MAY 10, 2021

Rachel Haighton	*40021349*
Cyril Girgis	*40028766*
Faisal Bari	*27517440*
Khaled Abdelkader	*40038214*

CAR SIMULATOR FINAL PROJECT

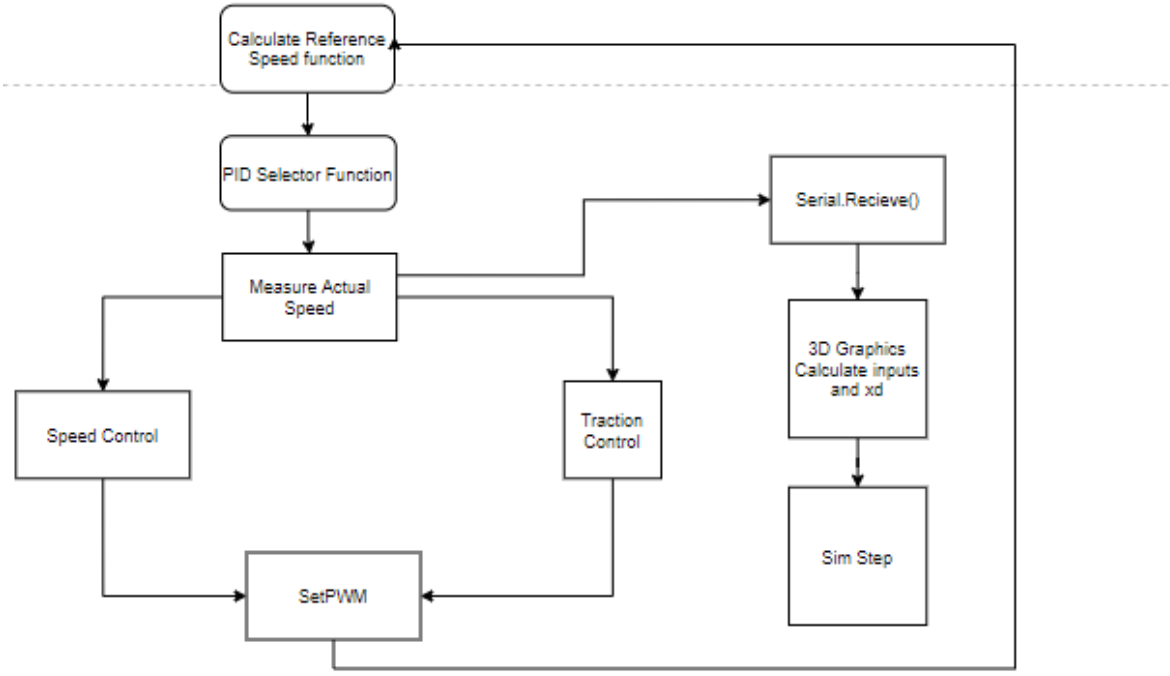
MECH 471

TABLE OF CONTENTS

HIL3 IMPLEMENTATION

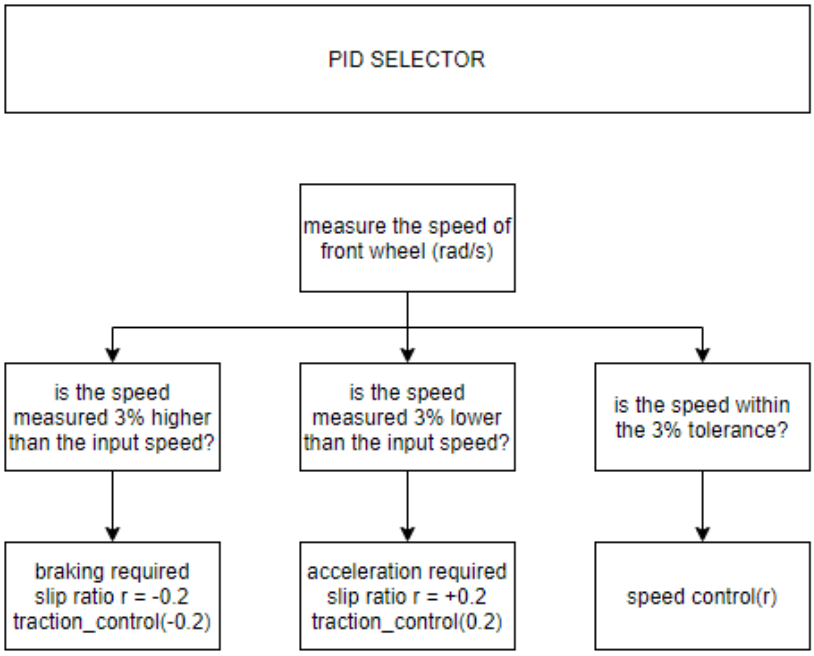
MANDATORY SECTION FLOW DIAGRAM AND LOGIC

The mandatory section is compromised of three main registry sections, two controller's functions and a function to set the reference values using serial communication and the PC keyboard. The original idea of this project was to input our desired speed, our desired steering yaw angle and breaking commands from the keyboard. The controllers would then perform the necessary computations for the PID loops and go through the hardware in the loop simulation. Finally, the feedback red from the simulation Arduino board was going to be used in the car model simulation. Thus, the inputs for the graphics simulation would be the actual speed and actual servo angle. In summary, the keyboard would have been used to input desired values while the graphics simulator would serve as a visual output to display the values on the simulator Arduino board. Unfortunately, we were never able to make Serial.Recieve() function properly and as a result, this flow diagram and loop was never fully completed, however, the rest of functions functioned properly and we were able to display the reference and actual values on the command line. Thus, this only prevented us from coupling it to 3D graphics but not from getting visual feedback.



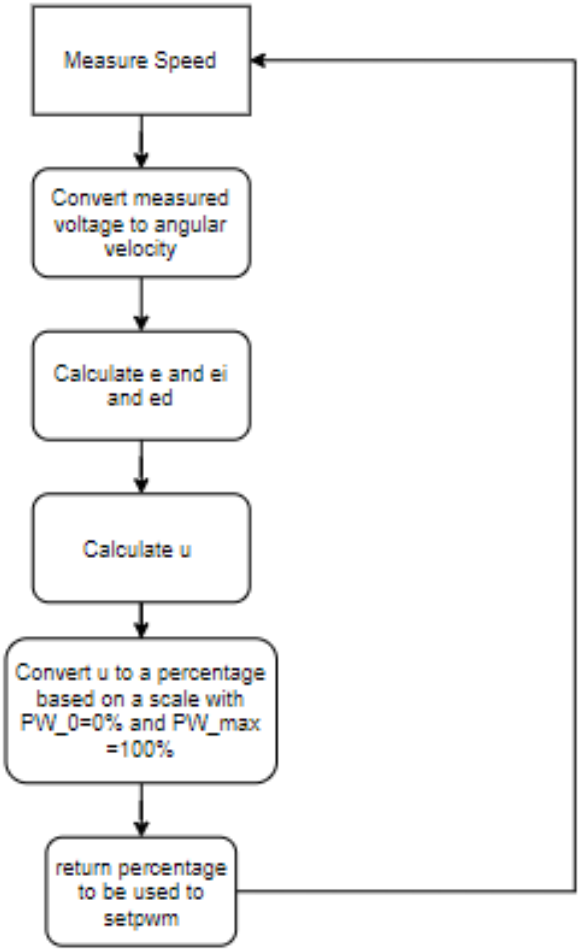
PID SELECTOR FUNCTION

If values read in a 3% tolerance are okay, the PID_selector reads the user input value and compares it to the values measured/read by the system. It then chooses a PID controller based on the results. While simulating the program on visual studios this proved to be ideal, once implemented using hardware in the loop simulation, the sampling period made for a bit of a less accurate traction and speed controller. As a result, the logic had to be adjusted so that it only uses traction control for the initial launch or initial breaking but then switches to the speed controller afterwards.

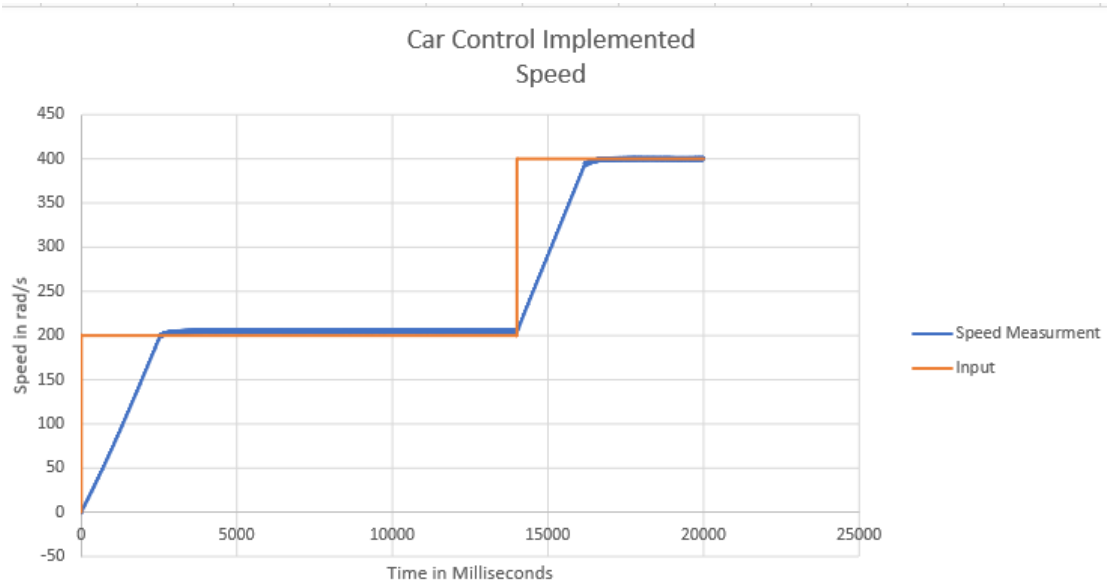


SPEED CONTROLLER

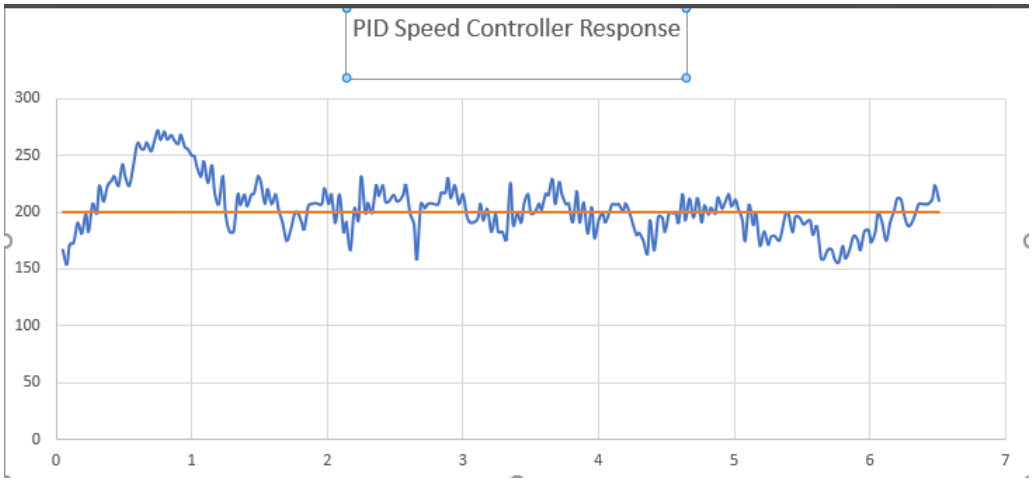
The speed controller was programmed on both visual studios and the Arduino hardware in the loop simulation. The flow diagram, as well as the results, can be seen in the figures below. The PC program ended up having a much better response. This was to be expected as it involves no hardware, no noise, and a much better sampling period. As a result, this can be viewed as the ideal response and can be used and compared to the actual response from the Arduino.



The PC response was with Kp= 50,ki = 10 and Kd=12 and also used anti wind up logic used in dynamixel smart servos with the code being in the PID simulator folder. The response as seen below is the ideal one that we had intended to reproduce using the Arduino boards. In the response graph below, the desired speed is changed throughout the simulation and this is thus the response to two separate step response inputs. The rising time is considered a bit slow but it allowed us to have minimal overshoot and no oscillations at steady state. The sim steps had dt =0.001 which makes for an extremely small sampling period.



Unfortunately, this controller did not produce similar results when using the Arduino boards. This can be due to the sampling period being much lower, a topic which will be covered later in this report, and due to the presence of much more noise when it comes to using actual hardware. When implementing and testing multiple PID controllers, we found that the derivative term led to a lot of instability, this is to be expected since the derivative of error tends to be a lot more susceptible to noise. As a result, a heuristic approach was used to design a PI controller. The proportional gain was increased till stable and consistent oscillations were reached. Then, the Ziegler- Nichols was used to get initial gains and then adjusted to optimize the results. We finally used a PI controller with Ki=0.6 and Kp=0.6 which netted the results below:

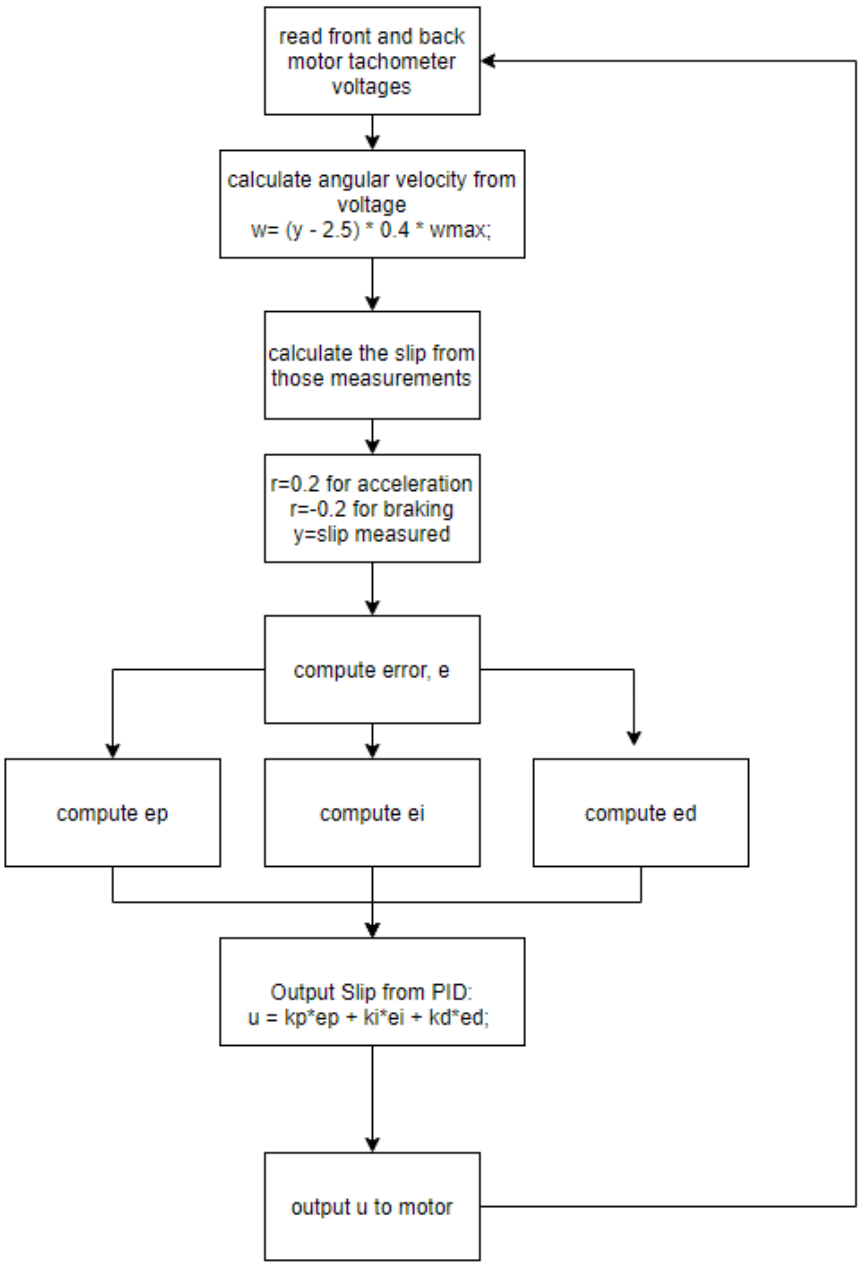


The response, while far from perfect, does have a good response time, even better than computer.
It does not have a consistent steady state error, but rather steady state oscillation that does not seem to dampen. The average amplitude over the timespan shown above is 204.3 rad/s which shows that it is converging towards the proper value, however, the overshoot is as large as 20% over the desired value even after reaching steady state oscillation.

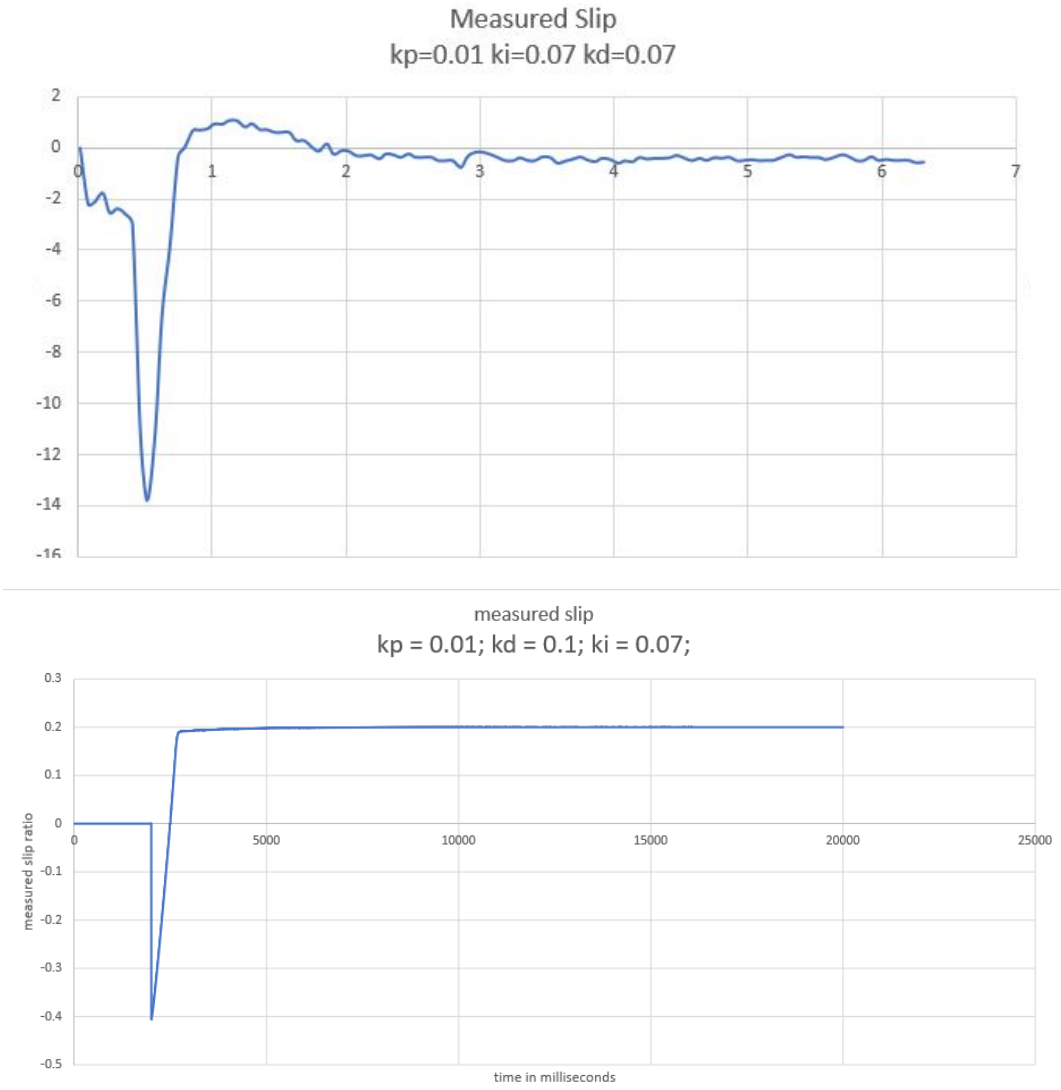
TRACTION CONTROLLER FOR BRAKING AND ACCELERATION

In this traction controller it is assumed that the forward motors will be altered, and the drive motor, wb, will be held constant.

$$Slip\ Ratio = \frac{w_f - w_b}{w_f}$$

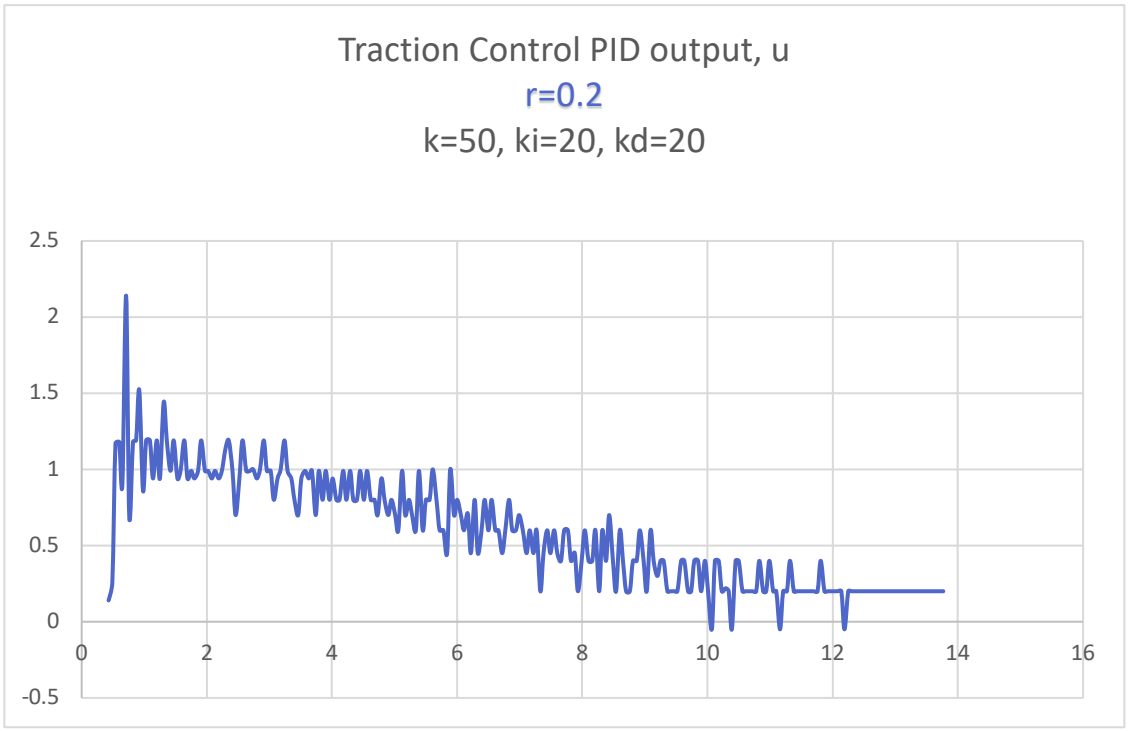


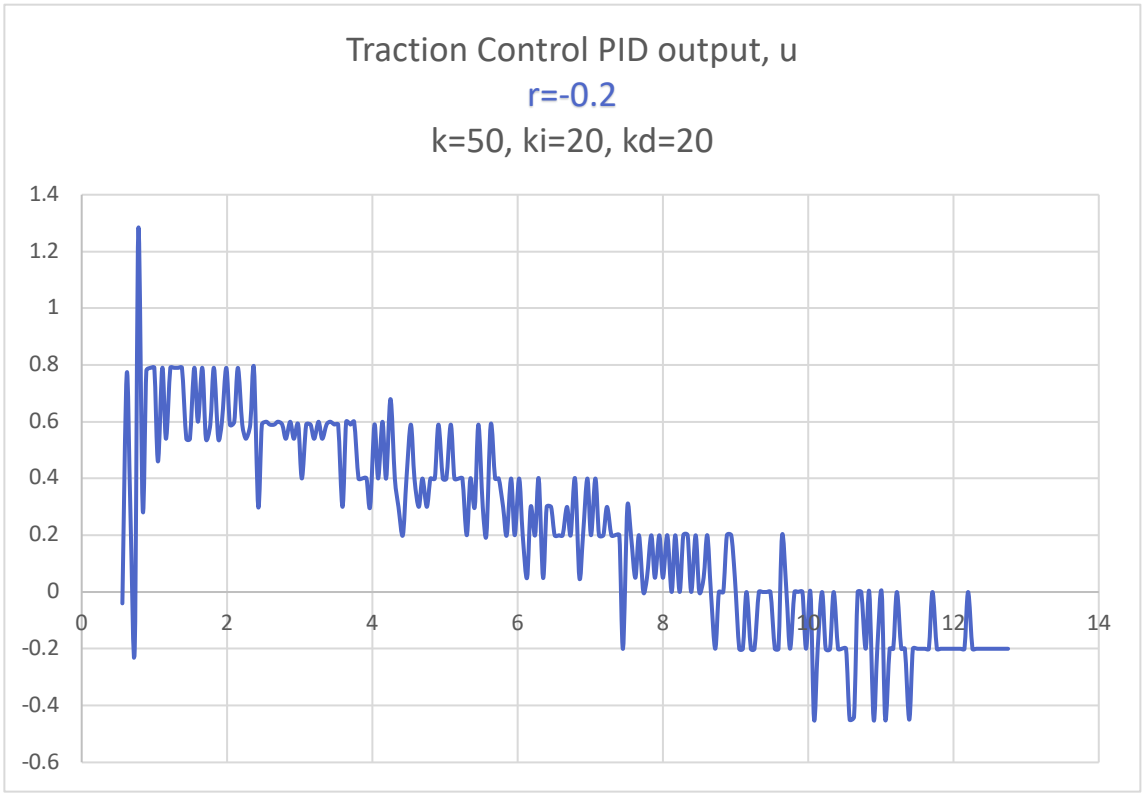
Here is the Initial traction control using the HIL3 simulator showed the following for r=0.2, it was improved later:



In the simulated graph the motors are fed 1V for 2 seconds and then traction control is turned on. While this shows great room for improvement in terms of tuning, it is a start.

Below is the PID gains used for braking (r= -0.2) and traction (r = 0.2). There is significant slip at the start, but it must rain before the rainbow.

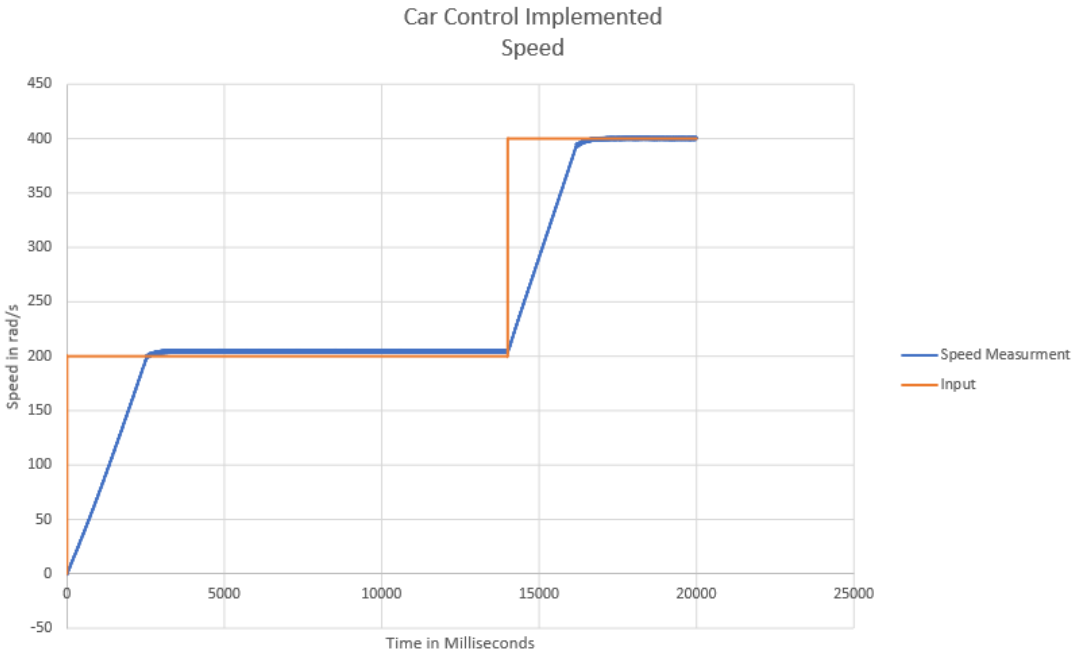

Traction Control PID output, u
r=0.2
k=50, ki=20, kd=20
2.5
2
1.5
1
0.5
0
-0.5
0
2
4
6
8
10
12
14
16


Traction Control PID output, u
r=-0.2
k=50, ki=20, kd=20
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0
2
4
6
8
10
12
14

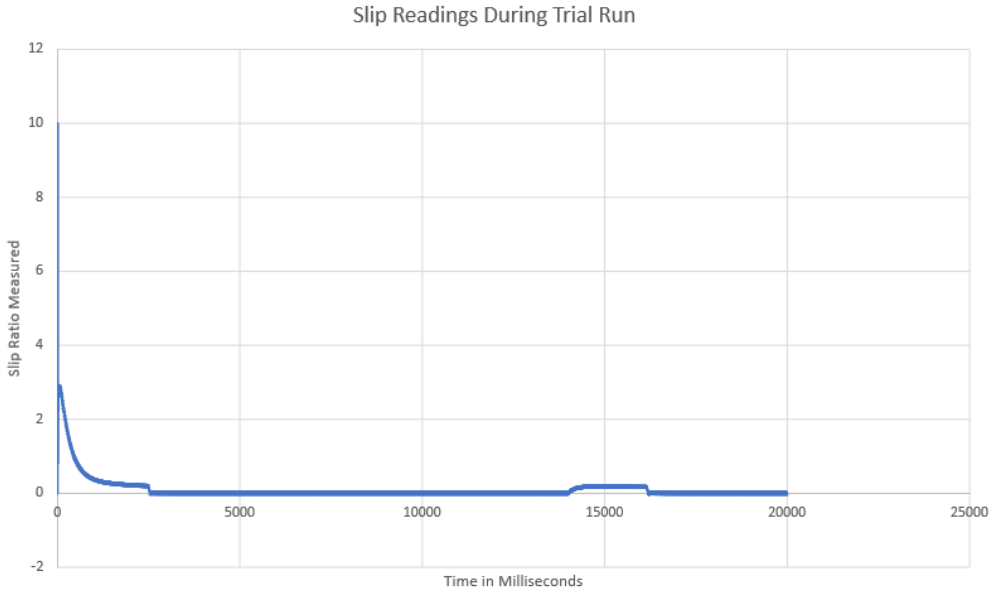
RESULTS COMBING THE CONTROLLERS UNDER HIL3

Once all the controllers were tuned to satisfaction, the HIL3 program was run using the PID selector. The speed was set using a series of if statements to replicate a step response.

Here is the simulation using Visual Studios:



In this simulation, the speed is set to 200rad/s and then 400rad/s after 14 seconds. Below is the corresponding slip readings, during the trial.



REGISTER PROGRAM LOGIC

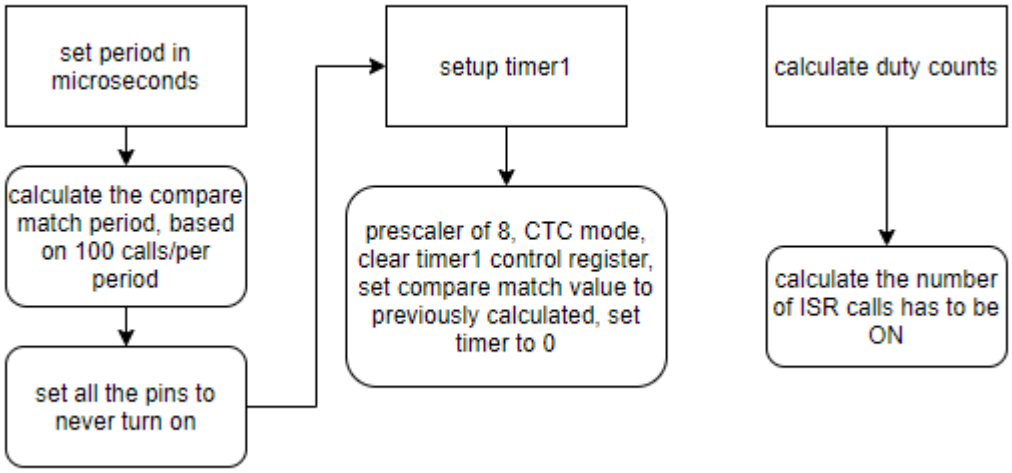
This section briefly describes the most important register programs, including the pwm function for the motors, the ADC, and the time count.

PWM FUNCTION

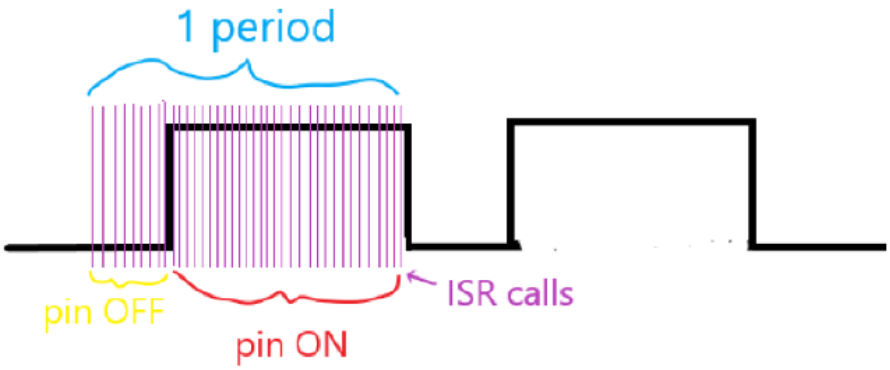
The PWM function creates a pulse width signal on any of the 14 digital pins. The user can set the period in microseconds and duty in percentage.

A period is divided up into 100 ISR compare match calls using timer 1. The compare match value is calculated based on the period, and the duty cycle tells the ISR when to turn the pin on or off.

A function is used to setup timer1, which is called one the period is set by the user.



In general, servo motors run on a frequency of 50Hz, this is equivalent to a period of 20ms.



At lower frequencies such as 10ms, the compare match interrupt was fired too often resulting in issues with the ADC and time counter functions. Since TIMER1_COMPA_vect is 12th and ADC_vect is 24 in the priority order, the program used to hang when it would need to read from the analog ports. Fortunately, the 20ms period does not pose this problem.

Unfortunately, as functions were added and integrated to the main program, the setpwm function did not continue working properly and we were unable to find the bug, there was input being sent and received between the arduino's but the controller was receiving no feedback. As a result, in the main program seen in the project, polling is used to set the duty of the motors. The prescaler is set to 64 and the overflow compare match value is given by :

$\frac{Period(seconds)}{Seconds\ per\ count}$= Period in terms of count

Once this is defined, the duty calculated from the pid loop and given as a percentage is converted to a number in counts using duty = percentage *Period in terms of count. The code within the interrupt function can be shown below, with z_9 being the duty in count and being the value that determines how long the motor should be turned on within the period.

```
ISR(TIMER1_COMPA_vect)
// timer1 output compare match interrupt function
{
//    (count-- > value) ? PORTB |= BIT(1) : PORTB &= ~BIT(1);
//pin 9
  static float prev, after;

  PORTB |= BIT(1); //on for z us
  prev = TCNT1;
  while((TCNT1-prev) < z_9+10){  }
  PORTB &= ~BIT(1);
  after = TCNT1;

 // Serial.print("\tafter - prev = ");
 // Serial.println(after - prev);
}
```

ADC FUNCTION

We think that our ADC function is what proved to be the limiting factor of our controllers. The function used is like the one seen in the lectures but in registry form. The voltage of the tachometer is red 200 times within the ADC interrupt with the sum being stored in a variable. A count of how many times the interrupt has been called is being used as well and is made a global volatile variable. Once count reaches 200, it stops calling the ADC conversion registry and the average voltage is calculated and returned.

A period in our program is approximatively 0.02 seconds with most of that time being spent within the ADC conversion functions. The prescaler is set at the largest value, attempts at lowering the prescaler led to a better sampling frequency but visibly poor resolution with the speed response being much worse off. We also tried to look into maybe lowering the sample size but this also led to more noise and a worse noise at numbers below 175 samples. A possible improvement that could have been made would have been adding an RC circuit.

TIMER (MICROS())

A need for the equivalence of “micros()” was identified for the PID controller. In the register programing version, Timer0 is used along with the TIMER2_OVF_vect interrupt.

TIMER2_OVF_vect simply increments the time by 255/16.0e6*8 each time it is called in order to output the time in microseconds.

```
ISR(TIMER2_OVF_vect)
{
  // increase offset by the roll time
  t2_offset += 255.0/16.0e6*8;

}
```

The timer2 clock must be setup before use though. The function get_time_us_setup() sets timer2 to have a prescaler of 8, with enabled overflow interrupts. This function also starts the clock at 0.

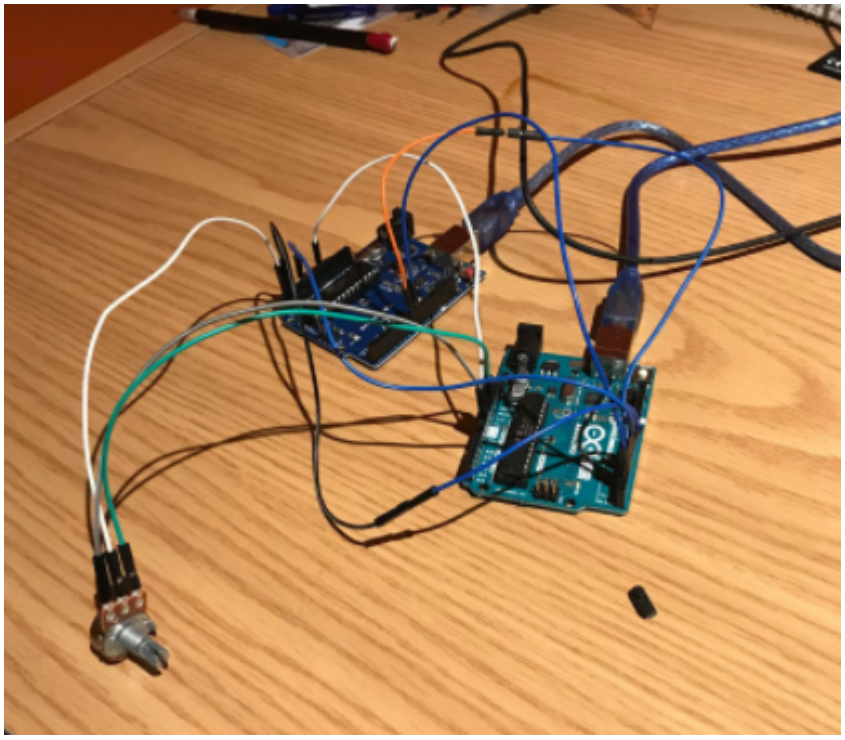
ADDITIONAL PROJECT COMPONENTS

SERIAL COMMUNICATION

Originally, as described in the first flow diagram, the intention was to have communication between the computer and the controller. The desired speed and the servo’s desired yaw would be set using the keyboard, sent to the Arduino using serial communication and then measured speed and yaw would be sent back to the pc and used as inputs in the graph simulation using the simple model.

We were able to send from the pc to the Arduino data as shown in the picture below. The running Arduino is communicating with our command line as this is what would typically be printed to the serial monitor. When sending data to control the reference speed or the steering angle. In the picture below we can see that the angle printed out increases. This is due to the left and right keyboard buttons being pressed which increments the yaw angle. Where lied the problem was when trying to control both speed and yaw at the same time. This would lead to either one or the other not responding properly. After several days of debugging without a solution we decided to stick to steering control using the computer. We found a work around by making the speed adjustable using a

potentiometer. The potentiometer is powered by the Arduino with the signal sent to A4 pin which then uses the ADC function to get the voltage, the voltage is then scaled to a value that's between 0 and 810 rad/s which is used as the reference speed. This proved to work as can be seen in the picture below where reference speed and angle is printed out. A video is also in the folder displaying both being changed at the same time



We were unfortunately not able to use Serial.Recieve() to get data as the values received were either unstable, garbage or very difficult to parse. Had we been able to do it, we would have sent as data the measured speed and measured yaw to be used as inputs.

STEERING CONTROL

Introduction:

The steering control will be used to control the direction of the servo, in which will control the direction of the robot/car. Steering is used when the user wants the robot/car to turn at a certain direction. A servo is used to change the direction of the two front wheels.

How it works:

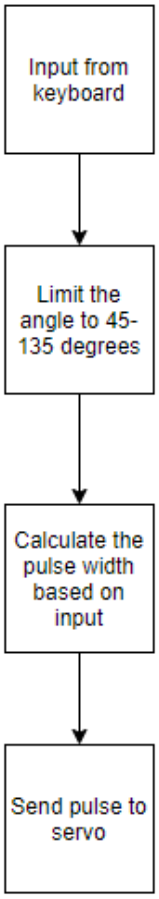
The servo is limited when it comes to rotation, some servos can turn up to 270 degrees, and for some it is much less. We will limit the rotation of the servo from 45 to 135 degrees. This way we can prevent the wheels from ever making a sharp 90 degree turn. The 90 degrees angle on the servo will give the straight motion of the robot/car. When the angle moves towards 45 degrees, we should see a rotation of the wheel towards the right. Likewise, if the angle of the servo were to move towards 135 degrees, the wheels should rotate leftward. The function steerin_control, takes in one parameter, and that's the angle in degrees. The input is received from the keyboard using serial communication. Timer0 was used to send a pulse to the servo, where the period is 2.5 ms. According

to servo specifications, a pulse width of 1.5 ms is 90 degrees. Using a simple equation, we can calculate the pulse width for any values between 45 and 135 degrees as shown below.

```
if(angle_degrees > 135){
  angle_degrees = 135;
}
else if (angle_degrees < 0){
  angle_degrees = 0;
}

pulse = ((0.5/90)*(angle_degrees-45)+1.25)*1000;
```

The steering control will remain open loop and a PID will not be implemented as a simple pulse to the servo should be sufficient. The code logic is as follows:



If a more comprehensive controller is required to control the angle of the steering and to also control the speed of each wheel when turning, then the following equations are to be used:

```
//steering equations
Rsteer = L/sin(yaw);
Rback = sqrt(Rsteer*Rsteer-L*L);
Rrobot = sqrt(Rback*Rback+(L/2)*(L/2));

w_innerfront = (v/(2*PI*tyre_radius)*((Rsteer-B/2)/Rrobot));
w_outerfront = (v/(2*PI*tyre_radius)*((Rsteer+B/2)/Rrobot));
w_rear = (v/(2*PI*tyre_radius)*((Rback)/Rrobot));
```

COLLISION CONTROL

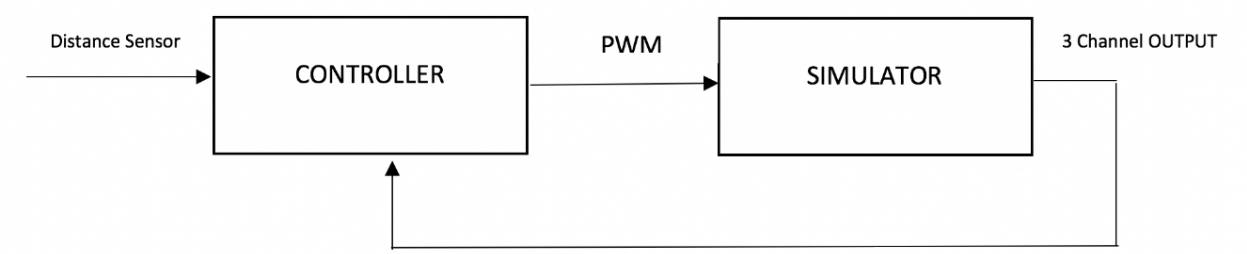
Introduction:

Collision Control Systems or Collision Avoidance Systems are the safety systems in vehicles to control the severity of collisions or try and reduce collisions as much as possible. In their most basic form, these Collision Control Systems control the speed of a car, monitor the distance between the vehicles and surrounding traffic, or provide a warning to the driver if things start getting unsafe.

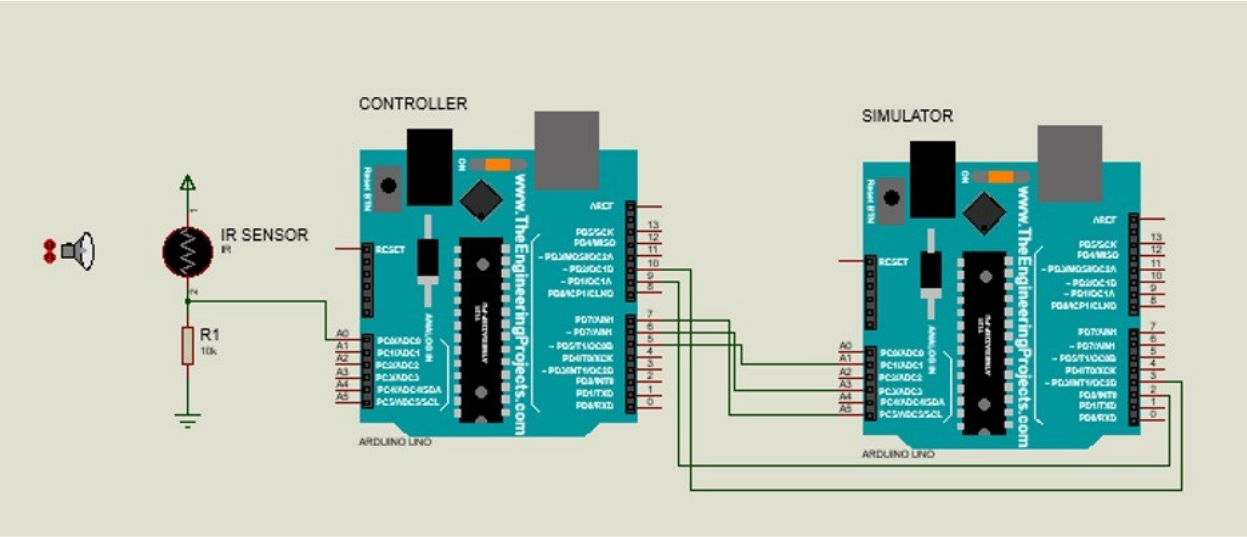
Procedure:

The system is a replication of a Collision Control System, which consists of two parts, the "Simulator" and the "Controller". This system works in such a way that the controller reads values from the distance sensor (Sharp IR in this case) and generates the PWM according to the distance calculated. Simulator then reads the PWM and generates the 3-channel output (analog signal) which is then again read by the controller. The controller then calculates the parameters which in the end governs the front wheel and back wheel velocity.

Block Diagram:



Schematic:



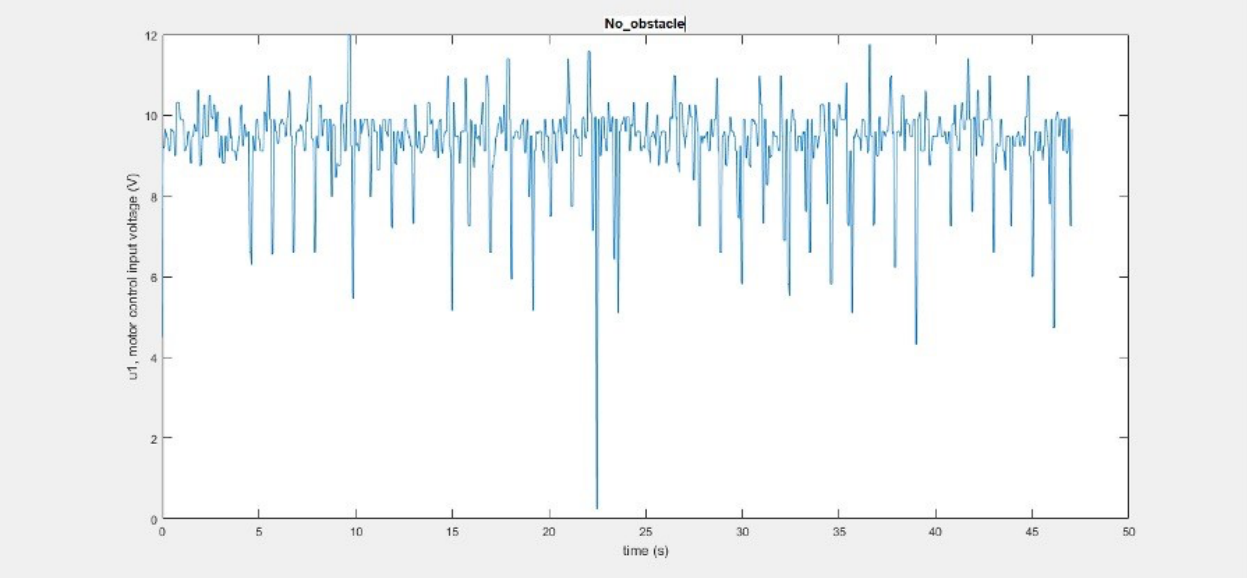
Case 1:

When there is no obstacle nearby, the PWM generated will be almost equal to maximum PWM defined in the program. The front and back wheel velocity will be maximum.

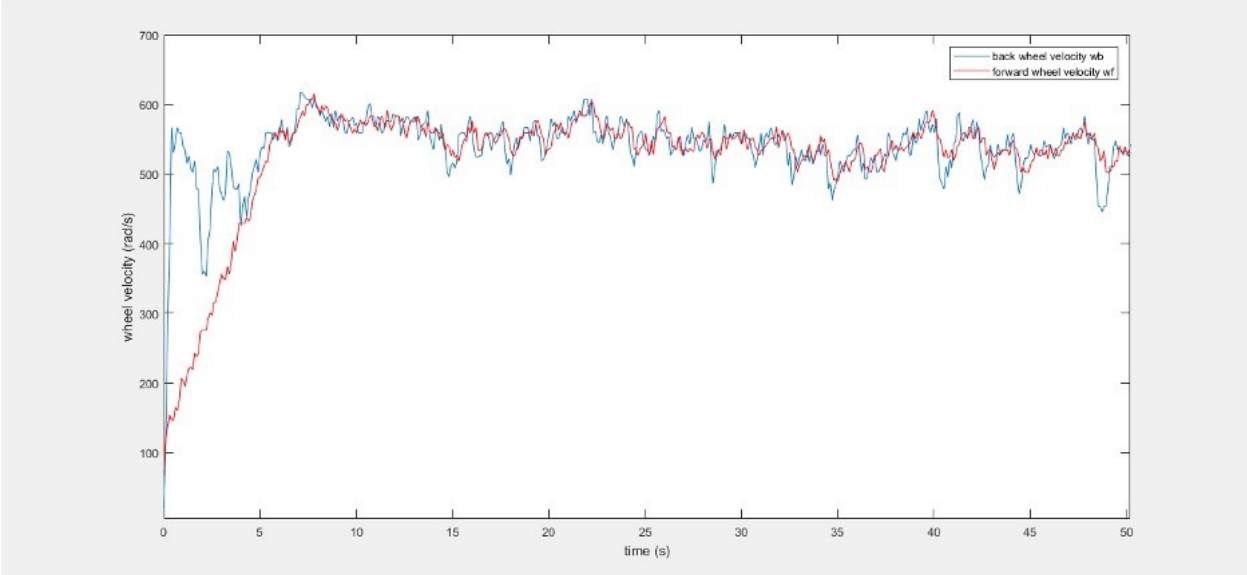
Results:

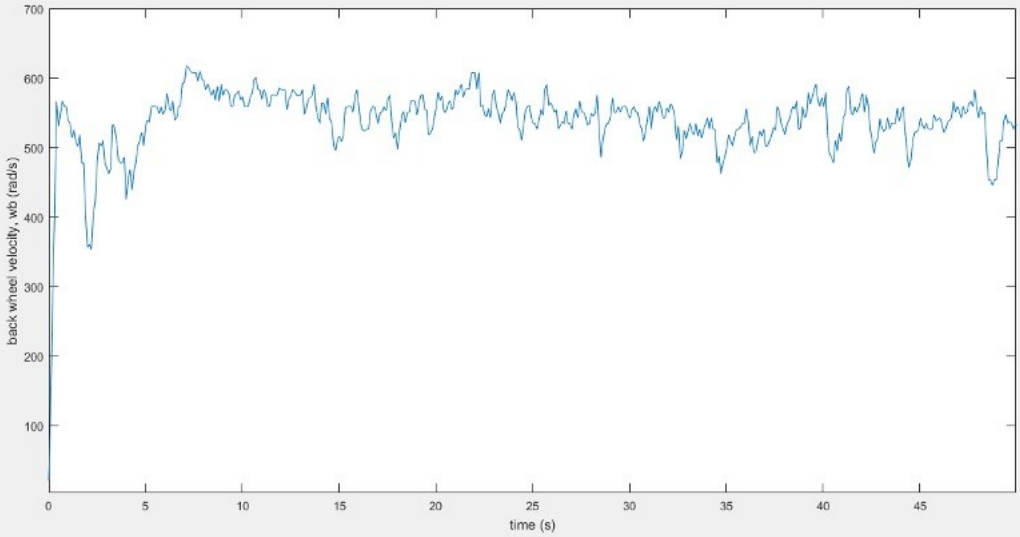
Simulator Graph:

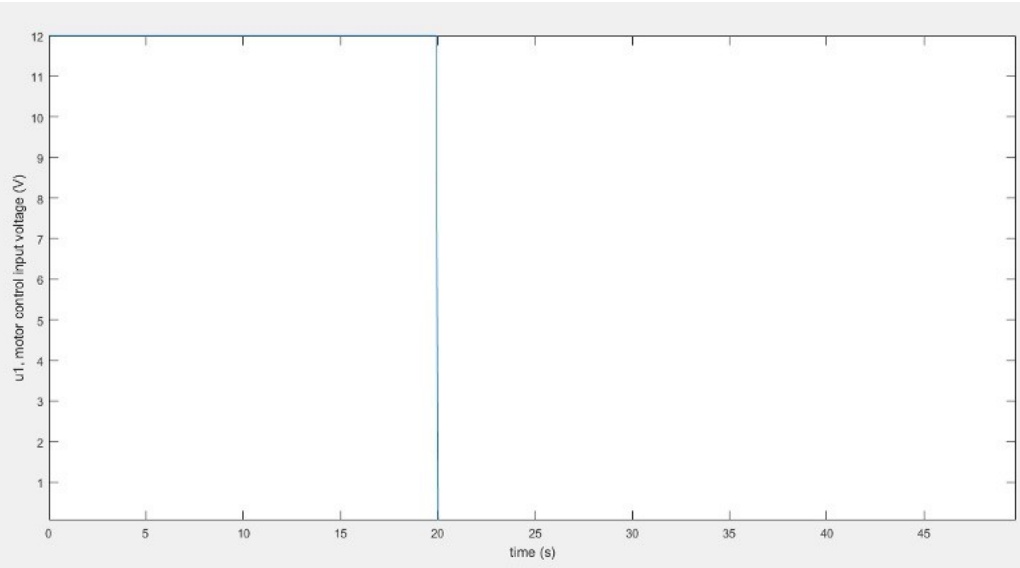
When there is no obstacle the from the simulator almost remains constant (noise can be removed through proper filtration). Hence the speed of the vehicle will be constant.

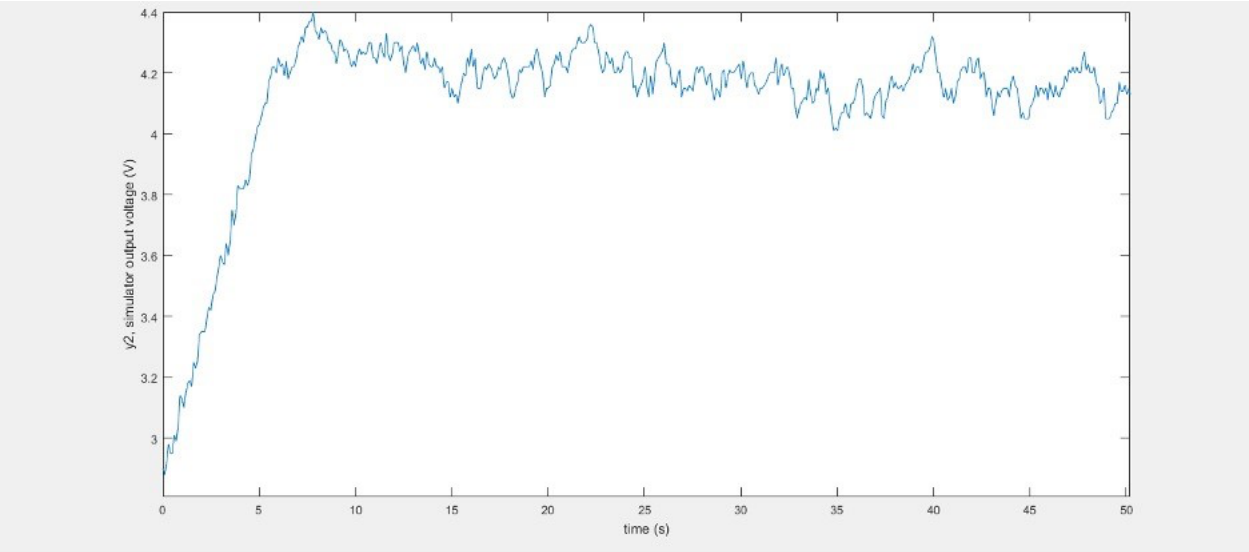


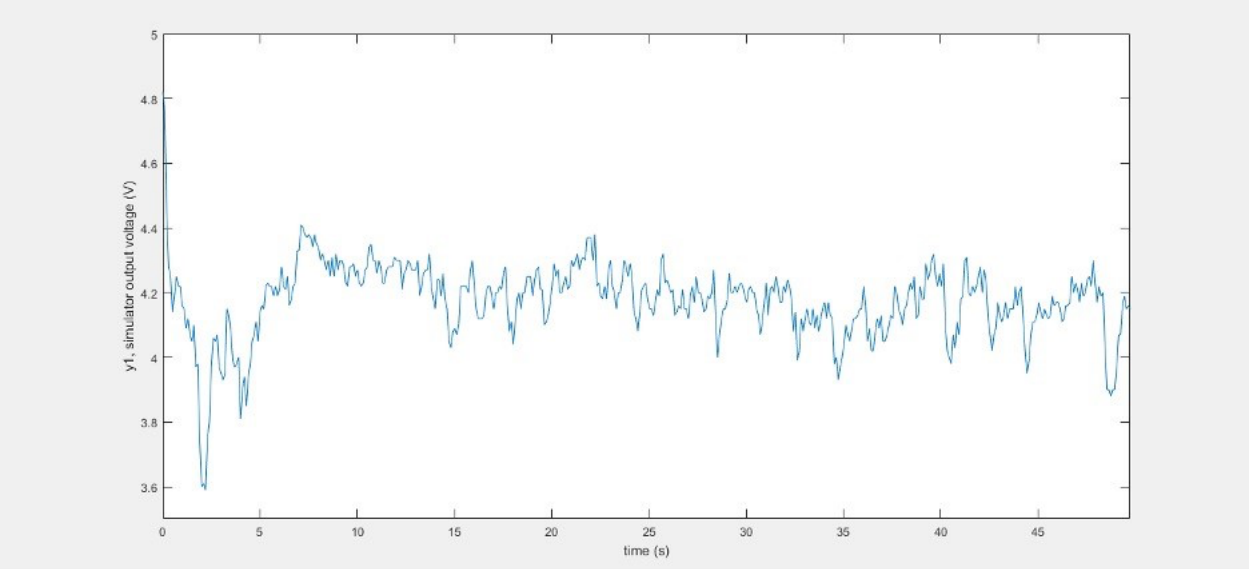
Controller Graphs:




back wheel velocity, wb (rad/s)
time (s)


u1, motor control input voltage (V)
time (s)





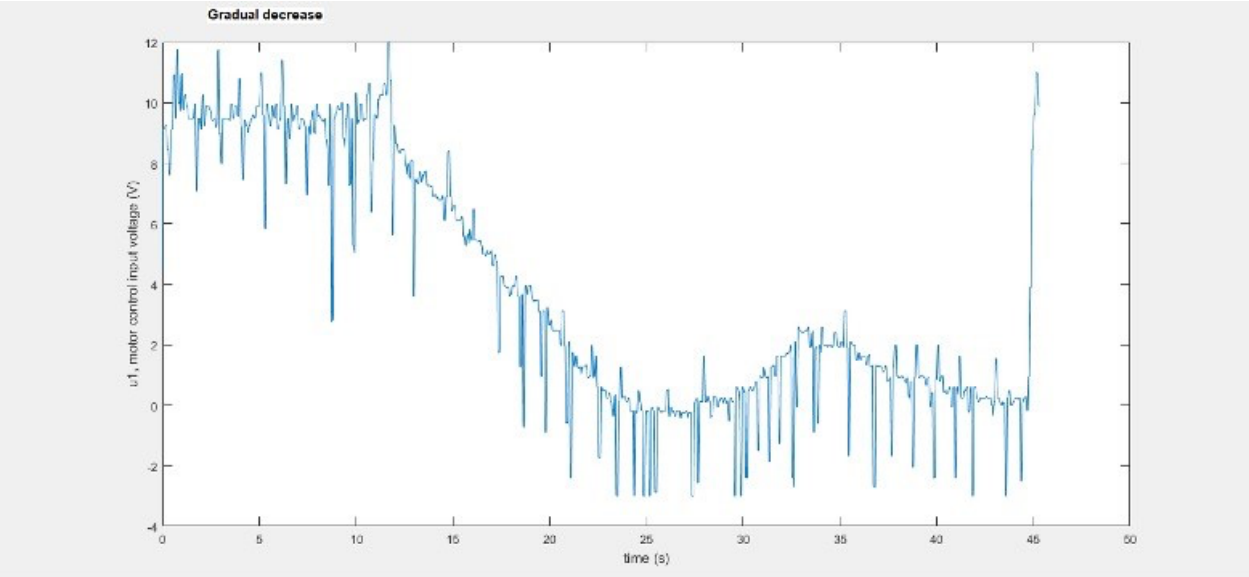
Case 2:

When the vehicle is approaching the object gradually, then according to the distance the voltages and the PWM will decrease gradually.

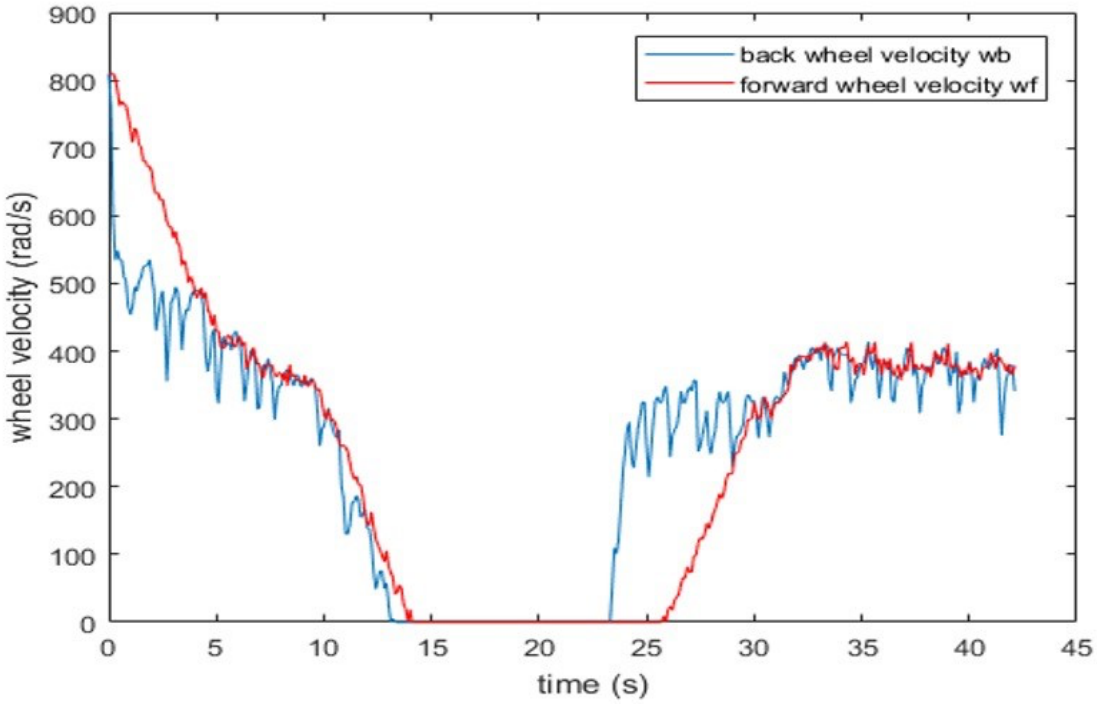
Results:

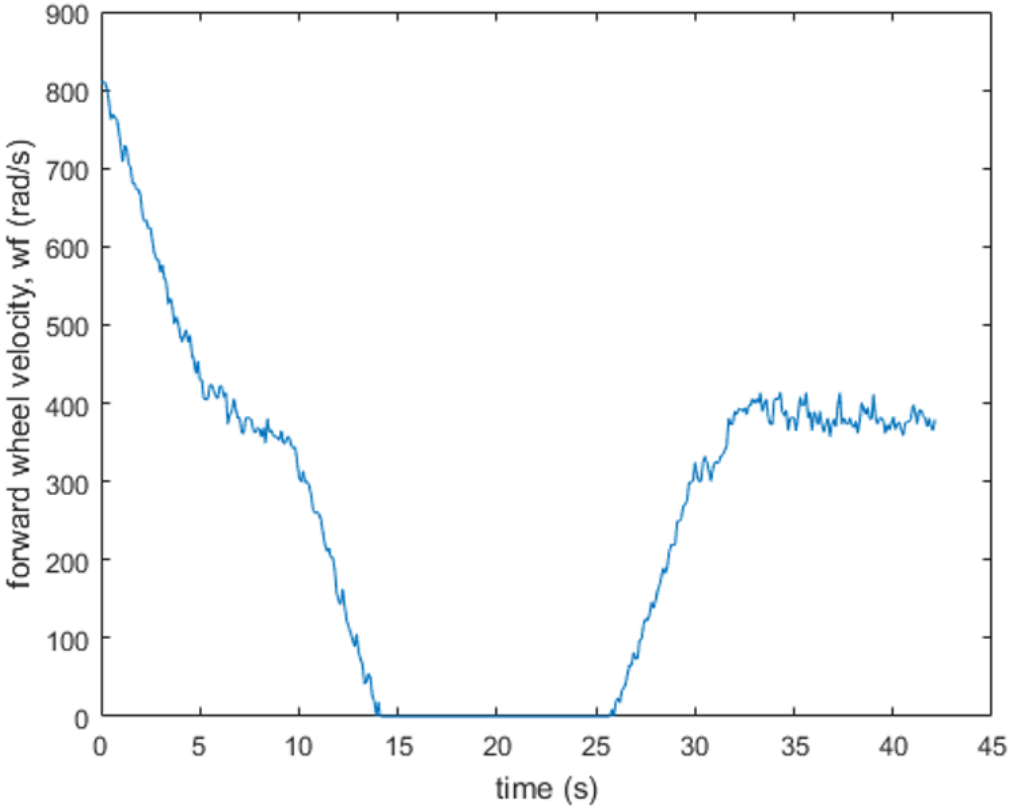
Simulator Graph:

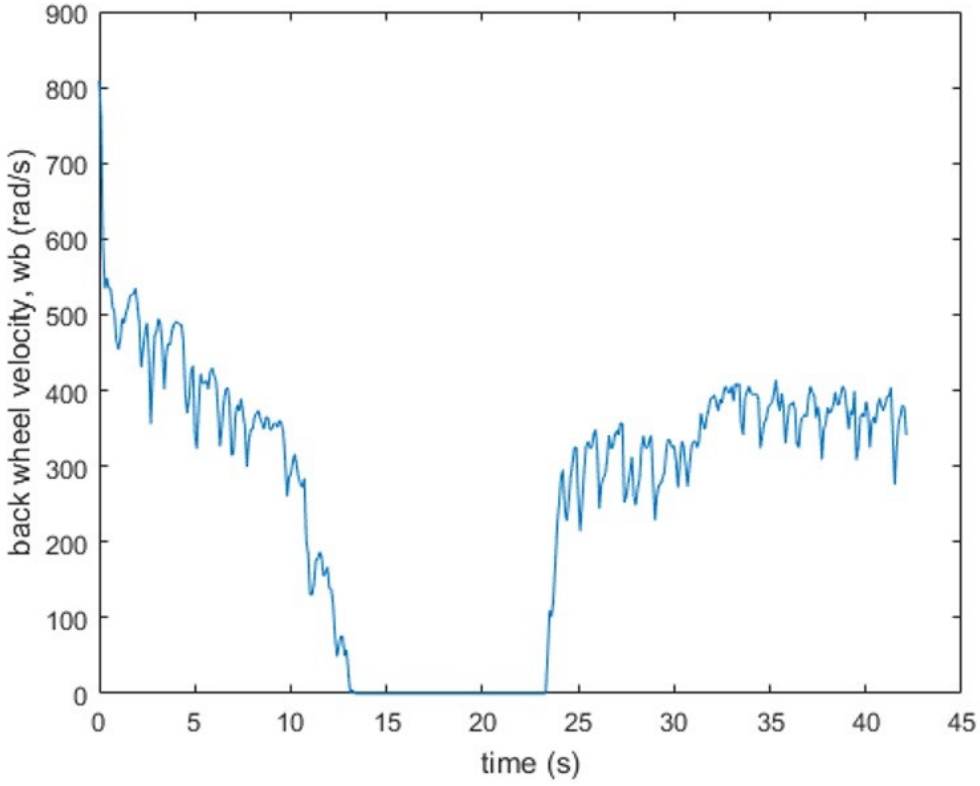
As the trend shows, the voltages decrease gradually as the vehicle approaches the object or obstacle and then rise as the obstacle moves away.

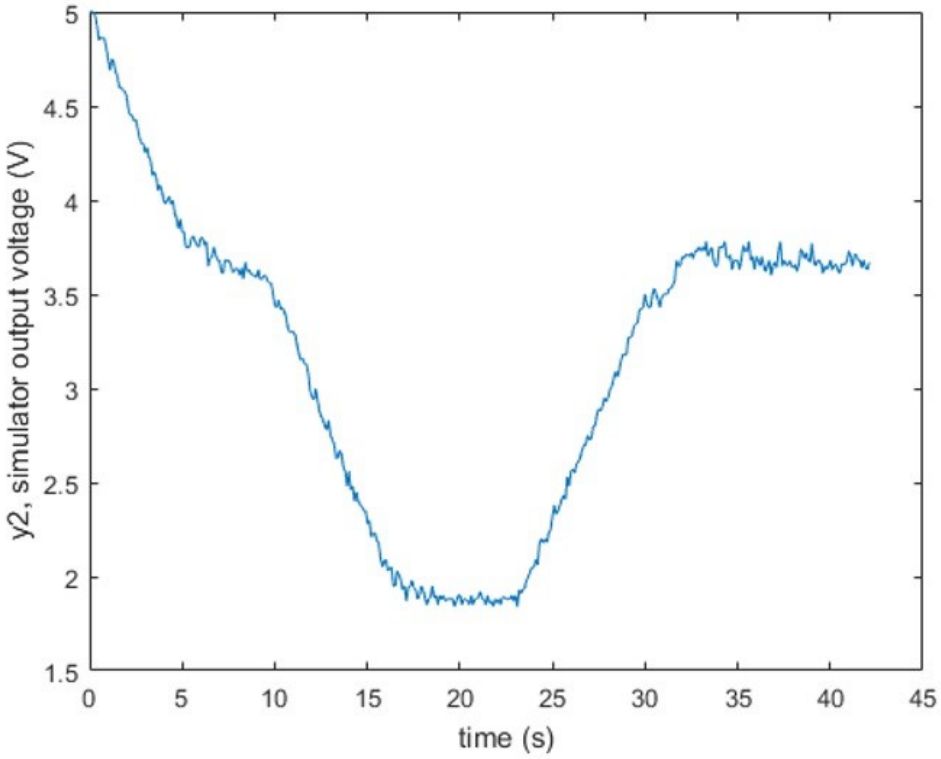


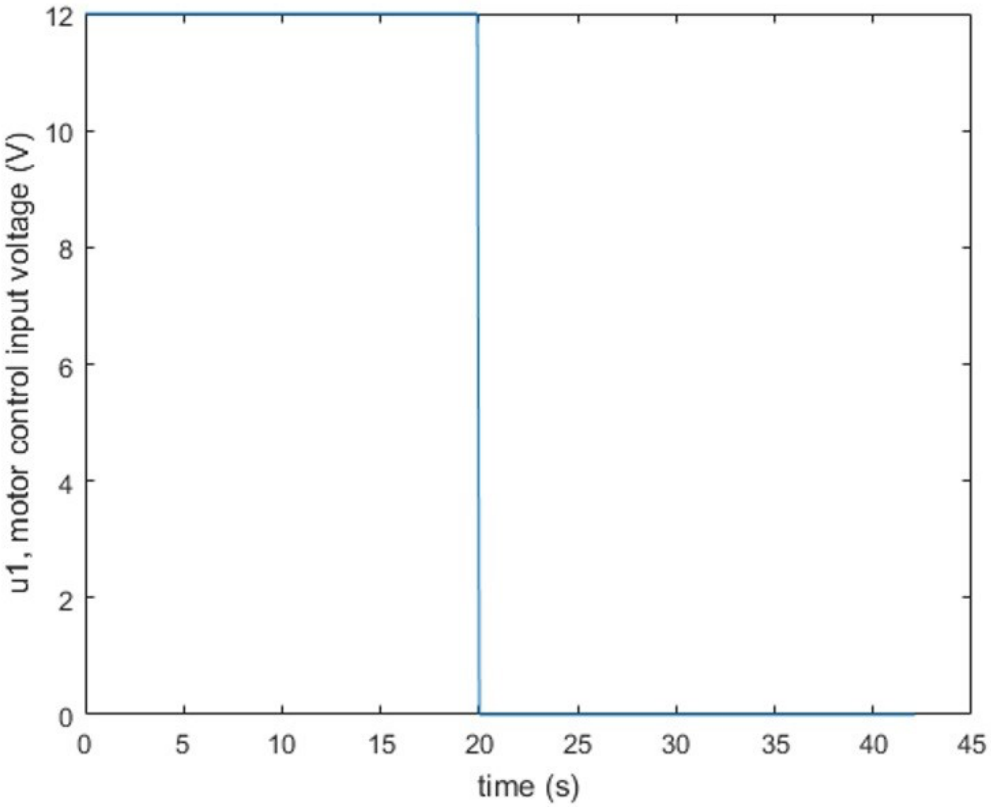
Controller Graphs:

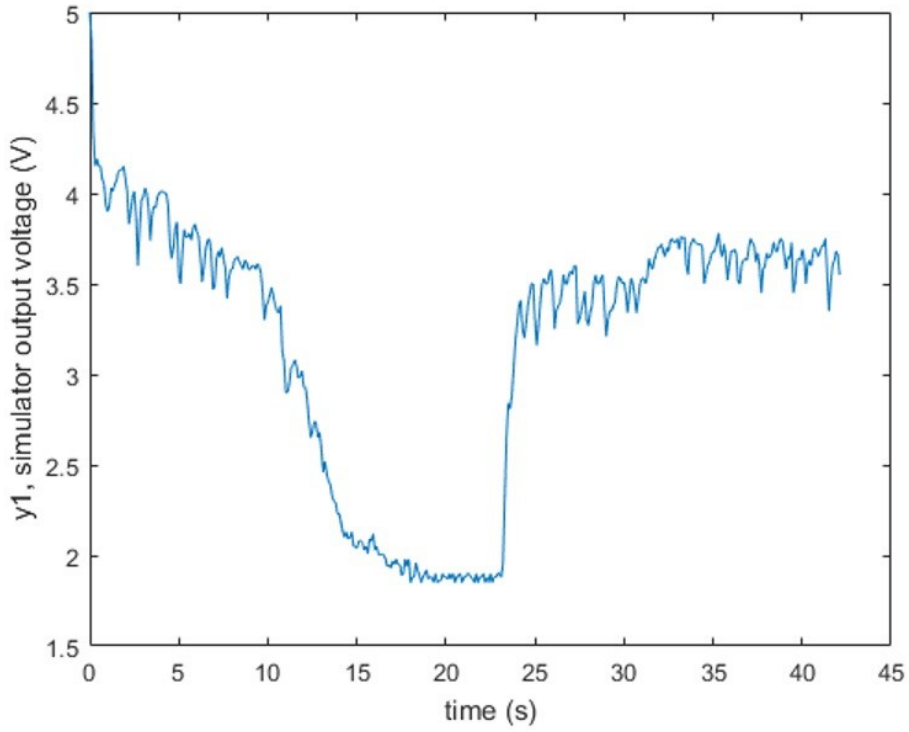



forward wheel velocity, wf (rad/s)
time (s)
0
100
200
300
400
500
600
700
800
900
0
5
10
15
20
25
30
35
40
45


back wheel velocity, wb (rad/s)
time (s)
0
100
200
300
400
500
600
700
800
900
0
5
10
15
20
25
30
35
40
45


y2, simulator output voltage (V)
time (s)
5
4.5
4
3.5
3
2.5
2
1.5
0
5
10
15
20
25
30
35
40
45


u1, motor control input voltage (V)
time (s)
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35
40
45



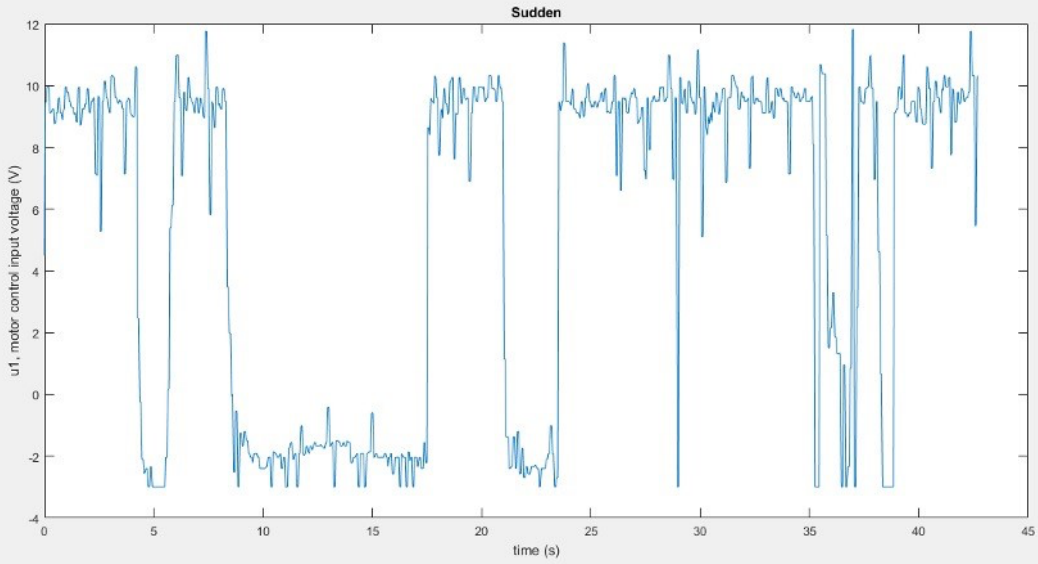
Case 3:

When an obstacle abruptly comes in the way of vehicle, then there will be a sudden drop in the voltages as well as in the PWM. This state resembles the emergency brakes of the system. And this is the most important part of the Collision Control System.

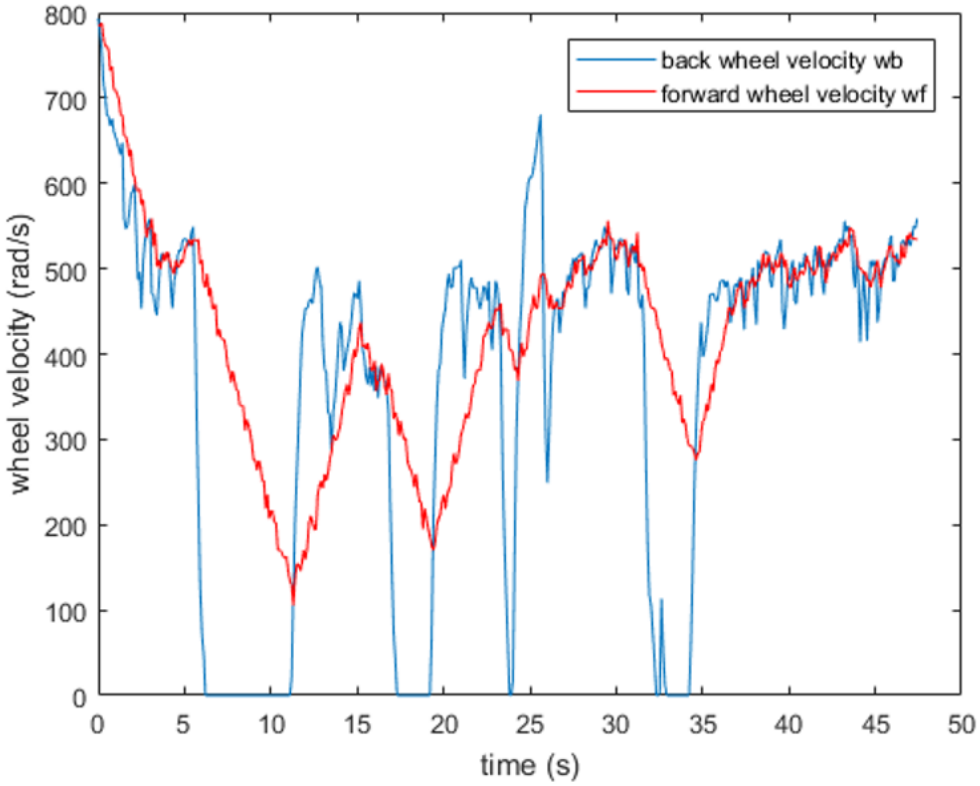
Results:

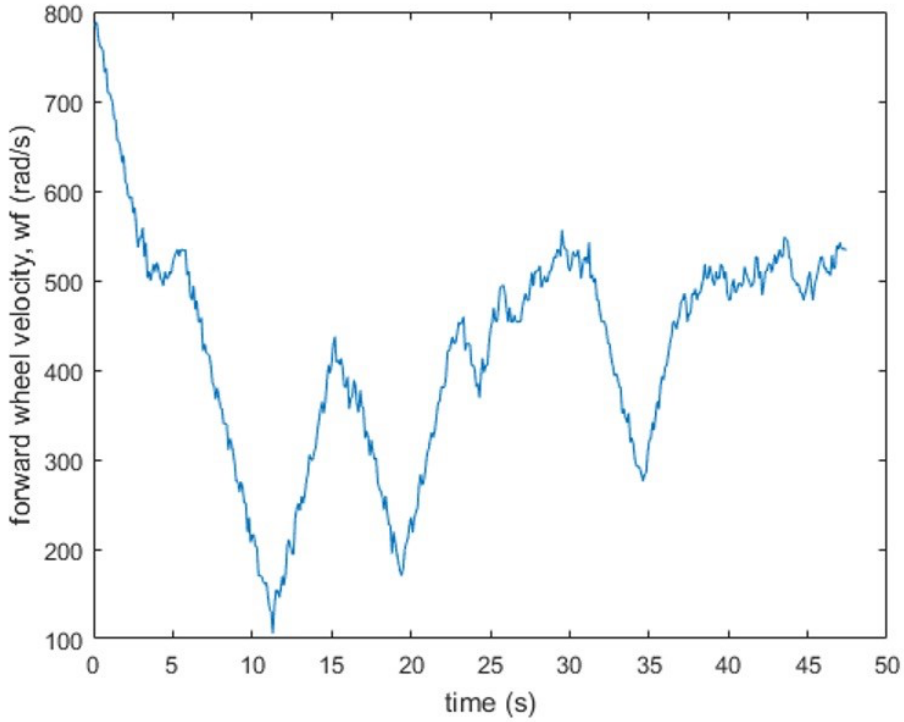
Simulator Graph:

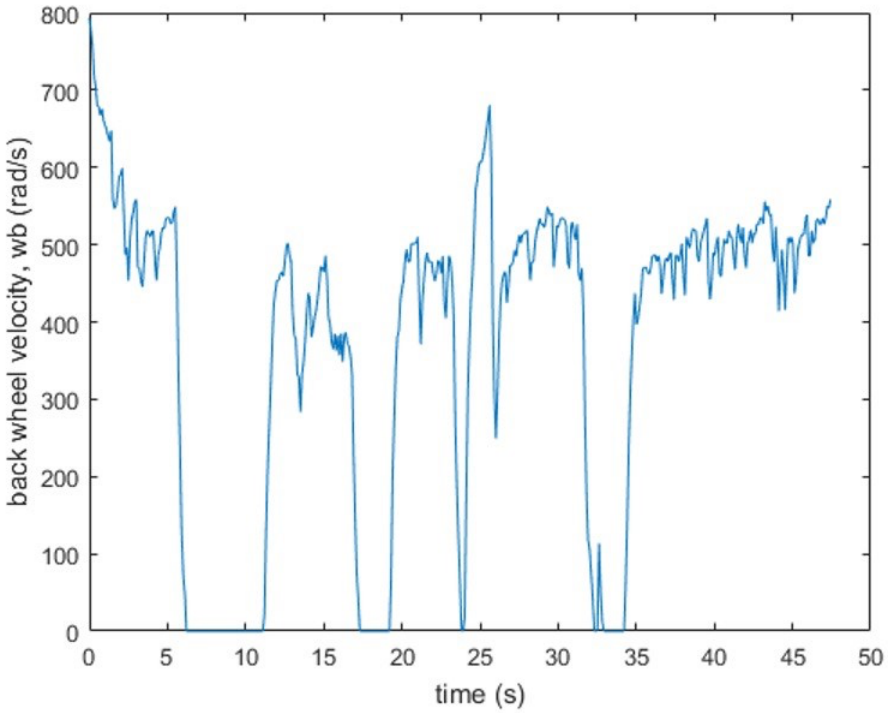
Here the graph shows the sudden drop in the voltages which were fed from the simulator to the controller.

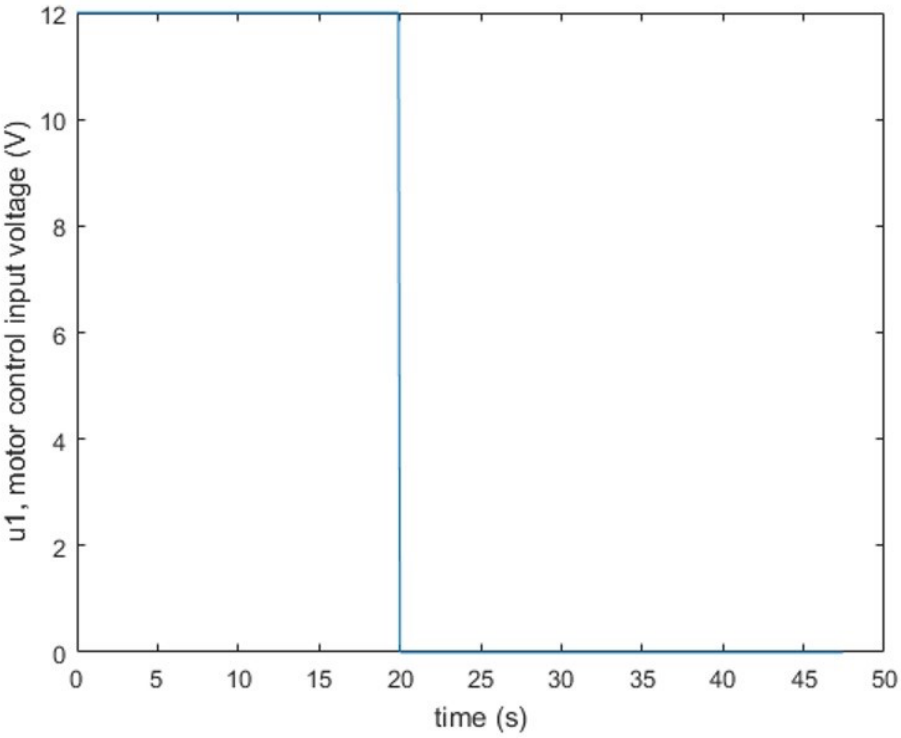


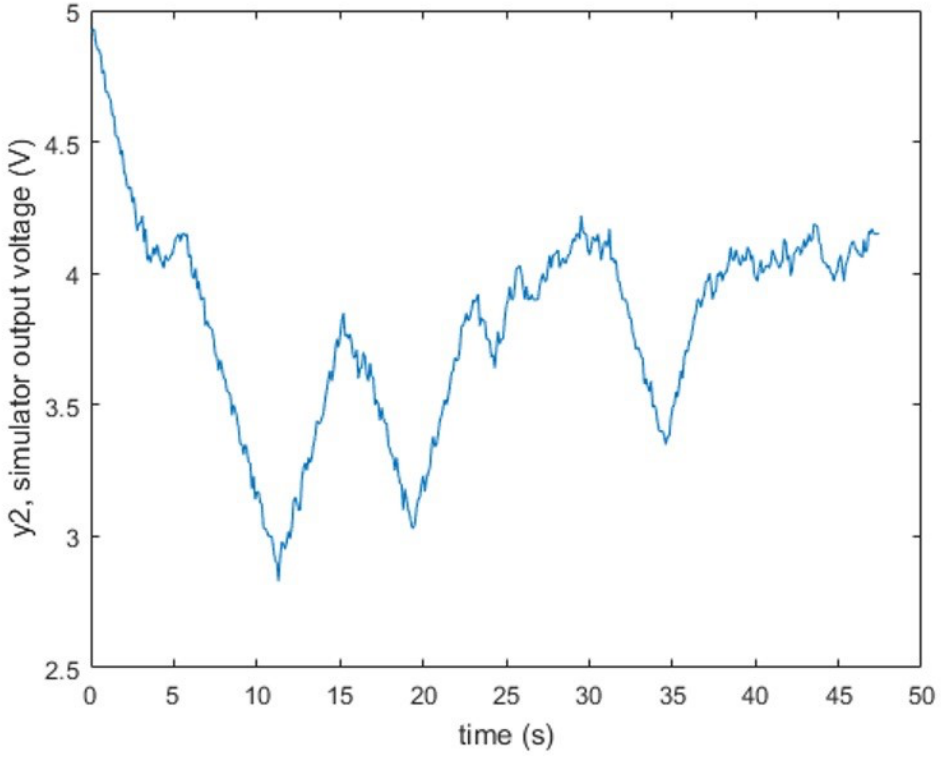
Controller Graphs:

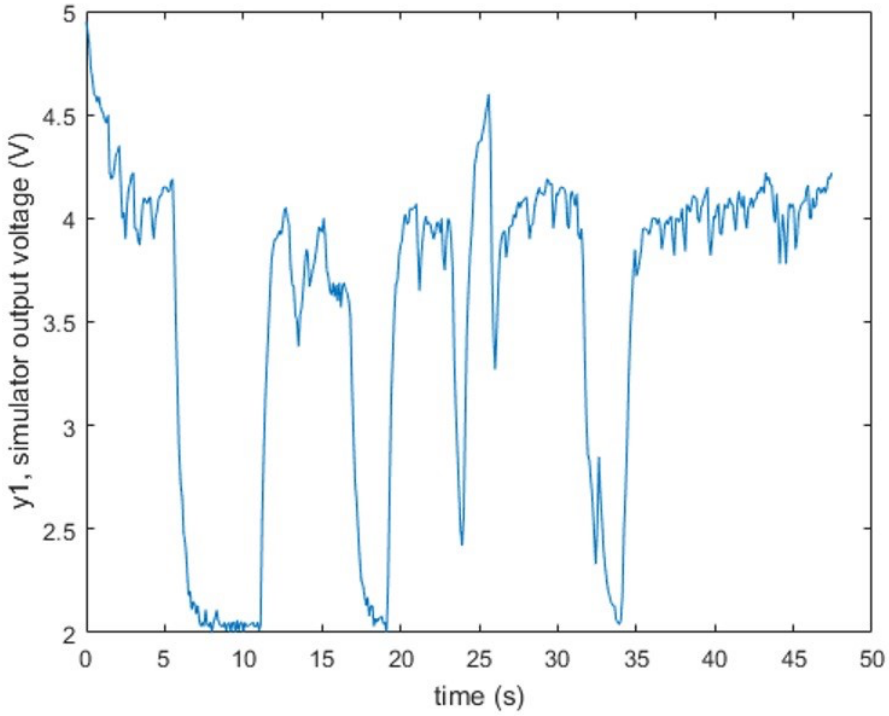





back wheel velocity, wb (rad/s)
time (s)
0
100
200
300
400
500
600
700
800
0
5
10
15
20
25
30
35
40
45
50


u1, motor control input voltage (V)
time (s)
0
2
4
6
8
10
12
0
5
10
15
20
25
30
35
40
45
50


y2, simulator output voltage (V)
time (s)
5
4.5
4
3.5
3
2.5
0
5
10
15
20
25
30
35
40
45
50


y1, simulator output voltage (V)
time (s)
5
4.5
4
3.5
3
2.5
2
0
5
10
15
20
25
30
35
40
45
50

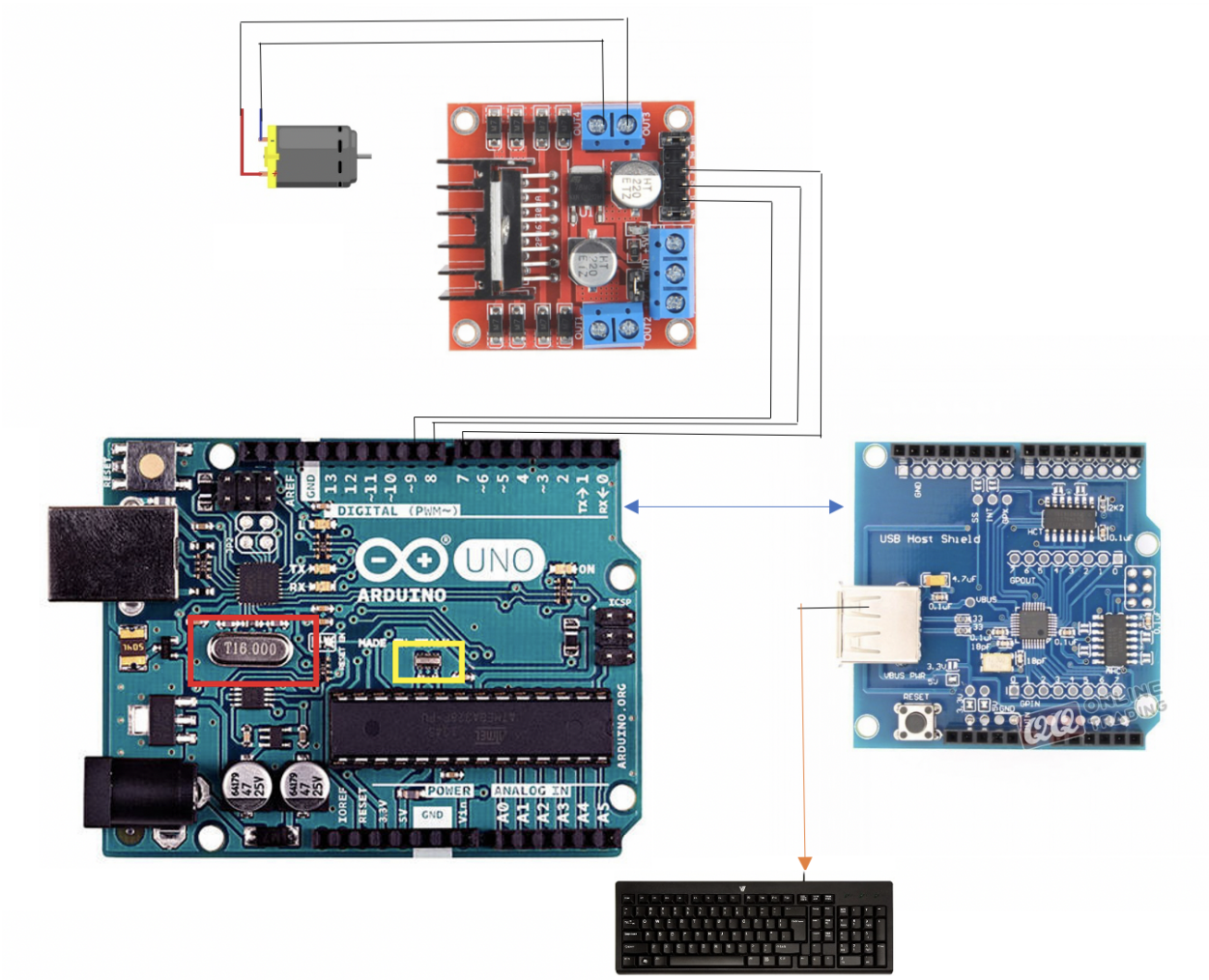
SERIAL COMMUNICATION II

Objective: To control speed of motor by computer keyboard

Introduction:

We will use host shield or pro mini for serial communication between the keyboard and the Arduino. We will read keyboard serially. When we receive bytes on Arduino from the keyboard serially then according to the byte, we will update the register PWM by setting duty cycle of it.

So, when byte of the UP key of the keyboard receives on Arduino which is 38 then we will increase PWM of register by adding duty cycle in previous duty cycle of that register. And when byte of DOWN key of keyboard receives on Arduino which is 40 then we will decrease PWM of register by subtracting duty cycle in previous duty cycle of that register. This is how we can control the speed of our motor.



Arduino Code:

```
const int NMAX=64;
int pwmA=100;
int pwmB=50;
int pwml;
int in_data;
int n,len;
static char buffer[NMAX];
void setup ()
  {
  Serial.begin (115200);
  Serial.println ();

  DDRD|=1<<PD7;                                          //////////////setting pin7 as output for motor driver ////////////////////
  DDRB|=1<<PB0;                                          ///////////////setting pin8 as output for motor driver ///////////////////////
  DDRB|=1<<PB1;                                          ////////////// setting pin9 as output for pwm ////////////////////////////
  DDRB|=1<<PB2;

  PORTD|=(1<<PD7);

  TCCR2A = _BV(COM2A1) | _BV(COM2B1) | _BV(WGM20);       ////////////////////setting pwm to phase correct mode ///////////////////////
  TCCR2B = _BV(CS22);                                    ////////////////////setting prescaller/////////////////////////////////////



  }  // end of setup

void loop ()
  {
   OCR2A =pwmA;                                           ////////////////////// settting duty cycle////////////////////////////////


   OCR2B = pwmB;

if (Serial.available()>0){                               ////////////////reading data from keyboard//////////////////////
   len=2;
   n = Serial.readBytes(buffer,len);
}

if (n==38){                                             ////////////////////////setting motor speed for increase if up button is pressed/////////////////
  pwmA=pwmA+10;
  pwmB=pwmB+10;
}

if (n==40){                                             ////////////////////////setting motor speed for increase if up button is pressed/////////////////
  pwmA=pwmA-10;
  pwmB=pwmB-10;
}

if (pwmA>250 || pwmB>250){
  pwmA=250;
  pwmB=250;
}

}
```

CONTRIBUTIONS

Member	List of Contributions
Faisal Bari	Steering control, braking control, debugging of serial communication (the receiving part, Arduino to computer)
Cyril Girgis	Speed control, ADC programming, Code integration between people, Set Input functions
Rachel Haighton	Register pwm, register version of micros(), traction & braking control, overall PID selector function, PID simulation in C++
Khaled Abdelkader	Collision control, Serial communication and register pwm.