The German University in Cairo

Mechatronics Engineering (MCTR601)

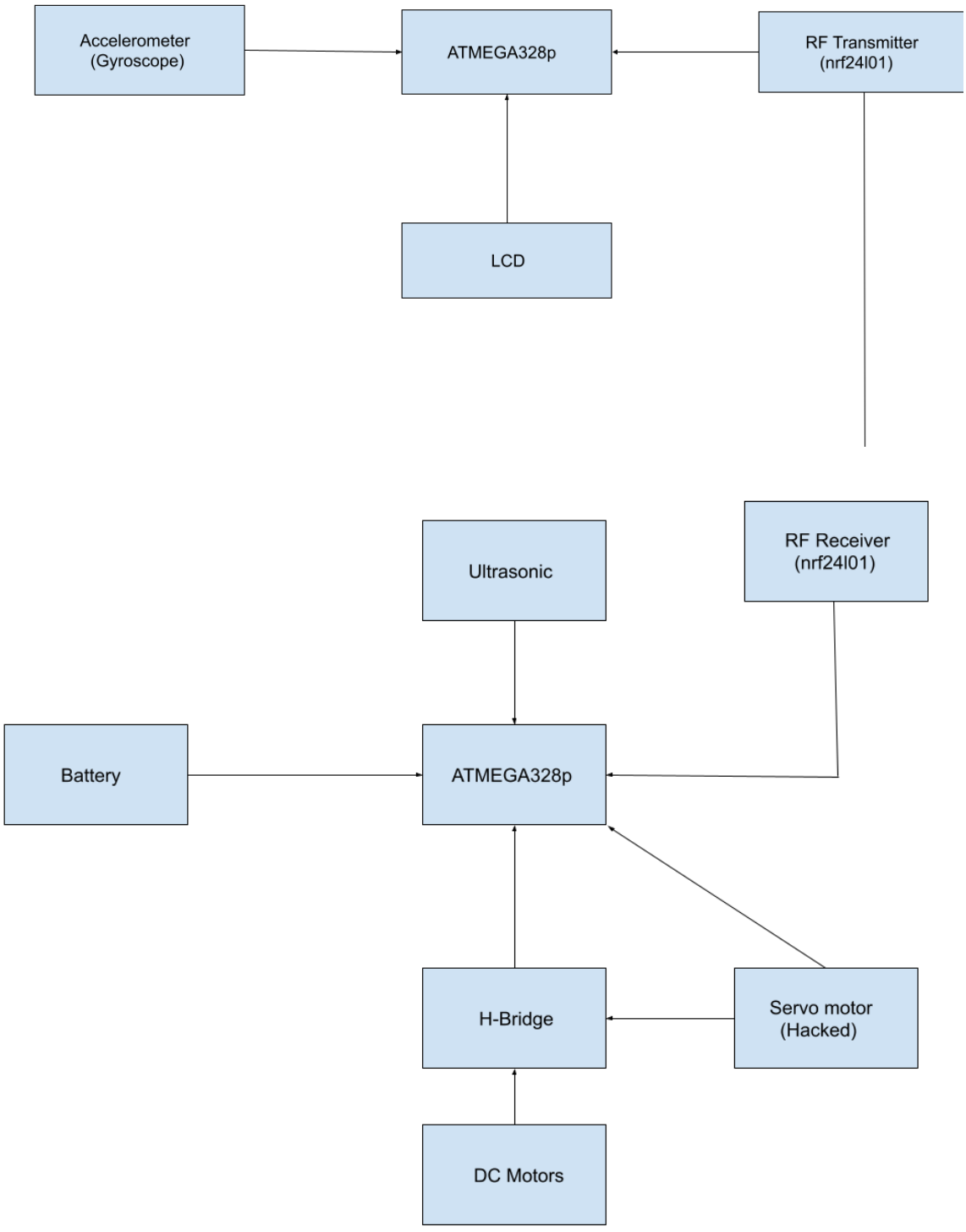
Hand Gesture-Controlled Ackerman Steering Car

Name	ID	Lab Number
Salaheldin Mohamed	49-18052	T-32
Zyad Tarek	49-16945	T-26
Shehab Ibrahim	49-7639	T-32

Table of Contents

1. Project Description

1.1. Functional diagram:



1.2. Project Idea:

Robots are playing an essential part in automation in various areas, including construction, military, medical, and industrial. The project idea(Hand Gesture car) is to convert the motion of the hand using an accelerometer (which is found in mpu6050) to read the hand motion and translate it into motion forward , backward , tire rotation right and tire rotation left . The positive pitch is translated into motion forward , the negative pitch backward , the positive roll tire rotation right and the negative roll tire rotation left . We used the RF transmitter and receiver for the data transfer from the hand rotation . Additionally , the project protects the car from crashing by using ultrasonic sensors to force a stop before collision with an obstacle and decrease the speed of the dc motors gradually if the distance between the car and the obstacle becomes within a certain range .

1.3 What the device does and how it works :

Simply the Hand gesture car gives a forward and backward motion in addition to the left and right turning with respect to the hand motion . After we used the Accelerometer found in mpu6050 and sending the data of the hand motion to the ATMEGA328p serially using I2C communication protocol (the microcontroller is the master and mpu6050 is the Slave) , we used a transmitter (nrf24l01) to transmit the data from the first Microcontroller used . A receiver (nrf24l01) then is used to receive this data from the transmitter . After receiving this data , we use them to control the speed of the DC Motors and the rotation of the hacked Servo Motor.

The 2 DC Motors for forward motion share the same PWM. We controlled the DC motor with the potentiometer (Hacked Servo motor) and calculated its error then used the feedback control to rotate the tires.

We also used the ultrasonic sensor so as to control the speed of the DC motors whenever the distance was within a certain limit (30 cm) and forced a stop to the car if the distance became less than 5 cm, otherwise we used the maximum speed of the DC motors.

1.4. The main actuators and sensors of the system :

For the main actuators and sensors in the system , we used the Accelerometer of the mpu6050 which is responsible for the car's motion and

rotation to the left and the right and the ultrasonic sensor to control the collision with obstacles while moving forward and force a stop when the distance between the car and the obstacle becomes less than 5 cm.

1.5. Actuators used in the system:

- DC Motors

1.6. Sensors used in the system:

- Accelerometer
- Ultrasonic sensor

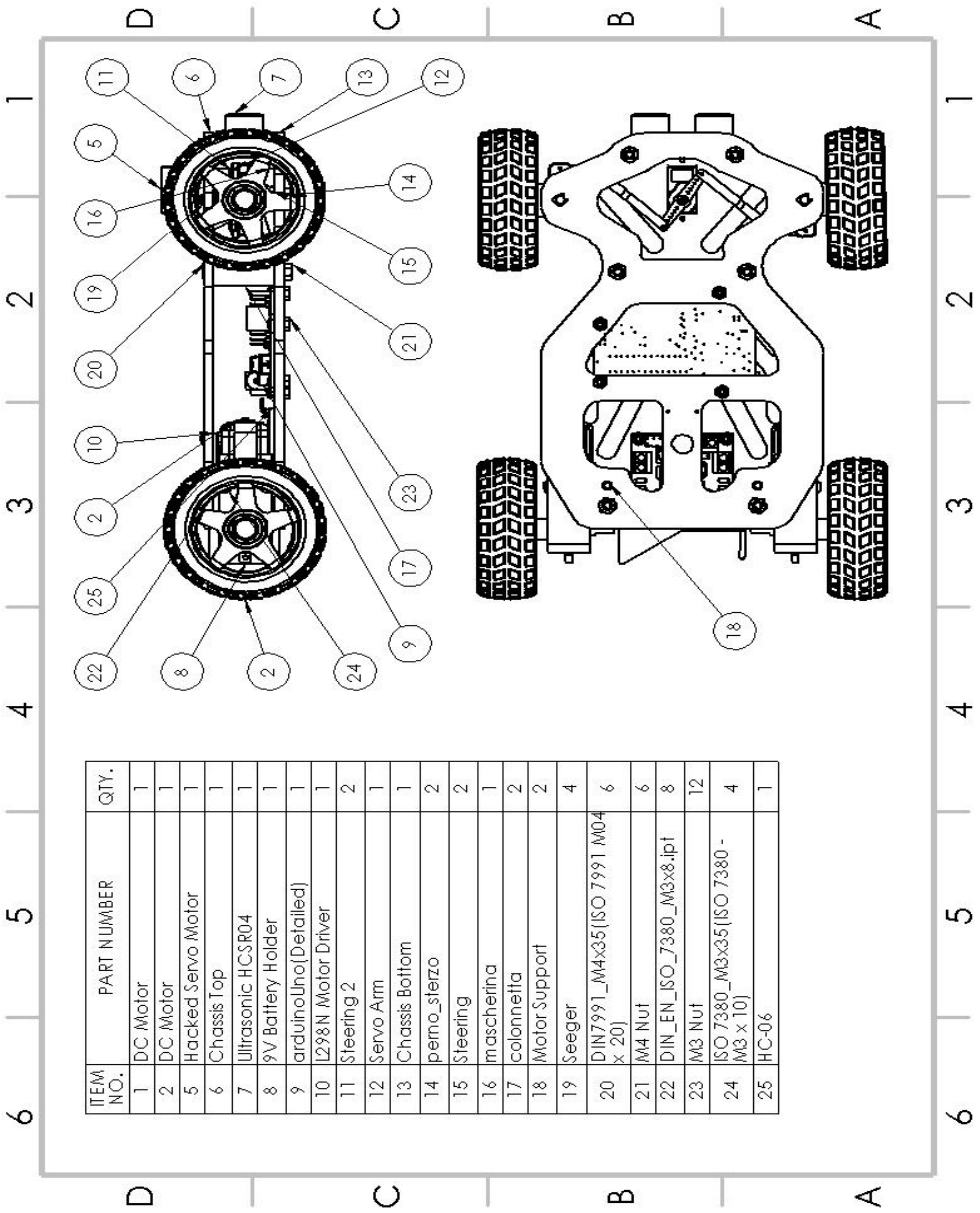
1.7. Components Used:

Name	Quantity
Arduino Uno.	2
DC Motors.	3
Accelerometer MPU 6050 GY-521.	1
Motor Driver L298N.	1
Ultrasonic Sensor HC-SR04.	1
Single chip 2.4 GHz Transceiver nRF24L01.	2
LCD(16*2).	1
Power Supply.	1

2. Methodology

2.1. Mechanical design

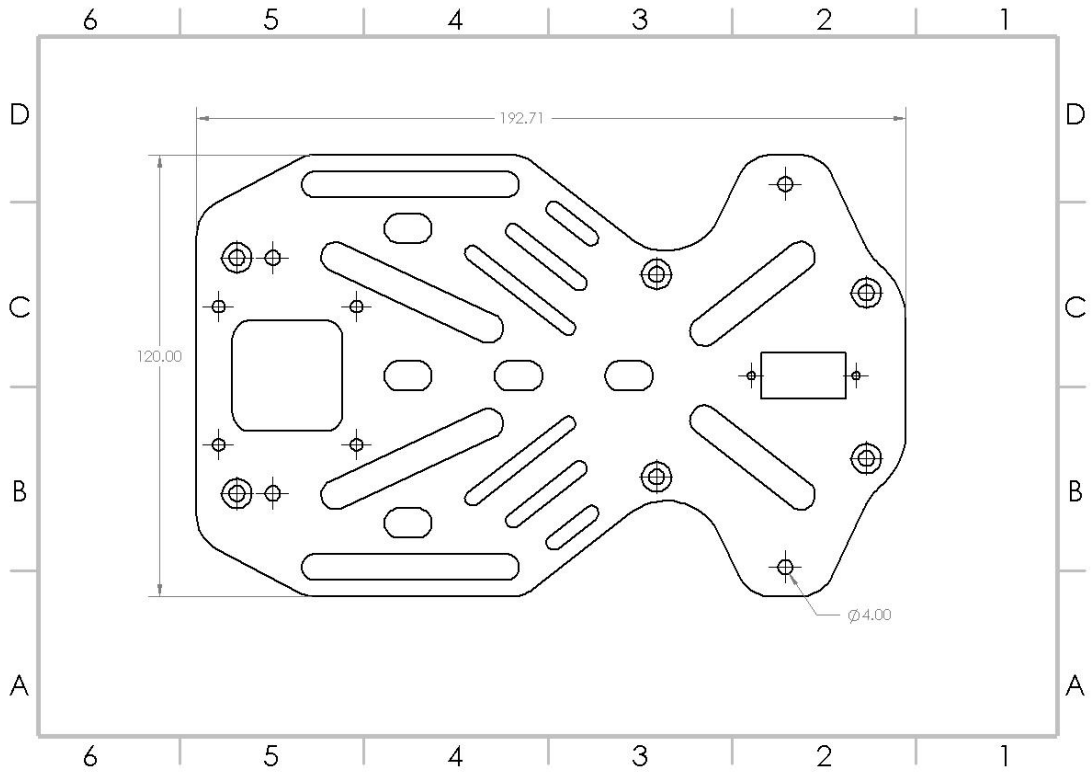
2.1.1. Parts:



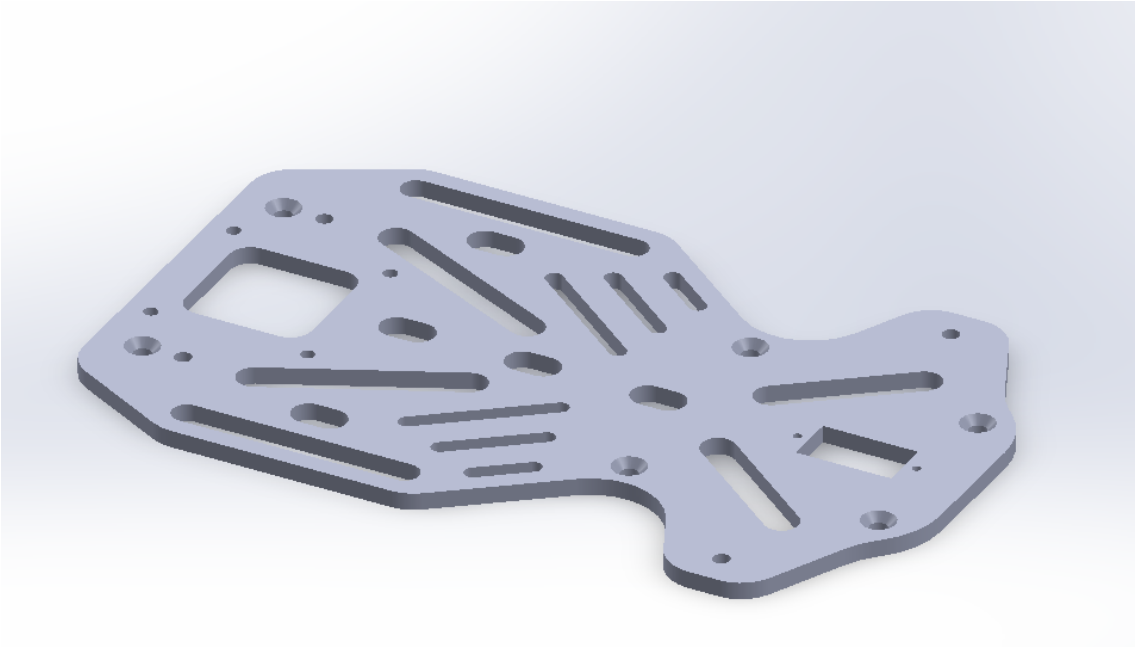
ITEM NO.	PART NUMBER	QTY.
1	DC Motor	1
2	DC Motor	1
5	Hacked Servo Motor	1
6	Chassis Top	1
7	Ultrasonic HCSR04	1
8	9V Battery Holder	1
9	arduinoUno(Detailed)	1
10	L298N Motor Driver	1
11	Steering 2	2
12	Servo Arm	1
13	Chassis Bottom	1
14	perno_sterzo	2
15	Steering	2
16	mascherina	1
17	colonnetta	2
18	Motor Support	2
19	Seeger	4
20	DIN7991_M4x35(ISO 7991 M04 x 20)	6
21	M4 Nut	6
22	DIN_EN_ISO_7380_M3x8.ipt	8
23	M3 Nut	12
24	ISO 7380_M3x35(ISO 7380 - M3 x 10)	4
25	HC-06	1

Following is a detailed description for some of the Mechanical parts in our project:

Dimensions of the body of the car is defined as follows:

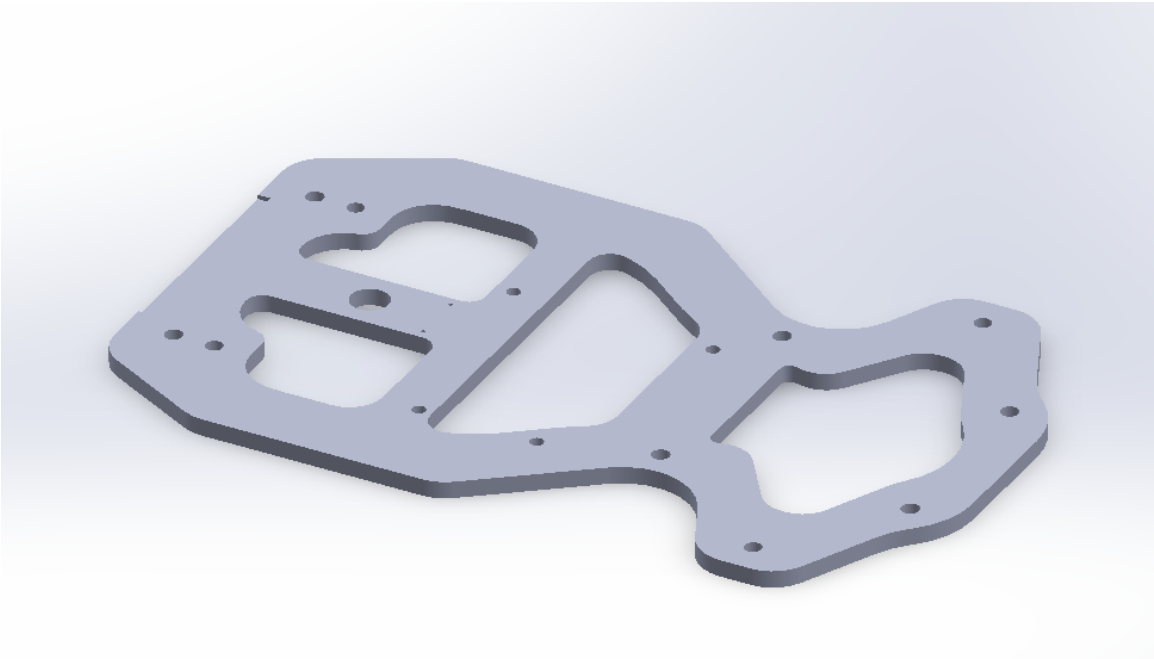


2.1.2. Upper Chassis:



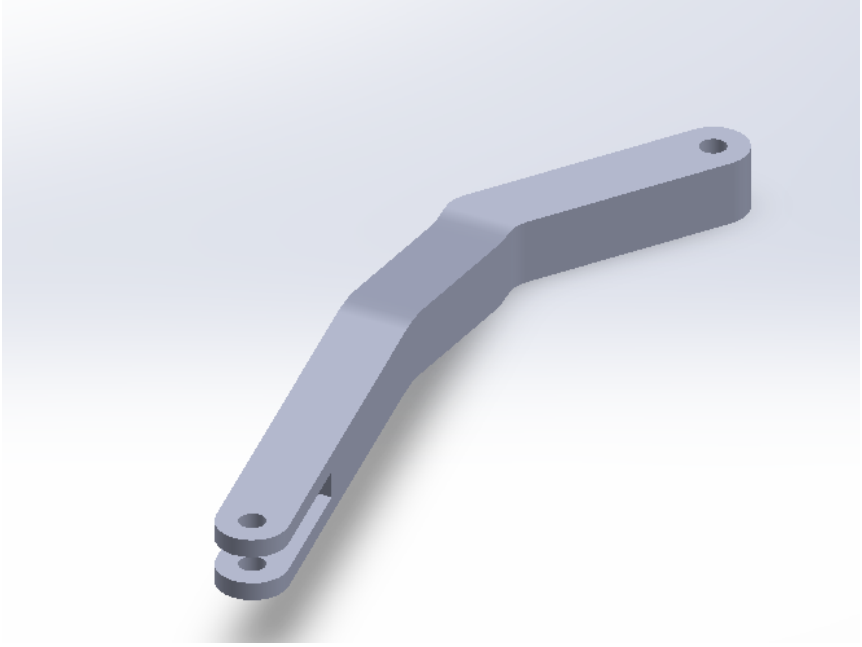
This is the Upper Part of The Car, it has many holes and slots for wires and connections, spaces for the front wheels to rotate.

2.1.3. Lower Chassis:



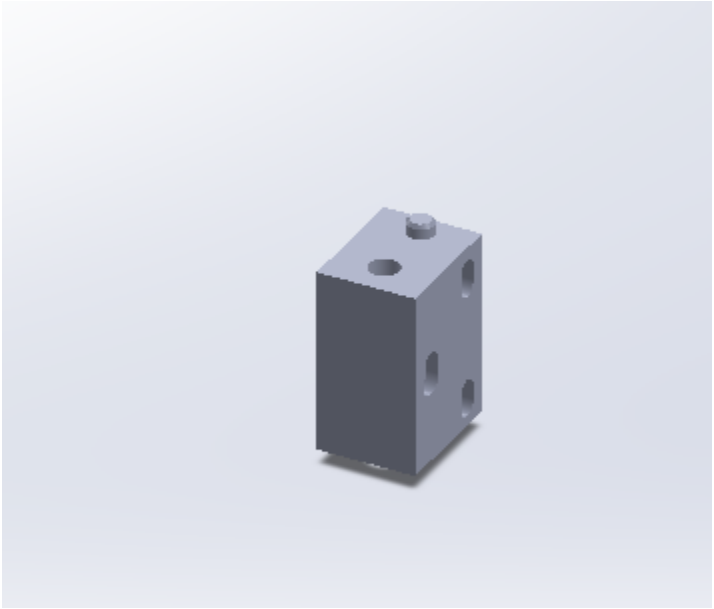
The same as the upper part.

2.1.4. Front Wheels' steering connector:



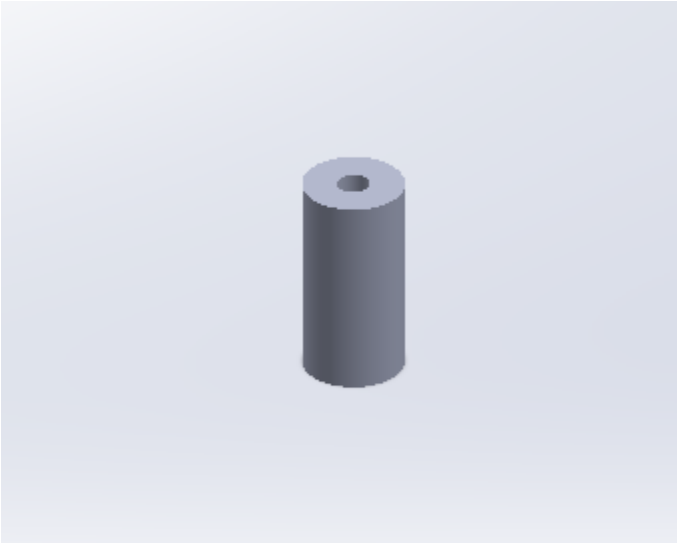
Connected to the front wheels of the car to connect them to the Servo-Motor and help then rotate.

2.1.5. Motor Supports:



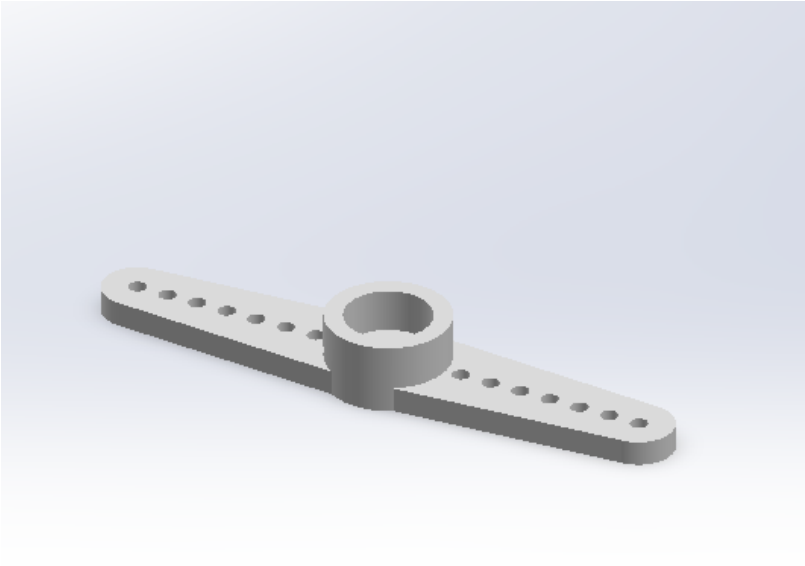
Supports the motor.

2.1.6. Body Supports:



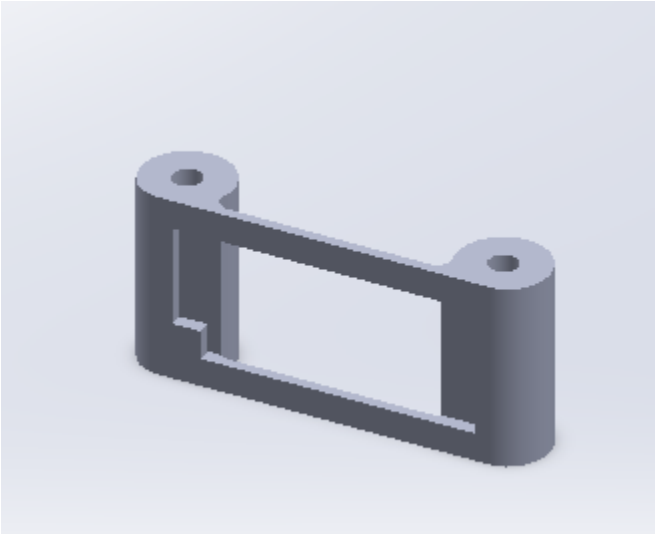
Supports each part of the kit while connecting the upper and lower chassis together with bolts through them.

2.1.7. Servo End:



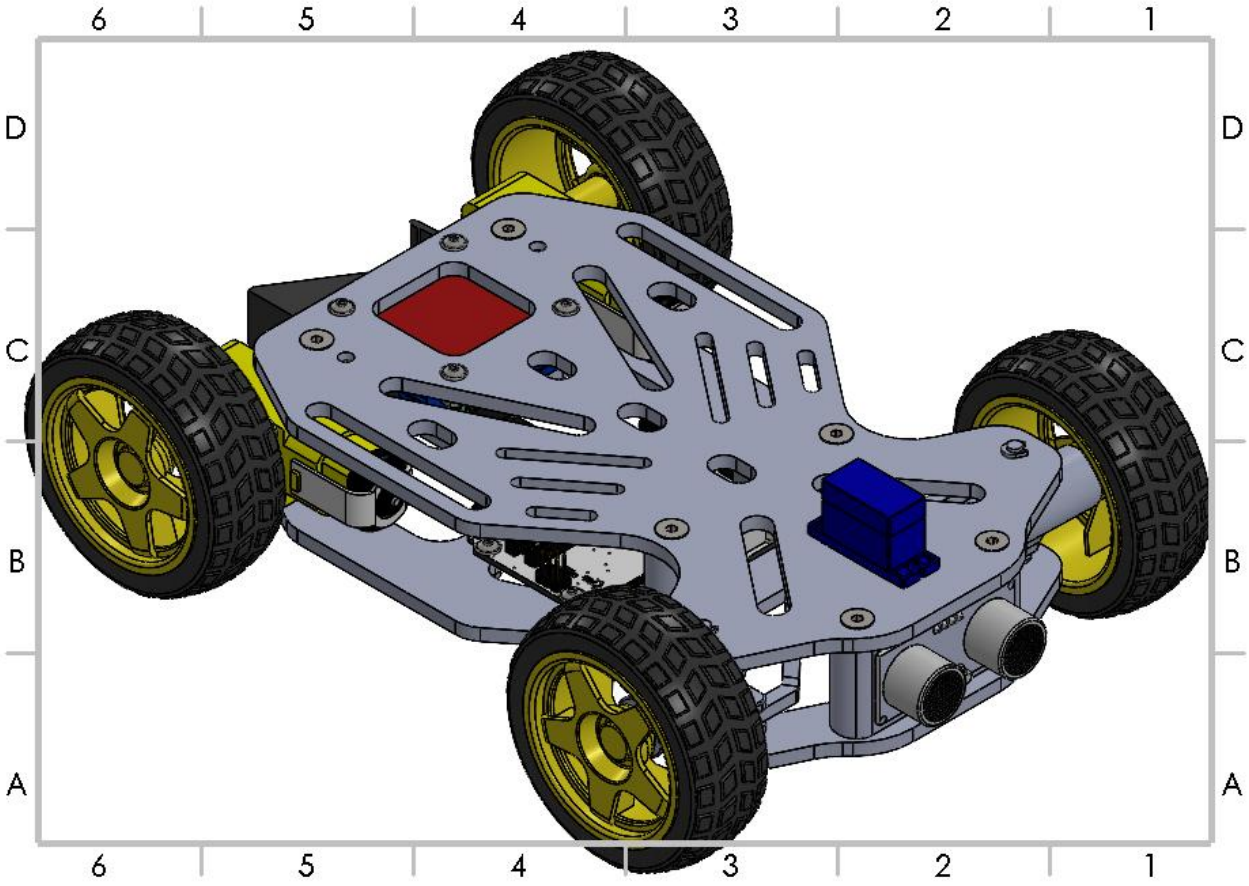
Connected to the end of the Servo-Motor and also connected to the front wheels' shafts to transmit the motion of the servo to the front wheels.

2.1.8. Ultra-Sonic Holder:



Hold, support and keep the Ultrasonic Sensor Fixed during motion.

Final Assembly:



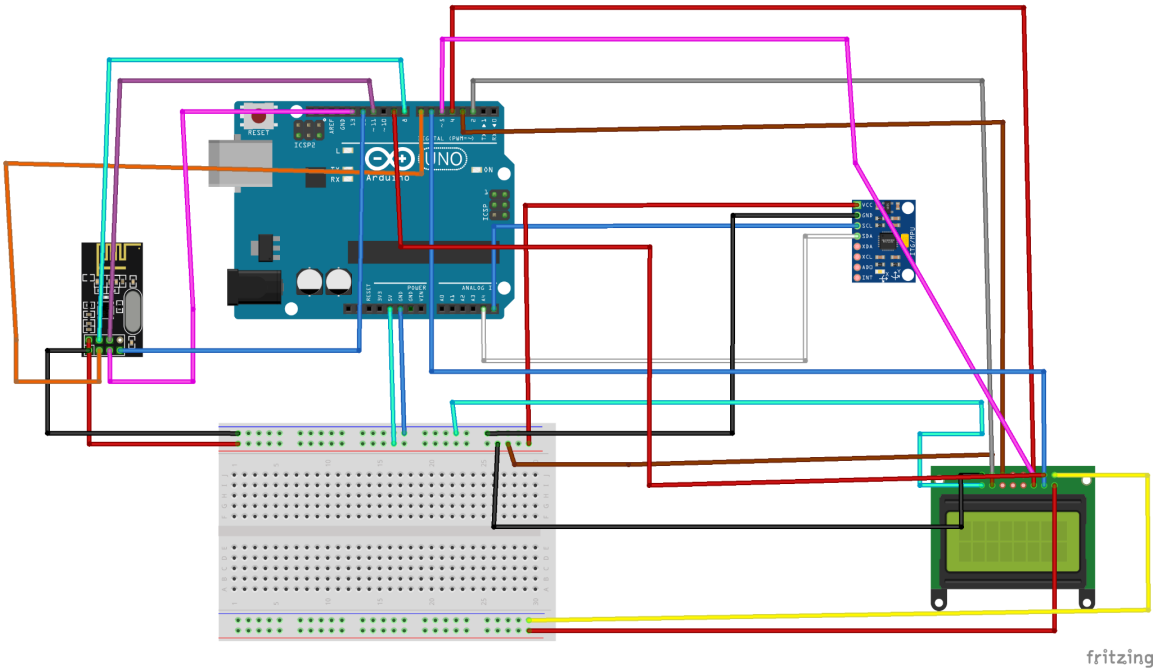
The car design was designed by Eng. Matteo Parmeggiani, we took the design, edited it to our needs and then moved on with the manufacturing process.

The chassis we used to make the car was laser cut and the material used was white transparent Acrylic with the thickness of 3mm.

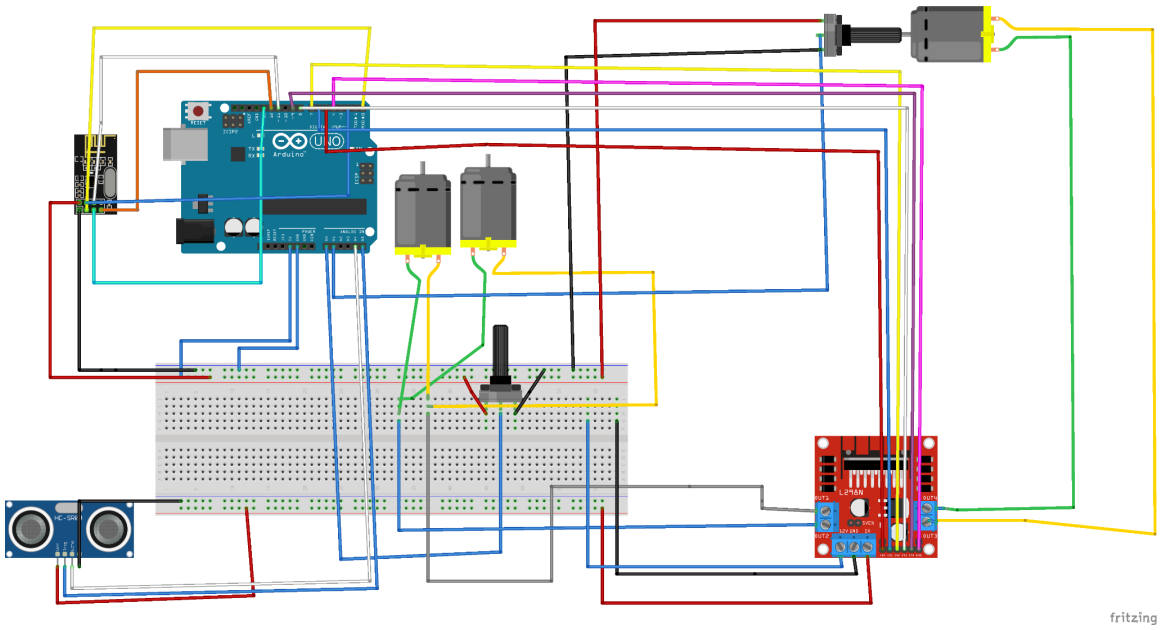
All supporting parts were 3d printed with white filament to fit with the car's body color, each part was printed separately and assembled together according to the designs.

In the appendix you can find more info about the journey of laser cutting and 3d printing the body of the car.

3. Electrical design



The circuit diagram and pin connection of the transmitter part of the project.



The circuit diagram and pin connection of the receiver part of the project.

4. Control

4.1. Modeling

4.1.1. Voltage-Angle: (For Servo-Motor)

A DC motor is an actuator that converts electrical energy to mechanical rotation using the principles of electromagnetism.

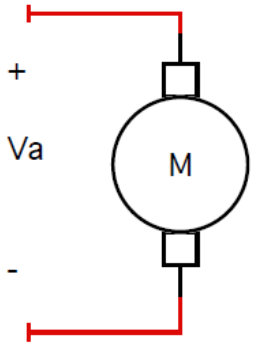


Figure 1: Circuit symbol for a DC servomotor

The circuit shown in Figure 2 models the DC servomotor. Note that an armature control current is created when the armature control voltage **"Va"** energizes the motor. The current flows through a series-connected armature resistance **"Ra"**, an armature inductance **"La"**, and the rotational component (the rotor) of the motor. The rotor shaft is typically drawn to the right with the torque **"Tm"** and angular displacement **"θm"** variables shown. **The motor transfer function is the ratio of angular displacement to armature voltage.**

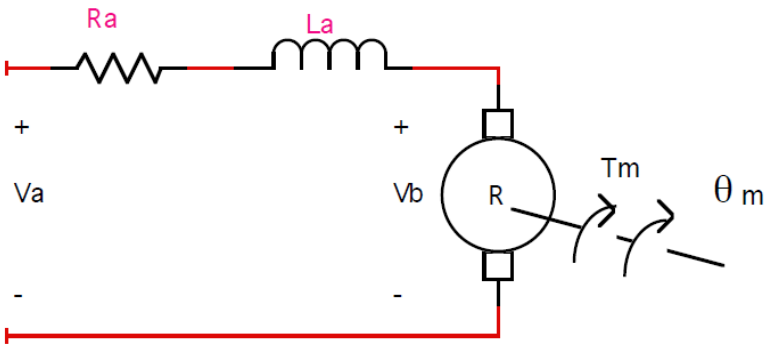


Figure 2: DC servomotor circuit theory model

Figure 3: The DC servomotor transfer function

Figure 3 shows the final Transfer functions which should be obtained for the Voltage-Angle relation:

Three equations of motion are fundamental to the derivation of the transfer function. Relationships between torque and current, voltage and angular displacement, and torque and system inertias are used.

Torque is proportional to the armature current. The constant of proportionality is called the torque constant and is given the symbol **"Kt"**. The time and frequency domain relationships are given as equations 1a and 1b.

(1a) $T_m(t) = K_t * I_a(t)$

(1b) $T_m(s) = K_t * I_a(s)$

The back electromotive force (back-emf) **"Vb"** is a result of the rotor spinning at right angles in a magnetic field. It is proportional to the shaft velocity. The constant of proportionality is called the back-emf constant and is given the symbol **"Kb"**. The time and frequency domain relationships are given as equations 2a and 2b.

(2a) $V_b(t) = K_b * \frac{d\theta_m}{dt}$

(2b) $V_b(s) = K_b * s * \theta_m(s)$

Torque can also be written as a relationship that depends on the load attached to the motor shaft. If the load inertia and damping behaviors are reflected back to the shaft the resulting equivalent inertia coefficient **"Je"** and viscous damping coefficient **"De"** can be used to model the mechanical system attached to the shaft. These equivalents combine the motor and load properties in a model with a single degree of freedom.

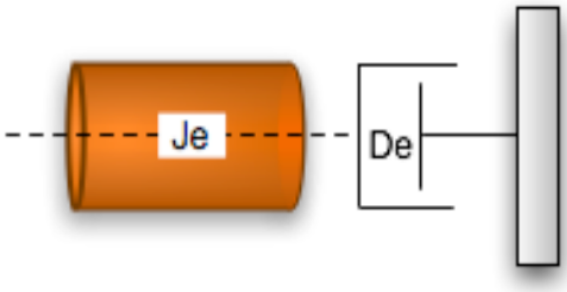


Figure 4: The mechanical system attached to the motor shaft

Figure 4 diagrams the equivalent system. The torque equation can be written by inspection by summing the forces acting at the shaft.

(3) $T_m = (Je * s^2 + De * s) * \theta m$

$$G_\theta = \frac{1}{Je * s^2 + De * s}$$

The DC servomotor transfer function block diagram can be created through an examination of the armature current equation. Ohm's law for impedances is used to write the armature current equation in the s-domain. Note that the equation includes **both real and imaginary** parts of the impedance. **The reactance is inductive.**

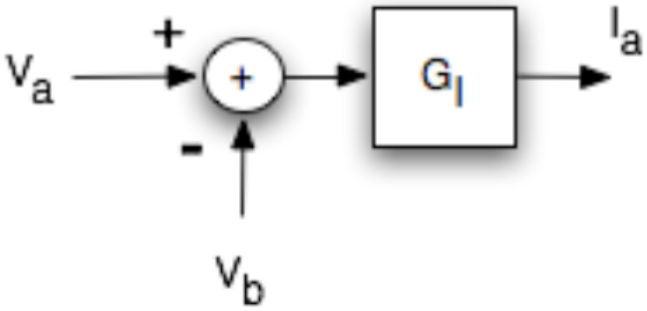


Figure 5: The block diagram after deriving armature current

(4) $I_a = \frac{V_a - V_b}{R_a + L_a * s} = G_I (V_a - V_b)$

Then $G_I = \frac{1}{L_a * s + R_a}$

Similarly, the equations of motion can be used to relate the armature current to torque, torque to angular displacement, and angular displacement to the back-emf.

The Final Block is:

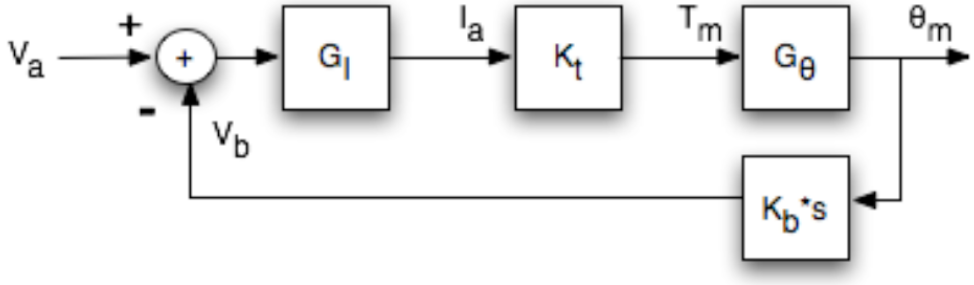
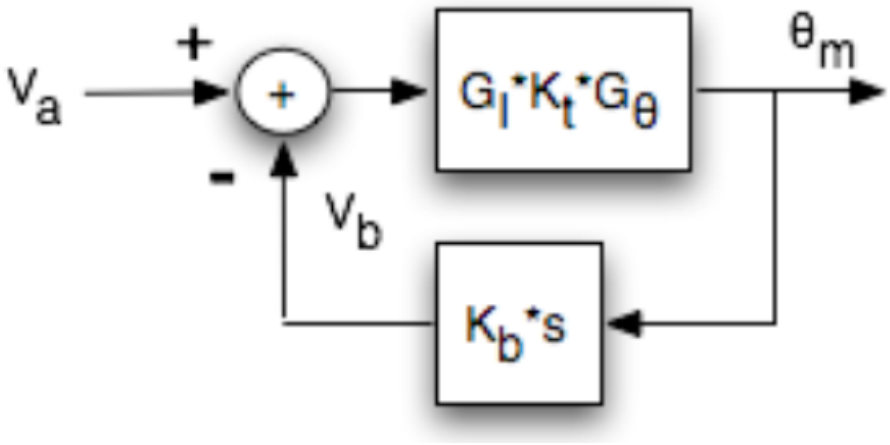


Figure 6: The complete block diagram of the DC servomotor

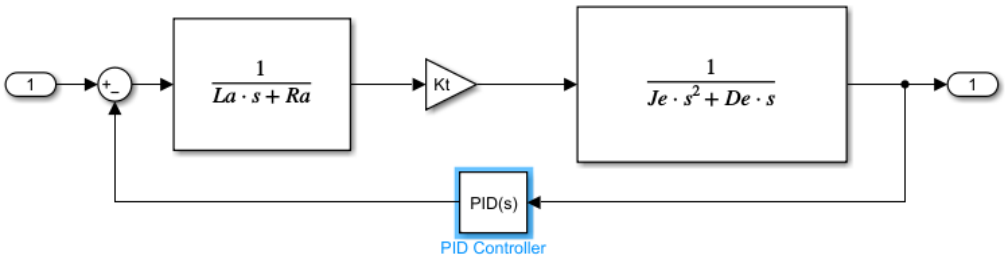
Note that this actuator is a feedback system that can be reduced to the standard form for derivation of the gain. The series gain path is reduced to an equivalent gain through multiplication. The reduced block diagram is shown in Figure 7. The forward path gain is traditionally called **G** while the feedback path gain is **H**.



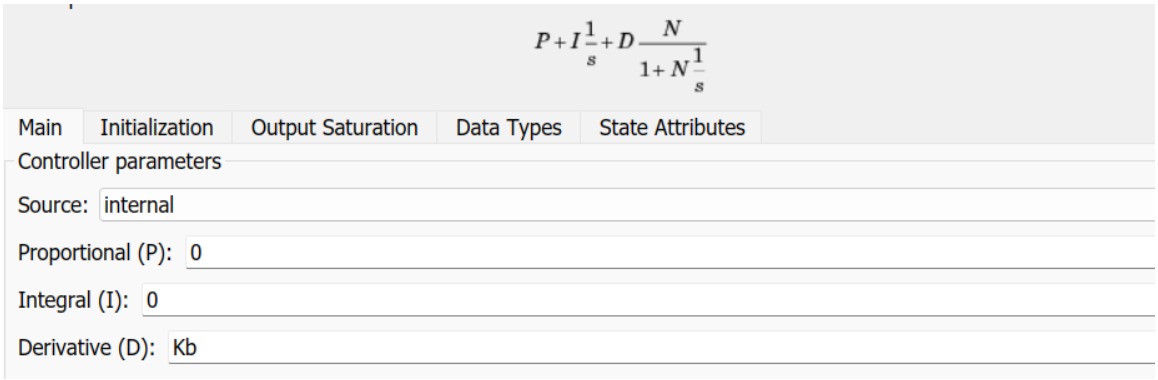
The gain can be calculated directly from the reduced block diagram:

$$G_m = \frac{\theta_m}{V_a} = \frac{G}{1+G*H} = \frac{G_l*K_t*G_\theta}{1+(G_l*K_t*G_\theta)*(K_b*s)}$$

Final Block is shown in the next Figure:



Note: PID here is used to obtain a (Kb*s) as a gain which is not in Simulink library



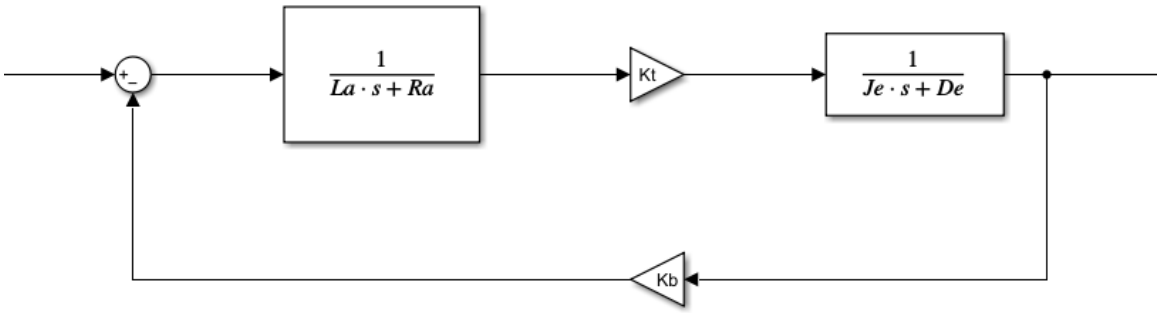
4.1.2. Voltage-Speed Control:

Using the same parameters as in the Angle equations and the same circuits, we can obtain the speed transfer function as the speed is the angle derivative.

For the Electrical part "Gi" will still be the same without any changes while the changes will be in the inertia part with

$$G_s = \frac{1}{Je*s+De}$$

The Transfer Function for the Speed will be as follows:



$$G = \frac{G_I * K_t * G_s}{1 + G_I * K_t * G_s * K_b}$$

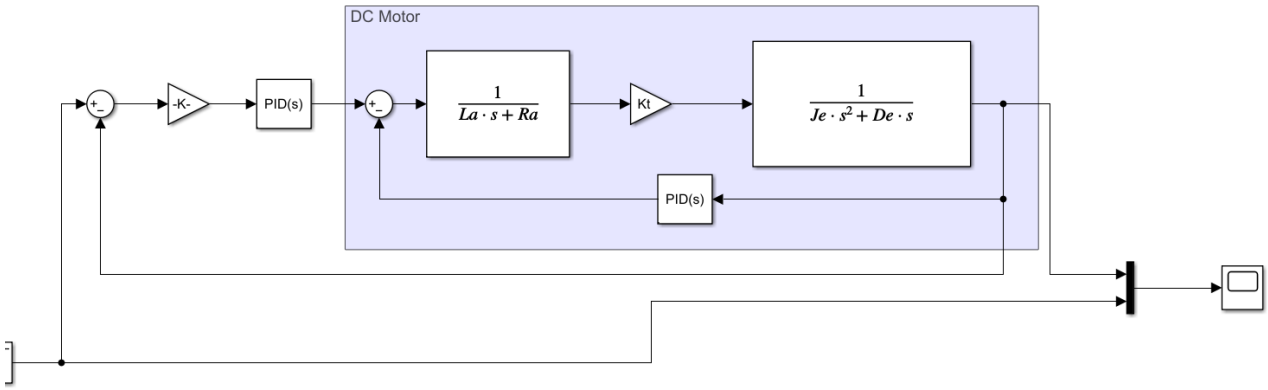
Here is a table with the Parameters used:

Parameter	Value	Name
Ra	**2.6 Ω**	**Motor Armature Resistance**
La	**180 µH**	**Motor ArmatureInductance**
Kt	**7.67E-3 N*m/A**	**Motor Torque Constant**
Kb	**7.67E-3 V/(r/s)**	**Back Emf Constant**
Je	**5.3E-7 kg*m2**	**Moment of inertia**
De	**7.7E-6 N*m/(r/s)**	**Viscous Damping coefficient**

4.1.3. Analysis

4.1.4. Analysis of Voltage-Angle:

Here is the Closed loop system with unity feed-back and without the usage of any controller:

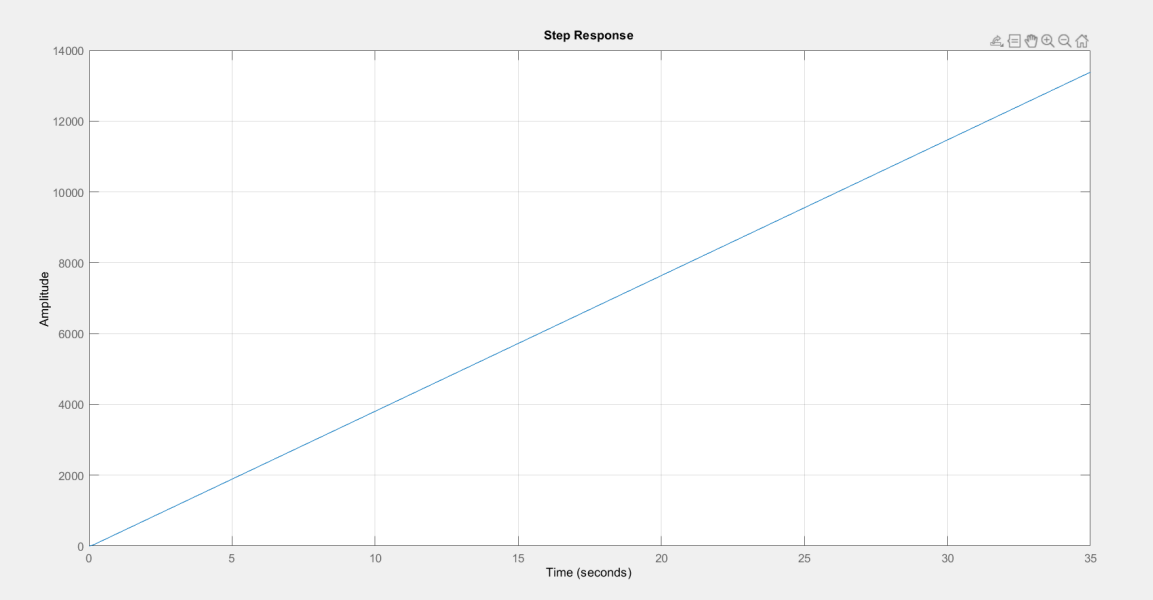


where K = 5/90 => a gain to convert angle into voltage

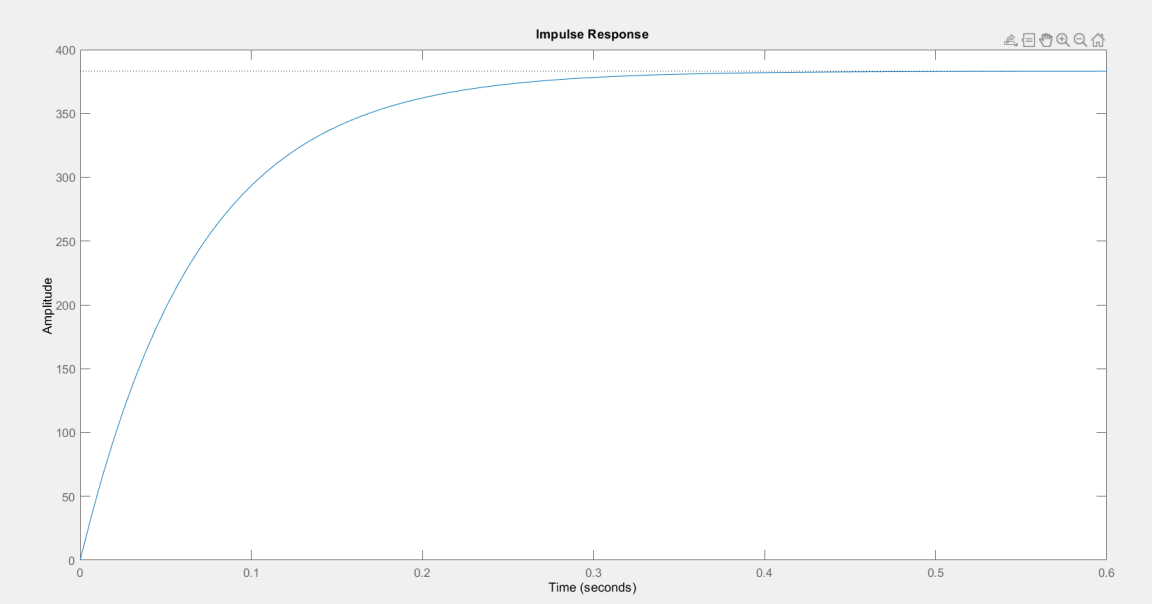
For open loop equation: $\frac{K_t}{(L_a*s+R_a)*(J_e*s^2+D_e*s)}$

Now Let's see the system Response for Different input signals:

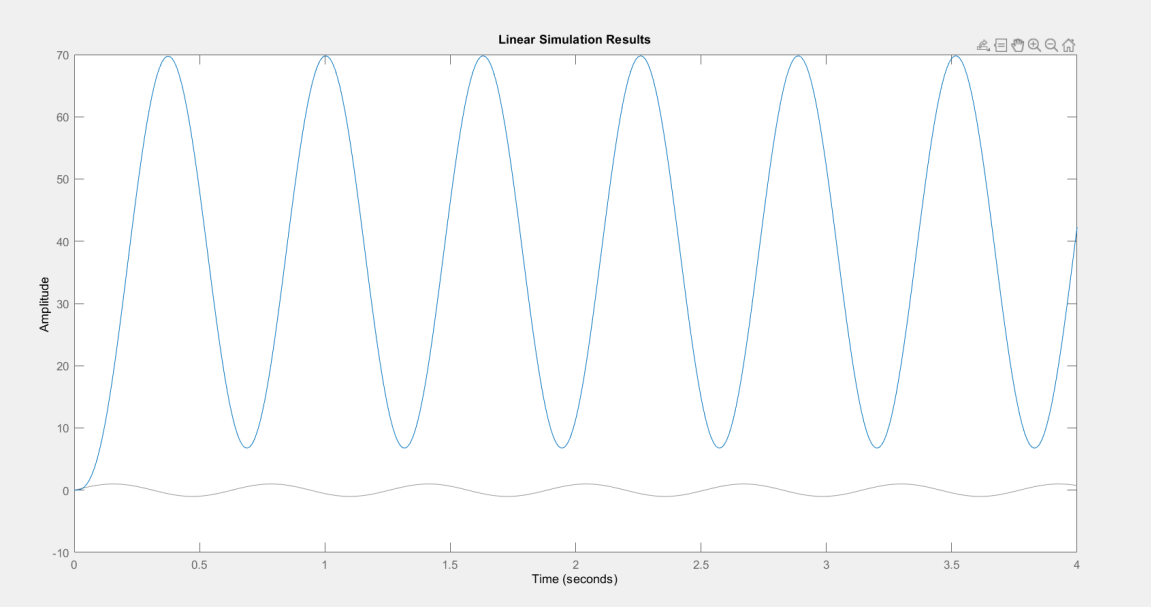
- **4.1.4.1. Step:**



- **4.1.4.2. Impulse:**

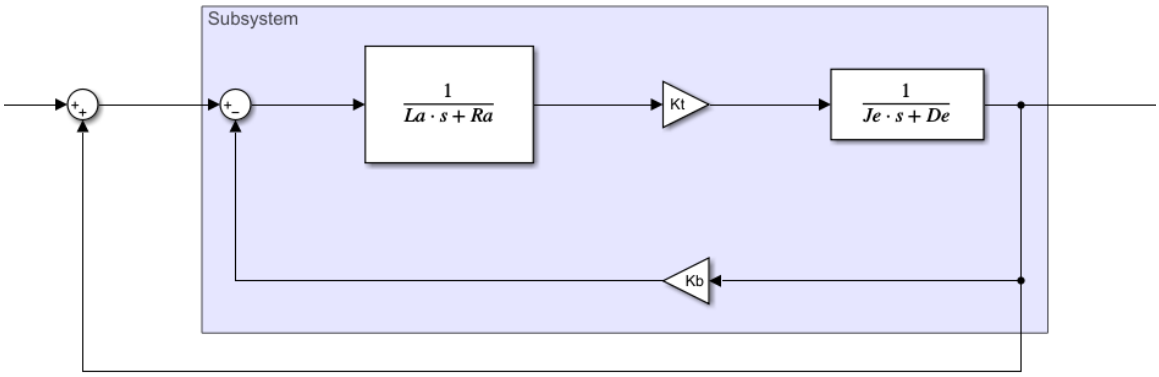


- **4.1.4.3. Sin input (Sin10t):**



4.1.5. Voltage-Speed:

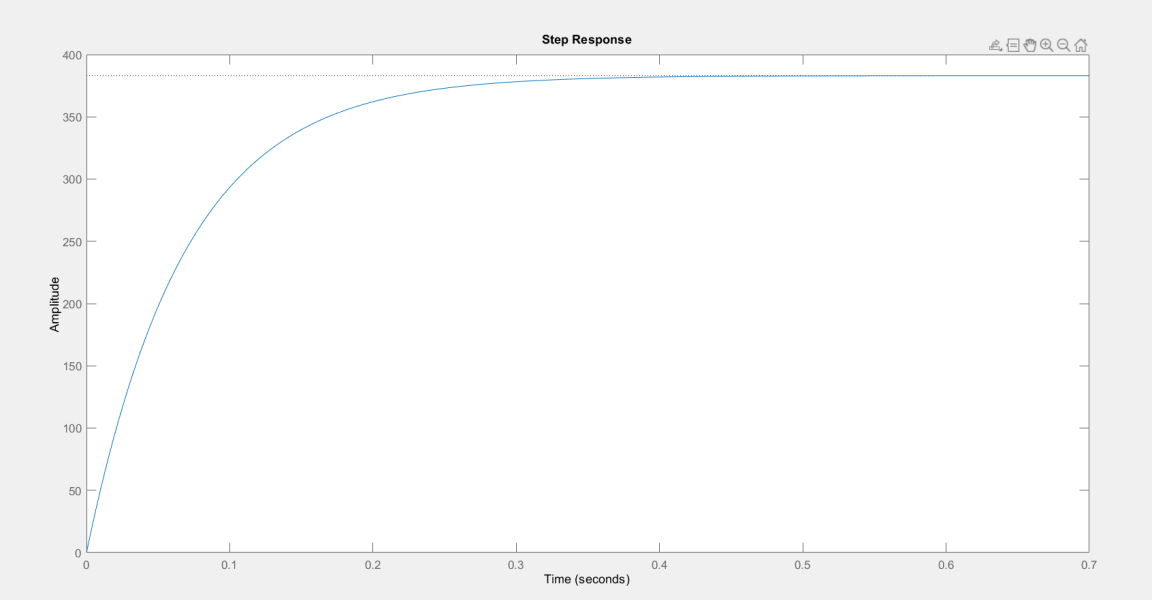
Here is the Closed loop system with unity feed-back and without the usage of any controller:



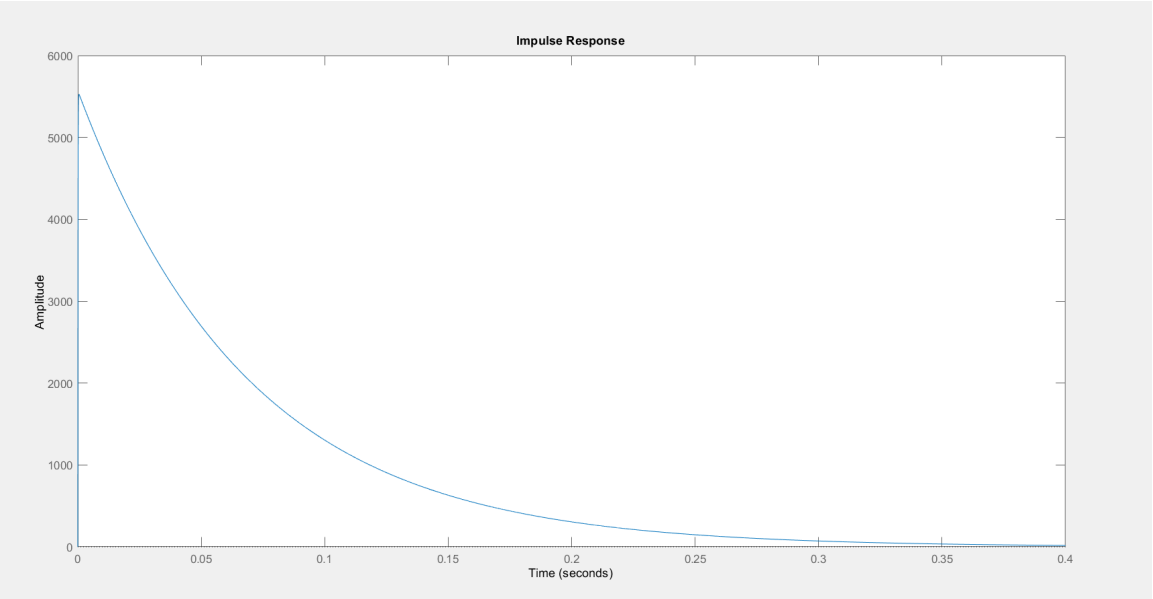
For open loop equation: $\frac{1}{(L_a{*}s{+}R_a){*}(J_e{*}s{+}D_e)}$

Now Let's see the system Response for Different input signals:

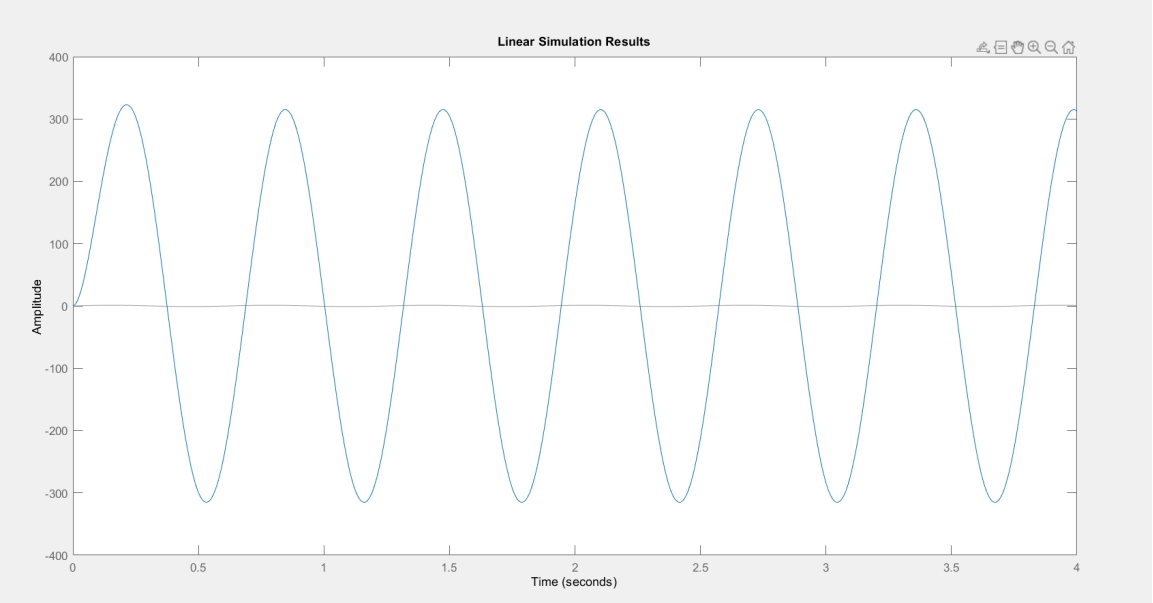
- **4.1.5.1. Step:**



- **4.1.5.2. Impulse:**

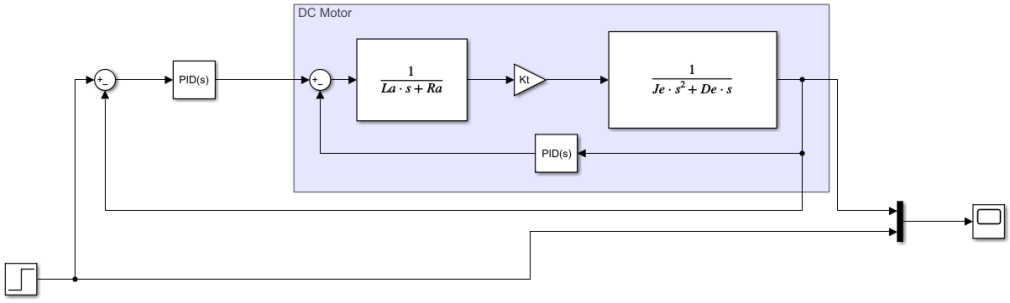


- **4.1.5.3. Sin Input(Sin10t) :**



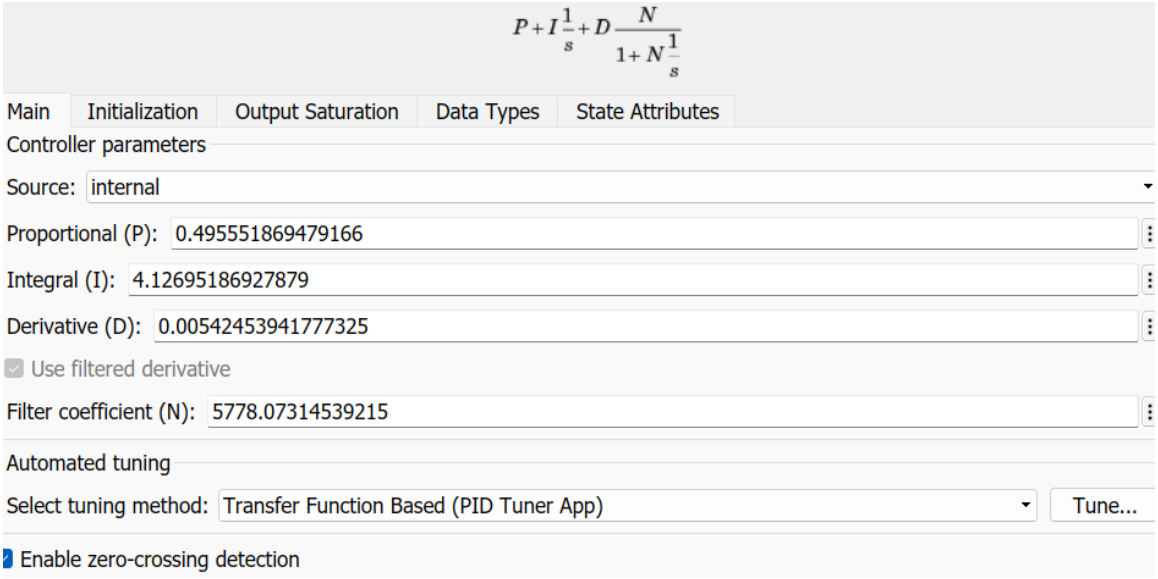
4.1.6. Controller Design:

Now it's clear that our system needs a controller to be more stable and to give the desired output.
As the Angle is the Derivative of the Speed , so it will be the same control for both.
Note: the Control was made with the aid of Simulink Auto-Tuning.

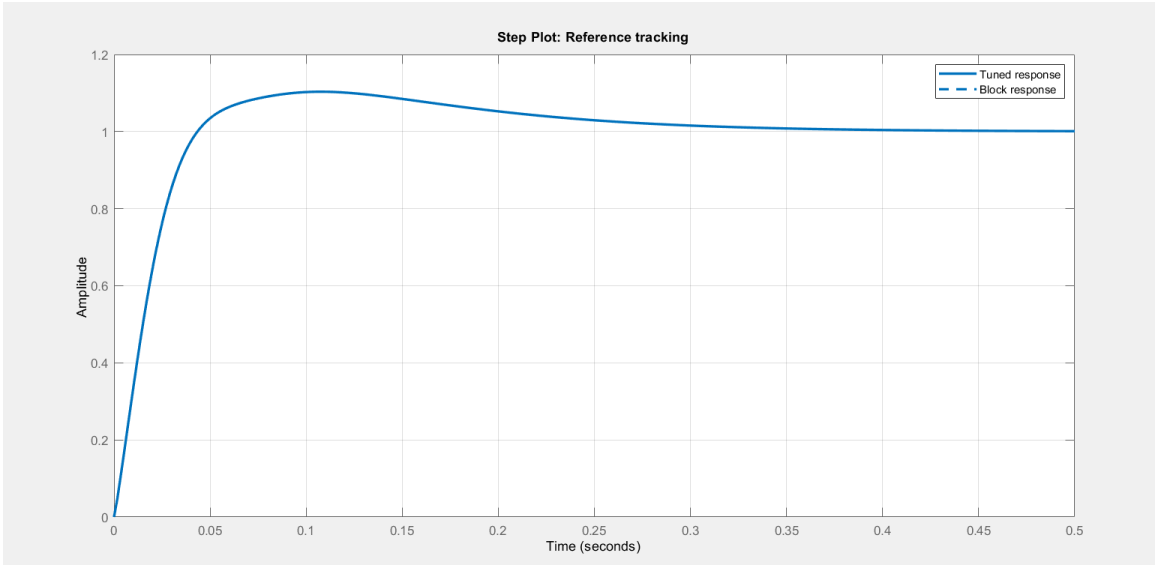


As shown in the Figure, the PID Controller is now connected and the control will be done for a step input and the output will be compared to the input using a scope.

- Tuning with Slower time response and a more Robust Transient Behavior: Values used:



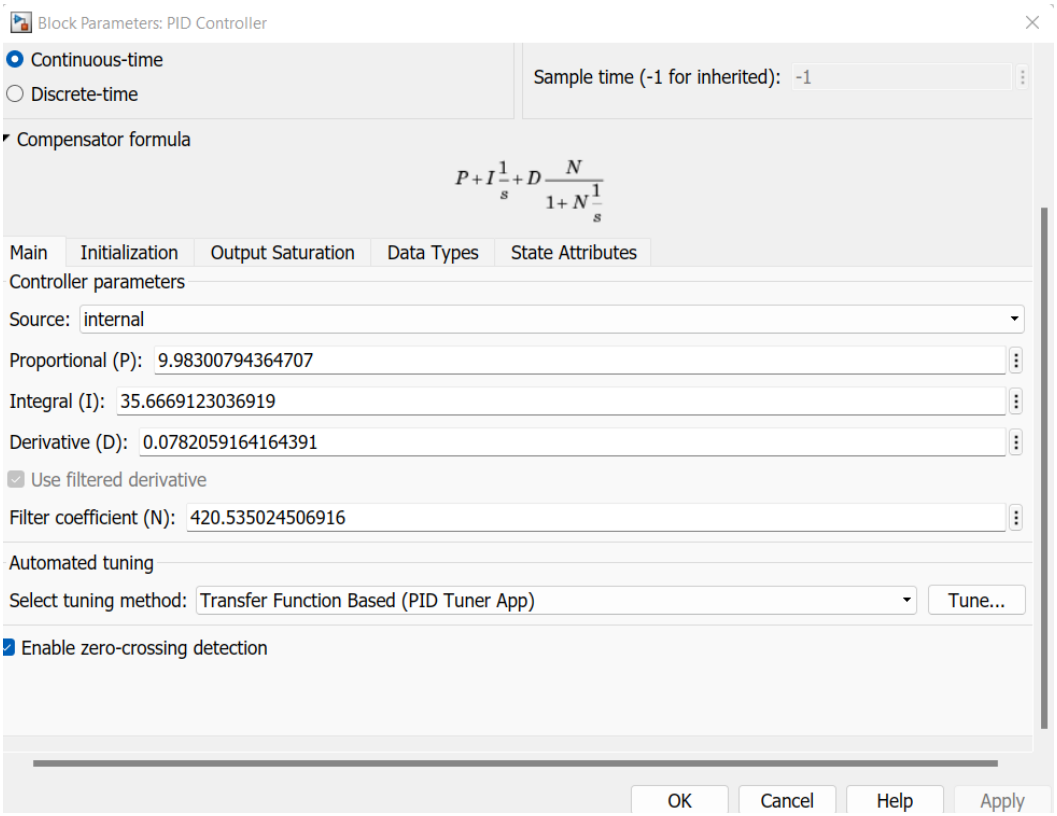
Response:



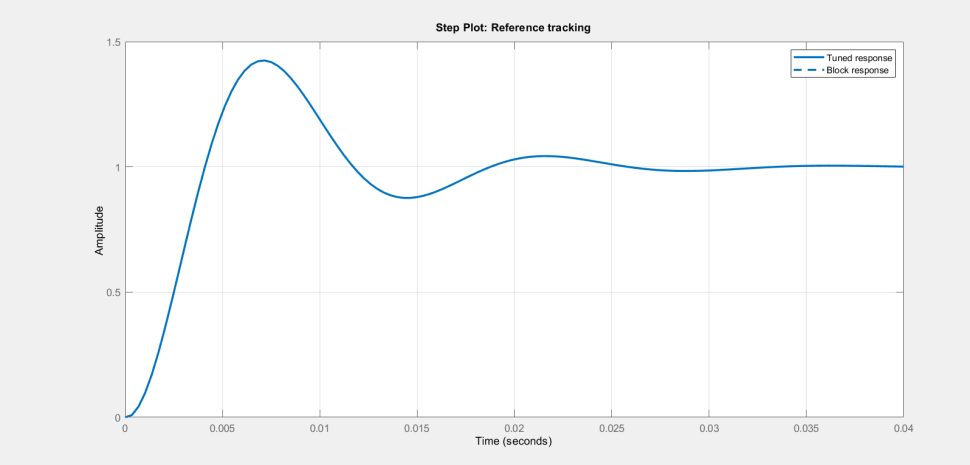
Comment: the system shows a more stable response but with an 0.1 over shoot and some delay in response.

- Tuning with Faster time response and a more aggressive Transient Behavior:

Values used:



Response:



Comment: the system shows a faster response but with an 0.5 overshoot and some oscillations.

- Tuning with the best time response and the best Transient Behavior:

Values used:

$$P + I\frac{1}{s} + D\frac{N}{1 + N\frac{1}{s}}$$

Main | Initialization | Output Saturation | Data Types | State Attributes

Controller parameters

Source: internal

Proportional (P): 8.3550418676458

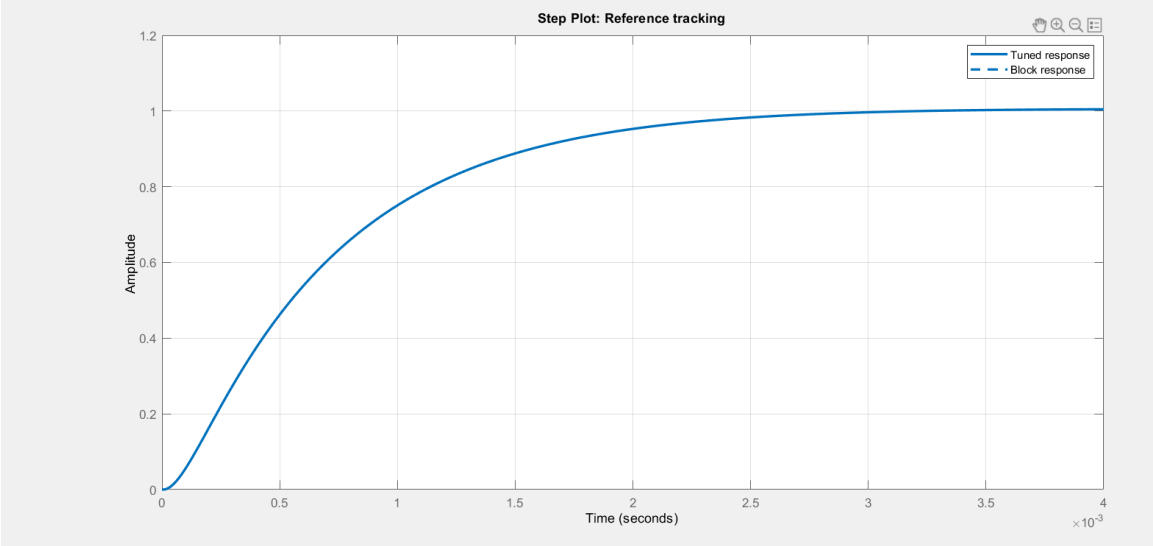
Integral (I): 64.8204339249882

Derivative (D): 0.239279943975786

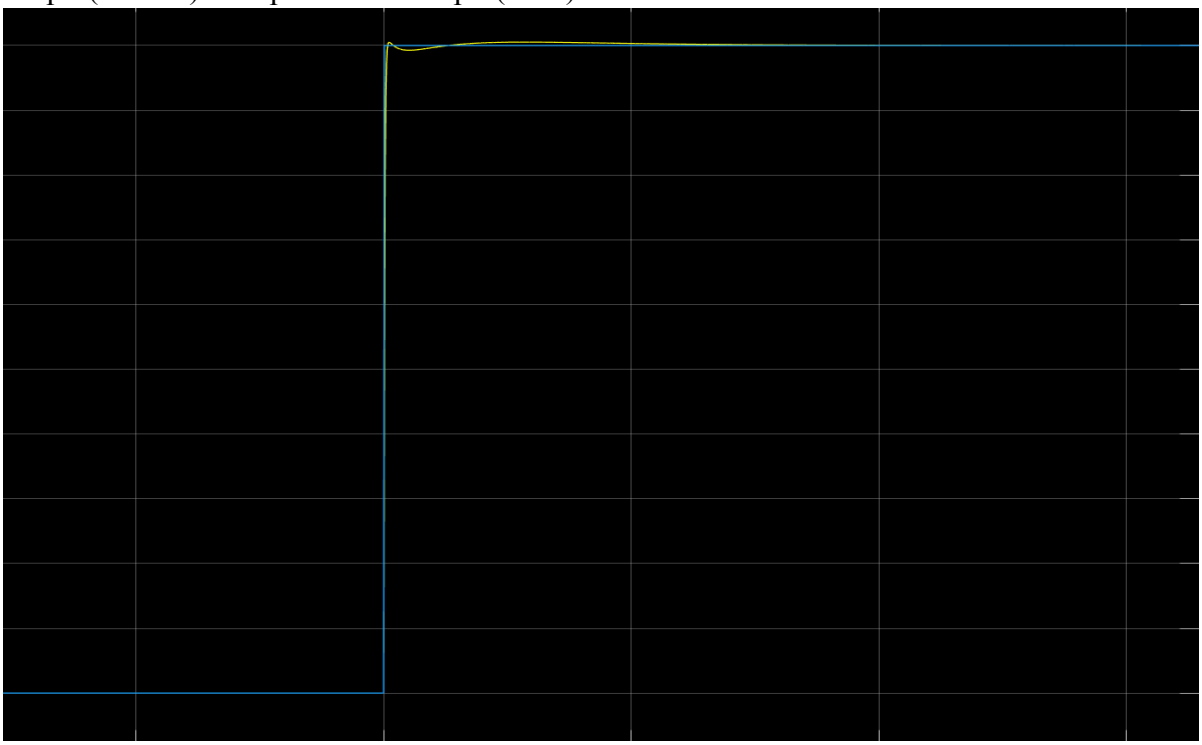
Use filtered derivative

Filter coefficient (N): 151978.808491552

Response:



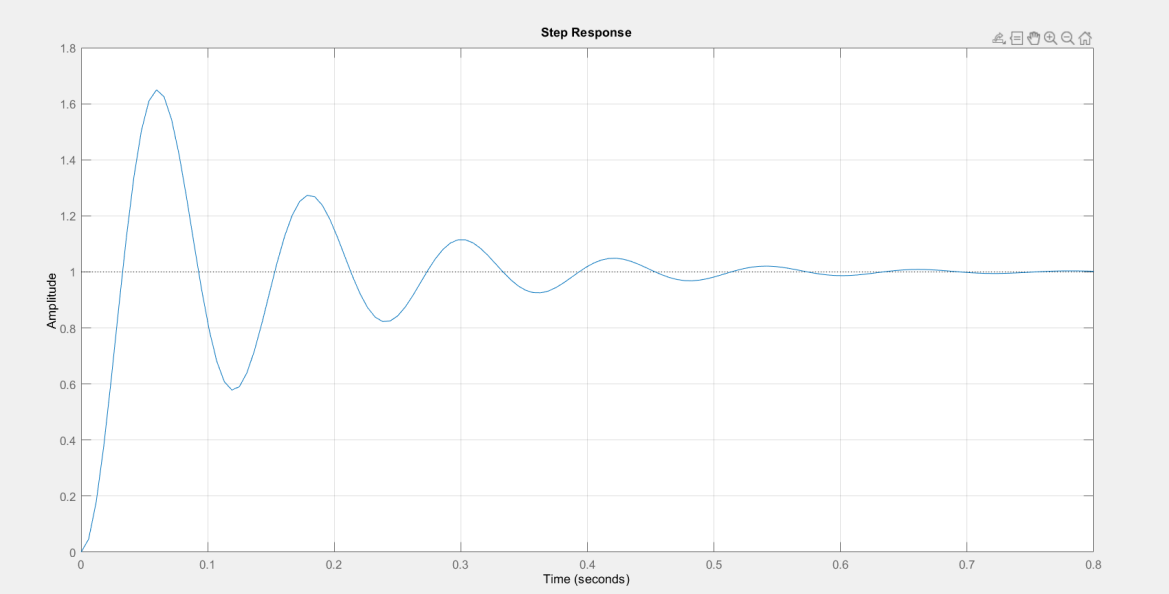
Output(Yellow) Compared to the input(Blue):



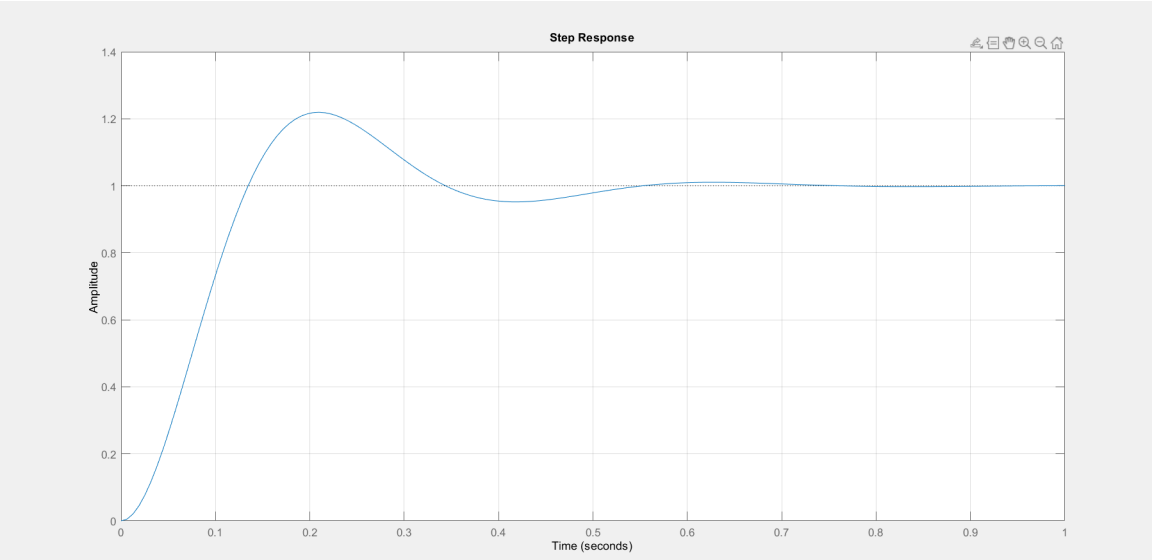
Note: All the previous simulations were theoretical analysis which won't be the same in the practical analysis , because the parameters used in the analysis were assumed and don't have to be the same as the real ones for our motor, also we ignored the effect of the load on the motor (wheels , micro-controller, H-bridge and any other load on the car).

So we had to do our practical analysis using trial and error method , using only a P controller and changing the Kp value obtaining the step response graph till reaching a stable and desired response.

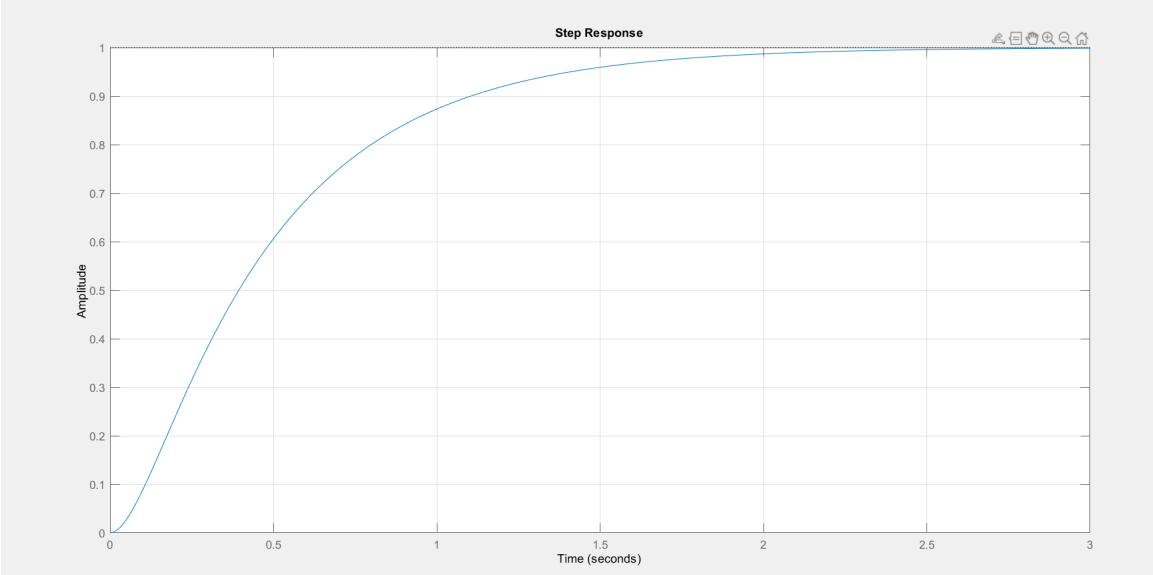
- Kp=0.5


Step Response
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
0.1
0.3
0.5
0.7

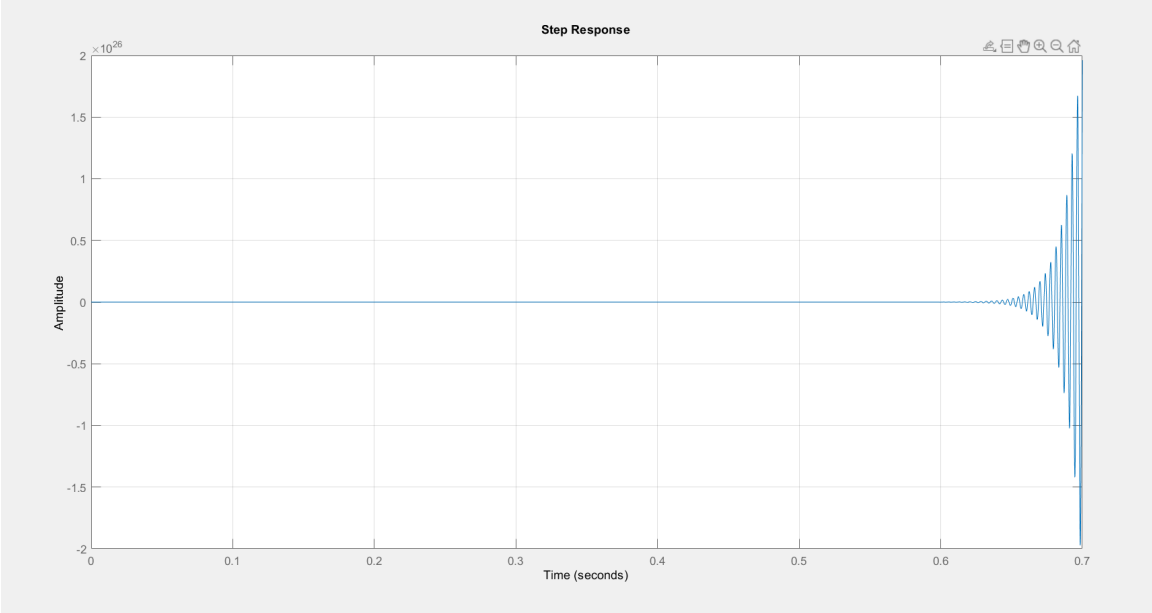
- Kp=0.05


Step Response
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0.1
0.3
0.5
0.7
0.9

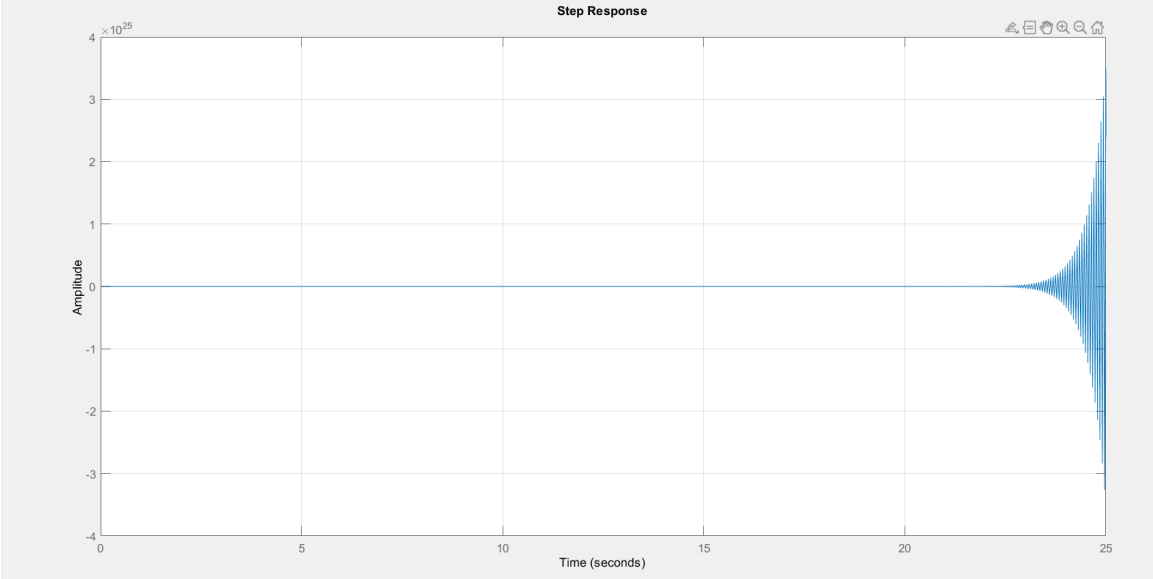
- Kp=0.005



- Kp=500



- Kp=50

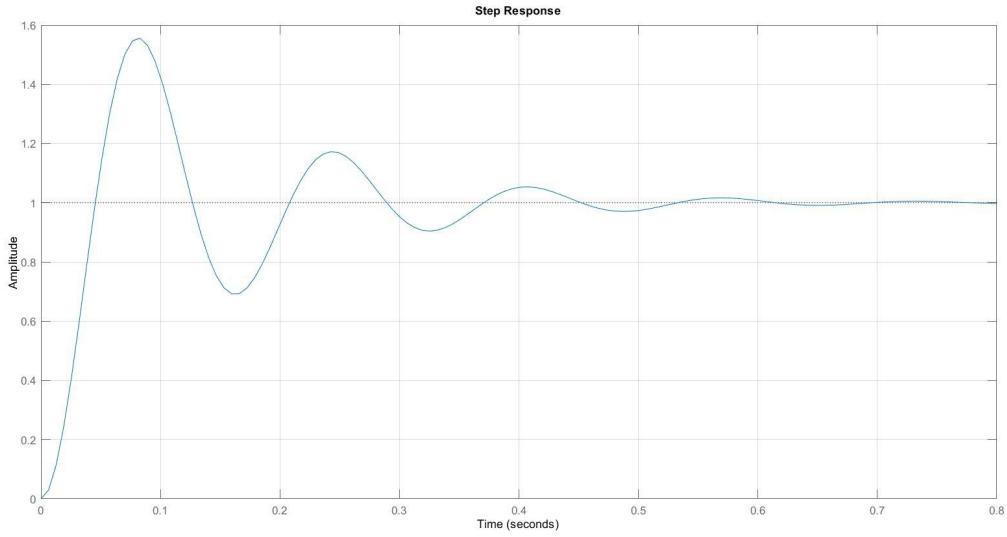


After Practical analysis , it was chosen to use a “P” controller with the Kp value equal=5.

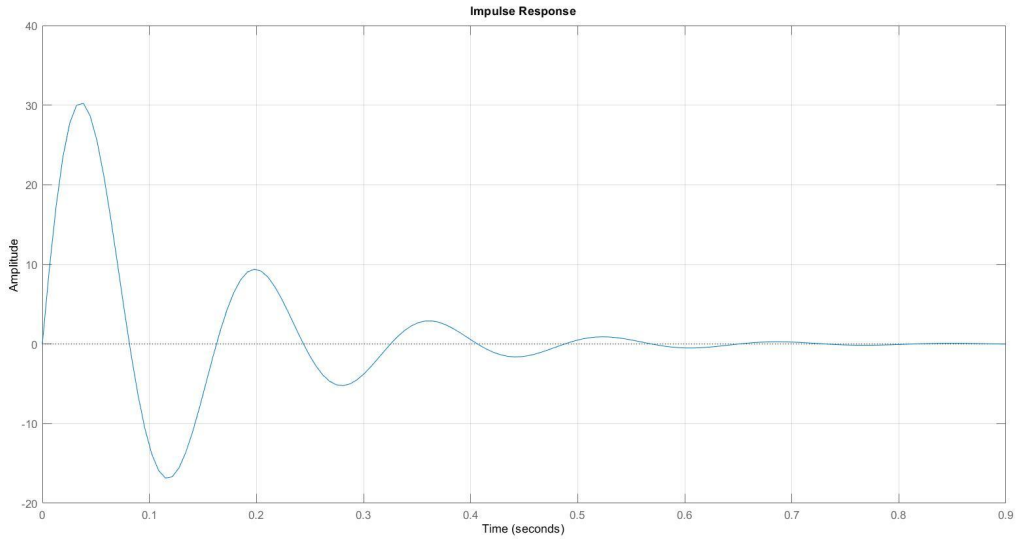
next is all plots for both speed and angle.

4.1.7. Angle:

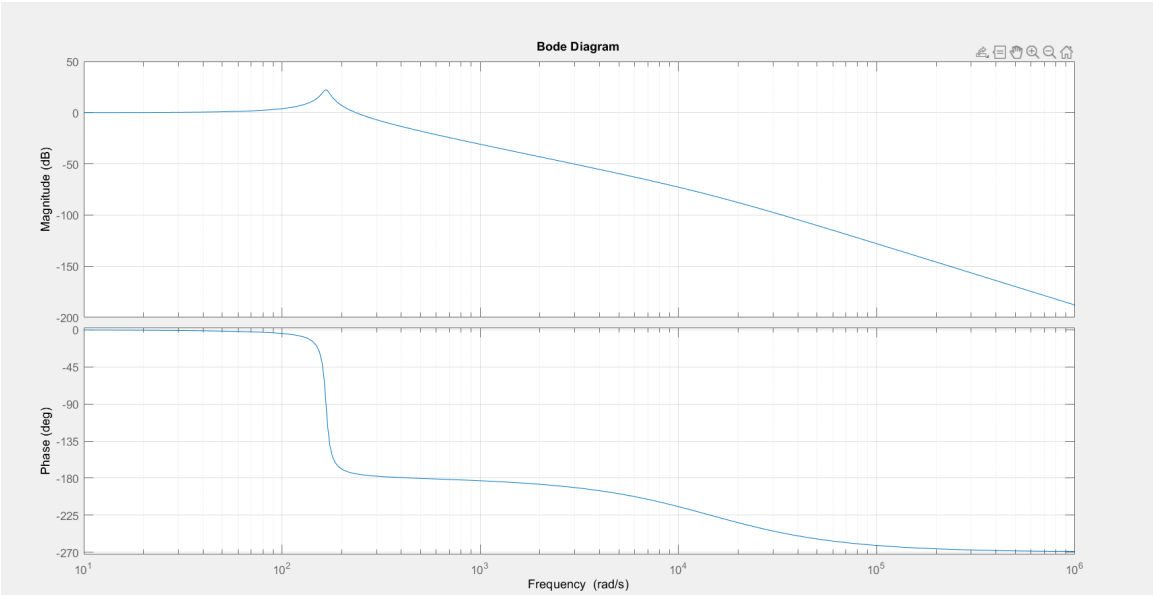
- **Step input:**



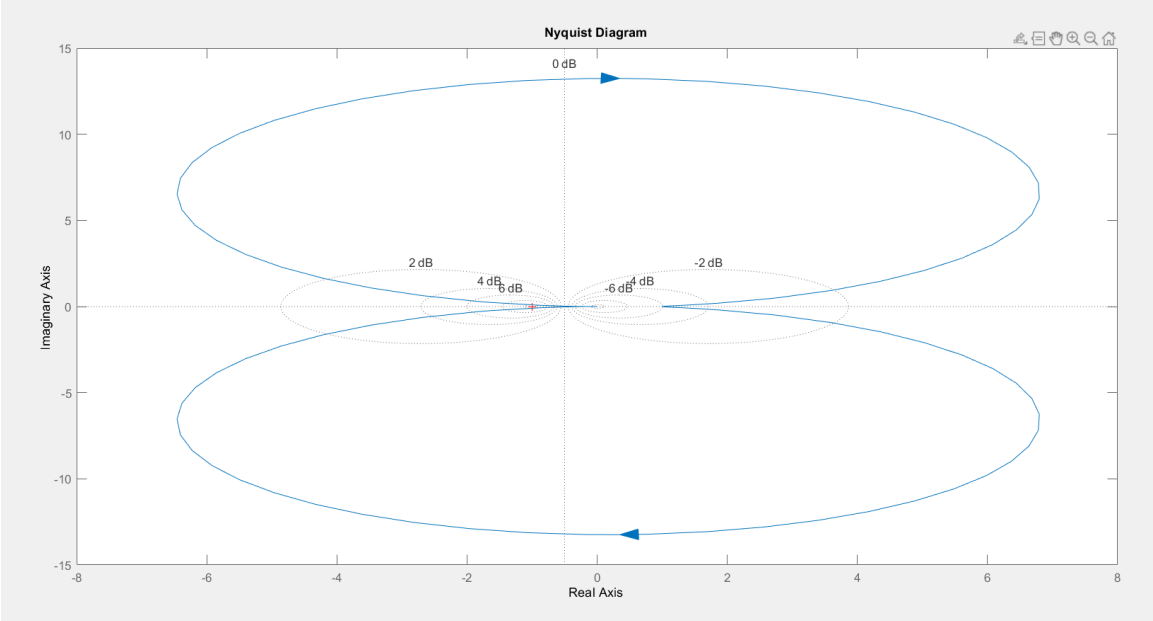
- **Impulse:**


Impulse Response
Amplitude
Time (seconds)
40
30
20
10
0
-10
-20
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

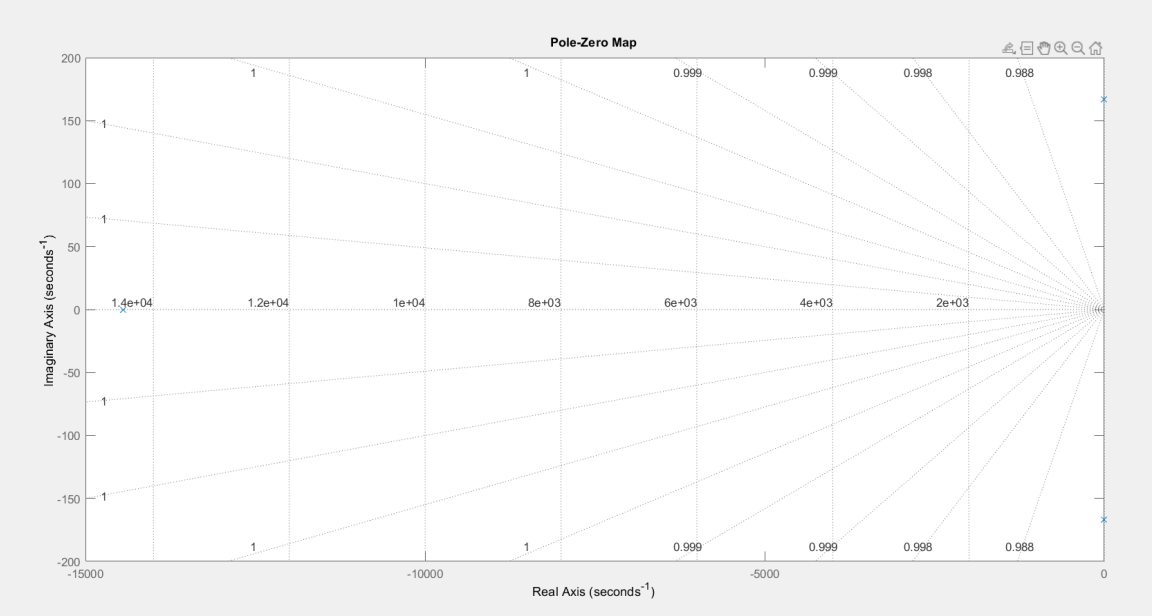
- **Bode Plot:**


Bode Diagram
Magnitude (dB)
Phase (deg)
Frequency (rad/s)
50
0
-50
-100
-150
-200
-45
-90
-135
-180
-225
-270
10^1
10^2
10^3
10^4
10^5
10^6

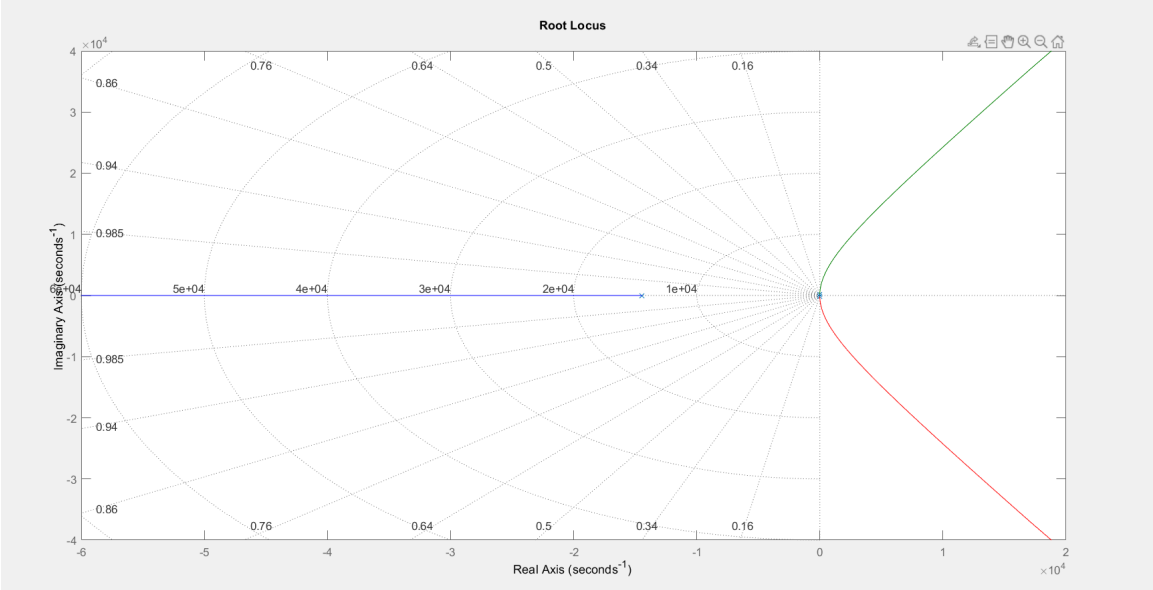
- **Nyquist:**



- **Pole-Zero:**

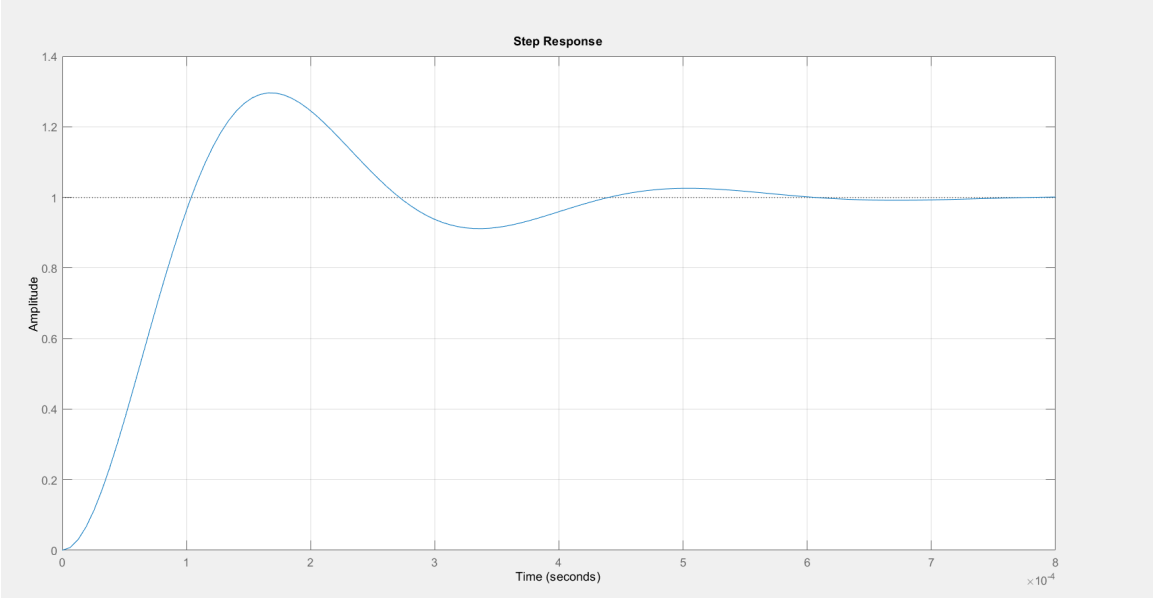


- **Root-Locus:**

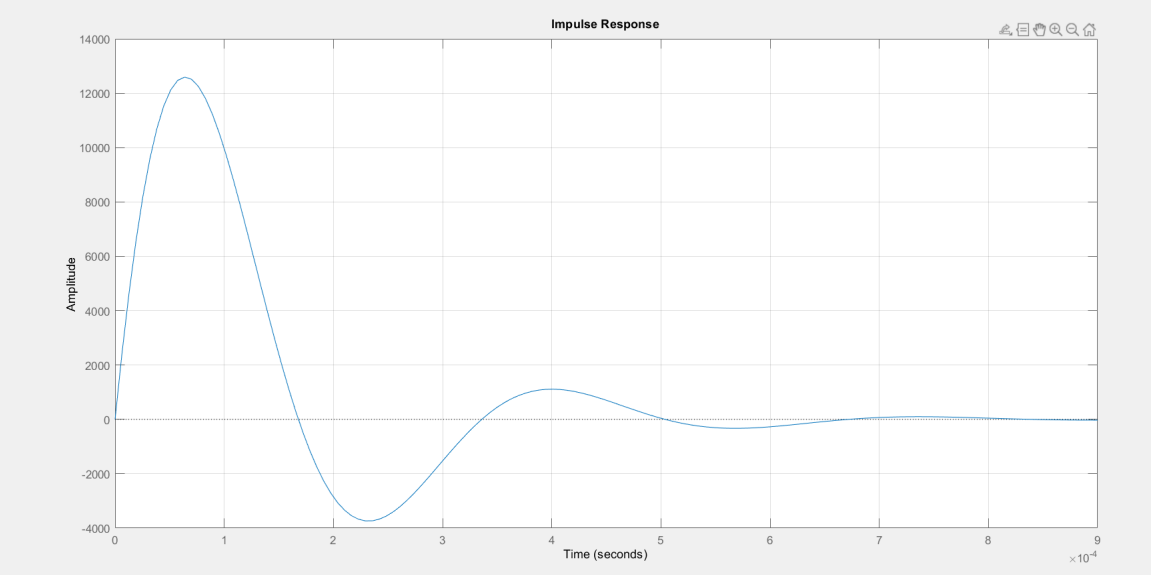


4.1.8. Speed:

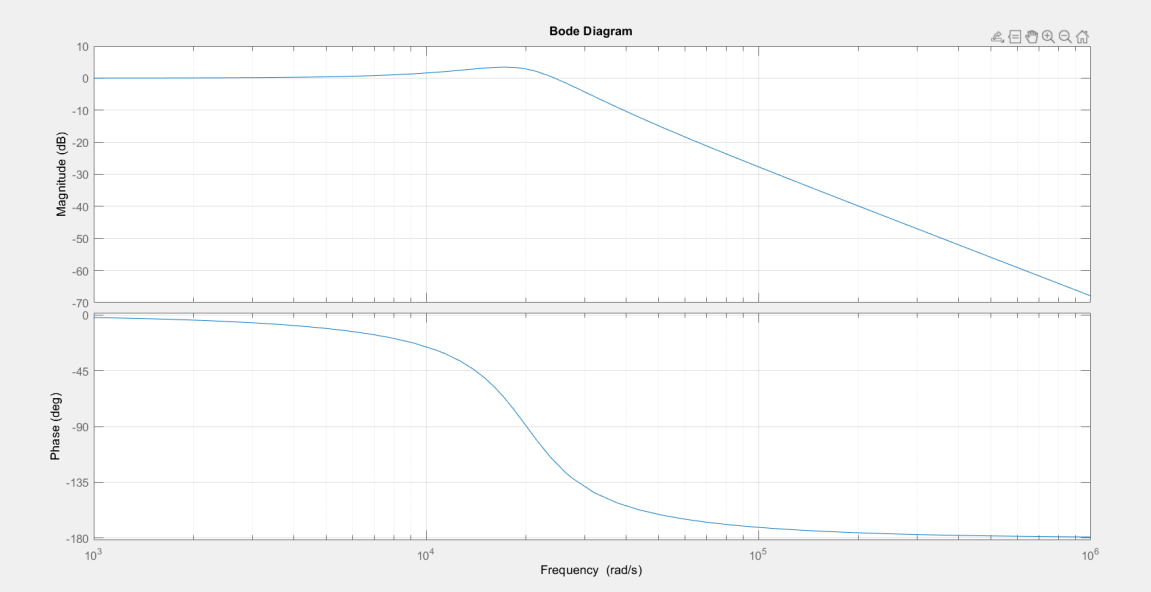
- **Step:**



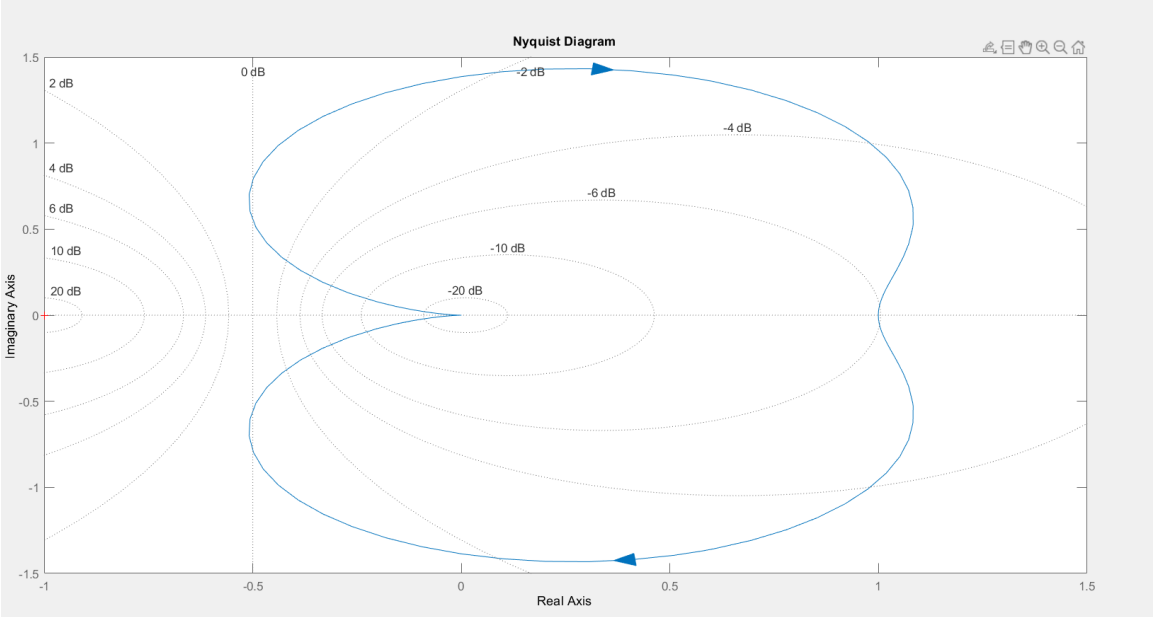
- **Impulse:**



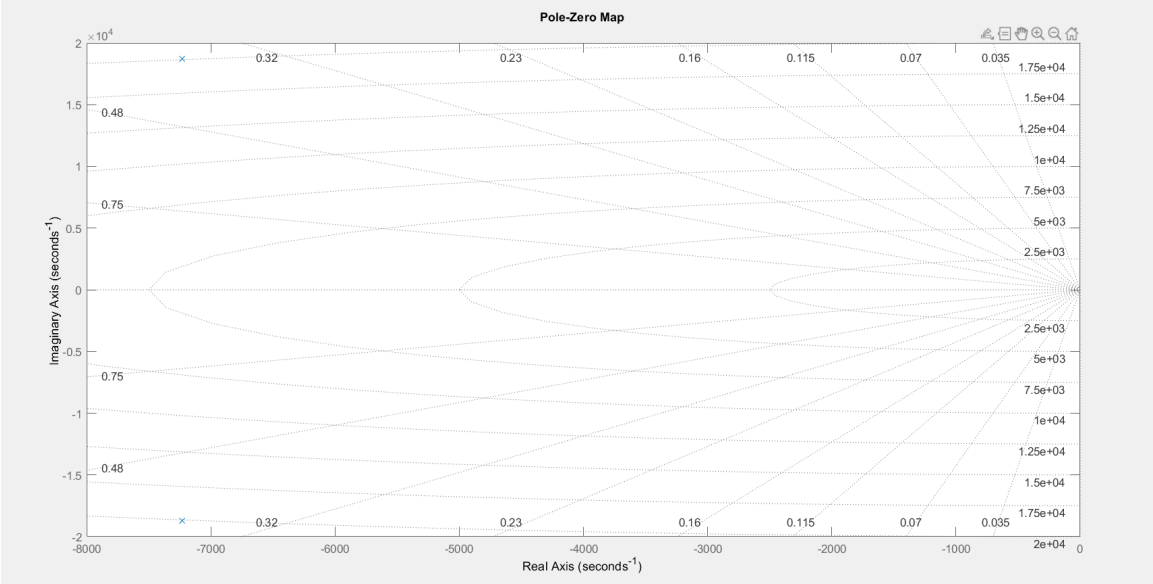
- **Bode Plot:**



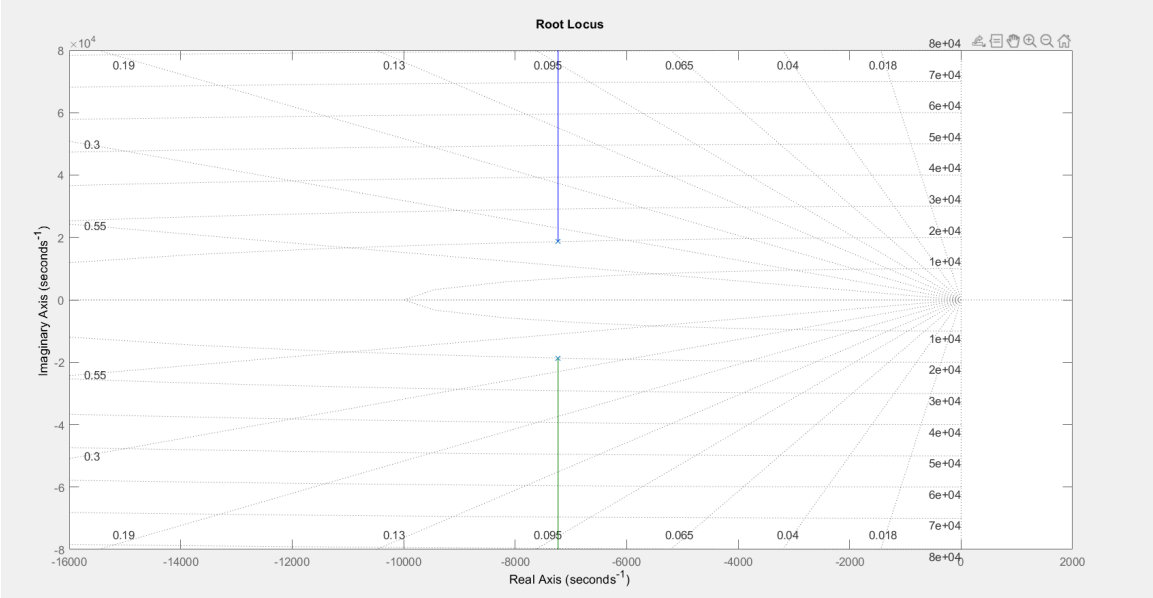
- **Nyquist:**



- **Pole-Zero:**

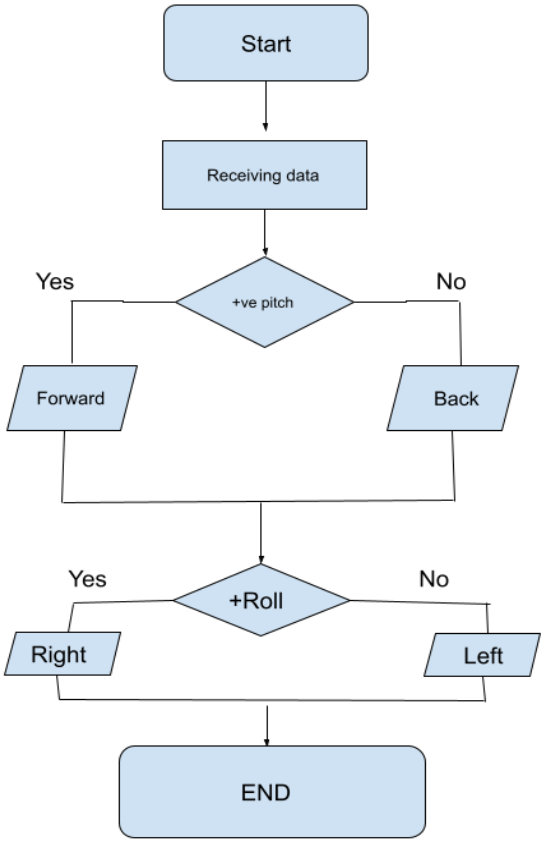


- **Root-Locus:**


Root Locus
×10^4
Imaginary Axis (seconds^-1)
Real Axis (seconds^-1)
0.19
0.13
0.095
0.065
0.04
0.018
0.3
0.55
8e+04
7e+04
6e+04
5e+04
4e+04
3e+04
2e+04
1e+04
-16000
-14000
-12000
-10000
-8000
-6000
-4000
-2000
0
2000

5. Programming

5.1. Flow Chart



5.2. Code for the Car

```
#include <PID_v1.h>
#include <Arduino_FreeRTOS.h>
#include <SPI.h>
#include <nRF24L01.h>
#include <printf.h>
#include <RF24.h>
#include <RF24_config.h>
```

```
#define echoPin A4
#define trigPin A5
#define F_MOTOR_CW 8
#define F_MOTOR_CCW 9
#define B_MOTORS_B 6
#define B_MOTORS_F 7
#define REF_F_MOTOR 15
#define MAX_LEFT_POT 720    // Maximum potentiometer value on
the hacked servo when turning left
#define MAX_RIGHT_POT 290  // Maximum potentiometer value on
the hacked servo when turning right
#define ALLOWANCE_DISTANCE 30.0  // The distance below which the
car would start decreasing its speed
#define ABOUT_TO_CRASH 5    // The distance below which (and
included) the car stops its motion
#define REF_INTERVAL 430;  // The interval of the potentiometer
(MAX_LEFT_POT-MAX_RIGHT_POT)
#define MAX_PWM 255
#define PWM_Pin 4
#define PWM_Pin_Forward 5

int backward;
float pitch;
double roll;
RF24 radio(0, 2);     // CE, CSN
const byte address[6] = "00001";
// Defining a placeholder (structure) to hold all the data to
be received from the transmitter
struct Data_Package
{
      int backward;
      float roll = 454;
      float pitch = 0.0;
};

Data_Package data;    // Create a variable with the above
structure
double actual = 0.0;
float E = 0.0;
```

```
float Kd = 0.0;
// FOR CONTROL
double Kp = 5;
int Ki = 0;
int Kdd = 0;
double U;
//PID myPID(&actual, &U, &roll,Kp,Ki,Kdd, DIRECT);
long duration;  // variable for the duration of sound wave
travel
int distance;   // variable for the distance measurement
/*-----------------------------------------------------------*/
/*---------------------- Tasks Section ----------------------*/
/*-----------------------------------------------------------*/

void UltraSonic(void *pvParameters)   // This is an UltraSonic
task.
{
      (void) pvParameters;

      pinMode(trigPin, OUTPUT);  // Sets the trigPin as an
OUTPUT
      pinMode(echoPin, INPUT);   // Sets the echoPin as an INPUT

      for (;;)   // A Task shall never return or exit.
      {
            digitalWrite(trigPin, LOW);

            vTaskDelay(0.1 / portTICK_PERIOD_MS);

            // Sets the trigPin HIGH (ACTIVE) for 10
microseconds)

            digitalWrite(trigPin, HIGH);

            vTaskDelay(0.1 / portTICK_PERIOD_MS);

            digitalWrite(trigPin, LOW);

            // Reads the echoPin, returns the sound wave travel
```

```
time in microseconds

            duration = pulseIn(echoPin, HIGH);

            // Calculating the distance

            distance = duration *0.034 / 2;  // Speed of sound
wave divided by 2 (go and back) distance

            if (backward == 0)    // checking if the moving is
forward motion
            {
                  if (distance < ABOUT_TO_CRASH)
                  {
                        Kd = 0;
                  }
                  else if (distance < ALLOWANCE_DISTANCE)
                  {
                        Kd = distance / ALLOWANCE_DISTANCE;
                  }
            }
            else
            {
                  Kd = 1;
            }

            // Kd will then be used in the left motors PWM
      }
}

void Receive(void *pvParameters) // This is an NRF recieve task.
{
      (void) pvParameters;
      pinMode(A0, INPUT);
      radio.begin();
      radio.openReadingPipe(0, address);
      radio.setPALevel(RF24_PA_MIN);
      radio.startListening();
```

```
      for (;;)   // A Task shall never return or exit.
      {
            // Serial.println("I'm in the receive");
            // Check whether there is data to be received
            if (radio.available())
            {
                  radio.read(&data, sizeof(Data_Package));    //
Read the whole data and store it into the 'data' structure
            }

            backward = data.backward;
            pitch = data.pitch;
            roll = data.roll;
            Serial.println("I'm Finished");
            Serial.println(roll);
      }
}

void ServoAngle(void *pvParameters)   // This is an ServoHacked
task.
{
      (void) pvParameters;

      pinMode(REF_F_MOTOR, INPUT);
      pinMode(PWM_Pin, OUTPUT); /*declare D3 pin as an output
pin */
      pinMode(F_MOTOR_CW, OUTPUT);
      pinMode(F_MOTOR_CCW, OUTPUT);
      //myPID.SetMode(AUTOMATIC);

      for (;;)   // A Task shall never return or exit.
      {
            //Kp = map(analogRead(A0), 0, 1023, 0, 40); //This
line was used to Online Tune the system using a potentiometer
            actual = analogRead(REF_F_MOTOR);
            //myPID.Compute();    // This was used when we used D
and I in the controller but we only were satisfied with P
            E = roll - actual;
            if (E < 0)
```

```
            {
                  E = E *(-1);
                  digitalWrite(F_MOTOR_CCW, LOW);
                  digitalWrite(F_MOTOR_CW, HIGH);
            }
            else
            {
                  digitalWrite(F_MOTOR_CCW, HIGH);
                  digitalWrite(F_MOTOR_CW, LOW);
            }

            U = Kp * E;
            analogWrite(PWM_Pin, U / REF_INTERVAL *MAX_PWM);
            // END SERVO CONTROL

      }
}

void BackMotors(void *pvParameters)   // This is an Back Motors
task.
{
      (void) pvParameters;
      pinMode(PWM_Pin_Forward, OUTPUT);
      pinMode(B_MOTORS_B, OUTPUT);
      pinMode(B_MOTORS_F, OUTPUT);

      for (;;)   // A Task shall never return or exit. {
            if (backward == 1)
            {
                  digitalWrite(B_MOTORS_F, LOW);
                  digitalWrite(B_MOTORS_B, HIGH);
                  analogWrite(PWM_Pin_Forward, Kd *pitch);

            }
      else
      {
            digitalWrite(B_MOTORS_F, HIGH);
            digitalWrite(B_MOTORS_B, LOW);
            analogWrite(PWM_Pin_Forward, Kd *pitch);
```

```
	}

	// Any code below this comment in this task will be read
by the arduino!!!
	//analogWrite(PWM_Pin_Forward, pitch);

	//dtostrf(roll, 5, 2, data.f);   // Converting double into
character array (for sending with NRF)

	// END BACK MOTORS CONTROL

}

void setup()
{
	Serial.begin(9600);
	while (!Serial);// Wait for Serial terminal to open port
before starting program

	Serial.println("");
	Serial.println("*******************************");
	Serial.println("        Program start          ");
	Serial.println("*******************************");

	xTaskCreate(
		UltraSonic, "UltraSonic", 128    // Stack size
	, NULL, 2  // priority
	, NULL);

	xTaskCreate(
		Receive, "Receive", 128    // Stack size
	, NULL, 2  // priority
	, NULL);

	xTaskCreate(
		ServoAngle, "ServoAngle", 128    // Stack size
	, NULL, 2  // priority
	, NULL);
```

```
      xTaskCreate(
            BackMotors, "BackMotors", 128    // Stack size
      , NULL, 2  // priority
      , NULL);

      vTaskStartScheduler();

}

void loop() {}
```

5.3. Code for the Hand Arduino

```
#include <LiquidCrystal_I2C.h>
#include <Key.h>
#include <Keypad.h>
#include <Arduino_FreeRTOS.h>
#include <LiquidCrystal.h>
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
#include <Wire.h>
#define echoPin 9
#define trigPin 4
#define MAX_LEFT_POT 720
#define MAX_RIGHT_POT 290
#define MAX_PWM 250

void Task(void *pvParameters);
void LCD(void *pvParameters);
const int rs = 2,
      en = 3,
      d4 = 4,
      d5 = 5,
      d6 = 6,
      d7 = 9;
LiquidCrystal_I2C lcd(0x27, 20, 4);
```

```
RF24 radio(7, 8);     // CE, CSN
//void calculate_IMU_error();
const byte address[6] = "00001";
struct Data_Package
{
      int backward;
      float roll = 454;
      float pitch = 0.0;
};

Data_Package data;    //Create a variable with the above
structure

float AccX, AccY, AccZ;
float accAngleX, accAngleY;
float AccErrorX, AccErrorY;
float roll;
float pitch;
const int MPU = 0x68; // MPU6050 I2C address
int c = 0;
float ref = 0.0;
long duration;  // variable for the duration of sound wave
travel

int distance;   // variable for the distance measurement

void LCD(void *pvParameters)
{
      //Serial.begin(9600);
      lcd.init();
      while (1)
      {
            lcd.backlight();
            lcd.setCursor(0, 0);

            lcd.print("Roll: ");
            lcd.print(data.roll);
            lcd.setCursor(0, 1);
            lcd.print("Pitch:");
```

```
            lcd.print(data.pitch);
            vTaskDelay(100 / portTICK_PERIOD_MS);
            lcd.clear();

      }
}

void Keys(void *pvParameters)    // This is a task.
{
      (void) pvParameters;

      const byte ROWS = 4;

      const byte COLS = 3;
      char keys[ROWS][COLS] = {
            {
                  '1', '2', '3' },

            {
                  '4', '5', '6' },

            {
                  '7', '8', '9' },

            {
                  '*', '0', '#' }
      };

      byte rowPins[ROWS] = { 0, 1, A0, A1
      };    //R1,R2,R3,R4

      //connect to the column pinouts of the keypad

      byte colPins[COLS] = { A2, A3, 10
      };    //C1,C2,C3,C4

      Keypad keypad = Keypad(makeKeymap(keys), rowPins, colPins,
ROWS, COLS);
```

```
      for (;;)
      {
            char key = keypad.getKey();
            if (key)
            {
                  Serial.println(key);
            }
      }
}

void test(void *pvParameters)    // This is a task.
{
      (void) pvParameters;
      for (;;)   // A Task shall never return or exit.
      {
            Serial.println("I'm a Dummy Print just here to test
FreeRTOS");
            Serial.println();
      }
}

void Task(void *pvParameters)    // This is a task.
{
      (void) pvParameters;
      radio.begin();
      radio.openWritingPipe(address);
      radio.setPALevel(RF24_PA_MIN);
      radio.stopListening();

      // FOR ACC

      Wire.begin();    // Initialize comunication
      Wire.beginTransmission(MPU);     // Start communication
with MPU6050     // MPU=0x68
      Wire.write(0x6B);     // Talk to the register 6B
      Wire.write(0x00);     // Make reset - place a 0 into the
6B register
      Wire.endTransmission(true);//end the transmission
```

```
      // END FOR ACC

      for (;;)   // A Task shall never return or exit.
      {
            // ACC
            // === Read acceleromter data ===     //
            Wire.beginTransmission(MPU);
            Wire.write(0x3B);     // Start with register 0x3B
(ACCEL_XOUT_H)
            Wire.endTransmission(false);
            Wire.requestFrom(MPU, 6, true);  // Read 6 registers
total, each axis value is stored in 2 registers
            //For a range of +-2g, we need to divide the raw
values by 16384, according to the datasheet
            AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;
      // X-axis value
            AccY = (Wire.read() << 8 | Wire.read()) / 16384.0;
      // Y-axis value
            AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0;
      // Z-axis value
            // Calculating Roll and Pitch from the accelerometer
data
            accAngleX = (atan(AccY / sqrt(pow(AccX, 2) +
pow(AccZ, 2))) *180 / PI) - AccErrorX;// We will consider the
errors to be zero(as default) for now
            accAngleY = (atan(-1 *AccX / sqrt(pow(AccY, 2) +
pow(AccZ, 2))) *180 / PI) - AccErrorY;
            data.roll = accAngleX;
            data.pitch = accAngleY;

            if (data.pitch < 0)
            {
                  data.pitch = -1 *data.pitch;
                  data.backward = 1;
            }
            else
            {
                  data.backward = 0;
            }
```

```
            data.pitch = map(data.pitch, 0, 90, 0, MAX_PWM);
            data.roll = map(data.roll, -90, 90, MAX_LEFT_POT,
MAX_RIGHT_POT);

            radio.write(&data, sizeof(Data_Package));

            Serial.println(data.pitch);
            Serial.println(data.roll);

      }
}

void setup()
{
      Serial.begin(9600);

      xTaskCreate(
            Task, "Calculate angles and send"     // A name just
for humans
      , 128 // Stack size
      , NULL, 2  // priority
      , NULL);

      xTaskCreate(
            LCD, "LCD Task", 128, NULL, 2, NULL);
      //
      //  xTaskCreate(
      //    Keys, "Keys Task", 128, NULL, 2, NULL);

      xTaskCreate(
            test, "test Task", 128, NULL, 2, NULL);

      vTaskStartScheduler();
}

void loop() {}

void calculate_IMU_error()
```

```
{
      // We can call this funtion in the setup section to
calculate the accelerometer and gyro data error. From here we
will get the error values used in the above equations printed
on the Serial Monitor.
      // Note that we should place the IMU flat in order to get
the proper values, so that we then can the correct values
      // Read accelerometer values 200 times
      while (c < 200)
      {
            Wire.beginTransmission(MPU);
            Wire.write(0x3B);
            Wire.endTransmission(false);
            Wire.requestFrom(MPU, 6, true);
            AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;
            AccY = (Wire.read() << 8 | Wire.read()) / 16384.0;
            AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0;
            // Sum all readings
            AccErrorX = AccErrorX + ((atan((AccY) /
sqrt(pow((AccX), 2) + pow((AccZ), 2))) *180 / PI));
            AccErrorY = AccErrorY + ((atan(-1 *(AccX) /
sqrt(pow((AccY), 2) + pow((AccZ), 2))) *180 / PI));
            c++;
      }

      //Divide the sum by 200 to get the error value
      AccErrorX = AccErrorX / 200;
      AccErrorY = AccErrorY / 200;
      c = 0;

}
```

6. Design Evaluation

The experimental results of the system shows that the car works perfectly and translates the hand tilt angles precisely using the controller we did.

Without the controller, the system would oscillate (just as mentioned in the control section of this report) and would not reach the required value in a timely manner.

The theoretical results though were not accurate due to some factor (mentioned below), it resembled the real life response to an extent.

Factors influenced our theoretical part:

1. **Different parameters:**
   Parameters used in obtaining the transfer function are assumptions and not necessarily the correct values for the used motor.

2. **Not considering some factors:**
   In analysis , we ignored all external loads on the system , because they weren't known, and considering all loads won't be easy.

3. **The Frequent usage:**
   The frequent usage of all components while making the project affected some parts of the project which led to different outputs than expected and even some unexpected errors.

4. **Low quality:**
   Some of the materials and components used in the project are not sufficient for such projects and that leads to the need to change them frequently or face a lot of errors. (ex. Servo-Motor , NRF)

7. Appendix

7.1. Angle Control Matlab code:

```
%DC Servo Motor Parameters
Ra=2.6; %Motor Armature Resistance
La=180E-6; %Motor ArmatureInductance
Kt=7.67E-3; %Motor Torque Constant
Kb=7.67E-3; %Back Emf Constant
Je=5.3E-7; %moment of inertia
De=7.7E-6; %Viscous Damping coefficient

Ga = tf(1,[Je,De,0]);
Gi =tf(1,[La,Ra]);
Gm = tf(Gi*Kt*Ga,[Gi*Kt*Ga*Kb,1]);
Kp =5;
K=5/90;
% add feedback command
sys = feedback(Gm*Kp*K,1);

step(sys);
impulse(sys);
bode(sys);
pzmap(sys);
rlocus(sys);
grid;
```

7.2. Speed Control Matlab code:

```
%DC Servo Motor Parameters
Ra=2.6; %Motor Armature Resistance
La=180E-6; %Motor ArmatureInductance
Kt=7.67E-3; %Motor Torque Constant
Kb=7.67E-3; %Back Emf Constant
Je=5.3E-7; %moment of inertia
De=7.7E-6; %Viscous Damping coefficient

Fi=tf(1,[La,Ra]);
Ft=tf(1,[Je,De]);
Fm=tf(Fi*Kt*Ft,(Fi*Kt*Ft*Kb)+1);

Kp=5;
K=5/90;
```

```
sys= feedback(Fm*Kp*K,1);
step(sys);
impulse(sys);
bode(sys);
pzmap(sys);
rlocus(sys);
grid;
```

ATMEGA328P Codes:

```
/*Beginning of Auto generated code by Atmel studio */
#include <Arduino.h>

/*End of auto generated code by Atmel studio */

/*Beginning of Auto generated code by Atmel studio */
#include <Arduino.h>

/*End of auto generated code by Atmel studio */
#include <LiquidCrystal.h>
#include <Wire.h>
     //Beginning of Auto generated function prototypes by Atmel
Studio
//End of Auto generated function prototypes by Atmel Studio

//Beginning of Auto generated function prototypes by Atmel
Studio
void calculate_IMU_error();
//End of Auto generated function prototypes by Atmel Studio

const int MPU = 0x68; // MPU6050 I2C address
float AccX, AccY, AccZ;
float GyroX, GyroY, GyroZ;
float accAngleX, accAngleY, gyroAngleX, gyroAngleY, gyroAngleZ;
float roll, pitch, yaw;
float AccErrorX, AccErrorY, GyroErrorX, GyroErrorY, GyroErrorZ;
float elapsedTime, currentTime, previousTime;
int c = 0;

LiquidCrystal lcd(1, 2, 4, 5, 6, 7);  // Creates an LCD object.
```

```
Parameters: (rs, enable, d4, d5, d6, d7)

void setup()
{
      Wire.begin();    // Initialize comunication
      Wire.beginTransmission(MPU);     // Start communication
with MPU6050    // MPU=0x68
      Wire.write(0x6B);     // Talk to the register 6B
      Wire.write(0x00);     // Make reset - place a 0 into the
6B register
      Wire.endTransmission(true);//end the transmission
      /*
      // Configure Accelerometer Sensitivity - Full Scale Range
(default +/- 2g)
      Wire.beginTransmission(MPU);
      Wire.write(0x1C);                     //Talk to the
ACCEL_CONFIG register (1C hex)
      Wire.write(0x10);                     //Set the register
bits as 00010000 (+/- 8g full scale range)
      Wire.endTransmission(true);
      // Configure Gyro Sensitivity - Full Scale Range (default
+/- 250deg/s)
      Wire.beginTransmission(MPU);
      Wire.write(0x1B);                     // Talk to the
GYRO_CONFIG register (1B hex)
      Wire.write(0x10);                     // Set the register
bits as 00010000 (1000deg/s full scale)
      Wire.endTransmission(true);
      delay(20);
      */
      // Call this function if you need to get the IMU error
values for your module
      calculate_IMU_error();
      delay(20);
      lcd.begin(16, 2);     // Initializes the interface to the
LCD screen, and specifies the dimensions (width and height) of
the display }

      lcd.print("Hello There");
```

```
      delay(3000);
}

void loop()
{
      // === Read accelerometer data ===     //
      Wire.beginTransmission(MPU);
      Wire.write(0x3B);     // Start with register 0x3B
(ACCEL_XOUT_H)
      Wire.endTransmission(false);
      Wire.requestFrom(MPU, 6, true);  // Read 6 registers
total, each axis value is stored in 2 registers
      //For a range of +-2g, we need to divide the raw values by
16384, according to the datasheet
      AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;     //
X-axis value
      AccY = (Wire.read() << 8 | Wire.read()) / 16384.0;     //
Y-axis value
      AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0;     //
Z-axis value
      // Calculating Roll and Pitch from the accelerometer data
      accAngleX = (atan(AccY / sqrt(pow(AccX, 2) + pow(AccZ,
2))) *180 / PI) - AccErrorX;     // AccErrorX ~(0.58) See the
calculate_IMU_error()custom function for more details
      accAngleY = (atan(-1 *AccX / sqrt(pow(AccY, 2) + pow(AccZ,
2))) *180 / PI) - AccErrorY;     // AccErrorY ~(-1.58)
      // === Read gyroscope data ===   //
      previousTime = currentTime;// Previous time is stored
before the actual time read
      currentTime = millis();     // Current time actual time
read
      elapsedTime = (currentTime - previousTime) / 1000;     //
Divide by 1000 to get seconds
      Wire.beginTransmission(MPU);
      Wire.write(0x43);     // Gyro data first register address
0x43
      Wire.endTransmission(false);
      Wire.requestFrom(MPU, 6, true);  // Read 4 registers
total, each axis value is stored in 2 registers
```

```
      // Correct the outputs with the calculated error values
      // GyroErrorZ ~ (-0.8)
      // Currently the raw values are in degrees per seconds,
deg/s, so we need to multiply by sendonds (s) to get the angle
in degrees

      // Complementary filter - combine acceleromter and gyro
angle values
      roll = accAngleX;
      pitch = accAngleY;

      // Print the values on the serial monitor

      // Prints "Arduino" on the LCD
      // delay(3000);

      //delay(3000);   // 3 seconds delay
      //lcd.setCursor(2,1); // Sets the location at which
subsequent text written to the LCD will be displayed
      //lcd.print("Hello");
      //delay(3000);
      //lcd.clear();   // Clears the display
      //lcd.print(roll);
      //lcd.print("/");
      //lcd.print(pitch);
      //lcd.print("/");
      //lcd.println(yaw);
      // lcd.clear(); // Clears the LCD screen

      // values should be casted to string/char array using itoa
or dtoa methods
      // Clears the display
      lcd.print("roll:");
      lcd.print(roll);
      lcd.setCursor(0, 1);
      lcd.print("Pitch:");
      lcd.print(pitch);
```

```
      delay(1000);
      lcd.clear();
      //lcd.print(millis() / 1000);
}

void calculate_IMU_error()
{
      // We can call this funtion in the setup section to
calculate the accelerometer and gyro data error. From here we
will get the error values used in the above equations printed
on the Serial Monitor.
      // Note that we should place the IMU flat in order to get
the proper values, so that we then can the correct values
      // Read accelerometer values 200 times
      while (c < 200)
      {
            Wire.beginTransmission(MPU);
            Wire.write(0x3B);
            Wire.endTransmission(false);
            Wire.requestFrom(MPU, 6, true);
            AccX = (Wire.read() << 8 | Wire.read()) / 16384.0;
            AccY = (Wire.read() << 8 | Wire.read()) / 16384.0;
            AccZ = (Wire.read() << 8 | Wire.read()) / 16384.0;
            // Sum all readings
            AccErrorX = AccErrorX + ((atan((AccY) /
sqrt(pow((AccX), 2) + pow((AccZ), 2))) *180 / PI));
            AccErrorY = AccErrorY + ((atan(-1 *(AccX) /
sqrt(pow((AccY), 2) + pow((AccZ), 2))) *180 / PI));
            c++;
      }

      //Divide the sum by 200 to get the error value
      AccErrorX = AccErrorX / 200;
      AccErrorY = AccErrorY / 200;
      c = 0;

}
```