Department of Mechanical and Industrial Engineering

MECH 471

Blue Death MK I

Project Report

Written by:

Gurvir Grewal 40062915

Muneeb Rehman 40058834

Alexandro Tomassini 40029074

Zihuan Zhang 40063079

Submitted to: Dr. Brandon Gordon

Concordia University

Montreal, QC, Canada

05/10/2020

Responsibilities:

Muneeb Rehman [Page 3-9.75]

- Designed, built, and programmed the real, physical, robot.
- Established shared communication such that the car-simulator controls the HIL (and sim_step), receives data from it, and outputs the data in the 3D graphics simulator.
- Made the speed PID controller that inputs forward desired velocity.
- Made the traction PID controller that keeps the slip ratio to 0.2 while accelerating.
- Made the Vision feature to detect the car's position and highlight the race track.
- Tested PIDs under asphalt, wet, snow and ice road surface conditions. Designed safety measures.
- Made the automated steering PID controller using the Virtual Force Field method.
- Built the neural network feature (made it for 472 but integrated for this project as well, yet to be tested).
- Made the RC filter circuit for the HIL.
- Converted analogWrite(…) into timer register level code for the y1,y2,y3 outputs on the HIL simulator.
- Converted analogRead(…) into ADC register level code to signals from the simulator, no polling.
- Converted servoWrite(…) into register level signal generators for the controller.

Alexandro Tomassini [Page 9.75-10]

- Designed an early version launch controller, traction controller and brake controller.
- Tested and simulated different PIDs in Matlab.

Zihuan Zhang [Page 11]

- Designed and tested an initial version of the PID speed controller in the car simulator and on Arduino
- Test the controller under different condition using MATLAB.

Gurvir Grewal [Page 12]

- Designed and coded a steering controller which utilizes vector analysis and PID to compare its position along the track's spline and remain at its center.
- Tested the controller under different condition using MATLAB.

The Real Car

The car was designed and built well before the project requirements were announced, everything is organized as classes and objects. The components were tested, and it is possible to control the real car via Bluetooth Serial communication using the keyboard keys. Using the Teensy 3.5, communication was surprisingly fast, and it was easier to read buffer memory because the data type sizes were the same as the one on the PC. The baud rate on the HC-05 Bluetooth module was also increased to its maximum limit using AT Commands. Unfortunately, since the actual build was no longer a requirement, the real bot was discontinued.

- Components included:
  - MPU6050 – Gyroscope Accelerometer
  - Teensy 3.5
  - HC-05 with maximum possible baud rate
  - RS 550 motor
  - Hall effect sensor (rotary encoder)
  - L298 Motor Driver – 7A
  - 4x 3.7 Lithium Ion batteries

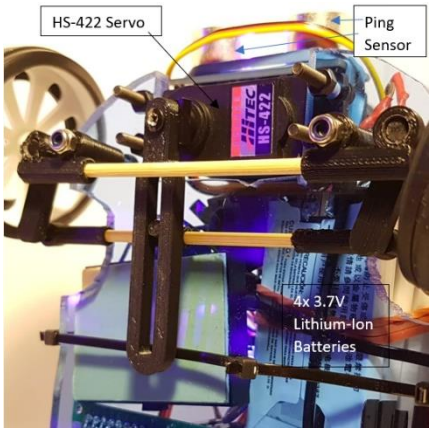

Figure 1: Steering Mechanism

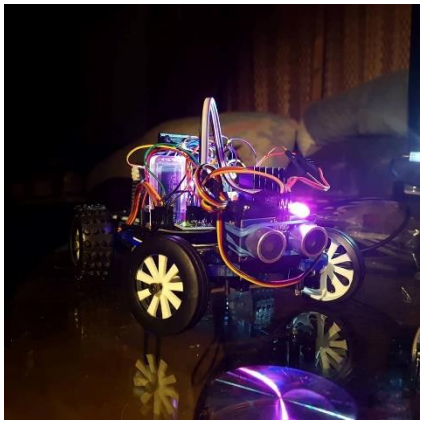

Figure 2: Blue Death Mk I

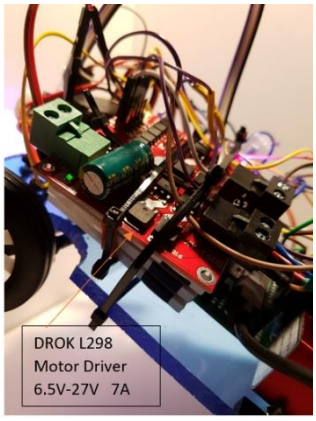

Figure 3: L298 Speed Controller 7 A

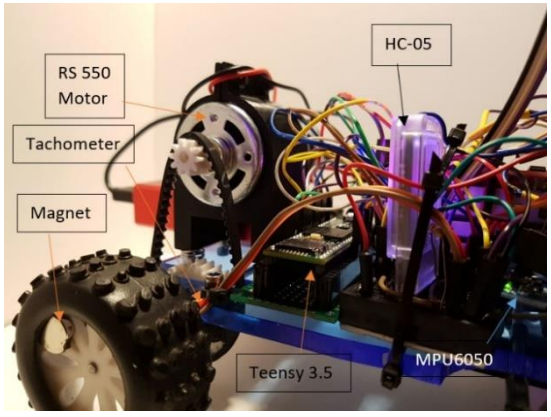

Figure 4: Side View

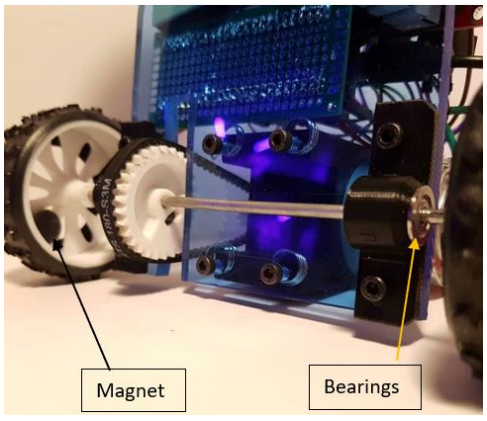

Figure 5: Bottom Rear View

The HIL

Adding 3x 0.1 µF capacitor and 3x 220kΩ resistors was a good combination to filter out noise from the simulator outputs, setup like the following pictures:

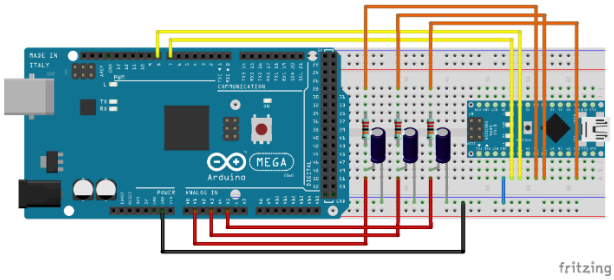

Figure 6: HIL Diagram

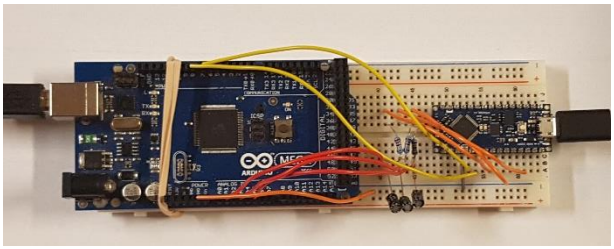

Figure 7: HIL Real View

NOTE: Simulator pin 10 used instead of pin 11, Arduino Nano does not have PWM on pin 11

As for the converting analogWrite, servoWrite and analogRead into register level programming code, the controller is reading when interrupted by the ISR(ADC_vect), and ISR(TIMER1_COMPA_vect) for the servo pulse generator. As for the simulator, since I was using an Arduino Nano Every (which has a different chip than the Arduino Nano), the PWM's are controlled by changing the Compare Register A and B values. The ADC readings are very fast since its designed to not have polling. Here is a side by side graph of the HIL with N=15 samplings and the plot from the 3D car simulator, without PID:

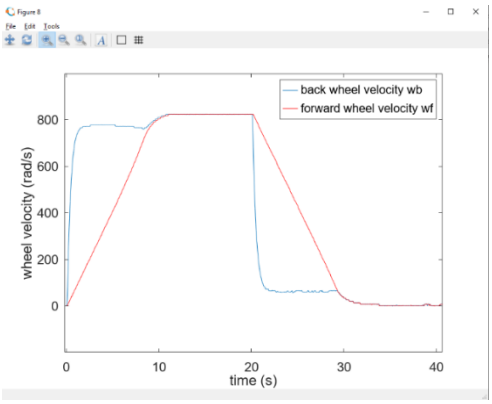

Figure 8: Wheel speed data from HIL

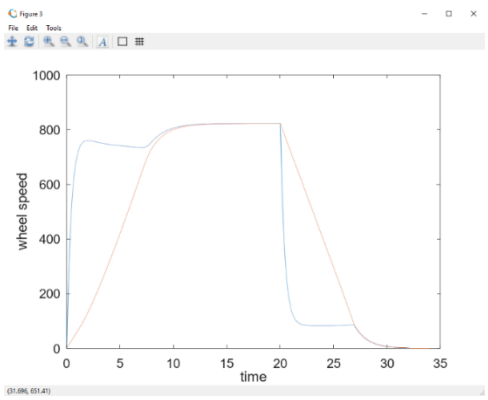

Figure 9: Wheel speed data from 3D simulator

```
void ADC_setup()
{
  ADMUX = 0;
  //ADMUX |= BIT(MUX0) | BIT(MUX2);
  ADMUX |= BIT(MUX0);
  ADMUX |= BIT(REFS0);

  // set ADC control and status register A
  ADCSRA = 0;
  ADCSRA |= BIT(ADEN); // ADC enable
  ADCSRA |= BIT(ADIE); // ADC interrupt enable

  ADCSRA |= BIT(ADPS1) | BIT(ADPS2); // 64 prescaler
  // this gives a conversion time of 60 microseconds


  ADCSRA |= BIT(ADSC); // start ADC conversion
}
```

Figure 10: ADC Setup for Controller

```
void timer1_setup()
{
  TCCR1A = 0;
  TCCR1B = 0;
  OCR1A = 5600;
  TIFR1 |= BIT(OCF1A);
  TIMSK1 = BIT(OCIE1A);
  TCCR1B |= BIT(WGM12);
  TCCR1B |= BIT(CS11) | BIT(CS10); //64 pre
  //TCCR1B |= BIT(CS10);  //no prescaler
  //TCCR1B |= BIT(CS11);
  TCNT1 = 0;
}
```

Figure 11: Timer1 Setup for 1.500-2.500ms per 20ms

3D Graphics – HIL Shared Memory

As part of the optional requirement, I wanted to simulate what was happening inside the HIL in real-time on the 3D graphics simulator. The inputs are saved inside an input.bin, which is read by the Serial program and pushed to the HIL or sim_step. The back and front wheel speeds are printed on the Serial monitor, character by character. The Serial program reads each character until it meets an "\n" break, the strings are separated where commas are located, and each section is converted into type doubles, stored in an array, then a binary file, and finally read by the 3D sim. The real-time simulation feature was successful, and the 3D sim can act as the controller while the HIL is running, the 3D sim can be used to illustrate the car in real-time as the HIL is running.

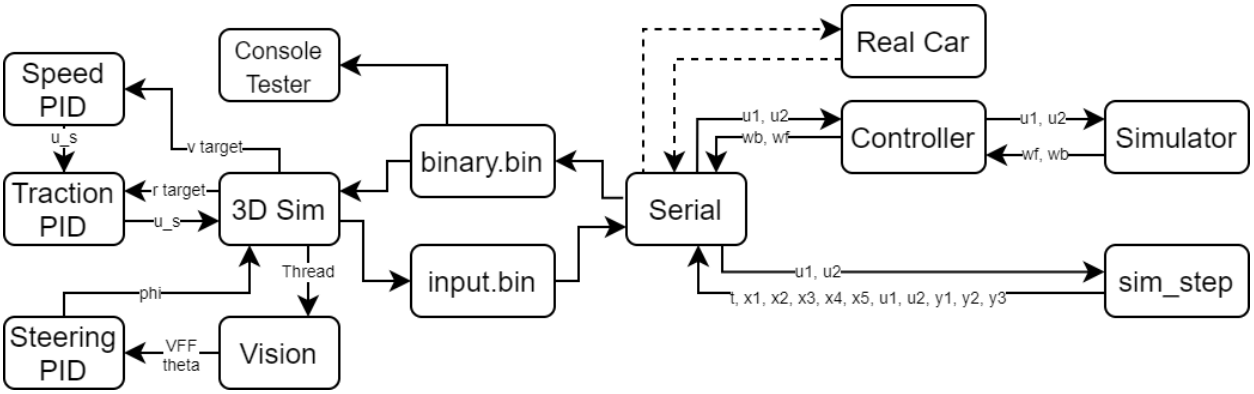

Figure 12: Real-Time HIL/Car_Sim Control

Speed PID

Instead of controlling the input voltage, the speed PID is designed to take the desired velocity of the user and output the voltage. Tuning the PID was proven difficult, since there would either be overshoots and voltage spikes if the P were too large, and a very high steady state error if the P were too small. Here are the results from the PID when the desired velocity is 20 m/s and a case where the target velocity steps down from 10 m/s to 5 m/s using the car_simulator.

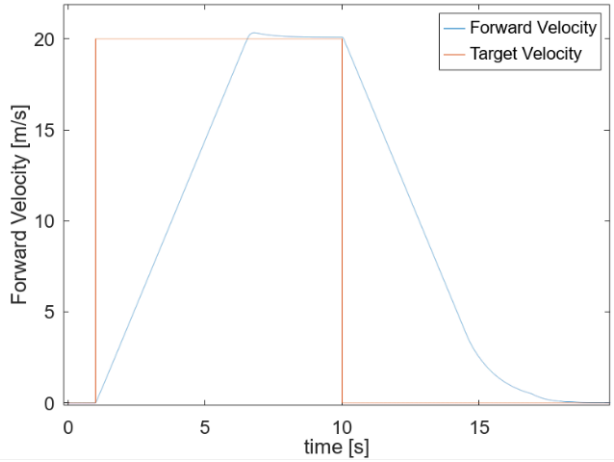

Figure 13: Single target velocity

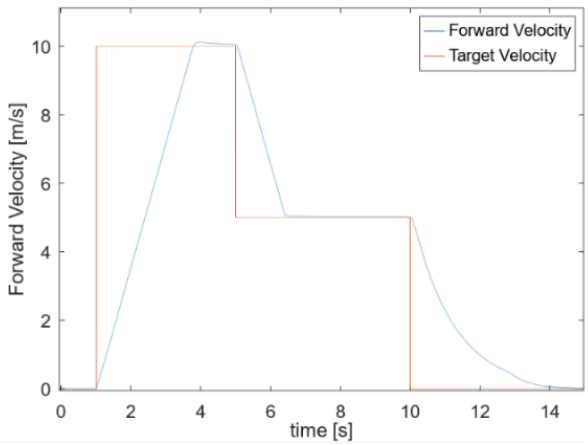

Figure 14: Step Down Target Velocity

Traction & Brake PID

The traction and Brake PID controller were designed to keep the slip ratio at [0.2] when accelerating and [-0.2] when decelerating. Both controllers can only work well after the speed controller, since the robot needs to identify whether it is accelerating, decelerating, or cruising to process which PID to run. The PIDs was tested for two cases on asphalt for acceleration and braking, as shown below:

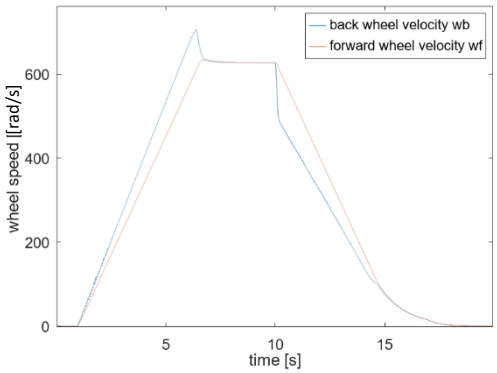

Figure 15: Wheel Velocity [wb/wf]

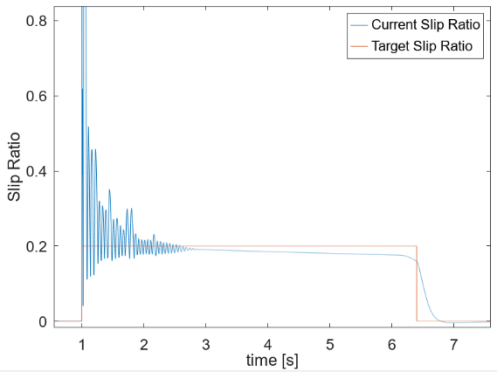

Figure 16: Slip Ratio Accelerating

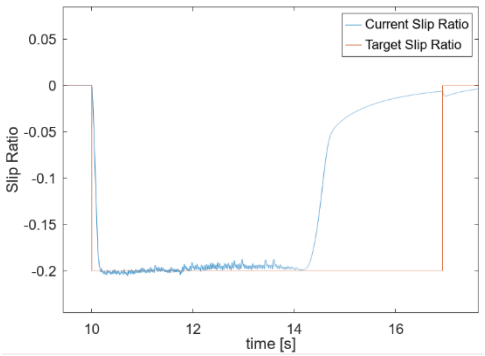

Figure 17: Slip Ratio Braking

Here are four cases with different road conditions at 20 m/s target velocity:

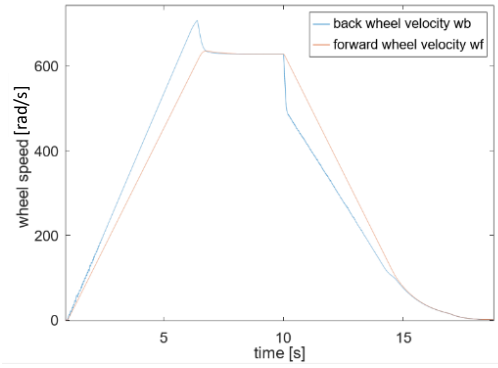

Figure 18: Asphalt Surface

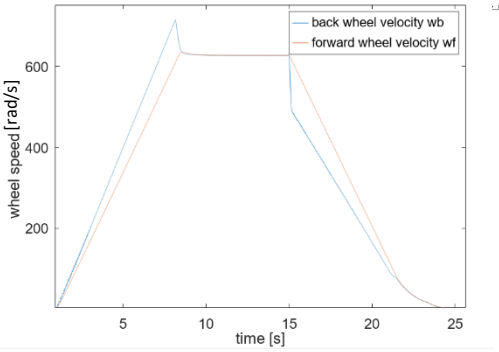

Figure 19: Wet Surface

Surface Condition	Launch Duration [sec]	Brake Duration [sec]
Asphalt	6.45	8.10
Wet	8.40	9.80
Snow	21.82	21.40
Ice	67.68	62.20

Table 1 - Acceleration/Braking Duration

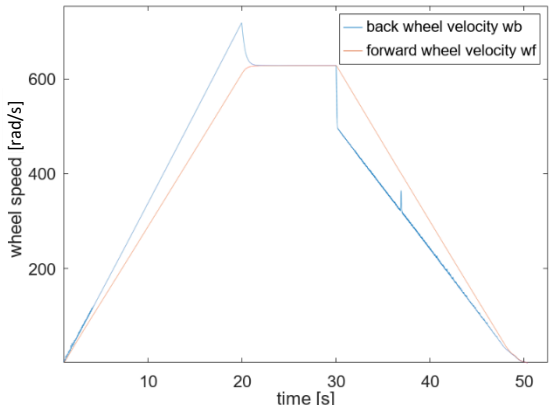

Figure 20: Snow Surface

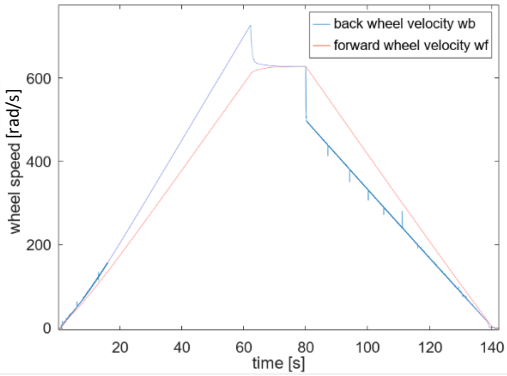

Figure 21: Ice Surface

Vision & Steering PID

The VFF version of the steering control takes the rgb image from Image View and does its processing on a separate thread, and the target theta is only updated when that thread is complete. Otherwise, the simulation time would slow down considerably as it would need to wait for VFF system to complete numerous loops. The car location on the Image_View window is first selected based on an HSV range specific to the car. Then a thresholding process converts everything into either 0 or 255, the label of the car is selected, and finally the centroid of that automatically selected label is tracked.

The VFF method, short for Virtual Force Field, is an exemplary concept widely used for autonomous vehicle control. Dozens of lines, or series of points, are drawn as vectors, but only the front side of the car is masked, while the back side has a hidden vector constant to counter those front vectors.

The resultant vector is the VFF_theta, the target theta that the steering needs to adjust to. Steering would have been impossible without the PID, mainly because the maximum allowable steering range causes the theta to overshoot easily. Tuning all three components of the PID, the steering adjusts itself quite well, even at the maximum velocity. Additionally, the car can avoid any obstacle in its course, which would have been impossible with the spline cross product distancing method.

To enable the Vision system and VFF control, the VFF_Feature switch must be turned on.

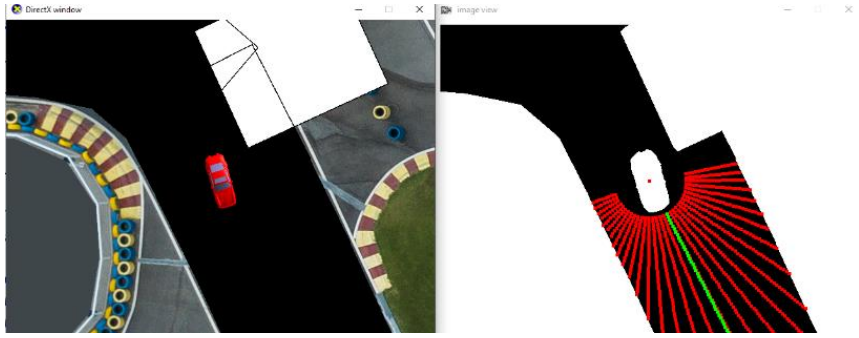

Figure 22: Car Simulator and Vision with VFF enabled

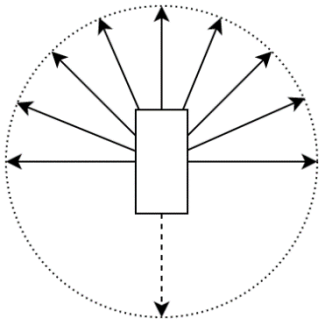

Figure 23: VFF Concept

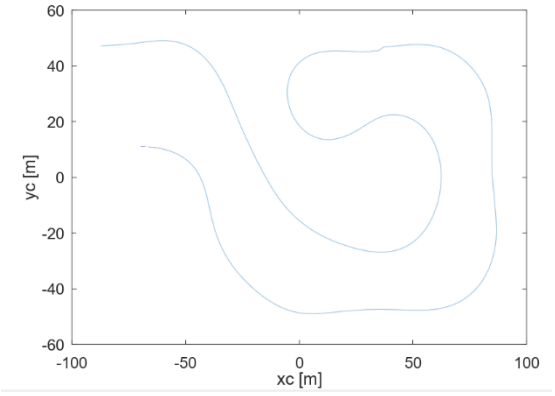

Figure 24: Racing with VFF enabled

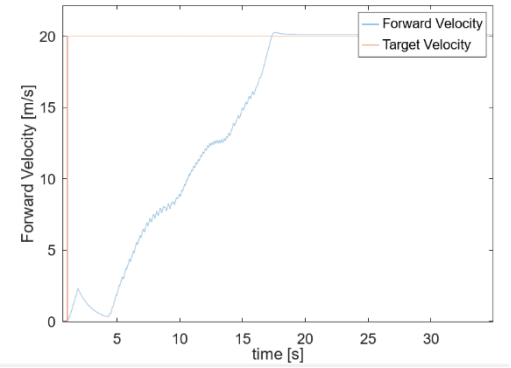

Figure 25: Car velocity with 20 m/s target

HIL & PID Results

Once the Speed, Traction and Braking PIDs were tested on the car simulator, they were transferred to HIL, with registry level ADC readings at N=15 sampling and timer1 servo writes on the controller and register level PWM outputs from the simulator, combined with RC filtration.
Here are the results with the simulator running on asphalt surface conditions:

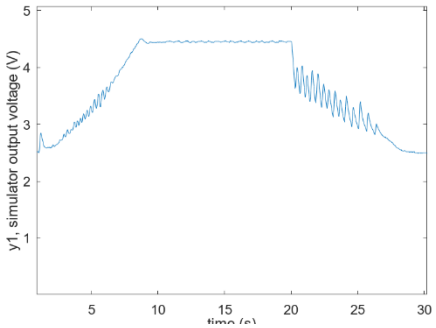

Figure 26: Y1 Output Readings

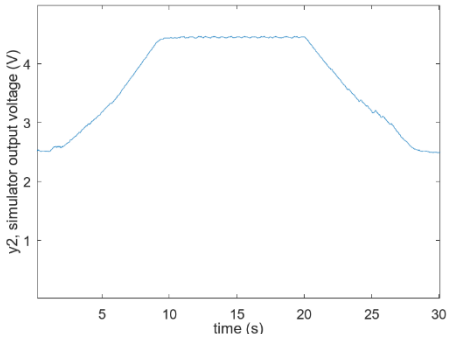

Figure 27: Y2 Output Readings

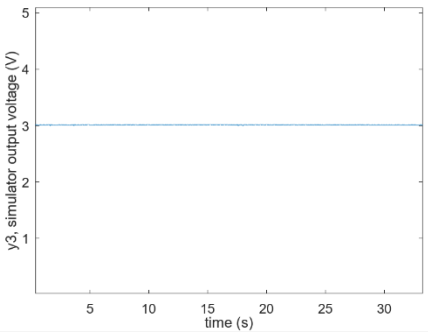

Figure 28: Y3 Output Readings

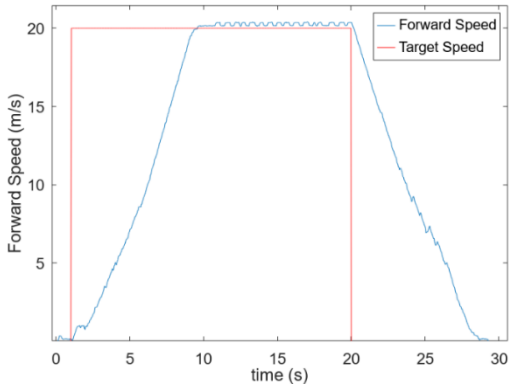

Figure 29: Forward Velocity with 20 m/s target

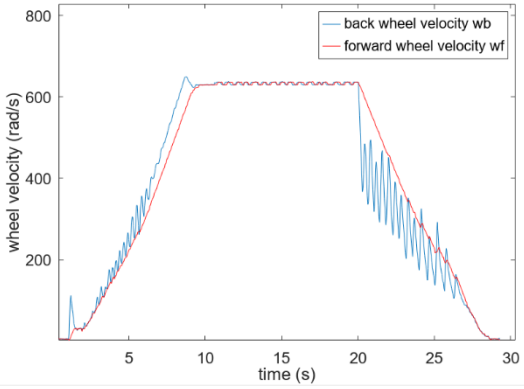

Figure 30: Wheel Velocity for wb & wf

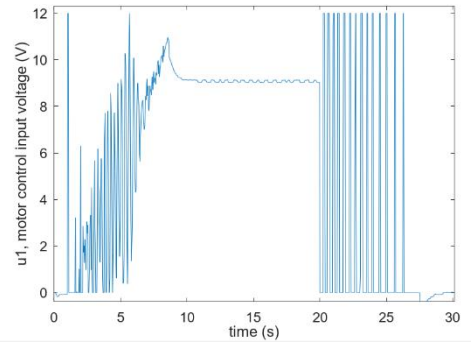

Figure 31: u1 Output Voltage

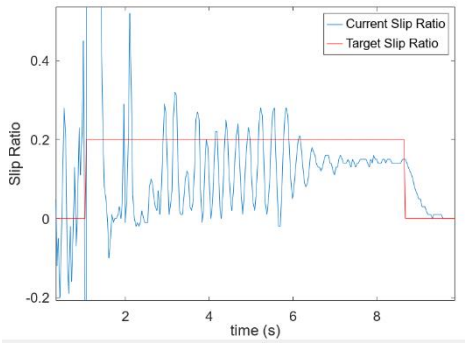

Figure 32: Slip Ratio at Acceleration

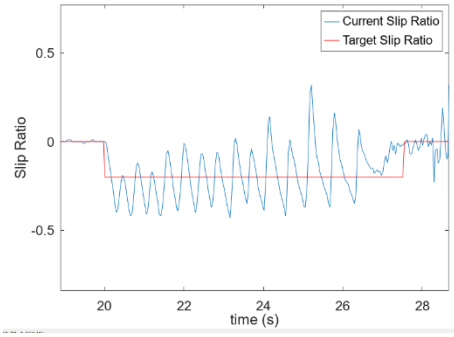

Figure 33: Slip Ratio at Braking

Thought the PID constant had to be adjusted to account for noise, the HIL results are very close to resembling the results from the car simulator.

PID Logic

The PID had the tendency to confuse itself when testing under different road conditions like snow or ice. This means that there needed to be extra safety mechanisms that would trigger the traction and brake controller only one at a time. The following diagram explains the PID logic and how its triggered.

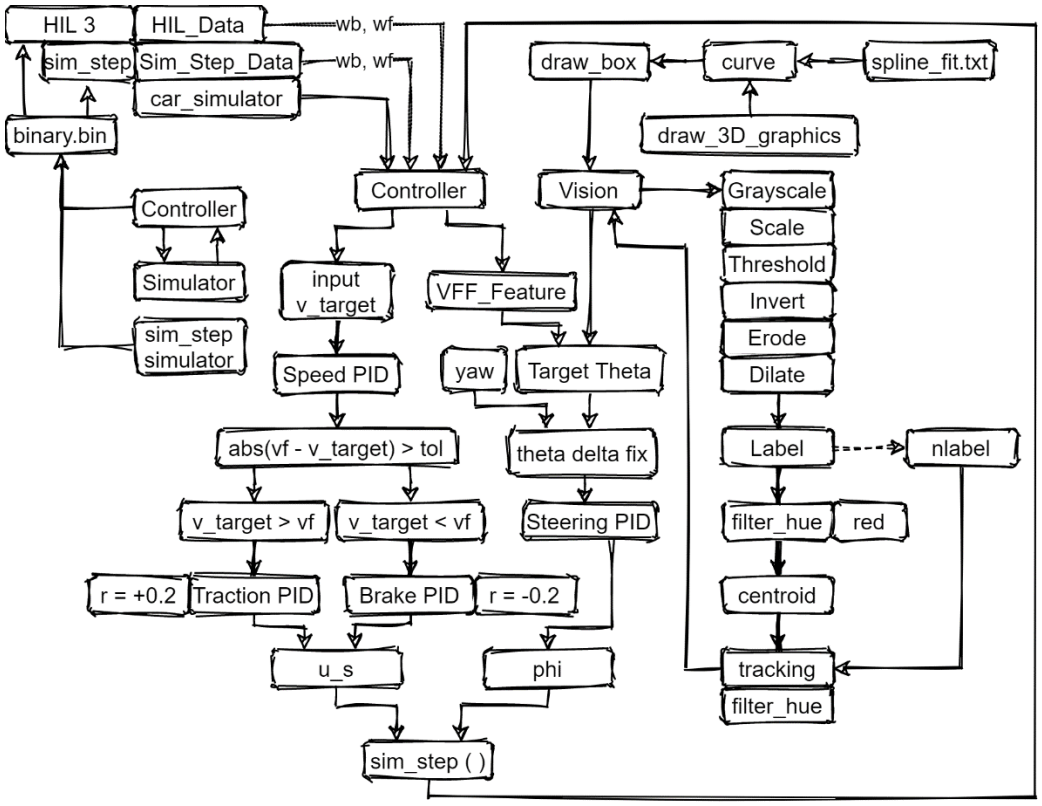

Figure 34- PID Logic Diagram

```
VFF_Feature = 0; //Can be slow since Vision processing takes alot more time than dt = 0.001s
bdmk1.VFF_control(VFF_Feature, u_s, us_max, u_phi, phi_max, t, 0.003);
bdmk1.speed_PID(1, velocity_target, wf, robot1.Rw, u_s, us_max, t, 0.003);
bdmk1.traction_PID(1, u_s, us_max, r, vf, wb, wf, velocity_target, t, 0.003);
bdmk1.brake_PID(1, u_s, us_max, r, vf, wb, wf, velocity_target, t, 0.003);
```

Figure 35- PID Function Calls

Early Controller Designs

Three PID controllers have been designed. The first PID was designed for the launch controller which attempted to accelerate the vehicle as fast as possible to a desired vehicle velocity. The second controller was a traction controller which is activated once the desired velocity has been reached, this will ensure no slip is present during the cruising phase. The final controller is the braking controller which will decelerate the vehicle as fast as possible. The controllers were designed to compare the vehicle's forward velocity to the forward velocity of the rear wheels. The slip ratio compares the forward velocity to the forward velocity of the rear wheel. In order to reduce slipping, the rear wheel velocity needs to converge and match the forward velocity of the vehicle.

The image below illustrates the three controllers in a single image. It is possible to notice the wheel speeds nearly match meaning the slip is very low. It took approximately 13 seconds to reach the maximum velocity and during the cruising, the slip was zero meaning both wheel velocities are identical. At 20 seconds, the brakes were applied, and the vehicle begins to decelerate. Through experimentation, it was found that for this specific controller, it was faster to slam the brakes and ignore the slipping to achieve the quickest deceleration from the vehicles top speed to zero which took approximately 15 seconds. This method is not ideal due to a loss of traction.

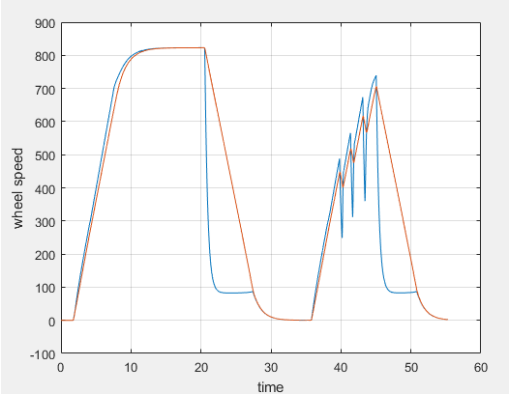

Figure 36: Sim-Step Controller Design

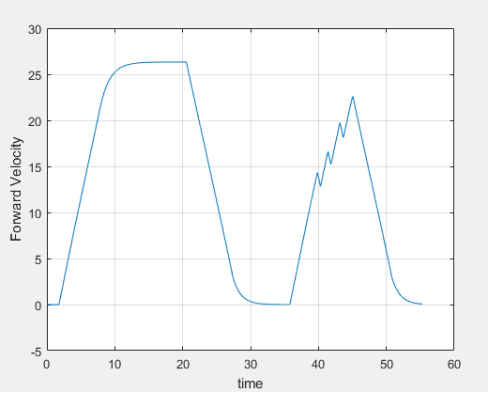

Figure 37: Sim-Step Controller Design

The launch controller and traction controller function in a very similar manner. The only difference between the two controllers is the desired slip ratio to be achieved. Through experimentation, it was realized that to achieve quick acceleration, it was important to have a small amount of slip. Due to this realization, the launch controller would ensure a slip ratio of 0.1 was present during the acceleration phase. Once the desired velocity was achieved, the launch controller would turn off and the traction controller would activate and maintain a slip ratio of 0.

The controllers designed in Figure 36 have been removed from the sim-step function and ported into the controller.cpp file which is focused solely on controlling the motor voltage. By making these adjustments, the simulations became more realistic. We notice in Figure 38, the behavior of the vehicle is not as smooth as seen in Figure 36. The controller was tested on an ice surface and the performance was not acceptable. As seen in Figure 40, a large amount of slipping is present which is undesirable.

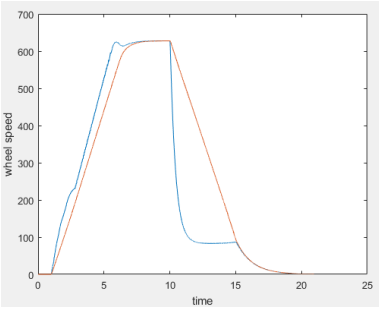

Figure 38: Early Controller Design On Asphalt

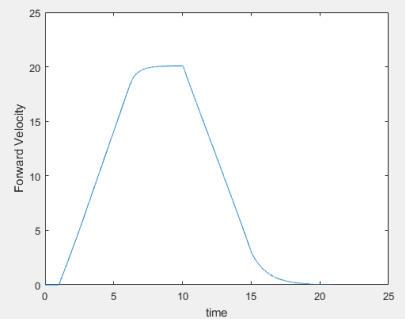

Figure 39: Forward Velocity [$\frac{m}{s}$] On Asphalt

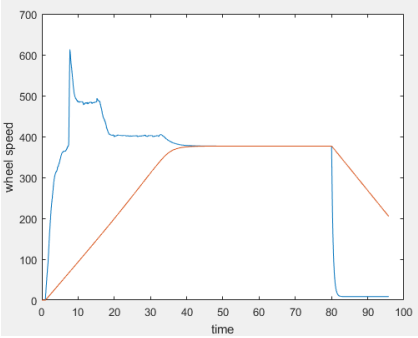

Figure 40: Early Controller Design On Ice

Initial Speed PID design

This PID speed controller was designed to maintain the forward speed of the car at a constant value, and it should be able to adjust the input voltage to the motor if there is a change in speed (acceleration and slowing down). The PID control gains were tuned manually by plotting the forward speed of the car vs time. Initially the kp gain was tuned to allow the car to reach its speed as fast as possible. However, after serval testing it was observed that the model was not capable of accelerating very fast. It takes about 10s to reach a forward velocity of 20m/s. For deceleration it was quite fast. It takes about 3-5 seconds to reduce the speed by 5m/s. As shown in the graphs below, the car was given a speed of 20m/s until t = 15s, then it decelerates to 15m/s and 10m/s at t = 30s and t = 45s. Finally the speed was increased to 18m/s. From the plot it was observed that the front wheel and back wheel were rotating at different speeds, however, after the motor become steady, the velocities become very close. The slip ratio in this case goes to 0 eventually, which indicates perfect rolling. During the testing, different speed setups were tested and they were all successful. It was also noted that with the same parameters, the response plot can be different. This is probably due to the calculation error inside the microcontroller. More iterations of tuning can improve the results. This function was also tested inside Arduino and same results were achieved.

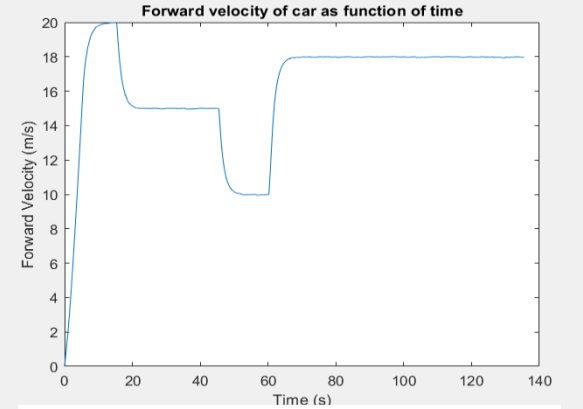

Figure 41- Forward Velocity Plot

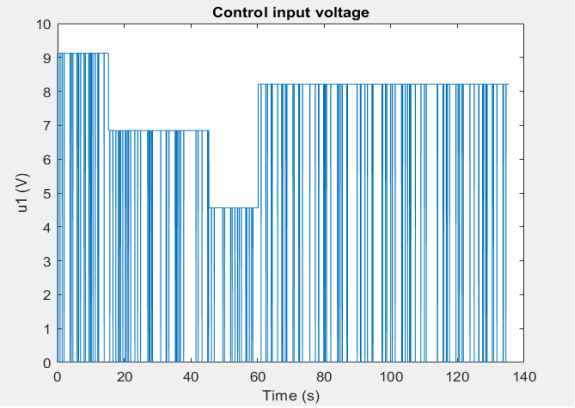

Figure 42 - u_s Voltage Plot

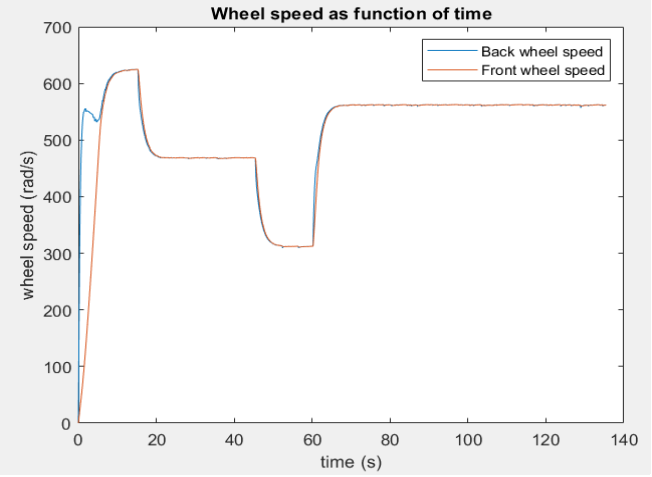

Figure 43- Wheel Speed Velocity at different target velocities

```
void speed_controller(double &v_req) {
    static double kp, ki, kd;//PID control gain parameter/
    static double e_p;
    double t1 = 0, dt; //initial time, time difference
    static double tp = 0;//previous time
    double u; //PID control output
    static double err, err_p; //control error and previous error
    double err_d = 0.0; //derivative of error
    double err_i = 0.0; //integral of error
    double v_car;
    t1 = robot1.t; //Get real simulation time
    v_car = robot1.x[4]; //Read the sensor / get output from the simualtion
    dt = t1 - tp;
    tp = t1; //save time for next call
    kp = 10.0;
    ki = 6.0;
    kd = 1.5;
    err = v_req - v_car;
    err_d = (err - e_p) / dt;
    e_p = err;
    err_i = err_i + err_i * dt;
    u = kp * err + ki * err_i + kd * err_d; //PID control input
    if (u > 12) u = 12; //cap the maximum voltage
    if (u < -12) u = -12;
    robot1.u[1] = u; // Write the voltage to the control input
}
```

Figure 44- Initial speed PID controller function

Steering PID Controller

This original steering controller is used without any graphics library involved. The controller utilizes vector calculations and the curve() function provided by Dr. BW Gordon to determine where the car's current position is and how to steer accordingly. The curve function is called and all the (x,y) coordinates of the spline that makes up the center of the track are recorded in arrays. The coordinates of the car are then compared to each element in these arrays where their distance magnitudes are computed. Then, the array index number ("best_index") of the current point on the spline that has the least distance from the car is recorded:

```
if (steer_init == 0) {
    bdmk1.steer_x[bdmk1.steer_index] = x;
    bdmk1.steer_y[bdmk1.steer_index] = y;
    bdmk1.steer_theta[bdmk1.steer_index] = theta;
    bdmk1.steer_index += 1;
}
```

```
for (loop_i; loop_i < steer_index; loop_i++) {
    //Calculate the difference in x and y position of the car to ALL
    current_x = abs((xc - steer_x[loop_i]));
    current_y = abs((yc - steer_y[loop_i]));
    current_mag = pow((pow(current_x, 2) + pow(current_y, 2)), 0.5);

    if (current_mag < closest_mag) {
        //If the point on the spline is the currently observed CLOSES
        closest_mag = current_mag;
        best_index = loop_i;    //array element of the spline that is
    }
}
```

With the closest point on the spline obtained, a car vector and spline vector are computed using another spline point, a few array elements earlier, as the origin/tail of the vectors. The cross product of the car and spline vector is computed where a negative result means the car is to the left of the center of the road, and a positive value is the right of the center of the road. The distance/magnitude between the car vector and spline vector is then computed, which is simply the cross product divided by the spline vector's magnitude, and this acts as the error we want to minimize using the PID:

```
//Distance between robot and spline curve is cross_prodcut/|curve_vector|
curve_vector_mag = pow((pow(curve_vector[0], 2) + pow(curve_vector[1], 2)), 0.5);
delta_distance = cross_product / curve_vector_mag;

error = delta_distance;
error_dot = (error - old_error) / time_delta;
int_error = int_error + error * time_delta;
phi = kp_PID * error + ki_PID * int_error + kd_PID * error_dot;
```

Figure 45- Vector Calculation and PID

After running the simulation with the implemented steering control, the car positioning and steering error is plotted in MATLAB and graphed below. The steering error is essentially the distance between the car position and the closest spline coordinate at that point in time.

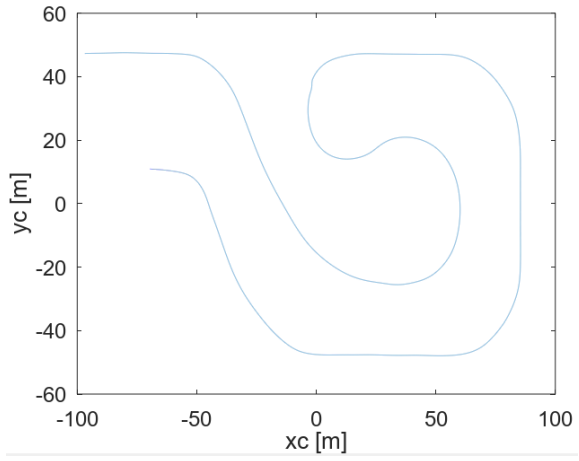

Figure 46- Car Positioning Plot on Dry Terrain

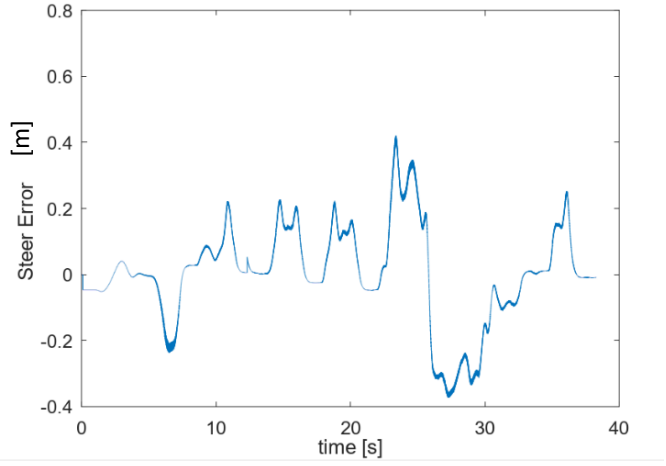

Figure 47- Distance Between Car from Spline (Steer Error) Calculated by PID