Lab Report 2 – RF Remote Control & Line Following

By Junaid Afzal
eeyja9@nottingham.ac.uk
Student ID: 4326724

Applied Electrical and Electronic Engineering: Construction Project
(EEEE1002 UNUK) (FYR1 18-19) (H61AEE)

Wednesday 6th February 2019
Partners: Emmanuel Boateng, Matthew Haywood & Oliver Jennings

Contents

1. Abstract

The main purpose of the two weeks was to modify the HMI from the previous week by converting it in to the form of a wireless handheld remote that can give continuous commands and display, to the user, real time telemetry information about the car. Then a line following mode, where an array of optical sensors will calculate the position of the car to the line, need to be designed, constructed and tested.

The full capabilities of the car, by the end of the two weeks, were that it could follow a line while from the remote one could see the current PID constants while also being able to adjust them. Also, if the joysticks were moved then the car would switch to RF remote mode. Here the car would take commands from the joysticks on the RF remote and the car would send telemetry data back to the remote where it would be displayed on an LCD. To return to line following mode the reset button on the Arduino would have to be pressed. However, this would clear any of the PID constants that may have been previously set.

2. Introduction

The main purpose of the two weeks was to modify the HMI from the previous week by converting it in to the form of a wireless handheld remote that can give continuous commands and display, to the user, real time telemetry information about the car. Then a line following mode, where an array of optical sensors will calculate the position of the car to the line, need to be designed, constructed and tested.

For the car to be able to follow a black line amongst a white surface, an optical sensor array, that outputs an analogue signal, needed to be designed and constructed. This array of sensors should vary their outputs depending upon how much of the line is directly below them. The array of optical sensors will then be tested by measuring the output voltage of each sensor when it is above the line and when it is not above the line. A sensor module needs to be constructed, which will hold the Arduino that will handle the sensor data. A demonstration of the advantages and disadvantages of a digital versus analogue approach to sensing the line also must to be completed. Next within the Arduino code, a weighted average formula to calculate the position of the car with respect to the line and a PID control formula to calculate the change in the motor speeds required for the car to follow the line, need to be created. These two formulae are then to be combined in to one so that the car can respond to the changes in the position of the line by adjusting the speed of the motors. The functionality of the car will then be tested by having it go around a track, with a white surface and a black line, and timing how long it takes to complete the track – if it can.

If the car is in line following mode, then the PID constants should be displayed on the screen of a handheld remote; adjusted via an input method on the remote; and then sent to the sensor Arduino. If the car is in RF remote mode, then an MPU6050 attached to the car should send data to the screen of the handheld remote and the car should no longer be following the line and instead be controlled via some human machine interface attached to the remote. A way of being able to switch from line following mode to RF remote mode must also to be implemented. In order to achieve this a handheld remote will need to be designed, constructed and tested.

3. Optical Sensor Array

The objectives in this chapter are to design and construct an optical sensor array that outputs an analogue signal which will vary depending upon how much of the line is directly below the sensor. The designing, constructing and testing of a comparator circuit is also to be carried out. A sensor module needs to be constructed, which will hold the Arduino that will handle the sensor data. The advantages and disadvantages of an analogue versus digital approach to sensing the line will be discussed in part iii. Discussion. The following components were used for the optical sensor array: six 5 mm round infrared LED lamps [1] for the IR emitters; six silicon PIN photodiodes [2] for the IR sensors; and three LM358 op-amps [3] to convert the current from the IR sensor to a voltage for an Arduino (sensor Arduino).

i. Design

The IR emitters will emit infrared light at the floor and the IR sensors will measure how much of this light has reflected back. For a white surface most of the infrared light emitted by the IR emitter will reflect back to the IR sensor but, for a black surface most of the infrared light will be absorbed. The IR sensor outputs a current that is proportional to the amount of infrared light it receives. Therefore, if a pair of IR emitter and IR sensor are above a white surface the IR sensor will produce a relatively large current and if they are above a black surface, such as a line, the sensor will produce a relatively small current. The sensors response is not a binary on-off, instead it is a continuous range from a minimum value to a maximum value, as the surface below reflects more infrared light. This will allow for greater accuracy as the relative position of each sensor to the line can be calculated by the current produced.

However, the sensor Arduino's analogue inputs do not accept a current and instead require a voltage from 0-5 V. So, to convert the current of the IR sensor, it could be passed through a transimpedance amplifier, Figure 1, or a comparator circuit, Figure 2**.** For a transimpedance amplifier the current will be converted to a voltage and scaled up, as the IR sensor outputs in μA, to a range of 0-5 V. The output of the amplifier will then be connected to one of the sensor Arduino's analogue inputs. The V_{Out} in Figure 1 is directly proportional to the value of R_f – the feedback resistor. A value of 1 MΩ was chosen since the output voltage was in μA and so this was high enough to scale the input to 0-5 V but low enough where it did cause small current to instantly clip to 5 V.

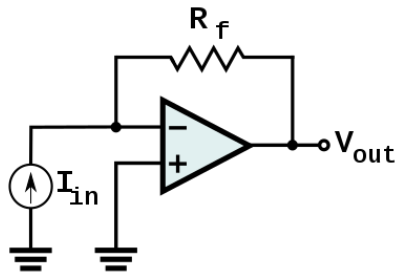

Figure 1 – An example of a transimpedance amplifier [4]

The circuit shown in Figure 2 is an example of a non-inverting comparator circuit. "When V_{IN} is greater than V_{REF}, … [V_{OUT}] will saturate towards the positive supply rail (+V_{cc}) and when V_{IN} is less than V_{REF} … [V_{OUT}] will saturate towards the negative supply rail (-V_{cc})" [5]. First V_{REF} is set to a value slightly more than the voltage produced, via the op-amp, when the IR sensor is above the line – i.e. its minimum value. Then when the IR sensor goes above the line V_{IN} will be less than V_{REF} and so V_{OUT} will saturate towards the negative supply rail (-V_{cc}), which will be connected to a digital pin on the Arduino.

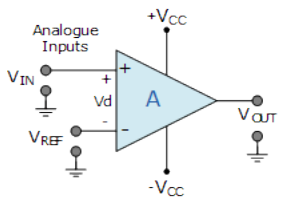

Figure 2 - Schematic of comparator circuit [5]

Six pairs of IR sensors and IR emitters were spaced evenly along the long edge of a rectangular stripboard. Also, the pair were oriented vertically instead of horizontally. As a result, more pairs could be included in the array and so line detection would be more accurate. If there were any more than six pairs, interreference, between each pair, would become a problem due to their close proximity. The pairs of IR sensors and IR emitters would be wired on one strip board while the op-amps on another – to simplify the wiring. The stripboards would then be place back to back, with a small separation, and held in with nuts and bolts. A 3D printed bracket, holding the optical sensor array, was then screwed in to the front of the car.

For wiring, one side of the IR sensors were connected to ground while the other side were connected to a input of separate op-amps. The IR emitter was connected to ground and 5 V and thus would always be emitting infrared light whilst the car was turned on. The op-amp was connected to ground and 5V, with the non-inverting input connected to ground and the inverting input connected to the output of an IR sensor. A resistor was the connected from the inverting input to the output of the op-amp to act as the feedback resistor - R_f in Figure 1. The output of each op-amp would then be connected to an analogue input of the sensor Arduino.

Header pins were soldered on to stripboard that the sensor Arduino would slot in to. This was so that the sensor Arduino could be removed in the case of a fault. The sensor Arduino and the car Arduino, which control the motors, were then connected via I^2C, to allow for wired communication. This would provide the sensor Arduino with 5 V and ground, which it could then pass on to the optical sensor array. The outputs of the op-amps were then soldered to the analogue inputs of the sensor Arduino – except A4 and A5 as they were being used for I^2C. This is what will be referred as the sensor module in later chapters.

ii. Results

The results obtained in Figure 3 were achieved by placing the entire optical sensor array above the line; recording the sensor Arduino's `analogRead()` of each IR sensor; and repeating for when the optical sensor array were not above the line. These values were then converted to the voltages by dividing by 1023, as that is the maximum analogue value, and then multiplying by five, as 1023 represents 5 V.mDue to the large headroom, that is the difference between supply voltage and the maximum output voltage, of the op-amp, the output voltage clipped at around 3.3 V, as seen in Figure 3, instead of the 5 V that can be accepted by the analogue inputs on the Arduino.

IR Sensor	Voltage when above the line (V)	Voltage when NOT above the line (V)
LLL	0.59	3.37
LL	0.64	3.13
L	0.78	3.47
R	0.44	3.37
RR	1.91	3.47
RRR	0.24	2.10

Figure 3 – Table for the minimum and maximum values for each IR sensor, with LLL meaning the left most sensor as seen if one was looking from above and behind the car

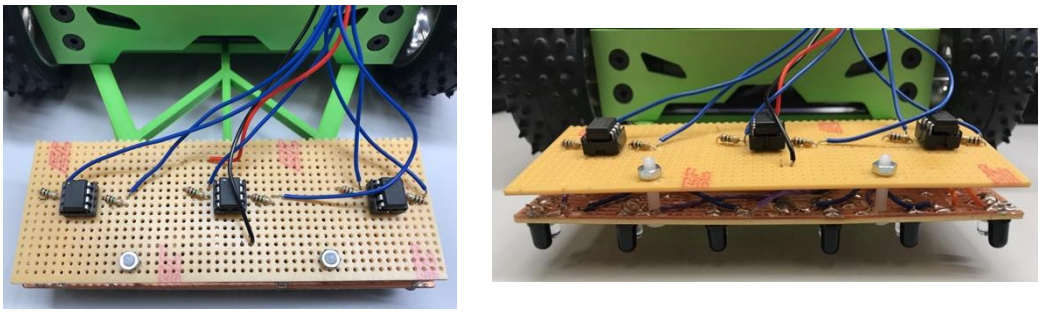

Figure 4 – Optical sensor array mounted on to the front of the car with a bracket

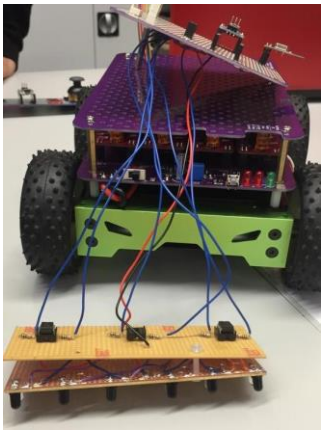

Figure 5 – Layout of the optical sensor array and sensor module connection method

When the comparator circuit in Figure 2 was given a low voltage, V_{OUT} was 0 V and when the comparator was given a high voltage, V_{OUT} was 5 V. V_{REF} would vary for each sensor due to manufacturing differences.

iii. Discussion

The voltage when above the line for the RR IR sensor and the voltage when NOT above the line for the RRR IR sensor, from Figure 3, appear to be anomalous. However, one would expect both voltages for each sensor to be anomalous if the fault was with the circuit thus, they are probably errors in measurement. These values should be repeated and if there is no significant change in their voltages, then there is a fault with their particular circuits. These anomalies would have decreased the line following performance and could explain why in Chapter 4. PID Control Scheme the car struggles to follow the line at times. Even though there were headroom limits on the op-amp outputs, the IR sensors still functioned correctly by giving low readings when above the line and higher readings as they got further away from the line, but the precision had been decreased due to the decreased range. This is reflected in the sensor Arduino code, appendix B, where the max values of each sensor for the `map()` function is around 700, with some as low as 400, instead of the maximum of 1023 – which represents 5 V.

The comparator circuit is an example of a 1-bit ADC due to its binary two states – 5 V or 0 V. This conversion from analogue to digital removes information from the output of the IR sensor. If the comparator circuit was used to interface between the optical sensor array and the sensor Arduino, there would be less accuracy. Also, the car would deviate further off course before it would react to the line, as the sensor only outputs a change when it is completely over the line – and only one sensor can be completely over the line at any single moment. Thus, one can say that the rate at which the car responds to line changes would have been decreased.

4. PID Control Scheme

The aims for this chapter are to implement, within the Arduino code, a weighted average formula to calculate the position of the car with respect to the line and a PID control formula to calculate the change in the motor speeds required for the car to follow the line. These two formulae are then to be combined in to one so that the car can respond to the changes in the position of the line by adjusting the speed of the motors.

i. Design

The PID control scheme is a type of control loop, with the equation stated in Figure 6. This control loop is relatively simple when compared others since you only need to adjust three constants and the mathematical equation can be represented in a way that most people can understand. The first term is the proportional term and it represents the current error. The second term is the integral term and it represents past errors. The third term is the differential term and it represents potential future errors.

$$u(t) = K_\mathrm{p} e(t) + K_\mathrm{i} \int_0^t e(t')\,dt' + K_\mathrm{d} \frac{de(t)}{dt},$$

Figure 6 – PID control equation

Sensor Module Code

The Arduino code on the sensor module, in appendix B, first declares the analogue pins each sensor is connected to and two floating point arrays. The first array is `SReading[]`, which contains the analogue output of every sensor – starting with the left most sensor and ending with the right most sensor. The next array is `SWeight[]`, which contains, in the same order as `SReading[]`, the weighting of each sensor. This weighting is calculated by centring the car on the line and then measuring the distance from each sensor to the line in millimetres. Positive values indicate the sensor is to the right of the line and negative values indicate the senor is to the left of the line. Next the other weighted average and PID variables are declared, with the variables involved in summation set to zero.

Assuming the car is in line following mode, see Chapter 5. Remote Control and HMI, `ReadSensorValues()` is called, which reads in the analogue value of the sensors. The minimum and maximum values were determined by printing all the sensor values, using the `PrintAllSensorAndSpeedInformation()` function and then placing the sensor over the line for the minimum value, and not over the line for the maximum value, and recording the values each time. The minimum and maximum values of the sensors are then used as the range that each sensor will be constrained to – note that every sensor has a different range due to differences in sensitivity of each sensor. The values are constrained so that the `map()` function does not receive values outside of its specified range and thus, it will stop any potential errors from occurring. The senor values are then mapped to an integer from zero to nine, with zero representing the sensor completely over the line and nine representing the sensor completely not over the line.

Next `CalculateLFSpeedAndDirection()`, containing the weighted average formula in Figure 7, PID formula and the direction the car should turn to stay on the line, is called. First the `WeightedTotal` is taken by looping through all the `SWeight[]` and `SReading[]` and taking a summation of the product. At the same time the `SReading[]` of all the sensors are summed together. When the car is centred on the line the weighted total will be equal to zero and thus the current error will also be zero. However, when the car moves to the left, the sensors on the right will be above the line and will give a lower sensor reading. Thus, the weighted total and the current error will become negative. Therefore, a negative error will mean the car should turn right and a positive error meaning the car should turn left.

```
for (int i=0; i<=5; i++) {
   WeightedTotal += SWeight[i] * SReading[i];
   SensorTotal += SReading[i];
}

CurrentError = WeightedTotal/SensorTotal;
```

Figure 7 – Weighted average and current error formulae

Next each PID variable is defined, with `Proportional` being the current error; `Integral` being the summation or running total of all errors; and `Differential` being the difference between the current and previous error – i.e. how quickly the error has changed from the previous error. Then the `PIDoutput`, as stated in Figure 6, is calculated by summing the products of the PID variables and their respective PID constants – Kp, Ki and Kd – which have been sent from the RF remote, refer to Chapter 5. Remote Control and HMI. Direction is determined by whether the `PIDoutput` is positive or negative and is represented by a L, R or N for left, right or neutral respectively. As explained in the previous paragraph a negative value will represent that the car has moved to the left of the line, so the line is to the right of the car and thus, the car should move to the right – R – to get the line back in to the center. A positive value will represent that the car has moved to the right of the line, so the line is to the left of the car and thus the car should move to the left – L – to get the line back in the center. The neutral direction was given a range, rather than a single value of zero, to smooth out the turning of the car and was tuned with trial and error. Thus, if `PIDoutput` is bigger than -15 but smaller than 15 then the car will continue moving forwards in a straight line and will only turn when the `PIDoutput` exceeds this range. Speed is calculated by making the `PIDoutput` positive, if applicable, and then constraining the number to a value from 0 to 255, as the motors have maximum value of 255.

```
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
Button = 0    9.00   0.00   8.00   9.00   8.00   9.00   Direction = L   Speed = 68
```

Figure 8 – Output of PrintAllSensorAndSpeedInformation() function

Figure 8 shows an example of what has been calculated up to this point, via `PrintAllSensorAndSpeedInformation()` function, which prints this information to the serial monitor. From left to right, the button value of zero shows that the car is in line following mode – refer to Chapter 5. Remote Control and HMI. The next six integers are the sensors values, with the left most value representing the left most sensor and the right most value representing the right most sensor – as one would observe if they were looking from above and behind the car. In this example one could infer the line's position as being to the left of the car and more precisely as directly under the second from the left sensor. The direction of 'L' confirms this conclusion. The direction and speed are then sent to the car Arduino via I^2C.

Car Code

The Arduino code controlling the cars motors, in appendix C, first defines that the `BaseSpeed` is 48. The faster the base speed, the less time the car will have to process line changes and will cause the error to increase. This will mean more correction will be needed, which will increase the likelihood of the car veering off course. Whilst a lower `BaseSpeed` will result in increased time to complete a course. Thus, 48 was a good compromise.

Again, assuming the car is in line following mode, the button value will be zero and so the direction and speed are read in via I^2C. Next the `PIDMotorControl()` function is called. Then the `Sum` is calculated which will increase the further away from the car gets from the line. Next a switch statement on the direction is created. For `case 'N'`, the car goes forwards at the base speed. For `case 'R'`, the right motors are set to spin backwards; the left motors are set to spin forwards; and all motors are set to the sum speed. This causes the car to rotate, centered on the car, to the right. For `case 'L'`, the left motors are set to spin backwards; the right are set to spin forwards; and all motors are set to the sum speed. This causes the car to rotate, centered on the car, to the left. In both case the car rotates until `PIDoutput` is back in the range of -15 and 15.

ii. Results

Initially a sum and difference approach were used to turn the car, where both motors would be going in the forwards direction, but one motor would have the sum value and the other the difference value. In practice when Kp was low the car reacted to weakly to the line. However, when Kp was increased the car would speed up and veer off the course at high speeds. Therefore, an approach of the car travelling until a certain threshold of `PIDoutput` had been exceeded and then rotating the car back within the threshold. This approach was more reliable.

Due to the lack of time and the unpredictability of the integral and differential components of the equation, only the proportional part of the PID equation was used. However, the car could still traverse a course – most of the time.

iii. Discussion

The weighted average formula and PID formula worked as intended and were assimilated in to a single function to give both a direction and speed.

The approach of rotating then car, instead of turning it, required a range of values instead of a single value of zero, for when the car should be going straight line. This is because for small values of `PIDoutput` and thus distances from the line, the car can never get back to the line exactly. This would cause the car to get stuck as it tried to rotate with an incredibly small speed. Therefore, the range acted as a dead zone and only when the distance from the line was large enough, did the car then try to rotate back on to the line. The final range of -15 to 15 is still susceptible to getting stuck but works in most cases.

The car could make it around the course most of the time but would sometimes start to oscillate when going around a corner. This is a common symptom of just using the proportional component. This is because the car is just reacting to what is happening right now and is oblivious to the previous errors, which is what the integral and differential component deal with.

Currently the calibration, of black and white sensor readings, is hard coded in to the sensor Arduino code. This means if the course was to change, in terms of black and white values, many variables would have to be changed manually. Instead what could have been implemented was a function within the `void setup()` of the RF remote code – so it only occurs once. This function would display a message on the LCD, to the user, that all sensors be placed above the white surface, and the user would confirm they have done this with a button press. The remote Arduino would then ask the senor Arduino to record the current analogue values of the sensors as that sensors maximum value. Then the sensor Arduino would send a confirmation message back to the remote to say it has completed that. Next the LCD on the RF remote would display a message to tell the user to put all sensors on the black surface, and the user would confirm they have done this with a button press. The remote would ask the senor Arduino to record the current analogue values of the sensors as that sensors minimum value. The sensor Arduino would send a confirmation message that it has been completed and the remote Arduino code would continue as described in Chapter 5. Remote Control and HMI. The minimum and maximum values would be used in the constrain and mapping functions described previously. This would allow the car to traverse multiple tracks with varying black and white values.

5. Remote Control and HMI

The objectives for this chapter were if the car was in line following mode, then the PID constants should be displayed on the screen of a handheld remote; adjusted via an input method on the remote; and then sent to the sensor Arduino. If the car was in RF remote mode, then an MPU6050 attached to the car should send data to the screen of the handheld remote and the car should no longer be following the line and instead be controlled via some human machine interface attached to the remote. A way of being able to switch from line following mode to RF remote mode also needed to be implemented.

i. Design

The nRF24L01+ modules were selected due to their inexpensive [6] and versatile design. The modules operate over the 2.4 GHz ISM radio frequency band; with data rates from 250 kbps to 2 Mbps; 125 separate channels; and each module can listen to six other modules simultaneously [7]. This will allow for lots of flexibility in what can be achieved now and in the future.

RF Remote Module

The remote control that will control the car (RF remote) was designed with the joysticks to act as the human to machine interface due to their common use in existing remote-control cars. They were placed horizontally opposite from each other, so to allow intuitive control of the car, and four buttons were also added near the bottom of the remote if more input methods were needed in the future. The 16x2 LCD screen was screwed in to the top of the remote which will display any relevant information. A rotary encoder was chosen to increment the PID constants for its ability to turn without end and was attached to the upper centre of the remote. An Arduino Nano, aka the remote Arduino, was soldered in the centre to make it easy to wire other components to it. The nRF24L01+ module and the remote Arduino will slot in to header pins, just in case the either module was to become faulty. The nRF24L01+ module will be on the back of the remote, so the module will be out of the way and be pointing towards the car.

A portable batter bank will power the remote Arduino via its USB port. The remote Arduino would subsequently power the rest of the board. The buttons were grounded but not connected to any digital pins as they have no use currently. From the remote Arduino both joysticks were connected to 5 V and ground whilst the nRF24L01+ module was connected to 3.3 V and ground [7], represented by red and black wires. The analogue outputs of the joysticks were connected to the analogue inputs A0 to A3, represented by blue wires. SPI connections from the nRF24L01+ to the Arduino Nano were represented by purple wires and the exact wiring diagram is displayed in Figure 1.

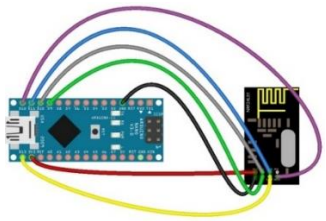

Figure 9 – Wiring diagram for nRF24L01+ with the Arduino Nano [8]

Sensor Module and Car Modifications

An nRF24L01+ module was added to the existing sensor board and will receive the data from the remote over the specific channel in the 2.4 GHz ISM radio frequency band and then send that information to the sensor Arduino. The nRF24L01+ module is wired the same as Figure 9. The MPU6050 was then attached to the car

Decoupling Capacitors

Between the 3.3 V and ground on the nRF24L01+ decoupling capacitors of 10 μF were added for when the transmission power would be set to high. They are also known as bypass capacitors since they bypass the power source when needed [9]. This is because at higher transmission powers fast changing current spikes and voltage transients become more common which would cause errors in information transmission over the 2.4 GHz ISM radio frequency band. Since decoupling capacitors “oppose quick changes of voltage, if the input voltage suddenly drops,

the capacitor provides the energy to keep the voltage stable. Similarly, if there is a voltage spike, the capacitor absorbs the excess energy" [9]. Thus, the capacitor keeps the input voltage stable.

RF Remote Mode and Line Following Mode

The RF remote code starts with the remote control deactivated and the car in line following mode. This is done by initialising the first value of `ButtonXYKpKiKdValues[]` to zero. The LCD attached to the remote, displays the mode the car is in, which in this case is "Line Following Mode". Kp, Ki and Kd values are displayed, with their corresponding values, on the three lines below.

When either one of the X or Y joysticks, right and left positionally, are moved in directions away from their rest point, left or right and up or down respectively, RF remote mode is activated. This will cause the car to now take commands from the RF remote whilst also sending MPU6050 data back to de displayed on the LCD. The was limited screen space thus only the forwards and backwards acceleration (`MyMPU6050.getAccX()`) and the pitch of the car (`MyMPU6050.getAngleY()`) were chosen to be sent to the remote to be displayed.

RF Remote Code

The Arduino code on the RF remote – refer to Appendix A – first defines the analogue pins that the joysticks and the pins of the nRF24L01+ modules CE and CSN pins are connected to. The first value of the `ButtonXYKpKiKdValues[]` array defines whether the car is in line following mode or in RF remote mode and is denoted with a zero or one respectively. This binary value is sent to all Arduinos to set them all to the same mode. The order of the different values of the `ButtonXYKpKiKdValues[]` array is reflected in the name of the array. The RF radio and serial monitor are started in `void setup()`.

If the car is in line following mode, then `ReadRotaryEncoder()` is first called. This checks if the button on the rotary encoder has been pressed and increments its value by one if the check returns true. Each button press selects the next PID constant on the LCD screen to be changed via a switch statement. To cycle back to the first PID constant, the `ButtonCount` value must exceed three, where it is reset back to one. If the rotary encoder position has changed then the currently selected PID constant needs to be incremented appropriately. `LCDLineFollowingMode()` displays the current mode at the top and then updates the PID constants. The `ButtonXYKpKiKdValues[]` array is then sent to the sensor Arduino to be used for the PID calculations, as long as the button value is still zero.

However, before that, at the beginning of `void loop()` the function `ReadJoyStickValuesAndCheck()` is called (Figure 10). When at rest JoyX gives an analogue value of 500 however due to random fluctuations a buffer of five is given either side of 500 so that no unintended movements occur. The values are constrained so that when the values are mapped between zero and 255, the number given to the mapping function is not outside of the range, which will prevent errors.

```
/*Analogue inputs are read in and converted to a number from -255 to 255*/
  if (analogRead(JoyX) > 505)       /*Right*/
    ButtonXYKpKiKdValues[1] = map(constrain(analogRead(JoyX), 505, 830), 505, 830, 0, 255);

  else if (analogRead(JoyX) < 495)  /*Left*/
    ButtonXYKpKiKdValues[1] = 0 - map(constrain(analogRead(JoyX), 60, 505), 505, 60, 0, 255);

  else                              /*Straight*/
    ButtonXYKpKiKdValues[1] = 0;
```

Figure 10 – RF remote code for reading in the right joystick (JoyX) which determines left and right movements

If the car is supposed to go right, then the X value will be positive and if the car is supposed to go left, then the X value will be negative. A zero-value represents a neutral direction. Similarly, if the car is supposed to go forwards, then the Y value will be positive and if the car is supposed to go backwards, then the Y value will be negative. Here a zero-value indicates the car should be stationary. This approach allows the sensor Arduino to determine the direction of the analogue value whilst not having to send any characters.

The X and Y values are checked, Figure 11, to see if at least one of them are non-zero. If they are the button state is set to one, which means that the car will switch from line following mode to RF remote mode.

```
if ((ButtonXYKpKiKdValues[1] != 0) || (ButtonXYKpKiKdValues[2] != 0)
      ButtonXYKpKiKdValues[0] = 1;
```

Figure 11 – RF remote code check for if the joysticks have been moved

In RF remote mode, the RF remote first starts listening for any MPU6050 data. If there is some it reads the values in to an array. Then `LCDRFMode()` is called which changes the top line of the LCD to "Remote Control Mode"; displays the current values of the X and Y joysticks; and displays the current acceleration and pitch of the car. Then the RF remote stops listening so that the `ButtonXYKpKiKdValues[]` array can be sent to the sensor Arduino.

Figure 12 – ButtonXYKpKiKdValues[] array that is being sent to the sensor Arduino from the RF remote Arduino when in line following mode (left) and in RF remote mode (right)

Sensor Module Code

The Arduino code on the sensor module – refer to Appendix B – has an identical `void setup()` to the RF remote except that the sensor also joins the I²C bus with the `Wire.begin()` function as the sensor. This is so that it can send data to the car Arduino. If the car is line following mode, that is `ButtonXYKpKiKdValues[0] = 0`, the PID control scheme outlined in 4. PID Control Scheme will be executed.

```
if (ButtonXYKpKiKdValues[1] > 0)
     XYDirection[0] = 'R';

 else if (ButtonXYKpKiKdValues[1] < 0) {
     XYDirection[0] = 'L';
     ButtonXYKpKiKdValues[1] *= -1;

 } else
     XYDirection[0] = 'N';
```

Figure 13 – Sensor module code for determining the direction of the right joystick (JoyX)

However, if `ButtonXYKpKiKdValues[0] = 1`, then `CalculateRFSpeedAndDirection()` is called – Figure 13. This examines the values for both joysticks and determines the direction, which is represented with a single character. I²C can only send data packets of one byte at a time. One byte is equal to a number from zero to 255, unsinged int, or if the int is signed then the range is -127 -to 127. Byte reconstruction is possible but, in an effort keep the I²C code simple, a decision was made to instead keep all values unsigned and send accompanying direction values, in the form single characters, to the car. Next `SendRFSpeedAndDirectionToCar()` is called the values and direction of both joysticks are sent to the car via I²C – note that the first value sent is the button state, as this will disclose to the car what the next values will be.

```
R X Value = 249 F Y Value = 251
R X Value = 249 F Y Value = 251
R X Value = 249 F Y Value = 251
R X Value = 249 F Y Value = 251
R X Value = 249 F Y Value = 251
R X Value = 249 F Y Value = 251
```

Figure 14 – The joystick values sent to the car Arduino from the sensor Arduino

Car Code

The Arduino code controlling the cars motors – refer to appendix C – calls the `receiveEvent()` function every time it receives I^2C information. The first value read is the button value and with that information the car understands what it should be reading in next. If this value is zero then the car enters the PID control scheme code, which has been previously explained in 4. PID Control Scheme.

However, if the button value is one then the `RFMotorControl()` function is called. It first works out the difference between the two joystick values, with an included error checking for when the difference could be negative, in which case the difference will changed to zero. The first switch statement determines the direction the motors should spin – forwards or backwards. The next switch statement, which is nested within the forwards and backwards directions of the first switch statement, determines the direction the car should be going and thus, which motor should be slowed down by giving it the difference value. This difference values allow the car, which does not have wheels that turn, to mimic the movements of a car with front wheels that turn. One can infer from this code that if one was to just move the right joystick, X joystick, the car would switch to RF remote mode but wouldn't move as the Y joystick is still stationary. Thus, this acts as an emergency stop feature when the car is in line following mode.

ii. Results

When checking the functionality of the remote, in Figure 15, we found that a few of the wires from the Arduino Nano to the RF module were in the wrong places. Such as CS and CE on the RF module, Figure 16, should be connected to D10 and D9 on the Arduino respectively, but were instead connected to D9 and D8. The caused the remote to not function correctly. After these soldering mistakes was addressed the remote worked as intended.

Figure 15 – RF Remote in line following mode and remote-control mode

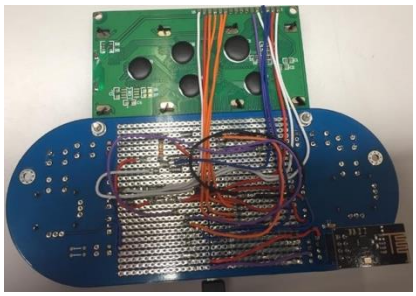

Figure 16 – The back of the RF remote

Figure 17 – MPU6050 sensor location on the car

The modification to the sensor module and addition of the nRF24L01+ caused the board to stop working – Figure 18. After immense troubleshooting it was found that the MISO and MOSI needed to be to be swapped.

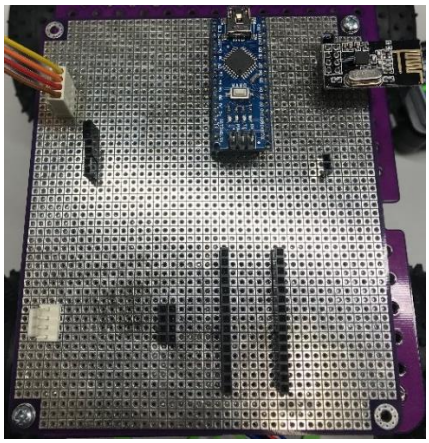

Figure 18 – Modified sensor module

The initial code for the remote would cause the readings for the joysticks to swap with each other by either the right joystick/X input being read twice; or the left joystick/Y input being skipped; or sometimes a random 120 would appear. This caused the car to move in unpredictable manner as the role of the either joystick would switch with no warning. To fix this the values were sent as a part of an array and stopped all irregular behaviour when sending data over RF and the car moved as intended.

iii. Discussion

When the car was in RF remote mode the car would stop following the line and take commands from joysticks on the remote. Data was sent from the MPU6050 and displayed on the RF remote's screen, while also displaying the current joystick values. This fulfilled the objectives at the beginning of the Chapter.

The method to switch from line following mode to RF remote mode was a success but returning to line following mode had its drawbacks. To leave line following mode and enter RF remote mode the joysticks had to be moved away from their rest point but, the method to return to line following mode was to press the reset button on the Arduino, as the code defaults to line following mode. An improvement to this method would be to have a separate button to switch to and from either mode. This would remove the need to re-enter the PID constants every time one enters line following mode. However, the PID constant were successfully displayed on the remote screen and could be incremented one at a time.

6. Conclusion

For the optical sensor array, some values in Figure 3 appear to be anomalous. However, one would expect both voltages for each sensor to be anomalous if the fault was with the circuit thus, they are probably errors in measurement and should be repeated. These anomalies would have decreased the line following performance and could explain why in Chapter 4. PID Control Scheme the car struggle to follow the line at times. Also, despite the headroom limits on the op-amp outputs, the IR sensors still functioned correctly by giving low readings when above the line and higher readings as they got further away from the line, but the precision had been decreased due to the decreased range.

The comparator circuit is an example of a 1-bit ADC due to its binary two states – 5 V or 0 V. This conversion from analogue to digital removes information from the output of the IR sensor output. If the comparator circuit was used to interface between the optical sensor array and the sensor Arduino, there would be less accuracy.

The weighted average formula and PID formula worked as intended and were assimilated in to a single function to give both a direction and speed. This accomplished the objective of being able to calculate the position of the car relative to the line because the speed increased as the car got further away from the line.

The approach of rotating then car, instead of turning it, required a range of values instead of a single value of zero, for when the car should be going straight line. This is because for small values of `PIDoutput` and thus distances from the line, the car can never get back to the line exactly. This would cause the car to get stuck as it tried to rotate with an incredibly small speed. Therefore, the range acted as a dead zone and only when the distance from the line was large enough, did the car then try to rotate back on to the line. The final range of -15 to 15 is still prone to the car getting stuck for specific corners but works for most corners. The car could make it around the course most of the time but would sometimes start to oscillate when going around a corner.

Currently the calibration, of black and white sensor readings, is hard coded in to the sensor Arduino code. This means if the course was to change, in terms of black and white values, many variables would have to be changed manually. Instead what could have been implemented was a function within the `void setup()`, so it only occurs once, on the RF remote code which would guide the user through a calibration scheme. This would allow the car to traverse multiple tracks with varying white and black values.

The objective of modifying the HMI from last week and converting it in to the form of a wireless handheld remote was achieved. Also, when the car was in RF remote mode the car would stop following the line and take commands from joysticks on the remote. Data was sent from the MPU6050 and displayed on the RF remote's screen, while also displaying the current joystick values which fulfilled the objective of sending telemetry data back to the remote.

The method to switch from line following mode to RF remote mode was a success but returning to line following mode had its drawbacks as it erased the PID constants. However, the PID constants were successfully displayed on the remote screen and could all be incremented one at a time.

The full capabilities of the car, by the end of the two weeks, were that it could follow a line while from the remote one could see the current PID constants while also being able to adjust them. Also, if the joysticks were moved then the car would switch to RF remote mode. Here the car would take commands from the joysticks on the RF remote and the car would send telemetry data back to the remote where it would be displayed on an LCD. To return to line following mode the reset button on the Arduino would have to be pressed. However, this would clear any of the PID constants that may have been previously set.

7. References

[1] "Farnell," [Online]. Available: http://www.farnell.com/datasheets/1697427.pdf?_ga=2.231674583.931628658.1549720259-1968236617.1549720259.

"Vishay," [Online]. Available: https://www.vishay.com/docs/81503/bpv10nf.pdf.

[2]

[3] "LM358 Op-Amp," [Online]. Available: http://www.ti.com/lit/ds/symlink/lm158.pdf.

[4] [Online]. Available: https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwiFw6257K7gAhWbDmMBHXDFDRYQjRx6BAgBEAU&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FTransimpedance_amplifier&psig=AOvVaw39QBI_IeAO2RcHo4u16dVK&ust=1549808422993659.

[5] "Comparator Circuit," [Online]. Available: https://www.electronics-tutorials.ws/opamp/op-amp-comparator.html.

[6] [Online]. Available: https://www.amazon.co.uk/Professional-NRF24L01-Antenna-Wireless-Transceiver/dp/B010N32SGM/ref=as_li_ss_tl?ie=UTF8&qid=1533729521&sr=8-3&keywords=NRF24L01&th=1&linkCode=sl1&tag=howtomuk-21&linkId=19b321c5950665d599608f11680affbf&language=en_GB.

[7] "Nordic Semiconductor (first doc)," [Online]. Available: https://www.nordicsemi.com/DocLib/search?query=nrf24l01.

[8] "Laser Harp," [Online]. Available: http://www.laserharp.com/projects/project-b.

[9] "Capacitor Guide," [Online]. Available: http://www.capacitorguide.com/coupling-and-decoupling/.

Appendix A

(Remote Code)

```
/*Libraries*/
#include <SPI.h>
#include "RF24.h"
#include <LiquidCrystal.h>
#include <Encoder.h>
#include "Remote_Template.h"

/*Pin Declarations*/
const int JoyX = A0, JoyY = A3, RS = 7, Enable = 6, D4 = 5, D5 = 4, D6 = 3, D7 = 2;
RF24 radio (9, 10);
LiquidCrystal MyLCD (RS, Enable, D4, D5, D6, D7);
Encoder MyEncoder (18, 19);

/*Global Variables*/
int ButtonXYKpKiKdValues[6] = {0,0,0,0,0,0}, GyroData[2] = {0,0};
int ButtonCount = 1, NewPosition;

void setup() {
  radio.begin();                              /*Start the radio module*/
  radio.setPALevel(RF24_PA_LOW);              /*Set the transmission power to low*/
  radio.setDataRate(RF24_250KBPS);            /*Set the speed to 250kbps*/
  radio.setChannel(CHANNELNUMBER);            /*Set the channel*/
  radio.openWritingPipe(rfAddresses[0]);      /*Transmit to the remote address*/
  radio.openReadingPipe(1, rfAddresses[1]);   /*Recieve sensor addressed packets*/
  radio.setPayloadSize(PAYLOADSIZE);          /*Set the number of bytes for the data packet*/

  Serial.begin (9600);

  MyLCD.begin(16, 4);
}


void loop() {
  ReadJoyStickValuesAndCheck();

    if (ButtonXYKpKiKdValues[0] == 0) {
      ReadRotaryEncoder();
      LCDLineFollowingMode();

    } else {
      radio.startListening();
      radio.read(&GyroData, sizeof(GyroData));
      LCDRFMode();
    }

//  PrintAllInfo();
  radio.stopListening();
  radio.write(&ButtonXYKpKiKdValues, sizeof(ButtonXYKpKiKdValues));
  delay(1);
}

void ReadJoyStickValuesAndCheck() {
/*Analogue inputs are read in and converted to a number from -255 to 255*/
  if (analogRead(JoyX) > 505)       /*Right*/
    ButtonXYKpKiKdValues[1] = map(constrain(analogRead(JoyX), 505, 830), 505, 830, 0, 255);

  else if (analogRead(JoyX) < 495)  /*Left*/
    ButtonXYKpKiKdValues[1] = 0 - map(constrain(analogRead(JoyX), 60, 505), 505, 60, 0, 255);

  else                              /*Straight*/
    ButtonXYKpKiKdValues[1] = 0;
```

```
  if (analogRead(JoyY) > 508)       /*Backwards*/
    ButtonXYKpKiKdValues[2] = 0 - map(constrain(analogRead(JoyY), 508, 890), 508, 890, 0, 255);

  else if (analogRead(JoyY) < 495)  /*Forwards*/
    ButtonXYKpKiKdValues[2] = map(constrain(analogRead(JoyY), 90, 495), 495, 90, 0, 255);

  else                              /*Stationary*/
    ButtonXYKpKiKdValues[2] = 0;

  if ((ButtonXYKpKiKdValues[1] != 0) || (ButtonXYKpKiKdValues[2] != 0)) /*If the joysticks have been*/
      ButtonXYKpKiKdValues[0] = 1;                                      /*moved then switch to remote*/
}                                                                        /*control*/

void ReadRotaryEncoder() {
  if (analogRead(A6) == 0) { /*When button is pressed analogue read gives a 0 value*/
      ButtonCount++;
      delay(100);

  } else
     ;

  if (ButtonCount > 3)       /*Reset value back to Kp when button is pressed when at Kd*/
      ButtonCount = 1;

  switch (ButtonCount) {
       case 1: /*Kp Values*/
           NewPosition = MyEncoder.read();
           if (NewPosition != ButtonXYKpKiKdValues[3])
              ButtonXYKpKiKdValues[3] = NewPosition;
           break;

       case 2: /*Ki Values*/
            NewPosition = MyEncoder.read();
            if (NewPosition != ButtonXYKpKiKdValues[4])
                ButtonXYKpKiKdValues[4] = NewPosition;
            break;

       case 3: /*Kd Values*/
            NewPosition = MyEncoder.read();
            if (NewPosition != ButtonXYKpKiKdValues[5])
                ButtonXYKpKiKdValues[5] = NewPosition;
            break;

       default:
          Serial.println("Button Count Error");
          break;
    }
}

void LCDLineFollowingMode () {
  MyLCD.setCursor(0, 0);
  MyLCD.print("Line Following Mode");

  MyLCD.setCursor(0, 1);
  MyLCD.print("Kp = ");
  MyLCD.print(ButtonXYKpKiKdValues[3]);

  MyLCD.setCursor(0, 2);
  MyLCD.print("ode Ki = ");
  MyLCD.print(ButtonXYKpKiKdValues[4]);

  MyLCD.setCursor(0, 3);
  MyLCD.print("    Kd = ");
  MyLCD.print(ButtonXYKpKiKdValues[5]);
}
```

```
void LCDRFMode () {
  MyLCD.setCursor(0, 0);
  MyLCD.print("Remote Control M");

  MyLCD.setCursor(0, 1);
  MyLCD.print("XValue = ");
  if (ButtonXYKpKiKdValues[1] == 0)
    MyLCD.print("0   ");
  else
    MyLCD.print(ButtonXYKpKiKdValues[1]);

  MyLCD.setCursor(0, 2);
  MyLCD.print("ode YValue = ");
  if (ButtonXYKpKiKdValues[2] == 0)
    MyLCD.print("0   ");
  else
    MyLCD.print(ButtonXYKpKiKdValues[2]);

  MyLCD.setCursor(0, 3);
  MyLCD.print("    Acc = ");
  MyLCD.print(GyroData[0]);

  MyLCD.print("  Angle = ");
  MyLCD.print(GyroData[1]);
}

void PrintAllInfo () {
  Serial.print("Button = ");
  Serial.print(ButtonXYKpKiKdValues[0]);
  Serial.print("   X Value = ");
  Serial.print(ButtonXYKpKiKdValues[1]);
  Serial.print("   Y Value = ");
  Serial.print(ButtonXYKpKiKdValues[2]);
  Serial.print("   Kp = ");
  Serial.print(ButtonXYKpKiKdValues[3]);
  Serial.print("   Ki = ");
  Serial.print(ButtonXYKpKiKdValues[4]);
  Serial.print("   Kd = ");
  Serial.println(ButtonXYKpKiKdValues[5]);
}
```

Appendix B

(Sensor Code)

```
/*Libraries*/
#include <SPI.h>
#include <Wire.h>
#include <MPU6050_tockn.h>
#include "RF24.h"
#include "Sensor_Template.h"

MPU6050 MyMPU6050 (Wire);
long timer = 0;

/*Pin Declarations*/
const int LLLSensor = A0, LLSensor = A1, LSensor = A2, RSensor = A3, RRSensor = A6, RRRSensor = A7;
RF24 radio(9,10);

/*Global Variables*/
float SReading[6] = {0,0,0,0,0,0};          /*Sensor Readings Array*/
float SWeigh[6] = {-60,-40,-15,15,35,60};   /*Weighting for each sensor in mm*/

float WeightedTotal = 0, SensorTotal = 0, PreviousError = 0, CurrentError;      /*PID variables*/
float Proportional, Integral = 0, Differential, PIDoutput;

int ButtonXYKpKiKdValues[6], int GyroData[2], Speed;      /*RF remote array*/
char XYDirection[2], Direction;

void setup() {
  radio.begin();                              /*Start the radio module*/
  radio.setPALevel(RF24_PA_LOW);              /*Set the transmission power to low*/
  radio.setDataRate(RF24_250KBPS);            /*Set the speed to 250kbps*/
  radio.setChannel(CHANNELNUMBER);            /*Set the channel*/
  radio.openWritingPipe(rfAddresses[0]);      /*Transmit to the remote address*/
  radio.openReadingPipe(1, rfAddresses[1]);   /*Recieve sensor addressed packets*/
  radio.setPayloadSize(PAYLOADSIZE);          /*Set the number of bytes for the data packet*/

  Serial.begin (9600);

  Wire.begin(SENSOR);        /*Join I2C as the sensor*/

  MyMPU6050.begin();                 /*Start the MPU sensor*/
  MyMPU6050.calcGyroOffsets(true);   /*Calculate the offsets*/
  delay(3000);                       /*3 second delay for offsets to be calculated*/
}

void loop() {
  radio.startListening();
  radio.read(&ButtonXYKpKiKdValues, sizeof(ButtonXYKpKiKdValues));
//  PrintAllIncomingInfo();

   if (ButtonXYKpKiKdValues[0] == 0) {
    ReadSensorValues();
    CalculateLFSpeedAndDirection();
//    PrintAllSensorAndSpeedInformation();
    SendLFSpeedAndDirectionToCar();

    PreviousError = CurrentError;    /*Save the current error as previous error for the next loop*/
    WeightedTotal = 0;               /*and reset the weighted total and sensor total back to 0*/
    SensorTotal = 0;
  }

  else if (ButtonXYKpKiKdValues[0] == 1) {
    CalculateRFSpeedAndDirection();
//    PrintAllJoyStickInfo();
    SendRFSpeedAndDirectionToCar();
```

```
    SendGyroDataToRemote();
    delay(75);
  }
}

void PrintAllIncomingInfo() {
  Serial.print("Button = ");
  Serial.print(ButtonXYKpKiKdValues[0]);
  Serial.print("   X Value = ");
  Serial.print(ButtonXYKpKiKdValues[1]);
  Serial.print("   Y Value = ");
  Serial.print(ButtonXYKpKiKdValues[2]);
  Serial.print("   Kp = ");
  Serial.print(ButtonXYKpKiKdValues[3]);
  Serial.print("   Ki = ");
  Serial.print(ButtonXYKpKiKdValues[4]);
  Serial.print("   Kd = ");
  Serial.println(ButtonXYKpKiKdValues[5]);
}

void ReadSensorValues() {
/*The analogue sensor is read, then constrained within a range,*/
/*then mapped (within that same range) to a value between 0 and 9*/
  SReading[0] = map(constrain(analogRead(LLLSensor), 120, 690), 120, 690, 0, 9);
  SReading[1] = map(constrain(analogRead(LLSensor), 130, 640), 130, 640, 0, 9);
  SReading[2] = map(constrain(analogRead(LSensor), 160, 710), 160, 710, 0, 9);
  SReading[3] = map(constrain(analogRead(RSensor), 90, 690), 90, 690, 0, 9);
  SReading[4] = map(constrain(analogRead(RRSensor), 390, 710), 390, 710, 0, 9);
  SReading[5] = map(constrain(analogRead(RRRSensor), 50, 430), 50, 430, 0, 9);
}

void CalculateLFSpeedAndDirection() {
/*Calculate the current error value (weighted average)*/
  for (int i=0; i<=5; i++) {
    WeightedTotal += SWeight[i] * SReading[i];
    SensorTotal += SReading[i];
  }

  CurrentError = WeightedTotal/SensorTotal;

/*Calculate PID value*/
  Proportional = CurrentError;
  Integral += CurrentError;
  Differential = CurrentError - PreviousError;

/*Kp = ButtonXYKpKiKdValues[3]      Ki = ButtonXYKpKiKdValues[4]        Kd = ButtonXYKpKiKdValues[5]*/
  PIDoutput = Proportional*ButtonXYKpKiKdValues[3] + Integral*ButtonXYKpKiKdValues[4] +
Differential*ButtonXYKpKiKdValues[5];

/*Work out direction*/
  if (PIDoutput > 15) {             /*Then line is to the left, so car needs to turn left*/
    Direction = 'L';
    Speed = constrain(PIDoutput, 0, 255);

  } else if (PIDoutput < -15) {     /*Then line is to the right, so car needs to turn right*/
    Direction = 'R';
    Speed = constrain(-PIDoutput, 0, 255);

  } else {                           /*Then line is in between sensors so no change*/
    Direction = 'N';
    Speed = 0;
  }
}
```

```
void PrintAllSensorAndSpeedInformation() {
  Serial.print("Button = ");
  Serial.print(ButtonXYKpKiKdValues[0]);
  Serial.print("\t");
  Serial.print(SReading[0]);
  Serial.print("\t");
  Serial.print(SReading[1]);
  Serial.print("\t");
  Serial.print(SReading[2]);
  Serial.print("\t");
  Serial.print(SReading[3]);
  Serial.print("\t");
  Serial.print(SReading[4]);
  Serial.print("\t");
  Serial.print(SReading[5]);
  Serial.print("\t");
  Serial.print("Direction = ");
  Serial.print(Direction);
  Serial.print("   Speed = ");
  Serial.println(Speed);
}

void SendLFSpeedAndDirectionToCar() {
  Wire.beginTransmission(CAR);
  Wire.write(ButtonXYKpKiKdValues[0]);
  Wire.write(Speed);
  Wire.write(Direction);
  Wire.endTransmission();
}

void CalculateRFSpeedAndDirection() {
    if (ButtonXYKpKiKdValues[1] > 0)
        XYDirection[0] = 'R';

    else if (ButtonXYKpKiKdValues[1] < 0) {
        XYDirection[0] = 'L';
        ButtonXYKpKiKdValues[1] *= -1;

    } else
        XYDirection[0] = 'N';


    if (ButtonXYKpKiKdValues[2] > 0)
        XYDirection[1] = 'F';

    else if (ButtonXYKpKiKdValues[2] < 0) {
        XYDirection[1] = 'B';
        ButtonXYKpKiKdValues[2] *= -1;

    } else
        XYDirection[1] = 'N';
}

void PrintAllJoyStickInfo () {
  Serial.print(XYDirection[0]);
  Serial.print(" X Value = ");
  Serial.print(ButtonXYKpKiKdValues[1]);
  Serial.print("\t");
  Serial.print(XYDirection[1]);
  Serial.print(" Y Value = ");
  Serial.println(ButtonXYKpKiKdValues[2]);
}
```

```
void SendRFSpeedAndDirectionToCar() {
  Wire.beginTransmission(CAR);
  Wire.write(ButtonXYKpKiKdValues[0]);
  Wire.write(ButtonXYKpKiKdValues[1]);
  Wire.write(ButtonXYKpKiKdValues[2]);
  Wire.write(XYDirection[0]);
  Wire.write(XYDirection[1]);
  Wire.endTransmission();
}

void SendGyroDataToRemote() {
  MyMPU6050.update();
  GyroData[0] = MyMPU6050.getAccX();
  GyroData[1] = MyMPU6050.getAngleY();

  radio.stopListening();
  radio.write(&GyroData, sizeof(GyroData));
}
```

Appendix C

(Car Code)

```
/*Libraires*/
#include <Wire.h>
#include <H61AEE_S02.h>
#include "Car_Template.h"

/*Global Variables*/
int ButtonValue;

char Direction;
int Speed;

int XValue, YValue;
char XDirection, YDirection;

int BaseSpeed = 48, Sum, Difference;

void setup() {
  car.setupVehicle();
  car.enableMotors(true);
  pinMode(YELLOW_LED, OUTPUT);
  pinMode(RED_LED, OUTPUT);
  Wire.begin(CAR);
  Wire.onReceive(receiveEvent);
  Serial.begin(9600);
}

void loop() {
  if (ButtonValue == 1) {
//    PrintAllJoyStickInfo();
    RFMotorControl();

  } else {
//    PrintAllLineFollowingInfo();
    PIDMotorControl();
  }
  delay (1);
}

void receiveEvent(int howMany) {
  ButtonValue = Wire.read();

  if (ButtonValue == 1) {
  XValue = Wire.read();
  YValue = Wire.read();
  XDirection = Wire.read();
  YDirection = Wire.read();

  } else {
  Speed = Wire.read();
  Direction = Wire.read();
  }
}

void PrintAllJoyStickInfo () {
  Serial.print(XDirection);
  Serial.print(" X Value = ");
  Serial.print(XValue);
  Serial.print("\t");
  Serial.print(YDirection);
  Serial.print(" Y Value = ");
  Serial.println(YValue);
}
```

```
void RFMotorControl() {
  Difference = YValue - XValue;
  if (Difference < 0)
        Difference = 0;

  switch (YDirection) {
    case 'N':
      car.setSpeed(ALL, 0);
      break;

    case 'F':
      switch (XDirection) {
        case 'N':
          car.setDirection(ALL, forwards);
          car.setSpeed(ALL, YValue);
          break;

       case 'R':
          car.setDirection(ALL, forwards);
          car.setSpeed(LEFT, YValue);
          car.setSpeed(RIGHT, Difference);
          break;

        case 'L':
          car.setDirection(ALL, forwards);
          car.setSpeed(LEFT, Difference);
          car.setSpeed(RIGHT, YValue);
         break;

        default:
          Serial.println("RF Remote X Error");
          break;
      }
      break;

    case 'B':
       switch (XDirection) {
        case 'N':
          car.setDirection(ALL, backwards);
          car.setSpeed(ALL, YValue);
          break;

       case 'R':
          car.setDirection(ALL, backwards);
          car.setSpeed(LEFT, YValue);
          car.setSpeed(RIGHT, Difference);
          break;

        case 'L':
          car.setDirection(ALL, backwards);
          car.setSpeed(LEFT, Difference);
          car.setSpeed(RIGHT, YValue);
          break;

        default:
          Serial.println("RF Remote X Error");
          break;
      }
      break;

    default:
        Serial.println("RF Remote Y Error");
        break;
  }
```

```
}

void PrintAllLineFollowingInfo () {
  Serial.print("Direction = ");
  Serial.print(Direction);
  Serial.print(" Speed = ");
  Serial.println(Speed);
}

void PIDMotorControl() {
  Sum = BaseSpeed + Speed;
  Difference = BaseSpeed - Speed;

  if (Sum > 255)
        Sum = 255;

  if (Difference < 0)
        Difference = 0;

  switch(Direction) {
    case 'N':
        car.setDirection(ALL, forwards);
        car.setSpeed(RIGHT, BaseSpeed);
        car.setSpeed(LEFT, BaseSpeed);
        break;

    case 'R':
        car.setDirection(RIGHT, backwards);
        car.setDirection(LEFT, forwards);
        car.setSpeed(RIGHT, Sum);
        car.setSpeed(LEFT, Sum);
        break;

    case 'L':
        car.setDirection(RIGHT, forwards);
        car.setDirection(LEFT, backwards);
        car.setSpeed(RIGHT, Sum);
        car.setSpeed(LEFT, Sum);
        break;

    default:
        Serial.println("Line Following Error");
        break;
  }
}
```