Traction Control Study for a Scaled Automated Robotic Car

By

Mark A. Morton

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

May 5, 2004
Blacksburg, Virginia

Keywords: sliding mode control, automated robotic car, simulated annealing, traction control, nonlinear control, hybrid model

Virginia Polytechnic Institute
and State University

Traction Control Study for a Scaled Automated Robotic Car

Mark A. Morton

(ABSTRACT)

This thesis presents the use of sliding mode control applied to a 1/10th scale robotic car to operate at a desired slip. Controlling the robot car at any desired slip has a direct relation to the amount of force that is applied to the driving wheels based on road surface conditions. For this model, the desired traction/slip is maintained for a specific surface which happens to be a Lego treadmill platform. How the platform evolved and the robot car was designed are also covered.

To parameterize the system dynamics, simulated annealing is used to find the minimal error between mathematical simulations and physical test results. Also discussed is how the robot car and microprocessor can be modeled as a hybrid system. The results from testing the robot car at various desired percent slip show that it is possible to control the slip dynamics of a 1/10th scale automated robotic car and thus pave the way for further studies using scaled model cars to test an automated highway system.

Acknowledgments

If children had teachers like Dr. Puskin Kachroo, everyone would aspire and strive for his/her full potential in everything they do. It has been with his great enthusiasm and wealth of knowledge that this author has been reenergized to learn more than humanly possible. Without Dr. Kachroo's guidance, patients, and inspiration this thesis would not have been possible; and this author would have missed out on much.

Other sources of great intelligence and inspiration have been Patricia Mellodge, Eric Moret, and Charles Lepple. They have given and shared seemingly subtle but truly important ideas and understanding, thank you.

To have even gotten to this point would not have been possible without the events God has set in motion and for which I am eternally grateful. I am also thankful for an incredibly supportive and loving family. Of which my wife, Siyuan Morton, has been the most incredibly supportive person I could ever have prayed. Without her I would be lost in a life without purpose and meaning /without life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Full size cars are now beginning to have traction control available on more and more vehicles; however, can these same concepts be applied to scaled vehicles? To best answer this question, a scaled model needs to be tested.

1.1 Motivation

Since the advent of the Model-T by Ford, buying vehicles has become increasingly more accessible for the general public. Inevitably, with an increased population of people and thus vehicles, the highways and roads are becoming increasingly congested with little or no room to expand. There is also the danger to road crews when the infrastructure needs repair. One possible solution to improve safety and limits on the ability to expand roads would be to have an automated system of robotic cars on the roads [1]. Such a system could potentially allow for an improved flow of vehicles, improved fuel efficiency, improved safety, and the list goes on [2]. An autonomous system of cars on roads may meet some opposition as people reluctantly give up control of their cars. However, automation is already a reality as people want safety features such as power brakes, power steering, antilock breaking, traction control, anti-slip, cruise control, and now adaptive cruise control. All these features build up to the next logical step of building an automated system of cars. Before implementing full-scale automated roads, concepts and reliability need to be tested to ensure public safety. To perform such tests on full-scale cars and roads would be too costly; therefore, scaled cars and roads need to be used. The dynamics involved in a full size car are very complex and numerous. It is therefore best to break the problem of testing and comparing the dynamics of a full size car to that of a scaled car into manageable parts. One of the many important parts of automating a car is the ability to control tire slip. To break the problem down even further, by controlling tire slip in just a longitudinal direction, lateral stability will also be improved [1]. Therefore, this thesis covers the dynamic component of controlling wheel slip implemented on a 1/10th scale robot car as a step towards modeling all the dynamics of full size vehicles.

1.2 Previous Research

Much research has been done to analyze the dynamics of vehicles and the components needed for acceleration. To control acceleration, the dynamics involved must first be understood. Looking at Figure 1.1 different surfaces have different coefficients of friction (μ) verses percent slip (λ). The challenge becomes adjusting to the specific traction/slip curves (based on road and tire conditions) and controlling/maintaining a specific amount of traction/slip. More details of the dynamics involved are covered in Chapter 2. For the scaled model implemented here only one specific surface was used in order to reduce the initial order of complexity. The surface used is covered in Chapter 4.

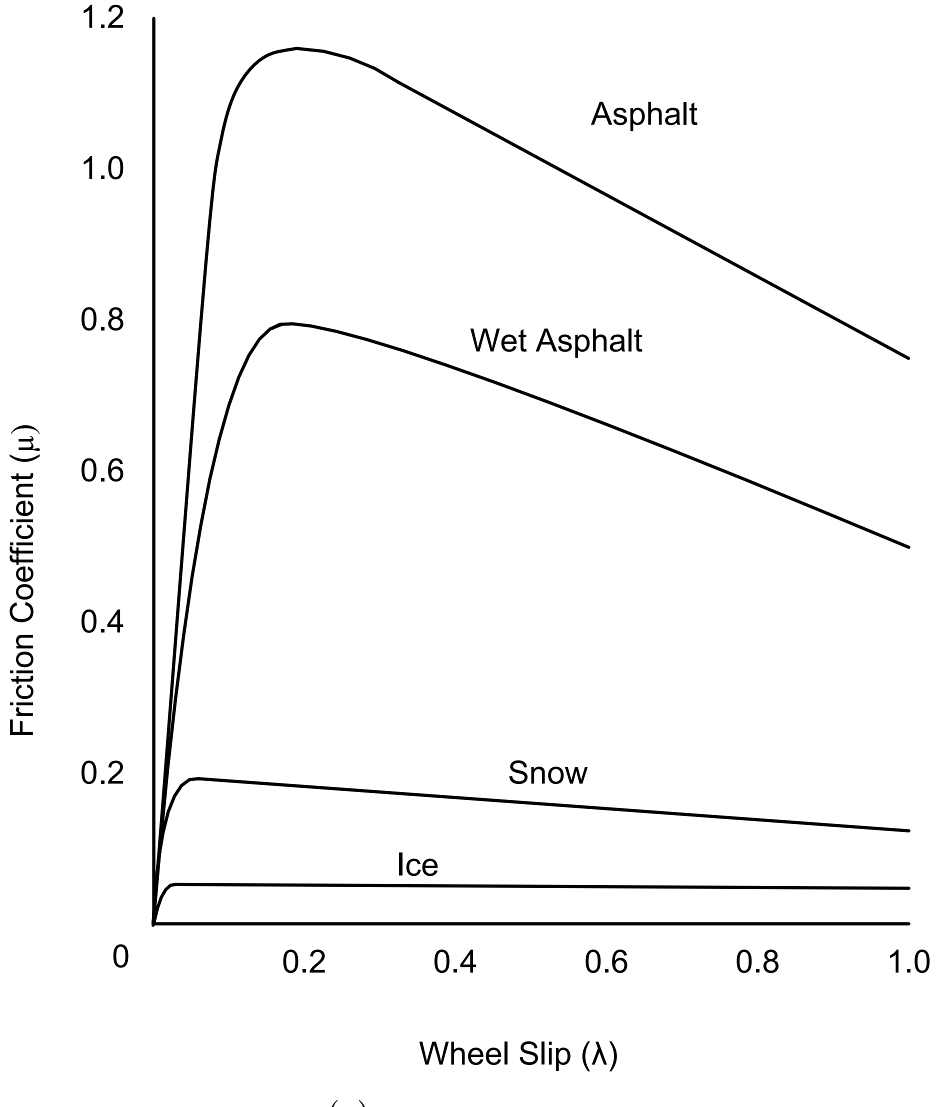


Figure 1.1: Traction-Slip/ $\mu(\lambda)$ curve for various road conditions, modeled from [2]

Work has been done to model the controlled slip of a vehicle as a continuous system [1]; however, a more accurate model would include the discrete components such as the processor(s). Since the scaled implemented system will actually be a hybrid of continuous and discrete components/systems, the hybrid model concept will be discussed.

1.3 Highlights and Outline of Thesis

This thesis is organized as follows:

- Chapter 2 covers the dynamics, mathematical model and control law design.
- Chapter 3 will explain the vehicle and circuit constructions.
- Chapter 4 shows the test platform construction used to test the robot car.
- Chapter 5 explains the development of the programs and dynamic parameters.
- Chapter 6 covers the final data collection and results.
- Chapter 7 discusses how the system is best represented as a hybrid model.
- Chapter 8 closes with conclusions and future work.

Chapter 2

Dynamics, Mathematical Model, and Control

Chapter 2 first explains some of the dynamics of tire slip followed by the mathematical equations used and finishes with an explanation of the control law used. These equations or a slight variation is what will be used later by the microprocessor on the robot car.

2.1 Background of Dynamics

As my grandfather used to say, "There are two things between you and the road, so don't skimp on them," those two things were shocks and tires. For this 1/10th scale model there was little to no suspension so these complex dynamics were ignored. Observations of the other major forces exerted on a pneumatic type tire accelerating in a linear direction are represented by Figure 2.1.

Now as a tire begins to accelerate, it deforms [3]. It is during this time that the $\mu(\lambda)$ curve is linear as seen in Figure 2.2 region O to A. As the tire stops deforming and tractive effort/engine torque increases, more of the tire tread begins to slip resulting in the nonlinear region from A to B. It is to the right of region B where too much slip occurs and traction is lost. For a pneumatic tire on a hard surface, the maximum acceleration is achieved when there is 15 to 20% slip according to [3]. To put traction another way and greatly simplifying the dynamics of acceleration, when a vehicle accelerates, force is applied from the engine to the tires which react with the road surface. If the force between the engine and the tires is greater than the force of friction between the tires and the road surface, the tire will begin to slip excessively (skid-for breaking). Figure 2.3 shows the complete $\mu(\lambda)$ curve for breaking and accelerating. In reference [3], longitudinal slip is defined mathematically as

$$Slip(\lambda) = \left(1 - \frac{V_V}{R_W \omega_W}\right) \tag{2.1}$$

where V_V is the linear velocity of the wheel/car, R_w is the radius of the tire, and ω_W is the angular velocity of the driving wheels. In [1], λ (slip) was used as the variable to be controlled because it has a direct relation to tractive effort (μ). In other words by maintaining a desired slip on a specific surface, the desired torque (input) can be determined.

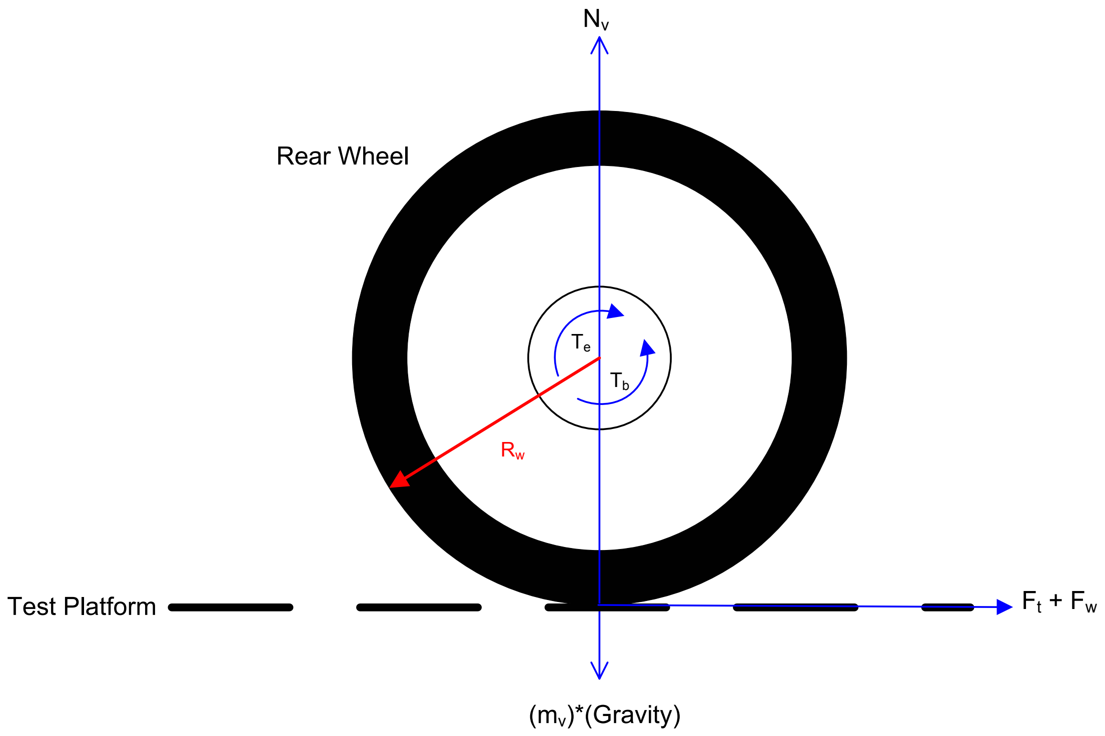


Figure 2.1: Forces acting on a tire in a linear direction, modeled from [1, 3]

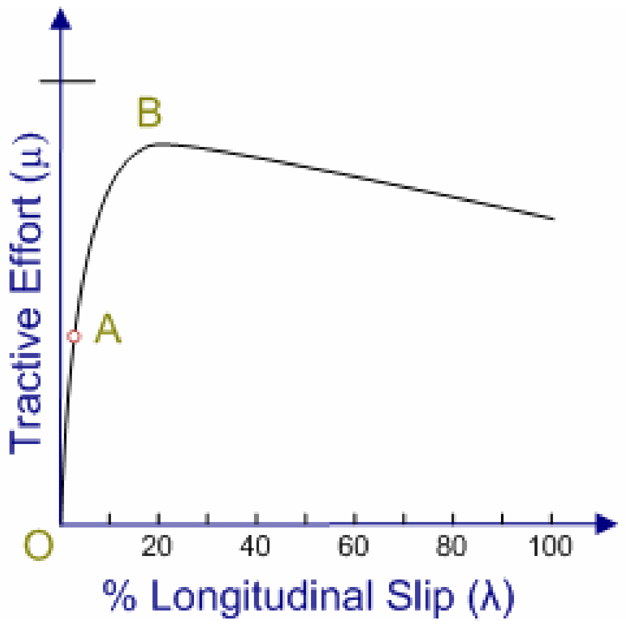


Figure 2.2: $\mu(\lambda)$ **curve, modeled from [3]**

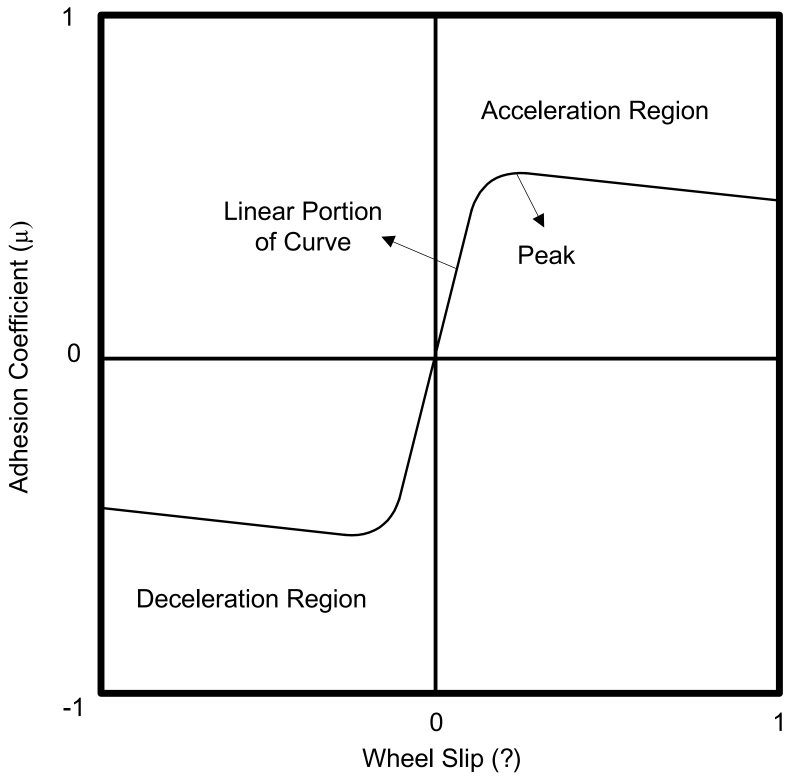


Figure 2.3: $\mu(\lambda)$ **curve for deceleration and acceleration, modeled from [4]**

2.2 Mathematical Model

Using the vehicle dynamics developed in the dissertation [1], the mathematical model was developed. For definitions of variables and their units, refer to Appendix A. The states of the system are as follows:

$$x_1 = \omega_v = \frac{V_V}{R_W} \tag{2.2}$$

$$x_2 = \omega_W \tag{2.3}$$

In the equations above, x_1 represents the angular velocity of the front wheels and x_2 represents the angular velocity of the rear wheels. For acceleration (as opposed to breaking) λ (representing percent slip) is as follows:

$$\lambda = \frac{(\omega_W - \omega_V)}{\omega_W} = \frac{x_2 - x_1}{x_2} = 1 - \frac{x_1}{x_2} \tag{2.4}$$

$$\because \lambda = 1 - \frac{x_1}{x_2} \Rightarrow 1 - \lambda = \frac{x_1}{x_2} \tag{2.5}$$

Appling Newton's Law to Figure 2.1, F_t is determined by the equation

$$F_t = N_V \times \mu \tag{2.6}$$

Using Newton's Law again and accounting for wind drag, the equations for linear and angular velocity were developed. The derivatives of these velocities were then used in Equations (2.9) and (2.10) where the dots over the variables indicate differentiation with respect to time.

$$\dot{V}_V = \frac{n_W F_t - F_V}{m_V} \tag{2.7}$$

$$\dot{\omega}_W = \frac{T_e - R_W F_t - R_W F_W}{J_W} \tag{2.8}$$

In order to have the states in a form where the control variable was present, the derivative of x_1 and x_2 was taken.

$$\dot{x}_1 = \dot{\omega}_V = \frac{\dot{V}_V}{R_W} = \frac{n_W F_t - F_V}{m_V} = \frac{n_W N_V \mu - F_V}{m_V R_W} \tag{2.9}$$

(Note: F_W was ignored as it has little effect on the resulting dynamics because the front wheel was assumed to not slip in order to determine the linear velocity of the car)

$$\dot{x}_2 = \dot{\omega}_W = \frac{T_e - R_W F_t - R_W F_W}{J_W} = \frac{T_e - R_W N_V \mu - R_W F_W}{J_W} \tag{2.10}$$

Again the aim was to control the amount/percent of slip on the rear wheels; therefore, the derivative of λ was performed and put into the general form for a single-input dynamic system, $x^{(n)} = f(\mathbf{x}) + b(\mathbf{x})u$ [1, 5, 6] for sliding mode control (the control algorithm used to control percent slip is discussed in section 2.3). This yielded

$$\dot{\lambda} = \left[\frac{F_V - n_W N_V \mu}{m_V R_W \omega_W} - (1-\lambda) R_W \frac{F_W + N_V \mu}{J_W \omega_W} \right] + \frac{(1-\lambda)}{J_W \omega_W} T_e \tag{2.11}$$

Here T_e, torque provided by the engine of a petrol car, is the control variable. However for the 1/10th scale model car, an electric motor was used. Therefore, the control variable needs to be in terms of volts. Using the linear model of a motor, torque was expressed in terms of volts (where V is in volts) by the following equation:

$$T_e = K_e(\frac{V - K_e\omega_W}{R_a}) \tag{2.12}$$

Substituting Equation 2.12 for torque in $\dot{\lambda}$ gives

$$\dot{x}_1 = \frac{n_W N_V \mu - F_V}{m_V R_W}$$

$$\dot{x}_2 = \frac{T_e - R_W F_t - R_W F_W}{J_W} \textbf{:} \text{ where } F_t = N_V \mu \textbf{:} \; T_e = K_e\left(\frac{V - K_e\omega_W}{R_a}\right) \quad \textbf{:} \; \omega_W = x_2$$

$$\dot{x}_2 = \frac{K_e\left(\frac{V - K_e\omega_W}{R_a}\right) - R_W N_V \mu - R_W F_W}{J_W}$$

$$\dot{\lambda} = \frac{d}{dt}\left(1 - \frac{x_1}{x_2}\right) = \frac{-x_2\dot{x}_1 + x_1\dot{x}_2}{x_2^2} = \frac{(1-\lambda)\dot{x}_2 - \dot{x}_1}{x_2}$$

$$\dot{\lambda} = \underbrace{\left[\frac{F_V - n_W N_V \mu}{m_V R_W \omega_W} - (1-\lambda)R_W \frac{F_W + N_V \mu}{J_W \omega_W} - \frac{(1-\lambda)K_e^{\,2}}{R_a J_W}\right]}_{f(\lambda)} + \underbrace{\frac{(1-\lambda)K_e}{R_a J_W \omega_W}}_{b(\lambda)} \underbrace{V}_{u} \tag{2.13}$$

It is important to understand that the microprocessor on the robot car gives desired voltage in terms of a PWM signal or percent of maximum voltage available (this is explained in subsequent chapters). Therefore, in computer simulations, the following equation was used to view volts in terms of actual volts applied and not a percentage.

$$V = \left(\frac{u}{1023} \times VoltsAvailable\right) \tag{2.14}$$

Note: 'u' here is between $\pm$ 1023 which represents the range for the PWM signal.

2.3 Control Law Design

Since the dynamics of the $\mu(\lambda)$ curve are nonlinear and the exact $\mu(\lambda)$ curve for the road surface is not known, sliding mode control was used. Sliding mode control is useful for non-linear systems where the boundaries of the system may be known, but the exact dynamics are not. In this system the range of the $\mu(\lambda)$ curve is known, but the exact curve for the road surface is not. In this part of the discussion, the actual control variable *V* will be referred as u .

In sliding mode control, the sliding surface *s* represents the amount of error in what is being controlled. What is trying to be controlled is the amount of slip (λ); therefore, $s : s = \lambda - \lambda_d$.

For the sliding mode control computer model, *F* and *b* were initially fixed and an actual value of μ was used in order to test and debug the simulation. Here *F* represents the upper bounds of the absolute error between f and $\hat{f}$. Also, *f* comes from Equation 2.13 and $\hat{f}$ is a best estimate of *f.*

.

In order to control the slip at a specific value, the $\mu(\lambda)$ curve must be known for the road surface. Using the function created in [4] represented here as Equation 2.15 an approximate $\mu(\lambda)$ curve can be generated, allowing $\dot{\lambda}$ to be solved for and λ to be controlled.

$$\mu(\lambda) = \frac{2\mu_p \lambda_p \lambda}{\lambda_p^2 + \lambda^2} \tag{2.15}$$

The ranges of μ and λ were obtained from interpreting the plot below in Figure 2.4. The ranges initially used were $0.0 \le \lambda \le 0.5$, $0.22 \le \mu_P \le 0.92$ and $0.1 \le \lambda_P \le 0.4$ where μ_P is the peak value of μ and λ_P is the peak value of λ on the $\mu(\lambda)$ curve.

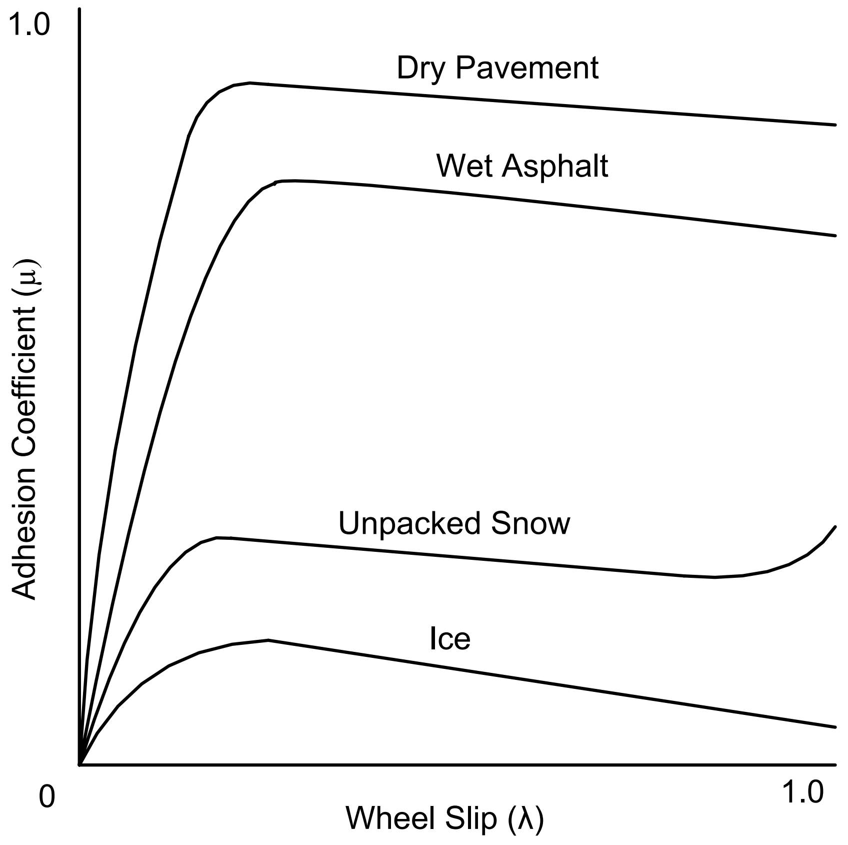


Figure 2.4: $\mu(\lambda)$ **curve for a full size car with pneumatic tires, modeled from [4]**

It is important to note that Equation (2.15), used to generate approximate adhesion coefficient plots, loses accuracy as λ approaches 1.0 and according to [4], is best suited in the range of $0 \le \lambda \le 0.3$. Because the area where wheel slip was to be maintained was approximately $0 \le \lambda \le 0.5$, the equation should be acceptable. The nominal curve was then fitted/approximated with a 6th order polynomial function. This function was then shifted up slightly and used as $\hat{\mu}$ (see Figure 2.5). The result from Equation (2.15) was assumed equal to the actual μ value of the road surface. These values were then used to determine $\hat{f}$, f, and F. Again, f is determined from Equation (2.13).

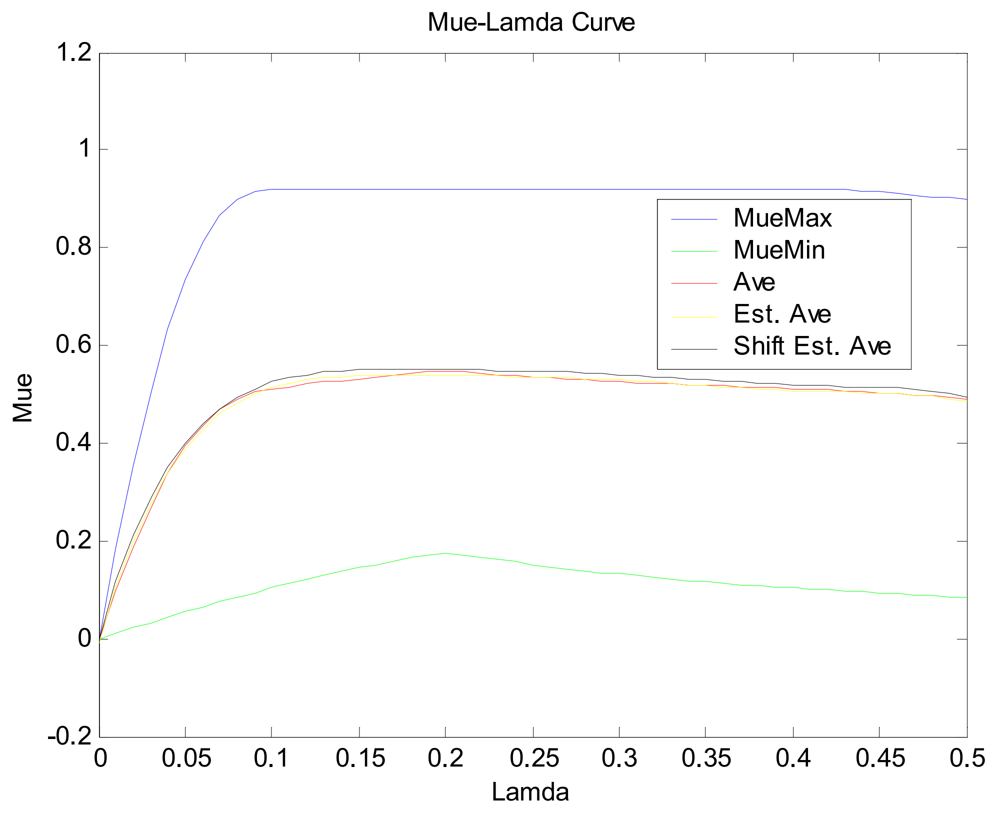


Figure 2.5: Generated from maximum and minimum ranges of Equation 2.15

Thus,

$$f(\lambda) = \left[\frac{F_V}{m_V R_W \omega_W} - \frac{(1-\lambda) R_W F_W}{J_W \omega_W} - \frac{(1-\lambda) K_e^{\ 2}}{R_a J_W} \right] - \left[\frac{n_W N_V}{m_V R_W \omega_W} + \frac{(1-\lambda) R_W N_V}{J_W \omega_W} \right] \mu \tag{2.16}$$

such that μ is determined from Equation (2.15) and F_V is equal to the wind drag coefficient c times the linear velocity squared. With Equation (2.16) in the form of $f = A + B\mu$, and $\hat{f}$ in the form of $\hat{f} = A + B\hat{\mu}$, again where $\hat{\mu}$ was the shifted estimated nominal curve of Equation (2.15) shown in Figure 2.5, the bounds of F could be calculated. This gave the upper bounds of F from the equation

$$F \geq \left| f - \hat{f} \right| \Rightarrow \left| A - B\mu - A - B\hat{\mu} \right| = \left| -B(\hat{\mu} - \mu) \right|$$

$$F \geq \left| \left(\frac{n_W N_V}{m_V R_W \omega_W} + \frac{(1-\lambda) R_W N_V}{J_W \omega_W} \right) (\hat{\mu} - \mu) \right| \tag{2.17}$$

To test the computer simulation, a mixture of approximated, assumed, and measured values for the car were used. As the code was proven to work, the initially assumed and fixed values of $\hat{f}$, $\hat{\mu}$, $\hat{b}$, and so forth were substituted with the equations derived above. An example would be how $\hat{f}$ was originally set equal to f times a percent multiplier.

```
f_hat  = 1.15*f;
```

The same line of code finally used and based on textbook sliding mode control was

```
f_hat  = [Fv/(mv*Rw*x(2)) - (1-Lamda)*Rw/(Jw*x(2)) - (1-Lamda)*K^2/(Ra*Jw*x(2))]
- [nw*Nv/(mv*Rw*x(2)) + (1-Lamda)*Rw*Nv/(Jw*x(2))]*Mue_hat;
```

Another example was where Mue had been used for Mue_hat in one stage of the development process. Functions $b(\lambda)_{\min}$ and $b(\lambda)_{\max}$ were derived as percentage values of $\frac{(1-\lambda)K_e}{R_a J_W \omega_W}$ from Equation (2.13). The simulation was also run with the gain margin $\beta = \sqrt{\frac{b_{\max}}{b_{\min}}}$ and with β equal to a constant value. Since β had little to no effect on the simulated results but added computation time to the robot car's microprocessor, β was set as a constant determined by using simulations to confirm similar results. The remaining equations derived for sliding mode control were $\hat{u} = -\hat{f}$, the gain $K = \beta(F + \eta) + (\beta - 1) * |\hat{u}|$, and the control law was $u = \frac{1}{\hat{b}}(\hat{u} - K * \operatorname{sgn}(s))$ [1, 5, 6].

Chapter 3

Vehicle and Circuit Construction

This chapter covers the construction of the vehicle, robot car's circuitry, and data collection circuitry. Some of the developmental thought processes and adjustments will also be discussed.

3.1 Vehicle Chip, Compiler, and Code Development

Before construction could begin, I/O processing and available material needs had to be assessed. The basic input needs were the linear velocity of the vehicle and the angular velocity of the driving wheels. As a chasse with an encoder mounted on the rear wheels was already available, this was used. Also encoders provide the best solution of cost, accuracy, and functionality for the revolutions measured. To determine the vehicle's linear velocity, the front wheel's radius and angular velocity were used. This calculation assumes the front wheel can be used to accurately measure and calculate the linear velocity of the vehicle. To measure the angular velocity of the front wheel, an encoder was mounted on the left front wheel (same side as rear encoder). The front left wheel had to be trimmed down and shifted out to accommodate the encoder, thus the right front tire was also shifted out for symmetry (see Figure 3.1).

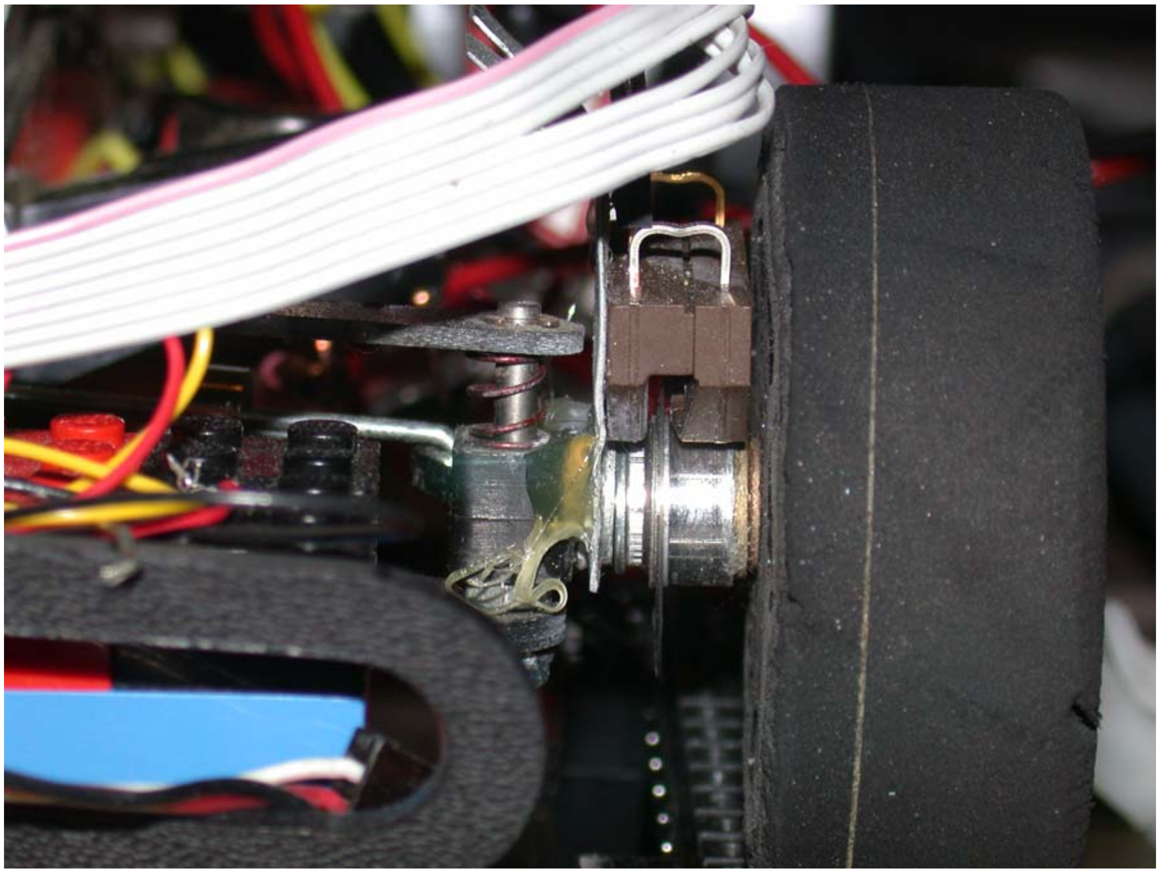


Figure 3.1: Picture of the left front wheel

To operate the robot car, a controller was needed that could handle the inputs from the front and rear wheel encoders, output a pulse width modulated (PWM) signal, and be fast enough to do these functions as well as compile the control algorithm as fast as possible to minimize delays. To control the robot car, the Microchip 16F877A was initially selected to develop and test a simple code referred to as the Follower code (see Appendix B). This code simply used interrupts to measure the velocity of the front and rear wheels, then increased/decreased power to the motor one bit at a time until the values from the wheels were the same. This chip operated at 20 MHz with an instruction cycle of 200ns. The next step was to try implementing the simplest control code simulated on the computer and run it on the car. Because implementing the control code would require multiplication, division, addition, and subtraction, various C compilers were tried. Otherwise, added development time and potential for introducing error would occur. Initially CC5X was tried, but the compiler was found too limiting. Also, the 16F877A chip was quickly running out of memory and the processing time was barely desirable even with using techniques to improve processing time. Therefore, it was decided that the Microchip 18F452 would be used which also had the same chip set as the 16F877A. The new chip also had a one-step multiplier, 100ns instruction cycle, greater number of interrupts, greater number of clocks, and increased amount of memory. High Tech C was the compiler finally used due to its capabilities. The Follower code was then improved using the two independent clock inputs as counters. The use of the counters was then implemented on the control algorithm.

3.2 H-bridge

To power the motor an RC motor controller was contemplated; however, the maximum resolution and frequency would have been much less than that of an H-bridge. The RC driver would have had 512 bits resolution in the forward direction verses 1023 bits for an H-bridge and a frequency of 1250 Hz [7]. Since the 18F452 chip has a minimum PWM output of 2.44 KHz and the RC battery being used had a maximum voltage of 8.52 volts fully charged and the motor could require an inrush current of up to 7 amps or more, off the shelf H-bridges were originally found unacceptable. Later one was found that could operate up to 10 kHz with a 5.2A output but inevitably still didn't perform as well as the "home made" h-bridge. The design for the H-bridge made was found on the internet [8] (Figure 3.2). To better accommodate the lower voltage from the battery, resisters R1 and R2 were replaced with 5K Ohm resisters and parts R9 and associated LED were removed (see Appendix C for parts list).

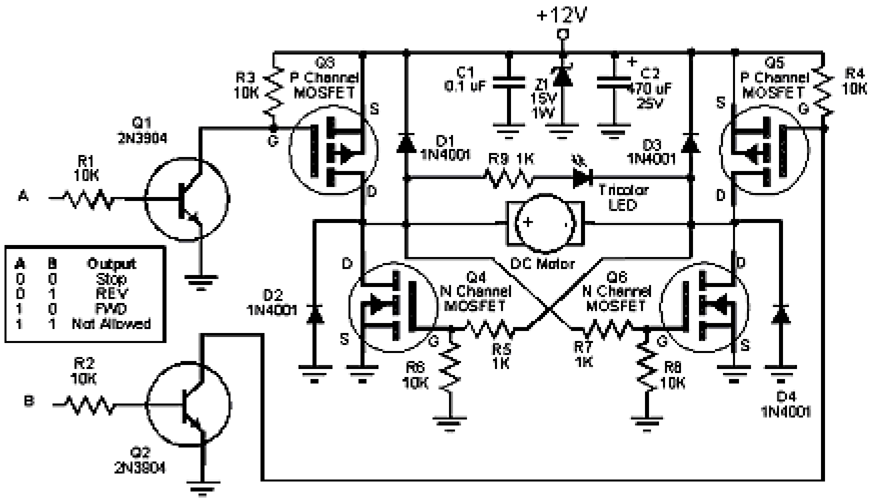


Figure 3.2: H-bridge [8]

The initial motor used was a standard stock radio control (RC) car motor. This later proved to have not enough low-end torque, even with a 21 to 81 gear ratio, so a rewound motor was used. The new motor had been rewound to have 100 turns per phase.

3.3 Vehicle Circuit

To have reverse on the H-bridge, the PWM signal and ground would have to be able to swap and since the processor only had one PWM signal generator, a multiplexer was used (part U8, Figure 3.3). With the multiplexer connected properly, only one PWM signal was needed and one bit/line for selecting direction.

To measure angular velocity from the 512 bits per revolution quadrature encoders, a D flip-flop was used [9] with the A and B channel inputs from the encoders to generate a 256 bit per revolution output per encoder. The direction the encoder rotates determines the connections of channels A and B giving an output only if the vehicle is moving in a positive direction. Since the objective was to control longitudinal slip, reverse should never need to be measured. On the final design, the outputs of the two channels from the D flip-flop went to the two independent counters on the 18F452 processor. The reason for using counters was that the processor was able to continue counting pulses from the D flip-flops while calculating the next control output, thus greatly minimizing the time delay. To aid in data analysis, the 10 bit PWM signal (otherwise known as the control variable u) plus the direction bit were sent to the Data Collection Circuit (DCC). Port B sent the upper 8 bits of the 10 bit PWM signal (B7 the greatest bit) and the lower 2 bits were sent from Port D4 and D5 (D4 the lowest bit).

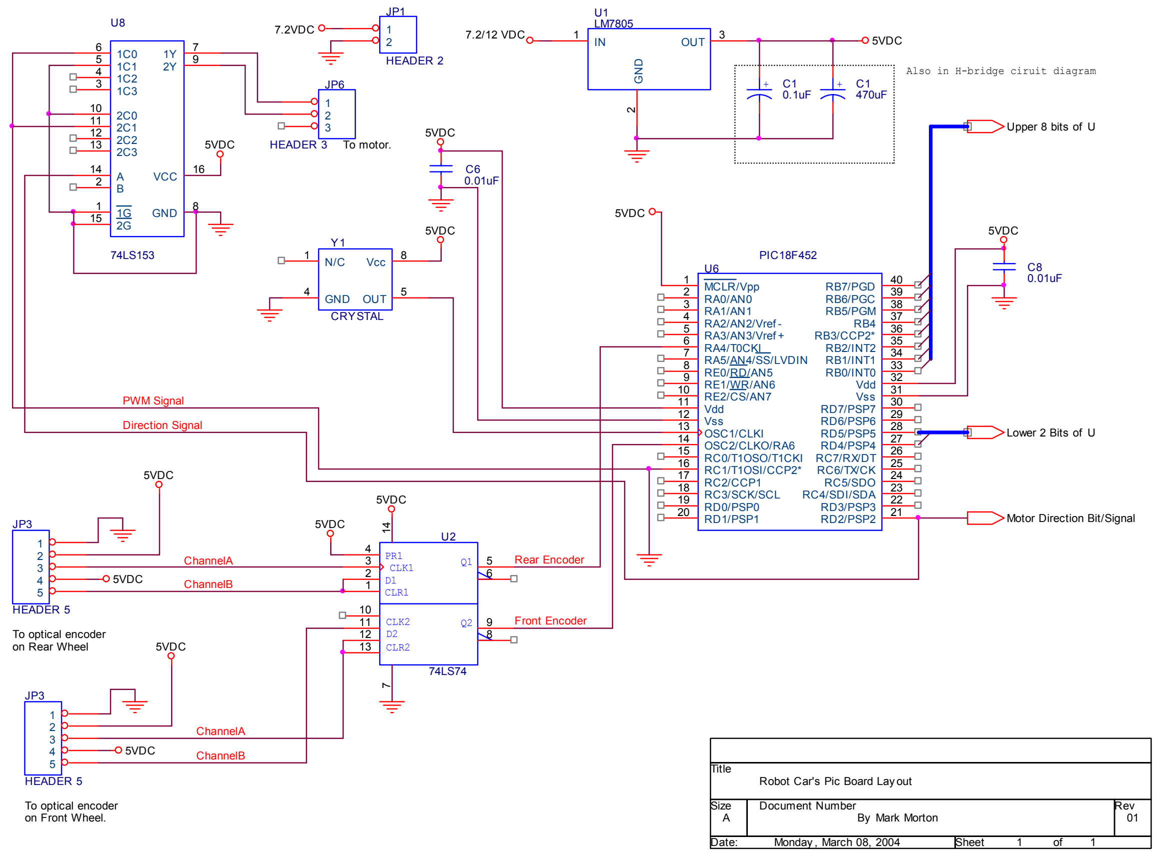


Figure 3.3: Robot car control circuitry

3.4 Data Collection Circuit and Code Development

Just as with the code on the car, Port B was the input for the highest 8 bits of the 10 bit value of the PWM signal. Ports D4 and D5 (with D4 the lowest bit) received the lowest two bits of the 10 bit PWM signal. Port D2 was the input for the direction bit (0 for forward, 1 for reverse). To minimize the size of the data collection file created by Matlab, the DCC would not begin sending data to the computer until Port D3 was high, which was triggered by the cars processor when the car was turned on. This allowed a velocity of ‘0’ to be captured for 1 to 3ms before the car began moving (see Figure 3.4 for a Block Diagram). As with the car’s processor Port A4 (counter/timer 1) taps off the output from the D flip-flop output from either the rear or front encoder, usually rear. Port C0 (counter/timer 0) received input from the car’s opposite encoder, usually front. The DCC and test platform were also equipped with an encoder and D flip-flop for future developments and tests.

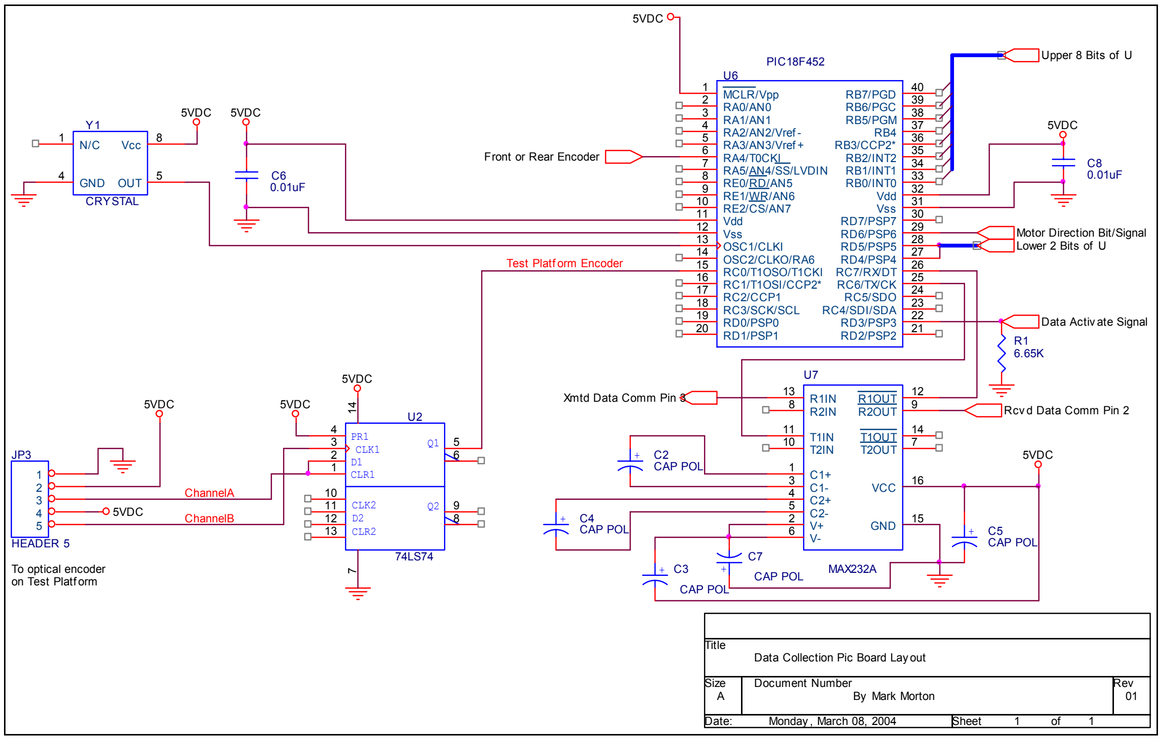


Figure 3.4: Data collection circuit diagram

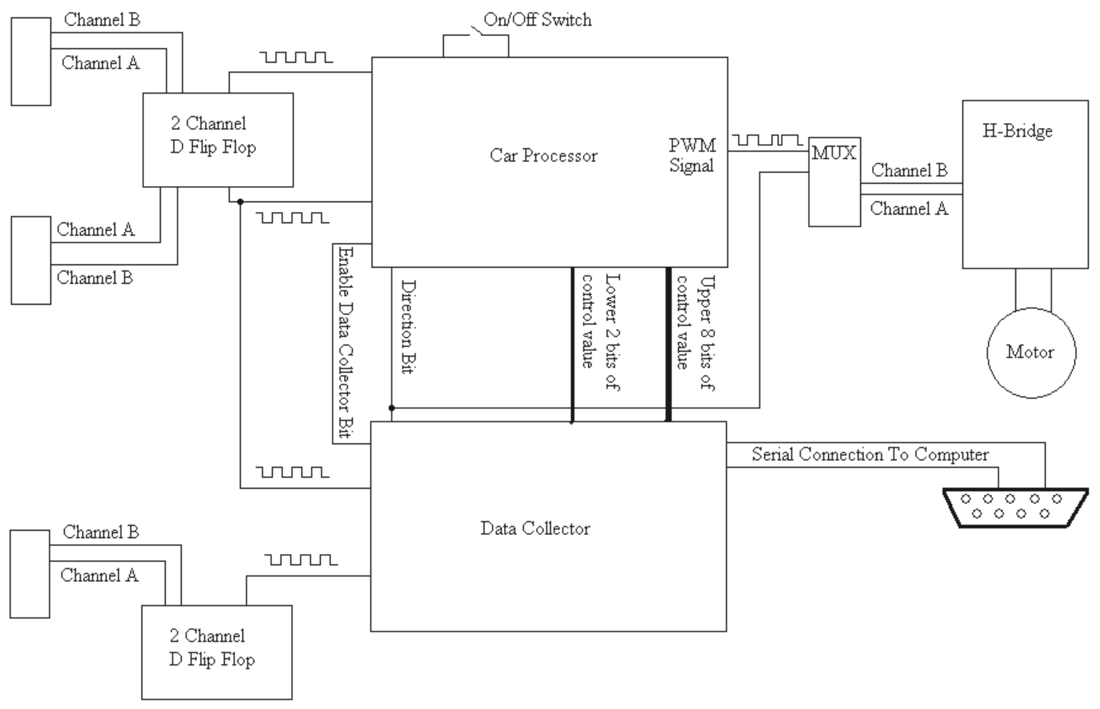


Figure 3.5: Block wiring diagram of robot car and data collection circuits

Originally, the idea was investigated to put a 512Mbyte memory chip on the car and then have the results downloaded to a computer. This idea had several disadvantages. The first disadvantage was it would have taken nearly every pin on the processor and thus required a lot of wiring time. This would have also meant that collecting data points would have only happened as often as the computation time took, which at the time was taking about 15ms. A 15ms sample rate would not have been often enough to generate a smooth enough plot. The biggest reason was the concern that data might need to be collected over time periods longer than memory would allow. As it was possible to collect and send data to a computer in real time to be stored at an interval faster than 1ms, this was the method used. When data is transmitted using a serial connection there is an overhead of 2 bits per 1 byte of information. The counts per interval of the rear and test platform encoders were handled as 1 byte each. Meaning 10 bits for each encoder per sampling interval was sent. The control value from the car was 10 bits plus 1 bit for direction. Since only 8 bits of data could be sent at a time, 2 bytes of data had to be sent from the car to the DCC. This is because there was one byte for the upper 8 bits of the control value and one byte containing the lower 2 bits of the control value plus direction. Meaning 5 bits from one of the bytes sent from the car to the DCC was Don't Cares. Summing it all up yields

10 bits (test platform) + 10 bits (rear wheel) + 10 bits (upper 8 bits of U) + 10 bits (lower 2 bits of U and 1 direction bit) = 40 bits per sampling period
(Note: 10 bits = 8 bits (data) + 2 bits (overhead)

The 9-pin communication port on the computer at the time was capable of 115200 bits per second (bps). So 40 bits per sampling period divided by 115200 bps allowed for sampling as fast as approximately every 0.347ms. For the velocities obtained, 1ms to 5ms sample rates were sufficient.

As the Follower Code was created only to test programming of the Robot Car's processor, no data collection was initially implemented just -- visual inspection of how well the rear wheel followed when the front wheel was spun by hand. The code was first developed using interrupts and then improved using counters. The Follower Code was then used to test the DCC with the computer. The DCC was setup to transmit collected data from the suspended car every 1ms (40Kbps). Windows HyperTerm software was used first to examine the data from the DCC and car. Software from [10] was then used to collect and store data to a file. First, a constant ASCII value from the DCC was recorded (see example output below). This was a double (16 bit) value, thus the repeating two numbers.

```
Terminal Hex log file
Date: 12/21/2003 - 1:34:39 PM
---------------------------------------------
33 35 33 35 33 35 33 35 33 35 33 35 33 35 33 35
33 35 33 35 33 35 33 35 33 35 33 35 33 35 33 35
```

Then data from the front wheel, rear wheel, and test platform was collected as 8 bit values. The sample data below shows the output of the front wheel being spun by hand while the other inputs remained at zero because they were set to send 0's from the car. The numbers represent the number of encoder tics counted over a 1ms time interval.

```
Terminal Hex log file
Date: 1/14/2004 - 12:48:08 PM
---------------------------------------------
05 00 00 06 00 00 06
00 00 06 00 00 06 00 00 06 00 00 06 00 00 06 00
00 06 00 00 06 00 00 06 00 00 06 00 00 06 00 00
```

Then to get an idea of what a good sampling rate would be, the maximum velocity of the rear wheel was measured with the car suspended. If sampling was too fast, only 0's and 1's would be recorded, too slow and very large values (perhaps greater than 8 bits) might be recorded but trends could not be analyzed. The result, with the car running on a 3000 mAh Ni-MH battery (about 8.3 volts), was a max tic count of 17 tics per sampling rate of 1ms. Since an 8-bit value was capable of 255 a 5ms sampling time was not too slow and with a thousand data points per second a smooth curve could be obtainable.

The next objective was to be able to manipulate and plot the data results. A Matlab code was then written to communicate, manipulate (if necessary), and store the data from the DCC (Appendix D). Using the Follower Code, data from the front wheel was collected and plotted (Figure 3.6).

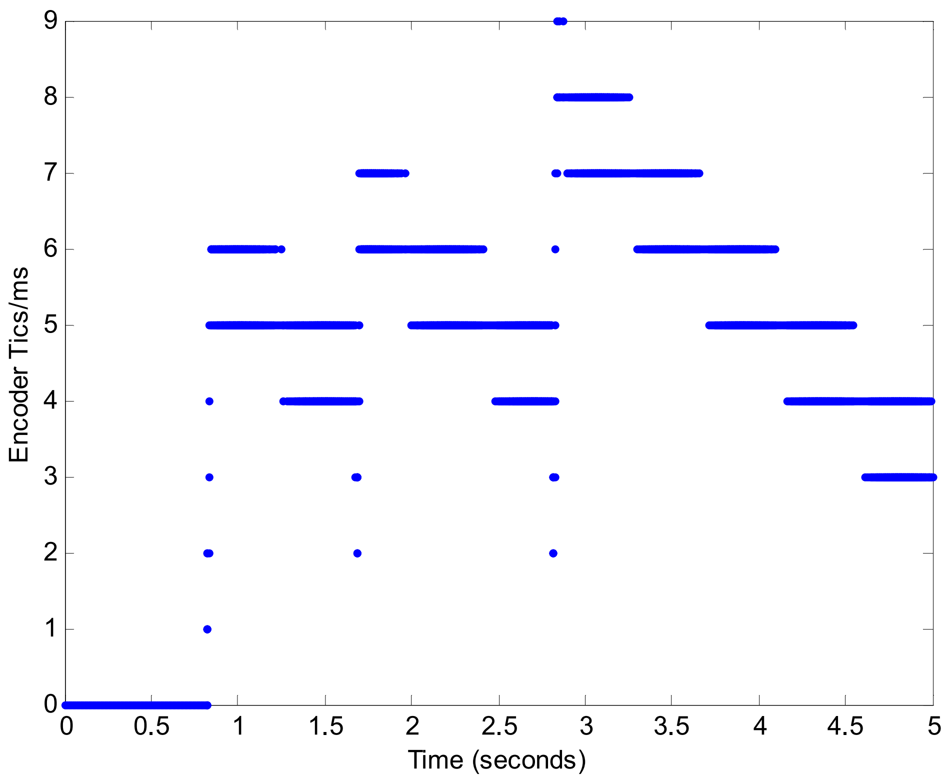


Figure 3.6: Result of data collected at 1ms intervals using the Follower Code

Originally the DCC was just going to be used to collect and analyze values of λ. When the control code did not perform as expected, it was obvious having the actual value of the control value 'u' from the car to compare with that of the computer simulation would be very useful (see Appendix E for DCC code). Therefore, the 10 bit PWM and 1 bit direction signal were added to the DCC. Because the lower two bits of the PWM signal and direction bit were sent together, to save data bytes sent, it became necessary for the Matlab code to untangle and recombine the bits received. For this, the upper 8 bits of the PWM signal (called UprBit in the Matlab code) were shifted to the left two bits and placed in a variable of size double. The lower two bits were filtered out from a separate 8 bit value called LwrBit and concatenated to the rest of the PWM signal. The direction bit was filtered and tested to determine the sign of the 10 bit PWM signal.

Chapter 4

Test Platform Development

The next physical construction hurdle was to devise a way to monitor the vehicle's performance without risk of damage.

4.1 Initial Platform

Before the idea of a platform to measure the car's velocity was determined, several other ideas were entertained. One idea was to have two photo gates to measure the car's average acceleration and another was to have several photo gates to get a better interpretation of the car's acceleration curve. Yet another idea was to video the car. Having the distance marked and knowing the frames per second, the acceleration could be crudely calculated. All three ideas had the risk of the car being damaged by an uncontrolled collision and the acceleration curve would not be measurably accurate. The idea was then suggested to place the rear wheels on a drum and from that, the test platform idea was launched.

The initial platform was merely the left side of the car's tires (side with encoders) setting on two Lego rubber tires with the right side supported by Lego blocks. The Lego tires were connected by two rubber bands, allowing the rear wheel to transpose its energy to the front wheel. This was only an initial test platform to get an idea of how well the traction control algorithm was functioning. As rubber bands stretch, later they were replaced by Lego chains (see Figure 4.1). Since many of the design parameters were not fully represented (example: mass of car, number of driving wheels …) this did not present a viable solution.

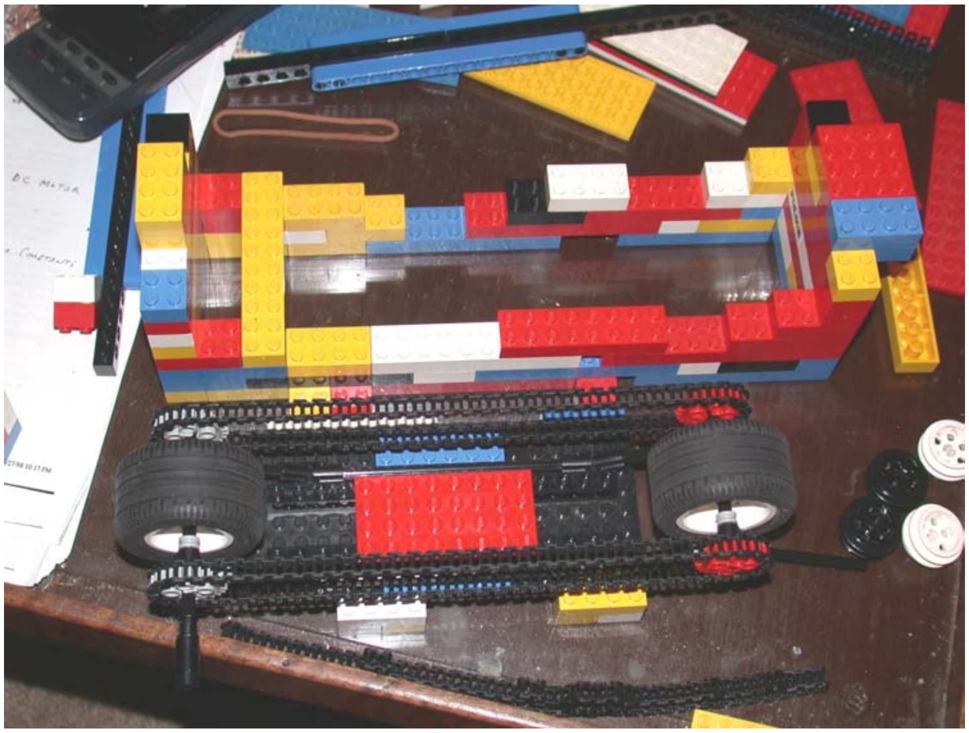


Figure 4.1: Initial test setup using wheels on one side

4.2 Different Platform Setups

Once the code and car were ready for full testing, a proper test platform needed to be created. Legos were selected because of their manufactured precision, versatility, availability, and cost. The next platform (Figure 4.2) was based on a conveyer type setup to try and best simulate a road. In the picture, the lower conveyer was removed to show the supporting structure.

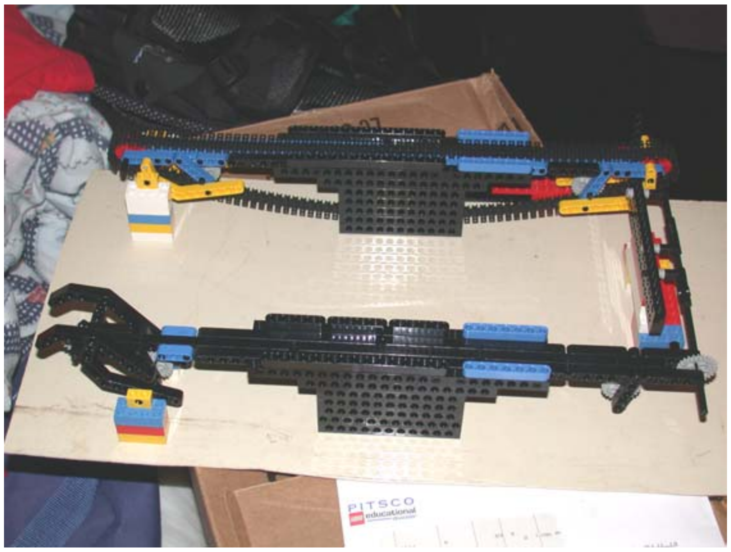


Figure 4.2: Updated test platform using conveyor belts

Due to the weight of the car, the dynamics of the gears, and the construction of the conveyor Legos, there was a great deal of friction. This coupled with the gripping ability of the tires caused there to be little or no difference between full speed and traction control. It was then determined to go back to a wheel type design for the car to set on but use wheels that were not malleable, thus had a lower coefficient of friction. Therefore, wheels were created from the gear and conveyor belt parts (Figure 4.3) and the Data Collection Circuit was added, but collection of the control variable 'u' between the car and data collection circuit had not been added at this point in time.

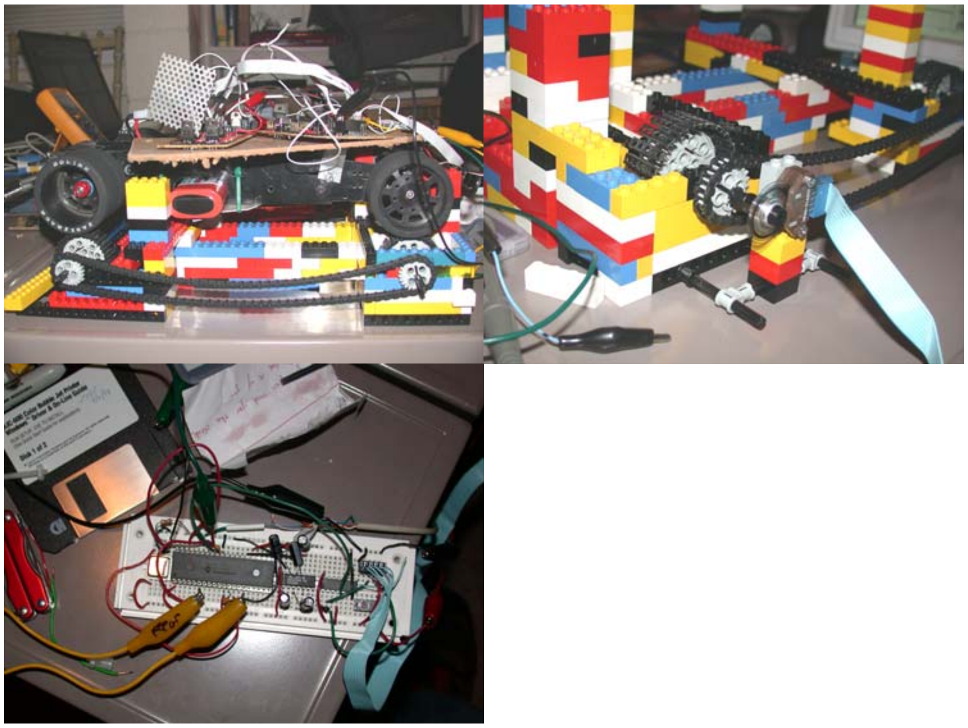


Figure 4.3: Reduced friction test platform design with Data Collection Circuit (DCC) added

4.3 Final Platform Setup

The car still had too much grip, though friction of the test platform was greatly reduced. Next the conveyor belt idea was tried again but with the center of the conveyor links fitting between two rails and the gearing was adjusted to just two big gears (Figure 4.4); this greatly reduced the friction, but the tires still had too much grip. So duct tape was placed around the tires. Along with the friction of the conveyor belt reduced, the car finally began to slip… a lot (Communication of the car's control value was also added). Eventually more weight needed to be added to the rear of the car to increase traction, possibly due to increased friction on the test platform from wear. A battery was added because its weight had already been measured.

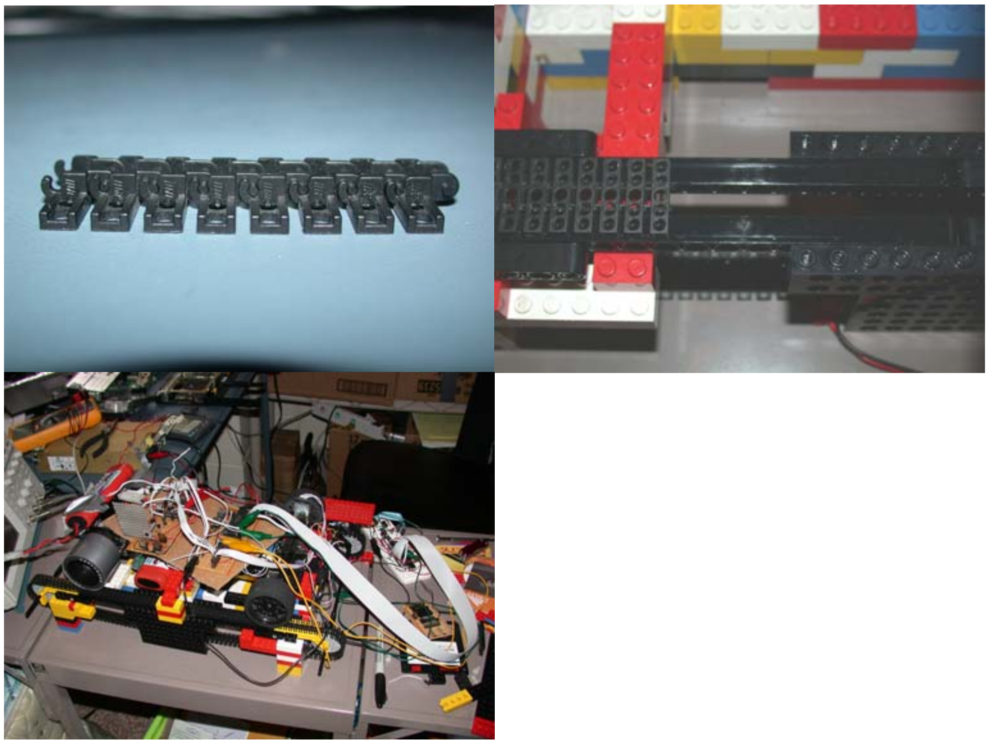


Figure 4.4: Final test platform setup

Chapter 5

Code and Development of Parameters

This chapter covers processes and analysis of the control value ranges and obtaining parameters of the system using Simulated Annealing.

5.1 Control Code and Analysis

Using the measured, calculated, and rough estimates for parameters, the complete control algorithm was implemented on the car. The result was that the rear tires were chattering. To help understand the possible ranges of outputs of the control algorithm and knowing the full range of possible inputs from the front and rear wheels, the control code in Matlab was modified to give a 3-D plot of the possible outputs (Figure 5.1) (Appendix F for code).

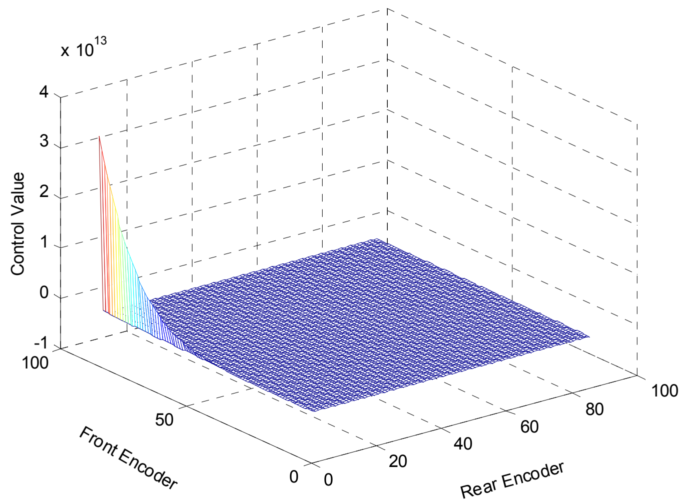
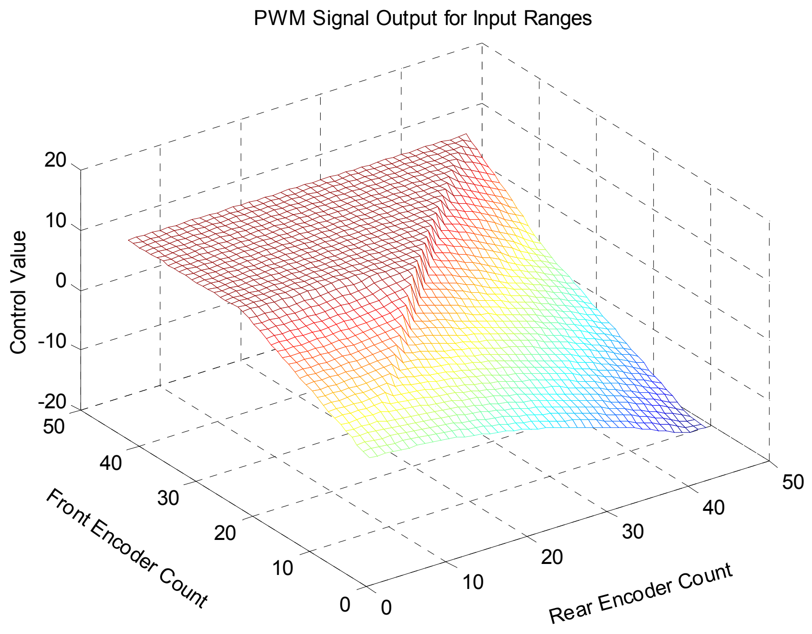
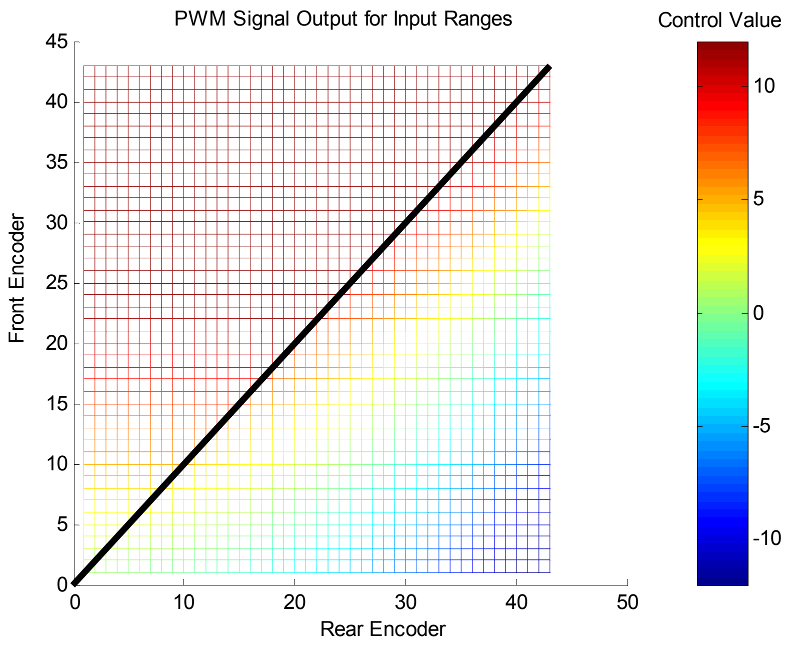


Figure 5.1: 3-D generated output of control variable from possible range of inputs

Obviously, the control's possible output values far exceeded the ± 1023 range of the PWM signal. Using the same program and manipulating the output of u and other variables, a plot was generated that closely resembles what might be expected (Figure 5.2 a & b). This plot can be used to explain the expected behavior of the system.



(a)



(b)

Figure 5.2: Possible Input/Output system response

Figure 5.2b is the plot as seen from looking down. The black diagonal line is the range of values where the velocity of the front wheel equals that of the rear. To the left of the black line, the front wheels are moving faster than the rear wheels, thus more power should be applied to the motor and even greater still when at higher speeds. To the right of the black line is when the rear wheels are moving faster than the front wheels. As explained earlier, while just to the right of the black line, some slip would be desirable (15 to 20% on a full sized car), so power would need to be increased, not decreased as indicated by this illustrating plot. This concept would form a channel along the diagonal. However, if there is too much slip, power to the rear wheels should be reduced. The greater the difference between the rear wheel and the front wheel, the bigger the decrease in power to the motor. This conceptual plot can now be added as a tool to recognizing when the system may or may not work, without implementing on the car and thus saving time.

5.2 Leader Code and Analysis

With the results of the control algorithm so far off the various combinations of uncertain and unknown variables would need to be compared with actual results from the car until a minimum error was found. In other words, the system needed to be parameterized. Before that was done, certain aspects of the data collection system needed to be analyzed. Also, because the batteries drained too quickly and introduce yet another very influential dynamic, a 12 volt 10 amp power supply was used from this point. However, even the power supply was not ideal as its output would drop after about the first 6-7 seconds. Fortunately this was generally enough time for the car to accelerate.

To simplify the system, a program called Leader Code was written to accelerate the vehicle (see Appendix G). This code simply increased the value of the PWM signal by one if the rear wheels' angular velocity was less than 5 encoder counts faster than the front wheels', otherwise the PWM signal would be decreased by one. The DCC sampling time was also adjusted to 0.0049999 seconds, the same as the robot car. For parameterizing the car with the Leader Code, three sets of data would need to be collected, the front encoder, rear encoder and PWM values. As the car was only able to transmit one set of data, it would be necessary for the DCC to collect the front and rear wheel encoder values separately while receiving the actual PWM values from the car. For this reason an analysis was done to compare the encoder values collected from the car to the values collected from the DCC on the rear wheels. This meant that the control value collected and sent from the car was replaced with the 8-bit value from the counter of the rear encoder collected by the car. If the independent values from the car and DCC were not close another means of collecting data would have to be found. The rear encoder data collected from the DCC and the car were then recorded and compared. Figure 5.3 and Figure 5.4 show the results from the car and the DCC respectively of the rear encoder values using the Leader Code to accelerate the car.

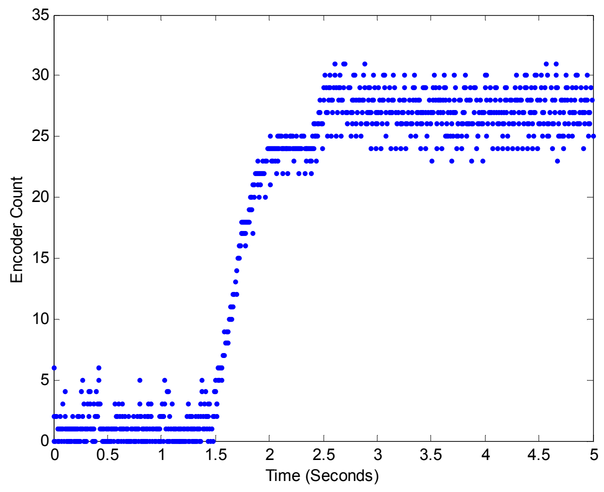


Figure 5.3: Rear wheel encoder count data collected from car.

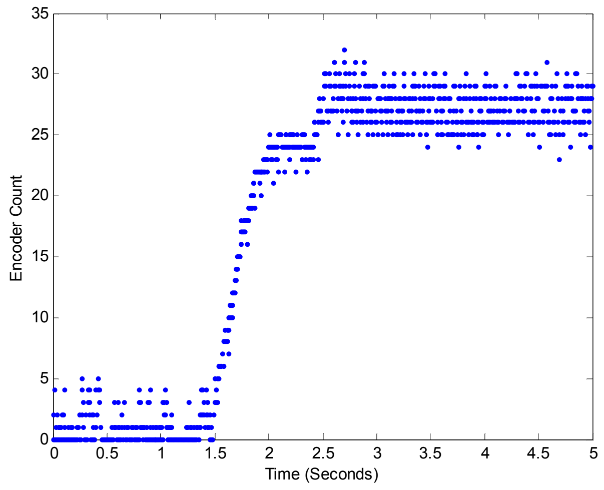


Figure 5.4: Rear wheel encoder data collected by Data Collector before shifting data

Figure 5.5 shows the difference between the two raw data values collected by the car and the DCC. Since the DCC actually begins collecting data before the robot car, the data from the DCC was then shifted eliminating leading 0's and then compared (see Figure 5.6). The maximum error after the shift was slightly less but not significantly different.

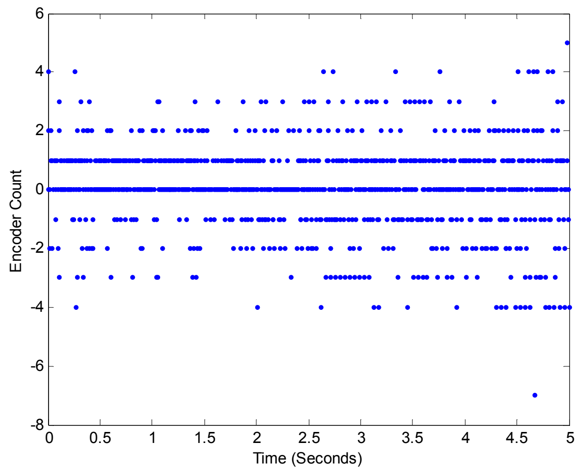


Figure 5.5: Difference between Car's rear wheel values and Data Collector's values before shift

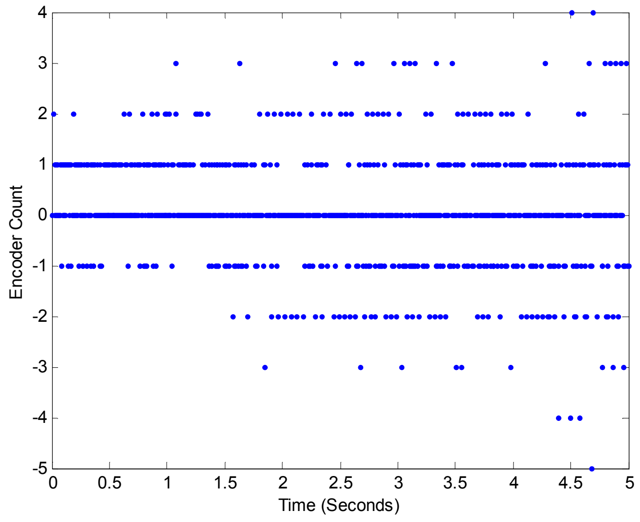
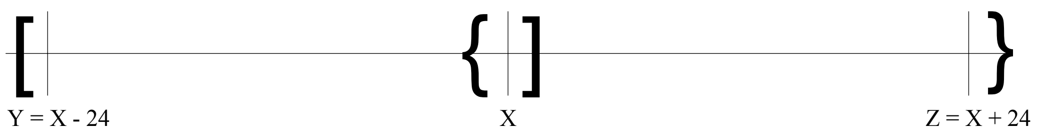


Figure 5.6: Difference between Car's Rear Wheel values and Data Collector's values after shift

From the original analysis, even without shifting the data, parameterizing the system should be possible. However, further analysis was continued in case averaging might have been needed to improve results.

To help view and understand the average error and difference between shifting and not shifting of the data, a Matlab program was written that averaged the data points except at the ends (see Appendix H for code with averaging on ends of data). The averaging was done by selecting an odd number of data points to average, in this case 51, and then taking one less than half that number (25) and average one less than that many data points before and after the selected data point with the actual data point being counted twice. This gives a slightly weighted value to the selected data point (Figure 5.7). The plot in Figure 5.8 shows the raw un-shifted plots from the DCC and robot car respectively followed by the plots of their mean values. The difference of the mean values was then also determined (Figure 5.9). The same plots were generated for the shifted data (Figure 5.10 & Figure 5.11). The difference of the mean shifted data and mean non-shifted data was done for comparison (Figure 5.12). So how was this data interpreted? Using this setup would generate some error when comparing the responses from the robot car and the simulation, but if necessary to improve accuracy, the data from the DCC could be shifted to give slight improvements, and if too much error was involved, it might be possible to then average the values for better results.



$$\text{Mean} = \text{Sum}(Y_{\text{range}} \ \& \ Z_{\text{range}}) \ / \ (Y + Z - 1)$$

Figure 5.7: Method for taking mean value of data points

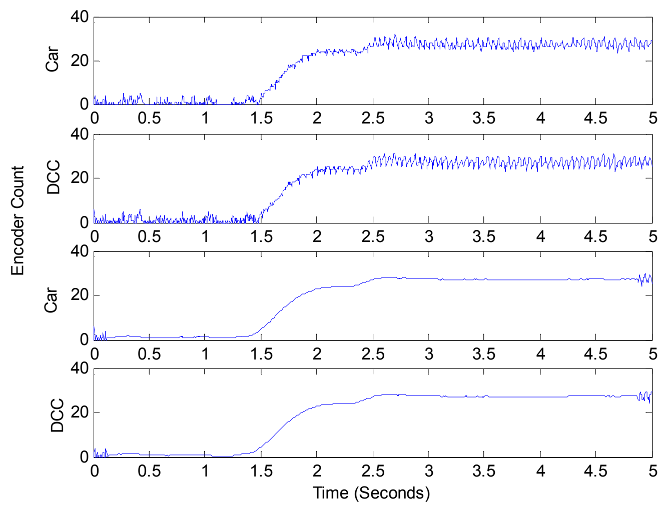


Figure 5.8: Rear Encoder plot from car, and DCC before shifting DCC data, followed by Rear Encoder values of Car and DCC after averaging.

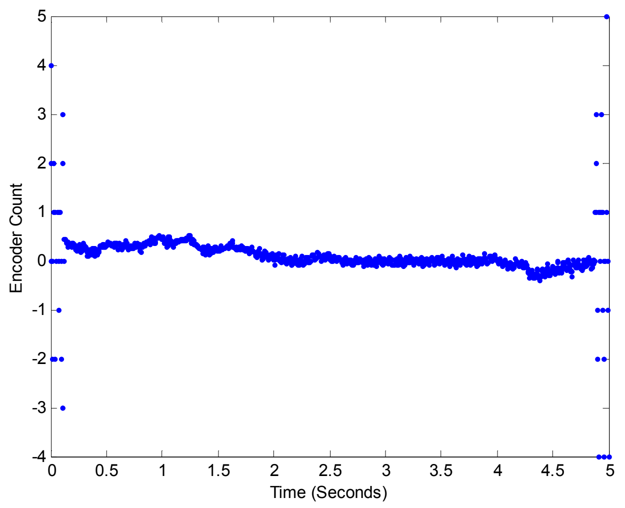


Figure 5.9: Difference of mean values before the rear encoder data was shifted

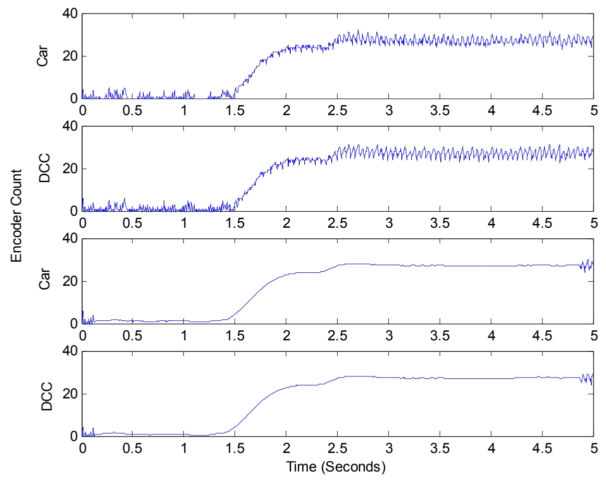


Figure 5.10: Rear Encoder plot from car, and DCC after shifting DCC data, followed by Rear Encoder values of Car and DCC after averaging.

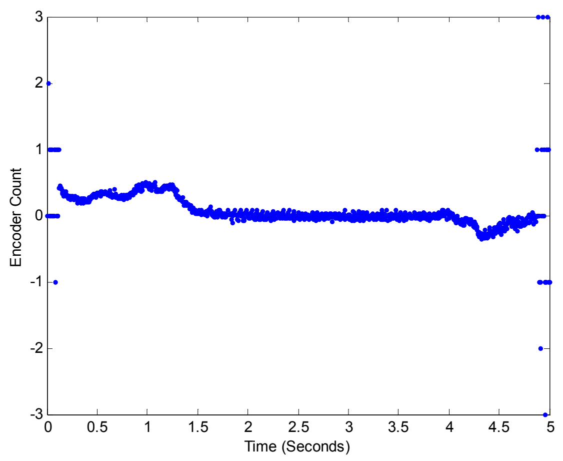


Figure 5.11: Difference of mean values after the rear encoder data was shifted

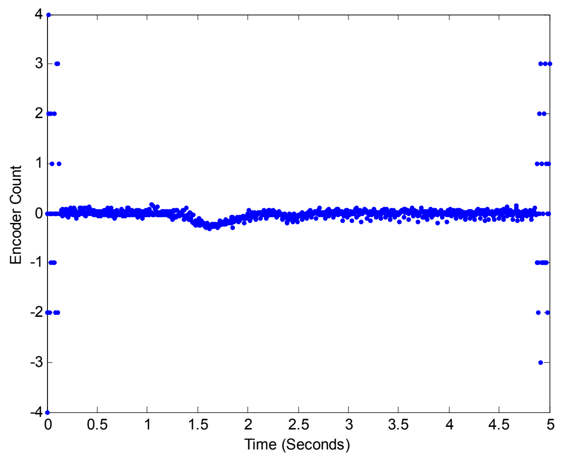


Figure 5.12: Error between shifted and un-shifted mean curves

5.3 Simulated Annealing

Because there were up to seven less than completely known values ($\lambda_{desired} : Lamda\_d, \mu_{Peak} : MueP, \lambda_{Peak} : Lamda\_P, c : c, N_V : Nv, F_W : Fw, J_W : Jw$) and it would be impossible to have an iterative search for the system parameters that yielded the best results, a random search method called Simulated Annealing was implemented[11, 12].

Simulated Annealing is a random search engine that picks a point on the function surface, governed by the sets of parameter unknowns, to be tested for a minimum difference (in this case). Before this new point is accepted as the lowest value, one of two governing criteria must be met. The first test is if the latest point has the lowest value. The second, in reference to the annealing process of metals, is dependent on a formula based on the present energy/temperature level of the process. If the energy level is high, it is very likely the new value will be accepted even though it may not be the smallest value; however, as the energy level is lowered, it becomes increasingly unlikely that the newest point will be accepted if it is not the lowest value. The process then repeats itself until a minimum criteria or some other limit such as time or maximum iterations is met. There are very many variations on this concept such as what types of random search generators are used or methods to find local minimums after each jump (see Appendix I for Simulated Annealing Code).

Knowing the surface was slippery, low ranges for $\lambda_{desired}$, μ_{Peak}, and λ_{Peak} were selected. Because the test platform had friction associated with it, the coefficient of wind drag as represented by the variable 'c', was initially given a wide range. The value of inertia, J_W, was given a wide range from 8.416425e-6, a factor of 10 smaller than that calculated, to 0.011 [13]. F_W, was also given a broad but low value range initially.

To determine the error between the actual system and the simulation, the recorded values of 'u' from the car over a certain time interval were input into the states along with the values of the front and rear wheel velocities collected by the DCC. Using these initial parameters, the derived equations were used to calculate the wheel velocities for the next moment in time. The following formula was then used to calculate error:

$$\sum (x(1)-x1)^2 + (x(2)-x2)^2 + (\lambda(1)-\lambda 1)^2 \tag{5.1}$$

Where x(1) and x(2) represented the car's calculated front and rear wheel velocities, respectively. While x1 and x2 were the measured front and rear wheel velocities, respectively. The value of $\lambda(1)$ came from simulation values while $\lambda 1$ came from measured values. The assumption made here was that the physical system was measured sufficiently fast enough that errors due to delays were at a minimum.

Initially the minimum total error of λ from this process was in the low 20's, however the error from the rear and front wheels was in the millions. To minimize the error from the front wheel, the small difference in radius from it and the rear wheel was then taken into account. This brought the summed error of the front wheel down to less than 0.004 (Note: because Simulated Annealing jumps around, this is only a sample value of what was typical).

To try and determine what was causing the error in the rear wheel values, the annealing program was modified to keep track of the simulated rear wheel results. When plotting the data collected from the rear wheel encoder and simulated rear wheel encoder over a range that was generating the largest error, it was observed the plots were the same but the scale was different (see Figure 5.13).

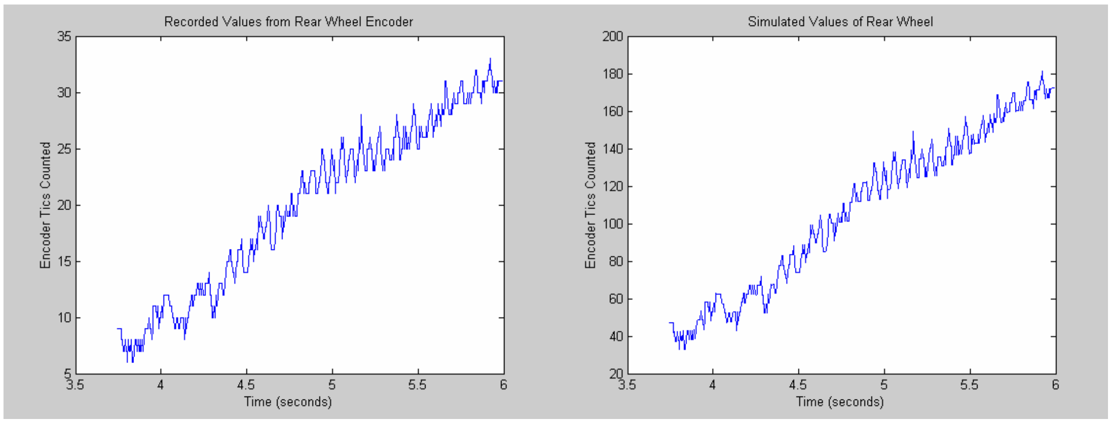


Figure 5.13: Measured and calculated results of rear wheel velocities

The ratios from the largest and smallest values were then compared. The ratios were almost the same. Since J_W was the only value in the denominator of the state for the rear wheels the upper range of J_W was increased into the hundreds, far exceeding calculated values. The annealing program was run terminating with a combined error of 0.0461. Considering J_W was several magnitudes too large to generate a low error, some other parameter had to be out of range. Considering the plot in Figure 2.4 was based on data from pneumatic tires on full sized automobiles and the tires on the robot car were stiff foam rubber surrounded by Duct Tape, the idea was considered that the lower range of the $\mu(\lambda)$ curve might not be realistic for this physical system. For this reason the ranges of μ_P, λ_P, and λ desired were extended in the lower range. This yielded total errors in the one thousandths! With good results from the Simulated Annealing program several runs were made yielding two sets of values that were used in simulations.

λ_d = 0.0018
μ_p = 0.0745
λ_p = 0.1551
c = 0.005
N_V = 1.3009
F_W = 0.4507
J_W = 0.1088
R_a = 1.506
K_e = 0.0159

Table 5.1: SA Results

λ_d = 0.0814
μ_p = 0.0102
λ_p = 0.0105
c = 0.0019
N_V = 1.4585
F_W = 0.1951
J_W = 0.0923
R_a = 1.2229
K_e = 0.0139

Table 5.2: SA Results

Notice in Table 5.2 that the λ_d (desired λ) was greater than the λ_p value. As described in [1], this is a region where control effort becomes significantly high and is very unstable, therefore the result of λ_d was lowered to less than 0.0105 when Table 5.2

results were used. Though not all results are within expected ranges, as these values yielded the closest match (lowest error) to collected data from the car by less than 40 times other parameter values, these values were used on the car.

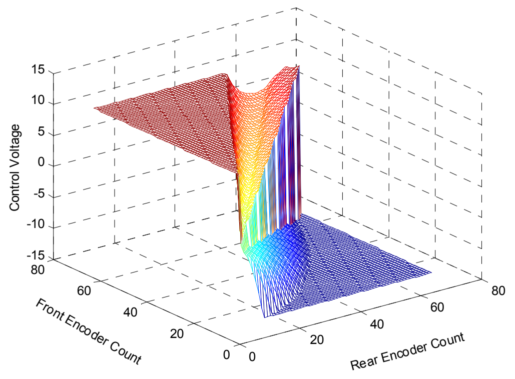
Chapter 6

Results

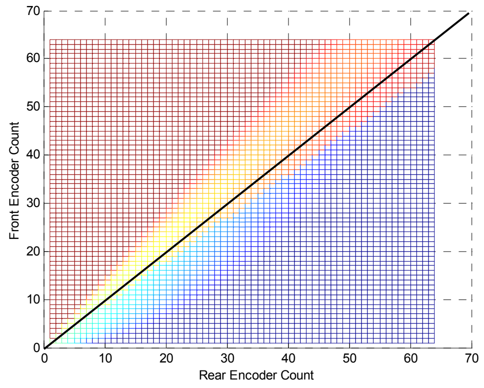
To improve the system response, processing time, and other dynamics, some parameters were adjusted on the simulation plot and car's code. After these adjustments, the measured data was then collected, plotted, and compared to simulation results.

6.1 Adjustments of Control Output

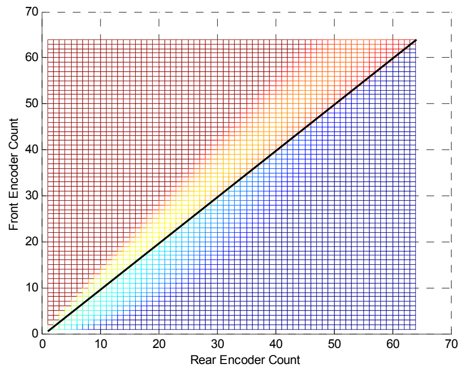
The final step was to adjust the control gains and the $\mu(\lambda)$ curve to give the best response. Initially the Simulated Annealing program was modified to find the optimum gain values using the control value and reduced chattering as the criteria. The 3-D plot previously developed was used to get an idea for reasonable value ranges for the annealing program. Different $\hat{\mu}(\lambda)$ curve approximations were also tried, but useful results were not obtained from the program. However, during the process of determining the range of gains and adjustments to the $\hat{\mu}(\lambda)$ curve from a 10th order polynomial to 6th order polynomial, a plot with potential (Figure 6.1 a & b) when using values from Table 5.2 produced promising results. With some adjustment to the gain η, the values were tried on the simulation. The results were also somewhat promising (Figure 6.2).



(a)

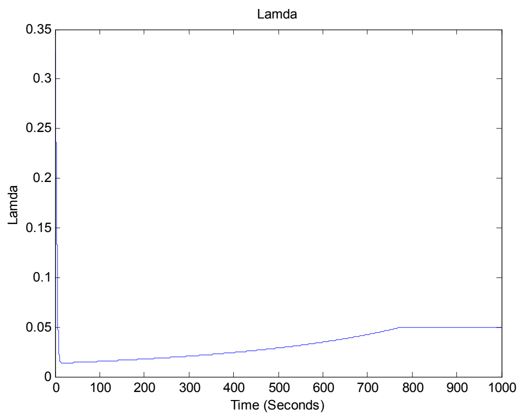


(b)



(c)

Figure 6.1: (a) 3-D plot with λ_d =0.1 (b) λ_d =0.1 from above (c) λ_d =0.01 from above



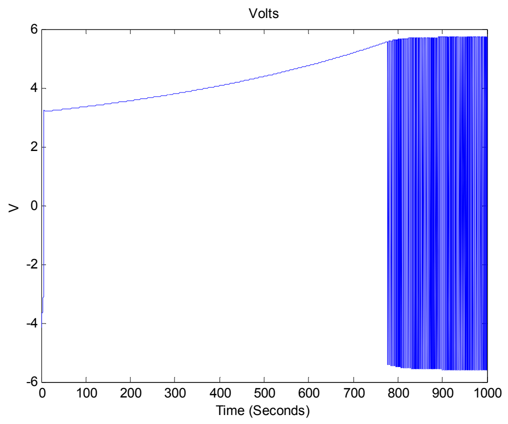


Figure 6.2: λ and controller voltage plots from computer simulation, respectively

The values were then run on the car with some success. The problem on the car was going to be that it would be very susceptible to chattering/instability when reducing λ_d

below 0.1 (Figure 6.1c). This should not be totally unexpected. As shown in Figure 2.3, the closer to the origin the closer the system is to breaking, which has a different set of governing dynamic equations. Also, the response of λ from the measured values varied very little if any. As previously mentioned, a channel might be expected, but this channel was right on the edge of the sliding surface/diagonal. A boundary layer was attempted but this made the system response worse as chattering occurred on the edge of the boundary layer and reduced the edge of the channel where the channel was too narrow. To better center the channel and increase the range of stability for λ_d a ratio was multiplied by the front wheel input value. Starting with the ratio of the rear wheel diameter divided by the front wheel diameter and making adjustments until the channel was better centered, led to a more stable system. The resulting ratio was the rear wheel diameter of 2.524 divided by 2.2035 (2.4035 = front wheel diameter). With this adjustment and an improvement on the $\hat{\mu}(\lambda)$ curve that decreased the car's processor's calculation time and provided a much better fit than a 6th order polynomial, the car was tested again with improved performance (Figure 6.3). From Equation (2.15) comes the new equation for $\hat{\mu}(\lambda)$:

$$\hat{\mu}(\lambda)=\left(\frac{2\mu_{p\max}\lambda_{p\max}\lambda}{\lambda_{p\max}^2+\lambda^2}+\frac{2\mu_{p\min}\lambda_{p\min}\lambda}{\lambda_{p\min}^2+\lambda^2}\right)\div 2 \tag{6.1}$$

where $\mu_{p\max}$ =0.94, $\mu_{p\min}$ =0.008, $\lambda_{p\max}$ =0.3, and $\lambda_{p\min}$ =0.008 was the improvement made to the $\hat{\mu}(\lambda)$ curve (Appendix J for robot car code).

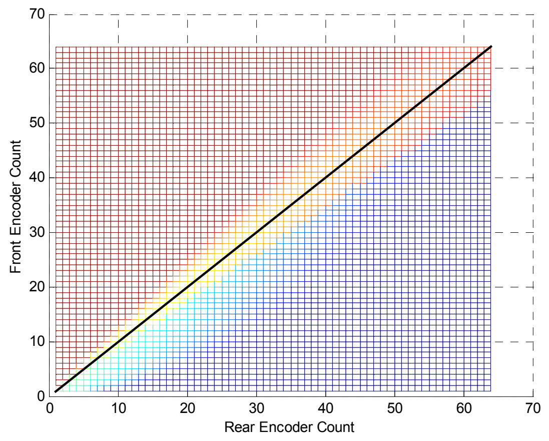


Figure 6.3: λ_d **=0.01 from above with adjustment to center the channel**

6.2 Test and Measurement Results

Using the Data Collector Circuit to record the front, rear wheel encoder values, and PWM output, plots were generated for analysis. For λ_d=0.1 results refer to Figure 6.4 thru Figure 6.7, and for λ_d=0.05 plots refer to Figure 6.8 thru Figure 6.11.

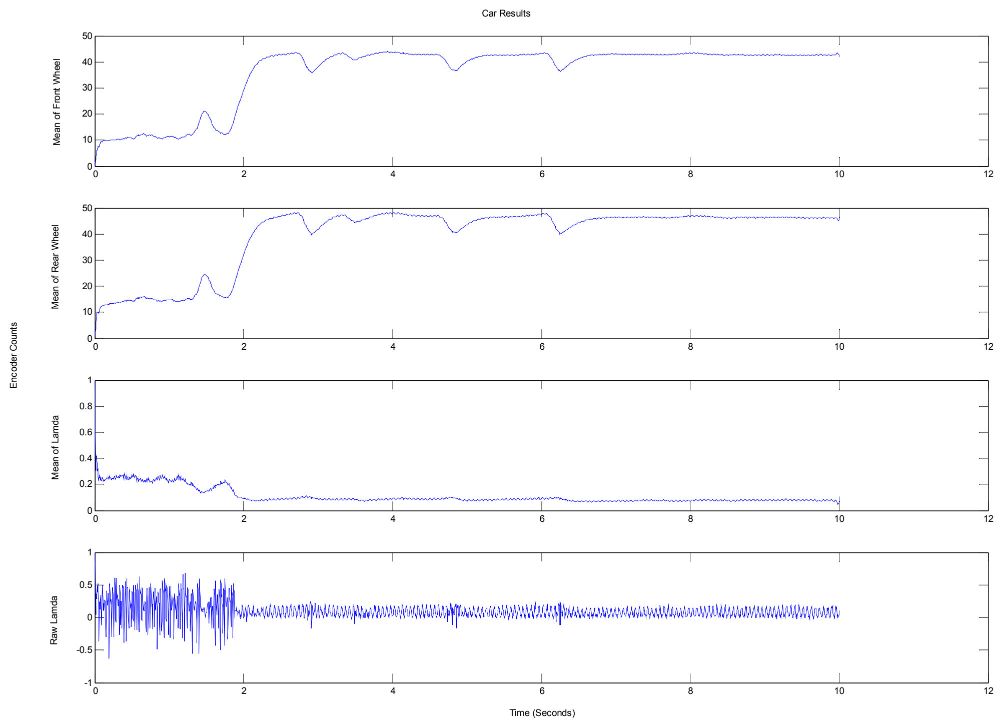


Figure 6.4: Front, rear mean encoder data, mean λ results, and raw λ results for λ_d=0.1

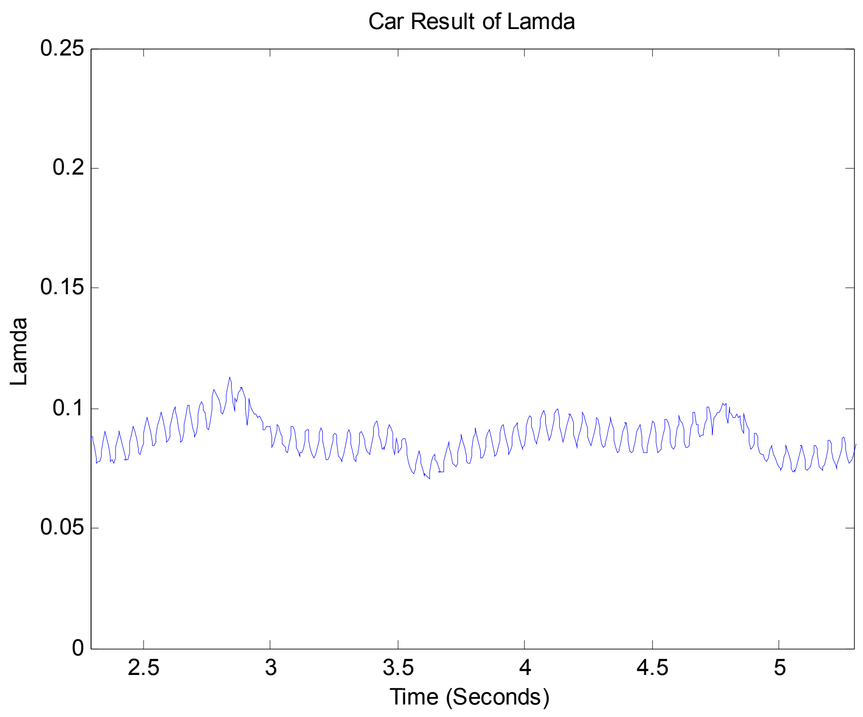


Figure 6.5: Zoomed in plot of mean λ results for λ_d=0.1

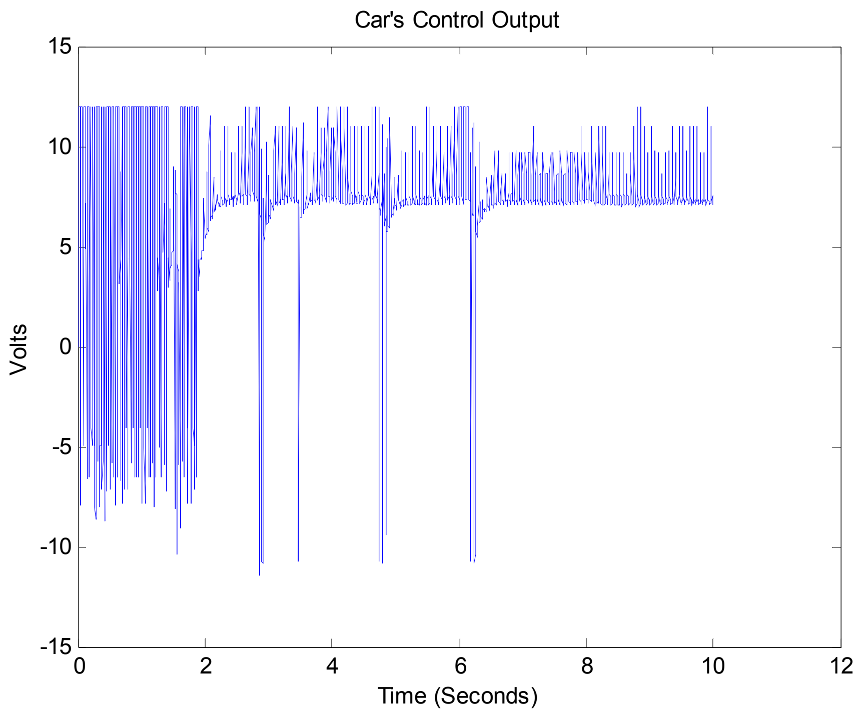
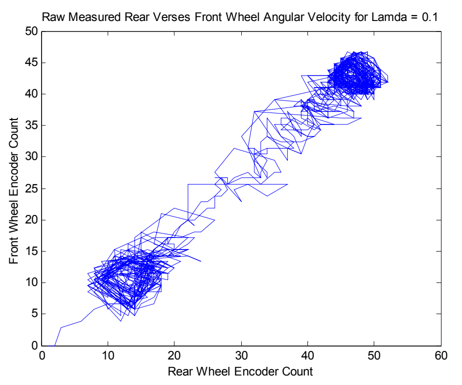
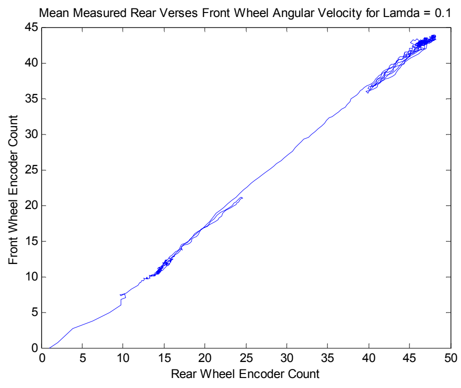


Figure 6.6: Raw PWM voltage car output for λ_d=0.1



(a)



(b)

Figure 6.7: (a) Raw measured front & rear velocities (b) Mean measured front & rear velocities for λ_d =0.1

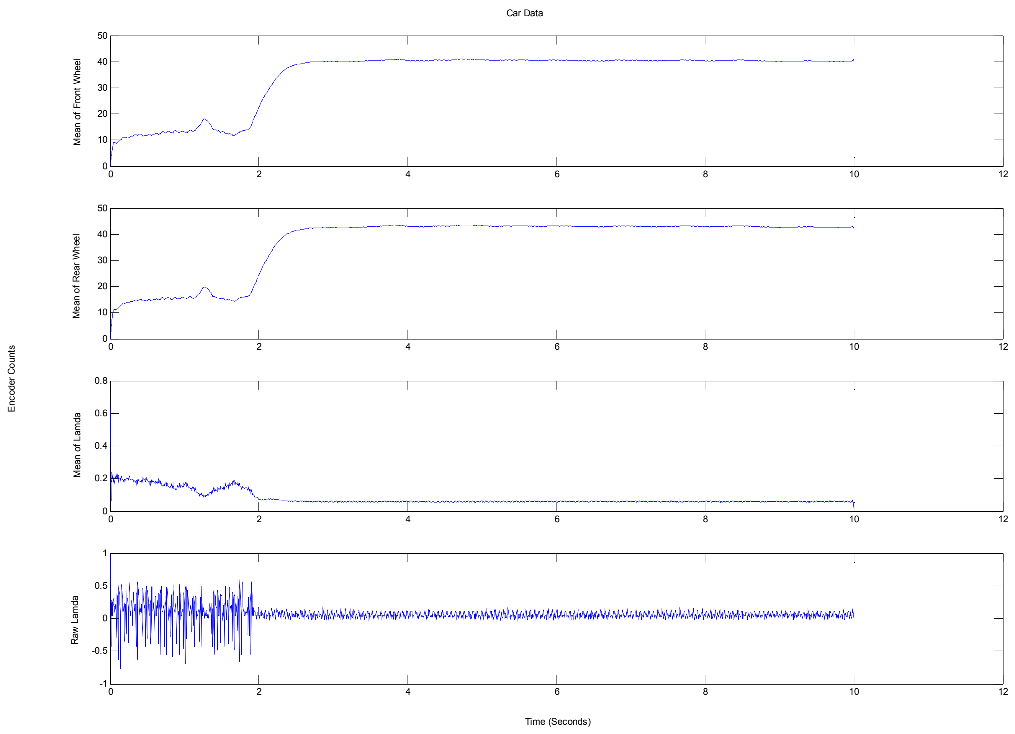


Figure 6.8: Front, rear mean encoder data, mean λ results, and raw λ results for λ_d =0.05

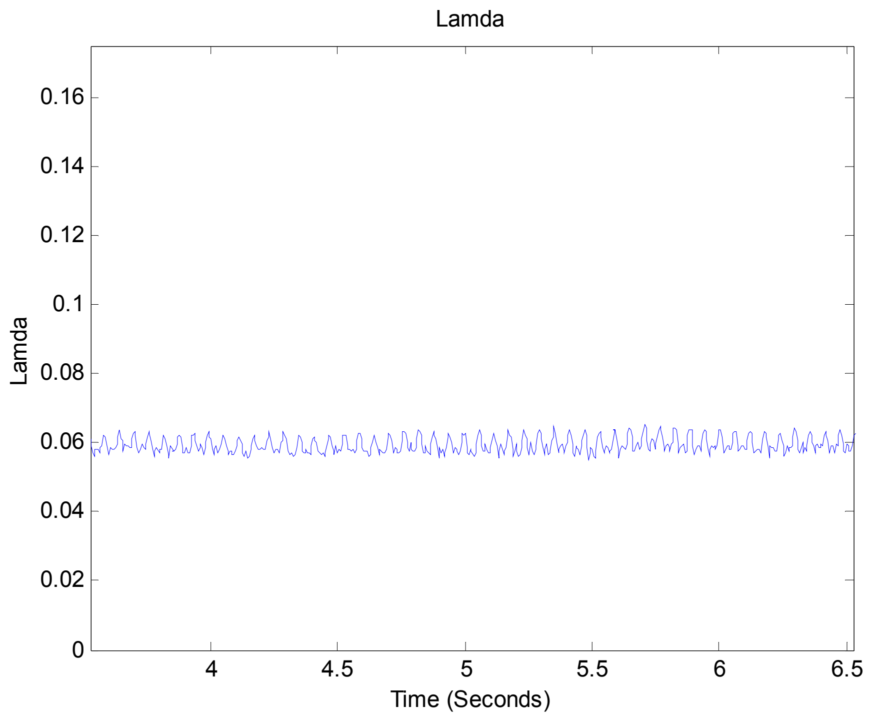


Figure 6.9: Zoomed in plot of mean λ results for λ_d=0.05

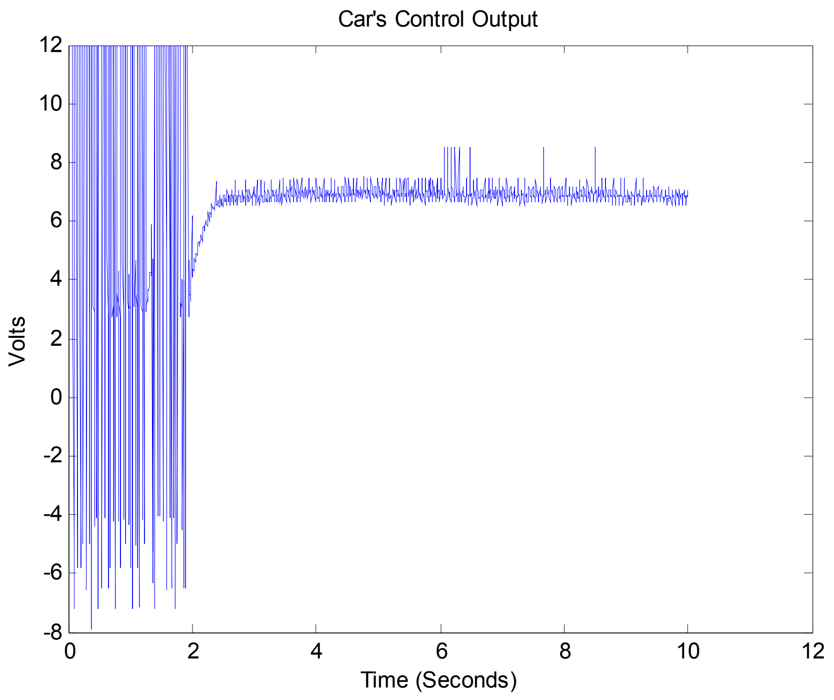
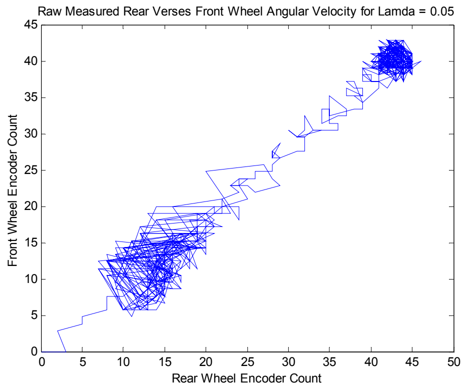
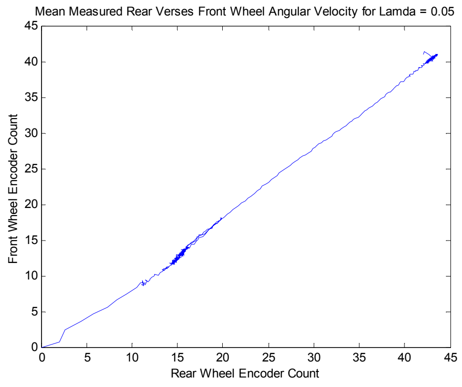


Figure 6.10: Raw PWM voltage output from robot car for λ_d=0.05



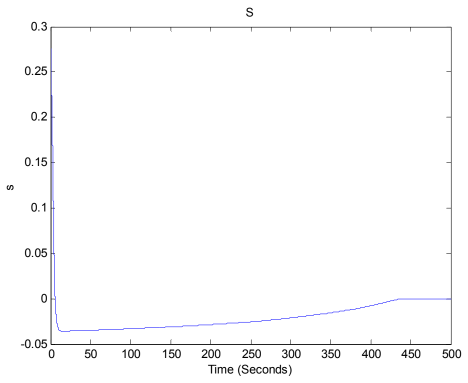
(a)



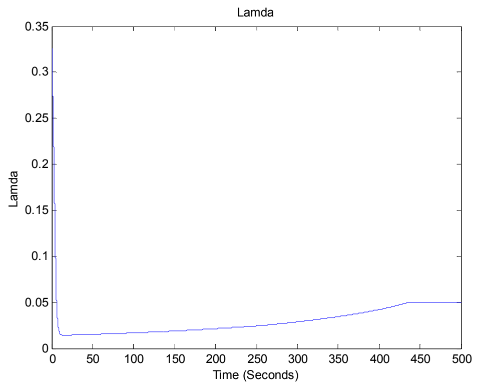
(b)

Figure 6.11: (a) Raw measured front & rear velocities (b) Mean measured front & rear velocities for λ_d=0.05

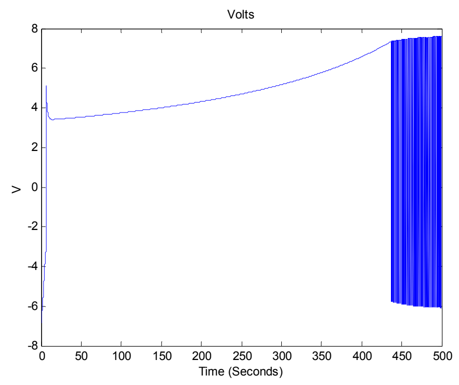
From Figure 6.5 and Figure 6.9, λ_d compared closely to what was expected. However, comparing other simulated results after the new adjustments, many values did not match accurately, such as control voltage values, velocities, and timing (Figure 6.12 and Figure 6.13 for simulation results). Considering the non-exact modeling of the car's dynamics and apparent differences in the $\mu(\lambda)$ curve from a full size car with pneumatic malleable tires compared to the foam tires wrapped in Duct Tape being used on the robot car, this should not be unexpected. Other dynamics not being modeled accurately were the friction forces in the test platform and the linearization of other dynamics such as those for the motor. With the new results, the computer simulation code was updated for comparison (see Appendix K for code)



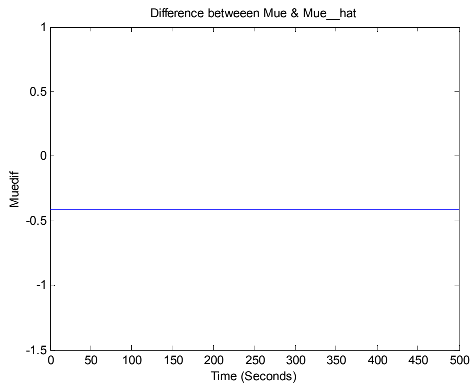
(a)



(b)

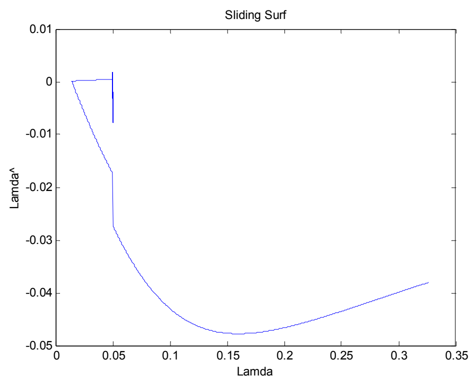


(c)

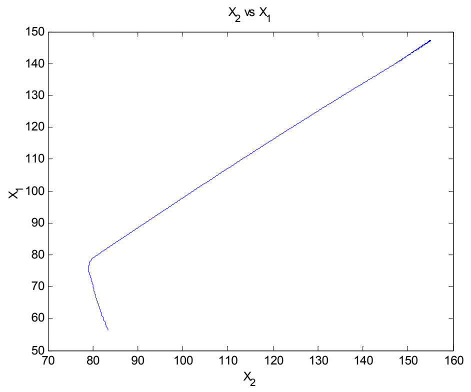


(d)

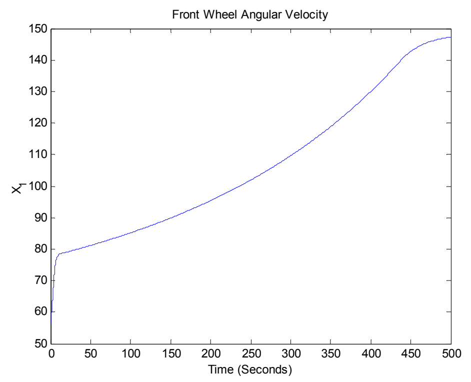
Figure 6.12: Results from adjusted simulation with λ_d =0.05



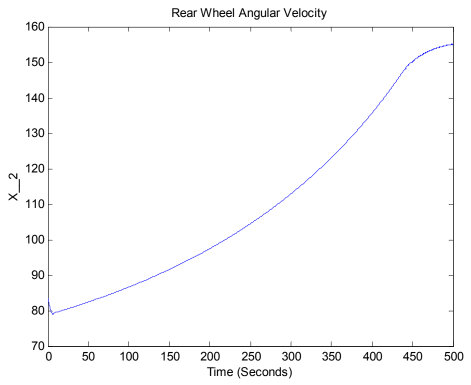
(a)



(b)



(c)



(d)

Figure 6.13: Results from adjusted simulation with λ_d =0.05

Even with these discrepancies, the results are close enough to encourage further study with improvements.

Chapter 7

Hybrid Model

Though the computer model simulates the dynamics of the car and responses of the control algorithm as if both happened continuously and to a high degree of precision, this was not the case. In reality, the processor takes time to process the car's situation and the inputs from the encoders were discrete. This was represented by the 3-D plots created earlier, and the control outputs were also integer values. The physical system (robot car) was continuous. This next chapter discusses the modeling of the system as a hybrid model. A hybrid model is a model that represents both the continuous and discrete components of a system.

7.1 Hybrid Model Analysis

Before expressing the dynamics of the robot car as a hybrid system, it is necessary to explain the model representation of the hybrid system. Using the example in Figure 7.1 [14],

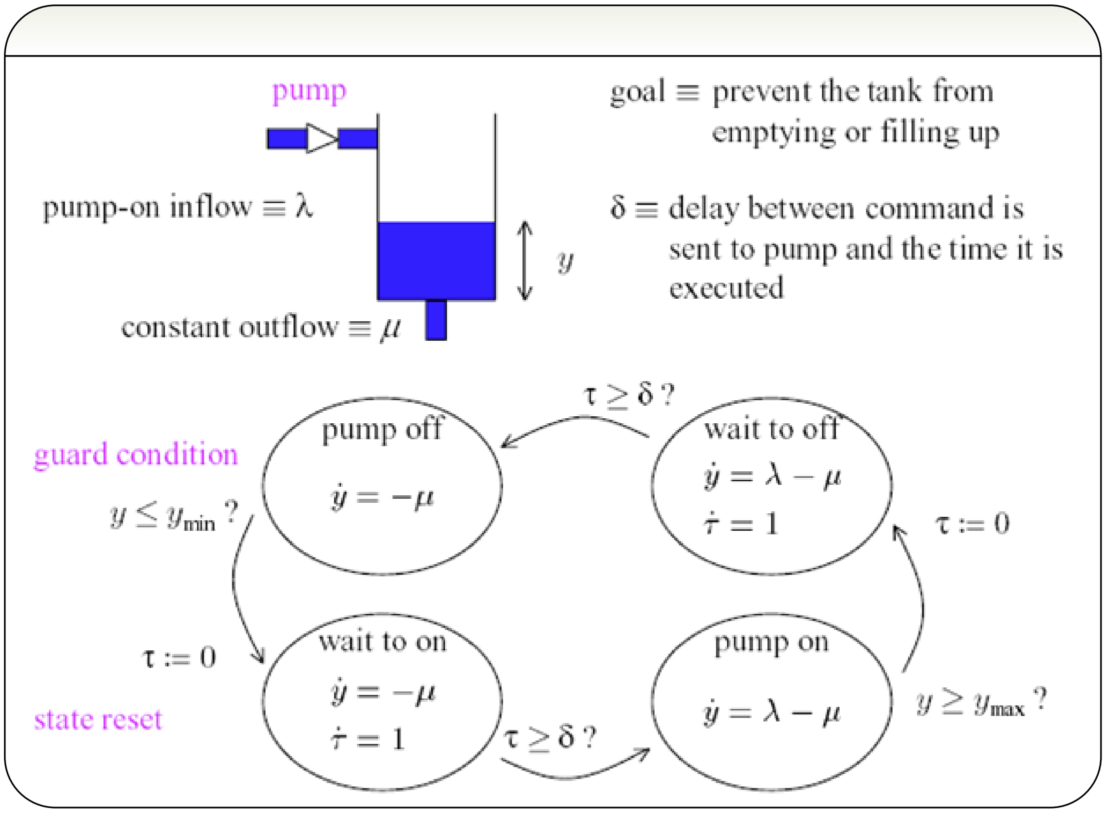


Figure 7.1: Hybrid model example of a Tank System [14]

and references [14, 15] a simple example of a tank being maintained with water within a certain range will be discussed. Just as in a State Machine each state represents an operation. In this example, there are four states: pump off, wait to off, pump on, and wait to on. For this hybrid system the depth of the tank is represented by the fluctuation 'y', which is continuous and therefore is represented in each state by ' $\dot{y}$ '. For this system, the continuous dynamics of a timer are also represented by ' τ ', where appropriate. When a state's dynamics no longer support the condition of the system, or a "guard condition" is met, an instantaneous transition in both discrete and continuous time is made to another state. If a parameter is changed as a result of the transition a "state reset" or "jump" is performed.

Before continuing with the hybrid model, the dynamics of the discrete system should first be discussed to help with clarity. Using Figure 7.2, this will be done.

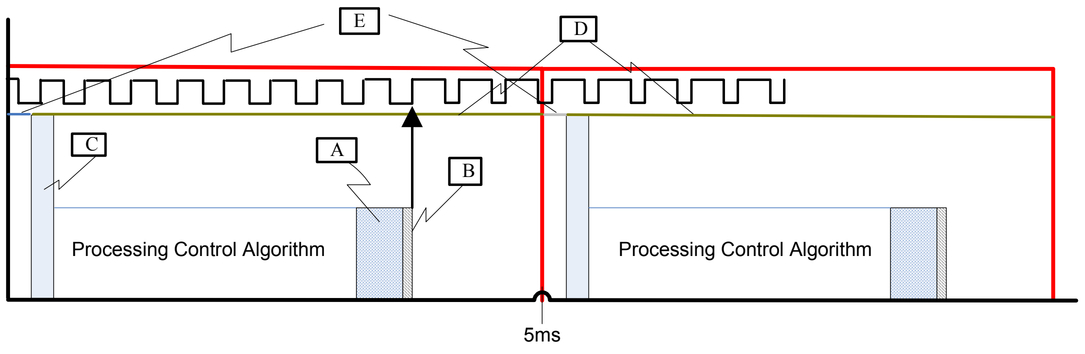


Figure 7.2: Timeline of discrete system on robot car's processor

Block 'A' in Figure 7.2 represents the variation in the processing time of the control algorithm which is always less than 5ms. Block 'B' represents the time for the PWM signal to change width. The PWM signal only changes width after a duty cycle has been completed ($1 \div 9.77 Khz$). Block 'C' indicates the time taken to update the encoder variables from the continually running counters. The section before block 'C' is where the 5ms interrupt timer is reset. Block 'C' is a source of error as the robot car is continuing to move as variables representing encoder values are updated. Block 'D' indicates the actual cycle time of the 5ms interrupts and Block 'E' is the overrun time until the interrupt timer is reset.

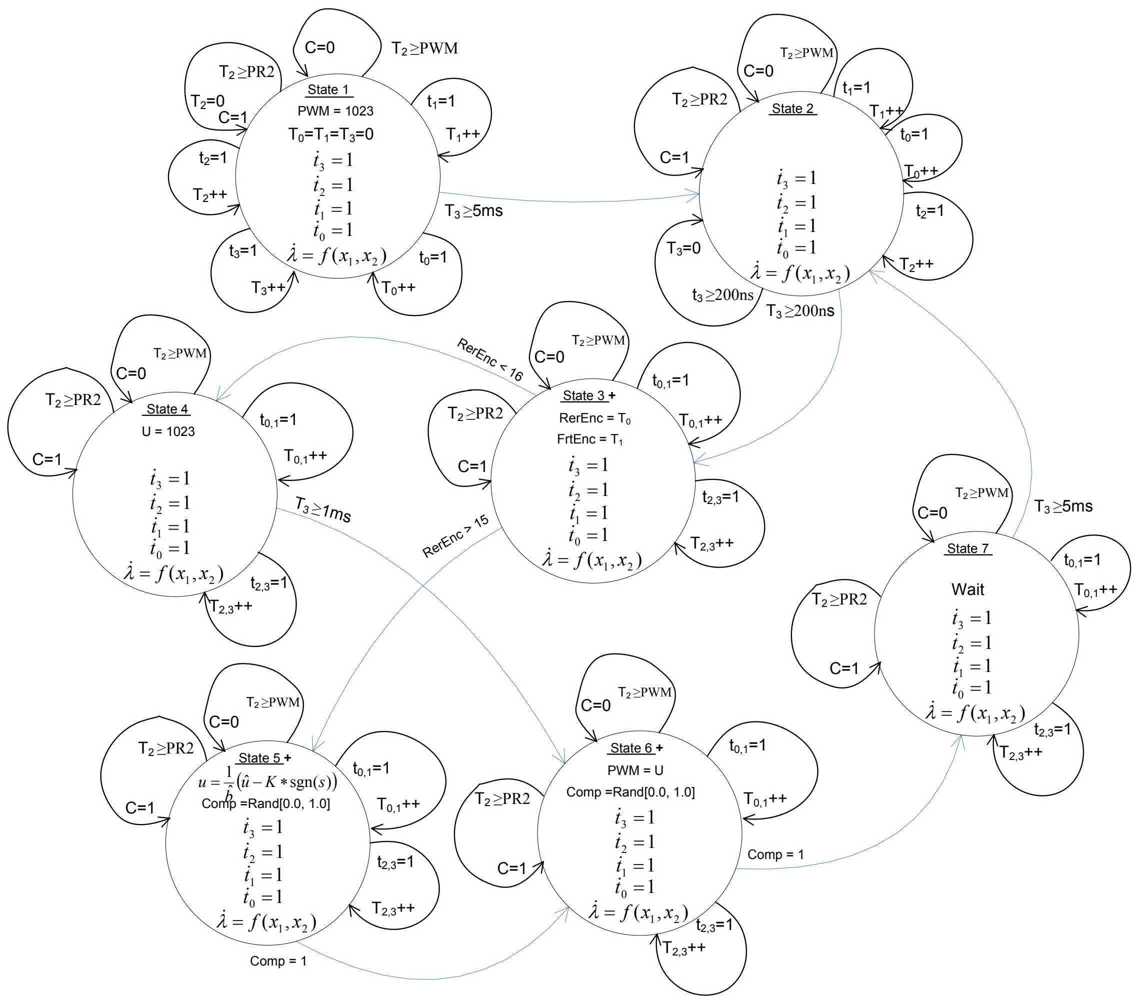


Figure 7.3: Hybrid representation of the robot car

The function $\dot{\lambda} = f(x_1, x_2)$, in the model shown in Figure 7.3 represents the continual dynamics of the robot car. Just as a State Machine representation of a computer program, each state here would normally represent a line of code, however, to keep the system manageable for discussion purposes, states with a "+" have been condensed to the most important line(s) of code. Another dynamic of this system not yet discussed is that states 5 and 6 are stochastic. This is because the compilation of the control algorithm varies in computation time depending on the input values. As modeling hybrid systems is still in its infancy [15] strict modeling techniques have not yet been developed. Therefore, in this thesis the Matlab random generator code is used to symbolize the stochastic property of these states.

State 1 represents the initialization of the car's code, and will be used to explain the parameters that all the states share. The continuous dynamics are represented by t_0, t_1, t_2, t_3, and $\dot{\lambda} = f(x_1, x_2)$; where T_0 thru T_3 represent the microprocessor's timers. The three loops off of State 1 at about 9 to 12 o'clock represent the PWM signal dynamics internal to the microprocessor. These dynamics generate the PWM duty cycle. The other two loops deal with the front and rear wheel encoder counter inputs. The loops that act like counters are combined on other states in order to conserve space and reduce confusion. State 2 represents the delay to update/reset the interrupt timer 3. State 3+ illustrates the reading of the front and rear wheel encoder counter values. Since the control algorithm used must have an angular velocity on the rear wheel a test was necessary to ensure the control algorithm was never implemented if the rear wheels were not moving. For this reason, if the rear wheel's angular velocity was below a certain point, full power would be applied to the motor. Initially a scaled exponential function was used, however because the electrical system's time response was significantly faster than that of the mechanical system the function was determined to be unnecessary. It may also be noticed that State 4 is not stochastic; this is because no computation is done here. State 5+ represents the computation part of the control algorithm and State 6+ represents the time taken to update the PWM signal. The microprocessor only updates the PWM signal at the end of every period (at most about 1.02e-4 seconds). The remaining of the 5ms is spent in a loop (as represented by State 7) until the interrupt from timer 3 starts the process over again.

Chapter 8

Conclusions and Future Work

The idea of modeling with computer and physical systems has been around for decades, and the models presented here may also be used as valuable tools to make improvements on the automated highway concept.

8.1 Conclusions

In conclusion the results indicate a useful system; however, from the results if the same tire setup is to be used, further analysis must be done to better characterize the $\mu(\lambda)$ curve. Otherwise, tires that better represent the dynamics of tires on full size vehicles will need to be used. This is most likely one of the larger sources of error, which in turn gave rise to discrepancies in the simulated annealing process. One significant source of error that was compensated for was the slight difference in front and rear tire sizes which could be a serious concern if this exact setup were to be implemented on full size vehicles.

8.2 Future Work

The numerous dynamics that even this simplified system represents provides for countless possibilities for future work. Therefore, it is the intent to just highlight some of the more important and fascinating possibilities for future work. To start with, modeling the $\mu(\lambda)$ curve for tires wrapped in Duct Tape is at the top of the list, followed by attempting to make more exact measurements on other system parameters. The next recommendation would be to add an accelerometer to eliminate the dependency on tire size, which would also be a viable solution on full size vehicles. The move to an accelerometer would also pave the way for future work on maximum acceleration and maximum deceleration. Future work on the control algorithm can also be done by applying a boundary layer and adding some integral control. Other additions that may be made would be to add developments leading to testing outside of a fixed test platform or adding other dynamics such as a suspension system. Further analysis could also be done on using the 3-D plot to generate a set of look-up tables to speed up processing time, or develop it into a generic solution for any type vehicle given certain parameters as inputs.

Bibliography

[1] P. Kachroo, "Nonlinear Control Strategy and Vehicle Traction Control." Berkeley, CA: UC at Berkeley, 1990.

[2] V. N. Müller, "Development of a robust driver model with parameter adaptation," in *Institut für Industrielle Informationstechnik (IIIT)*. Karlsruhe, Germany: Universität Karlsruhe (TH), 1996.

[3] J. Y. Wong, *Theory of Ground Vehicles*, Third ed. New York: John Wiley & Sons, Inc., 2001.

[4] C. Ünsal and P. Kachroo, "Sliding mode measurement feedback control for antilock breaking system," *IEEE*, vol. 7, pp. 271-281, 1999.

[5] J.-J. Slotine and W. Li, *Applied Nonlinear Control*. NJ: Prentice Hall, 1991.

[6] H. K. Khalil, *Nonlinear Systems*, 3rd ed. NJ: Prentice Hall, 2002.

[7] NE, "Novak Electronic Inc., Rooster/Super Rooster," www.teamnovak.com, 2003.

[8] E. Blanchard, "H-bridge using P and N channel FETs," http://www.geocities.com/fet_h_bridge/intro.html, 2001.

[9] P. Mellodge, "Control for a Path Following Robotic Car." Blacksburg: Virginia Tech, 2002.

[10] "Interface Security," www.workingtex.com, 2002.

[11] J. C. Spall, *Introduction to Stochastic Optimization: Estimation, Simulation and Control*. NY: John Wiley and Sons, 2003.

[12] R. Jang, "Software Computing Toolbox for Neuro-Fuzzy and Soft Computing," http://neural.cs.nthu.edu.tw/jang/book/soft/, 2004.

[13] S. A. Luis, "Mechanical Engineering Department, Texas A&M University (TAMU), MEEN 617 Notes: Handout 1, Modeling of Mechanical (Lumped Parameter) Elements," 2000.

[14] J. P. Hespanha, "Hybrid Control and Switched Systems." Santa Barbara, 2002, pp. Lecture Notes.

[15] A. v. d. Schaft, *An Introduction to Hybrid Dynamical Systems*. London: Springer, 2000.

Appendix A

Variables and Units

R_W = Radius of wheel, [m]

μ = Adhesion coefficient

λ = Wheel slip

N_V = Normal tire force [Newtons]

F_t = Tractive force (Average force of driving wheels)

J_W = Moment of inertia of wheel (MOI), [$Kg \cdot m^2$]

F_W = Wheel/Viscous friction, [$N \cdot \frac{S}{M}$ or $\frac{lb-\sec}{ft}$]

n_W = number of driving wheels

m_V = Mass of vehicle, [Kg]

F_V = Wind drag force [Newton]

T_e = Torque of engine [Newton]

T_b = Torque of brakes (not used)

V_V = Linear velocity of vehicle [meters/second]

ω_v = Angular velocity of front wheel [radians/second]

ω_W = Angular velocity of rear wheel [radians/second]

N_V = Normal force to ground, [N]

c = Wind drag coefficient

R_a = Motor armature resistance [Ohms]

K_e = Motor electrical constant [$\frac{V}{rad \cdot s}$]

Appendix B

Robot Car Follower Code

```
#include <pic18.h>
#include <math.h>

unsigned int RerEnc;
unsigned int FrtEnc;


void interrupt isr(void)
{
   if ( T0IF)    /* TMR0 overflow interrupt */
   {

/***********************************************/
                        if (RerEnc < FrtEnc)
                        {
                                    if (CCPR2L+1 > 255)
                                                CCPR2L = 0xFF;
                                    else
                                                CCPR2L = CCPR2L + 1;
                        }

                        if (RerEnc > FrtEnc)
                        {
                                    if (CCPR2L-1 < 0)
                                                CCPR2L = 0x00;
                                    else
                                                CCPR2L = CCPR2L - 1;
                        }
                        FrtEnc = 0;
                        RerEnc = 0;


               TMR0 = 0;   // For reseting Timmer 0 value
               T0IF = 0;  /* reset Timmer 0 flag  INTCON<2>*/
            }
            if (INT2IF)                          // RB0/INT ***front wheel count***
            {
                        FrtEnc++;
                        INT2IF = 0;  /* reset RB0/INT flag */
            }
            if (INT1IF)  //if (TMR1IF)              // (TMR1) Timer1 interupt/RC2 ****rear wheel count****
            {
```

```
                        RerEnc++;
                        INT1IF = 0; /* reset Timmer 1 capture flag */
            }
}

void main(void)
{

//          Direction of motor
            TRISE = 0x00;    // put port D in standard I/O
            PORTD = 0xFF;  // set port D as High
            TRISD = 0x00;    // set port D as output

            //  PWM for TMR2
            PR2     = 0xFF;                 // PWM period
            CCPR2L  = 0x00;                 // Upper 8 bits of 10 bit PWM duty cycle

            TRISC   = 0x00;                 // Make CCP2/RC1 pin an output for the PWM2 signal

            T2CON   = 0x05;                 // Set the TMR2 prescale value & enable Timer2 (2.44kHz)
            CCP2CON = 0xFF;                 // Lower 2 bits of 10 bit PWM duty cycle & Put in PWM Mode


// Setup RB1/Int to count "UP" pulses of REAR wheel
// Setup RB2/Int to count "UP" pulses of FRONT wheel
            INTCON  = 0xE0;  // Enable interupt
            INTCON2 = 0x74;  // Set Interupts to rising edge & Priority for TMR0 High
            INTCON3 = 0x18;                 // Set RB1 & RB2 to Low

// Timmer 0 Interupt
            T0CON = 0xC5;
            TMR0 = 0;  /* 256 @ 64 = 1.64ms/2 delay */
            INTCON  = 0xF0;  // Enable interupt

            FrtEnc = 0;
            RerEnc = 0;

            PIE1 = 0x00;
            PIE2 = 0x00;

   while (1)
            {
                        /* infinite loop */
                        NOP();
            }
}
```

Appendix C

Parts List

Part Label	Part Description	Part Number	MFG
R1, R2	Resister	5K Ohm	Generic part
R3, R4, R6, R8	Resister	10K Ohm	Generic part
Q3, Q5	P Channel Mosfet	IRF4905	Generic part
Q4, Q6	N Channel Mosfet	IRL3803	Generic part
D1, D2, D3, D4	Diode	1N4001	Generic part
Q1, Q2	NPN Transistor	2N3904	Generic part
Z1	Zenor Diode	15V, 1W	Generic part
C1	Capacitor	0.1 μF	Generic part
C2	Capacitor	470 μF	Generic part

Table C.1: H-Bridge Parts Lists

Part Label	Part Description	Part Number	MFG
	RISC processor	18F452	Microchip
	RS-232 Driver/Receiver	MAX232A	MAXIM
	Optical Encoder	HEDS-9140-I00	Agilent
	1" Disk 512 pulses per rev.		Agilent
	Crystal	ECS-2299A-400	ECS, Inc. International
	D-Flip Flop	DM74LS74N	FAIRCHILD

Table C.2: Data Collector's Parts Lists

Part Label	Part Description	Part Number	MFG
	RISC processor	18F452	Microchip
	Optical Encoder	HEDS-9140-I00	Agilent
	1” Disk 512 pulses per rev.		Agilent
	Crystal	ECS-2299A-400	ECS, Inc. International
	D-Flip Flop	DM74LS74N	FAIRCHILD
	Multiplexer	74S153	Teas Instruments
	5 volt regulator	7805	Generic part
			Generic part
			Generic part

Table C.3: Robot Car’s Control Circuit Parts List

Appendix D

Matlab Data Collection Code

```
% Test Data reader and Ploter %
% Input = COM1 @ 115200 bps
clear all;
close all;
clc;

Size = 8008;  % = 10sec of data w/4 inputs at 5ms sampling

s = serial('COM1', 'BaudRate', 115200, 'InputBufferSize', 1600000, 'Timeout', 25);
s.ReadAsyncMode = 'continuous';
fopen(s);
[A, count] = fread(s, Size, 'uchar');

fclose(s)

cc = 1;
for ii = 1: 4: count,

   RerEnc(cc) = A(ii);
   FrtEnc(cc) = A(ii+1);
%%%%%%%%%% Only used when not manipulating Upr and Lwr data bits
%     UprBit(cc) = A(ii+2);
%     LwrBit(cc) = A(ii+3);
%%%%%%%%%%%%
%%  for loading U  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   UprBit(cc) = A(ii+2);
   U(cc) = bitshift(UprBit(cc), 2);  % Load Upper 8 bits of 10 bits plus sign(+/-) into variable U

   LwrBit(cc) = A(ii+3);
   %%%% Determine and Set/Clear Lower 2 bits of 10 bits of U
   if (bitget(LwrBit(cc), 5) == 0)
      U(cc) = bitset(U(cc), 1, 0);
   else
      U(cc) = bitset(U(cc), 1);
   end

   if (bitget(LwrBit(cc), 6) == 0)
      U(cc) = bitset(U(cc), 2, 0);
   else
      U(cc) = bitset(U(cc), 2);
   end
```

```
    % Determine if U is possitive or negative

    if (bitget(LwrBit(cc),7) == 1)
        U(cc) = -U(cc);
    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    cc = cc + 1;
endt = [0:0.005:10.005];  %10s 4inputs
figure
plot(t,RerEnc,'.')
figure
plot(t,FrtEnc,'.')

% save DataFile FrtEnc RerEnc TrkEnc t;
count

%%%%%%% Other way to set/clear lower 2 bits of U
%     if (bitget(LwrBit(cc), 6)==1 && bitget(LwrBit(cc), 5)==1)
%         U(cc) = U(cc) + 3;
%     elseif (bitget(LwrBit(cc), 6)==1 && bitget(LwrBit(cc), 5)==0)
%         U(cc) = U(cc) + 2;
%     elseif (bitget(LwrBit(cc), 6)==0 && bitget(LwrBit(cc), 5)==1)
%         U(cc) = U(cc) + 1;
%     end
```

Appendix E

Data Collection Circuit (DCC) Code

```
#include <pic18.h>
#include <stdio.h>


#define USART_TX (RD3)  // Turns Output ON and OFF
#define BAUD 115200
#define FOSC 40000000L
#define NINE_BITS 0                    // 8-bit communication
#define OUTPUT 0
#define INPUT 1

#define DIVIDER ((int)(FOSC/(16UL * BAUD) -1))
#define SPEED 0x4

unsigned char TrkEnc;
unsigned char FrtEnc;
unsigned char RerEnc;

unsigned int LstTrk;
unsigned int LstRer;

void main(void)
{
/* Serial initialization */
            SPBRG = DIVIDER;
            TXSTA = (SPEED|NINE_BITS|0x20);
            RCSTA = (NINE_BITS|0x90);
            TRISC6=OUTPUT;
            TRISC7=INPUT;           /*initialize usart in serial.c*/
///////////////////////////

// Setup RB0/Int to count "UP" pulses of TestPlatform
            T2CON   = 0x00;  // Turn OFF TMR2

//          Direction of motor & to turn on data collection
            TRISE = 0x00;  // put port D in standard I/O
            PORTD = 0x00;  //
            TRISD = 0xFF;  // set port D as Input

//  10 bit PWM value output  (8 upperbits on PORTB and lower 2 bits on PORTD<7,6>
            PORTB = 0x00;  // set port RB as 0's
            TRISB = 0xFF;  // set port B as Input
```

```
//  Timmer 0 Counter  (Rear Wheel)
            T0CON = 0xA8;  ///8= no prescaler  ;  0=*2 prescaler  ////0xAF

//          Timmer 1 Counter  (Front Wheel/ TestPlatform
            T1CON = 0x87;

            T3CON = 0x85;

            TMR0H = 0;
            TMR0  = 0;
            TMR1H = 0;
            TMR1  = 0;
            TMR3H = 0xFB; // 15ms = B6C1; 52.4ms = 0000; 1ms = FB1D/FB27
            TMR3  = 0x27;   // For reseting Timmer 3 value

            TMR3H = 0xD9;             // 1ms = D93B w/o prescaler
            TMR3    = 0x3B;
//          TMR3H = 0x3C;             // 5ms = 3CF3 w/o prescaler
//          TMR3    = 0xFB;           // 5ms = 3CFB w/o prescaler

            TrkEnc = 0;
            FrtEnc = 0;
            RerEnc = 0;
            LstTrk = 0;
            LstRer = 0;

            PIE1 = 0x00;
            PIE2 = 0x02;  // Enables TMR3 Overflow Interrupt Enable bit

            INTCON  = 0xD0;  // Enable interupt

   while (1)
            {
                        NOP();

            }

}


void interrupt isr(void)
{


   if (TMR3IF)    /* TMR3 overflow interrupt */
   {
            TMR3H = 0xD9; // 0xD9
            TMR3  = 0x3B;   // 0x3B  //For reseting Timmer 3 value to 1ms

            if (LstRer > TMR0)
                        RerEnc = 65535-LstRer + TMR0;  //remember to start counting from '0'
            else
            RerEnc = TMR0 - LstRer;
            LstRer = TMR0;
```

```
		if (LstTrk > TMR1)
			TrkEnc = 65535-LstTrk + TMR1;
		else
		TrkEnc = TMR1 - LstTrk;
		LstTrk = TMR1;

		if (USART_TX)
		{
/*			printf("%c",FrtEnc);			*/
			printf("%c",RerEnc);
			printf("%c",TrkEnc);
			printf("%c",PORTB);  // UprPwr
			printf("%c",PORTD); // LwrPwr
		}
		TMR3IF = 0;  /* reset Timmer 3 flag  PIR2<1>*/
  }

}

void putch(unsigned char byte)
{
		/* output one byte */
		while(!TXIF)	/* set when register is empty */ // PIR1<4>
			continue;
		TXREG = byte;
}
```

Appendix F

Matlab Code For 3-D Plots of Control Outputs

```
% By:  Mark Morton
% Sliding Mode Control 3-D Plot for Tracktion Control

clc;
clear all;
close all;

MueP    = 0.0745;
Lamda_P = 0.1551;
Lamda_d = 0.01;  % desired Lamda from simulated annealing      % desired path
c       = 0.005;        % wind drag force coeficient
mv  = 1.6516;        % mass of car (Kg)
nw  = 2;          % number of driving wheels
Nv  = 1.3009;  % w/o batt 0.9328;  %w/ batt= 1.3009;        % normal force to ground (N)
Rw  = 0.0320548;  % radius of wheel (Meters)
Fw  = 0.4507;       % wheel viscous friction (N*Sec/Meter)
Jw  = 0.1088;      %0.5*m*Rw^2; % inertia (Kg*m^2)

Ke = 0.0159;  %6.7831e-3;      %
Ra = 1.506;  %5; %1.506;           % Armeture resistance
Volts = 12.0;       % Maximum voltage supplied to system (aka: battery)

N    = 0.2;
minn = 0.9;
maxx = 1.1;
Beta = 1.4;

z = 1;

Rfw = 2.4035;
Rrw = 2.524;  %2.54;  %2.524;
Rtw = 1.8085;
Ratio = 2/3; % for Track/Rw
Time        = 0.005*z;
% Time        = 0.00656;
% Time        = 0.015;

for FrtEnc = 1:1:64*z, %1:64, %1:96*z, %90, %270,  %900
    for RerEnc = 1:64*z, %90, %270,   %900
```

```
        x(1) = FrtEnc*6.283185/256.0/Time;  %*2.524/2.2035;   Compensation ratio
        x(2) = RerEnc*6.283185/256.0/Time;         Lamda   = 1-x(1)/x(2);

       Vv  = x(1)*Rw;     %initial velocity
       Fv  = c*Vv^2;       % wind drag force

      bmin    = minn*(1-Lamda)*Ke/(Ra*Jw*x(2));
      bmax    = maxx*(1-Lamda)*Ke/(Ra*Jw*x(2));
      b_hat   = sqrt(bmin*bmax);

     Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction coef.
     Mue_hat = ((2*0.94*0.3*Lamda)/(0.3^2+Lamda^2) + (2*0.008*0.008*Lamda)/(0.008^2+ …
            Lamda^2))/2 + 0.001;

     Muedif = Mue-Mue_hat;

% F   = abs(f_hat - f);
      F   = abs([(Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))] * (Mue-Mue_hat));

      s=Lamda-Lamda_d;

      Lamda)*Rw*Nv*Mue)/(Jw*x(2))] + ((1-Lamda)*u)/(Jw*x(2));
      u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='

%K   = F + N;             % Gain
     K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);
%
     u = 1/b_hat*(u_hat - K*sign(s))

   PerVolt = u/1023;                  % Percent Voltage to motor
   V = PerVolt * Volts;                % actual voltage supplied to motor

   if (V > 12)
      V = 12;
   end
   if (V < -12)
      V = -12;
   end

   if (u > 1023)
      u = 1023;
   end
   if (u < -1023)
      u = -1023;
   end

   U_out(FrtEnc, RerEnc) = u;

   V_out(FrtEnc, RerEnc) = V;

%          if (mod(RerEnc,100) == 1)
%             u
%          end
   end
end
figure
mesh(V_out)
```

Appendix G

Leader Code

```
#include <pic18.h>
#include <stdlib.h>
#include <math.h>

unsigned int RerEnc;
unsigned int FrtEnc;
unsigned int LstFrt;
unsigned int LstRer;
int Pwr;

/* service routine for timer 3 interrupt */

void interrupt isr(void)
{
//          unsigned int tics = TMR0;

            TMR3H = 0x3C; // 0xFB   ~1ms = FB1D w/prescaler
            TMR3  = 0xC2;   // 0x1D  //For reseting Timmer 3 value
                        if (LstRer > TMR0)
                                    RerEnc = 65535-LstRer + TMR0;  //remember to start counting from '0'
                        else
                                    RerEnc = TMR0 - LstRer;
                        LstRer = TMR0;

                        if (LstFrt > TMR1)
                                    FrtEnc = 65535-LstFrt + TMR1;
                        else
                                    FrtEnc = TMR1 - LstFrt;
                        LstFrt = TMR1;
/*********************************************/

            if (RerEnc > FrtEnc+5)  //1.0501352197*FrtEnc+5)
            {
                        Pwr = Pwr - 1;
            }
            else
                        Pwr = Pwr + 1;

            if (Pwr>=0)
            {           //**********check value of u ?????*************//
                        PORTD = 0x08;               // Bit RD2 = 0 for forward direction when Y1 on
                                                    //MUXconnected to IN1 on H-bridge
```

```
                if (Pwr > 1023)
                        Pwr = 1023;
                CCPR2L  = Pwr>>2; // & 1020;
                CCP2CON = 0xf ^ (Pwr<<4);
                PORTB   = CCPR2L;
                PORTD   = PORTD + (Pwr<<4);
        }
        else
        {       //**********check value of u ?????************//
                PORTD = 0x0C;  // Bit RD2 = 1 for reverse direction
                if (Pwr < -1023)
                        Pwr = 1023;
                else
                        Pwr = -Pwr;
                CCPR2L  = Pwr>>2; // & 1020;
                CCP2CON = 0xf ^ (Pwr<<4);
                PORTB   = CCPR2L;
                PORTD   = PORTD + (Pwr<<4);
                Pwr = -Pwr;
        }
        TMR3IF = 0;  /* reset Timmer 3 flag  PIR2<1>*/
}

void main(void)
{

//      Direction of motor
        TRISE = 0x00;  // put port D in standard I/O
        PORTD = 0x08;  // set port RD2 as LOW for forward direction
                                // and port RD3 HIGH to turn test platform output ON
        TRISD = 0x00;  // set port D as output

//  10 bit PWM value output  (8 upperbits on PORTB and lower 2 bits on PORTD<7,6>
        PORTB = 0x00;  // set port RB as 0's
        TRISB = 0x00;  // set port B as output

        //  PWM for TMR2
        PR2     = 0xFF;     // PWM period
        CCPR2L  = 0x55;  // Upper 8 bits of 10 bit PWM duty cycle



        T2CON   = 0x05;  // Set the TMR2 prescale value & enable Timer2 0x05 = (9.77kHz)
        CCP2CON = 0xFF;          // Lower 2 bits of 10 bit PWM duty cycle & Put in PWM Mode
        TRISC   = 0x01;  // Make CCP2/RC1 pin an output for the PWM2 signal

//  Timmer 0 Counter
        T0CON = 0xA8;  ///8= no prescaler  ;  0=*2 prescaler

//      Timmer 1 Counter
        T1CON = 0x87;

//      Timmer 3 Interupt on Over Flow
//      T3CON = 0b10110101
        T3CON = 0x85;   // B5=8*prescaler
        TMR0H = 0;
        TMR0  = 0;

        TMR1H = 0;
        TMR1  = 0;

        TMR3H = 0x3C; // 15ms = B6C1; 52.4ms = 0000; 5ms = E795; 1ms = FB1D;
        TMR3  = 0xC2;   // For reseting Timmer 3 value

        INTCON  = 0xC0;  // Enable interupt

        FrtEnc = 0;
        RerEnc = 0;
        LstFrt = 0;
        LstRer = 0;
        Pwr    = 0;

        PIE1 = 0x00;
        PIE2 = 0x02;  // Enables TMR3 Overflow Interrupt Enable bit

        CCPR2L  = 0xFF;  // Upper 8 bits of 10 bit PWM duty cycle

  while (1)
        {
                /* infinite loop */
                NOP();
        }
}
```

Appendix H

Matlab Mean Code

```
clear all;
clc;

% load F:\'serial test data'\'Annealing Prog'\Data7aB10s

Dfw = 2.4035;
Drw = 2.524;
Dtw = 1.8085;

x = FrtEnc*Dfw/Drw; %UprBit;  %FrtEnc;\
y = RerEnc;  %TrkEnc*Dtw/Drw;  %RerEnc;
x2 = x;
y2 = y;

count = length(t);
cc = 0;


neg = 0;
pos = 0;

% First and Last Data Points are NOT Averaged

m = 51  % must be an ODD number

for ii = 2: 1: (m-1)/2,
   for jj = ii-1: -1: 0,
      neg = neg + x(ii - jj);
      pos = pos + x(ii + jj);
   end
   x(ii) = (neg + pos)/(ii * 2);
   neg = 0;
   pos = 0;
end

for ii = ((m+1)/2): 1: length(t)-(m+1)/2,
   for jj = m: -2: 1,
      neg = neg + x(ii - (jj-1)/2);
      pos = pos + x(ii + (jj-1)/2);
   end
   x(ii) = (neg + pos)/(m + 1);
   neg = 0;
```

```
    pos = 0;
end

for ii = length(t)-(m-1)/2: 1: length(t)-1,
    for jj = length(t)-ii: -1: 0,
        neg = neg + x(ii - jj);
        pos = pos + x(ii + jj);
    end
    x(ii) = (neg + pos)/((length(t)-ii+1) * 2);
    neg = 0;
    pos = 0;
end

for ii = 2: 1: (m-1)/2,
    for jj = ii-1: -1: 0,
        neg = neg + y(ii - jj);
        pos = pos + y(ii + jj);
    end
    y(ii) = (neg + pos)/(ii * 2);
    neg = 0;
    pos = 0;
end

for ii = ((m+1)/2): 1: length(t)-(m+1)/2,
    for jj = m: -2: 1,
        neg = neg + y(ii - (jj-1)/2);
        pos = pos + y(ii + (jj-1)/2);
    end
    y(ii) = (neg + pos)/(m + 1);
    neg = 0;
    pos = 0;
end

for ii = length(t)-(m-1)/2: 1: length(t)-1,
    for jj = length(t)-ii: -1: 0,
        neg = neg + y(ii - jj);
        pos = pos + y(ii + jj);
    end
    y(ii) = (neg + pos)/((length(t)-ii+1) * 2);
    neg = 0;
    pos = 0;
end

FrtEnc = x;
RerEnc = y;
Diff = x-y;
clear x y cc ii jj neg pos;


figure
subplot(4,1,1)
plot(t, FrtEnc)
subplot(4,1,2)
plot(t,RerEnc)
subplot(4,1,3)
plot(t,1-FrtEnc./RerEnc)
```

```
subplot(4,1,4)
plot(t,1-x2./y2)


% save F:\'serial test data'\'Annealing Prog'\Data7bB10s
```

Appendix I

Matlab Simulated Annealing Code

I.1 Final Simulated Annealing Code

```
clear all;
close all;
clc;

tic;
%%%%%%%%%%%%  Sliding Mode Setup  %%%%%%%%%%%%%%%
global Mue Fv mv nw Nv Rw Fw Jw Ke V Ra x2
load Data7aB10s

figure
subplot(3,1,1), plot(t, RerEnc)
subplot(3,1,2), plot(t, FrtEnc)
subplot(3,1,3), plot(t, U)


Ke_Range = [2.7e-3: 1.0e-4: 24.74e-3];      %16.74e-3;  %6.7831e-3;      %
Ra_Range = [1.2: 0.1: 1.9];      %1.2;           % Armeture resistance
Lamda_d_Range = [0.001: 0.001: 0.3];      % desired path
MueP_Range = [0.01: 0.01: 0.3];
Lamda_P_Range = [0.01: 0.01: 0.3];
c_Range = [0.001: 0.001: 1.0];          % wind drag force coeficient
Nv_Range = [1.2: 0.01: 1.6516];            % normal force to ground (N)
Fw_Range = [0.01: 0.1: 7];          % wheel viscous friction (N*Sec/Meter)
Jw_Range = [1.116425e-6: 1.0e-4: 0.022]; %0.5*m*Rw^2; % inertia (Kg*m^2)

mv  = 1.6516;        % mass of car (Kg)
nw  = 2;          % number of driving wheels
Rw  = 0.0320548;    % radius of wheel (Meters)
Volts = 12.0;       % Maximum voltage supplied to system (aka: battery)

Error_Over = 0;
tspan = 0.0049999;
Time        = 0.0049999;
          Rfw = 1.20175;
             Rrw = 1.262;
% digits(7);
tic;
```

```
%%%%%%%%%%%%%%%  Setup for Annealing  %%%%%%%%%%%%%%%%%%%
Lowest_Error = 100000;
Highest_Error = 0.5;
Curr_Error = inf;
temp = 10e8;       % Choose the temperature to be large enough
fprintf('Initial Curr_Error = %f\n\n',Curr_Error);

NumCombo = 100;
rand('state',sum(100*clock))  % Resets it to a different state each time.

MaxTempItr = NumCombo*10;              % Max. # of trials at a temperature
MaxLcaLitr = NumCombo*1;               % Max. # of acceptances at a temperature
StopTolerance = 0.0001;          % Stopping tolerance
TempRatio = 0.98;                % Temperature decrease ratio
TempItr = 0;              % Number of trial moves

minE = inf;                      % Initial value for min. Curr_Error
maxE = -1;                       % Initial value for max. Curr_Error

%%%%%%%%%%%%%%%%%%%%%  Annealing loop  %%%%%%%%%%%%%%%%%%%%%%%
while (TempItr < MaxTempItr),  %(((maxE-minE)/maxE > StopTolerance) & (TempItr < MaxTempItr)),

   minE = inf;                   % Initial value for min. Curr_Error
   maxE = -1;                    % Initial value for max. Curr_Error
        LcaLitr = 0;             % Number of local iterations at a certain temp.

        while ((Curr_Error > 0.005) & (LcaLitr < MaxLcaLitr)),

    Ke      = rand * (max(Ke_Range)-min(Ke_Range)) + min(Ke_Range);
    Ra      = rand * (max(Ra_Range)-min(Ra_Range)) + min(Ra_Range);
    Lamda_d = rand * (max(Lamda_d_Range)-min(Lamda_d_Range)) + min(Lamda_d_Range);
    MueP    = rand * (max(MueP_Range)-min(MueP_Range)) + min(MueP_Range);
    Lamda_P = rand * (max(Lamda_P_Range)-min(Lamda_P_Range)) + min(Lamda_P_Range);
    c       = rand * (max(c_Range)-min(c_Range)) + min(c_Range);
    Nv      = rand * (max(Nv_Range)-min(Nv_Range)) + min(Nv_Range);
    Fw      = rand * (max(Fw_Range)-min(Fw_Range)) + min(Fw_Range);
    Jw      = rand * (max(Jw_Range)-min(Jw_Range)) + min(Jw_Range);

    Error_Sum = 0;
    rand('seed', 31415927)

%       for ii = 750:1200,
%       for ii = 2:length(t),
      for ii = 2:1200,

       x(1) = FrtEnc(ii)*6.283185/256.0/Time*Rfw/Rrw;
       x(2) = RerEnc(ii)*6.283185/256.0/Time;
       x2 = x(2);
       x1 = x(1);
       if (x(2) ~= 0),
          LamdaC   = 1-x(1)/x(2);

          Vv  = x(1)*Rw;    %initial velocity
          Fv  = c*Vv^2;     % wind drag force

          Mue = (2*MueP*Lamda_P*LamdaC) / (Lamda_P^2+LamdaC^2);  % adhesion/friction coef.
                                                  %(THE INPUT for the smallest curve)
      %%%%%%%%%%%%%%%%%% ODE Solver %%%%%%%%%%%%%%%%%%%%%%
        u = U(ii);
        PerVolt = u/1023;                % Percent Voltage to motor
        V = PerVolt * Volts;              % actual voltage supplied to motor

        [T, X] = ode45('states2',[0,tspan],x);
        x = X(length(X), [1,2]);

        LamdaM   = 1-x(1)/x(2);

       LC(ii) = LamdaC;
       LM(ii) = LamdaM;
       FW(ii) = x(1);
       RW(ii) = x(2);
       v(ii)  = V;

        Error1 = (x(2)-x2)^2;
        Error2 = (x(1)-x1)^2;
        Error3 = (LamdaM - LamdaC)^2;
        Error = Error1 + Error2 + Error3;
        Error_Sum = Error_Sum + Error;
        New_Error = Error_Sum;
      else
        Error_Sum = Error_Sum + 0;
      end

      %%%%%  If erro already over 3 no need to continue calculating error %%%%
      if (New_Error > Lowest_Error)
        Error_Over = 1
        break;
      end

     end


if (((rand < exp((Curr_Error-New_Error)/temp)) | ((New_Error-Curr_Error) < 0)) & Error_Over == 0),
        % accept it!

      Curr_Error = New_Error
                      minE = min(minE, Curr_Error);
                      maxE = max(maxE, Curr_Error);
      RWC = RW;
      if Curr_Error < Lowest_Error,
        Lowest_Error = Curr_Error
        LCL = LC;
        LML = LM;
        FWL = FW;
        RWL = RW;
        save LowError16
      end
      if Curr_Error > Highest_Error,
        Highest_Error = Curr_Error
        LCH = LC;
        LMH = LM;
        FWH = FW;
        RWH = RW;
        save HighError16
        end
  end
      rand('state',sum(100*clock))  % Resets it to a different state each time.
      Error_Over = 0;     % resets too high error value indicator

      LcaLitr = LcaLitr + 1;
end

            % Print information in command window
            fprintf('temp. = %f\n', temp);
            fprintf('Curr_Error = %f\n', Curr_Error);
            fprintf('[minE maxE] = [%f %f]\n', minE, maxE);
            fprintf('[LcaLitr TempItr] = [%d %d]\n\n', LcaLitr, TempItr);
            % Lower the temperature
            temp = temp*TempRatio;
  % Increment trial count
  TempItr = TempItr + 1;
end

fprintf('Compute Time = %f\n', toc);

      Ra
      Ke
      Lamda_d
      MueP
      Lamda_P
      c
      Nv
      Fw
      Jw

save AnnealResult16

toc;
```

I.2 Earliest Simulated Annealing Code

```
clear all;
close all;
clc;

tic;
%%%%%%%%%%%%%  Sliding Mode Setup  %%%%%%%%%%%%%%%%
global Mue Fv mv nw Nv Rw Fw Jw u
load data1aB5s
% load Data7aB10s
% load Data7bB10s
% load Data5aB10s

figure
```

```
subplot(3,1,1), plot(t, RerEnc)
subplot(3,1,2), plot(t,FrtEnc)
subplot(3,1,3), plot(t,U)

MueP_Range = [0.11: 0.01: 0.2];
Lamda_P_Range = [0.11: 0.01: 0.3];
c_Range = [0.01: 0.001: 0.05];        % wind drag force coeficient
Nv_Range = [1.0: 0.01: 1.6516];           % normal force to ground (Kg)
Fw_Range = [0.01: 0.001: 0.3];          % wheel viscous friction (N*Sec/Meter)
Jw_Range = [8.416425e-5: 1.0e-6: 0.011]; %0.5*m*Rw^2; % inertia (Kg*m^2)

Lamda_d = 0.25;     % desired path
%    c      = [0.001:0.001:0.05]; %%0.25;  % wind drag force coeficient
% %    m      = 0.016; % mass of wheel (Kg)
mv  = 1.6516;       % mass of car (Kg)
nw  = 2;          % number of driving wheels
Rw  = 0.0320548;    % radius of wheel (Meters)
tspan = 0.0049999;
Time       = 0.0049999;

digits(7);
tic;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Setup for Annealing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Lowest_Error = 1000;
Highest_Error = 10;
Curr_Error = inf;
temp = 10e6;   % Choose the temperature to be large enough
fprintf('Initial Curr_Error = %f\n\n',Curr_Error);

NumCombo = 1;
rand('seed', 31415927)
% rand('state',sum(100*clock))  % Resets it to a different state each time.

MaxTempItr = NumCombo*10;                 % Max. # of trials at a temperature
MaxLcaLitr = NumCombo*5;                  % Max. # of acceptances at a temperature
StopTolerance = 0.005;                 % Stopping tolerance
TempRatio = 0.25;               % Temperature decrease ratio
TempItr = 0;              % Number of trial moves

minE = inf;                          % Initial value for min. Curr_Error
maxE = -1;                           % Initial value for max. Curr_Error

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Annealing loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
while (((maxE-minE)/maxE > StopTolerance) & (TempItr < MaxTempItr)),
% while (TempItr < MaxTempItr),
   minE = inf;                  % Initial value for min. Curr_Error
   maxE = -1;                   % Initial value for max. Curr_Error
         LcaLitr = 0;           % Number of local iterations at a certain temp.

         while ((Curr_Error > 0.5) & (LcaLitr < MaxLcaLitr)),

      MueP    = rand * (max(MueP_Range)-min(MueP_Range)) + min(MueP_Range);
      Lamda_P = rand * (max(Lamda_P_Range)-min(Lamda_P_Range)) +…
min(Lamda_P_Range);
      c       = rand * (max(c_Range)-min(c_Range)) + min(c_Range);
      Nv      = rand * (max(Nv_Range)-min(Nv_Range)) + min(Nv_Range);
      Fw      = rand * (max(Fw_Range)-min(Fw_Range)) + min(Fw_Range);
      Jw      = rand * (max(Jw_Range)-min(Jw_Range)) + min(Jw_Range);
      Error_Sum = 0;

      for ii = 2:length(t),

         x(1) = FrtEnc(ii)*6.283185/256.0/Time;
         x(2) = RerEnc(ii)*6.283185/256.0/Time;
         if (x(2) ~= 0),
            LamdaC   = 1-x(1)/x(2);
%              if (LamdaC > 0.5)
%                 LamdaC = 0.5;
%              end

            Vv  = x(1)*Rw;     %initial velocity
            Fv  = c*Vv^2;      % wind drag force

            Mue = (2*MueP*Lamda_P*LamdaC) / (Lamda_P^2+LamdaC^2);
 % adhesion/friction coef.  (THE INPUT for the smallest curve)

         %%%%%%%%%%%%%%%%%% ODE Solver
%%%%%%%%%%%%%%%%%%%%%
            u = U(ii);
            u = (u-6.7831e-3 * x(2))/1.2;
            [T, X] = ode45('states',[0,tspan],x);
            x = X(length(X), [1,2]);

            LamdaM   = 1-x(1)/x(2);
%              if (LamdaM > 0.5)
%                 LamdaM = 0.5;
%              end

%           x(1) = FrtEnc(ii+1)*6.283185/256.0/Time;
%           x(2) = RerEnc(ii+1)*6.283185/256.0/Time;
%           LamdaC   = 1-x(1)/x(2);
            Error = (LamdaM - LamdaC)^2;
            Error_Sum = Error_Sum + Error;
            New_Error = Error_Sum;
         else
            Error_Sum = Error_Sum + 0;
         end
      end


                  if ((rand < exp((Curr_Error-New_Error)/temp)) | ((New_Error-Curr_Error)
< 0)),   % accept it!

         Curr_Error = New_Error
                               minE = min(minE, Curr_Error);
                               maxE = max(maxE, Curr_Error);
         if Curr_Error < Lowest_Error,
            Lowest_Error = Curr_Error
            save LowError3
         end
         if Curr_Error > Highest_Error,
            Highest_Error = Curr_Error
            save HighError3
         end
                  end

      LcaLitr = LcaLitr + 1;
         end

         % Update plot
         % Print information in command window
         fprintf('temp. = %f\n', temp);
         fprintf('Curr_Error = %f\n', Curr_Error);
         fprintf('[minE maxE] = [%f %f]\n', minE, maxE);
         fprintf('[LcaLitr TempItr] = [%d %d]\n\n', LcaLitr, TempItr);
         % Lower the temperature
         temp = temp*TempRatio;
   % Increment trial count
   TempItr = TempItr + 1;
end

fprintf('Compute Time = %f\n', toc);

save AnnealResult3

%%%%    control output plot   %%%%%%%%%%%5
for FrtEnc = 1:41, %90, %270,  %900
```

```
   for RerEnc = 1:41, %90, %270,   %900


        x(1) = FrtEnc*6.283185/256.0/Time;
        x(2) = RerEnc*6.283185/256.0/Time;

       Lamda   = 1-x(1)/x(2);
       if (Lamda > 0.5)
          Lamda = 0.5;
       end


       Vv      = x(1)*Rw;     %initial velocity
       Fv  = c*Vv^2;       % wind drag force

      bmin    = 0.999*(1-Lamda)/(Jw*x(2));
      bmax    = 1.001*(1-Lamda)/(Jw*x(2));
      b_hat   = sqrt(bmin*bmax);
%       Beta    = sqrt(bmax/bmin);
Beta = 1;
N   = 0.05;  % eta


      Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction
%coef.  (THE INPUT for the smallest curve)
      Mue_hat = -1239.8*Lamda.^6 + 2213.2*Lamda.^5 - 1590.6*Lamda.^4 + …
589.97*Lamda.^3 - 119.97*Lamda.^2 + 12.727*Lamda + 0.0;  % Nominal curve shifted
%up just a little

      Muedif = Mue-Mue_hat;

% Mue_hat = Mue;
   f_hat   = -((1-Lamda)*Fw*Rw)/(Jw*x(2)) + Mue_hat*[(Fv-Nv*nw)/(mv*Rw*x(2)) +..
((1-Lamda)*Rw*Nv)/(Jw*x(2))];

% F   = abs(f_hat - f);
      F   = abs([(Fv-Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))]*abs(Mue-
Mue_hat));


      s=Lamda-Lamda_d;

% Lam_dot =  [(Fv-Nv*nw*Mue)/(mv*Rw*x(2)) - ((1-Lamda)*Fw*Rw)/(Jw*x(2)) - ((1-
Lamda)*Rw*Nv*Mue)/(Jw*x(2))] + ((1-Lamda)*u)/(Jw*x(2));

      u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='
```

```
%K   = F + N;             % Gain

       K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

       u = 1/b_hat*(u_hat - K*sign(s));


      if (u > 1023)
         u = 1023;
      end
      if (u < -1023)
         u = -1023;
      end

       U_out(FrtEnc, RerEnc) = u;

    end
end
figure
mesh(U_out)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%
%%%%    minumum control output plot   %%%%%%%%%%%%%%%%%%%%%%%%

load LowError
for FrtEnc = 1:41, %90, %270,  %900
    for RerEnc = 1:41, %90, %270,   %900


         x(1) = FrtEnc*6.283185/256.0/Time;
         x(2) = RerEnc*6.283185/256.0/Time;

        Lamda   = 1-x(1)/x(2);
        if (Lamda > 0.5)
            Lamda = 0.5;
        end



        Vv      = x(1)*Rw;     %initial velocity
        Fv  = c*Vv^2;       % wind drag force

       bmin    = 0.999*(1-Lamda)/(Jw*x(2));
       bmax    = 1.001*(1-Lamda)/(Jw*x(2));
       b_hat   = sqrt(bmin*bmax);
%       Beta    = sqrt(bmax/bmin);
Beta = 1;
```

```
N   = 0.05;  % eta


      Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction
%coef.  (THE INPUT for the smallest curve)

      Mue_hat = -1239.8*Lamda.^6 + 2213.2*Lamda.^5 - 1590.6*Lamda.^4 + …
589.97*Lamda.^3 - 119.97*Lamda.^2 + 12.727*Lamda + 0.0;  % Nominal curve shifted
%up just a little

      Muedif = Mue-Mue_hat;



% Mue_hat = Mue;

      f_hat   = -((1-Lamda)*Fw*Rw)/(Jw*x(2)) + Mue_hat*[(Fv-Nv*nw)/(mv*Rw*x(2))
+ ((1-Lamda)*Rw*Nv)/(Jw*x(2))];

% F   = abs(f_hat - f);
      F   = abs([(Fv-Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))]*abs(Mue-
Mue_hat));


      s=Lamda-Lamda_d;

% Lam_dot =  [(Fv-Nv*nw*Mue)/(mv*Rw*x(2)) - ((1-Lamda)*Fw*Rw)/(Jw*x(2)) - ((1-
Lamda)*Rw*Nv*Mue)/(Jw*x(2))] + ((1-Lamda)*u)/(Jw*x(2));

      u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='

%K   = F + N;             % Gain

      K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

      u = 1/b_hat*(u_hat - K*sign(s));


     if (u > 1023)
        u = 1023;
     end
     if (u < -1023)
        u = -1023;
     end

      U_out(FrtEnc, RerEnc) = u;
```

```
   end
end
figure
mesh(U_out)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%
%%%%    maximumum control output plot   %%%%%%%%%%%

load HighError
for FrtEnc = 1:41, %90, %270,  %900
   for RerEnc = 1:41, %90, %270,   %900


        x(1) = FrtEnc*6.283185/256.0/Time;
        x(2) = RerEnc*6.283185/256.0/Time;

       Lamda   = 1-x(1)/x(2);
       if (Lamda > 0.5)
          Lamda = 0.5;
       end


       Vv      = x(1)*Rw;     %initial velocity
       Fv  = c*Vv^2;       % wind drag force

      bmin    = 0.999*(1-Lamda)/(Jw*x(2));
      bmax    = 1.001*(1-Lamda)/(Jw*x(2));
      b_hat   = sqrt(bmin*bmax);
%       Beta    = sqrt(bmax/bmin);
Beta = 1;
N   = 0.05;  % eta


      Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction
coef.  (THE INPUT for the smallest curve)
      Mue_hat = -1239.8*Lamda.^6 + 2213.2*Lamda.^5 - 1590.6*Lamda.^4 +
589.97*Lamda.^3 - 119.97*Lamda.^2 + 12.727*Lamda + 0.0;  % Nominal curve shifted
up just a little

      Muedif = Mue-Mue_hat;



% Mue_hat = Mue;
```

```
      f_hat   = -((1-Lamda)*Fw*Rw)/(Jw*x(2)) + Mue_hat*[(Fv-Nv*nw)/(mv*Rw*x(2))
+ ((1-Lamda)*Rw*Nv)/(Jw*x(2))];

% F   = abs(f_hat - f);
      F   = abs([(Fv-Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))]*abs(Mue-
Mue_hat));


      s=Lamda-Lamda_d;

% Lam_dot =  [(Fv-Nv*nw*Mue)/(mv*Rw*x(2)) - ((1-Lamda)*Fw*Rw)/(Jw*x(2)) - ((1-
Lamda)*Rw*Nv*Mue)/(Jw*x(2))] + ((1-Lamda)*u)/(Jw*x(2));

      u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='

%K   = F + N;             % Gain

      K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

      u = 1/b_hat*(u_hat - K*sign(s));


     if (u > 1023)
        u = 1023;
     end
     if (u < -1023)
        u = -1023;
     end

      U_out(FrtEnc, RerEnc) = u;

    end
end
figure
mesh(U_out)

tic;
```

I.3 Simulated Annealing Code to Find Gains

```
clear all;
close all;
clc;

tic;
```

```
%%%%%%%%%%%%  Sliding Mode Setup  %%%%%%%%%%%%%%%%
global Mue Fv mv nw Nv Rw Fw Jw Ke V Ra x2

%%%%%%%% Perameters from Simulated Annealing with Lamda, Mue, and Lamda_d
%at very low ranges
%%%%%%%%%%%%%% of values
MueP    = 0.0102;
Lamda_P = 0.0105;
Lamda_d = 0.008;  % desired Lamda from simulated annealing      % desired path
c       = 0.0019; %0.022;   % wind drag force coeficient

mv      = 1.6516;        % mass of car (Kg)
nw  = 2;          % number of driving wheels
Nv  = 1.4585        % normal force to ground (Kg)
Rw  = 0.032055;        % radius of wheel (Meters)
Fw  = 0.1951;       % wheel viscous friction (N*Sec/Meter)
Jw  = 0.0923;      %0.5*32.0*0.02659^2              %0.5*m*Rw^2; % inertia (Kg*m^2)

Ke = 0.0139;  %6.7831e-3;      %
Ra = 1.2229;           % Armeture resistance
Volts = 12.0;       % Maximum voltage supplied to system (aka: battery)

N_Range     = [0.0001: 0.00001: 9.0];
Phi_Range   = [0.01: 0.01: 5.0];
min_Range   = [0.01: 0.01: 9.0];
max_Range   = [0.01: 0.01: 9.0];
Beta_Range  = [0.01: 0.01: 12.0];

Error_Over = 0;

tspan = 0.5;
Time = tspan;
Rfw = 1.20175;
Rrw = 1.262;

tic;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  Setup for Annealing
%%%%%%%%%%%%%%%%%%%%
Lowest_Error = 100000;
Highest_Error = 2;
Curr_Error = inf;
temp = 10000; % Choose the temperature to be large enough
fprintf('Initial Curr_Error = %f\n\n',Curr_Error);

NumCombo = 1;
```

```
% rand('seed', 31415927)
rand('state',sum(100*clock))  % Resets it to a different state each time.

MaxTempItr = NumCombo*100;              % Max. # of trials at a temperature
MaxLcaLitr = NumCombo*10;               % Max. # of acceptances at a temperature
StopTolerance = 0.0001;            % Stopping tolerance
TempRatio = 0.90;           % Temperature decrease ratio
TempItr = 0;           % Number of trial moves

minE = inf;                 % Initial value for min. Curr_Error
maxE = -1;                  % Initial value for max. Curr_Error

%%%%%%%%%%%%%%%%%%%%%  Annealing loop
%%%%%%%%%%%%%%%%%%%%%%%%%
while (TempItr < MaxTempItr),  %(((maxE-minE)/maxE > StopTolerance) & (TempItr <
MaxTempItr)),

   minE = inf;              % Initial value for min. Curr_Error
   maxE = -1;               % Initial value for max. Curr_Error
         LcaLitr = 0;       % Number of local iterations at a certain temp.

         while ((Curr_Error > 0.005) & (LcaLitr < MaxLcaLitr)),

      N      = rand * (max(N_Range)-min(N_Range)) + min(N_Range);
      Phi    = rand * (max(Phi_Range)-min(Phi_Range)) + min(Phi_Range);
      minn   = rand * (max(min_Range)-min(min_Range)) + min(min_Range);
      maxx   = rand * (max(max_Range)-min(max_Range)) + min(max_Range);
      Beta   = rand * (max(Beta_Range)-min(Beta_Range)) + min(Beta_Range);

      Error_Sum = 0;
      rand('seed', 31415927)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
      FrtEnc = 1;
      RerEnc = 1;
       x(1) = FrtEnc*6.283185/256.0/Time;
       x(2) = RerEnc*6.283185/256.0/Time;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%
      x2 = x(2);  % value sent to ODE45 Function (rear wheel value)

      Lamda   = 1-x(1)/x(2);
      if (Lamda > 0.5)
         Lamda = 0.5;
      end
```

```
        Vv      = x(1)*Rw;     %initial velocity
        Fv      = c*Vv^2;       % wind drag force

        bmin    = minn*(1-Lamda)*Ke/(Ra*Jw*x(2));
        bmax    = maxx*(1-Lamda)*Ke/(Ra*Jw*x(2));
        b_hat   = sqrt(bmin*bmax);

        Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction
%coef.

Mue_hat = -9546.3*Lamda.^10 + 50170*Lamda.^9 - 1.1328e+005*Lamda.^8 +
1.4366e+005*Lamda.^7 - 1.1227e+005*Lamda.^6 + 55813*Lamda.^5 -
17584*Lamda.^4 + 3385.6*Lamda.^3 - 367.85*Lamda.^2 + 19.217*Lamda.^1 +
0.14133;

        Muedif = Mue-Mue_hat;

        f_hat   = [Fv/(mv*Rw*x(2)) - (1-Lamda)*Rw/(Jw*x(2)) - (1-
Lamda)*Ke^2/(Ra*Jw*x(2))] - [nw*Nv/(mv*Rw*x(2)) + (1-
Lamda)*Rw*Nv/(Jw*x(2))]*Mue_hat;

% F   = abs(f_hat - f);
      F   = abs([(Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))] * (Mue-…
Mue_hat));
% F = 0;  % for debuging

        s=Lamda-Lamda_d;

        u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='


        K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

% u = 1/b_hat*(u_hat - K*sign(s));

        if (abs(s) < Phi)
           sat = s/Phi;
        else
           sat = sign(s);
        end
        u = 1/b_hat*(u_hat - K*sat);

        PerVolt = u/1023;                  % Percent Voltage to motor
        V = PerVolt * Volts;                 % actual voltage supplied to motor
```

```
    if (V > 12)
       V = 12;
    end
    if (V < -12)
       V = -12;
    end

    for ii = 1:500,

       t(ii) = ii*tspan;

    %  PLOT VAR
       S_out(ii)       = s;
       Lam_out(ii)     = Lamda;
       U_out(ii)       = u;
       x_out(ii,:)     = x;
       Vv_out(ii)      = Vv;
       Mue_out(ii)     = Mue;
       Fv_out(ii)      = Fv;
       Mue_hat_out(ii) = Mue_hat;
       f_hat_out(ii)   = f_hat;
       F_out(ii)       = F;
       K_out(ii)       = K;
       Muedif_out(ii)  = Muedif;
       bmin_out(ii)    = bmin;
       bmax_out(ii)    = bmax;
       b_hat_out(ii)   = b_hat;
       V_out(ii)       = V;


  %%%%%%%%%%%%%%%%%% ODE Solver
%%%%%%%%%%%%%%%%%%%%%%
       [T, X] = ode45('states2',[0,tspan],x);
       x = X(length(X), [1,2]);

       x2 = x(2);
       Lamda = 1-x(1)/x(2);

       Lamda   = 1-x(1)/x(2);
       if (Lamda > 0.5)
          Lamda = 0.5;
       end

       Vv = x(1)*Rw;
       Fv  = c*Vv^2;       % wind drag force

       bmin    = minn*(1-Lamda)*Ke/(Ra*Jw*x(2));
```

```
        bmax    = maxx*(1-Lamda)*Ke/(Ra*Jw*x(2));
        b_hat   = sqrt(bmin*bmax);
%   b_hat = 1; % for debuging
%   Beta    = sqrt(bmax/bmin);

        Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  %
adhesion/friction coef.
%           Mue_hat = 0.052;
%   Mue_hat = Mue;   %  for debuging
Mue_hat = -9546.3*Lamda.^10 + 50170*Lamda.^9 - 1.1328e+005*Lamda.^8 + …
1.4366e+005*Lamda.^7 - 1.1227e+005*Lamda.^6 + 55813*Lamda.^5 - …
17584*Lamda.^4 + 3385.6*Lamda.^3 - 367.85*Lamda.^2 + 19.217*Lamda.^1 + …
0.14133;

        Muedif = Mue-Mue_hat;



        f_hat   = [Fv/(mv*Rw*x(2)) - (1-Lamda)*Rw/(Jw*x(2)) - (1-
Lamda)*K^2/(Ra*Jw*x(2))] - [nw*Nv/(mv*Rw*x(2)) + (1-
Lamda)*Rw*Nv/(Jw*x(2))]*Mue_hat;

% F   = abs(f_hat - f);
       F   = abs([(Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))] * (Mue-…
Mue_hat));
% F = 0;  % for debuging
        s=Lamda-Lamda_d;

        u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='

        K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

        if (abs(s) < Phi)
           sat = s/Phi;
        else
           sat = sign(s);
        end
        u = 1/b_hat*(u_hat - K*sat);

%    u = 1/b_hat*(u_hat - K*sign(s));

        PerVolt = u/1023;               % Percent Voltage to motor
        V = PerVolt * Volts;             % actual voltage supplied to motor
        if (V > 12)
           V = 12;
        end
        if (V < -12)
           V = -12;
```

```
         end

            Error1 = (s)^2;
            if (ii > 1)
               Error2 = ((V_out(ii) - V_out(ii-1)) / Time)^2;
            else
               Error2 = ((V_out(ii) - 0) / Time)^2;
            end
            Error = 0.3*Error1 + 0.7*Error2;
            Error_Sum = Error_Sum + Error;

         %%%%%  If erro already over 3 no need to continue calculating error %%%%
         if (isnan(Error_Sum) == 1)
            Error_Over = 1
            break;
         end

      end
      New_Error = Error_Sum/ii;

          if (((rand < exp((Curr_Error-New_Error)/temp)) | ((New_Error-Curr_Error) < …
0)) & Error_Over == 0),         % accept it!

         Curr_Error = New_Error
                          minE = min(minE, Curr_Error);
                          maxE = max(maxE, Curr_Error);

         if Curr_Error < Lowest_Error,
            Lowest_Error = Curr_Error
            save LowError1
         end
         if Curr_Error > Highest_Error,
            Highest_Error = Curr_Error
            save HighError1
         end
                  end
      Error_Over = 0;     % resets too high error value indicator

      LcaLitr = LcaLitr + 1;
         end

         % Update plot


         % Print information in command window
         fprintf('temp. = %f\n', temp);
         fprintf('Curr_Error = %f\n', Curr_Error);
```

```
        fprintf('[minE maxE] = [%f %f]\n', minE, maxE);
        fprintf('[LcaLitr TempItr] = [%d %d]\n\n', LcaLitr, TempItr);
        % Lower the temperature
        temp = temp*TempRatio;
   % Increment trial count
   TempItr = TempItr + 1;

   rand('state',sum(100*clock))  % Resets it to a different state each time.

end

fprintf('Compute Time = %f\n', toc);


save AnnealResult1

      N
      Phi
      minn
      maxx
      Beta
```

Appendix J

Robot Car Control Code

```
#include <pic18.h>
#include <stdlib.h>

#include <math.h>

unsigned char RerEnc;
unsigned char FrtEnc;
unsigned int Pwr;
unsigned int LstFrt;
unsigned int LstRer;
float x1, x2, Vv, K, Mue, Lamda, Fv, u_hat, s, bmin, bmax, b_hat, Mue_hat, f_hat, F;  signed int sgnS;

float u = 1023.0;
float Temp1 = 0.0;
float Temp2 = 0.0;
float Temp3 = 0.0;
float Temp4 = 0.0;
float Temp5 = 0.0;
float Temp6 = 0.0;
float Temp7 = 0.0;
float Muedif = 0.0;
float Ke        = 0.0159;
float Ra        = 1.506;
float MuePmax = 0.94;
float MuePmin = 0.008;
float Lamda_Pmax = 0.3;
float Lamda_Pmin = 0.008;
float MLmax       = 0.0;
float MLmin       = 0.0;
float LPmax       = 0.0;
float LPmin       = 0.0;
float MLP         = 0.0;
float LmP         = 0.0;
float N           = 0.2;     // Etta
float minn        = 0.9;
float maxx        = 1.1;  //  7.3130;  1.1;
float Beta        = 1.4;
float MueP        = 0.0745;          //peek adhesion/friction coef.
float Lamda_P     = 0.1551;          //peek Lamda on curve
float Jw          = 0.1088;          //0.5*m*Rw^2; inertia (Kg*m^2)
float Fw          = 0.4507;          //wheel viscous friction (N*Sec/Meter) ?0.51
float Nv          = 1.3009;          //(N)  normal force of vehicle
float Lamda_d     = 0.05;           //desired path
float c           = 0.005;          //wind drag force coeficient
float Rw          = 0.0320548;      //radius of rear wheel
unsigned int nw   = 2;              //number of driving wheels
float mv          = 1.6516;         //mass of vehicle (Kelo-grams)
float Time        = 0.005;          // (256-TMR0[value])*Prescale[value-Option Reg]*200nSec = approx
                                    //timer3 for interrupt time
/* service routine for timer 0 interrupt */

void interrupt isr(void)
{
//          unsigned int tics = TMR0;

            TMR3H = 0x3C; //
            TMR3  = 0xF3;   //For reseting Timmer 3 value

                        if (LstRer > TMR0)
                                    RerEnc = 65535-LstRer + TMR0;  //remember to start counting from '0'
                        else
                                    RerEnc = TMR0 - LstRer;
                        LstRer = TMR0;

                        if (LstFrt > TMR1)
                                    FrtEnc = 65535-LstFrt + TMR1;
                        else
                                    FrtEnc = TMR1 - LstFrt;
                        LstFrt = TMR1;
/*********************************************/
// x1 = vehicle's angular velocity
// x2 = driving wheel's angular velocity

                        x1 = FrtEnc*6.283185/256.0/Time*1.09076469253;  //=Vv/Rw = Wv = #tics*
                                                            //(6.28[rad/rev] / 256[tics/rev]) / elapsed time [second]
                        x2 = RerEnc*6.283185/256.0/Time;              //  =Ww

            if  (RerEnc>15)  // on 12 volts after Robot car has started moving
            {
                        Lamda = 1-x1/x2;

                        Temp1 = 1-Lamda;
                        Temp2 = Jw*x2;
                        Temp7 = Temp2*Ra;
                        Temp3 = Temp1*Ke/Temp7;

                        Vv = x1*Rw;                         // radius of wheel (Rw): Vv = x1(Wv)*Rw
                        Fv = c * Vv*Vv;                     //% wind drag force

            //          b = (1-Lamda)/(Jw*x(2));
                        bmin = minn*Temp3;
                        bmax = maxx*Temp3;
                        b_hat= sqrt(bmax*bmin);

/************************************************/
                        Temp7 = Lamda*Lamda;
                        Temp3 = MLmax*Lamda;
                        Temp4 = MLmin*Lamda;
                        Temp5 = LPmax+Temp7;
```

```
                    Temp6 = LPmin+Temp7;

        //Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction coef.
                    Mue = (MLP*Lamda)/(LmP+Temp7);

                    Mue_hat = (Temp3/Temp5 + Temp4/Temp6)/2 + 0.001;
/*************************************************/

                    Temp4 = mv*Rw*x2;
                    Temp5 = (nw*Nv)/Temp4;
                    Temp3 = Temp5 + Temp1*Rw*Nv/Temp2;

        f_hat = (Fv/Temp4 - Temp1*Rw*Fw/Temp2 - Temp1*Ke*Ke/(Ra*Jw)) - Temp3 * Mue_hat;
                    Muedif    = (Mue-Mue_hat);  // reuse bmin to save memory space

                    F    = fabs(Temp3 * Muedif);

                    s = Lamda - Lamda_d;       //s=Lamda-Lamda_d;

                    //u_hat   = -f_hat*[(Fv-Nv*nw*Mue)/(mv*Rw*x(2)) - ((1-Lamda)*Fw*Rw)/(Jw*x(2)) –
                            //((1-Lamda)*Rw*Nv*Mue)/(Jw*x(2))];
                    u_hat = -f_hat;

//                  K = F + N;

                    K  =  Beta*(F+N) + (Beta-1)*fabs(u_hat);
                    if (s>0)
                            sgnS = 1;
                    if (s<0)
                            sgnS = -1;

                    u = 1/b_hat*(u_hat - K*sgnS);  // on 12 volts
        }
        if (RerEnc < 16)
                    u = 1023;

                    if (u>=0)
                    {
                            PORTD = 0x08;  // Bit RD2 = 0 for forward direction when Y1 on
                                           // MUXconnected to IN1 on H-bridge
                            if (u > 1023)
                                    u = 1023;
                    }
                    else
                    {
                            PORTD = 0x0C;  // Bit RD2 = 1 for reverse direction
                            if (u < -1023)
                                    u = 1023;
                            else
                                    u = -u;
                    }
                    Pwr = u;  //re-use "FrtEnc" to convert "u" to an integer
                    CCPR2L  = Pwr>>2; // & 1020;
                    CCP2CON = 0xf ^ (Pwr<<4);

//                  PORTB   = RerEnc;                  // Eight bit value when Xmittng to DCC
                                           // OR //
                    PORTB   = CCPR2L;                    // Control value u = 10 bit value
                    PORTD   = PORTD + (Pwr<<4);

             TMR3IF = 0;    /* reset Timmer 3 flag  PIR2<1>*/
            }
void main(void)
{
//          Direction of motor & to turn on data collection
            TRISE = 0x00;  // put port D in standard I/O
            PORTD = 0x08;  // set port RD2 as LOW for forward direction
                                        // and port RD3 HIGH to turn test platform output ON
            TRISD = 0x00;  // set port D as output
//  10 bit PWM value output  (8 upperbits on PORTB and lower 2 bits on PORTD<7,6>
            PORTB = 0x00;  // set port RB as 0's
            TRISB = 0x00;  // set port B as output
//  PWM for TMR2
            PR2     = 0xFF;     // PWM period
            CCPR2L  = 0x55;  // Upper 8 bits of 10 bit PWM duty cycle

            T2CON   = 0x05;  // Set the TMR2 prescale value & enable Timer2 0x05 = (9.77kHz)
            CCP2CON = 0xFF;            // Lower 2 bits of 10 bit PWM duty cycle & Put in PWM Mode
            TRISC   = 0x01;  // Make CCP2/RC1 pin an output for the PWM2 signal
//  Timmer 0 Counter
            T0CON = 0xA8;  ///8= no prescaler
//          Timmer 1 Counter
            T1CON = 0x87;
//          Timmer 3 Interupt on Over Flow
            T3CON = 0x85;  // B5?  85 = no prescaler

            TMR0H = 0;
            TMR0  = 0;

            TMR1H = 0;
            TMR1  = 0;

            TMR3H = 0x3C;   // 5ms = 3CF3;
            TMR3  = 0xF3;

            INTCON  = 0xC0;  // Enable interupt

            MLP     = 2*MueP*Lamda_P;
             LmP    = Lamda_P*Lamda_P;
            MLmax  = 2*MuePmax*Lamda_Pmax;
            MLmin  = 2*MuePmin*Lamda_Pmin;
            LPmax  = Lamda_Pmax*Lamda_Pmax;
            LPmin  = Lamda_Pmin*Lamda_Pmin;

            FrtEnc = 0;
            RerEnc = 0;

            PIE1 = 0x00;
            PIE2 = 0x02;  // Enables TMR3 Overflow Interrupt Enable bit

            CCPR2L  = 0xFF;  // Upper 8 bits of 10 bit PWM duty cycle

  while (1)
  {  /* infinite loop */
```

```
            NOP();
  }
}
```

Appendix K

Matlab Continous Simulation Code

```
% By:  Mark Morton
% Sliding Mode Control Simulation for Tracktion Control

clc;
clear all;
close all;

% digits(7);

global Mue Fv mv nw Nv Rw Fw Jw Ke V Ra x2

Rfw = 1.20175;
Rrw = 2.524;
Ratio = Rrw/2.2035;

MueP    = 0.0745;
Lamda_P = 0.1551;
Lamda_d = 0.05;  %0.0018;  % desired Lamda from simulated annealing      % desired path
c       = 0.005; %0.022;   % wind drag force coeficient

mv      = 1.6516;        % mass of car (Kg)
nw  = 2;          % number of driving wheels
Nv  = 1.3009        % normal force to ground (N)
Rw  = 0.0320548;         % radius of wheel (Meters)
Fw  = 0.4507;       % wheel viscous friction (N*Sec/Meter)
Jw  = 0.1088;      %0.5*32.0*0.02659^2             %0.5*m*Rw^2; % inertia (Kg*m^2)

Ke = 0.0159;  %6.7831e-3;      %
Ra = 1.506;   %5;  %1.506;           % Armeture resistance
Volts = 12.0;       % Maximum voltage supplied to system (aka: battery)

N    = 0.2;
minn = 0.9;
maxx = 1.1;
Beta = 1.4;

tspan = 0.005;       % initialize step size
Time = tspan;

x(1) = 56.24; %10
x(2) = 83.45; %17
```

```
x2 = x(2);  % value sent to ODE45 Function (rear wheel value)

Lamda   = 1-x(1)/x(2);

Vv      = x(1)*Rw;     %initial velocity
Fv  = c*Vv^2;       % wind drag force

   bmin    = minn*(1-Lamda)*Ke/(Ra*Jw*x(2));
   bmax    = maxx*(1-Lamda)*Ke/(Ra*Jw*x(2));
   b_hat   = sqrt(bmin*bmax);

Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction coef.  (THE
                                                                   %INPUT for the smallest curve)
%%%% 6th degree polynomial
Mue_hat = -3.554*Lamda.^6 + 11.65*Lamda.^5 +...
      -14.974*Lamda.^4 + 9.5802*Lamda.^3 +...
      -3.1673*Lamda.^2 + 0.45142*Lamda.^1 +...
      0.031374;

Muedif = Mue-Mue_hat;


f_hat   = [Fv/(mv*Rw*x(2)) - (1-Lamda)*Rw*Fw/(Jw*x(2)) - (1-Lamda)*Ke^2/(Ra*Jw)] -
[nw*Nv/(mv*Rw*x(2)) + (1-Lamda)*Rw*Nv/(Jw*x(2))]*Mue_hat;

% F   = abs(f_hat - f);
F   = abs([(Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))] * (Mue-Mue_hat));
% F = 0;  % for debuging

s=Lamda-Lamda_d;

u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='

K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

      u = 1/b_hat*(u_hat - K*sign(s));

PerVolt = u/1023;                 % Percent Voltage to motor
V = PerVolt * Volts;               % actual voltage supplied to motor

if (V > 12)
   V = 12;
end
if (V < -12)
   V = -12;
end

Lam_dot = [(Fv-nw*Nv*Mue)/(mv*Rw*x(2)) - (1-Lamda)*Rw*((Fw+Nv*Mue)/(Jw*x(2))) - (1- …
             Lamda)*Ke^2/(Ra*Jw)] + (1-Lamda)*Ke/(Ra*Jw*x(2)) * V;


tic;

for ii = 1:200000,
   t(ii) = ii*tspan;
```

```
  %  PLOT VAR
  S_out(ii)       = s;
  Lam_out(ii)     = Lamda;
  Lam_dot_out(ii) = Lam_dot;
  U_out(ii)       = u;
  x_out(ii,:)     = x;
  Vv_out(ii)      = Vv;
  Mue_out(ii)     = Mue;
  Fv_out(ii)      = Fv;
  Mue_hat_out(ii) = Mue_hat;
  f_hat_out(ii)   = f_hat;
  F_out(ii)       = F;
  K_out(ii)       = K;
  Muedif_out(ii)  = Muedif;
  bmin_out(ii)    = bmin;
  bmax_out(ii)    = bmax;
  b_hat_out(ii)   = b_hat;
  V_out(ii)       = V;

%%%%%%%%%% ODE Solver %%%%%%%%%%%%%%%%%%%%%
  [T, X] = ode45('states2',[0,tspan],x);
  x = X(length(X), [1,2]);

  x2 = x(2);
  Lamda   = 1-x(1)/x(2);

  Vv = x(1)*Rw;
  Fv  = c*Vv^2;       % wind drag force

  bmin    = minn*(1-Lamda)*Ke/(Ra*Jw*x(2));
  bmax    = maxx*(1-Lamda)*Ke/(Ra*Jw*x(2));
  b_hat   = sqrt(bmin*bmax);

   Mue = (2*MueP*Lamda_P*Lamda) / (Lamda_P^2+Lamda^2);  % adhesion/friction coef.

%%% 6th degree polynomial
Mue_hat = -3.554*Lamda.^6 + 11.65*Lamda.^5 +...
     -14.974*Lamda.^4 + 9.5802*Lamda.^3 +...
     -3.1673*Lamda.^2 + 0.45142*Lamda.^1 +...
     0.031374;

  f_hat   = [Fv/(mv*Rw*x(2)) - (1-Lamda)*Rw*Fw/(Jw*x(2)) - (1-Lamda)*Ke^2/(Ra*Jw)] - …
            [nw*Nv/(mv*Rw*x(2)) + (1-Lamda)*Rw*Nv/(Jw*x(2))]*Mue_hat;

% F   = abs(f_hat - f);
  F   = abs([(Nv*nw)/(mv*Rw*x(2)) + ((1-Lamda)*Nv*Rw)/(Jw*x(2))] * (Mue-Mue_hat));

   s=Lamda-Lamda_d;

   u_hat = (-f_hat+0); %Lamda_d_dot);  %  /b_hat is taken care of in 'u='

   K  =  Beta*(F+N) + (Beta-1)*abs(u_hat);

        u = 1/b_hat*(u_hat - K*sign(s));

   PerVolt = u/1023;                 % Percent Voltage to motor
   V = PerVolt * Volts;               % actual voltage supplied to motor
   if (V > 12)
      V = 12;
   end
   if (V < -12)
      V = -12;
   end

   Lam_dot = [(Fv-nw*Nv*Mue)/(mv*Rw*x(2)) - (1-Lamda)*Rw*((Fw+Nv*Mue)/(Jw*x(2))) - (1- …
           Lamda)*Ke^2/(Ra*Jw)] + (1-Lamda)*Ke/(Ra*Jw*x(2)) * V;

   if (mod(ii,1000) == 1)
       Lamda-Lamda_d
   end

end

   save Simulation1
toc

figure;
plot(t,S_out);
title('S');
xlabel('Time (Seconds)');
ylabel('s');

figure;
plot(t,Lam_out);
title('Lamda');
xlabel('Time (Seconds)');
ylabel('Lamda');

figure;
plot(t,V_out);
title('Volts');
xlabel('Time (Seconds)');
ylabel('V');

figure;
plot(t,bmin_out)
title('bmin')
xlabel('Time (Seconds)');
ylabel('bmin');

figure;
plot(t,bmax_out)
title('bmax')
xlabel('Time (Seconds)');
ylabel('bmax');

figure;
plot(t,b_hat_out)
title('b_Hat')
xlabel('Time (Seconds)');
ylabel('b_Hat');

figure;
plot(t,U_out);
```

```
title('Percent Control Value U');
xlabel('Time (Seconds)');
ylabel('u');

figure;
plot(t,x_out(:,1));
title('Front Wheel Angular Velocity');
xlabel('Time (Seconds)');
ylabel('X_1');

figure;
plot(t,x_out(:,2));
title('Rear Wheel Angular Velocity');
xlabel('Time (Seconds)');
ylabel('X__2');

figure;
plot(x_out(:,2),x_out(:,1));
title('X_2 vs X_1');
xlabel('X_2');
ylabel('X_1');

figure;
plot(t,Vv_out);
title('Linear Velocity');
xlabel('Time (Seconds)');
ylabel('Vv (Meters/second)');

figure;
plot(t,Mue_out);
title('Mue');
xlabel('Time (Seconds)');
ylabel('Mue');

figure;
plot(t,Fv_out);
title('Fv');
xlabel('Time (Seconds)');
ylabel('Fv');

figure;
plot(t,Mue_hat_out);
title('Mue_Hat');
xlabel('Time (Seconds)');
ylabel('Mue_Hat');

figure;
plot(t,f_hat_out);
title('f__hat');
xlabel('Time (Seconds)');
ylabel('f__hat');

figure;
plot(t,F_out);
title('Maximum boundary F');
xlabel('Time (Seconds)');
ylabel('F');
```

```
figure;
plot(t,K_out);
title('Gain K');
xlabel('Time (Seconds)');
ylabel('K');

figure;
plot(t,Muedif_out);
title('Difference betweeen Mue & Mue__hat');
xlabel('Time (Seconds)');
ylabel('Muedif');

figure;
plot(Lam_out, Lam_dot_out);
title('Sliding Surf');
xlabel('Lamda');
ylabel('Lamda^')
```