Project Report

ECE 556 - Mechatronics

Dheeraj Vemula (#200178808)
Ojas Barve (#200213803)
Vinod K. Singla (#200149761)

December 7, 2017

TABLE OF CONTENTS

1 Introduction

The objective of this project is to design and implement a PID control for a Lego Mindstorms EV3 Unmanned Vehicle (UV) for different set of tasks. The Tasks laid out are namely:

1. **Path Tracking:**
   Implement a PID control for the UV to move from point A with coordinate (xA, yA) to Point B (xB, yB) as fast as possible by tracking a given path r;

2. **Parking:**
   Park as close as possible to the block (Point B) without touching it;

3. **Platooning:**
   Implement a PID control to regulate the UV speed to follow a moving UV in the front and maintain a constant distance.

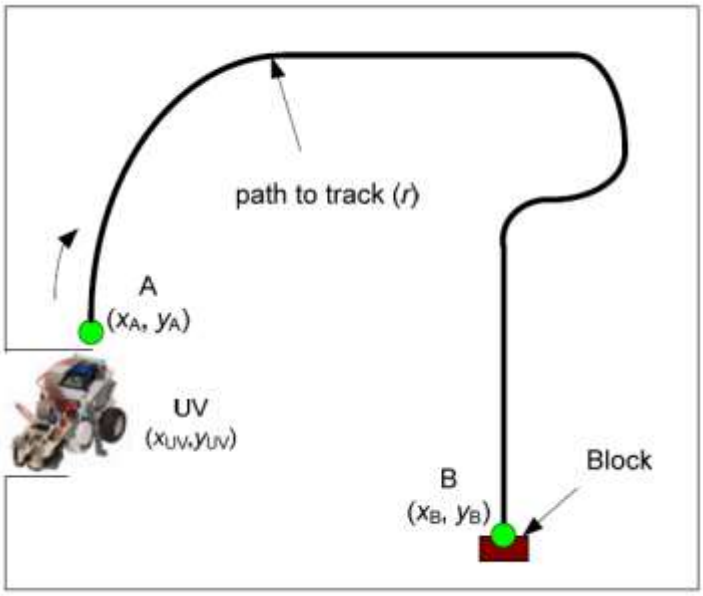


Figure 1. Shows an example layout of the track involved.

2 Project Setup

A major portion of the project was implementing the light sensor and ultrasonic sensor circuits as per the labs carried out throughout the semester. The sensors were interfaced with MATLAB using a trick circuit which made the EV3 detect the sensors as touch sensors. The fooling circuit is shown in Figure 2. Each of the individual sensor circuits are discussed below.

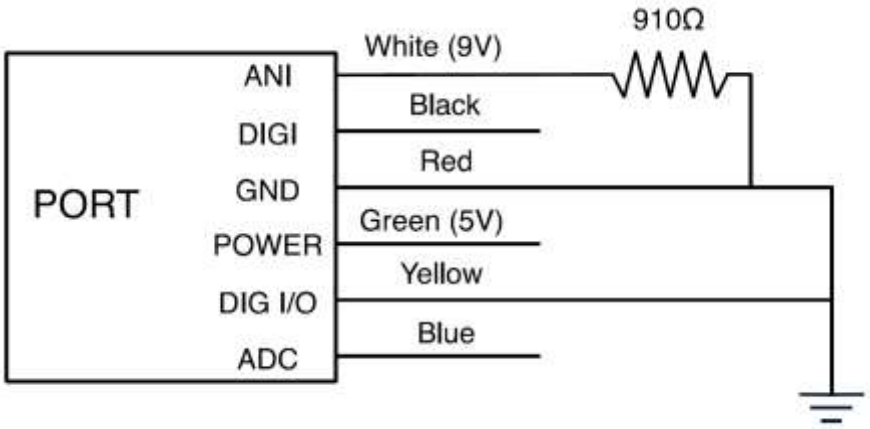


Figure 2. Circuit required to trick the EV3 to interface the sensors

2.1 Ultrasonic sensor

The Ultrasonic sensor used in the project was SRF04. We designed a IC 555 based astable multi-vibrator circuit. Below Figure 3(a). is the circuit setup on a bread board. The astable multi-vibrator circuit output was used to trigger the ultrasonic sensor. The echo pulses obtained from the ultrasonic sensor were fed to the EV3 brick. We implemented an analog as well as a digital filter to remove the noise from the system. Figure 3(b) shows the circuit for the implementation.

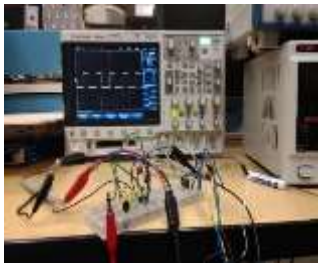


Figure 3 a)

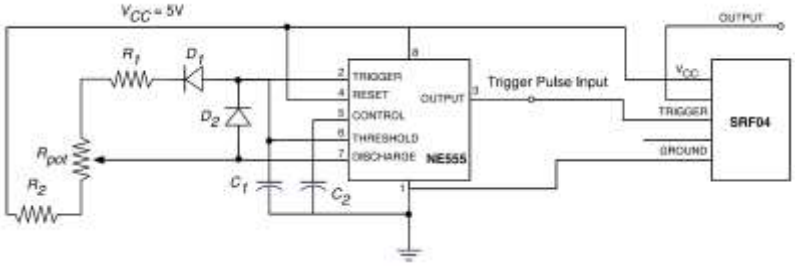


Figure 3 b)

Note the timer circuit we are using has one, 1 kilo-ohm resistance and one, 1 Mega Ohm Potentiometer. The aim is to keep on time of the trigger pulse to be greater than 10 micro seconds and the overall time period to be nearly 36 milliseconds. The reason to implement a potentiometer is to make it easy to adjust the duty cycle to our needs. The formulas used were:

$$t_1(HIGH) = 0.67 * R_A * C_1 \qquad t_2(LOW) = 0.67 * R_B * C_1$$

$$Period \equiv T = t_1 + t_2 \qquad Frequency \equiv f = \frac{1}{T}$$

In addition, the ultrasonic sensor had a low-pass filter (LPF) so that high frequency noise from the output signal can be removed. The Resistor and Capacitor values used were 150 kilo-ohm and 1µF. The cutoff frequency for the low pass filter implemented was 1 Hz. Figure 4 shows analog low pass filter outputs on an oscilloscope.

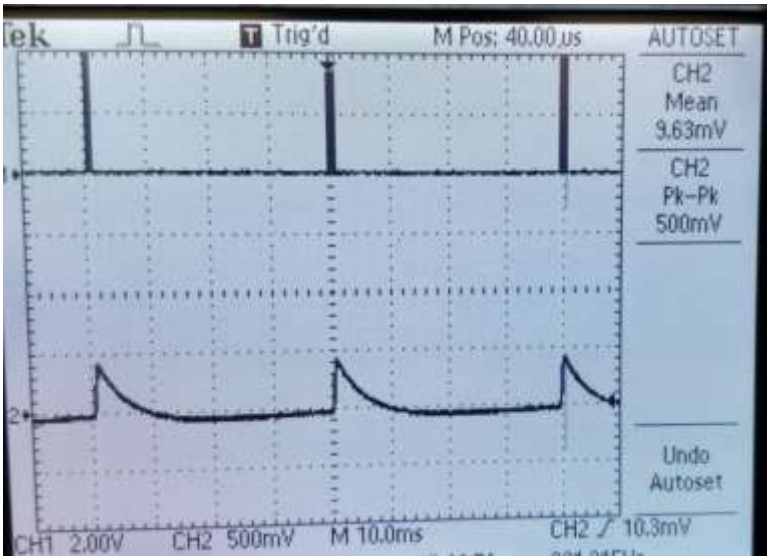


Figure 4. Low pass filter output

To further decrease noise in the output signal, we tested two additional types of digital filters. We performed mean and median filtering on the acquired data from the ultrasonic sensor and found that there was a tradeoff between accuracy and delay. The accuracy was increasing with increased number of samples used for filtering. However, acquiring more number of samples resulted in an increased delay. Also, mean filtering approach worked better than the median filtering approach overall. We calculated the mean values (moving averages) over the last 4 values. Figure 5 shows mean and median filtering with moving average for past 4 values.

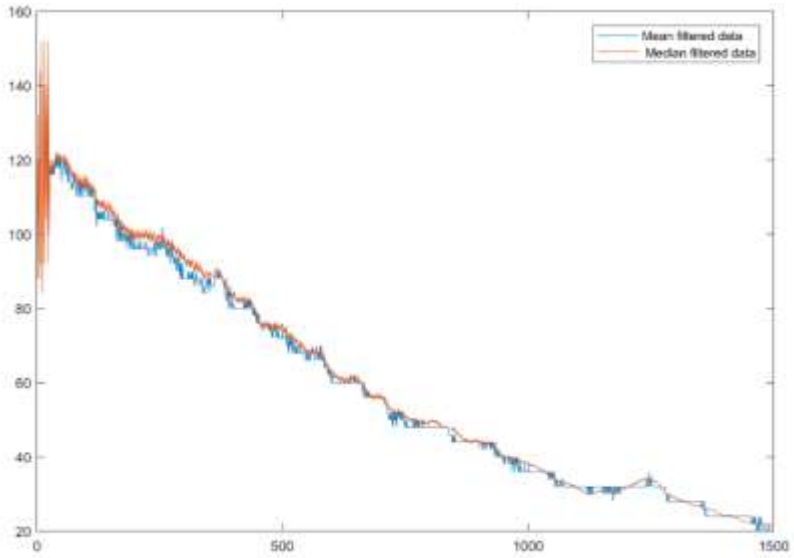


Figure 5. Mean and median filtering with moving average for past 4 values

2.2 Light sensor

For the light sensor circuit an OMRON EE-SF5B sensor was utilized. Figure 6. shows the circuit implementation.

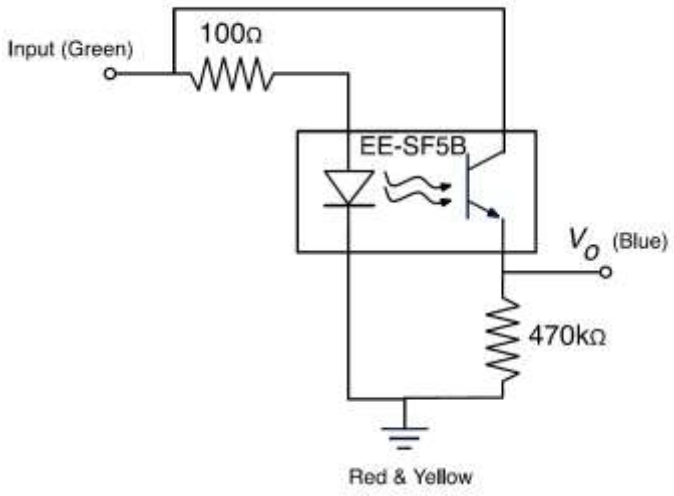


Figure 6. Light sensor circuit

As in case of ultrasonic sensor, a mean filtering approach with the mean value calculated for the last 12 values was utilized for removing noise from sensor output. Figure 7 shows mean filtering with moving average for left and right sensors.

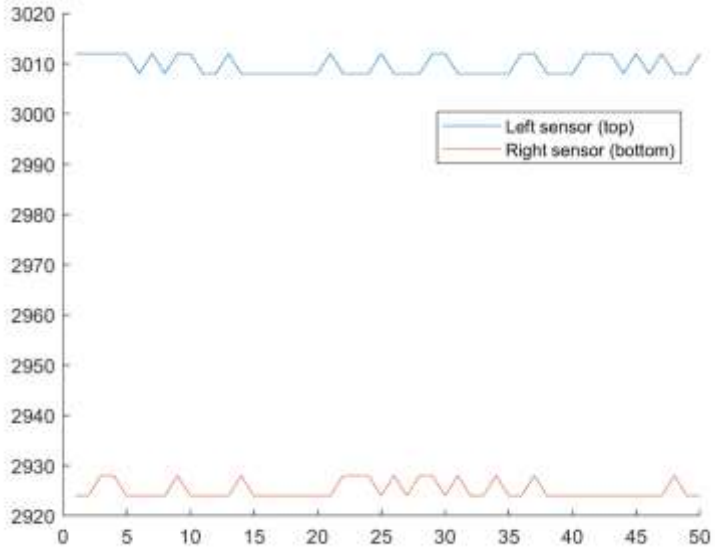


Figure 7. Mean filtering with moving average for left and right sensors.

To record the raw sensor reading for calculating tracking and smoothness error, the right light sensor was utilized. The configuration of light sensors is given in Figure 8 below.

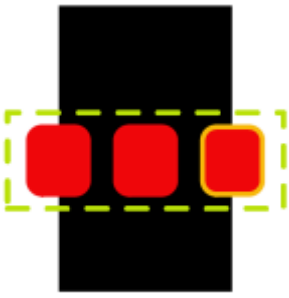


Figure 8. Configuration of light sensors

In the sections that follow, we describe the algorithm and code implementations.

3 Algorithm design and code implementation

The goal of the project was to achieve 3 major tasks using the UV that was built and programmed during the project. These were:

1. Path Tracking
2. Collision Avoidance
3. Platooning in a straight line

Each task presented unique challenges that were addressed using different algorithm strategies. Apart from the tasks themselves, there were several constraints in terms of data acquisition, data processing that presented additional challenges for the algorithm design. In this section, each tasks challenges w.r.t algorithm design and the strategies used to address them are discussed.

- **Path Tracking**

In this task, the UV had to track a path that consisted combination of straight lines, turns of different radii. The design of the bot consisted of 3 light sensors, placed at the front of the UV, that are used for path tracking. There are two sensors on the either edge of the track and one edge on the center of the track. The motion of the UV was controlled by controlling two motors connected to the left and right wheels of the UV. Straight line motion is achieved by setting equal speed on both the motor. Turning is achieved by differential speed between the motors, described by the strategy below:

For left turn, decrease the left motor speed and increase the right motor speed
For right turn, decrease the right motor speed and increase the left motor speed

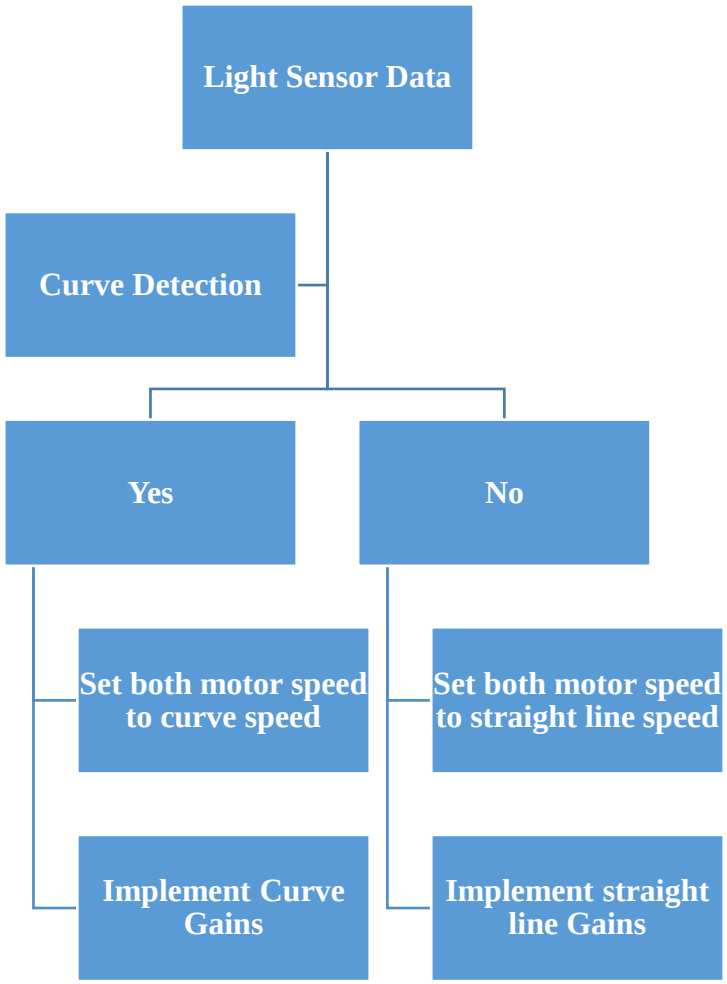
A Proportional – Derivative control was used to control the speed of the individual motors whose input was the error from the light sensors. The error is defined by the difference in the desired value and the measured error. The code shown below implements this approach.

```
% Calculate errors for control
    l_err = (rli_des - rli_l); % current iteration error
    l_err_o = (rli_des - rli_l_o); % previous iteration error

    r_err = (rli_des - rli_r); % current iteration error
    r_err_o = (rli_des - rli_r_o); % previous iteration error

    lc_out = kp*(l_err) + kd*(l_err - l_err_o)/T; % control action
    rc_out = kp*(r_err) + kd*(r_err - r_err_o)/T; % control action
```

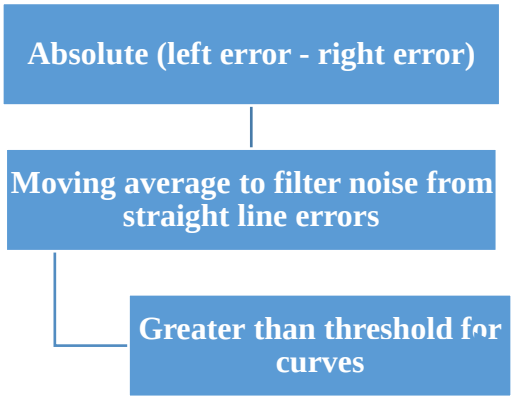
```
    lmotor.Speed = v + lc_out - rc_out; % apply control
    rmotor.Speed = v - lc_out + rc_out; % apply control

    rli_l_o = rli_l; % reassign value for next iteration
    rli_r_o = rli_r; % reassign value for next iteration
```

- **Gain and Speed Scheduling**

The major challenge in control was finding the gains and speed setting that would be common to both straight line tracking and path tracking. This challenge was addressed using gain scheduling – detecting curves and scheduling gains and speeds specific to curves and straight lines. The logic for gain scheduling is as follows:



Curve detection was implemented using the sensor data from both the edge sensors. The logic for curve detections is as follows:



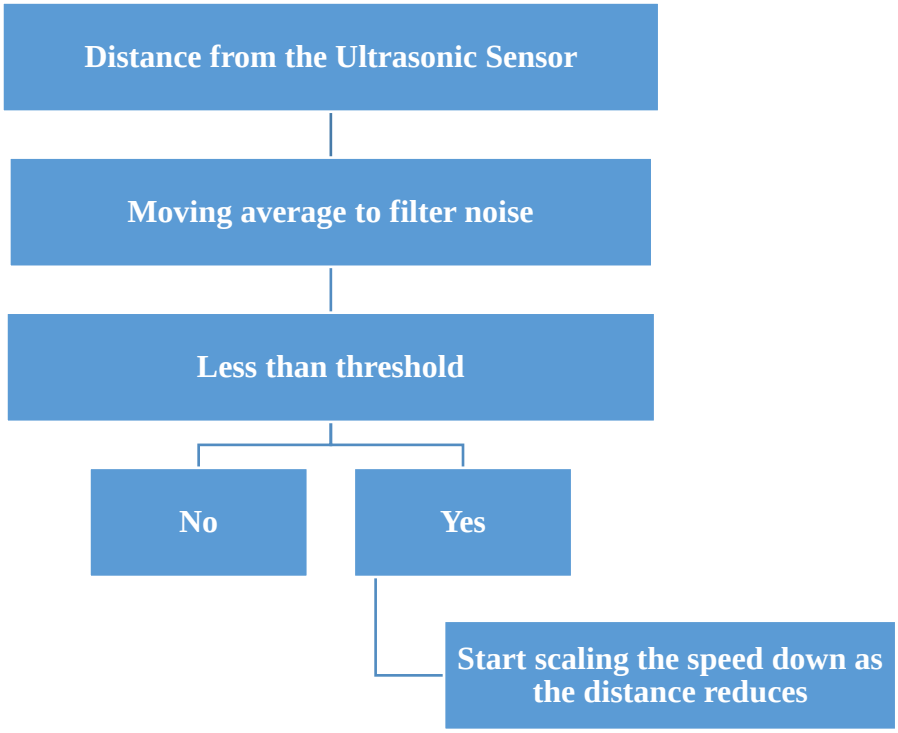
The code shown below implements this approach.

```
% Gain Scheduling based on curved or straight path
 if i > 12 % number of iteration found by trial & error
     mv_err(i) = mean(net_err(i - 12:i));
     if abs(mv_err(i)) > 55 % error value based on (0.5 to 1) feet curve
         kt = 0.4;
         kp = 0.7*kt;
         kd = 0.02*kt;
         v = vc;
     else
         kt = 0.3;
         kp = 0.7*kt;
         kd = 0.0*kt;
         v = vs;
     end
 end
```

- **Collision Detection**

In this task the UV had to stop before colliding with an obstacle as close as possible. The obstacle was detected using an ultrasonic sensor placed on the face of the UV. The major challenge in this task was coming to a sudden stop. Due to the inertia of the UV (mass and velocity before braking) there was an over shoot even after stopping motors.

The strategy was to slow down gradually when an object was detected at a distance and stop as close as possible. The logic that was implemented was as follows:



The code shown below implements this approach.

```
% current distance of object from ev3 in cm
    dist_u = 3  + ultra_scale*(ultra_mv(i) - ultra_raw(1));

    %  Slow EV speed and stop based on the ultrasonic sensor reading
    if dist_u < 20
        vc = (dist_u - 15)*(vc - 20)/45 + 20;
        vs = (dist_u - 15)*(vs - 20)/45 + 20;
        if 5 > dist_u && 2 < dist_u
            disp('stop')
            lmotor.stop
            rmotor.stop
            break
        end
    else
        vc = 45;
        vs = 65;
    end
```

- **Platooning on a Straight Line**

In this task, the UV is supposed to follow another UV (speeds ranging from 0-75% of maximum) while maintaining a constant distance of 20 cm. The distance from lead UV was measured using the ultrasonic sensor.

The strategy was to implement a scaling of velocity between the maximum and minimum as distance varies between 30 cm and 20 cm. Also, for stopping on red-mark, a threshold was applied on the light sensor reading to stop when a red-patch was detected on the track. The code shown below implements this approach.

```
%  follow based on separation error
sep_err = sep_des - dist_u;

% Calculate errors for path tracking
l_err = (rli_des - rli_l);
l_err_o = (rli_des - rli_l_o);

r_err = (rli_des - rli_r); % current iteration error
r_err_o = (rli_des - rli_r_o); % previous iteration error

lc_out = kp*(l_err) + kd*(l_err - l_err_o)/T; % control action
rc_out = kp*(r_err) + kd*(r_err - r_err_o)/T; % control action

v = min(-sep_err*vs/10, vs);

lmotor.Speed = max(0, v + lc_out - rc_out);
rmotor.Speed = max(0, v - lc_out + rc_out);

rli_l_o = rli_l;
rli_r_o = rli_r;
sep_err_o = sep_err;
dist_u_o = dist_u;

%  Stop based on the center sensor
if rli_stop_u > rli_m && rli_stop_l < rli_m
    disp('stop')
    lmotor.stop
    rmotor.stop
    break
end
```

4 Results and Discussion

Using the algorithms and code implementation discussed in the previous sections, the EV3 is able to track a complex test path created by a combination of 6 track modules (Figure 9) provided in the project description.

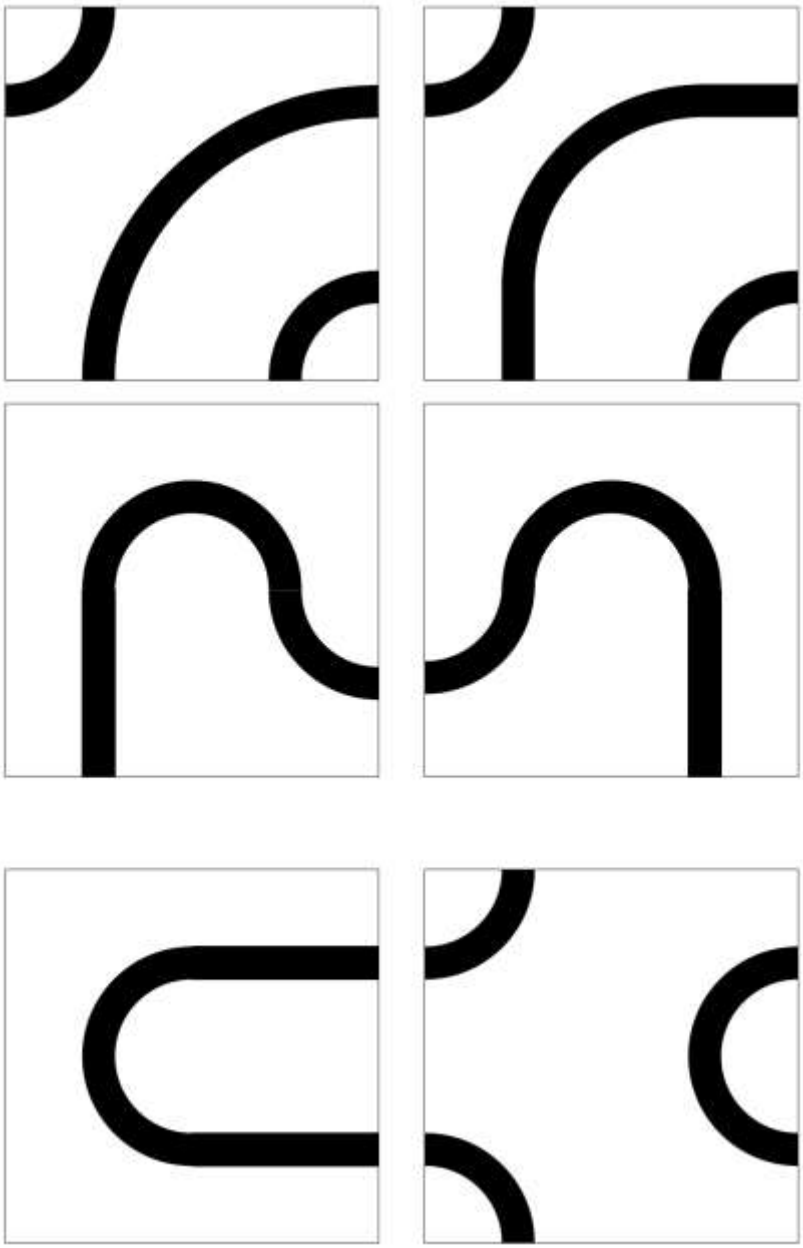


Figure 9. Track modules used to construct a test track

The results of each task are given in table below:

Task	Best Run-time (sec)	Meets stated requirements (Yes / No)
Path tracking	53.15	Yes, tracks path
Collision avoidance	NA	Yes, stops without colliding
Platooning	11.53	Yes, follows at 20 cm separation

Although the bot is able to perform all the tasks with repeatability some issues do exist which can be addressed in future work. These are highlighted below:

- The EV3 has some oscillations while tracking the path
- The total time taken to complete the track modules is relatively large
- The separation gap during platooning has some variations from desired gap.

These issues can be addressed in future using:

- An integral control can further smoothen out path tracking.
- Having a stricter control can help the bot to travel at increased set speed reducing total time required for path tracking.
- The platooning strategy can be modified to include a Kalman filter to estimate the tracked speeds in real time which can considerably increase performance.

5 References

[1] 2017 Fall ECE 456/556 project description and grading scheme

[2] ECE 556 lecture slides, Fall 2017

[3] MATLAB help, Web: https://www.mathworks.com/help/matlab/

[4] SR-04 datasheet, Web: https://www.robot-electronics.co.uk/htm/srf04tech.htm

[5] EE-SF5B datasheet, Web: http://www.mouser.com/ds/2/307/en-ee_sf5-1221359.pdf