Case Study: Model-Based Design of A Closed-Loop Speed Controller for A Hybrid System

Ronald Picard, Wang Yuxing, Sanchita Basak, Shreyas Ramakrishna
Vanderbilt University
Nashville, TN, USA

***Abstract*—Model-Based Design (MBD) has been used for decades to design controllers for hybrid systems. MBD requires special considerations when modeling, designing, and analyzing hybrid systems. Hybrid systems are often modeled and simulated in order to tune the controller parameters before testing on target hardware. We present an approach to modeling, design, and analysis of a closed-loop RC car speed controller. We model this system with state-space equations using estimated parameters, and tune a discrete PID controller so that an appropriate system response is achieved. With a successful model, we utilize Simulink's support package for Raspberry Pi to automatically generate and run an executable model on the target hardware. We present and discuss the challenges associated with hybrid system modeling and testing. With varied experiments we show the modeling challenges associated with hybrid systems and the gap between simulated models and executable models for controlling these systems. We show a successful PID controller in a Simulink simulation followed by unacceptable performance on target hardware caused by the high CPU usage of Simulink's auto generated code and the strict real-time sampling time requirements of our controller.**

***Index Terms*—Cyber Physical Systems, Hybrid Systems, PID controller, MATLAB, Simulink, Raspberry Pi.**

I. Introduction

The use of Model-Based Design (MBD) requires unique considerations when modeling, designing, and analyzing Hybrid Systems. Hybrid system control loops are often modeled and simulated in order to tune the controller parameters and predict the response before testing on target hardware. The design of these control loops often makes use of generative programming in order to prevent the introduction of software bugs and abstract away the low-level details from the system designer. In addition, hybrid systems introduce a degree of complexity that extends beyond the correctness of the software itself to processor scheduling, execution times, sample rates, and the conversion between analog an digital systems. We present and discuss a case study on the complexities of these systems utilizing Simulink's modeling, simulation, and automatic code generation capabilities in the study of a speed controller system for a micro RC car.

Outline: Section II provides an introduction into the problem space. The subsection II-A provides a background on the basics of model-based design, hybrid systems, state-space models, closed-loop control, PID control, and Simulink. Section II-B provides a description of the hardware testbed and its components, as well as the PWM controlled DC motor. The Section III provides the state-space modeling approach, and the estimated parameters. Section IV provides a description of the Simulink implementation for both the model Simulation and the Target Hardware. Section V provides the results of the simulation and target hardware test. The section also analyzes anomalies that occur in Hybrid Systems and describes the limitations of simulation for predicting these anomalies.

II. Background

In this section we introduce some fundamental topics required to understand the different sections of this work.

A. Basics

1) Model-Based Design: Provides an abstract way to identify and manipulate components of a system. Models may be easily understood by users and developers alike and provide an efficacious way of transferring domain knowledge from person to person. In addition, the use of automatic code generation from models relieves the system developers from the need to know low-level details of the operating systems and hardware components of the target devices.

2) Hybrid Systems: Consist of computing devices that interact with the world through the use of sensors and actuators. These systems require special considerations due to the nature of analog and digital signal conversions. In addition, these systems often requires special timing considerations in order to ensure proper functionally.

3) State-Space Models: Are linear a linear algebraic approach modeling systems that are described by differential equations. The common form of this is shown below in Equation 1. The matrix A is the state-variable transition matrix based on the state vector x at time t, matrix B is the state-variable transition matrix based on the input vector u at time t, matrix C is the output matrix based on the state vector x at time t, and matrix D is output matrix based on the input u at time t.

$$\begin{aligned} \vec{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned} \tag{1}$$

4) Closed-Loop Control: Closed-loop control systems involves feedback loop (usually negative) in which the measured response is subtracted from the reference value to provide an error signal to the controller. The controller then

converts this error signal into an appropriate control signal that causes the plant to provide a response closer to that of the reference value. An example negative feedback control block diagram is provided in Figure 1.

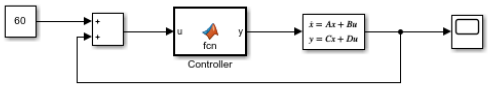

Fig. 1: Example Close-Loop Negative Feedback Control System

5) PID Control: Proportional, Integral, Derivative (PID) control is a common technique in which three gain values are tuned to provide an appropriate system response. The proportional control of PID controllers is simply the gain (amplifier) value of the signal. The integral portion of the control takes into account the historical values of the error signal and adjusts accordingly. Integral controllers are commonly utilized to remove steady-state error. Derivative control takes into the current rate of change of the system response and is commonly used to prevent a high overshoot of a steady-state value. A simple block of a PID controller is shown in Figure 2.

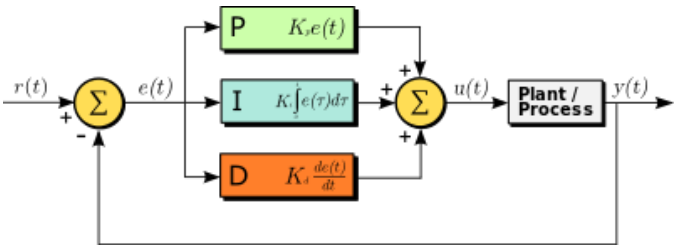

Fig. 2: PID Controller [1]

6) Simulink: Simulink is a model-based design tool that supports simulation and automatic code generation to target hardware. It provides a graphical framework for developing and simulating control software. The tool allows for the model-based creation of actor oriented diagrams in which results can be displayed to a scope. The tool allows simulation/execution time steps and sample time step to be specified. A limitation of Simulink is that it requires high a processing overhead that the target hardware must be able to support. Simulink provides a Raspberry Pi support package that allows the software to automatically generate, compile, build, and run an executable on a Raspberry Pi remotely.

B. DeepNNCar: Testbed for autonomous driving

DeepNNCar is an RC car testbed used for designing and testing driving controllers. Before discussing the capabilities of DeepNNCar's software control system, it is useful to know how the autonomous robotic vehicle was constructed. DeepNNCar is built upon the chassis of Traxxas Slash 2WD

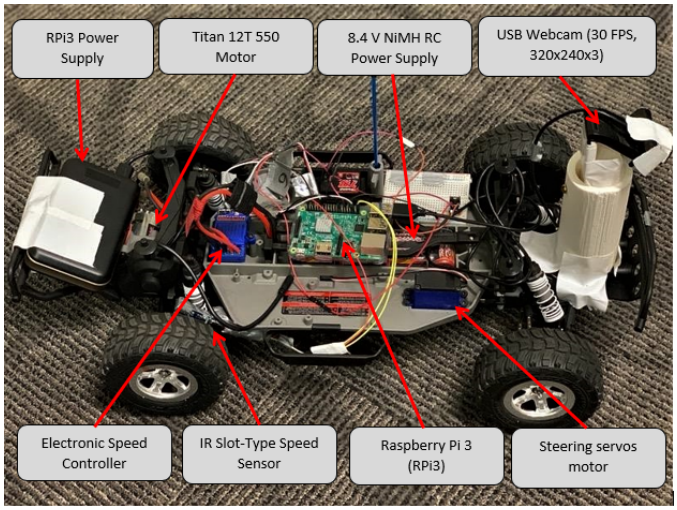

Fig. 3: DeepNNCar with different onboard components

1/10 Scale RC car. The RC car has two on-board motors: a steering servos and a Titan 12T 550 motor. An Electronic Speed Controller (ESC) allows voltage isolation and high current handling to separate the Titan 12T 550 motor voltage requirements from the rest of the system's requirements (5V) including that of the steering servos. The TQ Top Qualifier 2.4 GHz receiver is a PWM controller that translates the radio signals into digital steering and acceleration controls.

1) Sensor: A slot-type IR optocoupler is attached near the rear wheel chassis and connected to a GPIO pin of the Raspberry Pi 3B. After each revolution of the tire, the slot sensor is occluded which generates an interrupt read as a value of 1 in Simulink. The speed of DeepNNCar can then be calculated with the following equation:

$$speed_t = \lambda_t \times \pi \times d \tag{2}$$

where λ is the frequency of interrupt generation and d is the diameter of the tire. There are two constraints to this calculation. First, the system must timeout and detect when the car is not moving. We denote this timeout as T_{zd} or the zero-detection time. Secondly, the interrupt has a bounce time which we denote as T_{bt}. Therefore, λ is constrained by:

$$\lambda_t \in [\frac{1}{T_{zd}}, \frac{1}{T_{bt}}] \tag{3}$$

Any calculations regarding speed that fall outside of this range can be disregarded. From this information, the speed (m/s) of DeepNNCar is calculated.

2) RC Car Rewiring: The original wiring of the car comes with the two motors being controlled by the RF controller; however, we had to modify the wiring (see Figure 4) in order to control the servos and DC motors using Raspberry Pi 3B. The biggest takeaway is that the integrity of the architecture remained largely unchanged except for the addition of an IR slot sensor and the replacement of the TQ Top Qualifier 2.4

GHz with the Raspberry Pi 3B.

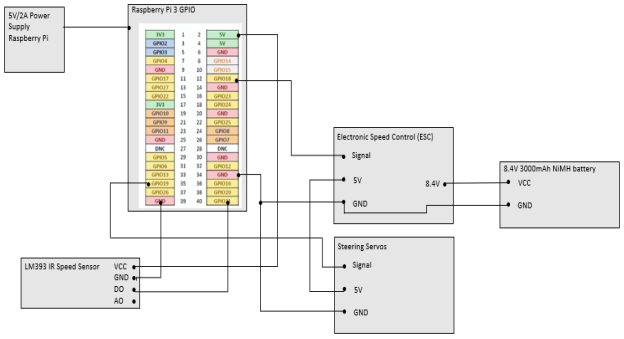

Fig. 4: The circuit schematic of DeepNNCar shows the rewiring we did in order to control the servos and the DC motor for steering and speed control

3) Controller and PWM Generation: We use the Raspberry Pi 3B to control the car. Raspberry Pi 3B has a Broadcom CBM2837 SoC with a quad-core ARM Cortex-A53 processor which we use to execute our programs. Each core has 32K I/D caches and a shared 512KB L2 cache. The pigpio python library is used to generate 100 Hz PWM pulses using the direct memory access (DMA) capabilities of the Raspberry Pi 3B. This allows for precise and responsive control over the two motors and is superior to the alternative approach of using software interrupts to generate a digital pulse. For the servos motor, a duty cycle of [10,20] maps directly to a a steering of -30 and 30 degrees. For the Titan 12T 550 motor, we operate within the range of [15,15.8] which corresponds to approximately [0,1] m/s when the car is subject to the experimental load and indoor testing conditions.

Pulse Width Modulation: Pulse width modulation (PWM) is briefly discussed because of its importance to robotics control and because the duty cycles controlling the acceleration of DeepNNCar. This technique can control a variety of actions, including the color of LEDs on a display or in our case the rotation of a motor shaft.

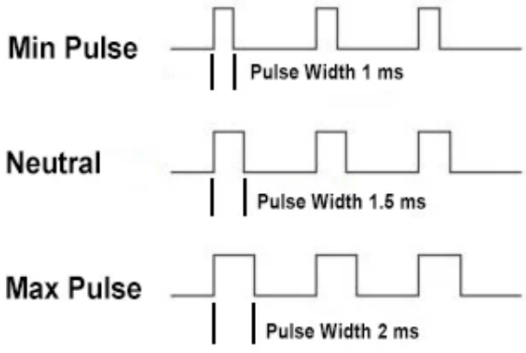

Fig. 5: From top to bottom, the duty cycles would be 10%, 15%, and 20% for a digital square signal of period 10 ms. For DeepNNCar, these correspond to left, straight, and right and reverse, neutral, and forward respectively

The Raspberry Pi 3B is used to generate digital PWM pulses that control the power received by the two DC motors. The duty cycle of a PWM signal is defined as the proportion that the square wave signal is a logical 1 to the signal's period (Figure 5). As the duty cycle of the signal increases, the power received by the motor increases. Beyond a given threshold, the direction of the rotation can also be controlled, allowing the car to go forward or reverse. However, for DeepNNCar's linear movement we are only concerned with forward acceleration. The specifics of the software used to configure and control the hardware of DeepNNCar is discussed in detail in the following sections.

III. Modeling

A. DC Motor State-Space

DC motors may be described by differential equations that relate current to angular velocity. The torque of a DC motor (assuming a constant magnetic field) is proportional to the armature current as illustrated by equation (4). The back electromagnetic force (EMF) is likewise proportional to the angular velocity as illustrated by equation (5). (6) and (7) describe the sum of all forces and Kirchhoff's voltage law surrounding the DC motor. Substituting (4) into (6) and (5) into (7) we obtain differential equations describing our DC motor in terms of current and voltage given by (8) and (9). From (8) and (9) we can derive the state-space model given by (10) and (11). [2]

$$[T = K_t * i] \tag{4}$$

$$[e = K_e * \dot{\theta}] \tag{5}$$

$$[J\ddot{\theta} + Ri = T] \tag{6}$$

$$[L\frac{di}{dt} + Ri = V - e] \tag{7}$$

$$[J\ddot{\theta} + Ri = Ki] \tag{8}$$

$$[L\frac{di}{dt} + Ri = V - K\dot{\theta}] \tag{9}$$

$$\begin{bmatrix} \ddot{\theta} \\ \mathrm{i} \end{bmatrix} = \begin{bmatrix} \frac{-b}{J} & \frac{K}{J} \\ \frac{-K}{L} & \frac{-R}{L} \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} V \tag{10}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta} \\ 1 \end{bmatrix} \tag{11}$$

B. Estimated Parameters

As illustrated by the state-space equations, the modeling parameters of the state-state model of the DC motor include the moment of inertia, damping ratio, back EMF, resistance, and inductance. Calculations performed can validate the values presented in a sample case [2]. The results reveal that the sample case is very close to the actual motor parameters of the Brushless TRAXXAS Titan 550 12T motor. The moment of inertia is estimated using Equation (12), where m was the mass of the DeepNNCar($3.2kg$), and r was the distance from the center of the motor to the center of the wheel($11.4cm$).

$$I = \frac{1}{2}mr^2 \tag{12}$$

The calculated inertia value of the car is 0.0208 $kg.m^2$. The back EMF is estimated by converting the rpm per volt value (Kv constant) to the unit of volts/rad/sec. The Kv constant is obtained from the TRAXXAS community forum which is run and supported by RC car enthusiasts [3]. The suggested replacement motor for the Titan 550 12T is a 3000Kv motor. The estimated back EMF is calculated to be 0.01 volts/rad/sec. The max torque of the motor obtained from the forum is $0.072N.m$. The damping value of the motor is assumed to be very small as supported by testing data with similar motor parameters [4]. Unfortunately, we cannot obtain an approximate value for the motor resistance and inductance from research. Nonetheless, the inertia, back EMF, torque and damping parameters estimated above are in the same magnitudes as the sample DC motor modeling case. The assumption is that the resistance and inductance coincide with the sample case.

IV. Design

This section covers discussion on modeling of the PID controller in simulation and hardware testbed.

A. Simulink Simulation Implementation

We utilize Simulink to design a closed-loop speed control system. The system consists of a reasonable reference speed of 0.2 m/s. The sampled response speed is subtracted from this reference speed to produce an error signal that is sent into the a discrete PID controller (Proportional P=2.19, Integral I=2.181, Derivative D=0.494, Filter Coefficient N=160.6). This control produces a duty cycle signal that is sent into a PWM generation block. The PWM generation sends a PWM signal through an H-Bridge amplification and into the DC motor. The DC motor plant is described by state-space equations which take in the amplified PWM signal along with a static and kinetic friction torque value and produces a velocity response. The DC motor has the following parameters: (Damping ratio:0.01, inertia:0.01, Back EMF: 0.01, Resistance:1, inductance:0.5). We convert single-ended analog signal to its digital representation in decimal value by using an A-D Converter. The implementation is shown in Figure 6, with adjustments to [5].

B. Simulink support package for Raspberry Pi

The Simulink Support Package for Raspberry Pi provides a library of modeling blocks for configuration and access to Raspberry Pi peripherals and a communication interface [6] [7]. This project utilizes the feature that writes to and reads from the Raspberry Pi digital I/O pins. After the controller design is completed, the model is compiled to an executable that runs on the Raspberry Pi 3B. Tuning is performed in Simulink while the algorithm runs on the hardware. To establish the communication between the Pi and Simulink, the support package installer automatically downloads and installs the Raspberry Pi support software onto the existing operating system of the Raspberry Pi micro SD card. The network is configured using the DeepNNCar's IP address. Despite Simulink's efficiency and simplicity in the automatic code generation process, the debugging process for the modeling blocks was quite tedious. Due to the inaccessibility of the source code in the modeling blocks, it was difficult to identify the intermediary states of the parameters and outputs. Moreover, the error messages provided were not specific enough to identify the errors. An oscilloscope was used in the process of debugging to check the outputs of the I/O pins on the Raspberry Pi 3B since the generated code was not available for inspection.

C. Simulink Hardware Implementation

The Simulink hardware implementation is shown in Figure 7. For the hardware implementation we started with the reference speed of 0.2 m/s. The sampled response speed calculated by a MATALB function is subtracted from the reference speed to produce the error signal to be fed to the discrete PID controller(Proportional P=0.013, Integral I=0.0001, Derivative D=0.0002, Filter Coefficient N=160.6). These values are quite different than what we used for the simulation. This illustrates the challenges with modeling and simulating hybrid systems, as values may differ on the hardware implementation. Two potential reasons for the discrepancy are that the estimated parameters are off slightly or the configuration of the system motor is more complex than our modeled version. The MATALB function takes as input the measured sensor impulse, the diameter of the wheel of the car, the previous sensor impulse time and the current sensor impulse time to calculate the speed using the circumference of the wheel and rotations per second. The PID controller produces a duty cycle signal which is sent into a non-linear saturation block to prevent invalid duty cycle values from being provided. The signal is then sent to Raspberry Pi 3B GPIO 18. The sensor impulses are then measured from GPIO 21 of the Raspberry Pi 3B, which acts as an input to the MATLAB function to be fed back in the closed loop controller to produce the error signal.

V. Analysis

A. Simulation Results

The results line up with what we intuitively expect. We have a steep rise in velocity, small overshoot, and then we level out to the steady-state velocity in about 3 seconds. The initial steep drop, into negative velocity is due to the step responses that we have set for static and kinetic friction. On board the target hardware, the velocity does not go negative, rather, it does not increase until the simulated response crosses the zero axis.

B. Hardware Results

The hardware tests produce spurious anomalies when it comes to sensor readings. A simple constant speed model, without feedback, can be successfully deployed onto the hardware causing continuous acceleration; however, errors arise when using the model with a closed-loop discrete PID

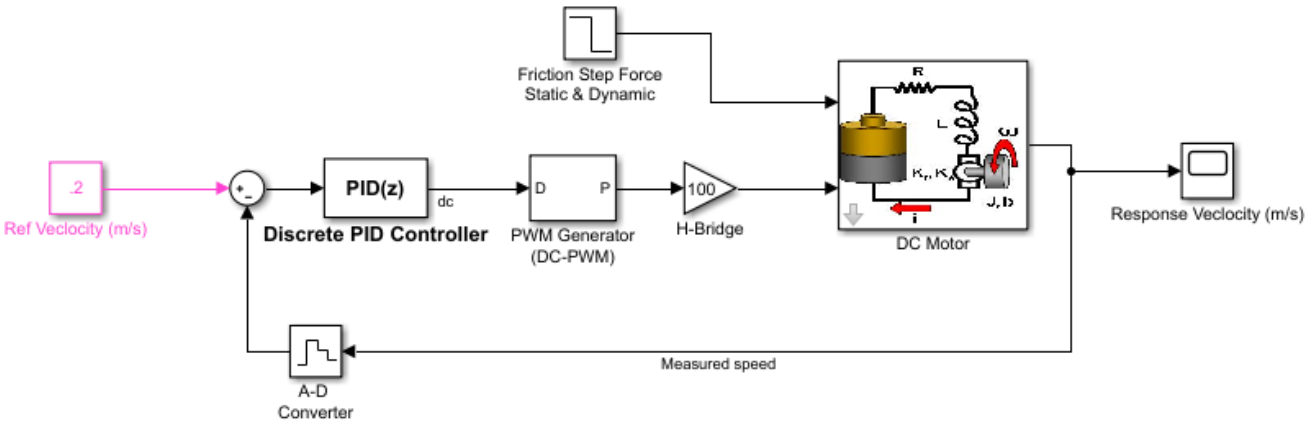

Fig. 6: Simulink Model: Simulation Block Diagram

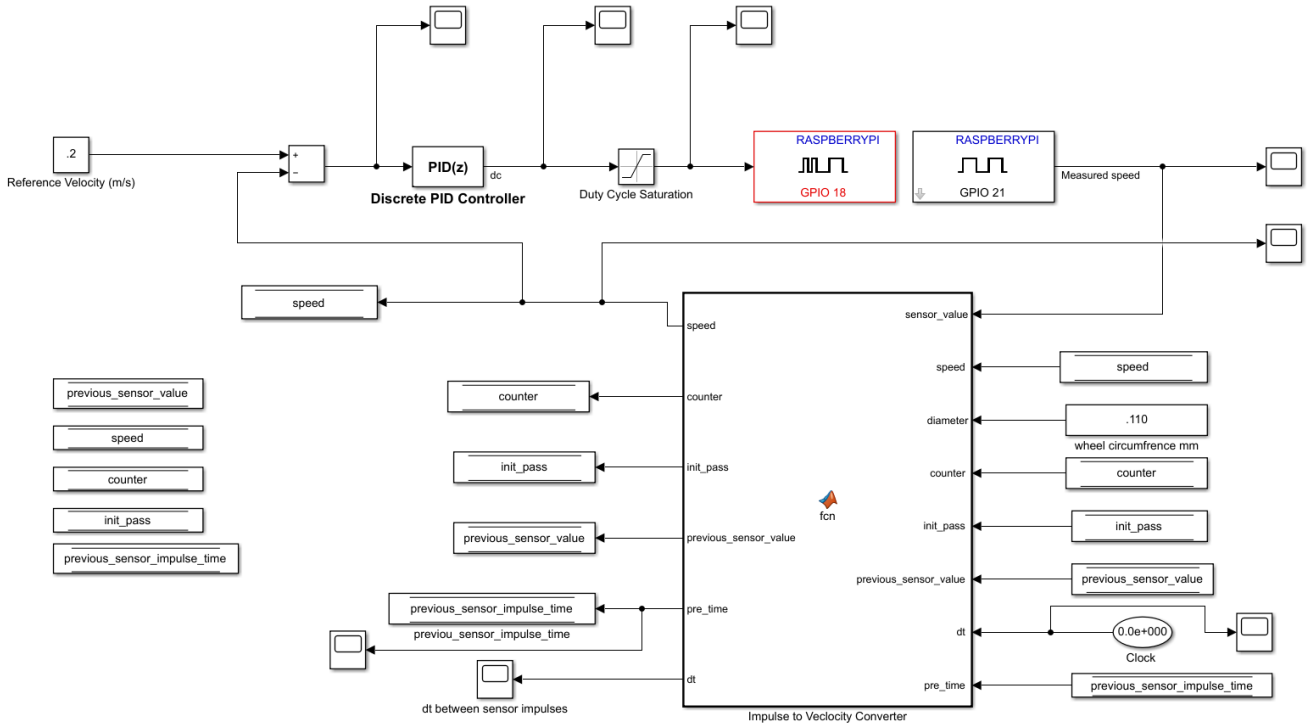

Fig. 7: Simulink Model: Target Hardware Block Diagram

controller and a non-linear saturation block. The crux of the matter comes down to the processing power of the Raspberry Pi Model 3B. The processor is not fast enough to handle the overhead of Simulink's generated executable. As a result, the execution cycle of the Simulink model slows down below real-time, resulting in delayed sensor sample time. This result is sporadic IR sensor impulse readings that vanish over time as the computation load increases and the execution time slows further. This results in limited and inaccurate feedback causing the closed-loop to function improperly. This anomaly, illustrates one of the challenges in Hybrid Systems and the importance of knowing worst case real-time execution times (WCET). It also illustrates the challenges of appropriately modeling such a hybrid systems. Though our state-space simulation provides acceptable results, the simulation is not enough to predict the scheduling anomalies that arise when the generated code is executed on the target hardware.

CPU Over Utilization: In order to capture the reading from the IR slot-type speed sensor, the sampling time of the Simulink executable must be small enough to capture sampling interval from the sensor. In general, there are two methodologies that can be employed to ensure consistent sample readings. The first requires Simulink to have a executable sampling time that is less than the on board sensor sampling time. This ensures that Simulink does not miss a sampled value from the sensor readings. In the DeepNNCar implementation,

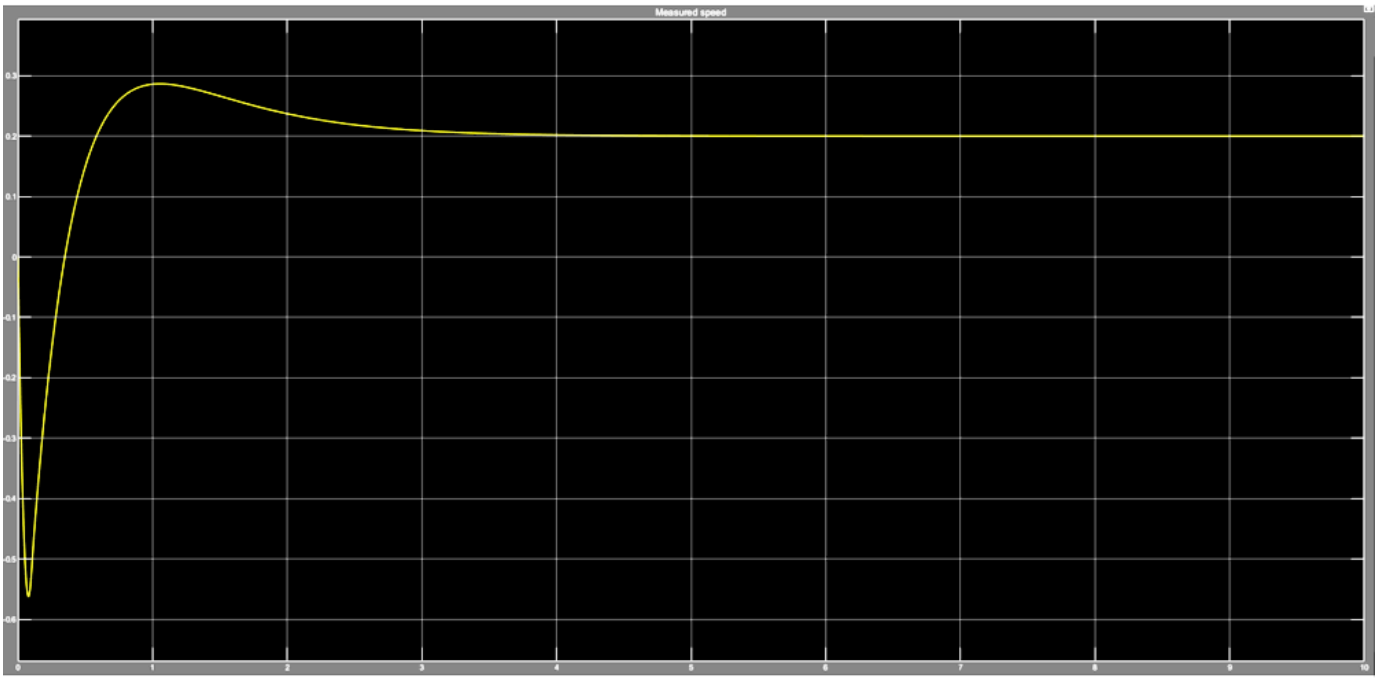

Fig. 8: Simulation Results

the sensor reads at a period of approximately 10ms. In order to guarantee that Simulink never misses a reading, the best solution is to set Simulink's sampling rate 1ms. Unfortunately, this requires additional processing power intensifying the issue of CPU over utilization, resulting in significantly degraded execution cycles times (far less than real-time) and more missed sensor readings. The second solution is to take a more moderate approach and decrease Simulink's sampling rate to a reasonable rate for the velocity ranges of the vehicle (e.g. 0.1s). With a 0.1s real-time sampling interval Simulink would still read the sensor readings as discrete impulses. However, even a sample time of 0.1s proved to be too fast for the Raspberry Pi 3B processor to handle. Just as with a sampling time of 1ms, due to the limitation in the Raspberry Pi 3B CPU performance, Simulink consumes all CPU resources which results in task overruns [8] and a less than real-time execution time. A task overrun refers to when a task is scheduled to execute before a previous instance of the same task has completed. This quickly creates a bottleneck in the CPU and delays all executions. The effect is that the execution time is far less than real-time. At start-up, the Simulink model is able to calculate the first few sensor readings from the car. However, as CPU overloads, the effective sample time becomes longer than specified, and sample points get missed. This is illustrated by figure 9 in which a table presents the difference between real-time and on board execution times under different code generation conditions. As illustrated the execution time is always significantly less than the real-time; this confirms execution timing delays are happening. The large demand for CPU resource is likely due to the overhead code in Simulink which we are unable to reduce. To conclude, the Raspberry Pi 3B is determined to have insufficient processing power to perform Simulink calculations with short sampling times.

Real-Time At Completion	Execution Time At Completion	Discrete PID Block Present	Non-Linear Saturation Block Present	Execution Time Step	Sample Time
4.92 seconds	3.012 seconds	No	No	auto	0.1 seconds
11.59 seconds	2.231 seconds	Yes	Yes	auto	0.1 seconds
18.87 seconds	11.883 seconds	Yes	Yes	auto	0.0001 seconds

Fig. 9: Real-Time Vs. Execution Time

VI. Conclusion

A. Summary

Model-Based Design has been used for decades in designing controllers for hybrid systems. In this work we have presented an approach to model-based design of a speed controller for a hybrid system. We have designed and tuned a PID controller for a DC motor in Simulink. We designed the DC motor state-space using differential equations relating current, voltage, and angular velocity. We estimated various parameters like inductance, inertia, damping ration, back EMF, and resistance to model the state-space of the motor. Using this setup we tuned the parameters of a discrete PID controller in a simulated closed-loop hybrid system to control the speed of the motor. This tuned model was then converted into a hardware model and deployed onto our hardware testbed; which is an RC car that uses Raspberry Pi 3B as its on board computing unit. We used Simulink's Raspberry Pi support library to deploy the hardware model on to the hardware testbed. We performed various experimental test runs varying the sampling times. We successfully deployed a simple constant speed model without

feedback onto the hardware; however, we were not successful in getting the closed loop system to work on the hardware due to processing limitations. We concluded from our experiments, that Simulink's generated code results in high processing overhead which completely utilizes the CPU resources. This degrades the model execution time to slower than real-time resulting in spurious errors due to the executed sampling rates falling below acceptable real-time thresholds. Through our work we have learned about the gap between the simulated models of hybrid systems and the actual implementations of those models on the target hardware. This was conferred from the differences between our simulated modeling results, and the results of real tests on the hybrid system testbed.

B. Future Work

As a future work, we would like to perform the closed loop PID experiment on a higher capacity computing unit like NVIDIA Jetson, to see if it could handle the processing overhead of Simulink, and perform at lower sampling rates.

REFERENCES

[1] Mathworks, "Pid controller design for a dc motor," 2019. [Online]. Available: https://en.wikipedia.org/wiki/PID_controller

[2] MATLAB and SIMULINK, "Dc motor speed: System modeling." [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?example=MotorSpeed&section=SystemModeling

[3] Hardcoretam, "Thread: Brushless equivalent of the trx titan 550 12t," Sept.2008, TRAXXAS COMMUNITY FORUM. [Online]. Available: https://forums.traxxas.com/showthread.php?474890-Brushless-equivalent-of-the-Trx-Titan-550-12T

[4] RadianDynamics, "Mechanical model of dc motor - damping coefficient problem," 2017, electrical Engineering Stack Exchange. [Online]. Available: https://electronics.stackexchange.com/questions/284446/mechanical-model-of-dc-motor-damping-coefficient-problem

[5] "Pid controller," 2019. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/26275-pid-controller-design-for-a-dc-motor

[6] MathWorks, "Raspberry pi support from simulink," 2019. [Online]. Available: https://www.mathworks.com/hardware-support/raspberry-pi-simulink.html

[7] ——, "Build raspberry pi projects using high-level programming and block diagrams," 2019. [Online]. Available: https://www.mathworks.com/discovery/raspberry-pi-programming-matlab-simulink.html

[8] Mathworks, "Detect and fix task overruns on raspberry pi hardware," 2019. [Online]. Available: https://www.mathworks.com/help/supportpkg/raspberrypi/ug/detect-and-fix-overruns-on-raspberry_pi-hardware.html