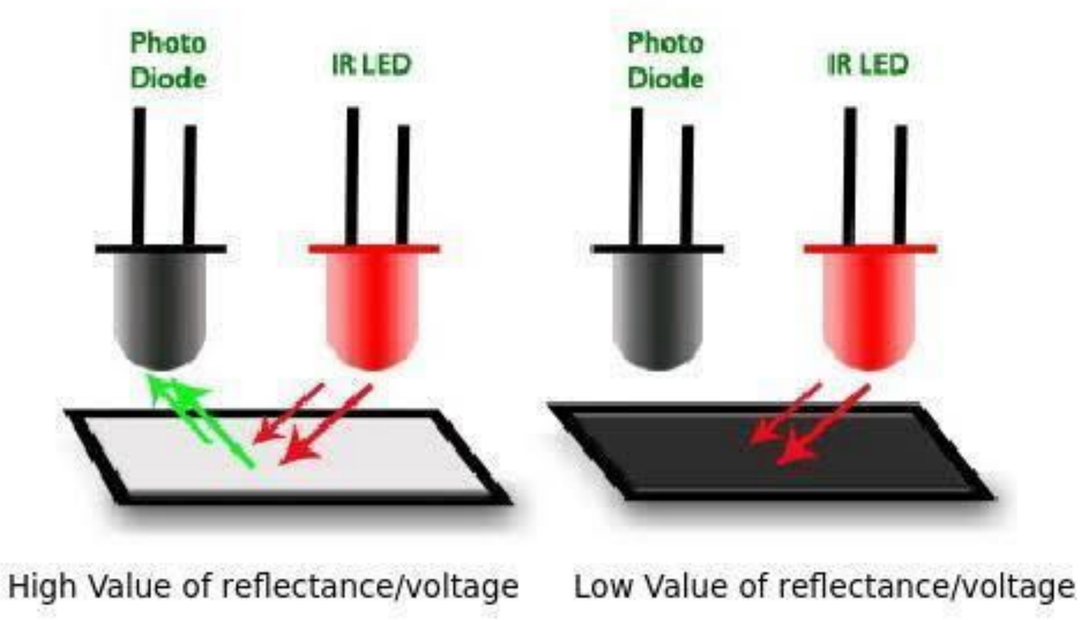


Photo Diode
IR LED
Photo Diode
IR LED
High Value of reflectance/voltage
Low Value of reflectance/voltage

An IR sensor consists of two diodes- an Infrared Light Emitting Diode (IR LED) and a Photo Diode.

An IR LED emits infrared rays. These rays are almost completely absorbed by a black surface (low reflectance) and almost completely reflected by a white surface (high reflectance).

A Photo Diode conducts electricity when IR rays fall on it, otherwise no current will pass through it.

When working in pair, these can used to identify a black or white surface. When there is black surface the IR rays will be absorbed and current through photodiode will be zero (the sensor will return 0 logic value).

When there is white surface the IR rays will be reflected and current through photodiode will be non-zero (the sensor will return 1 logic value).

Hence by using an IR sensor array, we can detect a black line on a white surface. For example, if an 8 IR sensor array returns values, S0=1, S1=1, S2=1, S3=0, S4=0, S5=1, S6=1, S7=1, then the black line is going through the middle of the sensor array (Sn represents pins connected to each sensor and returning a logic value 0 or 1). This value can be represented as an 8 digit binary number 11100111.

Hence combination of different values can be used to detect the position of the black line on the white surface and the bot can be accordingly programmed to achieve line following.

The IR sensors used have a potentiometer attached with the circuit. This potentiometer can set the range of distance for which the sensor has to work. Hence sensors can be calibrated for different positions.

There is also an advanced auto-calibrating and more accurate line sensor array available (e.g. model LSA08), which converts Boolean logic values like 11100111 into their corresponding decimal value, here 231, which is given as output and can be used as input to microcontroller.

PARTS USED BY US

A two floored chassis with two wheels

Two toy motors having maximum rpm 1000
ATmega 128 microcontroller

Cytron motor driver
circuit 12 V Li-ion battery

8 IR sensor array

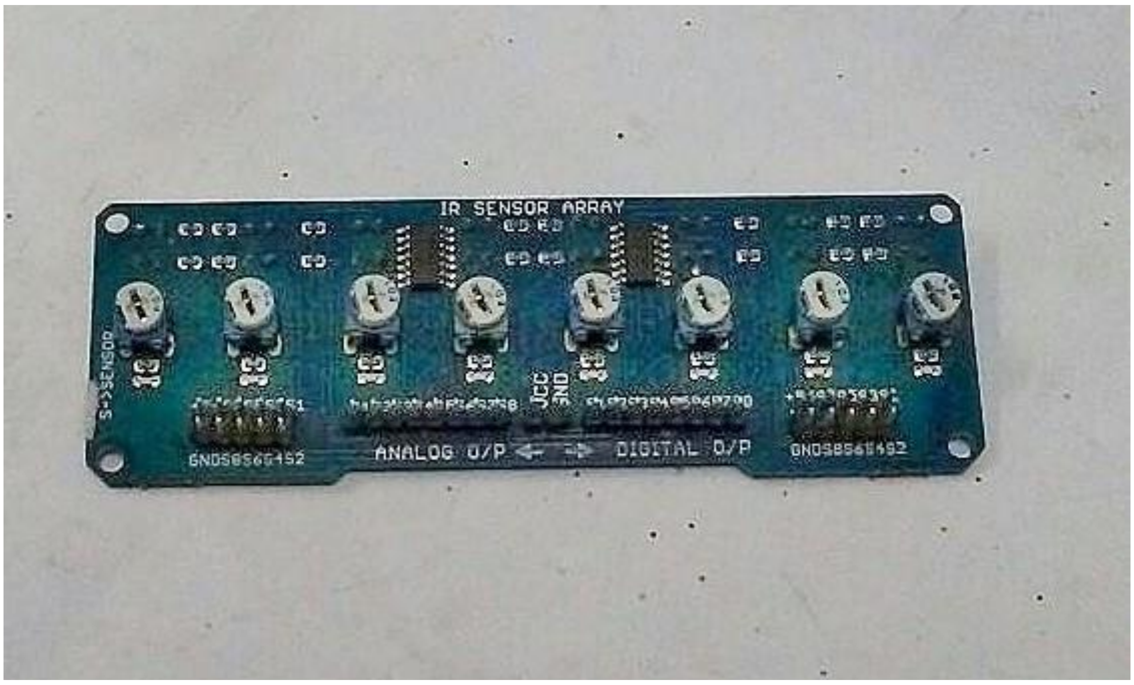
A voltage divider circuit using LM7805 IC
Jumper wires to make connections

Double-sided and electric insulation tapes

Two floored chassis with toy motors and wheels mounted on it

ATmega128 microcontroller

Cytron motor driver circuit



8 IR line sensors array

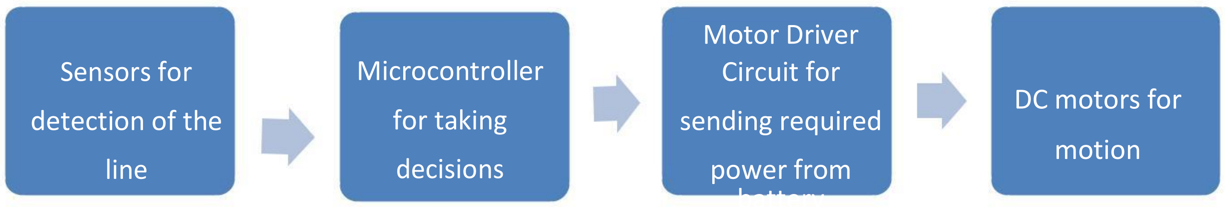
Voltage divider circuit using LM7805 IC

12V Li-ion battery

Jumper wires

BASIC DESIGN OF THE BOT

The line following robot used in this project can be considered to be made of four major blocks.

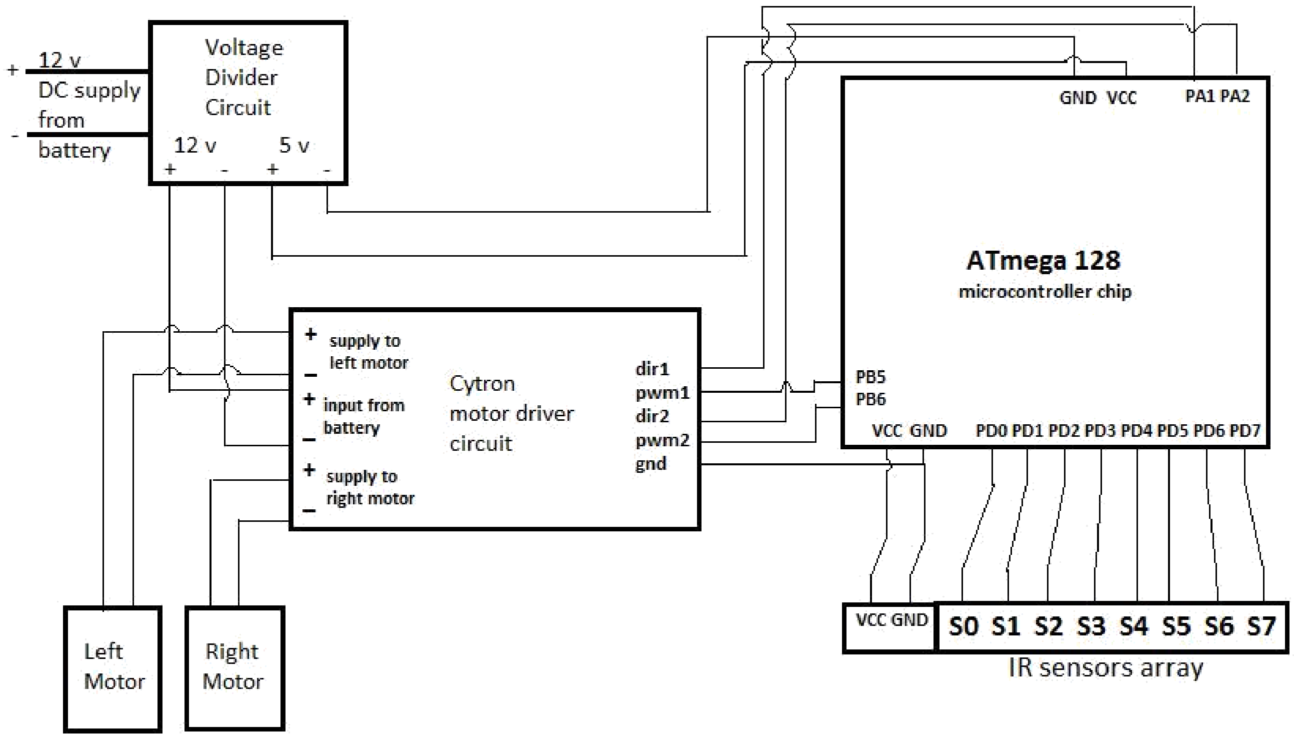

Sensors for detection of the line
Microcontroller for taking decisions
Motor Driver Circuit for sending required power from
DC motors for motion

The robot is powered using a 12v rechargeable Li-ion battery. A voltage divider circuit is made using the LM7805 IC. This circuit receives 12v supply from the battery and gives 5v supply to the microcontroller and 12v supply to the motor driver circuit.

The microcontroller receives input from the sensors and gives digital signals as output, ranging between 0 to 5v, to the motor driver circuit. The motor driver sets this signal on a scale of 0 to 12v and provides running power from the battery to the motors.

The motors do rotation accordingly and give motion to the robot. The motor driver used here, Cytron also provides direction pins.

The pins given on the Cytron motor driver circuit are dir1, pwm1, dir2, pwm2 and gnd. The pwm pins are for receiving voltage levels from the microcontroller. A dir pin when set high gives a motor rotation in a specific direction and when set low, in the opposite direction, this can be used for forward and backward rotations of the motor. The pins dir1 and pwm1 are used for one motor and the pins dir2 and pwm2 are used for a second motor. The gnd pin is for grounding the circuit.



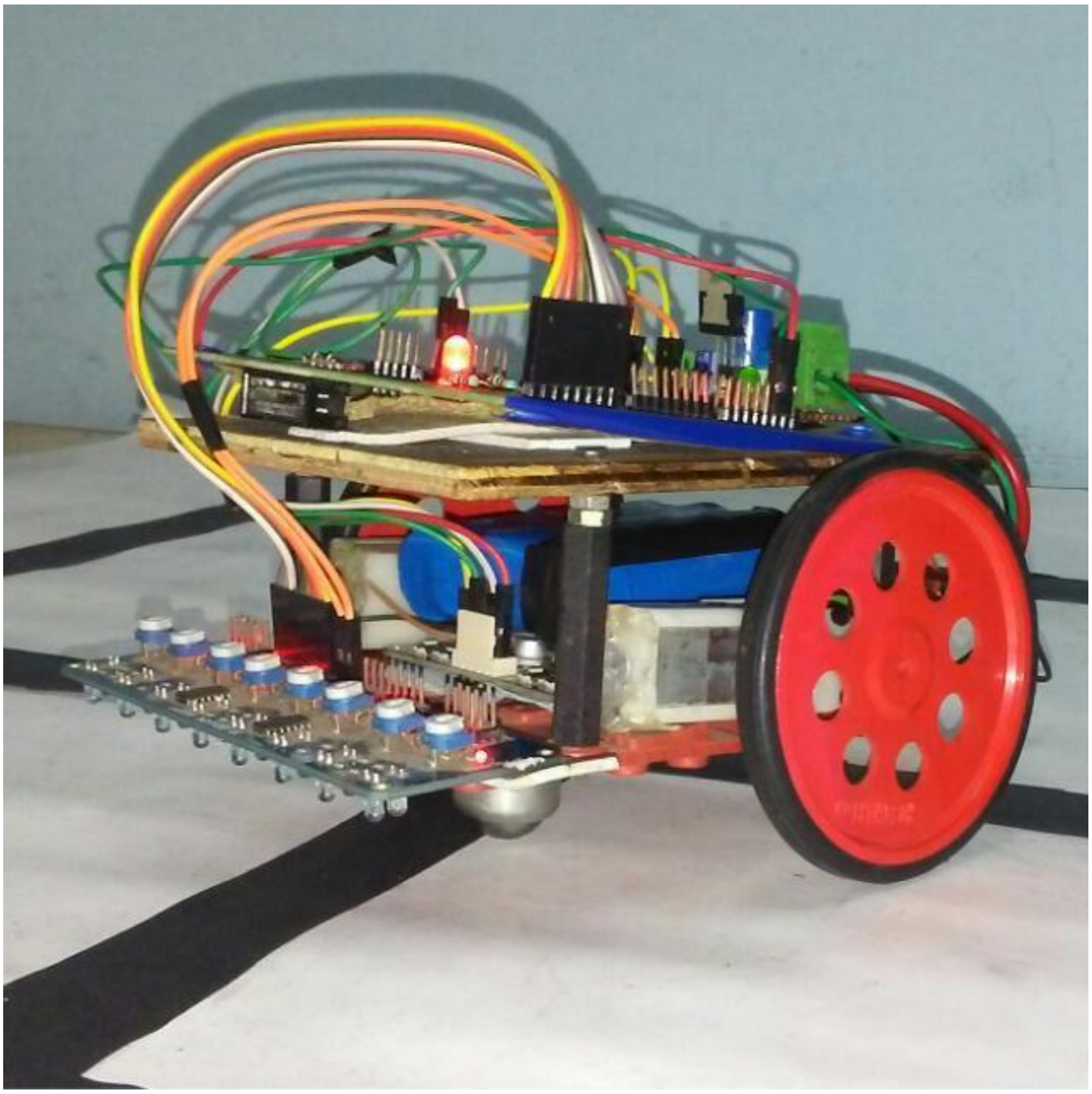
Here we will use PORT D for receiving input from the IR sensor array. Pin PD0 is connected to leftmost sensor S0 and pin PD7 is connected to rightmost sensor S7, all other sensors are connected accordingly.

The pins PB5 and PB6 are connected to pwm1 and pwm2 respectively. These pins give voltages, between 0 to 5v, which is scaled to 0 to 12v by the motor driver and is given to the motors. The voltage decides speed of rotation. The pin pwm1, i.e. pin PB5, is used for rotating the left motor and the pin pwm2, i.e. the pin PB6, is used for rotating the right motor.

PA1 and PA2 are connected to dir1 and dir2 respectively. When kept high, the motors do forward rotation, when kept low, the motors do reverse rotation. The pin dir1, i.e. PA1 is for direction of left motor and the pin dir2, i.e. PA2 is for rotation of right motor.

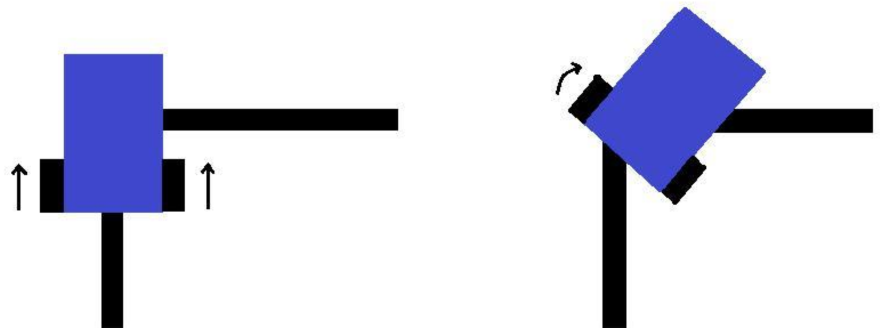
The gnd pin on the motor driver is connected to the ground pin on the microcontroller. Hence microcontroller and motor driver have a common ground.

Hence any port or pin can be used for different purposes. The only thing is that the bot is able to do proper motion at a proper speed.



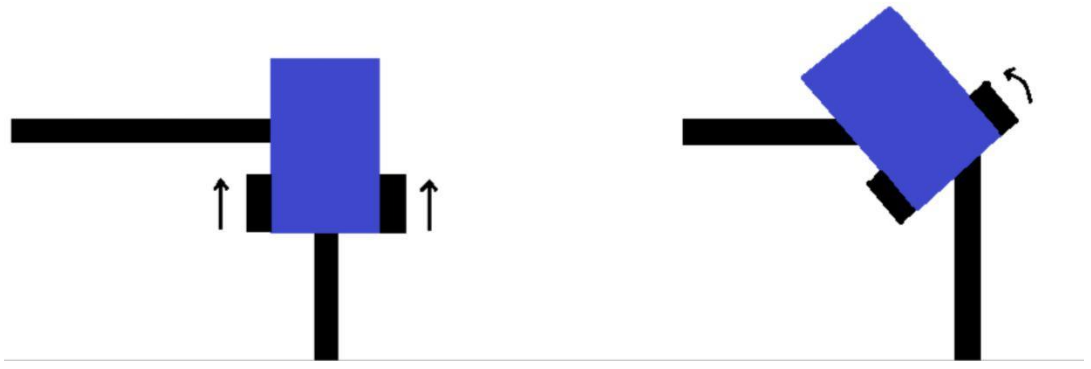
line follower bot

TURNING MECHANISM OF THE BOT



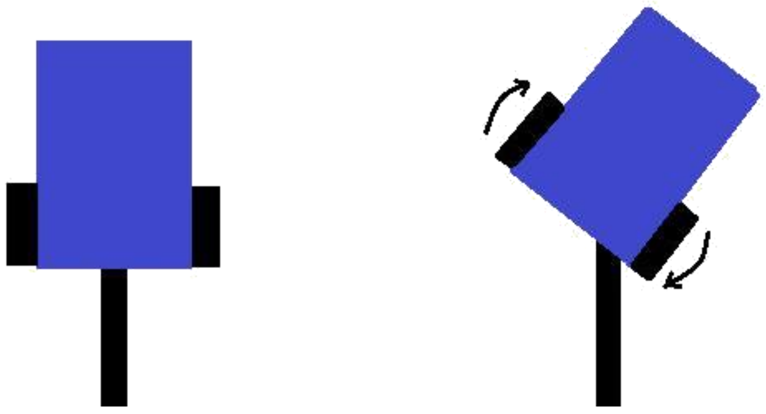
right turning of the bot

To make the bot turn right, the right wheel is stopped and the left wheel continues to rotate till the bot again comes back on straight path, after which both wheels start rotating with same speed.



left turning of the bot

To make the bot turn left, the left wheel is stopped and the right wheel continues to rotate till the bot again comes back on straight path, after which both wheels start rotating with same speed.



reverse turning at dead end

At a dead end in the path, the bot needs to do a reverse turn. For achieving this, both the motors are made to rotate in reverse direction, till the bot comes back on straight path. After this, the bot moves with both motors in forward direction.

In each case, the turning of the wheels has to be tuned for time intervals using delay functions in the program of the bot.

BASIC CODE FOR LINE FOLLOWING

A basic code in C, implementing these techniques is given.

This program is created for paths that are either straight or have sharp 90 degree turns.

```
//line following code

//black line on white surface

#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>

int readSensors();                    //function for reading sensor output

int mode,i, s[8];

int main(void)
{
        DDRD=0x00;              //declaring D port as input port
        DDRB|=0xFF;             //declaring B port as output port
        DDRA|=0xFF;             //declaring A port as output port

        while(1)
        {
                mode = readSensors();
                PORTA = 0b01100000;

                switch (mode)
                {
                        case 231:       //11100111 cases for straight path
                        case 207:        //11001111
                        case 243:       //11110011
                        case 199:       //11000111
                        case 227:       //11100011
```

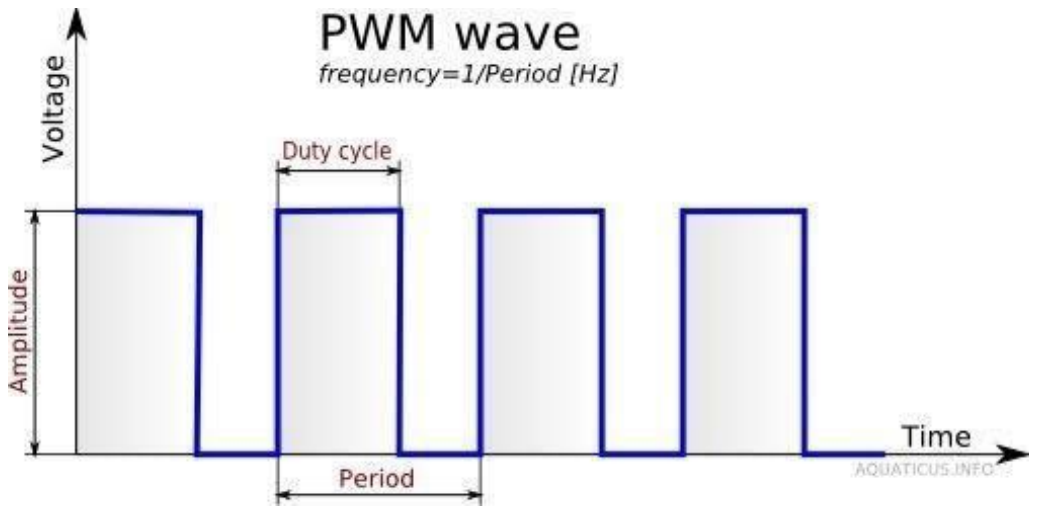
```
                    PORTB = 0b00000110;
                    break;

                case 15:        //00001111 cases for 90 degree left turn
                case 7:         //00000111
                case 31:        //00011111
                    PORTB = 0b00000010;
                    _delay_ms(300);
                    break;

                case 248:       //11111000 cases for 90 degree right turn
                case 240:       //11110000
                case 224:       //11100000
                    PORTB = 0b00000100;
                    _delay_ms(300);
                    break;

                default:
                    PORTB = 0b00000110;
            }
        }
}

int readSensors()
{
        s[8] = 0;
        int k=0;
        for(i=0;i<8;i++)
        {
                if(bit_is_set(PIND,i))  {s[i] = 1;}
                else                    {s[i] = 0;}
        }
        k = s[7] + s[6]*2 + s[5]*4 + s[4]*8 + s[3]*16 + s[2]*32 + s[1]*64 +
            s[0]*128;           //s[0] is leftmost and s[7] is rightmost

        return(k);
}
```

The pins PA1 and PA2 are high in this code, as we are aiming only for forward rotation of motors.

The delay values to be given to the _delay_ms() function has to be tuned experimentally for different turnings. The delay values decreases with increase in the rpm of motors used. The above code is very basic and won't give a very stable bot. For that purpose we need to apply pulse width modulation and PID control algorithm techniques.

PULSE WIDTH MODULATION (PWM)

A digital device like a microcontroller can give only two voltage values, 5V or 0V (5V for logic 1 and 0V for logic 0). This 0-5V voltage range can be magnified to a 0-12V voltage range with the help of an external power source and a motor driver circuit (higher voltage ranges can be obtained with other batteries and other motor drivers). So if a MCU returns these two values only, it can either run a motor at full rpm or 0 rpm. If we need to obtain a speed less than full, we need to obtain a voltage less than 5V from the MCU. This can be achieved through pulse width modulation (PWM) technique.



Pulse-width modulation uses a rectangular pulse wave whose pulse width is modulated resulting in the variation of the average value of the waveform. Hence analog voltage levels between 0 to 5 volts can be obtained with the help of this technique.

For a time period T of the wave, let the voltage be high for time T_{on} and be low for time T_{off}. Then duty cycle of the wave is defined as,

Duty cycle = $\frac{T_{on}}{T_{on}+T_{off}} \times 100\% = \frac{T_{on}}{T} \times 100\%$

The voltage obtained as output is averaged over the entire waveform.

Output Voltage = (Duty Cycle %) × maximum voltage.

Maximum voltage in the case of an atmega microcontroller is 5 volts.

Since this technique doesn't give any real analog voltage but the output voltage appears to be at an analog level, the speed of the PWM signal should be very fast to achieve this goal. Hence PWM is possible only with high frequency devices.

TIMER1 AND PWM GENERATION

Microcontrollers have dedicated circuits for specific applications. Such circuits are called peripherals. Timers are important peripherals.

A timer basically counts from 0 to a TOP value and then overflows back to 0. When it overflows it signals the processor for some action, if some action has to take place.

Here we will use a 16 bit timer, timer1 of the ATmega128 chip for generation of Fast PWM wave. For a 16 bit timer the maximum limit of the TOP value is 2^{16} i.e. 65536.

The 16-bit timer, Timer 1 on ATmega128 can be accessed through pins PB5 and PB6, referred as two channels of Timer1, channel A (PB5) and channel B (PB6).

Specific values have to be set in the Timer/Counter Control Registers (TCCR1A and TCCR1B) for Fast PWM generation.

Timer/Counter1 Control Register A (TCCR1A)

Bit	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10
Initial Value	0	0	0	0	0	0	0	0

All the bits in the TCCR1A register are read/write type.

Timer/Counter1 Control Register B (TCCR1B)

Bit	7	6	5	4	3	2	1	0
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Initial Value	0	0	0	0	0	0	0	0

All the bits in the above register are read/write type except register 5 (which is read only).

The clock speed to be used in the program is defined in the beginning of the program in the following manner,

```
#define F_CPU 8000000UL
```

This defines a processing speed of 8MHz for the program, referred as clock source output $Clk_{I/O}$. The clock source for the timer can be selected using the Prescalar.

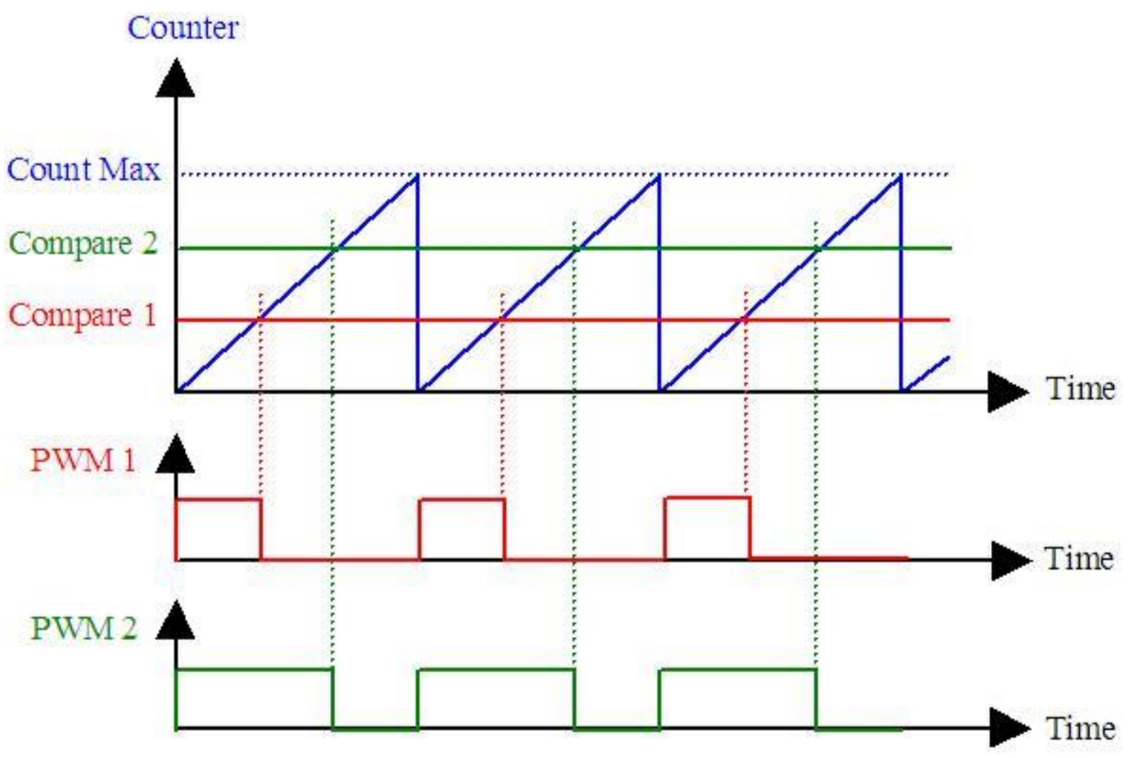
The prescalar is an electronic counting circuit used to reduce a high frequency electrical signal to a low frequency by integer division. The prescalar can be accessed using the CS1n (clock select) bits in TCCR1B. A fraction of the frequency of F_CPU can be used for timer using prescalar.

CS12	CS11	CS10	
0	0	0	no clock (timer stopped)
0	0	1	$Clk_{I/O}$/1 (no prescaling)
0	1	0	$Clk_{I/O}$/8 (from prescalar)
0	1	1	$Clk_{I/O}$/64 (from prescalar)
1	0	0	$Clk_{I/O}$/256 (from prescalar)

1	0	1	Clk$_{I/O}$/1024 (from prescalar)
1	1	0	external clock source
1	1	1	external clock source

The clock selected from above list would be used for timer1 functioning.

The main concept behind Fast PWM is explained below.



Generation of Fast PWM wave using Timer1

The timer keeps counting from 0 to a TOP value (Count Max) provided by the program. For 16 bit timer1, its maximum value can be 65536.

The program also provides two compare values, compare 1 and 2, lower than the TOP value, for the two channels of timer 1, A and B respectively.

Now in one cycle of counting, i.e. from 0 to TOP, the controller gives a high voltage in channel 1 till the counter value becomes equal to compare 1, after which it gives 0v for the rest part of the cycle and

again repeats the same in the next cycle. Hence, pulse
width modulation is achieved.

The output voltage is proportional to the pulse width, which in turn is proportional to the ratio of compare and TOP values.

Hence,

$$\text{Output Voltage} = \frac{\text{compare value}}{\text{TOP value}} \times$$

The maximum voltage in the case of a microcontroller is 5v. It can be set on a scale of higher voltage, like 12v, by using proper batteries and driver circuits.

The registers that are being used here for storing the TOP and compare values are,

ICR1 (Input Capture Register 1) – it will store the TOP value. Its maximum limit is 2^{16} (65536).

OCR1A (Output Compare Register 1A) – it will store the compare value for channel A (output voltage at PB5).

OCR1B (Output Compare Register 1B) – it will store the compare value for channel B (output voltage at PB6).

BASIC CODE FOR FAST PWM GENERATION

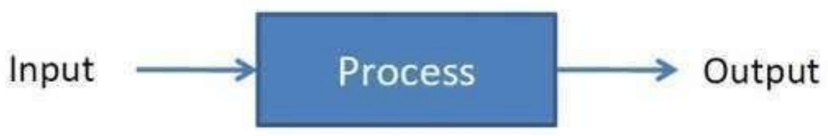
```
#define F_CPU 8000000UL

#include <avr/io.h>

void pwm_init();

void main()
{
        DDRB |= 0xFF;
        pwm_init();
        while (1)
        {
                OCR1A = 800; //one-fifth of maximum output
                OCR1B = 2000; //half of maximum output
        }
}

void pwm_init()
{
        TCCR1A |= (1<<COM1A1) | (1<<COM1B1) | (1<<WGM11);
        TCCR1B |= (1<<WGM12) | (1<<WGM13) | (1<<CS10);                //no prescaling
        ICR1 = 4000;       //top value for maximum output
}
```

In above code, ICR1 is 4000 which sets the upper limit. OCR1A is 800, so one-fifth of maximum output through channel A (PB5) and OCR1B is 2000, so half of the maximum output through channel B (PB6). Hence Fast PWM output is achieved.

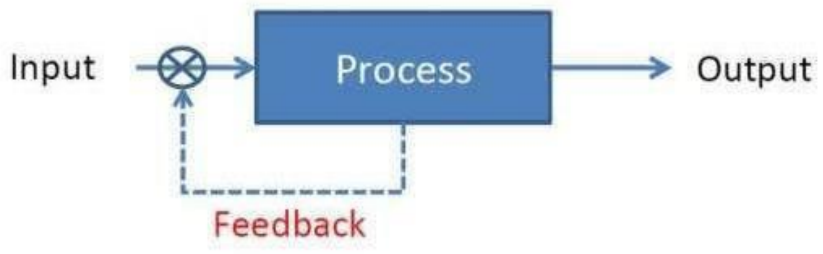
CONTROL SYSTEMS BASICS

When a number of elements are combined together to form a system which receives some input and produces desired output then the system is referred as control system. There are two types of control systems:-

1. **Open loop control systems** (non-feedback controller): In this system, the control action from the controller is independent of the process output, which is the process variable (PV) that is being controlled. A common example is electric drier; hot air comes out from it till a hand is present, irrespective of how much the hand has dried.



2. **Closed loop control systems** (feedback controller): In this system, the control action from the controller is dependent on the feedback from the process in the form of the value of the process variable (PV). A common example is that of an air conditioner, whose functioning depends upon the temperature of the room.

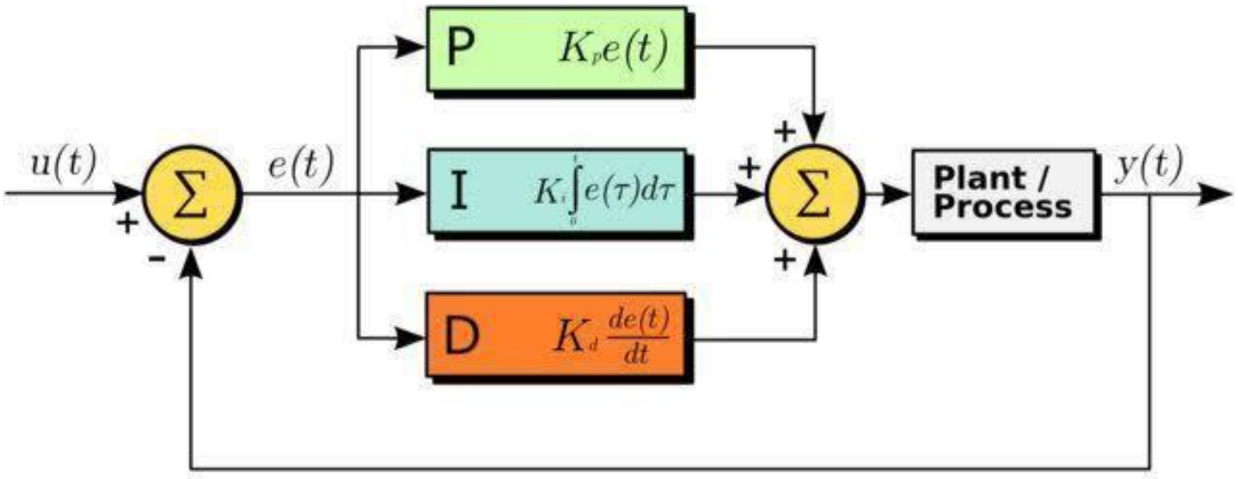


PROPORTIONAL-INTEGRAL-DERIVATIVE CONTROLLER (PID)

A proportional-integral-derivative controller (PID controller) is a closed loop control mechanism used to achieve error reduction in various systems.

A PID controller takes the feedback input of the process variable (PV) to be controlled. The Set Point (SP) is the desired value of PV which is to be achieved. The controller calculates error value *e(t)*, as the difference between the set point and the process variable and applies correction.

Hence error value *e(t)* = SP – PV(t), where SP is set point and PV is the process variable.



The correction value *v(t)* which is to be added to the process variable to reduce the error is give<u>n by</u> the equation,

$( ) = K_p ( ) + K_i \int 0 \ ( ) \qquad + K_d ( )$

This correction term is added to the base value y₀ of output variable *y(t)* to reduce error. Hence,

$( ) = \qquad 0 + ( )$ is the output of the controller.

The way a PID controller corrects the error can be understood by considering all the three parameters separately. The controller tries to reduce the difference between set point and process variable, i.e. error value to zero.

Proportional control (P): The most basic idea is, larger the error, larger should be the correction i.e. the correction term is proportional to the error produced, positive or negative. It accounts for the present value of error.

Integral control (I): This accounts for the past values of error. This term increases corrective action not only in relation with error but also in relation with the time for which it has persisted. A pure I controller can bring a system to zero error, but it would be extremely slow. Since the integration of error over some time cannot change sign along with the error value, sometimes the I control prompts overshooting which has to be accounted for. The I control is of great help when the system has to work against some external physical force.

Derivative control (D): This control doesn't consider the error, but the rate of change of error, and tries to bring this rate to zero. This accounts for the possible future values of error. Sometimes the error might not be very large at the particular moment of time but may be having a significant rate of change with respect to time, if not corrected; it will cause a large error in the future. That is why a correction proportional to this rate is required. This is tackled by the derivative controller.

APPLICATION OF PID IN DISCRETE SYSTEMS

When applied to discrete time systems, the analog integration and differentiation can be replaced by summation and difference, hence using a PSD (Proportional-Sum-Difference) controller.

The correction term can be rewritten for discrete systems as,

$$() = \quad p \; () + \quad {}_{i}\sum 0 \; ()\Delta \; + \quad {}_{d}\overline{{}^{\Delta\,(\,)}{}_{\Delta}}$$

The Δ term can be defined in the program as a small time interval for one cycle of computation, like 1 millisecond. Since this term is constant in the computation process it can be included with the constant values *K_p* and *K_d*, i.e. let $k_i = K_i\Delta$ and $k_d = K_d /\Delta$.

Hence, the correction term for a proportional-sum-difference (PSD) controller can be expressed as,

$$() = {}_{p}\,() + \quad {}_{i}\sum 0 \quad () + \quad {}_{d}\Delta\,()$$

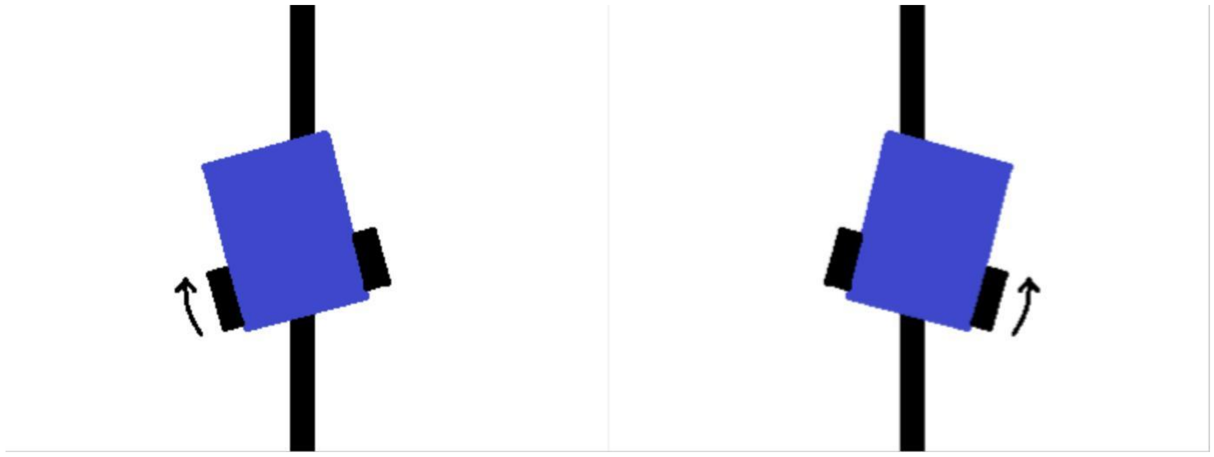
and the output *y(t)* of the controller, having base value y_o, will be expressed as,

$() = {}_{o} + \; ()$, which will result in control action.

APPLICATION OF PID CONTROLLER IN LINE FOLLOWER BOT

The goal to be achieved in the case of a line follower is that the centre of the robot should always be on the black line in white background. Deviation of the bot’s centre from the line is the error function *e(t)* in this case.

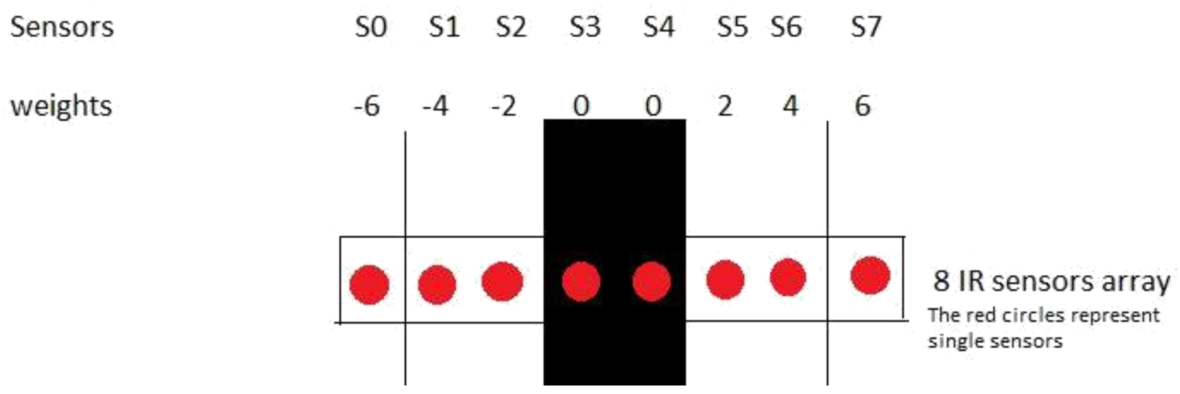
Here we will refer deviation to the left from the line as negative deviation and deviation to the right from the line as positive deviation.



negative deviation (left)   positive deviation (right)

The distance between two IR sensors should be so adjusted that only two IR sensors are there on the line at a point of time. For example, if the width if the line is 3 cm, then distance between two IR sensors should be 1.5 cm.

To calculate the deviation value, we are going to mathematically model the situation by giving different weightings to all the sensors. Here we are using 8 IR sensors, so starting from left; we will give weights like -6, -4, -2, 0, 0, 2, 4, and 6. The sum of the binary value returned by each sensor multiplied by its respective weight will give us the deviation value.



So if we use an array to store sensor values, like s[i], i being the index number varying from 0 to 7, we can store binary values for each number, 0 for white and 1 for black.

So deviation value,

*Deviation = (-6)*s[0] + (-4)*s[1] + (-2)*s[2] + (0)*s[3] + (0)*s[4] + (2)*s[5] + (4)*s[6] + (6)*s[7]*

In zero error condition, when bot is exactly on the line, the array s will be,

S = {1, 1, 1, 0, 0, 1, 1, 1},

So, deviation = (-6)*1 + (-4)*1 + (-2)*1 + (0)*0 + (0)*0 + (2)*1 + (4)*1 + (6)*1 = 0 (zero error deviation is zero).

The zero error deviation i.e. zero is our set point. The process variable is the current deviation at any point of time. Their difference is our error value.

Error value = desired deviation – current deviation = 0 – current deviation

Hence error value is positive for left deviation and negative for right deviation.

Also the position value can be calculated using the array s,

Position value K = s[0](128) + s[1]*(64) + s[2]*(32) + s[3]*(16) + s[4]*(8) + s[5]*(4) + s[6]*(2) + s[7]*(1)*

This position value is just the decimal conversion of the binary value returned by the sensors, for example, 231 for 11100111.

PSEUDO CODE FOR PID CONTROL

Here is a simple format for programming PID controller.

```
Start:
desired_deviation = 0
current_deviation = sensor_input
error = desired_deviation –
current_deviation error_sum = error_sum +
error error_difference = error – old_error
pid_output = $K_p$*(error) + $K_i$*(erro_sum) + $K_d$*(error_difference)
old_error = error
goto Start
```

For tuning the values of K_p, K_i, and K_d experimentally, to achieve a stable line following, the following points should be considered,

Starting with K_p, K_i and K_d, all equal to zero, we shall first tune K_p. We can start the with keeping $K_p = 1$, if the bot is slow and cannot turn quickly, K_p should be increased in gradual steps or else if the bot overshoots, the value should be decreased. The goal here is that the bot is somewhat able to follow the line. Since K_p is not enough to make the bot stable, there would be some wobbling in its motion. Leaving K_i for later, next we should attempt to tune K_d. K_d should be increased in smaller values of 0.25 or 0.5. The value should be increased till the amount of wobbling is reduced. Since derivative control is

about predicting future motion, tuned value of K_d will help reduce wobbling of the bot.
Now when a fairly stable line following is achieved with particular values of K_p and K_d, we should assign K_i a value between 0.25 and 1. Higher values of K_i cause overshooting and jerking and very low values give unperceivable results. So K_i needs to be tuned very precisely in order to achieve best results.
The values of the constants also mutually dependent to give better results, hence some experimentation with other combinations should also be done. These values also depend on the speed of the motors used, so they are to be tuned accordingly.

SAMPLE LINE FOLLOWING CODE USING PWM AND PID CONTROL

```
//Line following code
//black line on white surface

#define F_CPU 8000000UL
#define max           500
#define e_max       1000

#include <avr/io.h>
#include <avr/sfr_defs.h>
#include <util/delay.h>
#include <string.h>
#include <stdbool.h>

int sensor_weight[] = {-6,-4,-2,0,0,2,4,6};     //array for sensor weightings
int error, des_dev, cur_dev;      //variables for error, set-point and process variable
int e_sum=0, e_diff=0, e_old=0, pid_out;

int i, j, m, n, k;                  //variables for computing purposes

int s[8], h[8];                     //arrays for storing sensor input

int c=0, x=0, mode=0;

float kp = 15, ki = 0.25, kd = 4;               //PID constants

int speed = 500;                   //base speed for motor PWM

int readSensor();                    //for taking sensor input
void pwm_init();                     //setting registers for Fast PWM initialisation
float err_calc();                      //for calculating error value
float pid(float);                      //returns PID correction output
void travelPath();                   //for line following functions
void motorPIDcontrol(int);       //motor speed control using PID
void runExtraInch();               //for running a few steps ahead for alignment
void goAndTurn(char);             //for turning of bot
void motor_stop();                 //stops the motors
void main(void)
{
        DDRD = 0X00;            //port D is used as input port
        DDRB |= 0XFF;           //port B is used for output PB5 for left PB6 for right
        DDRA |= 0XFF;           //port A for motor direction PA1 for left PA2 for right

        pwm_init();
        while(1)
        {
                travelPath();
                _delay_ms(10);
        }
}

void pwm_init()
{
        TCCR1A |= (1<<COM1A1) | (1<<COM1B1) | (1<<WGM11);
        TCCR1B |= (1<<WGM12) | (1<<WGM13) | (1<<CS10);
        ICR1 = 4000;            //TOP value of 4000 for maximum rpm of motors
}

int readSensor()
{
        s[8] = 0; k=0;
        for(i=0;i<8;i++) {
                if(bit_is_set(
                PIND,i))
                {
                        s[i] = 1;
                }
                else
                {
                        s[i] = 0;
                }
        }
        k = s[0]*(128) + s[1]*(64) + s[2]*(32) + s[3]*(16) + s[4]*(8)
           + s[5]*(4) + s[6]*(2) + s[7]*(1);    //s[0] is leftmost and s[7] is rightmost

        return(k);
```

```
}

float err_calc()
{
        cur_dev=0;
        des_dev=0;
        h[8] = 0;
        m=0;

        for(i=1;i<7;i++)
        {
                if(bit_is_set(PIND,i))
                {
                        h[i] = 1;
                }
                else
                {
                        h[i] = 0;
                        m++;
                }
                cur_dev += h[i] * sensor_weight[i];  //cur_dev stores deviation value
        }                                  //deviation negative for left and positive for right

        if(m==0)
        {m=1;}

        cur_dev = (cur_dev / m) * 4000; //deviation value set on the scale of
                                          //motor rpm value for PID output
        error =des_dev - cur_dev;//error negative for right positive for left deviation
        return error;
}


float pid(float err)
{
        e_sum += err;
        e_diff = err - e_old;

        if (e_sum > e_max)          //condition for keeping the integral of error over a
                                    //period of time within a limit
        {
                e_sum = e_max;
        }
        else if (e_sum < -e_max)
        {
                e_sum = -e_max;
        }

        pid_out = (kp * err) + (ki * e_sum) + (kd *
        e_diff); e_old = err;

        if(pid_out > max)               //condition for keeping the pid output within
                                         the //limits of the base value of motor rpm
        {
                pid_out = max;
        }
        else if (pid_out < -max)
        {
                pid_out = -max;
        }
        return pid_out;
}

void motorPIDcontrol(int base)      //function for giving PID output to the motors
{
        OCR1A = base + (pid(err_calc()));
        OCR1B = base - (pid(err_calc()));
        PORTA |= ((1<<PINA1) | (1<<PINA2)); //both direction pins PA1 and PA2
                                             //are kept high for forward motion
}

void runExtraInch()
{
        OCR1A = 500;
        OCR1B = 500;
        _delay_ms(300);
        motor_stop();
}
```

```
void motor_stop()
{
        OCR1A = 0;
        OCR1B = 0;
        _delay_ms(500);
}

void travelPath()
{
        mode = readSensor();
        PORTA |= ((1<<PINA1) | (1<<PINA2));
        switch(mode)
        {
                case 255:                      //11111111 all white- end of line so U-turn
                        _delay_ms(300);
                        motor_stop();
                        PORTA &= ~(1<<PINA2); //right motor set for reverse rotation
                                              //for U-turning
                        OCR1A = 800;
                        OCR1B = 800;
                        _delay_ms(1800);
                        break;

                case 248:                      //11111000 cases for 90 degree right turn
                case 240:                      //11110000
                case 224:                      //11100000
                        motor_stop();
                        runExtraInch();
                        OCR1A = 800;
                        OCR1B = 0;
                        _delay_ms(1100);
                        break;

                case 31:                       //00011111 cases for 90 degree left turn
                case 15:                       //00001111
                case 7:                        //00000111
                        motor_stop();
                        runExtraInch();
                        OCR1A = 0;
                        OCR1B = 800;
```

```
                    _delay_ms(1100);
                    break;

                case 243:       //11110011 cases for straight and curved path following
                case 231:       //11100111
                case 227:       //11100011
                case 207:        //11001111
                case 199:       //11000111

                    motorPIDcontrol(speed);
                    break;

                default: //for any other case straight and curved path following
                    motorPIDcontrol(speed);
        }
}
```

In this code, when the bot is on the straight or smoothly curved path, the motion is controlled by PID. Hence, any error in traversing is removed for stable motion.

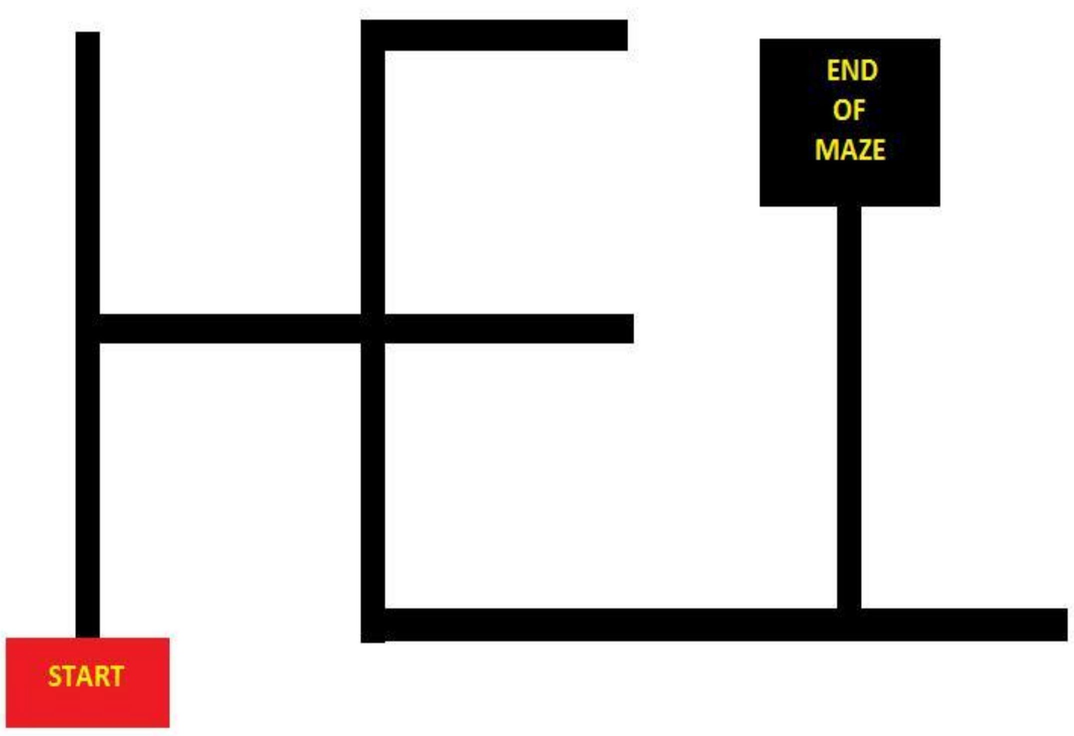
Special cases are coded for sharp left or right 90 degree turn for better precision. For acute angles, PID controller can do the correction and keep the bot on the trajectory.

The values given to the _delay_ms() function need to be tuned experimentally for different cases. Higher the rpm of motors used, lower will be the delay value for a turn.

The PID constants in the above code need to be tuned precisely by many repeated experiments. The logic developed here gives the bot an ability to traverse almost all trajectories, straight, curved or rectangular, with great ease, the key here being proper values for PID constants.

MAZE SOLVING WITH LINE FOLLOWER

A line follower bot can traverse along a line in a contrasting background. When there is a network or maze of lines, say black lines on a white surface, and the bot has to reach from a starting point to an end point, it needs programming logic to solve a multiple path maze. An example maze is given in the figure below.



Left-hand algorithm is one of many techniques that can be applied to solve a maze of this kind. In this algorithm, at every node in the maze, the bot will follow a priority rule for deciding between available path options. On a node, it follows the priority – left, straight, right – if a left is available, it will go left. If there is no left, but a straight path is there, it will go straight. If both left and straight are absent and right is the only option, it will go right.

Maze solving consists of two consecutive runs:-

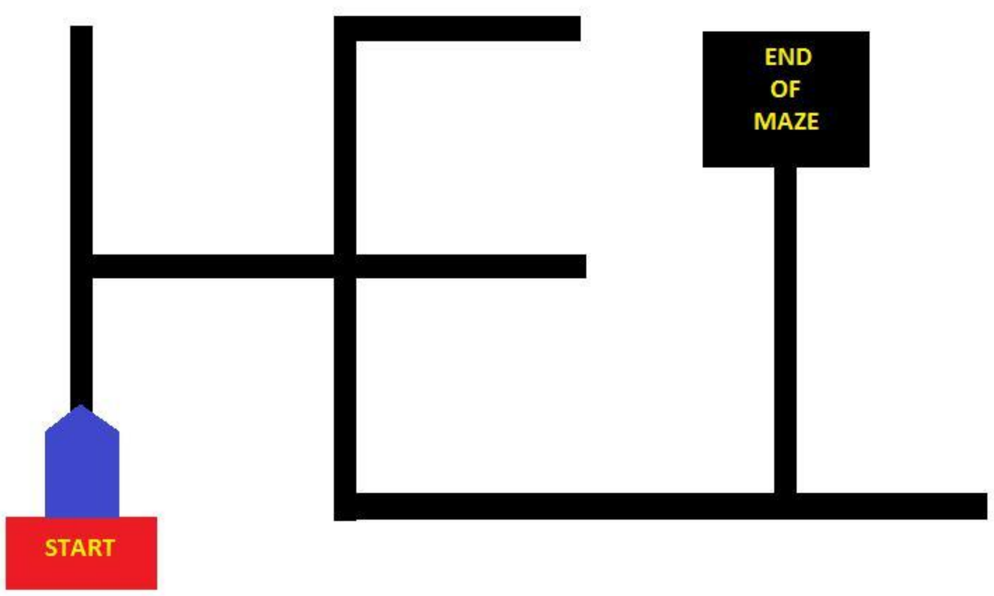
1. Dry run – in this the bot traverses in all possible ways and analyses the path using solving techniques like the left-hand algorithm. The bot keeps in its memory the data regarding turning at different nodes. This data is used in the program to find the shortest path.
2. Actual run – in this run the bot traverses the shortest path between the starting and ending points. The proper turning at each node is provided by the program, based on the data obtained from the dry run.

For storing the data regarding turning at different nodes obtained during the dry run, a character array path[] is used and the three decisions at a node – left, straight, right – are designated by characters 'L', 'S' and 'R' respectively. Since the bot also needs to take a U-turn at a dead end, we will designate it with the character 'U'. The array path[] is of sufficient length so that it can save the dry run data.

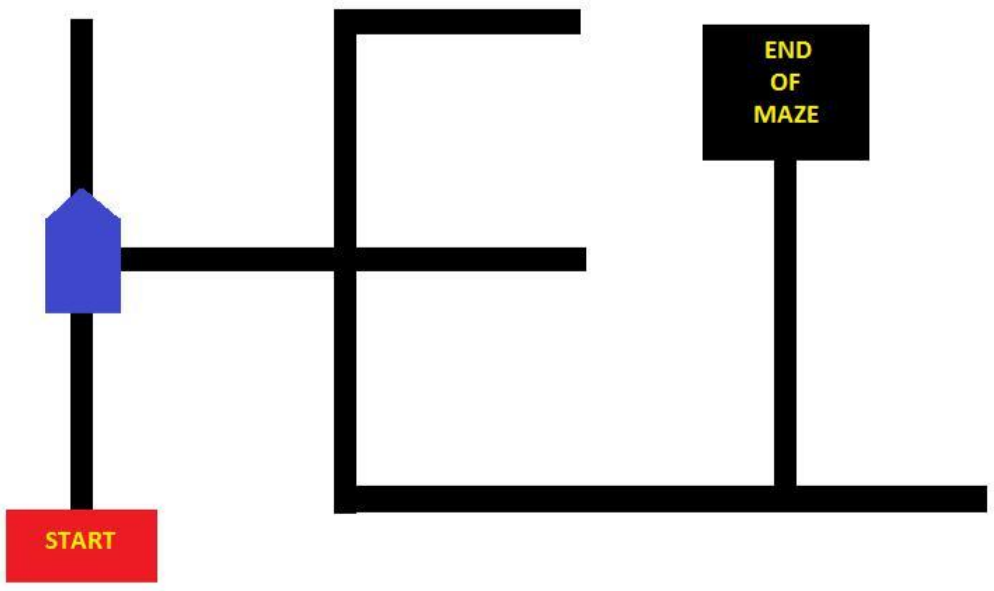
Later we will use a path reduction program to find out the shortest path from this data obtained during the dry run.

Now first we will discuss the dry run on the given maze and how we will deduce proper turning at a node from the data obtained in terms of characters. Later we can easily see how this can give us the shortest path in the maze.

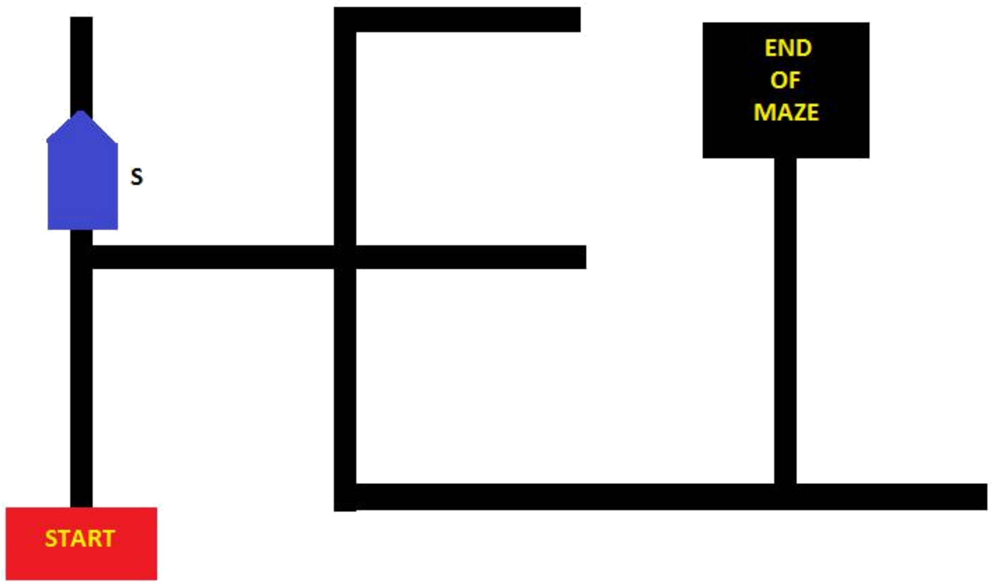
DRY-RUN (LEFT HAND ALGORITHM)



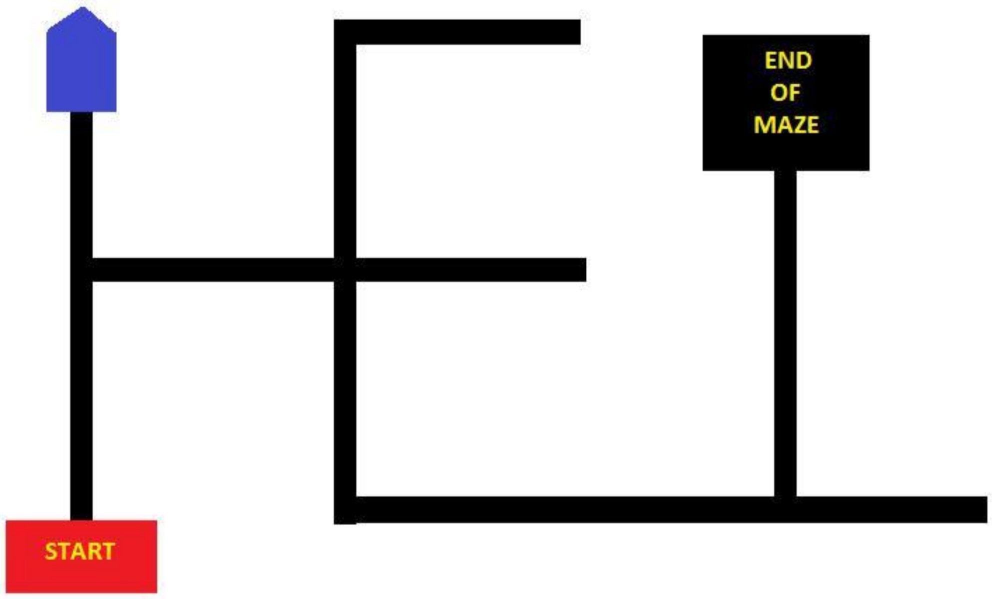
The bot will start from the starting point and will reach the first node.



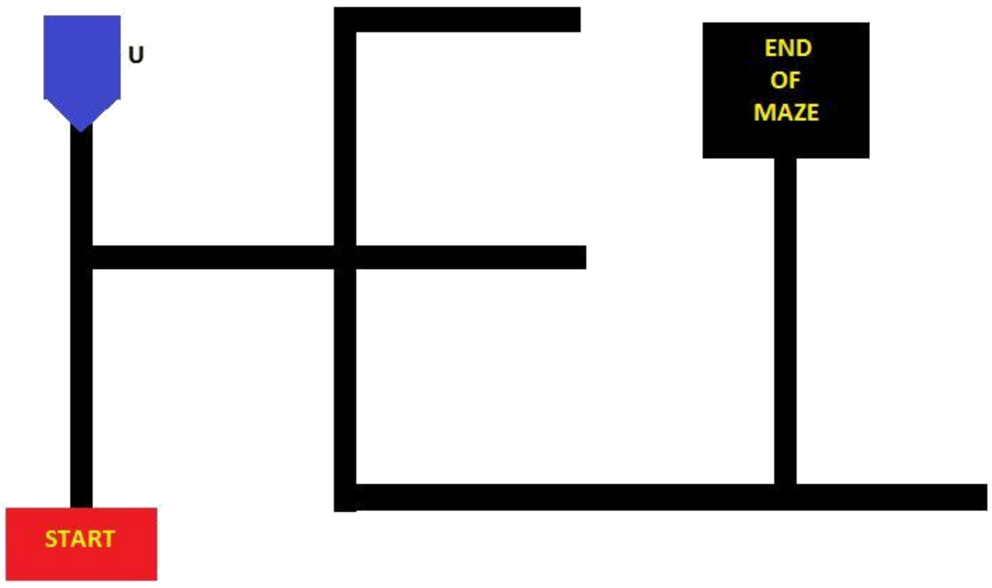
Here the bot has two options – straight or right.



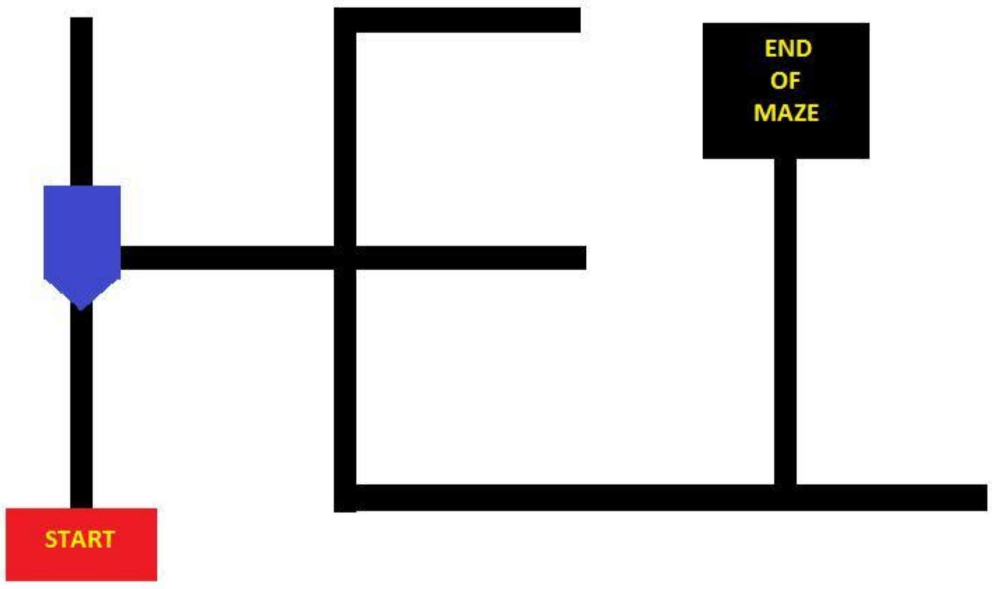
Based on priority, the bot will go straight and a character ‘S’ will be saved in the path array.



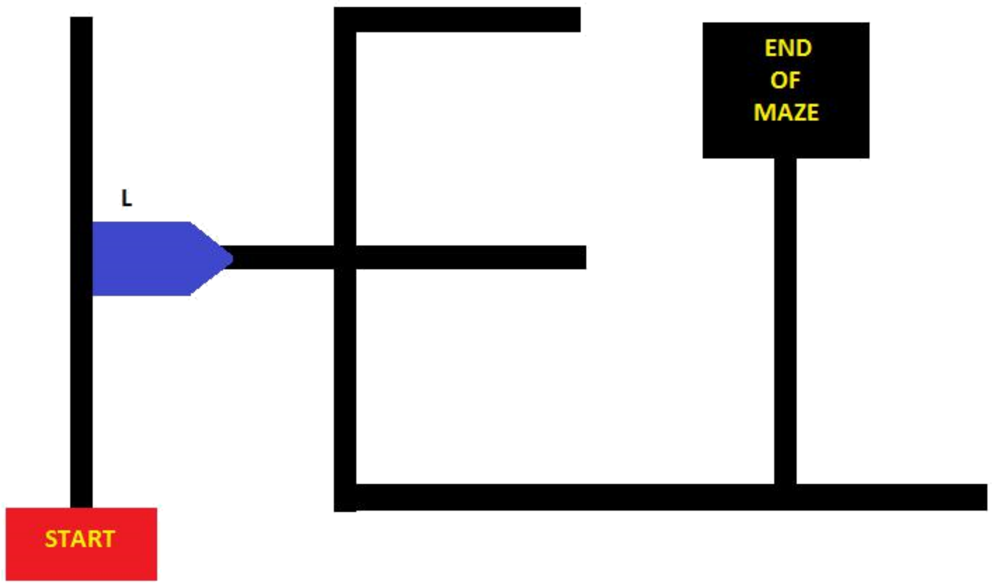
Further the bot meets a dead end and the only option here is a U-turn.



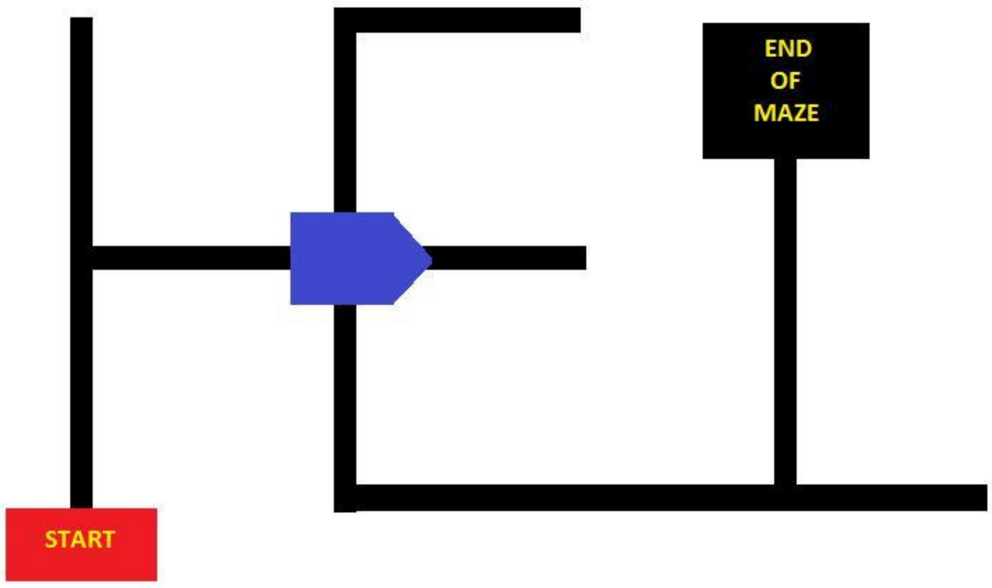
The bot makes a U-turn and saves the character ‘U’. Now path value is SU.



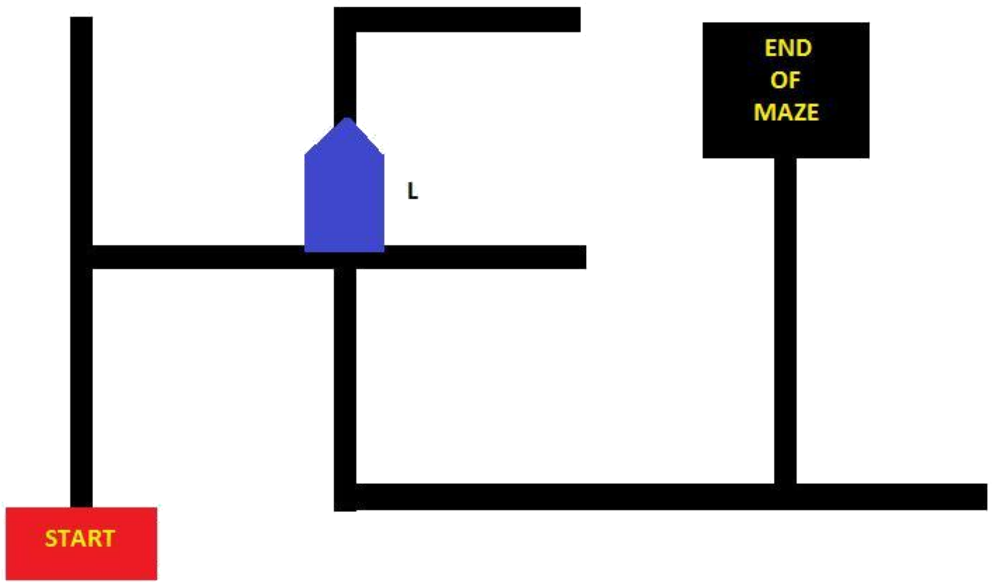
The bot reaches the previous node once again and this time has two options – straight or left.



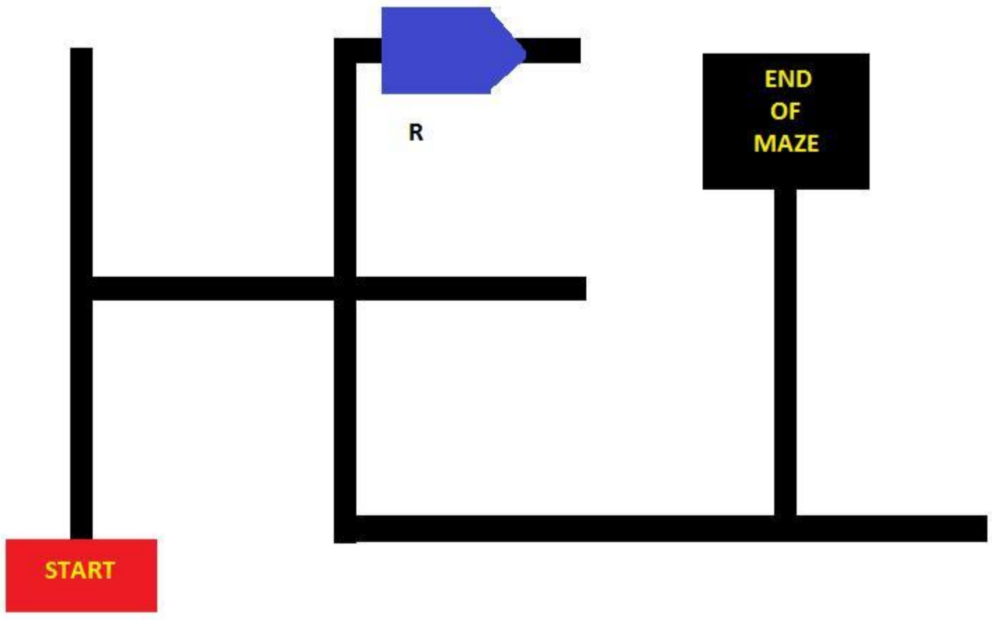
The bot chooses left on priority basis and saves character 'L'. Now path value is SUL, it can be seen that these steps can be replaced by a right turn so we will replace SUL by a 'R'.



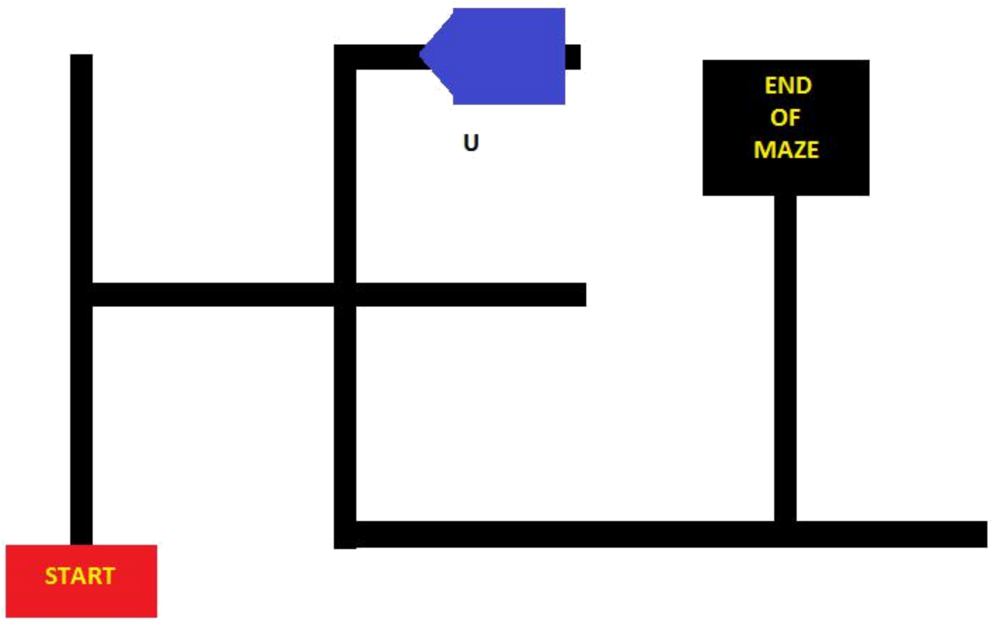
Now the bot reaches next node, here it has three options – left, straight or right.



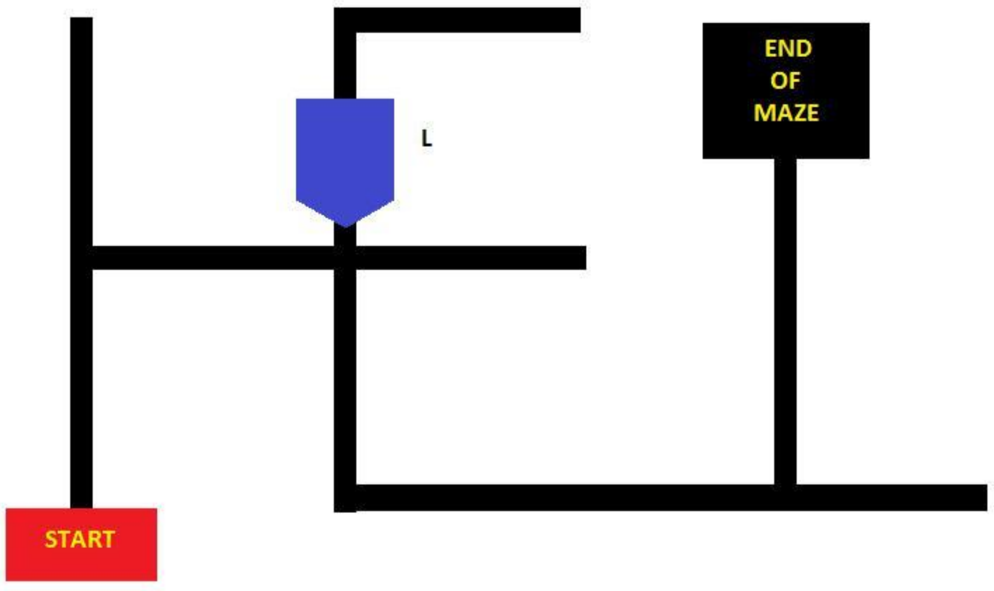
The bot takes left on priority basis. It saves the character ‘L’ (path R L).



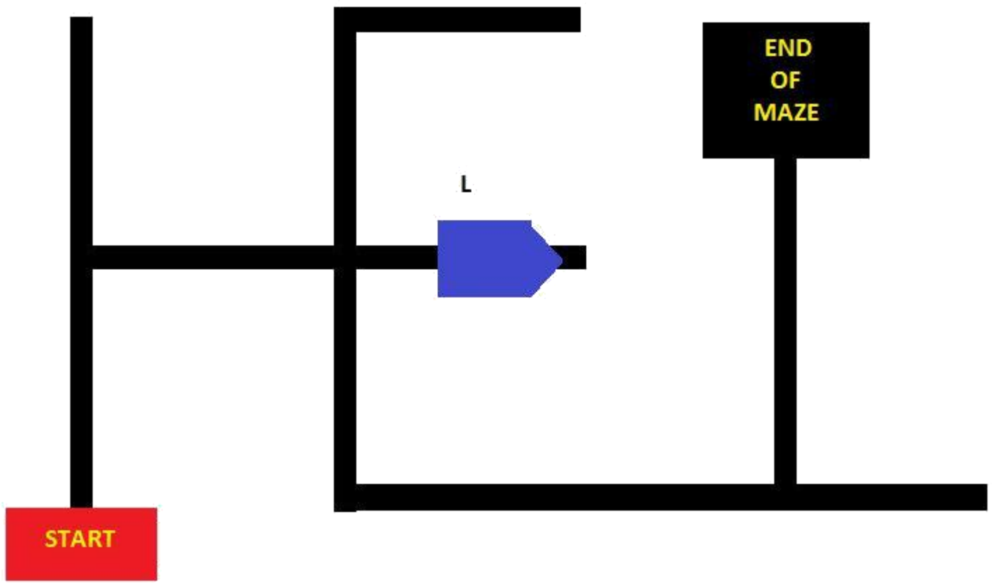
At next node only a right turn is there, so a value ‘R’ is saved (path R LR).



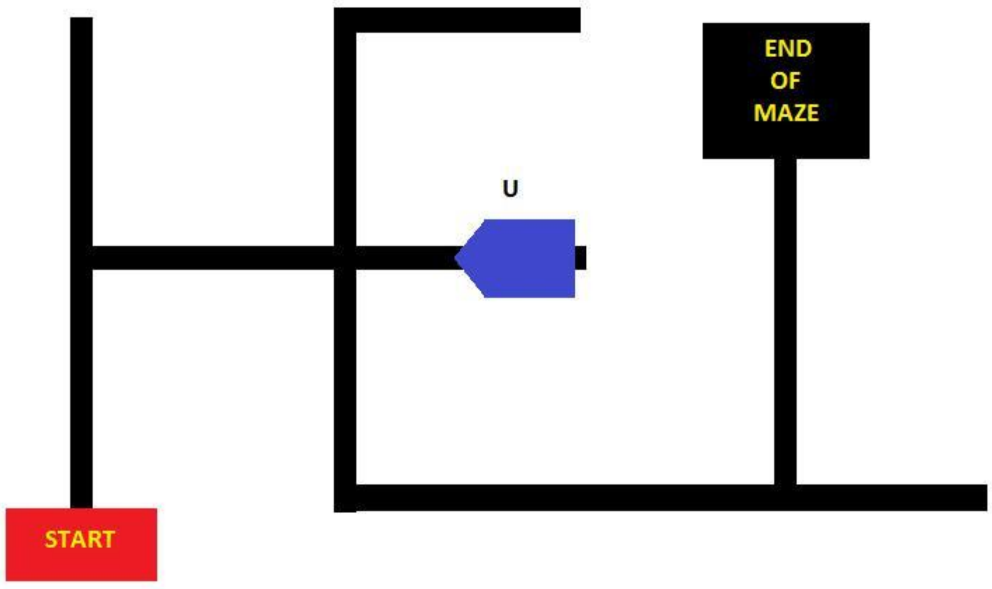
Since the bot meets a dead end, it takes U-turn and saves the character ‘U’ (path R LRU).



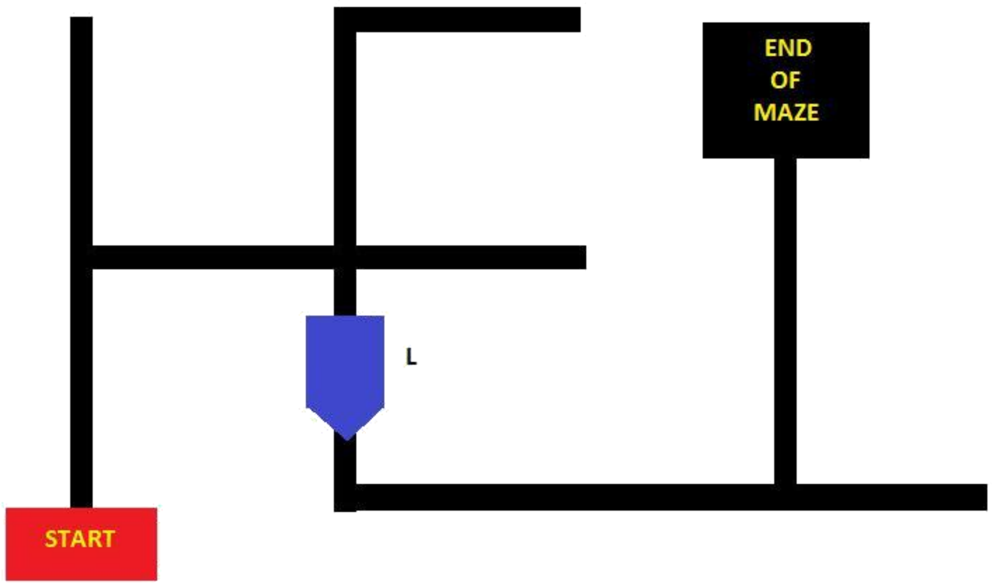
The bot takes a left turn and saves the character ‘L’ (path R LRUL).



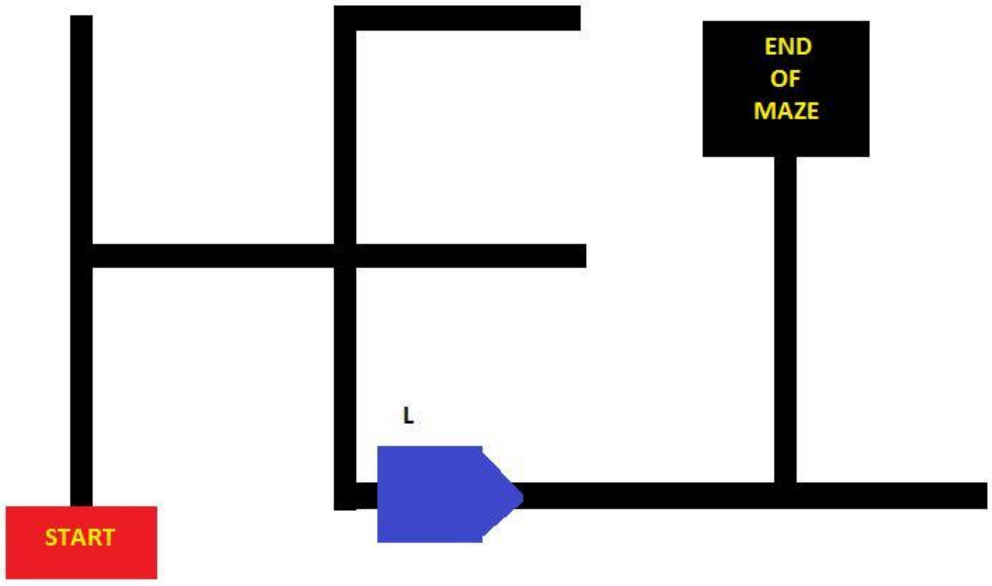
The bot again takes a left turn as per the priority and saves ‘L’ (path R LRULL). In the path value the sequence LRULL can simply be replaced by character ‘S’.
So path is R S.



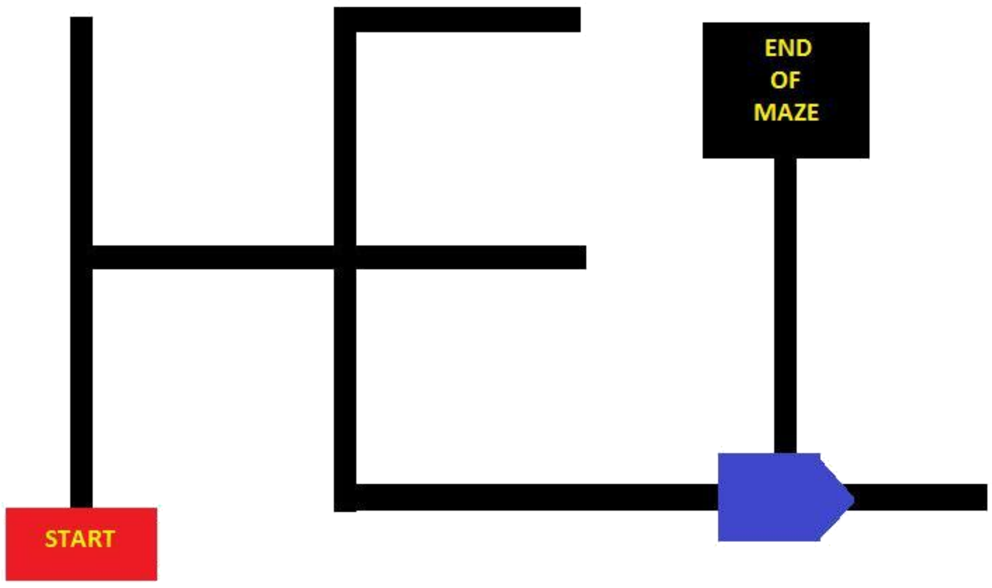
The bot meets a dead end and saves the character ‘U’. (path R SU)



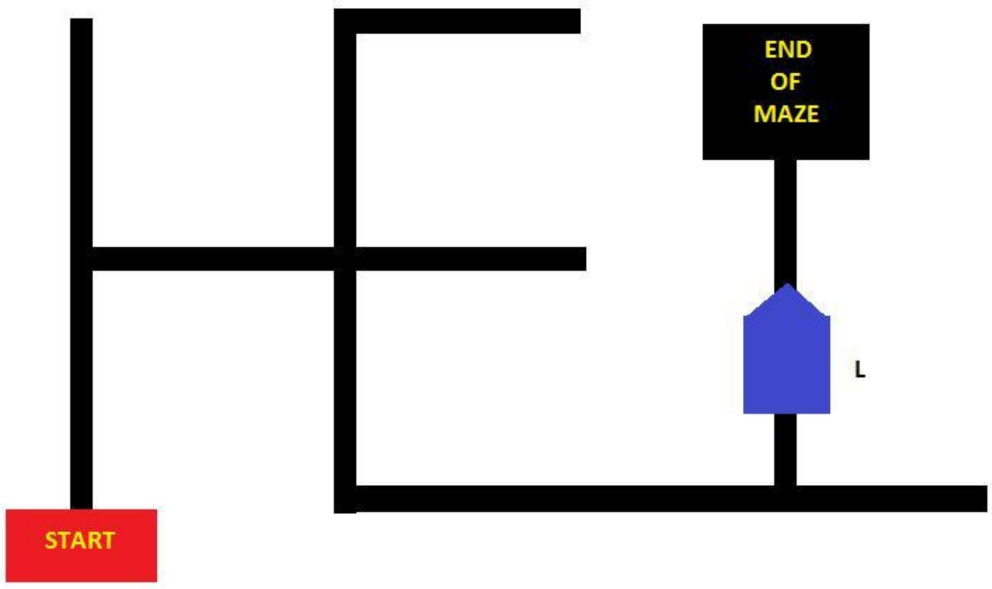
Again based on priority rule, the bot takes a left turn on the four-way. The path here being R SUL. As discussed before, the sequence SUL can be replaced by a ‘R’. So now the path is R R.



At next node only one option is there, a left turn. So the bot turns left and stores value ‘L’. (path R R L)



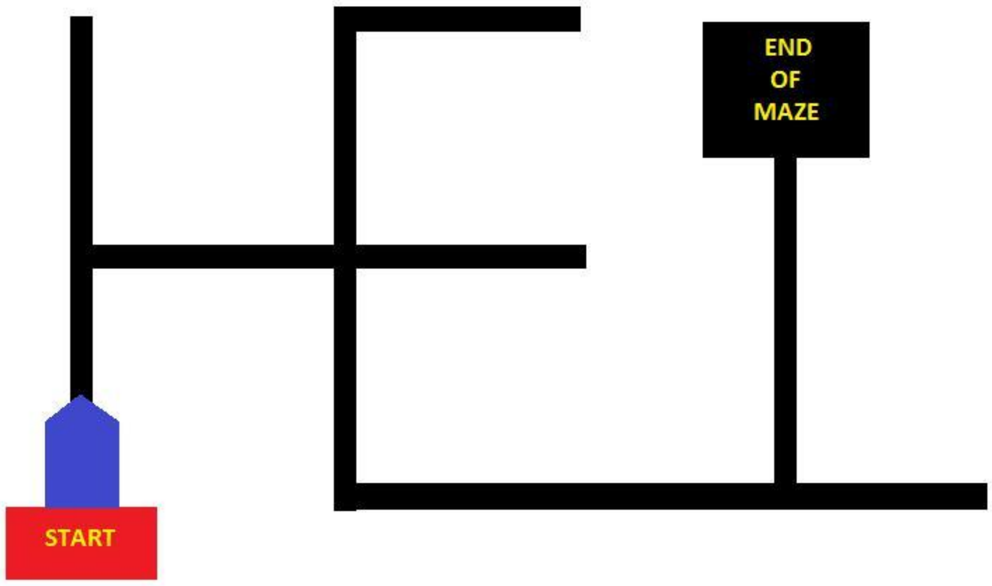
At this node there are again two options, left or straight.



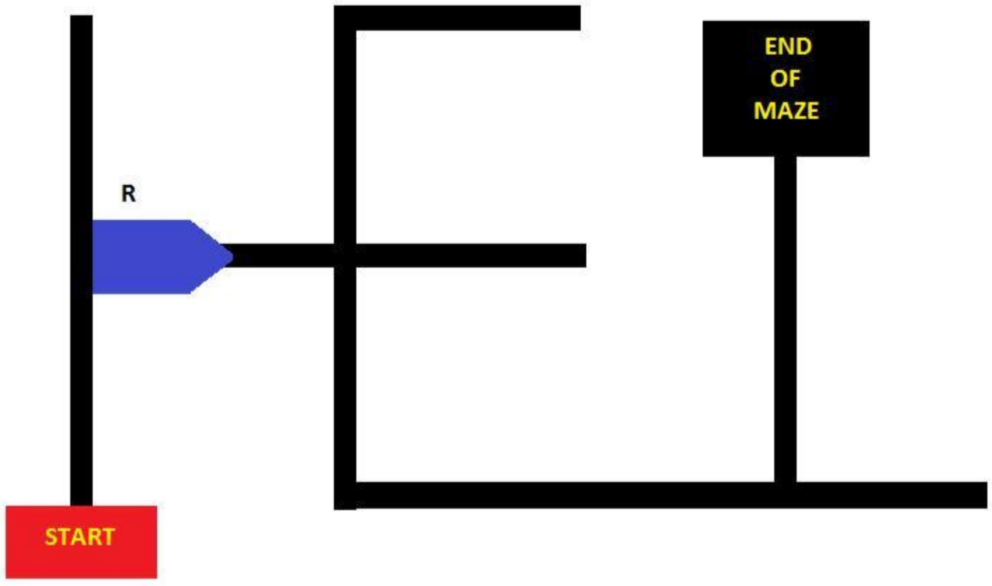
Because of the priority rule, the bot turns left and stores ‘L’. The path value is R R L L.

ACTUAL RUN (SHORTEST PATH)

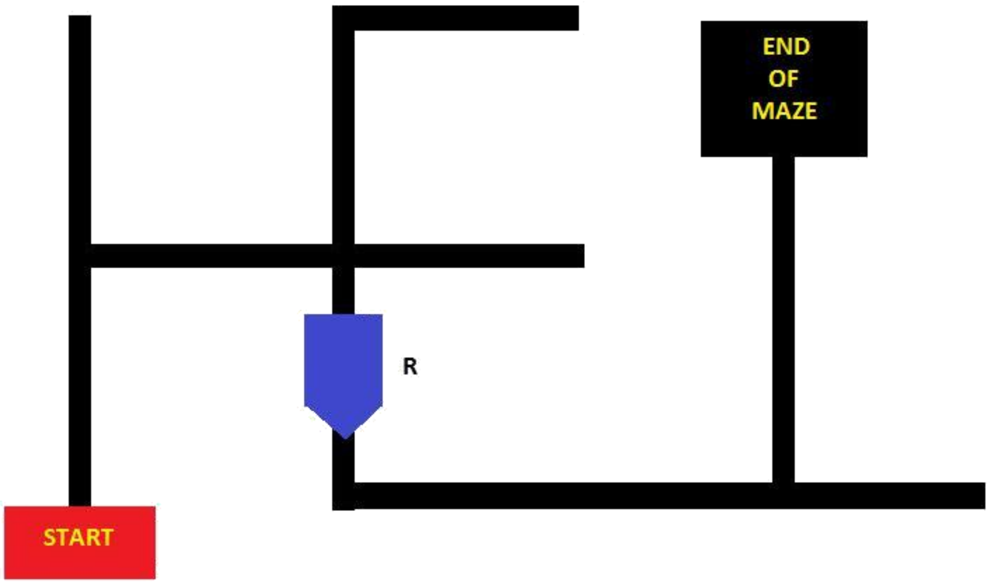
Now we have obtained the shortest path from the dry run data, i.e. R R L L.



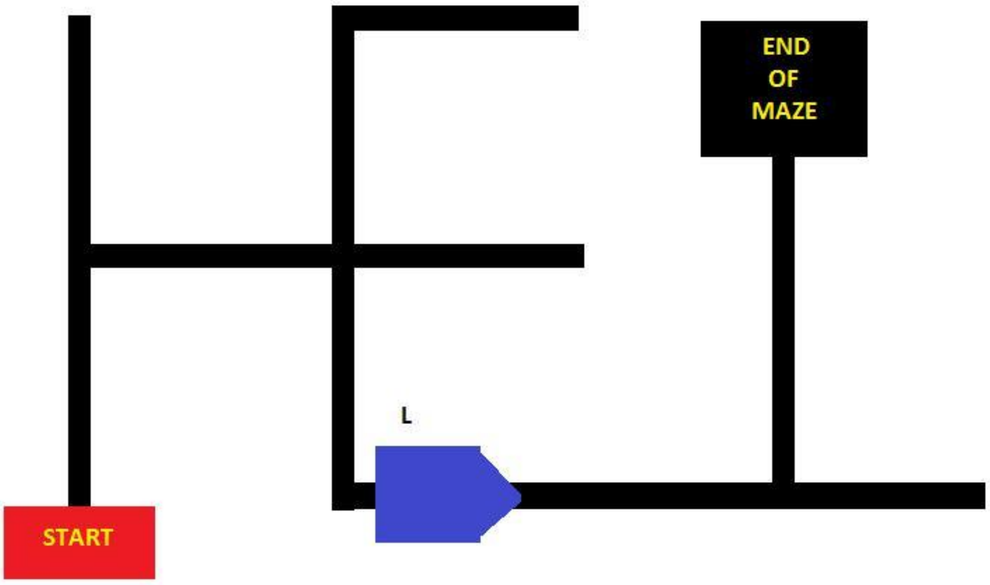
So if the bot starts the actual run with this data.



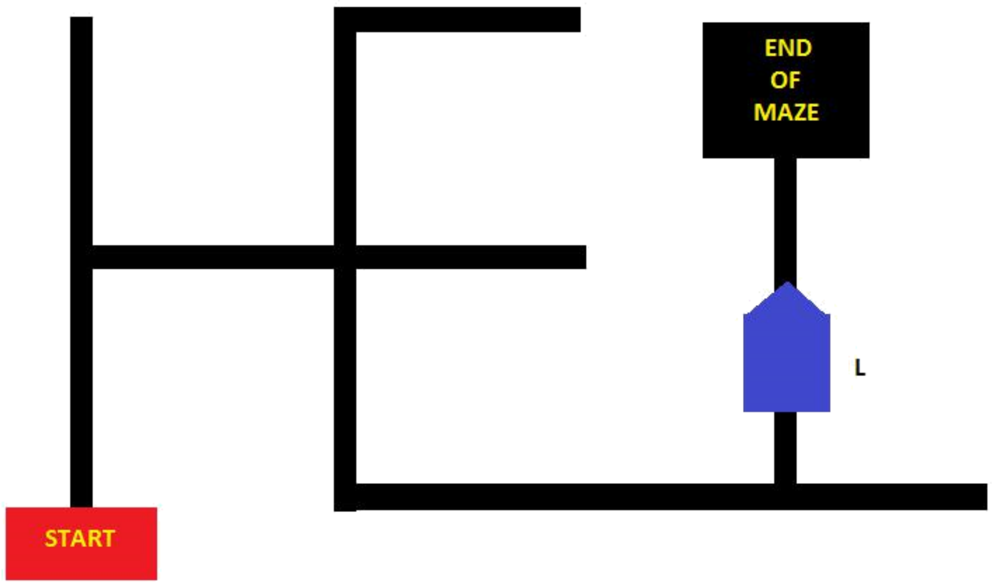
It takes right turn on the first node (**R** R L L).



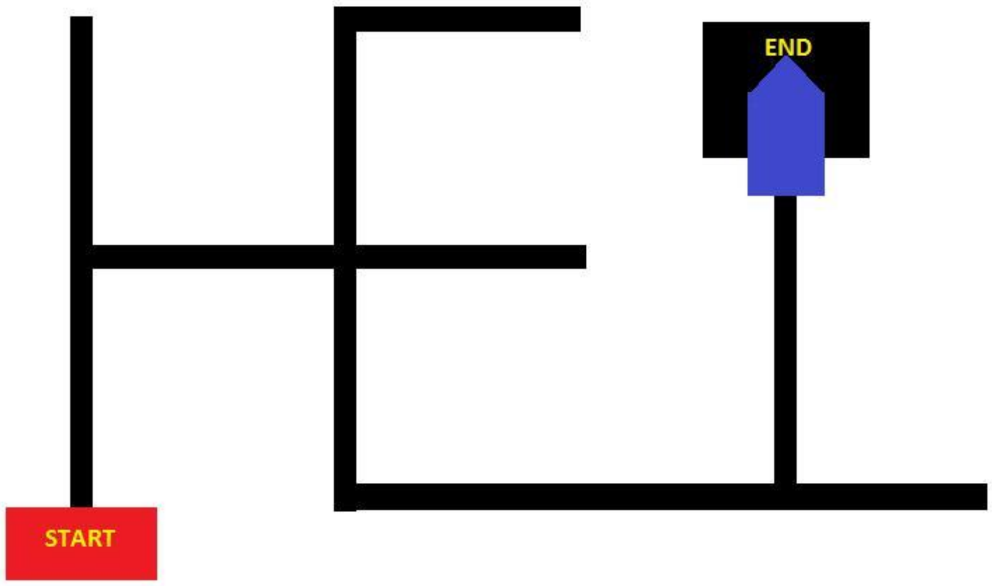
It takes right turn on the second node (R **R** L L).



At next node, it takes a left (R R **L** L).



And also a left turn on the last node (R R L **L**).



And hence it completes the actual run along the shortest path.

PATH REDUCTION PROGRAM

Here we will discuss a code in C based on the logic developed above in the left hand algorithm.

```
#include <stdio.h>
#include <string.h>

char path[]= {'S','U','L','L','R','U','L','L','U','L','L','L'}; //dry run data of the maze
int i, j;

int arrShift(int,int);          //shifts the data in array path to maintain order
void simplifyPath();            //replaces a sequence of characters by a proper character

void main()
{
        printf("Given Path \n");
        for(j=0;j<strlen(path);j++)     //to print the original dry run data
        {
                printf("%c",path[j]);
        }
        printf("\n");

        simplifyPath();

        printf("Reduced Path \n");
        for(j=0;j<strlen(path);j++)     //to print the reduced path data
        {
                printf("%c",path[j]);
        }

}
```

```
int arrShift(int i ,int len)
{
        for(i=i ; i<len-2 ; i++)
        {
                path[i] = path[i+2];
        }
        len = len-2;
        return len;
}

void simplifyPath()
{
        int len = strlen(path);
        int check = 0;

        start :
        for(i=0 ; i<len-2 ; i++)
        {
                if(path[i+1] == 'U')    //the logic is based on the presence of ‘U’
                {
                        switch(path[i])
                        {
                                case 'L':
                                        switch(path[i+2])
                                        {
                                                case 'L':
                                                        path[i+2] = 'S';
                                                        check = 1;
                                                        break;

                                                case 'S':
                                                        path[i+2] = 'R';
                                                        check = 1;
                                                        break;

                                                case 'R':
                                                        path[i+2] = 'U';
                                                        check = 1;
                                                        break;
                                        }
}

                    break;

                case 'S':
                    switch(path[i+2])
                    {
                        case 'L':
                            path[i+2] = 'R';
                            check = 1;
                            break;

                        case 'S':
                            path[i+2] = 'U';
                            check = 1;
                            break;
                    }
                    break;

                case 'R':
                    switch(path[i+2])
                    {
                        case 'L':
                            path[i+2] = 'U';
                            check = 1;
                            break;
                    }
                    break;
            }

            if(check == 1)
            {
                len = arrShift(i,len);
                check = 0;
                goto start;
            }
        }
    }

}
```

Let us discuss the logic of the code for path reduction.

When the simplifyPath() function is called in the main() function, it runs a loop from 0 to (length of path – 2). When a ‘U’ is located at (i+1)th position, the code checks the values at ith and (i+2)th positions.

On the basis of these values, a sequence of 3 letters is replaced by a single letter, which is stored at the (i+2)th position. Then we have coded a function arrShift(), which shifts the values from (i+2)th position to the ith position.

This process is continued till every ‘U’ in the input sequence is removed. The final result stored in the path[] array is the reduced path.

The combinations of characters and their replacements are listed below.

LUS -> R	SUL -> R
LUR -> U	RUL -> U
RUS -> L	SUR -> L
LUL -> S	RUR -> S

Now let’s apply these replacement rules in our input data.
Given combination is SULLRULLULLL.

Now when the first U is located in S**U**LLRULLULLL, SUL will be replaced by R. So the resultant combination is RLRULLULLL.

In RLR**U**LLULLL, the characters RUL will be replaced by U. Hence the resultant is RLULULLL.

Further in RL**U**LULLL, LUL will be replaced by S. So resultant is RSULLL.

In RSULLL, SUL is replaced by R, giving RRLL.

So the final result of the above code, which doesn’t consist of any U-turn, is RRLL. The bot will follow this combination to travel the shortest path in the maze between the starting and ending points.

This is a very popular maze-solving technique and can be used to solve mazes of all designs in a very short period of time.

MAZE – SOLVER CODE APPLYING LEFT HAND ALGORITHM

```
//maze solving code
//black line on white surface

#define F_CPU 8000000UL
#define max          500
#define e_Max       1000


#include <avr/io.h>
#include <avr/sfr_defs.h>
#include <util/delay.h>
#include <string.h>
#include <stdbool.h>


int sensor_weight[8] = {-6,-4,-2,0,0,2,4,6};    //sensor weightings
int error, des_dev, cur_dev;                    //variables for deviation and error

int s[8], h[8];                                 //arrays for storing input sensor data

int i, m, j, n, k;
int c=0, mode=0, temp=0, x=0, stop=0;

//PID constants
float kp=15, kd = 4, ki = 0;

float E_sum = 0 , e_old = 0, pid_out;
int speed=500;                              //base value for rpm of motors //array
char path[100];                             for storing path data of dry run
int pathLength=0;


void pwm_init();                        //pwm initialisation
int readSensor();                       //taking sensor input
float err_calc();                       //calculating error value
float pid(float);                       //calculating pid correction
void motorPIDcontrol(int);              //PID control of motors
void mazeSolve();                       //dry run and actual run
void runExtraInch();                    //moving only few steps
void motor_stop();                      //stopping the bot
void mazeEnd(); void                    //indicating end of maze
save_path(char); int                    //saves dry run data
arrShift(int,int); void                 //used with simplifyPath()
simplifyPath();                         //finds shortest path from dry run data


int main(void)
{
        DDRD = 0X00;            //port D declared as input port
        DDRB |= 0XFF;           //port B declared as output port
        DDRA |= 0XFF;           //port A declared as output port

        pwm_init();             //pwm initialisation

        while(1)
        {
                mazeSolve();
                _delay_ms(10);
        }
}

void pwm_init()
{
        TCCR1A |= (1<<COM1A1) | (1<<COM1B1) | (1<<WGM11);
        TCCR1B |= (1<<WGM12) | (1<<WGM13) | (1<<CS10);
        ICR1 = 4000;
}

int readSensor()
{
        s[8] = 0;
        k=0;
        for(i=0;i<8;i++)
        {
                if(bit_is_set(PIND,i))
                {
                        s[i] = 1;
                }
```

```
            else
            {
                    s[i] = 0;
            }
        }
        k = s[7] + s[6]*2 + s[5]*4 + s[4]*8 + s[3]*16 + s[2]*32 + s[1]*64
              + s[0]*128;                       //s[0] is leftmost s[7] is rightmost
        return(k);
}

float err_calc()
{
        cur_dev=0;
        des_dev=0;
        h[8] = 0;
        m=0;

        for(i=1;i<7;i++)
        {
            if(bit_is_set(PIND,i))
            {
                    h[i] = 1;
            }
            else
            {
                    h[i] = 0;
                    m++;
            }
            cur_dev += h[i] * sensor_weight[i]; //cur_dev stores deviation value
        }                                    //deviation negative for left positive for right

        if(m==0)
        {m=1;}

        cur_dev = (cur_dev / m) * 4000; //deviation value set on the scale of
                                         //motor rpm value for PID output
        error =des_dev - cur_dev;//error negative for right positive for left deviation
        return error;
}
```

```
float pid(float err)
{
        e_sum += err;
        e_diff = err - e_old;

        if (e_sum > e_max)          //condition for keeping the integral of error over a
                                    //period of time within a limit
        {
                e_sum = e_max;
        }
        else if (e_sum < -e_max)
        {
                e_sum = -e_max;
        }

        pid_out = (kp * err) + (ki * e_sum) + (kd *
        e_diff); e_old = err;

        if(pid_out > max)            //condition for keeping the pid output within
                                     the //limits of the base value of motor rpm
        {
                pid_out = max;
        }
        else if (pid_out < -max)
        {
                pid_out = -max;
        }
        return pid_out;
}

void motorPIDcontrol(int base)       //function for giving PID output to the motors
{
        OCR1A = base + (pid(err_calc()));
        OCR1B = base - (pid(err_calc()));
        PORTA |= ((1<<PINA1) | (1<<PINA2)); //both direction pins PA1 and PA2
                                             //are kept high for forward motion
}



void mazeSolve()
```

```
{
        while(!x)                                   //condition for dry run
        {
                mode = readSensor();
                switch(mode)
                {
                        case 00:                    //00000000 all black
                                motor_stop();
                                runExtraInch();
                                temp = 0;
                                temp = readSensor();
                                if(temp == 0)     //all black in next step then end of maze
                                {
                                        mazeEnd();
                                }
                                else       //any case in next step after all black - left turn
                                {
                                        OCR1A = 0;
                                        OCR1B = 800;
                                        _delay_ms(1100);




                                        s       //11111111 all white dead end
                                        ave_path('L');     //value 'L' saved for left turn
                                }
                                break;
                        case 255:
                                motor_stop();
                                _delay_ms(100);
                                PORTA &= ~(1<<PINA2); //right motor
                                OCR1A = 800;                //set for reverse rotation
                                OCR1B = 800;
                                _delay_ms(1800);
                                save_path('U');             //value 'U' saved for U-turn
                                break;
                                                                        case 224:
                        case 248:
                        case 240:
```

//11110000

//11100000

//11111000 cases for a right turn or straight

```
motor_stop();
runExtraInch();
temp = 0;
```

```
            temp = readSensor();

            if(temp == 255)   //case for a right turn
            {
                OCR1A = 800;
                 OCR1B = 0;
                _delay_ms(1100);
                save_path('R');  //value ‘R’ saved for right turn
            }
            else                    //case for straight
            {
            motorPIDcontrol(speed);
            save_path('S');              //value ‘S’ saved for straight
            }
            break;

        case 31:                    //00011111 cases for left turn
        case 15:                    //00001111
        case 7:                     //00000111
            motor_stop();
            runExtraInch();
            OCR1A = 0;
            OCR1B = 800;
            _delay_ms(1100);
            save_path('L');       //value ‘L’ saved for left turn
            break;

        case 243:            //11110011 cases for straight and curved path
        case 231:            //11100111                             following
        case 227:            //11100011
        case 207:            //11001111
        case 199:            //11000111
            motorPIDcontrol(speed);
            break;

        default: //for any other case straight and curved path following
            motorPIDcontrol(speed);
    }
}
```

```
while(x)                        //condition for actual run
{
        if ((c+1)==stop)
        {
                motor_stop();
        }

        //the two extreme position sensors are used for detection of node
        if((bit_is_clear(PIND,0)) || (bit_is_clear(PIND,7)))
        {
                char dir=path[c];    //path[] provides instruction for turning
                runExtraInch();
                switch(dir)
                {
                        case 'L':       //left turn
                                OCR1A = 0;
                                OCR1B = 800;
                                _delay_ms(1000);
                                c++;
                                break;

                        case 'R':       //right turn
                                OCR1B = 0;
                                OCR1A = 800;
                                _delay_ms(1000);
                                c++;
                                break;

                        case 'S':       //straight path
                                motorPIDcontrol(speed);
                                c++;
                                break;

                        default:        //for U-turning
                                motor_stop();
                                _delay_ma(100);
                                PORTA &= ~(1<<PINA2); //right motor
                                OCR1A = 800;            //set for reverse rotation
                                OCR1B = 800;
                                _delay_ms(1800);
                        }
                }


                else    //for straight and curved path following
                {
                        motorPIDcontrol(speed);
                }

        }
}

void runExtraInch()     //running few steps for alignment and node identification
{
        OCR1A = 500;
        OCR1B = 500;
        _delay_ms(330);
        motor_stop();
}

void motor_stop()
{
        OCR1A = 0;
        OCR1B = 0;
        _delay_ms(500);
}

void mazeEnd()    //indicates maze end and switches from dry run to actual run
{
        OCR1A = 0;
        OCR1B = 0;
        _delay_ms(10000);
        _delay_ms(10000);
        simplifyPath();
        stop = strlen(path);
        _delay_ms(2000);
        x=1;
        _delay_ms(100);
```

```
}

void save_path(char dir)     //saves dry run data in array path[]
{
          path[pathLength] = dir;
          pathLength++;
          _delay_us(1);
}

int arrShift(int i ,int len)
{
          for(i=i ; i<len-2 ; i++)
          {
                    path[i] = path[i+2];
          }
          len = len-2;
          return len;
}

void simplifyPath()          //for finding the shortest path using dry run data
{
          int len = strlen(path);
          int check = 0;

          start :
          for(i=0 ; i<len-2 ; i++)
          {
                    if(path[i+1] == 'U') //the logic is based on the presence of 'U'
                    {
                              switch(path[i])
                              {
                                        case 'L':
                                        switch(path[i+2])
                                        {
                                                  case 'L':
                                                  path[i+2] = 'S';
                                                  check = 1;
                                                  break;

                                                  case 'S':
```

```
                        path[i+2] = 'R';
                        check = 1;
                        break;

                        case 'R':
                        path[i+2] = 'U';
                        check = 1;
                        break;
                }
                break;

                case 'S':
                switch(path[i+2])
                {
                        case 'L':
                        path[i+2] = 'R';
                        check = 1;
                        break;

                        case 'S':
                        path[i+2] = 'U';
                        check = 1;
                        break;
                }
                break;

                case 'R':
                switch(path[i+2])
                {
                        case 'L':
                        path[i+2] = 'U';
                        check = 1;
                        break;
                }
                break;
        }
        if(check == 1)
        {
                len = arrShift(i,len);
                check = 0;
```

```
                                        goto start;
                              }
                    }
          }

}
```

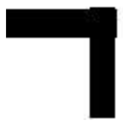
This code given here for solving a maze of black lines on a white surface includes all the concepts we have discussed so far.

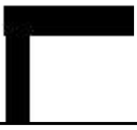
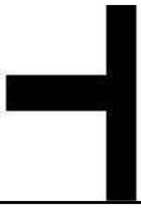
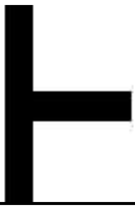
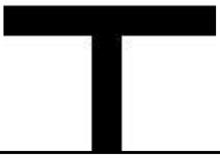
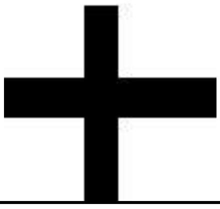
The code uses a function mazeSolve() to execute the dry and actual runs. There is a variable x, if x=0 the mazeSolve() executes dry run and if x=1 mazeSolve() executes actual run.

The values of x is updated in the mazeEnd() function, which is called at the end of dry run, so that in the next step of the code, actual run starts.

Here in the dry run, we have used run extra inch technique to determine the type of node we have reached. When the bot receives an input other than that for a straight line, it moves a few steps straight ahead with the help of runExtraInch() function. The data obtained there decides the nature of the node.

The possible types of nodes and their related combinations are tabulated below.

NODE	Combination	Next Step Combination
Left turn 	00011111 00001111 00000111	11111111

Right turn 	11111000 11110000 11100000	11111111
Straight and Left 	00011111 00001111 00000111	11110011 11100111 11100011 11001111 11000111
Straight and right 	11111000 11110000 11100000	11110011 11100111 11100011 11001111 11000111
Right and Left 	00000000	11111111
Four way 	00000000	11110011 11100111 11100011 11001111 11000111

Based on the above combination sequences, the node is determined and turning is decided on the basis of left hand algorithm. The data of each turning, in terms of characters, is stored in the path[] array.

The path reduction program consisting of simplifyPath() and arrShift() functions, find out the shortest path which is stored inthe path[] array.

Hence in the actual run, the bot uses this data present in path[] to traverse the shortest path in the maze from the starting point to the ending point.

The values given to the _delay_ms() function need to be tuned experimentally for different cases. Higher the rpm of motors used, lower will be the delay value for a turn.

The PID constants in the above code need to be tuned precisely by many repeated experiments. The logic developed here gives the bot an ability to solve various mazes of lines having straight, curved or rectangular paths, with great ease, the key here being proper values for PID constants.

A line follower bot is the most basic type of bot in the field of autonomous robotics and its application in maze solving is a nice logical problem to solve.

To make a line follower bot, one should be familiar with basic programming in language C and its use in programming the Atmel microcontroller chips. A knowledge of using basic electronic components and circuit designing will come handy in making a robust bot.

Scope of future development –

1. The code given here is applicable only for mazes that don’t have a loop. It needs to be upgraded for mazes that include one or more loops.
2. The left-hand rule can sometimes be very time consuming for solving a complete maze. A combination of different techniques will optimise maze solving.

These kinds of bots have very varied applications in automation and control engineering fields. An artificially automated environment can make a good use of line follower robots.

You can see the line follower bot built by us and its use in solving a maze by visiting the link given below.

https://www.youtube.com/watch?v=gYXfqpqtaw8

Any comments, suggestions and queries regarding this project are welcome. Any one of the following can be contacted,

Ankit Anand

B. Tech. (2nd year), Electrical Engineering
SV National Institute of Technology, Surat, India
Email ID – anand.ankit1997@gmail.com

Meet Gandhi

B.Tech. (2nd year), Electronics and Communication
Engineering SV National Institute of Technology, Surat, India
Email ID – meetj.gandhi@yahoo.com

Date: October 19, 2017 (Thursday)