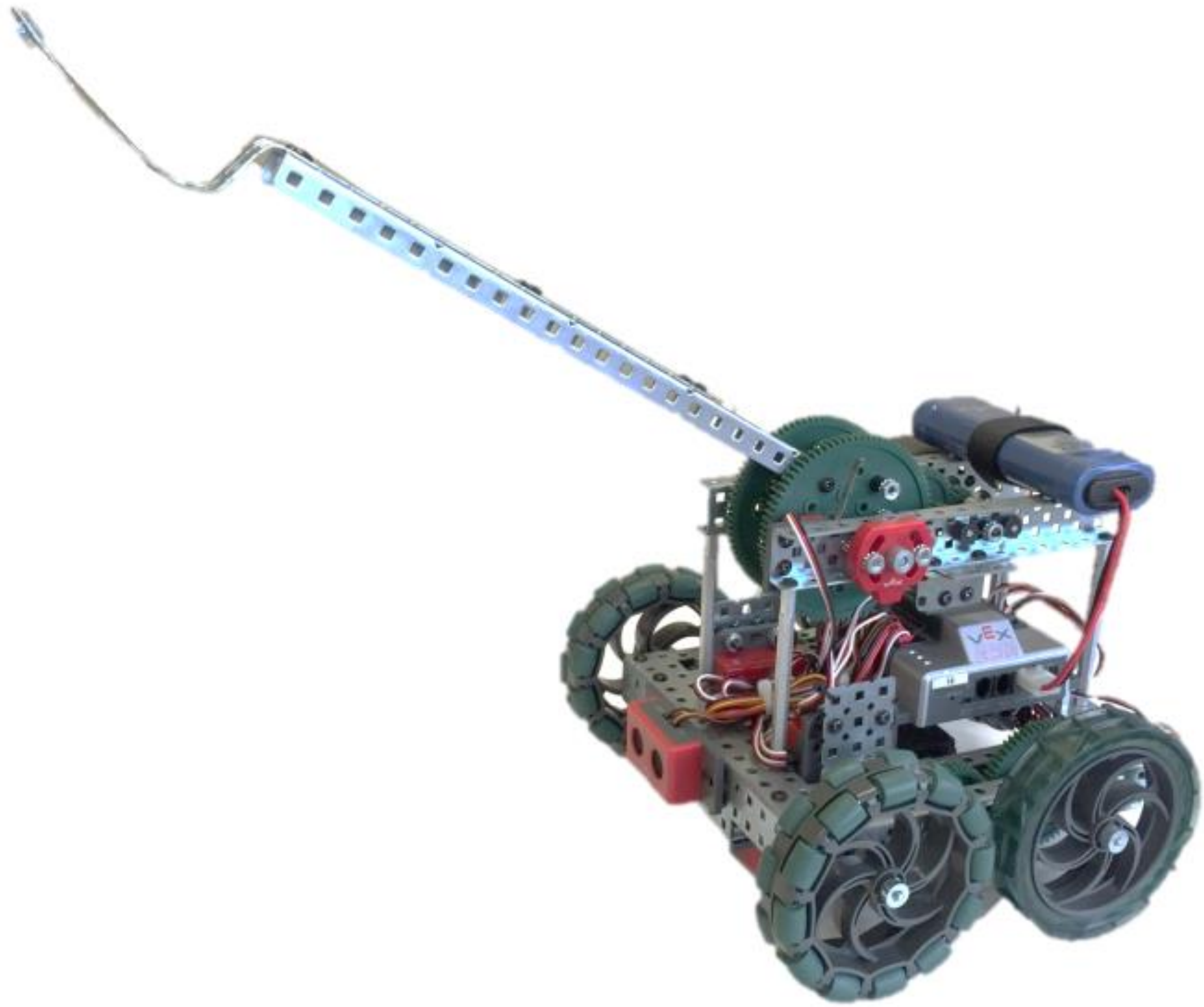
MECHENG 201 Project: Autonomous Warehouse Robot

Group A10; Devdass Krishnan & Shari Masina



1 Introduction

The Autonomous Warehouse Project was created to help the Dairy Industry Cooperative deliver stock around a warehouse due to the rising costs of labour. A scaled down version of the robot used in the warehouse to deliver stock was provided to us to aid in its design. The robot had to be designed ensuring that it would abide by the restrictions such as staying out of the Keep Out Zones, starting & finishing within the Human/Robot interaction zones and completing the task within a specified time limit to avoid exhausting the battery before finishing. The Robot's entire task must be completed autonomously.

2 Solution

The program used to control the Robot's movements was executed after the user had pressed the start button on the Robot – however, as soon as the Robot is tuned on, a support function, `armUp()` was used to raise the arm to its default 'up' position.

The 'main' function would run once per each trip the Robot made from the charging station. Within the 'main' function, many other functions were called upon to execute smaller, more specific parts of the task.

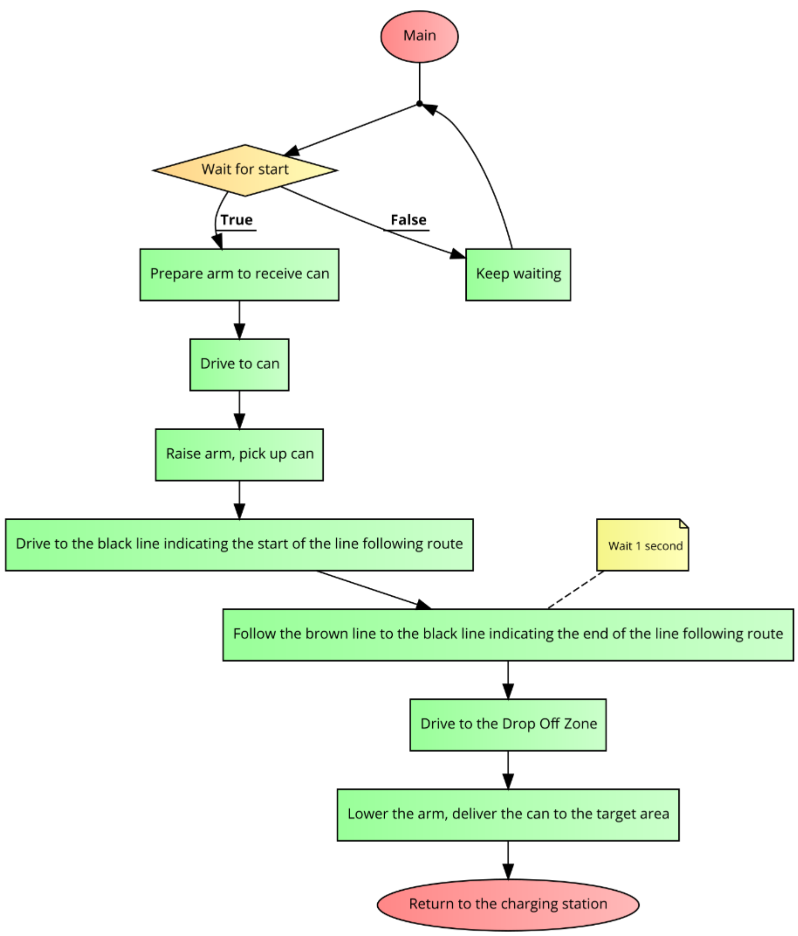


Figure 1: The 'main' function

The flowchart in *Figure 1* shows the order of events during one loop around the warehouse when delivering the can.

To complete the task at hand, we use the robot programming language RobotC to execute a program that includes a series of functions.

To prepare the arm we use the support function `armUp()` to raise the arm to its highest point, followed by the `waitForStart()` function. This function consists of a while loop that checks every 0.5 seconds if the start button has been pressed. Once the start button is pressed, the robot waits 0.5 seconds before operating. This ensures the robot does not move automatically once the power switch is on. We then use the `armDown()` support function as it moves the arm down to its lowest point and resets the arm encoder count.

Once the Robot operator presses the start button, the Robot's arm is lowered to a height that enables the Robot to grasp and raise the can using the `raiseArm()` function.
It then drives directly straight towards the can's 'pick up' location using the `driveStraight()` function but doesn't enter the 'keep out zone'.
The Robot is able to lift the can when the arm is threaded through the can's handle. The arm is then raised as far as possible whilst lifting the can.

A function called `turn_robot()` is used to pivot the robot 90° clockwise so that it faces the line-following section of the warehouse after the Robot reverses a sufficient distance. At this point, the Robot will be a small distance from a black line which indicates the start of the line-following section.

The Robot then drives towards the black line and stops immediately when the light sensors mounted underneath the Robot read a value that corresponds to a calibrated value of 'Black'. A delay of 1 second between the detection of the black line and initiating line-following is used which may signal the beginning of the line-following segment.

The function `linefollower()` is then activated (see figure 2 below) – the Robot drives from the black start line to the longer brown line. The line-following algorithm then drives the Robot at a steady speed along the brown line, around shelving units until it reaches a second black line, indicating the end of the line-following segment.

Once the light sensors have detected the second black line, the `linefollower()` function ends. The Robot then drives straight from the second black line towards the 'drop off' zone. Once the Robot reaches a predetermined distance from the 'drop off' zone, the main arm is lowered, placing the can onto the 'drop off' zone. The Robot's arm is then retracted by reversing away from the can.
Finally, the Robot then pivots 90° clockwise, drives towards the charging station, the main arm is raised fully, pivots 90° counter-clockwise and drives straight into the charging station to complete the run.

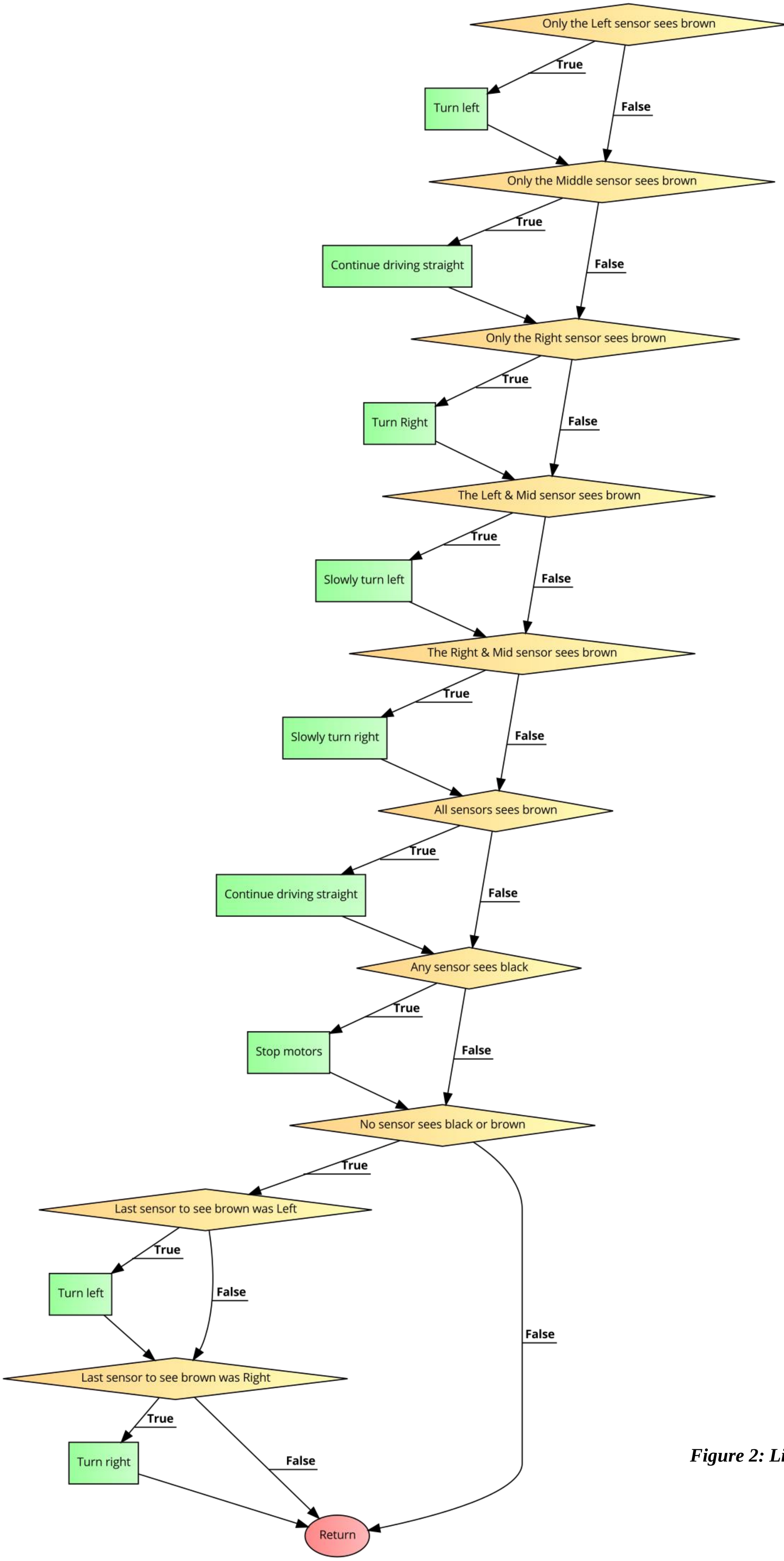


Figure 2: Line following algorithm

2.1 Special features

The function `driveStraight()` has an optional functionality which gives the Robot an option to stop once the light sensors detect a black line or any colour line depending on what the colour value is set to within the function. This is useful when driving towards the starting black line which indicates the start of the line-following segment.

The function also uses 2 different P controllers to control the distance to travel and to ensure the Robot drives straight. The motor powers decreased as the Robot approached the target distance, slowing the Robot down, decreasing the chance of 'overshoot'.

The left wheel's motor differed in delivered power compared to the right wheel's motor. Most likely due to internal frictions. To overcome this problem, a P controller was used. By monitoring the difference between individual wheel encoder counts, extra power was either added or subtracted from one motor (the left motor in this instance) to keep their angular velocities as similar as possible – thus ensuring the Robot travels in a straight line as opposed to an arc.

Loss of traction may occur when the Robot enters the `driveStraight()` function. This is because the power doesn't gradually increase from 0. This problem would affect the distance it would travel as the wheel encoder counts would change when the Robot is 'slipping'. We avoided this problem by implementing an acceleration condition using timers. When the `driveStraight()` function is initiated, the timer is reset. It then checks to see how long ago the function was called and apply a power constant (between 0 and 1.0) to the motors that correspond to the current time. This acceleration control is only required very briefly after the function is called.

3 Performance

On the day of assessment, we were given a total of 6 minutes (two 3-minute slots) to test our solution on the map layout.

3.1 Battery

We noticed during a practice run, our robot was moving slower than normal and realized our issue was the lack of battery. However, a full battery was not sufficient as it made the robot move too fast and therefore further affect our robot's performance. We decided to use a battery that was near full as it made our robot not only move at a reasonable speed, but also slow enough to detect sensor values when line following.

3.2 Starting point

The main factor that had a great impact on the robot's performance was our starting position at the beginning of the run. We noticed that if we placed the robot too far forward on the green patch, the robot would not reverse enough after picking up the can, and therefore after rotating 90 degrees to the right and driving forward, the robot sensors would miss the black line. We also positioned the robot on a slight angle to the left due to our left motor being very slightly more the powerful than the right motor. Applying a P controller `driveStraight()` proved to be quite effective, however there was a very small curve when driving straight for long distances. Every time the angle was slightly inaccurate, the robot would miss the can. This affected our time in our first test slot as we had to restart the robot every time our starting point was inaccurate.

However, in the second test slot we made our starting position more accurate and could run the course more times.

3.3 Line following

Our robot performed the line following function successfully. The steady speed of the robot enabled it to read sensor values effectively and therefore follow brown line to the end every time.

3.4 Can Placement

When the robot placed the can down we noticed that it would land roughly around the same area in the 30 points target on the bottom right side. The can's final position was dependent on the angle the robot is positioned after executing the line following function as the line following algorithm causes the Robot to 'bounce' between light sensors.

3.5 Overall:

We could run the whole course successfully only once in the first 3-minute slot, but could run the course 2 times in the second slot. After our first test slot, we could develop on what needed to be improved and therefore increase accuracy in our second test slot.

4 Improvement

4.1 Accuracy of delivery

- To improve our can placement on the target we can adjust our distance_to_travel input in the `driveStraight()` function so the robot could place the can closer to the target.
- Manually override the `driveStraight()` function and decrease the power of the left motor to prevent the robot driving more on an angle, as we noticed the can would always be placed slightly right of center.

4.2 Time

- Have tasks execute simultaneously, i.e; driving whilst moving the main arm. This will make the robot complete the task much faster.
- Increase the power in the motors during line following as much as possible, however; ensuring that the speed will be slow enough for the sensors to accurately read the colour values of the lines.
- Alter the line following function so that it drives more smoothly without moving side to side, i.e; implementing an edge following algorithm.

4.3 Starting Position

- Trial multiple runs to allow us to tune the kP value of `driveStraight()` so that the Robot drives as straight as possible.
- We could have increased the angle of the robot when turning toward the black line, to reduce the chance of the robot missing the black line.

5 Appendix

5.1 Source code – labTasks.c

```c
#pragma config(I2C_Usage, I2C1, i2cSensors)
#pragma config(Sensor, in1,    lightLeft,      sensorReflection)
#pragma config(Sensor, in2,    lightMid,       sensorReflection)
#pragma config(Sensor, in3,    lightRight,     sensorReflection)
#pragma config(Sensor, dgtl1,  btnStop,          sensorTouch)
#pragma config(Sensor, dgtl2,  btnStart,       sensorTouch)
#pragma config(Sensor, dgtl3,  sonar,          sensorSONAR_mm)
#pragma config(Sensor, dgtl5,  encRight,       sensorQuadEncoder)
#pragma config(Sensor, dgtl7,  encLeft,        sensorQuadEncoder)
#pragma config(Sensor, dgtl9,  LED_Right,      sensorDigitalOut)
#pragma config(Sensor, dgtl10, LED_Left,       sensorDigitalOut)
#pragma config(Sensor, dgtl11, armLimit_low,   sensorTouch)
#pragma config(Sensor, dgtl12, armLimit_high,  sensorTouch)
#pragma config(Sensor, I2C_1,  armEncoder,     sensorQuadEncoderOnI2CPort,    , AutoAssign )
#pragma config(Motor,  port2,           motorArm,      tmotorVex269_MC29, openLoop, reversed, encoderPort, I2C_1)
#pragma config(Motor,  port7,           motorRight,    tmotorVex269_MC29, openLoop)
#pragma config(Motor,  port8,           motorLeft,     tmotorVex269_MC29, openLoop, reversed)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "supportFunctions2017.c"

// -----------  DO NOT MODIFY anything ABOVE this line! ----------- //

// Student 1 - Name: Devdass Krishnan ID: 719113833
// Student 2 - Name: Shari Masina ID: 267183290

// Put your own function prototypes here.
void waitForStart();
float saturate(float input_num, float lower_limit, float upper_limit);
float percentage_to_level(float power_percentage);
void turn_robot(int degrees, int power_percentage, float counts);
void linefollower();
void driveStraight(int distance_to_travel, float kP, int detect);
int countToDistance(int encoder_count);
int detectBlackLine();
void raiseArm(int degrees, int percentagePower);

// Write your own program inside task main().
task main() {

    startTask(checkArm);        // DO NOT DELETE THIS LINE
    startTask(checkButtons);    // DO NOT DELETE THIS LINE
    // DO NOT PUT YOUR CODE BEFORE THIS LINE!!!
    // ------------ Put your own algorithms here ------------

    armUp(40);
    waitForStart();
    //prepare arm
    armDown(40);
    raiseArm(7, 40);
    //travel straight to the pick up
    driveStraight(1040, 0.8, 0);
    //pick up the can
    armUp(40);
    //reverse
    driveStraight(-200, 0.5, 0);
    //turn right
    turn_robot(-85, 40, 3.6893);
    //travel to the black line
    driveStraight(500, 0.8, 1);
    //wait for 1 second
    delay(1000);
    //follow the brown line
    motor[motorLeft] = 30;
    motor[motorRight] = 40;
    delay(600);
```

```
    linefollower();
    //drive straight to the drop off zone
    driveStraight(434, 0.5, 0);
    //drop off can
    raiseArm(-53, 40);
    //reverse a lil
    driveStraight(-200, 0.5, 0);
    //turn right
    turn_robot(-88, 40, 3.6893);
    //drive straight to the finish area
    driveStraight(1308, 0.5, 0);
    turn_robot(90, 40, 3.6893);
    armUp(70);
    driveStraight(430, 0.5, 0);

    // ------------------------------------------------------
    stopAllTasks(); // end of program - stop everything
}

//This function loops until the start button is pressed.
void waitForStart() {
    int result;
    result = SensorValue[btnStart];

    while (result == 0) {
        result = SensorValue[btnStart];
        delay(500); //checks every 0.5 seconds
    }
}
//This function raises or lowers the robot's arm
//Inputs: degrees - degrees to move the arm. percentagePower - the power (in percentage) at which to operate the motors.
void raiseArm(int degrees, int percentagePower) {
    //240.448 encoder counts per encoder shaft revolution. 1:21 between arm and encoder shaft.
    //(240.448/360) = 1 degree encoder shaft turn = 0.6679
    //Ans*21 = 1 degree arm swing = 14.026 encoder counts
    int currentCount = 0;
    SensorValue[armEncoder] = 0;
    float desiredCount = (degrees * 14.026);
    float powerLevel = percentage_to_level(percentagePower); //converts the percentage to a power level for the motor
    int modifier = 1;
    if (degrees < 0) {
        modifier = -1; //reverse the motor direction if degrees is a negative value
    }
    while (currentCount < desiredCount*modifier) {
        motor[motorArm] = modifier*powerLevel; //the use of 'modifier' here ensures that the count will always count upwards
        currentCount = modifier*SensorValue[armEncoder]; //update count
    }
    motor[motorArm] = 0;
}
//This function is a 'helper-function' used with driveStraight(). It is used to detect black lines.
//Outputs: 1 or 0 depending on whether or not any light sensor value rises above 2500.
int detectBlackLine() {
    //function will return '1' if a black line is detected by any of the light sensors
    if (SensorValue[lightLeft] > 2500)  {// 2500 = black colour threshold
        return 1;
    }
    if (SensorValue[lightMid] > 2500)  {
        return 1;
    }
    if (SensorValue[lightRight] > 2500)  {
        return 1;
    }
    else { //if sensors aren't reading 'black', return 0
        return 0;
```

```
    }
}
//This function converts a wheel encoder count to a distance in milimeters.
//Inputs: encoder counts from wheels.
//Outputs: the distance converted from encoder counts.
int countToDistance(int encoder_count) { //converts wheel encoder count to a distance in mm
    int distance = encoder_count * ((103*PI)/600);
    return distance;
}
//This function powers the motor depending on its direction to travel and how far.
//Inputs: distance_to_travel - the distance to travel in millimeters. kP - the P controller constant. detect - 1 or 0 depending on whether or not we want the robot to stop once it detects a black line.
void driveStraight(int distance_to_travel, float kP, int detect) {
    SensorValue[encRight] = 0; //reset encoder to 0
    SensorValue[encLeft] = 0; //reset encoder to 0

    int current_distance = 0;
    int current_count = 0;
    float modifier = 1.0; //used for acceleration part
    int power = 0;
    int enc_error;
    float delta_u, kP_straight = 0.5, error, u = 0;
    int exit = 1;

    clearTimer(T1);
    do {//This block is used to slowly accelerate the robot when it starts driving to avoid loss of traction
        //----------------------------------Acceleration block----------------------------------------
        if (time1[T1] < 333) {
            modifier = 0.333; //slowest
        }
        else if (time1[T1] < 666) {
            modifier = 0.666; //slower
        }
        else if (time1[T1] < 1000) {
            modifier = 1; //normal
        }

        //------------------------------------------------------------------------------------------
        current_count = SensorValue[encRight];
        current_distance = countToDistance(current_count); //Update the distance travelled every loop
        error = distance_to_travel - current_distance;
        u = kP * error; //P control used to slow the robot down when it gets close to its target distance
        power = saturate(u, -50, 50);
        enc_error = SensorValue[encRight] - SensorValue[encLeft];
        delta_u = kP_straight * enc_error; //P controller also used to ensure the robot drives straight
        //------------ optional functionality to stop moving once it detects a black line ------------
        if (detect == 1) {
            if (detectBlackLine() == 1) {
                motor [ motorLeft ] = 0;
                motor [ motorRight ] = 0;
                return;
            }
        }

        //------------------------------------------------------------------------------------------
        motor [ motorLeft ] = modifier * (power + delta_u);
        motor [ motorRight ] = modifier * power;

        if ((error < 40) && (error > -40)) { //TOLERANCE
```

```
            motor [ motorLeft ] = 0;
            motor [ motorRight ] = 0;
            exit = 0;
            return;
        }
    } while(exit);
}

//This function used to keep assigned powers within their limits
float saturate(float input_num, float lower_limit, float upper_limit){
    float output_num;
    float temp;

    if (lower_limit > upper_limit){
        temp = upper_limit;
        upper_limit = lower_limit;
        lower_limit = temp;
    }
    if (input_num > upper_limit) {
        output_num = upper_limit;
    }
    else if (input_num < lower_limit) {
        output_num = lower_limit;
    }
    else {
        output_num = input_num;
    }
    return output_num;
}
//This function converts a percentage into a level that the motors are able to use,
//Inputs: power_percentage - a percentage to be converted into a power level
//Outputs: a power level that will be passed to the motors.
float percentage_to_level(float power_percentage) {
    // Assuming the power level is allowed between -127 and +127
    // This function calls upon the saturate function
    float power_level = ((power_percentage / 100) * 127);
    power_level = saturate(power_level, -127, 127);
    return power_level;
}
//This function uses the wheel encoders to rotate the robot precisely a specific angle.
//Inputs: degrees - ammount to turn the robot body by. power_percentage - the power at which to operate the motors. counts - the number of counts that correspond to a 1 degree turn.
void turn_robot(int degrees, int power_percentage, float counts) {
    int modifier = 1;
    if (degrees < 1) { // -ve degrees = CW turns and -ve degrees = CCW turns.
            modifier = -1; // Will change to -1 if the user specifies a CW turn.
        }
    SensorValue[encRight] = 0; //reset encoder to 0
    SensorValue[encLeft] = 0; //reset encoder to 0
    float current_right_count = 0;
    float current_left_count = 0;
    float power_level = percentage_to_level(power_percentage);
    float right_count = degrees * counts * modifier; // A 1 degree robot turn corresponds to 3.6893 encoder counts.
    float left_count =  right_count; // By default, the left wheel will be turning backwards to create a CCW turn - so the encoder has to be counting backwards/opposite to right encoder.
    //int modifier = 1; // Will change to -1 if the user specifies a CW turn.
    int right_on = 0; // Is the right motor on?
    int left_on = 0; // Is the left motor on?

    motor [ motorLeft ] = -1 * power_level * modifier; // Multiply by -1 to create a CCW turn by default.
    left_on = 1;
    motor [ motorRight ] = power_level * modifier;
    right_on = 1;

    // The following code stops each motor once its encoder reaches its determined count.
```

```
    while ((right_on == 1) || (left_on == 1)) {
        // Update counts
        current_right_count = SensorValue[encRight] * modifier;
        current_left_count =  SensorValue[encLeft] * -1 * modifier; //left will be counting down as it is a CCW turn. We multiply by -1 so it counts upwards.
        // modifier is used to reverse this when the right encoder counts down during a CW turn.

        if (current_right_count >= right_count) {
            motor [ motorRight ] = 0; // stop right motor.
            right_on = 0;
        }
        if (current_left_count >= left_count) {
            motor [ motorLeft ] = 0; // stop left motor.
            left_on = 0;
        }
    }
}
//This function uses a basic line following algorithm to follow a brown line and stop once it sees a black line.
void linefollower() {
    //These are the ranges for the colour values
    int blueLow = 400, blueHigh = 1500;
    int brownLow = 1500, brownHigh = 2450;
    int blackLow = 2550, blackHigh = 4100;
    float modifier = 0.8; //to control the speed of the robot.
    int memory = 0; // Last sensor to see the line, 1 (left), 2 (mid), 3 (right)
    do {
    if ((SensorValue[lightLeft] > brownLow) && (SensorValue[lightMid] < blueHigh) && (SensorValue[lightRight] < blueHigh)) { //Only left sees black
        // turn left
        motor [ motorLeft ] = -30*modifier;
        motor [ motorRight ] = 40*modifier;
        memory = 1;
    }
    if ((SensorValue[lightLeft] < blueHigh) && (SensorValue[lightMid] > brownLow) && (SensorValue[lightRight] < blueHigh)) { //Only middle sees black
        // go straight
        motor [ motorLeft ] = 40*modifier;
        motor [ motorRight ] = 40*modifier;
        memory = 2;
    }
    if ((SensorValue[lightLeft] < blueHigh) && (SensorValue[lightMid] < blueHigh) && (SensorValue[lightRight] > brownLow)) { //Only right sees black
        // turn right
        motor [ motorLeft ] = 40*modifier;
        motor [ motorRight ] = -30*modifier;
        memory = 3;
    }
    if ((SensorValue[lightLeft] > brownLow) && (SensorValue[lightMid] > brownLow) && (SensorValue[lightRight] < blueHigh)) { //Left AND mid sees black
        // slowly turn left
        motor [ motorLeft ] = 30*modifier;
        motor [ motorRight ] = 40*modifier;
        memory = 1;
    }
    if ((SensorValue[lightLeft] < blueHigh) && (SensorValue[lightMid] > brownLow) && (SensorValue[lightRight] > brownLow)) { //Right AND mid sees black
        // slowly turn right
        motor [ motorLeft ] = 40;
        motor [ motorRight ] = 30;
        memory = 3;
    }
    if ((SensorValue[lightLeft] > brownLow) && (SensorValue[lightMid] > brownLow) && (SensorValue[lightRight] > brownLow)) { //ALL sees brown
        // go straight
        motor [ motorLeft ] = 40*modifier;
        motor [ motorRight ] = 40*modifier;
        memory = 2;
```

```
    }
    if ((SensorValue[lightLeft] > blackLow) || (SensorValue[lightMid] > blackLow) || (SensorValue[lightRight] > blackLow)) { //If any see black, stop
        // stop
        motor [ motorLeft ] = 0;
        motor [ motorRight ] = 0;
        return;
    }
    while ((SensorValue[lightLeft] < blueHigh) && (SensorValue[lightMid] < blueHigh) && (SensorValue[lightRight] < blueHigh)) { //While NONE of them see black
        if (memory == 1) { //If the last sensor to see it was the left one...  || memory == 2
            // Turn towards the left.
            motor [ motorLeft ] = -30*modifier;
            motor [ motorRight ] = 40*modifier;
        }
        else if (memory == 3) { //If the last sensor to see it was the right one...
            // Turn towards the right.
            motor [ motorLeft ] = 40*modifier;
            motor [ motorRight ] = -30*modifier;
        }
    }
    } while(1);
}
```