MMTO Internal Technical Memorandum #04-3

Smithsonian Institution &
The University of Arizona®

Control System Prototyping — A Case Study

D. Clark

July 15, 2004

Introduction

In the spring of 2004, MMTO began a project to upgrade the MMT servo control system to improve overall tracking and wind buffeting rejection. In support of this effort, a consultant, Ed Bell, was hired to assist with the controller development. The author, in turn, designed and built new hardware and software tools to collect system data, simulate its behavior, and implement a new prototype controller. The data collected with the new tools were used to refine and develop system models that subsequently drive controller design. Using this approach, provided you have a sufficiently accurate model, the control loop performance can be predicted with a great deal of confidence. This paper describes a case study done on a spare f/9 hexapod actuator that encapsulates all the steps involved in data collection and controller design that will be needed to implement a prototype controller on the MMT.

The Control System Prototyping Problem

It is often the case that engineers are given an electromechanical actuator of one sort or another and required to build a controller for it. In this case, "controller" would mean some sort of computer, with a user interface for interacting with the physical device. The usual solution is to design and build a "one of a kind" control system with whatever hardware and software would seem suited to the task. This approach is necessarily interdisciplinary, involving hardware designers, technicians, and software engineers to put together a complete package.

Much of this design work can be avoided, or at least enormously speeded up, with the use of modern computing tools. The computing tool of choice here is MatLab with Simulink and some add-on tool boxes to bridge the gap between pure simulation and realization of actual control hardware and software.

Why Use Matlab and Simulink?

MatLab is an industry-recognized software suite for scientific numerical applications. Simulink is the add-on toolbox for simulation of physical and software-based systems, using graphical design entry for building models of the physical systems and processes of interest to the design engineer. It contains many methods for analysis and visualization of data, making the work of putting together system models and presenting numerical results fairly easy and problem-free.

To extend the functionality of Simulink, MMTO owns licenses for a few more of The MathWorks' toolboxes for Simulink, most notably Real-Time Workshop and xPC Target. Real-Time Workshop is essentially a Simulink diagram parser that generates C-code for a particular target computing environment. The MathWorks supports over a dozen targets, with many 3rd-party vendors also providing target support. For rapid control system prototyping, we use xPC Target.

What is xPC Target and Why Do We Use it?

The MathWorks supplies an add-on Real-Time Operating System (RTOS) kernel they call xPC Target. It is designed to run on *any* x86-class PC, from desktop PCs to rack-mount units to single-board CPUs. It is a fairly lightweight RTOS that makes implementation of real-world applications from Simulink easy. Using this as the software target we get:

- Seamless integration with Matlab and Simulink, keeping all the design tools in a single computing environment.
- When new code is built it is automatically downloaded over the network to the target machine, making code changes and updates simple and fast.
- Native telemetry and data logging, facilities that are absolutely necessary for measurement of the real-world behavior of the system under test.
- Built-in GUIs for interacting with the target machine and collecting data, plus real time scopes for viewing signal levels during execution.
- The usual OS kernel services such as timers, a network stack, file i/o, and low-level i/o drivers, freeing the designer from having to write this.
- Bundled hardware i/o drivers written by The MathWorks. Over 150 different i/o boards are supported from dozens of vendors, obviating the need for custom-written hardware drivers.*
- Control system hardware is cheap and easily available, as PCs are a commodity item.
- Built-in benchmarking tools for estimating the total available performance from your particular target PC.
- Code optimization and control over the general execution of the software represented by the Simulink diagram are available if necessary.
- The size of the data logging circular buffer is controllable, and the kernel keeps track of whether the data log buffer has rolled over past the end.
- The kernel detects CPU overloads, i.e. when the control algorithm execution time exceeds the time available inside a single sample period.
- The kernel optionally logs the total execution time for your software, along with an average and maximum times, information that is necessary for deciding on design optimizations.

The MMTO target PC is a surplus desktop PC with an Athlon 750MHz CPU and 1GB of memory. The kernel only supports a few types of network cards, so we installed an Intel Pro 10/100 board from the supported hardware list. No hard drive is installed, as the kernel boots from a floppy disk. The i/o hardware is an SBS PCI-60A 6-channel PCI IP module carrier with 4 IP modules installed on it. These are an SBS IP-16ADC 16-channel analog input, an Acromag IP-230-4 4-channel analog output, an SBS IP-Quadrature 4-channel quadrature counter, and an SBS IP-Digital24 24-bit digital i/o unit. At present, we boot the kernel in text-only mode because using printf statements in custom code will print to the screen in this mode for debugging. Normally, you would use graphical mode to display graphics on the target machine.

*Note: Unfortunately, MMTO only owned one IP-module that was in fact supported. Custom Simulink drivers were written by the author to support the others. Source code will be provided to anyone who asks.

F/9 Actuator Test Article

The MMT f/9 hexapod has a spare actuator that was sent to the UA campus for repair. This was done and we began work on developing hardware changes for a future upgrade of the hexapod to the same hardware revision level as our f/5 unit. This made it convenient to use as a testbed for going through the process of control system prototyping while it was in the lab.

The actuator is a linear roller nut driven by a DC brush motor, with an incremental encoder on the motor shaft and a linear potentiometer located parallel to the actuator body measuring the relative displacement of the roller nut and body of the unit. The MMTO Electronics Group built a small test box with an amplifier and i/o conditioning to handle driving the DC motor and powering the encoder and potentiometer circuitry. The potentiometer is conditioned by an operational-amplifier circuit that allows the zero point and span of the output to be adjusted. We also changed the limit switch mounting to a setup similar to the f/5 arrangement. These give a motion range of about ±10mm from the center of the roller nut travel.

The entire actuator is mounted in a fixture to hold it in place and prevent the roller nut from turning so that linear motion is maintained. The potentiometer circuit was then adjusted to provide 0V at the center, and 0.25V per mm of travel (the leadscrew pitch is 1mm). The incremental encoder is counted 4X, giving 20,000 counts per mm. Since the 1mm pitch is so basic, it was decided to design the control system to handle position commands in mm.

F/9 Control System

The design goals for the controller were these:

1. A position control loop closed on the incremental encoder.
2. Support for motions of as small as 1 micron.
3. 1mm step changes in position completing within 5 seconds.
4. Absolute positioning based on a short startup initialization from the potentiometer reading.
5. Aware of limit switch status, stopping motion at limits and allowing reversing out of the limit.
6. Network-based position commands, with current position returned over the network.

To design the controller, we first must start with a model of the system. The model must be verified against actual measurements of the system, and then can be used to drive the controller design. The design can be iteratively tested on the model, with candidate controllers subsequently tested on the actual hardware. The model-based controllers are compared with the performance of the testbed controller as a check on the model/controller simulation.

F/9 System Models

For this project, we obtained models with two different methods: (1) reducing open-loop measurements of the actuator response to continuous-time Laplace transfer functions; and (2) using the physical constants of the DC motor and the actuator to develop a simulation-only model. As it turned out, both approaches were useful, with the final design utilizing a synthesis of the two methods.

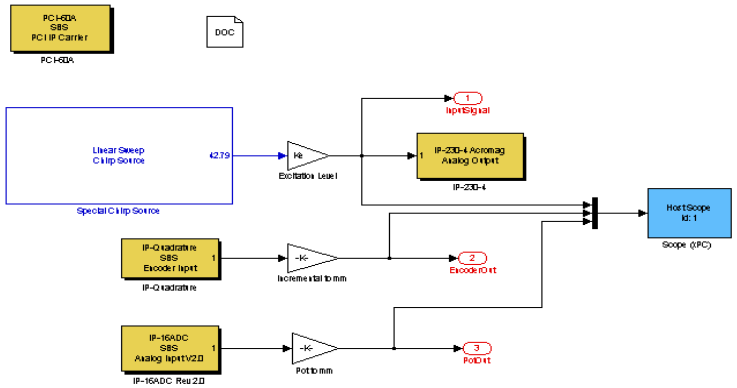
Open-Loop Method

In the open-loop method, a signal of a known amplitude and frequency is applied to the system under test, and the response signal is measured. The ratio of the input to the output becomes the transfer function magnitude response, and any phase shift input to output its phase response.

Simulink provides several signal sources for excitation. One of these is a continuous sinewave chirp signal that begins at some user-selected frequency and sweeps up to a selected target frequency at a specific target time. At first blush, this was a useful source block, but it soon was obvious that the "factory" chirp source doesn't provide enough cycles at low frequencies to give good input/output estimates. A custom Simulink chirp source block was then written by the author to provide a chirp with user-selectable frequencies, number of cycles per frequency, frequency step size, and final target frequency. This gives excellent transfer function estimation results. This chirp source is currently being used by both MMTO and LBTO for system excitation and response measurement.

Once the input/output data are in hand, Matlab has a function in the Signal Processing Toolbox called **tfe** that estimates a transfer function based on a sliding-window FFT of the input/output data. It returns complex numbers for the response, from which the magnitude and phase can be extracted. This complex vector is then passed to the function **invfreqs** to get the Laplace-transform numerator and denominator coefficients up to an order selected by the user.

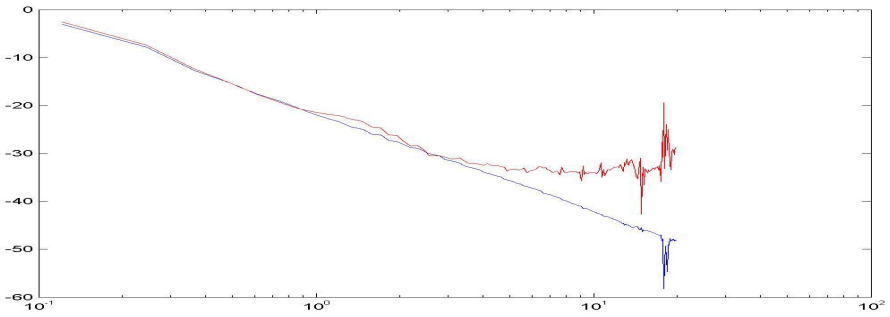
The Simulink diagram used to generate the real time code for the open-loop data collection is shown here:



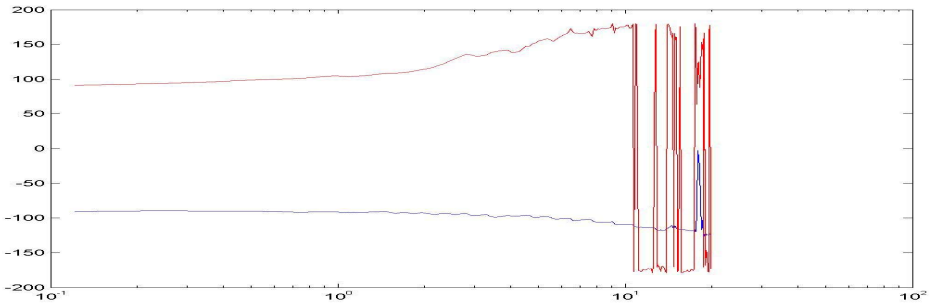
Yellow blocks are i/o drivers, blue are xPC Scopes, blue foreground are custom Simulink blocks, red are signal output ports exposed to the datalogging system, and black blocks are standard Simulink library blocks.

The chirp signal response was applied to **tfe** and results in these transfer function graphs:

Magnitude Response



Phase Response



As can be seen, the encoder and potentiometer are scaled to mm, and the excitation signal voltage is logged after passing through a user-selected gain. In addition, all the signals are multiplexed and presented to a Host Scope, a TCP/IP connection object that runs on the system host PC for real time display of the signals during operation. Logged data are stored on the target machine and transmitted over the network connection to the Matlab variable workspace when requested. The results can then be manipulated and stored on disk.

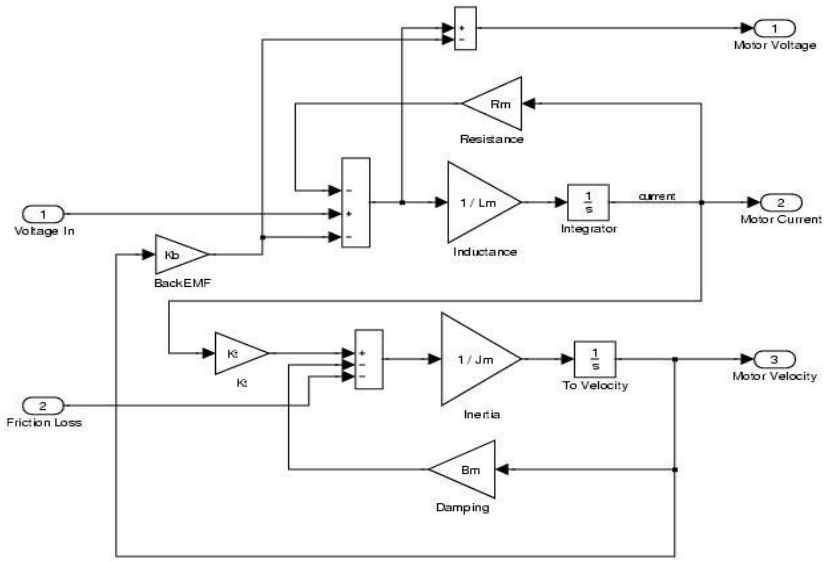
The noise of the potentiometer makes the response a bit nonlinear, so it was clear that the design would have to include low-pass filtering of the potentiometer output. The incremental encoder was quite clean.

Physical Model Method

With the physical model, the engineer represents the system under consideration in Simulink using the basic physical constants and behavior of the system. This can be time-consuming and error-prone, and it is highly recommended that physical models always be compared to actual measurements whenever possible.

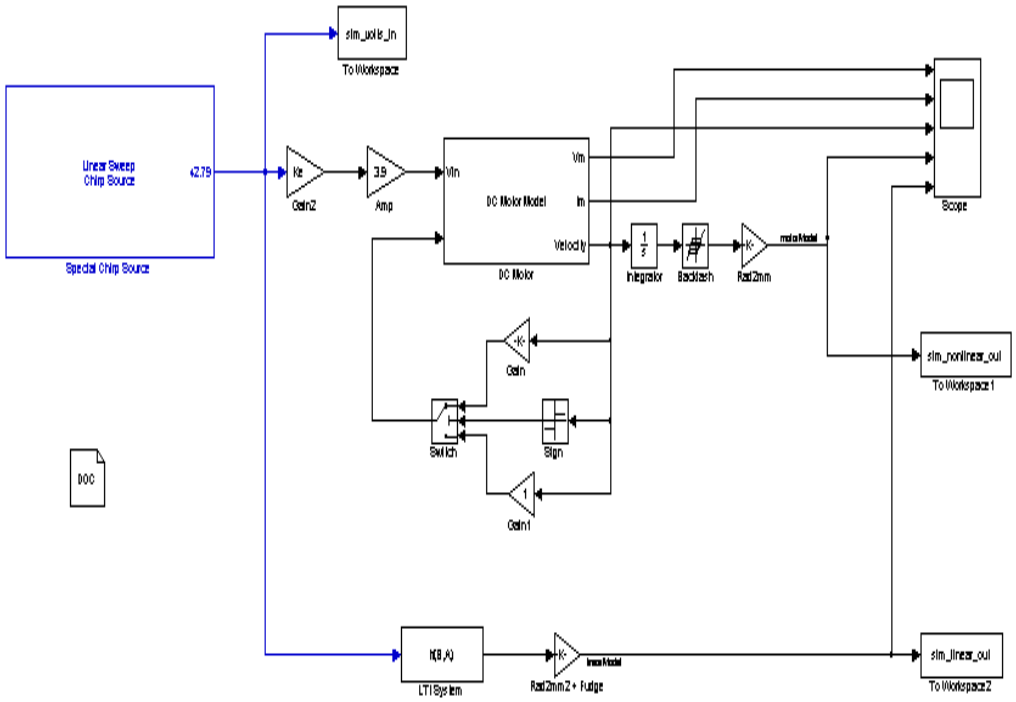
For this model, we can take into account nonlinear effects such as friction that are not available in the pure Laplace representation, and so this model complements the Laplace version of the model.

A Simulink diagram with a DC motor with nonlinear components in the loop, along with the continuous-time Laplace model, was developed for comparing the two modeling methods. Below is the DC motor model:



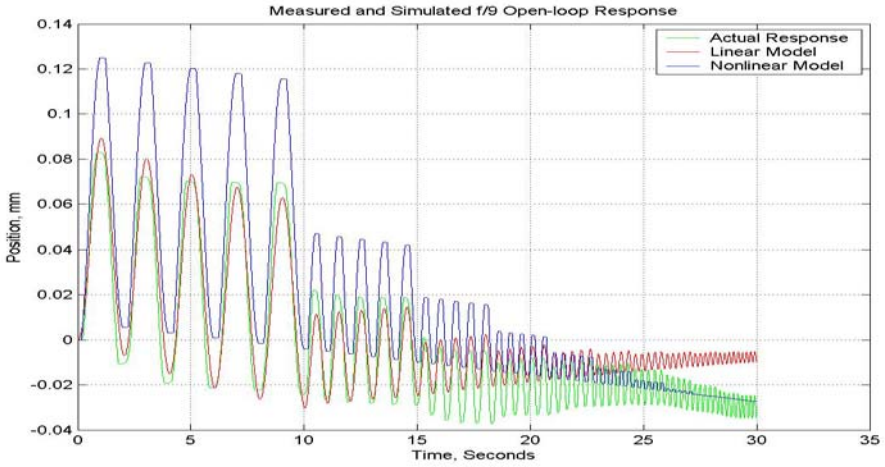
Since the test-fixture amplifier is a voltage source, this model lends itself well to the actual system. The input DC voltage is resisted by the motor resistance and inductance, producing a current through the motor, which then develops torque according to its torque constant, Kt. This torque, divided by the motor (and load) inertia, results in acceleration. A single integration gives the motor velocity, which feeds back and gives rise to the Back-EMF and velocity damping terms. An additional input for friction allows the friction torque loss to be added to the model.

The two models, linear and nonlinear, then appear in this Simulink diagram:



The DC motor has a friction term proportional to its velocity resisting the torque, and a hysteresis term in the position output, which is formed by an additional integration of the velocity. The velocity-friction term has a dependence on the sign of the velocity to represent the friction differential of the roller nut depending on the direction of motion. The continuous-time model, on the other hand, is just the **tfe**-produced Laplace coefficients supplied to an LTI System block. The input/output data are then copied to the Matlab workspace with To Workspace blocks. The input chirp signal is the identical block used in the open-loop real time model.

With the input/output from the Simulink model and the real time model data in the Matlab workspace, we can easily graph the results for comparison:

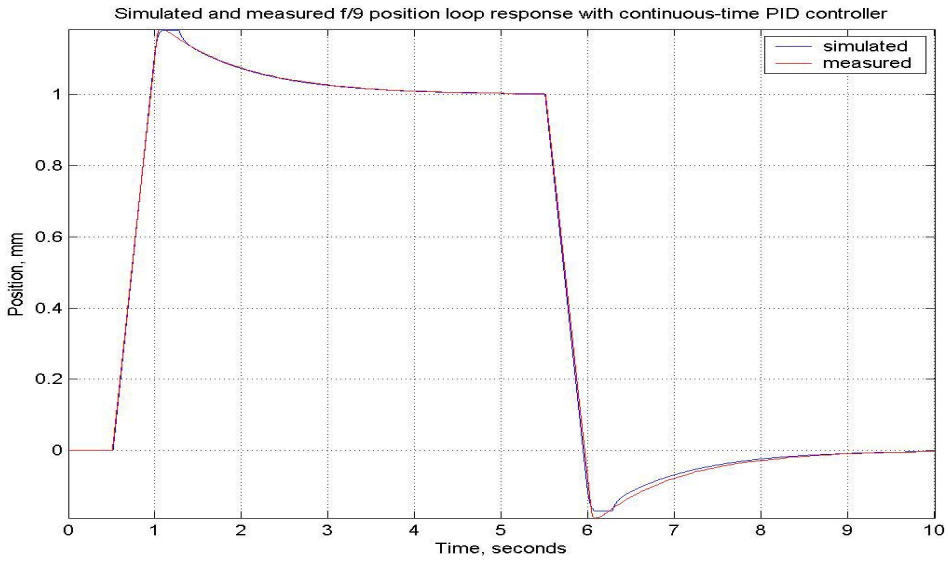


As is evident, the simplistic friction model overstates the friction behavior a bit, while the continuous-time model gives good general agreement with the amplitude and trend of the data. A close look at the nonlinear model and actual outputs shows the effect of the hysteresis at the signal peaks, which are clipped slightly. With this level of model agreement, it was decided to begin the control loop design, as increasing the model accuracy was judged not worth the additional time.

For a first cut at the design, a Simulink diagram was generated with the two models in place and a PID controller around the motor position signal. A standard Simulink PID block was inserted and a step signal source was used to simulate the closed-loop step response.

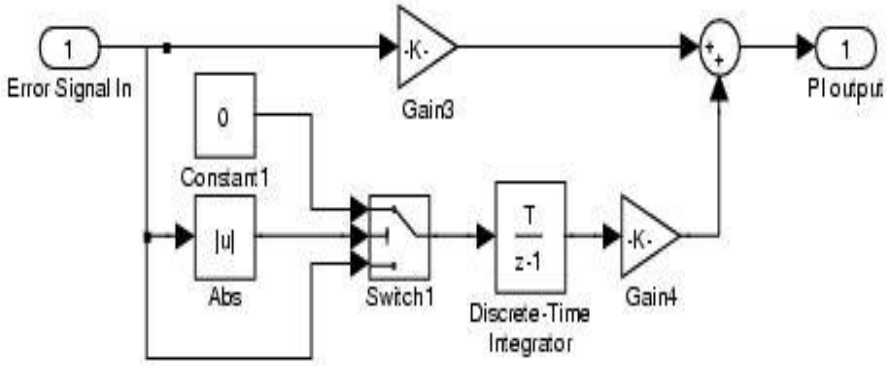
The design with the continuous-time PID block was iterated until an acceptable-looking response was achieved. This design was then transferred to a real time version and tested on the actuator.

The results comparing the measured PID loop response and the nonlinear model with the same PID control law are shown here:

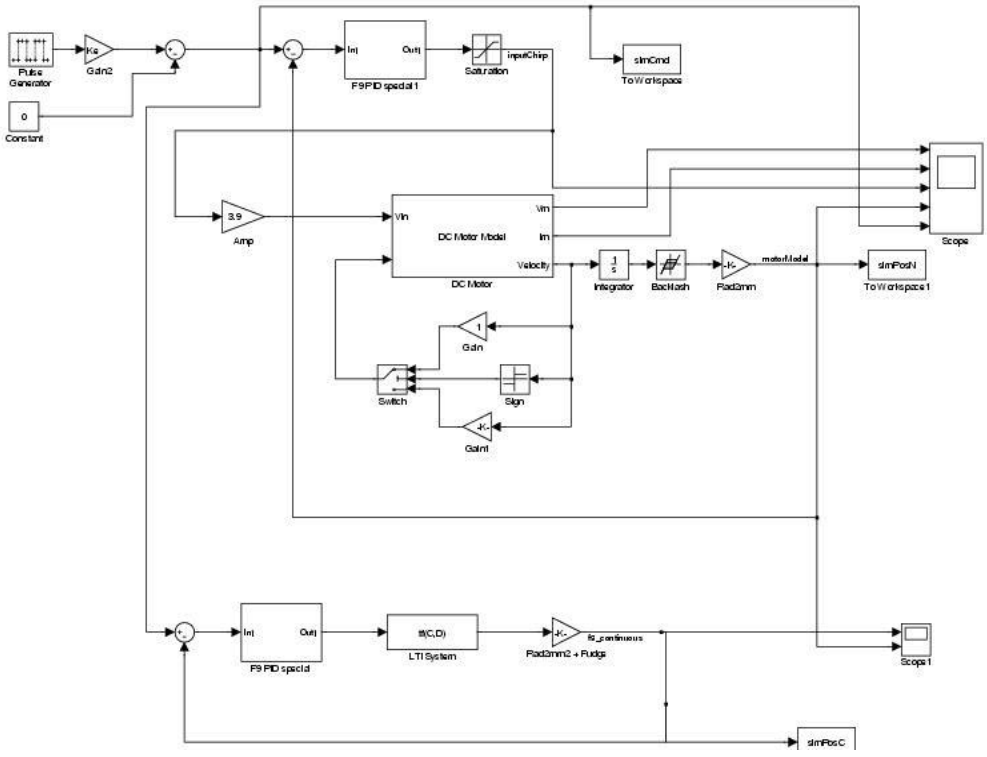


More work was done with this controller to eliminate the overshoot (due to the I, or integration, term) by adding Derivative (D) gain, but this led to very sluggish behavior that made small displacements take an unacceptable length of time.

The PID control was then abandoned in favor of a simpler discrete-time control law that maintains a Proportional gain and only applies an Integral when the position error is smaller than some (arbitrary) value:

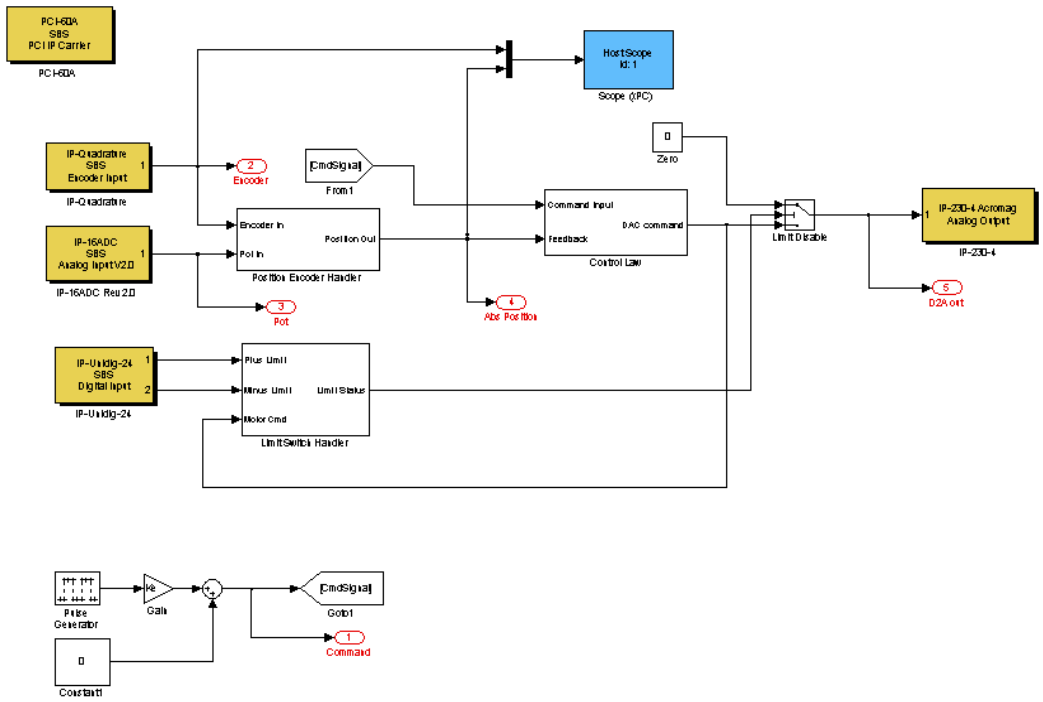


The two models were again tested with this new control law and tuned to acceptable performance levels. This Simulink diagram was used for the tuning:



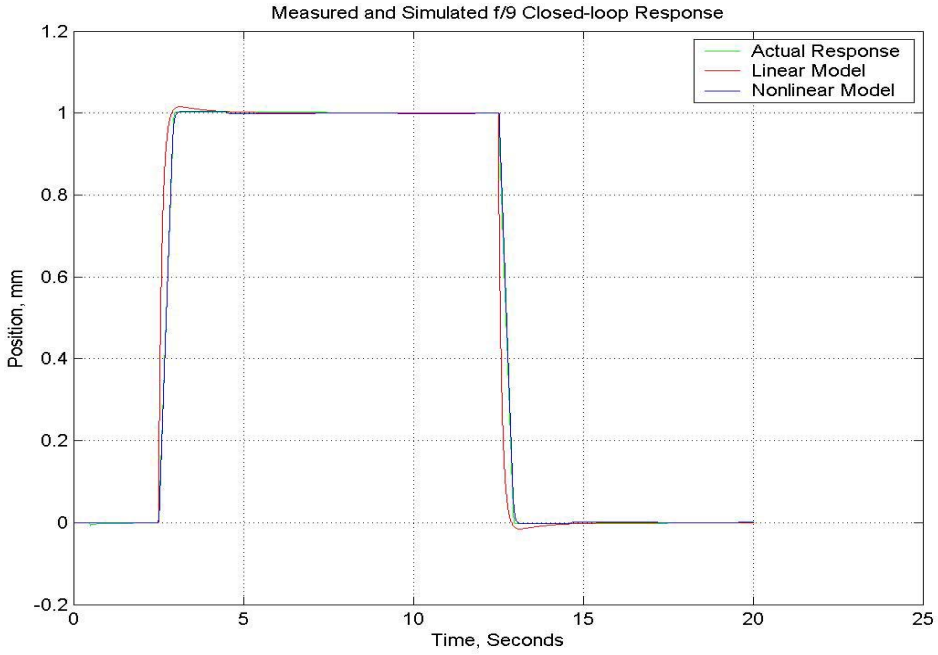
Again, the Simulink model output and the real time system output were compared as a check on the tuning and modeling accuracy.

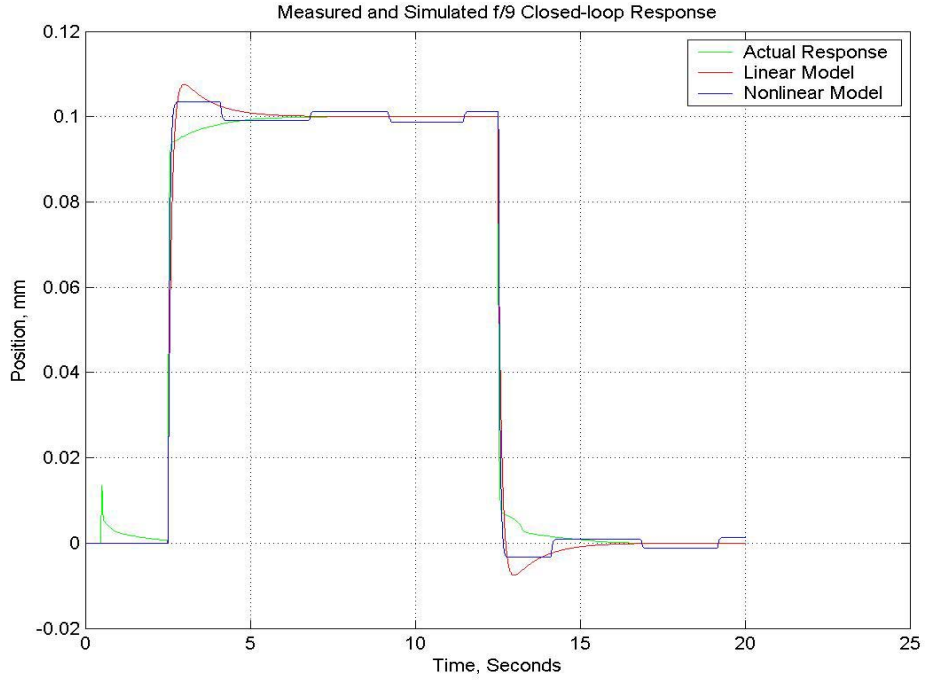
The real time version of the code was generated from this Simulink diagram:

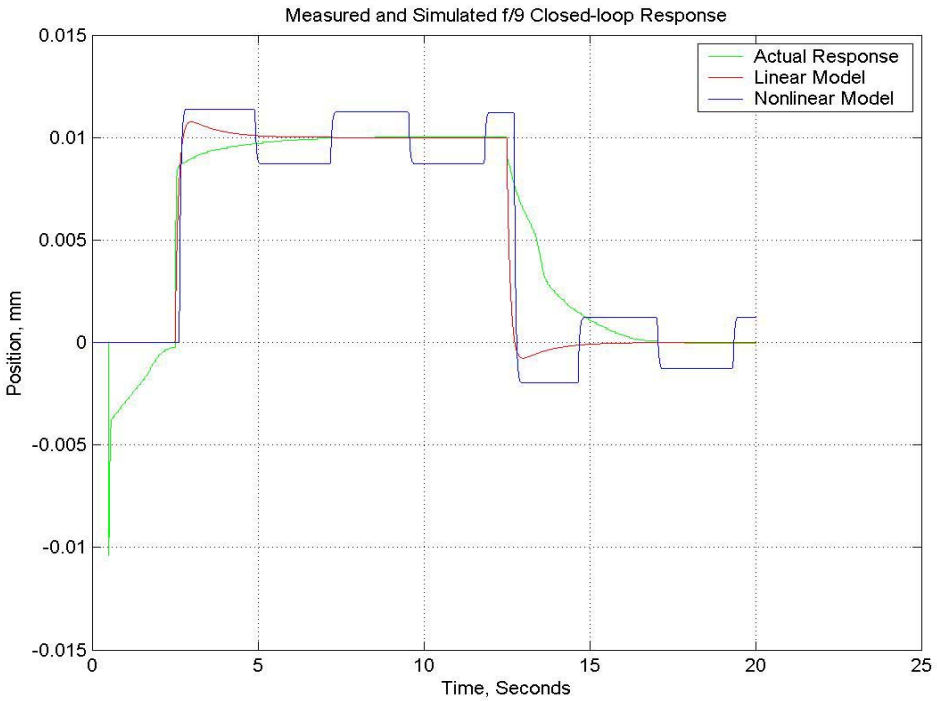


In this case, we logged the input/output signals in real time from the step command for later comparison to the models. We have also added limit switch handling and absolute positioning, discussed later in this paper.

Some comparisons of the models and the actual response are shown starting on the next page:


Measured and Simulated f/9 Closed-loop Response
Actual Response
Linear Model
Nonlinear Model
Position, mm
Time, Seconds
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
0
5
10
15
20
25


Measured and Simulated f/9 Closed-loop Response
Actual Response
Linear Model
Nonlinear Model
Position, mm
Time, Seconds
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
0
5
10
15
20
25



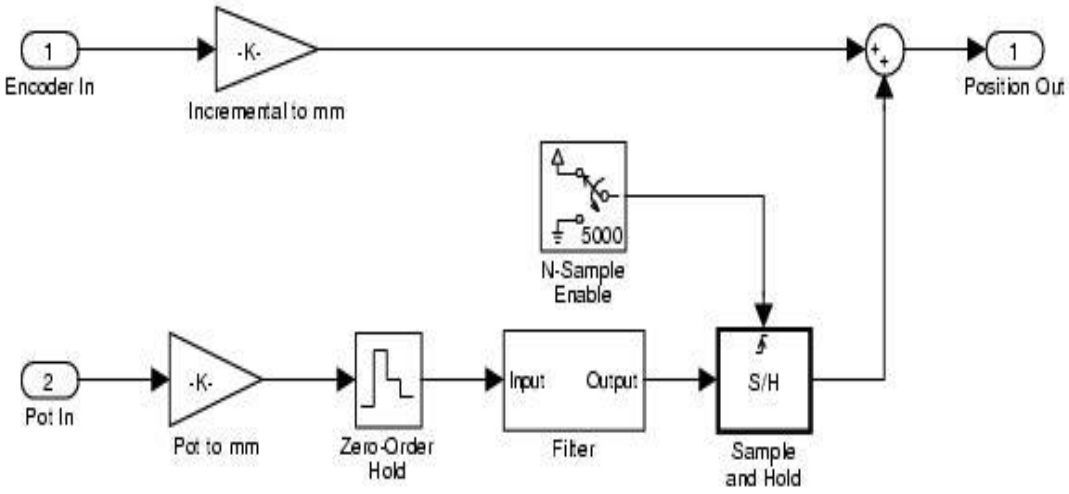
The models all gave good general agreement, though it is clear the nonlinear model's friction setup was off by a significant amount, especially for the small displacements. In every case, however, the Integral term took care of any stiction effects and the commanded motion was completed successfully. We met design goals #1-3 with this controller.

Extending Controller Functionality

The final three design goals of absolute positioning, limit switch awareness, and network i/o are also implemented and tested with the following additions to the real time code model.

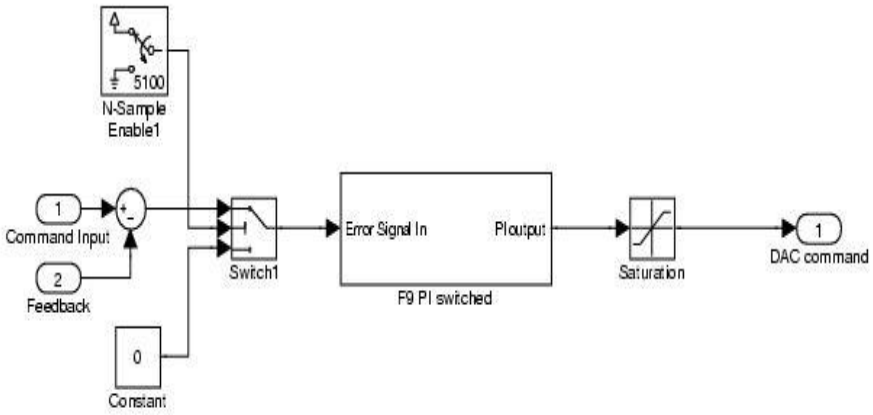
Absolute Positioning

Since we close the loop around the incremental encoder, which initializes at a position of 0 counts whenever the code starts up, we want to find an absolute position, latch the value, and add it to the incremental encoder position so that any subsequent motion is measured by the encoder and varies the absolute position. This is accomplished by the "Position Encoder Handler" subsystem in the controller diagram.



The encoder is scaled to mm and applied to a summing junction. The potentiometer is also scaled to mm and passes through a low-pass FIR filter with a low passband. For the first 5000 sample periods of the software, the sample and hold block is open, allowing the filter to settle. At 5001 samples (5.001s at the 1kHz loop update rate), the S/H goes into hold mode, latching the potentiometer value, which is then added to the encoder at the summing junction.

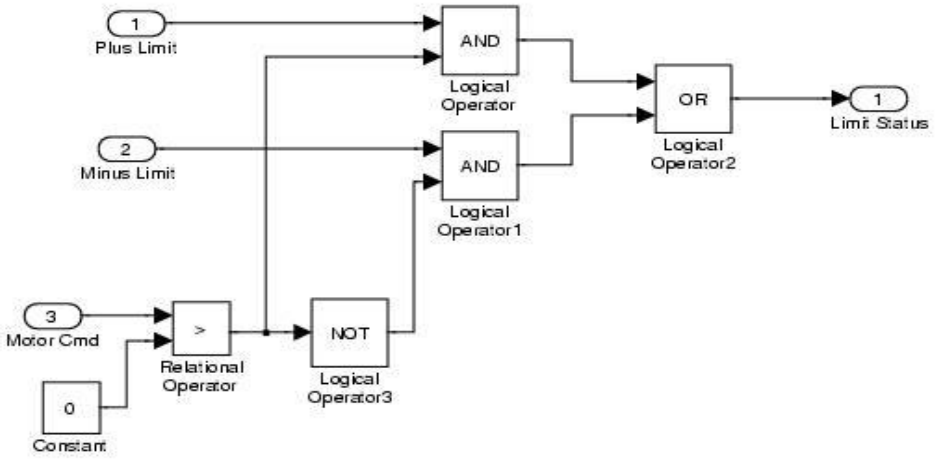
We had a race condition that occurred when previous to latching the absolute position value, the control law integrator would wind up and give erroneous output commands to the DAC. This was corrected by forcing the controller error signal to zero during the initialization phase:



Limit Switch Handling

We wanted to make sure limits couldn't be exceeded, and back away when the limit is tripped. We added two digital inputs tied to the normally-closed switch inputs, which have their common terminals wired to circuit ground. Since the IP-Digital24 has pull-ups on all its signal lines, when the switch is tripped the limit signal will go high. It will also be high if for some reason the limit input is disconnected.

The addition to the real time diagram for this is instantiated in the "Limit Switch Handler" subsystem in the controller:



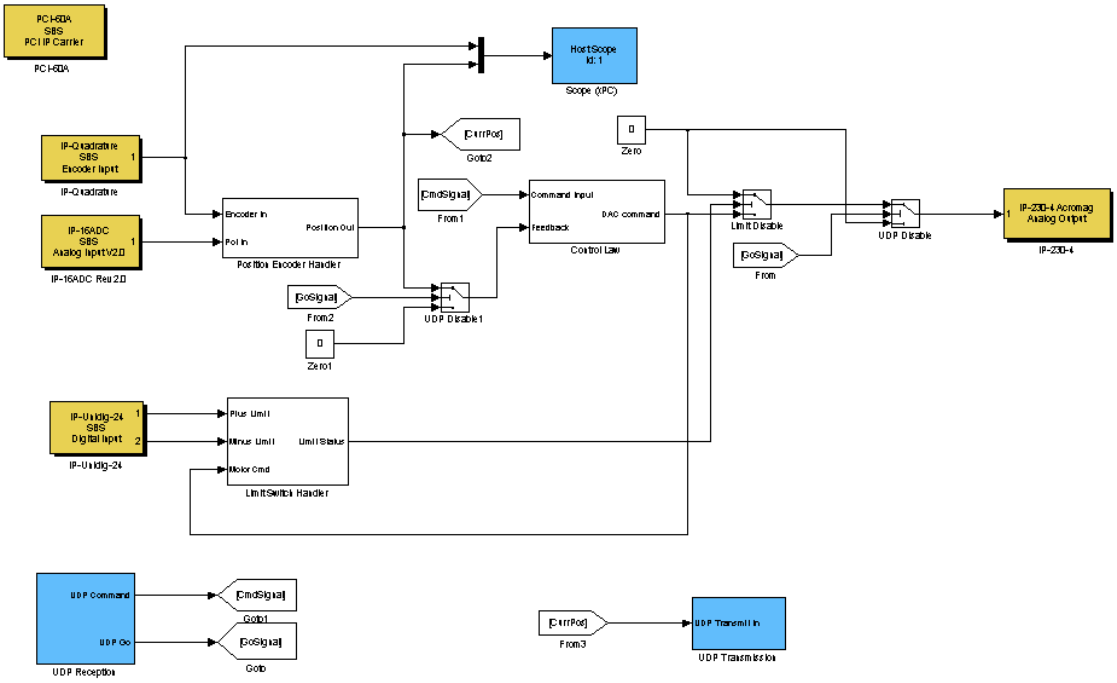
If the PI loop command is positive, the Plus Limit AND the command drive the OR gate, and likewise for the Minus Limit AND a negative command signal. If the OR operation outputs a 1, the DAC command is forced to 0V, stopping the motion. Conversely, if the limit is active and the command changes sign, the DAC is connected to the control law output and motion is enabled.

Network Commands and Returning Current Position

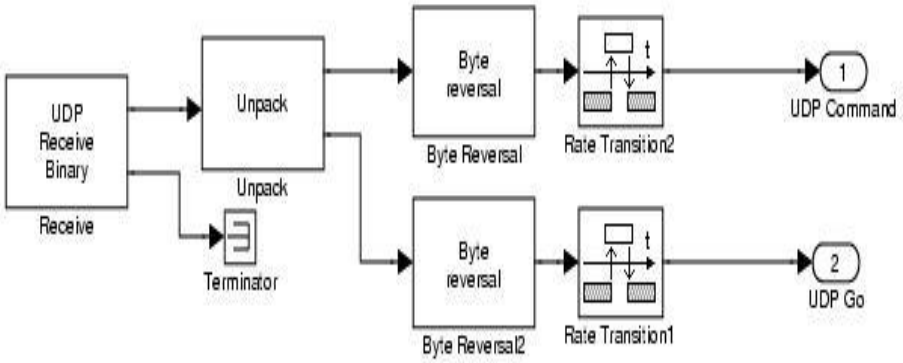
The final piece of the puzzle is adding network i/o for getting user inputs and displaying the current absolute position over the network. The xPC Target block library contains several blocks for doing UDP transmissions and reception, and so these were used along with an author-written Java GUI to do the user input/output.

The UDP receive block will hold the last-received value from the network, and starts up with a user-specified value, so that if new packets are not (or never) received, the behavior is well-defined. The default for the startup is 0.0, so we take advantage of that to provide some safety in the controller command handling; the reception expects two double-precision numbers, one of which allows the command to actually get into the controller, and the command itself. The "GoSignal" number is also used to force the feedback value to 0.0 so

that integrator windup after startup doesn't occur. Since UDP is officially connectionless and unreliable, the "Go Signal" is also used to qualify the received packet to try to ensure that the received data was not garbled; should the "GoSignal" number be different than expected by the UDP Disable switch, the DAC remains at 0V. The final version of the controller with UDP i/o looked like this:

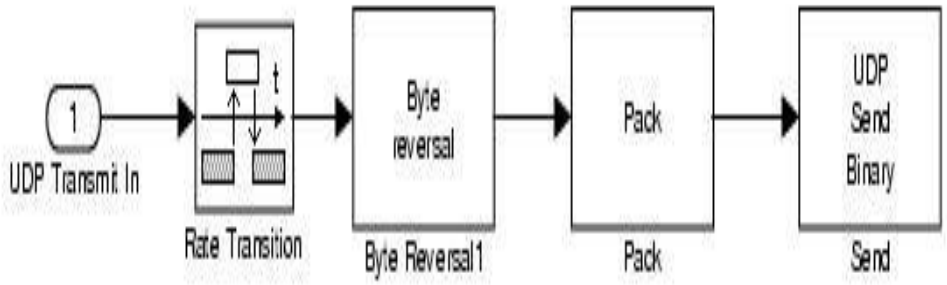


The UDP reception subsystem contained just a few blocks:



The UDP receive block samples the network at 0.5s intervals looking for new command packets. Should one not be received or is bad, it will hold the last value. A rate transition block gets the slow UDP data into the fast (1kHz) controller. *Simulink systems with more than one sample rate must have the slower sample times at integer multiples of the base (or fastest) sample rate.*

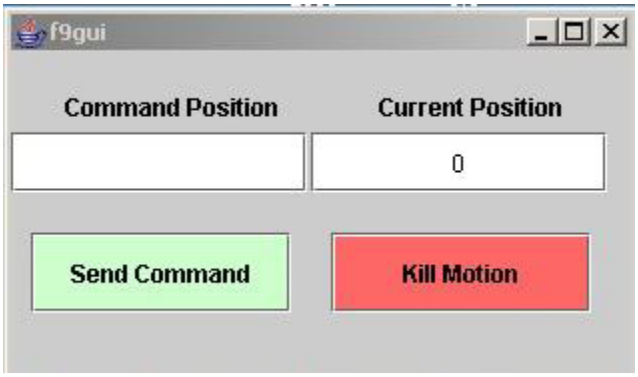
The UDP transmission is likewise simple with another rate transition for sending current position updates every 0.5s:



User Interface

Finally, some sort of user interface to test the whole controller was needed. One straightforward (and hopefully platform-independent) way to do this is to generate a GUI for Java. The author wrote this GUI so that command packets are sent over the network whenever the "Send Command" button is clicked. Clicking the "Kill Motion" button turns off the "Go Signal" number in the controller, zeroing the DAC and terminating any motion in progress. A Swing Timer listens for current position packets every second, and updates the GUI display accordingly. User commands are parsed and formatted into a number representation of NN.NNN, and current position numbers also receive this formatting, so that 1 micron commands are supported. No attempt to apply software limits are implemented here; the GUI happily sends commands that exceed the physical limits of the system. The controller just as happily complies until the limit switch is reached. Bad user entries are ignored and an error message is displayed on the GUI.

The final Java GUI window looks like this:



Conclusion

The Simulink environment, coupled with the integrated data collection and reduction tools available with xPC Target and MatLab, has proven to be a reliable and accurate way of quickly identifying system behavior, qualifying candidate controllers, and implementing them for testing. Built-in network i/o is simple and easy, and user interfaces can be efficiently integrated into a controller to complete the package. The rapidity of control software development using automated graphical programming software leads to tremendous gains in development cycle times. This will be a valuable tool for MMTO servo system design and testing.