Project 1 Report: Sensing Signal Processing

Rohit Ravikumar – Group 9

1. Signal Processing for One Ultrasonic Sensor

1.1 Problem Statement

To use ultrasonic sensors to find the distance from an obstacle by sampling the time data as fast as possible. The main aim of the project is to not only find the distance but to also minimize the noise by implementing a Kalman filter and making sure the sensor value converges to a stable reading with zero error within 10 seconds. The filter used must be designed from scratch, A calibration function must be defined to correlate the time values from the ultrasonic sensor to distance values. The distance must be printed in millimeters, along with the real time covariance value and, a buzzer and LED must be implemented to be activated when the error value does not change much, or the distance value has stabilized.

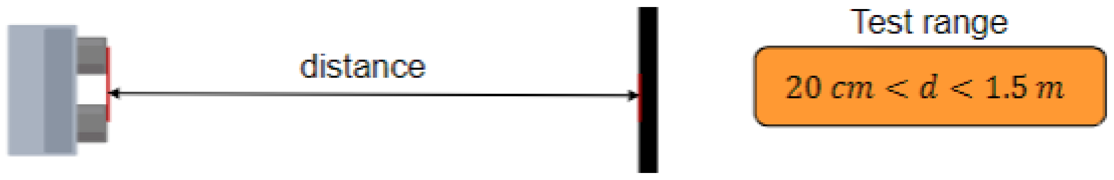


Fig 1.1.1

1.2 Technical Approach

The entire process was carried out in the following manner:

1) Sampling: Finding the sensor Variance and selection of the best sensor
2) Filtering: Implementing Kalman filter to remove noise
3) Calibration: Obtain the calibration function

<u>Sampling</u>

This was the most exhaustive process as each group was given 7 sensors and the variance of each of the sensors had to be calculated and the sensor with the least overall variance had to be selected for the final test. The sensor variance varied with the temperature, air pressure etc. hence the sampling had to be done multiple times and the variance calculated on a regular basis, to get some consistent readings.

The readings were taken at an interval of 10mm from 20mm upto 150mm.

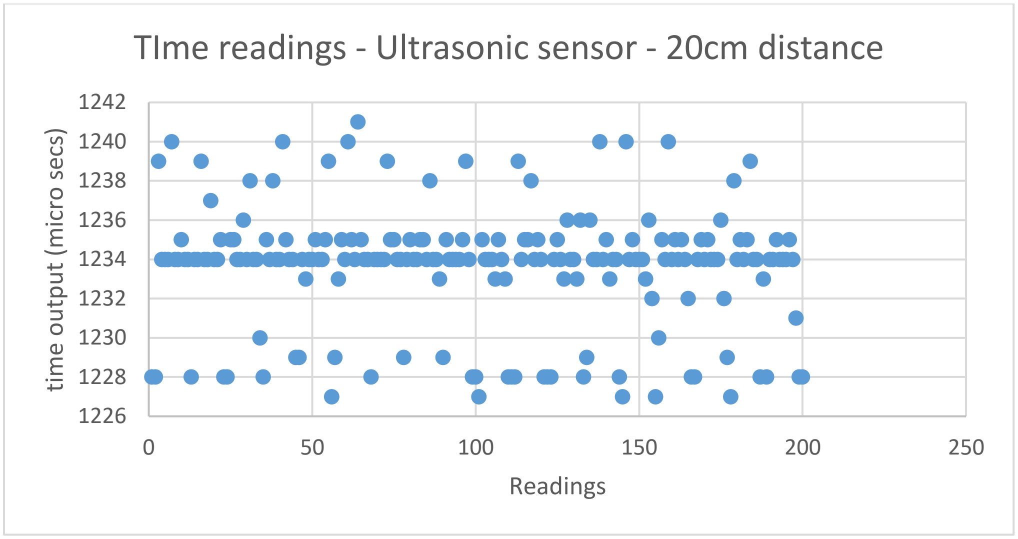
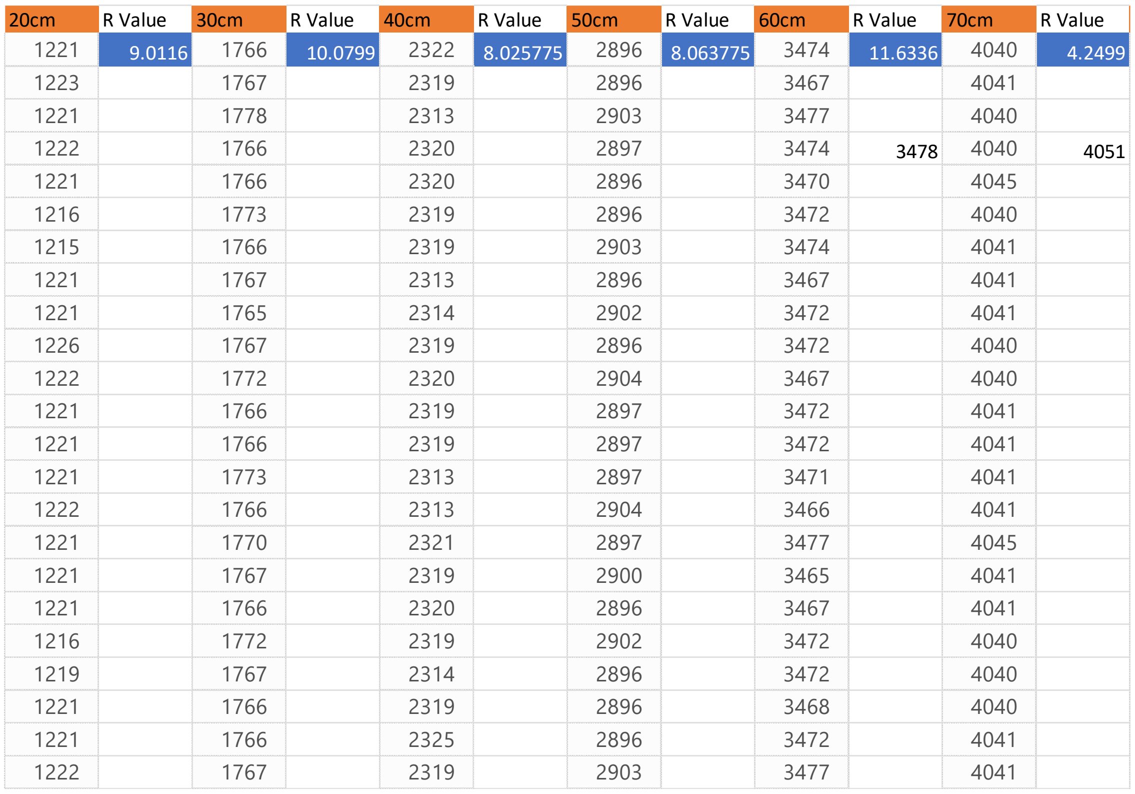


Fig 1.2.1 Time readings – Distance 20cm

The calculation of variance was done by first using an Arduino plugin on Excel which directly printed out the pulse in time data from the Arduino to an excel sheet.

Around 200-time samples were taken, and from this the variance of the readings was calculated using the VAR.P function on Excel.

The variance reading obtained from out sensors are as shown below:



20cm	R Value	30cm	R Value	40cm	R Value	50cm	R Value	60cm	R Value	70cm	R Value
1221	9.0116	1766	10.0799	2322	8.025775	2896	8.063775	3474	11.6336	4040	4.2499
1223		1767		2319		2896		3467		4041	
1221		1778		2313		2903		3477		4040	
1222		1766		2320		2897		3474	3478	4040	4051
1221		1766		2320		2896		3470		4045	
1216		1773		2319		2896		3472		4040	
1215		1766		2319		2903		3474		4041	
1221		1767		2313		2896		3467		4041	
1221		1765		2314		2902		3472		4041	
1226		1767		2319		2896		3472		4040	
1222		1772		2320		2904		3467		4040	
1221		1766		2319		2897		3472		4041	
1221		1766		2319		2897		3472		4041	
1221		1773		2313		2897		3471		4041	
1222		1766		2313		2904		3466		4041	
1221		1770		2321		2897		3477		4045	
1221		1767		2319		2900		3465		4041	
1221		1766		2320		2896		3467		4041	
1216		1772		2319		2902		3472		4040	
1219		1767		2314		2896		3472		4040	
1221		1766		2319		2896		3468		4040	
1221		1766		2325		2896		3472		4041	
1222		1767		2319		2903		3477		4041	

Fig 1.2.2 T values or samples taken to measure variance for Sensor 1

The variance vs distance values for sensor 1

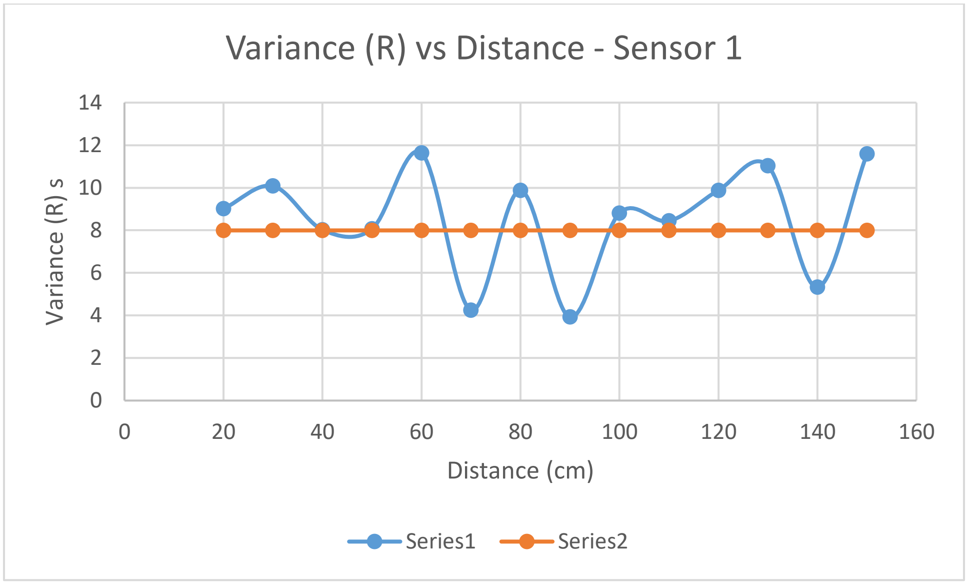


Fig 1.2.3 Variance vs Distance for Sensor 1

Now comparing both the graphs we see that the trend is very similar with an overall low R value except for a slight number of spikes. Sensor 1 had a lower average value than sensor 3 but still sensor 3 was selected for the final test as we found sensor 3 to be more reliable with the R values for a prolonged period.

Filtering:

The Kalman filter method was used to filter the noisy signals. The Kalman filter is a very efficient way to get accurate noiseless output with just a few simple lines of code. It takes a series of time readings and uses those readings to converge the error covariance value to 0 and the time value to a stable final time.

The Kalman filter works as follows:

Process Model:

yk = Ayk-1 + Buk + wk

zk = Hyk + vk

k: discrete time

y: system state to be estimated

u: system input (optional)

z: system measurement

A, B, H: constant matrices

Wk = Process noise (0,Q)

Vk = Measurement noise (0,R)

Prediction Step

State Estimate: **Ŷk = Ayk-1 + Buk**

Error Covariance: **Pk = APk-1AT + Q**

Correction Step:

Kalman Gain: **K = PkHT (HPkHT+ R)-1**

Updated State Estimate**: ŷk = ŷk+ K(zk - H ŷk)**

Updated Error Covariance: **Pk = (I - KH)Pk**

Calibration:

Once we get the Kalman filter values we can then move on to the Calibration part.

Calibration is basically developing a relation ship between the measured variable (input) and the physical variable (output).

The process that was adopted was that once the filtered readings were taken for distances from 20cm to 150cm then we could plot the values for the Distance (input) and time (output) from the sensor.

This calibration equation could then be used to directly convert the filtered time output from the ultrasonic sensor to the actual distance.

Yk(mus)	Dist (cm)
1209.85	20
1766.45	30
2311.24	40
2811.03	50
3450.17	60
3950.7	70
4555.55	80
5132.98	90
5700.28	100
6270.11	110
6837.59	120
7312.68	130
7965.87	140
8403.17	150

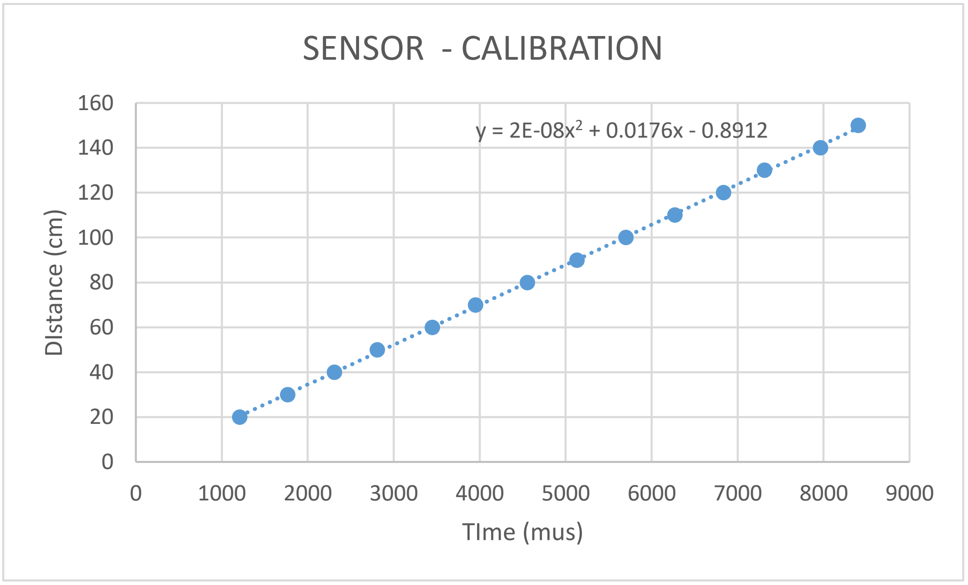


Fig 1.2.4 Calibration data for sensor 3

As we can see a second order line was fitted with the equation:

$$\mathbf{y = 2E\text{-}08x^2 + 0.0176x - 0.8912}$$

y is the distance in cms and x is the state estimate yk

<u>Testing:</u>

Finally, before testing a few tweaks had to be made to the calibration and output equation to compensate for the changing variance. An offset had to be used

1.3 Hardware and Software Implementation

The Hardware used to run 1.1 is as follows:

1) Arduino UNO

Fig 1.3.1 Arduino UNO (source google)

Bread Board:

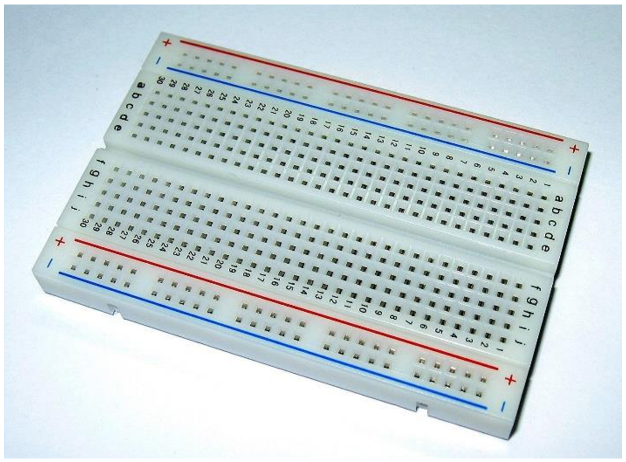


Fig 1.3.2 Bread Board (source google)

Jumper Wires, LED and Buzzer:

Fig 1.3.3 Bread Board (source google)

Ultrasonic sensor

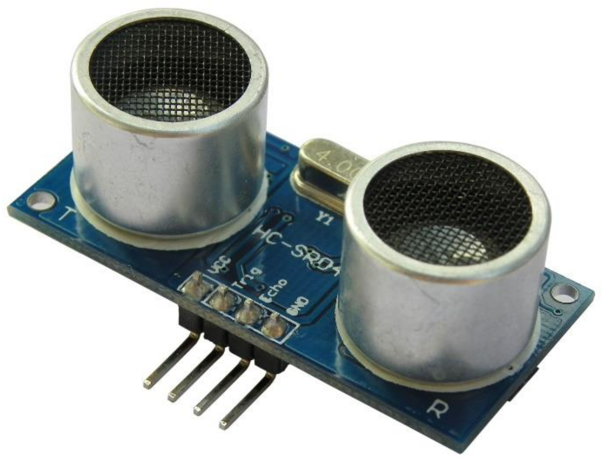


Fig 1.3.4 Ultrasonic Sensor (source google)

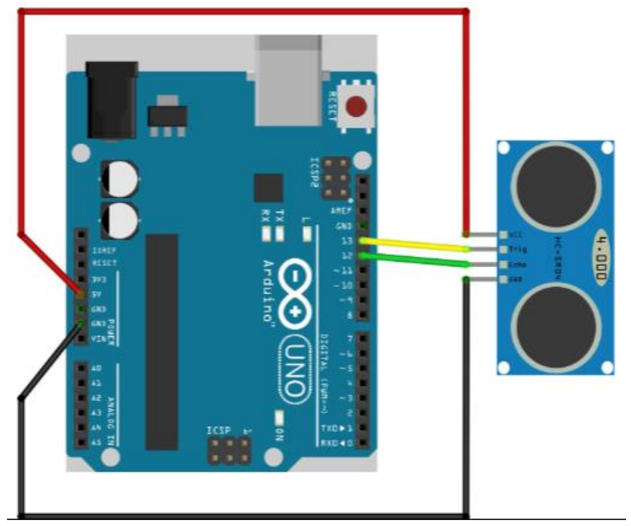
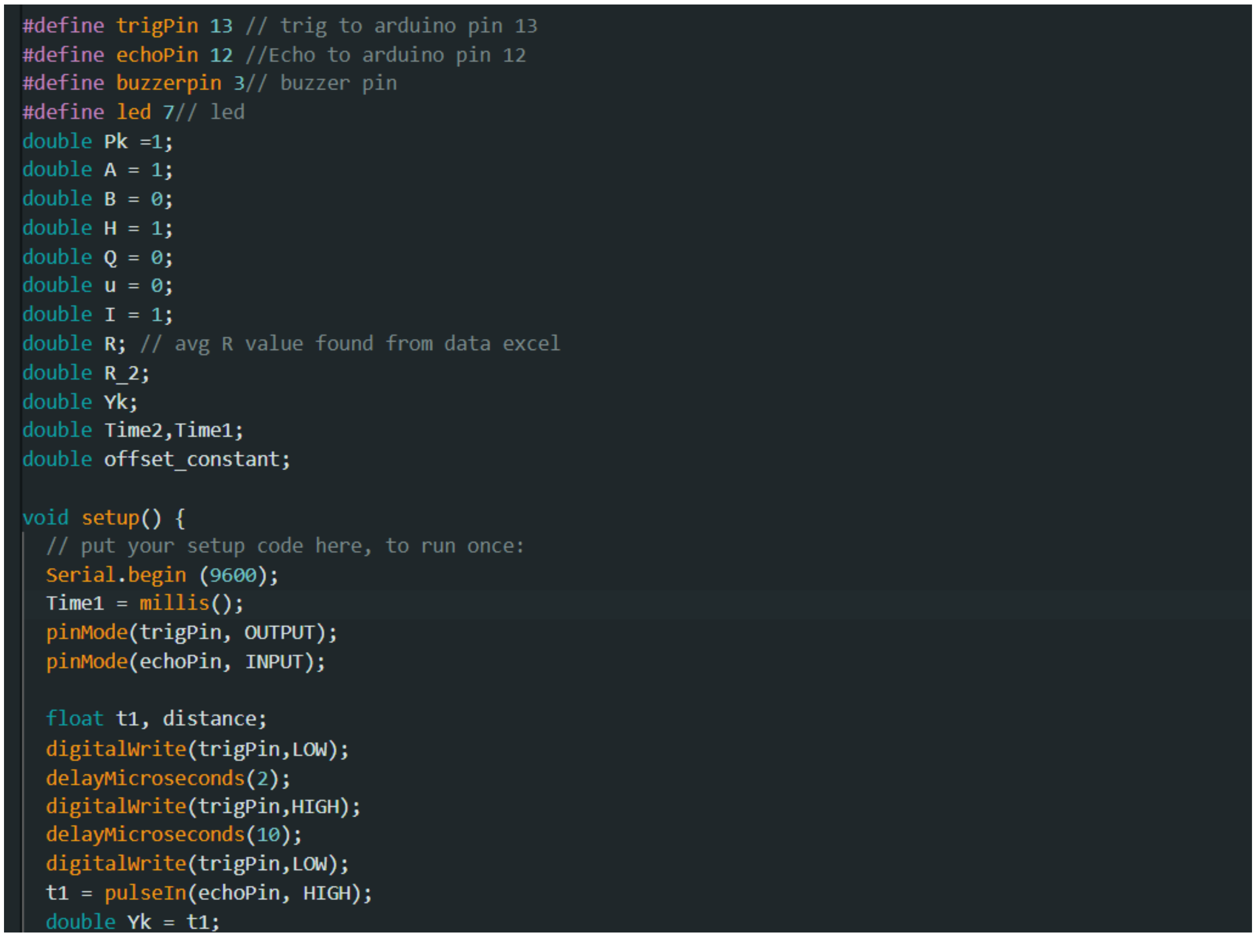


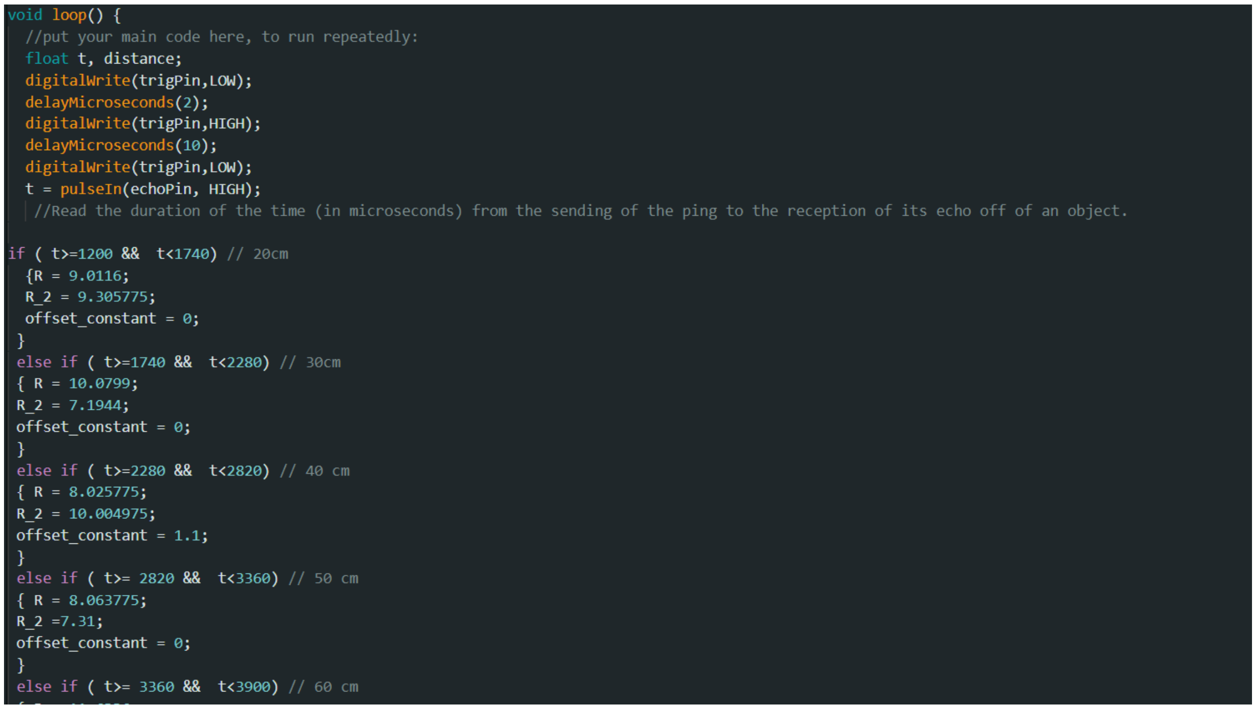
Fig 1.3.5 Wiring

Software:

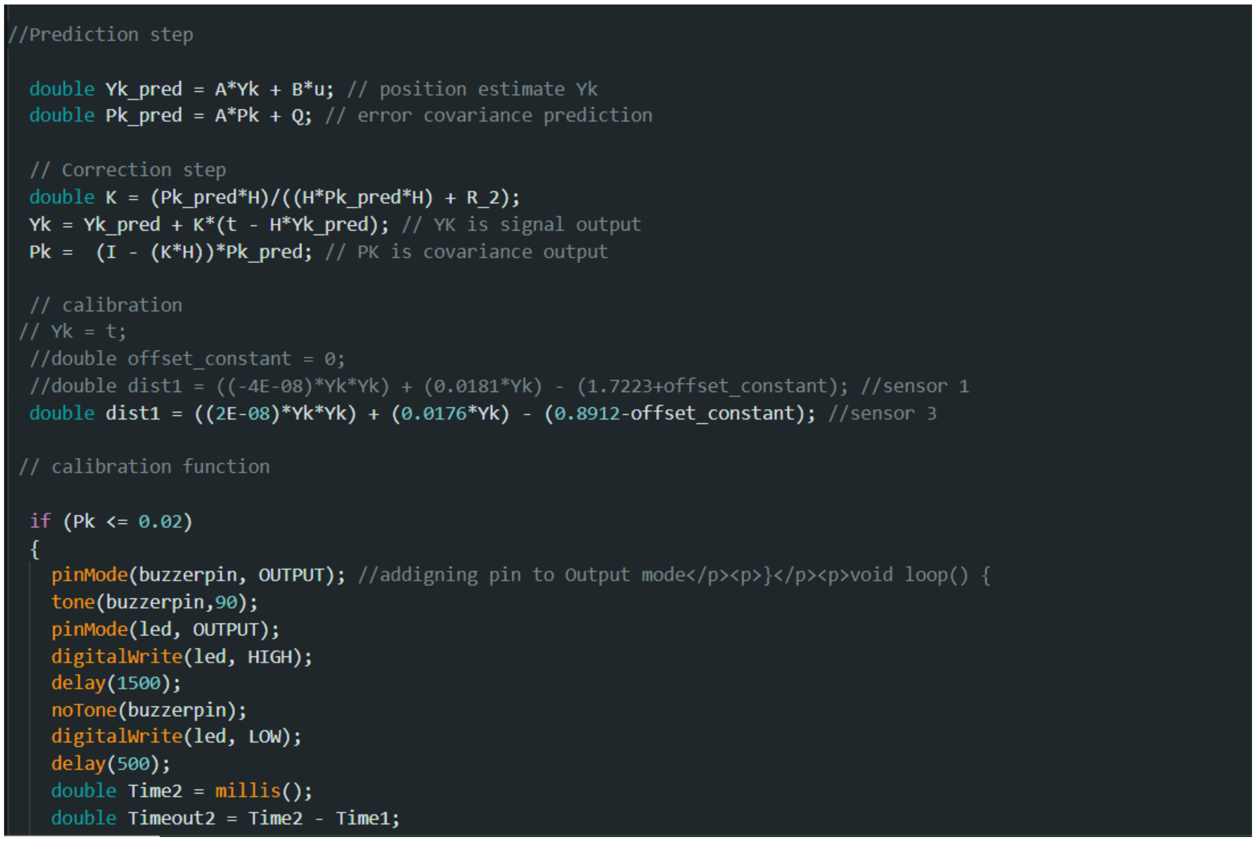
The entire code implementation on Arduino is as shown below:



```
#define trigPin 13 // trig to arduino pin 13
#define echoPin 12 //Echo to arduino pin 12
#define buzzerpin 3// buzzer pin
#define led 7// led
double Pk =1;
double A = 1;
double B = 0;
double H = 1;
double Q = 0;
double u = 0;
double I = 1;
double R; // avg R value found from data excel
double R_2;
double Yk;
double Time2,Time1;
double offset_constant;

void setup() {
  // put your setup code here, to run once:
  Serial.begin (9600);
  Time1 = millis();
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);

  float t1, distance;
  digitalWrite(trigPin,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin,LOW);
  t1 = pulseIn(echoPin, HIGH);
  double Yk = t1;
```



```
void loop() {
  //put your main code here, to run repeatedly:
  float t, distance;
  digitalWrite(trigPin,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin,LOW);
  t = pulseIn(echoPin, HIGH);
   //Read the duration of the time (in microseconds) from the sending of the ping to the reception of its echo off of an object.

if ( t>=1200 &&  t<1740) // 20cm
  {R = 9.0116;
  R_2 = 9.305775;
  offset_constant = 0;
 }
 else if ( t>=1740 &&  t<2280) // 30cm
 { R = 10.0799;
 R_2 = 7.1944;
 offset_constant = 0;
 }
 else if ( t>=2280 &&  t<2820) // 40 cm
 { R = 8.025775;
 R_2 = 10.004975;
 offset_constant = 1.1;
 }
 else if ( t>= 2820 &&  t<3360) // 50 cm
 { R = 8.063775;
 R_2 =7.31;
 offset_constant = 0;
 }
 else if ( t>= 3360 &&  t<3900) // 60 cm
```



```
//Prediction step

  double Yk_pred = A*Yk + B*u; // position estimate Yk
  double Pk_pred = A*Pk + Q; // error covariance prediction

  // Correction step
  double K = (Pk_pred*H)/((H*Pk_pred*H) + R_2);
  Yk = Yk_pred + K*(t - H*Yk_pred); // YK is signal output
  Pk =  (I - (K*H))*Pk_pred; // PK is covariance output

  // calibration
 // Yk = t;
  //double offset_constant = 0;
  //double dist1 = ((-4E-08)*Yk*Yk) + (0.0181*Yk) - (1.7223+offset_constant); //sensor 1
  double dist1 = ((2E-08)*Yk*Yk) + (0.0176*Yk) - (0.8912-offset_constant); //sensor 3

 // calibration function

  if (Pk <= 0.02)
  {
    pinMode(buzzerpin, OUTPUT); //addigning pin to Output mode</p><p>}</p><p>void loop() {
    tone(buzzerpin,90);
    pinMode(led, OUTPUT);
    digitalWrite(led, HIGH);
    delay(1500);
    noTone(buzzerpin);
    digitalWrite(led, LOW);
    delay(500);
    double Time2 = millis();
    double Timeout2 = Time2 - Time1;
```

Features of the Code:

- First the pins are defined for the Ultrasonic sensor, buzzer and the LED
- The first reading of the ultrasonic sensor is then initiated in the void setup.
- This reading is then used as the Yk(1) value in the void loop.
- An if loop is added to take the correct R value based on the time input range – for every 10 cms, and an offset value is also added to that.
- Then the output from the Kalman filter (Yk) is then sent to the calibration equation to get dist1 as the output.
- An if statement is added at the end, to exit the void loop when the Pk value reaches 0.02. This value was chosen as we noticed that at this Pk value, the error was almost minimal and also the time taken to converge to this corresponding value is under 10 seconds.

1.4 Experimental Results

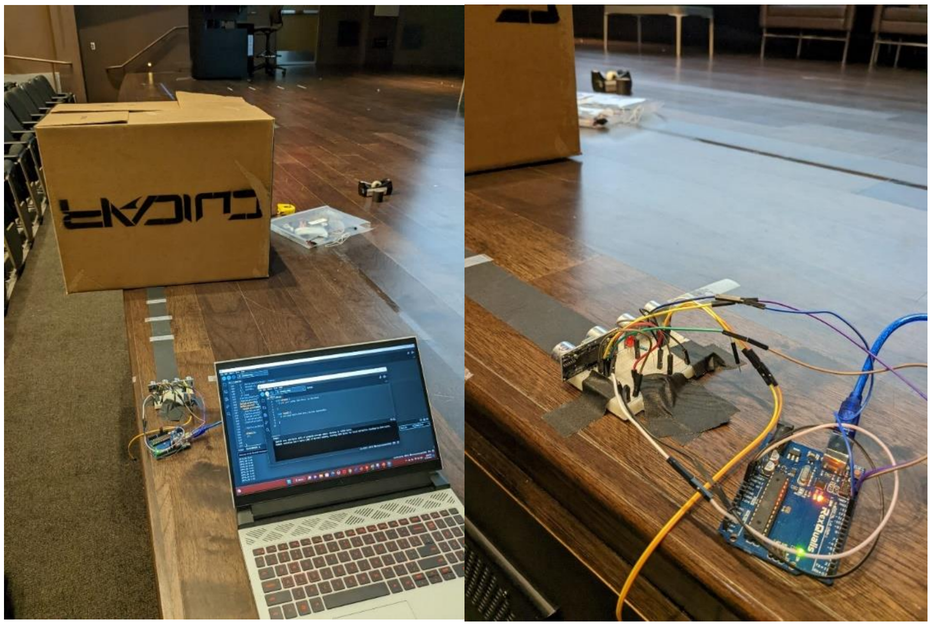
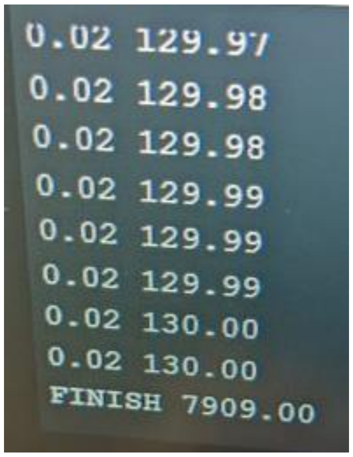


Fig 1.4.1 Ultrasonic Sensor (source google)

The readings from the Kalman filter were accurate upto 3mm and were converging well under distances of under 120 mm. After this distance, the time taken to converge increased but was still under the 10s limit

Shown below are some values from the ultrasonic sensor.

Distance - cm	**130**
Sensor Reading - cm	**130.00**
Convergence Time – milli seconds	**7909**



```
0.02 129.97
0.02 129.98
0.02 129.98
0.02 129.99
0.02 129.99
0.02 129.99
0.02 130.00
0.02 130.00
FINISH 7909.00
```

2. Sensor Fusion of Multiple Ultrasonic Sensors

2.1 Problem Statement

To use ultrasonic sensors to find the distance from an obstacle by sampling the time data as fast as possible. The main aim of the project is to not only find the distance but to also minimize the noise by implementing a Kalman filter and utilizing sensor fusion to make the value obtained quicker and more accurate as compared to using one sensor. The distance must be printed in millimeters, along with the real time covariance value and, a buzzer and LED must be implemented to be activated when the error value does not change much, or the distance value has stabilized.

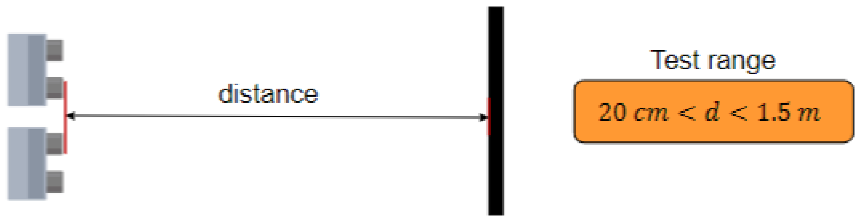


Fig 2.1.1

2.2 Technical Approach

The entire process was carried out in the following manner:

1) Sampling: Finding the sensor Variance and selection of the best sensor (already done in Part 1.1
2) Filtering: Implementing Kalman filter to remove noise (2 step correction)
3) Calibration: Obtain the calibration function (same as part 1.1)

<u>Sampling</u>

This was the most exhaustive process as each group was given 7 sensors and the variance of each of the sensors had to be calculated and the sensor with the least overall variance had to be selected for the final test. The sensor variance varied with the temperature, air pressure etc. hence the sampling had to be done multiple times and the variance calculated on a regular basis, to get some consistent readings.

The readings were taken at an interval of 10mm from 20mm upto 150mm.

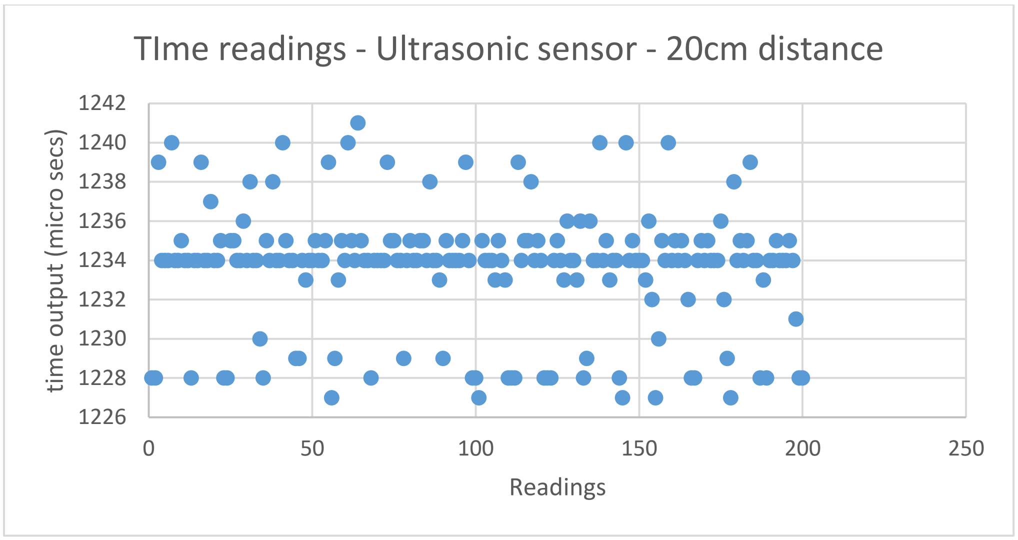
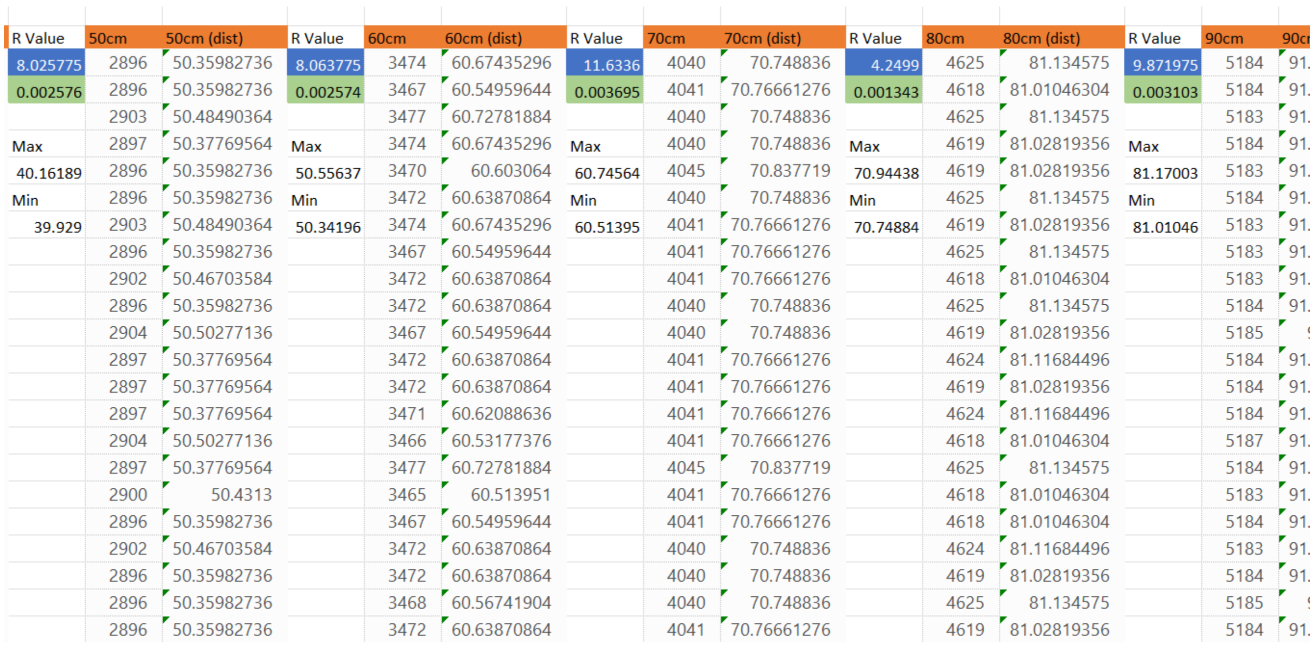


Fig 2.2.1 Time readings – Distance 20cm

The calculation of variance was done by first using an Arduino plugin on Excel which directly printed out the pulse in time data from the Arduino to an excel sheet.

Around 200-time samples were taken, and from this the variance of the readings was calculated using the VAR.P function on Excel.

The variance reading obtained from out sensors are as shown below:



R Value	50cm	50cm (dist)	R Value	60cm	60cm (dist)	R Value	70cm	70cm (dist)	R Value	80cm	80cm (dist)	R Value	90cm	90c
8.025775	2896	50.35982736	8.063775	3474	60.67435296	11.6336	4040	70.748836	4.2499	4625	81.134575	9.871975	5184	91.
0.002576	2896	50.35982736	0.002574	3467	60.54959644	0.003695	4041	70.76661276	0.001343	4618	81.01046304	0.003103	5184	91.
	2903	50.48490364		3477	60.72781884		4040	70.748836		4625	81.134575		5183	91.
Max	2897	50.37769564	Max	3474	60.67435296	Max	4040	70.748836	Max	4619	81.02819356	Max	5184	91.
40.16189	2896	50.35982736	50.55637	3470	60.603064	60.74564	4045	70.837719	70.94438	4619	81.02819356	81.17003	5183	91.
Min	2896	50.35982736	Min	3472	60.63870864	Min	4040	70.748836	Min	4625	81.134575	Min	5184	91.
39.929	2903	50.48490364	50.34196	3474	60.67435296	60.51395	4041	70.76661276	70.74884	4619	81.02819356	81.01046	5183	91.
	2896	50.35982736		3467	60.54959644		4041	70.76661276		4625	81.134575		5183	91.
	2902	50.46703584		3472	60.63870864		4041	70.76661276		4618	81.01046304		5183	91.
	2896	50.35982736		3472	60.63870864		4040	70.748836		4625	81.134575		5184	91.
	2904	50.50277136		3467	60.54959644		4040	70.748836		4619	81.02819356		5185	9
	2897	50.37769564		3472	60.63870864		4041	70.76661276		4624	81.11684496		5184	91.
	2897	50.37769564		3472	60.63870864		4041	70.76661276		4619	81.02819356		5184	91.
	2897	50.37769564		3471	60.62088636		4041	70.76661276		4624	81.11684496		5184	91.
	2904	50.50277136		3466	60.53177376		4041	70.76661276		4618	81.01046304		5187	91.
	2897	50.37769564		3477	60.72781884		4045	70.837719		4625	81.134575		5184	91.
	2900	50.4313		3465	60.513951		4041	70.76661276		4618	81.01046304		5183	91.
	2896	50.35982736		3467	60.54959644		4041	70.76661276		4618	81.01046304		5184	91.
	2902	50.46703584		3472	60.63870864		4040	70.748836		4624	81.11684496		5183	91.
	2896	50.35982736		3472	60.63870864		4040	70.748836		4619	81.02819356		5184	91.
	2896	50.35982736		3468	60.56741904		4040	70.748836		4625	81.134575		5185	9
	2896	50.35982736		3472	60.63870864		4041	70.76661276		4619	81.02819356		5184	91.

Fig 2.2.2 T values or samples taken to measure distance variance for Sensor 1

The variance vs distance values for sensor 1

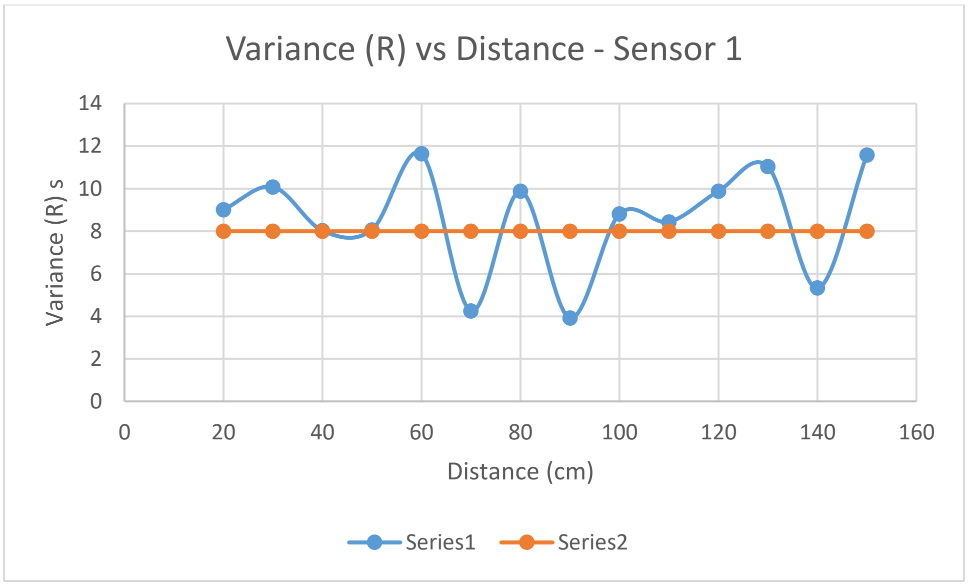


Fig 2.2.3 Variance vs Distance for Sensor 1

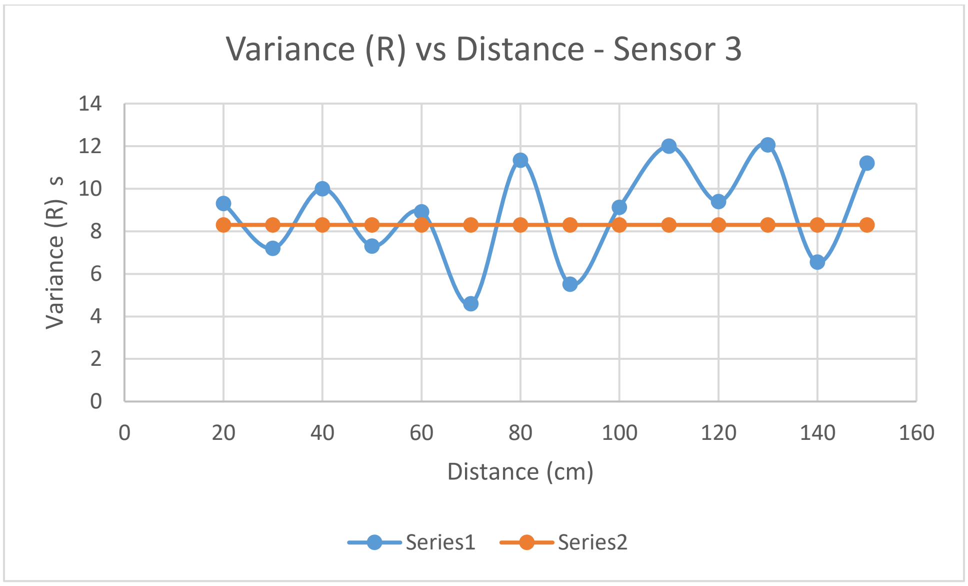


Fig 2.2.4 Variance vs Distance for Sensor 3

Filtering:

The Kalman filter method was used to filter the noisy signals. The Kalman filter is a very efficient way to get accurate noiseless output with just a few simple lines of code. It takes a series of time readings and uses those readings to converge the error covariance value to 0 and the time value to a stable final time.

The Kalman filter works as follows:

Process Model:

yk = Ayk-1 + Buk + wk

zk = Hyk + vk

k: discrete time

y: system state to be estimated

u: system input (optional)

z: system measurement

A, B, H: constant matrices

Wk = Process noise (0,Q)

Vk = Measurement noise (0,R)

Prediction Step

State Estimate: **Ŷk = Ayk-1 + Buk**

Error Covariance: **Pk = APk-1AT + Q**

Correction Step 1:

Kalman Gain: **K = PkHT (HPkHT+ R)-1**

Updated State Estimate**: ŷk = ŷk+ K(zk - H ŷk)**

Updated Error Covariance: **Pk = (I - KH)Pk**

Correction Step 2
K = P1,k(P1,k + R2)-1

ŷk = ŷ1,k + K(z2,k -ŷ1,k)

Pk = (I -K)P1,k

Calibration:

Once we get the Kalman filter values we can then move on to the Calibration part.

Calibration is basically developing a relationship between the measured variable (input) and the physical variable (output).

The process that was adopted was that once the filtered readings were taken for distances from 20cm to 150cm then we could plot the values for the Distance (input) and time (output) from the sensor.

This calibration equation could then be used to directly convert the filtered time output from the ultrasonic sensor to the actual distance.

Yk(mus)	Dist (cm)
1209.85	20
1766.45	30
2311.24	40
2811.03	50
3450.17	60
3950.7	70
4555.55	80
5132.98	90
5700.28	100
6270.11	110
6837.59	120
7312.68	130
7965.87	140
8403.17	150

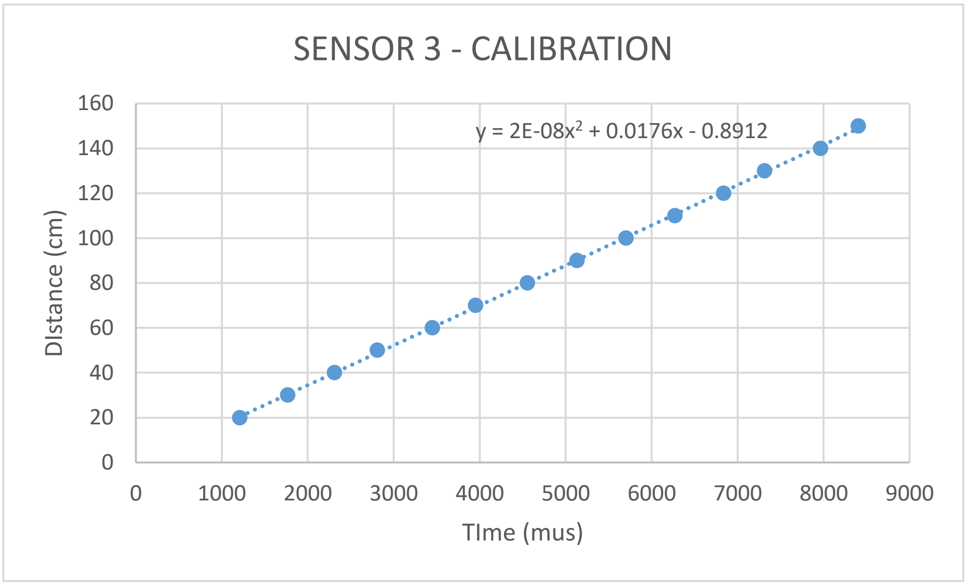


Fig 2.2.5 Calibration data for sensor 3

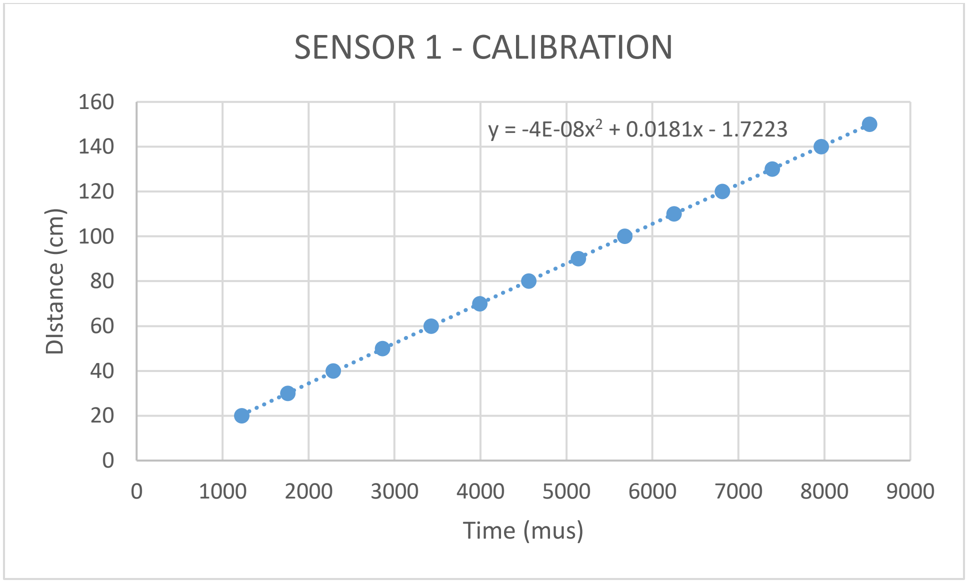


Fig 2.2.6 Calibration data for sensor 1

As we can see a second order line was fitted with the equation for sensor 3:

$$\mathbf{y = 2E\text{-}08x^2 + 0.0176x - 0.8912}$$

Calibration equation for sensor 1 is:

$$\mathbf{y = -4E\text{-}08x^2 + 0.0181x - 1.7223}$$

y is the distance in cms and x is the state estimate yk

Testing:

Finally, before testing a few tweaks had to be made to the calibration and output equation to compensate for the changing variance. An offset had to be used

2.3 Hardware and Software Implementation

The Hardware used to run 1.1 is as follows:

2) Arduino UNO

Fig 2.3.1 Arduino UNO (source google)

Bread Board:

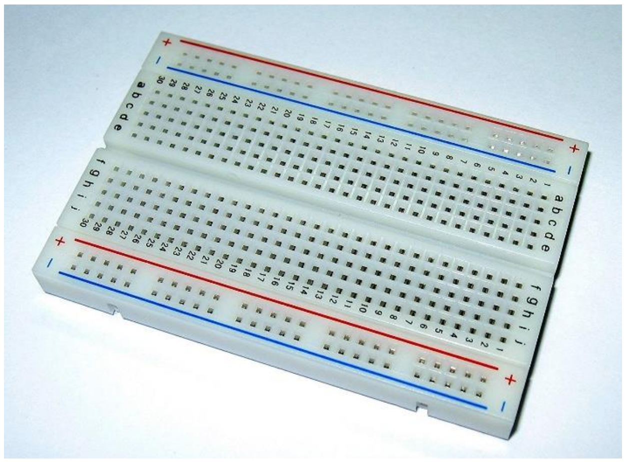


Fig 2.3.2 Bread Board (source google)

Jumper Wires, LED and Buzzer:

Fig 2.3.3 Jumper wires, LED, buzzer (source google)

Ultrasonic sensor:

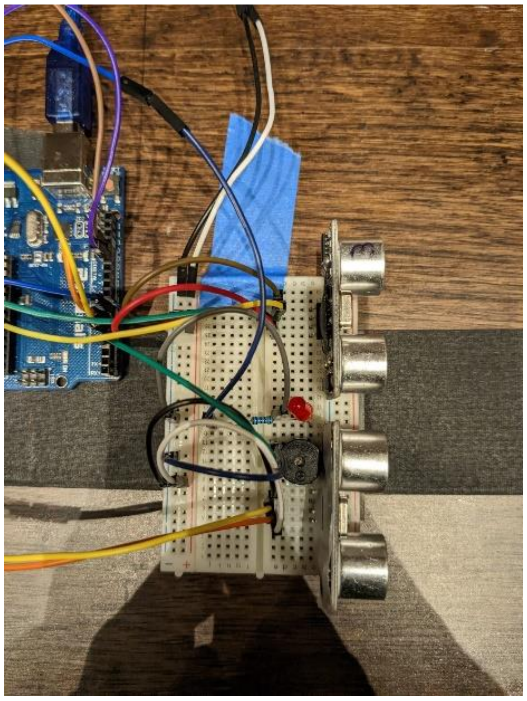


Fig 2.3.4 Ultrasonic sensor twin setup (source google)

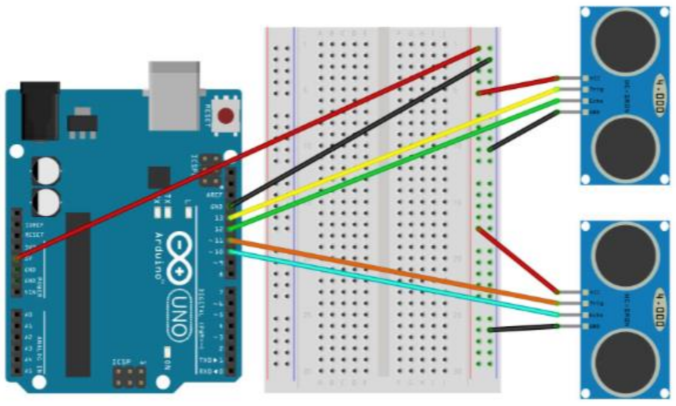
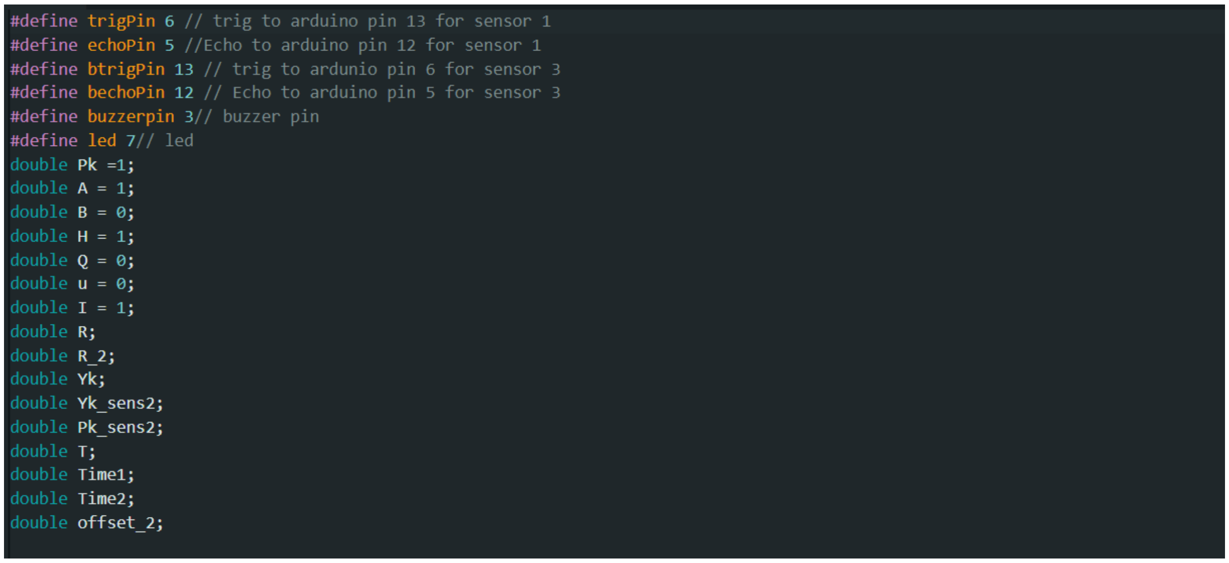


Fig 2.3.5 Wiring

Software:

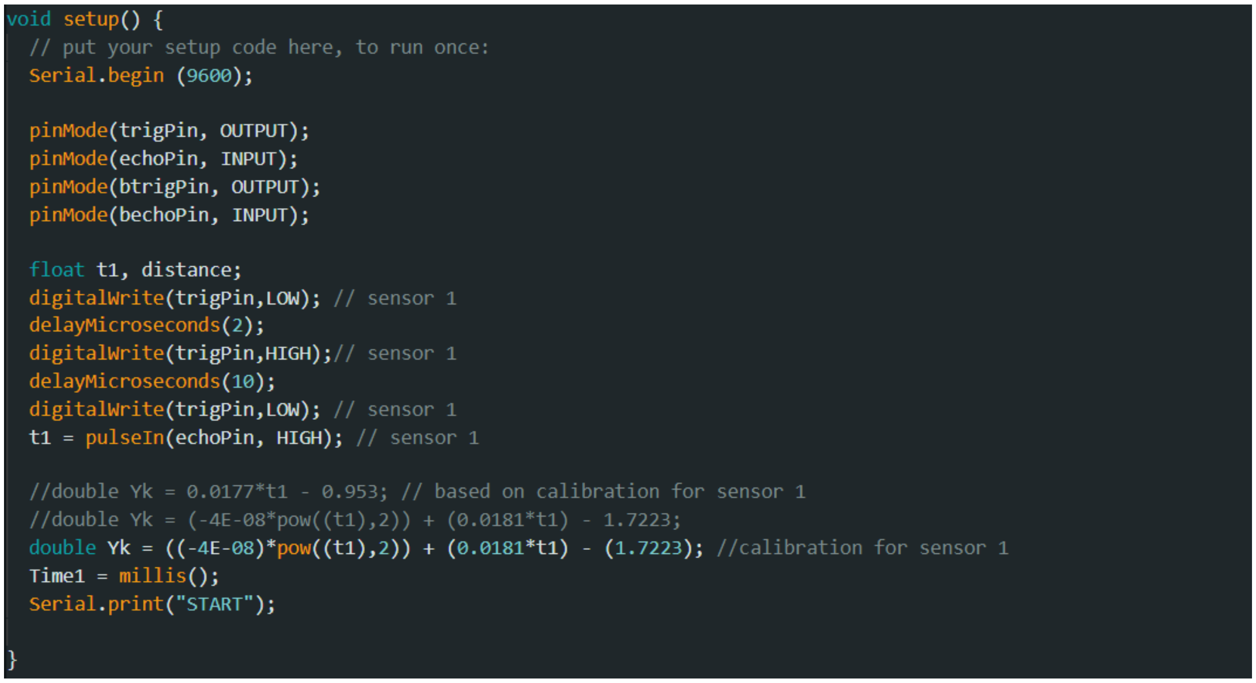
The entire code implementation on Arduino is as shown below:

1) Declaring the variables and the pin inputs for both the ultrasonic sensors and the buzzer and the pin.


```
#define trigPin 6 // trig to arduino pin 13 for sensor 1
#define echoPin 5 //Echo to arduino pin 12 for sensor 1
#define btrigPin 13 // trig to ardunio pin 6 for sensor 3
#define bechoPin 12 // Echo to arduino pin 5 for sensor 3
#define buzzerpin 3// buzzer pin
#define led 7// led
double Pk =1;
double A = 1;
double B = 0;
double H = 1;
double Q = 0;
double u = 0;
double I = 1;
double R;
double R_2;
double Yk;
double Yk_sens2;
double Pk_sens2;
double T;
double Time1;
double Time2;
double offset_2;
```

2) Void Setup:

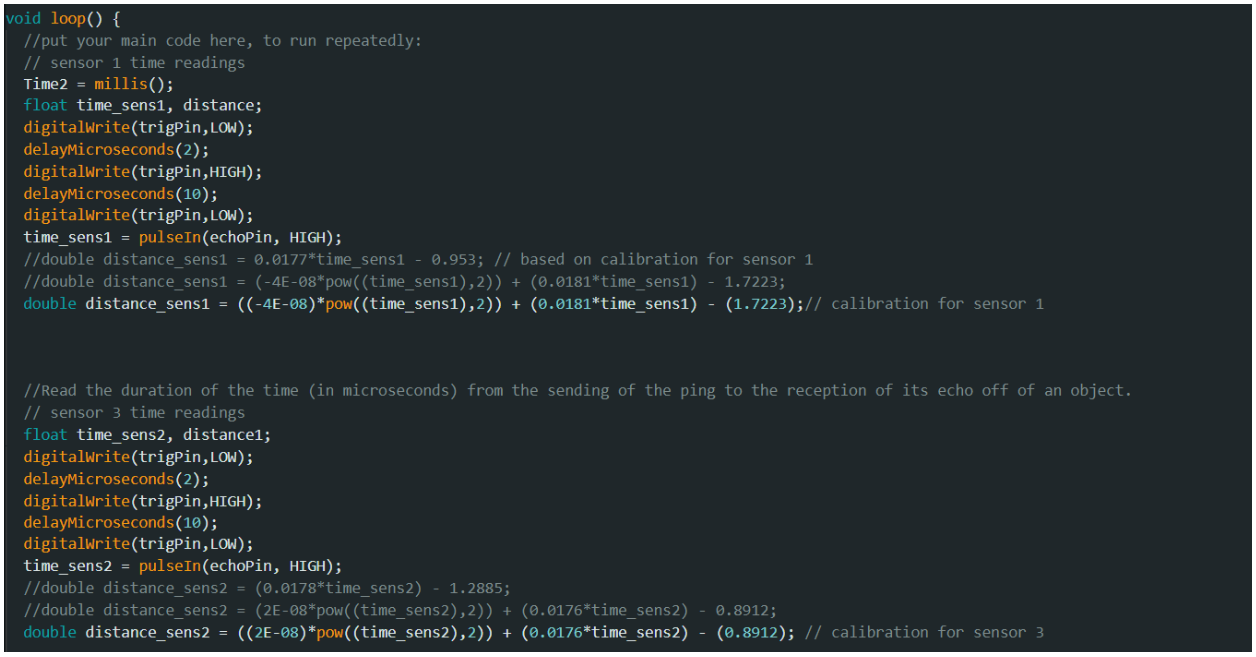
The first reading of the ultrasonic sensor is then initiated in the void setup. This acts as the Yk for sensor 1



```
void setup() {
  // put your setup code here, to run once:
  Serial.begin (9600);

  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  pinMode(btrigPin, OUTPUT);
  pinMode(bechoPin, INPUT);

  float t1, distance;
  digitalWrite(trigPin,LOW); // sensor 1
  delayMicroseconds(2);
  digitalWrite(trigPin,HIGH);// sensor 1
  delayMicroseconds(10);
  digitalWrite(trigPin,LOW); // sensor 1
  t1 = pulseIn(echoPin, HIGH); // sensor 1

  //double Yk = 0.0177*t1 - 0.953; // based on calibration for sensor 1
  //double Yk = (-4E-08*pow((t1),2)) + (0.0181*t1) - 1.7223;
  double Yk = ((-4E-08)*pow((t1),2)) + (0.0181*t1) - (1.7223); //calibration for sensor 1
  Time1 = millis();
  Serial.print("START");

}
```

3) Void Loop:

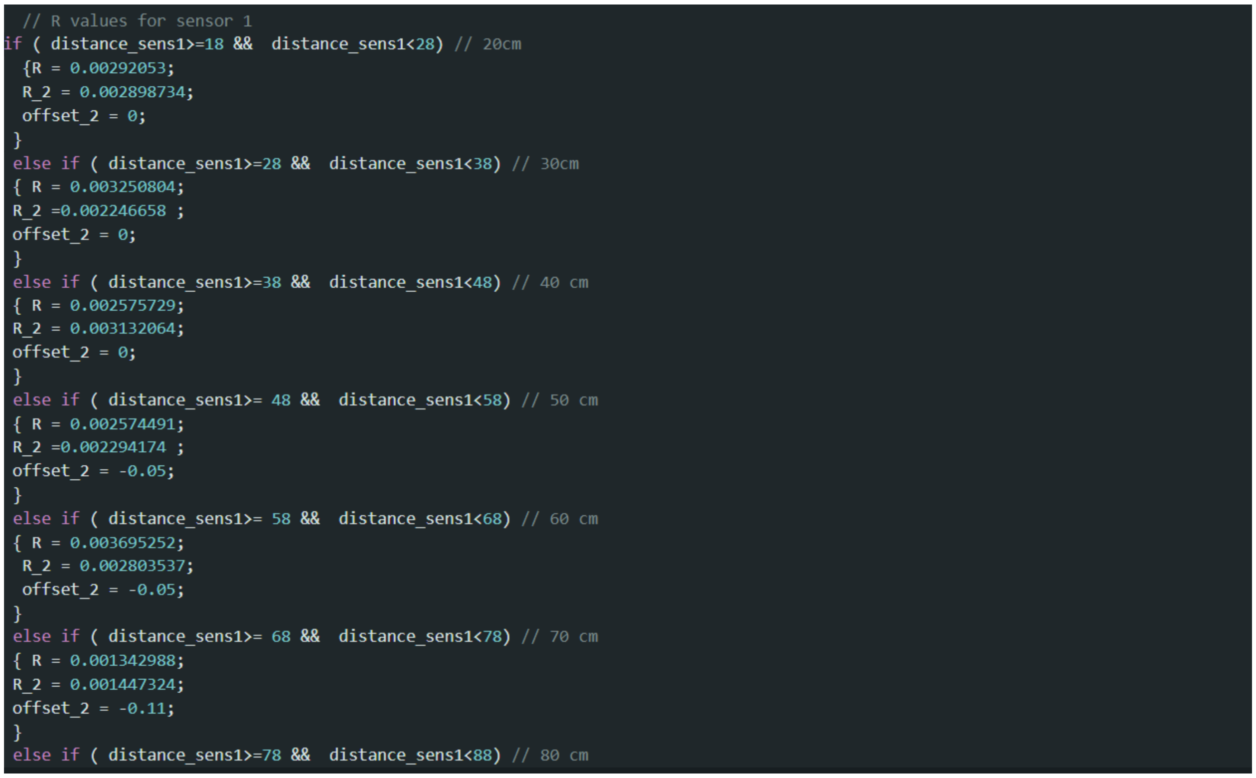
Both the sensors are initiated to take readings. time_sens_1 and time_sens_2 are ultrasonic sensor outputs, and as in this project we have to do a distance based Kalman filtering, the calibration equation obtained in the part 1.1 is used to get output distance distance_sens_1 and distance_sens2. This is then used as the Zk value for the sensors in the state prediction equations.



```
void loop() {
  //put your main code here, to run repeatedly:
  // sensor 1 time readings
  Time2 = millis();
  float time_sens1, distance;
  digitalWrite(trigPin,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin,LOW);
  time_sens1 = pulseIn(echoPin, HIGH);
  //double distance_sens1 = 0.0177*time_sens1 - 0.953; // based on calibration for sensor 1
  //double distance_sens1 = (-4E-08*pow((time_sens1),2)) + (0.0181*time_sens1) - 1.7223;
  double distance_sens1 = ((-4E-08)*pow((time_sens1),2)) + (0.0181*time_sens1) - (1.7223);// calibration for sensor 1



  //Read the duration of the time (in microseconds) from the sending of the ping to the reception of its echo off of an object.
  // sensor 3 time readings
  float time_sens2, distance1;
  digitalWrite(trigPin,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin,LOW);
  time_sens2 = pulseIn(echoPin, HIGH);
  //double distance_sens2 = (0.0178*time_sens2) - 1.2885;
  //double distance_sens2 = (2E-08*pow((time_sens2),2)) + (0.0176*time_sens2) - 0.8912;
  double distance_sens2 = ((2E-08)*pow((time_sens2),2)) + (0.0176*time_sens2) - (0.8912); // calibration for sensor 3
```

Similar to the Part 1.1, as the variance value changes with every 10 cm increment in the distance a piece wise if statement was used as shown below which accounted for the changing R value.

Offset_2 is also incorporated into this statement as the offset was necessary while getting the output as only so many calibrations can be done. The offset is a quick and easy way to get accurate readings based on trial and error.

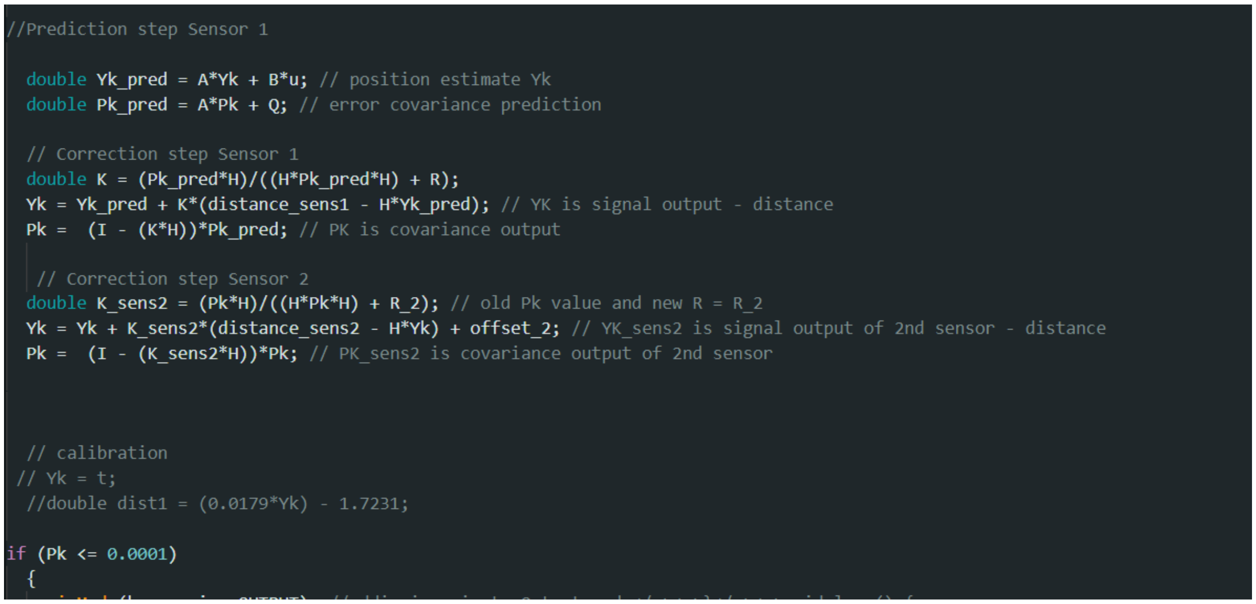


```
  // R values for sensor 1
if ( distance_sens1>=18 &&  distance_sens1<28) // 20cm
  {R = 0.00292053;
  R_2 = 0.002898734;
  offset_2 = 0;
 }
 else if ( distance_sens1>=28 &&  distance_sens1<38) // 30cm
 { R = 0.003250804;
 R_2 =0.002246658 ;
 offset_2 = 0;
 }
 else if ( distance_sens1>=38 &&  distance_sens1<48) // 40 cm
 { R = 0.002575729;
 R_2 = 0.003132064;
 offset_2 = 0;
 }
 else if ( distance_sens1>= 48 &&  distance_sens1<58) // 50 cm
 { R = 0.002574491;
 R_2 =0.002294174 ;
 offset_2 = -0.05;
 }
 else if ( distance_sens1>= 58 &&  distance_sens1<68) // 60 cm
 { R = 0.003695252;
  R_2 = 0.002803537;
  offset_2 = -0.05;
 }
 else if ( distance_sens1>= 68 &&  distance_sens1<78) // 70 cm
 { R = 0.001342988;
 R_2 = 0.001447324;
 offset_2 = -0.11;
 }
 else if ( distance_sens1>=78 &&  distance_sens1<88) // 80 cm
```

Kalman Filter:

The major difference between part 1.1 and part 1.2 is the implementation of 2 sensors and the updation of the Kalman equations. The prediction equation remains the same, using the variance of sensor 1, but now there are 2 correction steps. First the correction is done by the sensor 1 and then sensor 3 does the 2nd correction. This leads to a very accurate value and a fast convergence time. The speed of the convergence can be shortened by adding a smaller delay.

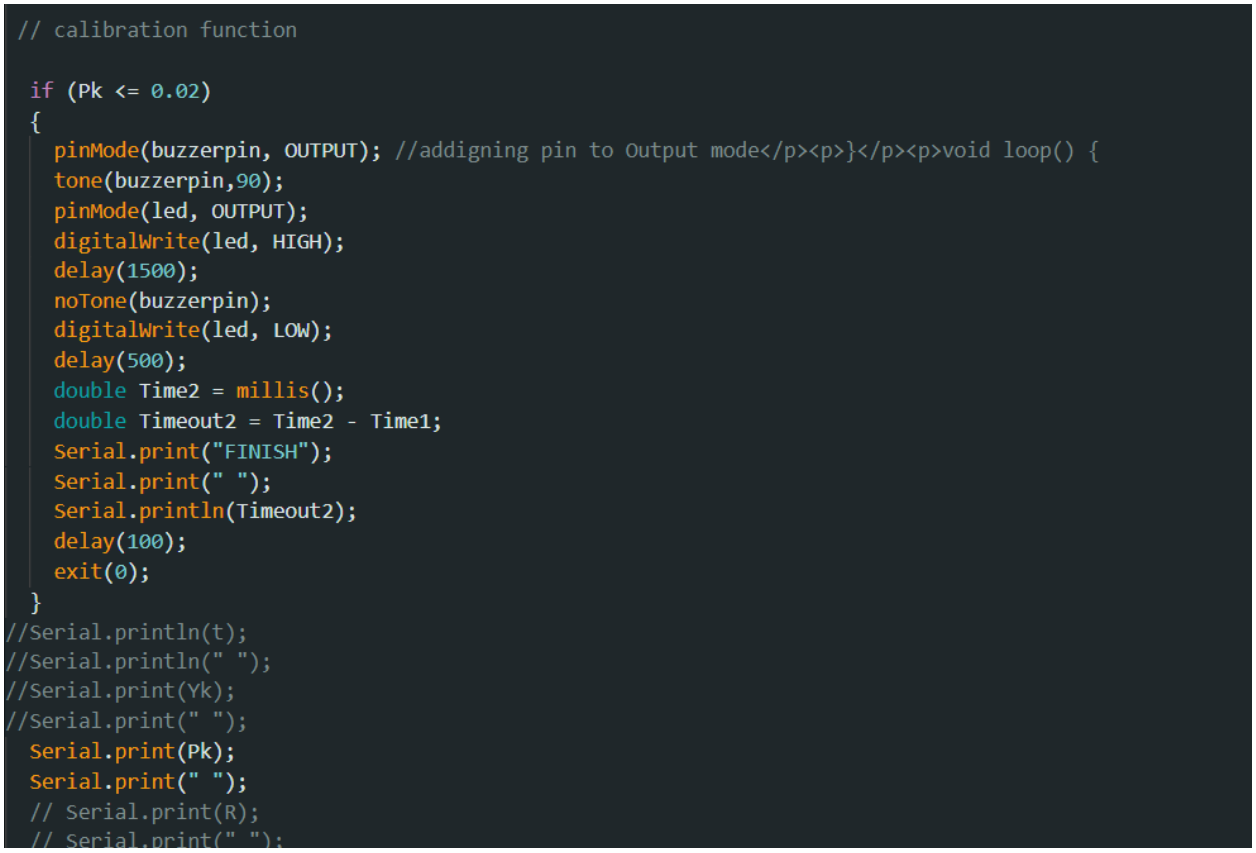
There will not be any calibration function after the Kalman as this is already calculating it in distance.



```
//Prediction step Sensor 1

  double Yk_pred = A*Yk + B*u; // position estimate Yk
  double Pk_pred = A*Pk + Q; // error covariance prediction

  // Correction step Sensor 1
  double K = (Pk_pred*H)/((H*Pk_pred*H) + R);
  Yk = Yk_pred + K*(distance_sens1 - H*Yk_pred); // YK is signal output - distance
  Pk =  (I - (K*H))*Pk_pred; // PK is covariance output

   // Correction step Sensor 2
  double K_sens2 = (Pk*H)/((H*Pk*H) + R_2); // old Pk value and new R = R_2
  Yk = Yk + K_sens2*(distance_sens2 - H*Yk) + offset_2; // YK_sens2 is signal output of 2nd sensor - distance
  Pk =  (I - (K_sens2*H))*Pk; // PK_sens2 is covariance output of 2nd sensor



  // calibration
 // Yk = t;
  //double dist1 = (0.0179*Yk) - 1.7231;

if (Pk <= 0.0001)
  {
```

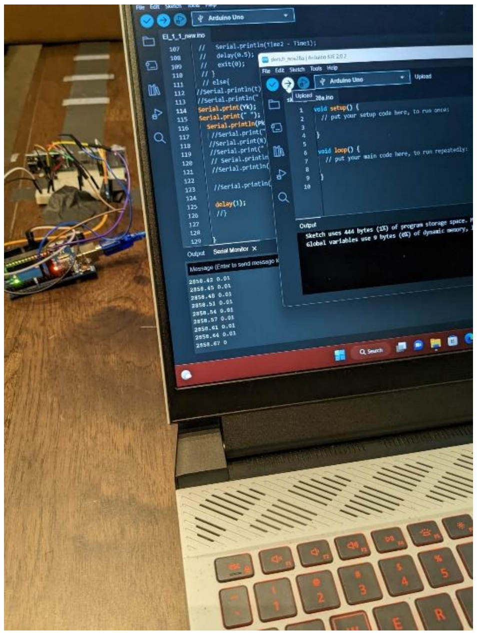
Stop statement:

This statement is added to exit the void loop and stop which will give us the final reading. The buzzer and LED are then activated when this is achieved.



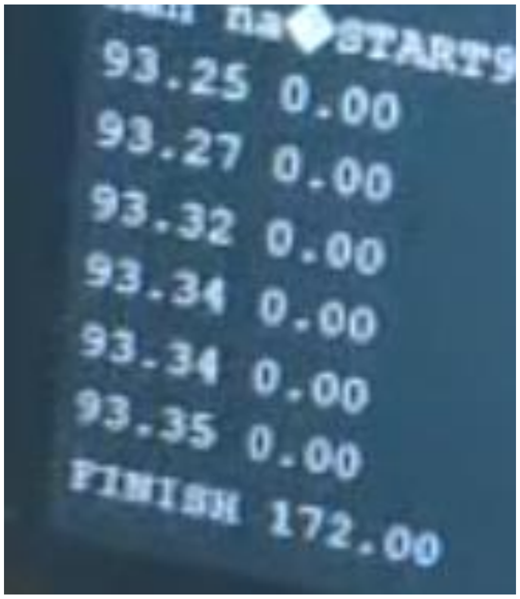
```
 // calibration function

  if (Pk <= 0.02)
  {
    pinMode(buzzerpin, OUTPUT); //addigning pin to Output mode</p><p>}</p><p>void loop() {
    tone(buzzerpin,90);
    pinMode(led, OUTPUT);
    digitalWrite(led, HIGH);
    delay(1500);
    noTone(buzzerpin);
    digitalWrite(led, LOW);
    delay(500);
    double Time2 = millis();
    double Timeout2 = Time2 - Time1;
    Serial.print("FINISH");
    Serial.print(" ");
    Serial.println(Timeout2);
    delay(100);
    exit(0);
  }
//Serial.println(t);
//Serial.println(" ");
//Serial.print(Yk);
//Serial.print(" ");
  Serial.print(Pk);
  Serial.print(" ");
  // Serial.print(R);
  // Serial.print(" ");
```

2.4 Experimental Results



Tests were conducted for different distances:

Distance - cm	**93**
Sensor Reading - cm	**93.35**
Convergence Time – milli seconds	**172**



```
START
93.25 0.00
93.27 0.00
93.32 0.00
93.34 0.00
93.34 0.00
93.35 0.00
FINISH 172.00
```

3. Conclusions and Discussions

3.1 Conclusions (a summary the results of different approaches)

Comparing the results of Project 1.1 and Project 1.2 we can see that the convergence time and the accuracy of the results increases in Project 1.2 because of 2 correction cycles.

As clearly seen from the above results; the convergence time for 1.1 is 7909 milli seconds vs 172 milli seconds

3.2 Discussions (a comparison of different approaches and potential future work to further improve each approach)

A sensor which is more stable to environmental changes can be used and a proper pedestal/ stand for the sensors to sit on must be used so that any error that arises can be eliminated.

Notice: The data in an individual report should be recorded individually. Students in one group can use the same hardware and software but cannot use the same data in the report.

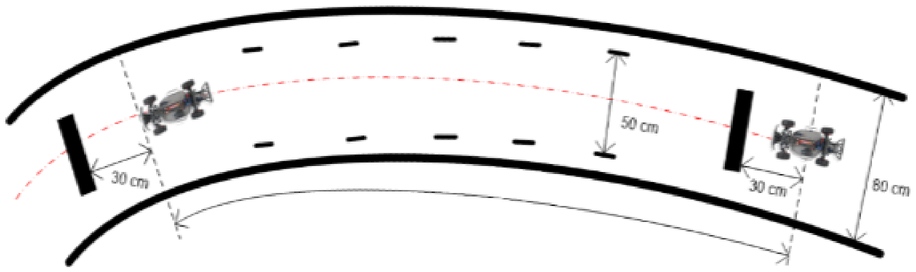
Project 2 Report: Adaptive Cruise Control and Autonomous Lane Keeping

Rohit Ravikumar – Group 9

1. Adaptive Cruise Control

1.1 Problem Statement

The intention or goal of this project is to implement a control strategy and control the throttle of the RC car to maintain adaptive cruise control. The car must initially be placed close to a box near the start line, and the car must adjust itself to exactly 30cms from the box autonomously. At the end, the car must come to a halt at a distance of 30cms from the box to complete the test.



1.2 Technical Approach

To control the steering, a PID controller is used that takes the reference point or set point as 30cm, which is specified in the rules for the task.

An ultrasonic sensor is placed in the front of the car that acts as the feedback input for the PID controller. This sensor samples the distance at a high rate and the throttle is controlled based on the error value and the gains set for the controller.

The value for 0 throttle is 90 and adding or subtracting this will make the car either accelerate in the forward or reverse direction.

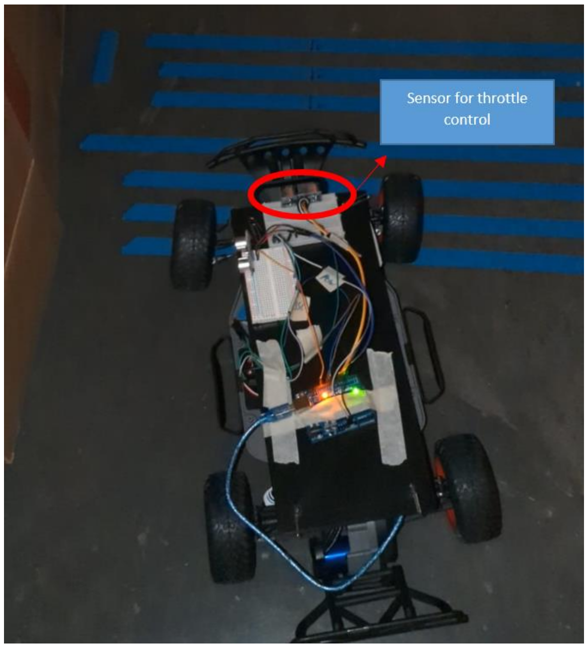


Fig 1.2.1 RC car – with front sensor

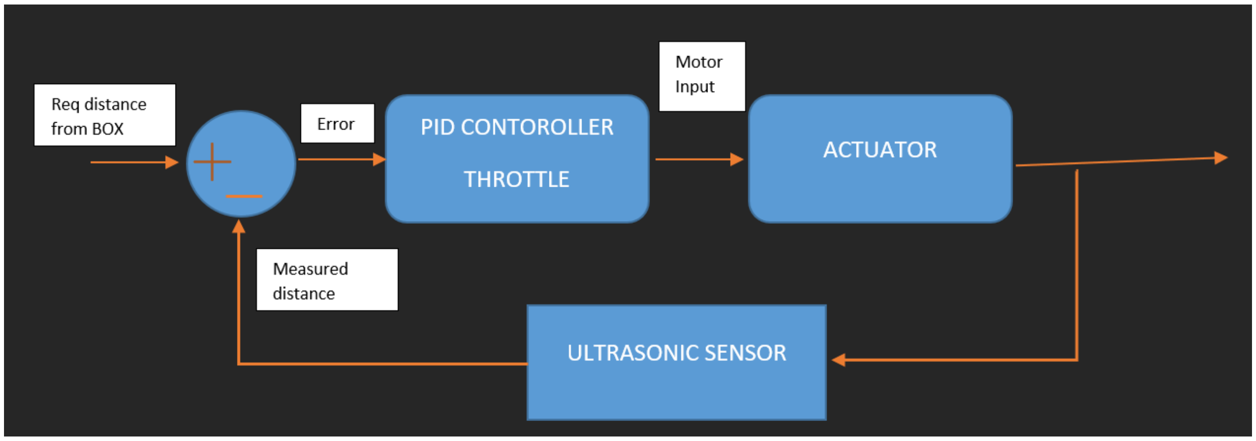


Fig 1.2.2 PID flow diagram

1.3 Hardware and Software Implementation

Hardware Wiring:

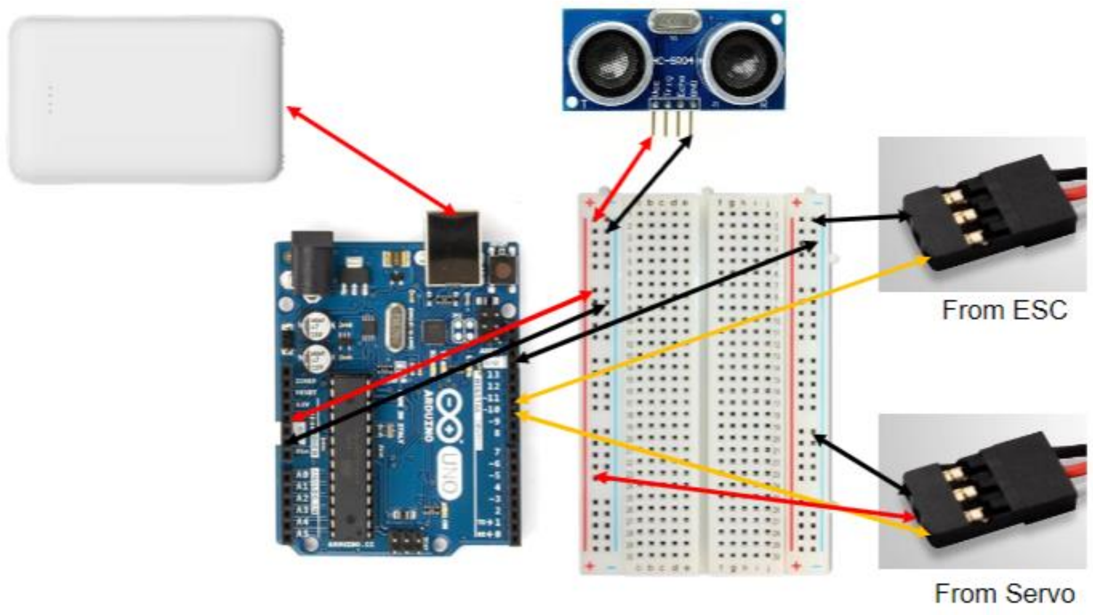


Fig 1.3.1 Hardware wiring

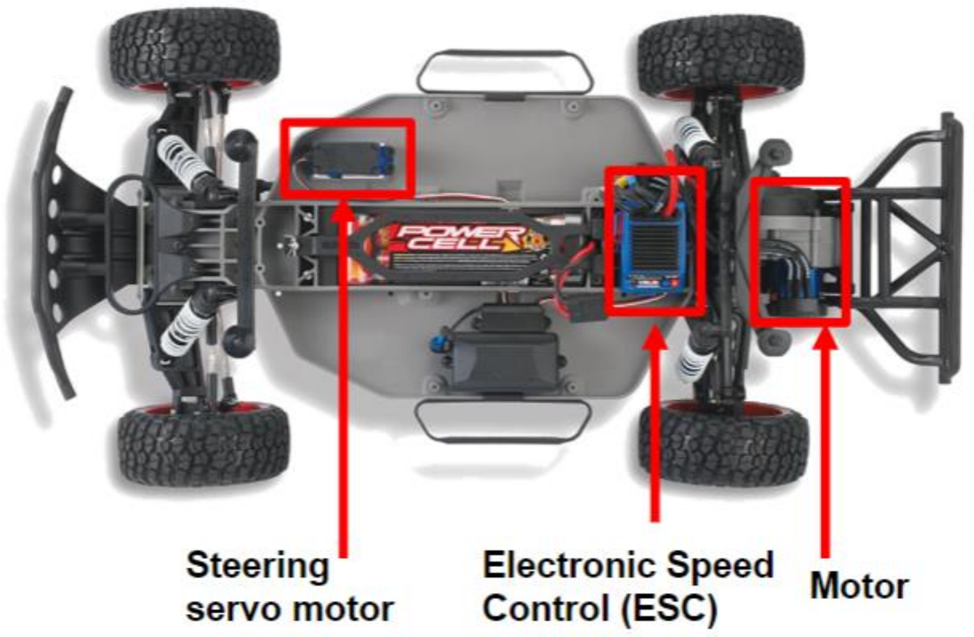


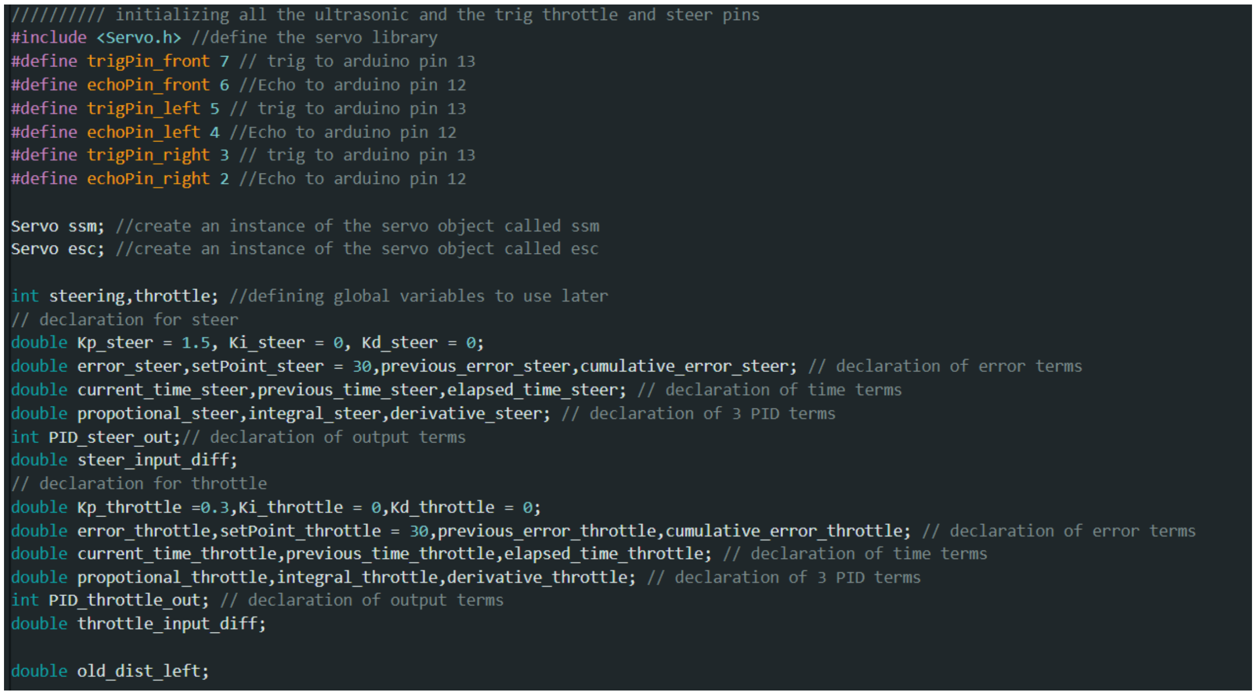
Fig 1.3.2 RC Car

Software Implementation:

1) Variables Initialization:

Initializing the Ultrasonic sensor pins and the servo library, along with the Servo ssm and the Servo esc.

The global variables to be used in the PID for the steering control are also initialized.



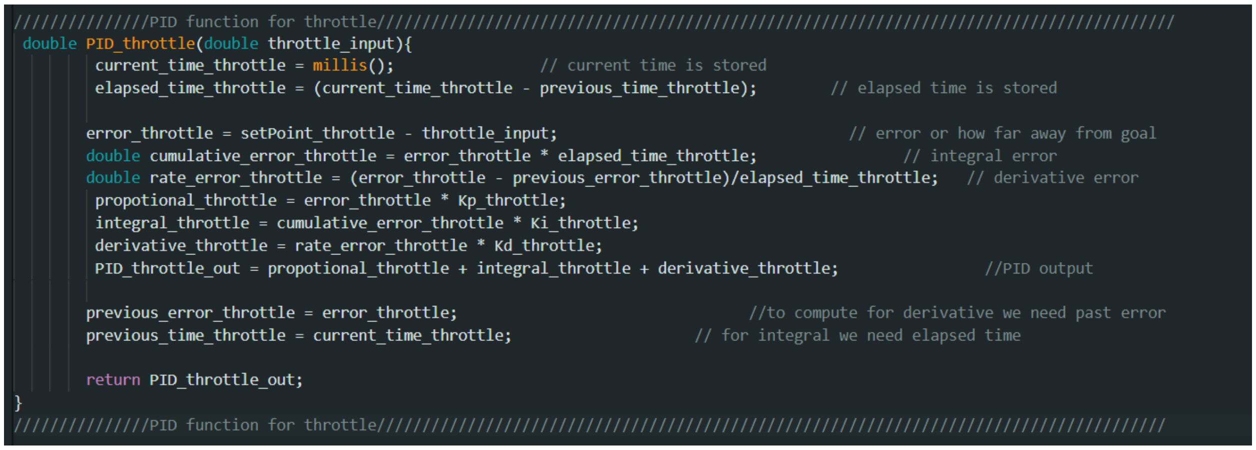
```
////////// initializing all the ultrasonic and the trig throttle and steer pins
#include <Servo.h> //define the servo library
#define trigPin_front 7 // trig to arduino pin 13
#define echoPin_front 6 //Echo to arduino pin 12
#define trigPin_left 5 // trig to arduino pin 13
#define echoPin_left 4 //Echo to arduino pin 12
#define trigPin_right 3 // trig to arduino pin 13
#define echoPin_right 2 //Echo to arduino pin 12

Servo ssm; //create an instance of the servo object called ssm
Servo esc; //create an instance of the servo object called esc

int steering,throttle; //defining global variables to use later
// declaration for steer
double Kp_steer = 1.5, Ki_steer = 0, Kd_steer = 0;
double error_steer,setPoint_steer = 30,previous_error_steer,cumulative_error_steer; // declaration of error terms
double current_time_steer,previous_time_steer,elapsed_time_steer; // declaration of time terms
double propotional_steer,integral_steer,derivative_steer; // declaration of 3 PID terms
int PID_steer_out;// declaration of output terms
double steer_input_diff;
// declaration for throttle
double Kp_throttle =0.3,Ki_throttle = 0,Kd_throttle = 0;
double error_throttle,setPoint_throttle = 30,previous_error_throttle,cumulative_error_throttle; // declaration of error terms
double current_time_throttle,previous_time_throttle,elapsed_time_throttle; // declaration of time terms
double propotional_throttle,integral_throttle,derivative_throttle; // declaration of 3 PID terms
int PID_throttle_out; // declaration of output terms
double throttle_input_diff;

double old_dist_left;
```

2) PID for Throttle:

The PID function for steering has been implemented using the throttle input as the input to the PID function. The steer input is the output from the ultrasonic sensor in distance. The PID function then uses this distance and subtracts it from the set point to give the PID output.

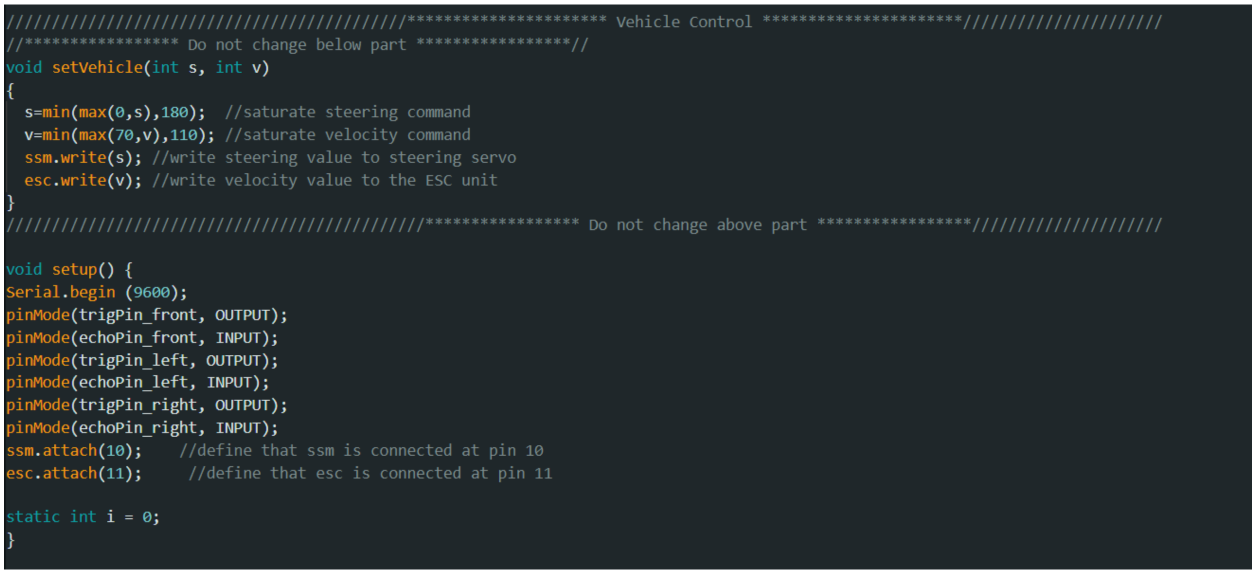
Input = throttle_input

Output = PID_throttle_out



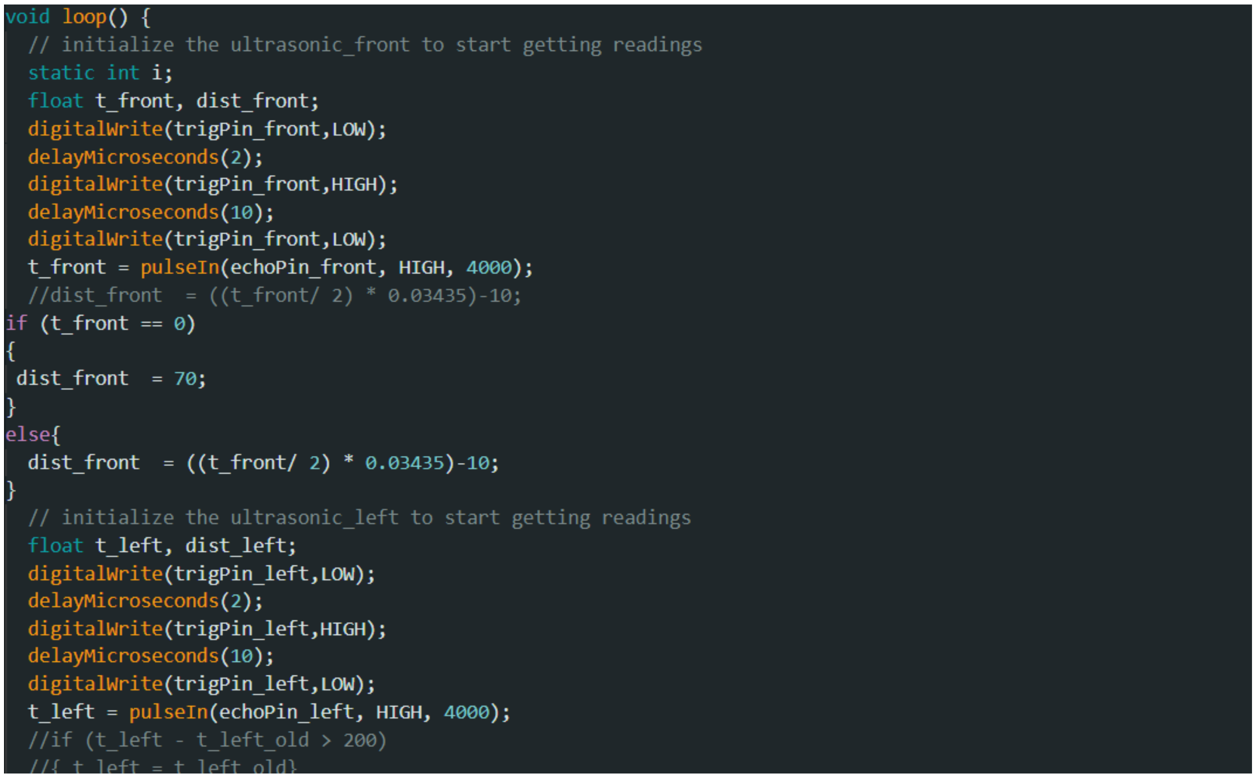
```
//////////////PID function for throttle/////////////////////////////////////////////////////////////////////////////////
 double PID_throttle(double throttle_input){
         current_time_throttle = millis();                // current time is stored
         elapsed_time_throttle = (current_time_throttle - previous_time_throttle);        // elapsed time is stored

        error_throttle = setPoint_throttle - throttle_input;                              // error or how far away from goal
        double cumulative_error_throttle = error_throttle * elapsed_time_throttle;               // integral error
        double rate_error_throttle = (error_throttle - previous_error_throttle)/elapsed_time_throttle;   // derivative error
         propotional_throttle = error_throttle * Kp_throttle;
         integral_throttle = cumulative_error_throttle * Ki_throttle;
         derivative_throttle = rate_error_throttle * Kd_throttle;
         PID_throttle_out = propotional_throttle + integral_throttle + derivative_throttle;                //PID output

        previous_error_throttle = error_throttle;                              //to compute for derivative we need past error
        previous_time_throttle = current_time_throttle;                  // for integral we need elapsed time

        return PID_throttle_out;
}
//////////////PID function for throttle////////////////////////////////////////////////////////////////////////////////
```

3) Void Set vehicle: This piece of code, controls the steering commands and limits it to a fixed value so as to not cause actuator burnout.

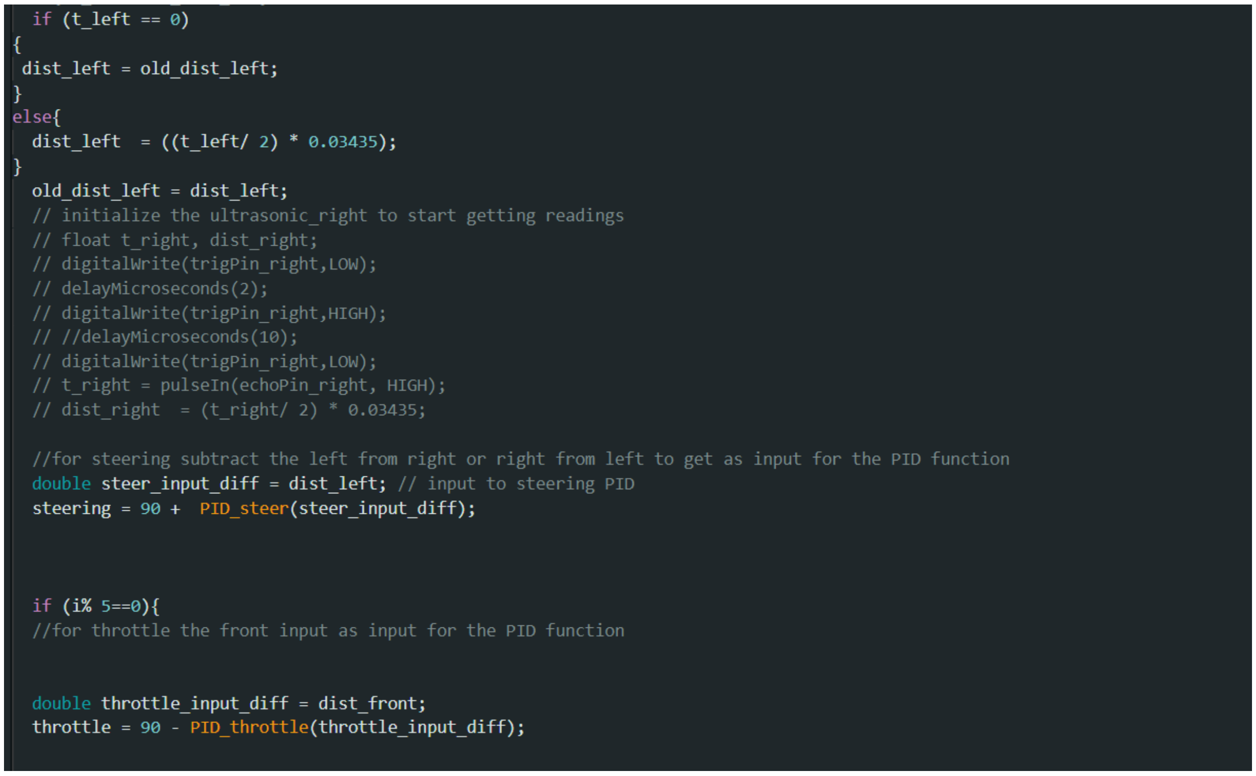


```
////////////////////////////////////////////*********************** Vehicle Control **********************//////////////////////
//***************** Do not change below part *****************//
void setVehicle(int s, int v)
{
  s=min(max(0,s),180);  //saturate steering command
  v=min(max(70,v),110); //saturate velocity command
  ssm.write(s); //write steering value to steering servo
  esc.write(v); //write velocity value to the ESC unit
}
/////////////////////////////////////////////***************** Do not change above part *****************////////////////////

void setup() {
Serial.begin (9600);
pinMode(trigPin_front, OUTPUT);
pinMode(echoPin_front, INPUT);
pinMode(trigPin_left, OUTPUT);
pinMode(echoPin_left, INPUT);
pinMode(trigPin_right, OUTPUT);
pinMode(echoPin_right, INPUT);
ssm.attach(10);    //define that ssm is connected at pin 10
esc.attach(11);     //define that esc is connected at pin 11

static int i = 0;
}
```

4) Void Loop: In the void loop, the ultrasonic sensor output is converted to a distance value using a calibration function.
This is then assigned to the variable dist_left.
This acts as the input to the PID_steer control. The output from the PID controller goes to the Void set vehicle function as a steering command which sets the vehicle steering angle and controls the car. The value is 90 when the error is 0 and the PID out put is added to this value.



```
void loop() {
  // initialize the ultrasonic_front to start getting readings
  static int i;
  float t_front, dist_front;
  digitalWrite(trigPin_front,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin_front,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin_front,LOW);
  t_front = pulseIn(echoPin_front, HIGH, 4000);
  //dist_front  = ((t_front/ 2) * 0.03435)-10;
if (t_front == 0)
{
 dist_front  = 70;
}
else{
  dist_front  = ((t_front/ 2) * 0.03435)-10;
}
  // initialize the ultrasonic_left to start getting readings
  float t_left, dist_left;
  digitalWrite(trigPin_left,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin_left,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin_left,LOW);
  t_left = pulseIn(echoPin_left, HIGH, 4000);
  //if (t_left - t_left_old > 200)
  //{ t_left = t_left_old}
```



```
  if (t_left == 0)
{
 dist_left = old_dist_left;
}
else{
  dist_left  = ((t_left/ 2) * 0.03435);
}
  old_dist_left = dist_left;
  // initialize the ultrasonic_right to start getting readings
  // float t_right, dist_right;
  // digitalWrite(trigPin_right,LOW);
  // delayMicroseconds(2);
  // digitalWrite(trigPin_right,HIGH);
  // //delayMicroseconds(10);
  // digitalWrite(trigPin_right,LOW);
  // t_right = pulseIn(echoPin_right, HIGH);
  // dist_right  = (t_right/ 2) * 0.03435;

  //for steering subtract the left from right or right from left to get as input for the PID function
  double steer_input_diff = dist_left; // input to steering PID
  steering = 90 +  PID_steer(steer_input_diff);



  if (i% 5==0){
  //for throttle the front input as input for the PID function


  double throttle_input_diff = dist_front;
  throttle = 90 - PID_throttle(throttle_input_diff);
```

1.4 Experimental Results

The throttle was tested extensively to give the best possible speed and not to be too fast so as to make the steering lag. Hence after extensive testing the final PID values for the throttle control were as follows:

Kp = 0.3

Kd = 0

Ki = 0

The Ki and Kd values are zero even though it made a lot of sense to have a non-zero Kd value, as that would reduce oscillations while stopping.

But this gave us a satisfactory result.

Another way to reduce the speed was to give the throttle input only once every 5 times the loop ran. This gave the car a very controlled throttle input and as the car was still moving at a decent pace we chose this configuration.

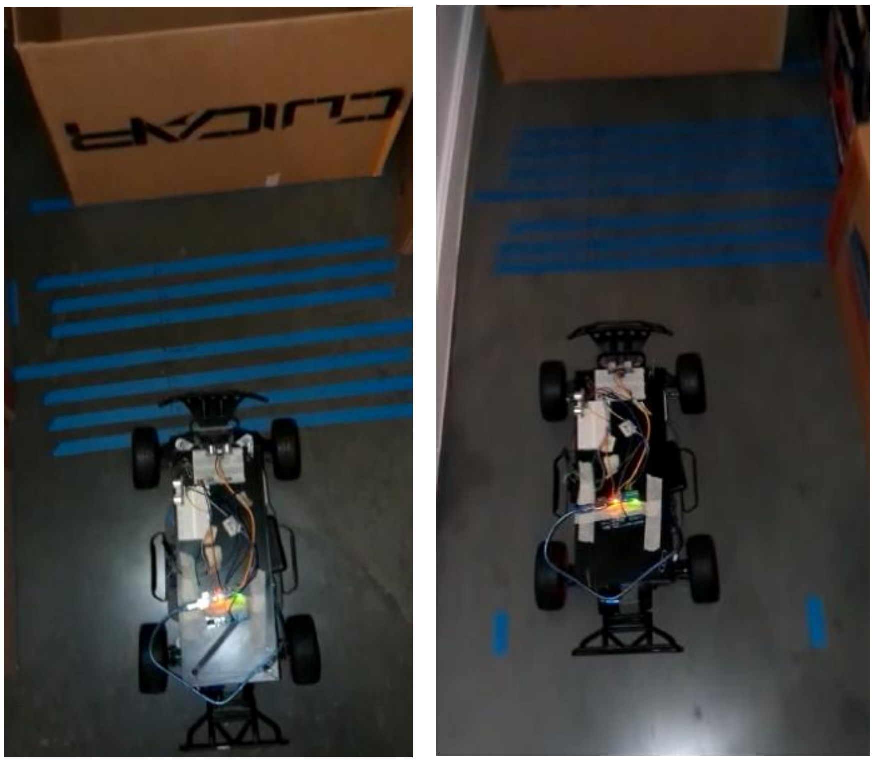


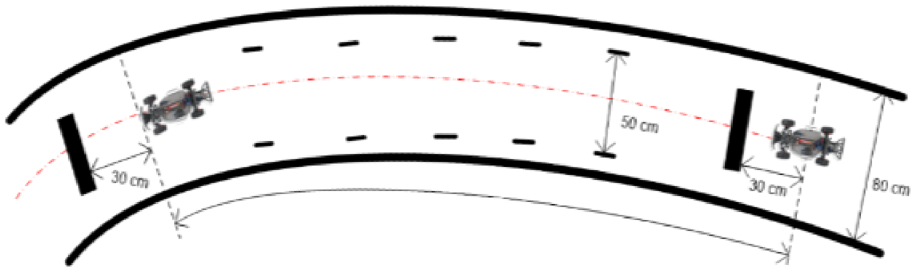
Fig 1.4.1 Final Test

The final test was successful, except for the final stopping part. The car had been charged completely before the test and hence due to a high SOC the torque value increased and the car carried too much momentum into the straight and hence ended up touching the box a bit. The Kp values had been tuned for a lower SOC value and hence torque was not so high as seen during the test.

2. Autonomous Lane Keeping

2.1 Problem Statement

The intention or goal of this project is to implement a control strategy and control the steering of the RC car to maintain the lane position. A wall or barrier is present on either side and the RC car has to reach the goal by maintaining its lane and completing the course as fast as possible.



2.2 Technical Approach

1) The steering system was completely decoupled from the throttle and tested separately.

Initially the steering wheel was un-calibrated and the steering neutral or for steering angle zero, the Steer value was Steering = 94.

It was then calibrated, and we got the final steer 90-degree value to Steering = 90.

2) Once the steering was tested with a basic code to just check its working, the PID for the steering wheel was implemented.

The process is as shown below

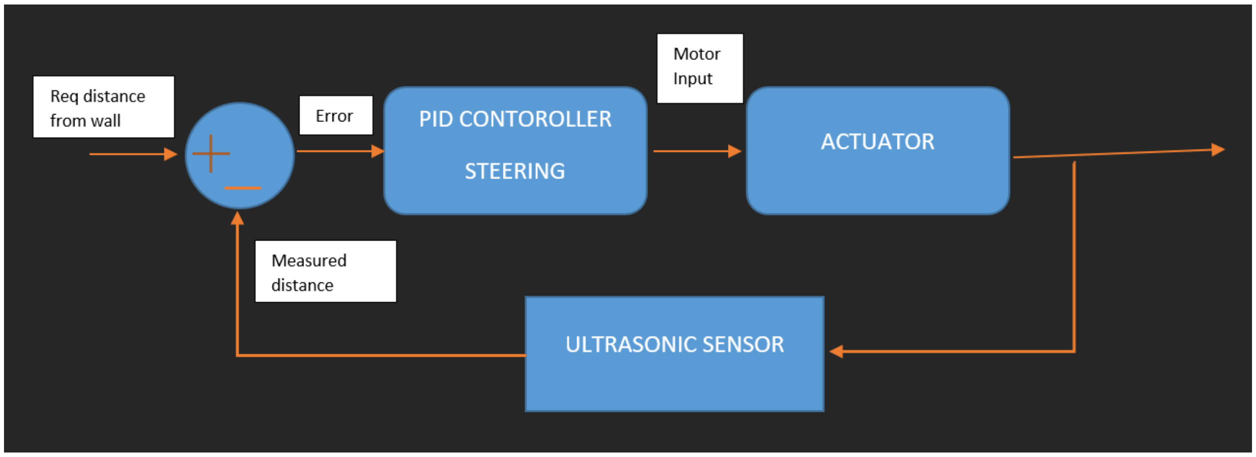


Fig 2.2.1 PID steering control flow diagram

The measured distance between the tracks or the lane width was 75 mm and the width of the plate on which the sensors are placed is 15mm. Hence the car must maintain a distance of:

(75 – 15) / 2 mm = 30mm from the wall to remain in the center of the lane markers

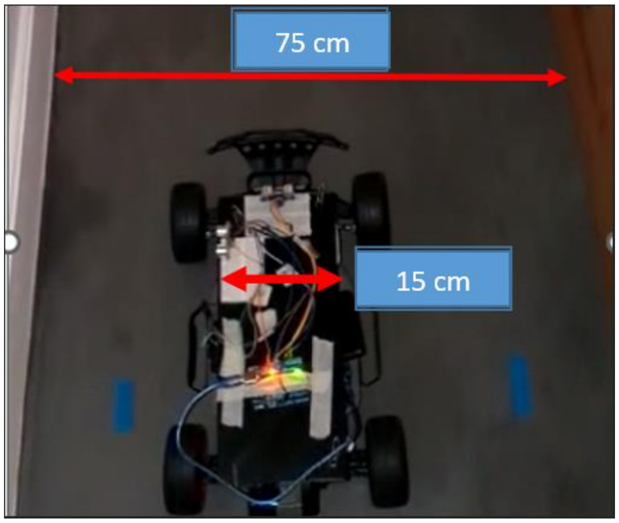


Fig 2.2.2 track and car dimensions

So, the setpoint for the PID controller was set to 30mm.

The car had to now maintain the distance from the edge of the sensor to the wall at a distance of 30mm. If the car went closer to the wall then the error would decrease and hence the PID output would become non zero and hence the car would steer away from the wall.

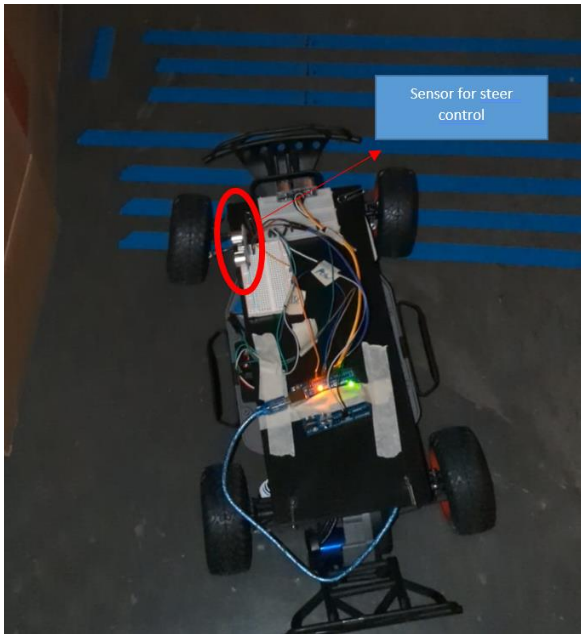


Fig 2.2.3 Sensor setup

2.3 Hardware and Software Implementation

Hardware Wiring:

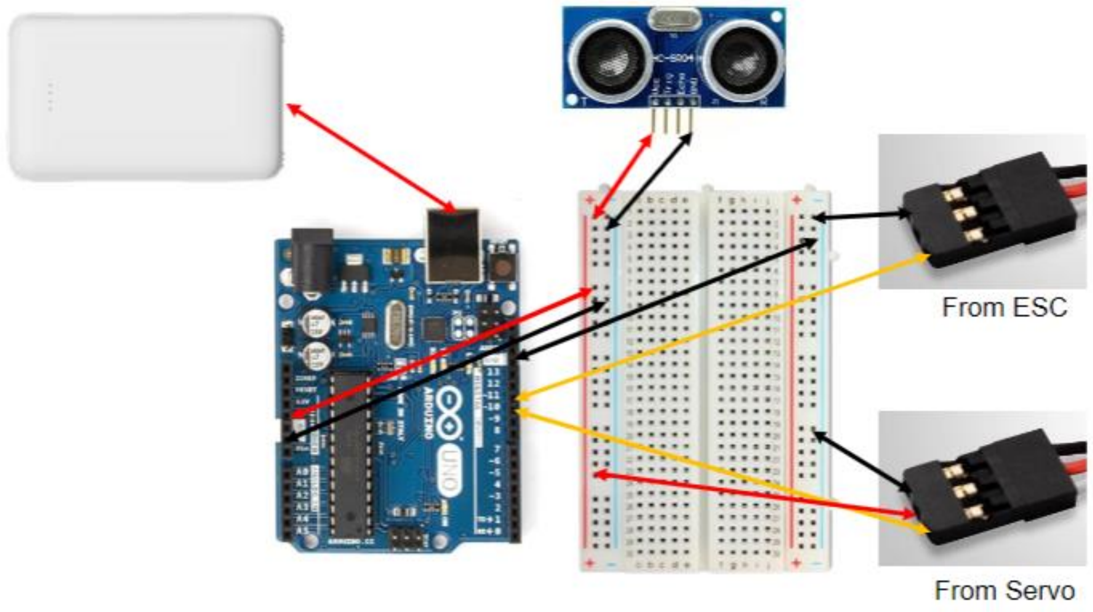


Fig 2.3.1 Hardware wiring

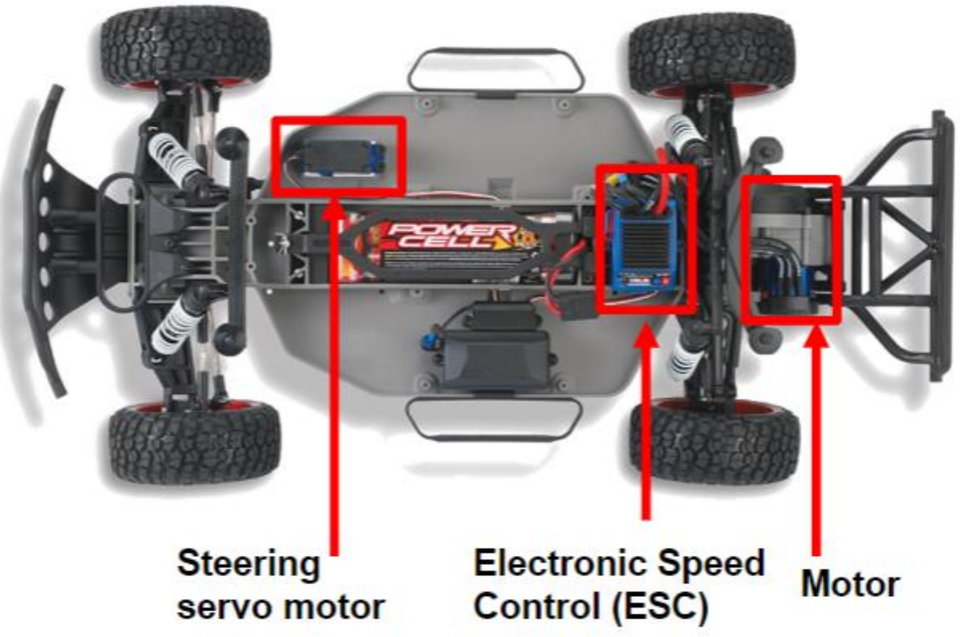


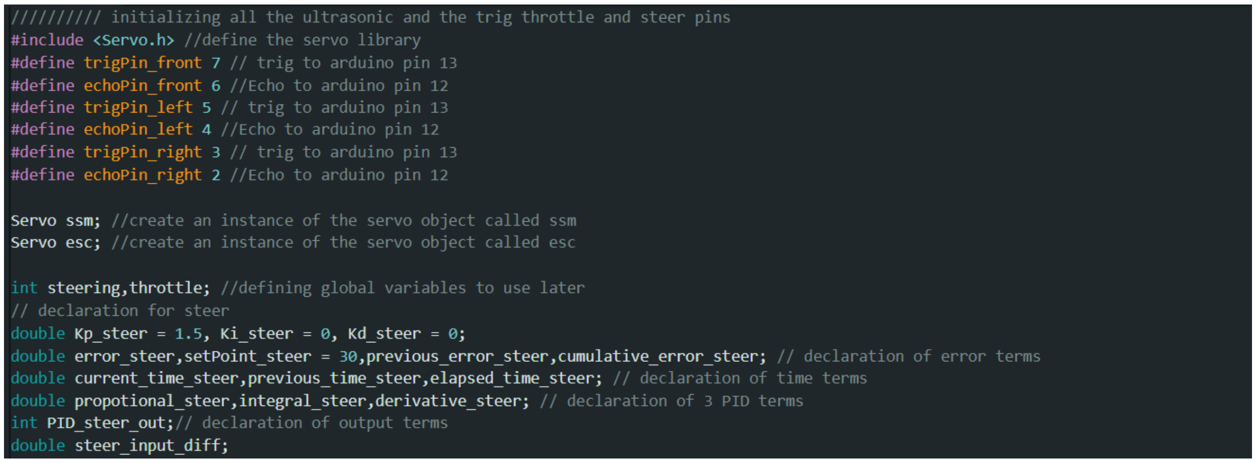
Fig 2.3.2 RC Car

Software Implementation:

1) Variables Initialization:

Initializing the Ultrasonic sensor pins and the servo library, along with the Servo ssm and the Servo esc.

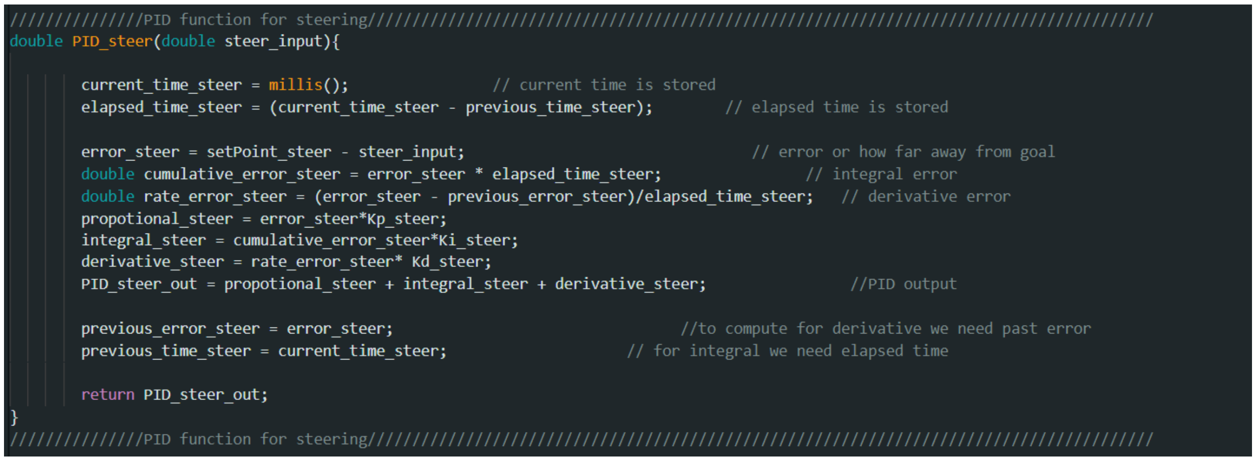
The global variables to be used in the PID for the steering control are also initialized.



```
////////// initializing all the ultrasonic and the trig throttle and steer pins
#include <Servo.h> //define the servo library
#define trigPin_front 7 // trig to arduino pin 13
#define echoPin_front 6 //Echo to arduino pin 12
#define trigPin_left 5 // trig to arduino pin 13
#define echoPin_left 4 //Echo to arduino pin 12
#define trigPin_right 3 // trig to arduino pin 13
#define echoPin_right 2 //Echo to arduino pin 12

Servo ssm; //create an instance of the servo object called ssm
Servo esc; //create an instance of the servo object called esc

int steering,throttle; //defining global variables to use later
// declaration for steer
double Kp_steer = 1.5, Ki_steer = 0, Kd_steer = 0;
double error_steer,setPoint_steer = 30,previous_error_steer,cumulative_error_steer; // declaration of error terms
double current_time_steer,previous_time_steer,elapsed_time_steer; // declaration of time terms
double propotional_steer,integral_steer,derivative_steer; // declaration of 3 PID terms
int PID_steer_out;// declaration of output terms
double steer_input_diff;
```

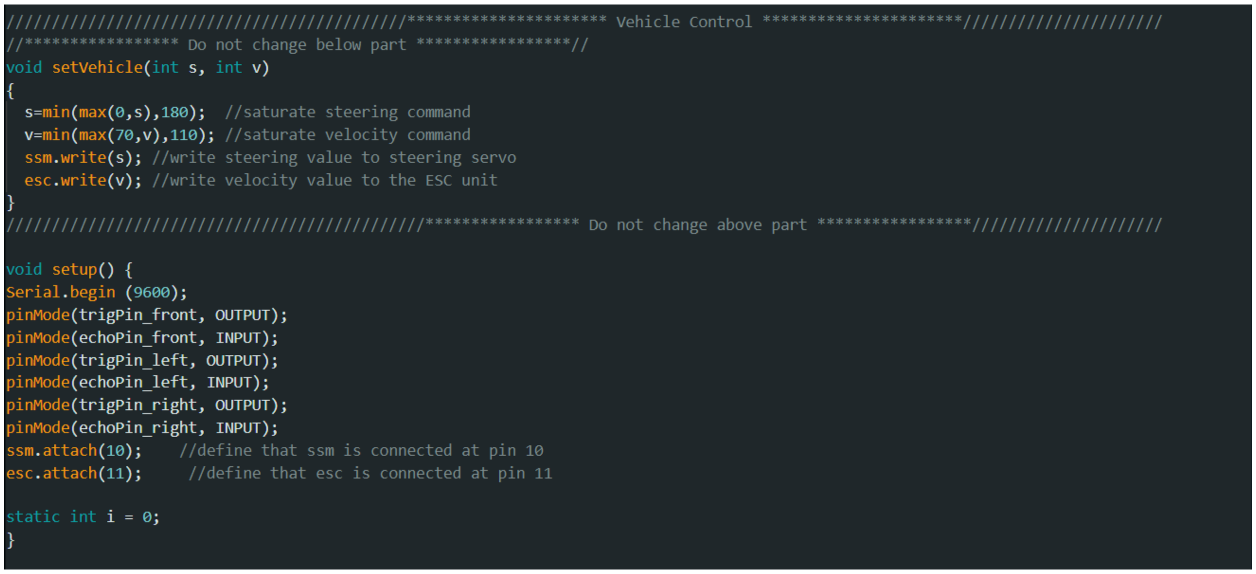
2) PID for Steering:

The PID function for steering has been implemented using the steer input as the input to the PID function. The steer input is the output from the ultrasonic sensor in distance. The PID function then uses this distance and subtracts it from the set point to give the PID output.



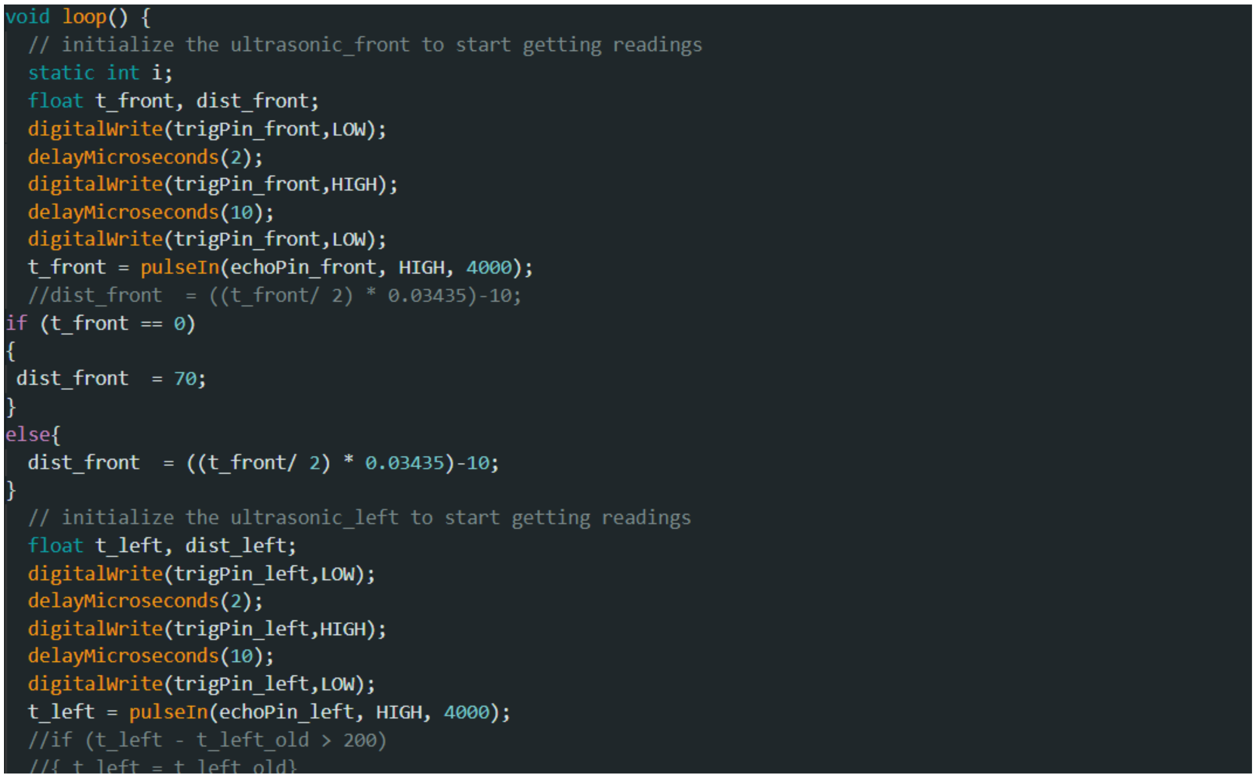
```
//////////////PID function for steering////////////////////////////////////////////////////////////////////////////////////////
double PID_steer(double steer_input){

        current_time_steer = millis();                  // current time is stored
        elapsed_time_steer = (current_time_steer - previous_time_steer);        // elapsed time is stored

        error_steer = setPoint_steer - steer_input;                               // error or how far away from goal
        double cumulative_error_steer = error_steer * elapsed_time_steer;               // integral error
        double rate_error_steer = (error_steer - previous_error_steer)/elapsed_time_steer;    // derivative error
        propotional_steer = error_steer*Kp_steer;
        integral_steer = cumulative_error_steer*Ki_steer;
        derivative_steer = rate_error_steer* Kd_steer;
        PID_steer_out = propotional_steer + integral_steer + derivative_steer;                //PID output

        previous_error_steer = error_steer;                               //to compute for derivative we need past error
        previous_time_steer = current_time_steer;                   // for integral we need elapsed time

        return PID_steer_out;
}
//////////////PID function for steering////////////////////////////////////////////////////////////////////////////////////////
```

3) Void Set vehicle: This piece of code, controls the steering commands and limits it to a fixed value so as to not cause actuator burnout.

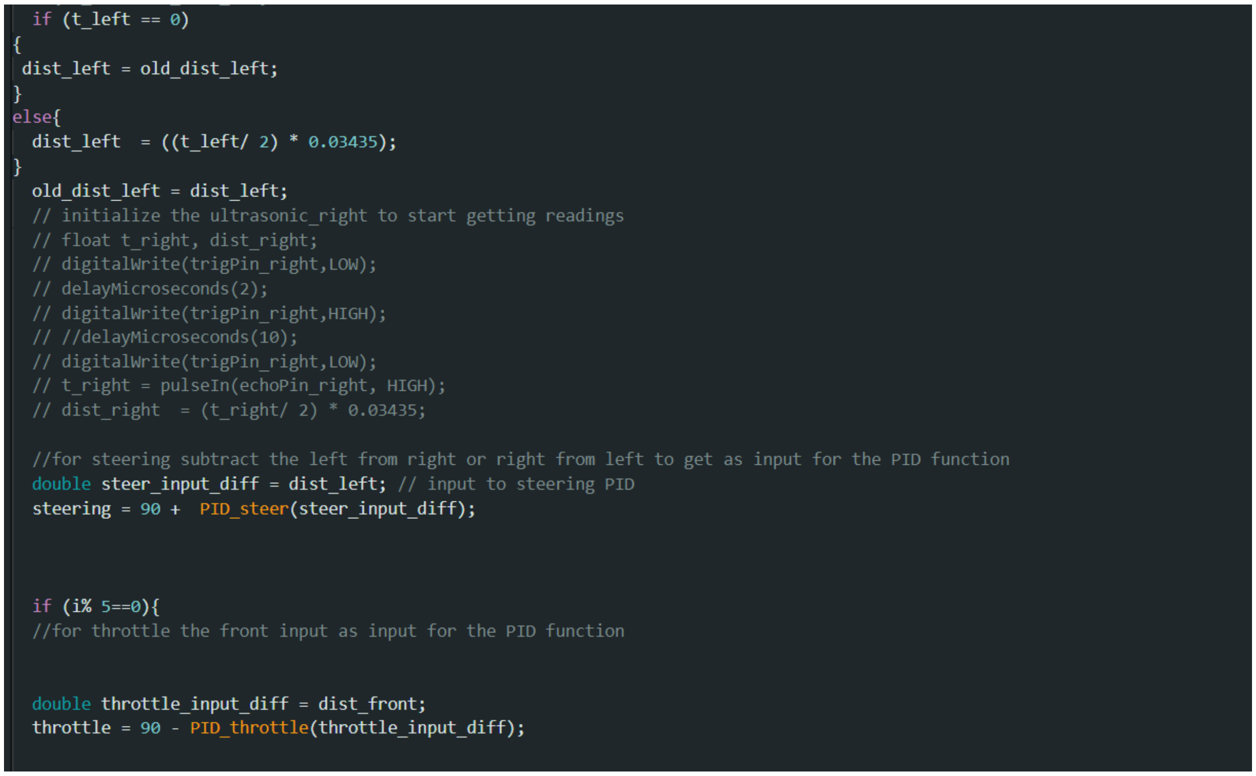


```
////////////////////////////////////////////*********************** Vehicle Control **********************/////////////////////
//***************** Do not change below part *****************//
void setVehicle(int s, int v)
{
  s=min(max(0,s),180);  //saturate steering command
  v=min(max(70,v),110); //saturate velocity command
  ssm.write(s); //write steering value to steering servo
  esc.write(v); //write velocity value to the ESC unit
}
/////////////////////////////////////////////***************** Do not change above part *****************////////////////////

void setup() {
Serial.begin (9600);
pinMode(trigPin_front, OUTPUT);
pinMode(echoPin_front, INPUT);
pinMode(trigPin_left, OUTPUT);
pinMode(echoPin_left, INPUT);
pinMode(trigPin_right, OUTPUT);
pinMode(echoPin_right, INPUT);
ssm.attach(10);    //define that ssm is connected at pin 10
esc.attach(11);     //define that esc is connected at pin 11

static int i = 0;
}
```

4) Void Loop: In the void loop, the ultrasonic sensor output is converted to a distance value using a calibration function.
   This is then assigned to the variable dist_left.
   This acts as the input to the PID_steer control. The output from the PID controller goes to the Void set vehicle function as a steering command which sets the vehicle steering angle and controls the car. The value is 90 when the error is 0 and the PID out put is added to this value.



```
void loop() {
  // initialize the ultrasonic_front to start getting readings
  static int i;
  float t_front, dist_front;
  digitalWrite(trigPin_front,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin_front,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin_front,LOW);
  t_front = pulseIn(echoPin_front, HIGH, 4000);
  //dist_front  = ((t_front/ 2) * 0.03435)-10;
if (t_front == 0)
{
 dist_front  = 70;
}
else{
  dist_front  = ((t_front/ 2) * 0.03435)-10;
}
  // initialize the ultrasonic_left to start getting readings
  float t_left, dist_left;
  digitalWrite(trigPin_left,LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin_left,HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin_left,LOW);
  t_left = pulseIn(echoPin_left, HIGH, 4000);
  //if (t_left - t_left_old > 200)
  //{ t_left = t_left_old}
```



```
  if (t_left == 0)
{
 dist_left = old_dist_left;
}
else{
  dist_left  = ((t_left/ 2) * 0.03435);
}
  old_dist_left = dist_left;
  // initialize the ultrasonic_right to start getting readings
  // float t_right, dist_right;
  // digitalWrite(trigPin_right,LOW);
  // delayMicroseconds(2);
  // digitalWrite(trigPin_right,HIGH);
  // //delayMicroseconds(10);
  // digitalWrite(trigPin_right,LOW);
  // t_right = pulseIn(echoPin_right, HIGH);
  // dist_right  = (t_right/ 2) * 0.03435;

  //for steering subtract the left from right or right from left to get as input for the PID function
  double steer_input_diff = dist_left; // input to steering PID
  steering = 90 +  PID_steer(steer_input_diff);



  if (i% 5==0){
  //for throttle the front input as input for the PID function


  double throttle_input_diff = dist_front;
  throttle = 90 - PID_throttle(throttle_input_diff);
```

2.4 Experimental Results

1) Upon rigorous testing, the final values for the PID steer were as follows:

Kp = 1.5

Kd = 0

Ki = 0

It was noticed that at higher Kp values, the steer motor over corrected and de stabilized the vehicle more leading to excessive yam moments in the car.

The Kd and Ki values were kept at 0 as introducing these only increased the instability and the car had a very jerky steering.

Initially, we were using 2 sensors to read the wall distance on either side and those values were subtracted to get the final input as the input to the PID. If that value was 0 then the car is in the center of the lane and if it was non-zero, then there was a steering input that was to be given.

But upon further investigation, we found that the car was not very responsive, and so we switched to a single sensor that read the side wall distance and tried to maintain that distance.

2) After a particular number of iterations, it was noticed that the car was not turning in properly, but instead turned into the corner. We then tested this using two cardboard boxes and found that when the cardboard was angles and brought near the sensor as shown in the figure below, instead

of the distance value decreasing, the value instead increased. This was corrected, by implementing a code which removed any erroneous value and took the previous distance values.

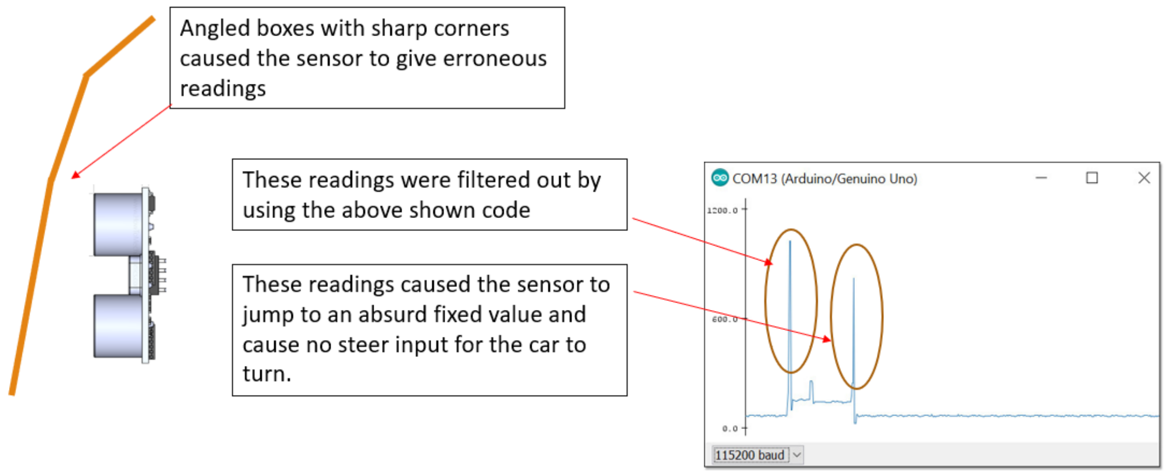

Angled boxes with sharp corners caused the sensor to give erroneous readings
These readings were filtered out by using the above shown code
These readings caused the sensor to jump to an absurd fixed value and cause no steer input for the car to turn.
COM13 (Arduino/Genuino Uno)
1200.0
600.0
0.0
115200 baud

Fig 2.4.1 Anomaly in readings

Final test was successful. The steering was spot on and the final Kp values were perfectly tuned to not cause over correction or under correction.

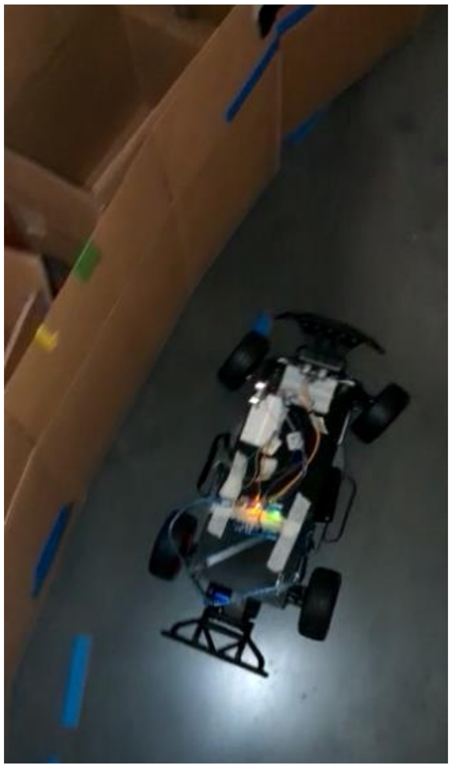


Fig 2.4.2 Final Test

3. Conclusions and Discussions

3.1 Conclusions (a summary the results of different approaches)

The final test was an overall success. The steering functioning didn’t depend on the battery Soc and hence there was no change in the functioning of that servo. The throttle on the other hand, had a few issues as the torque value increased on using a battery with a full SOC. All the Kp values for throttle were finalized based on the lower SOC value and hence it led to a bit of extra speed during the final test.

It was also noticed that when the car was tested with the sensor facing a wall instead of the cardboard box, the performance was very good, and it followed the wall with precise steering inputs.

3.2 Discussions (a comparison of different approaches, and potential future work to further improve each approach)

Overall using a PID controller was the best choice for the level of expertise possessed and the amount of time in hand. Complex control techniques such as reinforcement learning, and AI controllers may do the job much better and learn from rigorous testing.

Notice: The data in an individual report should be recorded individually. Students in one group can use the same hardware and software but cannot use the same data in the report.

CLEMSON UNIVERSITY

Rohit Ravikumar <rraviku@g.clemson.edu>

Thank you for completing a survey!
6 messages

EvaluationKIT Administrator <EKAdmin@clemson.edu> Fri, Dec 9, 2022 at 8:22 PM
To: Rohit Ravikumar <rraviku@clemson.edu>

Clemson University : Survey Certificate of Completion

202208 Fall 2022 Evaluations

Course: AUE-8350-001 : Auto Electronics : 202208-AUE-8350-001-83485-Jia

Instructor: Yunyi Jia

Submitted: 12/9/2022 8:20 PM

Student: Rohit Ravikumar

Thank you for completing a survey!

EvaluationKIT Administrator <EKAdmin@clemson.edu> Fri, Dec 9, 2022 at 8:24 PM
To: Rohit Ravikumar <rraviku@clemson.edu>

Clemson University : Survey Certificate of Completion

202208 Fall 2022 Evaluations

Course: AUE-8810-001 : Auto Sys Overview : 202208-AUE-8810-001-83479-Schmueser

Instructor: David Schmueser

Submitted: 12/9/2022 8:21 PM

Student: Rohit Ravikumar

Thank you for completing a survey!

EvaluationKIT Administrator <EKAdmin@clemson.edu> Fri, Dec 9, 2022 at 11:15 PM
To: Rohit Ravikumar <rraviku@clemson.edu>

Clemson University : Survey Certificate of Completion

202208 Fall 2022 Evaluations

Course: AUE-8810-001 : Auto Sys Overview : 202208-AUE-8810-001-83479-Paredis

Instructor: Chris Paredis

Submitted: 12/9/2022 11:13 PM

Student: Rohit Ravikumar

Thank you for completing a survey!

EvaluationKIT Administrator <EKAdmin@clemson.edu> Fri, Dec 9, 2022 at 11:17 PM
To: Rohit Ravikumar <rraviku@clemson.edu>

Clemson University : Survey Certificate of Completion

202208 Fall 2022 Evaluations

Course: AUE-8700-001 : Automotive Business Concepts : 202208-AUE-8700-001-88037-Mino

Instructor: Michael Mino

Submitted: 12/9/2022 11:14 PM

Student: Rohit Ravikumar

Thank you for completing a survey!

EvaluationKIT Administrator <EKAdmin@clemson.edu> Fri, Dec 9, 2022 at 11:23 PM
To: Rohit Ravikumar <rraviku@clemson.edu>

Clemson University : Survey Certificate of Completion

202208 Fall 2022 Evaluations

Course: AUE-8700-001 : Automotive Business Concepts : 202208-AUE-8700-001-88037-Brooks

Instructor: Johnell Brooks

Submitted: 12/9/2022 11:22 PM

Student: Rohit Ravikumar

Thank you for completing a survey!

EvaluationKIT Administrator <EKAdmin@clemson.edu> Fri, Dec 9, 2022 at 11:23 PM
To: Rohit Ravikumar <rraviku@clemson.edu>

Clemson University : Survey Certificate of Completion

202208 Fall 2022 Evaluations

Course: AUE-6600-001 : Dynamics of Vehicles : 202208-AUE-6600-001-89263-Schmid

Instructor: Matthias J. Schmid

Submitted: 12/9/2022 11:20 PM

Student: Rohit Ravikumar

Thank you for completing a survey!