Processor-in-Loop Control System Design Using a Non-Real-Time Electro-Magnetic Transient Simulator

by

Gregory Chongva

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg

Abstract

This thesis investigates using processor-in-loop techniques with non-real-time electro-magnetic transient simulation software for designing microcontroller-based systems. The behaviour of a microcontroller is included in the simulation by directly integrating the target microcontroller into an EMTP co-simulation. Additionally, to assist the design process, the optimization functionality of the EMTP program is extended to the microcontroller algorithm. Since non-realtime simulation does not require specialized test hardware to accurately simulate systems, it is both cheaper and able to be used earlier in the controller design process then hardware-in-loop real-time simulation. A component is created in the PSCAD / EMTDC program to integrate a generic controller running an arbitrary periodic algorithm into an EMTP simulation. The component operation is verified by creating a co-simulation of a three-phase induction motor V / f. speed control. The co-simulation results match the behaviour of the resulting system under a fairly broad range of operating conditions, highlighting the applicability of the technique.

Acknowledgements

The author would like to acknowledge the following individuals for their support and assistance:

- Mr. Michael Ward, for a small but immensely useful bit of programming advice.
- Mr. Erwin Dirks, for purchasing and helping bounce some ideas around.
- fellow student Maryam Salimi, for her simulation to start the ball rolling.
- fellow lab denizen Jessee Doerksen, for some excellent motor parameters.
- fellow greybeard and lab denizen Garry Bistyak, for some coaching regarding the fine art of induction motor control, and commiseration about the cruelty of old age.
- my supervisor Dr. Filizadeh., for his support and mostly successful attempts at understanding my Computerese point of view.
- my parents Frank and Clara Chongva, for looking after "The Boy" while I was up late at the University, among so many other things.
- Muoi Tran, for her patience through this last busy, busy year, and of course, for "The Boy".
- Anton Jun "The Boy" Chongva, for giving me joy, very practical lessons in determination, and hope for the future.

Contents

List of Figures

List of Tables

1 Introduction

"In theory, there is no difference between theory and practice. But, in practice, there is."

Jan L. A. van de Snepscheut

"Power is nothing without control."

Pirelli tire slogan.

This thesis deals with Software-in-Loop (SIL) and Processor-In-Loop (PIL) co-simulations for designing, physically realizing, and verifying a control system using a microcontroller and non-real-time simulation software. The specific system investigated is a constant voltage vs. frequency (V vs. f) speed control system for an induction motor. The software simulator used is PSCAD / EMTDC, an electromagnetic transient (EMT) simulation program.

This section of the thesis will attempt to give a brief overview of control systems, control system design and verification methodologies, and control system applications in the area of innovative power electronic systems.

1.1 Innovations in Power Electronic Systems

The need for innovative power electronic systems will only increase in the immediate future [1]. Some major growth areas requiring the use of power electronics are electric vehicle applications, power transmission control and protection systems, and alternate generation methods.

Various designs of electric motor are being investigated and implemented as an alternative to the internal combustion engine as a means of propulsion for automobiles [2]. Switching to renewable energy will reduce greenhouse gases, prevent degradation of the environment, and extend the life of existing non-renewable petroleum inventory. Vehicular applications require the electricity being applied to the motor be appropriately controlled, otherwise the resulting movement of the automobile could be unpleasant, wasteful, or even unsafe [3].

Increasing the efficiency of power transmission translates into more energy available for consumption. Two relatively new methods for increasing the efficiency of power transfer are the Flexible Alternating Current Transmission System, or FACTS, and the High Voltage Direct Current, or HVDC systems [1]. FACTS is a system which modifies the electrical characteristics of an ac transmission line, which HVDC allows power to be translated from the source ac to dc for transmission, and back to ac for distribution. Both of these systems require careful control of electrical energy to maximize the efficienty and stability of the power transmission.

One broad method of dealing with increased energy demand is integrating renewable energy sources into the conventional power grid. This involves the creation and harnessing of electrical energy from non-conventional sources such as solar power and wind energy. Generating and integrating power from these sources into the existing power grid requires careful control so as to maximize the power injection without disrupting the existing power flow.

1.2 Control Systems for Power Electronic Devices

Control systems are automated systems for maintaining control over a process. They consist, in the simplest form, of devices for measuring a process, called sensors, devices which can affect the process in some way, referred to as actuators, and devices which perform calculations of how to position the actuators in response to the given sensor data, or controllers.

All four examples of innovative electrical systems given in the previous section require the precise control of electrical power to be successful. This underscores the need for tools and methods to successfully create a control system for power electronic devices.

In the case of power electronic systems, such as HVDC, FACTS, or electric motor controllers, the sensor can be a voltage or current measuring device connected to an analog to digital converter, or in the case of a distributed system, a communication link. The actuator can be a power electronic switch, such as a thyristor, and its associated circuit and components. Due to their low cost, capabilities, and flexibility, digital microcontrollers are generally selected for the control equipment.

1.3 Simulation

Simulation is a process which creates a model of a system in order to more easily or safely investigate its properties in place of investigating the system itself. Due to the flexibility and processing capacity of modern digital computers, computer-based simulation is used to investigate a wide array of phenomena.

Generally, computer simulations represent the various components of a system as seperate software models. These models can vary in accuracy depending on the models used and the simulation requirements. The simulation software describes the behaviour of an aggregate system by calculating the effects of the interactions between its components.

Even though simulations are less accurate then live testing, simulation testing can be valuable to the design process. Simulation can show whether a system design is generally applicable to solving a problem before the system is even built. Simulation testing of an expensive, complex, and dynamic system also has less risk associated with it than live testing, since no actual components are used.

Simulations can be divided into two major types: non-real-time and real-time. In real-time simulations, simulation time passes at the same rate as normal time, while in non-real-time simulations, the simulated time passes at a different rate, depending on the calculations required to simulate the system as well as the capabilities of the simulation computer.

Depending on the accuracy or confidence level required, parts of the actual system being investigated can be integrated into the simulation. In such a system, a real-time simulator is used and selected external pieces of hardware are then interfaced with the simulator to form what is commonly known as a hardware-in-loop (HIL) simulation.

1.3.1 Co-simulation

Co-simulation is defined as the use of multiple simulation platforms in a single simulation of the same system. A simulation platform can consist of a software package or separate hardware representing some aspect of the simulation. In order to generate accurate simulation data, the platforms composing the entire simulation must have their simulation time synchronized in some manner, and also share data on the state of the simulated components.

Co-simulations can take advantage of pre-existing software to reduce the effort needed to simulate a system. An example of an electrical co-simulation with mathematics software is PSCAD electrical simulator and MatLab / Simulink mathematics package [5]. The PSIM electrical simulation packages and ModelSim circuit simulator have also been integrated for the purposes of co-simulation of electrical systems [6].

A microcontroller, when integrated into a simulation, can also be viewed as a simulation platform. In this case, integrating the controller provides the simulation with accurate controller computation and responses.

1.4 Optimization

Optimization is a process for searching for the minimum value of a function with an arbitrary number of input parameters [4]. Generally, an initial set of parameters is selected, and the function is evaluated at these parameter sets. Based on the function value returned for each set, and depending on the exact algorithm used, additional parameter sets are selected and have their function value calculated. By repeating this process, the optimization algorithm drives the search towards more desirable sets of parameters.

Optimization can be used to automate a portion of control system design. By identifying attributes in the control system as optimization parameters, and creating an objective function which calculates the quality of the response of a control system, the simulation can be run repeatedly by the optimization algorithm in an attempt to find optimal control parameters.

1.4.1 PSCAD / EMTDC Simulation Software

PSCAD / EMTDC is one of a number of software packages which simulate the behaviour of electrical systems undergoing transient behaviour [7]. PSCAD stands for Power Systems Computer-Aided Design, and is a graphical program allowing a user to create a model of an electrical system composed of various blocks representing portions of the electrical system equipment. EMTDC stands for Electro Magnetic Transient for DC, and simulates the behaviour of the resulting system. A wide variety of electrical phenomena can be investigated with PSCAD / EMTDC, from lightning strikes on transmission lines to all-terrain vehicle drive systems [8] [9].

PSCAD / EMTDC is a non-real-time simulator, with no definite correlation between simulation time and actual time. It has been interfaced with different mathematical software tools to expand its functionality and allow different systems to interact with PSCAD simulations [10]. C language routines are easily integrated into PSCAD simulations.

PSCAD / EMTDC includes analog control blocks for designing and simulating control systems as a part of a larger electrical system. However, no automated way exists in the package for converting a model of the control system comprised of these blocks into an arbitrary digital micro-controller, nor does PSCAD / EMTDC contain a method for integrating a controller directly into its simulations.

To allow computer-aided design of electrical systems, PSCAD / EMTDC has an optimization block incorporated in it which performs the Simplex, Genetic Algorithm, and Hooke-Jeeves optimization algorithms [11].

1.5 Motivation

Reducing the effort and expense required to create and verify new designs makes implementing these innovations more financially attractive, and can result in a better final system. Simulation is an activity which reduces the effort and expense of verifying new designs.

A non-real-time simulation of a control system design, although requiring no specialized simulation hardware, is not as accurate as the same design implemented on actual hardware. The details of the controller operation are generally not reflected as accurately in non-real-time simulation due to factors such as un-modeled components or conversion procedures.

In contrast, real-time hardware-in-loop simulations of control system designs excersise the actual microcontroller hardware and software, but require real-time hardware for proper integration of the control system during testing. Depending on the performance required by the simulation, this hardware could be expensive.

The main motivation of this work is to assist the physical realization of power electronic systems featuring control systems from non-real-time simulation to actual hardware. Microcontroller behaviour in the form of a simulation model is not easily translatable to actual hardware. If the form of the controller model used in the simulation could be translated into real hardware more accurately then existing models, higher confidence could be placed on simulation testing, reducing the amount of expensive hardware simulation required for the same level of confidence in the design.

1.6 Objectives

The primary objective of this thesis is to make non-real-time microcontroller simulation models more realistic by enabling software- and processor-in-loop simulation for modelling microcontroller behaviour. A component for an electromagnetic transient simulator (the PSCAD / EMTDC software in particular) is created to run accurate simulations of an arbitrary microcontroller running an arbitrary periodic task. To allow different microcontrollers to be accurately simulated, the microcontroller model is generic in nature.

A secondary objective of this work is to allow optimization algorithms of the simulator to work on software- and processor-in-loop simulations as for model-in-loop simulations. The microcontroller component is designed to allow the optimization functionality of PSCAD / EMTDC to modify the behaviour of the microcontroller.

To verify the effectiveness of the software- and processor-in-loop simulations, the component is used to design and physically realize a constant voltage vs. frequency, or V vs. f, control system for an ac induction motor using the component. A processor-in-loop simulation allowing optimization of the controller function is created using the microcontroller component. The system resulting from the optimized simulation is used to control a real induction motor. The migration between simulation and reality occurs using a minimum of specialized test hardware, and with a minimum of expense, time, and difference between the simulated and actual controller performance.

2 Simulation in Control System Design

This section will begin by reviewing the common phases of various system design processes. The use of simulation for testing throughout the different phases will be discussed, and a few types of simulation will be described.

2.1 System Design Process

Generally, the system design process starts with requirements gathering. These requirements describe the desired behaviour of the resulting system. Depending on the exact design process followed, requirements can be subject to change at any time during the development.

Once the requirements of the system substantially exist in some form, the process moves to the construction phase. This phase sees the various pieces of the design created. This could involve writing software, selecting and assembling hardware, including the sensors, actuators, and controller, and integration of digital circuits using a field programmable gate array, or FPGA. As a part of this phase, unit tests validating the behaviour of the various pieces are conducted. Due to the drastically different disciplines required, entirely seperate design streams may exist for the various components of the final system [12].

After the construction phase, the various components of the system are integrated until the entire system is assembled. Integration tests ensuring proper operation of the components when operating together are performed during this phase.

Finally, after the complete system has been assembled, the verification phase consists of system-level testing to ensure the system is meeting the stated requirements. This testing could be a live test, actually operating the system in the environment for which it was designed.

Some system design processes allow iterations of all four phases, overlapping of phases, or returning to previous phases due to unsatisfactory test results, or user requirements changes.

2.1.1 Simulation During the System Design Process

Design of a control system and tuning of its parameters may take different forms depending on the complexity of the situation at hand. In the most primitive case, a mathematical model of the system may be available, to which classical control system design techniques may be applied.

In most practical cases, however, the luxury of a reasonably comprehensive mathematical model may not be present. In such cases, computer simulation tools become a strong alternative. Instead of a precise set of equations, a computer routine is written to generate results approximating the operation of the modeled system. When placed in a program and interacting with other routines representing other systems, this collection of routines creates simulated data reflecting the operation of the aggregate system.

In addition to the unit and integration testing outlined above, testing via simulation can be incorporated into all phases of the system design process [12]. From initial requirements through to final testing, the design may exist in a number of different forms. Simulation appropriate to the form of the design at the current phase of development can be used for system verification [13].

Some work has been done towards creating flexible simulation systems [14] [15]. These systems are capable of reusing the same system component models for performing different types of simulation. Because of this, less effort is required to maintain a comprehensive set of simulation models usable throughout the design process.

2.2 Overview of Simulation Types

Figure 2.1 is an overview of the various types of simulation testing possible during the system design process. The diagram depicts a generic system and the environment in which it is to operate. From left to right, the system contains a control model, implemented as a piece of software running on a microcontroller. The microcontroller uses its input and output peripherals to read sensors and control actuators. The sensors and actuators control the physical equipment, which also reacts to its environment.

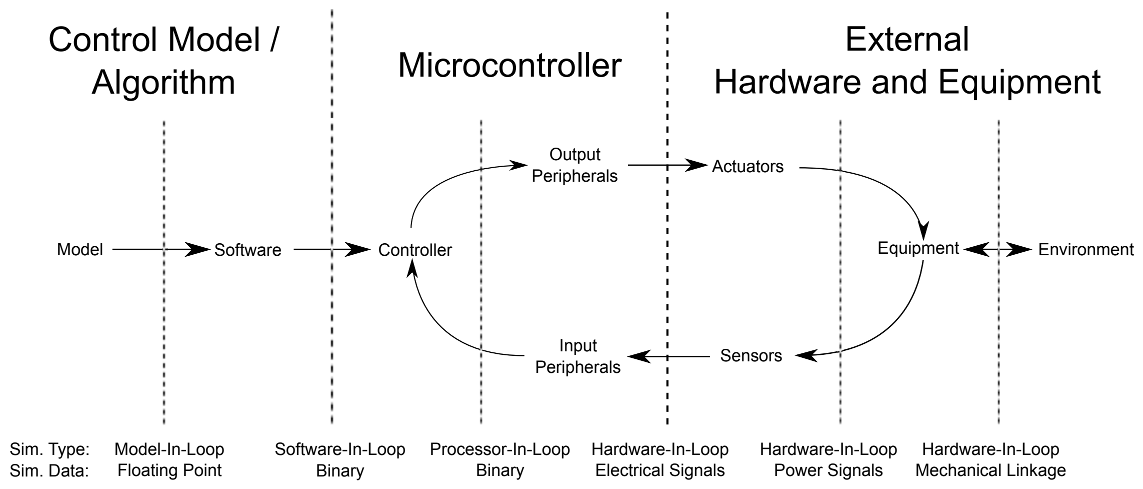


Figure 2.1: System Composition and Simulation Types

The dashed lines in the figure represent divisions between authentic and simulated portions of the design for the various types of simulation. The components to the left of each dashed line are authentic for that type of simulation, or are in a form which could be used in the final system design. The components to the right of each dashed line are not authentic, and are simulated in some way.

For each simulation type, the "Sim. Type" row holds the name of the type, and the "Sim. Data" row describes the form of the signals transfered between the real and simulated parts of the system. Floating point data refers to numbers extracted directly from the simulation, and binary data are numbers formatted as they would be received by the controller.

Initially, a design might only be simulated with model-in-loop techniques, designated by the dashed line at the extreme left of the diagram. When the controller model is translated into software, a software-in-loop simulation can be used for verification. Processor-in-loop simulations integrate the actual microcontroller hardware, and can be used to verify whether a specific controller has adequate performance. As the design progresses, the system can be simulated using more of the actual hardware of the design using the various hardware-in-loop simulations.

2.3 Non-real-time Simulation

As was mentioned in the introduction, a non-real-time simulation is one where the simulated time passes at a different rate than actual time. These types of simulation require less performance from the simulation platform than real-time simulations, as there is no practical limit on the amount of time the processing for a single timestep can take.

The following are different types of non-real-time simulations.

2.3.1 Model-In-Loop (MIL)

Model-in-the-loop simulation involves using a model to represent a system being designed. Generally, the ideal behaviour of the system is described, while the exact implementation details are completely ignored. This sort of representation would most likely occur during the initial requirements phase of the design process.

A model can be expressed in a number of different forms. Some examples are equations, a control block diagram, a modeling-specific language such as MathCad, or a general purpose language such as FORTRAN [16].

The model of the design can vary in sophistication and accuracy. For instance, the model might accurately represent the discretization effects in the system if the general performance of the target system is known.

The greatest benefit of this type of simulation is that it can be performed early in the design process, before software or hardware is available [17]. Also, since no restrictions are made on the form of the model, any modeling tool adequately describing the model can be used.

Disadvantages of using this type of simulation is that, depending on the software tools available, translating the model into real hardware can prove difficult [18]. Other, more accurate simulation or live testing would be required to increase confidence in the design [19] [20].

As the requirements portion of the design process continues, the design could be translated from a pure model into software routines for performing the model behaviour. The translation into prototype program code can be a manual or automatic process, depending on the software translation tools available [21].

2.3.2 Software-In-Loop (SIL)

Software-in-loop simulation involves incorporating the software routines of the actual controller software into the simulation. This representation of the controller behaviour is validated by simulating the action of the software routines as a part of a larger simulation.

The software-in-loop simulation can be more accurate to the final system behaviour than the model-in-loop, without any extra test hardware, and can be used to boost the confidence in a design [22]. As with model-in-loop simulation, testing of the controller code can still commence without controller hardware available.

The software-in-loop can be hosted in the simulator as if it was on an actual controller. The code used to integrate the software to the rest of the simulation can incorporate platform-specific behaviour towards this purpose [23] [24].

Multiple copies of the software can be instantiated as a part of a software-in-loop simulation. These multiple copies could represent multiple instances of a simulated device, such as a number of wind turbines comprising a wind farm, or a formation of spacecraft [25] [26].

And finally, software is not the only code which can be integrated into a simulation. Any portion which can be represented as computer files and simulated, such as HDL code for logic circuits, can be integrated as well [27].

2.3.3 Processor-In-Loop (PIL)

Processor-in-loop simulation is similar to software-in-loop simulation, except that the control software processing is done on the actual hardware, instead of on the simulation platform. This involves integrating the controller hardware platform into the simulation, with the simulation data and calculated results transferred between the target hardware and a host simulation computer through some sort of communications link [28].

Running the software directly on the controller hardware is a more accurate test by then running the same software on the simulation platform. Incorporating the actual hardware and software into the non-real-time simulation allows some insight into how long the code would take to execute, and the exact results the controller would calculate while running. This would give some idea of the required hardware performance for the design.

2.3.4 Emulator-in-loop

A hardware emulator is a piece of software which mimics the behaviour of a processor. They are useful for debugging software for a specific processor, giving the programmer full control over the execution of the software, or for debugging software without having the corresponding hardware available.

If the software for a control system was hosted on an emulator for the controller, the resulting simulation would fall somewhere between software-in-loop and processor-in-loop simulation types mentioned above.

2.4 Real-time Hardware-in-Loop (HIL) Simulation

As the hardware design is refined, and appropriate peripherals for performing the actual sensing and actuating of the system become available, the complete controller behaviour can be further validated by running the controller at full speed while applying simulated signals to it, either directly or through its actual sensors. This is refered to as hardware-in-loop, or HIL, simulation.

For real-time simulation, hardware specific to the simulation is configured to generate stimulus for the controller, and react to the controllers responses. In this manner, the control system is subject to signals identical to those experienced in actual operation.

The capabilities of the test hardware are crucial to the success of this type of simulation. Since the test hardware is masquerading as the system and equipment to the controller, the test hardware must be able to feed appropriate signals to the hardware and respond to the hardware requests as if it were the real equipment under control. Depending on the response rate of the controller, the test hardware might have to be fairly capable, and therefore expensive.

Generally in a simulation, the response of the system is calculated by some software running on a computer. In order to achieve performance sufficient for real-time simulation, part of the system can be simulated in hardware instead. For example, electrical circuits can be scaled down, and the smaller measurements for these component can represent the larger measurements of a real system [29].

Also, FPGAs with power circuit representations on them can be incorporated in a simulation [30]. If the same FPGA also hosts the controller under test, the latency for the communications link between the simulated power circuit and the actual controller becomes negligably small.

A few different types of real-time simulation exist, depending on what portion of the controller and system is included in the simulation [31]. These types are detailed on the right hand side of Figure 2.1, and described below.

2.4.1 Electrical Hardware-in-Loop

Only the controller and its peripherals are included in this type of simulation. Electrical signals are generated by the simulation hardware and connected to the controller. The simulation hardware interprets the electrical signals created by the controller and uses the input data to influence the state of the equipment being simulated.

2.4.2 Power Hardware-in-Loop

The controller, sensors, and actuators are included in this type of simulation. Simulated physical inputs are applied to the actual sensors, which are in turn read by the controller. The actuators in the system are similarly connected to the controller, and the simulation hardware reads their physical outputs, and incorporates the data into the simulation.

2.4.3 Mechanical Hardware-in-Loop

The controller, sensors, actuators, and controlled equipment are included in this type of simulation. Simulated physical inputs are applied to actual equipment to be controlled, and the equipment would be physically manipulated by the test hardware as if it were in use.

2.4.4 Hybrid Hardware-in-Loop

In addition to the above types of simulation, combinations are possible. For example, an arbitrary controller might have a few transducers built into its case, while the majority of the transducers would be connected through a wiring harness. If the built-in transducers are difficult to replace with injected electrical signals, they might be excited via physical signals under the control of the simulation. Sensors connected through a wiring harness would be better candidates for electrical signal injection.

2.5 Optimization of Control System Simulations

The goal of optimizing a simulated control system is finding the most desirable performance of the system. This would involve parameterizing the control algorithm, and running the simulation repeatedly while changing those parameters.

The optimization parameters for a controller design could be coefficients for any calculations in the control system. (e.g. proportional, integral, and derivative coefficients in a PID controller.) The objective function for the simulation could be some measurement of the quality of the process performance. (e.g. the cumulative difference, or error, between a requested speed and the actual speed of a motor)

The process of optimization could theoretically be used on any of the above types of simulations, as long as a measurement of performance could be generated for each run. However, since the optimization process could attempt to test a system with parameters leading to exceedingly bad performance, including large, expensive pieces of equipment in these sorts of simulations would not be advisable.

3 Controller Composition and Processor-in-Loop Simulation

This section will focus on providing detail of processor-in-loop simulation as it applies to power electronic systems. The general structure of common microcontrollers will be described by decomposing it into basic physical components. Since the types of simulation generally don't depend on the physical details of the actual controller being used, the controller components will be discussed generically.

Various types of simulations of microcontroller-based systems will also be reviewed. Comparisons and contrasts will be made between processor-in-loop (PIL), software-in-loop (SIL) and hardware-in-loop (HIL) simulations in a few different areas pertainent to control system design.

3.1 Generalized Controller Composition.

The general idea behind this work towards interfacing an arbitrary microcontroller with an electrical simulation is to divide the controller into its various components, and then simulate the various components as accurately as possible. The most broad division of the controllers functionality is to separate the processing of the controller from the action of its input and output peripherals. Figure 3.2 shows the controller structure overview.

3.1.1 Controller Processing

The controller processing is the data manipulation and calculations which are required for the controller to perform its function. These activities can be divided into two major parts: initialization processing and periodic task processing.

At start of controller operation, the controller initializes itself. The purpose of this is to set the controller to a state consistent with beginning to perform its task. During initialization, the controller peripherals are configured to operate as required, and any controller memory variables are set to values

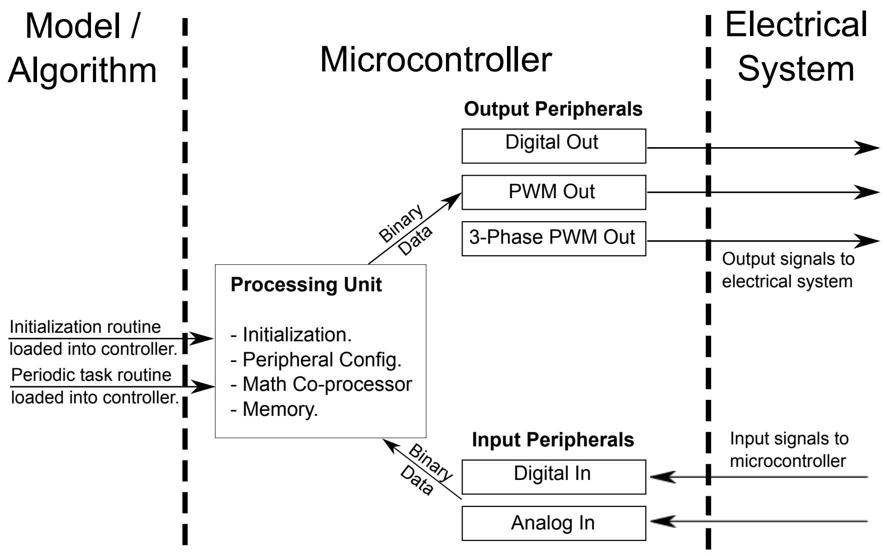


Figure 3.2: Controller Overview

appropriate for commencing operation.

Once the controller is initialized, the processor must calculate the required outputs based on the values of the inputs. This processing is performed periodically as new data becomes available.

Many different possibilities for organizing the operation of a controller exist, with real-time operating systems allowing multiple tasks to be organized and run at different periodic intervals, or triggered by external events. However, a control system can also be implemented by a single task, run at a constant frequency.

3.1.2 Controller Peripherals

Modern controllers can have a wide variety of peripherals specialized to accept or provide many different kinds of electrical signals. Peripherals generally convert between arbitrary electrical signals and a binary forms of the same signals suitable for controller processing. Various types of analog input circuits exist, but the basic operation is for the circuit to convert an analog voltage into a binary number for processing by the microcontroller.

Modern controllers have a wide variety of types of peripherals available, including digital inputs and outputs, analog inputs, PWM outputs, and various types of communication ports. The simplest input peripheral is a digital input, which monitors a single connection and returns either a 1 or a 0 to the processor, depending on the voltage level present on the connection. Similarly, the simplest output peripheral is the digital output, which provides either a high or low voltage depending on the value written to a specific location in the processor.

Analog input ports of microcontrollers also vary in number of channels, resolution of measurement, and can use analog multiplexers to reduce the number of conversion circuits required.

Pulse-width modulation, or PWM, is generally used for analog output peripherals of modern controllers. These peripherals output a pulse length in proportion to the digital number written by the processor. External circuitry is used to convert these pulses into actual voltages.

3.2 Software-, Processor-, and Hardware-in-loop Simulation Comparison

The following section compares and contrasts software-in-loop (SIL), processor-in-loop (PIL), and hardware-in-loop (HIL) in a few important categories.

The main feature of PIL simulations is the inclusion of the controller and controller software directly into the simulation. This is in contrast to SIL simulations, where only the software which would normally run on the controller is included in the simulation, and HIL, where the controller and its peripherals are included.

3.2.1 Usage During Design Process

The differences between the simulation types have some effects on the way they can be used during the design process, and the requirements for them to be used. Generally speaking, being able to use a type of simulation earlier in the design process is more desirable, since this allows more flexibility in terms of verifying the design.

PIL simulation requires at least a preliminary version of the hardware, but not necessarily with all the peripherals functioning. This is in contrast with SIL, which requires no hardware, and HIL, which requires hardware with working input and output peripherals. Because of this, SIL simulation can occur the earliest in the design process, before any of the actual controller hardware is available. PIL simulation will be the next available, as the microcontroller platform must be available, but none of the peripherals. Finally, HIL processing can be done when the hardware and some of its sensors become available.

3.2.2 Real-time Simulation Requirements

Both SIL and PIL are non-real-time simulation methods. Therefore, they don't have any real-time requirements, and thus are more flexible as to the simulation platform on which they can be run.

In contrast, HIL must be run at full speed to accurately test the hardware. This requires more performance from the simulation platform, and could possibly limit the amount of electrical network included in the simulation.

3.2.3 Interfacing Issues

The interfacing requirements for SIL, PIL, and HIL simulations are vastly different. SIL simulations require no specialized hardware whatsoever, since the controller operation is represented completely in software. All data is passed through memory structures or communication ports.

However, in SIL simulations, in order to accurately simulate the action of the software under test, the software must be presented with binary data as it would if run on an actual controller. Some software integration to perform conversion between simulation values and controller data must be performed to make the software run as if it were on the target controller.

PIL simulations, in addition to the value translation required for SIL simulations, also require some sort of communications port for exchanging simulated data between the simulation and the controller. The simulation platform tends to be a piece of software running on a general-purpose computer, so standard communications methods such as Ethernet or RS232 serial ports are usually available. Most controllers come with some version of a serial port, or possibly even Ethernet, so this is usually a trivial requirement. Also, additional memory is required on the controller to store the code and process the simulation messages, as well, but most controller product lines come with varying amounts of memory.

Also, for SIL and PIL, the different components of the simulation must be synchronized. A fixed timestep could be used, where the controller runs for the same amount of time as a simulation timestep. After every timestep, the data and results get passed back and forth [32] [28].

In contrast to SIL and PIL, the interfacing requirements for HIL simulations is much greater. HIL simulations require specialized test hardware for generating electrical signals in real-time for the controller to read. This simulation platform must be capable of simulating the target system, including calculating and generating every signal required by the controller, reading the results from the controller, and incorporating the results back into the simulation, all in real-time. This hardware can no longer be considered general purpose, and due to economies of scale, will be more expensive.

3.2.4 Speed of Simulation

Both SIL and PIL are non-real-time simulations. Because of this, they will both generally be slower then HIL.

Since PIL is a co-simulation, and requires communications and synchronization between multiple simulation platforms, PIL is generally slower then SIL.

3.2.5 Accuracy of Simulation

For SIL simulations, the development environment used to compile the code for the simulation is for the same platform as the simulation, not the target hardware. Libraries available on the controller might not be available or identical to those on the simulator.

For PIL and HIL simulations, the actual development tools and libraries are used to compile code for the simulation. This gives a good assurance that everything associated with the software is working properly, including the fact that the software fits onto the controller.

SIL simulations require some sort of assumption regarding the processing speed of the target controller and how long the processing requires, as the code is running on a seperate platform. In PIL and HIL simulations, since the code is running on the actual platform being simulated, the processing performance is accurately accounted for.

For SIL and PIL, only the processing portion of the controller operation is provided. The operation of the peripherals must be simulated in software. HIL simulations, since they feature the least simulated equipment, would tend to be the most accurate, and have less interfacing errors then PIL simulations. Peripheral simulation of varying accuracy can skew simulation results [33].

4 Development of a Software- and Processor-in-Loop Simulation

This section describes the development of a PSCAD / EMTDC component to enable software- and processor-in-loop simulations. The generic model of a microcontroller used to guide the component development is expanded, followed by an overview of the component and how it implements the generic model.

The component is used to create two different simulations: a basic development and troubleshooting simulation and a simulation of a constant voltage vs. frequency speed control system for an ac induction motor. Additionally, the induction motor case is run in two different configurations, with speed being measured by a large dynamometer and belt system, and also by a small quadrature encoder mounted on the end of the motor shaft.

To verify the accuracy of the component, the resulting control system is used to start and run the motor. The motor acceleration curves for simulated and actual starts of the motor are compared. The control system is run using a variety of different parameters for the PI controller.

Finally, the optimization functionality of PSCAD / EMTDC is used to improve the parameters of the PI controller. The proportional, integral, and anti-windup parameters are set to arbitrary values, the optimization algorithm is activated, and new parameters are generated. Both sets of parameters are verified by using them to operate the motor.

4.1 Generic Controller Model

In order to simplify the task of including the behaviour of an arbitrary microcontroller in PSCAD / EMTDC simulations, a generic model of the controller is created. The model focuses on the activity of the controller and controller peripherals pertinant to the simulation. The exact details of how the controller peripherals actually operate are simplified to a set of numeric parameters in order to allow different types of controllers to be easily considered for processor-in-loop simulations.

The following sections describe the specific behaviours modelled by the controller component.

4.1.1 Limited Access Processor Model

The action of the processor in this simulation is represented by two software routines. The first is an initialization routine, which is used to set the processor into a common initial state. The name of the initialization routine is *controller_reset()*.

During regular controller operation, the controller calculates output data based on the input data. For the purposes of this simulation, the calculation portion of the process is modelled as a single task running periodically on the controller. The task is assumed to be the highest priority task on the controller, and so will run without being pre-empted or delayed for any reason. The amount of time between consecutive runs of the task is called the task period. The amount of time required to complete the task is called the task delay, which is necessarily less then the task period. The name of the periodic task is *controller_process()*.

To further simplify the task of including the behaviour of an arbitrary microcontroller in PSCAD / EMTDC, the transfer of new data to or from the controller is prohibited during the processing portion of the simulation. The processor receives all required external data before running the periodic task. The task then runs to completion at full speed, calculating the required results without interruption by the simulation. After the task finishes, the results and elapsed time, or task delay, are transmitted back to the simulation. The simulation then continues, incorporating the new results after the simulation time has progressed by the reported task delay. This restriction on accessing new data during the task execution is observed both in simulation and in live running of the controller, to keep the performance of the two as close as possible.

The most accurate format for representing the two processes for an arbitrary controller is the C programming language. The vast majority of modern controllers have some sort of C compiler available. Representing the task in C code also allows the code to be run from inside PSCAD / EMTDC, allowing software-in-loop simulation.

4.1.2 Digital Input and Output Peripherals

The digital channels of the controller are represented by a series of bits, with one bit per input or output channel. For input channels, no delay occurs between a voltage change being registered at the controller and being readable by the controller. For output channels, no delay exists between the writing of the channel and the voltage changing at the controller output.

4.1.3 Analog Input Peripheral

Most controllers have some sort of analog input circuitry to convert analog voltages into digital representations for processing. Analog input peripherals are represented in generic form by a few different parameters.

The first set of parameters map the input voltage range into a binary data range. Both the voltage range and the corresponding binary range are represented by minimum and maximum values. The voltage range is assumed to map linearly to the binary range. Voltages outside the measureable range are converted to the closest valid binary limit. No consideration is given to special processing for out-of-range conditions.

The next parameter is the resolution of the conversion in bits, which represents the accuracy of the conversion. The data presented to the *controller_process()* task is 16 bits in length. If the actual resolution of the peripheral is less, the data is padded to occupy the most significant bits of this number.

The last parameter is a conversion delay. Different analog input circuits can require different amounts of time to convert voltages into binary numbers. This means that the analog data is delayed from being read by the controller. The effect of the data conversion delay is modelled by using voltages from earlier in the simulation.

An additional simplification of the analog input peripheral is all analog input channels for a controller are considered to be sampled simultaneously. Multiple serial channel conversions are represented by a single averaged delay for all channels, which is not strictly correct in all cases. Generally, the conversion times for the controllers examined for this thesis are in the order of microseconds.

4.1.4 PWM Output Peripheral

Analog output for modern controllers is generally performed through Pulse Width Modulation, or PWM, peripherals. The parameters required to generically represent a PWM channel are a clock period and a timer resolution in bits. Multiplying the clock period by the number of bits in one timer cycle gives the period of the PWM signal.

Actual PWM peripherals generally have a number of different operating modes, depending on whether the PWM timer value increases or decreases or both, at what point during a PWM timer sequence the output is affected by new values, and whether having a timer value greater then the required amplitude results in a high signal or a low signal. (i.e. whether the PWM output is inverting or non-inverting.) The peripheral model is implemented using a non-inverting mode with new values accepted only when the timer is at a minimum value, as this mode is common to all of the controllers initially considered [34] [35] [36].

4.1.5 3-Phase PWM Output Peripheral

3-phase PWM output is provided by coordinating 3 pairs of PWM switch outputs to provide three sine waves with a phase difference of 120 degrees between them. As in single-phase PWM, the 3-phase PWM frequency is determined by the 3-phase PWM clock period and number of bits of amplitude resolution. The actual switch output values are determined by the values for amplitude and frequency of the channel.

4.1.6 Timeline of Controller Activities.

In order to simplify the simulation and reduce the amount of communication required between PSCAD / EMTDC and the controller, a simplified model of the operation of the controller was implemented. The following is a description of simplified controller operation actions accounted for in the simulation.

When powered on, the *controller_reset()* routine is run to set the controller internal variables to some reasonable state.

After reset, the following steps are repeated periodically:

1. The A/D converter starts to convert the voltage experienced on its terminal.
2. A period of time lapses while the A/D converter converts the input voltage to a binary number.
3. When the A/D converter is finished, the controller immediately reads the most current digital input values, and then starts the *controller_process()* task.
4. A period of time called the task delay lapses while the *controller_process()* task calculates.
5. At the end of the task delay, the controller process writes the most recent output values to the PWM and digital output peripherals.
6. Immediately upon being written, the digital out channel voltages are set based on the channel value.
7. After the task period time expires since the start of the last A/D conversion, the A/D converter is retriggered, starting a repetition of the process.

During the above timeline, the PWM timer is not necessarily in synchronization with the above process. The value currently in the PWM register when the PWM timer reaches its lowest value is used to calculate the output voltage switch times.

As was mentioned earlier, microcontrollers in reality are flexible enough to perform these operations in any order at any time. The aforementioned timeline restricts the controller activities in order to easily integrate the controller behaviour into the PSCAD / EMTDC simulation.

4.2 Component Overview

A PSCAD / EMTDC block was written to allow connections from the simulation to a set of C routines which interface with the appropriate software and hardware.

4.2.1 Component Diagram

The PSCAD / EMTDC component has arrays of wiring connections for each type of signal. Figure 4.3 shows a Controller component with analog, digital, and arbitrary parameter inputs wired up on the left, as well as a component to extract one of the 3-phase ac channel switch outputs on the right. Also visible in the figure are conversions for the analog input signals as well as the blocks required to implement optimization.

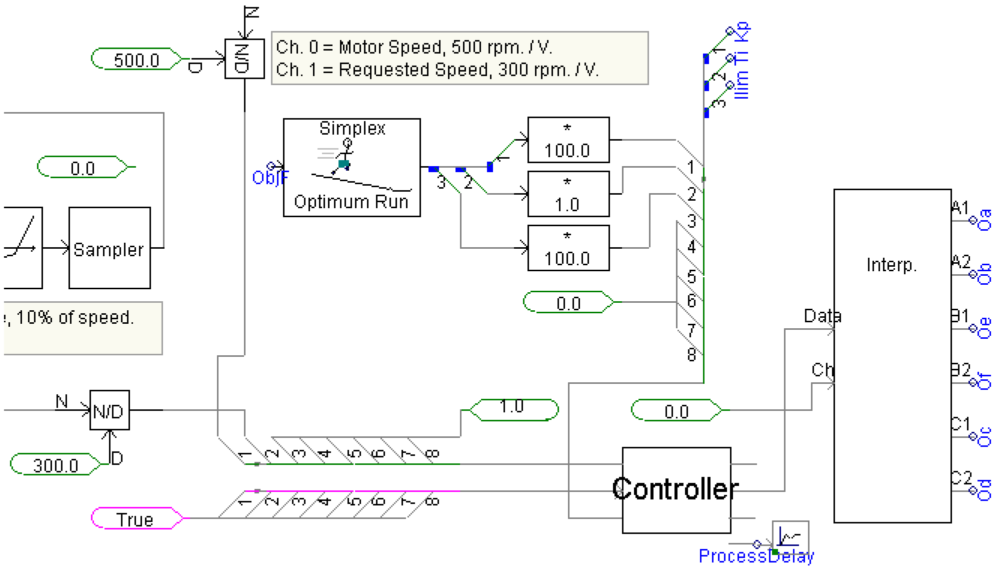
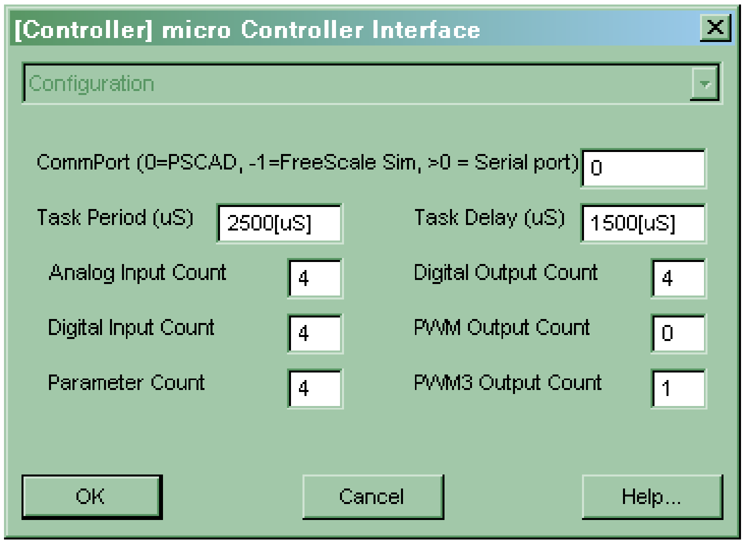


Figure 4.3: PSCAD / EMTDC Controller Component

4.2.2 Parameter Block

Figure 4.4 shows the numeric parameters available in the panel. As was mentioned earlier, the task period is the amount of time between the start of consecutive task runs. The task delay is the amount of time the task requires to complete, and must be less then the task period. The various count parameters are the number of each type of channel available on the controller.

Specifying the task delay or any of the count parameters is only valid when using software-in-loop simulation. When the microcontroller block is running in processor-in-loop mode, these parameters are calculated and reported by the microcontroller connected to the simulation.


[Controller] micro Controller Interface

Configuration

CommPort (0=PSCAD, -1=FreeScale Sim, >0 = Serial port)	0		
Task Period (uS)	2500[uS]	Task Delay (uS)	1500[uS]
Analog Input Count	4	Digital Output Count	4
Digital Input Count	4	PWM Output Count	0
Parameter Count	4	PWM3 Output Count	1

Figure 4.4: PSCAD / EMTDC Controller Parameter Block

4.2.3 Controller Data Structure

A C data structure is created to hold all input and output data, peripheral configuration data, as well as an array of arbitrary parameters for optimization. Using this structure has a number of positive effects, including keeping the C code consistent between PSCAD / EMTDC and the controller, hiding processor-dependant details of how data is read from or written to the peripherals, enforcing limited access to the processor, and revealing selected processor-specific information regarding the peripherals, such as clock speeds, bits of resolution, etc., for use by PSCAD / EMTDC.

The structure is identical in either PSCAD / EMTDC or on the controller. Implementing a coordinating structure this way adds to the processing time by requiring data to be transfered into and out of structure, reducing the processor usage efficiency. However, this time is accounted for in the simulation, and allows for a smooth transition from software-in-loop to processor-in-loop simulation.

A C header file describing this structure is included on page 70 in the appendix.

4.2.4 Peripheral Simulation Code

The PSCAD / EMTDC peripheral simulation code interfaces the controller data to the simulation. This code contains functionality for translating data between the microcontroller structure and the simulation for each type of peripheral in use.

4.2.5 Process Code

The *controller_reset()* and *controller_process()* routines are held in a single C file. For software-in-loop operation, this file is included in the PSCAD / EMTDC simulation. For processor-in-loop operation, the exact same file is included in the build of the controller code and installed on the controller.

The C program file implementing the *controller_reset()* and *controller_process()* routines for the constant voltage vs. frequency algorithm is included on page 72 in the appendix.

4.3 Controller Operating Software

The following sections give an overview of the software required for the controller to operate under PSCAD / EMTDC. Together, these routines implement a rudimentary two-task operating system, with a high-priority task for the process to be modelled in PSCAD / EMTDC, and a low-priority task for operating the controller lights, switches, simulation message passing, and debug modes of the controller.

4.3.1 Controller-Specific Peripheral Operation Code

The controller-specific integration code details the controller initialization and peripheral configuration and control. This code details the specifics of data reads from peripherals to structure, data writes from structure to peripherals, text and simulation messages to and from the communications port, and operation of the mode change buttons and indicator lights.

Additionally, the controller must be able to run the *controller_process()* routine periodically, and measure the amount of time required to complete it. This requires timer initialization, starting, and stopping, as well as some controller-specific arrangement of interrupt code for running the controller process periodically.

4.3.2 Controller-Independant Message-Passing Code

The physical hardware used to pass data between PSCAD / EMTDC and the microcontroller is the RS232 serial communications. Although RS232 is an old standard, and more modern and faster ways currently exist for moving data between processors, many microcontroller, as well as the GNU compiler of PSCAD / EMTDC, support it.

A simple protocol is created to pass various messages between PSCAD / EMTDC and the microcontroller. The protocol includes reset, data request and transfer, and process commands. The protocol also implements error checking and retransmission of missed messages, increasing the robustness of the link.

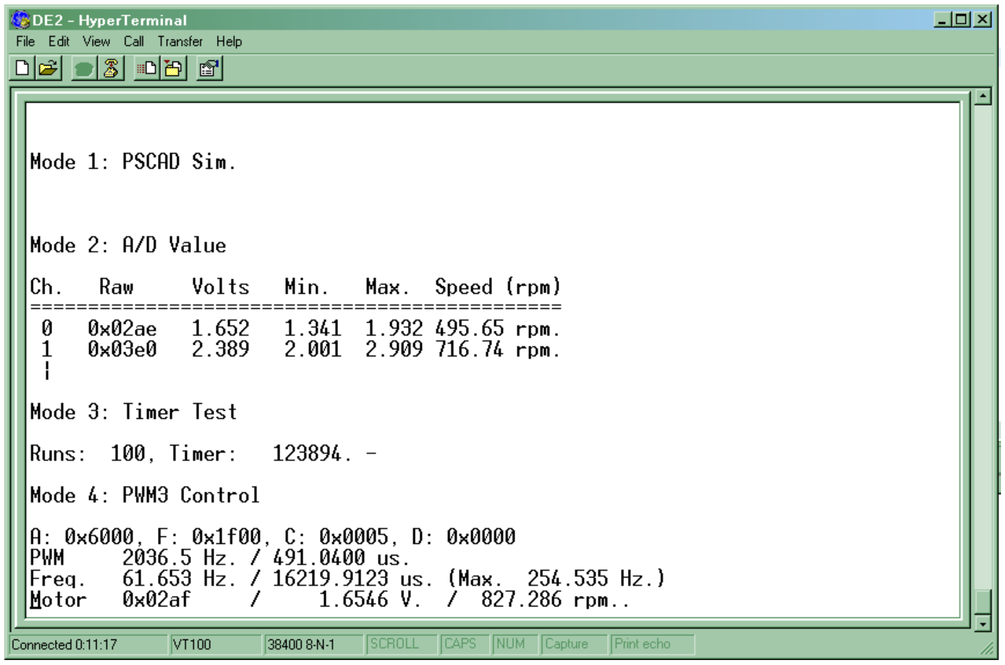
As a part of the simulation reset message, the operational parameters of the microcontroller peripherals are passed to PSCAD / EMTDC in the reset response message. This allows PSCAD / EMTDC to format the sensor data to the microcontroller and simulate the actuator signals from the microcontroller accurately.

Additionally, a number of arbitrary parameters are passed from PSCAD / EMTDC to the microcontroller. These parameters allow the behaviour of the microcontroller to be modified in PSCAD / EMTDC-based optimization.

4.3.3 Controller-Independant Operating Code

Some functionality is not required for simulation or running live, but assists in determining the performance of the microcontroller and its peripherals. This functionality is located on the microcontroller, but is not microcontroller-specific. In order to allow the reuse of this code on multiple microcontroller types, the microcontroller-specific configuration and code was kept segregated from the generic microcontroller code.

In addition to run mode and simulation mode, the microcontroller can be in one of four different test modes:


DE2 - HyperTerminal
File Edit View Call Transfer Help
Connected 0:11:17 VT100 38400 8-N-1 SCROLL CAPS NUM Capture Print echo

```
Mode 1: PSCAD Sim.


Mode 2: A/D Value

Ch.   Raw     Volts   Min.   Max.  Speed (rpm)
==============================================
 0   0x02ae   1.652   1.341  1.932 495.65 rpm.
 1   0x03e0   2.389   2.001  2.909 716.74 rpm.
 |

Mode 3: Timer Test

Runs:  100, Timer:   123894. -

Mode 4: PWM3 Control

A: 0x6000, F: 0x1f00, C: 0x0005, D: 0x0000
PWM     2036.5 Hz. / 491.0400 us.
Freq.   61.653 Hz. / 16219.9123 us. (Max.  254.535 Hz.)
Motor   0x02af     /     1.6546 V.  /  827.286 rpm..
```

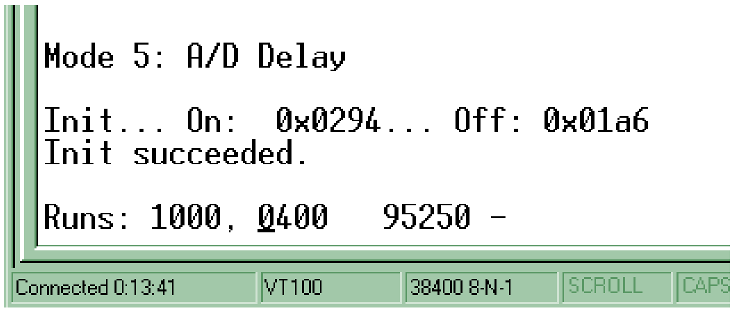
Figure 4.5: Screen Capture of Modes 1 Through 4

- A/D Value - In this mode, the microcontroller reads A/D channels and displays the raw data and converted voltage. For the purposes of setting up the target speed, the calculated motor speed for each channel is also displayed.

- Timer Test - This mode is for verification of the timer functionality of the microcontroller. The microcontroller runs the *controller_process()* routine repeatedly with no data and without writing the outputs to the peripherals, while measuring and displaying the amount of elapsed time taken. The microcontroller also toggles the onboard LEDs at the start and end of the time to allow independant verification of the timer via oscilloscope.

- PWM3 Manual Control - This mode allows manual control of the 3-phase PWM peripheral. The frequency, amplitude, PWM clock frequency, and deadtime of the 3-phase ac signal are all available as registers, which can all be manually written in this mode. Additionally, this mode has provisions for recording an operating amplitude and performing a step-change to that operating point. This functionality was used to generate the 20 Hz. and 40 Hz. signals for the motor parameter determination.

- A/D Delay - This mode measures the amount of time required for signal to be read by the A/D converter of the microcontroller. The microcontroller does this by manipulating the digital output line 0 and waiting for a corresponding change in the analog input channel 0. In order to use this mode, these two channels must be connected with a piece of wire.



```
Mode 5: A/D Delay

Init... On:  0x0294... Off: 0x01a6
Init succeeded.

Runs: 1000, 0400   95250 -
```

Figure 4.6: Screen Capture of Mode 5, Showing 95.25 uS. Average A/D Conversion Time

Additional microcontroller-specific code is required to run live. The peripherals, such as A/D and interrupt timers, require microcontroller-specific configuration to get them ready for use. Methods for reading input port data, writing to output ports, and interfacing with available buttons and LEDs are microcontroller-specific, as well.

4.4 Design Flow

This section details how a processor-in-loop simulation is created with an arbitrary microcontroller. The flow is divided into microcontroller-specific and simulation-specific development. Once the microcontroller-specific development has been completed, it can be reused on subsequent simulations.

4.4.1 Controller-Specific Development

A fair amount of development work needs to occur before a new microcontroller can be used with the microcontroller component. The most time-consuming portion is to write the microcontroller-specific module for performing the required configuration and activities on the specific microcontroller. Many microcontroller resources must be utilized, requiring a good knowledge of the microcontroller peripherals and functions.

The next step is to ensure the message passing and microcontroller support code will work on the new controller. Although microcontrollers from four different manufacturers were used, porting the code required very little in the way of modifications.

The next step is to verify the parameters of the microcontroller peripherals using different debugging modes of the microcontroller code. These parameters are then either hard-coded or written into the microcontroller data structure on reset.

The operation of the new microcontroller and peripherals is then verified with the troubleshooting test case. The message passing code, as well as the operation of the peripherals are all verified. The debug and troubleshooting modes of the microcontroller are used. The actual parameters are hard-coded into the structure for that microcontroller, and are passed to PSCAD / EMTDC as a part of the simulated reset process.

Once all of these processes are completed, the microcontroller can be used accurately with the component.

4.4.2 Simulation-Specific Development

Integrating the controller component in a simulation case requires instantiating the component and hooking the inputs and outputs up to the rest of the simulation. Input sensor voltages are modeled as scaled versions of live parameters such as motor speed or line voltage. PSCAD / EMTDC converts the voltages into binary data for the controller code to process as if the data had been read by the appropriate input peripheral. The PWM or 3-phase PWM signals are connected to power electronic switches in the simulation.

Implementing optimization is similar to that for a regular simulation. An array of optimization parameters are generated using the optimization run block and wired up to the arbitrary parameter input of the controller component. The controller code is then modified to access those parameters. An objective function is also required, to measure the quality of the performance of the resulting simulation.

Once a parameterized algorithm implementing the general desired behaviour is programmed for the microcontroller, a simulation designer can implement the microcontroller component in a simulation as a "black box", without needing to know the exact internals of the algorithm. The only requirement of the designer is information about the number of parameters to which the microcontroller will respond.

4.4.3 Software-In-Loop Operation

For software-in-loop, or SIL, operation, the *controller_reset()* and *controller_process()* routines are embedded directly in the PSCAD / EMTDC environment. These routines operate on the data in the local structure.

Since this code runs natively on the PC, and not on the target processor, the task delay, or time required to run the code, is not available, and must be estimated and entered into the controller block parameters. Another simplification for the purposes of facilitating software-in-loop simulation is a fixed value for the task delay in every timestep. In reality, different iterations of the *controller_process()* task might take different amounts of time, depending on the inputs and algorithm. A single task delay is an approximation, at best.

Also, the peripheral parameters, such as analog-to-digital resolution and conversion time, are hard-coded to reasonable values, or values close to those for the intended microcontroller, if known.

4.4.4 Processor-In-Loop Operation

For processor-in-loop, or PIL, operation, the initialization and task routines are compiled and installed on the target controller using the development tools specific to that microcontroller. The CommPort parameter in the PSCAD / EMTDC microcontroller component parameter block is set to the actual number of the serial port where the microcontroller is connected.

On reset, PSCAD / EMTDC reads in the peripheral parameters from the specific microcontroller. In this way, the correct parameters for the microcontroller being used are available to PSCAD / EMTDC.

Also on reset, the value for the process period is loaded into the microcontroller. The microcontroller uses this value for any calculations of elapsed time the algorithm may require, as well as to configure itself to automatically re-run the *controller_process()* routine at the requested rate.

Since the process code runs on the microcontroller, the microcontroller is able to determine the amount of time required to execute the code each timestep, and return the value to PSCAD / EMTDC with each set of results.

4.4.5 Live Operation

When the controller is first powered on, it defaults to simulation mode. Pressing the run mode button prompts the controller to start the peripherals and run the *controller_process()* routine periodically.

The period used is the task period specified in the component parameter block for the latest simulation, or the hard-coded default if no simulation was performed since the controller was last powered on. Similarly, if the algorithm references arbitrary parameters, the parameters from latest simulation will be used. If no simulations have been run since the controller was last reset, the default values hardcoded into the structure in the controller code will be used.

4.5 Physical Controllers.

A number of physical controllers were considered for use as an induction motor speed control.

4.5.1 PIC and Atmel Controllers

Initially, three different controllers were targeted to provide differing architectures, clock speeds, and development environments. The Cerebot 32MX4 development board with Microchip PIC32 processor was adapted for use in PSCAD / EMTDC, using Microchip's Development Environment for development and programming.

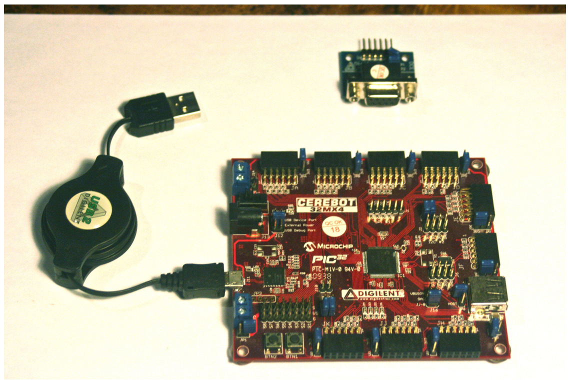


Figure 4.7: Cerebot 32MX4 Microcontroller Development System

The Cerebot II development board with Atmel AtMega64 processor was also adapted. The Atmel controller used AVR Studio from Atmel for development and programming.

Finally, the Atmel AtMega1284 processor in a breadboard-ready 40-pin DIP package was adapted, as well. Although the basic processor was the same as on the Cerebot II, the chip was capable of running at a wider variety of speeds, as well as being easily connected on a breadboard.

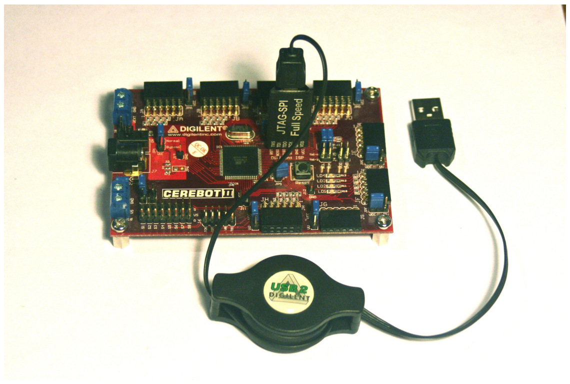


Figure 4.8: Cerebot II Microcontroller Development System

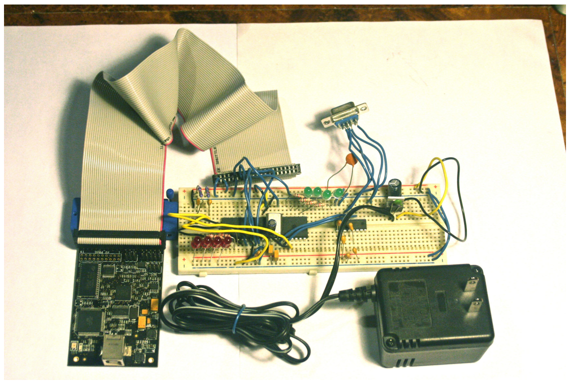


Figure 4.9: AtMega1284P 40-pin DIP Controller and AVR Dragon

4.5.2 FreeScale DEMO9S08MP16 Demonstration Board

The DEMO9S08MP16 demonstration board from FreeScale Semiconductor was also adapted for usage under PSCAD / EMTDC. This controller was designed specifically to be able to generate 3-phase ac, and included a special mode for generating switching signals appropriate for 3-phase ac by linking pairs of PWM channels together, as well as including switch dead-times in its output peripherals.

The FreeScale CodeWarrior development suite was used to development and programming for the this controller.

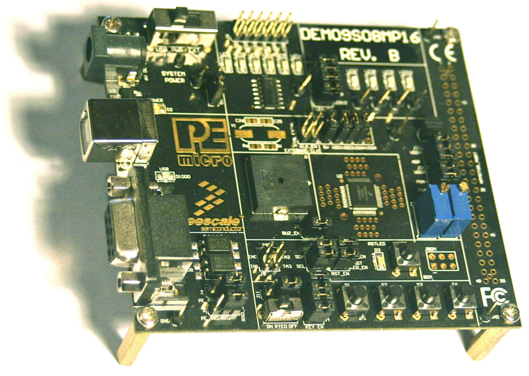


Figure 4.10: FreeScale DEMO9S08MP16 Demonstration Board

4.5.3 Altera DE2 Board / Cyclone II FPGA / Nios II Soft-core Processor

Finally, the DE2 evaluation board with an Altera Cyclone II FPGA was examined. This board contains a serial port, onboard LEDs and buttons, an LCD display, serial port connection, excessive amounts of off-chip SDRAM, and plenty of I/O pins. A Nios II soft-core processor is configured, containing all of the necessary peripherals, including an SPI bus for the A/D converter. The processor is also configured with a floating-point co-processor, in order to make floating point math much quicker.

Figure 4.11 shows a DB9 connector connected with red and brown wires in the foreground. This connector pinout matches the DB9 connector on the LabVolt thyristor component.

Although the DE2 board has some provision for A/D for its audio and video applications, an A/D chip, the MCP3202-B two-channel, 12 bit SPI (Serial Peripheral Interface) chip was selected to measure the motor speed from the dynamometer, as well as the requested speed channel from a potentiometer wired into the breadboard.

The process for creating a processor and C program out of an FPGA is somewhat involved. Altera has a suite of development software for generating working Nios II systems. The software packages used are listed below:

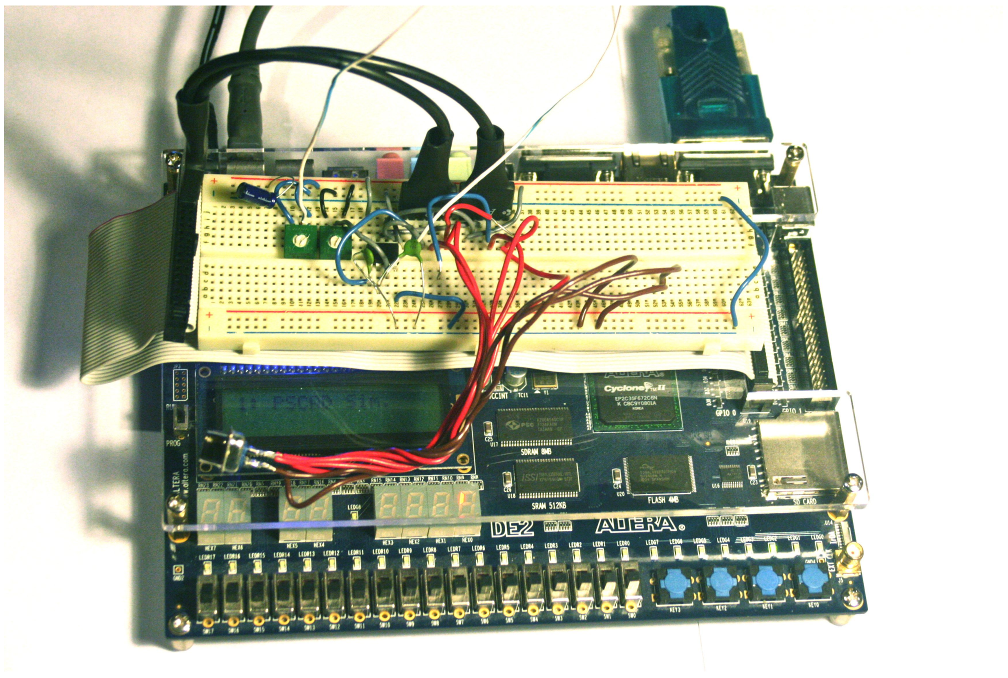


Figure 4.11: Altera DE2 Development and Education Board, w. MCP3202-B A/D Converter

- Qsys is a program for generating Nios II processor configuration by including interfaces for various peripherals.
- Quartus II is used to compile designs and create loadable files for the FPGA chip. Quartus II also allows the inputs and outputs of the various processor peripherals to be mapped to external pins on the device.
- Eclipse is an IDE for Nios II-specific C programming. Eclipse also integrates some tools to translate configuration information from Qsys into C macros to streamline peripheral access while programming.
- ModelSim is a simulator for testing HDL designs, from simple logic circuits to entire processor designs.

4.6 Development and Troubleshooting Test Case

The following is a short description of the development and troubleshooting test case, and how it was used.

4.6.1 Controller and Simulation Setup

This case consists of a simple test of the controller peripherals. The setup for this test case is fairly straightforward. The simulated analog input was connected to a simple circuit, and the actual controller analog input was connected to a potentiometer, enabling a small variable voltage to be applied to the input pin. The digital outputs were connected to red LEDs to allow observation of the signals when running live.

A few simple controller algorithms were implemented to manipulate the digital outputs and PWM outputs in response to the digital and analog inputs. The code was tested as a software-in-loop simulation, and then transfered to the various controllers to create processor-in-loop simulations.

4.6.2 Results

After much debugging, all controllers would behave much like the simulations described.

During development, the usage of some development and debugging environments allowed for breakpoints to be set in the code. When run as a part of PSCAD / EMTDC simulation, if the timeout period for PSCAD / EMTDC messages was removed, the combination of PSCAD / EMTDC and debugger for the controller could be used as an extended debugging platform. The code could be stopped, the debugging environment used to view or modify variables and step through code, and the simulation would wait patiently for the response messages to be sent by the controller.

The timing of the code execution gives valuable information regarding the performance of the controller and code. Certain controllers without floating point support would still compile and run the task. However, they would require a software library to perform the floating point math instead of a dedicated floating point co-processor. As a result, the task delay would be orders of magnitude longer to complete the required processing, resulting in unacceptable controller performance.

Executing different loops or branches can change the time required to run the task. If the timing requirements are strict enough, this could introduce jitter to the output.

After one development environment restricted its compiler optimizations due to time-limiting of its license, the task execution time was substantially longer then for the exact same source code under the original unrestricted compiler optimizations. Depending on how tight performance requirements are, this extra delay could also be significant. However, the longer task delay was observed in the results of the simulation.

Processor-in-loop simulations integrating the microcontroller did not require substantially longer amounts of time to complete the simulation then with similar model- or software-in-loop simulations.

4.7 3-Phase Induction Motor Speed Control Test Case.

In order to further confirm the accuracy of the PSCAD-based processor-in-loop simulations, a more involved simulation of a 3-phase induction motor speed control was created.

4.7.1 Controller Selection

Unfortunately, most of the microcontrollers used in the troubleshooting case were not able to control the 3-phase induction motor under the simplified microcontroller model required by the PSCAD / EMTDC component. None of the Atmel or Microchip microcontrollers were able to generate the coordinated switching signals for the 3-phase ac required to drive the thyristors and attached induction motor. Also, none of the microcontrollers had floating point co-processors, and so couldn't complete the required calculations in a reasonable amount of time.

The FreeScale microcontroller was able to generate 3-phase PWM signals. In fact, an application note explaining how to control a 3-phase induction motor appropriately for usage in a modern washing machine exists [37]. Unfortunately, this microcontroller did not have a floating-point coprocessor. The washing machine reference design used fixed-point math, requiring system math libraries, some of which were written in microcontroller-specific assembler language. This sort of code is not easily portable to the PSCAD / EMTDC environment for usage in SIL simulation.

Additionally, the FreeScale reference design also required additonal code run on the main processor to calculate the PWM register values as the phase advanced. This extra interrupt routine was not accounted for in the simplified microcontroller composition outlined in the previous chapter.

4.7.1.1 3-phase PWM-generating Peripheral for Nios II In order to fit into the generalized controller operation detailed above, a 3-phase PWM component was created using Verilog HDL. This component accepts an amplitude and frequency value for the required sine waves as calculated by the processor, and generates the individual amplitude values for each phase internally. This decision to move calculations from software into hardware has the effect of unloading the main processor of the repetitive 3-phase generation calculations, and is one component of embedded systems design flow [12].

The component uses the CORDIC algorithm for calculating trigonometric functions in hardware. CORDIC is short for COordinate Rotation DIgital Computer [38] [39].

The component features scalable and signed frequency and amplitude registers. The frequency values are signed, which allows generation of the 3-phase signal waves in either direction. An additional clock scaling register was implemented in order to slow down the 50 MHz system clock to drive the PWM timers and provide a range of possible PWM frequencies. Finally, a deadtime register was implemented to prevent both switches for a single phase from being on simultaneously.

4.7.2 Controller Task Algorithm

The speed control task of the controller can be stated in the following terms: Given a motor requested speed and an actual speed, what 3-phase voltage and frequency should be applied to the motor to minimize the error between requested and actual speeds?

The speed control algorithm is implemented by two main calculations: a PI controller with integral limiter and a constant voltage vs. frequency calculation. The complete algorithms are included in the appendices on page 72

4.7.2.1 PI Speed Control The PI controller is implemented to calculate an output frequency for the 3-phase voltage based on the current motor speed and requested speed. The equation for a PI controller is as follows [40]:

$$f_r(t) = K_p \left[e(t) + \frac{1}{T_I} \int e(t) dt \right]$$

where f_r is the requested frequency, $e(t)$ is the difference between the requested and actual speeds, K_p is the proportional constant, and T_I is the integal time constant.

An additional anti-windup parameter is used to limit the value of the integral, and therefore the amount of correction provided by the integral term.

4.7.2.2 Constant Voltage vs. Frequency Control After the electrical output frequency is determined, the voltage at which this frequency is applied to the motor must also be determined. Neglecting the voltage drop across the armature resistance and leakage reactance, the following is the equation governing the applied voltage, frequency, and air-gap flux density [41]:

$$V_a = \left(\frac{f_e}{f_{rated}} \right) \left(\frac{B_{peak}}{B_{rated}} \right) V_{rated}$$

where f_e and f_{rated} are the applied and rated magnetic flux, V_a and V_{rated} are the applied and rated voltages, and B_{peak} and B_{rated} are the peak and applied magnetic flux values.

If the rated voltage is applied, the equation is as follows:

$$B_{peak} = \left(\frac{f_{rated}}{f_e} \right) B_{rated}$$

Applying the full rated output voltage to the motor at a reduced frequency will cause the motor to experience excessive peak magnetic flux, possibly damaging it.

A constant voltage vs. frequency control algorithm is employed to limit the output voltage and therefore make the amount of magnetic flux applied constant. Applying a constant rated magnetic flux requires setting B_{peak} to B_{rated}, leaving:

$$V_a = f_e \frac{V_{rated}}{f_{rated}}$$

This states that for a constant magnetic flux, the required applied voltage is proportional to the desired applied frequency of the 3-phase sine waves.

4.7.3 Controller Task and Parameters

The ac induction motor PI speed control was programmed in C, using arbitrary parameters passed from PSCAD / EMTDC as the proportional gain and integral time constant. In order to prevent ”wind-up” of the controller, an additional parameter was used to limit the effect of the integral parameter.

The C code for the induction motor test case was fairly straightforward. Floating point math was used for the simplification of the calculations. The 50 MHz Nios II controller with floating point math coprocessor enabled had a task delay of less then 1.5 ms. for this task. In order to leave the controller time to perform other tasks such as reading the button inputs and blinking the LEDs to show it was still operating correctly, the task period was initially fixed at 2.5 ms.

Each run consisted of one simulation using each of the three sets of inertia and damping coefficients mentioned in the previous section, and three live data captures. The simulations were performed first, which also had the effect of loading the gain, integral, and integral windup parameters for the PI controller into the physical controller.

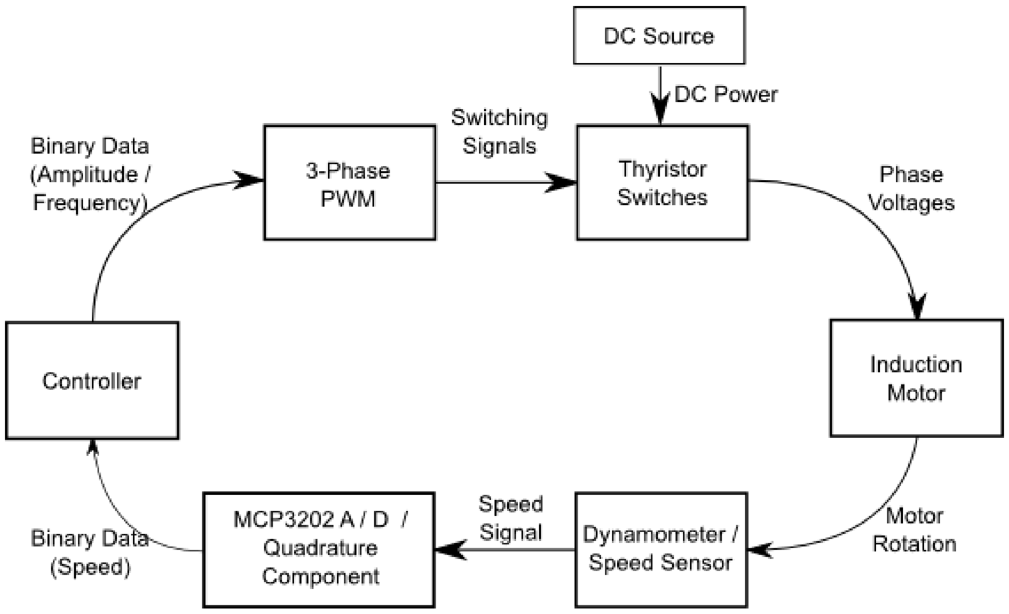


Figure 4.12: System Overview

4.7.4 Controller Wiring

The LabVolt equipment at the University of Manitoba Power Electronics Lab was used for experimentation with PSCAD / EMTDC and the controller. The complete list of LabVolt equipment used consists of:

- data acquisition system and PC software
- dc power supply
- thyristor bank
- induction motor
- prime mover / dynamometer

Figure 4.12 shows an overview of the system.

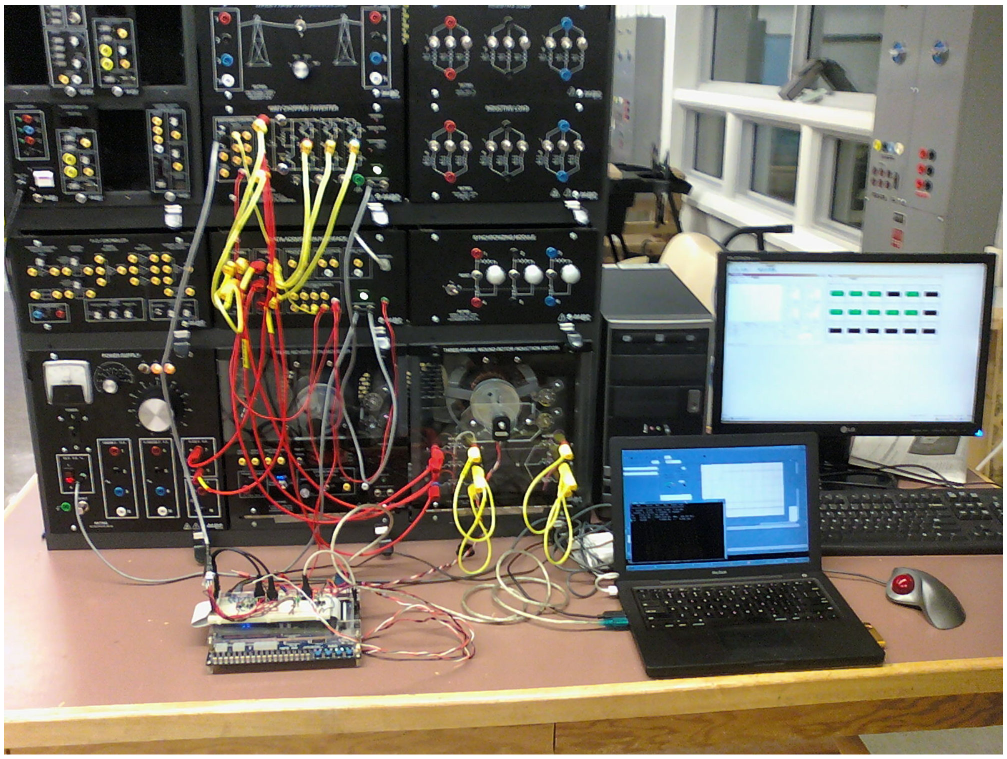


Figure 4.13: Equipment Setup

The target activity for the controller and motor was to start from rest, accellerate to 600 rpm. target speed, and dwell there with a minimum of error. The target speed of 1/3 the rated speed of the motor was chosen to shorten the length of simulation and data acquisition time required.

During the initial setup, the equipment was wired up as shown in Figure 4.12. The controller DB9 switch cable was connected to the DB9 cable of the thyristor unit. Figure 4.13 shows, clockwise from the upper left, the lab equipment, data acquisition workstation, simulation workstation, and Altera DE2 FPGA board.

The LabVolt data acquisition system was setup to capture data including the start and a period of stable running afterwards. The triggering in the data acquisition system in the LabVolt equipment was found to be somewhat inconsistent, so the speed curves captured during the runs were shifted in time with post-processing to line up their initial movements. The LabVolt data acquisition component was connected to measure the phase current going to the motor and the speed signal of the dynamometer. The speed signal sensitivity was set to 500 rpm. per volt in the software to match the dynamometer output.

The speed request channel of the A/D converter was connected to a potentiometer on the breadboard, and the sensitivity was set to 300 rpm. per volt. for full-scale range closer to that of the motor.

4.7.5 PSCAD / EMTDC Simulation Setup

A simulation used for developing the ac functionality is modified to match the equipment in the lab as closely as possible. Parameters for the power supply are modified by adding a small resistance to an ideal power supply model. This is done to match the supply voltage droop seen when running the physical motor.

Parameters for the induction motor are modified as well. The motor ratings, resistances, and reactances are all provided by a fellow student working with a similar motor.

A pair of C routines *controller_reset()* and *controller_process()* are written to perform PI control of the induction motor, using the constant voltage vs. frequency algorithm. These are reproduced in page 72 of the appendices.

The routines are parameterized to allow optimization by PSCAD / EMTDC, with the parameters being the proportional coefficient, integral time constant, and the anti-windup parameter. The routines are also compiled and installed in the DE2 / Nios II controller.

4.7.6 Induction Motor Inertia and Damping Parameters

The inertia and mechanical drag figures pose a problem, as they are difficult to measure. The equation of motion for induction motors is as follows [42]:

$$J\frac{d\omega}{dt} = T_m - T_e + D\omega$$

where J is the inertia of the system, ω is the velocity of the motor, D is the mechanical drag of the system, and T_m and T_e are the mechanical and electrical torques, respectively.

In order to generate inertia and mechanical drag values for the motor, the motor was physically driven with a constant 3-phase ac signal at both 20 Hz. and 40 Hz., using the 3-phase control mode in the controller code. Speed data was captured as the motor accellerated. The simulation was run repeatedly, modifying the inertia and mechanical drag parameters until the simulation curves matched the actual runs as closely as possible.

4.7.7 Dynamometer Configuration

Initially, the motor is connected to a dynamometer via a rubber belt in order to use the dynamometer speed measurement sensor. The unloaded dynamometer and belt combination add a significant amount of drag and inertia to the system.

4.7.7.1 Induction Motor Parameter Estimation Two attempts are made to calculate accurate values for the inertia and mechanical drag. In the first attempt, the motor is started using a constant 40 Hz. 3-phase ac signal, and the resulting accelleration curve is recorded. The simulation of the same situation is run repeatedly, manually varying the two parameters until the speed curves match reasonably well at 0.5 and 1.0 seconds, while neglecting the steady-state speed.

Unfortunately, the values for mechanical drag were not constant for the two cases, suggesting a more complex relationship between drag and speed when starting the motor. Simulations using these parameter sets and a linear combination of them were not as accurate. The coefficients are detailed in Table 4.1.

Run	Angular Moment of Inertia (S)	Damping (pu.)
40 Hz., Match @ 0.5 & 1.0 S.	2.31	0.40
20 Hz., Match S. S.	1.45	1.65
40 Hz., Match S. S.	1.75	0.99

Table 4.1: Motor and Dynamometer Configuration Parameters

The runs used to estimate the induction motor parameters are recorded in Figure 6.21 and 6.22.

Proportional Gain	Integral Time Constant	Integral Limit	Figure Number	Page Number	Notes
1.0	1.0	0.0	6.23	76	Integration Disabled
2.5	1.0	0.0	6.24	76	Integration Disabled
10.0	1.0	0.0	6.25	77	Integration Disabled
25.0	1.0	0.0	6.26	77	Integration Disabled
100.0	1.0	0.0	6.27	78	Integration Disabled
250.0	1.0	0.0	6.28	78	Integration Disabled
1.0	0.66	1000.0	6.29	79	
1.0	0.10	100.0	6.30	79	Optimization 1 Start Point
10.0	1.00	500.0	6.31	80	Optimization 2 Start Point
124.6	1.52	12.72	6.32	81	Optimization 1
4.53	1.71	66.8	6.33	81	Optimization 2
25.0	1.0	0.0	6.36	83	Noise Injection, Kp = 25.0
100.0	1.0	0.0	6.37	83	Noise Injection, Kp = 100.0
250.0	1.0	0.0	6.38	84	Noise Injection, Kp = 250.0

Table 4.2: PI Controller Parameters

4.7.7.2 Run Parameters The coefficients used for the various dynamometer runs are in Table 4.2. Generally, three sets of live data were captured to make sure the live runs were repeatable.

The simulations reproduced in Figures 6.32 and 6.33 were taken using sets of coefficients generated by an optimization run beginning on the coefficients in Figures 6.30 and 6.31, respectively. The objective function used during these optimizations was the integral of the square of the error between the actual speed and the desired speed.

During the proportional-only runs, some fluctuations were found in the high gain runs. Noise in the speed sensor line, combined with the extra sensitive gain settings were the source of the fluctuations. The noisy speed signal was readily visible in the live data from any of the runs. In order to make the simulation more accurately describe the physical equipment, an attempt was made to model this extra noise signal by adding pseudo-Gaussian noise of similar amplitude to the speed sense line of the simulation.

4.7.7.3 Results Initially, to reduce the size of the ModelSim simulations, the timer resolution of the 3-phase PWM peripheral was set to 8 bits. However, the sine waves created by this were observed to have a fair bit of noise associated with them. The component data width was widened to 10 bits, allowing a finer control over the switch timings and alleviating this problem.

Software-in-loop simulations were also attempted and compared to similar processor-in-loop simulations. When the task delay was setup to be approximately the same value for the two types of simulations, the resulting speed curves were found to be almost identical. After this result, processor-in-loop simulations were used to generate the rest of the data.

For the dynamometer case, the simulator results only matched the behaviour of the motor in a general way. When generating the initial motor parameters, no set of parameters could be found which would allow both the 20 Hz. and 40 Hz. runs to match the corresponding simulations exactly. As a result, all three sets of parameters were used to generate simulated curves. See Figures 6.21 and 6.22 on page 75 for plots of the configuration tests.

For the proportional-only runs, the simulation speed curve had generally the same general shape as the live runs, but did not match the live runs exactly. The absolute steady-state speeds did not match, either. The elastc belt connecting the motor and dynamometer seems to be stretching as the motor accellerates, causing the dynamometer speed reading to lag behind the actual speed of the motor. Figures 6.23 through 6.28 show the general shapes of the curves trending together, but the controller seems to be confused by the noise in the speed signal.

The controller also shows only the general shape of the simulation curves match that of the real runs. Some examples of this are visible in Figures 6.29 through 6.31, which show the similarity of the J = 2.31 and D = 0.40 simulation and the live runs. Figure 4.14 is representative of the similarity between simulated and real runs of the motor for the three different sets of parameters. The red, green, and yellow lines are accelleration curves for three live runs of the controller and motor, while the dark blue, light blue, and black lines are accelleration curves for simulations using different parameters for inertia and mechanical damping.

The effect of optimization on the parameters was to improve the performance of the controller simulation. The results of the live runs in Figures 6.32 and 6.33 are improved over those of the optimization start points. Unfortunately, the simulations still match the live data only generally, and also seem to be affected by noise in the speed signal.

An attempt was made to model the noise present in the speed signal channel. Since a deviation of

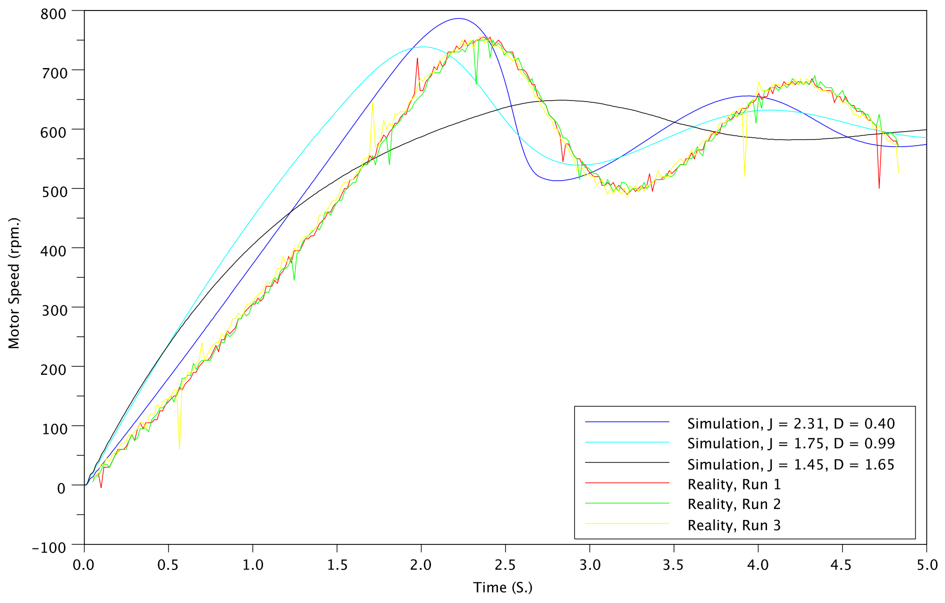


Figure 4.14: Dynamometer and Belt Speed Sensor, P = 1.0, K = 0.10, W = 100.

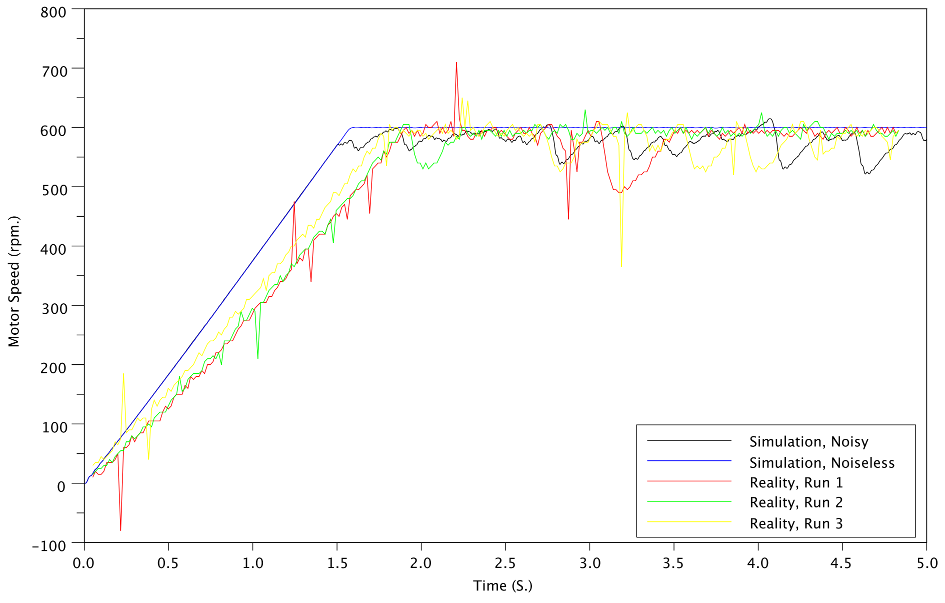


Figure 4.15: Dynamometer, Kp = 100.0, Noise Injection.

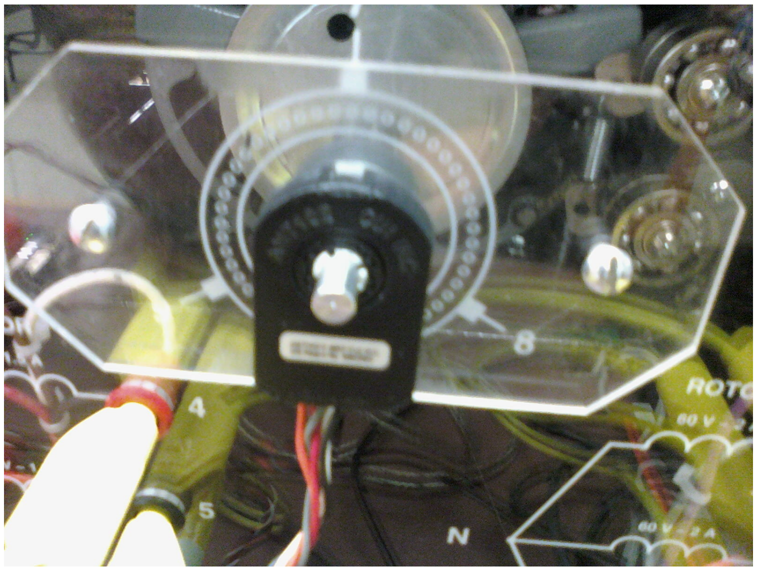


Figure 4.16: Quadrature Encoder Mounted on Motor Shaft

+-100 rpm. was captured during some of the speed runs, (see the real runs in Figures 6.21 and 6.22 for examples) a pseudo Gaussian noise generator was added to the speed signal in the simulation, and three more simulations were re-run. In Figures 6.36 through 6.38, the effect of the noise on the simulated run is evident, as the noisy simulation takes on some of the characteristics of the real runs. The figure for Kp = 100.0 with injected noise is reproduced in Figure 4.15.

4.7.8 Quadrature Encoder Configuration

The dyno and belt combination was replaced with a quadrature encoder mounted on the end of the shaft. This sensor generates a pair of square waves 90° out of phase based on the rotational position of the shaft. This sensor was used to eliminate the dynamics of the rubber belt and dyno mass and reduce the speed signal noise present in the dynamometer, and generate better agreement between the simulations and real runs. As can be seen from Figure 4.16, the encoder is relatively small. Internal to the sensor, the actual rotating part is smaller still, and made of light plastic, leaving the dynamics of the motor relatively unaffected.

The one drawback with the quadrature signal when compared to the voltage signal of the dynamometer is the LabVolt data acquisition system cannot accept a quadrature signal. In order to allow the speed signal to be captured, the controller code was modified to create an output PWM voltage signal matching that provided by the dynamometer.

4.7.8.1 Induction Motor Parameter Estimation Using this sensor greatly reduced the inertia and mechanical drag of the system, and allowed more accurate values to be generated for the induction motor component in PSCAD / EMTDC. The speeds for both 20 Hz. and 40 Hz. excitation matched within 10 rpm. between the simulated and measured values.

The runs used to estimate the induction motor parameters are recorded in Figure 6.39 and 6.40 on page 85. Only a single set of parameters was required to match both sets of 20 and 40 Hz. runs well. These parameters are given below:

Angular Moment of Inertia (S)	Damping (pu.)
0.539	0.142

Table 4.3: Table of Motor / Quadrature Sensor Coefficients

4.7.8.2 Run Parameters The coefficients used for the various runs are as follows:

The high-proportional gain runs in the first set of runs also exhibited the effects of noise. An attempt to add noise to the speed signal in the simulation was also made, resulting in the black lines in Figures 6.44 through 6.46.

The third set of simulations was taken using a set of coefficients generated by an optimization run beginning on the previous set of coefficients. The same objective function as the dynamometer case was used. The start point of Optimization 1 was Kp = 5.0, Ki = 1.0, and W = 1000.0, but unfortunately, real runs for this case were not performed.

In order to examine the effects of under-controlling the motor, an additional pair of runs were made with this configuration. As was previously mentioned, the controller was able to execute the task in less than 1.5 mS., and the task period was set to 2.5 mS. for the runs previously mentioned. Runs with the task

Proportional Gain	Integral Time Constant	Integral Limit	Figure Number	Page Number	Notes
1	1.0	0.0	6.41	86	Integration Disabled
2.5	1.0	0.0	6.42	86	Integration Disabled
10.0	1.0	0.0	6.43	87	Integration Disabled
25.0	1.0	0.0	6.44	87	Integration Disabled
100.0	1.0	0.0	6.45	88	Integration Disabled
250.0	1.0	0.0	6.46	88	Integration Disabled
1.0	0.66	1000.0	6.47	89	
1.0	0.10	100.0	6.48	89	
1.0	0.25	100.0	6.49	90	Optimization 2 Start Point
10.0	1.00	500.0	6.50	90	
5.0	0.20	500.0	6.51	91	
22.2	42.0	3478	6.52	92	Optimization 1
9.11	1.610	5.926	6.53	92	Optimization 2
9.11	1.610	5.926	6.54	93	Task Period = 10 ms.
9.11	1.610	5.926	6.55	93	Task Period = 100 ms.
25.0	1.0	0.0	6.56	94	Noise Injection, Kp = 25.0
100.0	1.0	0.0	6.57	94	Noise Injection, Kp = 100.0
250.0	1.0	0.0	6.58	95	Noise Injection, Kp = 250.0

Table 4.4: Table of PI Controller Parameters

period set to 10 mS. and 100 mS. were also performed.

A set of noise-injection runs was also created for this test case, with a smaller magnitude noise for the quadrature encoder. The results are shown in Figures 6.56 through 6.58.

4.7.8.3 Results Due to the reduced rotating mass and lack of elastic belt, the motor and speed encoder combination is a closer match to the simulated induction motor component then the motor-and-dyno combination. As a result, the parameters for this configuration were able to be set much more accurately, and the simulation curves matched the real runs much more closely.

Figures 6.41 through 6.51 in the appendix all show that the speed curves of the real runs match those of the simulations much more closely then in the dyno configuration. In particular, Figure 4.17 shows the performance of the system under the same conditions of Figure 4.14, outlining the better accuracy of the speed sensor configuration over the dyno-and-belt configuration.

Optimization for this configuration was similarly useful for generating better performance of the

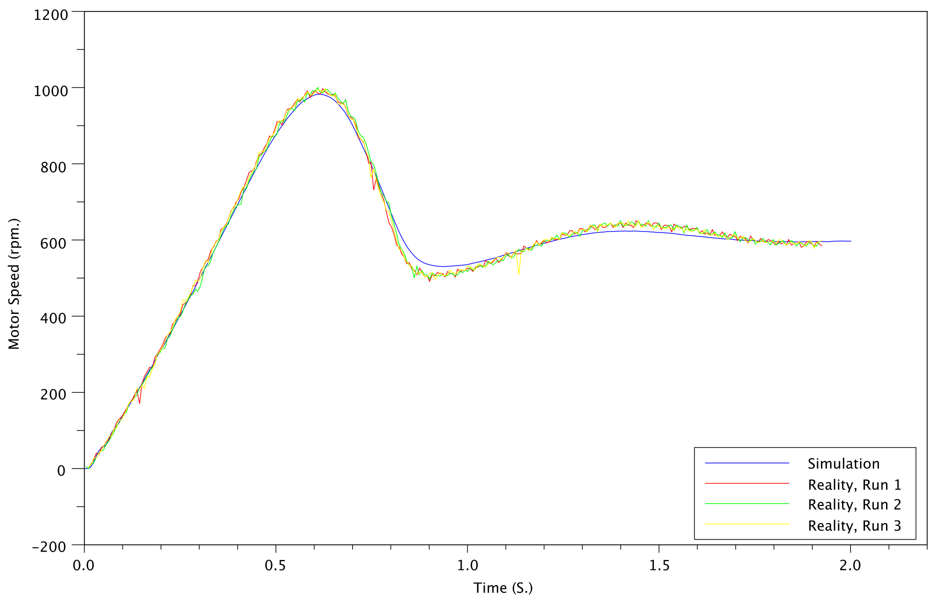


Figure 4.17: Quadrature Encoder, P = 1.0, K = 0.10, W = 100.

control. Figure 4.18 shows the performance of an arbitrary set of parameters. The system under these parameters exhibits an overshoot to 800 rpm. on start. The after-optimization performance depicted in Figure 4.19 shows much better speed control, going from rest to 600 rpm., and basically staying there.

Again, noise injection into the simulated speed signal had similar effects then in the dyno configuration above. The noise generated was less in magnitude then that in the dynamometer case, due to the belief that the quadrature encoder would provide less noise then the dynamometer. However, the motor exhibited approximately the same sort of fluctuations whether connected to the dyno or the encoder.

Under-controlling the motor by lengthening the task period could also be modelled accurately by the simulation. Figure 4.20 shows the simulated and real runs with the task period set to 100 mS., which does not provide enough control for the motor. Although the simulated and actual curves in the graph are not identical, they exhibit the same characteristics of overshoot and rapid speed fluctuation. The simulation could be used to investigate the minimum amount of processor usage dedicated to the control loop for successful operation of the controller.

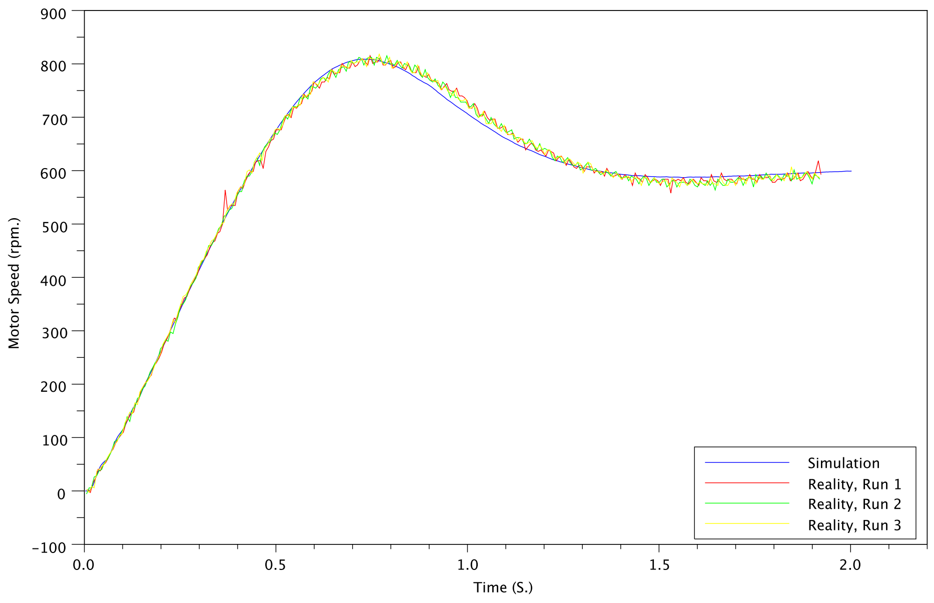


Figure 4.18: Quadrature Encoder, Before Optimization

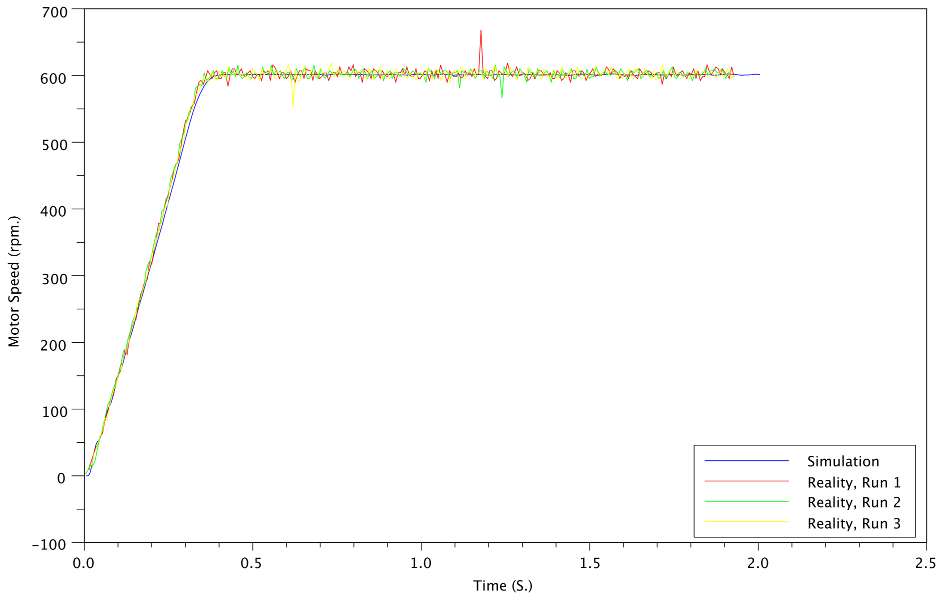


Figure 4.19: Quadrature Encoder, After Optimization

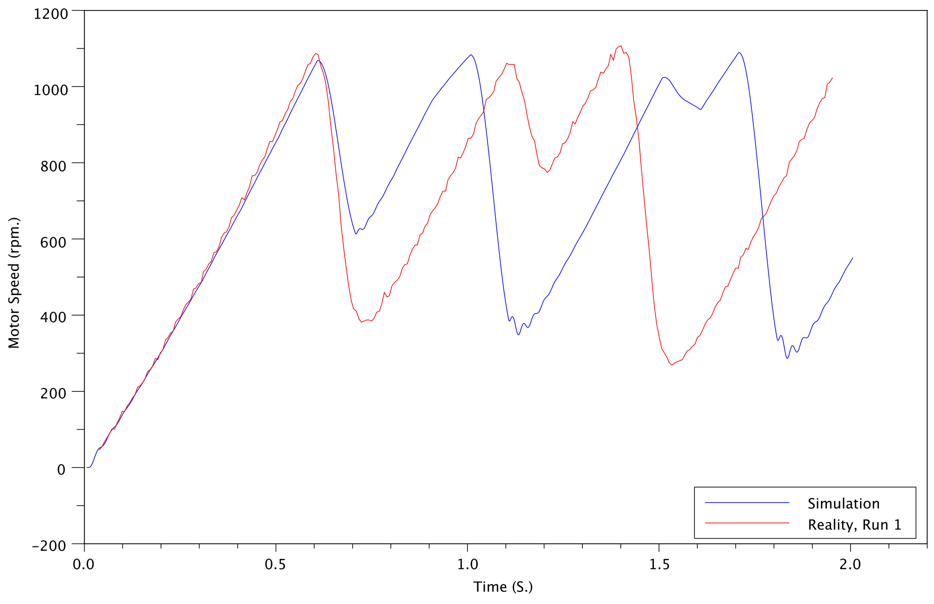


Figure 4.20: Quadrature Encoder, Task Period = 100.0 mS.

5 Conclusions, Contributions, and Recommendations for Future Work

5.1 Conclusions and Contributions

Accurate software-in-loop and processor-in-loop simulations are possible with PSCAD / EMTDC. Even though the controller model implemented is somewhat restricted and simplistic, the resulting simulation is accurate enough to predict the behaviour of a induction motor and speed control system within a reasonable tolerance.

With the limited access model of the processor used, processor-in-loop co-simulations can be completed in a time frame comprable to that of a similar single-platform simulation.

Multiple controller types can be used interchangably in processor-in-loop simulations, as long as the parameters for the peripherals of the specific controller are made available to PSCAD / EMTDC.

Optimization is also possible on software-in-loop and processor-in-loop simulations. Writing the controller processing routine to reference arbitrary parameters, and then allowing the optimization process to modify those parameters in repeated simulations makes modifying the action of a microcontroller in an optimizing simulation possible.

Simulation accuracy is reduced where the components of the simulation don't accurately reflect the behaviour of the real components in the system being simulated. The two main identified examples of this are the dyno-and-belt combination not being accurately represented by the PSCAD induction motor component and sensor noise present in the real system but not modelled accurately in the simulation. Replacing the dyno with the quadrature encoder for sensing speed allowed the real system to approach the ideal behaviour of the induction motor as represented in the simulator component. While the investigation of noise in the system was minimal, the simulation was able to more accurately mimic the behaviour of the noisy system with the addition of simulated noise.

With respect to the noise present in the sensors, non-real-time simulation is inferior to comprehensive hardware-in-loop testing, as hardware-in-loop testing including the dynamometer would show this behaviour. However, with sensor noise accurately accounted for, non-real-time simulation can remain accurate.

Between software-in-loop and processor-in-loop simulations, processor-in-loop simulations are more accurate due to having an accurate representation of the task delay. However, if a reasonable task delay can be assigned to each running of the task, the results of the two simulations become indistinguishable.

5.2 Recommendations for Future Work

The following section describes some ways in which the existing component simulation could be developed with future effort.

5.2.1 More Applications

The first recommendation for future work is to use the controller component to implement new controller applications. Any PSCAD simulation having a controller component which is to be physically realized could be modified to use the controller component.

In order to build confidence, applications requiring the least amount of controller performance should be attempted first, followed by more demanding applications. In particular, applications with slow event times might be attempted, such as battery charging controllers.

5.2.2 Controller Capability Expansion

The controller model on which the component is based could be expanded. The current model somewhat restrictive in that it doesn't allow some common controller functionality to be simulated. With additional development, more capable controllers could be integrated with the component, allowing more capable designs to be realized.

One simple upgrade would be to include a DSP core in the existing Nios II controller design. Other improvements could be including external interrupts or high-speed data sampling via direct memory access (DMA) in the simplified controller model.

5.2.3 Multi-Controller Simulations

Currently, the controller component is only able to be instantiated once in any simulation. Multiple instances of the controller would cause the single copy of the controller structure to be overwritten, spoiling the resulting simulation. Simulations of systems featuring multiple controllers, such as wind farms, are currently impossible with this component.

For SIL simulations, the component could be modified to use the storage arrays available in PSCAD for storing and referencing copies of the controller structure. In this way, multiple virtual copies of the controller could each work with its own data.

For PIL simulations, the controller would need to be able to switch context for the simulation. An alternative method for a simulation with a small number of controllers would be to use multiple serial ports and copies of hardware.

5.2.4 PSCAD-to-FreeScale CodeWarrior Emulator Co-simulation

A prototype communications link using Microsoft Component Object Model, or COM, was written between PSCAD and the True-Time Simulator and Real-Time Debugger included in the FreeScale Code-Warrior development environment. Messages and data could be passed through the link, and the message types available allowed sufficient control for injecting data into the emulator, extracting results from the emulator, and synchronizing the timesteps of the two simulators to run a complete controller simulation from within PSCAD.

Unfortunately, the FreeScale emulator wasn't able to immediately recognize the arrival of link messages from PSCAD, resulting in a very slow simulation. The actual speed of the link could be much higher, but could only be realized by continuously generating events on the emulator window, such as continuously resizing the window with the mouse. Due to this, and the fact that pursuing this development would result in a manufacturer-specific solution to the problem, the development was abandoned.

Using an emulator in this fashion would allow the simplified controller model to be eliminated, and the representation of the controller processing in the resulting PSCAD / emulator co-simulation could be as accurate as the emulator supplied.

5.2.5 PSCAD-to-ModelSim Co-simulation Improvements

Since the 3-phase PWM peripheral for the Nios II controller was developed using a hardware description language, or HDL, the ModelSim HDL simulator was investigated as a way of providing PSCAD with accurate simulated data regarding the peripheral.

PSCAD and ModelSim were integrated using shared memory and the DPI-C interface of SystemVerilog. In this way, instead of a C code approximation of the HDL component, the actual HDL component code was included directly into the PSCAD simulation. The co-simulation was able to work whether the simulation was software-in-loop or processor-in-loop.

Unfortunately, including ModelSim in the PSCAD simulation slowed the simulation down drastically. Although the results were accurate, ModelSim consumed 85% of the processor speed, making it somewhat impractical for the repeated simulations necessary for optimization. The simulation of the 3-phase PWM peripheral was much faster when approximated in C code. After verifying the operation of the C code approximation of the 3-phase PWM peripheral, this development was abandoned.

The current development of the PSCAD-to-ModelSim co-simulation was specific to the component being designed, but this doesn't have to be the case. Further development of the PSCAD-to-ModelSim co-simulation could use the Nios II bus as an interface point. This would allow various peripherals written for the Nios II to provide different types of switching to be quickly integrated to the simulation.

Another alternative for PSCAD / ModelSim co-simulations is to simulate the entire Nios II design in ModelSim. Although this would slow down the simulation, it would eliminate the need for hardware altogether. Additionally, by using ModelSim as a controller emulator, any processor design with an HDL description could be implemented in ModelSim.

5.2.6 HDL Simulation-Specific Peripheral ”Wrapper”

One drawback of the current system is all peripherals used by the controller need to be simulated from within the C code of the PSCAD / EMTDC component. The behaviour of the peripheral must be condensed into C code and included in the PSCAD / EMTDC component. This introduces opportunities for the behaviour of the actual peripheral and the peripheral simulation code to diverge.

An alternative to this for FPGA-based peripherals might be to create an HDL ”wrapper” for the switching peripheral. This wrapper would allow the switching peripheral to operate normally during live runs, accepting register values and manipulating the output switches as required.

However, during PIL simulation, the component would be disconnected from its output and clock circuit, run in synchronization with the PSCAD / EMTDC simulation, and have the resulting switching data transmitted back to the simulation. This development is similar to some work on simulating power systems on an FPGA [43].

The main drawback of this method is, to preserve the simulation accuracy, the switching info would have to be communicated to PSCAD / EMTDC more often then once every time the task runs.

5.2.7 Dyno and Belt Model Accuracy Improvement

The inability of the PSCAD induction motor component to accurately model the dynamometer and belt configuration raises the question of how to increase the accuracy of the simulation with regards to mechanical equipment. Creating a co-simulation with a physics simulation package such as Comsol Multiphysics, or developing a dynamic model of a power transmission belt and incorporating it into PSCAD are both possibilities.

References

[1] T. Garrity, “Innovation and trends for future electric power systems,” in *Power Systems Conference, 2009. PSC ’09.*, march 2009, pp. 1 –8.

[2] D. Howe, “Advanced electrical machines and actuators for new and emerging applications,” in *Electrical Machines and Systems, 2005. ICEMS 2005. Proceedings of the Eighth International Conference on*, vol. 1, sept. 2005, pp. 18 – 23 Vol. 1.

[3] A. J. Chongva and G. G. Chongva, “Personal observation of a small boy in an amusement park ride,” August 2011.

[4] E. K. P. Chong and S. H. Zak, ser. Wiley-Interscience series in discrete mathematics and optimization. Wiley, New York, 2001.

[5] M. Faruque, Y. Zhang, and V. Dinavahi, “Detailed modeling of cigre hvdc benchmark system using pscad/emtdc and psb/simulink,” *IEEE Transactions on Power Delivery*, vol. 21, no. 1, pp. 378 – 387, jan. 2006.

[6] P. Zumel, M. Garci anda Valderas, A. La andzaro, C. Lo andpez Ongil, and A. Barrado, “Co-simulation psim-modelsim oriented to digitally controlled switching power converters,” in *Control and Modeling for Power Electronics (COMPEL), 2010 IEEE 12th Workshop on*, june 2010, pp. 1 –7.

[7] N. Watson, J. Arrillaga, and I. of Electrical Engineers, *Power systems electromagnetic transients simulation*, ser. IEE power and energy series. Institution of Electrical Engineers, 2003. [Online]. Available: http://books.google.ca/books?id=b4TYkhpCb80C

[8] P. Liu, G. Wu, B. Sui, R. Li, and X. Cao, “Modeling lightning performance of transmission systems using pscad,” in *High Voltage Engineering and Application, 2008. ICHVE 2008. International Conference on*, nov. 2008, pp. 168 –171.

[9] A. Chevrefils and S. Filizadeh, “Modeling and transient simulation of an all-electric all-terrain vehicle (atv),” in *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, april 2007, pp. 401 –404.

[10] S. Filizadeh, M. Heidari, A. Mehrizi-Sani, J. Jatskevich, and J. Martinez, “Techniques for interfacing electromagnetic transient simulation programs with general mathematical tools ieee taskforce on inter-

facing techniques for simulation tools," *IEEE Transactions on Power Delivery*, vol. 23, no. 4, pp. 2610 –2622, oct. 2008.

[11] A. Gole, S. Filizadeh, R. Menzies, and P. Wilson, "Optimization-enabled electromagnetic transient simulation," *IEEE Transactions on Power Delivery*, vol. 20, no. 1, pp. 512 – 518, jan 2005.

[12] M. Loghi, T. Margaria, G. Pravadelli, and B. Steffen, "Dynamic and formal verification of embedded systems: A comparative survey," *International Journal of Parallel Programming*, vol. 33, pp. 585–611, 2005, 10.1007/s10766-005-8911-2. [Online]. Available: http://dx.doi.org/10.1007/s10766-005-8911-2

[13] S. Karimi, P. Poure, and S. Saadate, "An hil-based reconfigurable platform for design, implementation, and verification of electrical system digital controllers," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 4, pp. 1226 –1236, april 2010.

[14] J. Wegener and P. Kruse, "Search-based testing with in-the-loop systems," in *Search Based Software Engineering, 2009 1st International Symposium on*, may 2009, pp. 81 –84.

[15] J. Liu, J. Eker, J. W. Janneck, and E. A. Lee, "Realistic simulations of embedded control systems," in *Proc. Int. Federation of Automatic Control 15th World Congress*, 2002.

[16] L. Bui, S. Casoria, G. Morin, and J. Reeve, "Emtp tacs-fortran interface development for digital controls modeling," *Power Systems, IEEE Transactions on*, vol. 7, no. 1, pp. 314 –319, feb 1992.

[17] M. Cakmakci, Y. Li, and S. Liu, "Model-in-the-loop development for fuel cell vehicle," in *American Control Conference (ACC), 2011*, 29 2011-july 1 2011, pp. 2462 –2467.

[18] S. Filizadeh, A. Chevrefils, and D. Northcott, "Analysis and design of vehicular power systems using pscad/emtdc," in *Vehicle Power and Propulsion Conference, 2007. VPPC 2007. IEEE*, sept. 2007, pp. 463 –468.

[19] J. Bapiraju, J. Shenoy, K. Sheshadri, H. Khincha, and D. Thukaram, "Implementation of dsp based relaying with particular reference to effect of statcom on transmission line protection," in *Power System Technology, 2004. PowerCon 2004. 2004 International Conference on*, vol. 2, nov. 2004, pp. 1381 – 1385 Vol.2.

[20] E. Abiri, A. Rahmati, and A. Abrishamifar, "A sensorless and simple controller for vsc based hvdc systems," *Journal of Zhejiang University - Science A*, vol. 10, pp. 1824–1834, 2009, 10.1631/jzus.A0820504. [Online]. Available: http://dx.doi.org/10.1631/jzus.A0820504

[21] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, “Systematic embedded software generation from systemc,” in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, 2003, pp. 142 – 147.

[22] C. Xiang, M. Salem, T. Das, and X. Chen, “Real time software-in-the-loop simulation for control performance validation,” *SIMULATION*, vol. 84, no. 8-9, pp. 457–471, August/September 2008. [Online]. Available: http://sim.sagepub.com/content/84/8-9/457.abstract

[23] B. Kamiriski, K. Wejrzanowski, and W. Koczara, “An application of psim simulation software for rapid prototyping of dsp based power electronics control systems,” in *Power Electronics Specialists Conference, 2004. PESC 04. 2004 IEEE 35th Annual*, vol. 1, june 2004, pp. 336 – 341 Vol.1.

[24] D. Detjen, S. Schroder, and R. De Doncker, “Embedding dsp control algorithms in pspice,” *IEEE Transactions on Power Electronics*, vol. 18, no. 1, pp. 294 – 300, jan 2003.

[25] S. Demers, P. Gopalakrishnan, and L. Kant, “A generic solution to software-in-the-loop,” in *Military Communications Conference, 2007. MILCOM 2007. IEEE*, oct. 2007, pp. 1 –6.

[26] H. Min, Z. Guoqiang, Y. Hong, and T. Yafeng, “Processor-in-the-loop demonstration of coordination control algorithms for distributed spacecraft,” in *Information and Automation (ICIA), 2010 IEEE International Conference on*, june 2010, pp. 1008 –1011.

[27] M. Castoldi and M. Aguiar, “Simulation of dtc strategy in vhdl code for induction motor control,” in *Industrial Electronics, 2006 IEEE International Symposium on*, vol. 3, july 2006, pp. 2248 –2253.

[28] N. Gupta, S. Dubey, and S. Singh, “Pil based control algorithm for three-phase four-wire active filter for reactive and harmonic compensation under distorted supply,” in *Power Electronics, Drives and Energy Systems (PEDES) 2010 Power India, 2010 Joint International Conference on*, dec. 2010, pp. 1 –6.

[29] T. Qi and J. Sun, “Analog ic design for real-time simulation of power electronic circuits,” in *Control and Modeling for Power Electronics, 2008. COMPEL 2008. 11th Workshop on*, aug. 2008, pp. 1 –7.

[30] D. Word, R. Bednar, J. J. Zenor, and N. G. Hingorani, “High-speed real-time simulation for power electronic systems,” *SIMULATION*, vol. 84, no. 8-9, pp. 441–456, August/September 2008. [Online]. Available: http://sim.sagepub.com/content/84/8-9/441.abstract

[31] A. Bouscayrol, “Different types of hardware-in-the-loop simulation for electric drives,” in *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, 30 2008-july 2 2008, pp. 2146 –2151.

[32] J. Reeve and S. Lane-Smith, “Integration of real-time controls and computer programs for simulation of direct current transmission,” *Power Delivery, IEEE Transactions on*, vol. 5, no. 4, pp. 2047 –2053, oct 1990.

[33] Z. Jiang, R. Leonard, R. Dougal, H. Figueroa, and A. Monti, “Processor-in-the-loop simulation, real-time hardware-in-the-loop testing, and hardware validation of a digitally-controlled, fuel-cell powered battery-charging station,” in *Power Electronics Specialists Conference, 2004. PESC 04. 2004 IEEE 35th Annual*, vol. 3, june 2004, pp. 2251 – 2257 Vol.3.

[34] M. Technology, *PIC32MX3XX/4XX Family Data Sheet - 64/100-Pin General Purpose and USB 32-bit Flash Microcontrollers*, 2009.

[35] A. Corporation, *8-bit AVR Microcontroller with 64K Bytes In-System Programmable Flash (ATmega64 / ATmega64L)*, 2009.

[36] ——, *8-bit AVR Microcontroller with 16/32/64/128K Bytes In-System Programmable Flash (ATmega164A / ATmega164PA / ATmega324A / ATmega324PA / ATmega644A / ATmega644PA / ATmega1284A / ATmega1284PA)*, 2010.

[37] F. Semiconductor, *DRM115: 3-Phase AC Induction Motor Control with PFC Using MC9S08MP16*, nov. 2009.

[38] J. E. Volder, “The cordic trigonometric computing technique,” *Electronic Computers, IRE Transactions on*, vol. EC-8, no. 3, pp. 330 –334, sept. 1959.

[39] M. Fathallah, J. Chante, F. Calmon, and M. El-husseini, “Design of an optimized ic for control algorithms of ac machines: system testing and application,” in *Industry Applications Conference, 2001. Thirty-Sixth IAS Annual Meeting. Conference Record of the 2001 IEEE*, vol. 1, sep-4 oct 2001, pp. 103 –109 vol.1.

[40] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, ser. Addison-Wesley Series in Electrical and Computer Engineering: Control Engineering. Addison-Wesley Publishing Company, 1994.

[41] A. E. Fitzgerald, C. J. Kingsley, and S. D. Umans, ser. McGraw-Hill series in electrical engineering. Power and energy. McGraw-Hill, New York, 2003.

[42] B. Liang, B. S. Payne, A. D. Ball, and S. D. Iwnicki, "Simulation and fault detection of three-phase induction motors," *Mathematics and Computers in Simulation*, vol. 61, no. 1, pp. 1 – 15, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378475402000642

[43] O. Luci anda, I. Urriza, L. Barraga andn, D. Navarro, O. Jime andnez, and J. Burdi ando, "Real-time fpga-based hardware-in-the-loop simulation test bench applied to multiple-output power converters," *IEEE Transactions on Industry Applications*, vol. 47, no. 2, pp. 853 –860, march-april 2011.

6 Appendices

6.1 C Structure

```
/*******************************************************/
/* Controller.h - PSCAD-to-Controller process file. */
/*******************************************************/
/* by Greg Chongva. */
/* 611 9122         */

#ifndef INCLUDE_PSCAD_Controller
#define INCLUDE_PSCAD_Controller

typedef struct
{
        uint16_t uLo, uHi;
        real rLo, rHi;
} Conversion;

/* PSCAD_Controller structure. */
typedef struct
{
        /* Timing data. */
        uint32_t uTaskPeriod;           /* Period of task rate (uS). */
        uint32_t uTaskDelay;            /* time task took to run (uS). */


        /* Input Parameters. */
        uint8_t uDICount;               /* Digital input channel count. */

        uint8_t uAICount;               /* Analog input channel count. */
        Conversion cAI;                 /* Analog input conversion data. */
        uint32_t uAIDelay;              /* Analog input delay. (us) */

        /* Process Parameters. */
        uint8_t uParamCount;            /* number of arbitrary parameters. */

        /* Output Parameters. */
        uint8_t uDOCount;               /* Digital output channel count. */

        uint8_t uPWMCount;              /* Number of PWM channels. */
        Conversion cPWM;                /* PWM conversion data. */
        real rPWMClockPeriod;           /* Period of PWM clock (uS). */
        uint8_t uPWMResolution;         /* Resolution of PWM. (bits) */

        uint8_t uPWM3Count;             /* Number of PWM channels. */
        Conversion cPWM3;               /* 3-phase PWM amplitude conversion data. */
        real rPWM3ClockPeriod;          /* Period of PWM3 clock (uS). */
        uint8_t uPWM3Resolution;        /* Resolution of PWM3. (bits) */


        /* Input Channel Data. */
        uint8_t uDI;                    /* Digital input values. */
        uint16_t uAIAmp[4];             /* Analog input values. */

        /* Process Data. */
        real rParams[8];                /* Parameter data. */

        /* Output Channel data. */
        uint8_t uDO;                    /* Digital output values. */

        uint16_t uPWMAmp[4];            /* Value of PWM channels. */

        int16_t iPWM3Amp[4];            /* 3-phase PWM Amplitude. */
        int16_t iPWM3Freq[4];           /* 3-phase PWM Frequency. */

#ifdef PLATFORM_PSCAD
        /* Simulation-only structures. */
        PSCADTimerState *ptsTimers;
        PSCAD3PhaseState *p3psTimers;
#endif
} PSCAD_Controller;

/* One structure,only. */
extern volatile PSCAD_Controller pcController;

//
// Prototypes.
//

int PSCAD_main(void);
void timed_controller_process(void);

#endif
```

6.2 C Code for Induction Motor Control

```
/*****************************************************/
/* PSCAD_AC_Case.c - PSCAD-to-Controller process file. */
/*****************************************************/

/* by Greg Chongva. */
/* 611 9122        */

#include "Platform.h"
#include "Controller.h"
#include "Case.h"
#include "Convert.h"

#include "PI_Control.h"

/* PI Control Block Structure. */
struct PIBlock pidbOne;

/* Dynamically calculate maximum frequency based on controller parameters. */
real rMaxFreq = 0.0, rTime;
int16_t iPWM3FreqOld, iPWM3AmpOld;

/* Pre-run synchronization... yeah... */
void controller_reset(void)
{
        rTime = 0.0;

        /* GGC: June 18, 2010 - set digital outputs off. */
        pcController.uDI = pcController.uDO = 0x00;

        /* Initialize PI Blocks. */
#define MINAMP 0.333
#define MAXAMP 1.00
#define MAXFREQSLOPE 60.0
#define MAXFREQ 70.0

        pidbOne.rKProportional = pcController.rParams[0];
        pidbOne.rIntegralTimeConstant = pcController.rParams[1];
        pidbOne.rKIntegralLimit = pcController.rParams[2];
        pidbOne.rOutMax = 900;
        pidbOne.rOutMin = -900.0;
        pidbOne.rErrorLast = 0.0;
        pidbOne.rIntegralValue = 0.0;

        /* Initialize PWM3 channel. */
        pcController.iPWM3Freq[0] = 0;
        pcController.iPWM3Amp[0] = 0;

        /* Max. 1/4 revolution (simulated, Nios II hardware, too.) */
        /* PWM up and down ramps mean resolution counts twice. */
#ifdef PLATFORM_PSCAD
        rMaxFreq = PWM3_SINE_FREQ_MAX;

        printf("CaseAC.c: uTaskPeriod = %ld uS.\n", pcController.uTaskPeriod);
        printf("CaseAC.c: PWM3 period = %10.8f s, res. = %d bits.\n",
                pcController.rPWM3ClockPeriod, pcController.uPWM3Resolution);
        printf("CaseAC.c: rMaxFreq = %6.3f Hz.\n", rMaxFreq);
#else
        rMaxFreq = PWM3_SINE_FREQ_MAX(PWM3_CLOCKSCALE);
#endif

        iPWM3FreqOld = 0; iPWM3AmpOld = 0;

        return;
}

#define AICOUNT 2

void controller_process(void)
{
        uint8_t x;

        real rAI[AICOUNT];
        real rError;
        real rNode;                     /* Intermediate node value. */
        real rFreq, rAmp;       /* Outputs. */

        rTime += (real)pcController.uTaskPeriod / 1e6f;


        //
        // Analog Input Translation.
        //

        /* Ch. 0 = Motor Speed, 500 rpm. / V. */
        rAI[0] = DToA(&pcController.cAI, pcController.uAIAmp[0]) * 500.0;

        /* Ch. 1 = Requested Speed, 300 rpm. / V. */
        rAI[1] = DToA(&pcController.cAI, pcController.uAIAmp[1]) * 300.0;


        //
        // Calculations.
        //

        /* Error signal. */
        rError = rAI[1] - rAI[0];

        /* Control signal. */
        rNode = piController(&pcController, &pidbOne, rError);
        /* Frequency (Hz.) */
        rFreq = 0.03333 * (rNode + rAI[0]);

        /* Voltage Magnitude (normalized and limited). */
        // GGC: Aug. 17, 2011 - move curve to constant v/f line.
        if (rFreq <= 0.0)
        {
                rFreq = 0.0;
                rAmp = MINAMP;
        }
        else if (rFreq <= ((MINAMP * MAXFREQSLOPE) / MAXAMP))
        {
                rAmp = MINAMP;
        }
        else if (rFreq < MAXFREQSLOPE)
                rAmp = rFreq * MAXAMP / MAXFREQSLOPE;
        else if (rFreq < MAXFREQ)
        {
                rAmp = MAXAMP;
        }
        else
        {
                rFreq = MAXFREQ;
                rAmp = MAXAMP;
        }

        // Bounds checking.
        if (rAmp > 1.0)
                rAmp = 1.0;

        if (rFreq > rMaxFreq)
                rFreq = rMaxFreq;


        //
        // 3-Phase PWM Output Translation.
        //

        /* Max. Speed = rMaxFreq Hz. = 0x7fff ticks. */
        pcController.iPWM3Freq[0] =
                (uint16_t)((rFreq / rMaxFreq) * (real)0x7fff);

        /* Max. Normalized Amp. = 1.0. (by definition...) */
        pcController.iPWM3Amp[0] =
                (uint16_t)(rAmp * (real)0x7fff);

        return;
}
```

6.3 Dynamometer Speed Curves

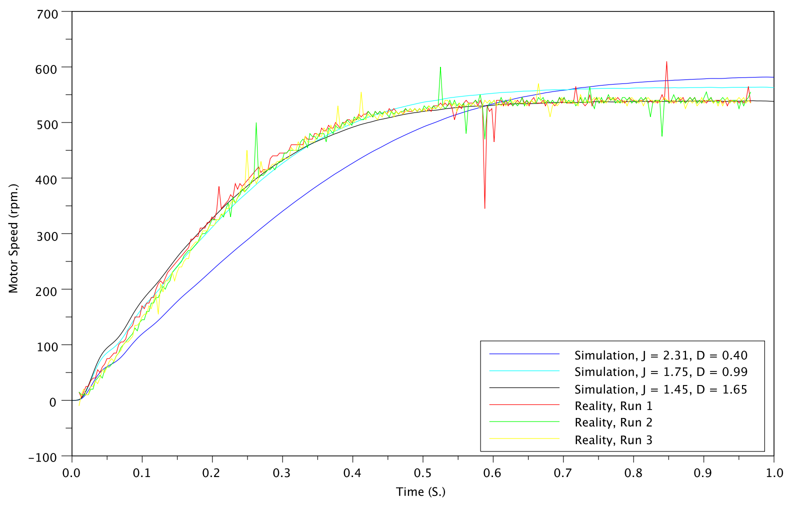


Figure 6.21: Dynamometer, Fixed 20 Hz. Signal

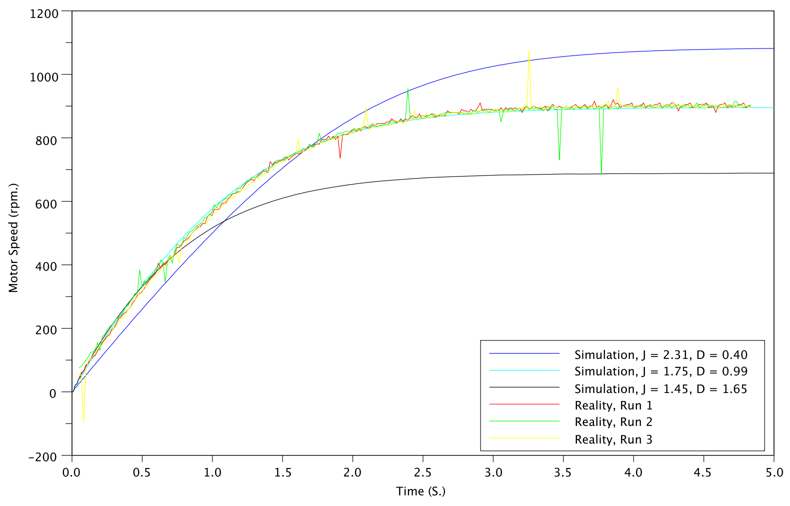


Figure 6.22: Dynamometer, Fixed 40 Hz. Signal

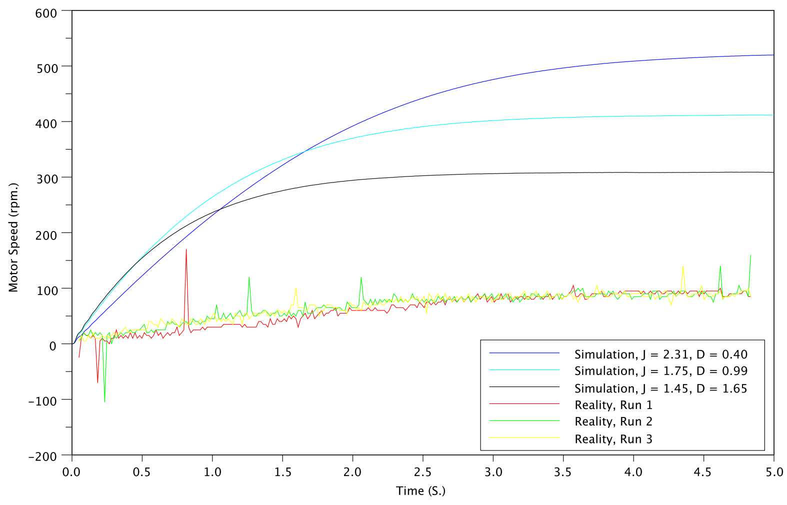


Figure 6.23: Dynamometer, Kp = 1.0

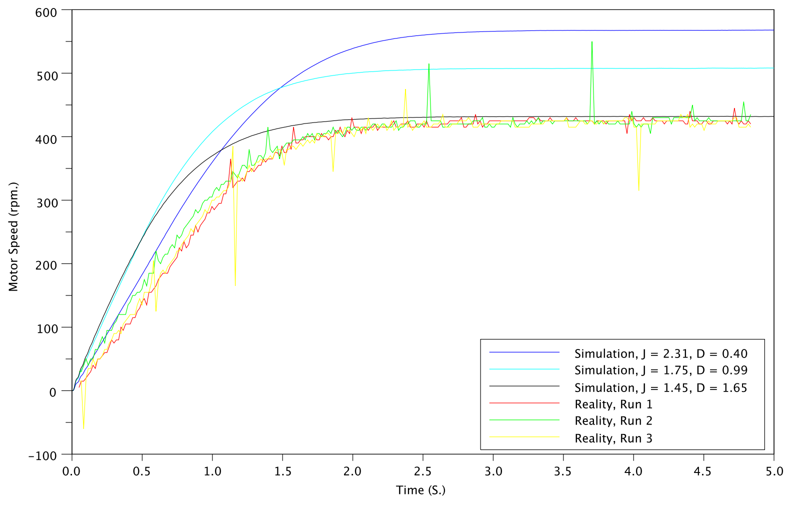


Figure 6.24: Dynamometer, Kp = 2.5

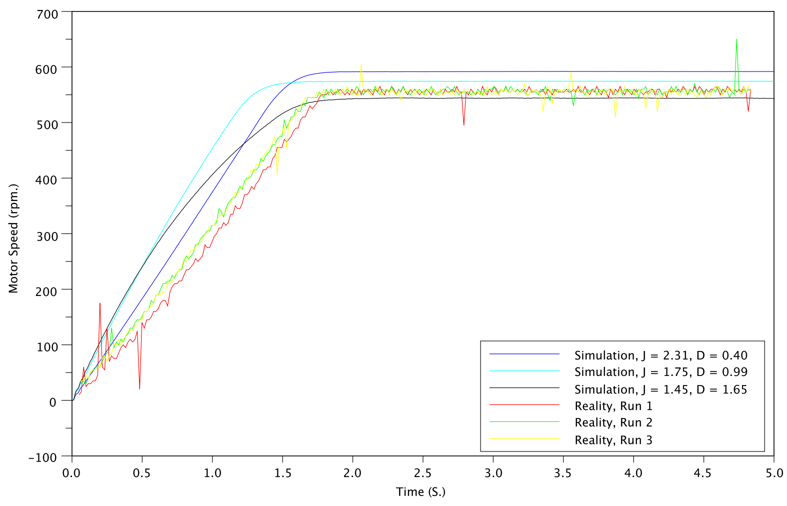


Figure 6.25: Dynamometer, Kp = 10.0

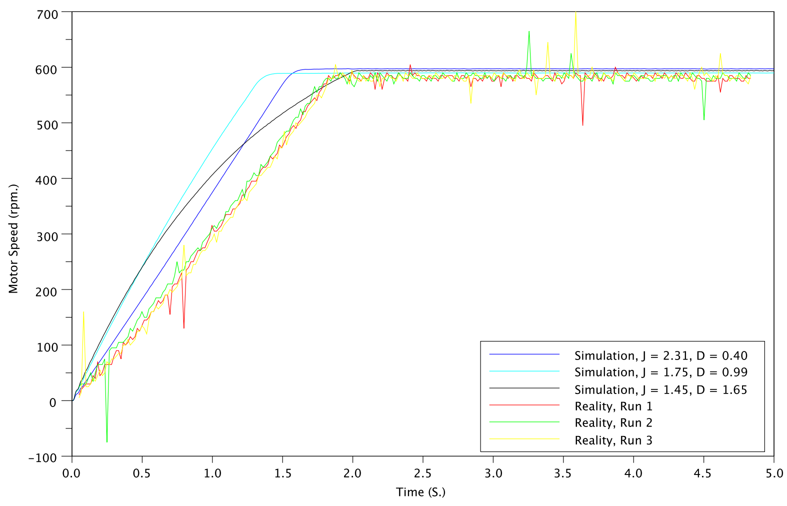


Figure 6.26: Dynamometer, Kp = 25.0

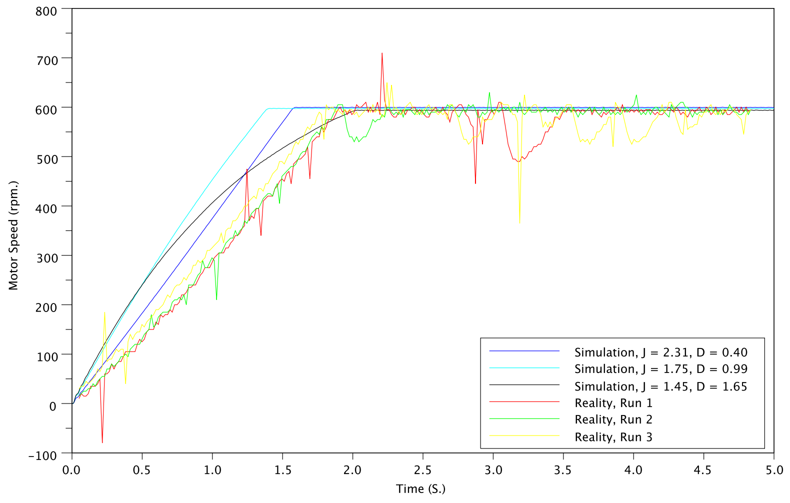


Figure 6.27: Dynamometer, Kp = 100.0

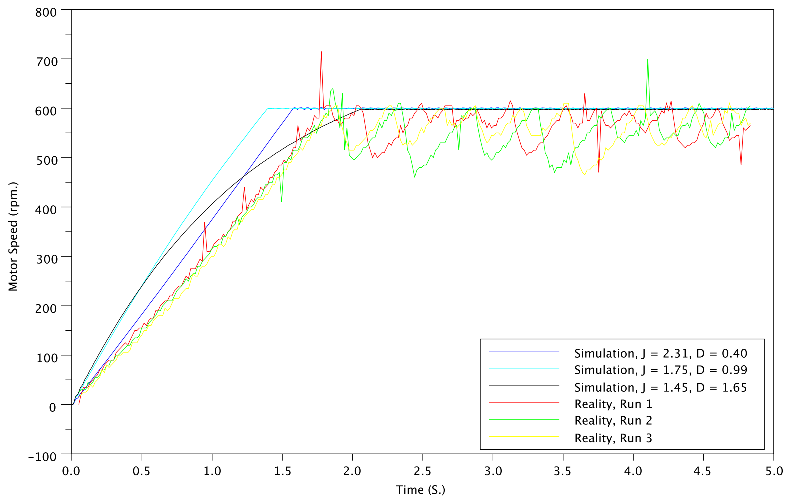


Figure 6.28: Dynamometer, Kp = 250.0

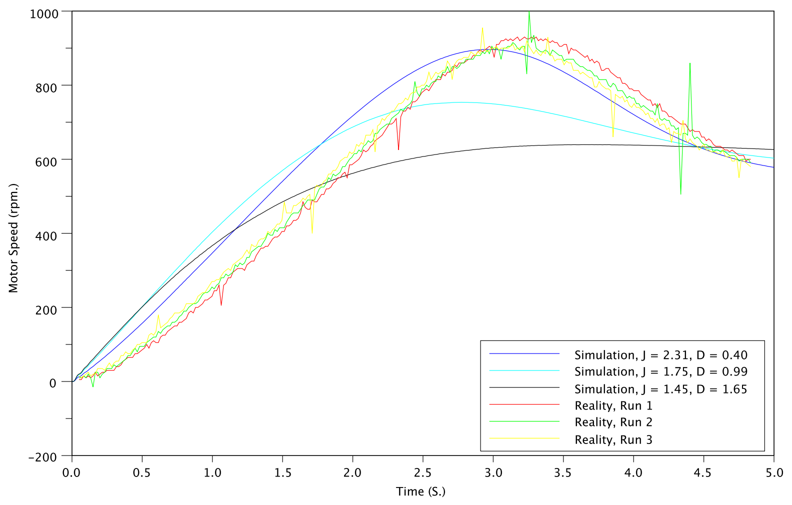


Figure 6.29: Dynamometer, P = 1.0, K = 0.66, W = 1000.

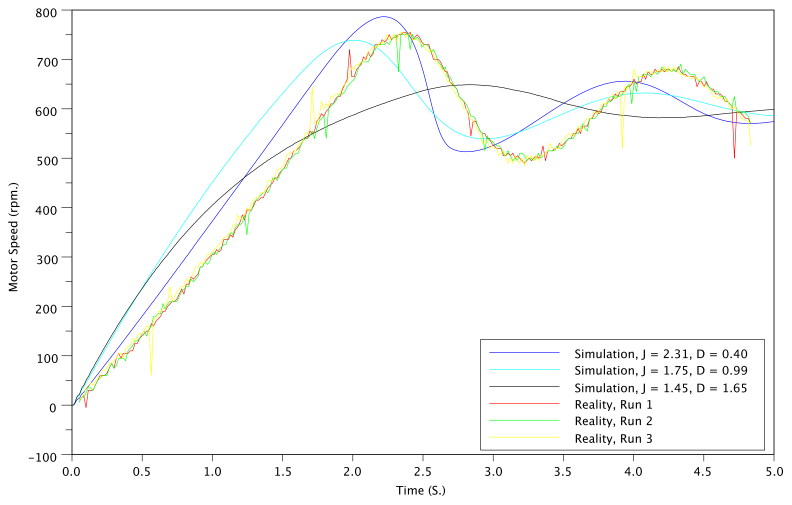


Figure 6.30: Dynamometer, P = 1.0, K = 0.10, W = 100.

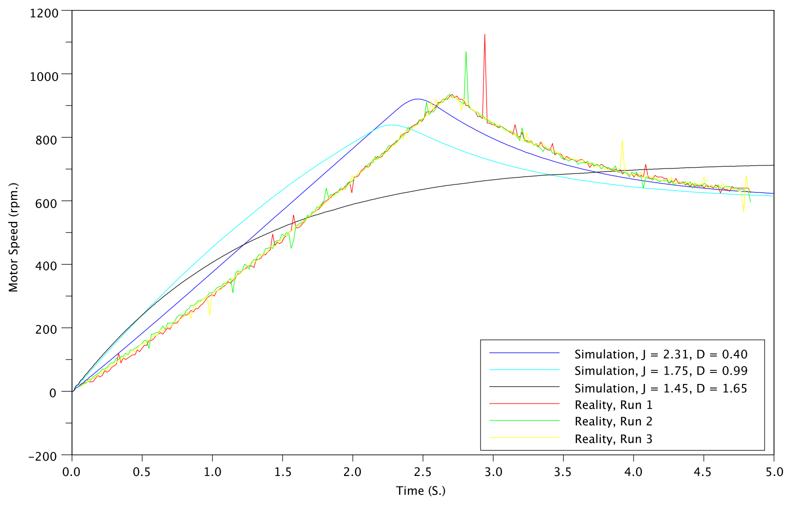


Figure 6.31: Dynamometer, P = 10.0, K = 1.0, W = 500.

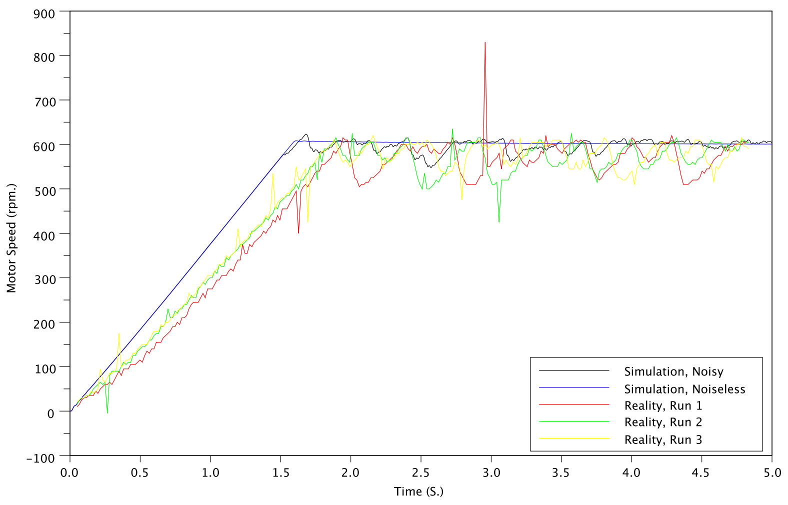


Figure 6.32: Dynamometer, Optimization 1

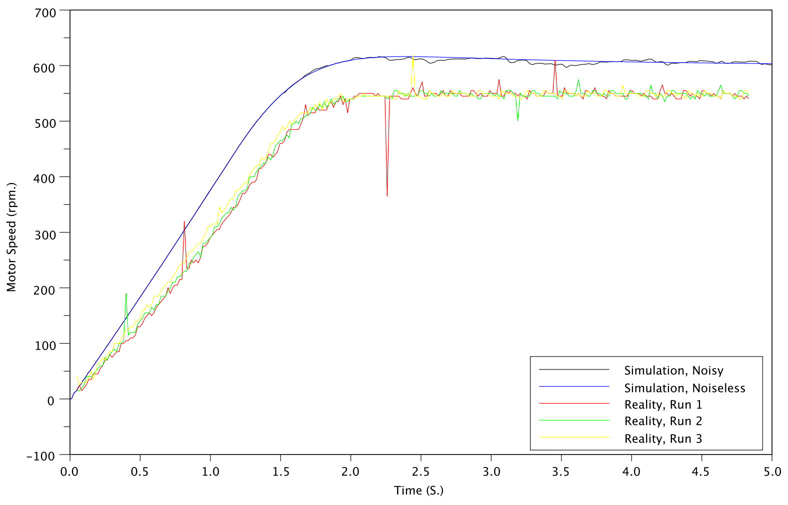


Figure 6.33: Dynamometer, Optimization 2

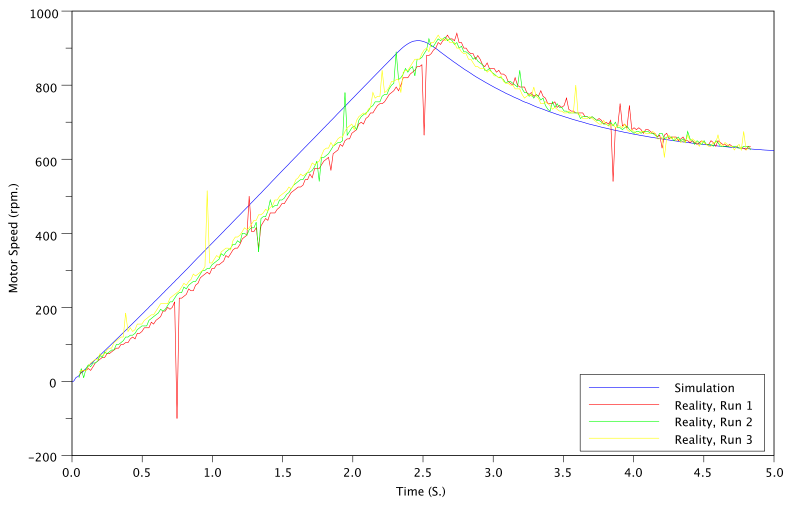


Figure 6.34: Dynamometer, Task Period = 2.5 mS.

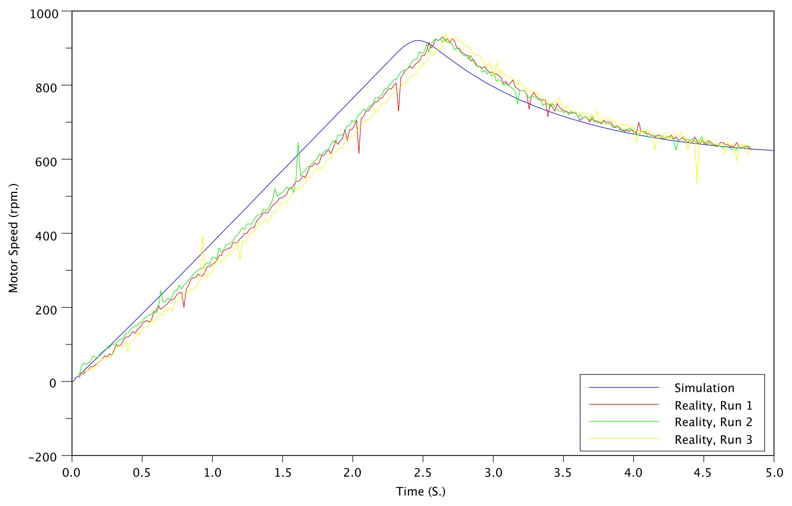


Figure 6.35: Dynamometer, Task Period = 5.0 mS.

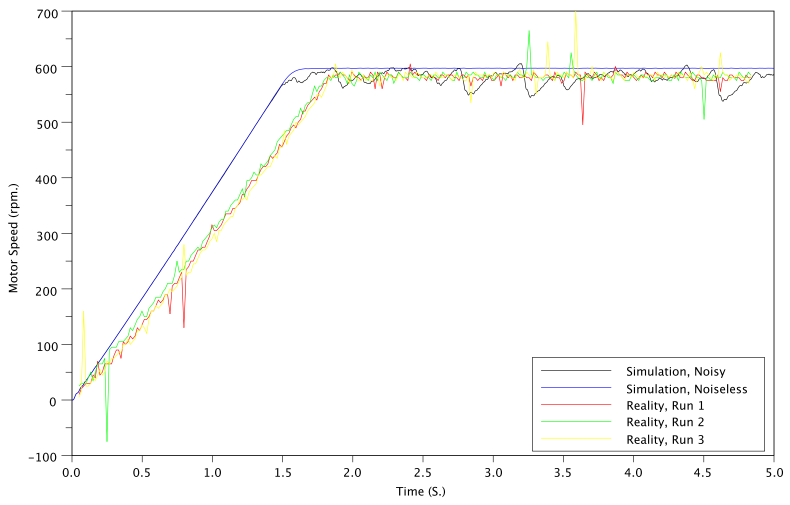


Figure 6.36: Dynamometer, Kp = 25.0, Noise Injection.

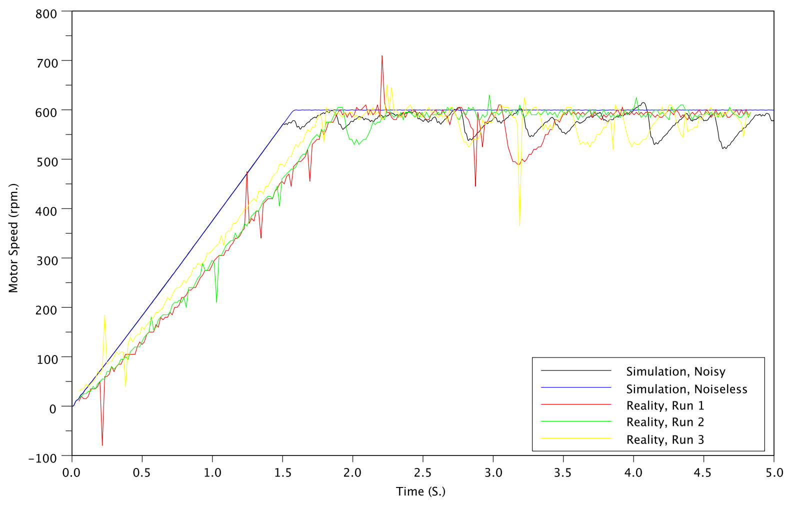


Figure 6.37: Dynamometer, Kp = 100.0, Noise Injection.

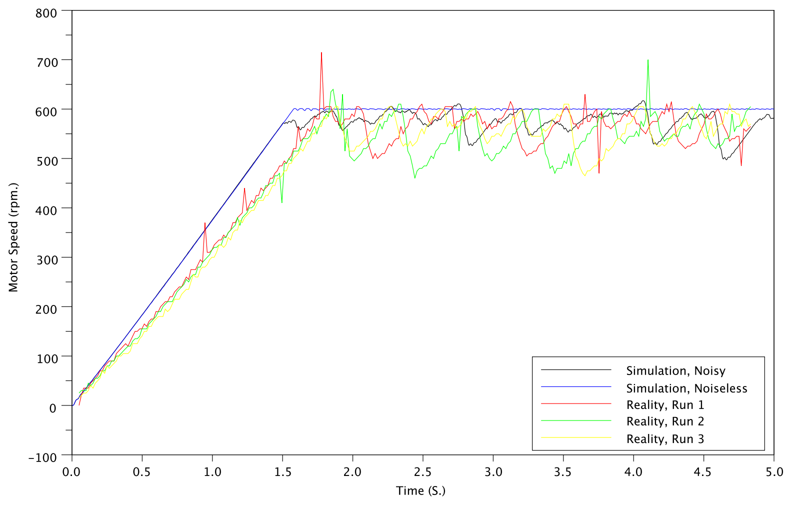


Figure 6.38: Dynamometer, Kp = 250.0, Noise Injection.

6.4 Quadrature Speed Encoder Curves

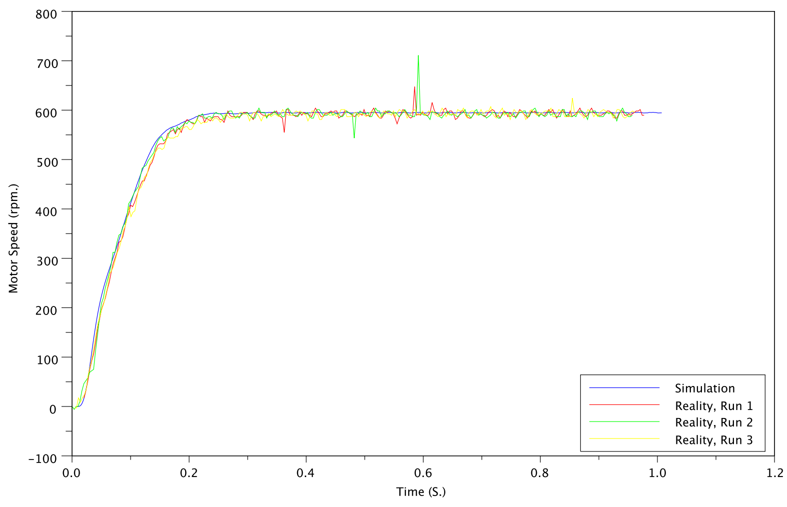


Figure 6.39: Quadrature Encoder, Fixed 20 Hz. Signal

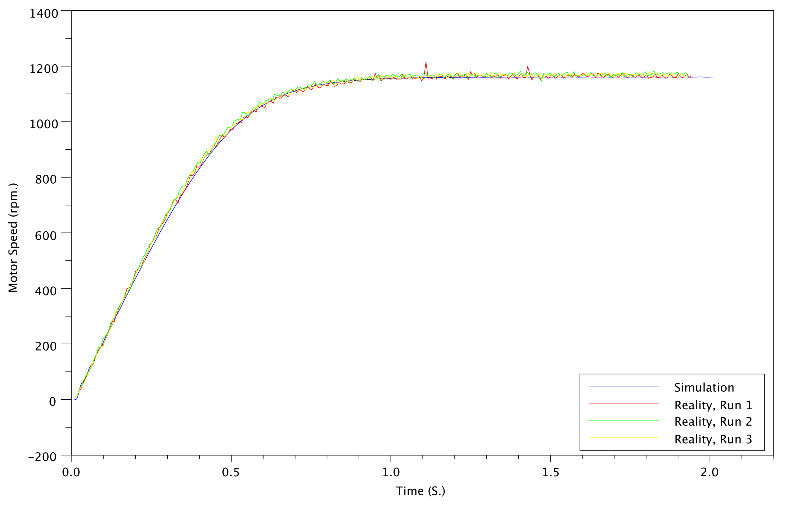


Figure 6.40: Quadrature Encoder, Fixed 40 Hz. Signal

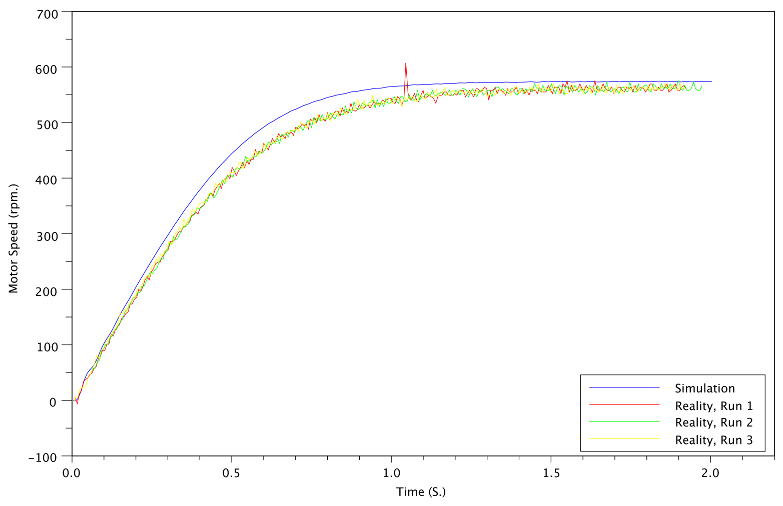


Figure 6.41: Quadrature Encoder, Kp = 1.0

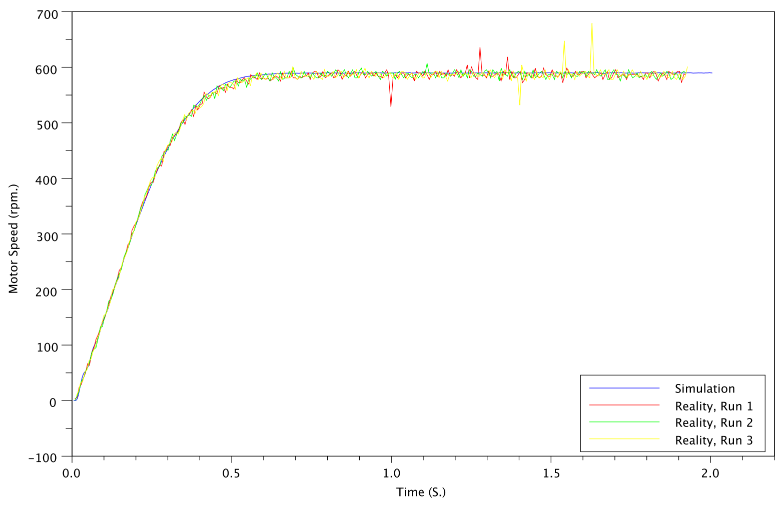


Figure 6.42: Quadrature Encoder, Kp = 2.5

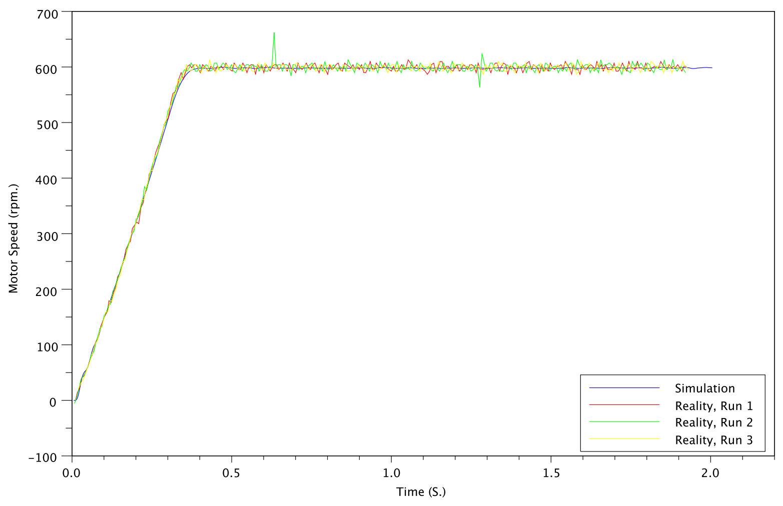


Figure 6.43: Quadrature Encoder, Kp = 10.0

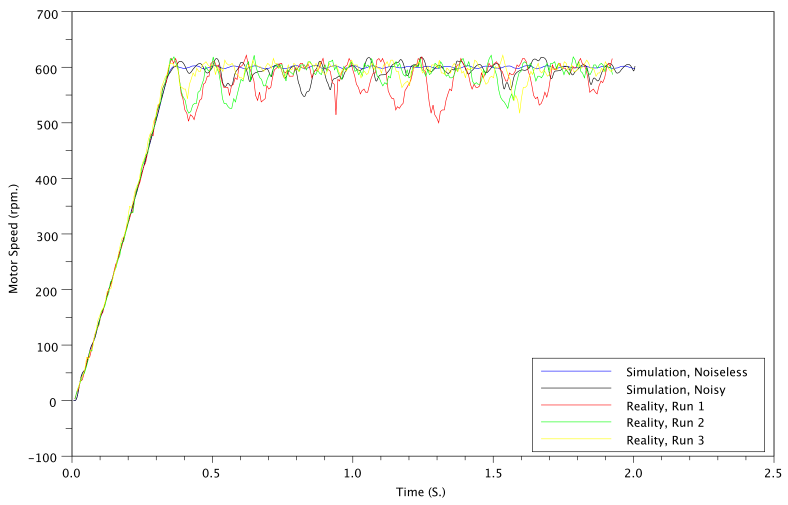


Figure 6.44: Quadrature Encoder, Kp = 25.0

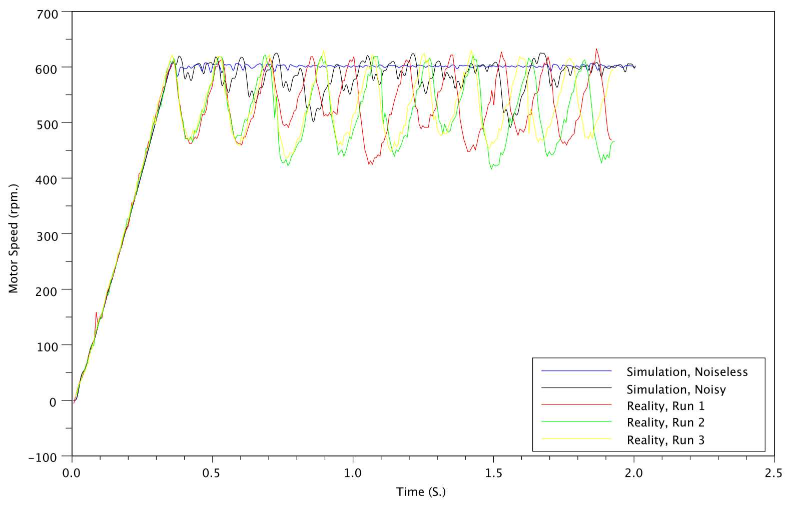


Figure 6.45: Quadrature Encoder, Kp = 100.0

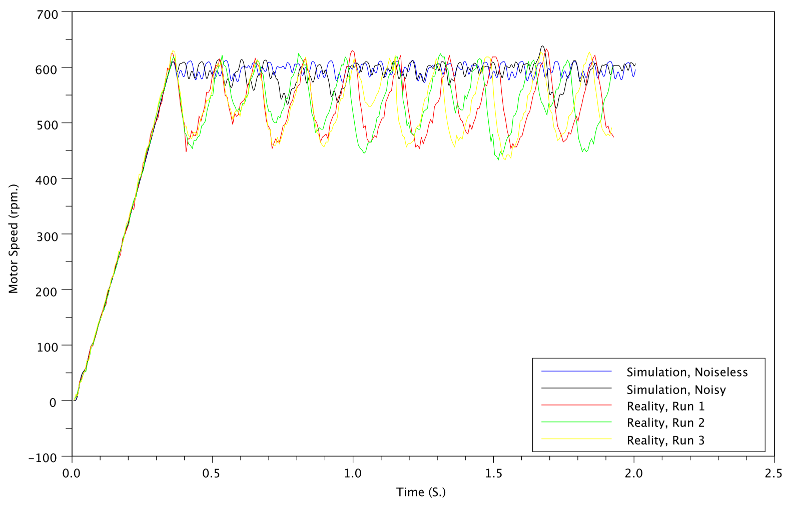


Figure 6.46: Quadrature Encoder, Kp = 250.0

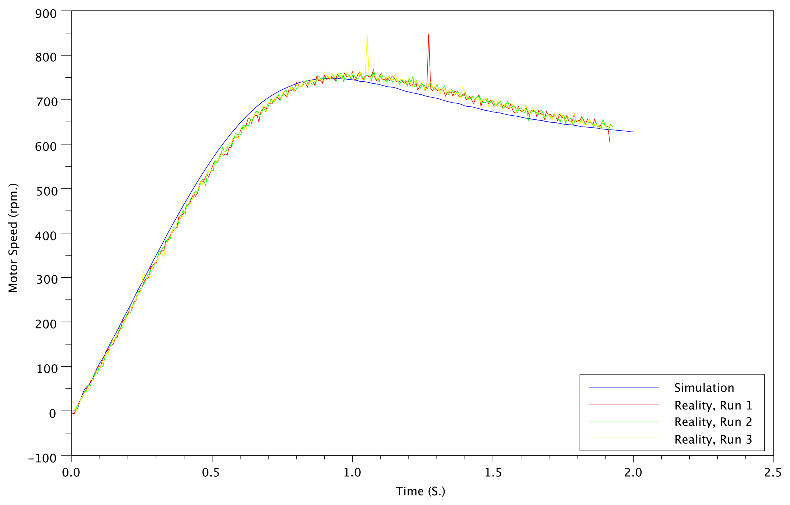


Figure 6.47: Quadrature Encoder, P = 1.0, K = 0.66, W = 1000.

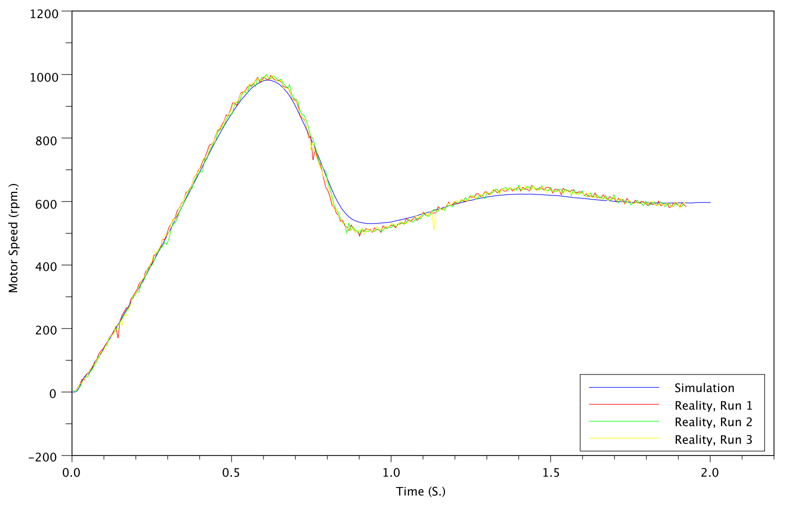


Figure 6.48: Quadrature Encoder, P = 1.0, K = 0.10, W = 100.

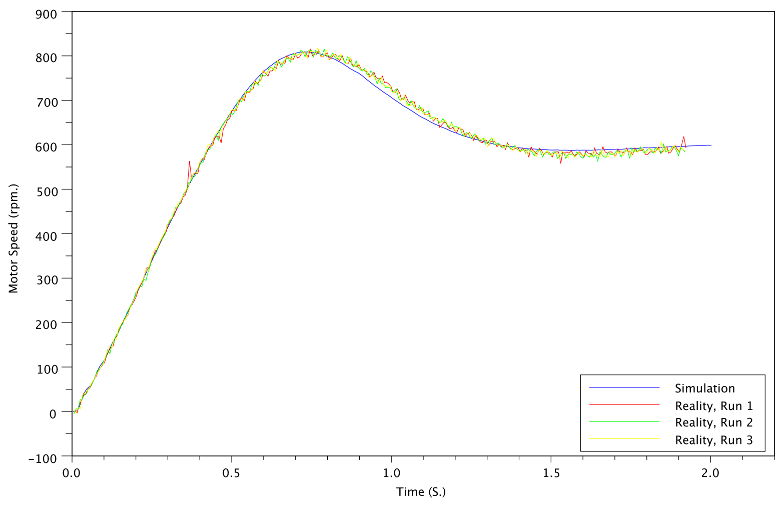


Figure 6.49: Quadrature Encoder, P = 1.0, K = 0.25, W = 100.

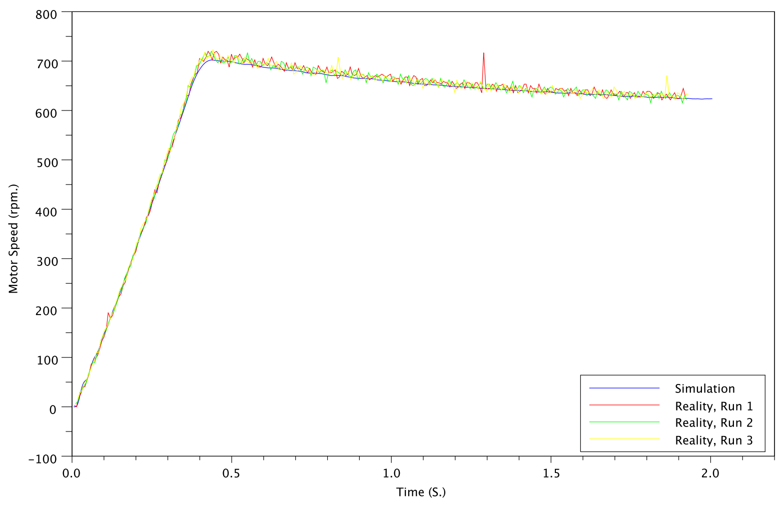


Figure 6.50: Quadrature Encoder, P = 10.0, K = 1.0, W = 500.

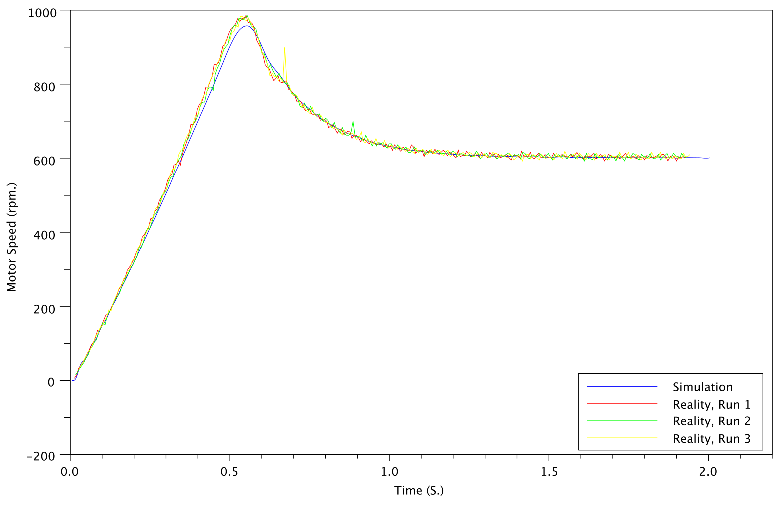


Figure 6.51: Quadrature Encoder, P = 5.0, K = 0.20, W = 500.

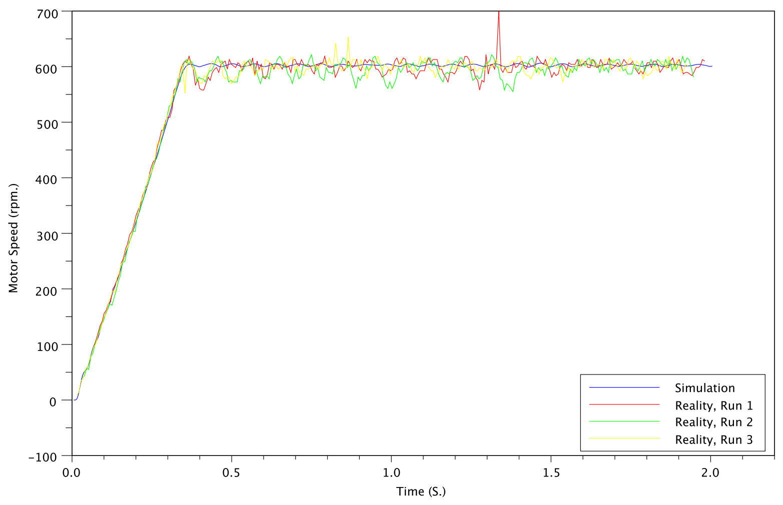


Figure 6.52: Quadrature Encoder, Optimization 1

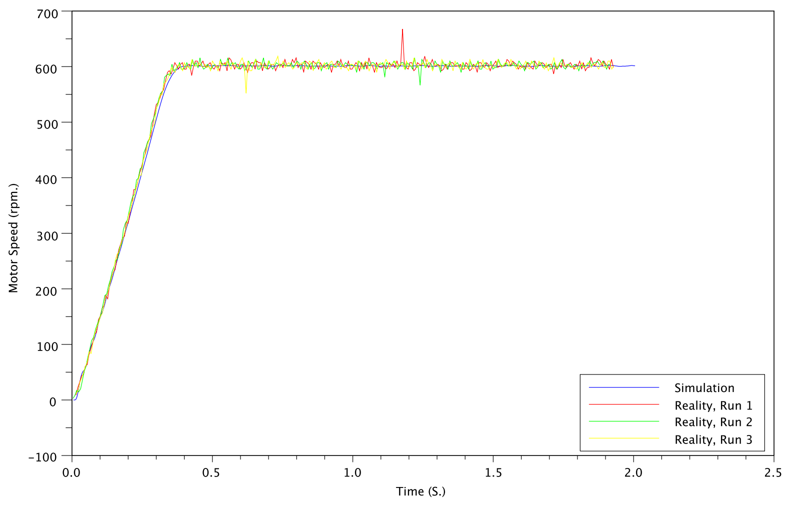


Figure 6.53: Quadrature Encoder, Optimization 2

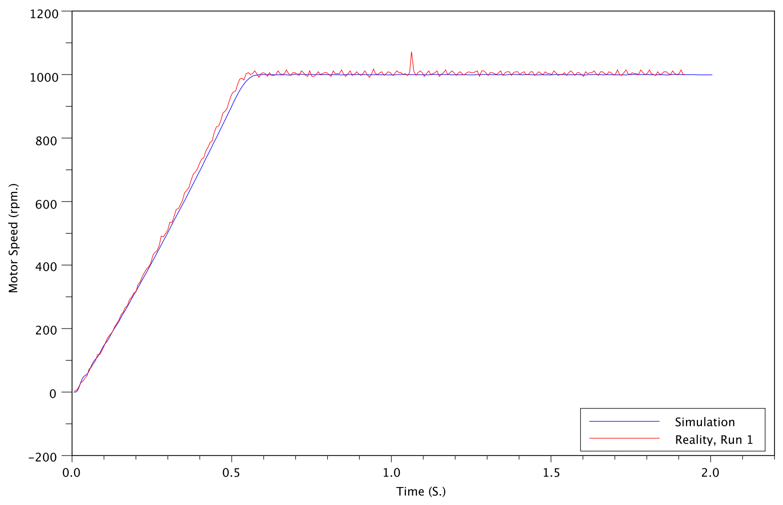


Figure 6.54: Quadrature Encoder, Task Period = 10.0 mS.

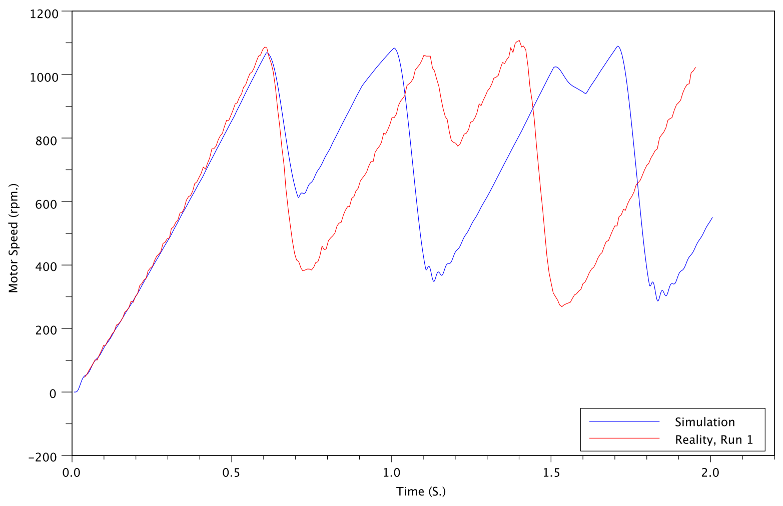


Figure 6.55: Quadrature Encoder, Task Period = 100.0 mS.

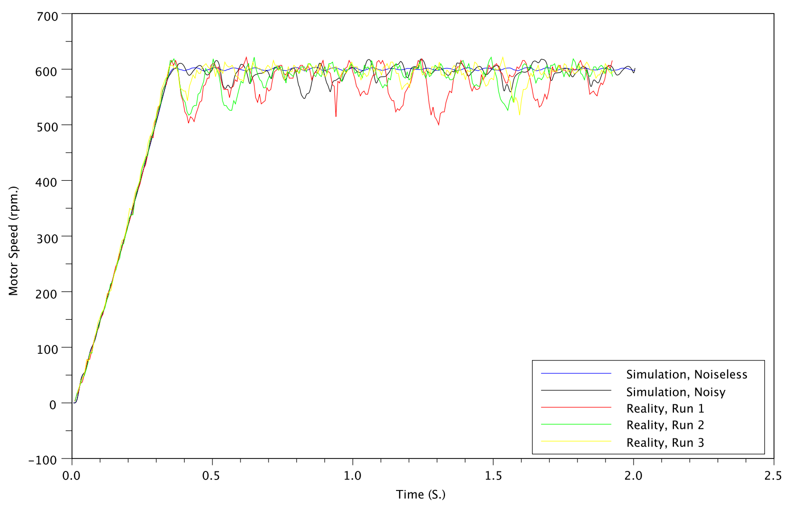


Figure 6.56: Quadrature Encoder, Kp = 25.0, Noise Injection.

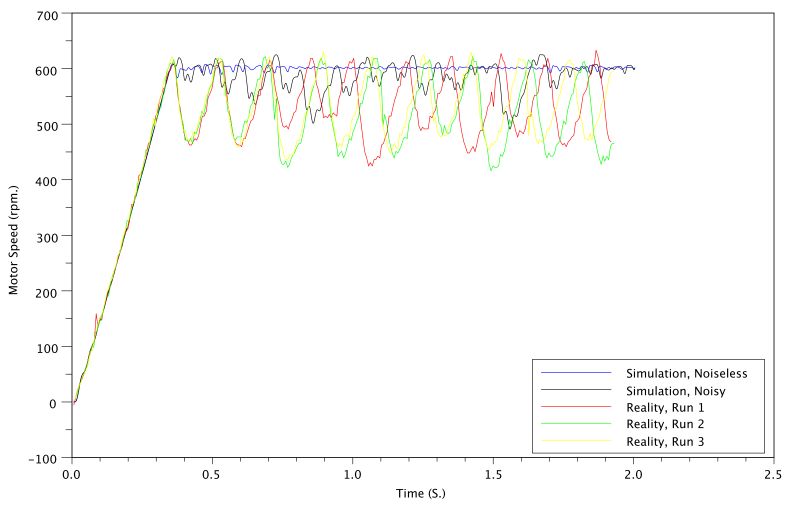


Figure 6.57: Quadrature Encoder, Kp = 100.0, Noise Injection.

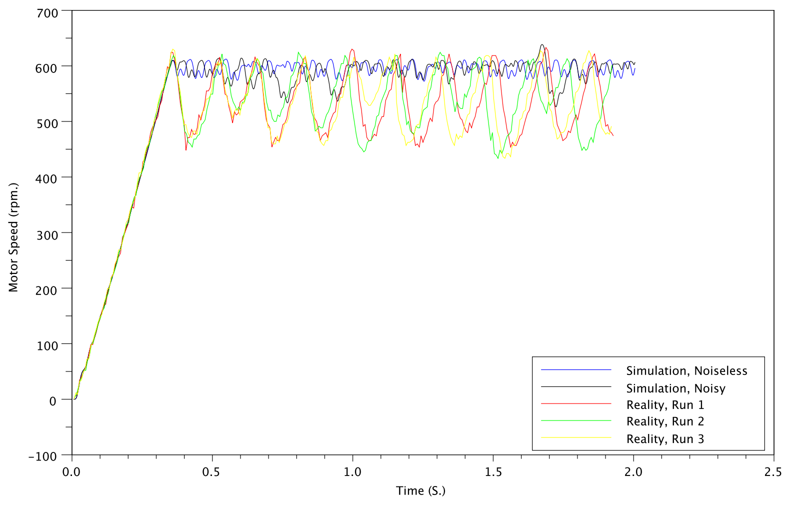


Figure 6.58: Quadrature Encoder, Kp = 250.0, Noise Injection.