EE3 Final Project Report

By Yolanda Chen & Ramtin Sharafoleslami

Fall 2020

1. Induction and Background

The main goal of this project is to implement code for a path-following car using RSLK. The car should be able to follow a predetermined black curved path, turn 180º around when it reaches the end of the path, and move back to the starting point and stop moving completely. Bonus credit was offered to students whose project car completed the above requirement within 11 seconds.

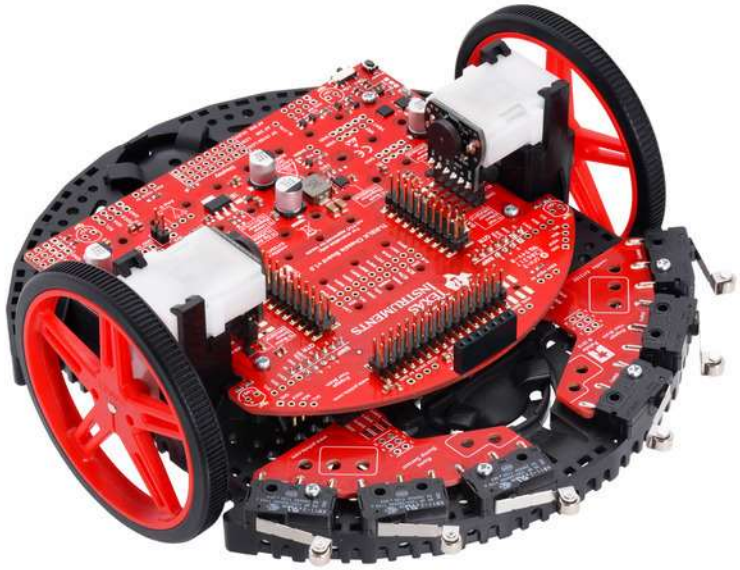


Figure 1. Natcar with MSP432. Source: Pololu
https://www.pololu.com/product/3670/pictures

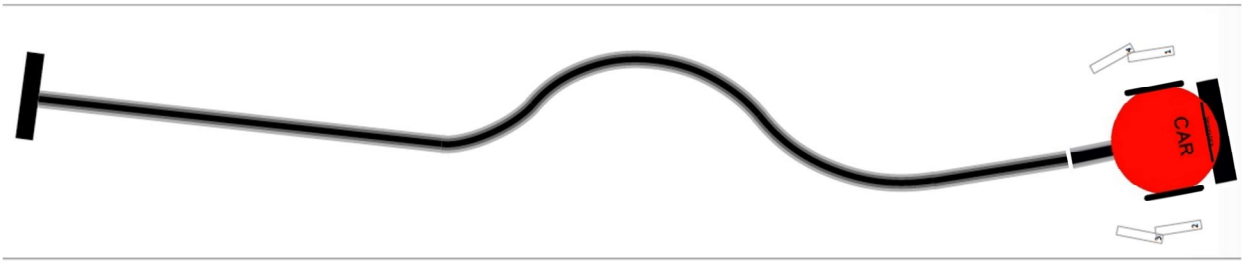


Figure 2. Predetermined black curved path. Source: Dr. Dennis Briggs [2] – UCLA

The implementation of the code utilized some form of PID control to change the motor speed according to the different values read from the IR sensors underneath the project car. The IR sensors attached to the project car are phototransistors.

PID control stands for proportional integral derivative control. It is a control loop mechanism that utilizes feedback. The goal of a PID is to find the error value between the desired setpoint to the current measured value and apply a correction algorithm based on proportional, integral, and derivative terms (hence, the name PID control) [3]. In our project, PID control enables the car to follow the predetermined black path. Our PID controller constantly calculates the error value (the desired setpoint is when the car is centered on the track and the current measured value is the position of the car at a given time) and applies our own correction algorithm to bring the car back on track. Our correction algorithm will be discussed in detail later on in this report.

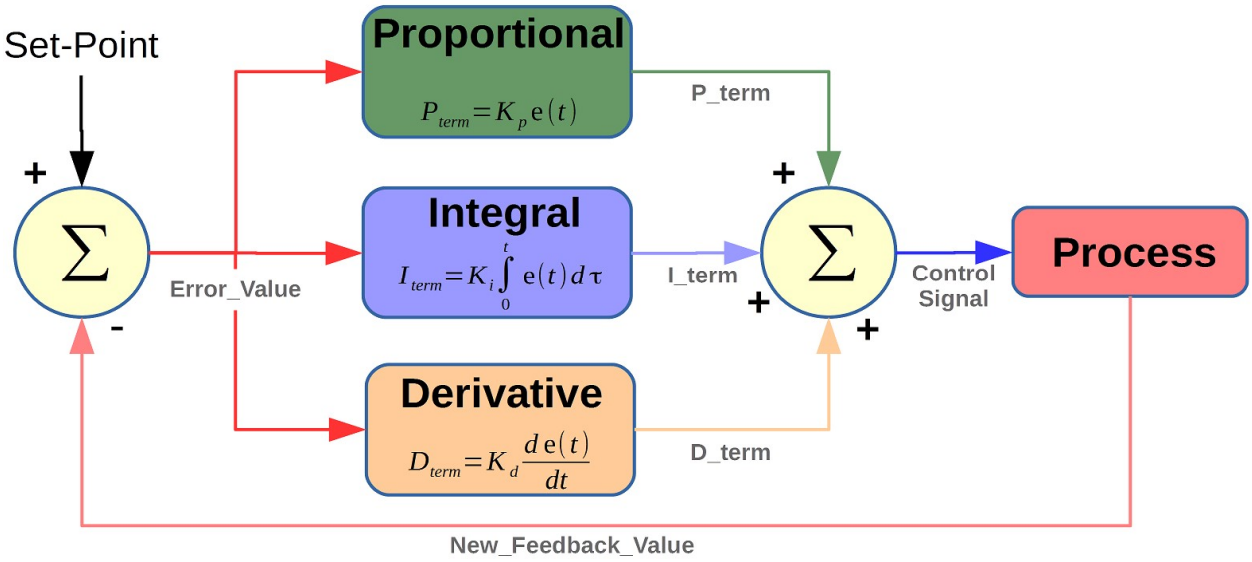


Figure 3. Diagram of a PID controller – Source: Microcontrollers Lab [3]
https://microcontrollerslab.com/pid-controller-implementation-using-arduino/

The circuit responsible for path detection relies on a phototransistor. Phototransistor is a type of transistor that displays a change of resistance based on the light intensity it receives. Figure 4 is a basic schematic of a phototransistor in a circuit. The combination of a phototransistor and a capacitor resembles an RC circuit with a variable resistance and therefore a variable time constant. This change in time constant is interpreted as dark or light (Figure 5) and enables the car to detect the path.

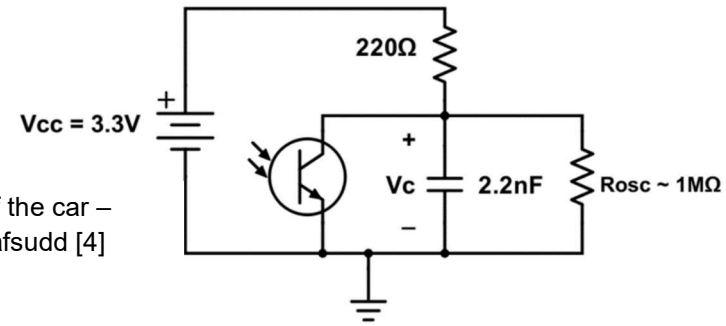


Figure 4. The phototransistor circuit of the car – ECE3 Lab Manual, Professor O.M. Stafsudd [4]

On the project car, there are 8 pairs of IR emitters and sensors arranged in an array underneath the project car, leaning closer to the front. As the project car moves forward, the IR LED will emit infrared light to the ground. The light will then be reflected by and surface and detected by the IR sensors. The received sensor values range from 0 to 2500 and will later be used to determine the speed of the left and right motors. Since the IR sensors receive light reflected from the ground, the color of the path becomes relevant. When a surface is white, it will reflect all wavelengths. On the other hand, when a surface is black, it will absorb wavelengths from the light. Therefore, the reflected light intensity should be different when the car is above a white and black surface. If the path in front of the car is white, the IR sensors will report low values (usually around 600 to 700). On the other hand, if the path in front of the car is black, the IR sensors will report high values (usually ranging from 2300 to 2500). Figure 5 gives a visualization of the process described above.

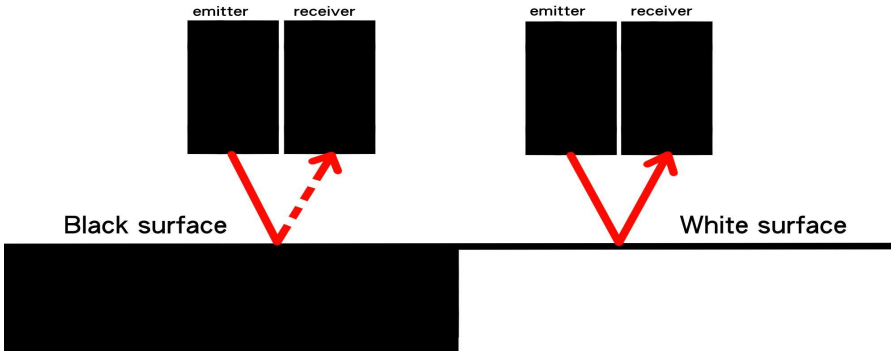


Figure 5. The reflection of IR light from different surfaces - Source: Yolanda Chen

However, although 8 IR emitters and sensors are supposedly identical electrical components, the values reported by the IR sensors may differ slightly. As a result, the recorded sensor data will be calibrated so that each sensor reports the same values. Below we will go over the technique taught by our TA, Xin Li [5].

To collect data from our project car, we placed our project car on a white sheet of paper. Then, we cut a small portion of the track with the black path in the middle and moved it across the sensors slowly from left to right. The sensors recorded the data every second. The following graph is the raw data collected from the sensors.

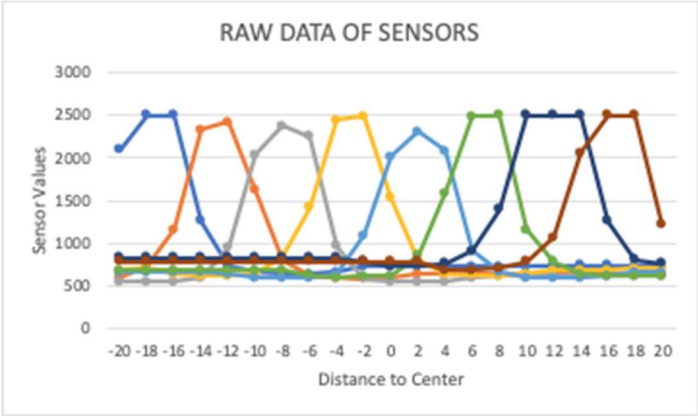


Figure 6. Graph of raw data from each - Source: Yolanda Chen

Note each curve represents the data collected by the sensors as we move the track across them. A peak is formed when the black section of the track is directly below the sensor. Also note that the highest and lowest value for each sensor is slightly different, proving our statement mentioned in the above paragraph.

After we collected the raw data of the sensors, we removed their offset by subtracting each value by the minimum value of each sensor, and then scaled the maximum values to 1000. By doing so, each sensor will map to the same values when they detect the black track. The following are the results for both steps.

Removing the offsets:

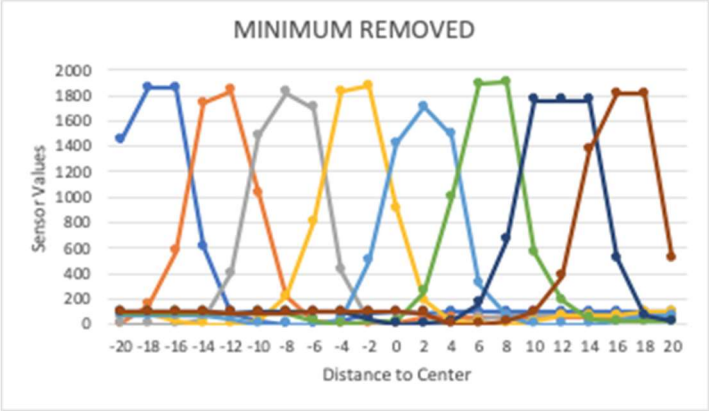


Figure 7. Graph of data with removed offset - Source: Yolanda

Normalizing the values so that the maximum goes to 1000:

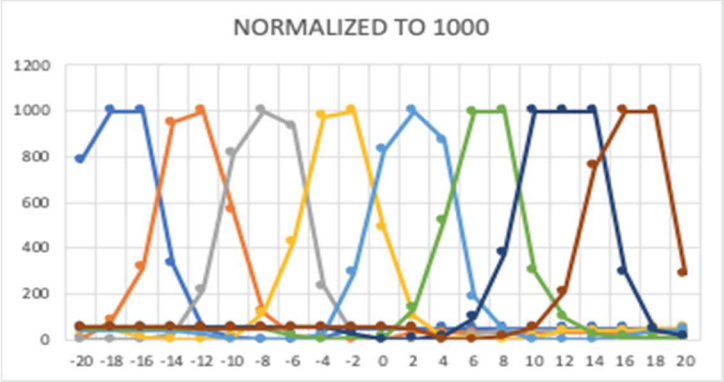


Figure 8. Graph of normalized data - Source: Yolanda

Finally, using the normalized values, we fuse the sensors data into one set of data. The fused data will be used to steer and determine the motor speed of the left and right motor of the project car. To fuse the data, we used the following idea introduced to us by Dr. Briggs [2]:

The collected array of sensor data are multiplied element by element with an array of values, either [-8, -4, -2, -1, 1, 2, 4, 8] or [-15, -14, -12, -8, 8, 12, 14, 15]. We then derive an integrated value by summing the scaled values. The integrated value is different in each scenario.

When the car is positioned at the center of the track, the sensors near the middle will detect high values while the sensors on the far left and right will return low values. Applying the weight and summing the eight values together, we will get an integrated value that is close to zero. However, when the car is positioned far left or right, sensors on the far left or right will be getting high values. Repeating the same procedure above, the integrated value will either be negative (left side off) or positive (right side off). Since the weights for far left and right have the largest magnitude, we know the car is way off track when the integrated value is a large positive or negative number. In that case, the car should steer towards the opposite direction. For example, if the left sensors are getting high values, we want to steer the car to the right, and vice versa. [2]

With this idea, we graph out the sensor fusion data of our calibrated data. We included the results for both weights in Figure 9.

The blue curve corresponds to scaling by [8,4,2,1] and the orange curve is for [15,14,12,8] scaling.

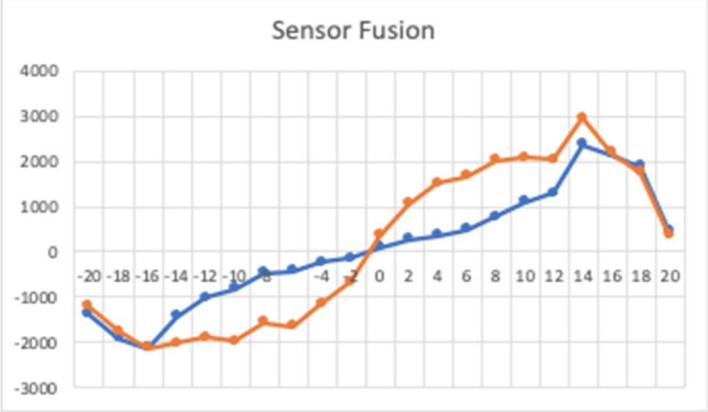


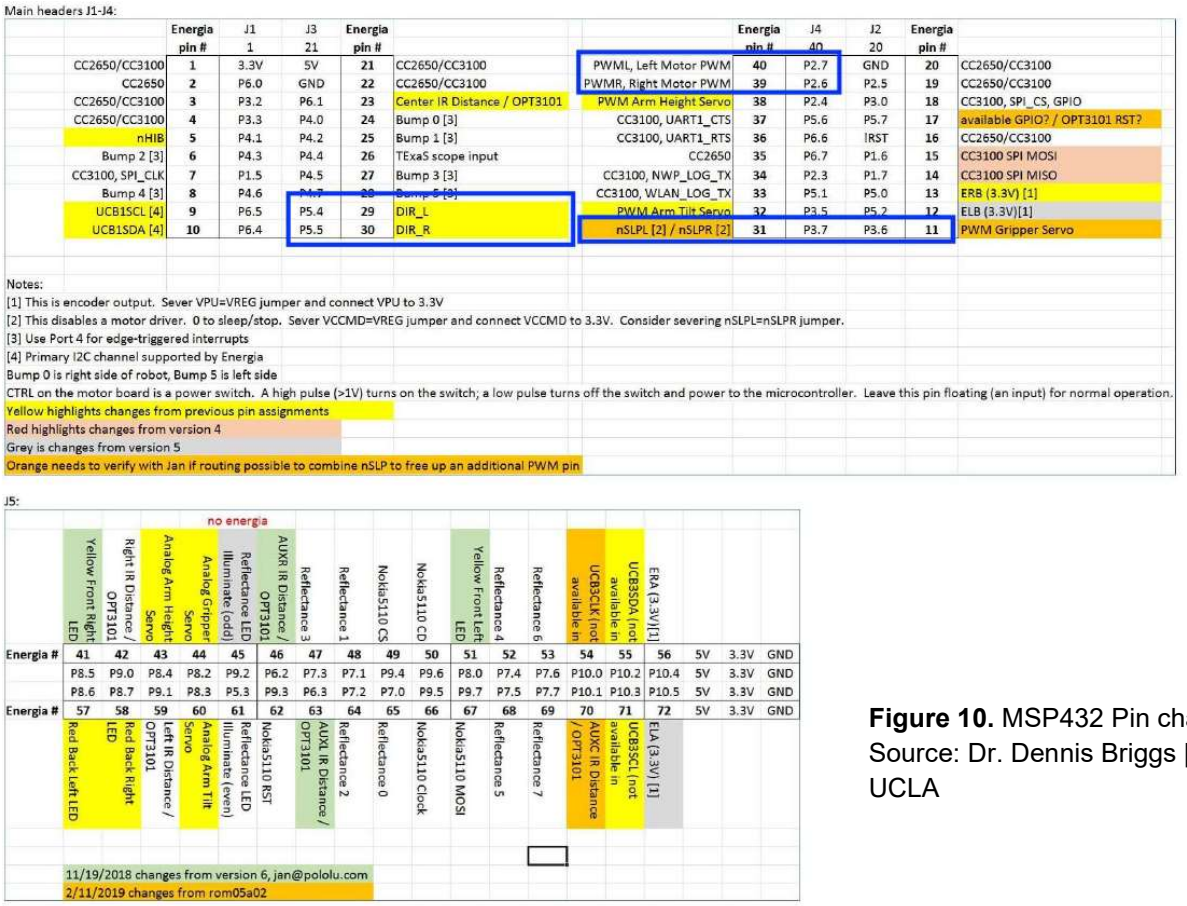
Figure 9. Fused sensor data as a function of position - Source: Yolanda Chen

2.1 Setups

The most important characteristic of the environment was sunlight intensity. The means for detecting the path relies heavily on the reflected infrared light. The sunlight contains infrared wavelengths which can cause distortion in sensors' readings. To find the offsets and maxima of the sensors, we collected the calibration data in an environment with the same sunlight intensity as the environment of the path. The default light intensity was assumed to be the intensity at 8am pacific standard time during fall. Note that this primary calibration was done before the implementation of the self-calibration algorithm. The self-calibration algorithm enables the car to function in environments with various IR intensities. In other words, the car will be able to collect a set of data, find their offsets and maxima, and adjust the inputs without any human interaction. To learn more about this algorithm see appendix I.

The paths were taped to a hard surface (floor) which enabled the car to move easily. Also, a hard surface ensured the distance between the path and the sensors would stay constant, so the light intensity was only affected by the color, not distance.

Prior to writing the code and preparing the car to track the straight and curved path, we performed several tests to ensure the motors and the sensors on the car were functional. The pin chart below shows the function of each pin.



Main headers J1-J4:

	Energia pin #	J1 1	J3 21	Energia pin #			Energia pin #	J4 40	J2 20	Energia pin #	
CC2650/CC3100	1	3.3V	5V	21	CC2650/CC3100	PWML, Left Motor PWM	40	P2.7	GND	20	CC2650/CC3100
CC2650	2	P6.0	GND	22	CC2650/CC3100	PWMR, Right Motor PWM	39	P2.6	P2.5	19	CC2650/CC3100
CC2650/CC3100	3	P3.2	P6.1	23	Center IR Distance / OPT3101	PWM Arm Height Servo	38	P2.4	P3.0	18	CC3100, SPI_CS, GPIO
CC2650/CC3100	4	P3.3	P4.0	24	Bump 0 [3]	CC3100, UART1_CTS	37	P5.6	P5.7	17	available GPIO? / OPT3101 RST?
nHIB	5	P4.1	P4.2	25	Bump 1 [3]	CC3100, UART1_RTS	36	P6.6	!RST	16	CC2650/CC3100
Bump 2 [3]	6	P4.3	P4.4	26	TExaS scope input	CC2650	35	P6.7	P1.6	15	CC3100 SPI MOSI
CC3100, SPI_CLK	7	P1.5	P4.5	27	Bump 3 [3]	CC3100, NWP_LOG_TX	34	P2.3	P1.7	14	CC3100 SPI MISO
Bump 4 [3]	8	P4.6	P4.7	28	Bump 5 [3]	CC3100, WLAN_LOG_TX	33	P5.1	P5.0	13	ERB (3.3V) [1]
UCB1SCL [4]	9	P6.5	P5.4	29	DIR_L	PWM Arm Tilt Servo	32	P3.5	P5.2	12	ELB (3.3V)[1]
UCB1SDA [4]	10	P6.4	P5.5	30	DIR_R	nSLPL [2] / nSLPR [2]	31	P3.7	P3.6	11	PWM Gripper Servo

Notes:
[1] This is encoder output. Sever VPU=VREG jumper and connect VPU to 3.3V
[2] This disables a motor driver. 0 to sleep/stop. Sever VCCMD=VREG jumper and connect VCCMD to 3.3V. Consider severing nSLPL=nSLPR jumper.
[3] Use Port 4 for edge-triggered interrupts
[4] Primary I2C channel supported by Energia
Bump 0 is right side of robot, Bump 5 is left side
CTRL on the motor board is a power switch. A high pulse (>1V) turns on the switch; a low pulse turns off the switch and power to the microcontroller. Leave this pin floating (an input) for normal operation.
Yellow highlights changes from previous pin assignments
Red highlights changes from version 4
Grey is changes from version 5
Orange needs to verify with Jan if routing possible to combine nSLP to free up an additional PWM pin

J5:

	Yellow Front Right LED	Right IR Distance / OPT3101	Analog Arm Height Servo	Analog Gripper Servo	Reflectance LED illuminate (odd)	AUXR IR Distance / OPT3101	Reflectance 3	Reflectance 1	Nokia5110 CS	Nokia5110 CD	Yellow Front Left LED	Reflectance 4	Reflectance 6	UCB3CLK (not available in	UCB3SDA (not available in	ERA (3.3V)[1]			
						no energia													
Energia #	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	5V	3.3V	GND
	P8.5	P9.0	P8.4	P8.2	P9.2	P6.2	P7.3	P7.1	P9.4	P9.6	P8.0	P7.4	P7.6	P10.0	P10.2	P10.4	5V	3.3V	GND
	P8.6	P8.7	P9.1	P8.3	P5.3	P9.3	P6.3	P7.2	P7.0	P9.5	P9.7	P7.5	P7.7	P10.1	P10.3	P10.5	5V	3.3V	GND
Energia #	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	5V	3.3V	GND
	Red Back Left LED	Red Back Right LED	Left IR Distance / OPT3101	Analog Arm Tilt Servo	Reflectance LED illuminate (even)	Nokia5110 RST	AUXL IR Distance / OPT3101	Reflectance 2	Reflectance 0	Nokia5110 Clock	Nokia5110 MOSI	Reflectance 5	Reflectance 7	AUXC IR Distance / OPT3101	UCB3SCL (not available in	ELA (3.3V) [1]			

11/19/2018 changes from version 6, jan@pololu.com
2/11/2019 changes from rom05a02

Figure 10. MSP432 Pin chart. Source: Dr. Dennis Briggs [2] - UCLA

Motors: To enable communication with the motors, The NSLP pins (31 & 11) must both be assigned HIGH. Once the motors were ready, the direction of the wheels could be determined by assigning HIGH or LOW to the digital direction pins (pins 29 & 30). Assuming the infrared sensors were at the front of the car, a LOW would correspond to the wheel rotating forward and a HIGH would be backward.

After getting familiar with the direction of the wheels, we tested the speed. Each PWM speed control pin (39 & 40) can take a value from 0 to 255. It is important to note that negative values are not acceptable and do not change the direction of the wheels. The higher the number, the faster the wheel rotates. However, due to electrical characteristics of the motors, the wheels do not rotate when the speed values are below 20.

Next, we made sure the two motors responded similarly to equal values. To do so, we assigned equal values to both of the speed control pins and tested the car on a straight line. Since the car did not deviate, we could safely conclude the motors had very similar responses.

Sensors: To be able to read the sensor values, we had to use the ECE3_read_IR function from the ECE3.h library. The argument of this function has to be a one-dimensional array with 8 spaces

to store integer values from each sensor in one space. To read the content of the array we had to print the values into the serial monitor. We chose a baud rate of 9600 for serial communication. Since the data collection rate of the car was very fast, for calibration purposes, we had to add a 1-second delay at the end of each iteration. As explained in the calibration section, the IR sensors did not have the exact same characteristics and adjustments had to be made to the raw input values. Once, the tests were over, we deleted all of the serial commands since they are not necessary for tracking and can slow down the execution significantly.

2.2 Procedure

The most challenging part of this project was determining the PID coefficients, Kd and Kp. To determine these values, we did a brief analysis to estimate what the values would be. Once the estimated value was found, we would try numbers close to that value. This method was much more efficient than plane guess and check.

Kp: The faster the car moves, the higher Kp must be. Here is an example: when the car is making a right turn, the right sensors detect the black strip. This means the proportional term must push the car to the right. Therefore, the left wheel should spin faster while the right wheel slows down. If the base speed is large, the proportional term will not have a significant effect, so the car gets off the track. Therefore, we assume Kp is linearly proportional to the base speed. It is also worth noting the sensor data was fused by [8,4,2,1] scaling since it would provide a larger error when the center of the car was far from the path, resulting in the proportional term pushing the car strongly towards the path.

We did not want to have a change of speed greater than half of the base speed. By looking into our calibration data, we realized the maximum error was 1500. By combining these two concepts, we reached the following relationship for Kp:

$$Kp = (1/1500)\,(baseSpd)\,(strFac)$$

The term ***strFac*** is the stirring factor. Later in the project, we saw for high speeds, Kp does not follow a linear relationship with the base speed. The ***strFac*** made the required adjustments and could only be obtained through testing. This term became even more important in following the curved track.

Kd: Once we found an appropriate Kp value that made the car roughly follow the line, we started adjusting Kd to achieve a smooth motion. The derivative term finds the error using the data scaled by [15,14,12,8] fusion. The reason is, when the car is away from the path and moving towards it, a large derivative term will cancel the proportional term and push the car away. As the car gets closer to the path, the derivative term starts getting larger so that the car does not pass the path and aligns itself smoothly. Therefore, we want the diffusion values to vary slowly away from the path and sharply close to the path (recall the derivative term calculates the change in the error value which corresponds to slope of the graph in Figure 9).

We mainly tested the car starting at positions 3 and 4 since they were the more extreme cases of positions 2 and 1. Therefore, we did not test the car at positions 1 and 2 for each base

speed, but once we reached the goal speed for the straight and curved track, we did those tests to be thorough.

2.3 Data

The earlier discussion in the procedure section only gives us a big picture for how the Kp and Kd should change with the base speed. To find the exact values we conducted several tests, results of which are provided in Tables 1 and 2.

Track	Base Speed	(1/1500) (baseSpd)/2	strFac	Kp	Kd	Starting Position	Deviation point	Deviation Direction	Comments
Straight	60	0.02	1	0.02	1	4	-	-	The car jiggled too much while tracking the line Therefore Kd must be smaller
Straight	60	0.02	1	0.02	0.5	4	-	-	The car jiggled while tracking the line Therefore Kd must be smaller
Straight	60	0.02	1	0.02	0.1	4	-	-	Success
Straight	80	0.02667	1	0.02667	0.1	4	-	-	Success
Straight	80	0.02667	1	0.02667	0.1	3	-	-	Success
Straight	100	0.0333	1	0.03333	0.2	4	-	-	Success
Straight	100	0.0333	1	0.03333	0.2	3	-	-	The car took some time to adjust itself to the track. Therefore Kd must be larger
Straight	100	0.0333	1	0.03333	0.5	4	middle	right	Failed to track the line. Kd must be smaller
Straight	100	0.0333	1	0.03333	0.3	4	-	-	Car jiggled while traching the line. Kd must be smaller
Straight	100	0.0333	1	0.03333	0.22	4	-	-	Success
Straight	100	0.0333	1	0.03333	0.22	3	-	-	Success
Straight	120	0.04	1	0.04	0.22	3	middle	left	Kp must be larger
Straight	120	0.04	1.2	0.048	0.22	4	-	-	Success
Straight	120	0.04	1.2	0.048	0.22	3	-	-	Success
Straight	150	0.05	1.2	0.06	0.22	4	middle	right	Kp must be larger
Straight	150	0.05	1.4	0.07	0.22	4	1/3 point	right	The car is not adjusting to the track. Kd mut be larger
Straight	150	0.05	1.4	0.07	0.3	4	1/3 point	right	Kp must be larger
Straight	150	0.05	1.6	0.08	0.3	3	-	-	Successs
Straight	150	0.05	1.6	0.08	0.3	4	1/3 point	left	Kp must be larger
Straight	150	0.05	1.8	0.09	0.3	4	beggining	left	The tracking is getting worse as we increase Kp. This is due to the large overshoot at the start. Kp must be much smaller and for a smaller Kp we have to decrease Kd as well
Straight	150	0.05	0.8	0.04	0.2	4	middle	-	Overshoots still visible
Straight	150	0.05	0.6	0.03	0.2	4	-	-	Success
Straight	150	0.05	0.6	0.03	0.2	3	-	-	Success
Straight	150	0.05	0.6	0.03	0.2	2	-	-	Success
Straight	150	0.05	0.6	0.03	0.2	1	-	-	Success

Table 1: Test values for straight track at different speeds – Source: Ramtin Sharafoleslami

For the curved path, to indicate at which points the car got off the track, we used the following numbers:

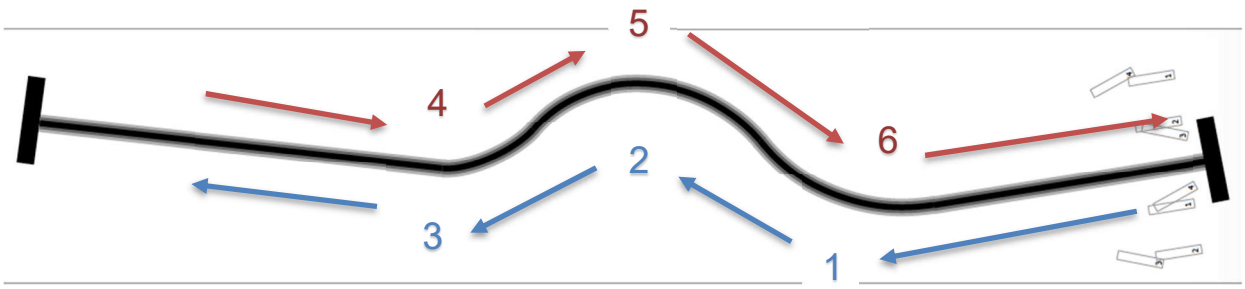


Figure 11: The numbers for deviation points - Source: Ramtin Sharafoleslami

Track	Base Speed	(1/1500) (baseSpd)/2	strFac	Kp	Kd	Starting Position	Deviation point	Deviation Direction	Comments
Curved	50	0.01667	1	0.0167	0.2	4	1	left	Failed to track the path. Kp must be larger
Curved	50	0.01667	1.5	0.025	0.2	4	2	right	Failed to track the path. Kp must be larger
Curved	50	0.01667	2	0.0333	0.2	4	5	left	Failed to track the path when returning. Kp must be larger
Curved	50	0.01667	2.5	0.0417	0.2	4	-	-	Success
Curved	50	0.01667	2.5	0.0417	0.2	3	-	-	Success
Curved	60	0.02	2.5	0.05	0.2	4	5	left	Failed to track the path when returning. Kp must be larger
Curved	60	0.02	3	0.06	0.2	4	-	-	Completed the path with jiggling at the beginning higher Kd recommended
Curved	60	0.02	3	0.06	0.24	4	-	-	Success
Curved	60	0.02	3	0.06	0.24	3	-	-	Success
Curved	70	0.0233	3	0.07	0.24	3	3	right	Failed to track the path. Kp must be larger
Curved	70	0.0233	3.2	0.0747	0.24	3	-	-	Success
Curved	70	0.0233	3.2	0.0747	0.24	4	2	right	Failed to track the path. Kp must be larger
Curved	70	0.0233	3.5	0.0817	0.24	4	2	right	Failed to track the path. Deviation might be due to unstable tracking before reaching point 1. Higher Kd required
Curved	70	0.0233	3.5	0.0817	0.26	4	-	-	Success
Curved	70	0.0233	3.5	0.0817	0.26	3	-	-	Success
Curved	70	0.0233	3.5	0.0817	0.26	2	-	-	Success
Curved	70	0.0233	3.5	0.0817	0.26	1	-	-	Success

Table 2: Test values for curved track at different speed - Source: Ramtin Sharafoleslami

As highlighted in green, each base speed has its Kp and Kd value which give the most accurate and smoothest tracking. Figures 12 and 13 show these values as a function of their base speeds on the straight track.

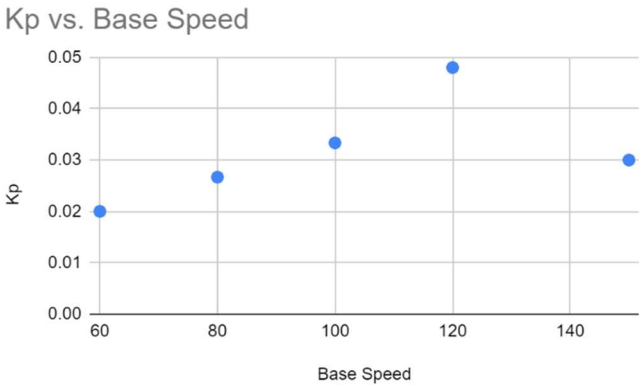


Figure 12: Graph of successful Kp values vs their corresponding base speeds- Source: Ramtin Sharafoleslami

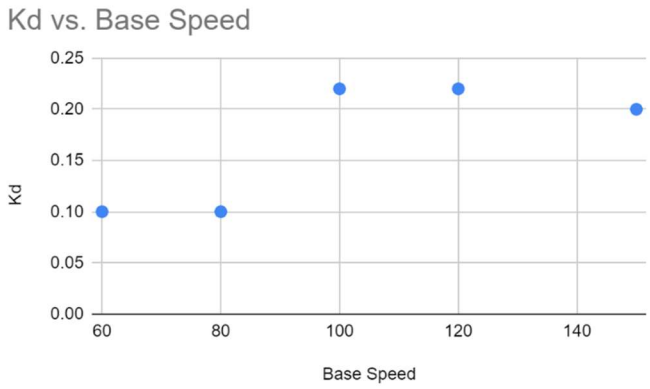


Figure 13: Graph of successful Kd values vs their corresponding base speeds - Source: Ramtin Sharafoleslami

2.4 Test Data Interpretation

Tables 1 and 2 both have a comments column in which the interpretation of the test data and the results are put. If the car deviated from the path, there could be two reasons, either the proportional term was not large enough to keep the car on the track or the derivative control was large so that it canceled the proportional term and pushed the car away. Most of the times, the problem could be solved by increasing the Kp value.

However, even though it sounds reasonable that a large Kp value would do a better job at keeping the car on the track, that was not the case for base speeds higher than 120. In Table 1, we see for the speed of 150, increasing the Kp value was generating worse results. This happened in starting positions 3 and 4 where the error value was large at the beginning. As a result, the proportional term pushed the car towards the track but since this push was very large, the car would go past the track. To avoid this, Kp had to be decreased but simply decreasing Kp would not work. Kd also had to decrease so that the two terms would balance each other out.

Once the proper Kp was found, determining the Kd value was easier. There were two scenarios which showed Kd had to be changed:

a. If the car could not adjust itself after travelling through 1/3 of the track, the Kd value was small.
b. If the car could adjust to the track quickly but then jiggled, it meant Kd was large.

Using these interpretations, we were able to find the proper Kp and Kd values for the car to track the straight path at the speed of 150 and the curved path at 70.

3. Results & Discussion

a. Test Discussions:

During the first two weeks of the project, we did not use PID control. The initial algorithm used was very simple and inefficient. Even though it was successful at tracking the straight path at very low speeds, it would not work for higher speeds and the curved path. The code for this algorithm is provided below:

```
leftCount = 0;
rightCount = 0;

  for (uint16_t i = 0; i < 4 ; i++)
  {
    if (calibValues[i] > 300 )
    {
      rightCount++;
    }
  }
    for (uint16_t i = 4; i < 8 ; i++)
  {
    if (calibValues[i] > 300 )
    {
      leftCount++;
    }
  }
```

```
    if ( rightCount > leftCount )
    {
      leftSpd = refRightSpd + 20;
      rightSpd = refRightSpd;
    }
    if ( rightCount < leftCount )
    {
      leftSpd = refLeftSpd;
      rightSpd = refRightSpd + 20;
    }
```

The main idea was to count how many dark spots the right sensors and the left sensors were counting and then compare the numbers to see if the path is on the left or the right side of the car. Then the speed of the wheels could be adjusted. In this case, the reference speed of both wheels was 20, meaning the maximum speed each wheel could reach after the adjustments was 40. This is much smaller than the 150 base speed that we were able to achieve using the PID control.

The reason this primary algorithm is not considered efficient is that the code contains two for loops and a comparison within each loop. This combination slows down the execution and a slow code cannot guide the car at high speeds.

Applying the PID concepts, enabled us to improve the functionality significantly. The tracking became smooth, the execution time decreased, and higher speeds were achieved. By looking at the PID code (*see appendix II)* we see there is less comparisons and no for loops.

As discussed in section 2.2, to determine the Kp value, we came up with a mathematical equation which helped estimating Kp with less trials. In fact, as shown in Figure 12, Kp followed our mathematical model for the speeds below 120. The graph is completely linear from speed 60 to 100 and then starts decreasing.

Finding Kd was not as straight forward and had to be found through experimentation. However, the from Figure 13 we can see as the speed was increasing, Kd was increasing too until we reached the speed 120. After that, since Kp started to decrease, the Kd graph also started to have a negative slope. We can also see unlike Kp, Kd values are not making a smooth function partly because they are determined after Kp.

Overall, the data meets the project goals. We were able to make a model for Kp and once we had Kp, we could also find Kd easily. By understanding the behavior of these two parameters, we were able to adjust them for the curved path. As a result, we reached a maximum speed of 150 for the straight track and 70 for the curved path. As discussed in section 3.b, our fastest car was able to complete the race at 9.1 seconds. This is the time that took the car to reach the end of the track and come back at its starting position. At the end of the track, there was a cross piece which had to be interpreted by the car as the end of the track and prompt it to return. The code for cross piece detection is under appendix II. It must be mentioned that the project had some limitations which did not allow us to go beyond certain speeds. We will go over them in the next section.

Other than completing the race, we implemented two additional algorithms for the car to adapt itself to the environment. These algorithms are self-calibration algorithm (*appendix I)* and path-recovery algorithm *(appendix III)*.

b. Race Day Discussions:

i. On Race Day, both of our project cars ran smoothly without any major issues. The first car completed in 9.1 seconds while the second car completed in 10.2 seconds. Therefore, both of our cars met the extra credit criteria.

While the first car met the criteria for extra credit on its first try, the second car did not. On the first try, the second car completed the race in around 11.5 seconds, which was very close to hitting the 11 second mark. As a result, we tried to adjust the base speed of the second car from the original 60 to 70. In the end, on its second try, the car completed in 10.2 seconds.

ii. Limitations of our code is mostly caused by it being specific to certain values such as speed and time. For example, as we were trying to improve the speed for the second car, we realized that when we increased the speed from 60 to 70, the car strayed away from the path using the same Kp and Kd values as the original. We ended up testing different Kp and Kd values for the base speed of 70.

Furthermore, there are some limitations to our cars and the track. We tried to increase the speed to 170 but at such a high speed, the friction between the wheels and the track were not large enough. As a result, even if the speeds commands were properly given, the car would not behave accordingly on the track. Later we realized that adding more weight to the car could improve the performance at high speeds.

iii. Even though we were careful with implementation of serial monitor and delays in the code, we did not take the charge of the batteries seriously. After we completed all the tests for the straight track, we did not replace the batteries and went straight to the curved path. After doing multiple tests, we found Kp and Kd values for speed of 90. The next day, we raised the speed to 110. The same Kp and Kd values worked. Then we tried 120 and they were still working. This caused suspicions to the point we tried timing the race. The time for completing the race with speeds of 90,110, and 120 turned out to be equal. From that we figured out the issue was with the batteries. After we replaced them, the car showed a completely different behavior even at the lower speeds. Therefore, we had to redo all the tests for the curved path since our previous values were invalid. We also tested the Kp and Kd values for the straight track (to make sure there was no error in them due to battery charge) but they were working properly.

4. Conclusions & Future Work

Our design was successful in finishing the track under 11 seconds without any deviation from the path. Throughout this project, we became familiar with the concept of PID control and learned the Kp and Kd values varied with the base speed of the car. We had an idea to derive an equation for Kp using the base speed but were unable to find an equation for Kd. If we had more time to improve our car, we would try to derive an equation of Kd based on the base speed and find a more advanced mathematical model for Kp which explained its non-linear behavior at higher speeds.

Another aspect of the project that can be improved is using different scaling factors for data fusion or implementation of an algorithm which is not specific to a track. By looking at tables 1 and 2, we see Kp and Kd are different for the straight and curved path at the same speeds. This means our code and coefficients are path dependent. By implementing a more advanced algorithm and PID coefficients which stay the same for different paths, our car can be more flexible and useful for other purposes.

5. References:

[1] Pololu Chassis Kit for TI-RSLK MAX. https://www.pololu.com/product/3670/pictures

[2] Dr. Briggs, Dennis M. A Conceptual Description of the PID Controller.

[3] PID Controller Implementation Using Arduino. https://microcontrollerslab.com/pid-controller-implementation-using-arduino/

[4] Professor Stafsudd, O.M. ECE3 Lab Manual.

[5] Li, Xin. Teaching Assistant. Lab 6, Calibration Notes.

APPENDIX I. Self-Calibration Algorithm

While doing the calibration data for the sensors, we realized that the sensors might detect slightly different values every time. In other words, each sensor doesn't necessarily report the same values for "sees black" every time. To make the sensor data more accurate, we implemented the code for self-calibration each time we turn on the car.
The self-calibration code is implemented in the setup() section of the entire code. We will collect data at the beginning of each run. To help the car collect sensor data, we will need to move the car back and forth vertical to the track. Below is the code for the self-calibration:

```
/* Self-Calibration Algorithm */
  ECE3_read_IR(sensorValues);
  for (int i = 0; i < 8; i++) {
    sensorMin[i] = sensorValues[i];
    sensorMax[i] = sensorValues[i];
  }
// Sensors start reading in data
  for (int i = 0; i < 100; i++) {
    ECE3_read_IR(sensorValues);
    for (int j = 0; j < 8; j++) {
      // set Min and Max to the current Min and Max
      if (sensorMin[j] > sensorValues[j])
        sensorMin[j] = sensorValues[j];
      if (sensorMax[j] < sensorValues[j])
        sensorMax[j] = sensorValues[j];
    }
    delay(50);
  }
```

To give us some time to get ready before the car starts to collect data, an LED on the project car will blink 5 times to notify us (See Appendix for the full code). When the sensors are collecting data, the LED will turn off. After around 5 seconds, the LED will start blinking again, indicating that data has been collected.

Note that in the code, we are actually trying to find the minimum and maximum values read by each sensor, because these two values are the most important values for calibration. The minimum and maximum values are stored in separate arrays of size 8, namely sensorMin[8] and sensorMax[8].

At the beginning of the loop() section of the entire code, the car will first read in sensor values. These newly collected data are raw values of the sensor that will later be calibrated using sensorMin and sensorMax. Below is the code for the calibration:

```
// calibrate the values
    for (int i = 0; i < 8; i++) {
```

```
      calibrated[i] = (sensorValues[i] - sensorMin[i]) * 1000 /
sensorMax[i];
    }
      fusionCur = (-8*calibrated[0] - 4*calibrated[1] -
                   2*calibrated[2] - 1*calibrated[3] +
                   1*calibrated[4] + 2*calibrated[5] +
                   4*calibrated[6] + 8*calibrated[7])/8;
    }
```

Note that this example uses the [8,4,2,1] scale for fusion.

APPENDIX II. Source Code with Cross Piece Detection

```
#include <ECE3.h>

uint16_t sensorValues[8]; // right -> left, 0 -> 7
float    calibValues [8]; // stores the calibrated data from the
sensors

float    farErr;
float    closeErr;
float    prevErr = 0;

int      leftSpd;
int      rightSpd;
int      baseSpd = 70;
int      sum;
bool     comingBack = false;

float    strFac = 3.5;
float    Kp;
float    Kd;

const int left_nslp_pin=31; // nslp ==> awake & ready for PWM
const int left_dir_pin=29;
const int left_pwm_pin=40;

const int right_nslp_pin=11; // nslp ==> awake & ready for PWM
const int right_dir_pin=30;
const int right_pwm_pin=39;

const int LED_RF = 41;

///////////////////////////////////
void setup() {
// put your setup code here, to run once:
  pinMode(left_nslp_pin,OUTPUT);
  pinMode(left_dir_pin,OUTPUT);
  pinMode(left_pwm_pin,OUTPUT);

  digitalWrite(left_dir_pin,LOW);
  digitalWrite(left_nslp_pin,HIGH);

  pinMode(right_nslp_pin,OUTPUT);
  pinMode(right_dir_pin,OUTPUT);
  pinMode(right_pwm_pin,OUTPUT);
```

```
  digitalWrite(right_dir_pin,LOW);
  digitalWrite(right_nslp_pin,HIGH);

  pinMode(LED_RF, OUTPUT);

  ECE3_Init();

  Serial.begin(9600);
  delay(2000); //Wait 2 seconds before starting
}

void loop() {
  //Store the reading from the 8 sensors in an array
  ECE3_read_IR(sensorValues);

  // ***************** Calibration ************************

  calibValues[0] = (sensorValues[0] - 483) * 1000 / 2500;
  calibValues[1] = (sensorValues[1] - 414) * 1000 / 2375;
  calibValues[2] = (sensorValues[2] - 437) * 1000 / 2470.8;
  calibValues[3] = (sensorValues[3] - 414) * 1000 / 2090;
  calibValues[4] = (sensorValues[4] - 460) * 1000 / 2114;
  calibValues[5] = (sensorValues[5] - 437) * 1000 / 2500;
  calibValues[6] = (sensorValues[6] - 483) * 1000 / 2500;
  calibValues[7] = (sensorValues[7] - 460) * 1000 / 2500;

  // ********************** Fusion ****************************

  farErr = ( -8*calibValues[0] -4*calibValues[1] - 2*calibValues[2] -
calibValues[3] + calibValues[4] + 2*calibValues[5] + 4*calibValues[6]
+ 8*calibValues[7] )/4.0;

  closeErr = ( -15*calibValues[0] -14*calibValues[1] -
12*calibValues[2] - 8*calibValues[3] + 8*calibValues[4] +
12*calibValues[5] + 14*calibValues[6] + 15*calibValues[7] )/8.0;

  sum = calibValues[0] + calibValues[1] + calibValues[2] +
calibValues[3] + calibValues[4] + calibValues[5] + calibValues[6] +
calibValues[7];

  // ********************** PID Control ***********************

  Kp = (1./1500.) * (baseSpd/2)* strFac;
  Kd =  0.26;
```

```
  rightSpd = baseSpd + Kp * farErr + Kd * (closeErr - prevErr);
  leftSpd  = baseSpd - Kp * farErr - Kd * (closeErr - prevErr);
  prevErr = closeErr;

  // ******************** Movement ************************

  if(sum > 3000 && !comingBack)   // end of the first trip
  {
    comingBack = true;
    digitalWrite(left_dir_pin,HIGH);
    digitalWrite(right_dir_pin,LOW);
    digitalWrite(LED_RF, HIGH);

    //Donut
    analogWrite(left_pwm_pin,120);
    analogWrite(right_pwm_pin,120);
    delay(430);

    //Getting Back On Track
    digitalWrite(left_dir_pin,LOW);
    analogWrite(left_pwm_pin,90);
    analogWrite(right_pwm_pin,90);
    delay(200);
  }
  else if(sum > 3000 && comingBack) // end of the trip
  {
    analogWrite(left_pwm_pin,0);
    analogWrite(right_pwm_pin,0);
      while(true){};
      // the car stays at the end of the path until it's restarted
  }
  else
  {
    // Execution
    digitalWrite(LED_RF, LOW);
    digitalWrite(left_dir_pin,LOW);
    digitalWrite(right_dir_pin,LOW);
    analogWrite(left_pwm_pin,leftSpd);
    analogWrite(right_pwm_pin,rightSpd);
  }
}
```

APPENDIX III. Path Recovery Algorithm

The following code segment can be placed under the movement section of the code in appendix II. This code is executed when sum of all sensor values drops below 400 which means all of them are detecting white color. At this point, the car stops immediately, goes in on reverse with the speed of 60 for 800 milliseconds. This time was found by experimentation and ensures at least one sensor goes over the black curve. If the car is still on the white area after the execution, this segment gets repeated until the black curve is detected.

```
else if(sum < 400)   // Path Recovery
{
  digitalWrite(left_dir_pin,HIGH);
  digitalWrite(right_dir_pin,HIGH);
  digitalWrite(LED_RF, HIGH);

  analogWrite(left_pwm_pin,0);
  analogWrite(right_pwm_pin,0);
  delay(400);
  analogWrite(left_pwm_pin,60);
  analogWrite(right_pwm_pin,60);
  delay(800);
```