Integrated Tool Suite
Phase I: Scoping and Design Study

David M. Auslander, PI, Mechanical Engineering
George Anwar,co-PI, Mechanical Engineering
Dennis Lieu, co-PI, Mechanical Engineering
Sanjit Seshia, co-PI, Electrical Engineering and Computer Science
Student Participants: Alan Zhang, Andrew Ficek, Brennan Kyong Un Chung
Jason Cheung, Jian Shi, Jonathan Shum, Jason Cheung
Matthew Cheung, Ryan O'Bannon

University of California, Berkeley

September 30, 2015
rev. 1: October 4, 2015

Contents

1 Integrated Tool Suite: Workflow **6**
1.1 Introduction . . . 6
1.2 The Toolset . . . 7
1.2.1 Modelica . . . 7
1.2.2 Functional Mock-Up Interface (FMI) . . . 8
1.2.3 Formal Specification of Performance . . . 8
1.2.4 Multi-Media System Components . . . 8
1.2.5 Translation Tools . . . 9
1.2.6 Testbeds . . . 9

2 Integrated Tool Suite: Curricular Development **10**

3 Control Architecture **10**

4 Mini Testbed: Getting an Early Handle on How to Do This **10**
4.1 *Ad Hoc* Simulation of Idealized Model . . . 12
4.1.1 Assumptions . . . 12
4.1.2 Conventional Solution . . . 13
4.1.3 Simulation With Modelica . . . 14
4.1.4 Simulation With Modelica: Equation Based . . . 14
4.1.5 Simulation With Modelica: Mechanical Object Based . . . 14

5 Testbed Analysis: General Procedures **17**
5.1 Introduction . . . 17
5.1.1 Motivation . . . 17
5.1.2 Testbed Overview . . . 17
5.1.3 Workflow Overview . . . 17
5.2 Computer Aided Design - Autodesk Inventor . . . 17

5.2.1 Introduction to Inventor . . . 17
5.2.2 Inventor Model Preparation . . . 18
5.3 Modelica . . . 18
5.3.1 Introduction to Modelica . . . 18
5.3.2 Inventor-to-Modelica Translator . . . 18
5.3.3 Inventor-to-Modelica Translator Software Capabilities . . . 18
5.3.4 Inventor-to-Modelica Translator Software Limitations . . . 18
5.3.5 Adding Multimedia Elements . . . 18
5.3.6 Introduction to Modelica Simulation . . . 19
5.3.7 Hybrid Simulation . . . 19
5.3.8 Separating the Controller from the Plant . . . 19
5.3.9 Modelica Control - State Machine . . . 19
5.3.10 Functional Mock-Up Interface (FMI) . . . 20
5.3.11 JModelica FMU Export Procedure . . . 20
5.4 LabVIEW . . . 21
5.4.1 Introduction to LabVIEW . . . 21
5.4.2 Introduction to LabVIEW RIO Architecture . . . 21
5.4.3 FPGA . . . 21
5.4.4 Feedback Control . . . 21
5.4.5 Supervisory Control . . . 23
5.4.6 Task Control . . . 23
5.4.7 Hybrid Simulation . . . 23
5.4.8 LabVIEW Control & Simulation Module . . . 23
5.4.9 Implementing the Control System . . . 24
5.4.10 Configuring LabVIEW FMI for Model Exchange . . . 24
5.4.11 FMU to LabVIEW . . . 26
5.4.12 Running the LabVIEW Simulation . . . 26
5.4.13 Calibrating Model Specifications . . . 26
5.5 Data Visualization . . . 26
5.5.1 LabVIEW Simulation to Inventor . . . 26
5.5.2 Inventor Visualization Tools . . . 27
5.6 Conclusion . . . 27

6 DC Motor Test Stand 27
6.1 Introduction . . . 27
6.1.1 Testbed Motivation . . . 27
6.1.2 General Procedures . . . 28
6.2 Physical Model . . . 28
6.2.1 Device Characterization . . . 28
6.2.2 Component Specifications . . . 28
6.3 Inventor Model . . . 29
6.3.1 Detailed Inventor Model . . . 29
6.3.2 Inventor Model Properties . . . 29
6.4 Modelica Model . . . 29
6.4.1 Applying Modelica Standard Library . . . 29
6.4.2 Inventor-to-Modelica Translation Software . . . 30
6.4.3 Adding Multimedia Elements . . . 30
6.4.4 Modelica Modeling Assumptions . . . 30
6.4.5 Modelica Simulation Environment . . . 30
6.4.6 Modelica to FMU . . . 31
6.5 LabVIEW . . . 31
6.5.1 FMU to LabVIEW . . . 31
6.5.2 LabVIEW Simulation . . . 31
6.6 Control System Design . . . 34

copyright ©2015 University of California, College of Engineering

6.6.1 Controller Designs . . . 34
6.6.2 Control System Design in Modelica . . . 34
6.6.3 Separating the Controller from the Plant . . . 34
6.6.4 Manually Tuning the PI Controller . . . 35
6.6.5 Control System Design in LabVIEW . . . 36
6.6.6 Comparing Simulation and Real Time Results . . . 36
6.6.7 Trapezoidal Profile Implementation . . . 36
6.7 Data Visualization . . . 39
6.7.1 LabVIEW Simulation to Inventor Dynamic Simulation . . . 39
6.8 Conclusion . . . 39
6.8.1 Workflow Verification . . . 39
6.8.2 Potentially Automated Features . . . 39

7 Inverted Pendulum 40
7.1 Introduction . . . 40
7.1.1 Testbed Motivation . . . 40
7.1.2 General Procedures . . . 40
7.2 Physical Model . . . 41
7.2.1 Device Characterization . . . 41
7.2.2 Component Specifications . . . 41
7.3 Inventor Model . . . 41
7.3.1 Detailed Inventor Model . . . 41
7.4 Modelica Model . . . 42
7.4.1 Inventor-to-Modelica Translation Software . . . 42
7.4.2 Modelica Model Verification . . . 42
7.4.3 Adding Multimedia Elements . . . 42
7.4.4 Modelica Modeling Assumptions . . . 43
7.5 LabVIEW . . . 44
7.5.1 FMU to LabVIEW . . . 44
7.5.2 LabVIEW Simulation . . . 44
7.5.3 Calibrating Model Specifications . . . 44
7.5.4 LabVIEW Model Verification . . . 45
7.6 Control System Design . . . 46
7.6.1 Controller Designs . . . 46
7.6.2 LabVIEW RIO Architecture . . . 46
7.6.3 Closed Loop Step Response Comparison . . . 46
7.6.4 Balancing the Inverted Pendulum . . . 46
7.7 Data Visualization . . . 48
7.7.1 LabVIEW Simulation to Inventor . . . 48
7.7.2 Autodesk Viewer API . . . 48
7.8 Conclusion . . . 50
7.8.1 Workflow Verification . . . 50
7.8.2 Potentially Automated Features . . . 50

8 Modeling Friction using the Modelica Standard Library 51
8.1 Introduction . . . 51
8.2 Modeling the Mass Prismatic System . . . 51
8.2.1 Modelica Model - Mass Prismatic System . . . 51
8.2.2 Modelica Model - Mass Prismatic Control . . . 51
8.3 Modelica to FMU . . . 52
8.4 FMU to LabVIEW . . . 53
8.5 Simulating the Frictionless Mass Prismatic System . . . 54
8.5.1 Frictionless System - Transient Response . . . 54
8.5.2 Frictionless System - Pulse Response . . . 54

copyright ©2015 University of California, College of Engineering

8.6 Introducing Friction into the Model . . . 55
8.6.1 Support Friction . . . 55
8.6.2 Support Friction with Stribeck Friction Table . . . 55
8.6.3 System with Support Friction - Transient Response . . . 56
8.6.4 System with Support Friction - Pulse Response . . . 56
8.6.5 System with Damping . . . 56
8.6.6 System with Damping - Transient Response . . . 58
8.6.7 System with Damping - Pulse Response . . . 58
8.7 Conclusion . . . 58

9 Five Axis Robotic Arm 60
9.1 Introduction . . . 60
9.1.1 Testbed Motivation . . . 60
9.1.2 General Procedures . . . 60
9.2 Physical Model . . . 61
9.2.1 Device Characterization . . . 61
9.2.2 Component Specifications . . . 61
9.3 Inventor Model . . . 61
9.3.1 Detailed Inventor Model . . . 61
9.3.2 Inventor Model Properties . . . 61
9.4 Modelica Model . . . 62
9.4.1 Inventor-to-Modelica Translator . . . 62
9.4.2 Modelica Model Verification . . . 62
9.4.3 Adding Multimedia Elements . . . 63
9.4.4 Modelica Modeling Assumptions . . . 65
9.4.5 Modelica to FMU . . . 65
9.5 LabVIEW . . . 65
9.5.1 FMU to LabVIEW . . . 65
9.5.2 Calibrating Model Specifications . . . 65
9.5.3 Trapezoidal Profile Control System Design . . . 66
9.5.4 Tuning the Control System Using Simulation . . . 66
9.5.5 Comparing Simulation and Real Time Results . . . 66
9.5.6 Optimizing the Real Time System . . . 68
9.6 Conclusion . . . 68
9.6.1 Workflow Verification . . . 68
9.6.2 Potentially Automated Features . . . 68

10 Slider Crank 70
10.1 Introduction . . . 70
10.1.1 Testbed Motivation . . . 70
10.1.2 General Procedures . . . 70
10.2 Physical Model . . . 70
10.2.1 Device Characterization . . . 70
10.2.2 Component Specifications . . . 71
10.3 Inventor Model . . . 71
10.3.1 Detailed Inventor Model . . . 71
10.3.2 Inventor Model Preparation . . . 71
10.4 Modelica Model . . . 71
10.4.1 Inventor-to-Modelica Translator . . . 71
10.4.2 Modelica Model Initialization . . . 72
10.4.3 Adding Multimedia Elements . . . 72
10.4.4 Modelica Modeling Assumptions . . . 73
10.4.5 Modelica Simulations . . . 73
10.5 Functional Mock-Up Interface (FMI) . . . 73

copyright ©2015 University of California, College of Engineering

10.5.1 FMI 2.0 Model Exchange . . . 73
10.6 Going Forward . . . 78

11 Go-Cart **78**
11.1 Introduction . . . 78
11.1.1 Testbed Motivation . . . 78
11.1.2 General Procedures . . . 78
11.1.3 Workflow Diagram . . . 78
11.2 Physical Model . . . 78
11.2.1 Device Characterization . . . 78
11.2.2 Component Specifications . . . 79
11.3 Inventor . . . 79
11.3.1 Detailed Inventor Model . . . 79
11.3.2 Inventor Model Preparation . . . 80
11.4 Modelica . . . 80
11.4.1 Inventor-to-Modelica Translator . . . 80
11.4.2 Adding Multimedia Elements . . . 81
11.4.3 Modelica Modeling Assumptions . . . 81
11.4.4 Modelica Simulation Environment . . . 81
11.5 Functional Mock-Up Interface (FMI) . . . 83
11.5.1 FMI 2.0 Model Exchange . . . 83
11.6 Going Forward . . . 83

12 Conclusions **83**

1 Integrated Tool Suite: Workflow

1.1 Introduction

The purpose of this project is to automate the workflow associated with the design and control of complex mechanical systems. The purpose of this phase of the project, the scoping and design phase, is to explore the technologies needed to achieve this goal and develop a proposal for the next phase of the project which would be to develop those technologies as far as possible, to provide skeleton application frameworks for using the technology, and to outline instructional programs to use the material at the undergraduate, graduate and professional levels.

The project is a cooperative venture of National Instruments Inc, Autodesk Inc, and the University of California, Berkeley.

Whereas the design (CAD) environment is very well developed as is the control environment, the simulation environment which can be used to design the control system is generally left to *ad-hoc* efforts and is often not done well, or not done at all. This is expressed by Figures 1 and 2 (these are from the draft integrated Tool Suite Scoping and Design Study slide set distributed by Andy Chang). The first figure shows the "first principles model." There is no systematic way of constructing this model from the CAD data, thus the characterization of that process as *ad-hoc*. The second figure shows a single "system specification" that is the basis for the design, the simulation and the control and is the goal of this project.

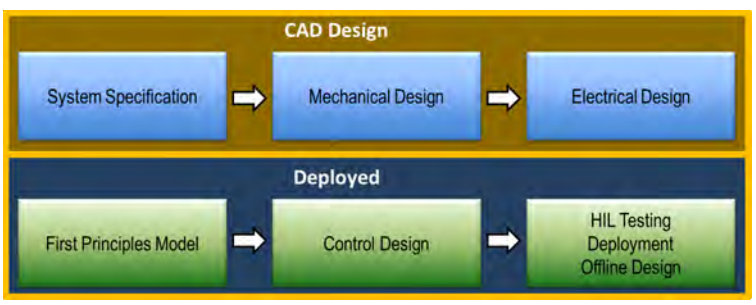

Figure 1: Double Source

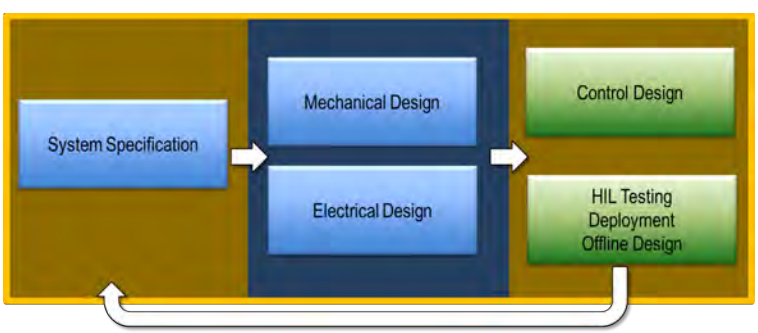

Figure 2: Single Source

Figure 3 shows the workflow that we are aiming for. The key steps are transferring information from Inventor to a simulation environment in such a way that the simulation can be constructed automatically. The second key step is porting the control code from the simulation environment where it was designed to the control implementation environment without having to rewrite the control code.

In summary, we are working towards a situation in which it will be possible for an engineer or a group, having Inventor and LabVIEW skills will be able to accomplish a full dynamic simulation of a system without the need for a simulation expert.

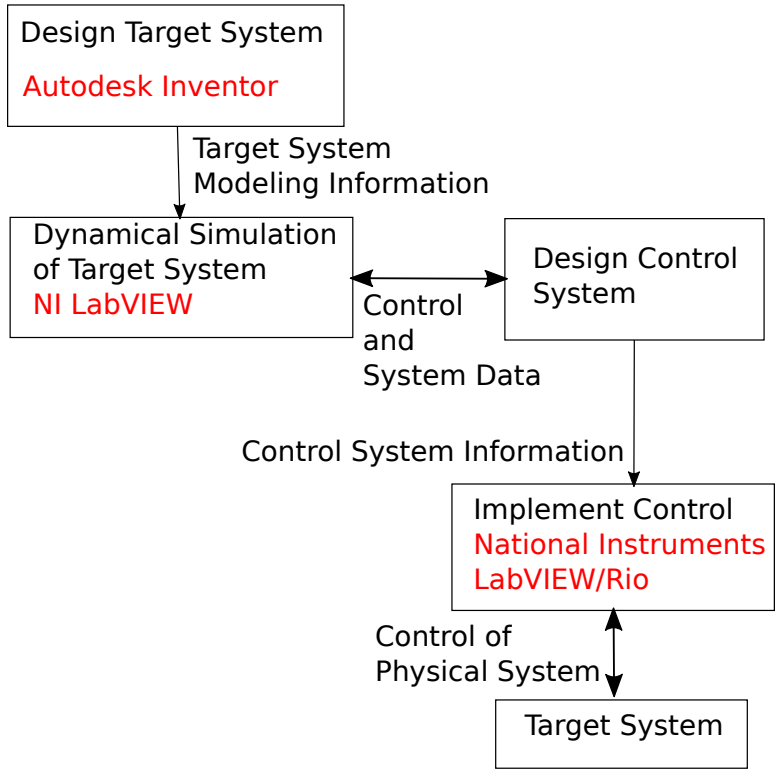

Figure 3: Integrated Tool Suite Workflow

1.2 The Toolset

The CAD tool for this study is Autodesk Inventor, the control hardware tool is the National Instrument RIO family of control modules, and the software environment for both control and simulation is National Instrument LabVIEW.

The core of the immediate work is evaluating software technology to fill the gap between Inventor and LabVIEW simulation. The biggest challenge in filling this gap is that mechanical system in general are defined by their constraints (bearings, joints, hinges, bushings, etc.) and thus require a combination of differential and algebraic equations to describe them mathematically (differential-algebraic equations or DAEs). These systems are also called acausal systems. Most simulation environments (including LabVIEW), however, require mathematical models for which the differentials are explicitly computable (that is, if the differential equations are expressed as a set of first order differential equations, the derivatives associated with each of those equations must be explicitly computable). These are called causal equations.

1.2.1 Modelica

The Modelica modeling and simulation standard is a candidate for filling the gap between acausal systems (those with constraints) and causal system (those for which any constraints have been accommodated mathematically). Modelica is different from conventional simulation in that it deals natively with acausal systems and DAE systems. Models in Modelica are expressed in true equations

rather than algorithmically. The "=" sign corresponds to the algebraic equal sign rather than to the "store-as" operator of standard computing languages (some computing languages have attempted to emphasize that difference by using the symbol ":=" for store-as, but the most popular languages don't do that; Modelica allows for algorithmic sections and uses the := operator for store-as).

It should be emphasized that while Modelica is a candidate for providing the necessary structure for the bridging Inventor to LabVIEW, it will be invisible to the intended users (recall that the skillsets we are aiming for include Inventor and LabVIEW, but not simulation specialty). We do have considerable interest in the possibility of using Modelica as part of the instructional program, however, and it also might be useful in enhancing or modifying cutting edge models before moving on to simulation.

The term "Modelica" used by itself refers to an open standard (see `www.modelica.org`) not to any particular implementation. There are a number of implementations available, some commercial, some freely available. Given the nature of this project, we are focusing on the open-source versions, of which we are using both OpenModelica and JModelica.

1.2.2 Functional Mock-Up Interface (FMI)

The prime candidate for the exchange of information among the components listed in Figure 3, above, is the FMI protocol. FMI was created around the Modelica environment as it is used to express solutions to simulation problems in the same terms as used by Modelica implementations. Any algebraic constraints associated with the problem are resolved and then the solution is presented as a causal simulation in the form of C functions. Thus, recipients of a Functional Mock-up Unit (FMU, a collection of information following the FMI protocol) do not have to deal with analysis or solution of DAEs; rather they need to be able to access and execute the solutions and parse the accompanying XML file that gives variable names and other relevant information about the solution.

An FMU can be targeted for co-simulation or model exchange. The major difference is that a co-simulation FMU includes the solver as well as the model; a model exchange FMU only includes the model information.

Both NI and Autodesk are currently exploring the use of the FMI protocol. If successful, this will provide the needed interchange protocol to bring models from Inventor to LabVIEW.

LabVIEW presently contains an experimental module to read FMUs based on the model-exchange format. Autodesk has experimental software to create a Modelica model from an Inventor model; this can be converted to an FMU using an existing Modelica implementation.

1.2.3 Formal Specification of Performance

Many engineering systems, particularly those designed for internal use, are designed and built without any formal specification for performance and without a carefully specified acceptance testing protocol. This can lead to very expensive misunderstandings when the product is delivered to the end user(s). As a part of this project, formal specifications for performance and a testing protocol will be built into the workflow.

1.2.4 Multi-Media System Components

A challenging aspect of this project will be the integration of system components that are not entirely mechanical. These include electromechanical, pneumatic, hydraulic components, etc. Components such as motors, pumps, etc., are ubiquitous in mechanical systems, and have their mechanical characteristics included in the Inventor model, but not non-mechanical properties such as resistance, inductance, torque constant, motor type, etc. These properties are essential to system simulation

so there must be a means of getting them into the model. However, it is equally essential that the primary target users not be required to create their own mathematical and computational models of multi-media components. This issue will not be approached in the present phase of the project (Scoping and Design). Rather, we will produce *ad hoc* models for these components.

1.2.5 Translation Tools

The ideal workflow would be to go directly from an Inventor model to an FMU (Functional Mock-up Unit). The FMU contains full model and simulation information that can be used in a standard simulation environment. That FMU would be loaded into LabVIEW's simulation module, providing the basis for control software design. Once the control system software was at a satisfactory point, that same software module could be combined with real-time software to implement control of the actual object.

There is not yet any module from Inventor to FMU, but there is a translator from Inventor to Modelica (using mostly the Modelica Mechanics Multibody Library). Then, as long as we use JModelica rather than OpenModelica, we can produce an FMU that LabVIEW can read.

There is a reasonable argument that the Inventor-to-Modelica translator is more useful than a direct Inventor-to-FMU translator because it is open and available for modification or addition whereas the FMU is closed. While the ideal workflow would not use the Modelica model, cutting edge projects that push the limits of the translation tools might require modification at the mathematical model level. In any case, an FMU translator should be capable of producing a Modelica model as well as the FMU.

Details of what works and what doesn't in these translation tools will be given below with the details of the various testbeds.

1.2.6 Testbeds

The Scoping phase of the project will proceed by defining a number of testbeds and exploring the workflow associated with those testbeds to identify areas that will work using existing tools, areas that need modified tools, and areas that are not currently working at all and will require new tools. The idea is that these testbeds are already existing mechanical systems – given the time available for the Scoping and Design phase it would be impossible to start designs from scratch and have to build the mechanical systems (to say nothing of the cost involved).

These testbeds will be reverse engineered into Inventor models and then we will demonstrate the workflow needed to simulate the system, design a control using the simulation, and then transfer the control software from the simulation to the real system.

There are several testbeds we are currently working on:

- Mini-Testbed – Seesaw: not a real testbed because it has no associated hardware but gives an opportunity to take an early look at the workflow and toolset
- Motor Test Stand – a bare-shaft DC brush motor on base with an incremental encoder
- Five-axis robot arm
- Swing-up inverted pendulum
- Go-Cart steering mechanism
- Go-Cart drive mechanism
- Slider crank
- Quad rotor aircraft

copyright ©2015 University of California, College of Engineering

2 Integrated Tool Suite: Curricular Development

Although the present stage of the project (Scoping and Design) is focused almost entirely on technical issues associated with achieving the desired workflow, the full end product of the project will include not only those technical achievements but also a set of curricula using the integrated tool suite workflow. We are planning to develop curricular modules in at least these areas:

- Undergraduate (lower and upper division)
- Graduate
- Professional

These curricular modules will represent the primary documentation for usage of the integrated tool suite. The differences among the modules will depend on the educational and experience levels expected of their users. For example, lower division undergraduate students have little mathematics or science background so an intuitive approach is important.

3 Control Architecture

When control is applied to the various testbeds it implements more than just feedback control and includes the lower layers of control needed to perform a useful task. Figure 4 shows these lower layers along with the physical layer and the interface to the physical layer. That interface is normally implemented using the FPGA facility of Compact-RIO devices and thus falls into the region between hardware and software. The feedback control layer is usually implemented as software, but, when particularly fast sample times are needed, can also be implemented in the FPGA. The supervisory layer provides setpoints to the feedback layer. Setpoint profiling is important in assuring consistency of control by keeping the error at the feedback layer always small. For motion control systems, for example, the most common profile for point-to-point moves is the trapezoidal profile having a constant acceleration phase, a constant velocity (cruise) phase, and a constant deceleration phase. The task control layer is used to put together sequences of supervisory modules to achieve functional requirements of a system. For example, a robot arm doing a pick-and-place task will have a sequence of point-to-point moves, interspersed with control of the end effector to pick up or release the target objects. The task control layer is often implemented with finite-state machines.

There are several levels of control above this, such as operator interface, factory control, etc., but those levels are not central to this project.

4 Mini Testbed: Getting an Early Handle on How to Do This

While we are working on several testbeds with significant mechanical complexity, some of the basic issues associated with the proposed workflow can be explored with much simpler problems. In particular, dealing with constraints and the resulting differential-algebraic-equations (DAEs) is something that we must be able to handle. Constraints are the essence of mechanical systems and most *ad hoc* simulation efforts go to a great deal of effort to define the explicit constraints out of the simulation by reducing the order and/or dimensionality of the system.

Figure 5 shows a typical "classroom model" of the typical children's park seesaw, with added springs. This system, while simple, has some nice properties for this analysis: it is nonlinear, and although it has two masses and two springs, the actual system order is much lower because of the constraining beam. We will attempt to analyze this system from several points-of-view in order to bring to the surface the problems that we need to deal with in order to automate the simulation process. Since this is a very preliminary analysis, we will not build or test actual physical models of this system.

copyright ©2015 University of California, College of Engineering

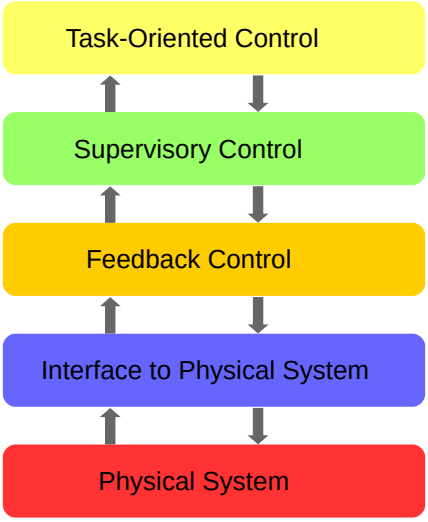

Figure 4: Control Architecture

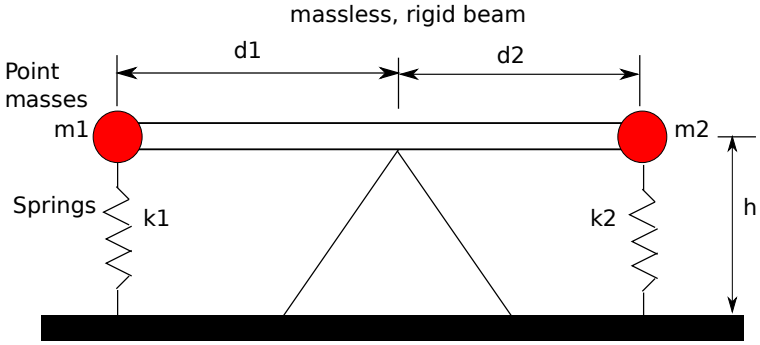

Figure 5: Idealized Model of a Seesaw

In contrast, Figures 6 and 7 show Inventor models of the same system. Unlike the idealized components of the previous model, the Inventor models have real beams and masses that are not point-masses. Although the two models nominally represent the same system, different designers came up with different designs. Reconciling these models is the essence of this project.

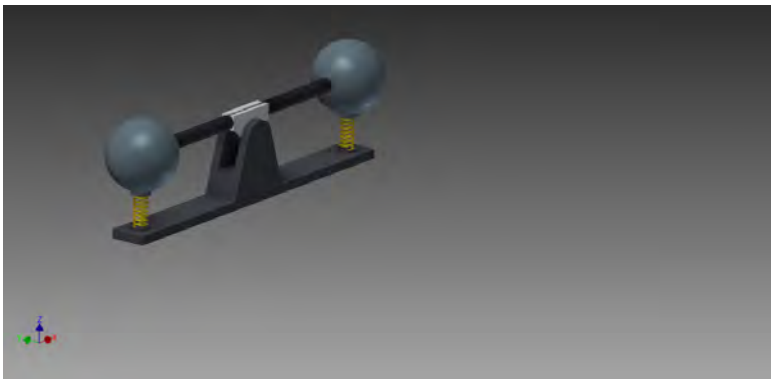

Figure 6: Inventor Model of a Seesaw (version 1)

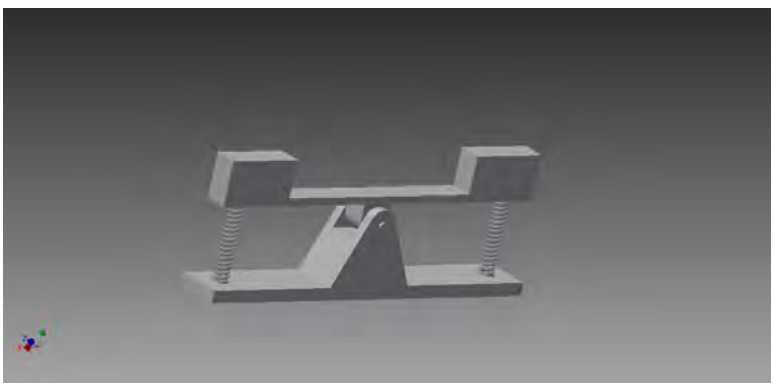

Figure 7: Inventor Model of a Seesaw (version 2)

4.1 *Ad Hoc* Simulation of Idealized Model

4.1.1 Assumptions

For the purpose of this part of the analysis, it is assumed that the Inventor models have been reduced to the idealized model shown in Figure 5. The simulations of this idealized model uses a physics-based model, which for this system is a differential-algebraic equation (DAE), and then uses analytical techniques to simplify the model as far as is convenient, leaving the rest to numerics. In this case, although it is perfectly obvious to almost everyone that the system is fully describable as a second order system because of the constraint imposed by the rigid, massless beam, for example, using beam angle and angular velocity as state variables, that step was purposely not taken in this case so that the system would remain in acausal (DAE) form and not be reduced to a causal system. Several assumptions are also made for the particular problem. These are:

- The beam-and-masses system is treated as two pendulums.
- The angular acceleration of the two masses must match.
- Longitudinal acceleration of the masses (in the direction of the beam) is ${v_n}^2/r$, where v_n is the velocity normal to the beam, and an appropriate sign convention is used.

- The beam is massless and thus cannot support any moment; therefore, the net moment associated with the normal forces applied from the beam to the masses must be zero.

With these assumptions, all of the variables that are unknown due to the algebraic constraints can be determined except the normal force applied from the beam to one of the masses (the other normal force can be found from the zero-net-moment assumption).

4.1.2 Conventional Solution

The conventional solution to this problem is to use a standard (causal) ordinary differential equation (ODE) solver and satisfy its need to have all derivatives explicitly computed by using either analytical or numerical methods to resolve the constraints.

The solution has been implemented in Java, using a Runge-Kutta 4/5 ODE solver and using the Nelder-Mead nonlinear simplex optimizer to find the unknown normal force the beam applies to one of the masses.

If the spring constants are both set to zero the system behaves as a simple pendulum. Figure 8. The parameter values are shown in Table 1. The initial condition is that the beam is horizontal and all velocities are zero.

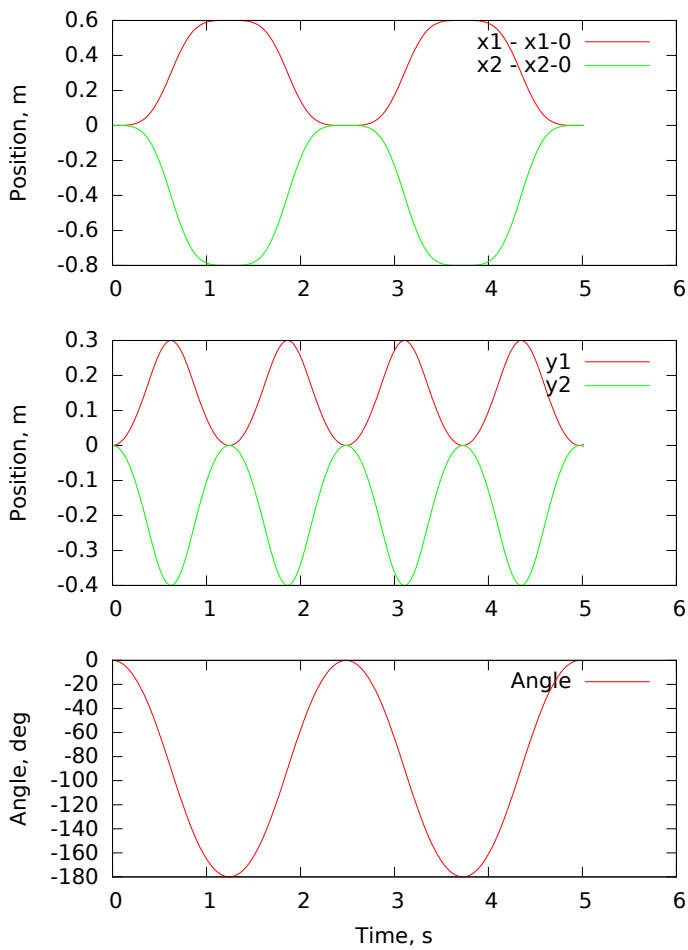

Figure 8: Conventional Solution, No Springs

Item	Value
d1	0.3 m
d2	0.4 m
h	0.2 m
m1	0.2 kg
m2	0.3 kg
k1	0 N/m
k2	0 N/m

Table 1: Parameters for Java Simulation, No Springs

When the springs are activated (both with spring constant of 20.0 N/m) the result of Figure 9 is obtained.

4.1.3 Simulation With Modelica

Modelica is different from conventional simulation in that it deals natively with acausal systems and DAE systems. Models in Modelica are expressed in true equations rather than algorithmically. The "=" sign corresponds to the algebraic equal sign rather than to the "store-as" operator of standard computing languages (some computing languages have attempted to emphasize that difference by using the symbol ":=" for store-as, but the most popular languages don't do that).

Two Modelica solutions will be shown: one based on hand-derived equations for the seesaw system and the other based on using mechanical system components predefined in the Modelica library.

4.1.4 Simulation With Modelica: Equation Based

Thus, for the seesaw system, the primary constraint that the angular accelerations of each of the masses are equal, and the constraint the the net moment on the beam be zero, are expressed by:

```
// Equations that join the two pendulums
fbN1 * d1 = -fbN2 * d2; // Moment on beam must be zero (it's massless)
aN1 / d1 = aN2 / d2;  // Angular accelerations must match
```

The rest of the model, although expressed as equations rather than as executable, algorithmic statements, looks very similar to code in the Java program (or any other programming language). However, the two lines above make all the difference! Modelica resolves these constraints internally to convert the model to causal form and then performs the simulation (Figure 10), whereas that process was done explicitly by the analyst in the conventional solution.

There are a number of implementations of Modelica currently available, both commercial and free. This solution was done using OpenModelica, an open source Modelica implementation.

4.1.5 Simulation With Modelica: Mechanical Object Based

After using Modelica at its basic, equation, level, it is useful to move up one level of abstraction and make use of the extensive libraries of components that are available. For the seesaw system in particular, the Modelica Standard Library (MSL), Mechanics, Multibody Library is extremely useful (this library is open source). Figure 11 shows an implementation of the seesaw using this library (thanks to Greg Morrow of NI for this model). To the extent that such libraries cover our needs, the step of needing to derive the system equations is not needed. This is important because that step usually requires a simulation and modeling specialist.

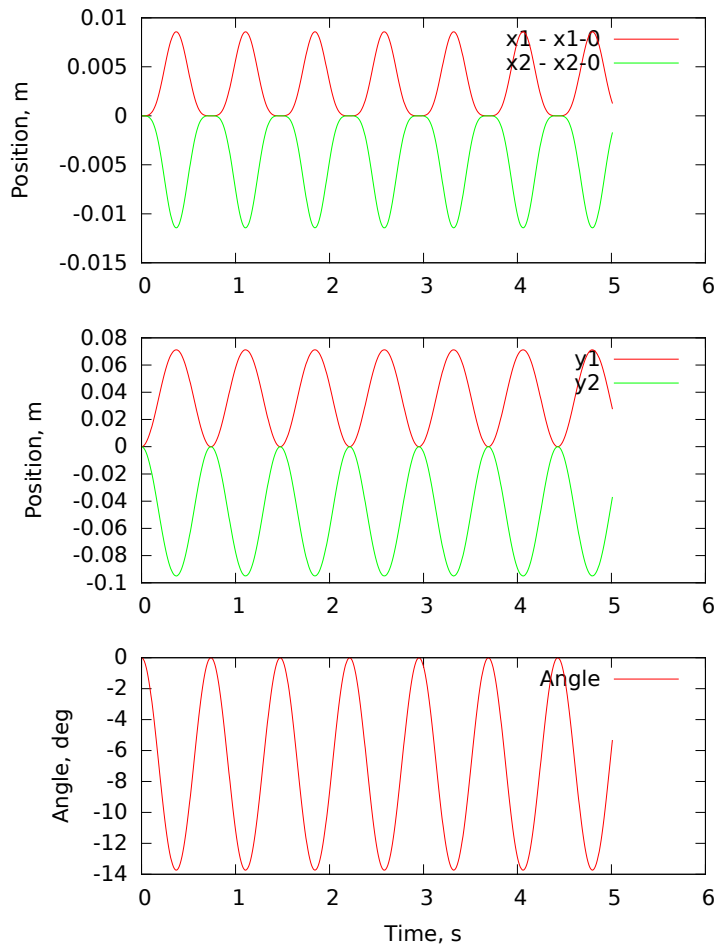

Figure 9: Conventional Solution, With Springs

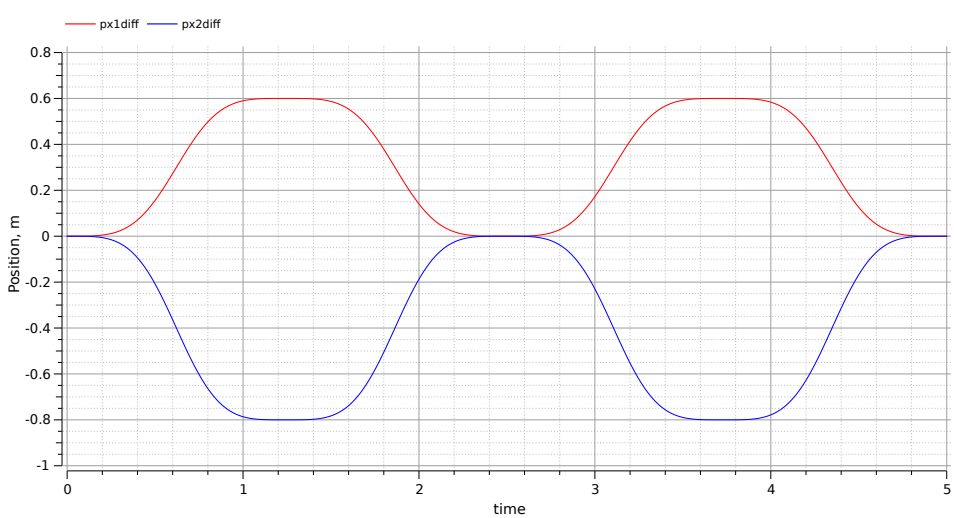

Figure 10: Modelica Simulation, No Springs

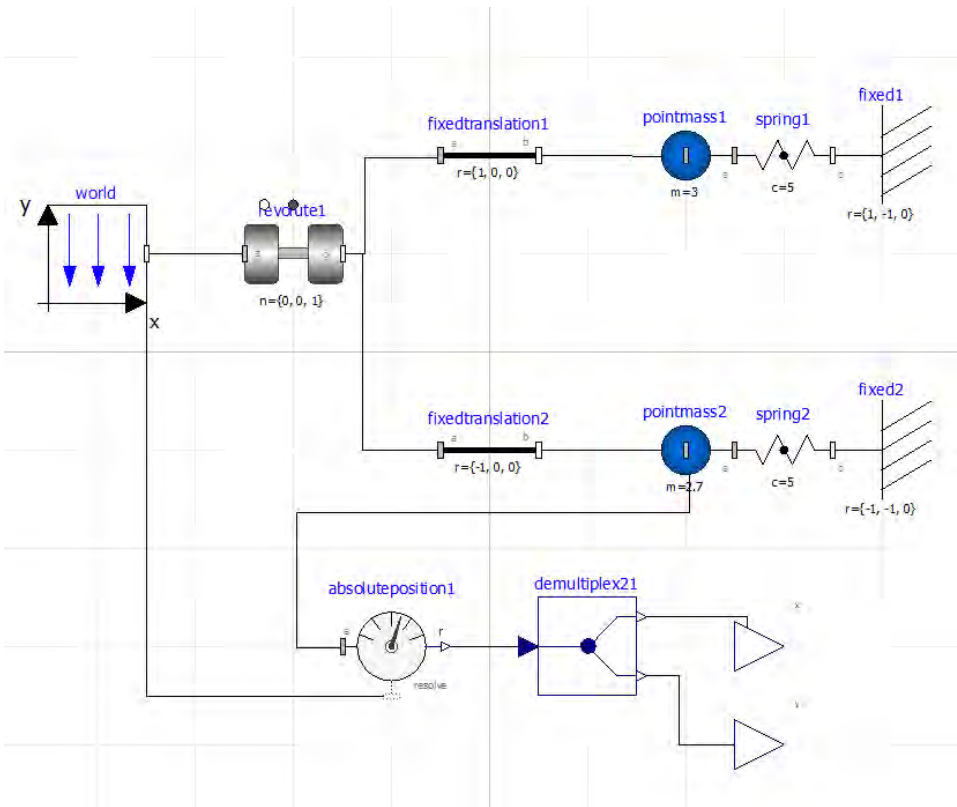

Figure 11: See Saw Model Based on Modelica Multi-Body Blocks

5 Testbed Analysis: General Procedures

5.1 Introduction

5.1.1 Motivation

The goal of this project is to automate the workflow associated with the design and control of complex mechanical systems - specifically to bring simulation earlier into the design process. Simulation allows users to evaluate and iterate on the design process before manufacturing and to design and test the control system in parallel with construction of the physical system. We start with a CAD model of our complex system then move the model into a simulation environment, where control software can be designed to control both the model in a simulation and, when it becomes available, the physical system.

5.1.2 Testbed Overview

These following testbeds illustrate a variety of properties found in real mechanical systems. Aside from the slider crank, we reverse engineer these physical systems to avoid bias from designing mechanisms that would adhere to the workflow.

1. DC Motor Test Stand
2. Inverted Pendulum
3. Five Axis Robotic Arm
4. Go-Cart Drivetrain
5. Go-Cart Steering
6. Slider Crank

For each testbed, we will run through our prospective workflow and document specific advantages and challenges unique to each system.

5.1.3 Workflow Overview

First, we begin with a model of our complex mechanical system in CAD environment. In this project, we use Autodesk Inventor and a prototype plug-in to convert our Inventor model into a Modelica model. We then use Modelica environments (OpenModelica and JModelica) as platforms to modify our model and export our model as an Functional Mock-up Unit - a file type based on the FMI 2.0 Standard which describes dynamic systems using differential, algebraic, discrete time equations. Next we import our model into a simulation environment. We use NI LabVIEW Control & Simulation to design our control system and produce simulation results. After we have designed our control system, we feed the simulation results from NI LabVIEW back to Autodesk Inventor for visualization purposes using Autodesk Inventor Dynamic Simulation. Once we are satisfied with the behavior of our system, we can manufacture our complex mechanical system and use NI LabVIEW RIO architecture to control the system in real time.

5.2 Computer Aided Design - Autodesk Inventor

5.2.1 Introduction to Inventor

To begin, we model our mechanical system in Autodesk Inventor. The CAD environment allows us add constraints and to specify material and mass properties. This information is crucial for modeling

copyright ©2015 University of California, College of Engineering

the dynamics of our system. We can use Inventor to create multiple iterations of a design to verify the behavior of our system before manufacturing.

5.2.2 Inventor Model Preparation

An important consideration for converting file types is to separate the assembly into as few components as possible. Create subassemblies by constraining parts with no relative movement together. This reduces the number of constraints in the assembly and makes the file easier to export to different software environments. Also, this process currently limits the types of constraints that can be used to model the system in Inventor. The process currently does not recognize complex constraints from the design module of Inventor.

5.3 Modelica

5.3.1 Introduction to Modelica

We will be using the Modelica modeling language as an intermediate step to convert the Inventor model into a LabVIEW block for control and simulation. Modelica is an open source equation based language used to model complex system and is used by many companies in the power and automotive industry. To modify Modelica models, we will be using OpenModelica Connection Editor (OMEdit) and JModelica because they are free open source environments. OMEdit provides a linked text and graphical editor with a simulation environment while JModelica provides a Python package with improved support for FMI 2.0 Model Exchange.

5.3.2 Inventor-to-Modelica Translator

We will also take advantage of specialized software from Autodesk to translate our CAD model directly into a Modelica model. In general, the translator uses Autodesk Inventor's Dynamic Motion Addin API to convert Inventor assemblies into Modelica models.

5.3.3 Inventor-to-Modelica Translator Software Capabilities

The translator is able to interpret basic constraints between parts and subassemblies by recognizing degrees of freedom between each component. It also embeds the current position and the mass property of components as Modelica's MultiBody bodies.

5.3.4 Inventor-to-Modelica Translator Software Limitations

There are currently several limitations to the software that prevent us from converting Inventor models into Modelica models. The software currently does not recognize Inventor Motion Constraints and has a difficult time handling redundant constraints in Inventor models. Also, the translator is unable to recognize 3D rigid body contacts, non-rigid body dynamics, and electronic components.

5.3.5 Adding Multimedia Elements

Although the translator currently does not export complex constraints, we can manually add multimedia elements and constraints not captured by the translator in OMEdit. We can take advantage of the Modelica Standard Library for ready-to-use components and add electronics, friction forces, and input and output terminals. Several commercial libraries and system specific libraries are also

copyright ©2015 University of California, College of Engineering

available and can be adapted to the workflow. The following list contains many elements available in the Modelica Standard Library that can be used to augment the translated Modelica model.

Missing Rigid Body Dynamics Rigid bodies, friction, gears, 3D joints

Electronics Sensors, machines, circuits, analog, digital

Complex Properties Thermal, fluid, magnetic components

Input and Output Terminals Interfaces crucial for block diagrams in LabVIEW

In general, we will need to calibrate these elements and verify their behavior before the control system design phase.

5.3.6 Introduction to Modelica Simulation

With a Modelica model, we can move forward with the process of designing a control system in Modelica. To do this, we can either directly modify the original system or separate the controller from the original system using files connected with input and output terminals. It is advantageous to separate the system from the controller so that the files can be transferred between software environments, and controllers can be reused and tuned without accessing the system.

5.3.7 Hybrid Simulation

To create a hybrid environment for control system design, use a when loop to sample discrete time intervals. This way, we can compare the simulation with the real time system. We can also compare the Modelica simulation with the LabVIEW simulation by configuring the same time steps.

5.3.8 Separating the Controller from the Plant

To create our controller in a separate file, create a new model with the control algorithm. Then to link the controller to the system, either reference the system using "Plant sys" in the Modelica code or drag and drop the block into the system through the visual editor in OMEdit where "Plant" is the name of the Modelica model. By declaring system as a variable in the controller, the controller can reference the system's input and output variables as instance variables. For example, if the system has input u and output y, we can get input or output of these values by calling sys.u or sys.y. Finally, we add our control algorithm in a discrete time environment by setting a sampling time and controlling the input into the system using a PID function with the appropriate parameters. Sample controller code is shown below in Figure 12. We have separated the PID function from the PID controller model so we can reuse it for different systems.

5.3.9 Modelica Control - State Machine

We can also develop higher levels of control using the Modelica environment. For example, we will implement a state machine to allow us to control multiple point-to-point moves. First, establish a discrete time environment by using a while loop to initialize the control at different time steps. Multi-rate problems can be solved with multiple discrete time zones. The control will guide the system through the following 4 states.

State 0 Initializes and brings the system under control at the initial position

State 1 Moves to the first position and transitions to State 2 once the move is done

State 2 Holds the position for a set amount of time and transitions to State 3 once time is up

State 3 Moves back to the original position

copyright ©2015 University of California, College of Engineering

```
model PIDController
  Plant sys;
  Real velSet = 50;
  parameter Real kpv = 0.022, kiv = 0.96, kdv = 0.0;
  parameter Real delT = 0.005;
  Real mVel;
  Real integVel, valPrev;
algorithm
  when sample(0, delT) then
    (mVel, integVel, valPrev) := PIDFunc(dt = delT, kp = kpv,
        ki = kiv, kd = kdv, integ = integVel, setpoint = velSet,
        val = sys.w, first = false);
    sys.u := mVel;
  end when;
end PIDController;
```

Figure 12: Separated Controller Code in Modelica (Note: Only PI is implemented)

With this state machine set up, we can control position using velocity as an inner loop (cascade) and a trapezoidal profile to get set points during the move. This requires defining algorithms to compute and run the trapezoidal moves.

5.3.10 Functional Mock-Up Interface (FMI)

Functional Mock-up Interface (FMI) is a tool independent standard that supports model exchange and co-simulation using a combination of xml-files and compiled C-code. It simplifies the exchange of dynamic system models between different modeling, simulation, and co-simulation tools. This standard allows us to transfer the model between Autodesk Inventor, Modelica, and LabVIEW. We will be using the FMI Standard 2.0 to evaluate the system of equations generated by Modelica in different environments. We can convert our file into an FMU using JModelica, which supports the FMI 2.0 protocol. When exporting an FMU, there are two options available in JModelica: Model Exchange or Co-Simulation.

Model Exchange Describes the dynamic system using differential, algebraic, discrete time equations

Co-Simulation Contains both the model and the solver. Also able to perform real-time data transfer between software environments

We will only need the model exchange format because because LabVIEW is able to solve and simulate the model generated without Co-Simulation.

5.3.11 JModelica FMU Export Procedure

First download JModelica and and start a Python 2.7 session. We will need to import the JModelica Python interface, PyModelica, to compile Modelica code into FMU. Then, describe the model_name and location of the mo_files with Python strings. The function, compile_fmu, will generate FMU for the model in the JModelica directory. The code for this procedure is shown in Figure 13

```
from pymodelica import compile_fmu
model_name = 'RLC_Circuit'
mo_files = 'RLC_Circuit.mo,RLC_Dependent.mo'
my_fmu = compile_fmu(model_name,mo_files,version=2.0)
```

Figure 13: FMU Export Procedure

5.4 LabVIEW

5.4.1 Introduction to LabVIEW

LabVIEW is a convenient tool for simulating dynamic systems, testing controller designs, and deploying control systems to real-time hardware all in an integrated environment. Given a complex mechanical system as a FMU, we can convert the file into a plant model, simulate its step response, and design a controller for the system. The original system parameters contained in the FMU can also be tuned in LabVIEW to achieve desired response without having to export another FMU.

To compare simulations with real-time systems, we developed a framework to design real-time control code. In the framework, the control code is separated into four distinct layers: FPGA, Feedback Control, Supervisory Control, and Task Control. Higher levels of control manage inputs to lower levels of control and the end user interacts solely with the Task Control layer as the other three layers are abstracted away.

5.4.2 Introduction to LabVIEW RIO Architecture

The NI LabVIEW reconfigurable I/O (RIO) architecture is an integral part of the NI graphical system design platform. It consists of four components: a processor, a reconfigurable field-programmable gate array (FPGA), modular I/O hardware, and graphical design software. We used the NI CompactRIO to control the DC motor test stand and the robotic arm and used the NI single-board RIO for the inverted pendulum. To interface with the different hardware, we need to write several LabVIEW FPGA programs to get access to unique input and output pins. At the FPGA level, we manipulate encoder counts and output PWM signals to motor. We can also do some basic arithmetic like calculating real time velocity on FPGA.

5.4.3 FPGA

Th FPGA layer is the lowest layer of control and communicates directly with the hardware. To control many of the testbeds, we use a CompactRIO-9068 with six NI9505 motor modules to control up to six motor and encoder systems. This layer is able to interact with the motor by calculating the location of the encoders and outputting the position and velocity of each axis. A portion of the code for the FPGA portion of the CompactRIO is shown in Figure 14.

5.4.4 Feedback Control

The Feedback Control layer is above the FPGA layer and produces inputs to the FPGA layer to implement control algorithms. The code is stored and run on the real-time portion of the CompactRIO to decrease communication delays. Outputs from the Feedback Control layer can be connected to the FMU for simulation but, to decrease latency, usage of local variables and other non-time dependent variables should be minimized. The Feedback Control layer is also designed to run independently from the following two layers which allow it to be encapsulated nicely into a simulation. An example of the feedback control system is located in Figure 15.

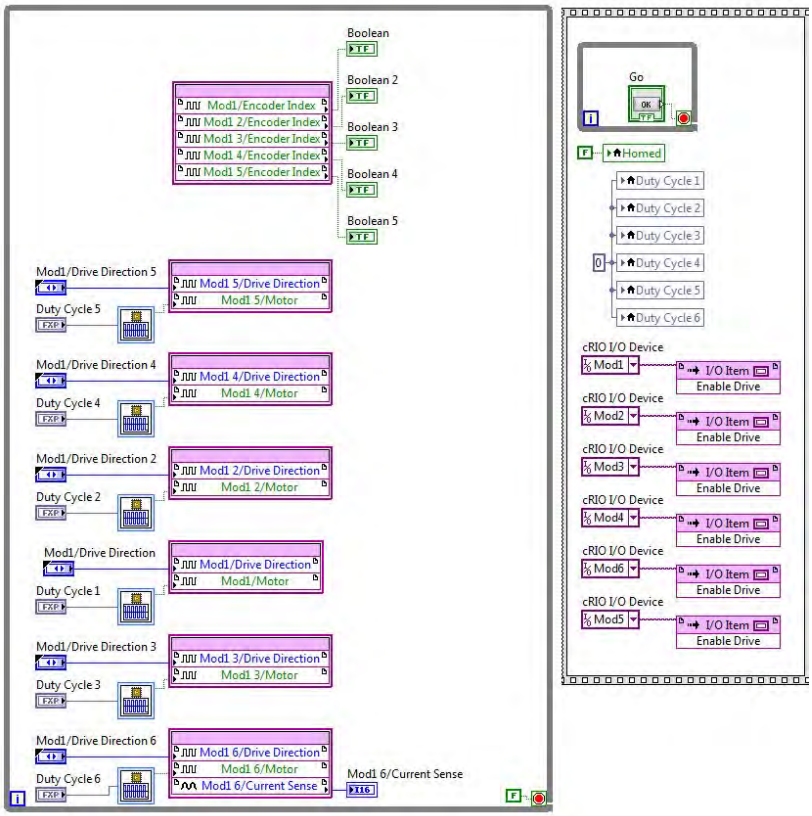

Figure 14: FPGA Example

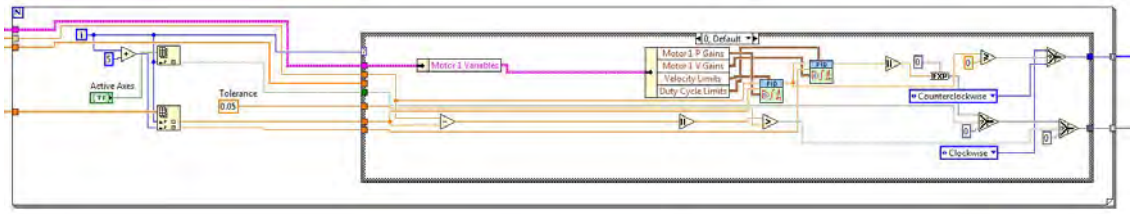

Figure 15: Feedback Control Example

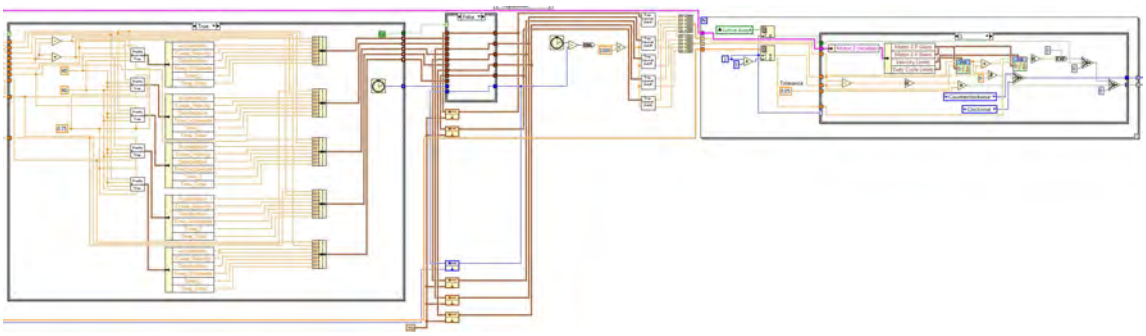

Figure 16: Supervisory Control Example

5.4.5 Supervisory Control

The Supervisory Control layer generates set points for the Feedback Control layer. This layer can use multiple techniques to generate the set points. For example, trapezoidal and S-shaped profiles can be generated for changes in set points. A sample of the code in shown in Figure 16. The Feedback Control and Supervisory Control layers were placed in the same VI on the CompactRIO.

5.4.6 Task Control

The Task Control layer is the highest level of the framework and abstracts the three previous layers away from the user. The level uses state machines to program various tasks and can be implemented on either the CompactRIO or a separate computer system.

5.4.7 Hybrid Simulation

The framework developed encapsulates and simplifies the integration of simulation and real-time control code. However, we need to create a hybrid environment by providing a discrete time environment for the simulations to ensure the simulation and real-time systems can be compared. The following steps show how to convert a subVI into discrete time.

1. Right-click the subVI and select subVI Node Setup.
2. Set Simulation subVI execution type to discrete.
3. Change the discrete timing period (sec) to -1. This sets the discrete time settings of the subVI to the discrete time settings of the simulation and simplifies the process.
4. To verify that the subVI is set to discrete time, check the top left corner of the subVI's icon. The subVI should now have a D on the top left corner of the icon.

Figure 17 shows an example of the discrete time control system for a simple motor with a cascade position control system.

5.4.8 LabVIEW Control & Simulation Module

LabVIEW Control Design and Simulation Module allows you to design control systems, simulate dynamic systems, and deploy control to real-time hardware. The core component of this module is the simulation loop. The simulation loop consists of 3 main parts: the input node, the main loop, and the output node. To set our discrete time steps, we change the parameters in the input node to set our ODE solver, discrete time steps, and simulation time. The input node is shown in Figure 18.

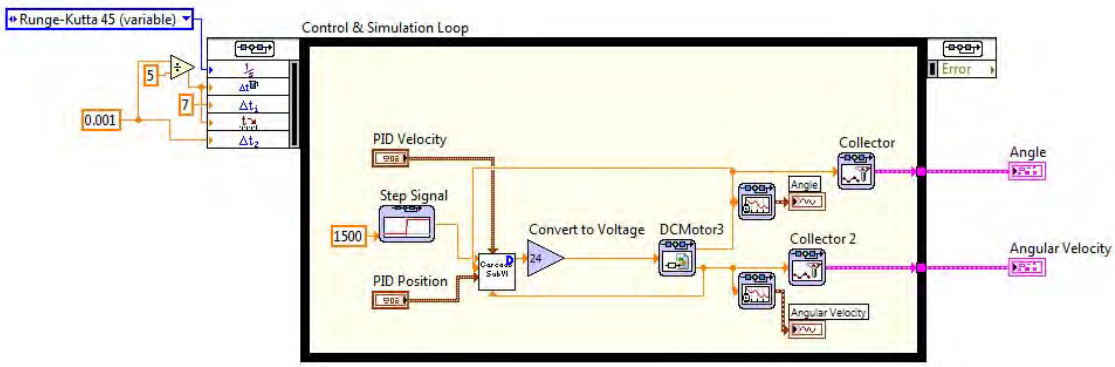

Figure 17: Discrete Time Control Example

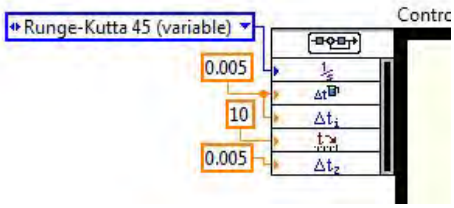

Figure 18: Simulation Input Node Example

5.4.9 Implementing the Control System

It is advised to place each level of control into a subVI to organize and reduce the total size of the VI. For example, the encapsulated control code can be stored within a central subVI and implement multiple levels of control. Depending on the system and application, certain levels of control can be implemented without the other levels. Figure 19 shows a central subVI integrated into the simulation environment.

The outputs of the subVI are fed directly into the FMU with multiple variables controlling model parameters. The FMU then outputs the positions and velocities of each axis. Figure 20 shows the FMU of the five axis robotic arm with different inputs controlling variables such as spring constant and damping. Any system can be implemented in this structure by using the continuous control code, FMU, and real-time variables.

5.4.10 Configuring LabVIEW FMI for Model Exchange

After installing LabVIEW and LabVIEW Control and Simulation Module, follow the next few steps to configure LabVIEW FMI for Model Exchange.

1. Within LabVIEW, open the VI Package Manager. Search for OpenG LabVIEW ZIP Library, and install it and any required dependencies.
2. Open to the file <LABVIEW Directory>/vi.lib/Simulation/EMI/Plugins/FMI for Model Exchange/EMI_FMI Configuration.ini
3. Make sure the file looks like this: [EMI_FMI] UseDefaultUnzipMethod=FALSE

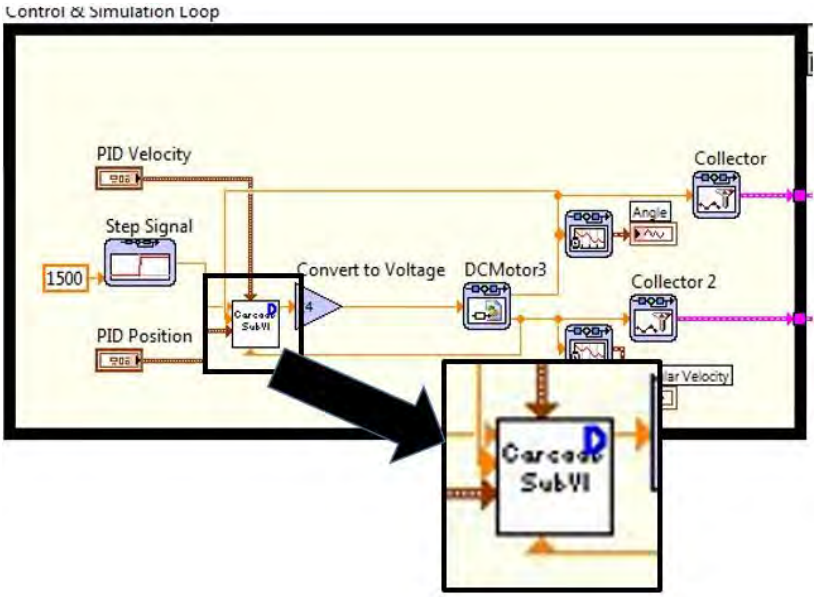

Figure 19: SubVI Example

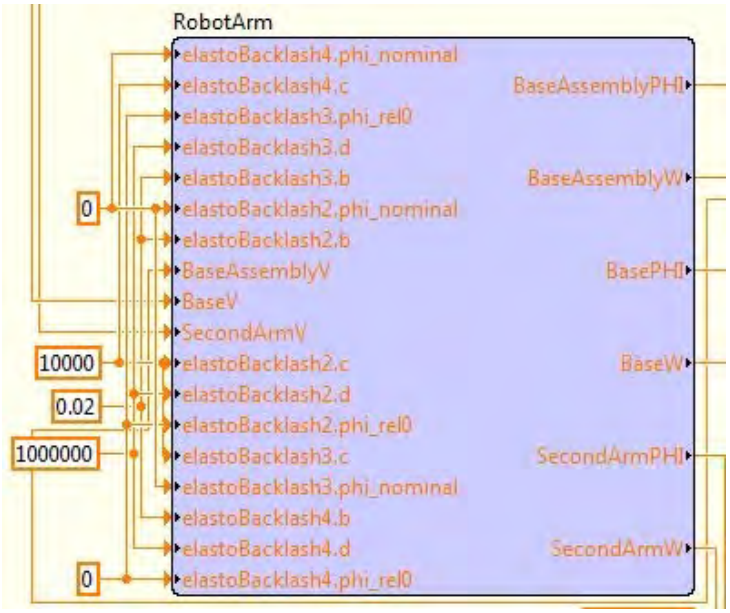

Figure 20: Simulation Framework FMU Example

5.4.11 FMU to LabVIEW

In a newly created LabVIEW VI, navigate to Control Design & Simulation Palette and select Control & Simulation Loop. All simulation functions need to be placed within a Control & Simulation Loop or a Simulation Subsystem. To import the FMU, select the FMI block under External Models Subpalette in the Configuration Dialog. The system parameters, dynamic relations, and input/output terminals will automatically be generated based on the construction of the FMU. In this step, we can also select system parameters to tune. After wiring an input signal to the FMI block and connecting an output terminal to a waveform graph, we can generate a plot of the step response of the system by running a simulation. We can also implement a controller as the input to the FMI block to observe the transient and steady state response of the dynamic system.

5.4.12 Running the LabVIEW Simulation

Here are the steps to creating a LabVIEW simulation with the FMU generated. In this example, we will be observing the step response of a system, "Plant", with input u and output y. We will input a step signal and observe the response via output y.

1. Create a new LabVIEW VI called Example.vi.
2. Place a Control & Simulation Loop from Control and Simulation Palette in the block diagram and drag the FMI block from Simulation Palette into the loop.
3. Select Plant.fmu from the Configuration Dialog. All input/output terminals will be generated automatically.
4. Select the Step Signal block from Signal Generation Palette and connect it to the input terminal on the Plant block. Set the initial value to 0, final value to 9.6, and time step to 0.01 s.
5. Drag the SimTime Waveform block from Simulation - Graph Utility Palette and connect it to output terminal y on the Plant block.
6. Double-click the terminals on the left side of the Simulation Loop and set the initial time to 0 s and the final time to 0.5 s in the Configuration Dialog.
7. Run the VI and observe the step response.

5.4.13 Calibrating Model Specifications

There are two ways of calibrating our model specifications. We can either use the Modelica simulation environment or LabVIEW simulation environment. Both processes require finding component specifications. For most of our testbeds, we do not have manufacturer specifications so we will tune our model by comparing data from simulation with data from the real time system.

It is also a good idea to introduce real-world nonlinearities into the system in LabVIEW. For example, we can add real-world voltage saturation limits to the motor or noise into the sensors. Nonlinearities that do not appear in simulation will most likely occur in the real world.

5.5 Data Visualization

5.5.1 LabVIEW Simulation to Inventor

LabVIEW simulation provides useful data about the position, velocity, and acceleration of all components through the Modelica model. This information can be used to visualize the behavior of the

copyright ©2015 University of California, College of Engineering

complex system. We can do this by exporting this data to Inventor and use the Dynamic Simulation tools available to see the assembly in motion.

5.5.2 Inventor Visualization Tools

Inventor provides a system for simulating and analyzing the dynamic characteristics of an assembly in motion. Although the main purpose of this interface is to define external forces and moments - we will use a separate feature and manually set each component to have a position profile that we received from LabVIEW.

1. Run the LabVIEW simulation and generate graphs for the position of each joint in the front panel.
2. Export the position data by right-clicking each graph and selecting Export Data to Excel.
3. In Excel, save the data as a .txt file. To save on computation time, it may be wise to sample the position data before importing it into Inventor.
4. Now open the CAD model in Autodesk Inventor.
5. Navigate to the Environments tab and select Dynamic Simulation.
6. In the Simulation Player module, select Construction Mode to select specific joints.
7. In the Navigation Panel, double click each joint and navigate to the DOF tab.
8. Click the Edit Imposed Motion image, and select Enable Imposed Motion.
9. Set the driving motion as position. Then, click on the graph on the right.
10. In the pop-up module, within the property of the selected sector box, select the input as a Spline.
11. Click on the arrow to the left and load a spline. Find the .txt file from LabVIEW and save it into Inventor.
12. Confirm the selection and run the visualization tool.

This process will scale even with multiple joints in a system. With this visualization tool, rather than interpreting multiple graphs independently, we can visually verify the behavior of the system as a whole.

5.6 Conclusion

This concludes the description of general procedures used in our workflow. The following testbeds demonstrate our progress in developing and determining the feasibility of our prospective automated workflow.

6 DC Motor Test Stand

6.1 Introduction

6.1.1 Testbed Motivation

In this section, we will explore the end-to-end process of our prospective automated workflow for the design and control of a DC motor. Motors are integral components in many complex mechanical system.

copyright ©2015 University of California, College of Engineering

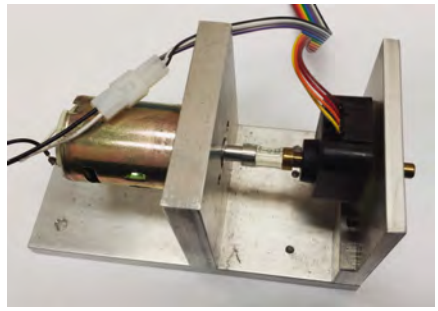

Figure 21: DC Motor Test Stand

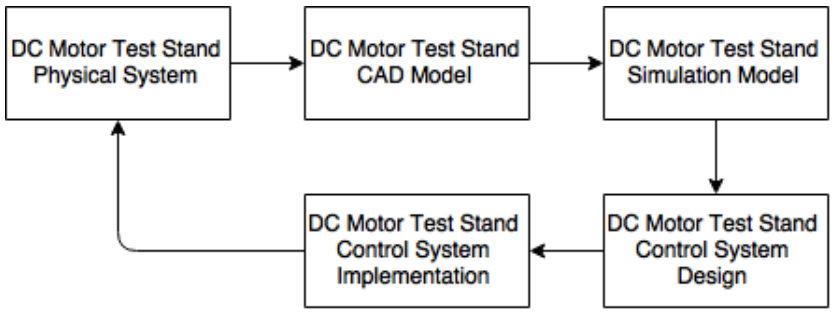

Figure 22: DC Motor Test Stand Workflow Diagram

6.1.2 General Procedures

We will start with a physical DC motor test stand, Figure 21, reverse engineer the process of designing it in Autodesk Inventor, and export details of the system to LabVIEW. Finally, using the exact same control schema, we will compare results of simulations of the motor in Modelica and LabVIEW with the results of connecting the control schema to the physical motor using hardware from National Instruments. We can see a diagram of our prospective workflow in Figure 22.

6.2 Physical Model

6.2.1 Device Characterization

The DC motor test stand consists of 3 main parts: the stand, the motor, and the encoder. The stand is a simple aluminum base that the motor and encoder can mount on.

6.2.2 Component Specifications

The encoder is an E5 Optical Kit Encoder from US Digital which tracks the angle of the shaft, and the motor is a direct current permanent magnet brush motor with the observed specifications shown in Table 2

copyright ©2015 University of California, College of Engineering

DC Motor Specification	Value
Nominal Voltage	24V
Nominal Current	0.365A
Nominal Angular Velocity	1000 rad/s
Inductance	0.514 mH
Resistance	4 ohm
Back-EMF Constant	0.0208 V/rad/s
Torque Constant	0.0208 Nm/Amp
Viscous Damping Coefficient	1.65E-5

Table 2: Observed Motor Parameters

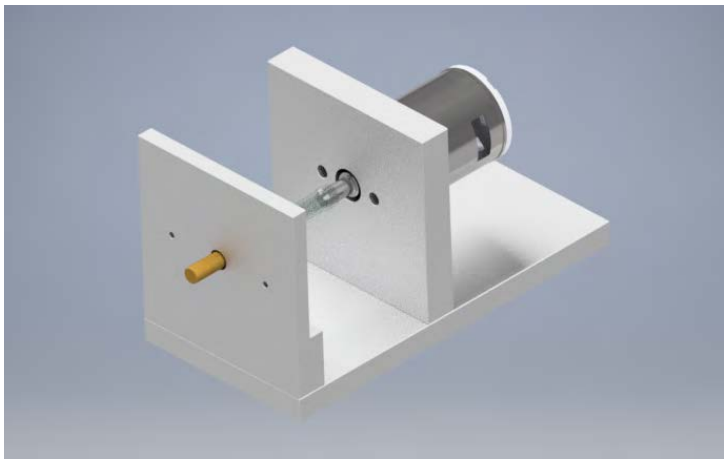

Figure 23: Isometric View of the DC Motor Test Stand

6.3 Inventor Model

6.3.1 Detailed Inventor Model

A model of the DC motor test stand is created in Inventor using dimensions from the test stand and importing an existing CAD model for the encoder (measures rotor angle) from US Digital. The DC motor is approximated using its external shell casing without the internal components. Each component is constrained with appropriate relationships and materials. The model is shown in Figure 23. The motor is attached to the vertical plate on the right and the encoder is just slightly visible on the vertical plate to the left. In work that follows the encoder will be modeled as an ideal angular position measurement.

6.3.2 Inventor Model Properties

The assembly of the DC motor test stand consists of two components: the fixture (test stand, encoder, and motor casing) and the axle of the motor. Because the fixture is a fixed component, there is only one rotational degree of freedom in the model.

copyright ©2015 University of California, College of Engineering

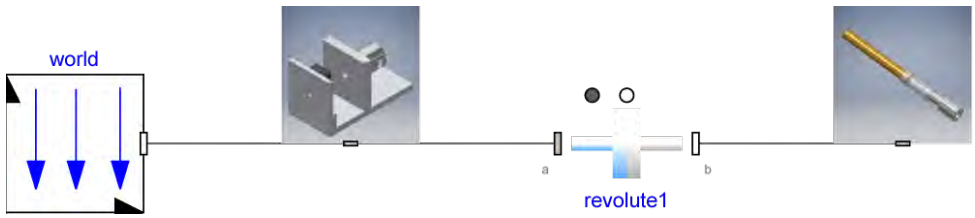

Figure 24: Result of Autodesk Inventor-to-Modelica Translation Software

6.4 Modelica Model

6.4.1 Applying Modelica Standard Library

One approach to generating an FMU for LabVIEW is to recreate a model for the DC motor test stand in Modelica. We can use a direct current permanent magnet motor from Modelica's library of electrical models with the appropriate motor specifications and inertial load or create our own motor model written in Modelica. We can also use Modelica's MultiBody library for a fixed constraint to position the motor in space. In general, this process may not scale well for more complex electro-mechanical systems as it may not be reasonable to model the same system multiple times.

6.4.2 Inventor-to-Modelica Translation Software

Another approach is to use specialized software from Autodesk to translate our CAD model directly into a Modelica model. The translator is able to interpret the constraints between the axle of the motor and the fixture (a cylindrical and a planar constraint) as Modelica's MultiBody revolute joint. It also embeds the current position and the mass property of components as Modelica's MultiBody bodies.

6.4.3 Adding Multimedia Elements

The translator does not recognize if the model contains electrical components. To add electrical components, we can create a motor model based on the motor parameters observed from the physical model and add an angle sensor to model the encoder. Another problem with the Modelica model is that the translator converts the rigid bodies from Inventor into point masses and removes properties such as rotational inertia. So we manually take the rotational inertia from our Inventor model and apply it to a rotational inertia element for the axle of the motor. We can also add friction forces and signal limiters to model our system more accurately. The transformation is shown in Figure 24 and Figure 25 below. It is key that components in the model maintain the input and output properties in the same units as in the original system. In this example, the system input is voltage to the motor and output is the angle of the shaft from the encoder.

6.4.4 Modelica Modeling Assumptions

In this example, rather than using the DC permanent magnet motor from the Modelica Standard Library, we created a custom model of a motor based on the two following governing equations:

$$I(t)K_t - B\frac{d\theta}{dt} = J\frac{d^2\theta}{dt^2} \tag{1}$$

$$V_{app} - K_{emf}\frac{d\theta}{dt} - RI(t) = L\frac{di}{dt} \tag{2}$$

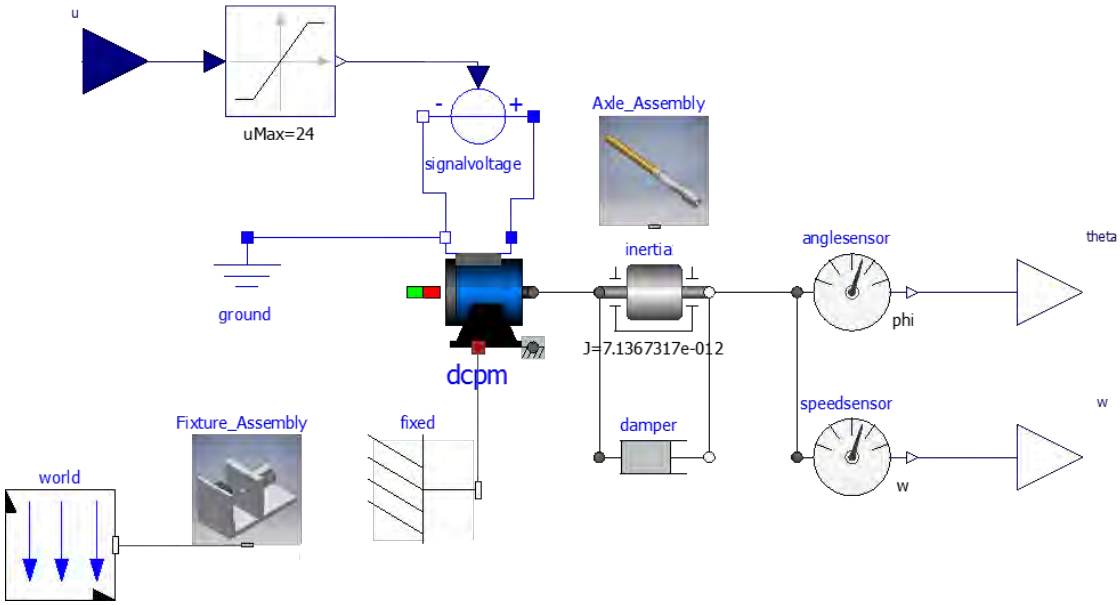

Figure 25: Augmented Modelica Model with Motor and Encoder

This allows us to directly measure certain properties of our motor and test the robustness of custom models.

6.4.5 Modelica Simulation Environment

Modelica also allow us to simulate the motor with a variety of inputs. For example, we can choose to input a step voltage to the motor and observe the angular position of the axle. An example of this is shown in Figure 26. To separate the plant from the controller, we can use real input and output connection terminals. Real input and output terminals interface between multiple files and allows us to reuse the same controller for various plants. In the simulation, Modelica displays various properties of each component throughout the simulation such as position, velocity, and acceleration. We will compare this simulation against the simulation in LabVIEW and the response of the physical DC motor test stand using the same controller.

6.4.6 Modelica to FMU

Using JModelica, we were able to convert our model from a Modelica model into an FMU without any problems. Note that the FMU must adhere to FMI 2.0 Model Exchange protocols to be accepted by LabVIEW. The model did, however, have trouble exporting to FMU using OMEdit. Certain models from the Modelica Standard Library seem to have problems adhering to the FMI standard.

6.5 LabVIEW

6.5.1 FMU to LabVIEW

In a new LabVIEW VI, we can import the FMU as an FMI block and wire an input signal, voltage, and an output signal to a waveform graph to generate a plot of the step response of the system. We can also implement a controller as the input to the FMI block to observe the transient and steady state response of the dynamic system.

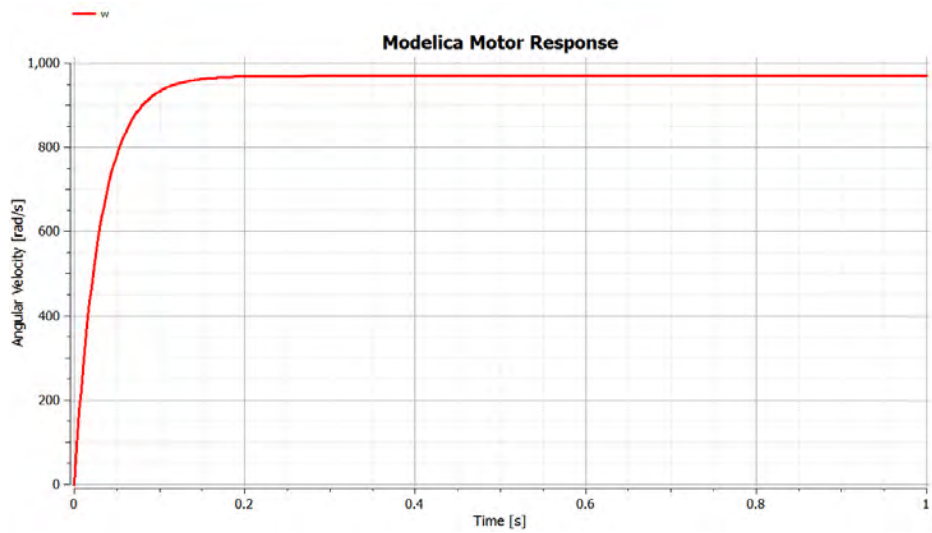

Figure 26: Motor Response to a 24 Volts Step Input

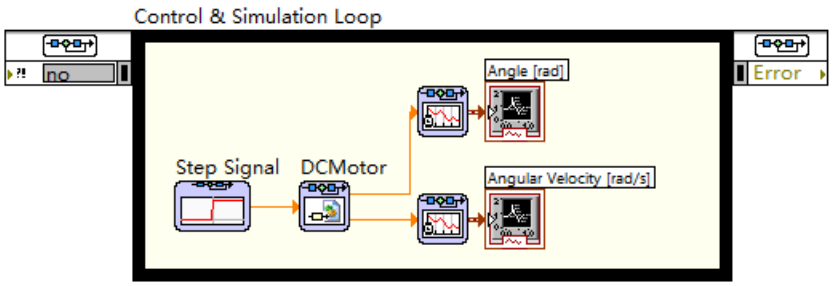

Figure 27: Block Diagram of the LabVIEW Simulation VI

6.5.2 LabVIEW Simulation

To simulate the open loop step response of the DC motor test stand, we add a simple 9.6 volts step input and observe the angle and angular velocity outputs from the DCMotor block. We create the block diagram shown in Figure 27 and observe the responses shown in Figure 28 and Figure 29.

6.6 Control System Design

Now that we have the motor modeled in multiple environments, we can begin to design a controller for our DC motor test stand in Modelica and LabVIEW. In this section, we will be be developing a cascade position controller for the motor. By using a cascade position controller, we will be able to tune the profile of the motor's angular velocity while also controlling the position. A block diagram of the system is shown in Figure 30.

6.6.1 Controller Designs

In this section, we will be implementing a cascade position control system. The cascade position control system consists of an upper-layer position control and a lower-layer velocity control. The position control is a simple P control with an input of desired position and an output of velocity

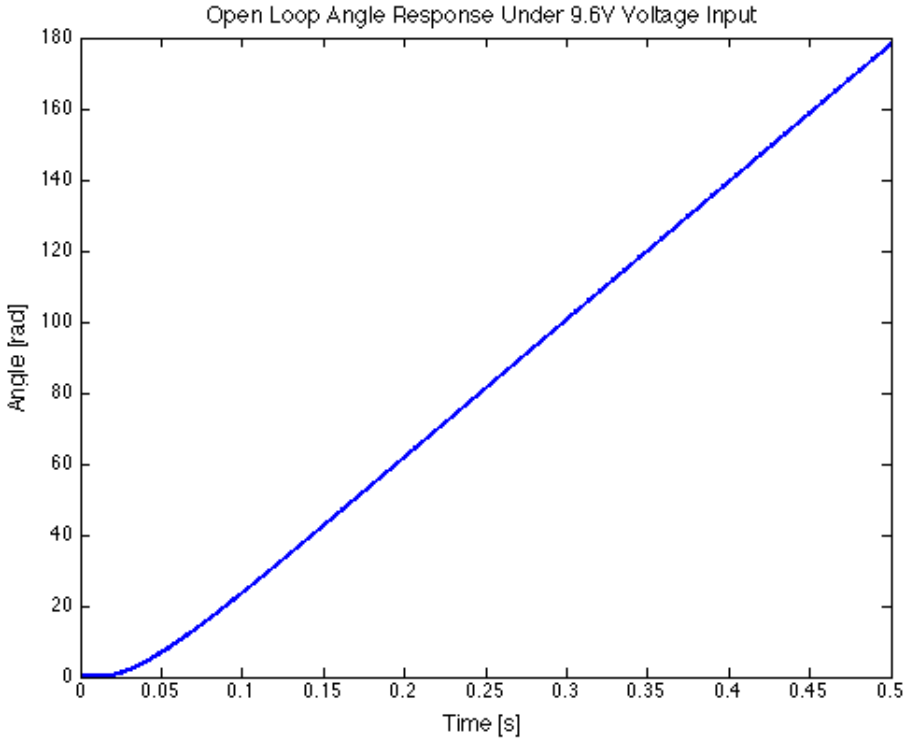

Figure 28: Angle Response Under a Step Input of 9.6 V

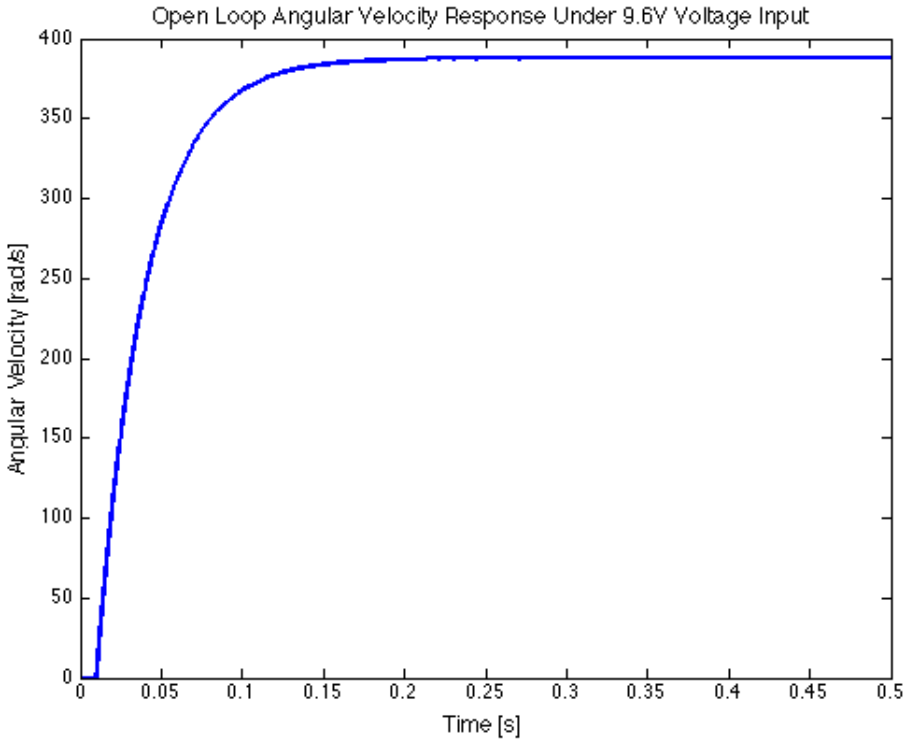

Figure 29: Angular Velocity Response Under a Step Input of 9.6 V

copyright ©2015 University of California, College of Engineering

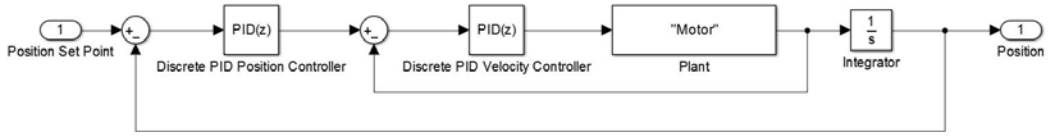

Figure 30: Block Diagram of Motor and Cascade Position Controller

setpoint. The velocity control is a PI control which takes in the desired velocity and outputs duty cycle. An integral control is necessary for the velocity control because it drives the steady state error of velocity to zero. Saturation limits are put on the velocity and duty cycle outputs to avoid exceeding the system's actuation limits.

6.6.2 Control System Design in Modelica

With the Modelica model of the DC motor test stand, we can move forward with the process of also designing a controller in Modelica. To do so, we can either directly modify the original Modelica model or separate the controller from the motor using files connected with input and output terminals. It is advantageous to separate the system from the controller so that the files can be transferred between software environments, and controllers can be reused and tuned without accessing the system.

6.6.3 Separating the Controller from the Plant

To create our controller in a separate file, create a new model with the control algorithm. Then to link the controller to the system, either reference the system using "DCMotorTestStand sys" in the Modelica code or drag and drop the DCMotorTestStand block into the system using the visual editor in OpenModelica's OMEdit. By declaring system as a variable in the controller, the controller can reference the system's input and output variables as instance variables. For example, if the system has input u and output y, we can get input or output these values by calling sys.u or sys.y. Finally, we add our control algorithm in a discrete time environment by setting a sampling time and controlling the input into the system using a PID function with the appropriate parameters. Sample controller code is shown below in Figure 31. We have separated the PID block from the model so we can reuse it for different systems.

6.6.4 Manually Tuning the PI Controller

With the control code setup in the Modelica environment, we can now tune the gains of the PID controller iteratively. For the motor, we will be using a PI controller because the derivative may have too much noise to accurately tune. Overall we have two values to choose and can perform multiple simulations to match the profile of the results to our desired profile. After setting our target velocity at 50 radians per second and Kpv = 0.022, kiv = 0.96, kdv = 0.0, we were able to reach the desired velocity and steady state conditions within 0.2 seconds. For the cascade position controller, we set our desired position at 600 radians with Kpv = 0.022, kiv = 0.96, kdv = 0.0 and Kpv = 2.5, kiv = 0.0, kdv = 0.0. The motor was able to ramp up to its final position of 600 radians within 2.5 seconds as shown in Figure 32.

```
model MotorVelCntrl
  DCMotorTestStand sys;
  Real velSet = 50;
  parameter Real kpv = 0.022, kiv = 0.96, kdv = 0.0;
  parameter Real delT = 0.005;
  Real mVel;
  Real integVel, valPrev;
algorithm
  when sample(0, delT) then
    (mVel, integVel, valPrev) := PIDFunc(dt = delT, kp = kpv,
        ki = kiv, kd = kdv, integ = integVel, setpoint = velSet,
        val = sys.w, first = false);
    sys.u := mVel;
  end when;
end MotorVelCntrl;
```

Figure 31: Separated Control Code in Modelica

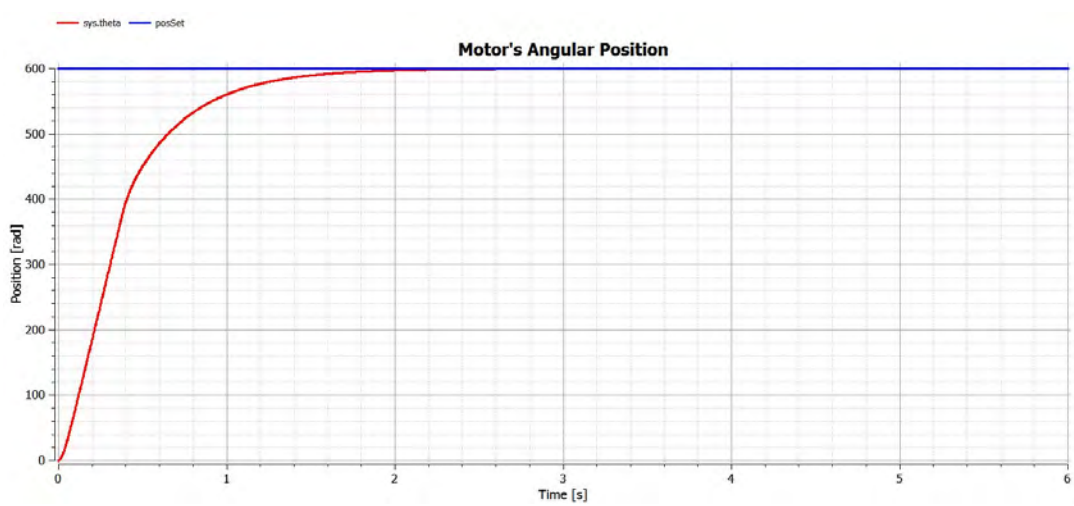

Figure 32: Cascade Position Controller and Motor Response

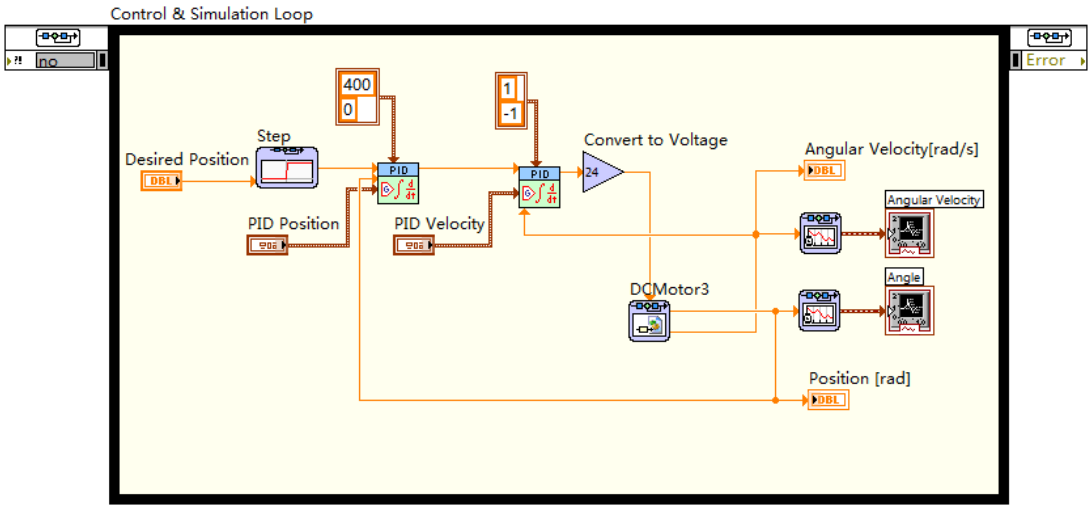

Figure 33: Block Diagram of LabVIEW Simulation

6.6.5 Control System Design in LabVIEW

The same cascade control is implemented in both LabVIEW Simulation and Real-Time environment. Figure 33 and Figure 34 show the block diagrams of the LabVIEW Simulation and Real-Time respectively. We set our desired position at 1500 radians with Kpp = 1.5 and Kvp = $6E-4$, Kvi = $4E-4$. We also include the saturation limit for the position controller to be 0 to 400 rad/s and the saturation limit for the velocity controller to be 0 to 1.

6.6.6 Comparing Simulation and Real Time Results

The position response and velocity response of the simulation and real-time are shown in Figure 35 and 36. It is observed that the position responses of the simulation and real time basically overlap each other and have the same settling time. The graph of the motor's velocity response also shows very similar characteristics but the simulation has a slight overshoot at the start which exceeds the saturation limit. Also, the physical motor appears noisy when it approaches speeds lower than 30 radians per second.

6.6.7 Trapezoidal Profile Implementation

We will now implement a trapezoidal profile control for point-to-point moves. A trapezoidal profile improves the system's behavior by using three phases of motion: constant acceleration, constant velocity and constant deceleration. For the motor, the rotor is accelerated from rest to a constant velocity then decelerates to the rotor's final position, where the final acceleration and final velocity are zero. Here, we implement the trapezoidal profile to the motor in both Simulation and Real-Time control. To quantify the profile, we set the following parameters: Desired position = 1000 radians, Acceleration = 100 radians per second squared, Deceleration = 100 radians per second square, Vcruise = 200 radians per second, and MinFracCruise = 0.3. The plots in Figure 37 and Figure 38 compare the position profile and velocity profile of the Simulation and Real Time control with the desired profiles.

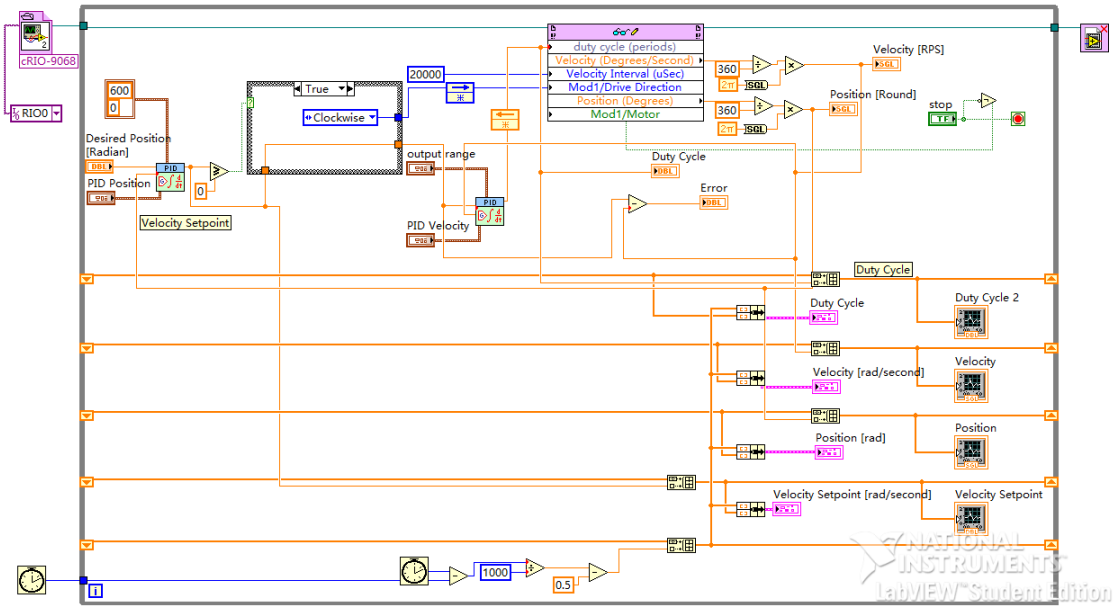

Figure 34: Block Diagram of LabVIEW Real Time

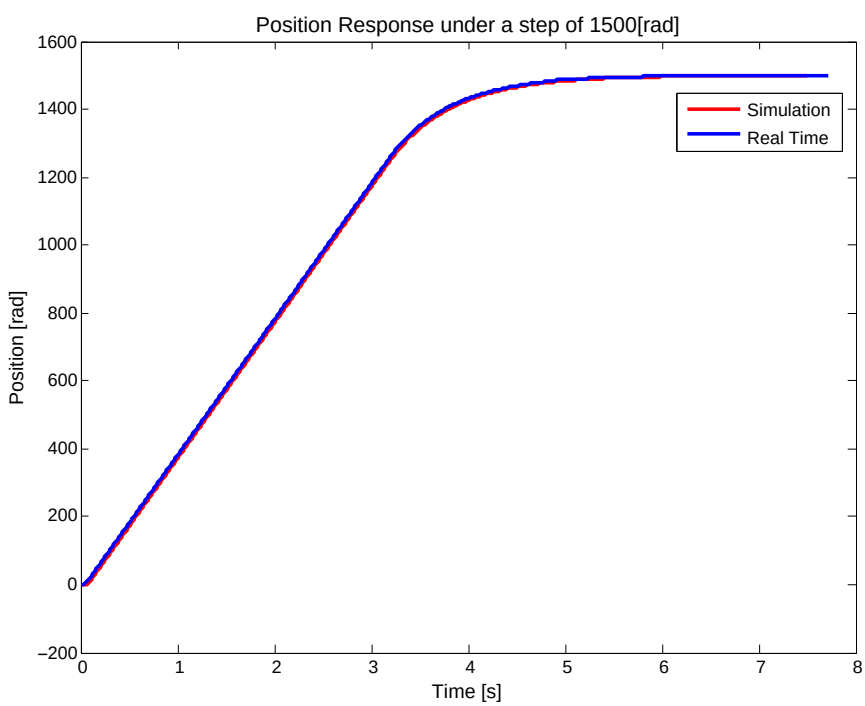

Figure 35: Position Response of the Simulation and Real-Time under a Step of 1500 rad

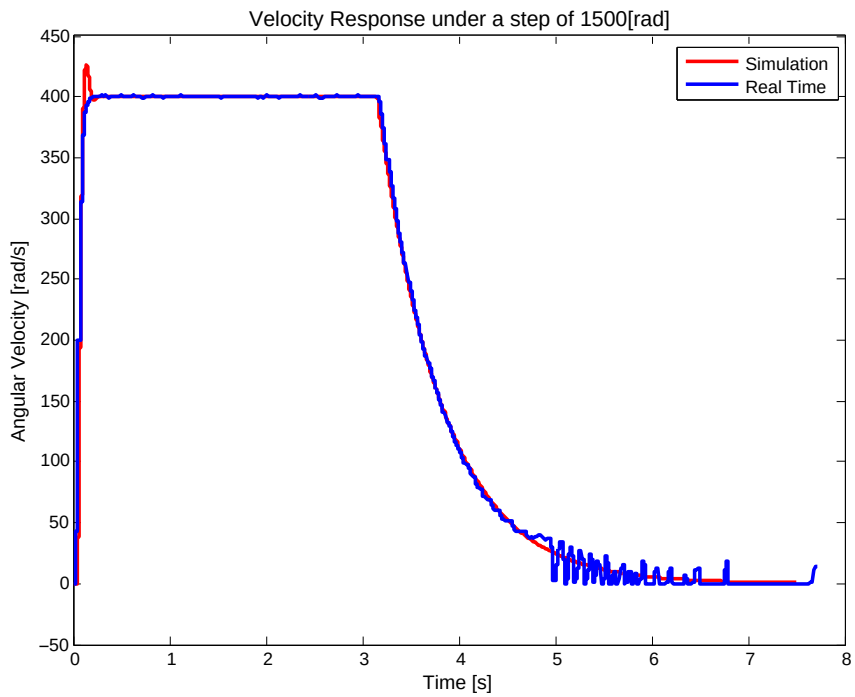

Figure 36: Velocity Response of the Simulation and Real-Time under a Step of 1500 rad

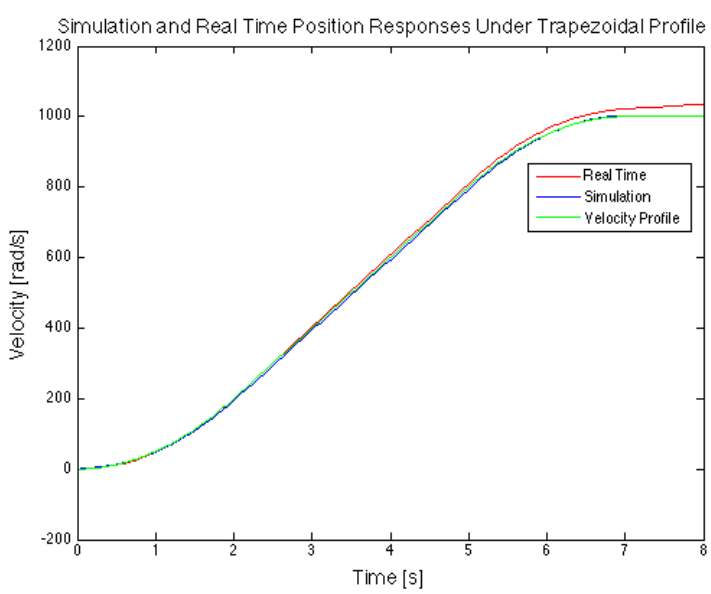

Figure 37: Position Profile of the Simulation and Real-Time Control

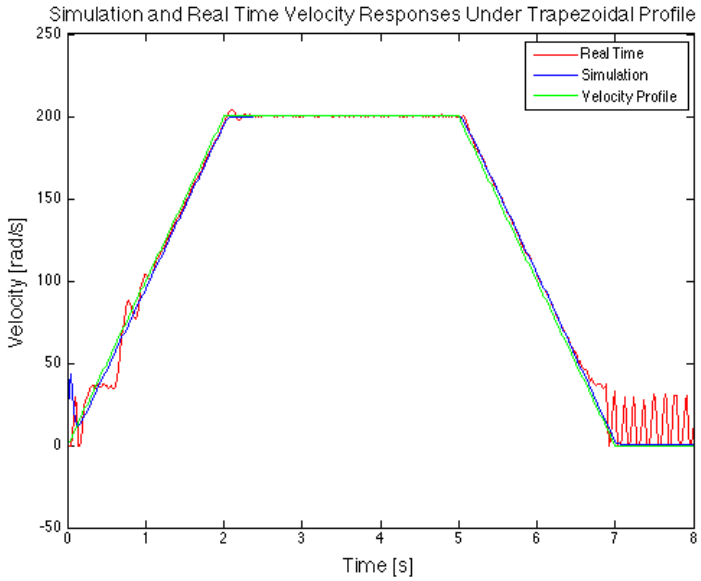

Figure 38: Velocity Profile of the Simulation and Real-Time Control

6.7 Data Visualization

6.7.1 LabVIEW Simulation to Inventor Dynamic Simulation

Using the Inventor Dynamic Simulation tools, we can take the data we received from LabVIEW and input a profile for the position of the DC Motor axle. In the Inventor model, we add a small flywheel so that we can see the DC Motor rotate more clearly.

6.8 Conclusion

6.8.1 Workflow Verification

In conclusion, we have been successfully able to show the entire end-to-end process for the design of the DC motor test stand. We started with a physical model, reversed engineered the process of designing it in Autodesk Inventor, and finally simulated and designed a controller for the motor in Modelica and LabVIEW. The controller for the simulation was also used to control the physical system using hardware from National Instruments.

6.8.2 Potentially Automated Features

This example highlights our need for adding multimedia components. Modelica provides a great way of doing this using their existing libraries but users will need to be able to produce custom parts or models for specific components that may be represented differently or not available.

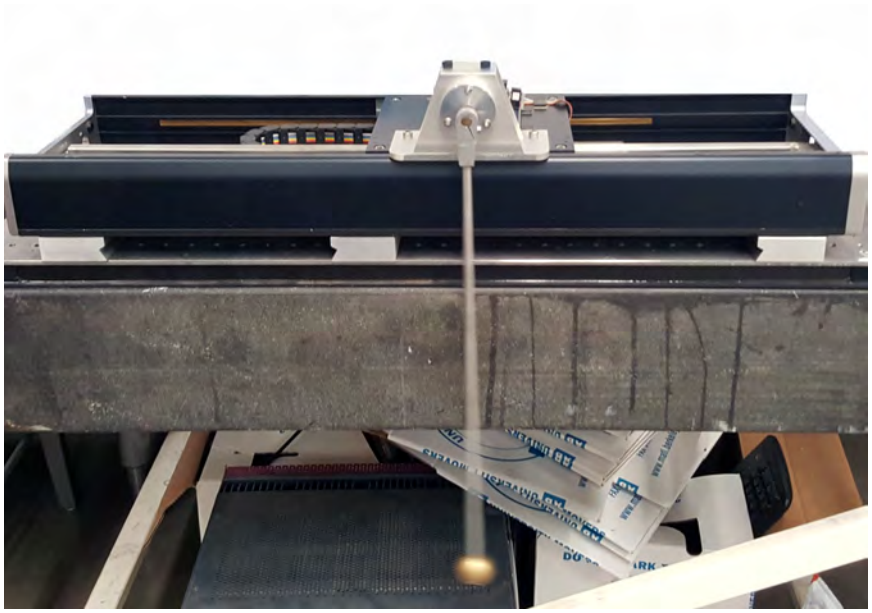

Figure 39: Inverted Pendulum Using High Precision Stage

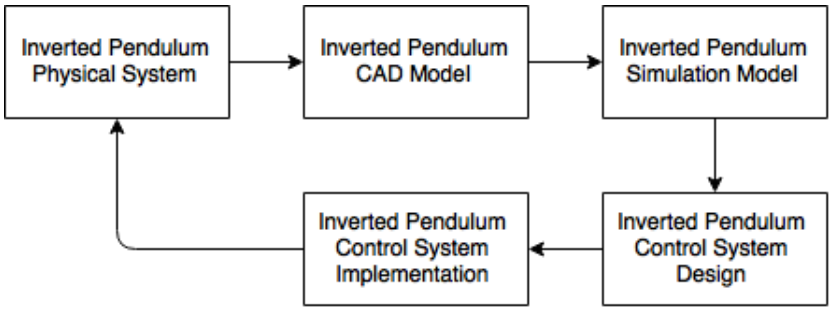

Figure 40: Inverted Pendulum Workflow Diagram

7 Inverted Pendulum

7.1 Introduction

7.1.1 Testbed Motivation

In this section, we will explore the end-to-end process of our prospective automated workflow for the design and control of an inverted pendulum (Figure 39). The inverted pendulum is an inherently unstable system and a classic problem in dynamics and control theory.

7.1.2 General Procedures

Like the DC motor test stand, we will start with a physical model of the inverted pendulum system, reverse engineer the process of designing it in Autodesk Inventor, and export details of the system to LabVIEW. Finally, using the exact same control schema, we will compare results of simulations of the system in Modelica and LabVIEW with the results of connecting the control schema to the physical system. We can see a diagram of our prospective workflow in Figure 40.

copyright ©2015 University of California, College of Engineering

7.2 Physical Model

7.2.1 Device Characterization

The inverted pendulum system consists of three main parts. The pendulum, the stage, and the base. The stage slides along the base and the pendulum rotates about a point on the stage. The pendulum is a aluminum rod with a brass weight attached at the bottom.

7.2.2 Component Specifications

There are two important electronic components in this complex mechanical system. The sliding stage is an industrial Bell Everman linear motor with a linear encoder to determine its position. It also uses a US Digital optical encoder to track the angular position of the pendulum.

7.3 Inventor Model

7.3.1 Detailed Inventor Model

A model of the inverted pendulum is created in Inventor using dimensions from the fixture. The assembly was separated into three main components: the base, which is a fixed assembly with weighted components, the linear stage, which is constrained to travel horizontally along the base, and the pendulum, which is pinned using a revolute joint onto the linear stage. The linear motor is approximated by a large beam, and a model of the US Digital optical encoder was supplied by the manufacturer. Weights are taken from each component to assure that the weight of the model corresponds to the weight of the physical system. The model is shown in Figure 41.

Figure 41: Isometric View of the Inverted Pendulum Assembly

copyright ©2015 University of California, College of Engineering

7.4 Modelica Model

7.4.1 Inventor-to-Modelica Translation Software

A Modelica model of the pendulum was created using the Inventor-to-Modelica Translation Software. The translated model is shown in Figure 42. The translator was able to convert all of the constraints from the CAD model correctly. To set the pendulum and stage in various positions, we need to modify the position vectors for the components in Modelica or adjust the Inventor model and translate the inverted pendulum assembly again. These values are difficult to calculate without modifying the original CAD model because the position vectors point from the origin of the original Inventor model to the object's center of mass in space.

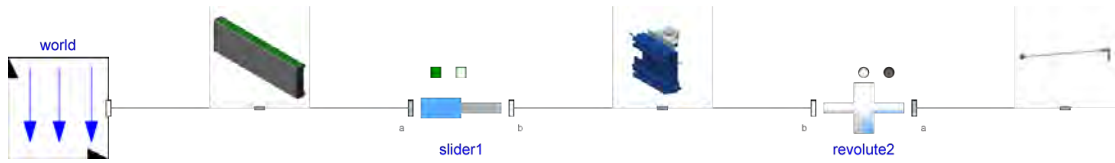

Figure 42: Translated Inverted Pendulum Modelica Model

7.4.2 Modelica Model Verification

We can use a Modelica simulation to verify the state of our system before adding additional components. We will add position and velocity sensors to the exported Modelica model on both the revolute joint and the prismatic slider joint. The Modelica Standard Library provides both relative and absolute sensors for translational and rotational movement. Typically, the initial conditions of the system are referenced as 0 but the initial condition of the prismatic slider joint was determined by the translator.

In this simulation, we observe that there is oscillation in the pendulum despite constraining the pendulum in a stable vertical position. These oscillations occur on the order of 10^{-5} radians. Ideally there should have a way in Inventor or Modelica to place the model into a stable equilibrium to prevent unexpected transient responses. We also observe that the Modelica model is more accurate if you constrain parts to coordinate planes in Inventor rather than in relation to other components. With these changes to the Inventor model, our stable inverted pendulum model ends up with oscillations on the order of 10^{-15} radians rather than 10^{-5} radians. This is very good considering the tolerance of the solver is 10^{-6}.

Within the revolute and slider joint Modelica blocks, there are parameters that can be used to describe the initial conditions of the inverted pendulum. By default, these initial conditions are set to 0. We were only able to change the initial velocities of both joints and were not able to describe the initial position or acceleration using this procedure.

7.4.3 Adding Multimedia Elements

Now that we have the core elements from our Inventor model translated into a Modelica model, we add the multimedia elements and relationships not captured by the translator: electronics, friction forces, and input and output terminals.

The inverted pendulum system has two main electronic components: the linear motor and the optical encoders. As shown in Figure 43, the linear motor is approximated by a conversion of the input current into a force on the prismatic joint between the base and the stage. To calibrate this

model, we will need to experimentally create a conversion factor and create limits on the current inputs. In the Modelica model verification process, we placed ideal sensor at all joints that can act as encoders for the model.

The next step is to add friction forces between each component at the joints. This will help remove the some of the unexpected transient responses seen in the Modelica model verification process. Friction forces will also need to be tuned using experimental data from our system.

Finally, input and output terminals are added for access to key values in LabVIEW environment. The input to the system is current into the linear motor, and the outputs are position and velocity for the stage and the pendulum. These terminals match the inputs and outputs of the physical system.

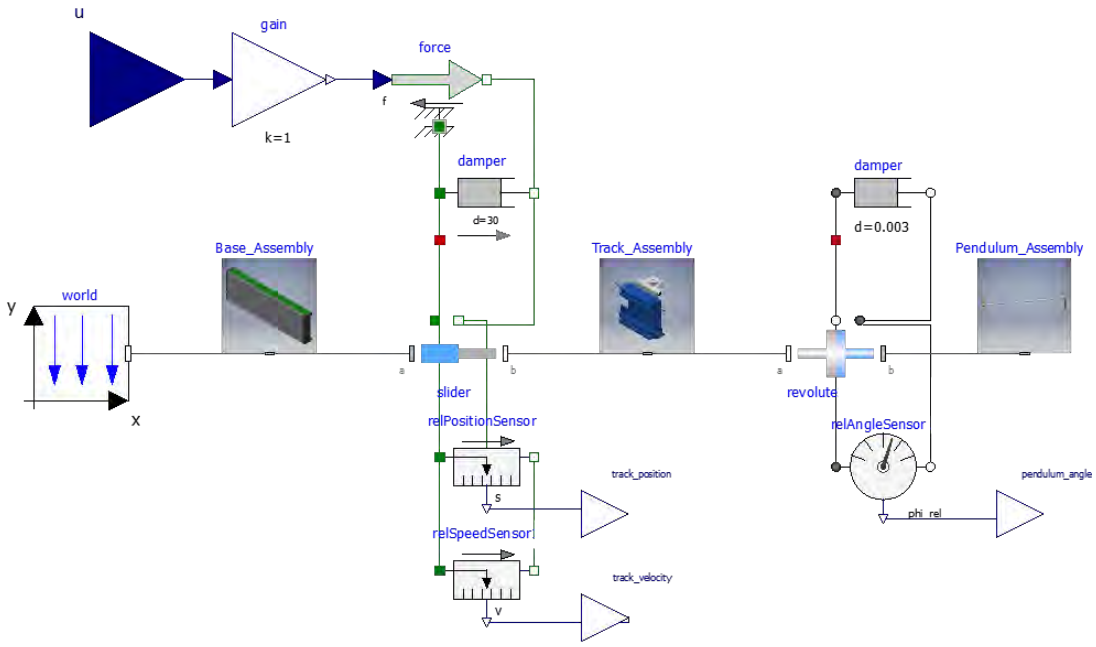

Figure 43: Augmented Inverted Pendulum Modelica Model with Dampers

7.4.4 Modelica Modeling Assumptions

Many approximations were made while adding multimedia elements. For example, the model of the linear motor hides many characteristics about the motor that may be important to us depending on the mechanical system we are designing. For instance, the velocity of the stage is not limited in our model.

There are several ways to model the fiction forces on the pendulum. The most complete models are the Modelica Standard Library support friction and bearing friction blocks for the prismatic and revolute joints respectively. Support friction and bearing friction require several parameters to execute. These parameters include the velocity dependent friction coefficient, the Coulomb force, the Stribeck effect, and the exponential decay of the friction force. Another approach to modeling friction is to just use a linear velocity dependent damping block. This requires only one damping coefficient for each joint but we may see residual oscillations without static friction components.

7.5 LabVIEW

7.5.1 FMU to LabVIEW

We export our Modelica model using JModelica into an FMU to transfer the model into LabVIEW. We can see our imported model in a Control and Simulation Loop in Figure 44. Since we have the physical inverted pendulum system, we want to calibrate our simulation model to be as accurate as possible. There are two platforms that can be used to calibrate our model specifications: the Modelica simulation environment or LabVIEW simulation environment. In this example, we import the FMU to LabVIEW and adjust all system parameters in the LabVIEW environment. The system parameters we will be adjusting include the current-to-force gain and friction coefficients.

Some things to note is that certain Modelica blocks did not transfer over to LabVIEW. For example, we had difficulties including support friction, bearing friction, dead-zone, and other nonlinear blocks. Although including these blocks did not stop JModelica from producing an FMU, the results from the LabVIEW simulation and the results from the Modelica simulation did not match up. For example, in the Modelica simulation the support friction blocks work as designed and inhibit motion until the input current cross a certain threshold. In the LabVIEW simulation, the support friction blocks lock the position of the components in place regardless of the magnitude of the current input.

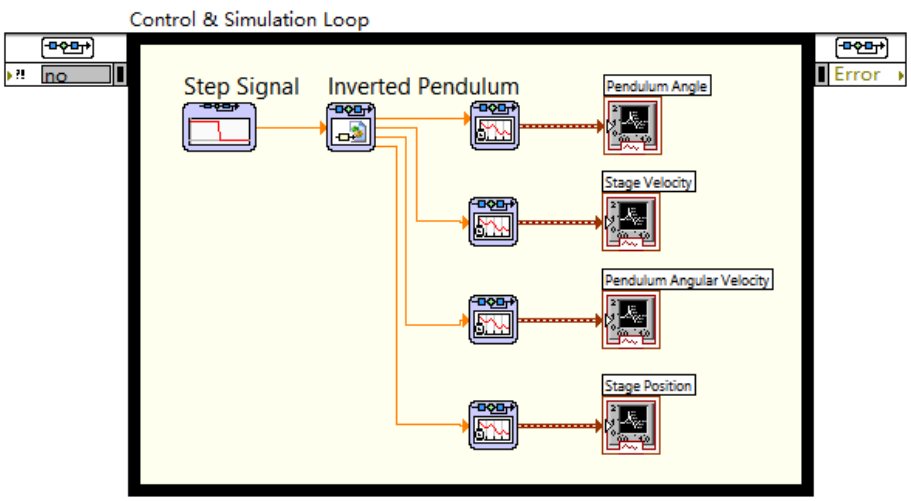

Figure 44: Imported Model in LabVIEW

7.5.2 LabVIEW Simulation

In LabVIEW, we can use simulation to tune the specifications of our system. Since we have the physical system, we can perform experiments comparing the simulation and the real time system to get accurate models of each component. For the inverted pendulum system, we will tune the linear motor gains and friction coefficients.

7.5.3 Calibrating Model Specifications

We can approximate a damping coefficient for the pendulum by observing the angular position of the pendulum of the real time system. We set the pendulum in a horizontal position, increase the damping on the stage, and tune the damping coefficient for the pendulum. A graph of both responses

are shown in Figure 45. We determine that the coefficient of friction for damping on the pendulum should be around $0.003\frac{N}{m^2}$. The frequency of oscillation is similar but the simulation lags behind the real time response. We cannot tune the frequency of oscillation without modifying the center of mass of the pendulum in the Modelica model.

We approximate the damping coefficient for the stage with a high value $5\frac{N}{m^2}$. Note that the stage does not move very much because there is a lot of friction between the base and the stage. To determine the gain from current-to-force for the linear motor, we graph current versus acceleration of the stage and the value of kinetic friction. Using Newton's first law and taking the friction into account, we map to the force and linearly approximate the gain required from current-to-force for the linear motor. We add a nonlinear block to limit the input of our signals to limit the forces on the stage in LabVIEW. This entire process could be automated by taking the actual data set and iteratively adjusting the damping until we have minimum error between data sets. For the linear motor, we would ideally like to create a Modelica linear motor block that has the same specification options from the linear motor manufacturer.

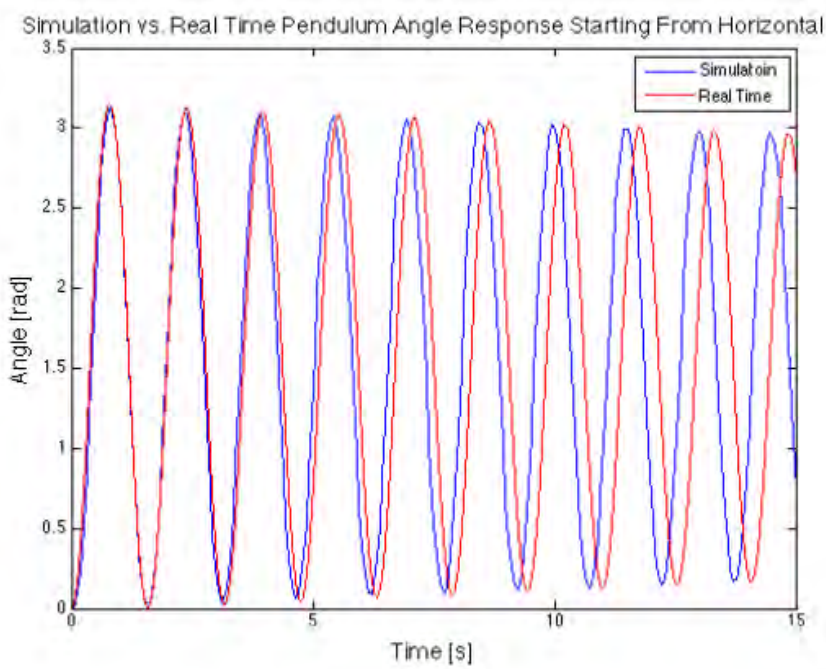

Figure 45: Tuning the Pendulum's Damping Coefficient

7.5.4 LabVIEW Model Verification

Ideally the LabVIEW simulation environment should behave the same as the Modelica simulation environment. Since LabVIEW is unable to handle static friction forces from our FMU, we increase the damping friction coefficients to compensate for effects of static friction. This problem may have been caused by process of converting a Modelica model to an FMU or importing an FMU from LabVIEW.

In the transient analysis of the pendulum falling from a horizontal position, we see accumulated error in the frequency of the oscillations. We have an effective pendulum length that is shorter than the true pendulum length and/or the viscous friction captured does not reflect the true friction forces on the pendulum because we did not model static friction against the pendulum axle.

7.6 Control System Design

7.6.1 Controller Designs

With our model calibrated, we now design control systems to simulate more complex features of our system and verify its behavior. We design a closed loop step response of the inverted pendulum with a PD controller and a step position input. The LabVIEW block diagram for the simulation is shown in Figure 46. Our set point for position is set at 0.05 meters and we tune the Kp and Kd values for the control system. Once we have our desired response in simulation, we move the control system to the real time system and gauge the accuracy of the simulation.

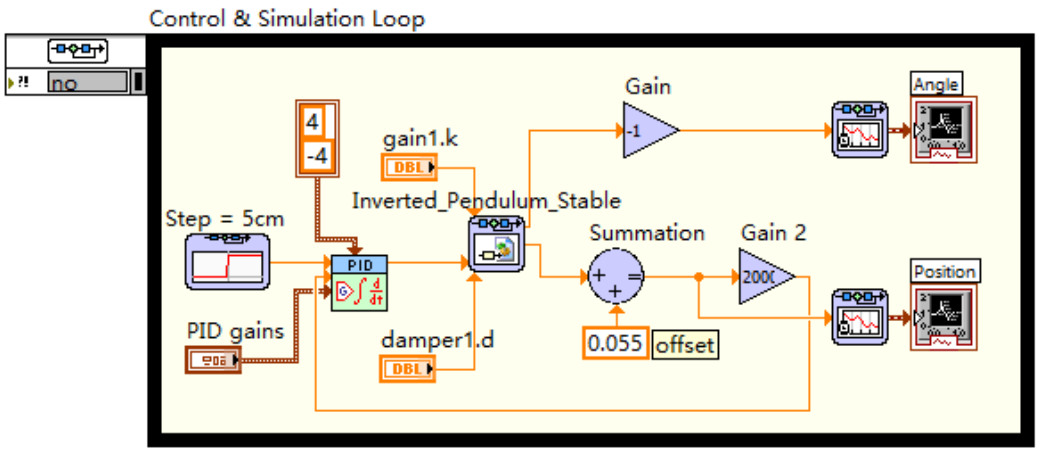

Figure 46: Inverted Pendulum Simulation Diagram

7.6.2 LabVIEW RIO Architecture

On the FPGA level, we have three while loops running at the same time. One of the while loops implements quadrature counters and computes the linear position and angle using digital signals A and B of a quadrature encoder. In the second while loop, data is read from or written to hardware at a fixed rate. The third while loop implements a PD controller and outputs drive command to the motor. On the real time level, the LabVIEW program allows us to select PD controller gain values and plots the position and angle responses with respect to time.

7.6.3 Closed Loop Step Response Comparison

We can compare the closed loop step response of the simulation and the response of the real time system. The graphs comparing angle and position are shown in Figure 47 and Figure 48. The systems use the exact same control system. Again, we can see that the pendulum position lags behind and the damping on the pendulum is a bit higher than expected in simulation. The position of the stage is more damped than the real time system as well.

7.6.4 Balancing the Inverted Pendulum

After observing the closed-loop step response, we move to a more complex controller design. We will create a control system to balance the inverted pendulum. We start with the simulation to get the full state feedback gains for position, velocity, angle and angular velocity. The block diagram of the simulation is in Figure 49. Since the Inverted pendulum is set to be perfectly vertical or zero radian

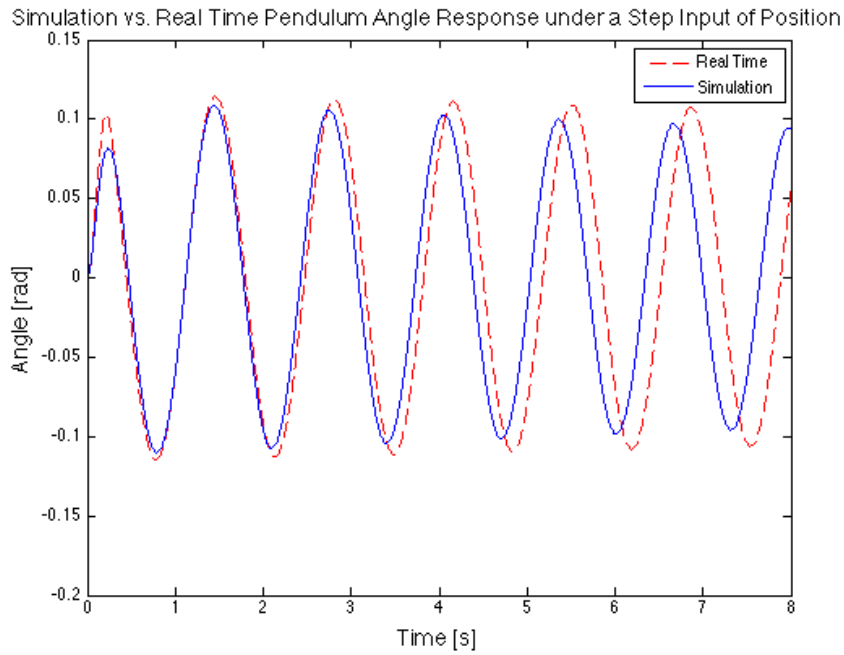

Figure 47: Inverted Pendulum Close-loop Step Response: Pendulum Angle

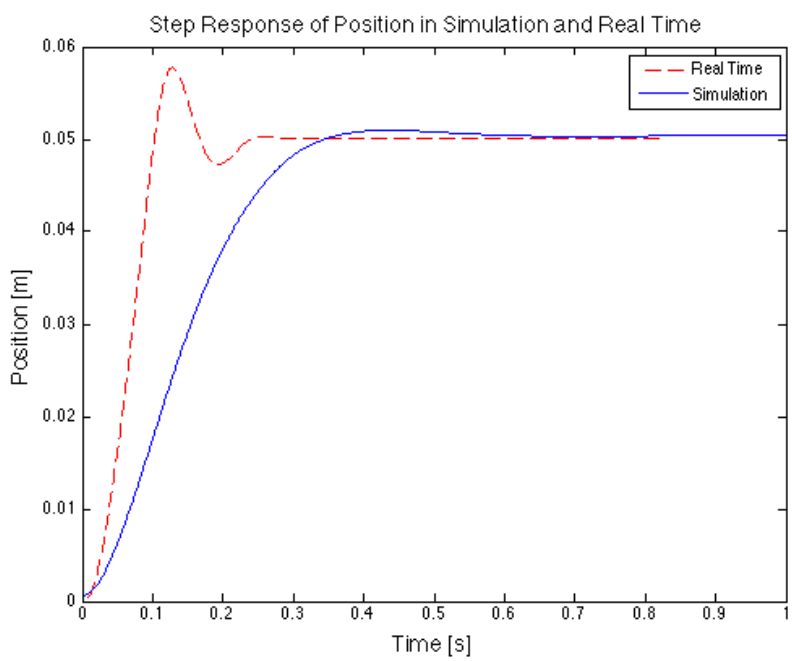

Figure 48: Inverted Pendulum Closed-loop Step Response: Stage Position

in the FMU model, it would keep in balance without the need of a controller. In this case, we set the equilibrium angle to be 0.01 radians so that the controller can function. It is determined that the choice of K1 = 4, K2 = 1, K3 = 13, K4 = 1, would make the pendulum balanced vertically upward. Then we implemented the same controller to the Real-Time system with the same feedback gains. The physical inverted pendulum is able to stabilize the pendulum using the same controller. The angle responses and position responses for both the simulation and the physical model are shown in Figure 50 and Figure 51. Although both systems are stable, the physical system oscillates at a frequency almost twice the frequency of the simulation and has larger magnitude of oscillation.

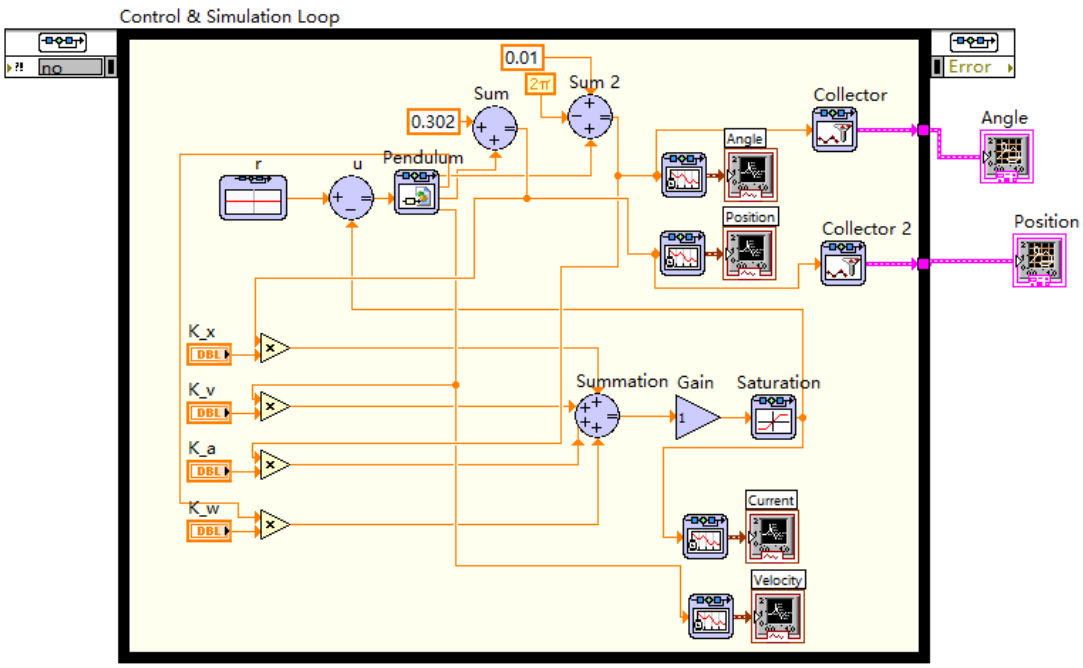

Figure 49: Block Diagram of the Inverted Pendulum Simulation

7.7 Data Visualization

7.7.1 LabVIEW Simulation to Inventor

With Inventor Dynamic Simulation, we can transfer the position data of the pendulum and track to LabVIEW and visualize the results of our simulation. This involves generating two graphs of time versus position for the slider and revolute joint.

7.7.2 Autodesk Viewer API

A useful tool to have would be an interface for users to interact with the model during simulation. We can use the Autodesk View and Data API to share our 3D model through the web. To maintain the subassemblies in our model, we converted our inverted pendulum model from Autodesk Inventor to Autodesk 360 then to WebGL. The View and Data API provides many extensions that would be useful for collaborative work. An extension similar to Inventor Dynamic Simulation would be useful to demonstrate the behavior of the system in real time and allow users to interact directly with the model during simulation. A prototype user interface is shown in Figure 52.

During the conversion procedure, constraints present in both the Autodesk Inventor and Autodesk

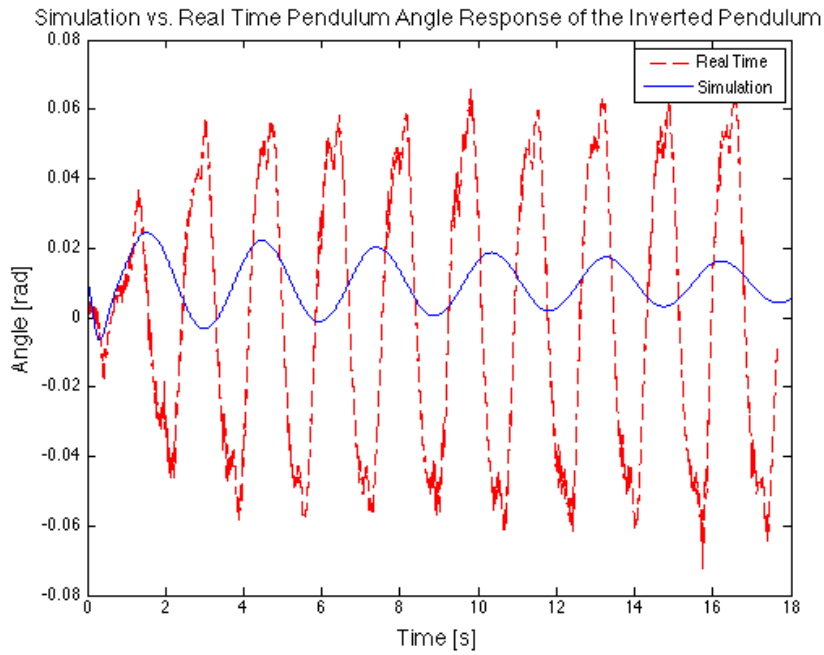

Figure 50: Simulation vs. Real Time Pendulum Angle Response of the Inverted Pendulum

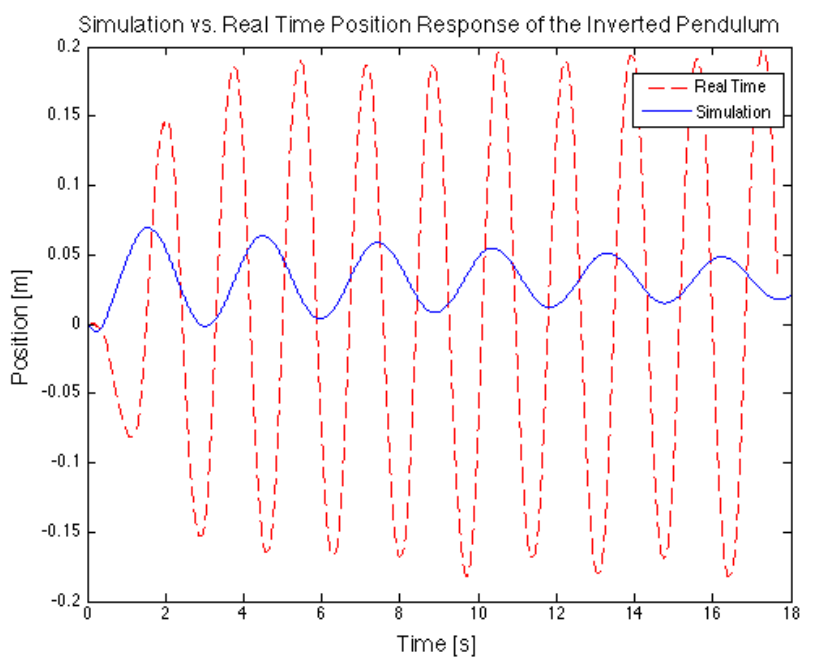

Figure 51: Simulation vs. Real Time Pendulum Position Response of the Inverted Pendulum

360 models are not retained in the WebGL model. As a result, the tool would need to enforce joints as degrees of freedom between components and convert the data from LabVIEW into movement at those joints. The inverted pendulum requires fixed joints, a slider joint, and a revolute joint.

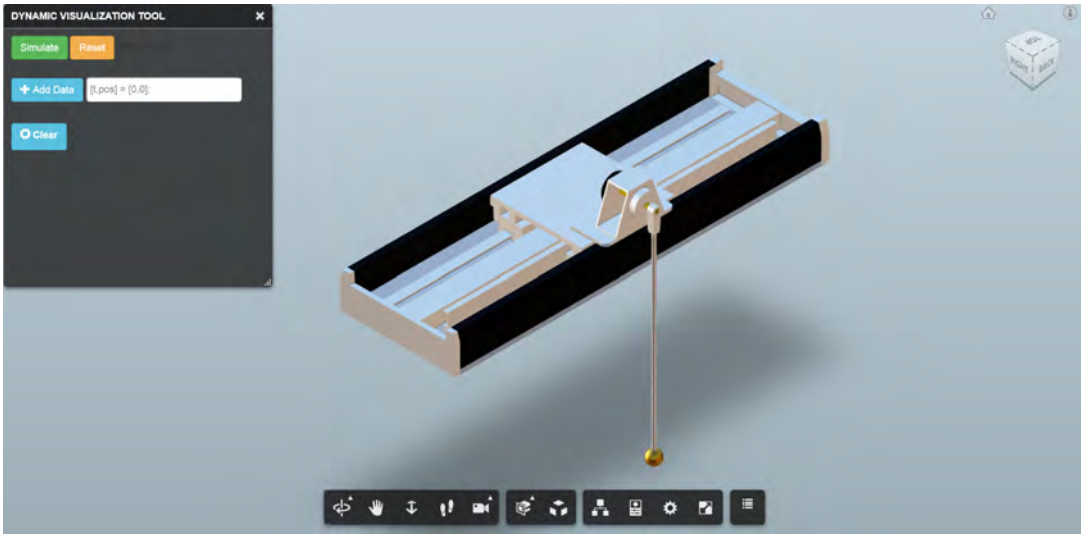

Figure 52: Conceptual User Interface for Dynamic Visualization

7.8 Conclusion

7.8.1 Workflow Verification

In conclusion, we have shown the entire end-to-end process for the design of the inverted pendulum system. We started with a physical model, reversed engineered the process of designing it in Autodesk Inventor, and finally simulated and designed a controller for the motor in Modelica and LabVIEW. The controller for the simulation was also used to control the physical system using hardware from National Instruments. Things that were particularly successful include translation from Autodesk Inventor to Modelica and adding multimedia components to the Modelica model using the Modelica Standard Library.

The main challenges for the inverted pendulum model include setting positions based on center of mass and getting support and bearing friction to work in LabVIEW through FMU. We have developed a way to use support friction in models and will describe the fix in the next section. Another challenge is the modeling of the linear motor in our system. Both limit our ability to model the system as accurately as possible.

7.8.2 Potentially Automated Features

This example highlights our need for certain Modelica blocks, friction, to get an accurate simulation model. It also shows the challenges of tuning virtual systems to match real time systems. Automating the tuning process would save on a lot of time and trial and error.

The inverted pendulum system would also benefit from being able to set initial conditions. There should be some way to have Inventor or Modelica automatically determine steady state conditions and start simulations at those locations.

Finally, a third potentially automated feature is to link the two environments - Inventor and LabVIEW - to provide visualizations during simulations. This would be an invaluable tool for users to communicate their designs to each other.

8 Modeling Friction using the Modelica Standard Library

8.1 Introduction

In this section, we will use a simple example to show how we can model friction in a system using the Modelica Standard Library. Modeling friction is very important for creating accurate models of mechanical systems. This section will provide a step-by-step guide through the process of modeling a simple mass constrained to travel along a fixed axis.

8.2 Modeling the Mass Prismatic System

8.2.1 Modelica Model - Mass Prismatic System

First in OMEdit, create a new model and name it 'MassPrismaticSystem.mo'. We will go through the following step by step instructions in detail to create the model.

1. From the Modelica Mechanics MultiBody Library, find the world frame and place it into the graphical editor. The default parameters enable gravity in the -y plane.
2. From the Modelica Mechanics MultiBody Joints Library, find the Prismatic Joint and attach it to the world frame. Set useAxisFlange = true and n = {1,0,0}. In Modelica, axis flanges allow you to control and read position of the joint. The normal vector, n, specifies the direction of travel for the prismatic joint. In this example, the mass will travel along the x-axis.
3. From the Modelica Mechanics MultiBody Parts Library, add a point mass onto the end Prismatic Joint. In the point mass's parameters, specify r_0.start = {1,0,0} . This is the position vector in relation to the world frame and places the point mass at that location. In the parameters, also change the mass to 0.5 kg.
4. From the Modelica Mechanics Translational Sensors Library, add relative position and velocity sensors onto the prismatic joint's flanges.
5. From the Modelica Mechanics Translational Sources Library, add a force input into the graphical editor and connect it to the prismatic joint's flanges. Make sure to turn on the fixed attribute if using the fixed flange on the force.
6. From the Modelica Blocks Interfaces Library, add corresponding input and output terminals. Name them before connecting them. Connect the input to the force and the outputs to the sensors.
7. In the icon view of OMEdit, move the input and output terminals outside the white box. This is so that we can connect signals to the block diagram if we reference this model in another model.

A graph of the completed model is shown in Figure 53.

8.2.2 Modelica Model - Mass Prismatic Control

Once we have our Modelica model completed, we can add the corresponding inputs/outputs in a separate Modelica model. That way, we can use the mass prismatic system as a block in a block diagram.

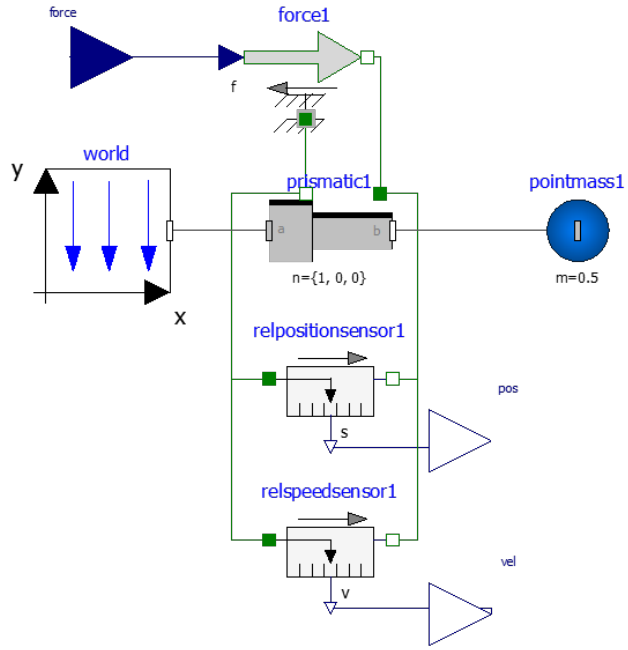

Figure 53: Frictionless Mass Prismatic System Modelica Model

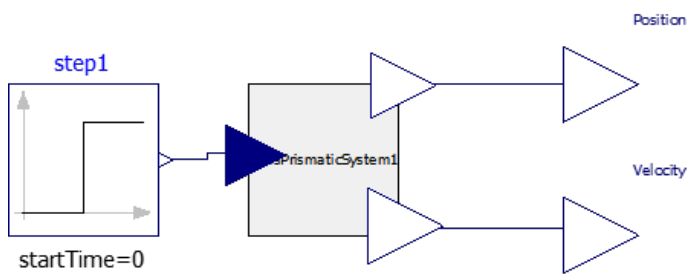

Figure 54: Frictionless Mass Prismatic System Modelica Model

1. From the Modelica Blocks Sources Library, place a pulse block into the graphical editor.
2. Drag and drop the mass prismatic system, 'MassPrismaticSystem', into the graphical editor.
3. Connect the source to the input of the mass prismatic system using the graphical editor
4. From the Modelica Blocks Interfaces Library, add another two output terminals. Again, name the outputs variables before connecting them.
5. Connect the output terminals to the outputs of the mass prismatic system.

A graph of this model is shown in Figure 54.

8.3 Modelica to FMU

Using the Modelica model of the system, we can generate an FMU to import into LabVIEW.

1. Start a Python 2.7 session.

```
from pymodelica import compile_fmu
model_name = 'MassPrismaticSystem'
mo_file = 'MassPrismaticSystem.mo'
my_fmu = compile_fmu(model_name, mo_file, version=2.0)
```

Figure 55: FMU Export Procedure

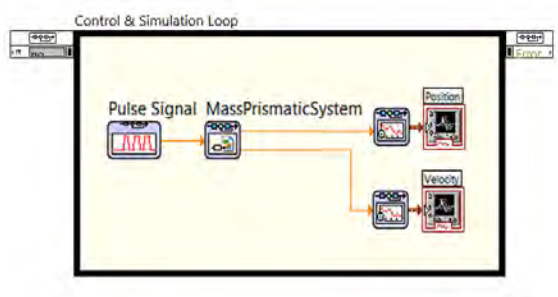

Figure 56: Frictionless Mass Prismatic System LabVIEW Model

2. Import the JModelica Python interface, PyModelica, to compile Modelica code into FMU.
3. Describe the model_name and location of the mo_files with Python strings.
4. The function, compile_fmu, will generate FMU for the model in the JModelica directory.

The code for this procedure is shown in Figure 55

8.4 FMU to LabVIEW

With the FMU completed, we can move the mass prismatic system into LabVIEW using the following steps.

1. Open LabVIEW 2014 and create a new Blank VI. Save the VI as MassPrismatic.vi.
2. In the Block Diagram, put a Control and Simulation Loop from the Control and Simulation Palette.
3. Drag an FMI for Model Exchange block into the Control and Simulation Loop. Select the FMU MassPrismaticSystem.fmu. All input/output terminals will be generated automatically with the FMU produced using JModelica.
4. Select the Pulse Signal block from Signal Generation Palette and connect it to the force input terminal.
5. Drag the SimTime Waveform block from Simulation - Graph Utility Palette and connect it to output terminal y on the Plant block.
6. Double-click the terminals on the left side of the Simulation Loop and set the initial time to 0 s and the final time to 5 s in the Configuration Dialog.
7. Run the VI and observe the position and velocity graphs in the front panel.

A graph of this model is shown in Figure 56.

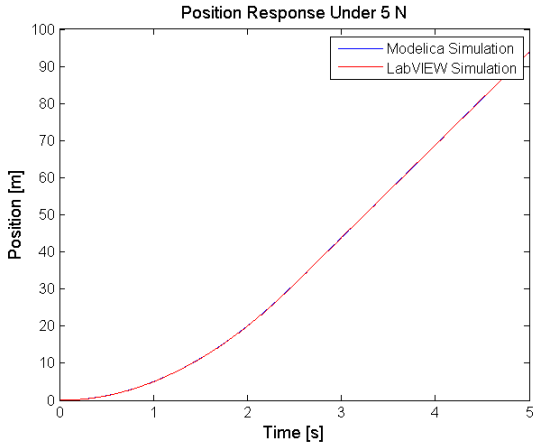

Figure 57: Frictionless Mass Prismatic System Position Simulation

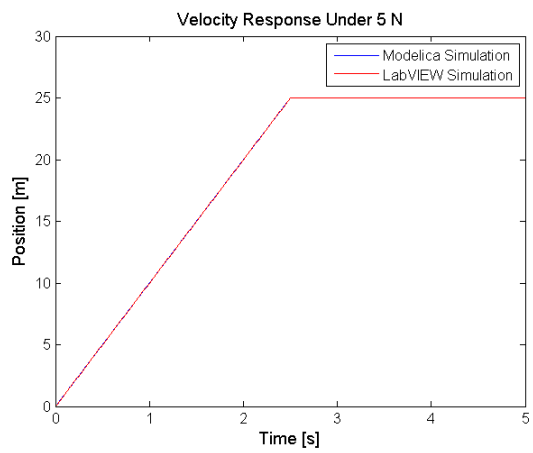

Figure 58: Frictionless Mass Prismatic System Velocity Simulation

8.5 Simulating the Frictionless Mass Prismatic System

8.5.1 Frictionless System - Transient Response

First, let us verify the zero input response. Set the amplitude of the pulse function to 0 and perform a simulation for 5 seconds, observing the position and velocity of the point mass. In the plot tab, we can select our relative speed and position sensors and plot the data versus time. Although there is gravity along the y axis, the prismatic joint constraints and restricts movement along the y axis. The responses from the LabVIEW simulation and the Modelica simulation are identical - there is no movement of the point mass.

8.5.2 Frictionless System - Pulse Response

Add a force of 5 N from 0 to 2.5 seconds. In LabVIEW, this can be done using a pulse signal with the following settings: Start time: 0, Amplitude: 5, Offset: 0, Duty Cycle: 50, Period: 5. In Modelica, the pulse source block should be configured with the following parameters: Amplitude: 5, Width: 50, Period: 5, Nperiod: 1, Offset: 0, StartTime: 0. We see the behavior of both systems is identical. The results are shown in Figure 57 and 58.

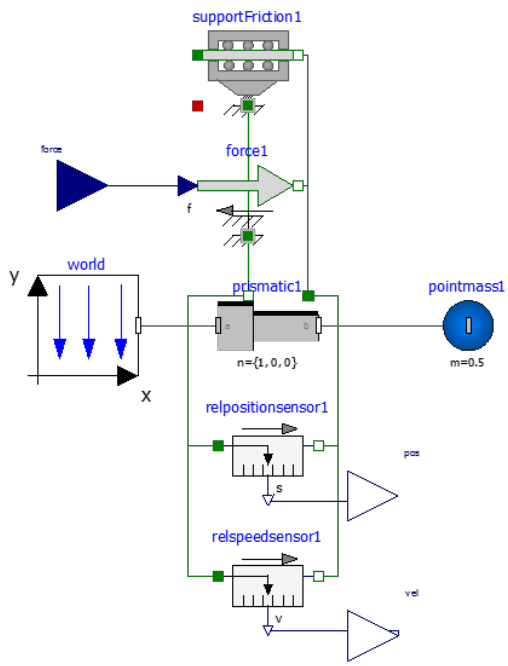

Figure 59: Frictionless Mass Prismatic System Modelica Model

8.6 Introducing Friction into the Model

8.6.1 Support Friction

Now that we have verified the frictionless model of the Mass Prismatic System, we can add support friction from Modelica Mechanics Translational Library. The Modelica Support Friction element describes Coulomb friction, a frictional force acting between a flange and the housing. The input to the friction block is a table of velocity and force. For example, a table could be f_pos = [0, 0; 1, 2; 2, 5; 3, 8]. The support friction block will linearly interpolate values within the table. Outside the table, the forces is assumed constant and takes the last value in the table. In this example, the block would have a static friction force of 1 N, linearly increase depending on the velocity to 8 N, then provide a constant 8 N of friction force after 3 meters per second. Now connect the support friction block to the flanges of the prismatic joint. For this example, let us set the friction to be a constant 1 N, f_pos = [0,1]. The Modelica model is shown in Figure 59. Repeat the process of exporting the system to FMU and importing the system to LabVIEW.

8.6.2 Support Friction with Stribeck Friction Table

Instead of guessing multiple forces and velocities for the support friction table, the GenerateStribeck-FrictionTable function from The Modelica Examples Utilities Library is able to automatically generate a table of forces and velocities using 6 inputs. In the Support Friction Block's parameters, we set include f_pos = Examples.Utilities.GenerateStribeckFrictionTable(F_prop = 1, F_Coulomb = 5, F_Stribeck = 10, fexp = 2, v_max = 12, nTable = 50). A graph of force versus velocity generated by the function is shown in Figure 60.

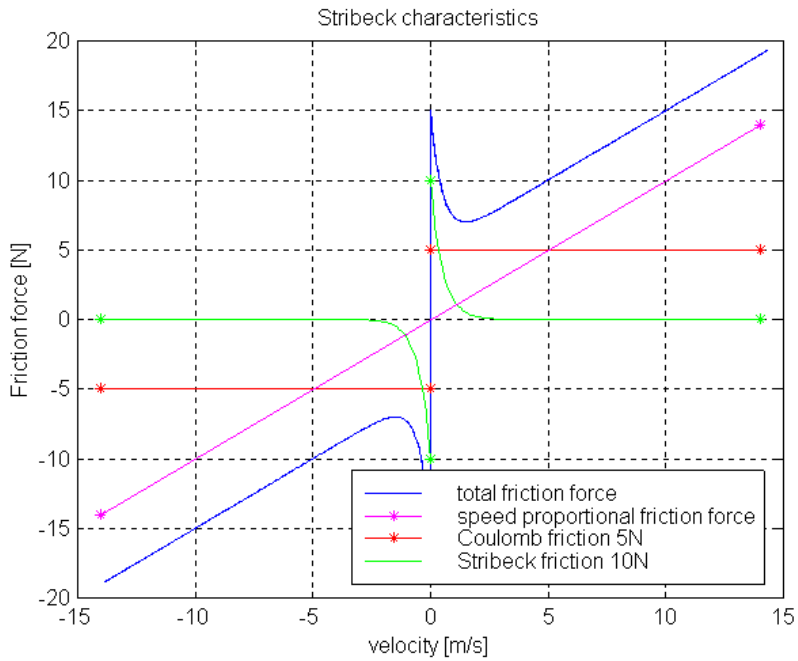

Figure 60: Example of Stribeck Friction Table

8.6.3 System with Support Friction - Transient Response

Now let us verify the zero input response again. We will set the support friction back to 1 N. Change the amplitude of the pulse to 0. Perform a simulation for 5 seconds and plot the results. Again, the behavior should be identical.

8.6.4 System with Support Friction - Pulse Response

In both Modelica simulation and LabVIEW simulation, let us repeat the same test and add of force of 1 N from 0 to 2.5 seconds. We expect that neither simulation moves due to the static friction force of 1 N. This holds true for both models, but once we change the force to 5 N, we note some discrepancies between the models. We see the expected response from the Modelica simulation but not from the LabVIEW Simulation. These graphs are shown in Figure 61 and Figure 62. The LabVIEW Simulation seems to have difficulties recognizing that the input force exceeded the static friction threshold.

This problem is a result of problems handling the initialization of the LabVIEW simulation. Given a non-zero input at time zero, the states get trapped at zero. To fix this problem in the LabVIEW simulation, we add a delay into the input and activate the step response at $t = 1e^{-6}$ seconds. This small fix solves the problem with friction in LabVIEW and the response in LabVIEW now matches the response in Modelica. The LabVIEW simulation with this fix is shown in Figure 63.

8.6.5 System with Damping

Because the solution to issues involving implementing friction in the LabVIEW simulation was not resolved until recently, we modeled friction using the Modelica Standard Library Damping block instead. This block is a linear velocity dependent damper element. Remove the support friction

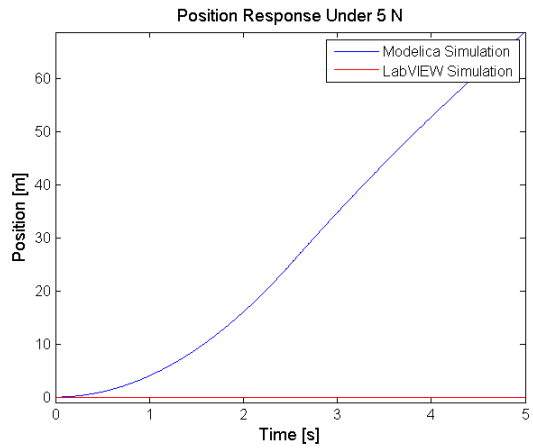

Figure 61: Friction Mass Prismatic System Position Simulation

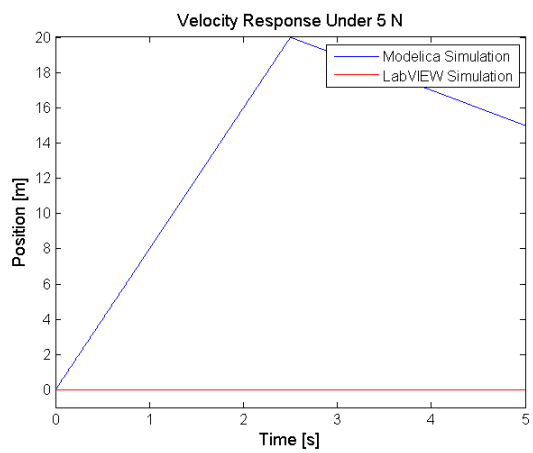

Figure 62: Mass Prismatic System with Friction Velocity Simulation

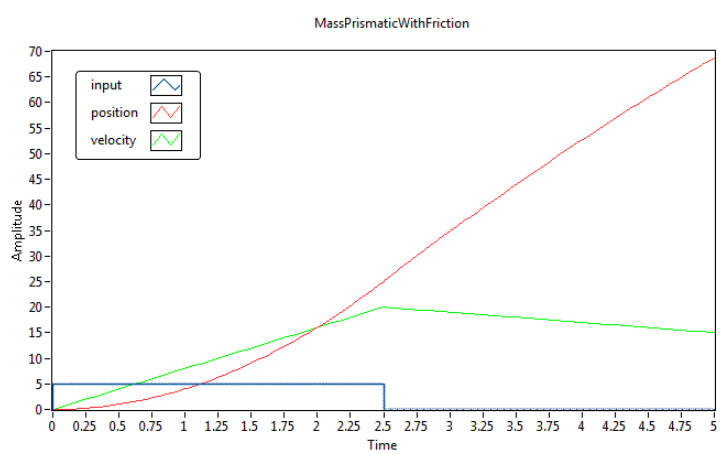

Figure 63: LabVIEW Support Friction With Delay

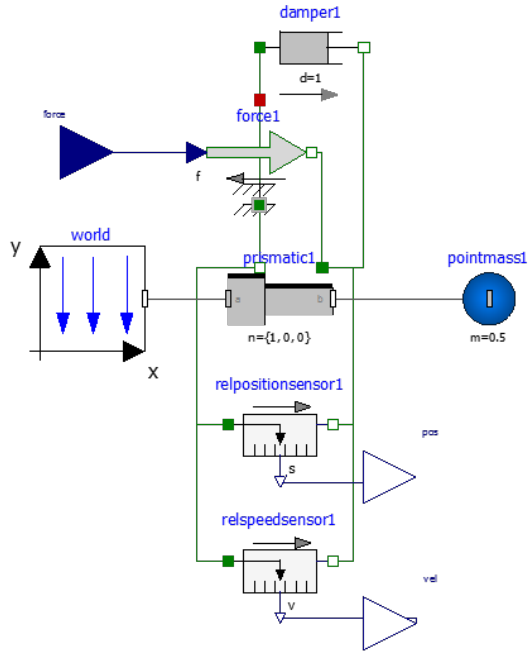

Figure 64: Mass Prismatic System with Damping

block from the Modelica model and replace it with the damping block. Set the parameters of the damper to 1 N.s/m and repeat the process of exporting the system to FMU and importing the system to LabVIEW.

8.6.6 System with Damping - Transient Response

Again, we can verify the zero input response again to ensure that the point mass doesn't move. Change the amplitude of the input to 0 and perform the simulation for 5 seconds. In both the LabVIEW simulation and the Modelica simulation, the position and velocity remain constant as expected.

8.6.7 System with Damping - Pulse Response

Let us repeat the experiments we did with the support friction block. After running the simulation, we can see that the simulations here match up. One problem with using damping is that the mechanical behavior modeled may be completely different. For example, systems that have high amounts of dry friction will be modeled poorly using the damping block. The damping block is a good alternative for well lubricated systems with less dry friction because it only requires a single parameter to tune.

8.7 Conclusion

In summary, we have shown two approaches to modeling friction. The first approach is to use the Modelica Support Friction block for translational joints. The support friction block provides a way to model static friction, but models exported with this component require a small change in the input signal in the LabVIEW simulation environment. The second approach is to use the Modelica Damping block. This approach doesn't account for static friction but is still able provide a velocity

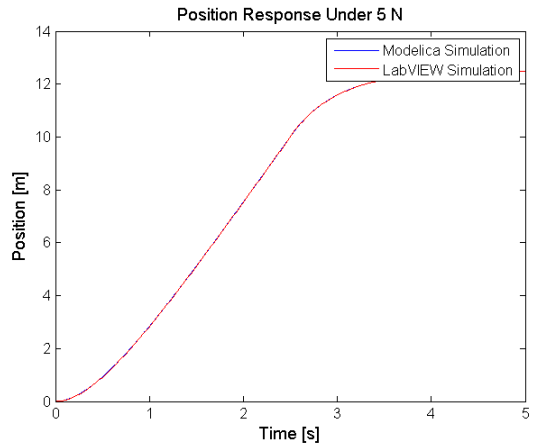

Figure 65: Mass Prismatic System with Damping Position Simulation

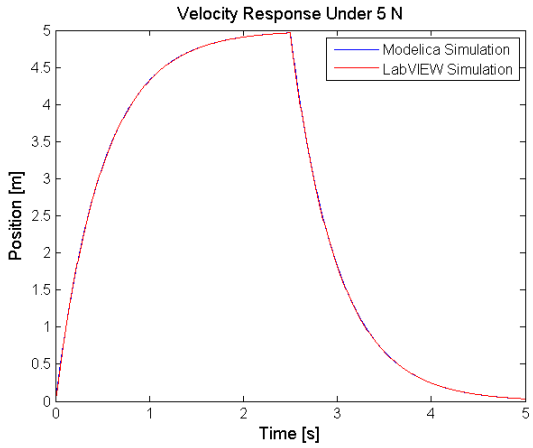

Figure 66: Mass Prismatic System with Damping Velocity Simulation

Figure 67: Five Axis Robot

dependent damping force. For the models creating using this workflow to improve, the process from Modelica to LabVIEW through FMU must retain all properties of the original Modelica model. We expect both simulation environments to perform identically.

9 Five Axis Robotic Arm

9.1 Introduction

9.1.1 Testbed Motivation

In this section, we will explore the end-to-end process of our prospective automated workflow for the design and control of a five-axis (plus gripper) robotic arm (Figure 67). High accuracy multi-axis robotic arm are commonly designed and controlled in manufacturing environments.

9.1.2 General Procedures

Like the DC motor test stand and inverted pendulum, we will start with a physical model of the five axis robotic arm, reverse engineer the process of designing it in Autodesk Inventor, and export details of the system to LabVIEW. Finally, using the exact same control schema, we will compare results of simulations of the system in Modelica and LabVIEW with the results of connecting the control schema to the physical system. We can see a diagram of our prospective workflow in Figure 68.

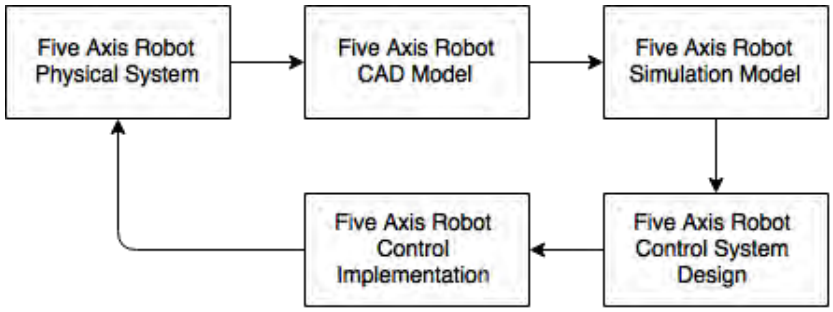

Figure 68: Five Axis Robot Workflow Diagram

9.2 Physical Model

9.2.1 Device Characterization

The robot is a SCORBOT-ER 4u, a versatile training robot designed by Intelitek for labs and classrooms. It has five rotational axes and supports a wide range of motion. In total, we have six motors that control five rotational axes and one gripper. The fourth and fifth axes are arranged in a unique differential drive configuration. When the motors are turned in the same direction at the same speed, the drive rolls. When the motors are turned in opposite directions at the same speed, the drive pitches. Overall, each motor has an optical encoder that reads the motor's position and velocity. The motors are connected to each axis via gears and belts. We control each motor using CompactRIOs and NI9505 motor modules rather than using SCORBASE robotics programming and control software.

9.2.2 Component Specifications

Basic information about the robot can be found on Intelitek's website. The data sheet includes a description of each component, each axes' range of movement, and each axes' effective speed. The robot can be homed using fixed micro switches at each axis. The system is designed to have repeatability tolerance of 0.18 mm at the tip of the gripper.

9.3 Inventor Model

9.3.1 Detailed Inventor Model

A model of the robot is created in Inventor using dimensions from the physical system. Mass properties and degrees of freedom are included in the model but belts and gears were not included as we knew the Inventor-to-Modelica translator would not support these features. Weights are taken from each component to assure that the real weight corresponds to the weight of the model. An image of the model is shown below in Figure 69.

9.3.2 Inventor Model Properties

The Inventor model takes advantage of Inventor motion constraints to simulation the robot's range of motion. For example, these constraints connect each motor to its corresponding axes. One thing to note is that the belts and wires are not included in the Inventor model. In the real system, the differential drive utilizes two bevel gears control pitch and roll of the end effector. This is

Figure 69: Five Axis Robot CAD

challenging to model in Inventor due to the number of redundant constraints. The Inventor-to-Modelica Translator also does not currently recognize these types of assemblies and constraints; therefore, we need to limit the Inventor model to having only three axes.

9.4 Modelica Model

9.4.1 Inventor-to-Modelica Translator

A Modelica model of the robot is created using the Inventor-to-Modelica Translation Software. The translated model is shown in Figure 70. The translator is able to convert all the constraints from the CAD model correctly but converted a few more components than needed. We do not need the motor shafts that are translated since we can include their properties in the motor when attached directly to the revolute joints. Similar to the inverted pendulum model, to set each axis of the robot in various positions, we need to modify the position vectors for the components in Modelica or adjust the Inventor model and translate the robot assembly again.

9.4.2 Modelica Model Verification

We can use a Modelica simulation to verify the state of our system before adding multimedia components. We will add position and velocity sensors to the exported Modelica model at each of the revolute joints to measure the response. The Modelica Standard Library provides both relative and absolute sensors for translational and rotational movement. Typically, the initial conditions of the system are referenced as 0, but sometimes the sensors start reading at 2π.

In Figure 71, we observe that the robot does fall over despite constraining the robot to a fixed position. Axis 1 doesn't move much because it rotates along the plane orthogonal to gravity. Axis 2 and 3 rotate a considerable amount due to gravity. This is expected as the multiple axes act as a stacked inverted pendulum with small amounts of friction from the motors. There should have a way in Inventor or Modelica to place the model into a stable equilibrium to prevent unexpected transient responses. Also, each axis of the robot can rotate 360 degrees and is not limited by the

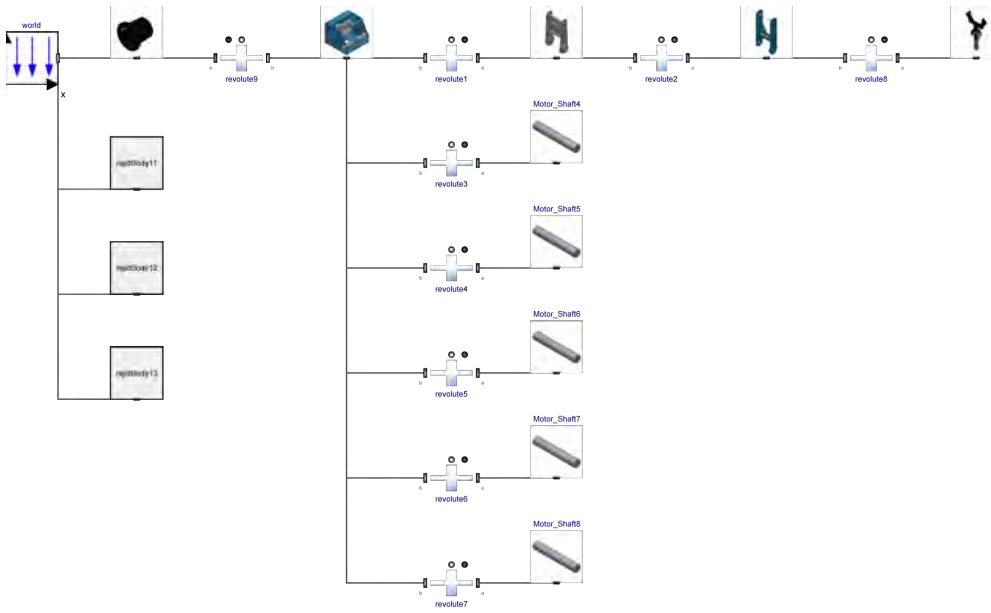

Figure 70: Three DOF Robot Modelica Translation

physical components clashing into each other. A good feature to have in Modelica would be the ability to place limits on the positions of each joint.

9.4.3 Adding Multimedia Elements

Now that we have the core elements from our Inventor model translated into a Modelica model, we add the multimedia elements and relationships not captured by the translator: electronics, gearing ratios, backlash, friction forces, and input and output terminals. The inputs to our system will be a voltage for each motor and the outputs will be rotational position and velocity of each axis or revolute joint.

For each of the motors, we use the Modelica Standard Library DC permanent magnet motor and tune the parameters based on specifications from the manufacturer, which include the back EMF constant and the rotational inertia of the rotor. We perform experiments on the motor to find the nominal voltage, current, and angular velocity. An interesting note is that the motor does include viscous friction and damps the movement of the revolute joints.

Next, we add the gear ratios for each axis using the Modelica Standard Library ideal gear. This component has one driving shaft and one driven shaft, converting rotations from the motor to the revolute joint.

Although we are not able to use the Modelica Standard Library bearing friction block, we can use the Modelica Standard Library spring damper block to model the elasticity and damping in the system. We also attach this component at each revolute joint.

Finally, we add the fourth and fifth axes of the robot in Modelica by using an equation based model of the interface. We are unable to use the existing Modelica Standard Library Mechanics blocks to model the differential drive mechanism. As a result, the equation based model does not account for inertial change in the system so the spring damper block does not affect the angle of the fourth and fifth axes. The complete model is shown in Figure 72.

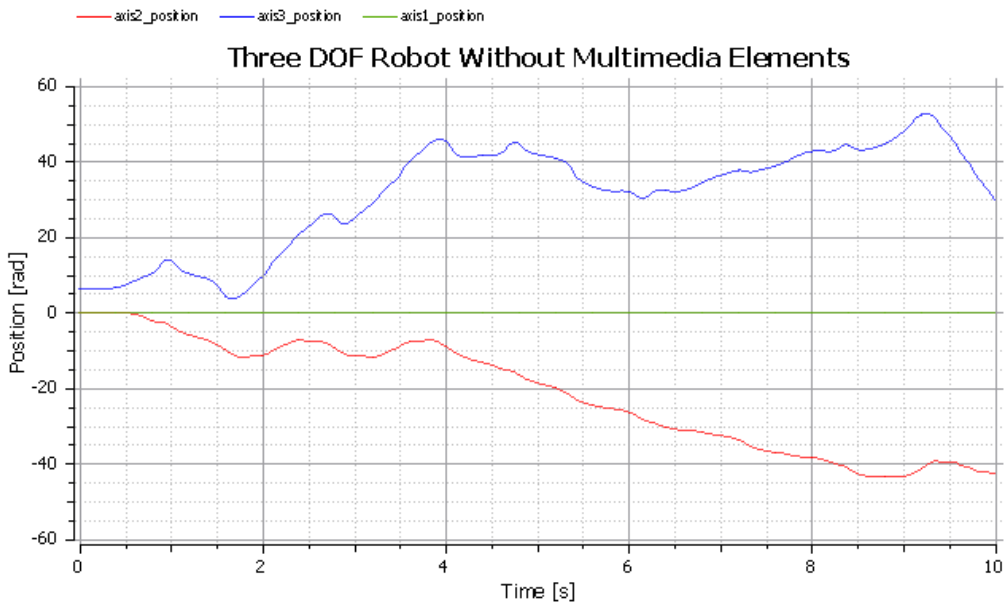

Figure 71: Three DOF Robot Modelica Simulation Without Multimedia Elements

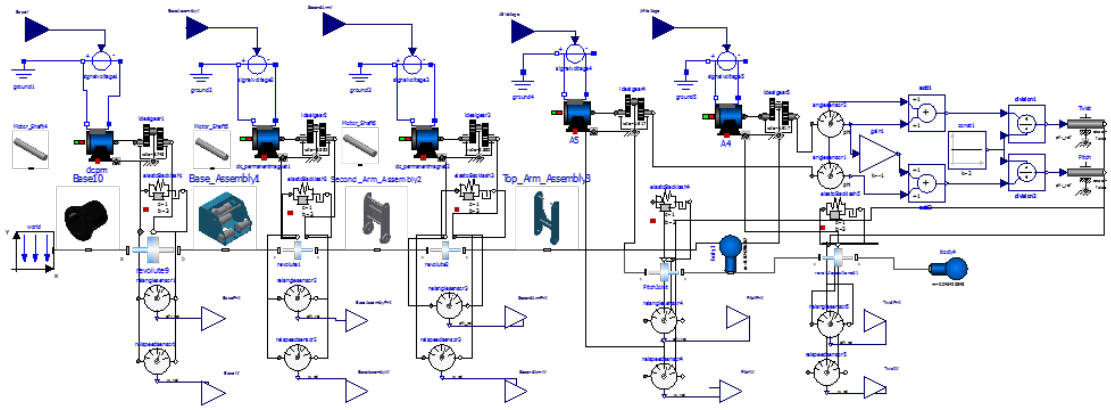

Figure 72: Five Axis Robot Complete Modelica Model

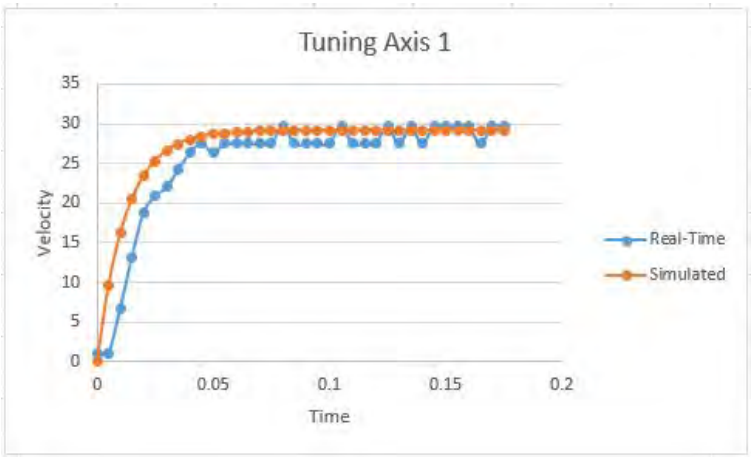

Figure 73: Comparing Angular Velocities to Tune Axis 1

9.4.4 Modelica Modeling Assumptions

We assume that each component can be replaced with a body mass. Also, we are unable to convert the support and bearing friction or backlash block from Modelica to LabVIEW so we assume that the combination of the friction from the DC permanent magnet motor and the spring damper block are sufficient. There are also challenges in modeling a physical gearbox so an ideal gearbox with no losses or holding torque was used instead. Finally, no limits to the rotation of the robot were applied.

9.4.5 Modelica to FMU

Even with our equation based model of the fourth and fifth axes, there are no problems exporting the Modelica model as an FMU using JModelica.

9.5 LabVIEW

9.5.1 FMU to LabVIEW

We import the FMU into our LabVIEW VI, and select the system parameters we would like to tune. One issue we found is that there is a limit to how many variables that can be chosen to tune, 15 variables. This number also includes the number of input terminals. With 5 input variables for the robot, only 10 tunable variables remain. A simple workaround is to set a few variables in Modelica before exporting as an FMU. Otherwise, the system may need to be separated into multiple FMUs using several input and output terminals to connect them.

9.5.2 Calibrating Model Specifications

To calibrate the model's frictional properties, we tune the spring and damping coefficients in the FMU by comparing the real-time and simulation. The easiest way to do this is to compare the settling times of the angular velocity for each axis as a result of constant voltage inputs. Figure 73 shows an example of the process involved in tuning an axis.

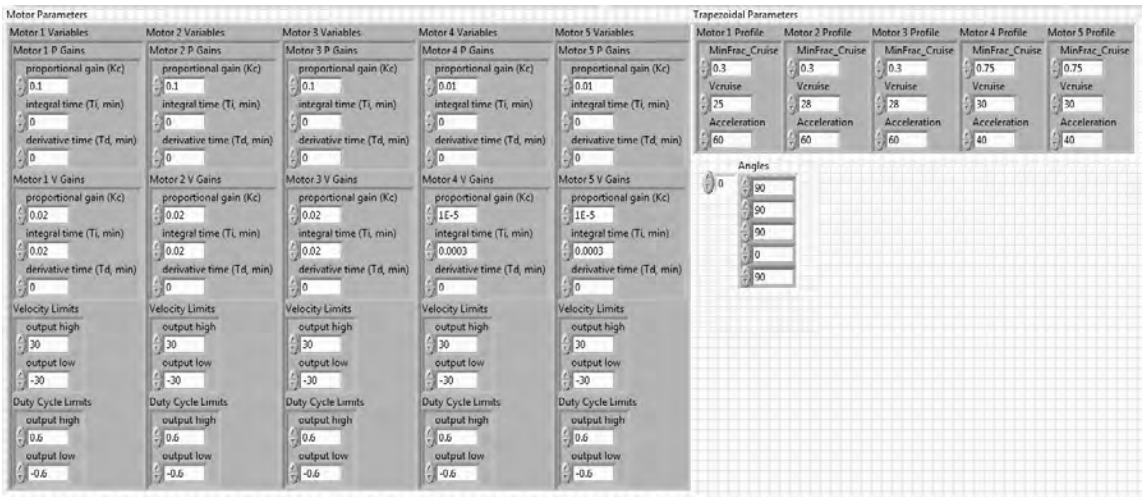

Figure 74: Parameters Tuned Using Simulation

9.5.3 Trapezoidal Profile Control System Design

To control the robot, we implement a trapezoidal motion controller. The trapezoidal profile includes 3 phases. In the first phase, the system undergoes constant acceleration. In the second phase, the system undergoes constant velocity. In the third and final phase, the system undergoes constant deceleration to reach its final position. An S-curve motion profile can also be implemented for smoother motion.

9.5.4 Tuning the Control System Using Simulation

After tuning the FMU, we tune the first and second layers of the control system. These layers include the cascade loop and the trapezoidal profile generator. The SubVI control scheme is placed into a SubVI, and the inputs to the SubVI are desired positions, measured positions, and measured velocities. Tuning the control system involves trial and error due to inconsistencies in the outputs from the FMU, which caused large spikes in the velocity profiles. Next, we implement the same control system for both the simulation and real-time system and compare the results.

9.5.5 Comparing Simulation and Real Time Results

Once we obtain satisfactory results from the simulation, we port the control system to the real-time system and compare the responses. The real-time system is generally more robust compared to the simulation and has a wider range of possible gains. Figure 74 shows the simulation parameters, and Figure 75 shows the result of moving each axis 90 degrees using a trapezoidal profile.

The first three axes reach the correct steady state position in both the simulation and real-time system; however, the real-time system responds slower than the simulation. We also see a lot of noise in the velocity profile of the real time system. In the real-time system, axes 4 and 5 do not reach their target steady state conditions and produce inaccurate velocity profiles. This may be due to excessive static friction that is unaccounted for in the model. A more robust simulation model with more accurate models of each component is required to improve these results.

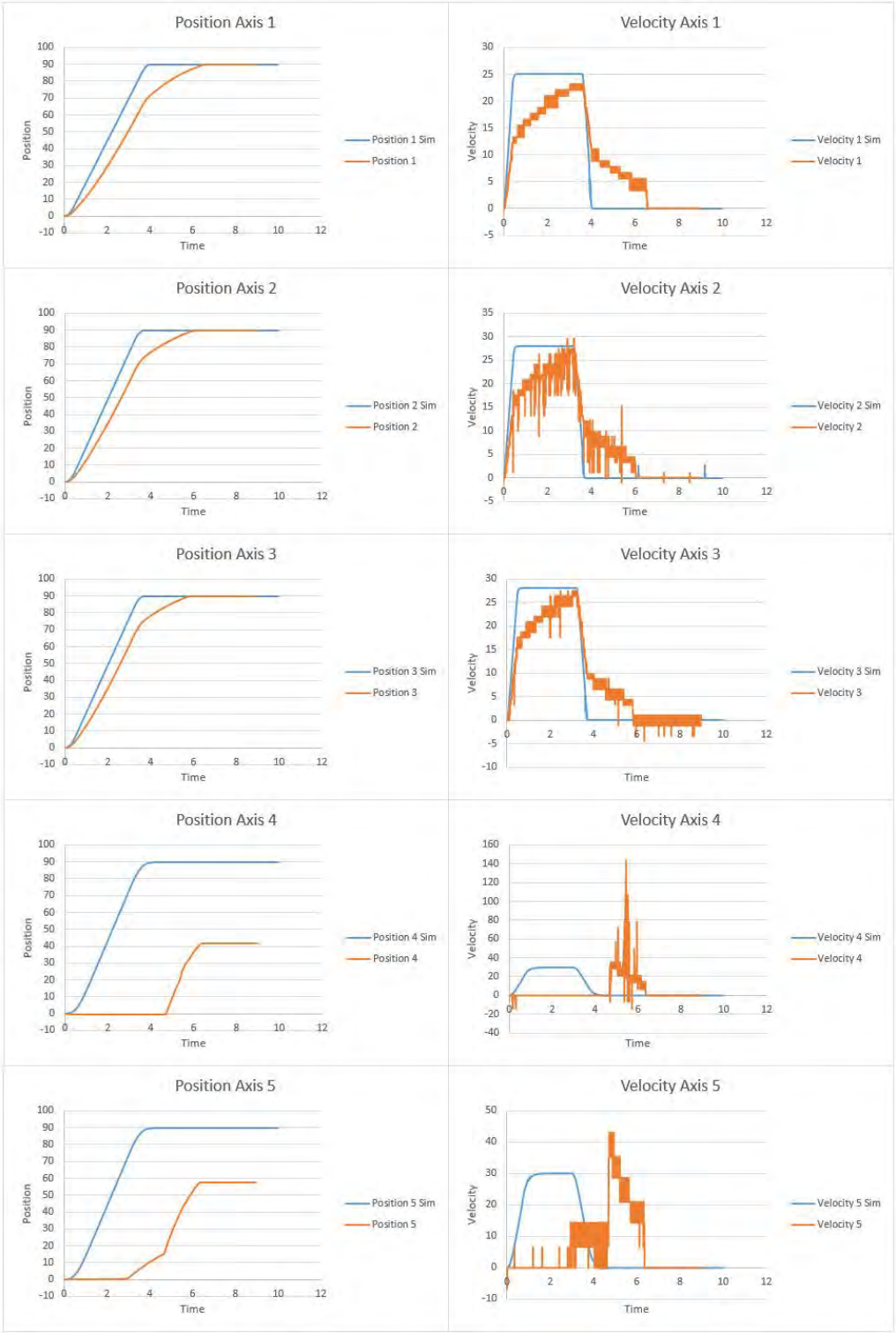

Figure 75: Comparing the Simulation and Real-Time System Using the Same Control System

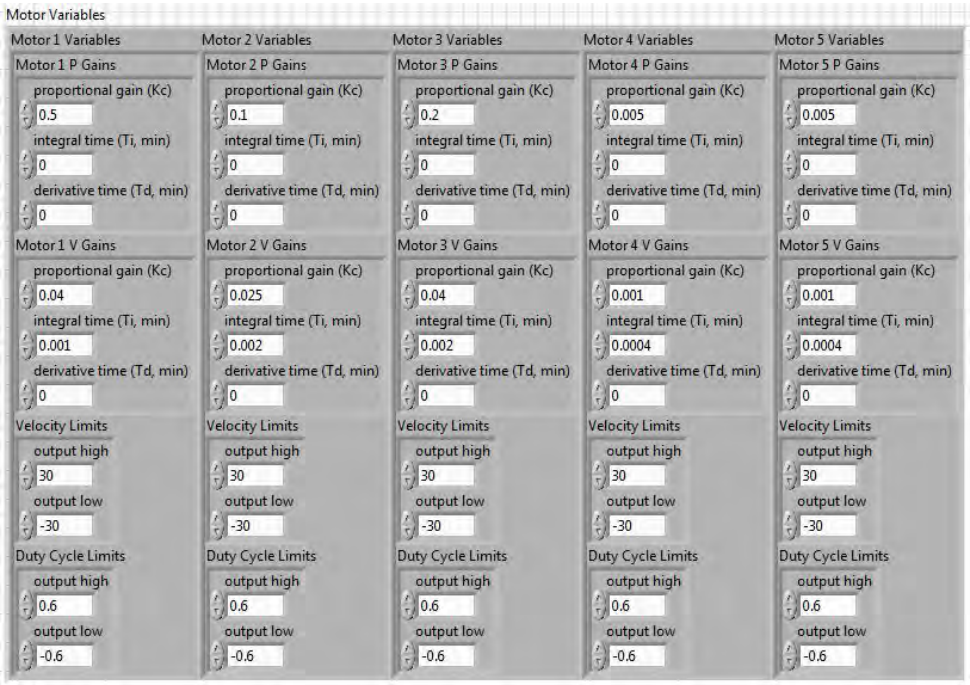

Figure 76: Optimized Real-Time Parameters

9.5.6 Optimizing the Real Time System

Next, the real-time system is tuned separately from the simulation to optimize the system's response. Although the cascade control gains are different, the system uses the same trapezoidal profile gains. To stabilize the response of the real-time system, we increase the proportional gains and lower the integral gains. Figure 76 shows the updated gains, and Figure 77 shows the behavior of the real-time system, which appear to follow the simulation results more closely. The noise in the simulation may be a result of errors in calculating the velocity and resolutions of the encoders.

9.6 Conclusion

9.6.1 Workflow Verification

The main challenges for the five-axis robotic arm include describing complex mechanical mechanisms, tuning multiple components of a system independently, and ensuring that optimizing the behavior of the simulation also optimizes the behavior of the real-time system.

9.6.2 Potentially Automated Features

Certain mechanical components are not described by the Modelica MultiBody Library. The differential drive described by the 4th and 5th axes would have benefited from a model based on existing MultiBody components. Non-ideal gears and pulleys also may improve the accuracy of the model.

The testbed exemplifies our need for an automated process to simplify the rather repetitive process of tuning multiple components. For example, we needed to tune motor and friction for each axis. Ideally, components described in Modelica accurately reflect the physical components to skip this process.

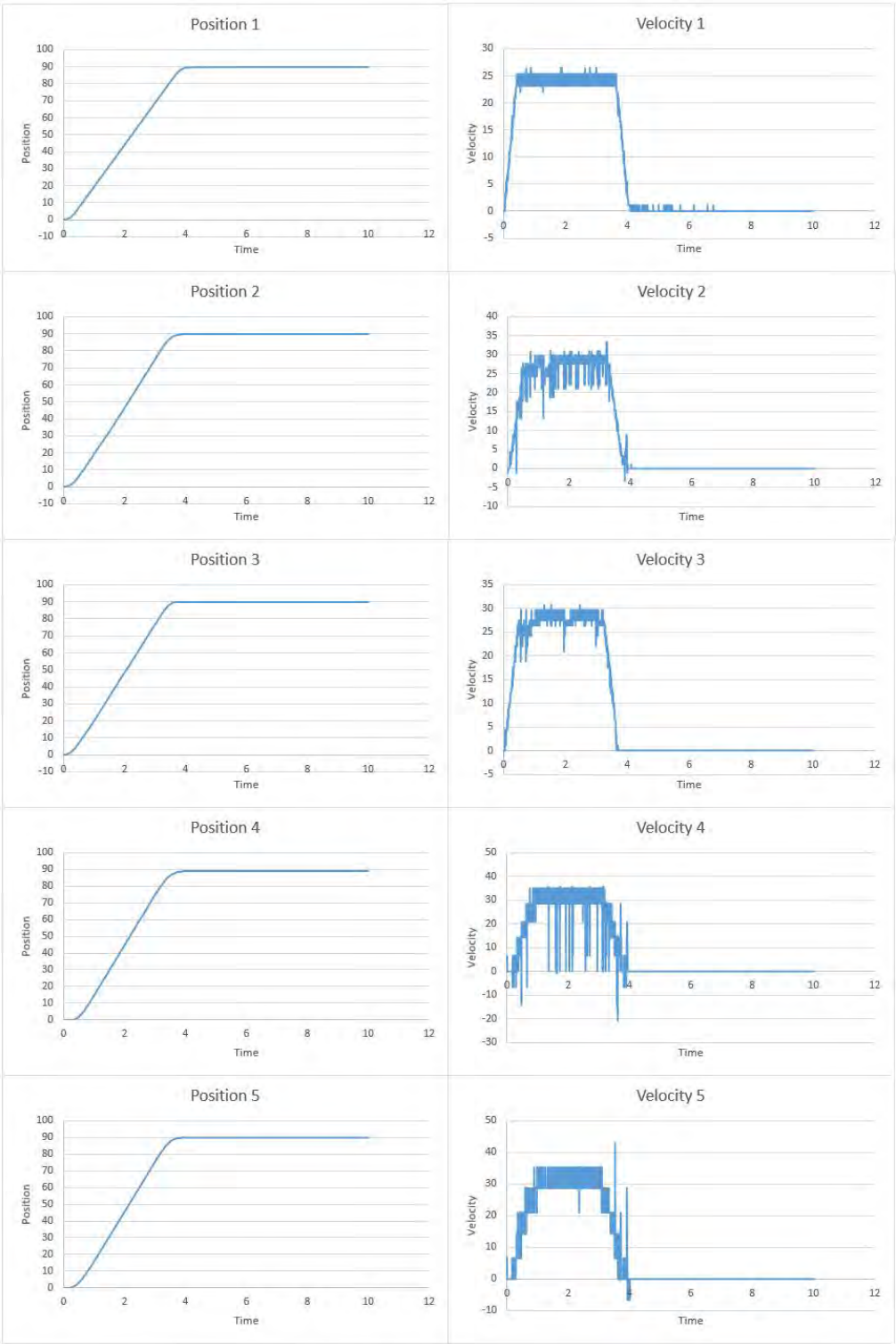

Figure 77: Optimized Real-Time System Response

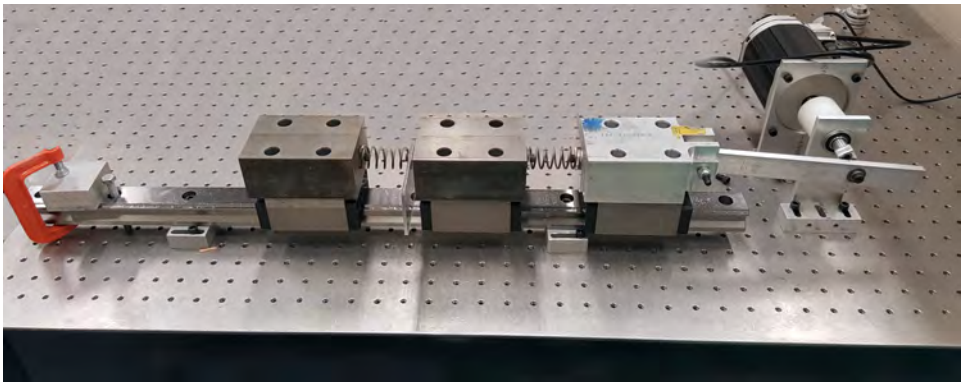

Figure 78: Slider Crank Workflow Diagram

Finally, we can imagine a system that is also able to automatically tune the control system for the simulation and present gains for stable responses. The auto-tune software could iteratively improve the existing control system by using search algorithms. This would be extremely helpful since the simulation for the robot is not as robust as the real-time system.

10 Slider Crank

10.1 Introduction

10.1.1 Testbed Motivation

The slider crank system described in this section is a four-bar mechanism that contains a series of three masses connected by two springs (Figure 78). The key system elements we will explore in this section are resonance frequencies, spring dynamics, and kinematic loops.

10.1.2 General Procedures

We do not start with the physical model in this section. Instead, we will use this testbed as a design challenge and start from a CAD model in Inventor. From Inventor, we will convert the CAD model into a Modelica model and add multimedia components. After adding these components, we will export the system to LabVIEW and develop control for the simulation. Once we have taken the model through the workflow, we will manufacture the physical system and compare the results of the simulation versus the real-time system. The complete workflow is visualized in Figure 79.

10.2 Physical Model

10.2.1 Device Characterization

The slider crank system consists of a motor, a four-bar mechanism, and three masses. The three masses are placed along the track, with a spring in between each mass. The slider crank converts the rotational movement from the motor into translational movement.

copyright ©2015 University of California, College of Engineering

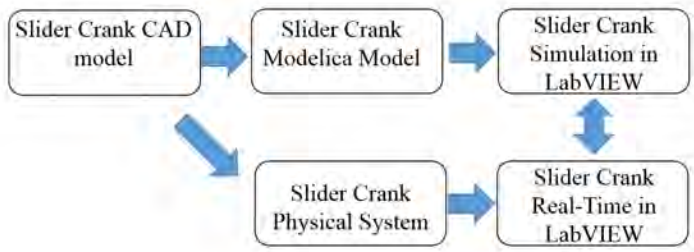

Figure 79: Slider Crank Workflow Diagram

10.2.2 Component Specifications

The main electrical components in the system include ultrasonic sensors, a CPM-MCPV-2321P-RQN Motor from Teknic, couplings from Teknic, and bearings from McMaster-Carr.

10.3 Inventor Model

10.3.1 Detailed Inventor Model

The Inventor model of the slider crank system contains seven subassemblies:

1 The fixed motor

2 The fixed track for the masses to slide along

3 The rotating short arm

4 The long arm

5-7 Three individual mass/carriage assemblies

The mass/carriage assemblies are constrained to move linearly on the track. Revolute joints are added between the motor shaft and short arm, between the short arm and long arm, and between the long arm and the first mass. The completed model can be seen in Figure 80.

10.3.2 Inventor Model Preparation

Although spring elements can be added in Inventor for visualization purposes, we do not model the springs prior to converting our model because the Inventor-to-Modelica translator does not support these elements. With the slider crank, it is important to be able to specify initial conditions. For example, we would like to set the model to start in equilibrium and set the locations of each of the three masses on the track. We can do this by applying distance constraints in Inventor and deleting them prior to exporting.

10.4 Modelica Model

10.4.1 Inventor-to-Modelica Translator

Figure 81 shows the raw Modelica model exported using the Inventor-to-Modelica translator. Since the slider crank is a four-bar mechanism, we have a kinematic loop in the Modelica model, which currently isn't supported by the Inventor-to-Modelica Translator. Upon closer inspection, the "Point-

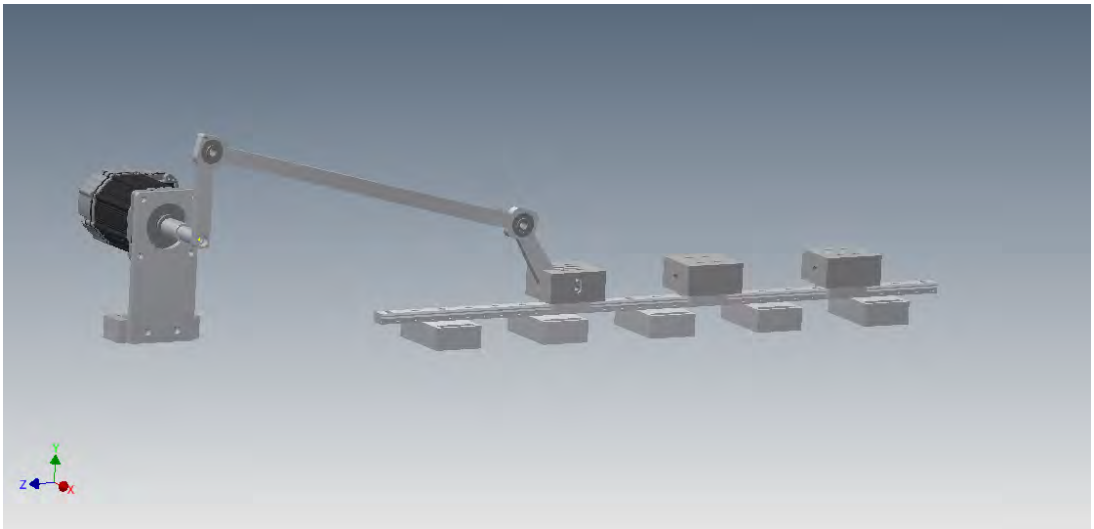

Figure 80: Slider Crank System

Line" joint seemed to prevent the model from translating. Once we removed this joint (keeping the model in the exact same position), we are able to transfer the model of the slider crank from Inventor to Modelica.

10.4.2 Modelica Model Initialization

Setting the initial conditions (position, velocity, acceleration) for joints in the slider crank in Modelica is challenging. For example, all joints of the same type (i.e. revolute, slider) inherit from the same Modelica class. As a result, initialization parameters apply to all joints that inherit from the same class.

It is also difficult to change the position of the Modelica MultiBody Bodies. In Modelica, joints are located in space using frames referenced from the origin. The position data is contained within the model of the joint and is accessible outside of the model through the parameters. After experimenting with different initialization settings, we end up with a working model.

10.4.3 Adding Multimedia Elements

Now that we have the core elements from our Inventor model translated into a Modelica model, we add and modify the multimedia elements and relationships not captured by the translator: revolute joint with cut forces, a DC permanent magnet motor, and input/output blocks.

Since the slider crank is a four-bar mechanism, we add a special revolute joint from the Modelica MultiBody Library to deal with the planar loop. This revolute joint solves the problems related to cut forces and cut torques.

We set the motor properties that were given by the manufacturer and approximate the properties that we were unable to find (e.g. nominal voltage, nominal current).

We also create a Modelica model from scratch using as much information as we can from the Inventor model. We match general shapes and masses, as well as include the special revolute joint for looping in Modelica. Rather than specify each joint as having frames at the origin, we build the

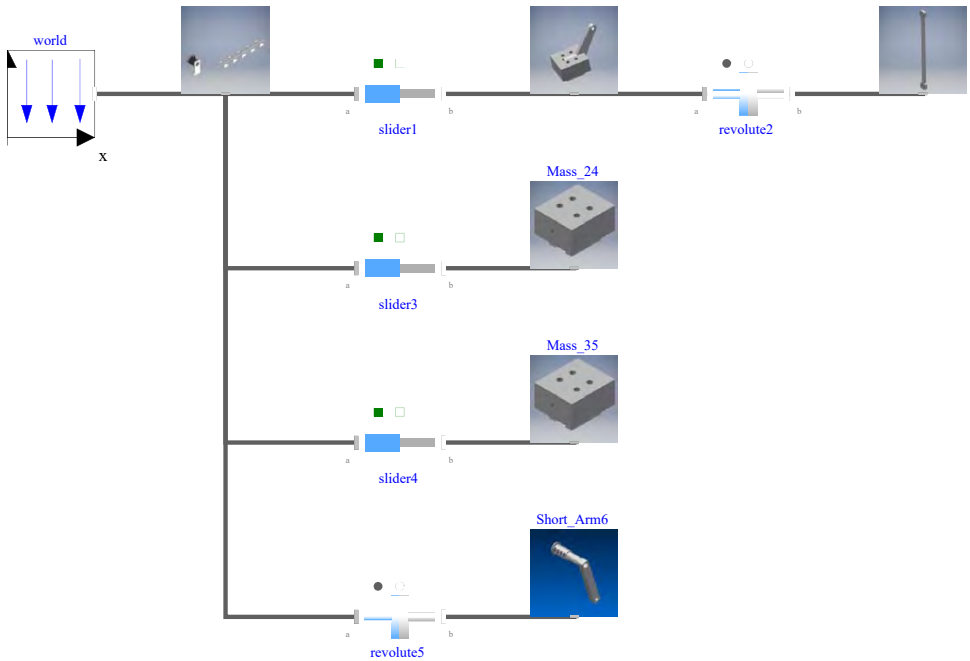

Figure 81: Exported Modelica Model

model so that each piece was given a position relative to the component it was connected to. Figure 82 shows the augmented Modelica model and Figure 83 shows a Modelica model made from scratch. We create two models to verify multiple methods of modeling the same system.

10.4.4 Modelica Modeling Assumptions

In this model, we avoid modeling and tuning friction. The main focus of this testbed is to explore resonance frequencies, springs, and how to model four-bar mechanisms.

10.4.5 Modelica Simulations

Both the augmented Modelica model and the Modelica model created from scratch simulate as expected. We can see plots for the position of each mass in the simulation in Figures 84, 85, and 86. In both cases, the motor is driven with a constant input of 3 volts. The slight differences in position is a result of specifying relative positions but we see that the shape and frequency of the response is almost identical. One thing to note is that the models break down after inputting between three to five volts into the motor. At this point, the masses begin to cross each other because springs from the Modelica MultiBody Library do not contain hard stops and are force based.

10.5 Functional Mock-Up Interface (FMI)

10.5.1 FMI 2.0 Model Exchange

As of now, we have not been able to produce a working FMU model in LabVIEW, which prevents us from completing the workflow for the slider crank.

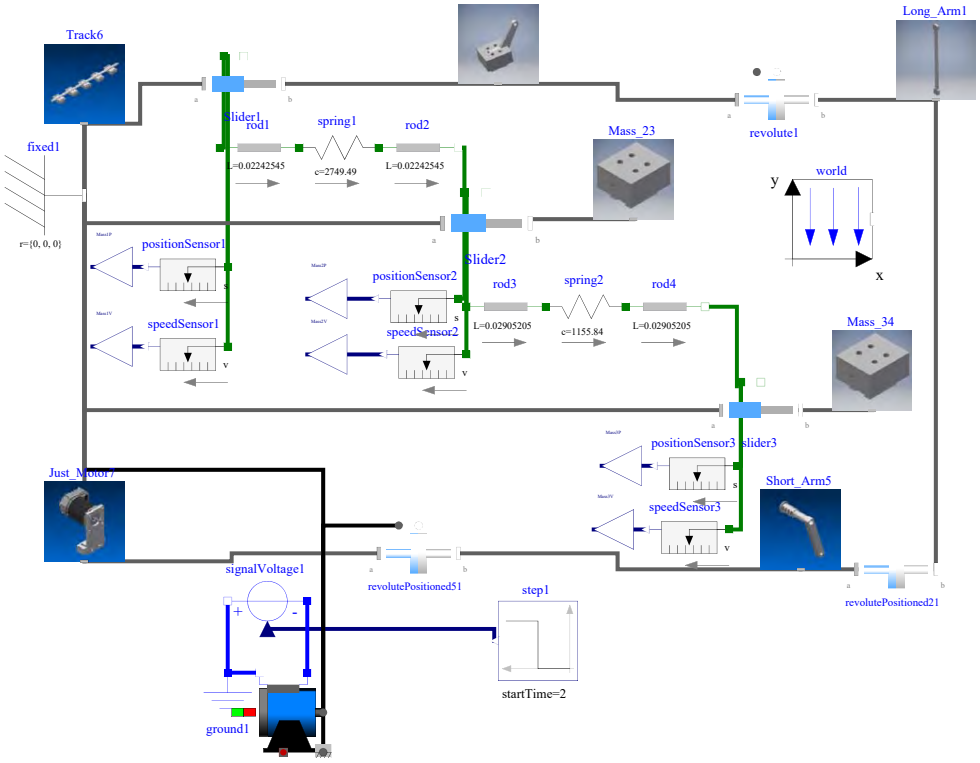

Figure 82: Augmented Modelica Model

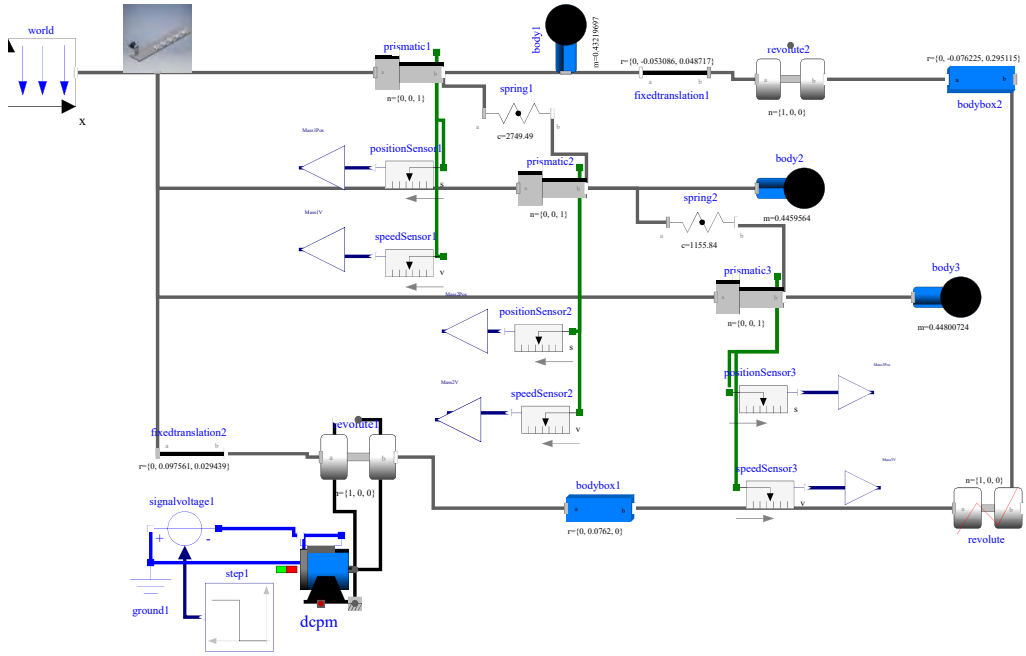

Figure 83: Modelica Model Created from Scratch

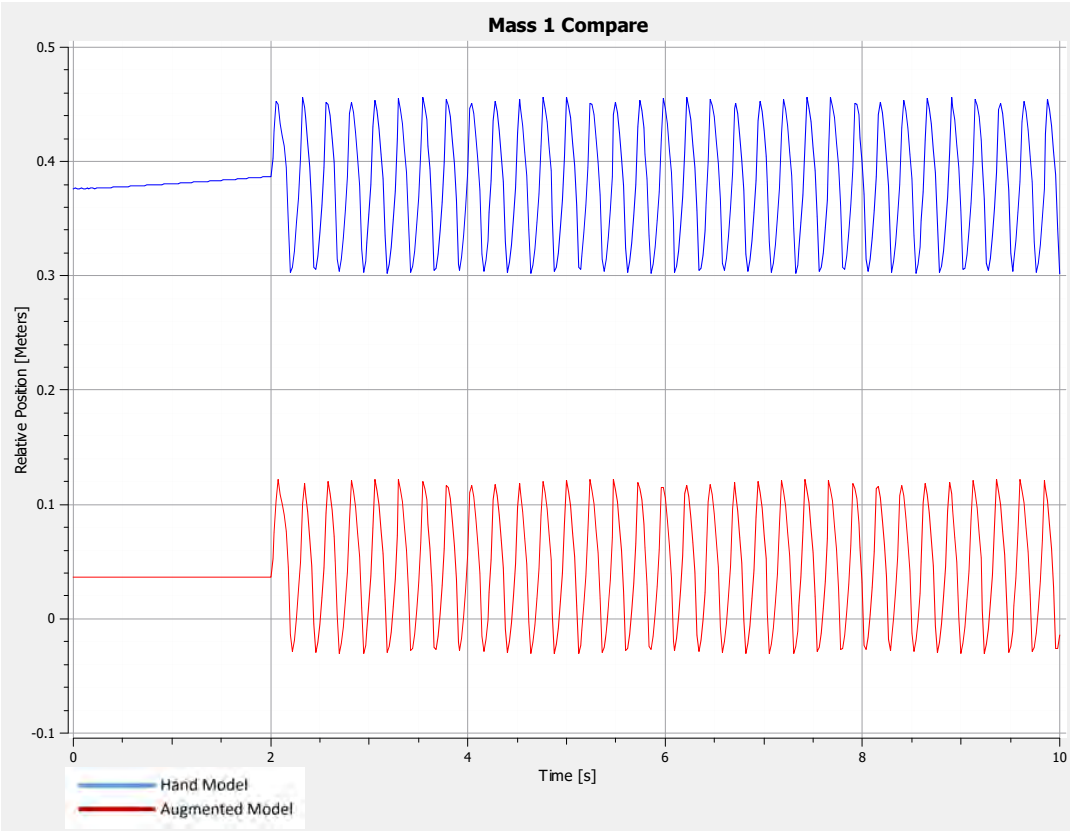

Figure 84: Position Data for Mass 1

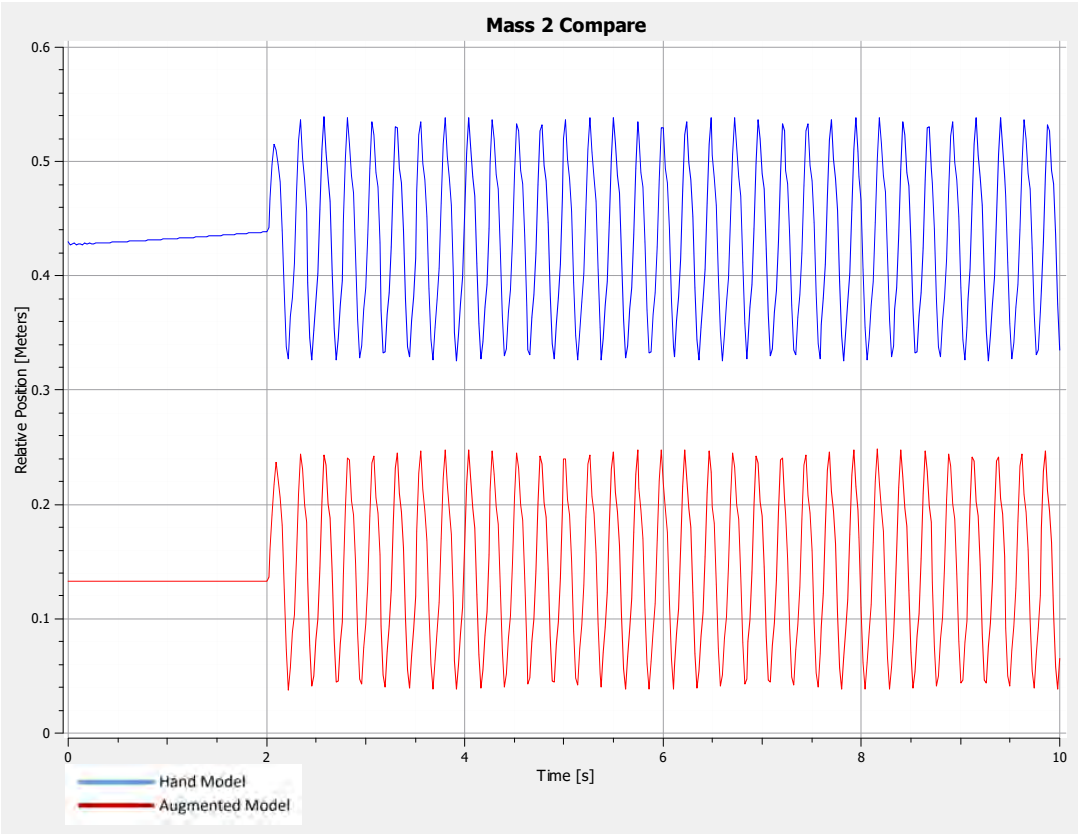

Figure 85: Position Data for Mass 2

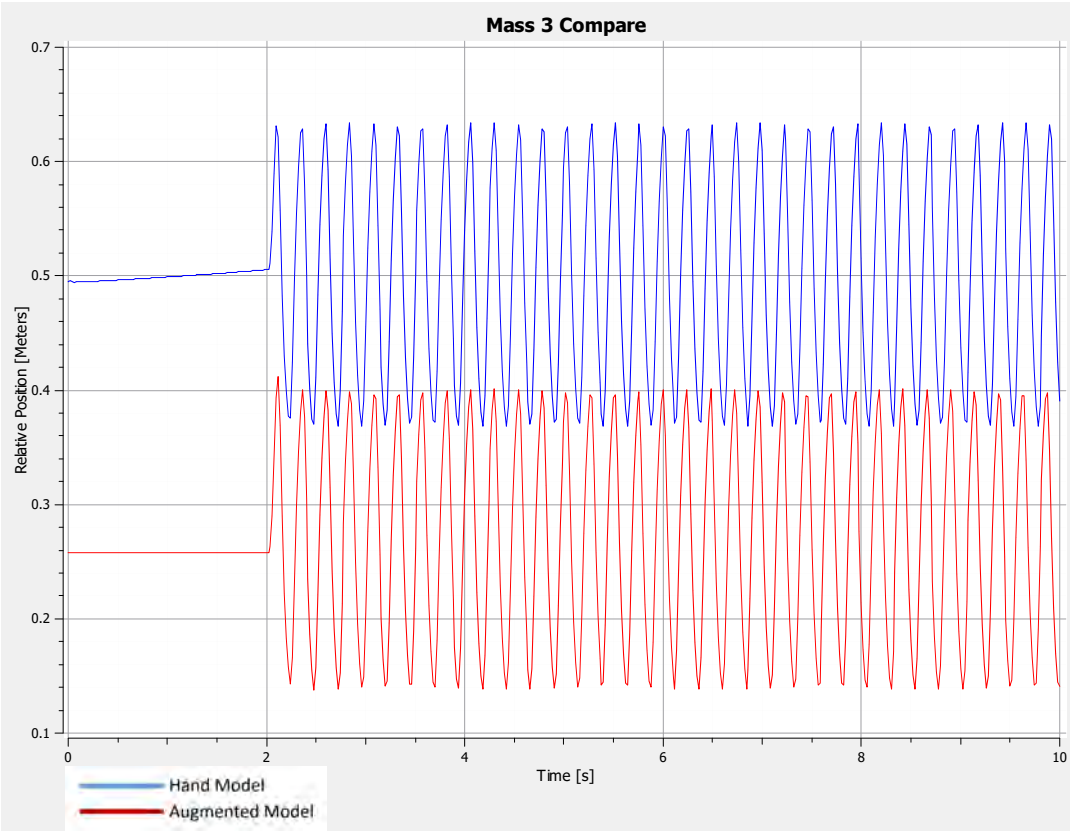

Figure 86: Position Data for Mass 3

10.6 Going Forward

The next step for the slider crank testbed is to produce a working model in LabVIEW and to compare our simulation with the response from the physical system. We hope to be able to effectively model the resonance frequencies of the system, include accurate models for friction at each joint, and include hard-stops for the springs. At the time, we have demonstrated that Modelica can model the slider crank but many challenges relating to initialization will need to be addressed to help automate this process.

11 Go-Cart

11.1 Introduction

11.1.1 Testbed Motivation

The goal of this testbed is to model the mechanics of a go-cart. With this system, we are concerned with modeling the slippage and rolling of the tires, the drivetrain, and the steering system.

11.1.2 General Procedures

We start with the physical go-cart and model the drivetrain and steering system in Inventor separately. Then we convert the Inventor model to a Modelica model using the Inventor-to-Modelica translator. Finally, we export an FMU from Modelica to develop control using the LabVIEW simulation environment.

11.1.3 Workflow Diagram

Although it is difficult to evaluate the accuracy of the model, we will be following the standard workflow seen in the previous testbeds. The workflow diagram can be seen below in Figure 87.

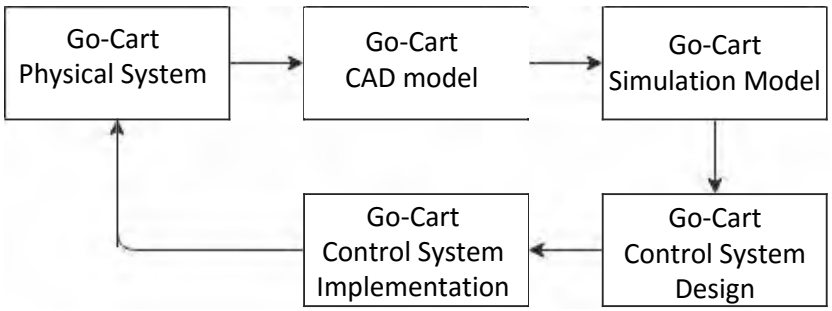

Figure 87: Go-Cart Workflow

11.2 Physical Model

11.2.1 Device Characterization

The go-cart is a very complex mechanical system built by the INSTAR lab at UC Berkeley. In this section, we will explore the drivetrain system and the steering mechanism. The drivetrain consists

of namely the chassis, the motor, the two axles (the back 'driven' axle as well as the front axle), and the two sets of wheels. The steering mechanism is made up of two simple four-bar loops and shares a single revolute joint that attaches to a steering column and steering wheel. The go-cart is shown in Figure 88.

Figure 88: Physical Go-Cart

11.2.2 Component Specifications

Important components to note include the motor and the tires. The drivetrain motor is a Motenergy ME0201014201 PMAC Motor with a voltage range of 24-72 volts. Rapter Slick tires have an 11 inch outer diameter, 5 inch inner diameter, and a 6 inch width.

11.3 Inventor

11.3.1 Detailed Inventor Model

A model of the go-cart drivetrain is created in Inventor using a preexisting model of the go-cart. All masses will be approximated because the physical system is difficult to disassemble and measure. The entire assembly consists of the rear and front axles, wheels, driving sprocket, driven sprocket, chain, and chassis. The sprockets included in the Inventor model are off-the-shelf components that included 3D models created by the manufacturer.

We initially place the wheels and axle together in an Inventor subassembly so the model treats them as one solid part. Then we include the driven sprocket and constrain it with the axles. The driving sprocket is rotationally mated to the driven sprocket using a 2:1 gear ratio. A chain is included using the roller chains component in Inventor. The front axle and all of the rear axle assembly are each positioned to match the architecture of the go-cart.

11.3.2 Inventor Model Preparation

To prepare the drivetrain model for Inventor-to-Modelica translator, we decompose the model into four separate assemblies.

1 The front axle and wheels combination

2 The back wheels, axle and driven gear combination

3 The driving gear

4 The frame

We then approximate the mass of the go-cart and create a small and very dense ring to attach to the frame at the center to model the weight of the entire go-cart. Finally, we remove the belt drive and constraints connecting the driving and driven gears since the Inventor-to-Modelica translator is unable to recognize these constraints. The completed drivetrain model can be seen in Figure 89.

The steering mechanism is unable to export because of the kinematic loops created by the constraints in Inventor. To fix this, we remove two revolute joints (one from each loop) while being careful to maintain the positions of each bar. We can add the revolute joints back into the model once the model is converted into a Modelica model. The completed steering model can be seen in Figure 90.

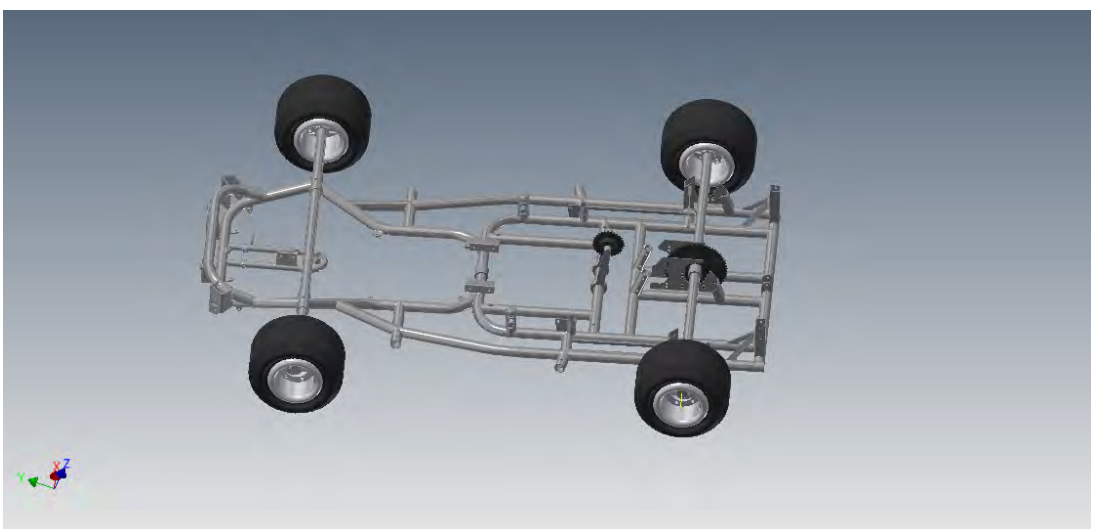

Figure 89: Drivetrain CAD Model

11.4 Modelica

11.4.1 Inventor-to-Modelica Translator

After modifying the Inventor models, the translator successfully generates Modelica models for both systems with a few exceptions. The first problem is related to kinematic looping. Once wheel elements are attached to the models, the closed kinematic loop is unsolvable. We solve this by substituting the rigid connection between wheel components by attaching a very stiff spring. The second problem is a challenge with initialization. Since the Inventor-to-Modelica translator creates joints inherits from the same model, the model cannot run without initialization errors. We solve this problem by duplicating the model of the revolute joint and replacing one of the existing revolute

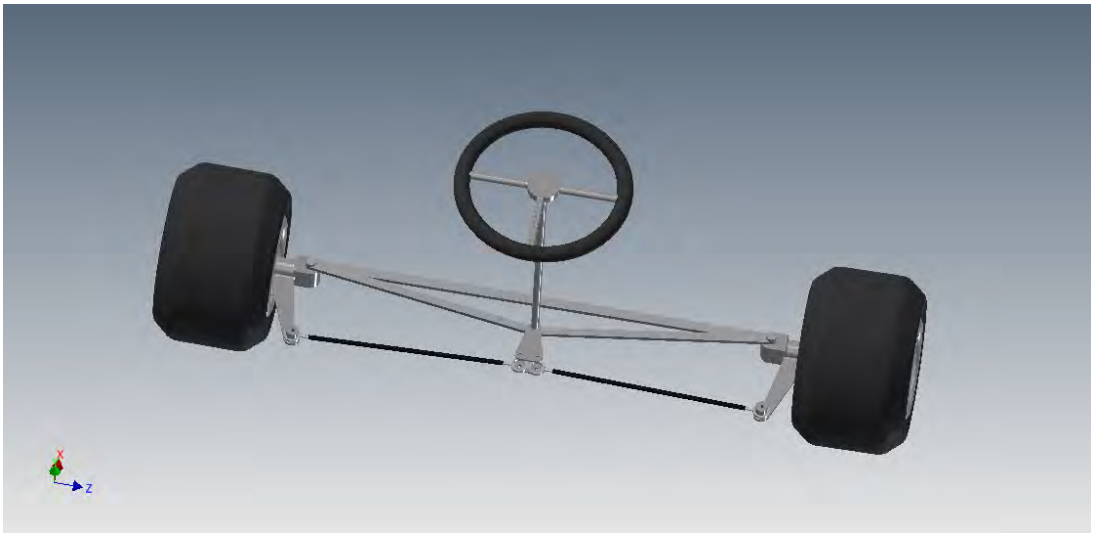

Figure 90: Steering CAD Model

joints with this new joint. For the steering system, an additional planar loop revolute joint from the Modelica MultiBody Library is needed to close to kinematic loop.

11.4.2 Adding Multimedia Elements

We take advantage of the Modelica Standard Library to add components into both the drivetrain and steering models. Since the Inventor-to-Modelica translator doesn't recognize the wheels as unique components, we attach wheels with low masses and inertias to the revolute joints at the axle.

We also add a DC permanent motor with properties that approximate the real motor properties. The completed drivetrain model is shown in Figure 91.

In the steering assembly, we drive the steering wheel with a torque to simulate a driver or self-driving control system. The completed steering Modelica model is shown in Figure 92.

11.4.3 Modelica Modeling Assumptions

We are currently unable to model slippage of the wheel with open source Modelica libraries. The Modelica Wheels and Tires Library looks promising but models in this library are entirely mathematical and do not include support for OMEdit's graphical editor.

11.4.4 Modelica Simulation Environment

We run a simulation of the drivetrain. We use a simple motor to drive the axle and and results can be seen in Figure 93. We also simulated a torque acting on the steering wheel. In the simulation, the torque is applied, stop, then reversed. Although the simulation is able to produce results, the solver produces "Matrix is Singular" errors. The steering simulation is shown in Figure 94.

In the graph of the position of the tires, the top curve is the angular positions of the four wheels, and the bottom curve is the angular velocities of the wheels. We are able to drive the two wheels which in turn pushes the cart and causes the front wheels to turn as well.

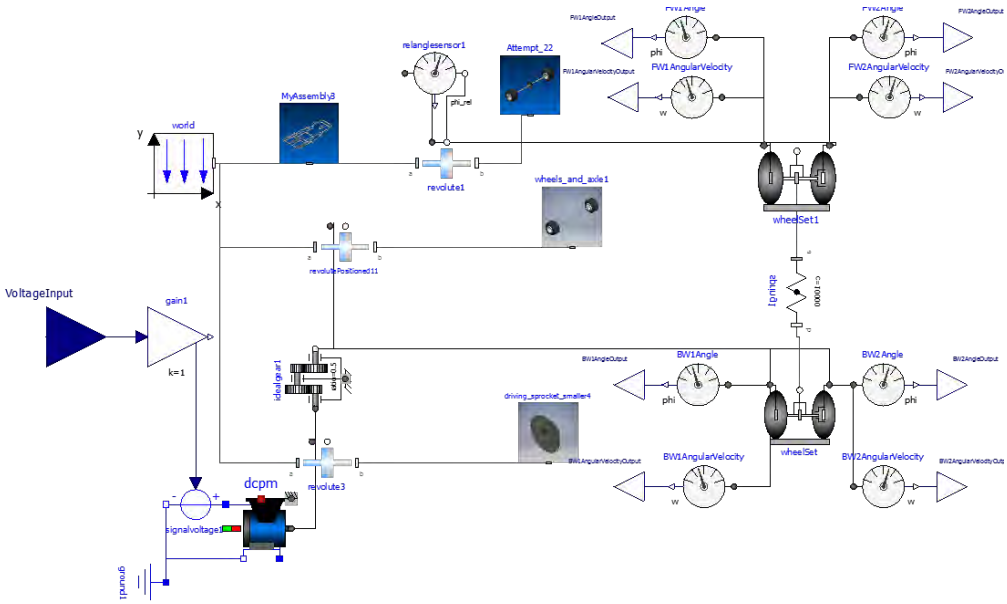

Figure 91: Drivetrain Modelica Model

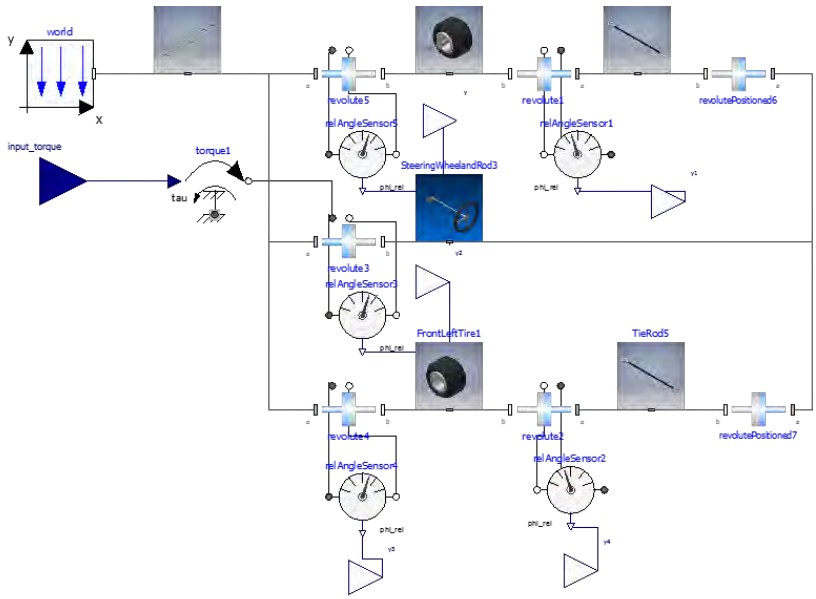

Figure 92: Steering Modelica Model

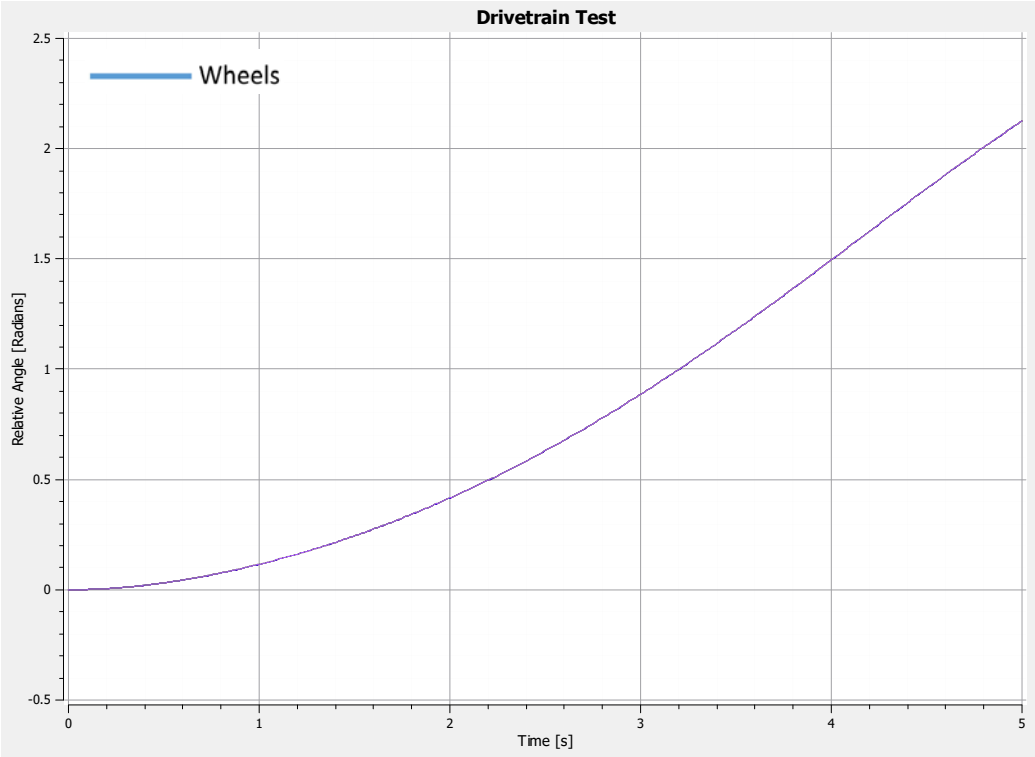

Figure 93: Drivetrain Simple Test

11.5 Functional Mock-Up Interface (FMI)

11.5.1 FMI 2.0 Model Exchange

Without understanding the singularity errors in Modelica, we are unable to produce a working FMU for either model.

11.6 Going Forward

So far we have modeled most mechanical components and behavior of both systems but are unable to model slippage and rolling of the tires. We are still experimenting with different libraries and methods to solve this in Modelica. We are also having trouble producing an error free simulation in Modelica. This prevents us from creating an FMU to bring to LabVIEW for simulation and control system design.

12 Conclusions

We have demonstrated the feasibility of a workflow that greatly simplifies the creation and use of dynamic simulations in the design and construction of mechanical systems and the implementation of controls for those systems.

Modelica has proven to be an effective intermediate representation to translate 3D models in Inventor to simulations that can be run in the LabVIEW simulation environment. Modelica's Functional Mock-up Interface (FMI) provides a well-defined protocol for porting models from one environment to another.

Figure 94: Steering Test

Despite showing feasibility, our work in this phase of the project has also turned up significant gaps. Each of the testbed sections above highlights specific gaps in the translation process that relate to the mechanics of the specific systems. These gaps tend to relate to particular mechanical structures and also to the fact that the translation software from Autodesk is still in very preliminary form.

A more meta issue is that the model in Inventor does not retain detailed information about the non-mechanical portions of the system. Most notably in these examples and testbeds those are the sensor, actuator and electronic components. For example, the mechanical characteristics of motors are retained in the modeling information but not the electrical characteristics such as winding resistance, torque constant, or even type of motor. In the testbeds documented here those components were inserted manually into the Modelica models before porting them to LabVIEW simulation.

These are solvable problems and the overall conclusion is that an integrated workflow from CAD to control can be created.

copyright ©2015 University of California, College of Engineering