Controlling the Speed of a Magnetically-Suspended Rotor with Compressed Air

by

Robert P. Jantz

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science in Mechanical Engineering

Auburn, Alabama
May 9, 2011

Keywords: dSPACE, PD control, Simulink, air turbine, magnetic bearings, transfer function, variable coefficients

Copyright 2011 by Robert P. Jantz

Approved by

George T. Flowers, Chair, Professor of Mechanical Engineering
David Bevly, Philpot-WestPoint Stevens Associate Professor of Mechanical Engineering
Song-yul Choe, Associate Professor of Mechanical Engineering
Dan Marghitu, Professor of Mechanical Engineering

Abstract

Magnetic bearing research is generally concerned with the development of the automatic control systems required to levitate a rotor, driven by either an electric motor or air turbine. The regulation of rotor speed in research has often been accomplished by manual methods, turning a rheostat in the case of an electric motor and by adjusting a regulator or valve if the drive is by an air turbine.

Automatic control of rotor speed would facilitate research using magnetic bearings. This control would assist the development of adaptive disturbance rejection techniques since rotor speed could be easily adjusted for any change in the desired rejection frequency. Automatic speed control could also be central to health and containment strategies for magnetically suspended flywheels used in the control of space structures. If cracks are detected in a flywheel through health monitoring, the speed of the rotor/flywheel could be automatically reduced to a level where the damaged flywheel could be temporarily operated.

This work details the development and implementation of a control system to automatically and precisely regulate the speed of a magnetically suspended rotor and flywheel. Development began with the installation of an electronic flow control valve and all instrumentation needed to measure rotor response. Once all hardware was in place, a Simulink model of the entire system, actuator (electronic valve) and plant (air turbine, magnetic bearings, rotor and flywheel), was created. This model was developed using system identification techniques where a step input is applied to the plant and its response is measured. A transfer function of the plant was derived from these tests, and it relates volumetric flow rate of air to rotor speed and uses variable coefficients. The operation of the electronic valve was too complex to be described by differential equations or transfer functions. Thus, a model of it was written in software and included in Simulink using an embedded MATLAB function.

The Simulink model was then used to develop a speed controller for the simulated system. Simulations established the type of controller and its gains. A proportional-derivative or PD controller was found to accurately regulate the speed of the simulated system. When complete, this controller was combined with the actual magnetic-bearing controller to create the software necessary for bearing and speed control. This software was then executed on dSPACE hardware to provide overall control of the system - actuator, bearings, rotor and flywheel. Numerous tests were conducted on this system to tune the gains of the speed controller. Once tuned, the PD controller developed in the simulated environment worked exceptionally well on the real system. The controller can maintain the actual speed of the rotor to within a few rpm of the desired for speeds ranging from 200 to over 6000 rpm.

The final part of this work involved developing instrument panels from which to operate the bearings and control the speed of the rotor. Instrument panels similar to those found in automobiles were created using ControlDesk, an integrated software development environment provided with dSPACE. A series of panels were designed and created so that variables necessary to operate and precisely control the bearings and rotor could be monitored and easily adjusted.

Acknowledgements

I thank my advisor Dr. George T. Flowers for funding this research and for his assistance and encouragement in completing this project. I wish to also acknowledge the excellent education I've received at Auburn and thank those professors who provided it. A special thank you goes to my brother James and to colleague Bruce Shue. James is currently a graduate student in mechanical engineering at Auburn, and he built special hardware needed for the project and assisted me in numerous other ways. I shared a lab with Bruce for two years, and he could answer questions on many engineering subjects. Bruce was definitely the "go to guy" for me and others when the subjects were difficult.

Special thanks go to Nathan Martz for designing and constructing the magnetic bearings. Without Nathan, there would not be a magnetic-bearing system to control. Fred Buchanan and others from the Auburn University mechanical shop simplified the network of air supply pipes in the lab to inexpensively provide the air flow necessary to spin the rotor. Henry Cobb, engineer and director of the Research Electronics Support Facility at Auburn built the project box for and added the LVDT to the flow control valve. Modeling of the system would have been nearly impossible without Henry's work. Finally, I thank Bruce Smith from Control and Power, Inc. of Birmingham, AL for providing gratis advice, pipe fittings and the flow meter used in developing the speed controller for the rotor and flywheel.

Table of Contents

Abstract ii

Acknowledgements iv

List of Figures viii

List of Tables xi

Nomenclature xii

Chapter 1 - Introduction and Literature Review 1

1.1 Air-Driven Rotors 1

1.2 Process Identification 2

1.3 Transfer Functions 3

1.4 Controller Gains 4

Chapter 2 - Existing System 7

2.1 Bearing Hardware 7

2.2 dSPACE 8

2.3 Bearing Controller 11

2.4 Instrument Panel 16

2.5 Flywheel 18

Chapter 3 - Hardware Upgrades 19

3.1 Air Delivery 19

3.2 Speed Measurement 21

3.3 Flow Control 23
Chapter 4 - System Model 26
4.1 Transfer Function 27
4.2 Step Response 30
4.3 Variable Coefficients 34
4.4 MagBear - Masked Subsystem 37
4.5 Valve Position 37
4.6 Stepping Rates 40
4.7 Aalborg Valve 43
Chapter 5 - Controller Model 46
5.1 Proportional Control 46
5.2 Simulated and Actual Responses 48
5.3 Proportional-Derivative Control 51
5.4 Good Gain 53
5.5 Gain Scheduling 57
Chapter 6 - Controller Implementation 63
6.1 Controller Design 63
6.2 Testing and Tuning 68
Chapter 7 - Operating the Rotor 76
7.1 Starting the System 76
7.2 Activating the Bearings 77
7.3 Setting or Changing Speeds 79

7.4 Stopping the Rotor and System 82

7.5 All Parms 83

7.6 Amplifier Status and Calibration 84

7.7 Control Parameters 87

7.8 Input/Output 90

Chapter 8 - Conclusions and Future Work 93

References 96

Appendix A - MATLAB Utility Program 100

Appendix B - Embedded MATLAB Functions 103

Appendix C - Python Programs 110

List of Figures

2-1 Existing Magnetic-Bearing System 7

2-2 Connector Panels 9

2-3 Simulink Block Diagram 10

2-4 ControlDesk Instrument Panel 11

2-5 Bearing Controller 13

2-6 Original Instrument Panel 17

3-1 Rotameter 20

3-2 Digital Tachometer 21

3-3 Retro-Reflective Sensor 22

3-4 Aalborg Valve and Accessories 24

4-1 Simulink Model of Magnetic-Bearing System 26

4-2 Free-Body Diagram of Rotor and Flywheel 27

4-3 Rotor Speed vs. Time 31

4-4 Model of Individual Transfer Function 32

4-5 Step Response of Transfer Function 33

4-6 Torque Constant vs. Rotor Speed 35

4-7 Damping Coefficient vs. Rotor Speed 36

4-8 Simulink Model with Variable Coefficients 37

4-9 Flow Rate vs. Valve Position 39

4-10 Position Lookup Table 43

5-1 Proportional Controller 46

5-2 Simulated Response with P Control 47

5-3 Bearing and Speed Controller 49

5-4 PD Controller 52

5-5 Simulated Response with PD Control 53

5-6 Determination of Good Gain 54

5-7 Simulated Response with Modified Good Gain PD Control 56

5-8 PD Gains vs. Rotor Speed 59

5-9 PD Controller with Gain Selection 60

5-10 P, D and PD Control Signals 61

6-1 Speed Controller 64

6-2 Low-Pass Filter 67

6-3 P Control - 800 rpm 69

6-4 PD Control - 800 rpm 69

6-5 P Control - 1750 rpm 70

6-6 PD Control - 1750 rpm 70

6-7 P Control - 2500 rpm 71

6-8 PD Control - 2500 rpm 71

6-9 P Control - 3300 rpm 72

6-10 PD Control - 3300 rpm 72

6-11 P Control - 4000 rpm 73

6-12 PD Control - 4000 rpm 73

6-13 Changing Set Point - 4000 to 1000 rpm 74

6-14 Changing Set Point - 500 to 2500 rpm 75

7-1 layout_start Instrument Panel 77

7-2 Schematics Instrument Panel 78

7-3 Setting Rotor Speed 80

7-4 Viewing Velocity Plot 81

7-5 Stopping the Rotor 83

7-6 Amp Status/Cal Instrument Panel 85

7-7 Saving Calibration Data 87

7-8 Control Parms Instrument Panel 88

List of Tables

4-1 Step Response Data 32

4-2 Flow Rate and Valve Position 39

4-3 Stepping Rate and Slope 41

5-1 Simulated and Actual Response Data 50

5-2 Good Gain Values for K_d 55

5-3 Modified Good Gain Values for K_d 57

6-1 Controller Input and Output 65

6-2 Controller Configuration 68

Nomenclature

c damping coefficient (N-m-sec)

J moment of inertia of rotor and flywheel (kg-m^2)

K_d derivative gain (volts-sec/rpm)

K_i integral gain (volts/rpm-sec)

K_p proportional gain (volts/rpm)

Q volumetric flow rate of air (m^3/sec or scfm)

T_d derivative time (sec)

T_i reset time (sec)

T_{ou} time between first overshoot and first undershoot (sec)

T_q torque constant (N-sec/m^2)

t_{ss} time to steady-state (sec)

Greek Letters:

ω angular velocity of rotor and flywheel (rad/sec or rpm)

$\dot{\omega}$ angular acceleration of rotor and flywheel (rad/sec^2)

ω_{ss} steady-state speed of rotor and flywheel (rad/sec or rpm)

$\dot{\theta}$ angular velocity of rotor and flywheel (rad/sec or rpm)

$\ddot{\theta}$ angular acceleration of rotor and flywheel (rad/sec^2)

Chapter 1 - Introduction and Literature Review

A great deal has been written regarding the use of magnetic bearings to suspend a rotor. This literature generally describes the development of the control systems used to keep the rotor suspended and away from the magnets used for suspension. In testing of the magnetic bearing controllers, an electric motor is often used to spin the rotor or in some instances, a turbine driven by compressed air has been used. The regulation of rotor speed in research has often been accomplished by manual methods, turning a rheostat in the case of an electric motor and by adjusting a regulator or valve if the drive is by an air turbine. For those systems found using compressed air drive, none of them employed automatic speed control.

1.1 Air-Driven Rotors

Compressed air was employed to drive a flywheel used to study position stability in high temperature superconducting (HTSC) magnetic bearings [1]. An air turbine was not used to spin the flywheel; instead, compressed air was blown over the flywheel until it reached the desired speed. Hikihara again used compressed air in other tests conducted on HTSC magnetic bearings [2]. In these later tests, vanes were attached to the rotor and compressed air was directed against them to revolve the rotor. The air supply was shut off once the rotor was up to speed. NASA used an air impeller or air turbine to drive the rotor in their study of a passive magnetic bearing flywheel [3]. The speed of this rotor was controlled by manually adjusting a pressure regulator positioned in the air supply line leading to the impeller. A stroboscope was then used to monitor the speed of the rotor. This form of speed control was again employed by NASA to investigate permanent magnetic bearings for spacecraft applications [4]. An air turbine also drove the rotor in the magnetic bearing system built by Matras to study adaptive disturbance rejection [5]. The speed of this rotor was not maintained automatically but was set by manually adjusting a shutoff valve. It is this system of Matras' to which automatic speed control was applied.

1.2 Process Identification

A model of the magnetic-bearing system was needed before a speed controller could be developed. The transfer function portion of this model was derived from the equation of motion for the system. However, most of the coefficients of the transfer function were unknown but were found experimentally by a method known as system identification or real-time identification. This method is often used in the chemical process industries to build a model from which a controller can be designed. Generally, process dynamics are determined by applying a deterministic input (pulse, ramp or step) of one controlled variable and then measuring the output of the process until it reaches steady-state. The dynamics of the system are then given by the transient response.

Kealy and O'Dwyer compared various open and closed loop process identification techniques in the time domain [6]. The most common method used for PID controller development is one where a step input is applied to a system that is initially at rest. These step tests are completed open loop, and the model obtained is first order with dead time, where dead time is the delay between the application of the step and initial system response.

Open-loop step tests were conducted by Gajan et al. to determine process response to the addition of chlorine into a water treatment system [7]. Here, a number of tests were performed at various flow rates of chlorine. Each test was repeated several times to obtain a valid process model. Open-loop process identification was also employed to develop controllers to minimize energy consumption in complex distillation columns [8]. Step inputs were used to perturb the distillation process, and its output was registered until steady-state conditions were achieved. Since this process was non-linear, the size of the steps had a very large influence on system identification. A series of smaller steps were necessary to model the system in a linear manner.

Audits of control systems found in pulp and paper mills have also been conducted with open-loop step tests [9]. The performance of these systems has been found to degrade over time, resulting in lower production and decreased product quality. Performance degradations are the

result of process variations which ultimately affect the process transfer function and controller tuning. During an audit, open-loop identification tests are performed to establish the causes of process variations. These causes are then corrected and controllers retuned for the "new" process. Furthermore, many of the processes found in the manufacture of pulp and paper are nonlinear. However, it has been shown that linear control techniques can be applied to nonlinear processes [10]. These techniques were partially developed from open-loop process identification tests. Here, the change in a chemical concentration to a step change in water flow rate was shown to be nonlinear. However, linear controllers were developed and applied to regulate the chemical process.

1.3 Transfer Functions

The input to the magnetic bearing system is volumetric flow rate of air, and the output is the angular velocity of the rotor. A number of step tests were conducted using different flow rates to establish the dynamics of this system. With the rotor initially at rest, each test was performed by suddenly introducing a fixed flow rate of compressed air into the turbine. The velocity response of the rotor was then recorded, and from this response, the transfer function relating flow rate to speed was determined. As a result of the step tests, it was found that the system could not be characterized by a single transfer function. Instead, multiple transfer functions were necessary to describe the system depending on the desired or set speed (operating point). Since these transfer functions all had the same form, the system was modeled using a single transfer function with variable coefficients.

Transfer functions with variable coefficients are often used in the development of adaptive controllers. If process dynamics vary depending on the operating point, then a different transfer function describes the system at each operating point. Once these functions are known, a single controller can be designed that can adapt or alter its gains depending on current operating conditions. Brazauskas and Levisauskas used adaptive transfer function based control to regulate the temperature in an industrial methane tank [11]. Controllers for machining processes have also been developed using adaptive transfer functions. Milling for example is a variable process because the cutting force needed by the mill varies depending

on the depth of cut and the material being milled [12]. For productivity reasons, it is desirable to maximize the allowable cutting force applied by the mill. One way to maximize this force is to manipulate the feed rate or the rate at which material is fed to the mill. This rate can be adjusted using variable gain controllers. Depending on design, gains can be altered directly based on operating conditions or modified as a function of the manipulated variable (e.g. feed rate) [13].

1.4 Controller Gains

It was decided to try a proportional-integral-derivative (PID) controller to regulate the speed of the rotor. A PID controller was selected since it is employed extensively in a variety of industries, provides satisfactory control over a wide range of processes and is straight forward to implement [14]. Depending on the source, PID controllers account for anywhere from 50% to 95% of the regulators used in closed-loop industrial processes [15][16]. PID controllers and their variants PI and PD are also easy to build in Simulink. Furthermore, once designed, most PID controllers need to be tuned (i.e. gains adjusted) for the actual process, and this tuning was done using ControlDesk instrument panels.

There are numerous methods for calculating PID gains, including Ziegler-Nichols, Cohen-Coon, Skogestad's, Relay, Good-Gain and the ever popular trial and error. The last method requires no explanation. In the early 1940s, Ziegler and Nichols (Z-N) developed two methods for obtaining preliminary values of controller gains. With the process reaction method, controller gains are established from the open-loop step response of the process. Recall that this same type of response is used in system identification. The ultimate cycle method also developed by Z-N is a test performed closed loop using only proportional control. Here, the gain is increased until process output shows sustained oscillations. At this point, the gain necessary to produce these oscillations is determined to be the ultimate gain, and from it, the proportional gain K_p, reset time T_i and derivative time T_d are calculated. The integral gain K_i is equal to K_p/T_i,and the derivative gain is T_dK_p [17].

The process reaction method is simple to use once the process response curves have been obtained. However, this method cannot be applied if just PD control is desired [15]. In addition, Z-N often leads to the overestimation of gains, especially with the ultimate cycle method since this approach operates a process near its stability limit [18]. The Ziegler-Nichols method was not used since a PD controller worked best with the magnetic-bearing system.

The Cohen-Coon (C-C) method of estimating gains is similar to the process reaction approach developed by Ziegler-Nichols. Cohen-Coon like Z-N measures the open-loop response of a process to a step change in one input. From this response, the proportional controller gain K_c, reset time T_i and derivative time T_d are calculated. C-C differs from Z-N in that these three coefficients are determined from percentage changes in input and output [15]. Ziegler-Nichols constructs these values more directly from the process response curve. Similar to Z-N, Cohen-Coon is simple to use once the process response is known. However, C-C can only be applied to processes having a large time delay (transportation lag or dead time) between input and output. It is for this reason that the method of Cohen-Coon was not used.

Skogestad's method is another model-based approach to estimating controller gains. Similar to the methods of Ziegler-Nichols and Cohen-Coon, Skogestad requires the process response or reaction to a step input of one variable. Also similar to these other methods, Skogestad calculates the proportional gain K_p, integral time T_i and derivative time T_d from the response curve. However, unlike the Z-N open-loop method, Skogestad's approach can be applied to processes other than just those that are first order with dead time. In addition to this process type, Skogestad's method can be used with four other types of systems [19]. The gains determined with the Skogestad approach were much too great for use in both the simulated environment and the actual system.

In 1984, Åström and Hägglund proposed the relay method for estimating controller gains [20]. This method can be applied to both simulated processes and actual ones. When used on an actual process, the relay method is superior to the second approach of Ziegler-Nichols since relay minimizes the possibility of operating a plant near its stability limit [18]. In the relay method, an on-off relay replaces the controller in the feedback loop and applies step-like

control to the process. Control is held in one direction until process output exceeds the set point. Then, control is reversed until the output falls below the set point at which time control is again reversed. The constant reversal of control causes the process to oscillate at the same frequency of the control but in the opposite direction. The period of these oscillations is a close approximation of the Ziegler-Nichols ultimate period found from their ultimate cycle method. From the period generated by the relay, the ultimate gain is determined and then K_p, T_i and T_d are found using the formulas of Ziegler-Nichols. Relay was not tested since the next and last method to be discussed provided adequate gain estimates.

Another experimental approach to estimating controller gains is the Good Gain method. This method does not require a process model, but one can be used if available. Good Gain is intuitive, and its aim is acceptable stability. Acceptable stability is achieved when the first undershoot of the proportionally controlled response is barely observable [19]. To use Good Gain, operate the process with just proportional control, and adjust the proportional gain K_p until acceptable stability is achieved. If integral control is to be included, set the integral time T_i equal to $1.5T_{ou}$ where T_{ou} is the time between the first overshoot and first undershoot of the response. If derivative control is added to either P or PI control, the derivative time T_d is initially set to $T_i/4$. T_d may need to be adjusted somewhat from this initial estimate. The integral and derivative gains K_i and K_d are then calculated as always - $K_i = K_p/T_i$ and $K_d = T_dK_p$. Good Gain was used in conjunction with a process model to establish initial gain estimates for the actual speed controller.

Chapter 2 - Existing System

Automatic speed control was applied to an existing magnetic-bearing system. Prior to discussing the development of the speed controller, an overview of the current system will be provided. This overview will include all bearing hardware, the dSPACE system providing the interface between the hardware and bearing-controller software, the Simulink bearing controller and the ControlDesk instrument panel for operating and monitoring the rotor and flywheel.

2.1 Bearing Hardware

Figure 2-1 shows a flywheel on a rotor suspended by two magnetic bearings.

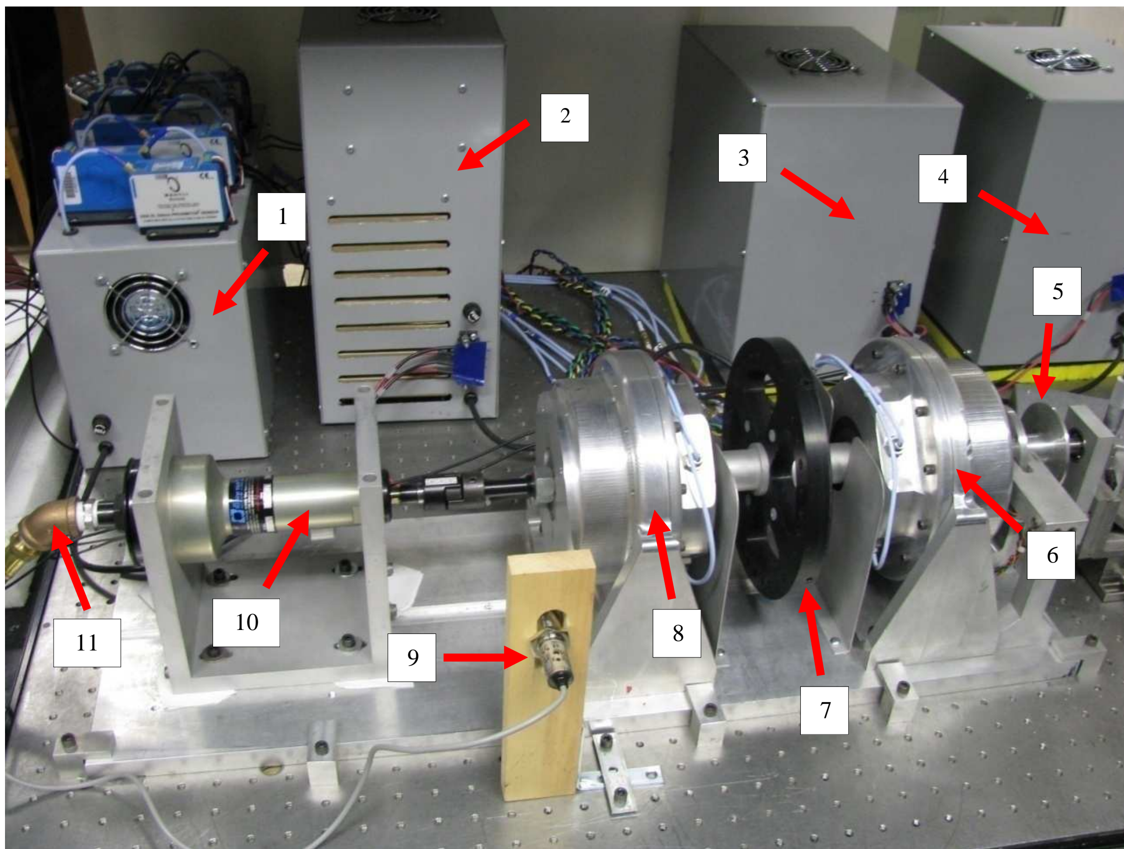

Figure 2-1 Existing Magnetic-Bearing System

This experimental device was constructed by the Air Force Research Laboratory (AFRL) as a part of the Flywheel Attitude Control Energy Transmission System (FACETS) program. At one time, the Air Force was investigating the use of magnetically suspended flywheels for combined energy storage and attitude control of space structures. When the FACETS program was complete, AFRL donated the test setup shown in Figure 2-1 to Auburn University. A condensed description of the FACETS bearing is given here. A formal discussion of it along with complete construction details are provided by Matras [5] and Barber [21].

The two magnetic bearings are located by arrows 6 and 8 in Figure 2-1. For future reference, arrow 6 identifies bearing number one and arrow 8 bearing number two. Each bearing consists of 4 pole pairs or 4 electromagnets located radially around the rotor or shaft supporting the flywheel (arrow 7). All electrical hardware needed to energize the electromagnets and suspend the rotor are indicated by arrows 1 through 4. The power supplies for the bearings are pointed to by arrows 3 and 4. Arrow 2 is directed at the box containing the amplifiers for energizing the magnets. Arrows 5 and 9 point to the sensors for the mechanical and electronic tachometers; arrows 10 and 11 identify the air turbine which spins the rotor and the air supply line to the turbine.

Arrow 1 points to the hardware providing position measurements of the rotor relative to the magnets. Position or displacement readings are calculated from voltages produced by proximity probes located inside each bearing. There are 4 proximity probes per bearing, one for each magnet, for a total of 8 probes. The rotor is then suspended or floated by constantly varying the current to the magnets so the rotor never contacts the magnets. The required currents are determined by a constant analysis of the displacement measurements. Current management and position analysis are accomplished with control software that will soon be discussed.

2.2 dSPACE

A system manufactured by dSPACE [22] provides the interface between the host computer and all bearing hardware. The dSPACE system consists of the following hardware:

- DS1005 processor,
- DS2003 analog to digital (A/D) converter,
- DS2002/2003 A/D connector panel,
- DS2103 digital to analog (D/A) converter, and
- DS2103 D/A connector panel.

The dSPACE processor and A/D and D/A converters reside in an expansion box connected to the host computer with a fiber-optic cable. Each connector panel provides the physical connections for input to and output from the computer and processor. Pictured in Figure 2-2 are the two connector panels.

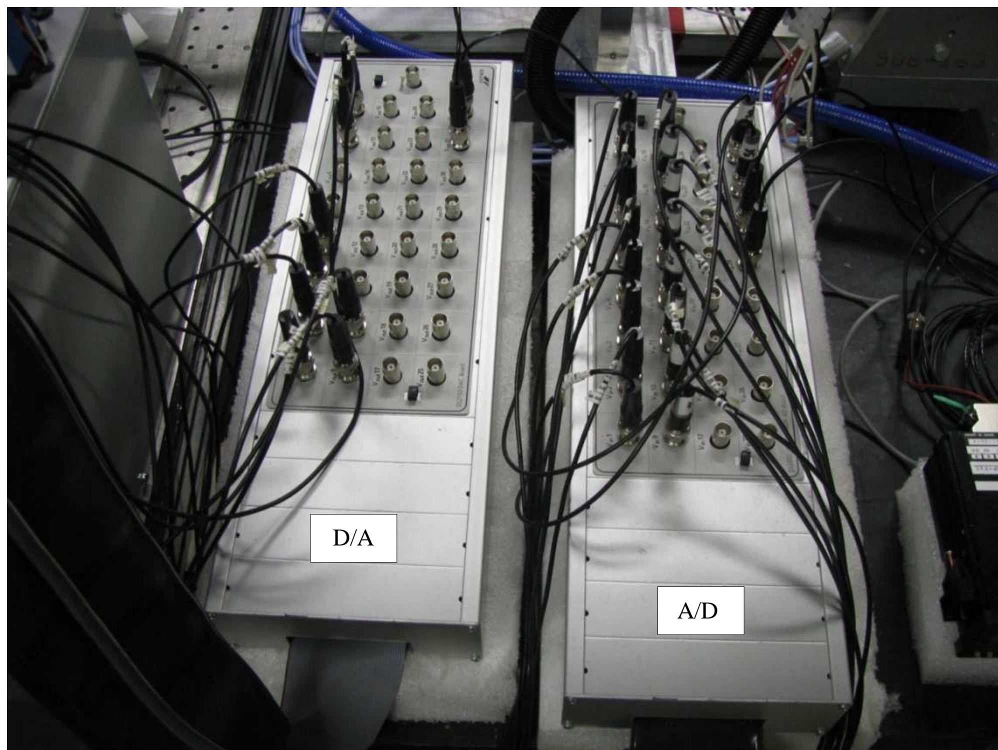

Figure 2-2 Connector Panels

The A/D panel provides connections for up to 32 input signals from sensors generating analog voltages. For the magnetic bearings, there are 8 signals from the Proximitor probes and 8 from the amplifiers. The flow-control valve and tachometers generate 4 other inputs needed to

control the rotor. Thirty-two connections are also available for the output of control signals to actuators. For the FACETS bearing, there are 8 output signals to the amplifiers plus 2 others required to operate the valve.

Control programs process inputs to and generate output from dSPACE. A control program starts as a Simulink block diagram, a sample of which is shown in Figure 2-3.

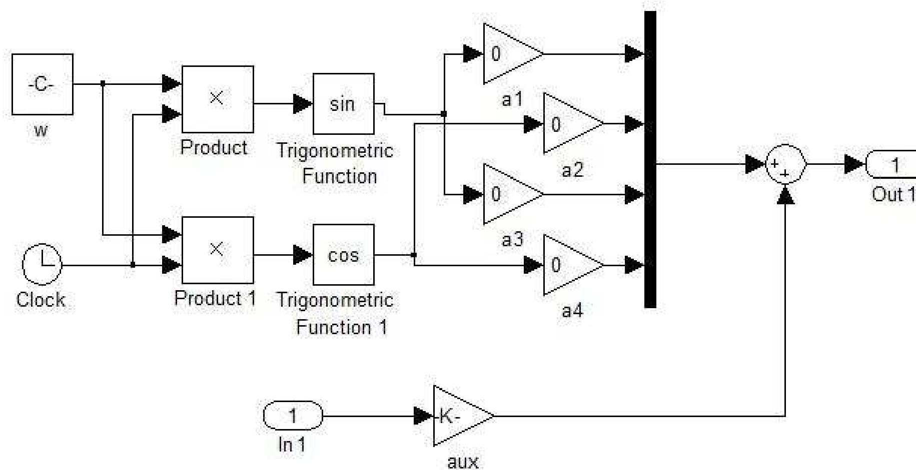

Figure 2-3 Simulink Block Diagram

A dSPACE provided software module call Real-Time Workshop (RTW) converts a block diagram to a C-language program. This program is then compiled by RTW and executed by the DS1005 processor to control an actual system.

Controlling a system is simplified by another dSPACE software module, ControlDesk. ControlDesk provides for the creation of instrument panels from which control code can be executed and modified in real time. A sample instrument panel is pictured in Figure 2-4.

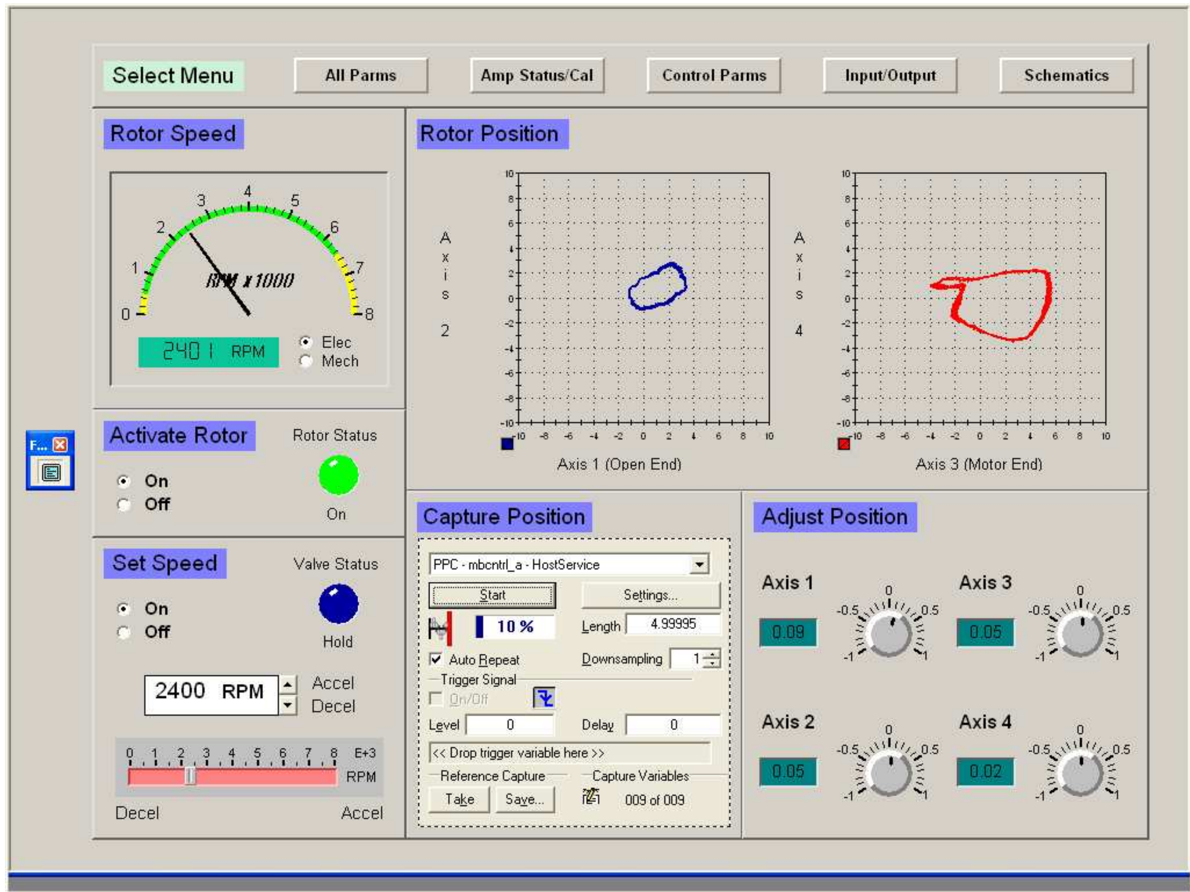

Figure 2-4 ControlDesk Instrument Panel

Variables in a Simulink block diagram can be easily connected to ControlDesk instruments (e.g. the knobs shown in Figure 2-4) for display or manipulation. In addition, the functionality of an instrument panel can be extended by the user with other software features of ControlDesk.

2.3 Bearing Controller

Modeling and control of the FACETS magnetic bearing were completed by Matras [5]. He developed a state-space model of the AFRL unit and used a proportional - integral -derivative or PID controller to actively manage the bearings. This model was implemented in Simulink. Matras then added adaptive disturbance rejection (ADR) with output redefinition to his model. ADR is used to reject or compensate for forces due to rotor imbalance and base

motion disturbances. These forces would likely be seen in any magnetic bearing mounted on a space structure such as a satellite. If these forces are ignored or uncompensated for in the bearing controller, the rotor could contact the electromagnets and damage or destroy them.

The bearing controller and its development are well documented by Matras. Thus, only a brief description of the controller will be presented here. Pictured in Figure 2-5 is the top-level Simulink block diagram for the magnetic-bearing controller.

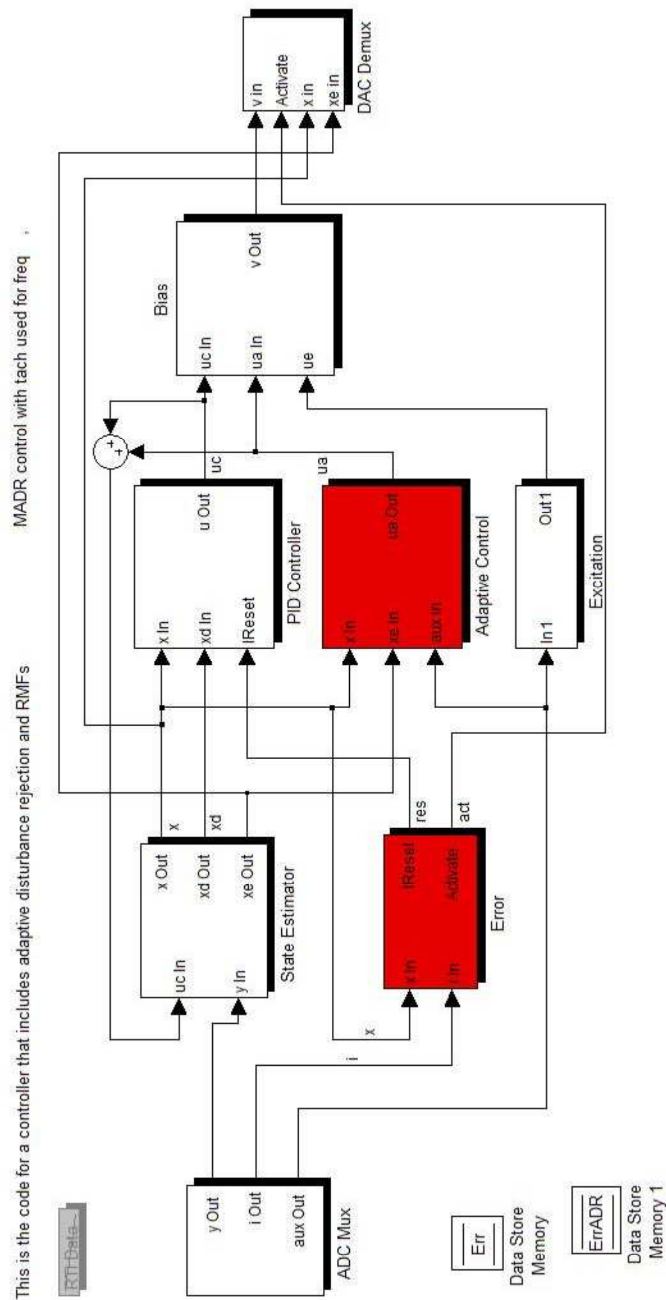

Figure 2-5 Bearing Controller

Each block in the diagram represents a component or a subsystem of the complete model. The subsystem representation provides a condensed view of each block. Refer to [5] for an expanded view of the subsystems.

ADC Mux

Subsystem *ADC Mux* represents in Simulink the A/D converter and connector panel discussed in Chapter 2.2. This subsystem collects inputs into the system and routes them to other subsystems for processing. Recall from the earlier discussion of the hardware that there are 8 inputs (signals or analog voltages) from the Proximitor probes, 8 from the amplifiers powering the bearings and 4 signals from the tachometers and flow-control valve.

State Estimator

Inputs to *State Estimator* are rotor position from *ADC Mux* and control current to the bearings. This subsystem calculates actual position and estimates the position and velocity states for input to other subsystems.

PID Controller

The *PID Controller* as its name implies implements proportional-integral-derivative control action using the measured position of the rotor, integrated position and estimated velocity of the rotor as inputs. The sole output is a control signal. This subsystem was constructed so that different controller gains could be applied to each bearing.

Adaptive Control

As described by Matras [5], adaptive control is only applied to the axes of bearing number 1. The *Adaptive Control* subsystem includes its own subsystem to generate control signals based on user-specified rejection frequencies. The input to *Adaptive Control* is measured rotor position, estimated states (rotor position and speed) and any auxiliary inputs applied through the *ADC Mux* block. The outputs of this block are the adaptive control signals for *Bearing 1*.

Bias

Bias generates the control voltages used for suspending the rotor. Inputs from both the PID and adaptive controllers are added to user-specified bias voltages to generate composite voltages. The rotor can be biased towards one magnet or another by specifying a separate bias

voltage for each magnet. Composite voltages are then output to the bearings from the *DAC Demux* subsystem. In addition, an extra signal can also be applied to the composite voltages through the *ADC Mux* and *Excitation* subsystems.

Excitation

The description of the *Excitation* subsystem is taken directly from Matras [5]. This block generates sinusoidal excitation signals that can be controlled for each channel (axis) separately (Channels will be described later in this thesis.). It also feeds the signal for the auxiliary input through to the *Bias* subsystem. The auxiliary input is used for the signal generated by the DSA (dynamic signal analyzer) when determining frequency response data.

Error

Detecting current and position errors and taking any precautionary actions are the functions of the *Error* block. Current and position errors occur when either exceeds a certain value. If an error occurs, the user is notified, and if configured to do so, the bearings will be shut off. However, turning off the bearings could seriously damage them if the rotor is spinning. Thus, this block is almost always configured to only notify the user when an error is detected. Error conditions are numerical coded and stored in the *Err* and *ErrADR* data storage areas shown in Figure 2-5. Inputs to *Error* are the measured position of the rotor and the currents flowing to the bearings. Outputs are an integral gain reset and possible bearing deactivation.

DAC Demux

Subsystem *DAC Demux* is the Simulink interface to the D/A converter and connector panel covered in Chapter 2.2. Inputs to this subsystem are bearing control voltages output from *Bias*, a rotor status signal from *Error* and the measured and estimated positions of the rotor with respect to the magnets. Either the actual or estimated positions can be selected by the user for use in measuring frequency response data. There are 8 control voltages output from *DAC Demux*. These signals are then routed to the amplifiers shown in Figure 2-1. The

amplifiers convert the voltages to currents to generate the magnetic attractive forces in the bearings. Voltages needed to operate the flow-control valve are also output from *DAC Demux*.

2.4 Instrument Panel

Operating the FACETS bearing requires modifying variables within each subsystem of the block diagram given in Figure 2-5. For example, a variable in the *Error* subsystem must be changed to power on and off the bearings. Refer to [5] for an expanded view of *Error* and of all subsystems shown in Figure 2-5. Changing the values of variables is accomplished through instrument panels or layouts created with ControlDesk. The instrument panel originally constructed to operate the FACETS bearing is pictured in Figure 2-6.

Figure 2-6 Original Instrument Panel

The pictured layout is very complete, but it is difficult to use given the large amount of information available in one view. Thus, as part of the current development effort, the original layout was logically reorganized into five panels or dialogs. Each dialog is accessed and displayed individually from a top-level instrument panel. Since the instrument panel in Figure 2-6 has been redone, the operation of the FACETS bearing from it will not be explained. Instead, the new dialogs will be presented later in this thesis, and at that time, it will be explained how to use these dialogs to operate the bearing.

2.5 Flywheel

Arrow 7 in Figure 2-1 points to the flywheel which is a part of the current magnetic-bearing system. This flywheel was designed and constructed by Barber, and the details of its design can be found in [21]. Barber used this flywheel and the FACETS bearing developed by Matras to investigate a health monitoring system for flywheels supported by magnetic bearings and used on satellites for energy storage and attitude control. Flywheels of this type could operate at speeds of up to 100,000 rpm [21]. If a crack were to develop at such a high speed, it could quickly lead to a complete failure of the flywheel.

Health monitoring provides for the early detection of developing cracks and other structural flaws such as the delamination of composite materials used in the construction of energy-storing flywheels. Cracks produce vibrations in rotating machinery. Matras used the FACETS bearing to develop and refine adaptive disturbance rejection or ADR. This technique automatically adjusts the bearing controller to compensate for vibration due to rotor imbalances and base-motion disturbances. With ADR, any change in the balance of the rotor can be identified by observing the automatic adjustments of the adaptive control gains. Barber used ADR as the basis of his health monitoring system. Any change in balance of the flywheel due to a crack for example was identified by monitoring the adaptive control gains. How cracks and crack growth were simulated in a flywheel is explained in [21].

Chapter 3 - Hardware Upgrades

The FACETS bearing and flywheel will be used to further develop ADR and health monitoring of flywheels. To assist future development, it was decided to implement automatic speed control of the rotor. As delivered from the Air Force Research Laboratory, rotor speed was regulated by manually adjusting a shutoff valve positioned in the air supply line to the turbine. Compared to manual speed regulation, automatic control offers the following advantages by:

- providing an easy way to set or change speeds,
- maintaining a set speed under varying air flow conditions,
- enabling tests requiring varying speeds,
- synchronizing rotor speed and ADR frequency,
- providing for the containment of damaged flywheels, and by
- compensating for other disturbances.

Before a speed controller could be developed, a number of hardware upgrades were required of the system. These upgrades are detailed in the following sections of Chapter 3.

3.1 Air Delivery

Compressed air is delivered to the laboratory room housing the FACETS bearing from a common source serving numerous labs. When the air turbine was first connected to the air supply, there was only enough compressed air to spin the rotor to 1500 rpm. This was the speed as reported by the optical encoder or mechanical tachometer affixed to the end of the rotor as seen in Figure 2-1, arrow 5. Rotor speed was verified with a stroboscope. From prior tests conducted at AFRL, the bearing had been operated to 13,000 rpm [5]. Health monitoring studies also required speeds in excess of 1500 rpm [21].

Two options were available for increasing the supply of air to the turbine - (1) improve the piping in the lab or (2) buy an air compressor. Opting for the former and cheaper alternative, the assistance of the Auburn University mechanical shop was enlisted. This department is

responsible for all building maintenance, and they simplified and improved the piping network in the lab. All pipes smaller than one-half inch in diameter were replaced with one-half inch pipes. The flow path from the pipe entering the lab to the turbine was straightened by removing a number of bends and elbows. Small diameter shutoff valves were also replaced with one-half inch models. With these piping changes complete (and inexpensive), there was enough air to turn the rotor to over 6000 rpm, an increase in speed of more than 300%. Pressure in the supply pipe ranged from about 110 psi with the shutoff valve closed to around 90 psi with the valve fully opened and the rotor spinning.

The volume of air consumed by the turbine was needed to develop the speed control system. To determine air consumption, a volumetric flow meter was installed in the air supply line to the turbine. This meter is pictured in Figure 3-1.

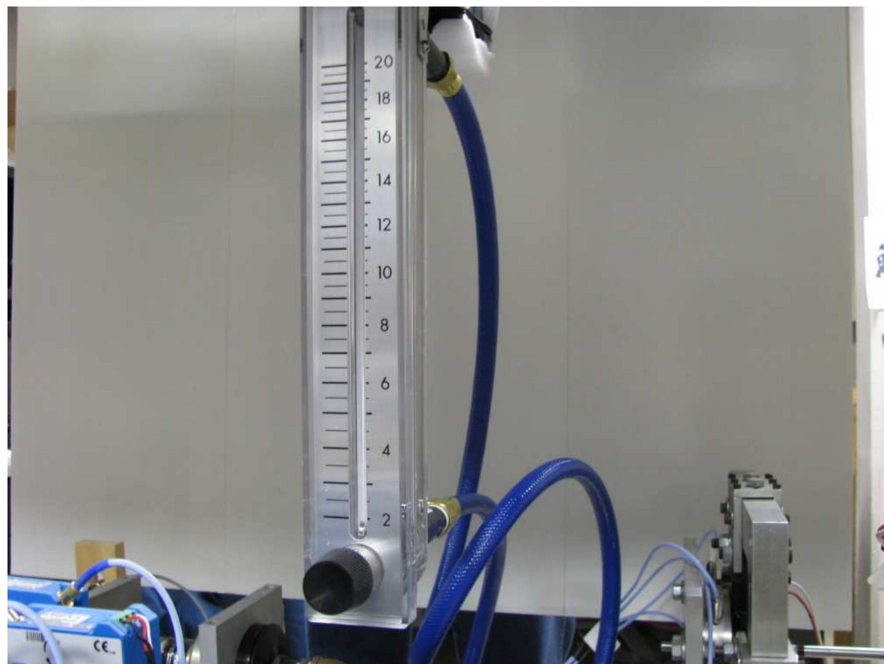

Figure 3-1 Rotameter

There are many different types of flow meters, and the one shown in Figure 3-1 is a variable area meter or rotameter. An explanation of its operation is given by Holman [23], and he classifies this type of meter as a drag effect device. The rotameter provides flow

measurements by sight and in standard cubic feet per minute (scfm) of air. This device was provided gratis by Bruce Smith of Control & Power, Inc, an industrial supplier of controls, instrumentation and automation based in Birmingham, Alabama. With the rotameter installed, it was found that about 10 scfm of air at 100 psi are required to start the flywheel moving. Nearly 20 scfm of air at 90 psi are flowing to the air turbine with the shutoff valve fully open. The flow meter slightly reduced the maximum speed of the rotor, but the data provided by the meter were essential to the development of a speed controller.

3.2 Speed Measurement

The optical encoder or mechanical tachometer mentioned previously was never well calibrated. Thus, it was decided to purchase an electronic tachometer so that rotor speed would be accurately reported. After reviewing the available offerings, a Shimpo model DT-5TX digital tachometer was purchased, and this instrument is pictured in Figure 3-2.

Figure 3-2 Digital Tachometer

The Shimpo tachometer was chosen since it met our requirements regarding sensor input and signal output. It was also competitively priced and available from numerous suppliers. The DT-5TX accepts input from a number of different speed sensors, including magnetic pick-ups, retro-reflectors and rotary-pulse generators. A retro-reflector was chosen since it is easy to install and is effective over a distance of from 1 inch to 3 feet from the measurement source. This wide range offers great flexibility in locating the sensor. However, range is dependent on the diameter of the object whose rpm is being measured.

Figure 2-1 provided a picture of the retro-reflective sensor in place with the FACETS bearing. Another view of the sensor is given in Figure 3-3.

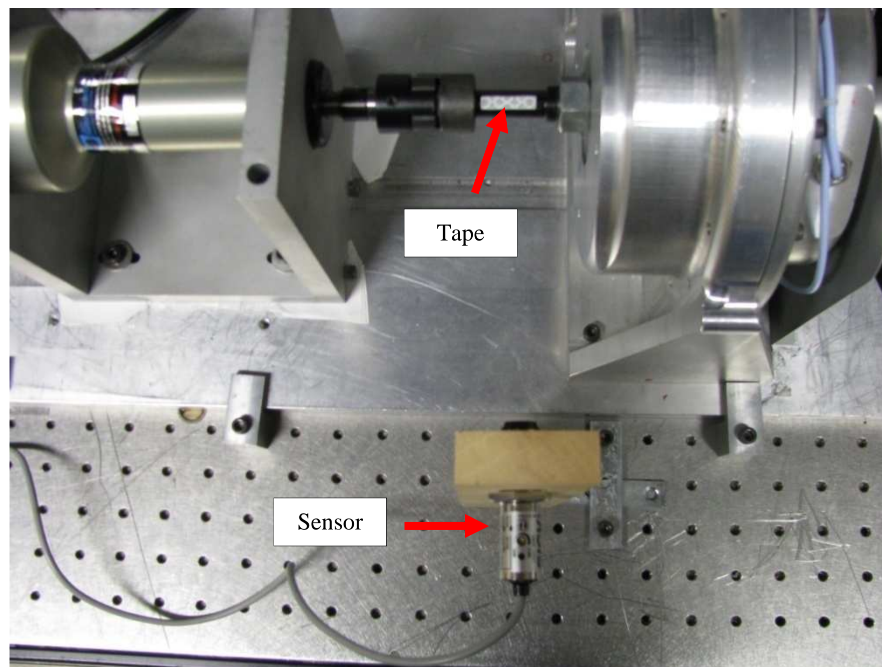

Figure 3-3 Retro-Reflective Sensor

The retro-reflector detects a revolution of the rotor each time its light is returned or reflected back to it by a small piece of photo-reflective tape (visible in Figure 3-3) attached to the rotor. The sensor then only needs to be aimed at the tape to detect rpm. Aim or alignment is assisted by an LED mounted on top of the sensor and by the homemade mounting bracket. This cost-

effective hardware aids alignment by providing for sensor adjustments in three directions - vertically, parallel with and perpendicular to the rotor.

In addition to its digital display, the tachometer is equipped with an analog voltage output module that generates a 0 to 10 volt DC signal linearly proportional to measured rpm. This signal is then input directly into dSPACE through the A/D converter. Furthermore, the tachometer must be configured prior to use. Configuration is performed with buttons located on the front panel of the tach. These buttons can be seen in Figure 3-2. There are a number of variables requiring configuration, and for complete setup information, refer to the Shimpo DT-5TX instructional manual [24].

3.3 Flow Control

An electronic valve controllable with dSPACE was required to implement automatic speed regulation of the rotor. Electronically-controlled valves are available in at least two types - pressure regulators and flow-control valves. Pressure regulators adjust air flow by venting excess air to the atmosphere. This is an undesirable feature since it is noisy and disruptive, especially when operating in a laboratory or small room. Venting probably isn’t noticeable if a pressure regulator is used within a large, open building such as a factory. Because they vent, pressure regulators were deemed unsuitable for speed control.

Flow-control valves regulate air flow by adjusting the position of an obstruction or stopper in the flow path through the valve. The stopper’s position is controlled either by a solenoid or a stepper motor. A solenoid-operated valve was selected initially because it had a high flow capacity and provided valve-position feedback. Position feedback correlates stopper position with volumetric flow rate of air, and this relationship was required to develop the speed controller. The solenoid valve tested required just a single voltage signal to position the stopper. Stopper position was then supposed to be linearly proportional to the applied voltage - 0 volts to close and 10 volts to fully open. In tests, the solenoid-operated valve worked very poorly.

The action of the valve was non-linear and not repeatable. The valve acted like an on-off device - it was either closed or fully open. In fact, it required only about 1 volt to completely open the valve. In addition, valve control was not symmetric. It required more voltage to close the valve a certain amount than to open it the same amount. Furthermore, the solenoid-operated valve would become unstable, and the stopper would oscillate if the voltage was increased or reduced too rapidly. Automatic speed control would have been difficult to implement with this particular valve.

A flow-control valve operated by a stepper motor was then tested. This type of valve uses a stepper motor rather than a solenoid to control the position of the stopper. The valve selected is built by Aalborg Instruments and Controls of Orangeburg, NY. Unfortunately, the Aalborg valve did not provide position feedback, but this feature was soon added. In practice, this stepper-motor valve works wonderfully and is pictured in Figure 3-4.

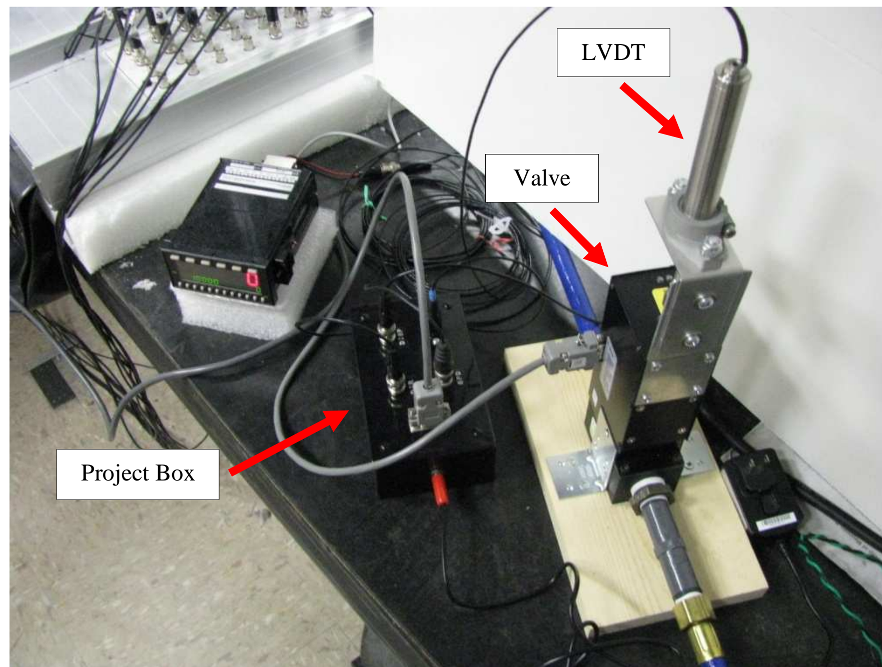

Figure 3-4 Aalborg Valve and Accessories

Note the project box next to the valve and the linear voltage differential transformer (LVDT) attached to the top of the valve. The project box was built by the Research Electronics

Support Facility at Auburn University, and this capable department also added the LVDT. Electrical connections between the stepping motor valve (SMV) and the A/D converter are provided by the project box while the LVDT adds the position feedback missing from the as-purchased Aalborg valve.

The Aalborg SMV requires two voltage signals for operation, one for direction and another for speed. This valve opens if the direction signal is between 7.6 and 12 volts. A direction signal of less than 2.3 volts will close the valve; stopper position is held at 0 volts. In practice, a voltage of 8.5 is used to open the valve, and a 1.5 volt signal is used to close the valve. Refer to the Aalborg instruction manual for detailed operating instructions [25].

The speed at which the SMV opens and closes (i.e. raises and lowers the stopper) is controlled by a voltage signal of from 0+ to 2.5 volts. Low voltages provide slow stepping rates while higher voltages yield faster stepping rates. During testing, it was found that the valve would respond to a voltage as small as 0.05 volt. Air flow can be precisely controlled at this low voltage. For practical purposes, the fastest stepping rate should be limited to 0.7 volt since rates greater than this result in temporary instability in the magnetic bearings.

Chapter 4 - System Model

With the required hardware in place, a speed controller could now be developed. Development began with the creation of a Simulink model representing the actual system - valve, air turbine, rotor, flywheel and magnetic bearings. This model is shown in Figure 4-1.

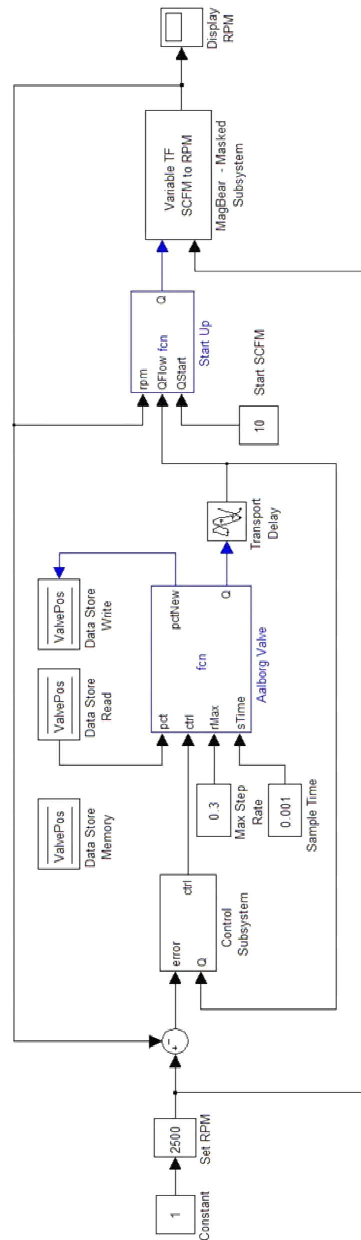

Figure 4-1 Simulink Model of Magnetic-Bearing System

The system model contains three major components which are labeled in the block diagram - *Control Subsystem*, *Aalborg Valve* and *MagBear - Masked Subsystem*. Chapter 4 will detail the creation of the system model and its use in developing the actual speed controller.

4.1 Transfer Function

An equation for the angular velocity of the rotor is needed to build the model of the magnetic bearing (magbear) subsystem. This subsystem will include the air turbine, rotor, flywheel and magnetic bearings. Rotor velocity can be found from the equation of motion (EOM) for the subsystem, and this equation was developed from the free-body diagram of the rotor and flywheel shown in Figure 4-2. Forces due to gravity and the bearing supports are not shown since they do not contribute to the desired EOM.

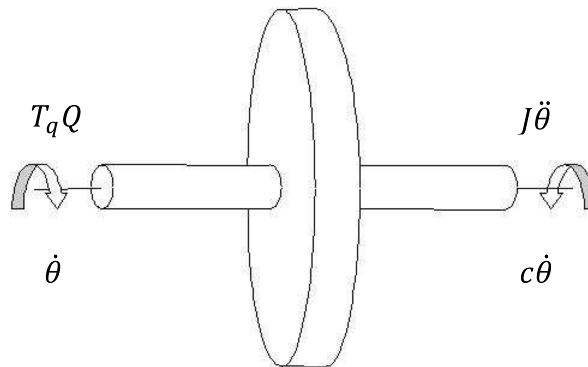

Figure 4-2 Free-Body Diagram of Rotor and Flywheel

The torque exerted on the rotor by the air turbine is a function of the volumetric air flow rate Q. This function was assumed to equal an unknown torque constant T_q multiplied by Q. The units of torque are *N-m*, flow rate m^3/sec and the torque constant $N\text{-}sec/m^2$. Moment of inertia J is that of the rotor and flywheel about the axis of rotation. Barber found this moment of inertia to be $0.004903\ kg\text{-}m^2$ [21] At this point, the damping coefficient c of the system is unknown but when found, will be in units of *N-m-sec*. Finally, the angular velocity or speed of the rotor and flywheel is given by $\dot{\theta}$ in *rad/sec*.

The differential equation of motion follows and was found by summing moments about the rotational axis:

$$J\ddot{\theta} + c\dot{\theta} = T_q Q \tag{4.1}$$

Equation 4.1 can also be expressed in terms of ω where $\omega = \dot{\theta}$ and $\dot{\omega} = \ddot{\theta}$:

$$J\dot{\omega} + c\omega = T_q Q \tag{4.2}$$

Since the magnetic-bearing subsystem is described by a single equation of motion (Equation 4.2) with a single state variable ω, a transfer function will be used to model the subsystem.

The transfer function can be found from 4.2 where flow rate Q is the input to the system and speed ω is the output. Assuming zero initial conditions, the Laplace transform of Equation 4.2 is:

$$Js\omega(s) + c\omega(s) = T_q Q(s) \tag{4.3}$$

Rearranging 4.3 yields the transfer function relating input to output:

$$\omega(s) = \frac{T_q Q(s)}{Js + c} \tag{4.4}$$

The rotational speed of the rotor ω is measured by the tachometer; volumetric flow rate Q is measured by the rotameter, and the moment of inertia J has been previously determined. The torque constant T_q and the damping coefficient c are the unknowns. At this point, it is assumed that the magbear system is linear and time-invariant (LTI) since a Laplace transform was used to derive Equation 4.4. This assumption implies that T_q and c are constant over the speed range of the rotor.

Equation 4.4 is of the form,

$$\frac{b_0}{s + a_0} \tag{4.5}$$

where $b_0 = T_q/J$ and $a_0 = c/J$. Applying a step input of Q to the system yields:

$$\omega(s) = \left(\frac{Q}{s}\right)\left(\frac{b_0}{s + a_0}\right) \tag{4.6}$$

The solution to this equation by partial fraction expansion is:

$$\omega(t) = \frac{Qb_0}{a_0}[1 - e^{-a_0 t}] \tag{4.7}$$

Substituting for a_0 and b_0 yields the velocity of the rotor with respect to time:

$$\omega(t) = \frac{QT_q}{c}\left[1 - e^{-ct/J}\right] \tag{4.8}$$

Note from Equation 4.8 when time $t_0 = 0$ seconds, the speed $\omega(0)$ is zero. This is an expected result since at time zero, the step input of air has not yet been applied. As time increases after the application of Q, velocity reaches a steady-state value at time steady-state t_{ss} of:

$$\omega(t_{ss}) = \frac{QT_q}{c} \tag{4.9}$$

A units check shows the speed of the rotor to be in *rad/sec*. The speed at any time t in terms of the steady-state speed ω_{ss} for a step input of Q is:

$$\omega(t) = \omega_{ss}\left[1 - e^{-ct/J}\right] \tag{4.10}$$

Equations 4.9 and 4.10 were used to experimentally determine the torque constant T_q and damping coefficient c of the actual system comprised of the air turbine, rotor, flywheel and magnetic bearings. These coefficients were found by applying a step input of air Q to the system and allowing the flywheel to reach a steady-state speed. The response of the system from time t_0 to time t_{ss} was captured using the data acquisition capabilities of ControDesk. Equation 4.10 was then fit to the response data which resulted in the coefficients for a_0. With a_0 known, c could now be found since $a_0 = c/J$ and J had already been determined from prior research. Equation 4.9 was then used to determine T_q since ω_{ss} , Q and c were all now known.

4.2 Step Response

Step inputs in standard cubic feet per minute, *scfm*, and cubic meters per second, m^3/sec, shown in parentheses, of 10 (0.004720), 12 (0.005663), 14 (0.006607), 16 (0.007551), 18 (0.008495) and 20 (0.009439) were used to generate speed vs. time response curves for the system. All steps were applied with the control valve fully open. The valve was not used to throttle the air flow to the desired value. With this approach, the valve's contribution to the system's response was constant for all inputs. The desired step was achieved with the flow adjustment knob on the rotameter. This adjuster can be seen in Figure 3-1. Once the flow rate was set, data collection was activated and so were the bearings. Each test ran until the rotor reached its steady-state speed. At this point, data collection was turned off, and curves were fit to the data using MATLAB. However, data collected by ControlDesk has to be converted to a *mat* file format before MATLAB can process it. This conversion is accomplished from ControlDesk using a built-in utility.

Once the data were converted, MATLAB's curve fitting tool, *cftool*, was used to fit curves to the data. The curves fit with *cftool* were one-term exponentials of the form given in Equation 4.10. These curves are shown in red in Figure 4-3, and the actual response data are drawn in black.

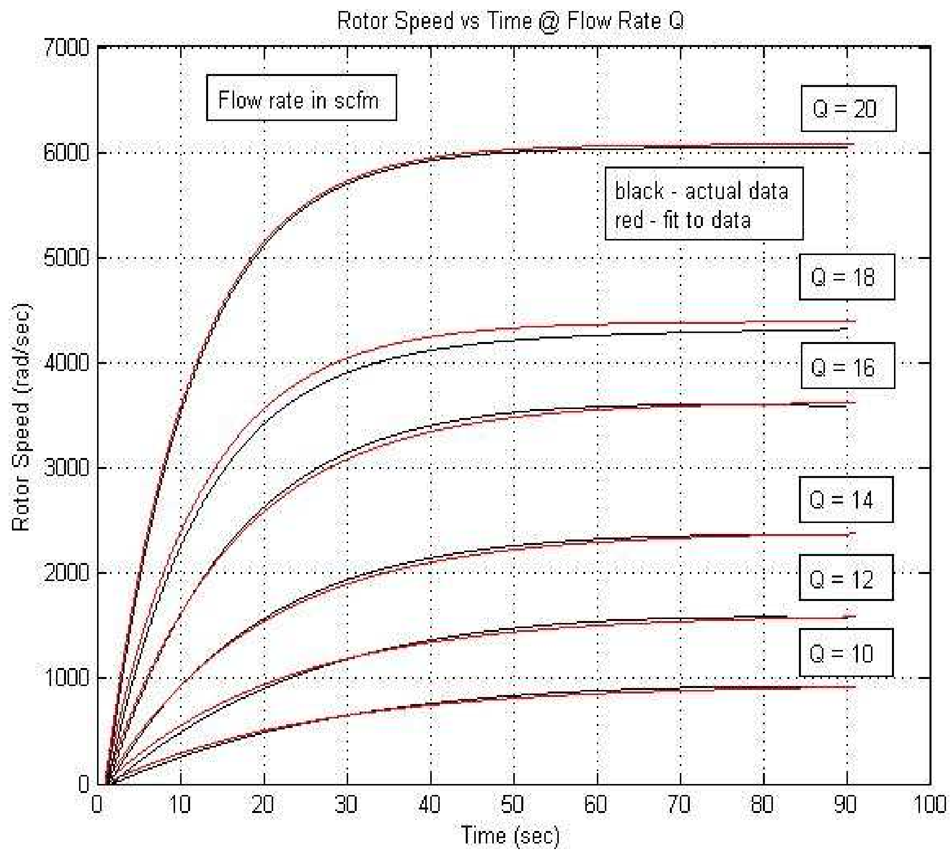

Figure 4-3 Rotor Speed vs. Time

Except for a slight divergence at 18 scfm, Equation 4.10 fits the data very well. The curves do not begin at time zero since there was a slight time lag between the start of data collection and the actuation of the bearings.

Table 4-1 summarizes the data presented in Figure 4-3 and lists the values of c and T_q found from fitting Equation 4.10 to the step response data. Note that rotor speed can be converted to rpm by multiplying the values in *rad/sec* by $60/2\pi$.

Q *scfm*	Q 10^{-3} m^3/sec	ω_{ss} *rpm*	ω_{ss} *rad/sec*	a_0 *1/sec*	c 10^{-4} *N-m-sec*	T_q $N\text{-}sec/m^2$
10	4.720	945	99	0.040	1.9612	4.11
12	5.663	1609	168	0.046	2.2554	6.71
14	6.607	2393	251	0.054	2.6476	10.04
16	7.551	3629	380	0.065	3.1870	16.04
18	8.495	4391	460	0.087	4.2656	23.09
20	9.439	6076	636	0.098	4.8049	32.39

Table 4-1 Step Response Data

It is seen from Table 4-1 that the damping coefficient c and torque constant T_q vary with the air flow rate and speed of the rotor. However, these variables are constant for a given speed. Thus, while the magbear system cannot be described by a single LTI transfer function, it can be considered as a series of these transfer functions, each one specific to a set speed.

A transfer function was created for each air flow rate shown in Table 4-1. A simple Simulink model was then constructed to test how accurately these transfer functions predicted the steady-state speed of the rotor. This model is shown in Figure 4-4 for a step input of 0.009439 m^3/sec or 20 scfm of air.

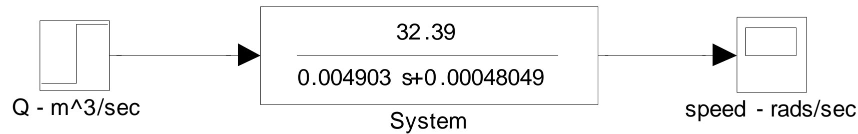

Figure 4-4 Model of Individual Transfer Function

The response or output of this model is given in Figure 4-5.

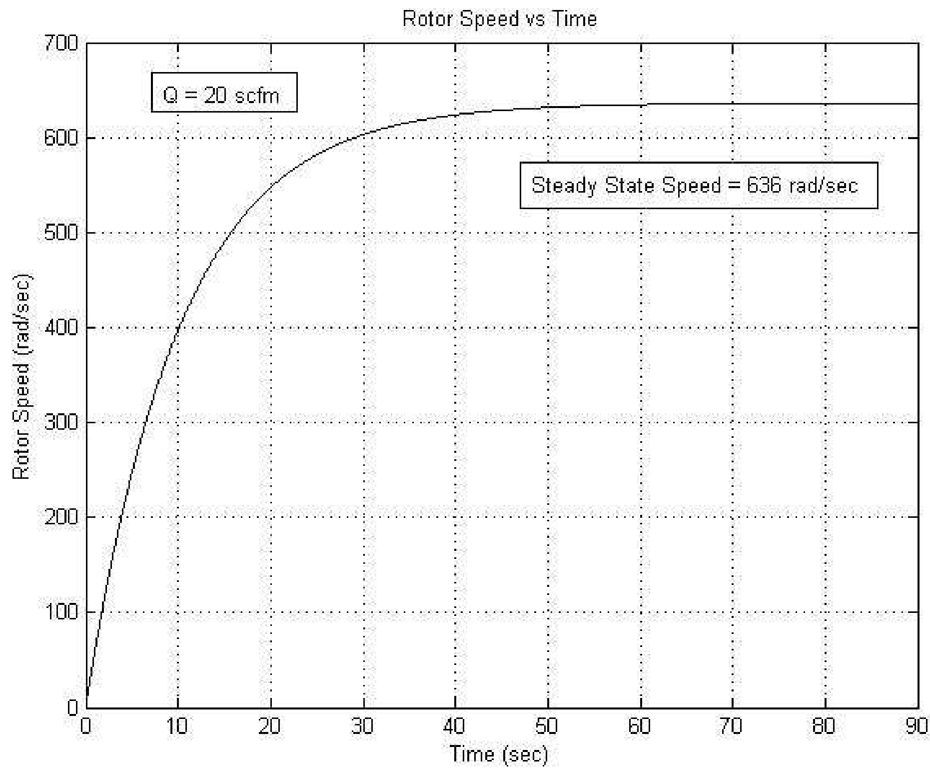

Figure 4-5 Step Response of Transfer Function

The steady-state speed from the transfer function, 636 *rad/sec*, is identical to that determined experimentally. In fact, the transfer functions for each step input predict a steady-state speed nearly identical to that found by experiment.

A single LTI transfer function does not characterize the magnetic-bearing system since it is not linear as evidenced in Figure 4-3 and Table 4-1. However, a single transfer function can be used to model the magbear system provided coefficients T_q and c are varied with input.

4.3 Variable Coefficients

A transfer function with variable coefficients can be created in Simulink. One way to do so requires rearranging the transfer function as a sum of integrators and then placing this function in a masked subsystem block. This approach was used here.

The transfer function representing the magnetic-bearing system was derived earlier and given in Equation 4.4. This equation is repeated below:

$$\omega(s) = \frac{T_q Q(s)}{Js + c} \tag{4.4}$$

With a little algebra, Equation 4.4 can be rearranged as the difference of two integrators (*1/s*):

$$\omega(s) = \frac{1}{J}\left[\frac{QT_q}{s} - \frac{\omega(s)c}{s}\right] \tag{4.11}$$

In the form of Equation 4.11, the coefficients T_q and c of the integrators can be varied as a function of rotor speed. How T_q and c vary was found by fitting curves to the data given in Table 4.1. These curves were again fit using MATLAB's *cftool*. It was found that the torque constant T_q changes as a function of speed according to the following quadratic polynomial:

$$T_q = 4.557 \times 10^{-7} * (\omega^2) + 2.313 \times 10^{-3} * (\omega) + 2.218 \tag{4.12}$$

In Equations 4.12 and 4.13, ω is given in *rpm* instead of *rad/sec* since ultimately, the input to the simulated and actual speed controllers will be desired rotor speed in user-friendly *rpm*. A quadratic polynomial also represented the variance of the damping coefficient c with speed:

$$c = 1.117 \times 10^{-12} * (\omega^2) + 5.007 \times 10^{-8} * (\omega) + 1.459 \times 10^{-4} \tag{4.13}$$

Graphs of T_q and c are provided in Figures 4-6 and 4-7.

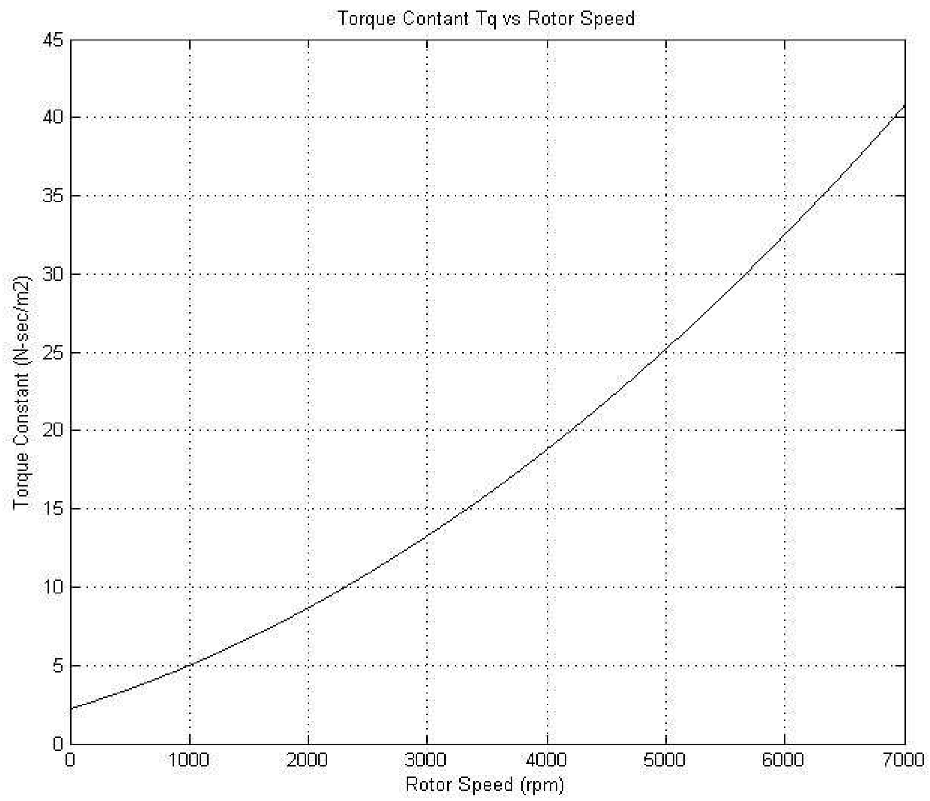

Figure 4-6 Torque Constant vs. Rotor Speed

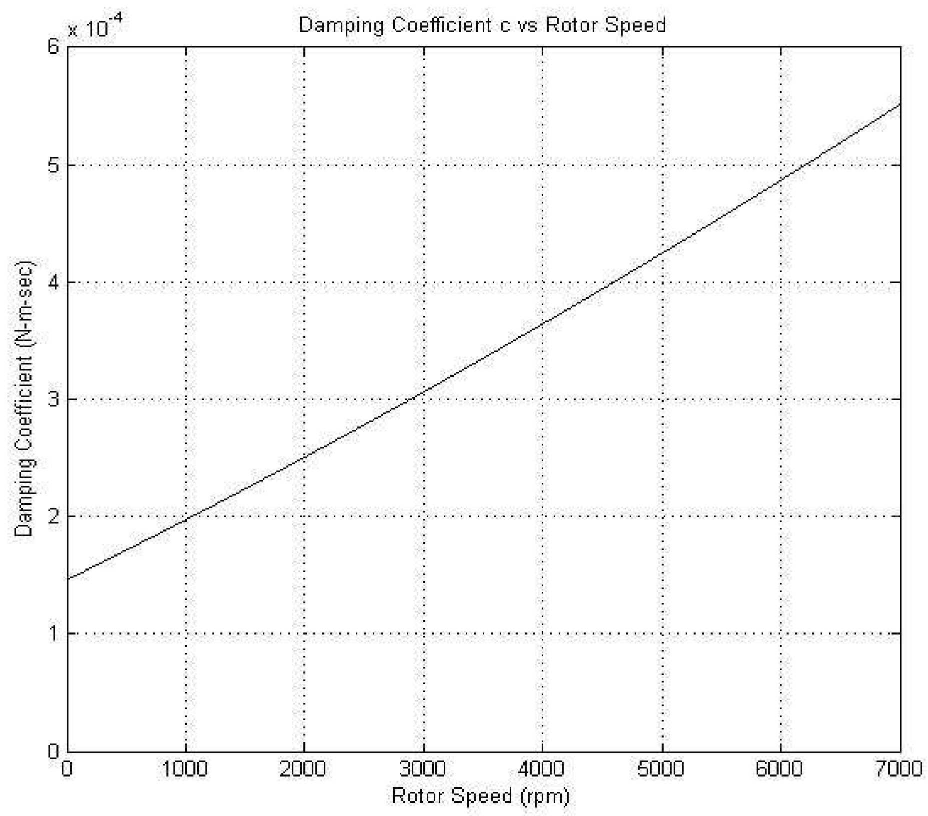

Figure 4-7 Damping Coefficient vs. Rotor Speed

4.4 MagBear - Masked Subsystem

A Simulink model was then constructed from the transfer function, Equation 4.11, and Equations 4.12 and 4.13. This model is pictured in Figure 4-8, and it simulates the dynamics of the air turbine, rotor, flywheel and magnetic bearings.

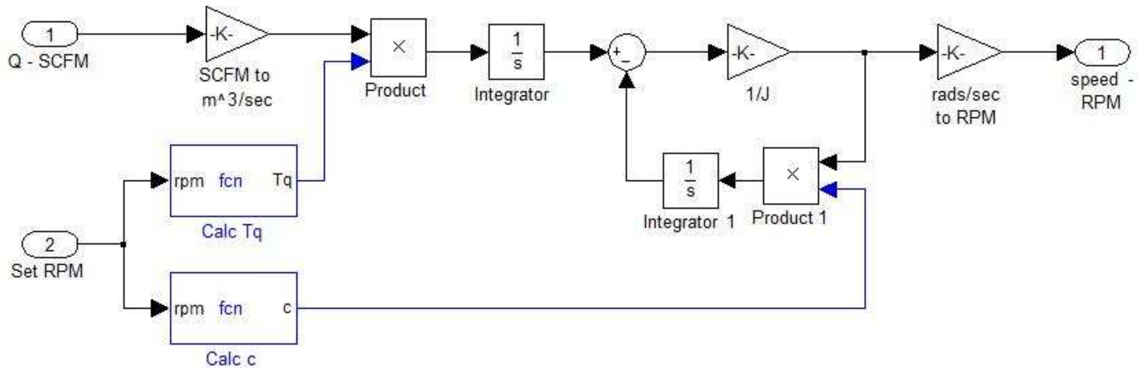

Figure 4-8 Simulink Model with Variable Coefficients

The embedded MATLAB functions *Calc Tq* and *Calc c* contain Equations 4.12 and 4.13. *Set RPM* is the steady-state rotor speed desired by the user, and *speed - RPM* is the instantaneous rotor speed calculated by the model. Coefficients T_q and c are not varied continuously with speed but are instead set according to the specified steady-state speed, *Set RPM*. These coefficients could be constantly varied by using *speed - RPM* in place of *Set RPM* as the input to functions *Calc* T_q and *Calc c*. Continuously-varying coefficients were tried, but when used, the transient speeds of the rotor were overstated by the model. The transfer function model shown in Figure 4-8 is contained in the block labeled *MagBear - Masked Subsystem* of Figure 4-1. This subsystem accurately modeled the air turbine, rotor, flywheel and bearings.

4.5 Valve Position

The model of the magnetic-bearing system developed thus far is given in Figure 4-8, and it requires air flow as input. Ultimately, the input to the completed model and to the actual system will be desired rpm. To achieve a certain speed, the control valve must flow a certain

quantity of air. How much air it flows depends on the position of the valve or how far the valve is opened. As delivered, there was no way to determine air flow as a function of position. To obtain this data, an LVDT was added to the valve. This device was seen previously in Figure 3-4, and the LVDT tracks the position of the valve by following the motion of the valve's stopper.

With the LVDT in place, tests were conducted to produce a curve of volumetric air flow rate in scfm versus position as a percent of fully open. To conduct these tests, a control panel to be discussed later was constructed in ControlDesk to operate the valve. The valve was then opened until a certain flow rate was achieved as indicated by the rotameter. At this flow rate, the valve's position was read from the output of the LVDT. Output was calibrated to read as a percentage of full open. The valve's position was recorded for flow rates ranging from 6 to 20 scfm in increments of 1 scfm. Six scfm is about the minimum amount of air required to keep the rotor spinning and 20 scfm is the maximum air flow available from the building's compressed air source. The results of the flow tests are summarized in Table 4-2.

Q (scfm)	*% Open*
0	0
6	6
7	7
8	10
9	11
10	13
11	14
12	16
13	18
14	20

15	23
16	27
17	31
18	39
19	50
20	100

Table 4-2 Flow Rate and Valve Position

A graph of the data in Table 4-2 is shown in Figure 4-9, and this graph was constructed using MATLAB's *cftool*.

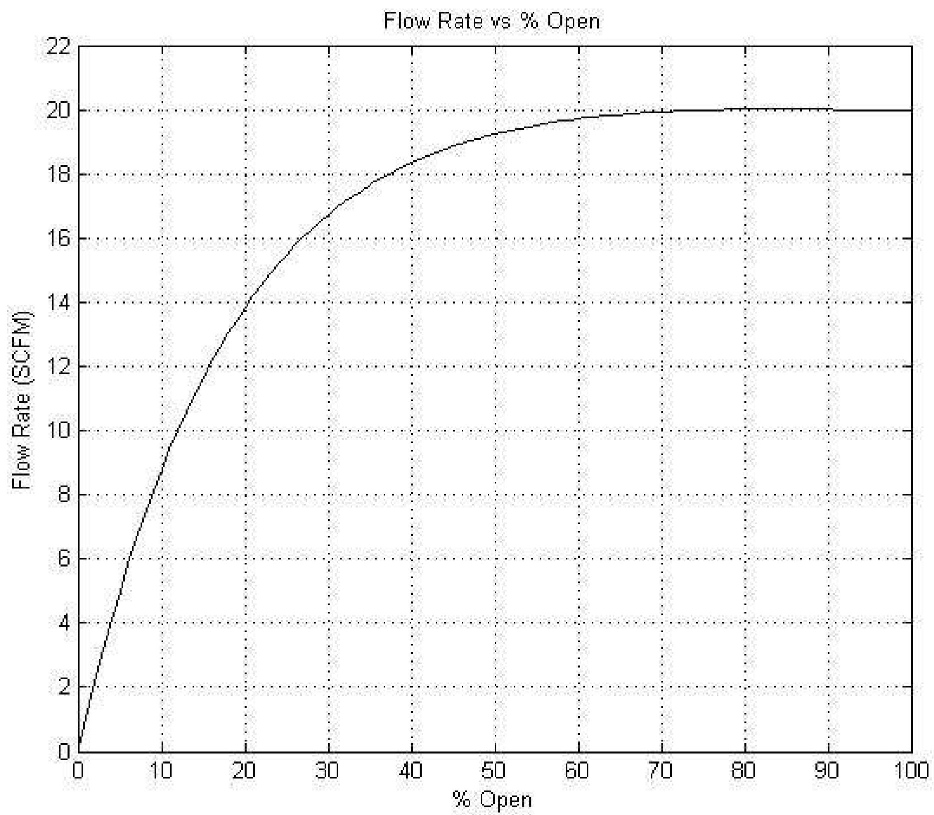

Figure 4-9 Flow Rate vs. Valve Position

The curve in Figure 4-9 is given by the following equation:

$$Q = 21.29 * e^{-0.0006*(\%\ open)} - 21.29 * e^{-0.05416*(\%\ open)} \quad (4.14)$$

Equation 4.14 defines flow rate in *scfm* as a function of valve position. Position as a function of stepping rate and time must also be known before an accurate model of the valve can be created.

4.6 Stepping Rates

The Aalborg valve requires two voltage signals for operation - one for direction and the other for stepping rate. Stepping rate determines how fast the valve opens and closes. Both direction and rate will be contained in the control signal (voltage) produced by the yet-to-be-designed speed controller. Direction is determined from the sign of the signal - open if positive and close if negative. The absolute value of the control signal will be the stepping rate in volts.

To complete the model of the valve, its position as a function of time at various stepping rates must be determined. If position is known at any time, the instantaneous flow rate can be calculated from Equation 4.14. This flow rate is then the input to the *MagBear - Masked Subsystem* shown in Figure 4-8.

The valve will accept a rate signal from 0+ to 2.5 volts [25]. Tests were thus conducted at stepping rates of from 0.1 to 1.5 volts in increments of 0.1 volt. Testing at rates greater than 0.7 volt probably wasn't necessary since any rate greater than 0.7 volt can produce temporary instability in the bearings. Each test began with the selection of a stepping rate. Data acquisition was then activated in ControlDesk followed by the actuation of the valve. The position of the valve as reported by the LVDT was captured as a function of time as the valve moved from completely closed to fully open. Each test then produced a position versus time curve or more accurately a line for each stepping rate. Regardless of rate, the relationship between position and time was always a straight line with a *y* intercept of 0. However, the slope of each line was unique to a stepping rate.

The slopes of the position versus time plots were found in the usual way - *cftool*, and are given in Table 4-3. Slope is the velocity of the valve stopper in units of *% open/sec*.

Stepping Rate (*volts*)	Slope (*% open/sec*)
0.1	0.2372
0.2	0.4195
0.3	0.6012
0.4	0.8827
0.5	1.1460
0.6	1.4720
0.7	1.8980
0.8	2.2460
0.9	2.6850
1.0	3.0890
1.1	3.5690
1.2	3.9910
1.3	4.4600
1.4	4.9350
1.5	5.4080

Table 4-3 Stepping Rate and Slope

At the risk of stating the obvious, if a control signal or stepping rate of 0.2 volt is applied for 0.5 second, then from Table 4-3 the valve opens $0.4195 \times 0.5 = 0.12024\%$. Interpolation is required if the control signal is not equal to one of the rates given in Table 4-3. Interpolation is performed linearly by the valve model according to Equation 4.15.

$$y_2 = \frac{(x_2 - x_1)(y_3 - y_1)}{(x_3 - x_1)} + y_1$$

(4.15)

In Equation 4.15, y_2 is the slope to be found, and x_2 is the stepping rate calculated by the controller. The values x_1 and x_3 are the stepping rates from Table 4-3 which bound x_2. Similarly, y_1 and y_3 are the slopes which bound y_2. Thus, if the controller sets the stepping rate at 0.24 volt, the valve position changes by:

$$y_2 = \frac{(0.24 - 0.2)(0.6012 - 0.4195)}{(0.3 - 0.2)} + 0.4195 = 0.4922 \ \ \% \, open/sec$$

If the valve steps at 0.24 volts for 0.5 sec, then it opens or closes (negative % open) $0.4922 \times 0.5 = 0.2461\%$.

Stepping rate, slope (stopper velocity) and time are all required to determine the incremental change in valve position for an interval of stepping time. The relationship between these four variables is displayed graphically in Figure 4-10. A MATLAB program, *plot3DStepRates.m*, was written to generate Figure 4-10. A listing of this program is given in Appendix A.

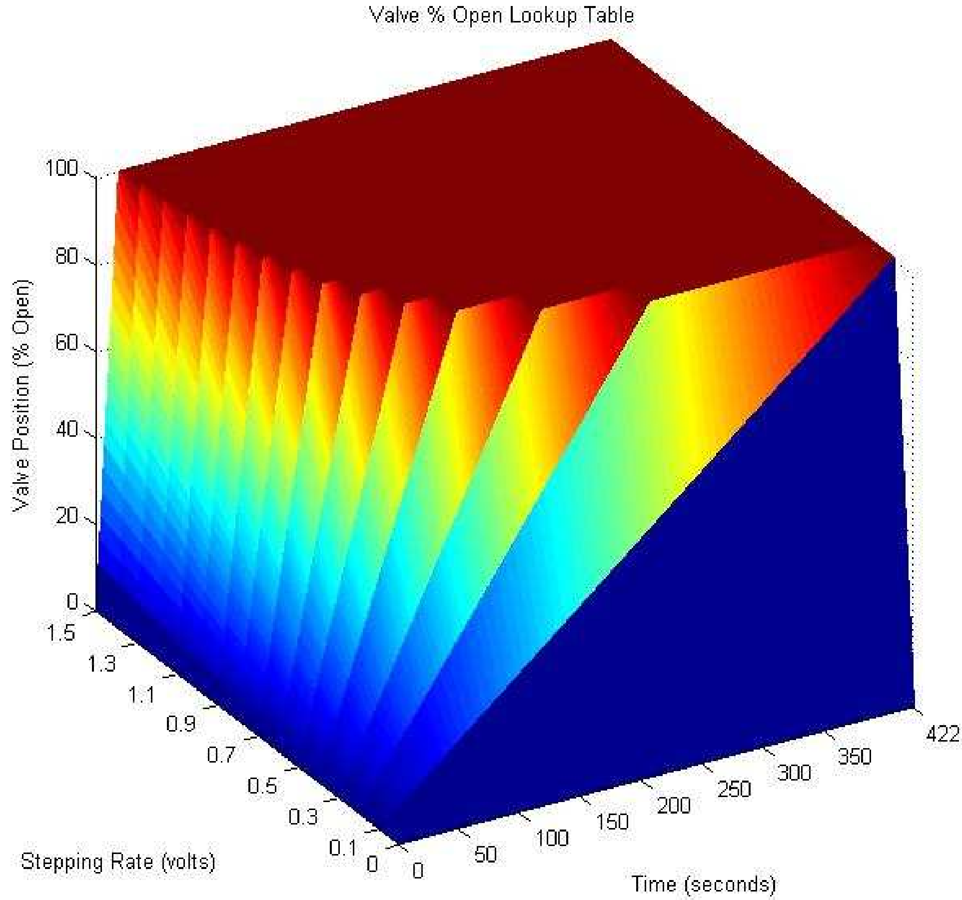

Figure 4-10 Position Lookup Table

Figure 4-10 is implemented as a lookup table by the valve model. Given a stepping rate and time interval, the model calculates the incremental change in valve position according to Figure 4-10. Note that a stepping rate is not used directly; it is converted to a slope using Table 4-3 and Equation 4.15. Incremental change in air flow is then determined by the model using Equation 4.14. The complete model of the Aalborg valve is discussed next.

4.7 Aalborg Valve

Lookup tables are used to control systems that are difficult to model with differential equations and analytical functions. For example, lookup tables are employed to control

automotive fuel injection systems [26]. In practice, these tables are experimentally generated from extensive testing of the actual system. This approach was used here to develop a description of the Aalborg valve since the dynamics of the valve were too complex to be described by a differential equation or transfer function. Therefore, the valve was modeled using a lookup table, and this table was shown graphically in Figure 4-10. The table was incorporated into the Simulink model of the magnetic-bearing system as an embedded MATLAB function or program. This program is contained in the block labeled *Aalborg Valve* in Figure 4-1. A program listing for the valve is contained in Appendix B, and the operation of this program and related portions of the Simulink model are discussed next.

As seen in Figure 4-1, there are four inputs to the *Aalborg Valve* function - *pct*, *ctrl, rMax* and *sTime*. The current position (% open) is given in *pct* and is read from persistent memory (*ValvePos*). The control signal *ctrl* is the stepping rate determined by the controller. This signal can be either positive or negative. Positive *ctrl* opens the valve and negative *ctrl* closes it. *rMax* is the maximum stepping rate permitted by the user, and *sTime* is the time that *ctrl* is applied. *sTime* is equal to the fundamental sample time or the integration time step used for the simulation. The MATLAB program uses the control signal *ctrl*, Table 4-3 and Equation 4.15 to determine the rate (% open/sec) at which the valve's position is changing. This rate is then multiplied by *sTime* to determine the incremental change in position during one time sample. The new position of the valve *pctNew* is found by taking the previous position *pct* and adding (valve opening) or subtracting (valve closing) the incremental position. Equation 4.14 and *pctNew* (i.e. % open) are then used to determine flow rate *Q*. Flow rate is routed to the *MagBear - Masked Subsystem* transfer function (Figure 4-8) and *pctNew* is stored for use in the next time step. This completes the description of the valve model, but there are a few other items to note from Figure 4-1.

A *Transport Delay* one percent greater than the *Sample Time* was required to eliminate algebraic loop errors from the Simulink model shown in Figure 4-1. Delaying a signal is a known solution to this type of error, and best accuracy is achieved when the delay is larger than the simulation step size [27]. The simulation would not run without the *Transport Delay*, and delaying the flow rate slightly had no effect on the results of the model.

Another embedded MATLAB function *Start Up* was incorporated into the valve model to account for the inertia of the rotor and flywheel when starting from rest. During testing it was determined that about 10 scfm of air at 100 psi were required to initiate flywheel movement. The simple *Start Up* function models this starting inertia by preventing rotation until the flow rate Q equals 10. Function *Start Up* outputs Q as determined by the valve whenever rpm exceeds zero since the flywheel once spinning will continue to turn with less than 10 scfm of air applied to it.

It was also found during initial testing that valve motion is delayed when moving from the fully closed position. When closed, the valve does not open immediately after a voltage is applied; there is a delay before the valve begins to open and air starts flowing. This delay is modeled with a negative initial value of *pct* (% Open) in *Data Store Memory - ValvePos*. Using a negative value at the start of a simulation forces the simulation to run for a time before *pct* becomes positive and air begins to flow in the model.

Chapter 5 - Controller Model

The model of the magnetic-bearing system shown in Figure 4-1 is nearly complete. The development of the plant (air turbine, rotor, flywheel and magnetic bearings) transfer function and that of the Aalborg valve were covered in Chapter 4. To complete the system model, a speed controller must be designed for the simulated system. This controller will be contained in the *Control Subsystem* block of Figure 4-1, and it will be implemented in the actual system pictured in Figure 2-1.

The speed controller for the magnetically-suspended rotor must limit large oscillations in rpm and achieve acceptable settling times. In addition, the desired speed should be maintained and not deviate from the set point. This implies that steady-state errors must be minimized or eliminated. While these requirements do not place specific numbers on the design criteria, the requirements will ensure the desired response for the intended purposes of the magnetic-bearing system - support the studies of ADR and flywheel health.

5.1 Proportional Control

To test the Simulink model developed thus far, a simple proportional controller was added to the *Control Subsystem* seen in Figure 4-1. A block diagram of this proportional controller is shown in Figure 5-1.

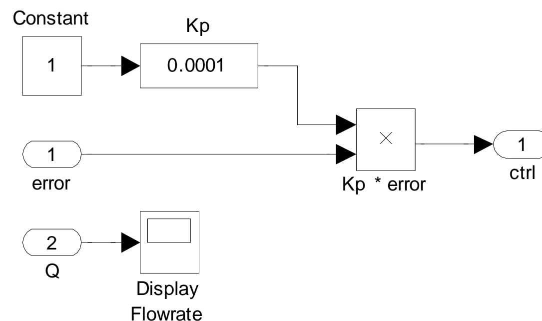

Figure 5-1 Proportional Controller

A proportional gain K_p of 0.0001 volt/rpm was chosen to test the model. This gain was not scientifically determined but resulted from the operating characteristics of the valve. The

Aalborg valve accepts a voltage or stepping rate of from 0+ to 2.5 volts. Thus, the signal from the controller must be within this range. Through testing, it was established that stepping rates above 0.7 volt resulted in temporary bearing instability and that the valve would not respond to stepping rates less than about 0.05 volt. So, a practical stepping rate (control signal) would be between 0.05 and 0.7 volt. The rate determined by the controller is K_p *x error* were *error* is the difference of desired or set rpm and actual or measured rpm. With the existing supply of compressed air, the flywheel can reach speeds in excess of 6000 rpm. Rounding up, the maximum error is thus *7000 - 0 = 7000 rpm*. To not exceed a stepping rate of 0.7 volt, K_p must equal $0.7\ volt/7000\ rpm = 0.0001\ volt/rpm$.

Incorporating proportional control into the model produced realistic behavior. The results of one simulation are given in Figure 5-2 for a desired speed or set point of 945 rpm.

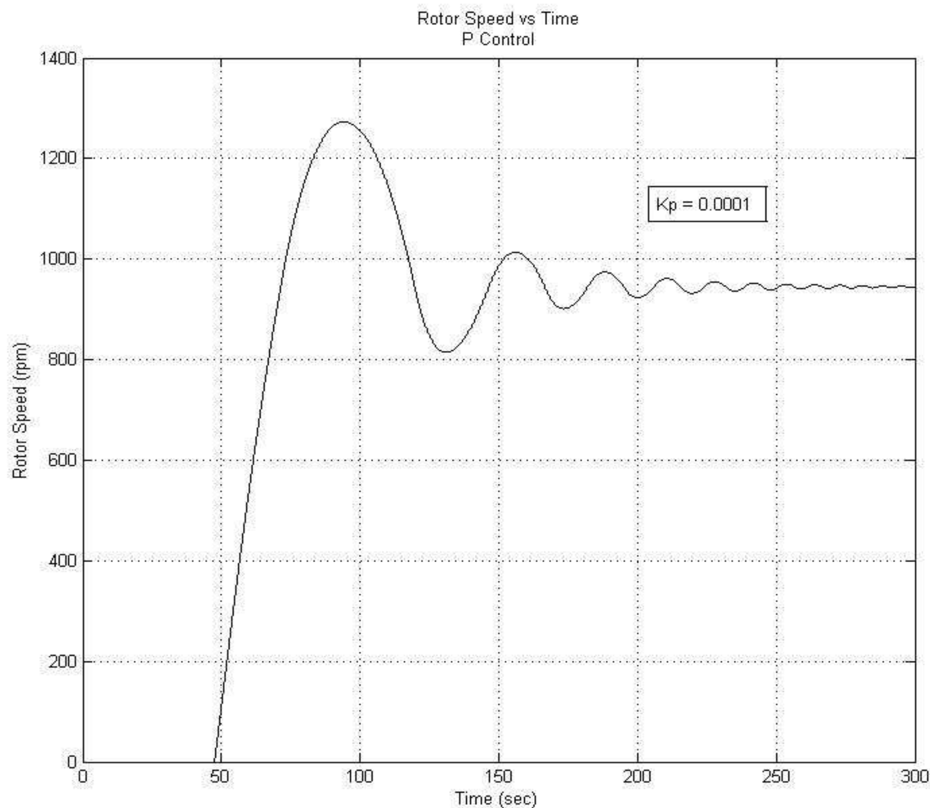

Figure 5-2 Simulated Response with P Control

Notice from the figure that there is a time delay of 50 seconds before the rotor starts moving. This delay is caused by both the starting inertia of the system and the delay in initial valve movement as discussed in Chapter 4. Once moving, the speed of the rotor increases quickly and overshoots the set speed by 330 rpm. The steady-state speed equal to one percent of the desired is then reached in about 250 seconds. The actual rpm oscillates about the desired prior to reaching steady-state, but there is no steady-state offset from the desired speed.

Oscillatory behavior is not expected from a first-order system but is present in the model. From Equation 4.4, the transfer function modeling the air turbine, rotor, flywheel and magnetic bearings is first order. It's the operation of the valve that imparts the oscillations in rpm, and valve behavior is not incorporated in the transfer function. With proportional control, the control signal only changes sign when the measured speed moves above or below the desired set point. The valve too then only reverses direction with a change in sign of the control signal. Oscillations in speed result since valve position does not start to change until after the actual rpm has exceeded the desired or the actual has fallen below the set point. In effect, the valve has increased the order of the system model by one, and second order systems oscillate. The increase in system order results from the valve acting as an integrator (*1/s*). The lack of steady-state errors is also explained by the valve as integrator.

5.2 Simulated and Actual Responses

Simulated responses were compared with actual system responses to determine the accuracy of the model given in Figure 4-1. For these comparison tests, a *Speed Control* subsystem was added to the existing bearing controller. This subsystem is shown in blue in Figure 5-3.

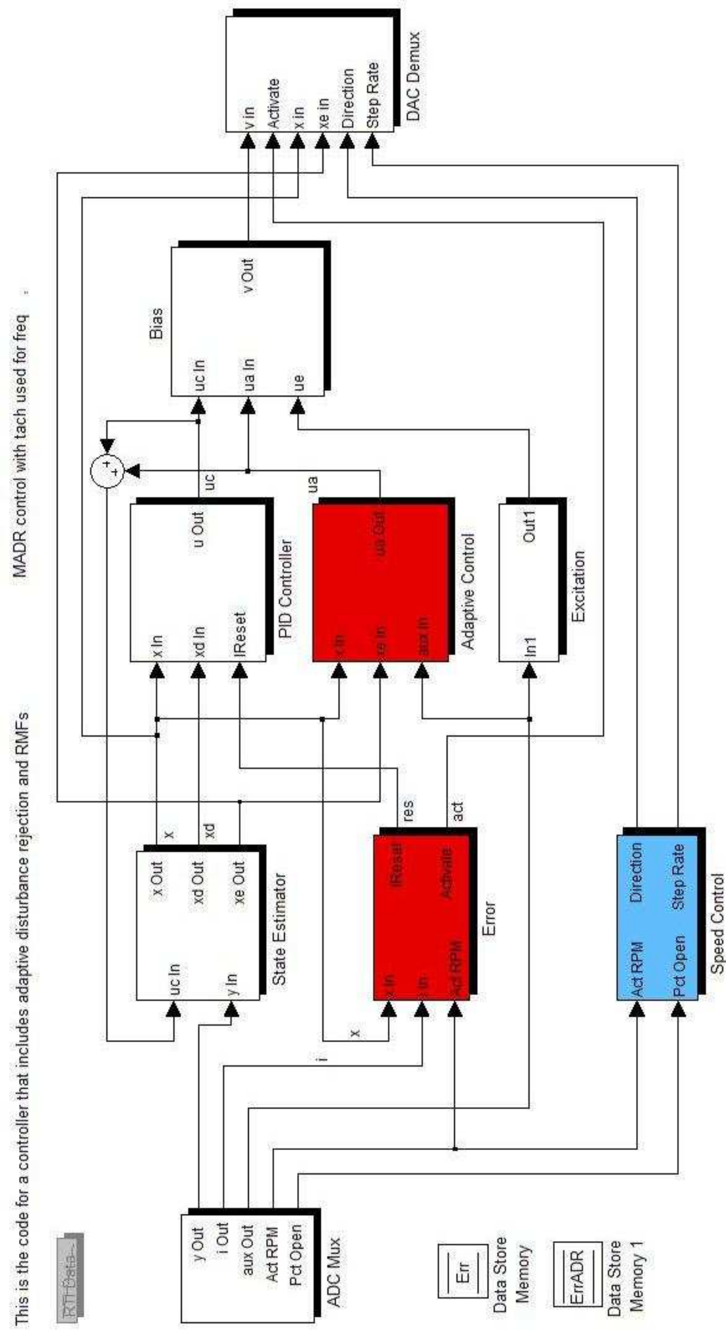

Figure 5-3 Bearing and Speed Controller

At this point, the *Speed Control* block contained only a proportional controller like that shown in Figure 5-1. Tests were conducted with both the model and the actual system using as set points the steady-state speeds reached with different step inputs of air. These speeds were shown in Figure 4-3 and given in Table 4-1. Test results are summarized in Table 5-1 with data from the actual system shown in parentheses.

Set Point (rpm)	Valve Delay (sec)	Peak Time (sec)	Peak Speed (rpm)	Overshoot (%)	First Under Time (sec)	First Under Speed (rpm)	Time to SS 1% (sec)
945	47 (53)	94 (87)	1273 (1265)	35 (34)	131 (112)	814 (791)	242 (263)
1609	31 (47)	83 (90)	2015 (1955)	25 (22)	123 (114)	1477 (1458)	196 (227)
2393	22 (35)	77 (83)	2840 (2610)	19 (9)	119 (105)	2276 (2271)	166 (190)
3629	18 (28)	74 (86)	4021 (3797)	11 (5)	117 (102)	3550 (3525)	135 (146)
4391	18 (27)	75 (92)	4680 (4463)	7 (2)	115 (108)	4336 (4378)	131 (110)
6076	18 (?)	108 (?)	6073 (?)	None (?)	None (?)	None (?)	108 (?)

Table 5-1 Simulated and Actual Response Data

All simulations and system tests were conducted with the proportional gain K_p set to 0.0001 volt/rpm and the maximum stepping rate limited to 0.3 volt. This stepping rate produced the best results with respect to the design criteria given at the beginning of Chapter 5. The valve delay for the simulations was set to -20%, and a starting inertia equivalent to 10 scfm was used. Data from the actual system for a set point of 6076 rpm are not available due to a minor electrical problem with one bearing.

A number of observations can be made from the data listed in Table 5-1. Using the actual system as the basis, the modeled valve begins to open from 8 to 37% sooner than the real valve. Peak rpm almost always arrives sooner (7 - 14%) than it actually does and at a slightly

higher (0.6 - 9%) speed. Overshoot is also greater in the simulated environment; overshoot is a maximum of 35% of the set speed at 945 rpm and just 7 % at 4391 rpm. In reality, overshoot is 34% and 2% at these same set points. The first valley in the modeled speed trace occurs later (6 -17 %) but at nearly the same rpm (0.2 - 3% higher). Finally, settling time taken at 1% of the set speed is generally faster (8 - 19%) in the model than with the actual magnetic-bearing system.

Overall, the simulated environment provides a good representation of the actual system. The model accurately calculates rotor speed at the first peak and valley and at steady-state. There is no steady-state error in the actual system, and none is predicted by the model. Also, the modeled valve will oscillate forever trying to exactly maintain the set speed. This is true of the real system, and to reduce wear on the Aalborg valve, its operation will be automatically deactivated when actual speed is close to set speed. The user will specify a band of rpm (a dead band) around the set point for which control is turned off, and the valve's position is held constant. The practice of limiting control around a set point is often done in industry to prolong the life of actuators [9].

The system model could be improved by adjusting simulated valve motion such that peaks and valleys in the velocity profile occur when they do in the real system. However, there is enough variability in the motion of the actual valve not to warrant additional development of the valve model. It was found during testing that the rate of valve movement was not always symmetric or repeatable. Given the same stepping rate in either direction, the valve closes more slowly than it opens. This is especially true at the low stepping rates required for smooth operation of the rotor.

5.3 Proportional-Derivative Control

It can be seen from Figure 5-2 and Table 5-1 that simulated and actual rotor speeds oscillate prior to achieving steady-state. To reduce oscillations, derivative control is often used with proportional to add damping to a system. Thus, derivative action was then tried on the model

prior to testing it on the actual system. The *Control Subsystem* pictured in Figure 5-1 was altered to form the proportional-derivative (PD) controller shown in Figure 5-4.

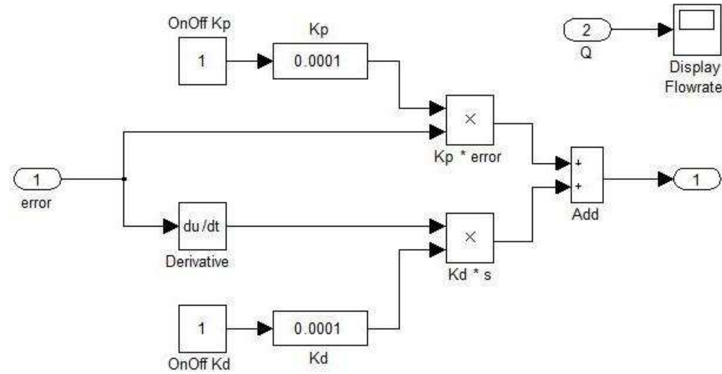

Figure 5-4 PD Controller

To test the model with PD control, the derivative gain K_d was set equal to the proportional gain K_p. With K_p and K_d set to 0.0001, the model's response to a desired rpm of 2500 is shown in Figure 5-5.

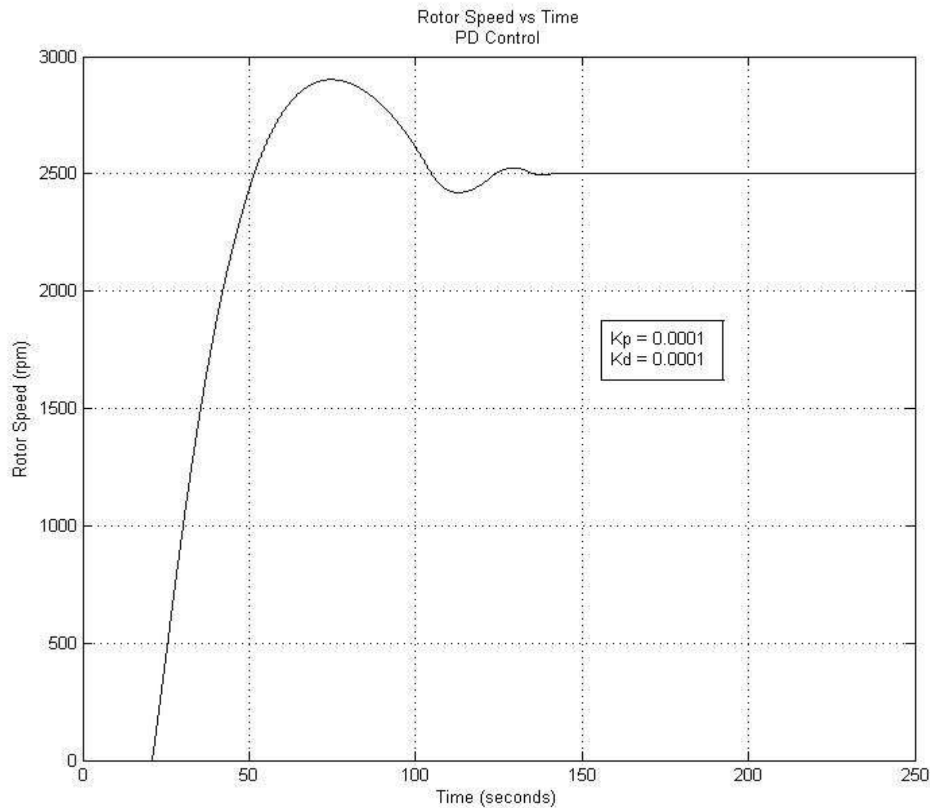

Figure 5-5 Simulated Response with PD Control

Regardless of set point, simulated system response curves with PD control all resemble that shown in Figure 5-5. The benefits of derivative control can be seen by comparing the system's response using P-control (Figure 5-2) with that using PD-control (Figure 5-5). For a given set point, adding derivative action reduces peaks and valleys, limits oscillations and decreases the time to reach steady-state. Further reductions in these measures can be achieved by increasing the derivative gain K_d. Practically, if K_d is increased too much, system response becomes sluggish. To determine initial values of K_d for the actual system, the Good Gain method was applied to the results of numerous system simulations.

5.4 Good Gain

Recall from Chapter 1 that the Good Gain method does not require a process model. Since a model had been developed, it was used to provide initial estimates of K_d for the actual system.

Good Gain calls for adjusting the proportional gain K_p until the first undershoot is barely perceptible. When this is achieved, the time to the first peak or overshoot t_o and that to the first valley or undershoot t_u are measured. The difference in these two times is $T_{ou} = t_u - t_o$, and it is used to find K_d. To illustrate Good Gain, the response curve for a set point of 2000 rpm is shown in Figure 5-6. The times t_o and t_u are shown on the graph along with the Good Gain formulas for calculating the derivative gain. These formulas are $T_i = 1.5T_{ou}$, $T_d = T_i/4$ and $K_d = K_pT_d$.

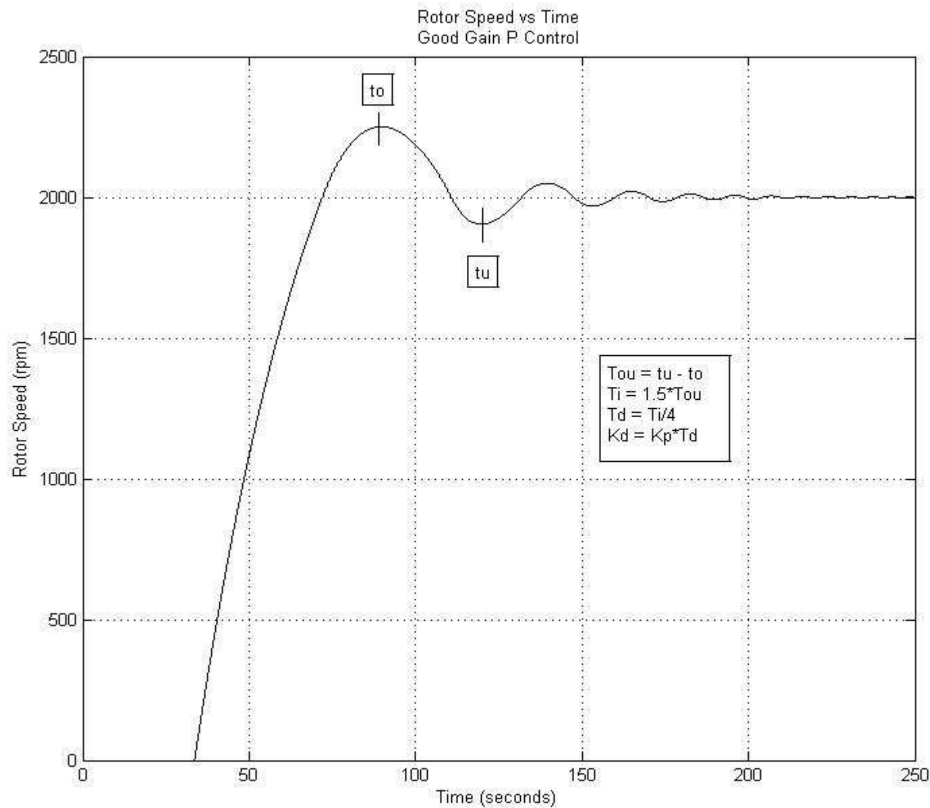

Figure 5-6 Determination of Good Gain

Note from Figure 5-6 that the first valley is more than barely perceptible. With the system model, K_p could have been adjusted to any value until a barely perceptible first valley was achieved. The range of K_p is limited with the actual system since the minimum stepping rate of the Aalborg valve is 0.05 volt. To more accurately model reality, K_p for all simulations was set so that the modeled valve always stepped at 0.05 volt. This low rate results in the

minimum undershoot possible and more closely approximates conditions specified by Good Gain. For example, using the minimum stepping rate for a set point of 2000 rpm results in K_p = 0.05 volt/2000 rpm = 0.000025 volt/rpm.

A series of simulations were conducted to determine initial estimates of K_d. For all these tests, the minimum possible stepping rate and the set point were used to set K_p. Simulations were run for desired speeds of from 500 to 6000 rpm in increments of 500 rpm, and test results are given in Table 5-2.

Set Point (rpm)	K_p x 10^{-5} (volts/rpm)	T_{ou} (sec)	$T_i = 1.5T_{ou}$ (sec)	$T_d = T_i/4$ (sec)	$K_d = K_pT_d$ x 10^{-5} (volt*sec/rpm)
500	10.00	37.0	55.5	13.88	138.80
1000	5.00	35.0	52.5	13.13	65.60
1500	3.33	32.0	48.0	12.00	39.60
2000	2.50	30.0	45.0	11.25	28.10
2500	2.00	28.0	42.0	10.50	21.10
3000	1.67	25.0	37.5	9.38	15.70
3500	1.43	23.0	32.5	8.63	12.30
4000	1.25	20.0	30.0	7.50	9.40
4500	1.11	19.0	28.5	7.13	7.90
5000	1.00	17.0	25.5	6.38	6.40
5500	0.91	16.0	24.0	6.00	5.50
6000	0.83	15.0	22.5	5.27	4.70

Table 5-2 Good Gain Values for K_d

It is seen from Table 5-2 that the calculated values of K_d using Good Gain are too high. The derivative gains are much greater than the proportional ones used in the simulations. In determining K_d, Good Gain suggests using an initial estimate of $T_i/4$ for the derivative time T_d. However, if this results in unacceptable values of K_d, Good Gain recommends dividing T_i

by a number greater than 4. Before increasing this divisor, simulations were run using the derivative gains shown in Table 5-2. Regardless of set point, these gains added too much damping to the system. It was then decided to adjust K_d to minimize undershoot in the model. Simulations were then run at the same set points shown in Table 5-2 to determine derivative gains that minimize undershoot at each desired speed. For each set point, the proportional gains remained unchanged from those given in Table 5-2. Using minimum undershoot as the design criterion resulted in response curves like the one shown in Figure 5-7 (1500 rpm set point) and the derivative gains listed in Table 5-3.

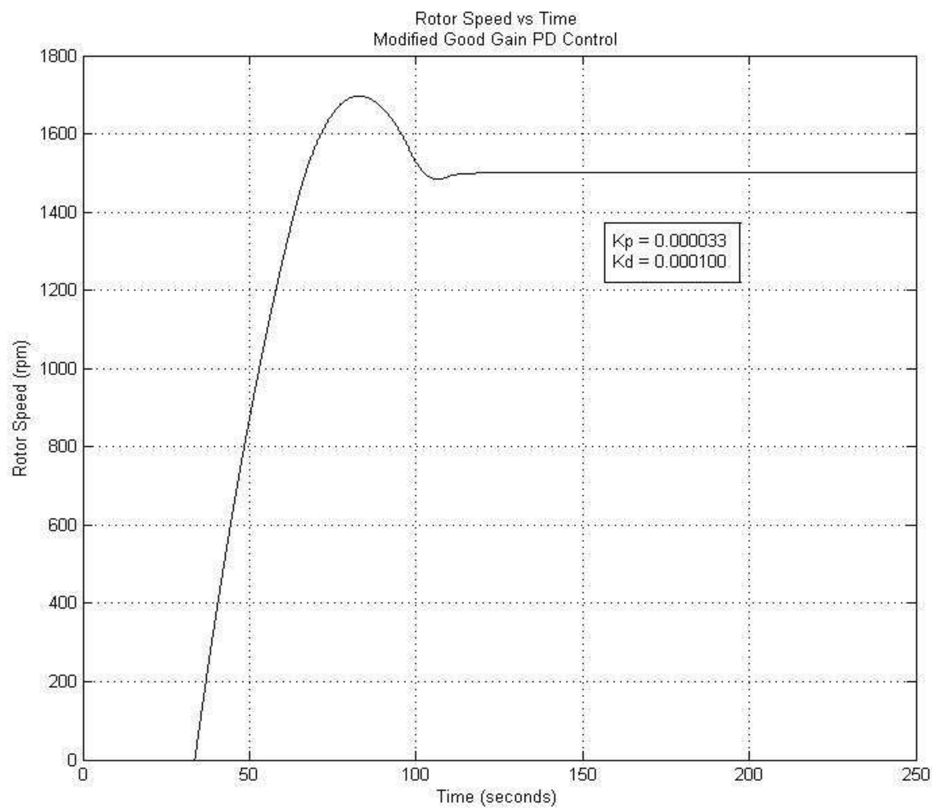

Figure 5-7 Simulated Response with Modified Good Gain PD Control

Set Point (rpm)	K_p x 10^{-5} (volts/rpm)	K_d x10^{-5} (volt*sec/rpm)
500	10.00	30.0
1000	5.00	16.0
1500	3.33	11.0
2000	2.50	7.0
2500	2.00	5.0
3000	1.67	4.0
3500	1.43	3.0
4000	1.25	2.0
4500	1.11	2.0
5000	1.00	1.0
5500	0.91	1.0
6000	0.83	1.0

Table 5-3 Modified Good Gain Values for K_d

The derivative gains listed in Table 5-3 can also be determined from the data in Table 5-2 if a derivative time equal to $T_i/16$ is used. Values of K_d found using $T_i/16$ are more realistic than those calculated from $T_i/4$. Even with a divisor of 16, the derivative gains are still higher than the proportional ones used in the simulations. However, no additional simulations were conducted to determine initial estimates of K_d. Their final values were found by tuning the controller of the actual system.

5.5 Gain Scheduling

Design criteria for the speed controller were given at the beginning of this chapter. Recall that the controller must limit large oscillations in rpm and achieve acceptable settling times. These design criteria were achieved in the just-performed simulations by limiting the stepping rate and incorporating derivative control. From Table 5-3, it is seen that desirable performance can

be attained at various speeds provided the controller gains K_p and K_d are varied with rpm. These gains can be dynamically altered through gain scheduling, a method used to adjust controller parameters as a function of a process variable. The concept of gain scheduling originated in connection with the development of flight control systems [28]. Today, gain scheduling is a standard technique for aircraft control, and it is also used in automobile engine control units [29]. For the magnetic-bearing system, the process variable is the desired speed of the rotor, and the controller parameters are the proportional and derivative gains.

True gain scheduling was not implemented in the simulations, but the gains were automatically adjusted based on the desired speed. Actual gain scheduling would vary the controller parameters for each change in set point. However, the set point was not altered during a simulation. In the actual system, as in aircraft control, the set point can vary constantly. The goal with each simulation was to achieve the response shown in Figure 5-7 for any given set point. As is evident from Table 5-3, a consistent response requires different gains for each desired speed. These gains were selected automatically using the same approach as that used in Chapter 4.3 to chose the coefficients for the transfer function. To select the appropriate gains, equations for K_p and K_d as functions of rpm were needed.

To find these equations, curves were fit to the data in Table 5-6 using MATLAB's *cftool*, and these curves are shown in Figure 5-8.

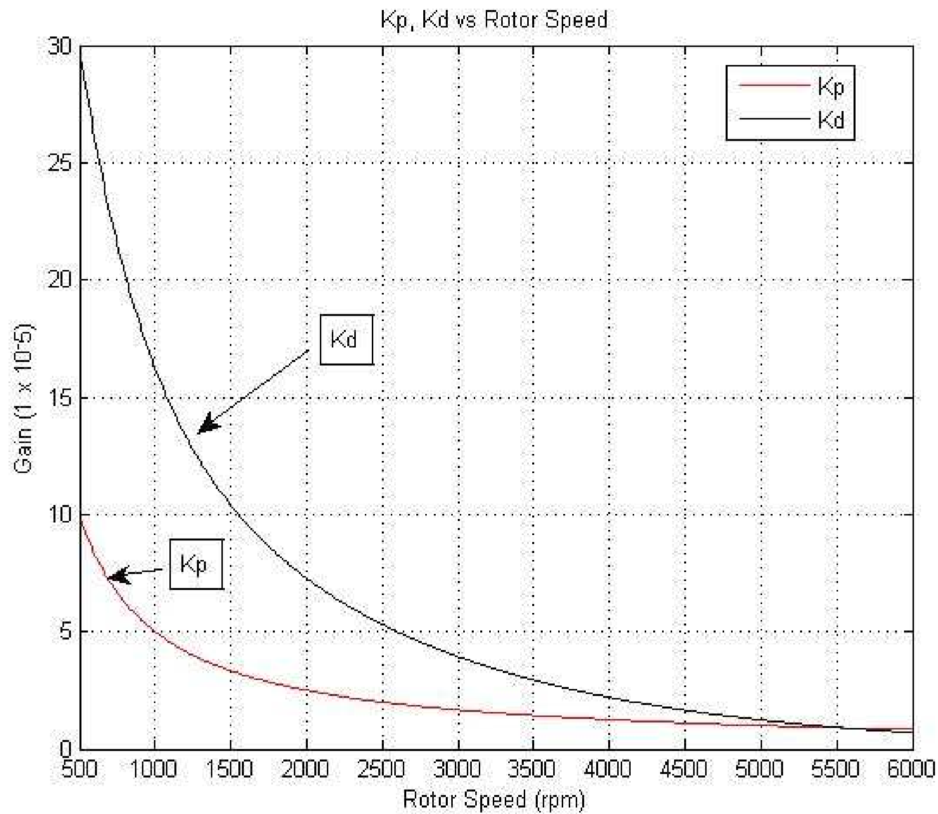

Figure 5-8 PD Gains vs Rotor Speed

The equations for the gains are:

$$K_p = 5001 * \omega^{-1} \tag{5.1}$$

$$K_d = 46.74 * e^{-0.00249*\omega} + 21.99 * e^{-0.00058*\omega} \tag{5.2}$$

To use Equations 5.1 and 5.2, the angular velocity of the rotor, ω, must be in units of *rpm*. Note from Figure 5-8 that for any speed below 500 rpm, the gains are determined using a set point of 500 rpm. Equations 5.1 and 5.2 were incorporated into the PD controller shown in Figure 5-4 to yield a controller, pictured in Figure 5-9, with automatic gain selection. The equations for K_p and K_d are contained in the embedded MATLAB function *Schedule Gains*.

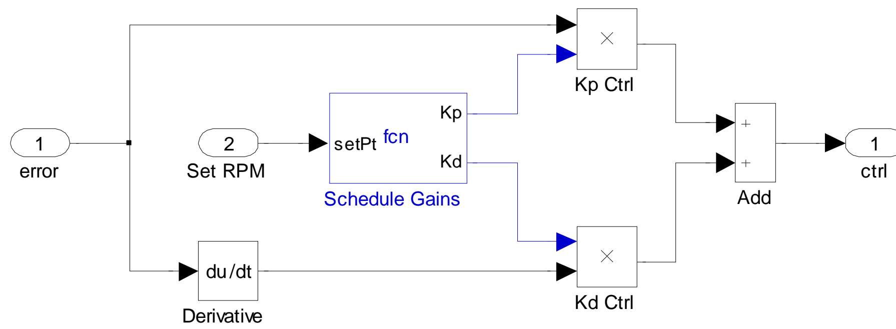

Figure 5-9 PD Controller with Gain Selection

With the automatic-gain-selection controller, the simulated system's velocity profile is always the same regardless of set point. This profile is identical to that plotted previously in Figure 5-7 for a desired speed of 1500 rpm. If the set point had been 3660 rpm or any other speed, the velocity plot would have looked the same as that in Figure 5-7. Thus, the automatic gain selection controller achieves the design objectives of limited overshoot/undershoot and reasonable settling time for all set points.

Prior to discussing the implementation of the actual proportional/derivative controller, it is enlightening to examine a PD control signal. Figure 5-10 shows individual proportional and derivative signals plus the PD composite.

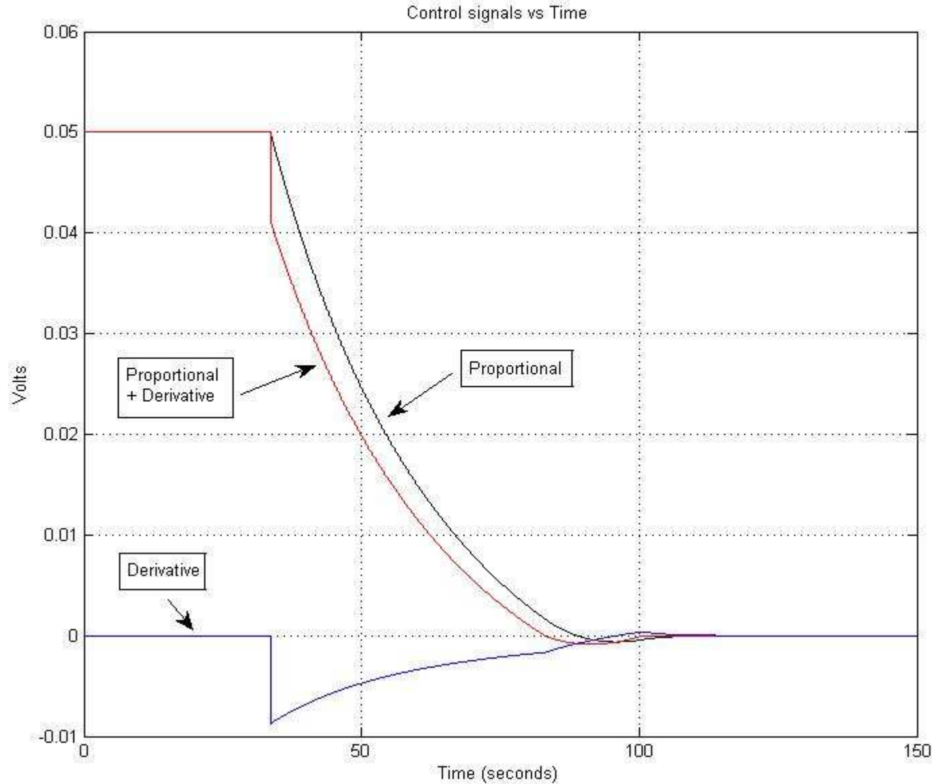

Figure 5-10 P, D and PD Control Signals

Figure 5-10 was generated from a simulation without automatic gain selection using a desired speed of 3500 rpm and typical values of K_p and K_d. It can be seen from Figure 5-10 how derivative action anticipates future actuator response and adds damping to a system. The sign of the composite PD signal changes sooner than that for the proportional signal. When the control signal changes sign (i.e. crosses through zero volts), the direction of valve motion changes too. For this simulation, the proportional signal first changes sign at 89 seconds which as expected occurs at the set point of 3500 rpm. When derivative action is added to proportional, the PD signal changes sign at 83 seconds and 3380 rpm. The speeds were read from the velocity plot which is not shown. With just proportional control, the valve changes direction (starts to close) once the set point is reached, resulting in overshoot. The PD controller anticipates the set point and changes the direction of valve motion prior to reaching the desired rpm, thereby minimizing overshoot. Limiting overshoot has the same effect as

adding damping to a system - both act to slow response. If undershoot were present in the simulation, the anticipatory nature of PD control would be seen to minimize it also.

Chapter 6 - Controller Implementation

The actual speed controller was designed using the knowledge gained from the simulated system. Using this system, it was determined that the following were required to regulate the speed of the magnetically-suspended rotor:

1) provide proportional control for basic speed regulation,
2) incorporate derivative action to minimize oscillations in speed,
3) add adjustable gains to tailor response,
4) limit stepping rates to provide consistent response,
5) deactivate derivative control at higher speeds where overshoot is minimal,
6) provide a "dead band" to reduce wear on the flow-control valve, and
7) allow manual operation of the valve for testing purposes.

Simulations also showed that integral action was not necessary since the system displayed no steady-state error. Recall that the valve is acting as an integrator.

6.1 Controller Design

To meet the requirements for speed regulation, the controller shown in Figure 6-1 was constructed in Simulink.

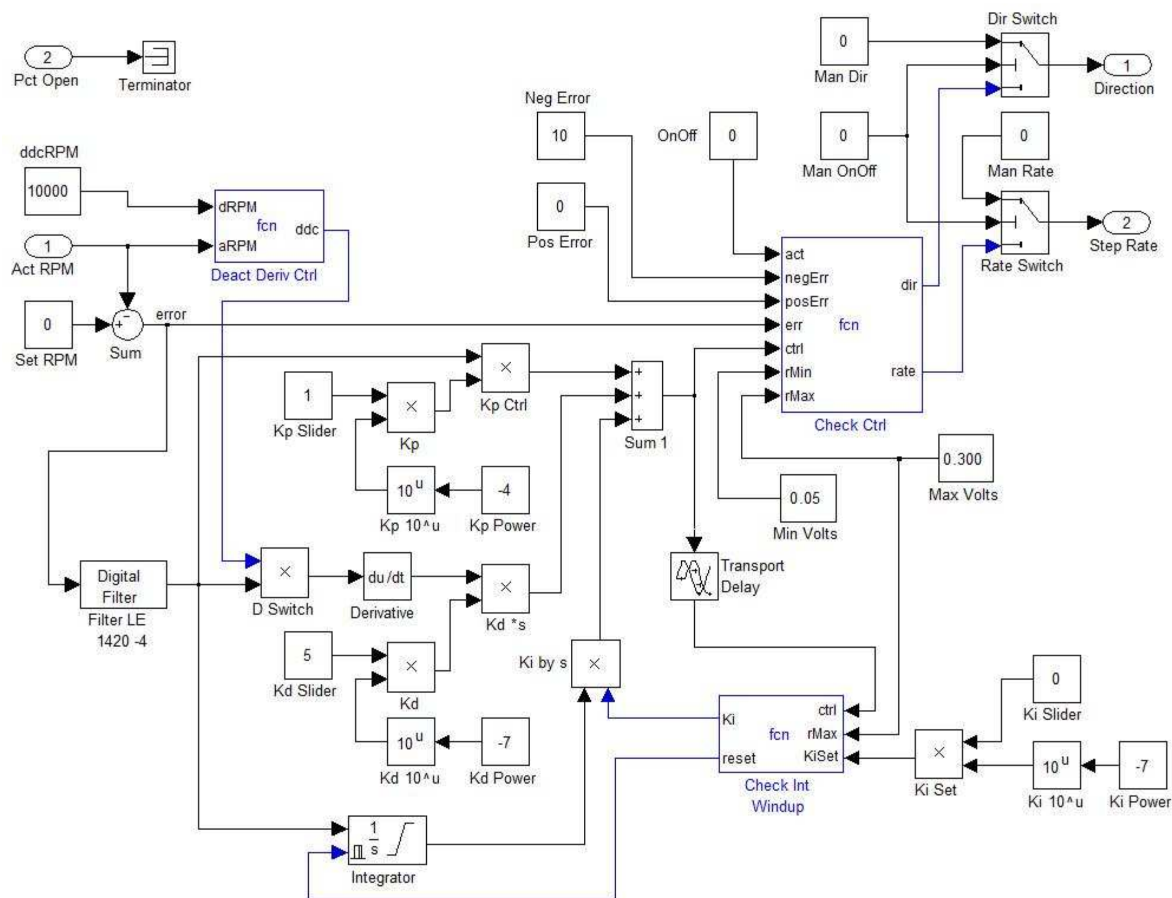

Figure 6-1 Speed Controller

This controller is contained within the blue *Speed Control* block of the complete magnetic-bearing controller given previously in Figure 5-3. It was also explained earlier in Chapter 2.2 how block diagrams are converted to executable code through Real-Time Workshop for use by dSPACE.

The block diagram in Figure 6-1 is that for a PID controller of the form $\left(K_p + \frac{K_i}{s} + K_d s\right)$. While integral action is not currently needed to regulate the speed of the rotor, it was added in case of a future requirement. There are numerous inputs to and outputs from the controller. Inputs are of two types - user and system. User inputs are entered through instrument panels to be discussed in the next chapter. Inputs from the magnetic-bearing system are read by the

controller via the A/D converter. Outputs are written to the system by the controller through the D/A converter. All input and output is summarized in Table 6-1.

Variable Name	I/O Type	Units	Function
Act RPM	system in	rpm	Current speed of the rotor measured by the tachometer shown in Figures 3-2 and 3-3.
Pct Open	system in	% open	Current position of valve determined by the LVDT shown in Figure 3-4; not used.
Set RPM	user in	rpm	Set point - desired speed of the rotor.
ddcRPM	user in	rpm	Rotor speed above which derivative action is turned off.
Neg Error	user in	rpm	Upper limit of dead band.
Pos Error	user in	rpm	Lower limit of dead band.
OnOff	user in	none	Switch for automatic speed control.
Man OnOff	user in	none	Switch for manual speed control.
Man Dir	user in	none	Direction of valve travel - open, close or hold - when in manual mode.
Man Rate	user in	volts	Stepping rate for valve when in manual mode.
Max Volts	user in	volts	Maximum stepping rate allowed.
Min Volts	user in	volts	Minimum stepping rate allowed.
K_p Power, K_p Slider	user in	volts/rpm	Proportional gain.
K_i Power, K_i Slider	user in	volts/rpm*sec	Integral gain; not used.
K_d Power, K_d slider	user in	volts*sec/rpm	Derivative gain.
Direction	controller out	volts	Direction of valve travel - open, close or hold.
Step Rate	controller out	volts	Stepping rate for valve.

Table 6-1 Controller Input and Output

Most of the variables in Table 6-1 are self explanatory; however, a few require further explanation. The variables *Neg Error* and *Pos Error* together specify a band of rpm (a dead band) around the set point for which control is turned off, and the valve's position is held. It was mentioned in Chapter 5.2 that dead bands are used in industry to reduce wear on actuators. For example, if the set point is 3600 rpm and *Neg Error = 10* and *Pos Error = 5*, then the valve's position is held constant between 3595 and 3610 rpm. It was found during initial testing of the controller that a properly specified dead band has the added benefit of reducing the amplitude of oscillations in speed.

Variables *Min Volts* and *Max Volts* place limits on the stepping rate sent to the valve. The stepping rate (control signal) determined by the speed controller is often less than the minimum stepping rate of the valve. This rate is about 0.05 volt, and *Min Volts* ensures that the valve will always respond. *Max Volts* is used in place of automatic gain selection. It was also determined during testing that limited stepping rates (<= 0.3 volt) combined with small but constant controller gains provide more consistent response to changes in desired speed. Thus, the automatic adjustment of controller gains is not absolutely necessary. A big change in set point can result in high stepping rates even with the use of small gains. If stepping rates exceed 0.3 volt, large speed oscillations and temporary bearing instability result. The valve can step at 2.5 volts, but at this rate, it would be impossible to control the speed of the rotor.

Addition blocks of the speed controller requiring explanation are the three embedded MATLAB functions, *Check Ctrl*, *Check Int Windup* and *Deact Deriv Ctrl*, the *Digital Filter* and *Transport Delay*. Function *Check Ctrl* determines stepping rate and valve direction based on controller input and the user inputs just described. *Check Int Windup* is part of an anti windup circuit that limits integrator output if integral action is used. For now, integral control is not needed and is disabled from the *Control Parameters* instrument panel by setting the value of the K_i slider to zero. This instrument panel and all others used to control the rotor will be fully described in the next chapter. Function *Deact Deriv Ctrl* turns off derivative control when rotor speed exceeds a user-specified rpm. Maximum rotor speed is limited by the supply of compressed air to about 6500 rpm. There is little overshoot or undershoot due to changing set points at high rpm, so derivative action has little use at high speeds. To keep

derivative control active at all times, set the *Deact Deriv Control* variable on the *Control Parameters* panel to an unattainable speed (e.g. 10,000 rpm). Refer to the program listings in Appendix B if more detailed explanations are needed of the embedded MATLAB functions.

It was explained during the discussion of the simulated system that a *Transport Delay* was needed to eliminate algebraic loop errors from the model. A *Transport Delay* was used with the actual controller for the same reason. This delay had no effect on controller operation since integral action is not used. Recall that if a delay is needed, it should be slightly larger than the integration time step [27].

Pictured in Figure 6-2 is the low-pass filter added to remove noise from the tachometer signal and to improve the performance of derivative action.

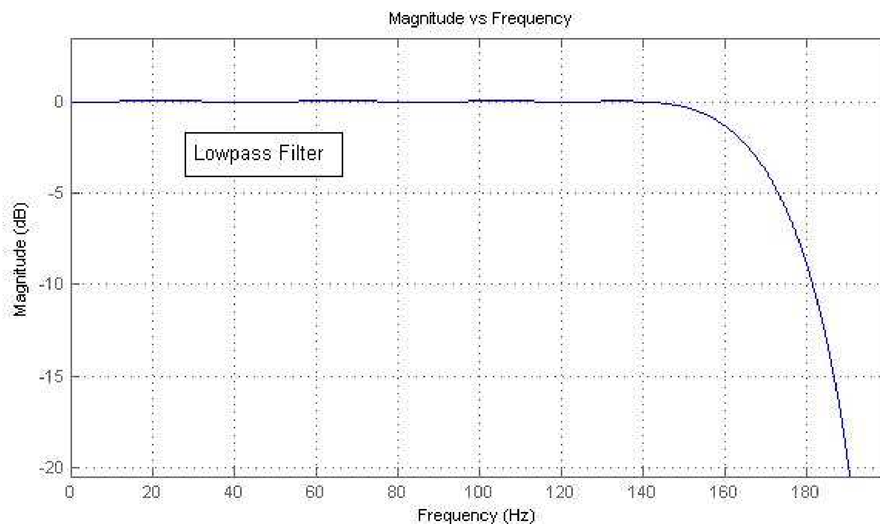

Figure 6-2 Low-Pass Filter

MATLAB's Filter Design and Analysis Tool *fdatool* was used to construct a filter having a pass frequency (end of pass band) of 140 Hz and a stop frequency (beginning of stop band) of 200 Hz. A sampling rate of 400 Hz or twice the stop frequency is employed. The pass and stop frequencies correspond to rotor speeds of 8400 rpm and 12,000 rpm, respectively. With the current supply of compressed air, the rotor will spin to between 6000 (100 Hz) and 7000 rpm (117 Hz). If additional air flow becomes available and raises the maximum rpm limit, a

new filter having higher pass and stop frequencies can easily be designed with the *fdatool* to replace the current filter.

6.2 Testing and Tuning

Through testing, a single controller configuration was established that accurately regulates the speed of the rotor for any set point from 200 rpm to maximum rpm. The variables and their values for this one configuration are listed in Table 6-2.

Variable Name	Value
ddcRPM	10,000 rpm
Neg Error	10 rpm
Pos Error	5 rpm
Max Volts	0.3 volt
Min Volts	0.05 volt
K_p Power, K_p Slider	0.0001 volt/rpm
K_i Power, K_i Slider	0
K_d Power, K_d slider	0.0000005 volt*sec/rpm

Table 6-2 Controller Configuration

The PD controller configured as shown in Table 6-2 provides excellent control when starting at rest and when changing from one set point to another (e.g. raising the speed from 932 to 3145 rpm or lowering the speed from 5689 to 2166 rpm). Once steady-state is reached, rotor speed varies only a few rpm either way from the set point. Any set point equal to or greater than 200 rpm could be chosen to demonstrate the controller’s capabilities. For illustrative purposes, velocity versus time plots are shown for desired speeds of 800, 1750, 2500, 3300

and 4000 rpm. These plots were generated for a rotor initially at rest. There are two graphs for each set point - one where only proportional control is used and the other where derivative action is added to proportional. The advantages of PD over P control are evident from the figures. Velocity plots will also be shown for changing set points, and all graphs of velocity versus time were captured with ControlDesk.

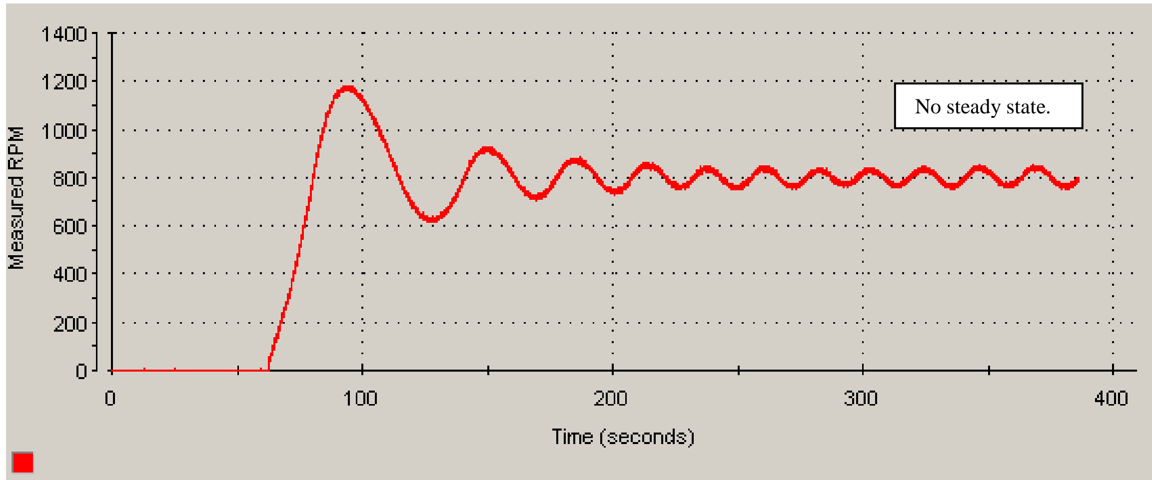

Figure 6-3 P Control - 800 rpm

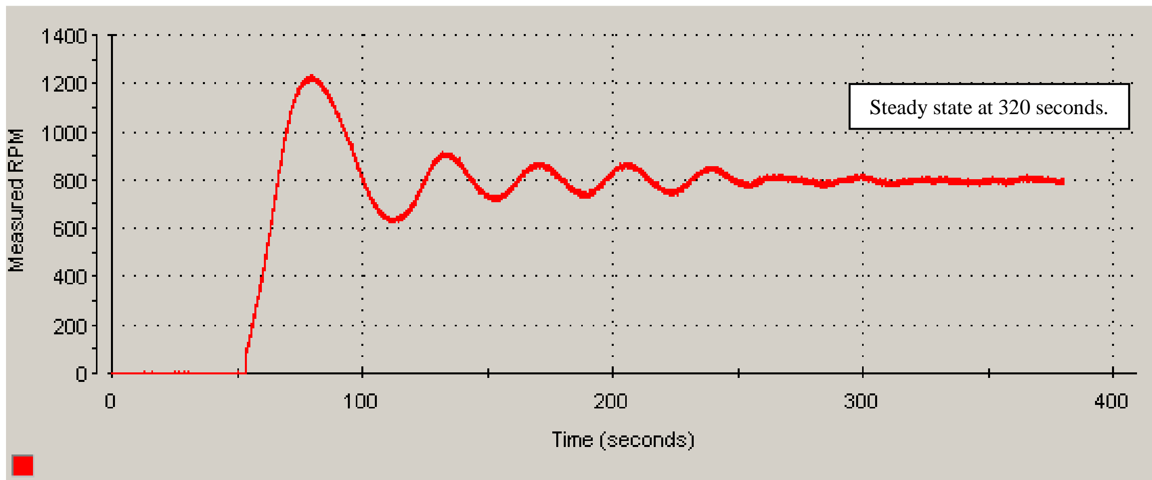

Figure 6-4 PD Control - 800 rpm

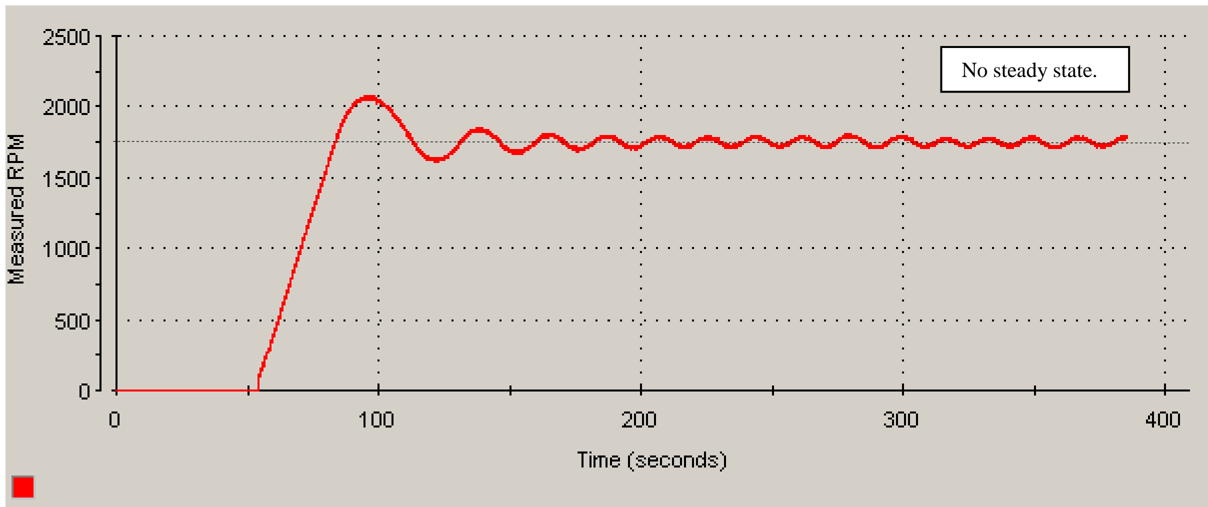

Figure 6-5 P Control - 1750 rpm

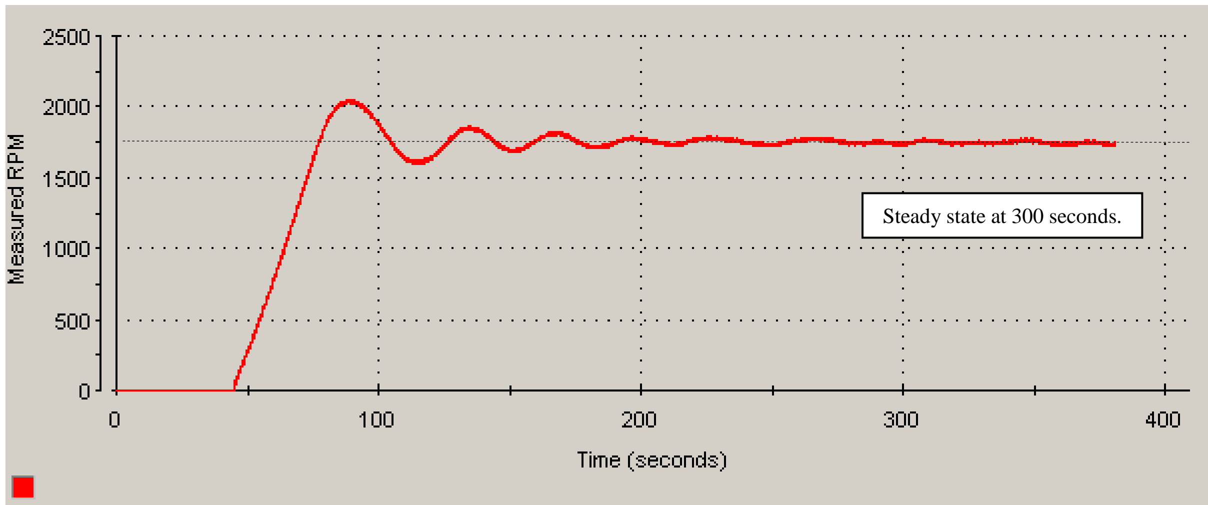

Figure 6-6 PD Control - 1750 rpm

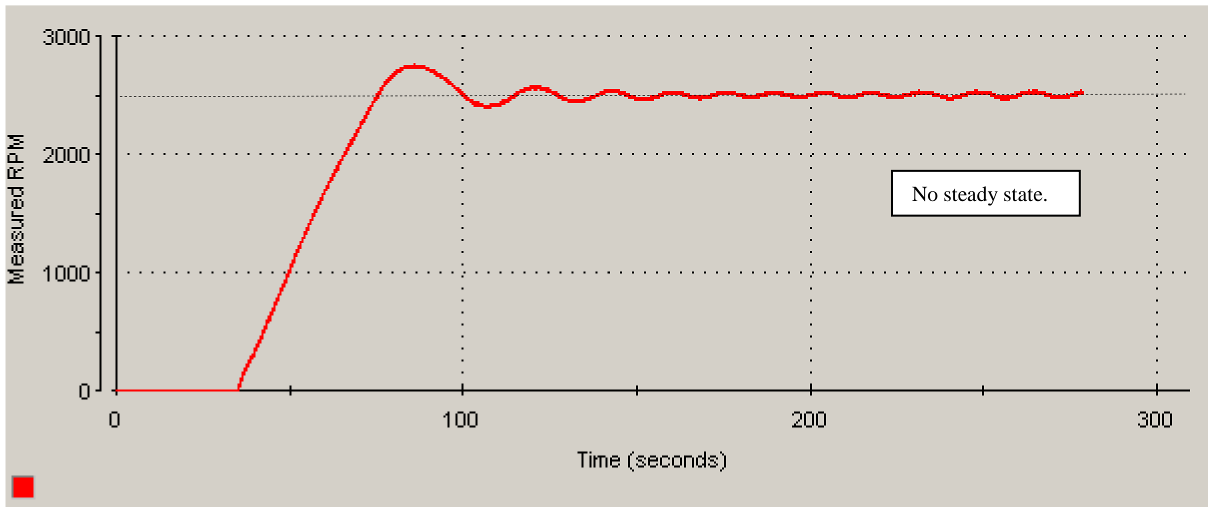

Figure 6-7 P Control - 2500 rpm

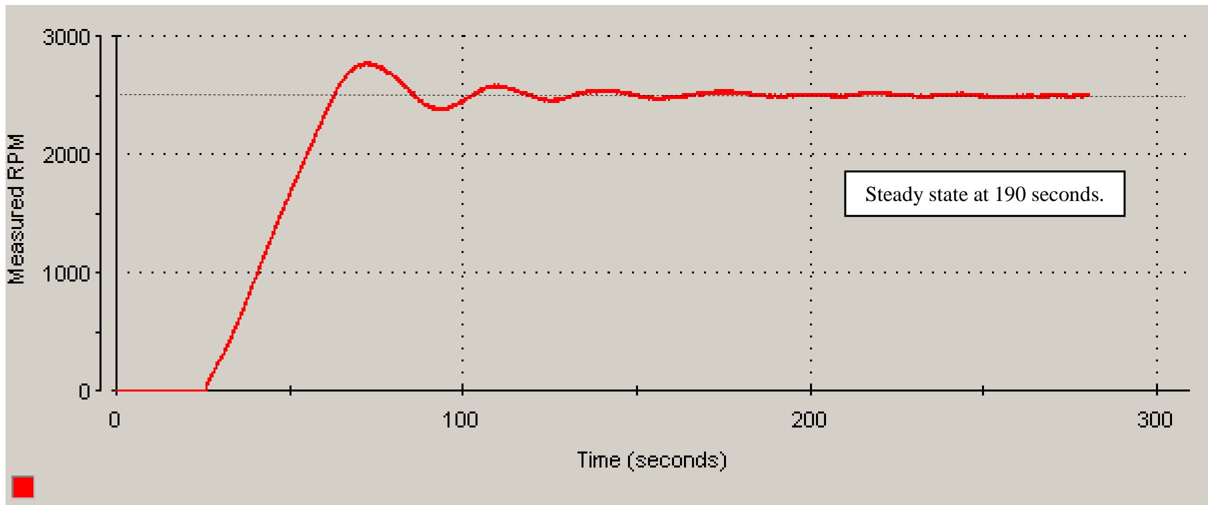

Figure 6-8 PD Control - 2500 rpm

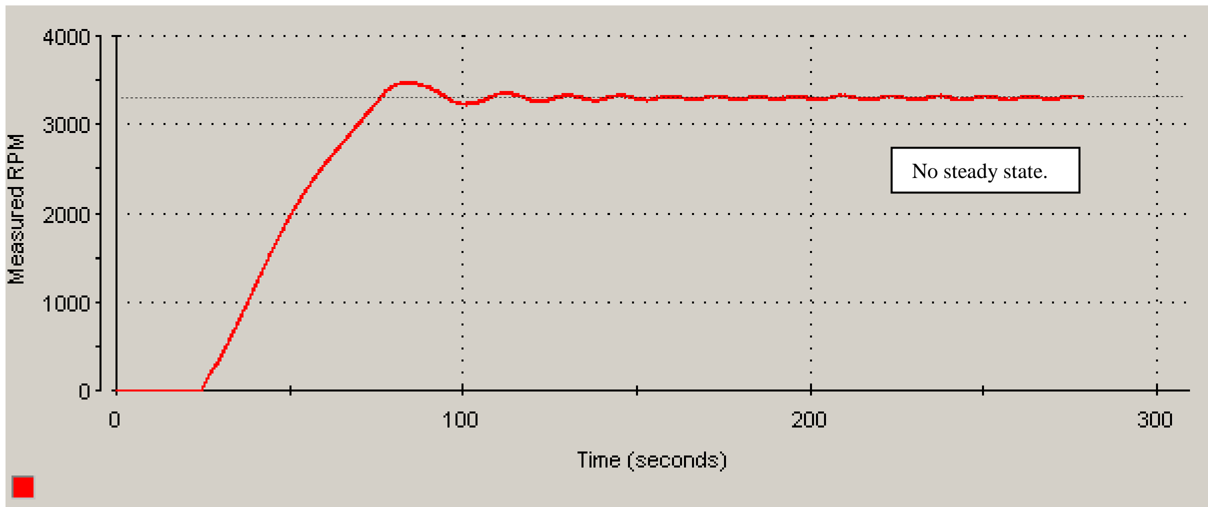

Figure 6-9 P Control - 3300 rpm

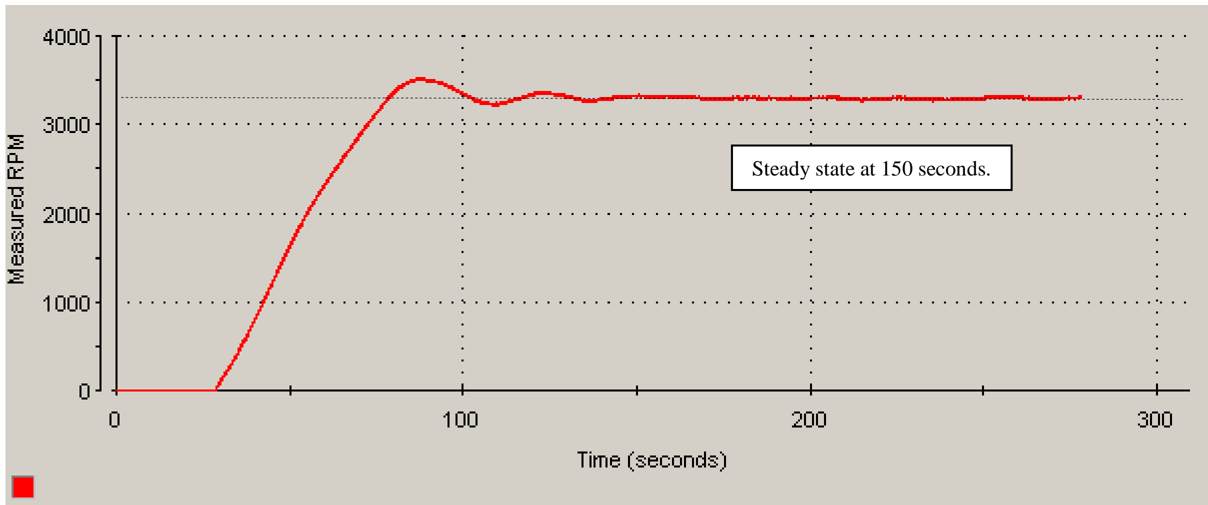

Figure 6-10 PD Control - 3300 rpm

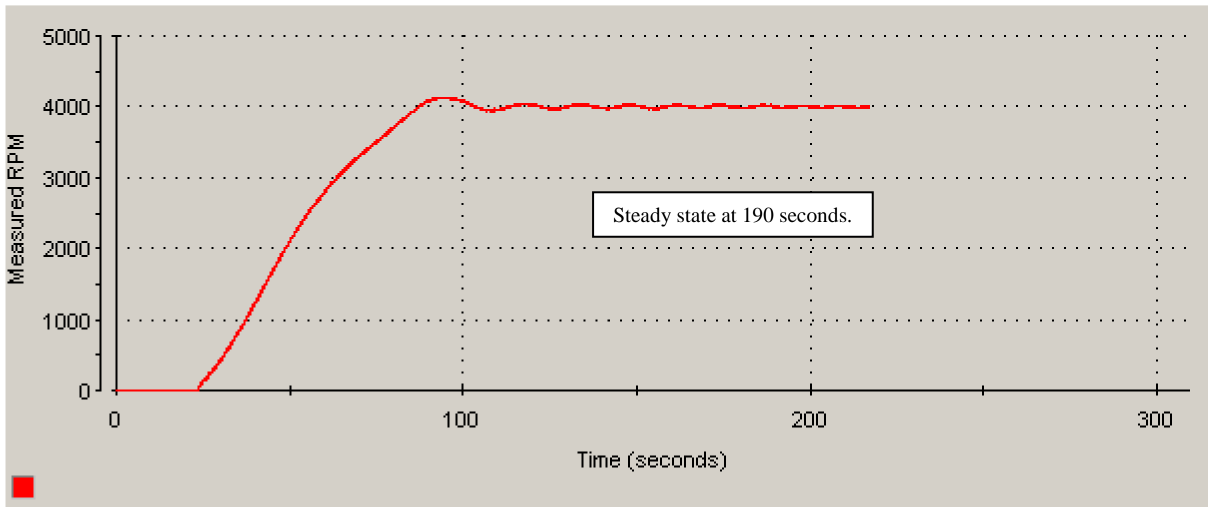

Figure 6-11 P Control - 4000 rpm

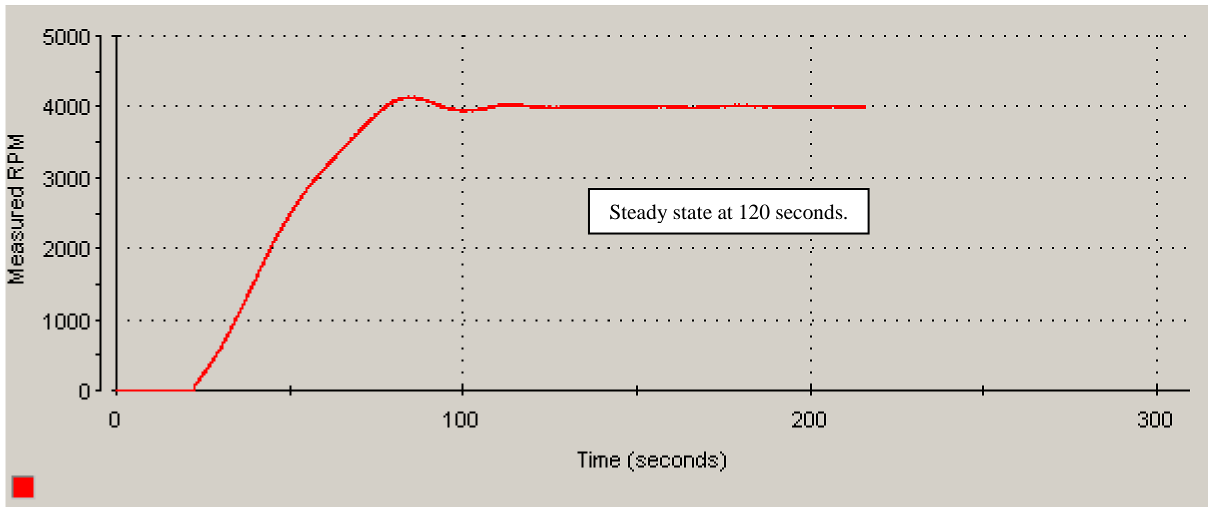

Figure 6-12 PD Control - 4000 rpm

It is seen from the data that proportional control alone cannot minimize or eliminate oscillations about the set point. These oscillations are especially noticeable at lower speeds and they persist for all set points shown. With just proportional control, a true steady-state condition is never reached except at 4000 rpm. However, PD control minimizes or eliminates

oscillations about the desired rpm, and steady-state is attained at the time shown on each graph.

Velocity data for a change in set point are given in Figures 6-13 and 6-14. These figures begin with the rotor initially spinning at a steady-state speed. Figure 6-13 graphs the velocity response when the set point is reduced from 4000 to 1000 rpm. The speed response for an increase in desired speed from 500 to 2500 rpm is shown in Figure 6-14. Both plots were generated using the controller configuration given in Table 6-2.

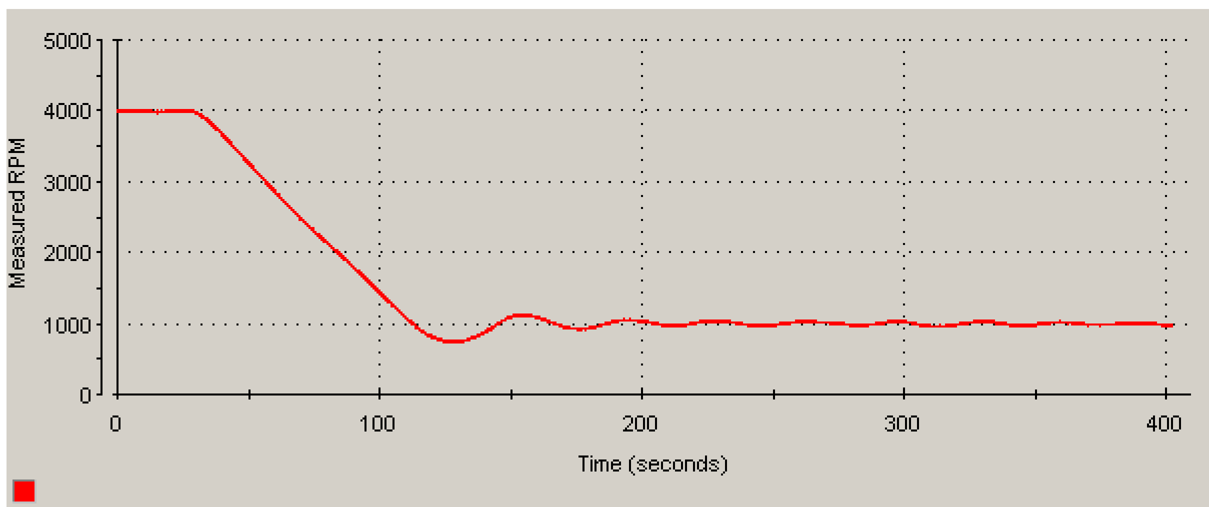

Figure 6-13 Changing Set Point - 4000 to 1000 rpm

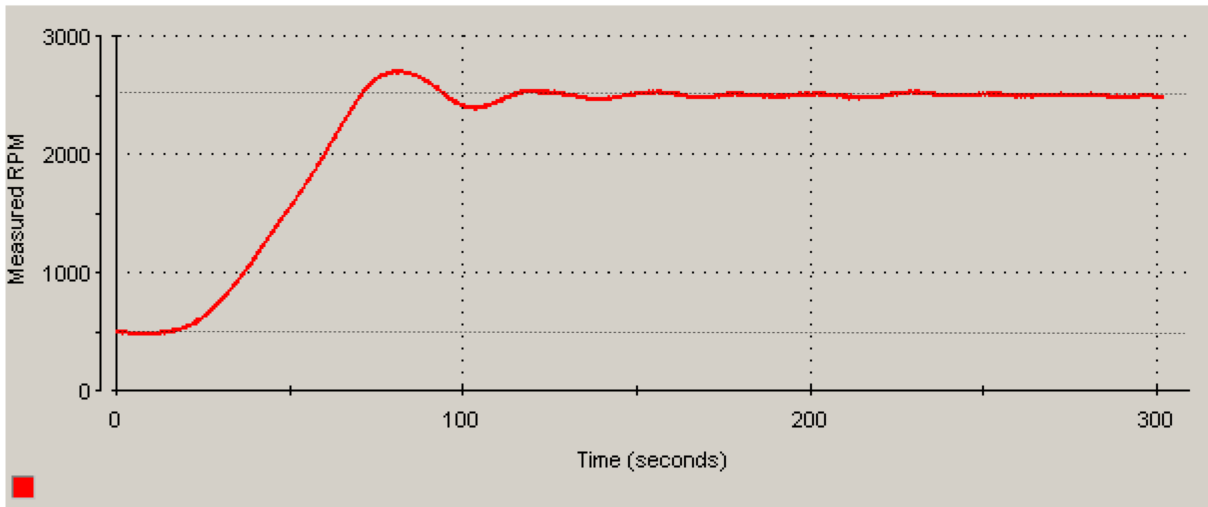

Figure 6-14 Changing Set Point - 500 to 2500 rpm

Note the slight oscillations about the final set point of Figures 6-13 and 6-14. These oscillations resulted from disturbances (i.e. pressure variations) in the air supply to the turbine. When Figures 6-13 and 6-14 were generated, other labs were using air from the common air supply serving the entire building. The responses of the rotor and controller to changing line pressures are seen in these minor speed oscillations about the final set point. It is evident from Figures 6-3 through 6-14 that the PD controller accurately regulates the speed of the rotor when starting from rest, when changing set points and when compensating for disturbances.

Chapter 7 - Operating the Rotor

Operation and control of the rotor is accomplished through ControlDesk using a series of layouts or instrument panels. The functions of each panel will be discussed during or after the procedures are given for starting the system, activating the bearings, setting and changing rotor speeds and stopping the rotor. Note that some familiarity with ControlDesk is needed to execute these procedures.

7.1 Starting the System

To begin operation,

1. Ensure the manual air supply valve is closed.
2. Plug in the electronic tachometer shown in Figure 3-2
3. Plug in the Aalborg flow-control valve pictured in Figure 3-4. If it is not already closed, this valve will close when plugged in. A green light on top of the valve body will be lit when the valve is fully closed.
4. Slowly and completely open the manual air supply valve.
5. Start ControlDesk, and load experiment *mbcntrl_a* found in directory *C:\Users\Robert\Alex_New*. All instrument panels will flash as they are activated. The panel *layout_start* shown in Figure 7-1 should be displayed in ControlDesk. If it isn't, make *layout_start* the active panel.
6. Start animation mode.
7. Select full-screen viewing. At this point, the screen should look identical to Figure 7-1.

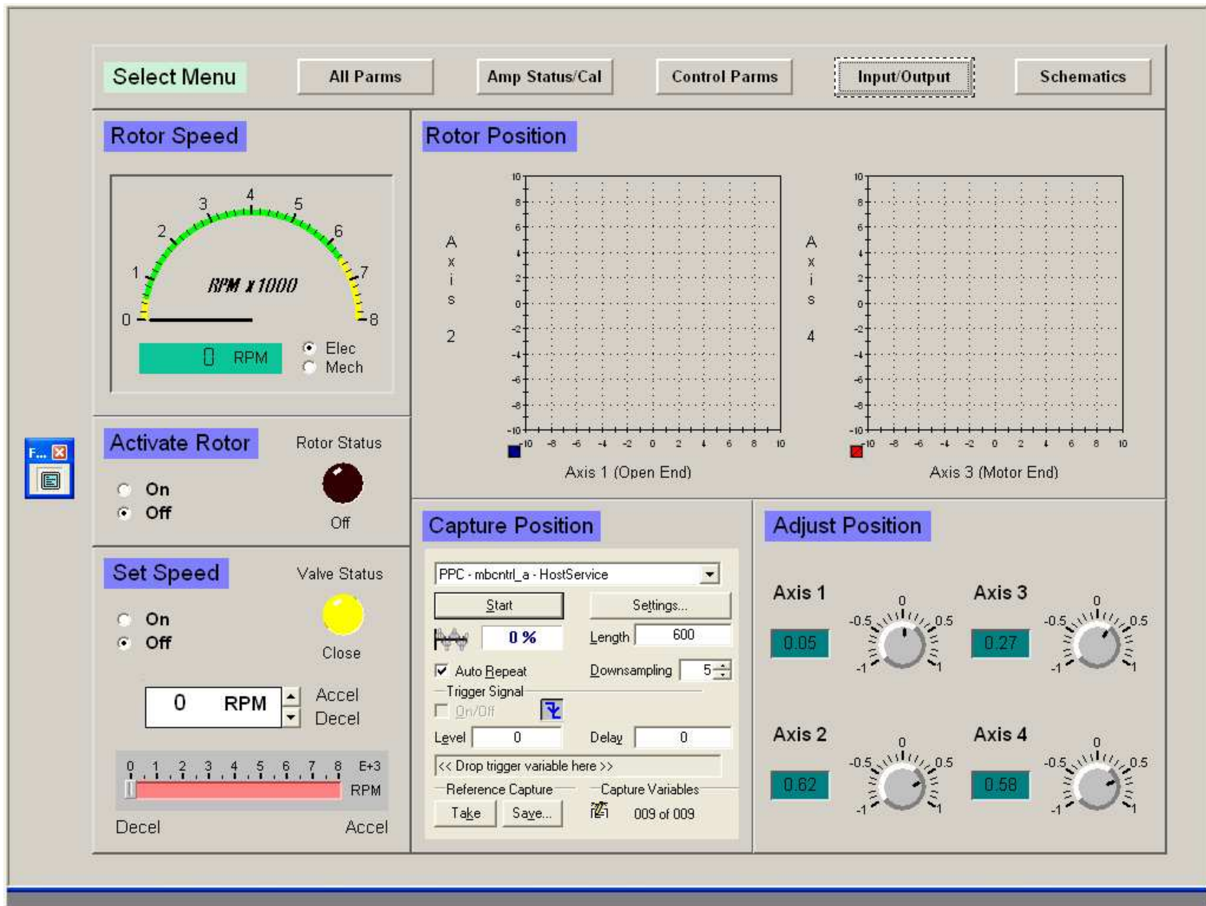

Figure 7-1 layout_start Instrument Panel

7.2 Activating the Bearings

To activate the magnetic bearings,

1. Click the *On* radio button in the *Activate Rotor* section of *layout_start*. The *Rotor Status* light will turn green, and the status message will change to *On*.
2. Turn on the two amplifiers pictured in Figure 2-1.
3. Turn on the two power supplies also shown in Figure 2-1. A “clunk” sound may be heard from the bearings when the hardware is powered on.
4. Lightly spin the rotor/flywheel by hand. If there is drag, use the knobs in *Adjust Position* to eliminate the drag.

To view the current position of the rotor as located between the magnets, enter a data capture *Length* of 5 (seconds) or less in the *Capture Position* dialog. Then, press the *Start* button in *Capture Position*, and the rotor's position will be plotted on the two graphs contained in section *Rotor Position*.

Each axis, *Axis 1*, *Axis 2*, *Axis 3* and *Axis 4*, refers to a pair of magnets, and the orientation of each axis is shown on the *Schematics* layout. To view these axis definitions, press the *Schematics* button located at the top, right corner of *layout_start*. The *Schematics* layout shown in Figure 7-2 will then be displayed.

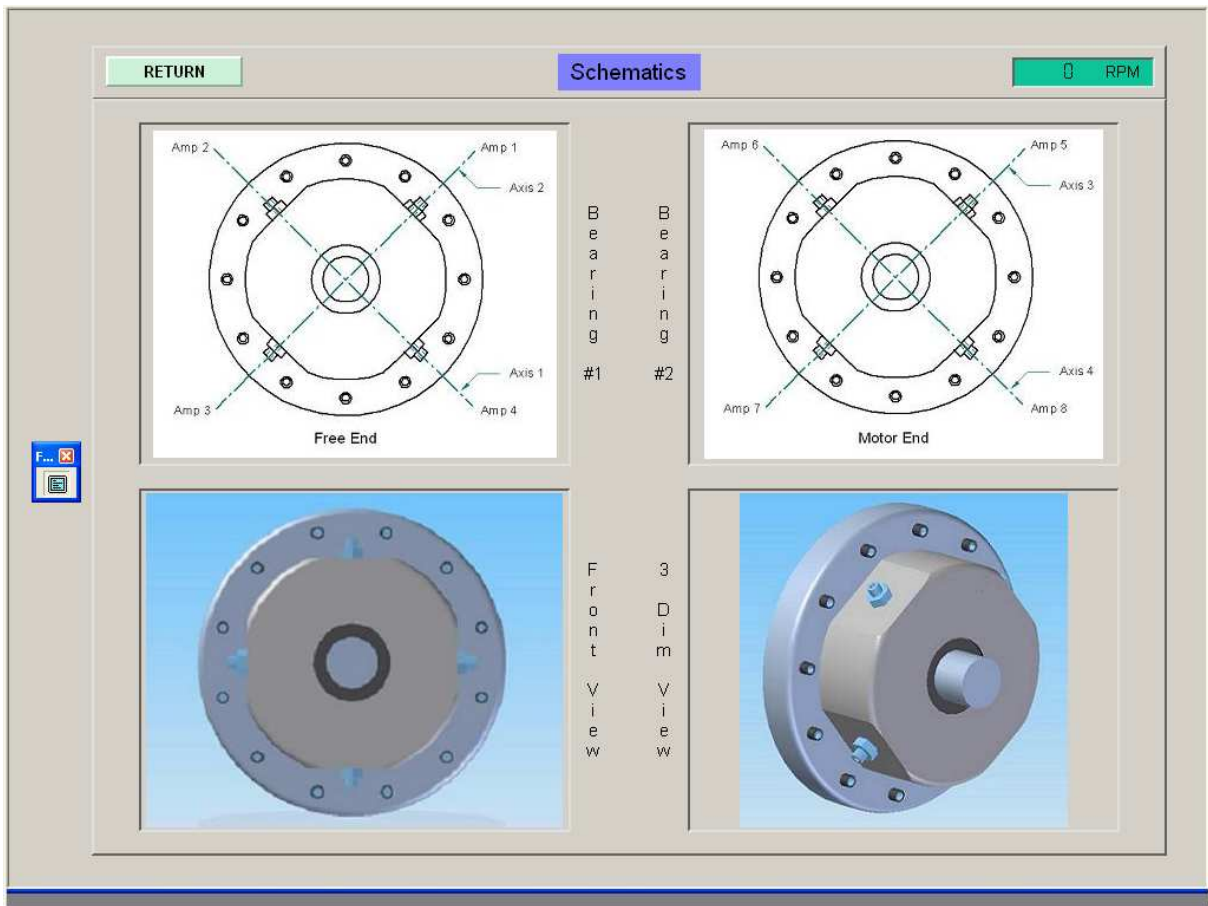

Figure 7-2 Schematics Instrument Panel

To return to *layout_start*, press the bluish-green *Return* button at the top, left corner of the *Schematics* panel. For future reference, every panel except *layout_start* contains a *Return* button. Pressing this button will always return you to *layout_start*.

7.3 Setting or Changing Speeds

Once the rotor is properly positioned, it can be spun up to speed. To do so from *layout_start*,

1. Choose either *Elec* or *Mech* in section *Rotor Speed*. The electronic tachometer pictured in Figure 3-2 will report the rotor's speed if radio button *Elec* is chosen. If *Mech* is selected, rotor speed will be measured by the mechanical tachometer shown in Figure 2-1. It is recommended that *Elec* be used.
2. Click the *On* radio button under *Set Speed* to activate automatic speed control. The *Valve Status* light will turn dark blue, and the status message will change to *Hold*.
3. Enter the desired rpm in *Set Speed* by 1) typing a number into the *RPM* numeric input box, 2) scrolling to the set point using the up and down arrows located next to the *RPM* box or 3) dragging the *RPM* slider to the desired speed. The *RPM* box and slider will always display the same rpm regardless of which instrument was used to set the speed. How these instruments were synchronized will be discussed at the end of Chapter 7.3. Compressed air will begin flowing once the speed is set, and the rotor will begin turning when there is sufficient air flow to overcome the start up inertia discussed in Chapter 4.7.
4. Wait for the rotor to reach the desired speed, or change to another set point as outlined in Step 3. The *layout_start* dialog will look as it does in Figure 7-3 for a desired speed of 2400 rpm. A graph of rotor rpm versus time may be viewed by pressing the *Input/Output* button at the top of *layout_start*. This velocity plot is contained in section *Plotter Out* of the *Input/Output* layout shown in Figure 7-4. This layout will be discussed in more detail later in Chapter 7.

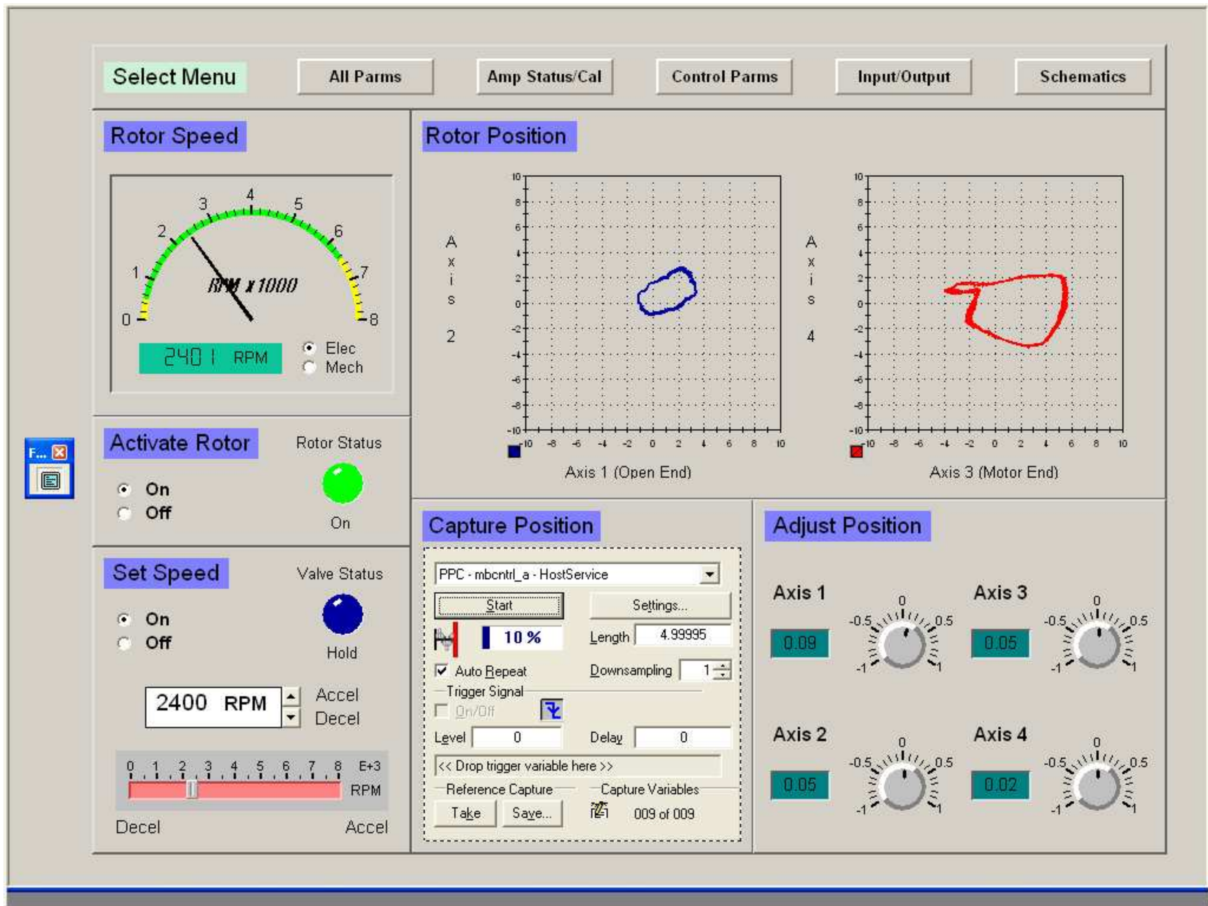

Figure 7-3 Setting Rotor Speed

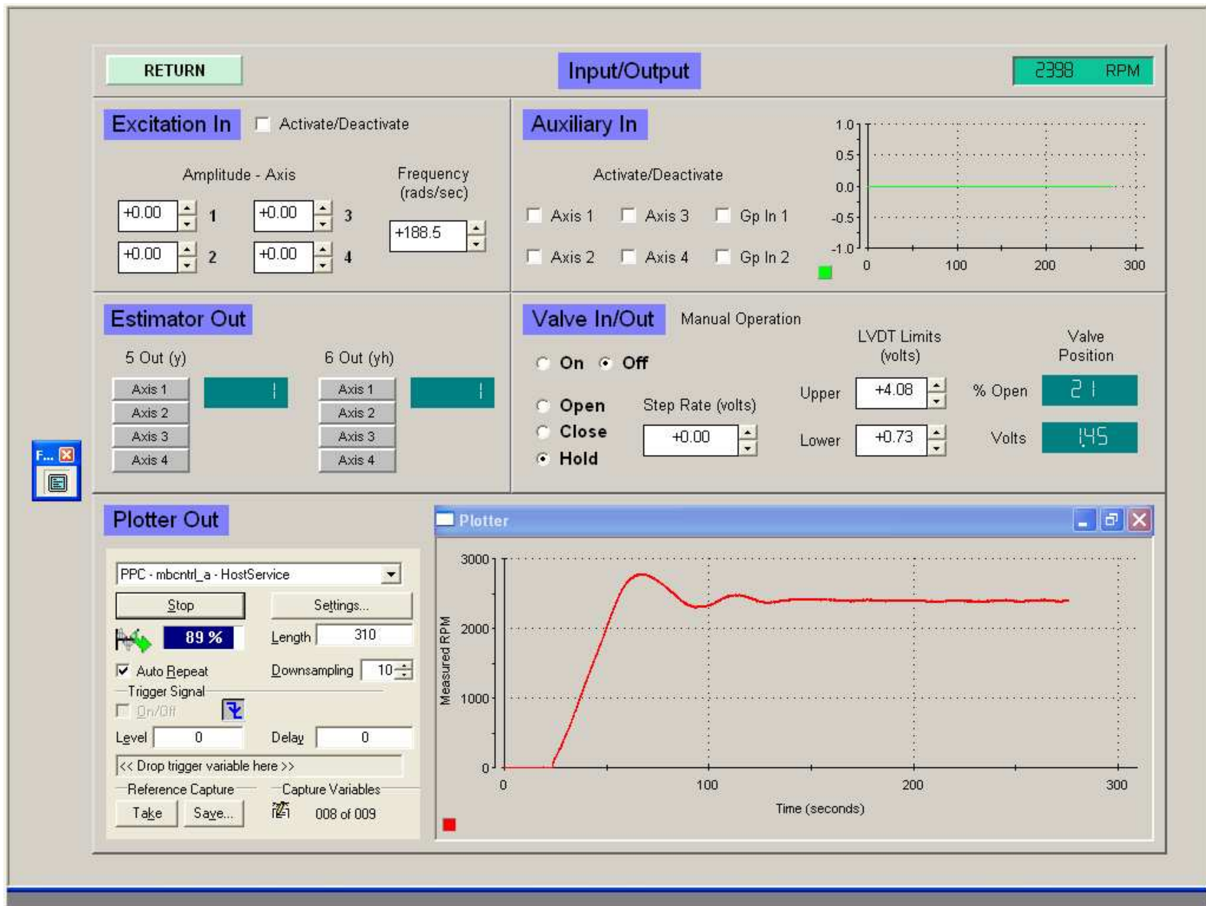

Figure 7-4 Viewing Velocity Plot

Synchronizing the *RPM* box and slider was accomplished through a special program written in the Python interpretive language. ControlDesk provides a Python-based programming facility for the user to extend the functionality of layouts and instruments. This facility was used to keep the aforementioned instruments in sync. If a Python program is added to extend the operation of a layout, this program is automatically given the same name as the layout prepended with an underscore. Many lines of Python code were written to integrate the different layouts and to add safety and convenience features to rotor control. Some of these special features will be noted during the discussion of the remaining layouts. Complete listings of all Python programs are provided in Appendix C. These programs are identified by a .py extension.

7.4 Stopping the Rotor and System

To stop the rotor,

1. Click the *Off* radio button under *Set Speed* on *layout_start* (shown in Figures 7-1 and 7-3). The rotor will slowly coast to a stop as the supply of compressed air is gradually shut off by the flow-control valve.
2. Once the rotor COMPLETELY stops spinning, click the *Off* radio button under *Activate Rotor*. If you attempt to turn off or deactivate the rotor while it is spinning, a warning is displayed, and the rotor remains activated. Damage to the bearings could result if they are accidently turned off while the rotor is moving. Shown in Figure 7-5 is the warning message, and it reads "The rotor cannot be deactivated unless it is at rest." The Python programming facility of ControlDesk was used to implement this safety feature.
3. Return to the ControlDesk development environment by clicking on the icon immediately to the left of the *Activate Rotor* section. Refer to Figure 7-5 at the end of these steps for the location of the icon.
4. Exit animation mode.
5. Close the manual air supply valve. This is very important for safety reasons, so do not forget to close this valve. If you are done for the day, also complete Steps 6 through 9.
6. Turn off the amplifiers.
7. Turn off the power supplies.
8. Unplug the electronic tachometer.
9. Unplug the Aalborg flow-control valve.

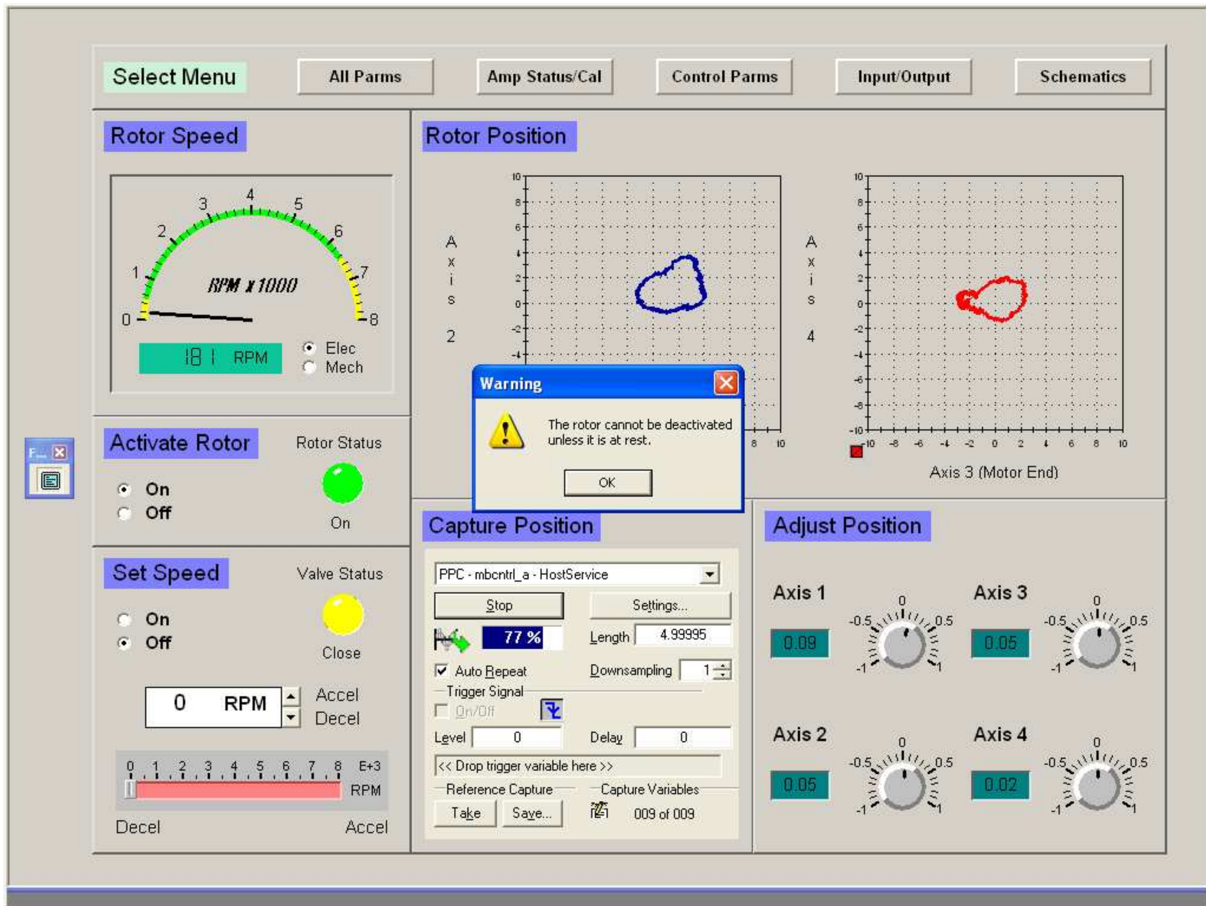

Figure 7-5 Stopping the Rotor

Thus far, three of the six instrument panels or layouts have been discussed within the context of rotor operation. An overview will now be provided of the three remaining layouts, *All Parms*, *Amp Status/Cal* and *Control Parms*. These panels are used to monitor and tune the magnetic-bearing system, and each panel is accessed by pressing the named button at the top of *layout_start*. Additional features of the *Input/Output* layout shown in Figure 7-4 will also be covered in the next sections.

7.5 All Parms

The *All Parms* layout was pictured earlier in Figure 2-6, and it is the original instrument panel created to control the magnetic bearings. *All Parms* was included with the new series of

panels to provide a reference to earlier work. While the original instrument panel is still functional, it should NOT be used to operate the rotor. *All Parms* does not include automatic speed control or any of the special features written for the new layouts. One such feature is that which prevents accidental deactivation of a spinning rotor.

The extensive functionality provided by *All Parms* has been logically redistributed among the five new instrument panels. If additional functions are needed in the future, they can be included on a new layout. Access to the new panel would then be easily accomplished by adding another button to the top of *layout_start*. The ease by which the new system can be extended was lacking in the single panel of *All Parms*. Rotor operations would have been greatly complicated by adding even more instruments onto this single layout.

7.6 Amplifier Status and Calibration

Pictured in Figure 7-6 is layout *Amp Status/Cal*, and it is used to monitor and adjust the eight power amplifiers. This instrument panel is activated by pressing the *Amp Status/Cal* button on *layout_start*.

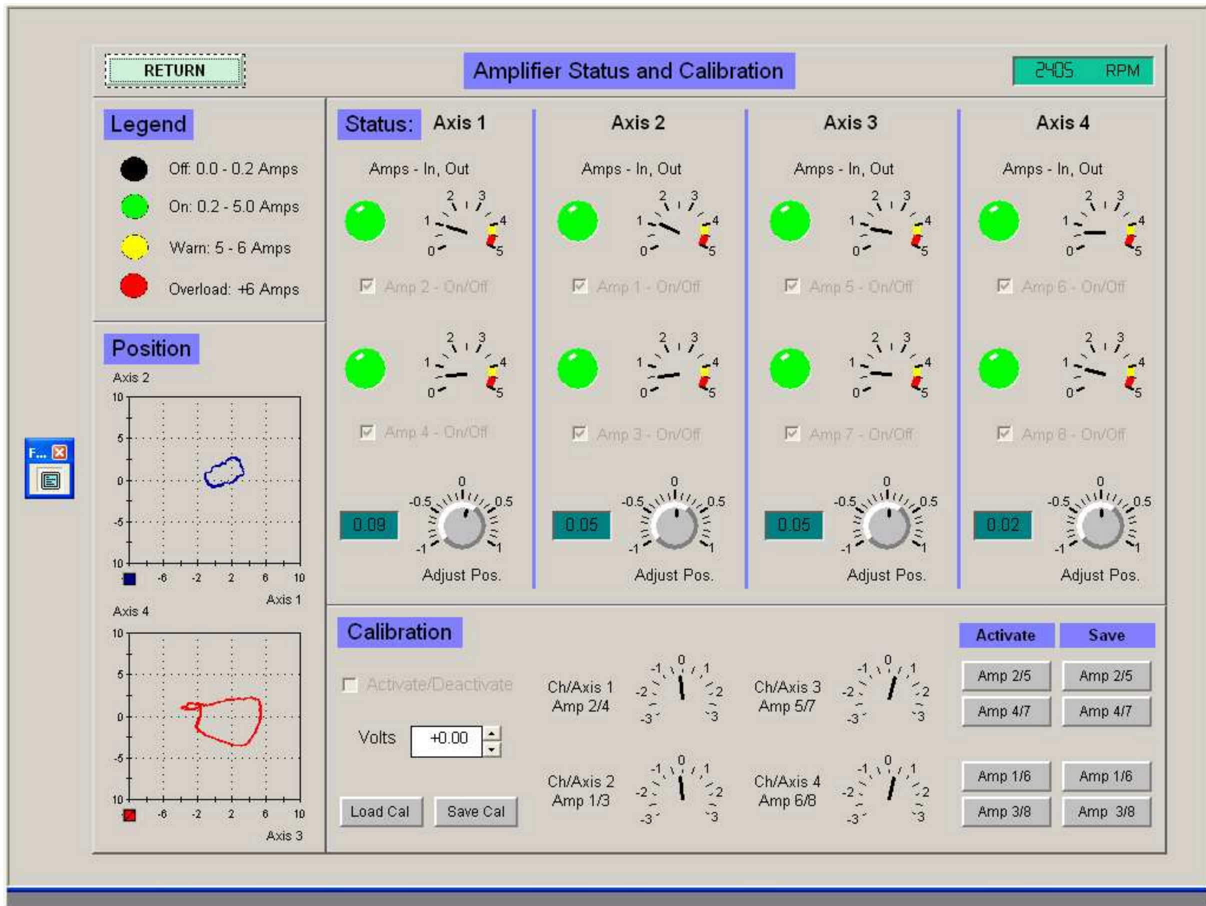

Figure 7-6 Amp Status/Cal Instrument Panel

There are four columns labeled *Axis 1*, *Axis 2*, *Axis 3* and *Axis 4* shown in the *Status* section of this layout. Each column corresponds to one axis of a magnetic bearing. The location and orientation of each axis and the two amplifiers assigned to each are shown on the *Schematics* panel pictured in Figure 7-2. Each light or LED and gauge pair within a column displays the current read from and the current written to the named amplifier. The four knobs labeled *Adjust Pos.* and the *Position* plots are duplicates of the same instruments found on *layout_start*. These instruments are included on *Amp Status/Cal* to allow manual positioning of the rotor while observing changes in amplifier current. Awareness of these changes can help diagnose bearing problems if current overloads occur during rotor adjustment or operation. Note that any change in position made from either *layout_start* or *Amp Status/Cal* is reflected in both layouts.

Calibration was developed to eliminate rotor vibrations that occurred at certain speeds and to resolve collocation issues [5]. Collocation or arrangement issues arise since the proximity probes sensing rotor position are not in the same location as the magnetic actuators that position the rotor. Refer to [5] for a discussion of collocation and calibration. Calibration can also be better understood by studying the Python program *_amp adjust.py* given in Appendix C. Only a stepwise procedure for calibration will be outlined in the following discussion.

Calibration can only be performed if the rotor is at rest and both automatic and manual speed control are off. If these conditions are not met, the *Activate/Deactivate* checkbox under *Calibration* will be automatically dimmed and not selectable as it is in Figure 7-6. The individual amplifier *On/Off* checkboxes in the *Status* section of Figure 7-6 will also be dimmed if the rotor is spinning. Deactivating these checkboxes is another safety feature implemented with the programming facility of ControlDesk. Calibration will vary the amplifiers on and off, and it has been noted that turning off an amplifier while the rotor is spinning could damage the magnetic bearings.

To calibrate a system that is at rest,

1. Click the *Activate/Deactivate* checkbox. This will turn off all amplifiers.
2. Adjust the voltage in the *Volts* input box, and note changes to the values displayed on the gauges for each axis (also called a channel).
3. Press the *Amp 2/5* button under the *Activate* heading in the *Calibration* section. The checkboxes for amplifiers 2 and 5 will be selected and these amplifiers will be turned on.
4. Press the *Amp 2/5* button under the *Save* heading in the *Calibration* section. This operation will store calibration data in variables defined in the Python program *_amp adjust.py*. Appendix C contains the listings for all Python programs.
5. Repeat steps 3 and 4 for *Amp 4/7*, *Amp 1/6* and *Amp 3/8*.
6. Press the *Save Cal* button to save the new calibration values to a file. Saving calibration data to a file is accomplished with a standard Windows save file dialog as

shown in Figure 7-7. Once the data are stored in a file, the *Adjust Pos* knobs are updated with the new calibration values.

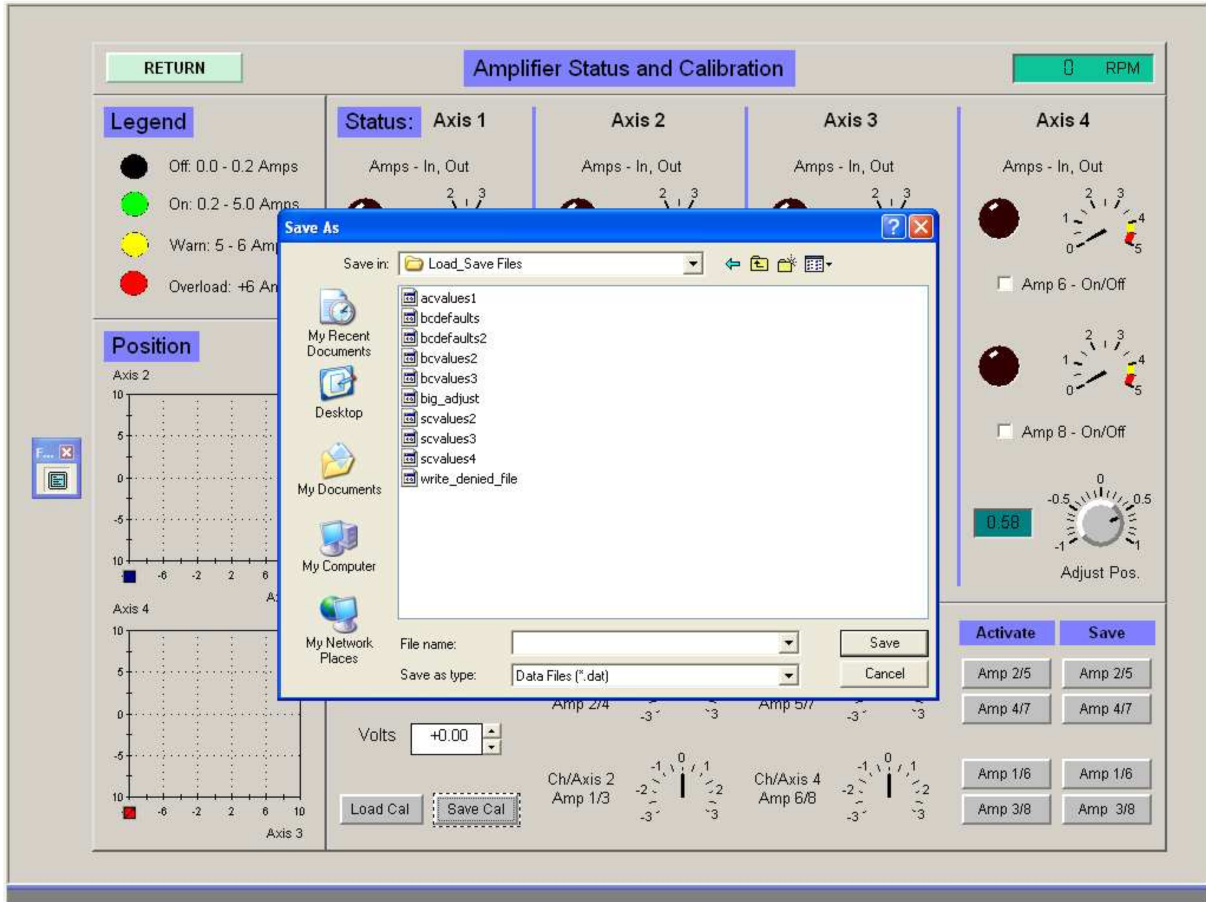

Figure 7-7 Saving Calibration Data

If you want to use a previously saved calibration, press *Load Cal*. A standard Windows open file dialog will then be displayed from which any saved calibration can be selected. The save and load functions using Windows dialogs were also implemented with the programming capabilities of ControlDesk.

7.7 Control Parameters

All of the variables needed to adjust and tune the bearing and speed controllers are available on a separate layout. This instrument panel is shown in Figure 7-8, and it is displayed by

pressing the *Control Parms* button located at the top of *layout_start*. Note that *Control Parms* also contains the adjustable settings for the adaptive controller used for disturbance rejection.

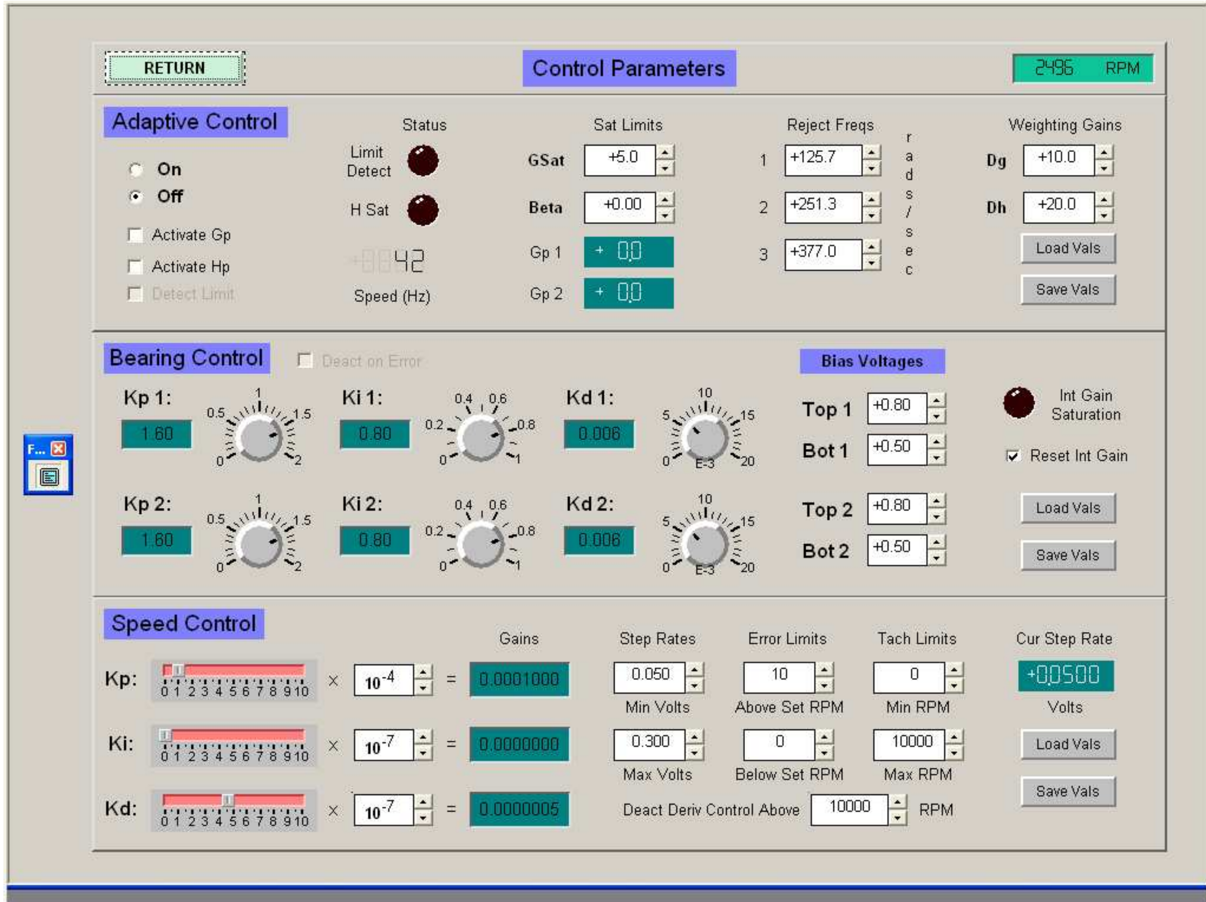

Figure 7-8 Control Parms Instrument Panel

Since the development of the automatic speed controller did not involve adaptive disturbance rejection (ADR), the *Off* radio button in section *Adaptive Control* was always selected. ADR was used in prior studies to compensate for disturbances (e.g. vibration of the base supporting the magnetic bearings) acting on the system. The values of all *Adaptive Control* parameters shown in Figure 7-8 are those developed by Matras [5]. Select the *On* radio button to activate ADR, and then adjust the various values using [5] as a reference according to your needs.

The *Load Vals* and *Save Vals* buttons found in all three sections of *Control Parms* work exactly like the *Load Cal* and *Save Cal* buttons located on the *Amp Status/Cal* layout. All of the adjustable parameters displayed in an individual section of the *Control Parms* layout can be saved to a file by pressing *Save Vals*. *Load Vals* will set the instruments to the values read from a file saved previously with *Save Vals*. Both the save and load functions were implemented using Windows file dialogs.

Instruments for displaying and adjusting the gains of the two magnetic-bearing controllers are given in section *Bearing Control*. All variable names containing the number 1 (e.g. *Kp1* and *Top1*) apply to the bearing supporting the free end of the rotor. This end contains the hardware drive for the mechanical tachometer. Variable names ending with the number 2 (e.g. *Kp2* and *Top2*) pertain to the bearing nearest the air turbine. The naming convention for the bearings was shown on the *Schematics* layout pictured in Figure 7-2. Since there is a controller per bearing, gains for one bearing may be tailored independently of those for the other bearing. However, the gains used for the development of the automatic speed controller are those determined by Matras and shown is Figure 7-8. *Bias Voltages* are used to further adjust rotor position, and the bias values shown are those developed previously and used throughout speed controller testing.

The *Deact on Error* checkbox has been permanently dimmed (i.e. it cannot be selected). This feature automatically shuts off the bearings if an error (e.g. current overload) is detected. *Deact on Error* was included in the new instrument panels to maintain functional consistency with the original *All Parms* layout. Since automatic deactivation of the bearings could result in more damage than a current overload, this feature cannot for now be selected.

Nearly all of the *Speed Control* variables and their values were discussed in Chapter 6. It is worthwhile though to review a few items. The gains *Kp*, *Ki* and *Kd* can be infinitely adjusted using a combination of instruments. Remember, the value of *Ki* displayed under *Gains* should be zero! Each gain is determined by multiplying the number from a slider by a power of 10 from a numerical input box. This power is changed with the arrows located next to the box. The use of two instruments instead of one such as a knob provides for the selection of a much

greater range of values. This almost unlimited range reduced development time for the speed controller since gains could be quickly changed without the need to constantly redesign an instrument to include unforeseen values.

Other variables worth noting are the *Error Limits*, *Above Set RPM* and *Below Set RPM*. These values specify the actuator dead band covered in Chapter 6.1and used to extend the life of the flow-control valve. The values of *Tach Limits, Min RPM* and *Max RPM* must match the values entered at the front panel of the electronic tachometer when it is configured. Tachometer configuration is covered in [24]. If these values do not match, the measured rpm of the rotor will be in error. Finally, *Cur Step Rate* reports the current stepping rate of the valve. This value is interesting to watch since the valve usually steps at the minimum rate specified by *Step Rates*, *Min Volts*.

7.8 Input/Output

The *Input/Output* instrument panel was pictured previously in Figure 7-4. This panel was introduced in Chapter 7.3 when the step-wise procedures were given for setting or changing the speed of the rotor. All operations available from the *Input/Output* layout will now be covered.

The functions performed by *Excitation In* and *Auxiliary In* are linked together. Neither type of input is active unless the *Activate/Deactivate* checkbox under *Excitation In* is selected. *Excitation In* superimposes a sinusoidal signal onto the bearing control voltages. Each bearing axis seen in Figure 7-2 can be excited separately with a sine wave by selecting an amplitude for the wave from the input boxes labeled *1, 2, 3* and *4* under heading *Amplitude - Axis*. The frequency for all waves is common and is chosen from the input box below *Frequency (rads/sec)*. These sinusoidal signals are used to test ADR. In addition, an auxiliary signal from an external source can be added to the sine waves. This signal would be input into the system through the A/D converter. To apply an external signal to a bearing axis or to one of the adaptive control gains, select the check boxes contained within the *Auxiliary In* section. These

boxes are labeled *Axis 1*, *Axis 2*, *Axis 3* and *Axis 4* and *Gp In 1* and *Gp In 2*. The auxiliary signal will be displayed on the graph to the right of the check boxes.

Estimator Out provides the ability to output on an axis basis the measured *5 Out (y)* and estimated *6 Out (yh)* positions of the rotor. Pressing one of the buttons labeled *Axis 1, Axis 2 Axis 3 or Axis 4* displays the selected axis number and routes the position signal for that axis to the D/A converter.

Select the *On* radio button in section *Valve In/Out* to manually operate the Aalborg flow-control valve. Manual operation is used for testing purposes and for calibrating the LVDT which measures valve position. Automatic speed control is turned off when in manual mode. Direction of valve travel or valve position is then determined by choosing one of the radio buttons *Open*, *Close* or *Hold*. The speed at which the valve opens or closes is specified in the *Step Rate (volts)* input box. None of the automatic speed control settings contained on layout *Control Parms* apply when the valve is operated manually. The valve will step at whatever rate is given in the *Step Rate (volts)* input box.

Calibration of the LVDT pictured in Figure 3-4 is performed with the *LVDT Limits (volts)*, *Upper* and *Lower* numerical input boxes. The *Upper* limit is set to the voltage value shown in the *Valve Position*, *Volts* display box when the valve is fully opened as indicated by the red light on top of the Aalborg flow-control valve. The *Lower* limit is set to the voltage displayed when the valve is completely closed as indicated by the green light on the valve. These two voltages are used to calculate the valve's current position which is then displayed in the *% Open* display box. When calibration is complete, select the *Off* radio button to exit manual valve mode and to reactivate automatic speed control. The exclusion of one mode of valve operation from the other is accomplished with switch blocks seen at the upper right corner of the speed controller block diagram shown in Figure 6-1.

The *Plotter* displayed in Figure 7-4 graphs rotor velocity as a function of time. However, a *Plotter* instrument can be easily and dynamically configured (i.e. while the rotor is spinning) to graph practically any variable in a Simulink block diagram. The *Plotter* on the *Input/Output*

layout was used during the development of the speed controller to display values of numerous variables providing visual proof of an idea's success or failure.

Plotter characteristics (e.g. axis labels) are customized using the *Settings* ... button found on the *CaptureSettings* dialog positioned to the left of the velocity plot. Any number of *Plotters* and *CaptureSettings* dialogs can be included in a series of layouts. Rotor positions are plotted on *layout_start* (Figures 7-1, 7-3 and 7-5), and rotor speed is graphed on the *Input/Output* layout. Unfortunately, the *Length* parameter specified in the individual *CaptureSettings* dialogs is shared among all plots. *Length* in seconds is the refresh rate at which all graphs are redrawn. The value of *Length* used by ControlDesk is that from the *CaptureSettings* dialog used to start data capture. With just a single refresh rate, some graphs may be redrawn more frequently than desired.

Chapter 8 - Conclusions and Future Work

The simulated system provided direction for the design of the actual speed controller. Simulations showed that a maximum stepping rate of 0.3 volt and a proportional gain K_p of 0.0001 would provide accurate speed control. These values were used in the final controller configuration. In addition, the model demonstrated the benefits of derivative action, and derivative control is part of the actual speed regulator. Simulated speed responses contained no steady-state errors and none exist in the real system. Thus, integral control was not necessary for regulating the speed of the rotor. The need for an actuator dead band was also determined from the model. While never shown, graphs of valve motion versus rotor speed from any simulation showed that the flow-control valve continually oscillates as it tries to maintain zero deviation from the set point. The Aalborg valve behaves identically to the simulated one, and to limit oscillations and increase valve life, a configurable dead band was included in the speed controller.

A gain-scheduled controller was used in the simulations to maintain a consistent response for different set points. It was seen from velocity data that actual rotor response varies depending on the desired rpm. The magnitudes of the transient speed oscillations are more pronounced at lower speeds. These magnitudes could be reduced by altering the controller gains K_p and K_d depending on set point. Automatic gain selection was not included since maintaining a steady-state speed is of the most importance.

The simulated system did not provide a realistic estimate of the derivative gain K_d. This deficiency may be the result of the procedure used for estimating the gains since the Good Gain method applied to speed responses from the actual system also overestimated K_d. The minimum derivative gain found from simulations using modified Good Gain was 1 x 10^{-5}. The actual value used for the single controller configuration was 5 x 10^{-7}. This value is two orders of magnitude less than the estimated K_d.

It is doubtful whether any of the methods mentioned in the introduction would have provided close estimates for the derivative gain. All methods except possibly Relay do not directly

include the actuators in determining gains. For the magnetic-bearing system, the electrical and mechanical characteristics of the Aalborg valve set K_d. When a K_d of 1 x 10^{-5} was used with this valve, it provided way too much anticipatory action or damping. The flow-control valve would begin closing too soon, and rotor speed could not even reach the set point. System testing established the actual value of K_d. The modified Good Gain method used in the simulations did however provide a starting point for determining the derivative gain for the actual system.

A formal study of stability was not conducted in the development of the speed controller. It was determined through simulations and system testing that stepping rates greater than 0.3 volt produced temporary instability in the bearings. The Routh or root-locus methods could be used to formally establish stability criteria. This task would be complicated since the magnetic-bearing system was only partially modeled with a transfer function. Valve action is responsible for system stability, and the valve was represented as a lookup table using an embedded MATLAB program. Automotive fuel-injection systems are modeled and implemented in a similar manner. The methods used to establish stability for these automotive systems could possibly be applied to the speed controller. It could be however, that in industry, stability is determined experimentally.

Rotor response would be improved by decreasing transient speed oscillations. These transients are most pronounced with dramatic changes in set point. The severity of the oscillations could be reduced by using valve position and flow rate data or by using a gain scheduled controller. With the addition of the LVDT, the valve's position is always known. From system testing, it is also known what quantity of air as a function of valve position is required to maintain a certain rpm. Using this data, if a big change in set point was required, the valve could be repositioned and held at the flow rate needed for the desired speed. Control would be reactivated once the new set point was reached. Oscillations in rpm would be reduced since flow rates would never become too little or too great at any one time. Very low stepping rates achievable with a gain scheduled controller could also improve transient response. However, the current valve will not step slower than 0.05 volt.

The use of adaptive disturbance rejection (ADR) techniques requires that rotor speed be precisely known. The new speed measurement and control system should facilitate ADR studies, and these would provide a good test of the new system's capabilities. Other future work would involve incorporating automatic speed control with flywheel health and containment strategies. If defects were detected in a flywheel, the speed of the rotor could be automatically reduced to an rpm where the flywheel could be safely operated, or the flywheel could be completely stopped.

References

[1] T. Hikihara, H. Adachi, S. Ohashi, Y. Hirane and Y. Ueda, "Levitation Drift of Flywheel and HTSC Bearing System Caused by Mechanical Resonance," *Physica C*, 1997, pp. 34-40.

[2] T. Hikihara, H. Adachi, F. Moon, and Y. Ueda, "Dynamical Behavior of Flywheel Rotor Suspended by Hysteretic Force of HTSC Magnetic Bearing," *Journal of Sound and Vibration,* 1997, pp 871-887.

[3] M. Siebert, B Ebihara, R. Jansen, R.L. Fusaro, W. Morales, A. Kascak and A. Kenney, "A Passive Magnetic Bearing Flywheel," *NASA Technical Publication*, February, 2002.

[4] W. Morales, R. Fusaro and A. Kascak, "Permanent Magnetic Bearing for Spacecraft Applications," *NASA Technical Publication*, January, 2008.

[5] A.L. Matras, "Applied Adaptive Disturbance Rejection Using Output Redefinition on Magnetic Bearings," *Ph. D Thesis*, University of Colorado, Boulder, CO, 2005.

[6] T. Kealy and A. O'Dwyer, "Comparison of Open and Closed Loop Process Identification Techniques in the Time Domain," *Dublin Institute of Technology Conference Paper*, September, 2002.

[7] K. Gajam, Z. Zouaoui, P. Shaw and Z. Chen, "Design of an Adaptive Self-Tuning Smith Predictor for a Time Varying Water Treatment Process," *International Symposium on Advanced Control of Chemical Processes*, July, 2009.

[8] M. Serra i Pratt, "Energy Optimisation and Controllability in Complex Distillation Columns," *Universitat Politècnica de Catalunya, Department of Chemical Engineering*, Barcelona, June, 2000.

[9] W.L. Bialkowski, "Mill Audits Can Cut Costs by Reducing Control Loop Variability", *Pulp and Paper Magazine*, December, 1998.

[10] M. Nikolaou and P. Misra, “Linear Control of Nonlinear Processes: Recent Developments and Future Directions”, University of Houston, Chemical Engineering Dept., *CEPAC*, 2001.

[11] K. Brazauskas and D. Levisauskas, “Adaptive Transfer Function-Based Control of Nonlinear Process. Case study: Control of Temperature in Industrial Methane Tank,” *International Journal of Adaptive Control and Signal Processing*, Volume 21, Issue 10, December 2007, pp. 871-884.

[12] Y.H. Peng, “On the Performance Enhancement of Self-Tuning Adaptive Control for Time-Varying Machining Processes,” *International Journal of Advanced Manufacturing Technology*, August 2004, pp. 395-403.

[13] A.G. Ulsoy, Y. Koren and L.K. Lauderbaugh, “Variable Gain Adaptive Control Systems for Machine Tools,” *Progress Report for NSF Grant MEAM 811269*, October, 1983.

[14] C. Carnevale, A. Piazzi and A. Visioli, “Noncausal Open-Loop Control with Combined System Identification and PID Controller Tuning,” *International Conference on Control*, September, 2008.

[15] J. Bennett, A. Bhasin, J. Grant and W.C. Lim, “PID Tuning via Classical Methods,” *The Michigan Chemical Process Dynamics and Controls Open Text Book*, October, 2007.

[16] J. Zhong, “PID Controller Tuning: A Short Tutorial,” *Purdue University, Mechanical Engineering Department*, Spring 2006.

[17] W.J. Palm III, *System Dynamics*, McGraw-Hill, New York, 2005, pp. 783-785.

[18] D. I. Wilson, “Relay-based PID Tuning,” *Automation and Control*, Feb/March 2005.

[19] *http://techteach.no/publications/books/dynamics_and_control/tuning_pid_controller.pdf*

[20] V. VanDoren, “Relay Method Automates PID Loop Tuning,” *Control Engineering*, September 1, 2009.

[21] K.L. Barber, “Health Monitoring for Flywheel Rotors Supported by Active Magnetic Bearings,” *Master’s Thesis*, Auburn University, Auburn, AL, 2006.

[22] http://www.dspace.com

[23] J. P. Holman, *Experimental Methods for Engineers*, McGraw-Hill, New York, 1978, pp. 231-233.

[24] *DT-5TX/DT-5TS Digital Panel Mount Tachometer Instruction Manual*, NIDEC-Shimpo America Corporation, Itasca, IL, 2000.

[25] *Operating Manual SMV Stepping Motor Valve*, Aalborg Instruments and Controls, Orangeburg, NY, 2001.

[26] W.J. Palm III, *System Dynamics*, McGraw-Hill, New York, 2005, page 613.

[27] MATLAB Help documentation.

[28] K. J. Åström and B. Wittenmark, *Adaptive Control, Second Edition*, Addison-Wesley Publishing Company, Inc., New York, 1995, p. 19.

[29] W.J. Rugh and J.S. Shamma, “Research on Gain Scheduling,” *Automatica 36*, 2000, pp. 1401-1425.

[30] George T. Flowers and Stephen G. Ryan, “Development of a Set of Equations for Incorporating Disk Flexibility Effects in Rotordynamical Analyses,” *ASME 1991 International gas Turbine and Aeroengine Congress and Exposition, June 3, 1991, pp.* V005T14A008-V005T14A008.

[31] Y. Gowayed, Y., G. Flowers, F. Hady, and J. Trudell, “Optimal design of multi-direction composite flywheel rotors,” *Polymer Composites*, Vol. 23, No. 3, 2002, pp. 433-441.

[32] Valeta Carol Chancey, George T. Flowers, and Candice L. Howard, “A Harmonic Wavelet Approach for Extracting Transient Patterns from Measured Rotor Vibration Data,” *ASME*

Turbo Expo 2001: Power for Land, Sea, and Air, June 4-7, 2001, pp. V004T03A008-V004T03A008.

[33] Robert N. Dean, George T. Flowers, A. Scotte Hodel, Grant Roth, Simon Castro, Ran Zhou, Alfonso Moreira, Anwar Ahmed, Rifki Rifki, Brian E. Grantham, David Bittle, and J. Brunsch, "On the Degradation of MEMS Gyroscope Performance in the Presence of High Power Acoustic Noise," *IEEE International Symposium on Industrial Electronics (ISIE2007)*, June 4-7, 2007, pp. 1435-1440.

[34] Alex Matras, George Flowers, Robert Fuentes, Mark Balas, and Jerry Fausz, "Suppression of Persistent Rotor Vibrations Using Adaptive Techniques," *ASME Journal of Vibration and Acoustics,* Vol. 128, No. 6, December, 2006, pp. 682-689.

[35] Klaus H. Hornig and George T. Flowers, "Parameter Characterization of the Bouc/Wen Mechanical Hysteresis Model for Sandwich Composite Materials using Real Coded Genetic Algorithms," *International Journal of Acoustics and Vibration*, Vol. 10, No. 2, 2005, pp. 73-81.

[36] Roland Horvath, George T. Flowers, and Jerry Fausz, "Passive Balancing of Rotor Systems Using Pendulum Balancers," *ASME Journal of Vibration and Acoustics*, August 2008, pp. 041011.

[37] F.S. Wu, and G.T. Flowers, "Disk/Shaft Vibration Induced by Bearing Clearance Effects: Analysis and Experiment," *ASME Journal of Vibration and Acoustics*, Vol. 118, No. 2, 1996, pp. 204-208.

[38] George T. Flowers, Gyorgy Szasz, Victor S. Trent, and Michael E. Greene, "A Study of Integrally Augmented State Feedback Control for an Active Magnetic Bearing Supported Rotor System," *ASME 1997 International Gas Turbine and Aeroengine Congress and Exhibition*, June 2, 1997, pp. V004T14A041-V004T14A041.

Appendix A

MATLAB Utility Program

plot3DStepRates.m

```
%{

plot3DStepRates.m:  plots valve position as a function of time and stepping rate. Refer to
Figure 4-10.

%}

%{
 X = to right = time axis.
 Y = out of page = step rate axis = volts axis.
 Z = up = % open axis.
%}

% Stepping rate in volts.
rate = [ 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 ];

% The following slopes were determined using MATLAB's cftool operating on data captured
% with ControlDesk. Units of slope are (% open)/sec. Refer to Table 4-3.
slope = [ 0.0 0.2372 0.4195 0.6012 0.8827 1.146 1.472 1.898 2.246 2.685 3.089 3.569 ...
 3.991 4.460 4.935 5.408 ];

% Calculate the time required to fully open the valve at each step rate. Skip slope = 0 since it
% could result in division by 0 in the following loop.
for i = 1:length(slope)
 j = i + 1;
 if j > length(slope)
  break;
 end
 tOpen(i) = 100/slope(j);
 fprintf('\n -> At a step rate of %2.1f, it takes %4.1f seconds ', i/10, tOpen(i))
 fprintf('to fully open the valve. \n')
end

% Determine the maximum time for the plot.
tOpenMax = max(tOpen);
tOpenMax = ceil(tOpenMax);
```

```
% Generate time data in increments of 1 second.
clear x
x(1) = 0;
for i = 2:tOpenMax
  x(i) = i; % Note, x at 1 second is skipped.
end

% Set the step rates at each second. Step rates are constant at each time interval.
yVals = struct([]);
for i = 1:length(rate)
  clear yData;
  for j = 1:tOpenMax
    yData(j) = rate(i);
  end
  yVals(i).data = yData;
end

% Calculate % open at each second for each step rate.
zVals = struct([]);
for i = 1:length(slope)
  clear zData;
  for j = 1:tOpenMax
    zData(j) = x(j)*slope(i);
    if zData(j) > 100
      zData(j) = 100; % Do not exceed 100% open.
    end
    zVals(i).data = zData;
  end
end

% Generate the XZ planes for each step rate, and connect the planes with surfaces.
clear X
X = [x; x]; % Time.

for i = 1:length(rate)
  clear Y Z
  j = i + 1;
  if j > length(rate)
    break;
  end
  Y = [ yVals(i).data; yVals(j).data ]; % Stepping rate.
  Z = [ zVals(i).data; zVals(j).data ]; % Percent open.
  surf(X, Y, Z)
  hold on
```

```
end

shading flat
axis([0 max(x) 0 max(rate) 0 100])
set(gca,'XTick',[0 50 100 150 200 250 300 350 tOpenMax])
set(gca,'YTick',[0 0.1 0.3 0.5 0.7 0.9 1.1 1.3 1.5])

grid on
title('Valve % Open Lookup Table')
xlabel('Time (seconds)')
ylabel('Stepping Rate (volts)')
zlabel('Valve Position (% Open)')
```

Appendix B

Embedded MATLAB Functions

Aalborg Valve

```
function [pctNew,Q] = fcn(pct,ctrl,rMax,sTime)
%{
Model of the Aalborg flow-control valve.
%}

% The minimum stepping rate is fixed by the design of the valve. This rate was determined
% experimentally to be between 0.045 and 0.05 volts.
rMin = 0.05;

% The following slopes were determined using MATLAB's cftool operating on data captured
% with ControlDesk. Units of slope are (% open)/sec. Refer to Table 4-3.
slope = [ 0.2372 0.4195 0.6012 0.8827 1.146 1.472 1.898 2.246 2.685 3.089 3.569 ...
3.991 4.460 4.935 5.408 ];

% Decide if interpolation is required.
posFlag = -1;

% Preallocate variables or this function will not compile.
pctIncr = 0.0;
x1 = 0.0; x3 = 0.0;
y1 = 0.0; y2 = 0.0; y3 = 0.0;

% Determine whether to open or close the valve.
if ctrl < 0
  dir = -1; % Close.
elseif ctrl > 0
  dir = 1; % Open.
else
  dir = 0; % Hold.
end

% The control signal must be positive for interpolation.
ctrl = abs(ctrl);

if ctrl == 0
  pctIncr = 0;
  posFlag = 1;
```

```
end

% Do not exceed the maximum step rate set by the user. Do not step slower than possible
% with the valve.
if ctrl > rMax
  ctrl = rMax;
end

if ctrl < rMin
  ctrl = rMin;
end

% Determine the boundary curves for the interpolation. The following is a lookup table, and it
is % shown graphically in Figure 4-10.
if posFlag < 0
  if 0 < ctrl && ctrl <= 0.1
    x1 = 0; y1 = 0;
    x3 = 0.1; y3 = slope(1);
  elseif 0.1 < ctrl && ctrl <= 0.2
    x1 = 0.1; y1 = slope(1);
    x3 = 0.2; y3 = slope(2);
  elseif 0.2 < ctrl && ctrl <= 0.3
    x1 = 0.2; y1 = slope(2);
    x3 = 0.3; y3 = slope(3);
  elseif 0.3 < ctrl && ctrl <= 0.4
    x1 = 0.3; y1 = slope(3);
    x3 = 0.4; y3 = slope(4);
  elseif 0.4 < ctrl && ctrl <= 0.5
    x1 = 0.4; y1 = slope(4);
    x3 = 0.5; y3 = slope(5);
  elseif 0.5 < ctrl && ctrl <= 0.6
    x1 = 0.5; y1 = slope(5);
    x3 = 0.6; y3 = slope(6);
  elseif 0.6 < ctrl && ctrl <= 0.7
    x1 = 0.6; y1 = slope(6);
    x3 = 0.7; y3 = slope(7);
  elseif 0.7 < ctrl && ctrl <= 0.8
    x1 = 0.7; y1 = slope(7);
    x3 = 0.8; y3 = slope(8);
  elseif 0.8 < ctrl && ctrl <= 0.9
    x1 = 0.8; y1 = slope(8);
    x3 = 0.9; y3 = slope(9);
  elseif 0.9 < ctrl && ctrl <= 1.0
    x1 = 0.9; y1 = slope(9);
```

```
    x3 = 1.0; y3 = slope(10);
  elseif 1.0 < ctrl && ctrl <= 1.1
    x1 = 1.0; y1 = slope(10);
    x3 = 1.1; y3 = slope(11);
  elseif 1.1 < ctrl && ctrl <= 1.2
    x1 = 1.1; y1 = slope(11);
    x3 = 1.2; y3 = slope(12);
  elseif 1.2 < ctrl && ctrl <= 1.3
    x1 = 1.2; y1 = slope(12);
    x3 = 1.3; y3 = slope(13);
  elseif 1.3 < ctrl && ctrl <= 1.4
    x1 = 1.3; y1 = slope(13);
    x3 = 1.4; y3 = slope(14);
  elseif 1.4 < ctrl && ctrl <= 1.5
    x1 = 1.4; y1 = slope(14);
    x3 = 1.5; y3 = slope(15);
  else % Greater than 1.5; use 1.5 volts for the step rate.
    pctIncr = slope(15)*sTime;
    posFlag = 1;
  end
end

% If required, interpolate between the boundary curves.
if posFlag < 0
  % y2 = slope and x2 = ctrl
  y2 = ((ctrl-x1)*(y3-y1))/(x3-x1) + y1; % Equation 4.15.
  pctIncr = y2*sTime;
end

% Percent open = current position plus or minus percent increment.
pctNew = pct + dir*pctIncr;

% Check for valve saturation - the valve cannot be opened more than 100%.
if pctNew > 100
  pctNew = 100;
end

% Coefficients of the Flow Rate (SCFM) vs. Valve Position (% Open) curve (Figure 4-9).
%. This curve was generated using MATLAB's cftool operating on data captured with
% ControlDesk.
a1 = 21.29;
b1 = -0.0006;
c1 = -21.29;
d1 = -0.05416;
```

```
% Calculate how much air is currently flowing. pctNew can be less than zero since valve
% delay is modeled as a negative initial value in 'Data Store Memory.'
if pctNew < 0
  Q = 0;
else
  Q = a1*exp(b1*pctNew) + c1*exp(d1*pctNew); % Equation 4.14.
end
```

Appendix B

Embedded MATLAB Functions

Speed Controller

Check Ctrl

```
function [dir,rate] = fcn(act,negErr,posErr,err,ctrl,rMin,rMax)
% This block supports the Embedded MATLAB subset.
% See the help menu for details.

%{
 This function determines the direction of valve motion (open, close or hold) and the rate of
opening and closing.
 inputs:
        act = sets automatic speed control to on or off.
        negErr = upper limit of dead band.
        posErr = lower limit of dead band.
        err = difference in set and measured rpm.
        ctrl = stepping rate as determined by the controller.
        rMin = minimum stepping rate allowed by the user.
        rMax = maximum stepping rate allowed by the user.
 outputs:
        dir = direction of valve travel - open, close or hold.
        rate = stepping rate in volts to valve.
%}

% Close the valve fairly rapidly if speed control is turned off.
if act < 0.5
  rate = rMax; % e.g. 0.3 volts.
  dir = 1.5; % 1.5 volts to close.
  return;
end

% Hold the valve's position if the measured rpm is within the range of the user-specified
% allowable error (i.e. dead band). Remember that positive overshoot equals negative error
% since error = set speed - measured speed.
if err < 0 % Measured speed greater than set speed.
  if err >= -negErr % Negative error is entered as a positive number.
    rate = 0;
    dir = 0;
    return;
```

```
  end
end

if err > 0 % Measured speed less than set speed.
  if err <= posErr
    rate = 0;
    dir = 0;
    return;
  end
end

if err == 0 % Measured speed equals set speed.
  rate = 0;
  dir = 0;
  return;
end

% The sign of the control signal determines whether to open or close the valve.
if ctrl < 0
  dir = 1.5; % 1.5 volts to close.
elseif ctrl > 0
  dir = 8.5; % 8.5 volts to open.
else
  dir = 0; % 0 volts to hold the current position.
end

% The stepping rate must be a positive voltage.
ctrl = abs(ctrl);

% Limit the stepping rates according to user preferences.
if ctrl <= rMin
  rate = rMin;
  return;
end

if ctrl >= rMax
  rate = rMax;
  return;
end

% Use the stepping rate determined by the controller.
rate = ctrl;
```

Check Int Windup

```
function [Ki,reset] = fcn(ctrl,rMax,KiSet)
% This block supports the Embedded MATLAB subset.
% See the help menu for details.

if ctrl > rMax % Deactivate integral control.
  Ki = 0;
  reset = 1;
else % Continue with integral action.
  Ki = KiSet;
  reset = 0;
end
```

Deact Deriv Ctrl

```
function ddc = fcn(dRPM,aRPM)
% This block supports the Embedded MATLAB subset.
% See the help menu for details.

% Deactivate derivative control when rotor speed exceeds a user-specified rpm.
if aRPM >= dRPM
  ddc = 0; % Turn off.
else
  ddc = 1; % Turn on.
end
```

Appendix C

Python Programs

_amp adjust.py

```
# -*- coding: cp1252 -*-

# NOTE:  All checkboxes on the "amp adjust" dialog are disabled if the rotor is spinning.

def On_Instrumentation_ampadjust_btnCalMode_StateChanged(State):
    """
    Syntax      : On_Instrumentation_ampadjust_btnCalMode_StateChanged(State)

    Purpose     : StateChanged event handler; activate/deactivate calibration mode.

    Parameters  : State
    """
    from cdautomationlib import Instrumentation

    cLayout = Instrumentation().Layouts.Item("control parms")
    btnIGReset = cLayout.Instruments.Item("btnIGReset")
    btnIGReset.CheckState = 1

    Layout  = Instrumentation().Layouts.Item("amp adjust")
    btnAmp1 = Layout.Instruments.Item("btnAmp1")
    btnAmp2 = Layout.Instruments.Item("btnAmp2")
    btnAmp3 = Layout.Instruments.Item("btnAmp3")
    btnAmp4 = Layout.Instruments.Item("btnAmp4")
    btnAmp5 = Layout.Instruments.Item("btnAmp5")
    btnAmp6 = Layout.Instruments.Item("btnAmp6")
    btnAmp7 = Layout.Instruments.Item("btnAmp7")
    btnAmp8 = Layout.Instruments.Item("btnAmp8")

    if State == 1: # Activate cal mode - turn off all amplifiers.
      btnAmp1.CheckState = 0
      btnAmp2.CheckState = 0
      btnAmp3.CheckState = 0
      btnAmp4.CheckState = 0
      btnAmp5.CheckState = 0
      btnAmp6.CheckState = 0
      btnAmp7.CheckState = 0
      btnAmp8.CheckState = 0
      btnAmp8.CheckState = 0
```

```
  else: # Deactivate cal mode - turn on all amplifiers.
    btnAmp1.CheckState = 1
    btnAmp2.CheckState = 1
    btnAmp3.CheckState = 1
    btnAmp4.CheckState = 1
    btnAmp5.CheckState = 1
    btnAmp6.CheckState = 1
    btnAmp7.CheckState = 1
    btnAmp8.CheckState = 1
    btnAmp8.CheckState = 1

  return  # Just to mark the end of the function.


# Determine if calibration can be performed.

def calDim(): # Dim - the checkboxes are "dimmed" or disabled.

 import win32con
 import win32ui

 from cdautomationlib import Instrumentation

 aLayout  = Instrumentation().Layouts.Item("amp adjust")
 btnCalMode = aLayout.Instruments.Item("btnCalMode")

 if btnCalMode.CheckEnabled:
  return False # Okay to calibrate.

 # If calibration mode is not enabled, then no calibration can be performed.

 msg1 = "Calibration cannot be performed unless the rotor is at rest and\n"
 msg2 = "both 'Set Speed' and Valve In/Out 'Manual Operation' are off."
 msg = msg1 + msg2
 win32ui.MessageBox(msg, "Information", win32con.MB_OK |
win32con.MB_ICONINFORMATION)
 return True


# Determine if calibration is activated.

def calOff():

 import win32con
```

```
 import win32ui

 from cdautomationlib import Instrumentation

 aLayout  = Instrumentation().Layouts.Item("amp adjust")
 btnCalMode = aLayout.Instruments.Item("btnCalMode")

 if btnCalMode.Value == 1:
   return False # Okay to calibrate.

 # If calibration is not checked, then no calibration can be performed.

 msg ="Please activate calibration mode."
 win32ui.MessageBox(msg, "Information", win32con.MB_OK |
win32con.MB_ICONINFORMATION)
 return True


def On_Instrumentation_ampadjust_btn25up_Click():
   """
   Syntax     : On_Instrumentation_ampadjust_btn25up_Click()

   Purpose    : Click event handler; activate amps 2 and 5.

   Parameters : None

   """

   if calDim():
     return

   if calOff():
     return

   from cdautomationlib import Instrumentation
   Layout = Instrumentation().Layouts.Item("amp adjust")

   btnAmp2 = Layout.Instruments.Item("btnAmp2")
   btnAmp5 = Layout.Instruments.Item("btnAmp5")

   # Activate desired amplifiers.

   btnAmp2.CheckState = 1
   btnAmp5.CheckState = 1
```

```
    return


def On_Instrumentation_ampadjust_btn47dn_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btn47dn_Click()

    Purpose     : Click event handler; activate amps 4 and 7.

    Parameters  : None

    """

    if calDim():
      return

    if calOff():
      return

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("amp adjust")

    btnAmp4 = Layout.Instruments.Item("btnAmp4")
    btnAmp7 = Layout.Instruments.Item("btnAmp7")

    # Activate desired amplifiers.

    btnAmp4.CheckState = 1
    btnAmp7.CheckState = 1

    return


def On_Instrumentation_ampadjust_btn16up_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btn16up_Click()

    Purpose     : Click event handler; activate amps 1 and 6.

    Parameters  : None

    """
```

```
    if calDim():
      return

    if calOff():
      return

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("amp adjust")

    btnAmp1 = Layout.Instruments.Item("btnAmp1")
    btnAmp6 = Layout.Instruments.Item("btnAmp6")

    # Activate desired amplifiers.

    btnAmp1.CheckState = 1
    btnAmp6.CheckState = 1

    return


def On_Instrumentation_ampadjust_btn38dn_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btn38dn_Click()

    Purpose     : Click event handler; activate amps 3 and 8.

    Parameters  : None

    """
    if calDim():
      return

    if calOff():
      return

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("amp adjust")

    btnAmp3 = Layout.Instruments.Item("btnAmp3")
    btnAmp8 = Layout.Instruments.Item("btnAmp8")

    # Activate desired amplifiers.

    btnAmp3.CheckState = 1
```

```
    btnAmp8.CheckState = 1

    return


def On_Instrumentation_ampadjust_btnSave25_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btnSave25_Click()

    Purpose     : Click event handler; save calibration values to global variables.

    Parameters  : None

    """

    if calDim():
      return

    if calOff():
      return

    global num24up
    global num57up

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("amp adjust")

    btnAmp2 = Layout.Instruments.Item("btnAmp2")
    btnAmp5 = Layout.Instruments.Item("btnAmp5")

    # Amps 2 and 4 equal axis 1/channel 1.
    num24avg = Layout.Instruments.Item("Pos1")
    # Amps 5 and 7 equal axis 3/channel 3.
    num57avg = Layout.Instruments.Item("Pos3")

    num24up = num24avg.Value
    num57up = num57avg.Value

    btnAmp2.CheckState = 0
    btnAmp5.CheckState = 0

    return
```

```
def On_Instrumentation_ampadjust_btnSave47_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btnSave47_Click()

    Purpose     : Click event handler; save calibration values to global variables.

    Parameters  : None

    """

    if calDim():
      return

    if calOff():
      return

    global num24dn
    global num57dn

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("amp adjust")

    btnAmp4 = Layout.Instruments.Item("btnAmp4")
    btnAmp7 = Layout.Instruments.Item("btnAmp7")

    # Amps 2 and 4 equal axis 1/channel 1.
    num24avg = Layout.Instruments.Item("Pos1")
    # Amps 5 and 7 equal axis 3/channel 3.
    num57avg = Layout.Instruments.Item("Pos3")

    num24dn = num24avg.Value
    num57dn = num57avg.Value

    btnAmp4.CheckState = 0
    btnAmp7.CheckState = 0

    return


def On_Instrumentation_ampadjust_btnSave16_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btnSave16_Click()

    Purpose     : Click event handler; save calibration values to global variables.
```

```
    Parameters  : None

    """

    if calDim():
      return

    if calOff():
      return

    global num13up
    global num68up

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("amp adjust")

    btnAmp1 = Layout.Instruments.Item("btnAmp1")
    btnAmp6 = Layout.Instruments.Item("btnAmp6")

    # Amps 1 and 3 equal axis 2/channel 2.
    num13avg = Layout.Instruments.Item("Pos2")
    # Amps 6 and 8 equal axis 4/channel 4.
    num68avg = Layout.Instruments.Item("Pos4")

    num13up = num13avg.Value
    num68up = num68avg.Value

    btnAmp1.CheckState = 0
    btnAmp6.CheckState = 0

    return


def On_Instrumentation_ampadjust_btnSave38_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btnSave38_Click()

    Purpose     : Click event handler; save calibration values to global variables.

    Parameters  : None

    """
```

```
  if calDim():
    return

  if calOff():
    return

  global num13dn
  global num68dn

  from cdautomationlib import Instrumentation
  Layout = Instrumentation().Layouts.Item("amp adjust")

  btnAmp3 = Layout.Instruments.Item("btnAmp3")
  btnAmp8 = Layout.Instruments.Item("btnAmp8")

  # Amps 1 and 3 equal axis 2/channel 2.
  num13avg = Layout.Instruments.Item("Pos2")
  # Amps 6 and 8 equal axis 4/channel 4.
  num68avg = Layout.Instruments.Item("Pos4")

  num13dn = num13avg.Value
  num68dn = num68avg.Value

  btnAmp3.CheckState = 0
  btnAmp8.CheckState = 0

  return


# Display the given message and ask for yes or no response.

def ShowMessageBoxYN(msg):

 import win32con
 import win32ui

 rc = win32ui.MessageBox(msg, "Warning", win32con.MB_YESNO |
win32con.MB_ICONWARNING)

 return rc # Yes = 6; No = 7


# Display Windows save file dialog.
```

```
def ShowSaveFileDialog(initPath):

 import win32con
 import win32ui
 import os

 # Use "None" for NULL.
 Dlg = win32ui.CreateFileDialog(False, None, None, win32con.OFN_HIDEREADONLY|
  win32con.OFN_FILEMUSTEXIST|win32con.OFN_PATHMUSTEXIST,
  "Data Files (*.dat)|*.dat|Text Files (*.txt)|*.txt|All Files (*.*)|*.*||", None)

 Dlg.SetOFNInitialDir(initPath)

 # Save file; prompt to overwrite.
 while True:
  if Dlg.DoModal() == win32con.IDOK:
   fName = Dlg.GetPathName()
   if os.path.isfile(fName): # Does the file already exist?
    msg = "File " + fName + " exists."
    msg = msg + "\nDo you want to overwrite it?"
    # If the file exists, ask permission to overwrite.
    rc = ShowMessageBoxYN(msg)
    if rc is 6: # Yes, overwrite.
     return Dlg.GetPathName()
    else: # No, do not overwrite; prompt for another name.
     continue
   else: # New file.
    return Dlg.GetPathName()
  else:
   # Ensure that the string "None" is returned if Cancel is pressed.
   return "None"


# Display Windows open file dialog.

def ShowOpenFileDialog(initPath):

 import win32con
 import win32ui

 # Use "None" for NULL.
 Dlg = win32ui.CreateFileDialog(TRUE, None, None, win32con.OFN_HIDEREADONLY|
  win32con.OFN_FILEMUSTEXIST|win32con.OFN_PATHMUSTEXIST,
  "Data Files (*.dat)|*.dat|Text Files (*.txt)|*.txt|All Files (*.*)|*.*||", None)
```

```
  Dlg.SetOFNInitialDir(initPath)
  if Dlg.DoModal() == win32con.IDOK:
    return Dlg.GetPathName()
  else:
    # Ensure that the string "None" is returned if Cancel is pressed.
    return "None"


# Display the given message.

def ShowMessageBox(type, msg):

  import win32con
  import win32ui

  if type == 0: # Error.
    win32ui.MessageBox(msg, "Error", win32con.MB_OK | win32con.MB_ICONERROR)
  elif type == 1: # Information.
    win32ui.MessageBox(msg, "Information", win32con.MB_OK |
win32con.MB_ICONINFORMATION)
  else: # Warning.
    win32ui.MessageBox(msg, "Warning", win32con.MB_OK |
win32con.MB_ICONWARNING)

  return # Used to mark the end of a function.


def On_Instrumentation_ampadjust_btnSaveCal_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btnSaveCal_Click()

    Purpose     : Click event handler; save calibration data to a file.

    Parameters  : None

    """

    global num13up
    global num13dn
    global num24up
    global num24dn
    global num57up
    global num57dn
```

```
global num68up
global num68dn

if calDim():
  return

if calOff():
  return

import os
fileName = ShowSaveFileDialog(os.getcwd())

if fileName == "None": # The Cancel button was pressed.
  return

import cdautomationlib
Inst = cdautomationlib.Instrumentation()
from cdautomationlib import Instrumentation
Layout = Instrumentation().Layouts.Item("amp adjust")

knbCal1 = Layout.Instruments.Item("knbCal1")
knbCal2 = Layout.Instruments.Item("knbCal2")
knbCal3 = Layout.Instruments.Item("knbCal3")
knbCal4 = Layout.Instruments.Item("knbCal4")

"""
Note the following:
  num13up - amps 1 and 6 on; num13dn - amps 3 and 8 on.
  num24up - amps 2 and 5 on; num24dn - amps 4 and 7 on.
  num57up - amps 2 and 5 on; num57dn - amps 4 and 7 on.
  num68up - amps 1 and 6 on; num68dn - amps 3 and 8 on.
"""
adj13 = -(num13up + num13dn)/2 # Channel 2.
adj24 = -(num24up + num24dn)/2 # Channel 1.
adj57 = -(num57up + num57dn)/2 # Channel 3.
adj68 = -(num68up + num68dn)/2 # Channel 4.

"""
Save the calibration values to a file. The format of the file is:
channel 2 value
channel 1 value
channel 3 value
channel 4 value
"""
```

```
    error = 0
    info = 1
    import sys
    try:
      file = open(fileName, "w")
      file.write("%f\n" % adj13)
      file.write("%f\n" % adj24)
      file.write("%f\n" % adj57)
      file.write("%f" % adj68)
      file.close()
      msg = "%s \nsuccessfully written." % fileName
      print msg
      ShowMessageBox(info, msg)
    except IOError, (errno, strerror):
      msg = fileName
      msg = msg + "\nIO error(%s): %s" % (errno, strerror)
      print msg
      ShowMessageBox(error, msg)
      return

    # Update each knob.

    knbCal1.Value = adj24
    knbCal2.Value = adj13
    knbCal3.Value = adj57
    knbCal4.Value = adj68

    Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal1", "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal2", "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal3", "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal4", "WriteData")

    return # Used to mark the end of a function.


def On_Instrumentation_ampadjust_btnLoadCal_Click():
    """
    Syntax      : On_Instrumentation_ampadjust_btnLoadCal_Click()

    Purpose     : Click event handler; read calibration data from a file.

    Parameters  : None

    """
```

```
if calDim():
  return

if calOff():
  return

import os
fileName = ShowOpenFileDialog(os.getcwd())

if fileName == "None": # The Cancel button was pressed.
  return

import cdautomationlib
Inst = cdautomationlib.Instrumentation()
from cdautomationlib import Instrumentation
Layout = Instrumentation().Layouts.Item("amp adjust")

knbCal1 = Layout.Instruments.Item("knbCal1")
knbCal2 = Layout.Instruments.Item("knbCal2")
knbCal3 = Layout.Instruments.Item("knbCal3")
knbCal4 = Layout.Instruments.Item("knbCal4")

error = 0
info = 1
numLines = 0

# Ensure there are only 4 lines in the file. This is a simple check to keep the user from
# selecting an erroneous file.

try:
  file = open(fileName, "r")
  for line in file:
    numLines = numLines + 1
  file.close()
except IOError, (errno, strerror):
  msg = fileName
  msg = msg + "\nIO error(%s): %s" % (errno, strerror)
  print msg
  ShowMessageBox(error, msg)
  return

if numLines is not 4:
  msg = fileName
```

```
  msg = msg + "\nis not a valid data file."
  print msg
  ShowMessageBox(error, msg)
  return

"""
Read the calibration values from a file. The format of the file is:
channel 2 value
channel 1 value
channel 3 value
channel 4 value
"""
import string
try:
  file = open(fileName, "r")
  adj13 = float(string.strip(file.readline()))
  adj24 = float(string.strip(file.readline()))
  adj57 = float(string.strip(file.readline()))
  adj68 = float(string.strip(file.readline()))
  file.close()
  msg = "%s \nsuccessfully read." % fileName
  print msg
  ShowMessageBox(info, msg)
except IOError, (errno, strerror):
  msg = fileName
  msg = msg + "\nIO error(%s): %s" % (errno, strerror)
  print msg
  ShowMessageBox(error, msg)
  return

# Update each knob.

knbCal1.Value = adj24
knbCal2.Value = adj13
knbCal3.Value = adj57
knbCal4.Value = adj68

Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal1", "WriteData")
Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal2", "WriteData")
Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal3", "WriteData")
Inst.ConnectionController.ProcessAnimationEvent("amp adjust://knbCal4", "WriteData")

return # Used to mark the end of a function.
```

Appendix C

Python Programs

_aux input.py

```
#  -*- coding: cp1252 -*-

def manageAmps(Value):
    """
    Perform the following depending on Value:
      Value = 1:  enable the amplifier checkboxes.
      Value = 2:  disable the amplifier checkboxes.
      Value = 3:  enable the amplifier checkboxes, and turn on the amplifiers.

    Note: this same function is in _layout_start.py
    """

    from cdautomationlib import Instrumentation

    aLayout  = Instrumentation().Layouts.Item("amp adjust")
    btnAmp1 = aLayout.Instruments.Item("btnAmp1")
    btnAmp2 = aLayout.Instruments.Item("btnAmp2")
    btnAmp3 = aLayout.Instruments.Item("btnAmp3")
    btnAmp4 = aLayout.Instruments.Item("btnAmp4")
    btnAmp5 = aLayout.Instruments.Item("btnAmp5")
    btnAmp6 = aLayout.Instruments.Item("btnAmp6")
    btnAmp7 = aLayout.Instruments.Item("btnAmp7")
    btnAmp8 = aLayout.Instruments.Item("btnAmp8")

    if Value == 1 or Value == 3:
      btnAmp1.CheckEnabled = True
      btnAmp2.CheckEnabled = True
      btnAmp3.CheckEnabled = True
      btnAmp4.CheckEnabled = True
      btnAmp5.CheckEnabled = True
      btnAmp6.CheckEnabled = True
      btnAmp7.CheckEnabled = True
      btnAmp8.CheckEnabled = True

    if Value == 2:
      btnAmp1.CheckEnabled = False
      btnAmp2.CheckEnabled = False
```

```
    btnAmp3.CheckEnabled = False
    btnAmp4.CheckEnabled = False
    btnAmp5.CheckEnabled = False
    btnAmp6.CheckEnabled = False
    btnAmp7.CheckEnabled = False
    btnAmp8.CheckEnabled = False

  if Value == 3:
    btnAmp1.CheckState = 1
    btnAmp2.CheckState = 1
    btnAmp3.CheckState = 1
    btnAmp4.CheckState = 1
    btnAmp5.CheckState = 1
    btnAmp6.CheckState = 1
    btnAmp7.CheckState = 1
    btnAmp8.CheckState = 1

  return # Just marks the end of this function.


def On_Instrumentation_auxinput_manCtrlRadioButton_UserInteraction
(DispId,Value,EventId):
  """
  Syntax:  On_Instrumentation_auxinput_manCtrlRadioButton_UserInteraction
(DispId,Value,EventId)

  Purpose:  UserInteraction event handler; activate manual operation of the flow control
valve.

  Parameters:  DispId,Value,EventId

  """

  import win32con
  import win32ui

  from cdautomationlib import Instrumentation
  import cdautomationlib

  Inst = cdautomationlib.Instrumentation()
  Layout  = Instrumentation().Layouts.Item("layout_start")
  airButton = Layout.Instruments.Item("airOnOffRadioButton")
  rotorStatus = Layout.Instruments.Item("rotorStatusMessage")
  digitalTach = Layout.Instruments.Item("mainDigitalTachDisplay")
```

```
rpm = digitalTach.Value

cpLayout  = Instrumentation().Layouts.Item("control parms")
limitADRButton = cpLayout.Instruments.Item("btnLimitDetectADR")
auxLayout  = Instrumentation().Layouts.Item("aux input")
manButton = auxLayout.Instruments.Item("manCtrlRadioButton")

# Turn off automatic speed control.
airButton.Value = 0

# Update the automatic "Set Speed" radio buttons.
Inst.ConnectionController.ProcessAnimationEvent("layout_start://airOnOffRadioButton",
  "WriteData")

# Enable ADR limit detect only if the rotor is at rest and the air is off.

if airButton.Value == 0 and manButton.Value == 0:
  checkVal = 0
else:
  checkVal = 1

if rpm < 1 and checkVal == 0:
  limitADRButton.CheckEnabled = True
else:
  limitADRButton.CheckState = 0
  limitADRButton.CheckEnabled = False

# The air can be turned off at anytime.
if manButton.Value == 0:
  return

# Make sure the amplifiers are all on before spinning up the rotor.

aLayout  = Instrumentation().Layouts.Item("amp adjust")
btnCalMode = aLayout.Instruments.Item("btnCalMode")
btnCalMode.CheckEnabled = True
btnCalMode.CheckState = 0

# Arg = 3; enable the amplifier checkboxes, and turn on the amplifiers.
manageAmps(3)

return # Just marks the end of this function.
```

```
def On_Instrumentation_auxinput_manPosRadioButton_UserInteraction
(DispId,Value,EventId):
    """
    Syntax:  On_Instrumentation_auxinput_manPosRadioButton_UserInteraction
(DispId,Value,EventId)

    Purpose:  UserInteraction event handler; hold the valve's position when in manual mode.

    Parameters:  DispId,Value,EventId

    """

    from cdautomationlib import Instrumentation
    import cdautomationlib

    Inst = cdautomationlib.Instrumentation()

    auxLayout  = Instrumentation().Layouts.Item("aux input")
    posButton = auxLayout.Instruments.Item("manPosRadioButton")
    rateNumInput = auxLayout.Instruments.Item("manStepRateNumericInput")

    val = posButton.Value

    # Set the step rate to zero if the hold button was pressed.
    if val < 0.5:
      rateNumInput.Value = 0

      # Update the step rate in the numerical input box.
      Inst.ConnectionController.ProcessAnimationEvent("aux
input://manStepRateNumericInput", "WriteData")

    return # Just marks the end of this function.
```

Appendix C

Python Programs

_control parms.py

```
#  -*- coding: cp1252 -*-

def On_Instrumentation_controlparms_actADRRadioButton_UserInteraction
(DispId,Value,EventId):
    """
    Syntax:  On_Instrumentation_controlparms_actADRRadioButton_UserInteraction
(DispId,Value,EventId)

    Purpose     : UserInteraction event handler; activate adaptive control.

    Parameters  : DispId,Value,EventId

    """

    from cdautomationlib import Instrumentation

    cLayout  = Instrumentation().Layouts.Item("control parms")
    ADRButton = cLayout.Instruments.Item("actADRRadioButton")
    btnGpAct = cLayout.Instruments.Item("btnGpAct")
    btnHpAct = cLayout.Instruments.Item("btnHpAct")
    limitADRButton = cLayout.Instruments.Item("btnLimitDetectADR")
    manButton = cLayout.Instruments.Item("manCtrlRadioButtion")

    sLayout  = Instrumentation().Layouts.Item("layout_start")
    airButton = sLayout.Instruments.Item("airOnOffRadioButton")
    digitalTach = sLayout.Instruments.Item("mainDigitalTachDisplay")
    rpm = digitalTach.Value

    # Value = 0.0 if "Off" was pressed; Value = 1.0 if "On" was pressed.
    if ADRButton.Value == 0:
      btnGpAct.CheckState = 0
      btnHpAct.CheckState = 0
      limitADRButton.CheckState = 0

    elif ADRButton.Value == 1:
      btnGpAct.CheckState = 1
      btnHpAct.CheckState = 1
```

```
    limitADRButton.CheckState = 0

    """ Direct from Alex:
    Important Note:  limit detect should not be used when the motor is running.
    Shutting off the bearing while spinning can cause equipment damage.
    """
    # Therefore, enable ADR limit detect only if the rotor is at rest and the air is off.

    if airButton.Value == 0 and manButton.Value == 0:
      checkVal = 0
    else:
      checkVal = 1

    if rpm < 1 and checkVal == 0:
      limitADRButton.CheckEnabled = True
    else:
      limitADRButton.CheckState = 0
      limitADRButton.CheckEnabled = False

  else:
    return

  return # Just marks the end of this function.


# Return True if the rotor is stopped; return False if the rotor is still moving.

def rotorStopped():

  from cdautomationlib import Instrumentation

  sLayout  = Instrumentation().Layouts.Item("layout_start")
  airButton = sLayout.Instruments.Item("airOnOffRadioButton")
  digitalTach = sLayout.Instruments.Item("mainDigitalTachDisplay")
  rpm = digitalTach.Value

  auxLayout  = Instrumentation().Layouts.Item("aux input")
  manButton = auxLayout.Instruments.Item("manCtrlRadioButton")

  if airButton.Value == 0 and manButton.Value == 0:
    checkVal = 0
  else:
    checkVal = 1
```

```
  if rpm < 1 and checkVal == 0:
    return True

  # The the user why bearing control values cannot be loaded or saved at this time.

  msg1 ="Cannot Load/Save values unless the rotor is at rest,\n"
  msg2 = "and both 'Set Speed' and 'Man. Valve Control' are off."
  msg = msg1 + msg2
  ShowMessageBox(1, msg)
  return False


# Display the given message.

def ShowMessageBox(type, msg):

  import win32con
  import win32ui

  if type == 0: # Error.
    win32ui.MessageBox(msg, "Error", win32con.MB_OK | win32con.MB_ICONERROR)
  elif type == 1: # Information.
    win32ui.MessageBox(msg, "Information", win32con.MB_OK |
win32con.MB_ICONINFORMATION)
  else: # Warning.
    win32ui.MessageBox(msg, "Warning", win32con.MB_OK |
win32con.MB_ICONWARNING)

  return # Used to mark the end of a function.


# Display the given message and ask for yes or no response.

def ShowMessageBoxYN(msg):

  import win32con
  import win32ui

  rc = win32ui.MessageBox(msg, "Warning", win32con.MB_YESNO |
win32con.MB_ICONWARNING)

  return rc # Yes = 6; No = 7
```

```
# Display Windows open file dialog.

def ShowOpenFileDialog(initPath):

  import win32con
  import win32ui

  # Use "None" for NULL.
  Dlg = win32ui.CreateFileDialog(TRUE, None, None, win32con.OFN_HIDEREADONLY|
    win32con.OFN_FILEMUSTEXIST|win32con.OFN_PATHMUSTEXIST,
    "Data Files (*.dat)|*.dat|Text Files (*.txt)|*.txt|All Files (*.*)|*.*||", None)

  Dlg.SetOFNInitialDir(initPath)
  if Dlg.DoModal() == win32con.IDOK:
    return Dlg.GetPathName()
  else:
    # Ensure that the string "None" is returned if Cancel is pressed.
    return "None"


# Display Windows save file dialog.

def ShowSaveFileDialog(initPath):

  import win32con
  import win32ui
  import os

  # Use "None" for NULL.
  Dlg = win32ui.CreateFileDialog(False, None, None, win32con.OFN_HIDEREADONLY|
    win32con.OFN_FILEMUSTEXIST|win32con.OFN_PATHMUSTEXIST,
    "Data Files (*.dat)|*.dat|Text Files (*.txt)|*.txt|All Files (*.*)|*.*||", None)

  Dlg.SetOFNInitialDir(initPath)

  # Save file; prompt to overwrite.
  while True:
    if Dlg.DoModal() == win32con.IDOK:
      fName = Dlg.GetPathName()
      if os.path.isfile(fName): # Does the file already exist?
        msg = "File " + fName + " exists."
        msg = msg + "\nDo you want to overwrite it?"
        # If the file exists, ask permission to overwrite.
        rc = ShowMessageBoxYN(msg)
```

```
      if rc is 6: # Yes, overwrite.
        return Dlg.GetPathName()
      else: # No, do not overwrite; prompt for another name.
        continue
    else: # New file.
      return Dlg.GetPathName()
  else:
    # Ensure that the string "None" is returned if Cancel is pressed.
    return "None"


def On_Instrumentation_controlparms_loadBCValsPushButton_Click():
  """
  Syntax     : On_Instrumentation_controlparms_loadBCValsPushButton_Click()

  Purpose    : Click event handler; read bearing control settings from a file.

  Parameters : None

  """

  if rotorStopped() is False:
    return

  import os
  fileName = ShowOpenFileDialog(os.getcwd())

  # The Cancel button was pressed.
  if fileName == "None":
    return

  import cdautomationlib
  Inst = cdautomationlib.Instrumentation()
  from cdautomationlib import Instrumentation
  Layout = Instrumentation().Layouts.Item("control parms")

  sldBt1 = Layout.Instruments.Item("Bt1")
  sldBb1 = Layout.Instruments.Item("Bb1")
  sldKp1 = Layout.Instruments.Item("Kp1")
  sldKd1 = Layout.Instruments.Item("Kd1")
  sldKi1 = Layout.Instruments.Item("Ki1")
  sldBt2 = Layout.Instruments.Item("Bt2")
  sldBb2 = Layout.Instruments.Item("Bb2")
  sldKp2 = Layout.Instruments.Item("Kp2")
```

```
sldKd2 = Layout.Instruments.Item("Kd2")
sldKi2 = Layout.Instruments.Item("Ki2")

error = 0
info = 1
numLines = 0

# Ensure there are only 10 lines in the file. This is a simple check to keep the user from
# selecting an erroneous file.

try:
  file = open(fileName, "r")
  for line in file:
    numLines = numLines + 1
  file.close()
except IOError, (errno, strerror):
  msg = fileName
  msg = msg + "\nIO error(%s): %s" % (errno, strerror)
  print msg
  ShowMessageBox(error, msg)
  return

if numLines is not 10:
  msg = fileName
  msg = msg + "\nis not a valid data file."
  print msg
  ShowMessageBox(error, msg)
  return

import string
try:
  # Read the control settings from a file, and convert each value to a floating
  # point decimal number.
  file = open(fileName, "r")
  # Bt1.Value = float(string.strip(file.readline())) <- This will not work. Each
  # line must be first read into a separate variable. This variable can then be
  # assigned to the instrument.
  valBt1 = float(string.strip(file.readline()))
  valBb1 = float(string.strip(file.readline()))
  valKp1 = float(string.strip(file.readline()))
  valKd1 = float(string.strip(file.readline()))
  valKi1 = float(string.strip(file.readline()))
  valBt2 = float(string.strip(file.readline()))
  valBb2 = float(string.strip(file.readline()))
```

```
    valKp2 = float(string.strip(file.readline()))
    valKd2 = float(string.strip(file.readline()))
    valKi2 = float(string.strip(file.readline()))
    file.close()
    msg = "%s \nsuccessfully read." % fileName
    print msg
    ShowMessageBox(info, msg)
  except IOError, (errno, strerror):
    msg = fileName
    msg = msg + "\nIO error(%s): %s" % (errno, strerror)
    print msg
    ShowMessageBox(error, msg)
    return

  # Now, the values read from the file can be assigned to the instruments.

  sldBt1.Value = valBt1
  sldBb1.Value = valBb1
  sldKp1.Value = valKp1
  sldKd1.Value = valKd1
  sldKi1.Value = valKi1
  sldBt2.Value = valBt2
  sldBb2.Value = valBb2
  sldKp2.Value = valKp2
  sldKd2.Value = valKd2
  sldKi2.Value = valKi2

  # Update the instruments.

  Inst.ConnectionController.ProcessAnimationEvent("control parms://Bt1", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Bb1", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Kp1", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Kd1", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Ki1", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Bt2", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Bb2", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Kp2", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Kd2", "WriteData")
  Inst.ConnectionController.ProcessAnimationEvent("control parms://Ki2", "WriteData")

  return # Used to mark the end of a function.


def On_Instrumentation_controlparms_saveBCValsPushButton_Click():
```

```
"""
Syntax      : On_Instrumentation_controlparms_saveBCValsPushButton_Click()

Purpose     : Click event handler; save bearing control settings to a file.

Parameters  : None

"""

if rotorStopped() is False:
  return

import os
fileName = ShowSaveFileDialog(os.getcwd())

if fileName == "None":
  return

from cdautomationlib import Instrumentation
Layout = Instrumentation().Layouts.Item("control parms")

sldBt1 = Layout.Instruments.Item("Bt1")
sldBb1 = Layout.Instruments.Item("Bb1")
sldKp1 = Layout.Instruments.Item("Kp1")
sldKd1 = Layout.Instruments.Item("Kd1")
sldKi1 = Layout.Instruments.Item("Ki1")
sldBt2 = Layout.Instruments.Item("Bt2")
sldBb2 = Layout.Instruments.Item("Bb2")
sldKp2 = Layout.Instruments.Item("Kp2")
sldKd2 = Layout.Instruments.Item("Kd2")
sldKi2 = Layout.Instruments.Item("Ki2")

error = 0
info = 1
import sys
try:
  # Write the control settings to a file. For consistency, the order in which the
  # values are written is identical to that used by Alex.
  file = open(fileName, "w")
  file.write("%f\n" % sldBt1.Value)
  file.write("%f\n" % sldBb1.Value)
  file.write("%f\n" % sldKp1.Value)
  file.write("%f\n" % sldKd1.Value)
  file.write("%f\n" % sldKi1.Value)
```

```
      file.write("%f\n" % sldBt2.Value)
      file.write("%f\n" % sldBb2.Value)
      file.write("%f\n" % sldKp2.Value)
      file.write("%f\n" % sldKd2.Value)
      file.write("%f\n" % sldKi2.Value)
      file.close()
      msg = "%s \nsuccessfully written." % fileName
      print msg
      ShowMessageBox(info, msg)
    except IOError, (errno, strerror):
      msg = fileName
      msg = msg + "\nIO error(%s): %s" % (errno, strerror)
      print msg
      ShowMessageBox(error, msg)
      return

    return # Used to mark the end of a function.


def On_Instrumentation_controlparms_loadCruiseValsPushButton_Click():
    """
    Syntax      : On_Instrumentation_controlparms_loadCruiseValsPushButton_Click()

    Purpose     : Click event handler; read speed control settings from a file..

    Parameters  : None

    """
    if rotorStopped() is False:
      return

    import os
    fileName = ShowOpenFileDialog(os.getcwd())

    # The Cancel button was pressed.
    if fileName == "None":
      return

    import cdautomationlib
    Inst = cdautomationlib.Instrumentation()
    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("control parms")

    KpSlider = Layout.Instruments.Item("KpCruiseSlider")
```

```
KiSlider = Layout.Instruments.Item("KiCruiseSlider")
KdSlider = Layout.Instruments.Item("KdCruiseSlider")
KpPower = Layout.Instruments.Item("KpCruisePowerSelection")
KiPower = Layout.Instruments.Item("KiCruisePowerSelection")
KdPower = Layout.Instruments.Item("KdCruisePowerSelection")
stepMin = Layout.Instruments.Item("stepRateMin")
stepMax = Layout.Instruments.Item("stepRateMax")
negErr = Layout.Instruments.Item("negErrorLimit")
posErr = Layout.Instruments.Item("posErrorLimit")
tachMin = Layout.Instruments.Item("tachRPMMin")
tachMax = Layout.Instruments.Item("tachRPMMax")
ddcRPM = Layout.Instruments.Item("ddcAboveRPM")

error = 0
info = 1
numLines = 0

# Ensure there are only 13 lines in the file. This is a simple check to keep the
# user from selecting an erroneous file.

try:
  file = open(fileName, "r")
  for line in file:
    numLines = numLines + 1
  file.close()
except IOError, (errno, strerror):
  msg = fileName
  msg = msg + "\nIO error(%s): %s" % (errno, strerror)
  print msg
  ShowMessageBox(error, msg)
  return

if numLines is not 13:
  msg = fileName
  msg = msg + "\nis not a valid data file."
  print msg
  ShowMessageBox(error, msg)
  return

import string
try:
  # Read the control settings from a file, and convert each value to a floating
  # point decimal number.
  file = open(fileName, "r")
```

```
 # Bt1.Value = float(string.strip(file.readline())) <- This will not work. Each
 # line must be first read into a separate variable. This variable can then be
 # assigned to the instrument.
 valKpS = float(string.strip(file.readline()))
 valKiS = float(string.strip(file.readline()))
 valKdS = float(string.strip(file.readline()))
 valKpP = float(string.strip(file.readline()))
 valKiP = float(string.strip(file.readline()))
 valKdP = float(string.strip(file.readline()))
 valSMin = float(string.strip(file.readline()))
 valSMax = float(string.strip(file.readline()))
 valNErr = float(string.strip(file.readline()))
 valPErr = float(string.strip(file.readline()))
 valTMin = float(string.strip(file.readline()))
 valTMax = float(string.strip(file.readline()))
 valDDC = float(string.strip(file.readline()))
 file.close()
 msg = "%s \nsuccessfully read." % fileName
 print msg
 ShowMessageBox(info, msg)
except IOError, (errno, strerror):
 msg = fileName
 msg = msg + "\nIO error(%s): %s" % (errno, strerror)
 print msg
 ShowMessageBox(error, msg)
 return

# Now, the values read from the file can be assigned to the instruments.

KpSlider.Value = valKpS
KiSlider.Value = valKiS
KdSlider.Value = valKdS
KpPower.Value = valKpP
KiPower.Value = valKiP
KdPower.Value = valKdP
stepMin.Value = valSMin
stepMax.Value = valSMax
negErr.Value = valNErr
posErr.Value = valPErr
tachMin.Value = valTMin
tachMax.Value = valTMax
ddcRPM.Value = valDDC

# Update the instruments.
```

```
    Inst.ConnectionController.ProcessAnimationEvent("control parms://KpCruiseSlider",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://KiCruiseSlider",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://KdCruiseSlider",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control
parms://KpCruisePowerSelection", "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control
parms://KiCruisePowerSelection", "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control
parms://KdCruisePowerSelection", "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://stepRateMin",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://stepRateMax",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://negErrorLimit",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://posErrorLimit",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://tachRPMMin",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://tachRPMMax",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://ddcAboveRPM",
"WriteData")

    return # Used to mark the end of a function.


def On_Instrumentation_controlparms_saveCruiseValsPushButton_Click():
    """
    Syntax      : On_Instrumentation_controlparms_saveCruiseValsPushButton_Click()

    Purpose     : Click event handler; save speed control settings to a file..

    Parameters  : None

    """
    if rotorStopped() is False:
      return

    import os
```

```
fileName = ShowSaveFileDialog(os.getcwd())

if fileName == "None": # The Cancel button was pressed.
  return

from cdautomationlib import Instrumentation
Layout = Instrumentation().Layouts.Item("control parms")

KpSlider = Layout.Instruments.Item("KpCruiseSlider")
KiSlider = Layout.Instruments.Item("KiCruiseSlider")
KdSlider = Layout.Instruments.Item("KdCruiseSlider")
KpPower = Layout.Instruments.Item("KpCruisePowerSelection")
KiPower = Layout.Instruments.Item("KiCruisePowerSelection")
KdPower = Layout.Instruments.Item("KdCruisePowerSelection")
stepMin = Layout.Instruments.Item("stepRateMin")
stepMax = Layout.Instruments.Item("stepRateMax")
negErr = Layout.Instruments.Item("negErrorLimit")
posErr = Layout.Instruments.Item("posErrorLimit")
tachMin = Layout.Instruments.Item("tachRPMMin")
tachMax = Layout.Instruments.Item("tachRPMMax")
ddcRPM = Layout.Instruments.Item("ddcAboveRPM")

error = 0
info = 1
import sys
try:
  # Write the speed control settings to a file.
  file = open(fileName, "w")
  file.write("%f\n" % KpSlider.Value)
  file.write("%f\n" % KiSlider.Value)
  file.write("%f\n" % KdSlider.Value)
  file.write("%f\n" % KpPower.Value)
  file.write("%f\n" % KiPower.Value)
  file.write("%f\n" % KdPower.Value)
  file.write("%f\n" % stepMin.Value)
  file.write("%f\n" % stepMax.Value)
  file.write("%f\n" % negErr.Value)
  file.write("%f\n" % posErr.Value)
  file.write("%f\n" % tachMin.Value)
  file.write("%f\n" % tachMax.Value)
  file.write("%f\n" % ddcRPM.Value)
  file.close()
  msg = "%s \nsuccessfully written." % fileName
  print msg
```

```
    ShowMessageBox(info, msg)
  except IOError, (errno, strerror):
    msg = fileName
    msg = msg + "\nIO error(%s): %s" % (errno, strerror)
    print msg
    ShowMessageBox(error, msg)
    return

  return # Used to mark the end of a function.


def On_Instrumentation_controlparms_loadACValsPushButton_Click():
  """
  Syntax      : On_Instrumentation_controlparms_loadACValsPushButton_Click()

  Purpose     : Click event handler; read adaptive control settings from a file.

  Parameters  : None

  """
  if rotorStopped() is False:
    return

  import os
  fileName = ShowOpenFileDialog(os.getcwd())

  # The Cancel button was pressed.
  if fileName == "None":
    return

  import cdautomationlib
  Inst = cdautomationlib.Instrumentation()
  from cdautomationlib import Instrumentation
  Layout = Instrumentation().Layouts.Item("control parms")

  gLim = Layout.Instruments.Item("GSatLimit")
  bLim = Layout.Instruments.Item("BetaSatLimit")
  freq1 = Layout.Instruments.Item("rejectFreq1")
  freq2 = Layout.Instruments.Item("rejectFreq2")
  freq3 = Layout.Instruments.Item("rejectFreq3")
  gGain = Layout.Instruments.Item("DgGain")
  hGain = Layout.Instruments.Item("DhGain")

  error = 0
```

```
info = 1
numLines = 0

# Ensure there are only 7 lines in the file. This is a simple check to keep the
# user from selecting an erroneous file.

try:
  file = open(fileName, "r")
  for line in file:
    numLines = numLines + 1
  file.close()
except IOError, (errno, strerror):
  msg = fileName
  msg = msg + "\nIO error(%s): %s" % (errno, strerror)
  print msg
  ShowMessageBox(error, msg)
  return

if numLines is not 7:
  msg = fileName
  msg = msg + "\nis not a valid data file."
  print msg
  ShowMessageBox(error, msg)
  return

import string
try:
  # Read the control settings from a file, and convert each value to a floating
  # point decimal number.
  file = open(fileName, "r")
  # Bt1.Value = float(string.strip(file.readline())) <- This will not work. Each
  # line must be first read into a separate variable. This variable can then be
  # assigned to the instrument.
  valGLim = float(string.strip(file.readline()))
  valBLim = float(string.strip(file.readline()))
  valFreq1 = float(string.strip(file.readline()))
  valFreq2 = float(string.strip(file.readline()))
  valFreq3 = float(string.strip(file.readline()))
  valGGain = float(string.strip(file.readline()))
  valHGain = float(string.strip(file.readline()))
  file.close()
  msg = "%s \nsuccessfully read." % fileName
  print msg
  ShowMessageBox(info, msg)
```

```
    except IOError, (errno, strerror):
      msg = fileName
      msg = msg + "\nIO error(%s): %s" % (errno, strerror)
      print msg
      ShowMessageBox(error, msg)
      return

    # Now, the values read from the file can be assigned to the instruments.

    gLim.Value = valGLim
    bLim.Value = valBLim
    freq1.Value = valFreq1
    freq2.Value = valFreq2
    freq3.Value = valFreq3
    gGain.Value = valGGain
    hGain.Value = valHGain

    # Update the instruments.

    Inst.ConnectionController.ProcessAnimationEvent("control parms://GSatLimit",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://BetaSatLimit",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://rejectFreq1",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://rejectFreq2",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://rejectFreq3",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://DgGain",
"WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://DhGain",
"WriteData")

    return # Used to mark the end of a function.


def On_Instrumentation_controlparms_saveACValsPushButton_Click():
    """
    Syntax      : On_Instrumentation_controlparms_saveACValsPushButton_Click()

    Purpose     : Click event handler; write adaptive control settings to a file.

    Parameters  : None
```

```
"""
if rotorStopped() is False:
  return

import os
fileName = ShowSaveFileDialog(os.getcwd())

if fileName == "None":
  return

from cdautomationlib import Instrumentation
Layout = Instrumentation().Layouts.Item("control parms")

gLim = Layout.Instruments.Item("GSatLimit")
bLim = Layout.Instruments.Item("BetaSatLimit")
freq1 = Layout.Instruments.Item("rejectFreq1")
freq2 = Layout.Instruments.Item("rejectFreq2")
freq3 = Layout.Instruments.Item("rejectFreq3")
gGain = Layout.Instruments.Item("DgGain")
hGain = Layout.Instruments.Item("DhGain")

error = 0
info = 1
import sys
try:
  # Write the adaptive control settings to a file.
  file = open(fileName, "w")
  file.write("%f\n" % gLim.Value)
  file.write("%f\n" % bLim.Value)
  file.write("%f\n" % freq1.Value)
  file.write("%f\n" % freq2.Value)
  file.write("%f\n" % freq3.Value)
  file.write("%f\n" % gGain.Value)
  file.write("%f\n" % hGain.Value)
  file.close()
  msg = "%s \nsuccessfully written." % fileName
  print msg
  ShowMessageBox(info, msg)
except IOError, (errno, strerror):
  msg = fileName
  msg = msg + "\nIO error(%s): %s" % (errno, strerror)
  print msg
  ShowMessageBox(error, msg)
```

```
  return

return # Used to mark the end of a function.
```

Appendix C

Python Programs

_layout_start.py

```
#  -*- coding: cp1252 -*-

def On_Instrumentation_layout_start_setSpeedSlider_Changing(Value):
    """
    Syntax      : On_Instrumentation_layout_start_setSpeedSlider_Changing(Value)

    Purpose     : Changing event handler; synchronize speed input instruments.

    Parameters  : Value

    """

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("layout_start")
    comboBox = Layout.Instruments.Item("setSpeedNumericalInput")
    slider = Layout.Instruments.Item("setSpeedSlider")

    rpm = slider.Value
    rpm = round(rpm) # Eliminate everything to the right of the decimal point.

    # Ensure the combo box stays in sync with the slider.
    comboBox.Value = rpm

    return # Just marks the end of this function.


def
On_Instrumentation_layout_start_setSpeedNumericalInput_UserInteraction(DispId,Val
ue,EventId):
    """
    Syntax:  On_Instrumentation_layout_start_setSpeedNumericalInput_UserInteraction
(DispId,Value,EventId)

    Purpose     : UserInteraction event handler; synchronize speed input instruments.

    Parameters  : DispId,Value,EventId
```

```
    """

    from cdautomationlib import Instrumentation
    Layout = Instrumentation().Layouts.Item("layout_start")
    comboBox = Layout.Instruments.Item("setSpeedNumericalInput")
    slider = Layout.Instruments.Item("setSpeedSlider")

    # Ensure the slider stays in sync with the combo box.
    slider.Value = comboBox.Value

    return # Just marks the end of this function.


def manageAmps(Value):
    """
    Perform the following depending on Value:
      Value = 1:  enable the amplifier checkboxes.
      Value = 2:  disable the amplifier checkboxes.
      Value = 3:  enable the amplifier checkboxes, and turn on the amplifiers.

    Note: this same function is in _control parms.py
    """

    from cdautomationlib import Instrumentation

    aLayout  = Instrumentation().Layouts.Item("amp adjust")
    btnAmp1 = aLayout.Instruments.Item("btnAmp1")
    btnAmp2 = aLayout.Instruments.Item("btnAmp2")
    btnAmp3 = aLayout.Instruments.Item("btnAmp3")
    btnAmp4 = aLayout.Instruments.Item("btnAmp4")
    btnAmp5 = aLayout.Instruments.Item("btnAmp5")
    btnAmp6 = aLayout.Instruments.Item("btnAmp6")
    btnAmp7 = aLayout.Instruments.Item("btnAmp7")
    btnAmp8 = aLayout.Instruments.Item("btnAmp8")

    if Value == 1 or Value == 3:
      btnAmp1.CheckEnabled = True
      btnAmp2.CheckEnabled = True
      btnAmp3.CheckEnabled = True
      btnAmp4.CheckEnabled = True
      btnAmp5.CheckEnabled = True
      btnAmp6.CheckEnabled = True
      btnAmp7.CheckEnabled = True
      btnAmp8.CheckEnabled = True
```

```
    if Value == 2:
      btnAmp1.CheckEnabled = False
      btnAmp2.CheckEnabled = False
      btnAmp3.CheckEnabled = False
      btnAmp4.CheckEnabled = False
      btnAmp5.CheckEnabled = False
      btnAmp6.CheckEnabled = False
      btnAmp7.CheckEnabled = False
      btnAmp8.CheckEnabled = False

    if Value == 3:
      btnAmp1.CheckState = 1
      btnAmp2.CheckState = 1
      btnAmp3.CheckState = 1
      btnAmp4.CheckState = 1
      btnAmp5.CheckState = 1
      btnAmp6.CheckState = 1
      btnAmp7.CheckState = 1
      btnAmp8.CheckState = 1

    return # Just marks the end of this function.


def  On_Instrumentation_layout_start_actRotorRadioButton_UserInteraction
(DispId,Value,EventId):
    """
    Syntax      : On_Instrumentation_layout_start_actRotorRadioButton_UserInteraction
(DispId,Value,EventId)

    Purpose     : UserInteraction event handler; activate magnetic bearings.

    Parameters  : DispId,Value,EventId

    """

    # NOTE - a MATLAB function has been included in the Error model block to ensure that
    # the rotor cannot be deactivated unless it is at rest.

    import win32con
    import win32ui

    from cdautomationlib import Instrumentation
    import cdautomationlib
```

```
    Inst = cdautomationlib.Instrumentation()
    cLayout  = Instrumentation().Layouts.Item("control parms")
    actADRButton = cLayout.Instruments.Item("actADRRadioButton")
    limitADRButton = cLayout.Instruments.Item("btnLimitDetectADR")

    auxLayout  = Instrumentation().Layouts.Item("aux input")
    manButton = auxLayout.Instruments.Item("manCtrlRadioButton")

    Layout  = Instrumentation().Layouts.Item("layout_start")
    actButton = Layout.Instruments.Item("actRotorRadioButton")
    airButton = Layout.Instruments.Item("airOnOffRadioButton")
    digitalTach = Layout.Instruments.Item("mainDigitalTachDisplay")

    if actButton.Value == 0: # Deactivate the rotor.

      # Set the button to "On" if the rotor cannot be deactivated.
      if digitalTach.Value > 0:
        actButton.Value = 1

Inst.ConnectionController.ProcessAnimationEvent("layout_start://actRotorRadioButton",
        "WriteData") # Update the button with the new value.

        # Tell the user why the bearing cannot be deactivated.
        msg ="The rotor cannot be deactivated\nunless it is at rest."
        win32ui.MessageBox(msg, "Warning", win32con.MB_OK |
win32con.MB_ICONWARNING)
        return

      # Deactivate adaptive control; activate integral gain.

      btnGpAct = cLayout.Instruments.Item("btnGpAct")
      btnHpAct = cLayout.Instruments.Item("btnHpAct")
      btnIGReset = cLayout.Instruments.Item("btnIGReset")

      actADRButton.Value = 0
      btnGpAct.CheckState = 0
      btnHpAct.CheckState = 0
      btnIGReset.CheckState = 1

      # Update the original "pid" layout.

      pLayout  = Instrumentation().Layouts.Item("pid")
      btnOnOff = pLayout.Instruments.Item("btnOnOff")
```

```
      btnOnOff.CheckState = 0

    elif actButton.Value == 1: # Activate the rotor.

      # Deactivate ADR limit detect.
      limitADRButton.CheckState = 0

      # If the air is on, gray-out ADR limit detect.

      if airButton.Value == 1 or manButton.Value == 1:
        checkVal = 1
      else:
        checkVal = 0

      if checkVal == 1:
        limitADRButton.CheckEnabled = False
      else: # Air is off.
        limitADRButton.CheckEnabled = True

      # Turn on the amplifiers to activate the rotor.

      aLayout  = Instrumentation().Layouts.Item("amp adjust")
      btnCalMode = aLayout.Instruments.Item("btnCalMode")
      btnCalMode.CheckEnabled = True
      btnCalMode.CheckState = 0

      # Arg = 3; enable the amplifier checkboxes, and turn the amps on.
      manageAmps(3)

    else:
      return

    return # Just marks the end of this function.


def On_Instrumentation_layout_start_airOnOffRadioButton_UserInteraction
(DispId,Value,EventId):
    """
    Syntax  : On_Instrumentation_layout_start_airOnOffRadioButton_UserInteraction
(DispId,Value,EventId)

    Purpose     : UserInteraction event handler; activate automatic speed control.

    Parameters  : DispId,Value,EventId
```

```
    """

    from cdautomationlib import Instrumentation
    import cdautomationlib

    Inst = cdautomationlib.Instrumentation()
    Layout  = Instrumentation().Layouts.Item("layout_start")
    airButton = Layout.Instruments.Item("airOnOffRadioButton")
    rotorStatus = Layout.Instruments.Item("rotorStatusMessage")

    auxLayout  = Instrumentation().Layouts.Item("aux input")
    manButton = auxLayout.Instruments.Item("manCtrlRadioButton")

    # Turn off manual valve control.
    manButton.Value = 0
    Inst.ConnectionController.ProcessAnimationEvent("aux input://manCtrlRadioButton",
        "WriteData") # Update the button with the new value.

    # The air can be turned off at anytime.
    if airButton.Value == 0:
      return

    # "Rotor Status" must be "On" before the air can be turned on.
    status = rotorStatus.Value

    # Make sure the amplifiers are all on before spinning up the rotor.

    aLayout  = Instrumentation().Layouts.Item("amp adjust")
    btnCalMode = aLayout.Instruments.Item("btnCalMode")
    btnCalMode.CheckEnabled = True
    btnCalMode.CheckState = 0

    # Arg = 3; enable the amplifier checkboxes, and turn on the amplifiers.
    manageAmps(3)

    return # Just marks the end of this function.


def  On_Instrumentation_layout_start_ampStatusPushButton_UserInteraction
(DispId,Value,EventId):
    """
    Syntax: On_Instrumentation_layout_start_ampStatusPushButton_UserInteraction
(DispId,Value,EventId)
```

```
   Purpose     : UserInteraction event handler; configure "amp adjust" instruments.

   Parameters  : DispId,Value,EventId

   """

   from cdautomationlib import Instrumentation

   Layout  = Instrumentation().Layouts.Item("layout_start")
   digitalTach = Layout.Instruments.Item("mainDigitalTachDisplay")
   rpm = digitalTach.Value

   airButton = Layout.Instruments.Item("airOnOffRadioButton")

   aLayout = Instrumentation().Layouts.Item("amp adjust")
   btnCalMode = aLayout.Instruments.Item("btnCalMode")

   auxLayout = Instrumentation().Layouts.Item("aux input")
   manButton = auxLayout.Instruments.Item("manCtrlRadioButton")

   # Enable calibration mode and amplifier checkboxes only if the rotor is at rest,
   # and the air is turned off.

   if airButton.Value == 0 and manButton.Value == 0:
     checkVal = 0
   else:
     checkVal = 1

   if rpm < 1 and checkVal == 0:
     btnCalMode.CheckEnabled = True
     manageAmps(1) # Arg = 1;  enable the amplifier checkboxes.
   else:
     btnCalMode.CheckEnabled = False
     manageAmps(2) # Arg = 2;  disable the amplifier checkboxes.

   return # Just marks the end of this function.


def  On_Instrumentation_layout_start_controlParmsPushButton_UserInteraction
(DispId,Value,EventId):
   """
   Syntax:  On_Instrumentation_layout_start_controlParmsPushButton_UserInteraction
(DispId,Value,EventId)
```

```
Purpose     : UserInteraction event handler; configure "control parms" instruments.

Parameters  : DispId,Value,EventId

"""

from cdautomationlib import Instrumentation
import cdautomationlib

Inst = cdautomationlib.Instrumentation()

cLayout  = Instrumentation().Layouts.Item("control parms")
limitADRButton = cLayout.Instruments.Item("btnLimitDetectADR")

auxLayout  = Instrumentation().Layouts.Item("aux input")
manButton = auxLayout.Instruments.Item("manCtrlRadioButton")

sLayout  = Instrumentation().Layouts.Item("layout_start")
actButton = sLayout.Instruments.Item("actRotorRadioButton")
airButton = sLayout.Instruments.Item("airOnOffRadioButton")
digitalTach = sLayout.Instruments.Item("mainDigitalTachDisplay")
rpm = digitalTach.Value

# Enable ADR limit detect only if the rotor is at rest and the air is off.

if airButton.Value == 0 and manButton.Value == 0:
  checkVal = 0
else:
  checkVal = 1

if rpm < 1 and checkVal == 0:
  limitADRButton.CheckEnabled = True
else:
  limitADRButton.CheckState = 0
  limitADRButton.CheckEnabled = False

return # Just marks the end of this function.
```

Appendix C

Python Programs

_mbcntrl_a.py

```
def On_ExperimentManager_ExperimentOpened(ExperimentFilePath):
    """
    Syntax      : On_ExperimentManager_ExperimentOpened(ExperimentFilePath)

    Purpose     : Fired after an experiment is opened.

    Parameters  : ExperimentFilePath

    """


def On_Instrumentation_StartAnimation():
    """
    Syntax      : On_Instrumentation_StartAnimation()

    Purpose     : Fired when animation is started.

    Parameters  : None

    """

    # The following globals are used for calibration in the "amp adjust" layout.
    global num13up
    global num13dn
    global num24up
    global num24dn
    global num57up
    global num57dn
    global num68up
    global num68dn

    # Initialize global variables.
    num13up = 0
    num13dn = 0
    num24up = 0
    num24dn = 0
```

```
num57up = 0
num57dn = 0
num68up = 0
num68dn = 0

# names - contains all layouts in this experiment.
names = [ "amp adjust", "aux input", "bearing schematic", "control parms",
  "layout_start", "pid" ]

from cdautomationlib import Instrumentation
import cdautomationlib

# Count the number of open layouts.
Inst = cdautomationlib.Instrumentation();
numOpen = Inst.Layouts.Count

# Determine which layouts need to be opened.
i = 0
if numOpen > 0:
  for i in range(0, numOpen):
    obj = Inst.Layouts[i]
    name = obj.Name
    names.remove(name)

# A full pathname is needed to open a layout.
import os
pathname = os.getcwd()

# Open layouts which are not already opened.
numClose = len(names)
if numClose > 0:
  i = 0
  for i in range(0, numClose):
    name = names[i]
    fullName = pathname + "\\" + name + ".lay"
    Instrumentation().Layouts.Add(fullName)

# For now, never allow the bearing to be totally deactivated when an error occurs.
# Total deactivation while the rotor is spinning can result in bearing damage. Refer
# to Alex's doctoral thesis.

cLayout = Instrumentation().Layouts.Item("control parms")
limitDetect = cLayout.Instruments.Item("btnLimitDetect")
limitDetect.CheckState = 0
```

```
limitDetect.CheckEnabled = False

limitDetectADR = cLayout.Instruments.Item("btnLimitDetectADR")
limitDetectADR.CheckState = 0
limitDetectADR.CheckEnabled = False

 pLayout = Instrumentation().Layouts.Item("pid")
limitDetect = pLayout.Instruments.Item("btnLimitDetect")
limitDetect.CheckState = 0
limitDetect.CheckEnabled = False

# Deactivate adaptive control.

ADRButton = cLayout.Instruments.Item("actADRRadioButton")
ADRButton.Value = 0
GpButton = cLayout.Instruments.Item("btnGpAct")
GpButton.Value = 0
HpButton = cLayout.Instruments.Item("btnHpAct")
HpButton.Value = 0

# Deactivate calibration mode.

aLayout = Instrumentation().Layouts.Item("amp adjust")
calButton = aLayout.Instruments.Item("btnCalMode")
calButton.Value = 0

# Make the top-level layout, layout_start, the active layout.

Layout = Instrumentation().Layouts.Item("layout_start")
Layout.Activate()

# When animation starts, turn off speed control, and set the rpm to 0.

button = Layout.Instruments.Item("airOnOffRadioButton")
button.Value = 0
comboBox = Layout.Instruments.Item("setSpeedNumericalInput")
comboBox.Value = 0
slider = Layout.Instruments.Item("setSpeedSlider")
slider.Value = 0

auxLayout = Instrumentation().Layouts.Item("aux input")
manButton = auxLayout.Instruments.Item("manCtrlRadioButton")
manButton.Value = 0
```

```
    # Call events to update variables.

    Inst.ConnectionController.ProcessAnimationEvent("amp adjust://btnCalMode",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://actADRRadioButton",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://btnLimitDetect",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://btnLimitDetectADR",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://btnGpAct",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("control parms://btnHpAct",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("aux input://manCtrlRadioButton",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("layout_start://airOnOffRadioButton",
     "WriteData")

Inst.ConnectionController.ProcessAnimationEvent("layout_start://setSpeedNumericalInput",
     "WriteData")
    Inst.ConnectionController.ProcessAnimationEvent("layout_start://setSpeedSlider",
     "WriteData")

    return # Just marks the end of this function.
```