e-Yantra Robotics Competition

Implementation Analysis

Line Follower

Team name	**KOLKATA ROBOSOME**
Team Leader name	**SOUMYADIP GHOSH**
College	**JADAVPUR UNIVERSITY, KOLKATA – 700032.**
Email	**soumyadip93@gmail.com**
Date	**24/01/2013**

Design Analysis

Ques-1 3x line sensors are interfaced on firebird robot with ATmega2560. Explain how you will interface the array of 7x line sensors? (15)

Initially the FIREBIRD robot has 3 white line sensors which uses the ADC of the main master micro-controller ATmega2560. To connect the 7x white line sensor, we need to interact with the slave micro-controller via the SPI bus. The following steps illustrate how the 7x line sensor module is integrated in the FIREBIRD system.

1) The main circular PCB is carefully separated from the black wooden base after opening the battery and motor connections. The 4 potentiometers provided along with the 7x module are then carefully soldered in the specified place just above the previously installed potentiometers for the 3x module
2) The 3x sensor module is then opened and the 7x module is inserted.
3) Now, to enable SPI communication between the master and slave, jumper J4 needs to be inserted.
4) Next, we have to write appropriate code for utilizing the 4 extra sensors which is represented via the flowchart below: -

FUNCTION TO CONFIGURE PINS DEDICATED TO SPI COMMUNICATION.

```
void  spi_pin_config(void)
{
        DDRB = DDRB | 0x07;
        PORTB = PORTB | 0x07;
}
```

FUNCTION TO INITIALISE SPI BUS AT CLOCK RATE OF 921600 Hz.

```
void spi_init(void)
{
        SPCR = 0x53;
        SPSR = 0x00;
        SPDR = 0x00;
}
```

FUNCTION TO SEND BYTE TO SLAVE ATmega8 AND GET ADC CHANNEL DATA FROM IT.

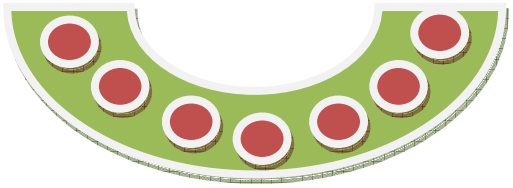
```
unsigned char spi_master_tx_and_rx (unsigned char data)
{
        unsigned char rx_data = 0;
        PORTB = PORTB & 0xFE; // make SS pin low
        SPDR = data;
        while(!(SPSR & (1<<SPIF))); //wait for data
transmission to complete
        _delay_ms(1); //time for ADC conversion in the
slave microcontroller
        SPDR = 0x50; // send dummy byte to read back
data from the slave microcontroller
        while(!(SPSR & (1<<SPIF))); //wait for data
reception to complete
        rx_data = SPDR;
        PORTB = PORTB | 0x01; // make SS high
        return rx_data;
}
```

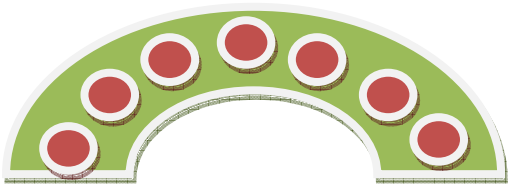
IN THE FUNCTION FOR READING SENSOR VALUES, THE APPROPRIATE CHANNEL HAS TO BE SPECIFIED.

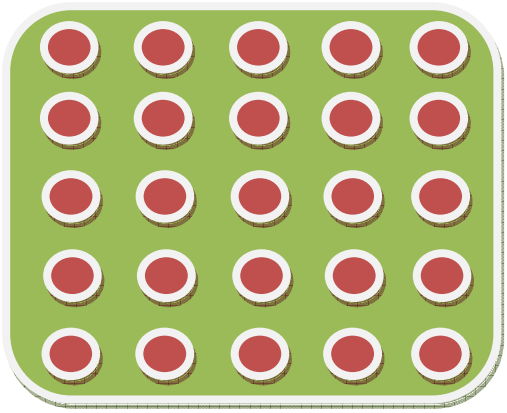
```
Fourth = spi_master_tx_and_rx(0);//4th sensor
Fifth = spi_master_tx_and_rx(1);//5th sensor
Sixth = spi_master_tx_and_rx(2);//6th sensor
Seventh = spi_master_tx_and_rx(3);//7th sensor
First = ADC_Conversion(3);//1st sensor
Second = ADC_Conversion(2); //2nd sensor
Third = ADC_Conversion(1); //3rd sensor
```

Ques-2 According to your understanding what should be the most appropriate sensor array pattern that would solve the line follower problem? (10)

a 

b 

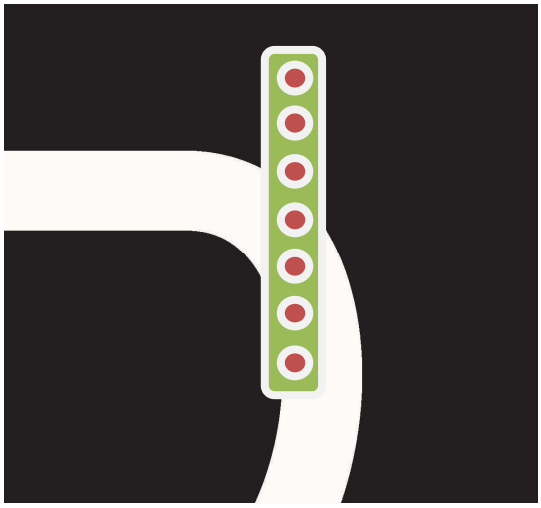
c 

d 

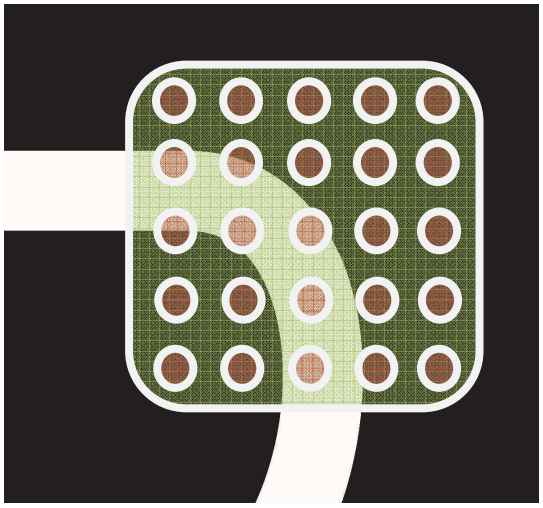
After carefully studying the line follower arena, we concluded that each of the above mentioned sensor arrays have its own merits and demerits. We highlight them through some salient curves and junctions from the arena. First let us consider a right turn:-

CASE – 1(a)

CASE – 1(b)



CASE – 1(c)



CASE – 1(d)

In the above cases, we find that if we consider that the position of the 4th sensor is more or less the same in all the cases:-

a) In case 1(a) of convex shaped sensor array, we see that during the right convex turn, only 2 sensors, i.e, the 4th and the 5th are on the white line. So this position will produce lesser turning effect.

b) In case 1(b) of concave sensor array, almost 4 sensors come over the white line. So the control variable generated via PID algorithm will be more and hence this will produce faster turning effect.

c) In case 1(c) of straight sensor array, 4 sensors are over the white line. So this case is similar to 1(b) and generates fast turning effect.

d) In case 1(d) of 5 X 5 sensor matrix, 3 sensors out of a column of 5 cover the white line. So this will also generate similar control value as the percentage of sensors over the line(3 out of 5) is similar to the last 2 cases(4 out of 7). But here the remaining columns are not utilized and are hence redundant.

In this particular convex curve, cases 1(b) and 1(c) are most fitting. Case 1(d) also satisfies the faster turning effect requirements but is not economic as many sensors are going unutilized.

Let us now consider a sharp left turn.

CASE – 2(a)

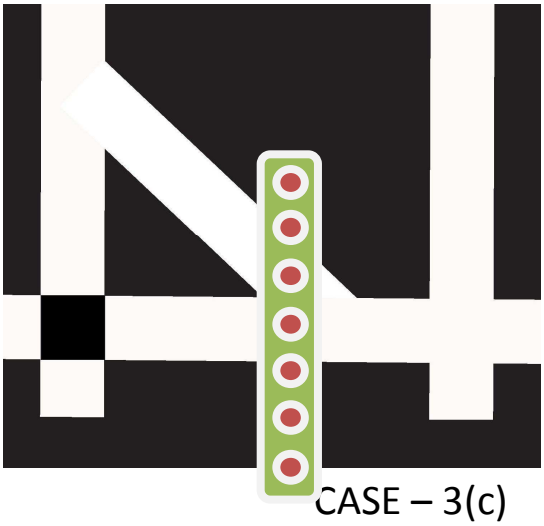
CASE – 2(b)

Here we see clearly that the concave sensor array in case 2(a) has only 2 sensors on the line while in case 2(b), 4 sensors are on the line. So case 2(b) will produce faster and more turning effect. Hence the convex sensor array is better than the concave one in this case.

Now let us consider a case in the grid.

CASE – 3(a)

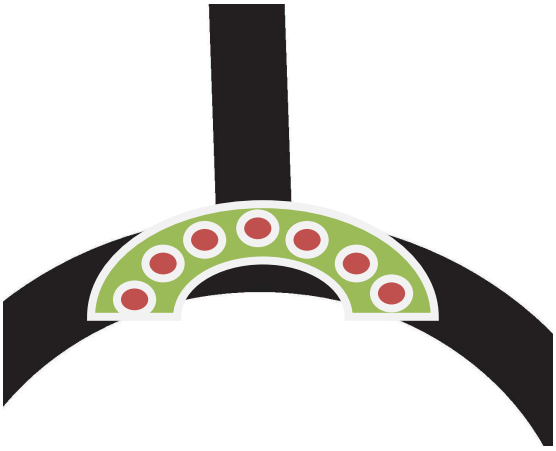
CASE – 3(b)



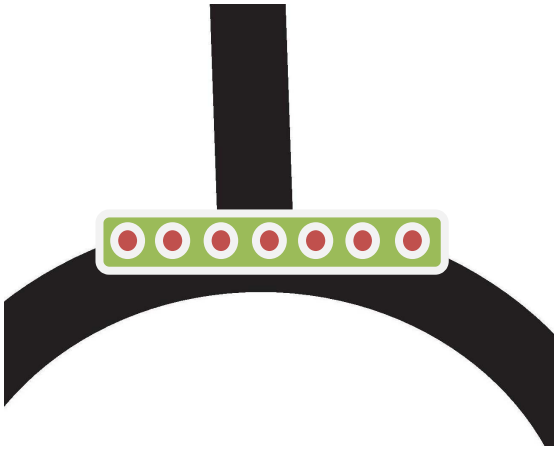
CASE – 3(c)

Here in case 3(a), 4 sensors are on the white line and hence if the necessity arises to follow the shorter path along the hypotenuse in the grid, it can be done to a great extent, fast and easily. The cases 3(b) and 3(c) have 2 sensors on the line and so they take time to turn. Hence the concave sensor array is most befitting in this case like case 1.

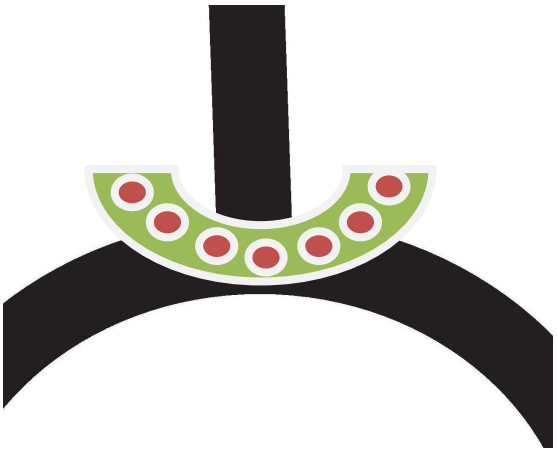
Let us now consider the case of the start of the loop in the black line.



CASE – 4(a)



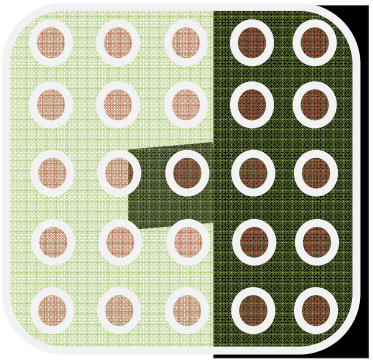
CASE – 4(b)



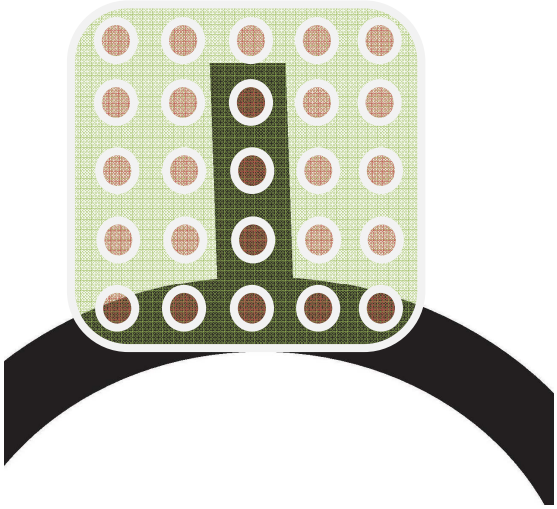
CASE – 4(c)

In this case, 4(a) has all sensors on the line, 4(b) has almost all on the line, but 4(c) has only 3 on the line. In our algorithm, the condition used to detect the start of the loop is when all the sensors are on the line. So 4(c) is not suitable.

Now let us discuss another case

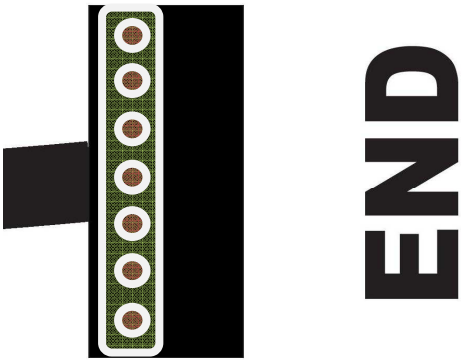


END

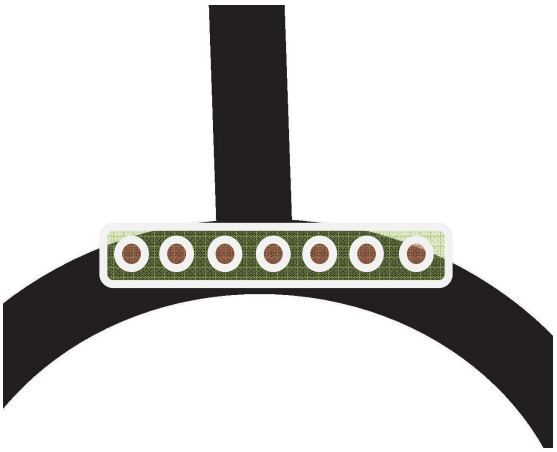


CASE – 5(a)

CASE – 5(b)



CASE – 5(c)



CASE – 5(d)

Here we compare the straight sensor array and the 5 x 5 matrix sensors for 2 regions – the start of the loop and the END point T-junction. In case of the straight sensor array, we see that both these regions generate the same sensor states, i.e, all 7 sensors are on the line(Fig 5(c) and 5(d)). However, for the 5 X 5 matrix, we see a different scenario. In case of the T-junction, 2 rows of 5 sensors and the middle sensor of the next row are on the line(Fig 5(a)) but in case of the start of the loop, 1 complete row and the middle sensors of all the next rows cover the line (Fig 5(b)). Clearly they generate different sensor states and can be distinguished easily which is difficult for the straight sensor array. So the 5 X 5 matrix sensors is advantageous here.

After considering all these 5 cases (non-exhaustive), we arrive at a conclusion that all the 4 different sensor configurations have their own pros and cons. The straight sensor array and the concave sensor array are suitable in majority of the regions. The concave sensor array, because of its shape, can figure out the curvature of the line beforehand which is advantageous. However, the straight sensor array is most general and fits more or less well in all the cases.

Since the straight sensor array is generic and is applicable in most of the cases, we use them in our line follower theme.

Algorithm Analysis

Q-1 Draw a flowchart illustrating the algorithm used to complete the entire task. (50)

The flowchart mentioned below gives a tentative algorithm in the sequence of the code of the Embedded C program. At first, the functions necessary for the main algorithm are described along with relevant details. Then the main algorithm is illustrated where the necessary functions are called to perform specific tasks to solve the line follower problem.

START PROGRAM

INCLUDE NECESSARY HEADER FILES AND DEFINE CRYSTAL FREQUENCY

```
1) #define F_CPU 14745600
2) #include <avr/io.h>//for I/O Operations
3) #include <avr/interrupt.h>//for Interrupt Handling
4) #include <util/delay.h>//for generating delay
5) #include <math.h> //included to support power function
6) #include "lcd.h" //for including LCD related functions
```

DECLARE GLOBAL VARIABLES NECESSARY FOR DIFFERENT FUCTIONS IN THE ALGORITHM

```
1) //Sensor values after ADC conversion. The actual integer value of sensors is stored here
   unsigned char First, Second, Third, Fourth, Fifth, Sixth, Seventh;

2) //Sensor threshold values. These values represent the threshold for each sensor, i.e,
   //the value below which a sensor distinctly detects a white line or white background
   //and above which it distinctly detects a black line or black background. These
   //variables are specified separately so that it is easy to finetune the program after
   //calibration in the actual arena
   int t1, t2, t3, t4, t5, t6, t7; //values will be assigned to them after calibration.
   // 1 denotes the extreme left sensor when rear of the robot is towards the user

3) //Array to store the current state of all the 7 sensors. This is a 7 element array
   //where the index 0 stores current state of 1st sensor and so on. In the first half,
   //when the bot is traversing a white line on black background, if a sensor comes over a
   //white line, its corresponding index in the array becomes 1(high), otherwise on black
```

DECLARE GLOBAL VARIABLES NECESSARY FOR DIFFERENT FUCTIONS IN THE ALGORITHM (CONTD)

```
    //background, it remains 0(low). In the second half,when bot is traversing a black line
    //on white background, a black line makes the corresponding sensor state go 1(high)
    //while the white background makes the state 0(low). This reversing of logic from
    //white line following to black line following is specified later.
    int arr[7]; //the array will be initialized later while reading sensor values

4)  //To store the current state of the sensors as stated above in a single integer value,
    //we define a variable "arr_value". For example, an arr_value of 0001000 on the white
    //line(black back) means only the 4th sensor is on the white line, while an arr_value
    //of 0001111 on the black line(white back) means 4,5,6,7th sensors are on black line
    //while others are on white background.
    unsigned long int arr_value;//here type specifier is "long int" because the maximum
                //value can be 1111111 which exceeds the range of "int" data type in C.

5)  //An array "prev_values" is maintained to store previous values of "arr_value". The
    //number of previous values stored is taken to be 10 here, however it may vary. This
    //array actually stores the history of the sensor values which is useful in the
    //following cases: a) negotiating sharp turns fast b) reversing the logic from white
    //line to black line following, etc. How it is used in each of the aforementioned cases
    //will be specified in the appropriate functions
    unsigned long int prev_values[10];

6)  //logic reverse variable used when white line changes to black line
    int line = 0;//0 is used when following white line, when following black line, 1 is
    //used.

7)  //Two variables "x1" and "x2" govern the kind of line following and modify the way
    //the indexes in the array "arr" are intialised. Intially, x1 = 1 and x2 = 0.
    //So, while following white line, if a particular sensor detects white line, its value
    //is set to x1(=1) and others are set to x2(=0). An example is given below
    /*if(First < t1)//t1 is threshold value
    {
          arr[0] = x1;
    }
    else
    {
          arr[0] = x2;
    }*/
    //Now when the logic reverses, i.e, the bot becomes black line following, x1 is made 0
    //and x2 is made 1. So after logic reversal, when any sensor comes under black line,
    //its ADC value is more than the threshold value(here t1) and hence the else block of
    //the conditional statement gets executed, i.e, the corresponding sensor state(here
    //arr[0]) becomes x2(now equal to 1). The opposite happens on white background now.
    //This kind of reasoning minimizes the code to a great extent and makes programming
    //robust.

    int x1 = 1;//later made to 0 when black line following
    int x2 = 0;// later made to 1 when black line following

8)  //Variables to monitor state of position encoder
    unsigned long int ShaftCountLeft = 0; //to keep track of left position encoder
    unsigned long int ShaftCountRight = 0; //to keep track of right position encoder

9)  //Weighted average value of sensors. The way in which this value is calculated in
    //illustrated in the function "readSensors()" later.
    float s;

10) //Variables for P control function(Proportional Control), details described later
    float pGain = k;//Proportional Gain, k is calibrated depending on the turns
    float control;//control variable to determine degree of turning.
```

CONFIGURING PORTS AND PINS IN THE MASTER AND SLAVE MICRO-CONTROLLER FOR VARIOUS FUNCTIONALITIES

1)
```
//Configure LCD port which is needed to display values on the LCD
void lcd_port_config (void)
{
       DDRC = DDRC | 0xF7; //all the LCD pin's direction set as output
       PORTC = PORTC & 0x80; // all the LCD pins are set to logic 0 except PORTC 7
}
```

2)
```
//ADC pin configuration which is needed to convert analog sensor values to digital
void adc_pin_config (void)
{
       DDRF = 0x00; //set PORTF direction as input
       PORTF = 0x00; //set PORTF pins floating
       DDRK = 0x00; //set PORTK direction as input
       PORTK = 0x00; //set PORTK pins floating
}
```

3)
```
//Configure SPI port which is needed for interaction between master and slave micro-
//controllers. In this project, we need to get the 4, 5, 6, 7th white line sensor values
//via SPI from the slave ATmega8.
void spi_pin_config (void)
{
       DDRB = DDRB | 0x07;
       PORTB = PORTB | 0x07;
}
```

4)
```
//Configure ports to enable robot's motion
void motion_pin_config (void)
{
       DDRA = DDRA | 0x0F;
       PORTA = PORTA & 0xF0;
       DDRL = DDRL | 0x18;   //Setting PL3 and PL4 pins as output for PWM generation
       PORTL = PORTL | 0x18; //PL3 and PL4 pins are for velocity control using PWM.
}
```

5)
```
//Configure Buzzer. Buzzer is turned ON when robot reaches END point in the arena.
void buzzer_pin_config (void)
{
       DDRC = DDRC | 0x08;//Setting PORTC 3 as output
       PORTC = PORTC & 0xF7;//Setting PORTC 3 logic low to turnoff buzzer
}
```

6)
```
//Configure INT4 (PORTE 4) pin as input for the left position encoder
void left_encoder_pin_config (void)
{
       DDRE  = DDRE & 0xEF;  //Set the direction of the PORTE 4 pin as input
       PORTE = PORTE | 0x10; //Enable internal pullup for PORTE 4 pin
}
```

7)
```
//Configure INT5 (PORTE 5) pin as input for the right position encoder
void right_encoder_pin_config (void)
{
       DDRE  = DDRE & 0xDF;  //Set the direction of the PORTE 5 pin as input
       PORTE = PORTE | 0x20; //Enable internal pullup for PORTE 5 pin
}
```

ENABLE POSITION ENCODER INTERRUPTS

```
//Position encoders are useful while traversing the grid in the white line
//following part. The actual procedure for traversing the grid is described
//later
```

1)
```
//Initializes the left position encoder interrupt.
void left_position_encoder_interrupt_init (void) //Interrupt 4 enable
{
	cli(); //Clears the global interrupt
	EICRB = EICRB | 0x02; // INT4 is set to trigger with falling edge
	EIMSK = EIMSK | 0x10; // Enable Interrupt INT4 for left position
				//encoder
	sei();   // Enables the global interrupt
}
```

2)
```
//Initializes the right position encoder interrupt
void right_position_encoder_interrupt_init (void) //Interrupt 5 enable
{
	cli(); //Clears the global interrupt
	EICRB = EICRB | 0x08; // INT5 is set to trigger with falling edge
	EIMSK = EIMSK | 0x20; // Enable Interrupt INT5 for right position
				//encoder
	sei();   // Enables the global interrupt
}
```

INTERRUPT SERVICE ROUTINE (ISR) DEFINATION

1)
```
//ISR for left position encoder
ISR(INT4_vect)
{
	ShaftCountLeft++;//increment left shaft position count
}
```

2)
```
//ISR for right position encoder
ISR(INT5_vect)
{
	ShaftCountRight++;//increment right shaft position count
}
```

INTIALISATION OF PORTS, PINS AND CHANNELS

1) `//This function initializes the ports and pins already configured before`

```
void port_init()
{
       lcd_port_config();
       adc_pin_config();
       spi_pin_config();
       motion_pin_config();
       buzzer_pin_config();
       left_encoder_pin_config();
       right_encoder_pin_config();
}
```

2)
```
// Timer 5 initialized in PWM 8 bit fast mode needed for velocity control
// Prescale:256, TOP=0x00FF, Timer Frequency:225.000Hz
void timer5_init()
{
       TCCR5B = 0x00;//Stop
       TCNT5H = 0xFF;//Counter higher 8-bit value to which OCR5xH value
                     //is compared with
       TCNT5L = 0x01;//Counter lower 8-bit value to which OCR5xH value is
                     //compared with
       OCR5AH = 0x00;//Output compare register high value for Left Motor
       OCR5AL = 0xFF;//Output compare register low value for Left Motor
       OCR5BH = 0x00;//Output compare register high value for Right Motor
       OCR5BL = 0xFF;//Output compare register low value for Right Motor
       OCR5CH = 0x00;//Output compare register high value for Motor C1
       OCR5CL = 0xFF;//Output compare register low value for Motor C1
       TCCR5A = 0xA9;//COM5A1=1, COM5A0=0; COM5B1=1, COM5B0=0; COM5C1=1
                     //COM5C0=0
       TCCR5B = 0x0B;//WGM12=1; CS12=0, CS11=1, CS10=1 (Prescaler=64)
}
```

3)
```
//Function to Initialize ADC Channels
void adc_init()
{
       ADCSRA = 0x00;
       ADCSRB = 0x00;//MUX5 = 0
       ADMUX = 0x20;//Vref=5V external --- ADLAR=1 --- MUX4:0 = 0000
       ACSR = 0x80;
       ADCSRA = 0x86;//ADEN=1 --- ADIE=1 --- ADPS2:0 = 1 1 0
}
```

4)
```
//Function To Initialize SPI bus at clock rate of 921600hz
void spi_init(void)
{
       SPCR = 0x53; //setup SPI
       SPSR = 0x00; //setup SPI
       SPDR = 0x00;
}
```

DEFINING BUZZER FUNCTIONS

1)
```
//Function to turn ON the buzzer
void buzzer_on (void)
{
	unsigned char port_restore = 0;
	port_restore = PINC;
	port_restore = port_restore | 0x08;
	PORTC = port_restore;
}
```

2)
```
//Function to turn OFF the buzzer
void buzzer_off (void)
{
	unsigned char port_restore = 0;
	port_restore = PINC;
	port_restore = port_restore & 0xF7;
	PORTC = port_restore;
}
```

MOTOR DIRECTION AND VELOCITY CONTROL FUNCTIONS

1)
```
//Function used for setting motor's direction
void motion_set (unsigned char Direction)
{
	unsigned char PortARestore = 0;

	Direction &= 0x0F;//removing upper nibble for protection
	PortARestore = PORTA;//reading PORTA original status
	PortARestore &= 0xF0;//making lower direction nibble to 0
	PortARestore |= Direction;//adding lower nibble for forward command
				//and restoring the PORTA status
	PORTA = PortARestore;//executing the command
}
//In the above function, different hexcodes as arguments set corresponding
//directions for locomotion as follows:-
//motion_set (0x06) - FORWARD,i.e, both wheels forward
//motion_set(0x04) - SOFT LEFT,i.e, left wheel stationary & right forward
//motion_set(0x02) - SOFT RIGHT,i.e, left wheel forward & right stationary
//motion_set(0x05) - LEFT,i.e, left wheel backward & right forward
//motion_set(0x0A) - RIGHT,i.e, left wheel forward & right backward
//motion_set(0x09) - BACK,i.e, both wheels backward
//motion_set (0x00) - STOP,i.e, both wheels stationary
//Accordingly, functions can be defined for each motion separately,
//e.g - forward(), left(), etc.
```

2)
```
//Function for velocity control using PWM by Timer 5
void velocity (unsigned char left_motor, unsigned char right_motor)
{
	OCR5AL = (unsigned char)left_motor;
	OCR5BL = (unsigned char)right_motor;
	//Since timer 5 has maximum value of 255, its OCR5AL and OCR5BL
	//registers cannot exceed that value. Hence maximum velocity of
	//motor corresponds to the value of 255 in the registers. Also,
	//for practical cases, a value less than 100 is too low to actuate
	//the motor. So the values assigned must be within 100 to 255.
}
```

FIXED LOCOMOTION FUNCTIONS

```
//These functions are used for moving the robot forward for backward by a
//certain specified distance, turning left or right by certain degrees, etc.
//These functions may be utilized while traversing the grid of white lines.

1) //Function used for moving robot forward or backward by specified distance
   void linear_distance_mm(unsigned int DistanceInMM)
   {
          float ReqdShaftCount = 0;
          unsigned long int ReqdShaftCountInt = 0;
          ReqdShaftCount = DistanceInMM / 5.338; //division by resolution to get
                                                 //shaft count
          ReqdShaftCountInt = (unsigned long int) ReqdShaftCount;
          ShaftCountRight = 0;
          while(1)
          {
                 //the lcd_print function is defined in the "lcd.h" header file.
                 //Here the three variables below may be displayed so as check
                 //if the position encoder interrupts are working fine or not.
                 //During final run, they may be by-passed.
                 lcd_print(1, 1, ShaftCountLeft, 3);
                 lcd_print(1, 5, ShaftCountRight, 3);
                 lcd_print(1, 9, ReqdShaftCountInt, 3);
                 if(ShaftCountRight > ReqdShaftCountInt)//If right(or left) shaft
                                                 //exceeds the required shaft count
                 {
                        break;
                  }
           }
           stop(); //Stop action
   }

2) //Function used for turning robot left or right by specified degrees.
   void angle_rotate(unsigned int Degrees)
   {
          float ReqdShaftCount = 0;
          unsigned long int ReqdShaftCountInt = 0;
          ReqdShaftCount = (float) Degrees/ 4.090; //division by resolution to
                                                   //get shaft count
           ReqdShaftCountInt = (unsigned int) ReqdShaftCount;
           ShaftCountRight = 0;
           ShaftCountLeft = 0;
           while (1)
          {
                 lcd_print(1, 1, ShaftCountLeft, 3);
                 lcd_print(1, 5, ShaftCountRight, 3);
                 lcd_print(1, 9, ReqdShaftCountInt, 3);
                 if((ShaftCountRight >= ReqdShaftCountInt) | (ShaftCountLeft >=
                     ReqdShaftCountInt))
                        break;
           }
          stop(); //Stop action
   }

   //The "angle_rotate" function may be implemented in normal left/right
   //mode or soft left/right mode. In normal mode, there are 88 pulses for 360
   //degrees rotation,i.e, 4.090 degrees per count. But in soft mode, there are
   //176 pulses for 360 degrees rotation,i.e, 2.045 degrees per count. So in soft
   //mode, if x degrees turning is to be achieved actually, 2*x must be passed as
   //argument to the "angle_rotate" function since only one wheel is moving.
```

FUNCTIONS TO OBTAIN ALL THE SEVEN WHITE LINE SENSOR VALUES

1)
```
//For the first 3 white line sensors(sensors 1,2,3), we use the ADC Conversion
//of the ATmega2560 master as is shown in the function below
//This Function accepts the Channel Number and returns the corresponding value
unsigned char ADC_Conversion(unsigned char Ch)
{
	unsigned char a;
	if(Ch>7)
	{
		ADCSRB = 0x08;
	}
	Ch = Ch & 0x07;
	ADMUX= 0x20| Ch;
	ADCSRA = ADCSRA | 0x40;//Set start conversion bit
	while((ADCSRA&0x10)==0);//Wait for ADC conversion to complete
	a=ADCH;
	ADCSRA = ADCSRA|0x10;//clear ADIF(ADC Interrupt Flag) by writing 1 to it
	ADCSRB = 0x00;
	return a;
}
```

2)
```
//For the 4,5,6,7th sensors, we have to use the ADC channel of the ATmega8.Hence
//we need to obtain it via a function to transfer data over SPI bus
//Function to send byte to the slave microcontroller and get ADC channel data
//from the slave microcontroller
unsigned char spi_master_tx_and_rx (unsigned char data)
{
	unsigned char rx_data = 0;
	PORTB = PORTB & 0xFE; // make SS pin low
	SPDR = data;
	while(!(SPSR & (1<<SPIF))); //wait for data transmission to complete
	_delay_ms(1); //time for ADC conversion in the slave microcontroller
	SPDR = 0x50; // send dummy byte to read back data from the slave
			// microcontroller
	while(!(SPSR & (1<<SPIF))); //wait for data reception to complete
	rx_data = SPDR;
	PORTB = PORTB | 0x01; // make SS high
	return rx_data;
}

//Thus the seven sensor values are obtained as follows:-
//First = ADC_Conversion(3);
//Second = ADC_Conversion(2);
//Third = ADC_Conversion(1);
//Fourth = spi_master_tx_and_rx(0);
//Fifth = spi_master_tx_and_rx(1);
//Sixth = spi_master_tx_and_rx(2);
//Seventh = spi_master_tx_and_rx(3);
```

FUNCTION TO CALCULATE WEIGHTED AVERAGE VALUE OF 7 SENSORS

```
//This function calculates the weighted average value of the 7 sensors and
//also assigns value to the sensor-related variables and arrays

float readSensors()
{
	float avgSensors = 0.0;//initialized to 0

	First = ADC_Conversion(3);//1st sensor
	Second = ADC_Conversion(2);//2nd sensor
	Third = ADC_Conversion(1);//3rd sensor
	Fourth = spi_master_tx_and_rx(0);//4th sensor
	Fifth = spi_master_tx_and_rx(1);//5th sensor
	Sixth = spi_master_tx_and_rx(2);//6th sensor
	Seventh = spi_master_tx_and_rx(3);//7th sensor
	if(First < t1)//the logic of this code has already been discussed
	{		//before.
		arr[0] = x1;
	}
	else
	{
		arr[0] = x2;
	}
	if(Second < t2)
	{
		arr[1] = x1;
	}
	else
	{
		arr[1] = x2;
	}
	if(Third < t3)
	{
		arr[2] = x1;
	}
	else
	{
		arr[2] = x2;
	}
	if(Fourth < t4)
	{
		arr[3] = x1;
	}
	else
	{
		arr[3] = x2;
	}
	if(Fifth < t5)
	{
		arr[4] = x1;
	}
	else
	{
		arr[4] = x2;
	}
```

FUNCTION TO CALCULATE WEIGHTED AVERAGE VALUE OF 7 SENSORS (CONTD)

```
        if(Sixth < t6)
        {
                arr[5] = x1;
        }
        else
        {
                arr[5] = x2;
        }
        if(Seventh < t7)
        {
                arr[6] = x1;
        }
        else
        {
                arr[6] = x2;
        }

        arr_value = arr[0] * 1000000 + arr[1] * 100000 + arr[2] * 10000 +
                    arr[3] * 1000 + arr[4] * 100 + arr[5] * 10 + arr[6];
        //arr_value is generated from the arr array.

        //calculate weighted mean
        avgSensors = (float)((arr[0] * 1 + arr[1] * 2 + arr[2] * 3 + arr[3] * 4 +
                      arr[4] * 5 + arr[5] * 6 + arr[6] * 7)/ (arr[0] + arr[1] +
                      arr[2] + arr[3] + arr[4] + arr[5] + arr[6]);
        //the procedure for calculation of the weighted mean involves assigning
        //a weightage to each sensor in order,i.e, 1st sensor has weightage 1
        //while 7th sensor has weightage 7 and then calculating the mean. For
        //example, a sensor state of 0001111 means avgSensors value is
        //(0 * 1 + 0 * 2 + 0 * 3 + 1 * 4 + 1 * 5 + 1 * 6 + 1 * 7)/(0+0+0+1+1+1+1)
        //which yields 5.5. The purpose is to keep this avgSensors value at 4.0.
        //Accordingly, the error from 4.0 is calculated and a control variable is
        //generated based on the error to initiate turning.
}
```

FUNCTION TO GENERATE CONTROL VARIABLE ACCORDING TO PID ALGORITHM

```
//This function generates the control variable. However, here we use only
//proportional gain as integral and derivative gain are not of much use to
//us till now. But we can modify the algorithm later.
float PID(float cur_value, float req_value)//In our case, cur_value is the
{                                          //average sensor value and req_value is 4.0
    float pid;//the local control variable
    float error;
    error = req_value - cur_value;
    pid = (pGain * error);//pGain obtained from calibration
    if(pid < 0) pid *= -1;//we extract only magnitude of control without sign
    return pid;
 }
```

FUNCTIONS RELATED TO THE ARRAYS

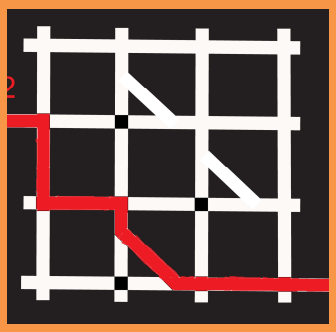
1)
```
//Function to initialize the array cells with 0 before use since in C language, they
//are filled with garbage by default.
//This function is invoked in the main block as specified later
void array_initialise()
{
	int i;//local counter variable
	for(i=0; i < 10; i++)
	{
		prev_values[i] = 0;
	}
	for(i = 0; i < 7; i++)
	{
		arr[i] = 0;
	}
}
```

2)
```
//Function to shift cells in array "prev_values[10]" left by 1 when new
//"arr_value" has to be stored at the extreme right cell. The extreme left
//"arr_value" is thus removed
void shift(int value)//argument "value" is the new state to be stored in
{			//the rightmost cell.
	int i;//local counter variable
	for(i = 0; i < 7 - 1; i++)
	{
		prev_values[i] = prev_values[i+1];
	}
	prev_values[6] = value;
}
```

3)
```
//Function used to perform comparisons among cells in prev_values array.
int prev_values_search(long int value, int start_index, int end_index)
{
	//Here we search for the existence of a particular value of 1 or more
	//sensors in the 7x sensor array. "value" is the search parameter,
	//"start_index" is the sensor number from which we are considering
	//and "end_index" is the sensor number upto which we are considering.
	//For e.g, a function call of prev_values_search(111, 3, 5) will check
	//if any of the previous sensor states have the state 1 for the 3rd,
	//4th and 5th sensors. Care must be taken that the value provided for
	//searching must have number of digits in accordance with the
	//start_index and end_index,i.e, they must have (end_index – start_index + 1)
	//number of digits. For checking values of non-consecutive sensors, the
	//function can be called separately for each sensor.
	int i;//local array loop counter
	int flag = 0;//status of the search. If a match is found, this becomes 1.
	for(i = 0; i < 10; i++)
	{
		long int temp = prev_values[i];//storing previous values in temporary
					//variable
		temp /= (long int)pow(10, 7 - end_index);//removing the tail end from
					//the entire 7 digit long integer.
		temp %= (long int)pow(10, end_index - start_index + 1);//removing the
					//front end from the entire 7 digit long integer.
		if(value == temp) //found match
		{
			flag = 1;
			return flag; //return 1(positive)
		}
	}
	return flag; //return 0(negative)
}
```

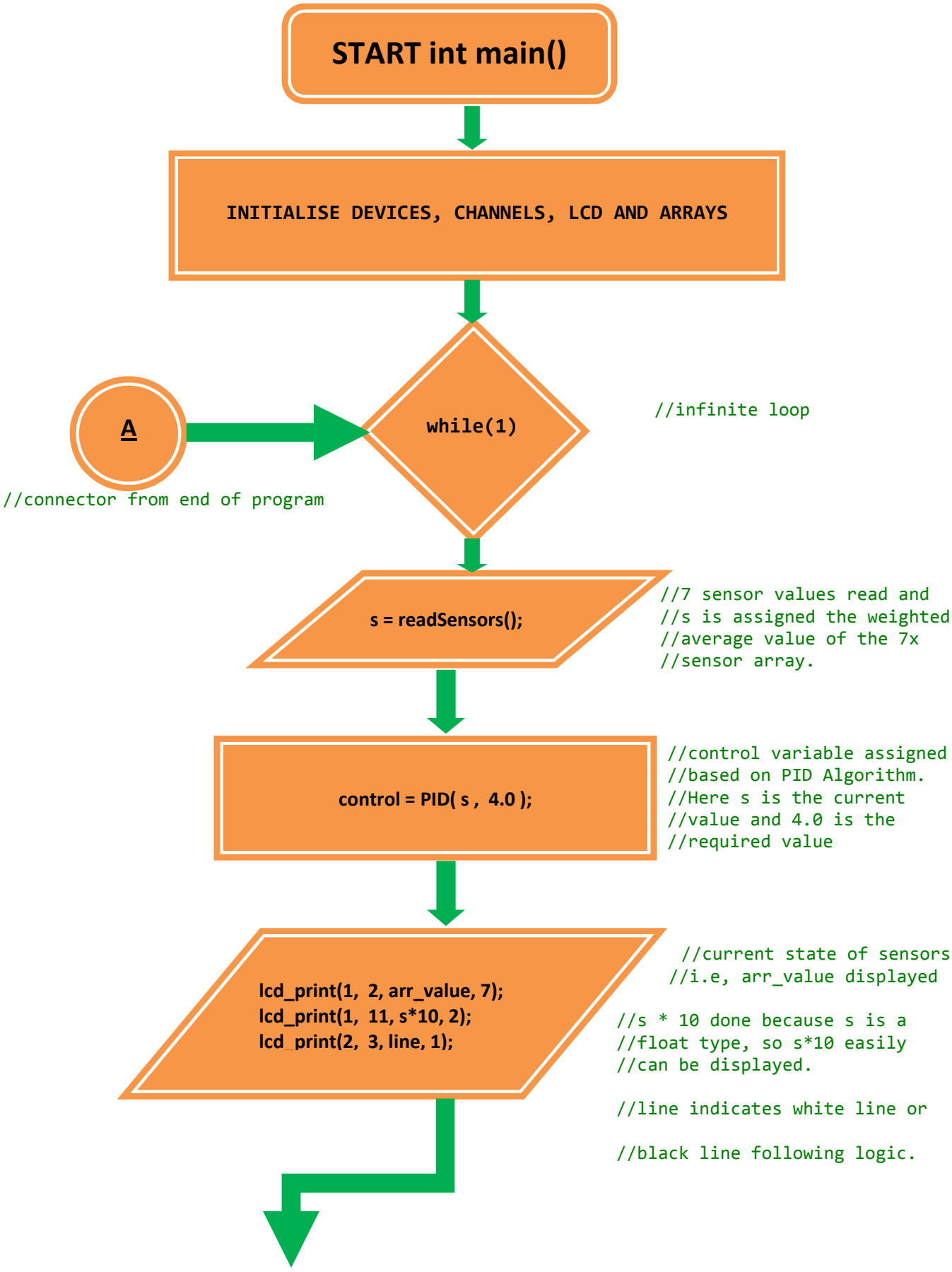
FUNCTION FOR GRID TRAVERSAL

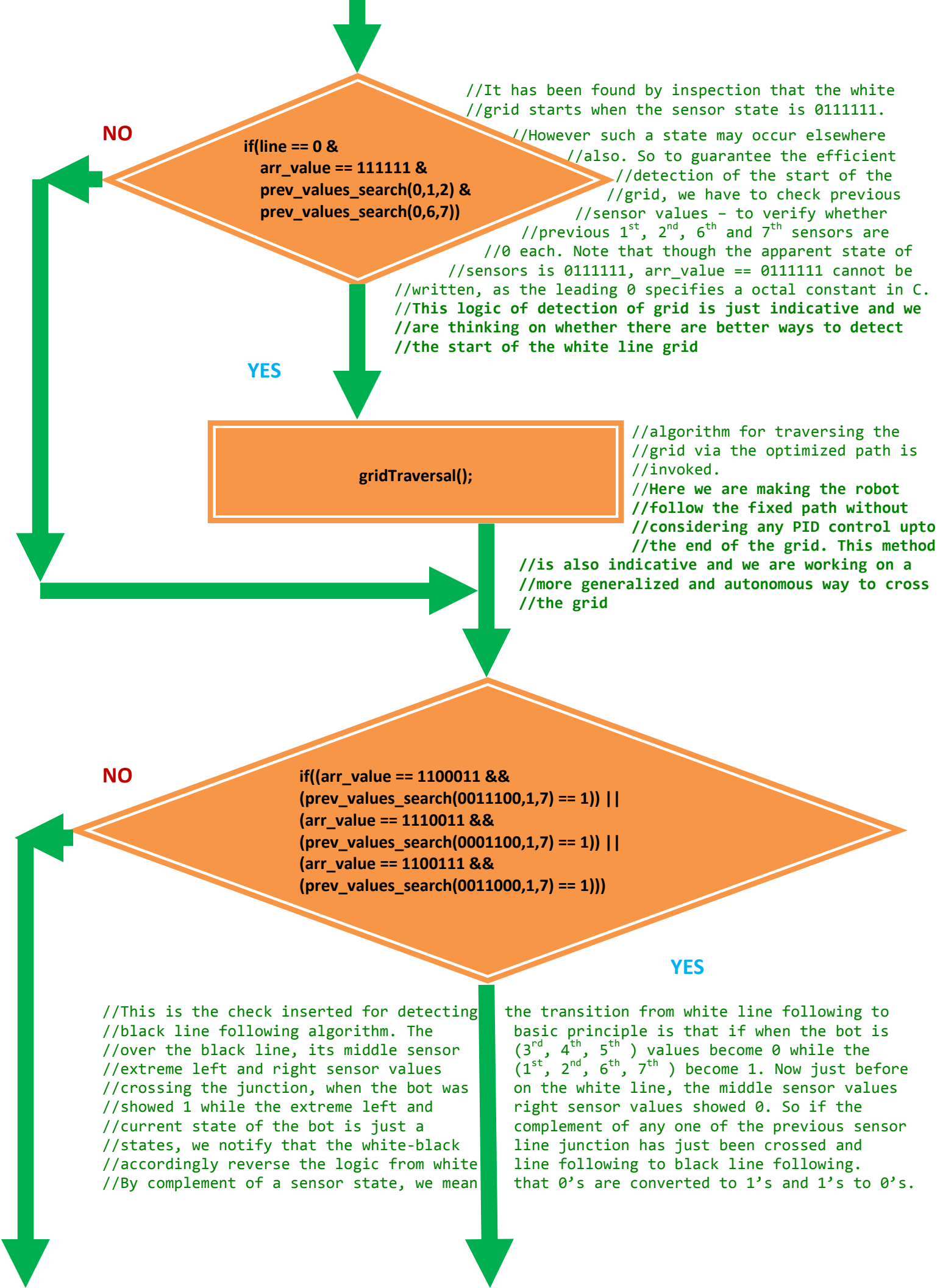
```
//This function specifies the algorithm for traversing the grid.

void gridTraversal()
{
	forward();
	velocity(150, 150);
	forward_mm(260); //moves forward by 26 cm
	right_degrees(45); //right turn by 45 degrees
	forward_mm(160); //moves forward by 16 cm
	right_degrees(45); //right turn by 45 degrees
	forward_mm(8); //moves forward by 8 cm
	left_degrees(90); //left turn by 90 degrees
	forward_mm(180); //moves forward by 18 cm
	right_degrees(90); //right turn by 90 degrees
	forward_mm(180); //moves forward by 18 cm
	left_degrees(90);//left turn by 90 degrees
}
```

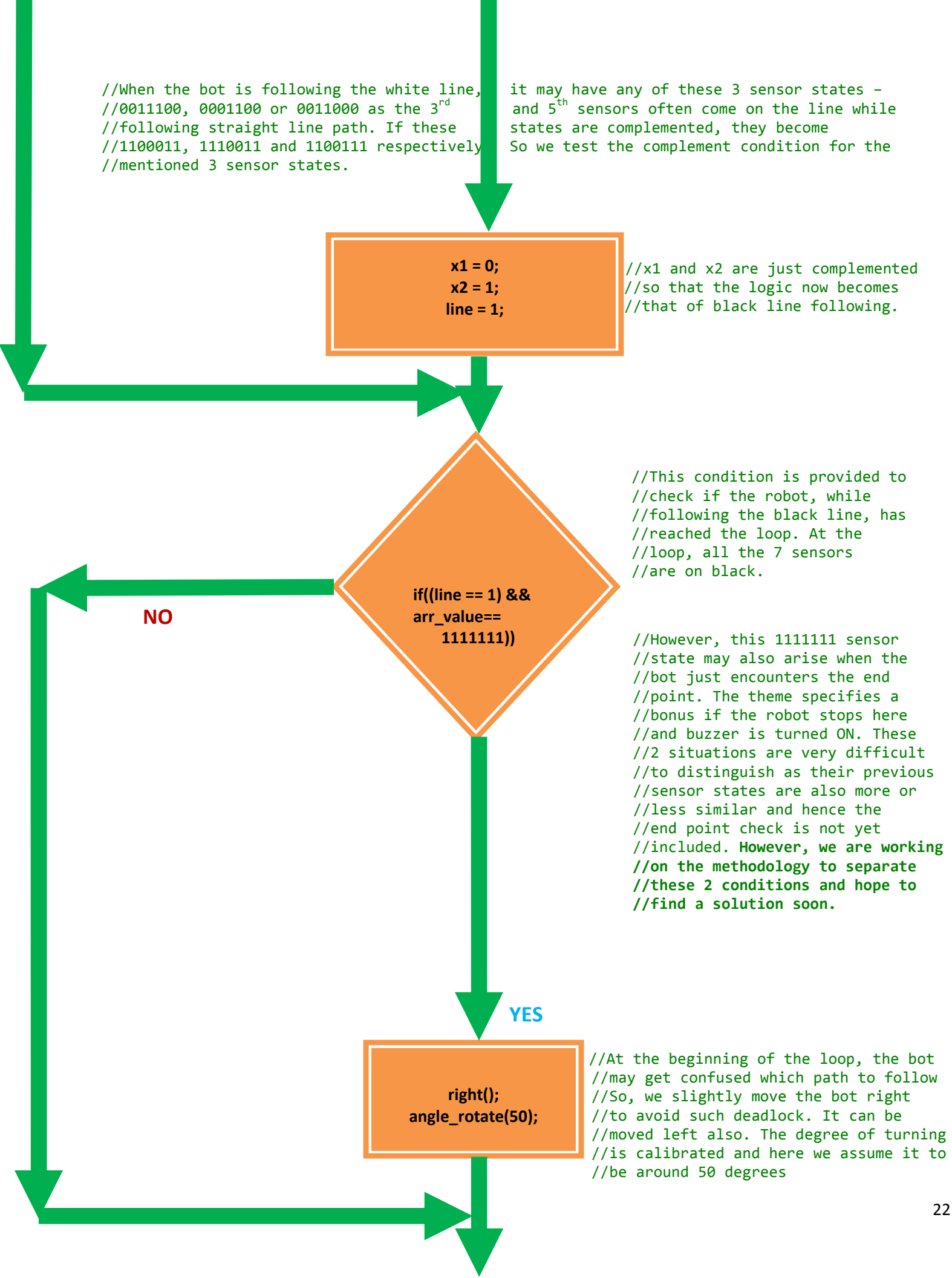


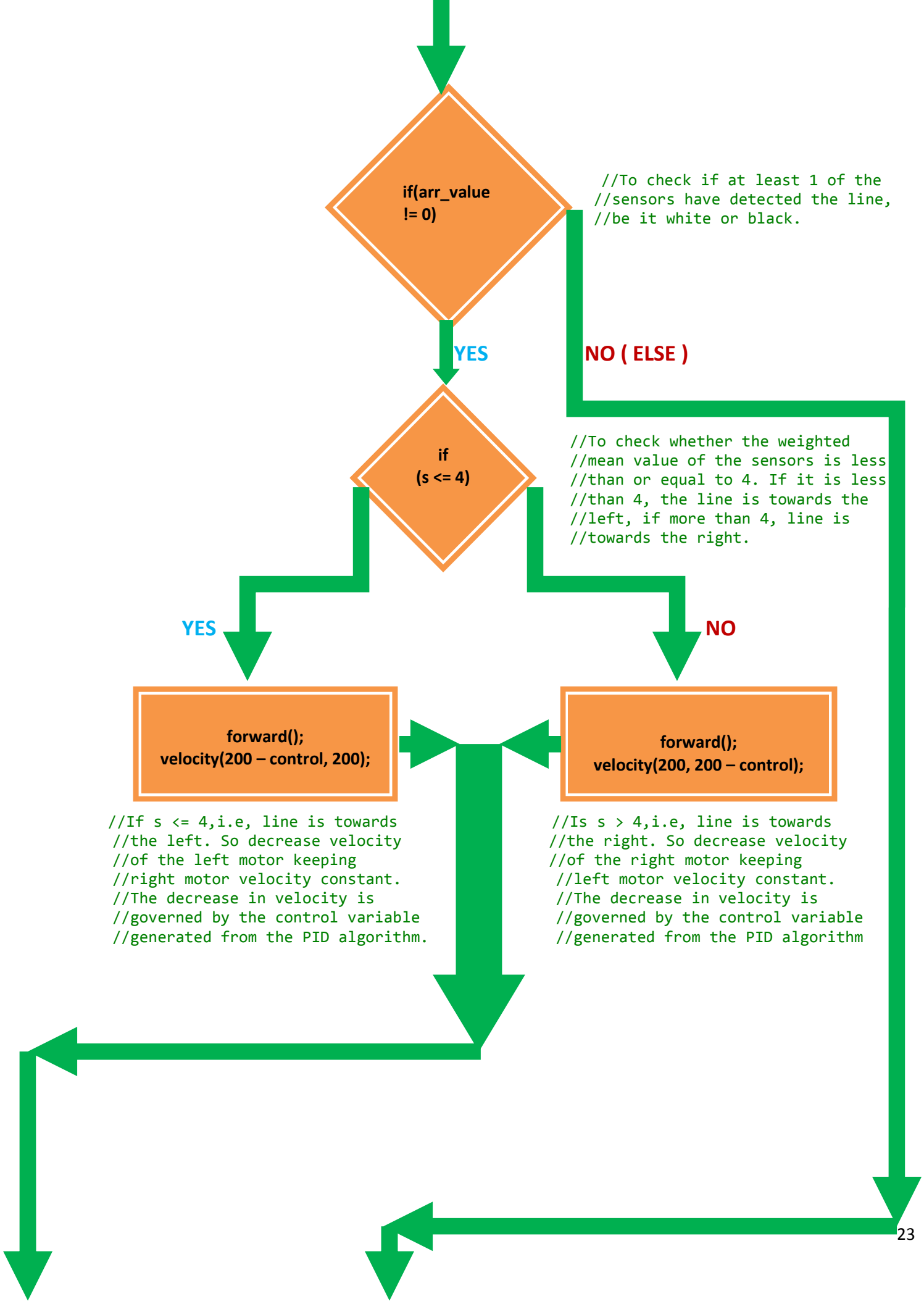
```
//The most optimized path of following the grid is shown above by the red line. It is
//exactly this path that is being followed in the above function after measurement of
//distances and angles. Here we make the robot follow a fixed path (the most optimized
//path shown above) on detection of the start of the grid. However, such an approach of
//fixed locomotion is not generic and is difficult to apply when the arena is not known
//beforehand.

//Due to the complexity and relative difficulty of the grid traversal part, we are
//suggesting the above indicative algorithm for now. However, we are working to find a
//better, more general way to traverse the grid and hope to come up with a solution
//soon.
```

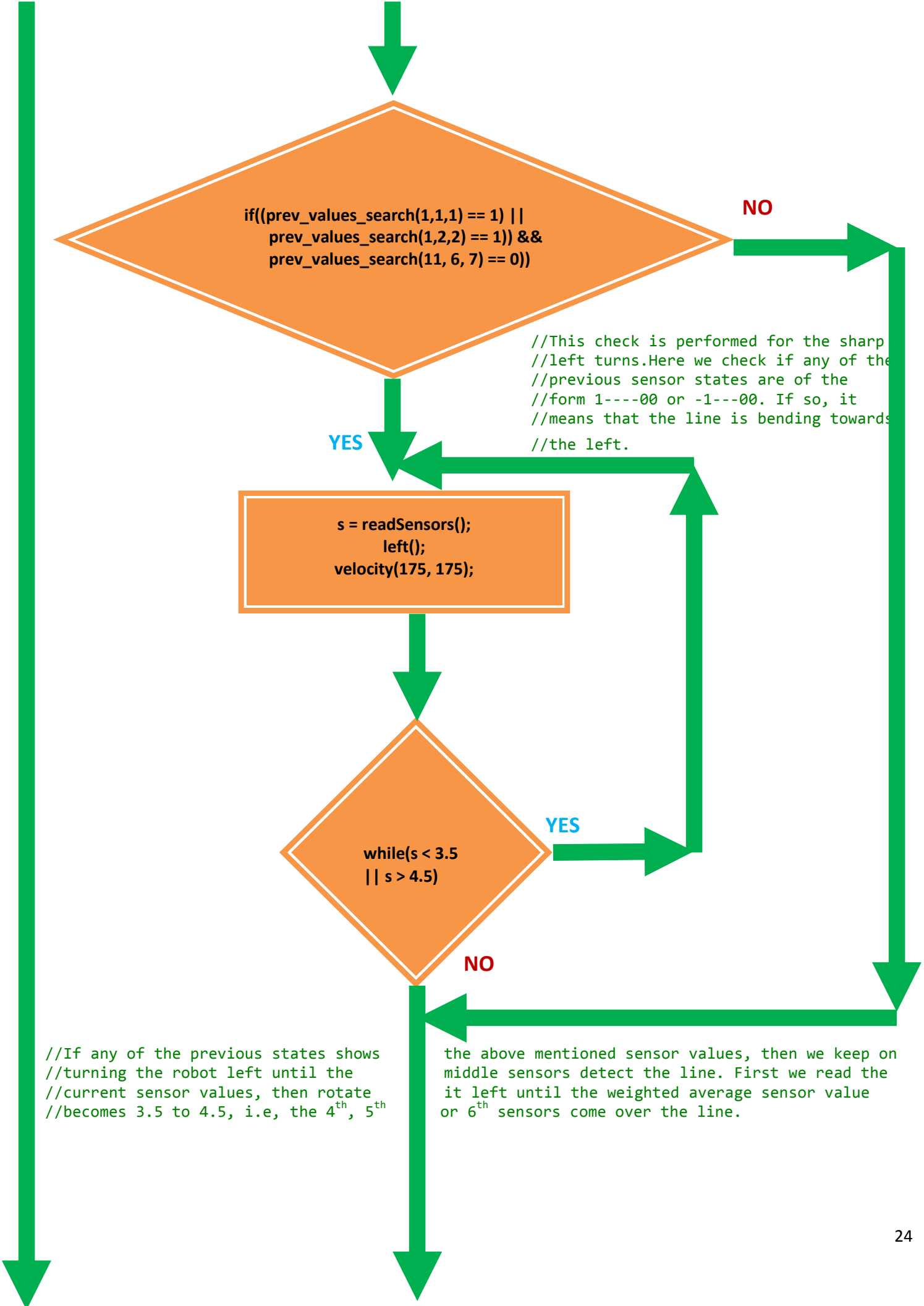
THE MAIN ALGORITHM CORRESPONDING TO THE MAIN BLOCK IN THE EMBEDDED C PROGRAM

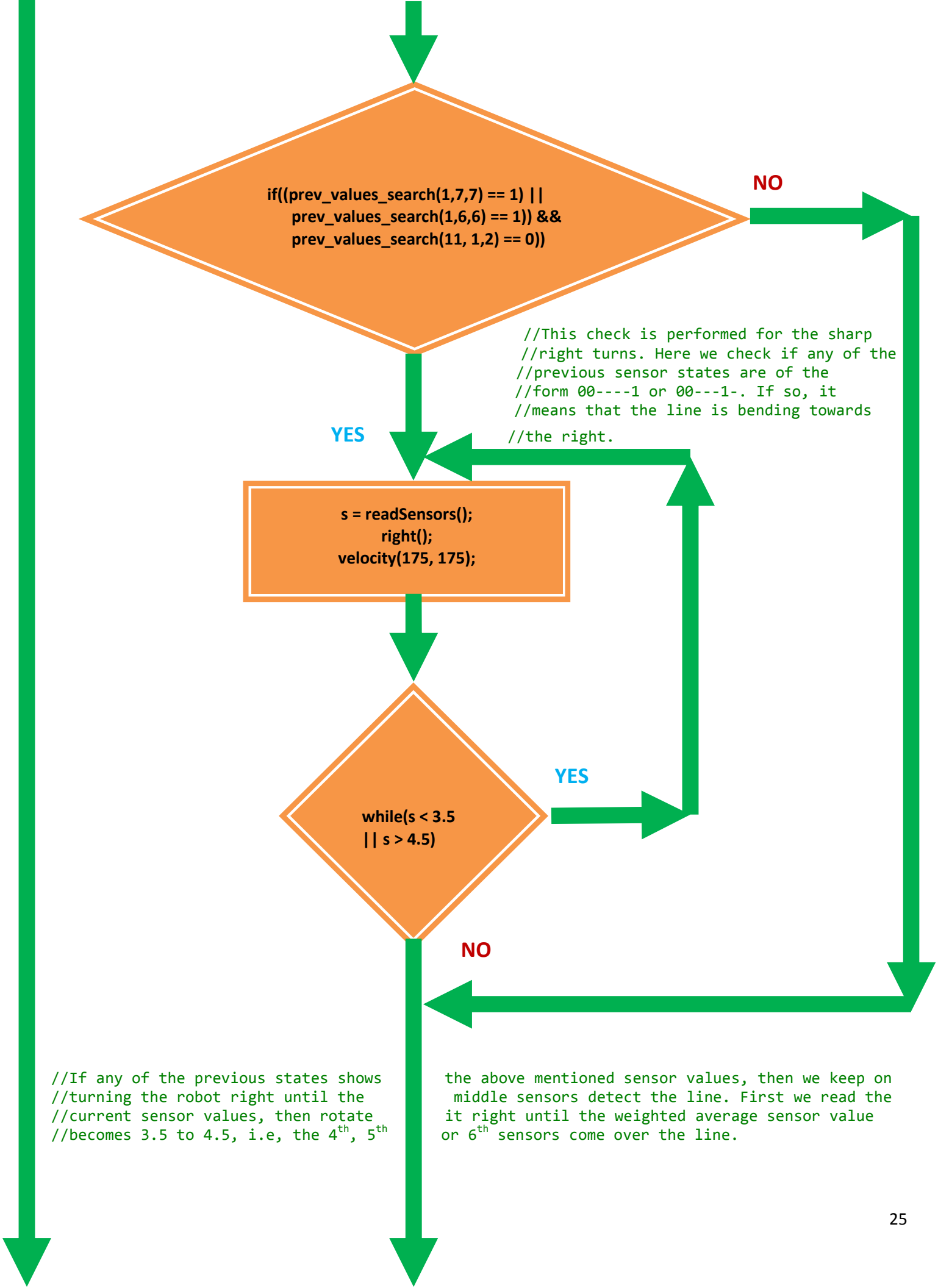


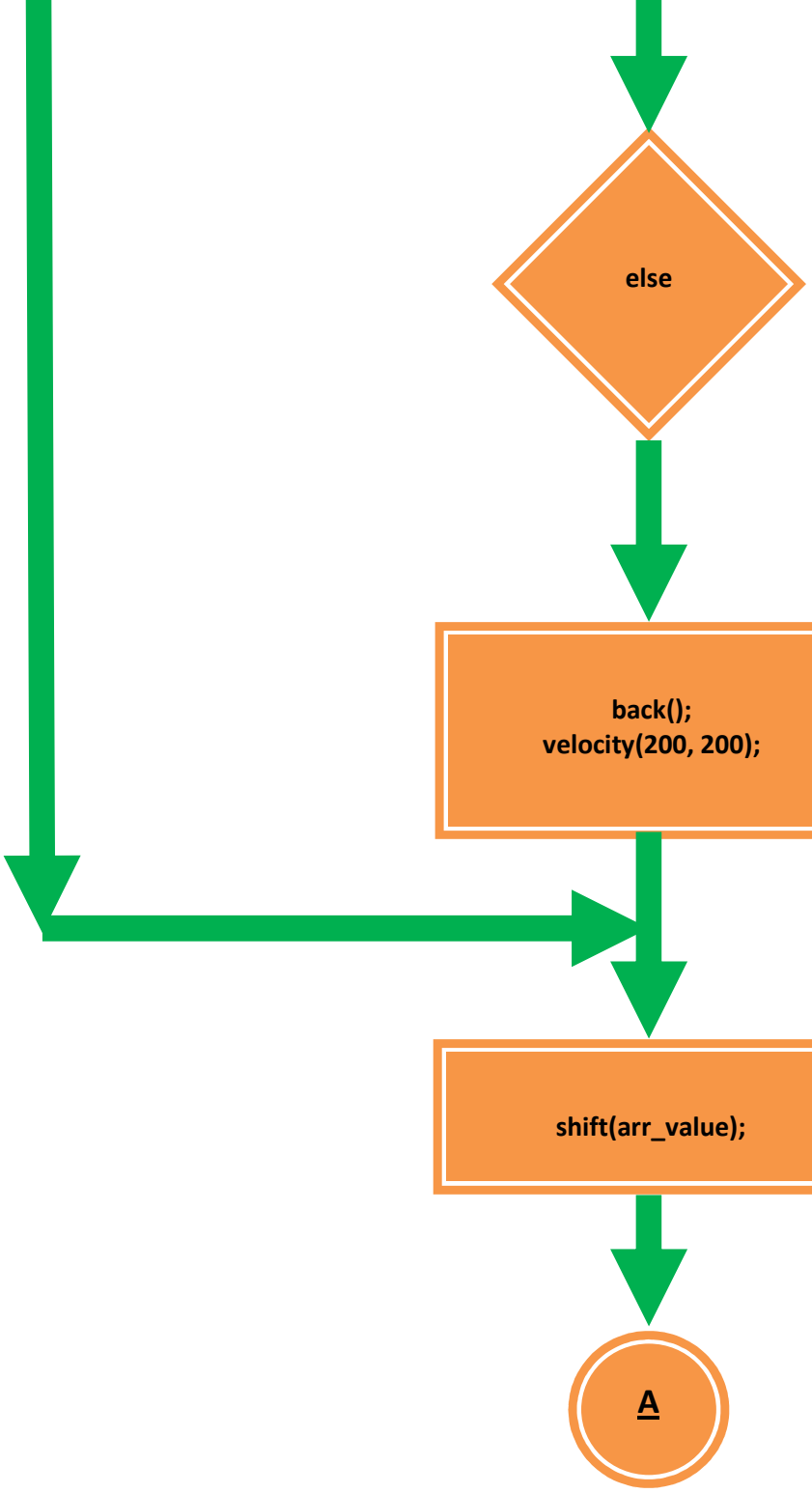

NO
if(line == 0 &
arr_value == 111111 &
prev_values_search(0,1,2) &
prev_values_search(0,6,7))
//It has been found by inspection that the white
//grid starts when the sensor state is 0111111.
//However such a state may occur elsewhere
//also. So to guarantee the efficient
//detection of the start of the
//grid, we have to check previous
//sensor values – to verify whether
//previous 1st, 2nd, 6th and 7th sensors are
//0 each. Note that though the apparent state of
//sensors is 0111111, arr_value == 0111111 cannot be
//written, as the leading 0 specifies a octal constant in C.
//This logic of detection of grid is just indicative and we
//are thinking on whether there are better ways to detect
//the start of the white line grid
YES
gridTraversal();
//algorithm for traversing the
//grid via the optimized path is
//invoked.
//Here we are making the robot
//follow the fixed path without
//considering any PID control upto
//the end of the grid. This method
//is also indicative and we are working on a
//more generalized and autonomous way to cross
//the grid
NO
if((arr_value == 1100011 &&
(prev_values_search(0011100,1,7) == 1)) ||
(arr_value == 1110011 &&
(prev_values_search(0001100,1,7) == 1)) ||
(arr_value == 1100111 &&
(prev_values_search(0011000,1,7) == 1)))
YES
//This is the check inserted for detecting
//black line following algorithm. The
//over the black line, its middle sensor
//extreme left and right sensor values
//crossing the junction, when the bot was
//showed 1 while the extreme left and
//current state of the bot is just a
//states, we notify that the white-black
//accordingly reverse the logic from white
//By complement of a sensor state, we mean
the transition from white line following to
basic principle is that if when the bot is
(3rd, 4th, 5th) values become 0 while the
(1st, 2nd, 6th, 7th) become 1. Now just before
on the white line, the middle sensor values
right sensor values showed 0. So if the
complement of any one of the previous sensor
line junction has just been crossed and
line following to black line following.
that 0's are converted to 1's and 1's to 0's.


//When the bot is following the white line,
//0011100, 0001100 or 0011000 as the 3rd
//following straight line path. If these
//1100011, 1110011 and 1100111 respectively
//mentioned 3 sensor states.
it may have any of these 3 sensor states –
and 5th sensors often come on the line while
states are complemented, they become
So we test the complement condition for the
x1 = 0;
x2 = 1;
line = 1;
//x1 and x2 are just complemented
//so that the logic now becomes
//that of black line following.
//This condition is provided to
//check if the robot, while
//following the black line, has
//reached the loop. At the
//loop, all the 7 sensors
//are on black.
if((line == 1) &&
arr_value==
1111111))
NO
//However, this 1111111 sensor
//state may also arise when the
//bot just encounters the end
//point. The theme specifies a
//bonus if the robot stops here
//and buzzer is turned ON. These
//2 situations are very difficult
//to distinguish as their previous
//sensor states are also more or
//less similar and hence the
//end point check is not yet
//included. However, we are working
//on the methodology to separate
//these 2 conditions and hope to
//find a solution soon.
YES
right();
angle_rotate(50);
//At the beginning of the loop, the bot
//may get confused which path to follow
//So, we slightly move the bot right
//to avoid such deadlock. It can be
//moved left also. The degree of turning
//is calibrated and here we assume it to
//be around 50 degrees


if(arr_value != 0)
//To check if at least 1 of the
//sensors have detected the line,
//be it white or black.
YES
NO (ELSE)
if (s <= 4)
//To check whether the weighted
//mean value of the sensors is less
//than or equal to 4. If it is less
//than 4, the line is towards the
//left, if more than 4, line is
//towards the right.
YES
NO
forward();
velocity(200 – control, 200);
forward();
velocity(200, 200 – control);
//If s <= 4,i.e, line is towards
//the left. So decrease velocity
//of the left motor keeping
//right motor velocity constant.
//The decrease in velocity is
//governed by the control variable
//generated from the PID algorithm.
//Is s > 4,i.e, line is towards
//the right. So decrease velocity
//of the right motor keeping
//left motor velocity constant.
//The decrease in velocity is
//governed by the control variable
//generated from the PID algorithm


if((prev_values_search(1,1,1) == 1) ||
prev_values_search(1,2,2) == 1)) &&
prev_values_search(11, 6, 7) == 0))
NO
//This check is performed for the sharp
//left turns.Here we check if any of the
//previous sensor states are of the
//form 1----00 or -1---00. If so, it
//means that the line is bending towards
//the left.
YES
s = readSensors();
left();
velocity(175, 175);
while(s < 3.5
|| s > 4.5)
YES
NO
//If any of the previous states shows
//turning the robot left until the
//current sensor values, then rotate
//becomes 3.5 to 4.5, i.e, the 4th, 5th
the above mentioned sensor values, then we keep on
middle sensors detect the line. First we read the
it left until the weighted average sensor value
or 6th sensors come over the line.


if((prev_values_search(1,7,7) == 1) ||
prev_values_search(1,6,6) == 1)) &&
prev_values_search(11, 1,2) == 0))
NO
//This check is performed for the sharp
//right turns. Here we check if any of the
//previous sensor states are of the
//form 00----1 or 00---1-. If so, it
//means that the line is bending towards
//the right.
YES
s = readSensors();
right();
velocity(175, 175);
while(s < 3.5
|| s > 4.5)
YES
NO
//If any of the previous states shows
//turning the robot right until the
//current sensor values, then rotate
//becomes 3.5 to 4.5, i.e, the 4th, 5th
the above mentioned sensor values, then we keep on
middle sensors detect the line. First we read the
it right until the weighted average sensor value
or 6th sensors come over the line.




```
//If none of the above conditions
//arise, it means that the robot
//has lost its path.
```

```
//In order to make the bot regain
//the line, we move it back till
//either of the sensors detect the
//line. This is like a recovery
//mechanism.
```

```
//The present sensor state that is
//stored in arr_value is inserted
//into the prev_values array.
```

```
//'A' is a connector to the
//"while(1)" condition before.
//So here the program proceeds
//to the next iteration.
```