Development of a Multi-mode Adaptive Controller and Investigation of Gain Variations with Speed and Balance Changes

by

James W. 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 3, 2014

Keywords: ControlDesk, dSPACE, Simulink, adaptive control
disturbance rejection, magnetic bearings

Approved by

George T. Flowers, Professor of Mechanical Engineering and Dean of the Graduate School
David Bevly, Albert J. Smith, Jr. Professor of Mechanical Engineering
Song-yul Choe, Associate Professor of Mechanical Engineering

Abstract

Magnetic bearings offer a number of advantages over conventional rolling element bearings. Magnetic bearings provide support for rotating systems through magnetic levitation rather than by mechanical contact, nearly eliminating the energy losses attributable to friction in standard bearings. Low power consumption is one characteristic of magnetic bearings that has encouraged their use in an increasing number of applications. Another is the ability to use the bearing itself as an actuator in a controller that can alter the orbit of the rotating system within the bearing to reduce or eliminate the detrimental effects of disturbances acting on the system. In addition, controller outputs can potentially be used as an indicator of the general health or integrity of the system.

This work details the development of a multi-mode adaptive controller for a magnetic bearing system that is capable of suppressing disturbances acting at synchronous and asynchronous frequencies and caused by rotating imbalances and base motion. The work was based on an existing adaptive controller that formed part of the overall control system for a well sorted and well developed magnetically suspended rotor and flywheel. The development of the controller made extensive use of system modeling techniques and model-in-the-loop simulations. Development also required continual refinement of the system model and on-going reconfiguration of the operating environment since the ever increasing complexity of the controller often exceeded the real-time capabilities of the processor.

The modes of the controller, or the methods used by it to determine the frequency of the disturbance acting on the system, include discrete Fourier transform, rotor speed and manual observation. The adaptive controller was shown to produce excellent disturbance rejection and vibration suppression in all of the three modes. The capabilities of the controller operating in the first mode were demonstrated with simulated disturbances and in the second and third

modes with software simulations, simulated disturbances and physical changes in the balance of the rotor and flywheel.

This work also details the efforts to evaluate the predictive capability of adaptive controller gains. The correlation between gain variations and balance state has been demonstrated, but a repeatable and unambiguous response of the gains to a synchronous disturbance undetectable by other means has not been well established. The sensitivity of the gains to variations in rotor speed increases the difficulty of this task. Software simulations of the adaptive controller operating in speed mode showed the potential of using the gains as an indicator of a change in the balance or health of the system, but actual tests conducted on the magnetic bearing system were not as encouraging.

Acknowledgements

I thank my advisor Dr. George T. Flowers for funding this research and for his assistance and encouragement in completing this project. I also wish to acknowledge the excellent education I've received at Auburn and thank those professors who provided it. A special thank you goes to my brother Robert, to colleague Thomas Bitner and to Alex Matras. Both Robert and Thomas recently earned their Master of Science degrees in mechanical engineering from Auburn. Robert proved invaluable in learning the magnetic bearing system and patiently spent many hours educating me on both the basics and the details of ControlDesk, Simulink and system modeling. Thomas was always willing to listen whenever I encountered problems and often suggested creative and ultimately effective ways to solve them. Dr. Matras also earned his MSME degree from Auburn and was responsible for much of the design, construction and development of the magnetic bearing system. He always capably answered every question that I asked saving me the countless hours it would have taken to answer them on my own.

I would also like to thank the mates from next door. Fellow graduate students Fuxi Zhang, Haoyue Yang, Jessen Soobramaney and Li Chong all provided a hand at one time or another over the duration of the project. I would also like to thank John Marcell from the Mechanical Engineering Department and Bob Holland from General Machinery Company. John capably supervised the movement of the magnetic bearing system from its original location in Wilmore to its current one in the recently constructed Advanced Research Laboratory. Both John and Bob helped sort out the problem of an inadequate supply of air to the magnetic bearing system immediately after the move, and Bob reviewed the entire air distribution network in the new lab building to ensure that the all requirements could be met.

Table of Contents

List of Figures

List of Tables

Nomenclature

A_n	Fourier coefficient - cosine
A_p	state matrix
B_n	Fourier coefficient - sine
B_p	input matrix
C_p	output matrix
F	imbalance force (N)
f_n	Fourier frequency (Hz)
G_p	adaptive gain
H_p	adaptive gain
l	distance of mass imbalance from center of rotor (m)
m	imbalance mass (kg)
N	number of samples
n	coefficient number
r	sample number
u_d	disturbance function
u_p	input vector
x_p	state vector - rotor displacement
y	amplitude (mils)
y_p	output vector

Greek Letters:

Γ_p disturbance matrix

ΔG weighting matrix - G_p

ΔH weighting matrix - H_p

Δf fundamental cyclic frequency (Hz)

Δt sampling interval (sec)

φ_d disturbance vector

ω frequency (Hz, rad/sec, RPM)

Chapter 1 - Introduction and Literature Review

Magnetic bearings use magnetic levitation rather than mechanical contact to support loads such as rotors and shafts, and they provide an alternative to rolling element bearings for many applications especially those subject to extreme conditions. Magnetic bearings are generally classified as either passive or active. Passive bearings use permanent magnets and often electrodynamic effects to provide levitation without any control system [1]. Active magnetic bearings use electromagnets or a combination of electro and permanent magnets and an active control system, since the bearings are unstable by nature, to provide stable, reliable operation.

Most of the research conducted on magnetic bearings has been with active systems. In addition, nearly all magnetic bearing systems built for and used in commercial applications are active. However, passive systems have been and are continuing to be investigated, though few passive systems are employed in industrial applications [2], [3]. The magnetic bearing system used for the work presented here is an active one, and all discussions that follow will be with regard to active systems.

Magnetic bearings possess a number of characteristics that make them a superior alternative to conventional systems in many applications. These characteristics include the ability to tolerate high temperatures and to operate at high rotational speeds with very little frictional loss and with low power consumption. In fact, the objective of much of the current research on active systems is to quantify the extreme capabilities of the bearings. As one example, Mekhiche et.al. demonstrated the capability of magnetic bearings to operate at high temperatures and speeds [4]. They successfully operated an active system at 600^{o} C and at rotor speeds of 50,000 RPM. No other type of bearing can perform adequately under these conditions, and this system could extend the use of magnetic bearings to gas turbines and aircraft engines of novel design.

The ultimate load carrying capacity of magnetic bearings has not been precisely determined, but theoretical calculations indicate that it is less than that of conventional bearings by a factor of four or so. However, an active magnetic bearing system has been built for hydroelectric

power generation that supports a rotor with a mass of 50 tons. More typical of the loads carried by magnetic bearings are the rotors for turbo machinery used in the petroleum industry. These generally have a mass measured in the range of tons [5].

Magnetic bearing systems vary greatly in size, and in principle, there appears to be no upper limit on bearing size. One system developed for testing high-speed tires has a stator with an inside diameter of 6 meters [5]. At the other extreme are the magnetic bearings used in micro devices such as guidance systems, medical instruments and miniature motors. Small motors have been constructed that use a magnetically suspended rotor with dimensions on the order of a few millimeters. One example was built with a stator of 5 mm inside diameter, a rotor of 1.5 mm outside diameter and an air gap of 0.1 mm [6].

An illustration of a simplified active magnetic bearing system appears in Figure 1-1.

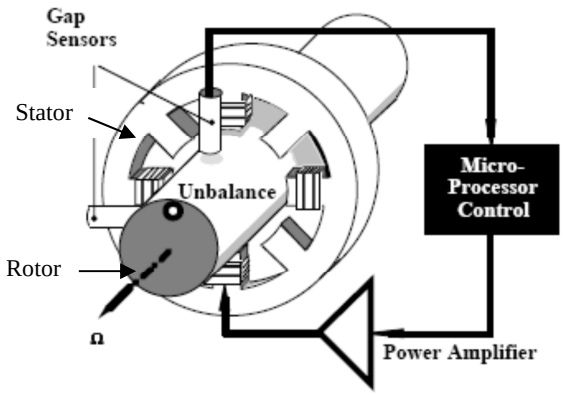


Figure 1-1 Active Magnetic Bearing System [5]

The electromagnets are formed by the poles of the stator, and each pole generates an attractive force. The power amplifier supplies current to pairs of electromagnets located on opposite sides of the rotor. The microprocessor calculates the output to send to the magnets to properly position the rotor. The microprocessor's calculations are based on the control laws developed for the system. Finally, the gap sensors and their associated electronics measure the distance (air gap) between the stator and rotor and provide the necessary feedback to the microprocessor.

The unbalance or imbalance shown in the Figure represents the fact that rotating machinery is seldom perfectly balanced. Imbalances generate inertial forces if the machine rotates about its geometric center, and these forces produce vibration. If the imbalances are large enough, the resultant vibration can become destructive especially at the speeds obtainable with magnetic bearings. One advantage of active magnetic bearing systems is the ability to incorporate vibration suppression and disturbance rejection mechanisms into their controllers. These can reduce and even eliminate the effects caused by the imbalance as well as by other disturbing forces acting indirectly on the bearings.

Some of the first attempts at vibration suppression included the application of passive and semi active mass balancers to rotating shafts supported by conventional bearings. Inoue et.al. describe a rotor containing cylindrical cavities in which small balls are placed. The researchers noted that under certain conditions the balls will arrange themselves to compensate for the unbalance of the rotor and reduce the dynamic loading on the bearings [7]. Similarly, Bovik and Hogfors demonstrated that plane rotors cut with grooves for the free movement of damped particles exhibit autobalancing [8].

One of the first genuinely active systems that utilized adaptive compensation to minimize the vibration in a bearing system was described by Burrows and Sahinkaya [9]. Following this influential work, numerous techniques for unbalance compensation and disturbance rejection have been developed, tested and successfully implemented. Of these, one of the earliest was based on notch filters. Initial work with these proved their ability to suppress disturbances, but it was often at the expense of stability since the filters were placed in the feedback path where they can affect the dynamic behavior of the bearing system [10], [11].

Several approaches were subsequently developed to solve the stability problems associated with the use of notch filters. These are broadly classified as adaptive feed forward compensation, and interestingly enough, some also use notch filters [11]. Most controllers of this type introduce synchronous signals into the control loop to cancel or reduce synchronous imbalance forces. Generally, the signal is a sinusoid of the proper phase and amplitude necessary to neutralize the disturbance. In addition, the controllers do not alter the closed loop

dynamics of the system and therefore, do not produce destabilizing effects [12], [13]. The feed forward approaches differ from one another based on the mechanism used to generate the compensating signal and the process used to adapt the signal to an imbalance [11].

Several researchers have developed controllers designed to account for uncertainties in model development and for variations in parameters that describe the plant model. These controllers often incorporate disturbance rejection into their designs. One example of earlier work with parametric techniques is that of Lum et.al. [14]. They developed a controller that performed "on-line" identification of imbalance parameters that were then used to update the controller for an active magnetic bearing system resulting in closed loop stability by continuous parameter update.

More recently, Huang and Lin developed a dynamic output feedback controller for a magnetic bearing system that included adaptive imbalance compensation derived from a linear-in-the-parameter imbalance force model [15]. This model provides an estimate of the forces generated by rotor eccentricity and mass imbalance. The estimate is based on a linear, parametric representation of the centripetal force vector, and adaptive compensator signals are generated synchronously.

Another existing approach to vibration control and disturbance rejection is disturbance observer based compensation (DOBC). Grochmal and Lynch wanted to provide precise tracking of rotor orbits in a magnetic bearing system by reducing static offset, the deviation of the rotor position from its set point, and synchronous vibration [16]. They developed a controller based on a hierarchical system that included both velocity and disturbance observers. The velocity observer functioned continuously to provide stable positioning while the disturbance observer provided rejection only at steady-state operation. Their controller integrated the observers with nonlinear state feedback to estimate and suppress the ill effects of static offset and synchronous vibration.

One limitation to disturbance observer based compensation is the requirement in most cases that disturbance characteristics or parameters be reasonably well known. If they are not, a

disturbance may not be modeled accurately. An adaptive controller based on an inaccurate model can lead to poor closed loop performance and instability. One approach to applying DOBC to a system with uncertain disturbance parameters was demonstrated by Wen [17]. He characterized disturbances as being composed of both known and unknown components and then developed a controller based on an "auxiliary" observer that estimated the latter. The controller provided good arbitrary disturbance attenuation.

Much of the earlier research on disturbance rejection and active magnetic bearing systems was focused on the identification and suppression of disturbances occurring at synchronous (rotor) frequencies. In fact, most of the references cited previously discussed the development and testing of controllers that were designed to eliminate the effects of synchronous disturbances exclusively. However, more recent work has extended the earlier work often by using novel control strategies and more sophisticated plant models to accommodate transient disturbances, multiple frequencies and base motion.

Burrows et.al. in a survey of work done on adaptive control of active magnetic bearings presented a unique controller based on open loop adaptive control (OLAC) principles that calculated control forces based on displacement measurements of the rotor [18]. The measurements were analyzed with a Fourier transform (the capability was built into the controller) to determine the amplitude and phase of the control forces. This approach to adaptive control did not require the development of a system model or any knowledge of the system parameters. One drawback to the controller was the delay in the application of the calculated control forces. Since the transform could not be completed during the current control cycle, the forces were applied to a following cycle.

The controller described in [18] was extended by Abulrub et.al. to improve responsiveness to a rapid and potentially destructive change in balance conditions [19]. Rapid response is critical to ensuring the integrity of rotating machinery if a sudden change in balance occurs. These researchers implemented a recursive version of OLAC termed recursive open loop adaptive control (ROLAC) that utilizes a recursive Fourier transform to speed up the

calculations of control forces. The forces can be determined during the current control cycle and applied to it rather than to a subsequent one.

Fourier coefficients, calculated in real time, have also been used in controllers designed to simultaneously suppress multiple disturbances, to improve transient response and to reduce the energy consumption of magnetic bearing systems. Cole et.al. developed a closed loop form of synchronous vibration control that used parallel loops of recursive Fourier transforms to determine the controller effort necessary to suppress multiple disturbances occurring at harmonics of the rotational speed [20]. Keogh et.al. used dynamic feedback of Fourier coefficients computed from rotor position to optimize the response of a closed loop controller to transient disturbances [21]. In addition, Sahinkaya and Hartavi used a recursive Fourier coefficient calculation to measure the orbit size of the rotor in a magnetic bearing system. The calculated size was compared to the optimal one developed analytically. The orbits were then adjusted accordingly to minimize the energy consumed by the system [22].

Efforts to develop bearing controllers capable of suppressing disturbances caused by base motion are not entirely new. Cole et.al. developed a controller capable of attenuating vibration caused by forces directly applied to the rotor as well as those indirectly applied through the supporting structure [23]. The controller design was based on the H_∞ control theory that made possible the development of a controller optimized for more than one input.

More recent approaches to dealing with base motion disturbances have been developed in response to the use of magnetically suspended devices in moving vehicles. The military has built an electro-optical sight for target tracking that uses magnetic bearings to improve stability. The sight is mounted in a moving vehicle, so base motion response must be addressed by the bearing controller. Kang et.al. describe the development of a sliding mode controller for the electro-optical sight that is capable of suppressing external disturbances even in the presence of system parameter variations [24].

A general method for rejecting disturbances that can be applied to a magnetic bearing system is discussed by Fuentes and Balas [25]. Their approach is based on a type of adaptive control

termed model reference whereby a plant subjected to persistent disturbances is directed to track a reference model with no disturbances applied. The frequencies of the disturbances must be known, but knowledge of their amplitudes is not required. In addition, the amplitudes can vary with time. Fuentes and Balas demonstrated the effectiveness of their approach, when applied to a general system, with numerical simulations. Subsequent research efforts by Matras and Barber have shown that the methods of [25] are equally as effective at identifying and rejecting disturbances acting on a flywheel system supported by active magnetic bearings [26], [27], [28].

Chapter 2 - Adaptive Disturbance Rejection Control

In this Chapter, the control laws developed by [25] to suppress or reject persistent disturbances acting on a general system will be briefly discussed. A simple, linear system subjected to a disturbance will serve as the basis for the discussion. A state space representation of the system will then be developed, and the results of computer simulations demonstrating the effectiveness of adaptive disturbance rejection (ADR) control applied to the system will be summarized.

2.1 Control Laws

A simple linear system with a persistent disturbance can be represented in state space according to Equations 2.1 and 2.2.

$$\dot{x}_p = A_p x_p + B_p u_p + \Gamma_p u_d \tag{2.1}$$

$$y_p = C_p x_p \tag{2.2}$$

The state, input and output terms are as expected. The last term in the state equation represents the disturbances applied to the system and is composed of a vector disturbance function u_d and a real valued matrix operator Γ_p that maps the disturbance onto the system state vector.

The elements of the disturbance function consist of linear combinations of scalar functions multiplied by amplitudes and unit vectors. The scalar functions can be constant or sinusoidal, or they can be other types of waveforms as long as the phase is known. For example, steady state errors and rotating imbalances can be represented by constant and sinusoidal scalar functions, respectively. Also, the phase of a sinusoidal disturbance does not have to be known if the disturbance is replaced by two sinusoidal ones that are 90^o out of phase.

Given the vector disturbance function u_d, a control law can be defined that uses adaptive techniques to suppress the effects of disturbances applied to a system. The control law is

based on the method developed by [25] and discussed in detail when applied to an active magnetic bearing system by [28] and is shown in Equation 2.3.

$$u_p = G_p y_p + H_p \varphi_d \tag{2.3}$$

The adaptive terms G_p and H_p are defined in Equations 2.4 and 2.5.

$$\dot{G_p} = -y_p y_p^T \Delta G \tag{2.4}$$

$$\dot{H_p} = -y_p \varphi_d^T \Delta H \tag{2.5}$$

The adaptive gain G_p is applied to the output of the system y_p, and adaptive gain H_p is applied to the vector of scalar functions φ_d, often referred to as the disturbance vector, described earlier. Gain G_p is essentially a stabilizing gain that responds to any nonzero output from the system. Gain H_p scales the disturbance vector to the amplitudes necessary to cancel the effects of the disturbances. Weighting matrices ΔG and ΔH determine how quickly the gains adapt to the disturbance. The transpose of vectors in Equations 2.4 and 2.5 is required to satisfy the necessary matrix algebra.

2.2 Simulated System

The linear state space model defined by Equations 2.1 and 2.2 can be extended to describe a rotating system by including displacements in both orthogonal directions X and Y. Only translational displacements will be considered, but rotational ones could be as well since ADR control rejects each type similarly. The displacements are assumed to be uncoupled, and the stiffness K and damping C in both directions are the same.

The state space representation for this system is given below where variable M represents the overall mass of the system:

$$\dot{z} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -K/M & 0 & -C/M & 0 \\ 0 & -K/M & 0 & -C/M \end{bmatrix} z_p + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} u_p + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/M & 0 \\ 0 & 0 & 0 & 1/M \end{bmatrix} u_d \tag{2.6}$$

$$y_p = [1 \quad 1 \quad 0 \quad 0] z_p \tag{2.7}$$

The state vector is ordered as follows:

$$z_p = \begin{bmatrix} z_{x1} & z_{y1} & z_{x2} & z_{y2} \end{bmatrix}^T = [x \quad y \quad \dot{x} \quad \dot{y}]^T \tag{2.8}$$

The control output vector is made up of the system or plant inputs required to suppress the disturbance in both the X and Y directions:

$$u_p = [0 \;\; 0 \;\; u_{p,x} \;\; u_{p,y}]^T \tag{2.9}$$

The disturbance applied to the simulated system is a rotating imbalance that represents the less than perfect balance that characterizes nearly all rotating systems. The imbalance is modeled as a point mass *m* located at a fixed distance *l* from the center of rotation. The imbalance generates a force with an amplitude *F* defined by Equation 2.10 where variable ω denotes the rotational speed of the system.

$$F = ml\omega^2 \tag{2.10}$$

Disturbance function u_d includes the disturbance forces acting on the system and is composed of two sinusoids with amplitude *F* and frequency ω as shown below:

$$u_d = [0 \;\; 0 \;\; F \sin \omega t \;\; F \cos \omega t]^T \tag{2.11}$$

Disturbance vector φ_d is also formed by sine and cosine functions of the same frequency ω:

$$\varphi_d = [0 \;\; 0 \;\; \sin \omega t \;\; \cos \omega t]^T \tag{2.12}$$

The block diagram for the simulated system with adaptive disturbance rejection control is shown in Figure 2-1.

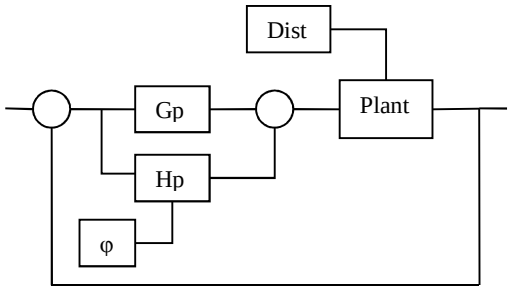


Figure 2-1 Plant with ADR Controller

The reference input is set to zero since no displacement of the system is desired, and the disturbance input is composed of the two persistent sinusoids. The controller includes only that part necessary for disturbance rejection, and feedback is provided by the measured displacements of the plant.

2.3 Simulation Results

Using the system model described by Equations 2.3 through 2.12, computer simulations were performed to demonstrate the effectiveness of adaptive disturbance rejection control. A MATLAB program (See Appendix A for a listing.) was written that determines the model's response to the imbalance disturbance with and without ADR control. The model represents an unbalanced rotating system, and the imbalance is the applied disturbance. Therefore, the frequency of the disturbance and the rotational speed of the system were the same.

Parameters that must be specified to the program for each simulation include the mass, natural frequency and damping ratio of the system, the frequency and magnitude of the disturbance and values of the weighting matrices ΔG and ΔH for the controller. Many of the inputs required for the program were based on previous work by Matras et.al. [29]. For example, the damping ratios and disturbance forces used for most of the simulations were quite small. Damping ratios less than 0.05 and disturbance forces in the vicinity of 0.1 N were generally used. However, much larger values of each were also shown to have little effect on the ability of the adaptive methods to suppress the disturbances.

Adaptive gain G_p is typically necessary to provide initial stability [29] and could be expected to have little effect on this model. In fact, a number of simulations were run that showed this to be the case. Therefore, matrix ΔG was set equal to the identity matrix for each simulation. Gain H_p is then primarily responsible for suppression and accordingly, the value of ΔH was varied to provide the fastest rejection of the disturbance for each test. In addition, disturbance frequencies above, below, and near the natural frequency of the system were tried.

Characteristic values and results for a representative simulation appear in Figure 2-2.

Mass = 5 kg
Imbal. Force = 0.04 N
Nat'l Freq. = 20 Hz
Dist. Freq. = 10 Hz
ΔH = -28,000

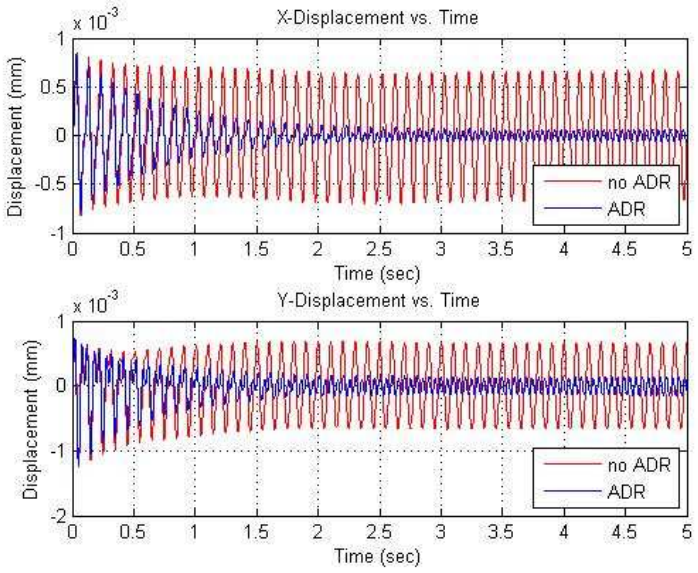


Figure 2-2 Representative Disturbance Rejection - Case 1

For the Case shown in Figure 2-2, displacements in both axis directions are suppressed in about two seconds. Note that the sign on the ΔH term must be negative when the disturbance frequency is below the natural frequency of the system [27].

Mass = 5 kg
Imbal. Force = 0.13 N
Nat'l Freq. = 20 Hz
Dist. Freq. = 18 Hz
ΔH = -22,000

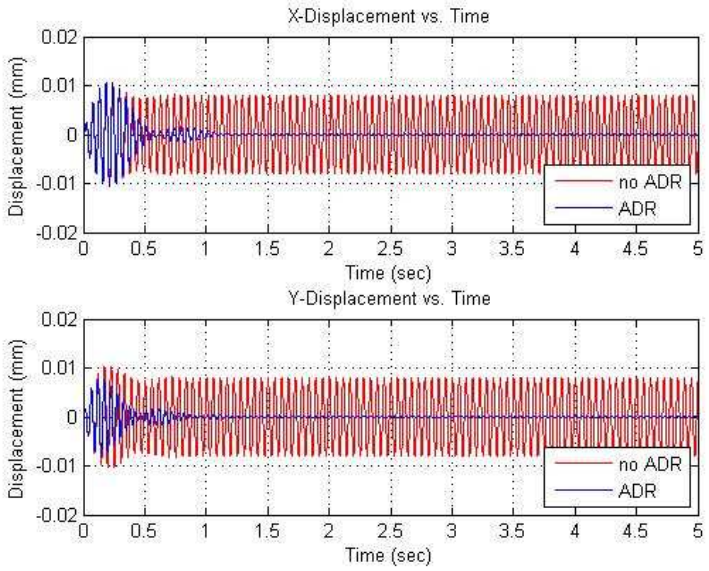


Figure 2-3 Disturbance Rejection - Case 2

Figure 2-3 (2-4) illustrates the rejection of a disturbance with a frequency just less (more) than the natural frequency of the system. For the simulations shown in Figures 2-3 and 2-4, displacements caused by disturbances acting near the natural frequency of the system are rejected more quickly than they were in Case 1 despite an order increase in their magnitude. The imbalance mass was the same for Cases 1 through 3, but the imbalance force was different in each Case because the rotational speed was different in each also. Note again, that the sign on matrix ΔH must be negative for the results shown in Figure 2-3 and positive for those shown Figure 2-4.

Mass = 5 kg
Imbal. Force = 0.19 N
Nat'l Freq. = 20 Hz
Dist. Freq. = 22 Hz
ΔH = 22,000

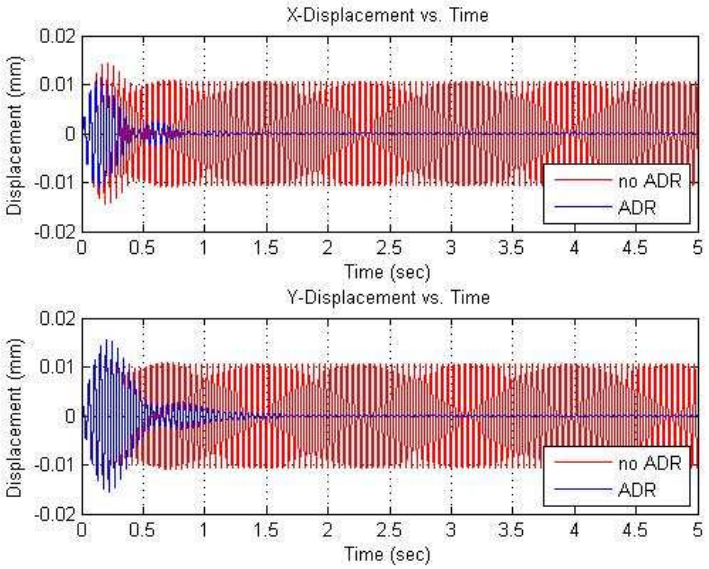


Figure 2-4 Disturbance Rejection - Case 3

The next simulation (Figure 2-5.) illustrates the Case for when the disturbance frequency is a multiple of the natural frequency of the system. Again, the disturbance is suppressed as quickly as it was in the preceding simulations. Here, the imbalance mass was smaller than that used previously so that the imbalance force despite the higher frequency would remain within the range of the forces from prior simulations.

Mass = 5 kg
Imbal. Force = 0.06 N
Nat'l Freq. = 20 Hz
Dist. Freq. = 40 Hz
ΔH = 80,000

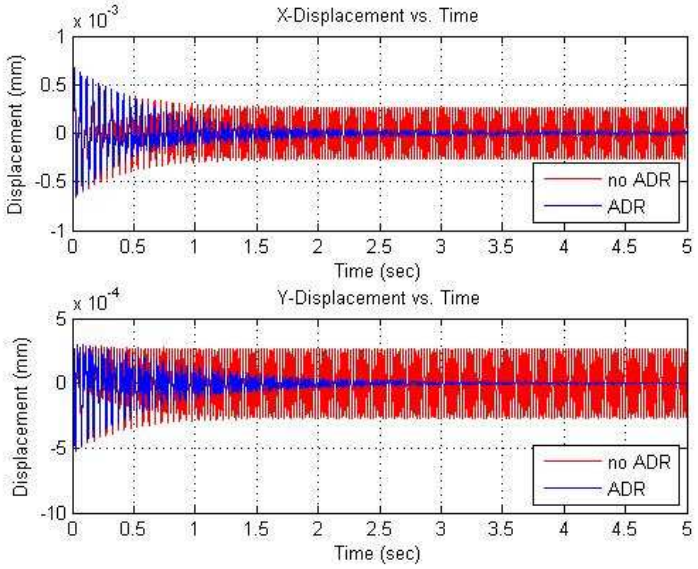


Figure 2-5 Disturbance Rejection - Case 4

Finally, the results shown in Figures 2-6 and 2-7 demonstrate that neither the magnitude of the disturbance force nor the amount of damping present in the system affects the ability of adaptive methods to suppress disturbances.

Mass = 5 kg
Imbal. Force = 0.40 N
Nat'l Freq. = 20 Hz
Dist. Freq. = 10 Hz
ΔH = -28,000

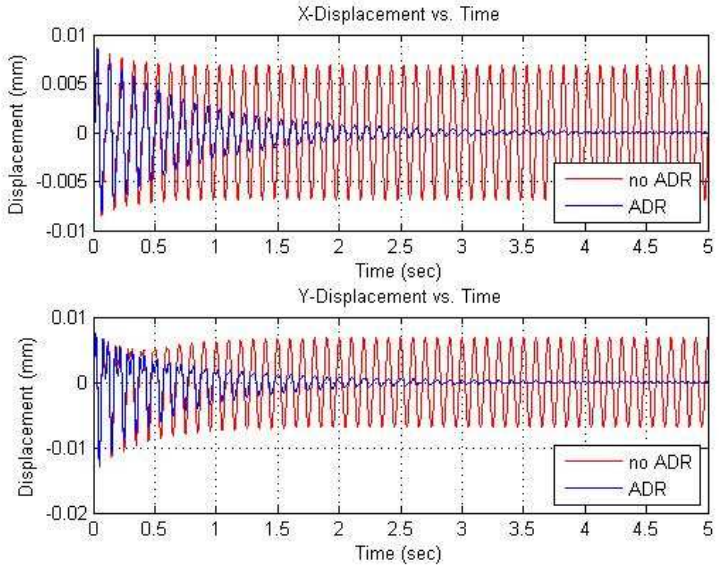


Figure 2-6 Disturbance Rejection - Case 5

For Case 5, the force was increased by one order over the first simulation, yet the time required to reject is identical for the two tests (Note that damping and ΔH are the same.). For the final case, illustrated in Figure 2-7, a damping ratio of 0.30 was used as compared to a value of 0.02 in the first simulation. As can be seen, the times to reject were nearly the same regardless of the amount of damping present. However, the value of ΔH had to be adjusted to account for the greater damping.

Mass = 5 kg
Imbal. Force = 0.04 N
Nat'l Freq. = 20 Hz
Dist. Freq. = 10 Hz
ΔH = -40,000

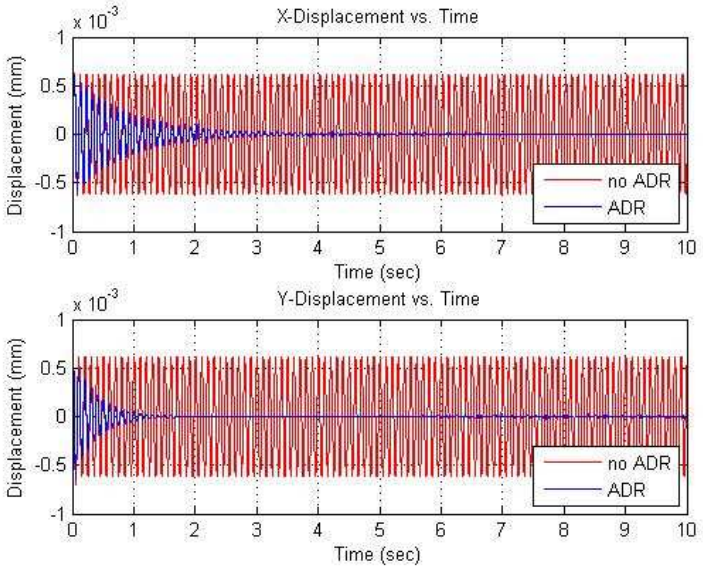


Figure 2-7 Disturbance Rejection - Case 6

It's also instructive to observe the response of the adaptive gain H_p during disturbance rejection. A graph of the gain as a function of time for the first case discussed is shown in Figure 2-8 (Graphs of H_p for the other cases are very similar to the graph for this case).

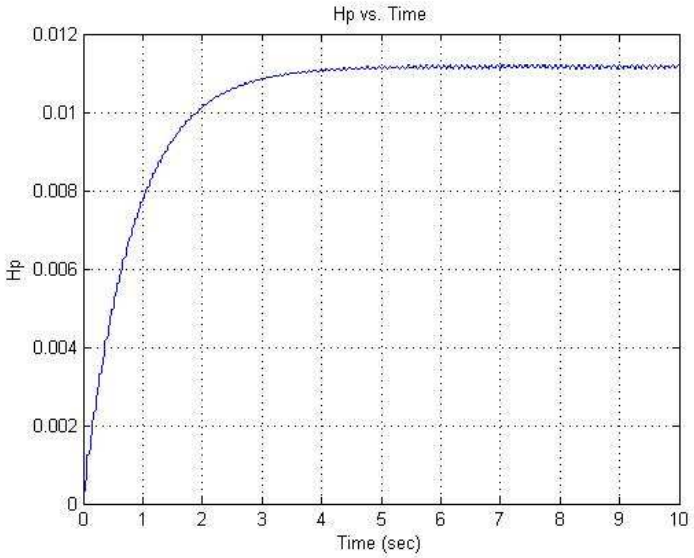


Figure 2-8 Gain H_p - Case 1

Notice that the gain steadily increases until the disturbance is suppressed at which point the gain settles to a nearly constant value. Researchers have investigated whether a change in the

slope of or the appearance of a discontinuity in the trace of H_p, once the gain has adapted to a disturbance, indicates a change in the balance of a system [26], [27], [28]. If a relationship can be established between H_p and a balance state, then a change in the gain could demonstrate a change in the balance of a system that could occur as a result of the development of a crack or defect. Thus, gain H_p could be used as a predictor of the overall health of a system.

Chapter 3 - Existing System

In this Chapter, an overview of the magnetic bearing system as it existed at the outset of this research is presented. The overview includes brief summaries of the bearing and speed-control hardware, the dSPACE system, the Simulink bearing/speed controller and the ControlDesk instrument panels. In addition, summaries are also provided of the significant changes and enhancements made to the system as part of this research. Subsequent Chapters will discuss these changes in detail.

3.1 Bearing and Speed-Control Hardware

The magnetic bearing system was originally designed and developed by the Air Force Research Laboratory (AFRL) as part of the Flywheel Attitude Control Energy Transmission System (FACETS) program. The AFRL was created in 1997 through the consolidation of several laboratories and a research office. The FACETS program included investigation into the use of magnetically suspended flywheels for both energy storage in and attitude control of space vehicles. At the conclusion of the FACETS program, the entire system was donated to Auburn University.

The magnetic bearing system largely as it was received from the AFRL is shown in Figure 3-1. (Photo courtesy of R. Jantz [30].)

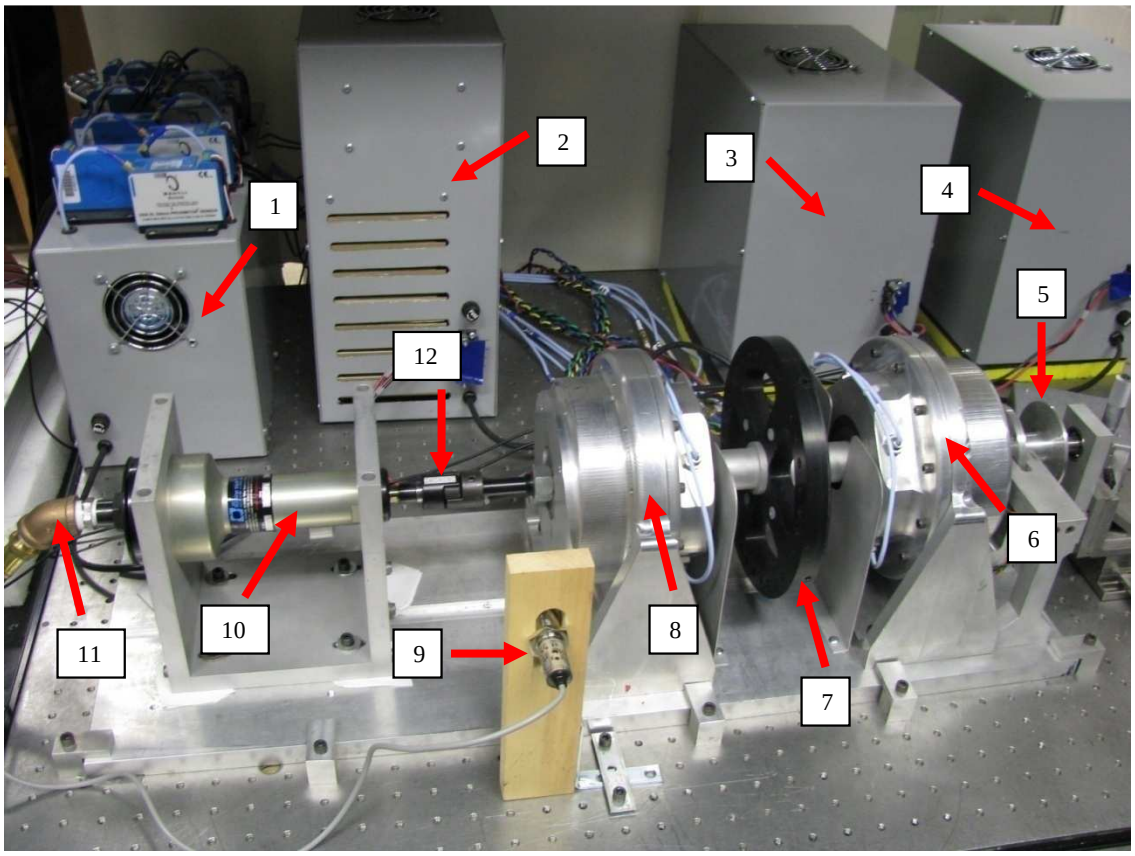


Figure 3-1 Magnetic Bearing System

The major elements, identified by numbered arrows, are:

1) proximity system - provides measurements of the displacement of the rotor along the bearing axes,
2) amplifiers - modulate the output of the power supplies to provide the current necessary to energize the magnetic bearings,
3) power supply,
4) power supply,
5) sensor - provides the input signal to the mechanical tachometer,
6) magnetic bearing one - contains four radially disposed electromagnets for supporting the free end of the rotor,
7) flywheel,

8) magnetic bearing two - contains four radially disposed electromagnets for supporting the motor (turbine) end of the rotor,
9) sensor - provides the input signal to the electronic tachometer,
10) air turbine - powers or spins the rotor and flywheel,
11) air supply line and
12) flex coupler.

Further details about the hardware can be found in [30], and a comprehensive discussion of the FACETS system including details of its design and construction can be found in [26] and [28].

Hardware necessary to support the addition of speed control to the original magnetic bearing system is pictured in Figure 3-2 (Photo courtesy of [30].)

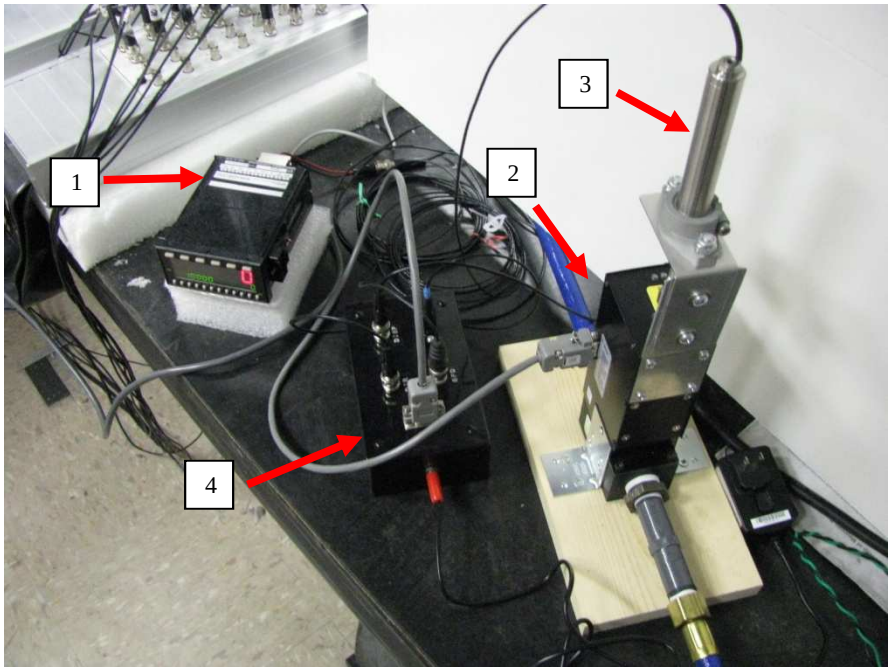


Figure 3-2 Speed Control Hardware

and includes the following items identified by the numbered arrows:

1) electronic tachometer - measures the rotational speed of the rotor,

2) stepping motor valve (SMV) - regulates the flow of compressed air to the air turbine,
3) linear voltage differential transformer (LVDT) - provides measurements of the position of the SMV from fully closed to fully open and
4) project box - provides the electrical connections between the analog-to-digital (A/D) converter and the SMV.

For a complete discussion of the development and implementation of the speed control system, refer to [30].

Figures 3-1 and 3-2 depict the magnetic bearing system almost exactly as it exists today since the hardware with two minor exceptions was not changed for this project. The exceptions include careful realignment of the major components and the replacement of the flex coupler between the air turbine and bearing two to improve the rotor orbits (See Section 4.1.).

3.2 dSPACE System

The interface between the host computer and the bearing/speed-control hardware is provided by a system manufactured by dSPACE [31]. Although no changes were made to the dSPACE system during this project, summaries of the major hardware and software elements of the system follow.

The dSPACE hardware includes the components listed and briefly described below:

1) DS1005 processor - executes the control code for the bearing system generated from the Simulink bearing/speed controller block diagram,
2) DS2003 analog-to-digital (A/D) converter - converts analog signals received from the hardware (e.g. proximitors and tachometers) to digital for input to the processor,
3) DS2002/2003 A/D connector panel - provides physical connections for input signals,
4) DS2103 digital-to-analog (D/A) converter - converts digital signals sent from the processor (e.g. control currents and signals) to analog for output to the bearing/speed control hardware and
5) DS2103 D/A connector panel - provides physical connections for output signals.

The processor, A/D converter and D/A converter are enclosed in an expansion box that is pictured in Figure 3-3.

Figure 3-3 dSPACE Expansion Box

The A/D and D/A connector panels are shown in Figure 3-4. (Photo courtesy of [30].)

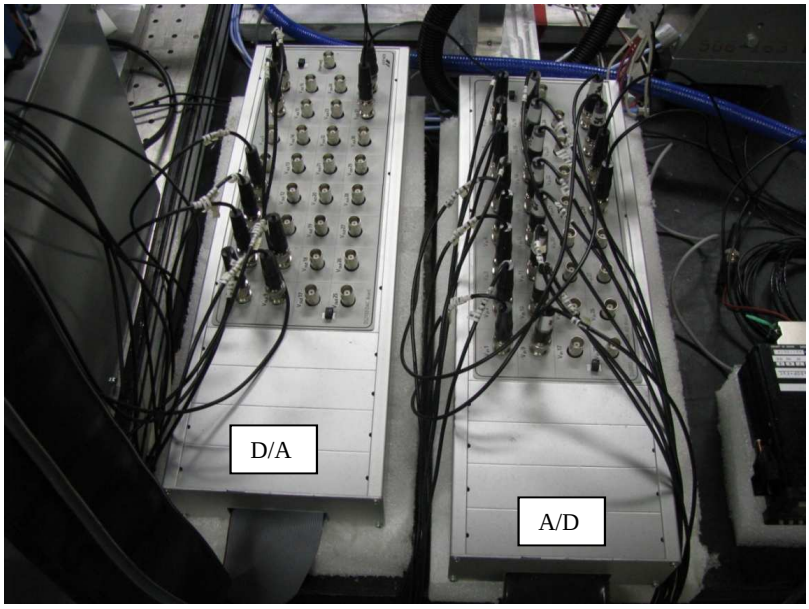


Figure 3-4 A/D and D/A Connector Panels

The dSPACE software consists of the components listed and briefly described below:

1) Real-Time Workshop - generates C-language programs from Simulink block diagram models, compiles the programs and loads the resulting executables on the dSPACE processor,
2) Real-Time Interface - provides the link necessary for connecting Simulink block diagrams to dSPACE hardware in block library form and
3) ControlDesk - provides software for creating layouts and instrument panels used to manage and control systems.

ControlDesk allows Simulink block variables to be displayed and changed through a number of different instruments so that systems can be monitored and controlled in real time. Variables that can be displayed through ControlDesk instruments include outputs from Simulink blocks, while those that can be changed include configuration parameters for Simulink blocks and arguments passed to S-functions. Several panels developed with ControlDesk are shown in this document (See Figures 4-14, 4-15 and 4-16.).

3.3 Simulink Bearing/Speed Controller

The modeling and development of the control system for the magnetic bearings were done by [26] using Simulink. This system includes both a proportional-integral-derivative (PID) controller for actively managing the bearings and an adaptive-disturbance-rejection (ADR) controller for neutralizing the effects of disturbances that can upset the balance of the rotor but that cannot be offset by PID control alone. The modeling and development of the PID speed control system were done by [30] also using Simulink. The bearing and speed controllers were combined into a single system to produce the bearing/speed controller. The top-level Simulink block diagram for this is shown in Figure 3-5.

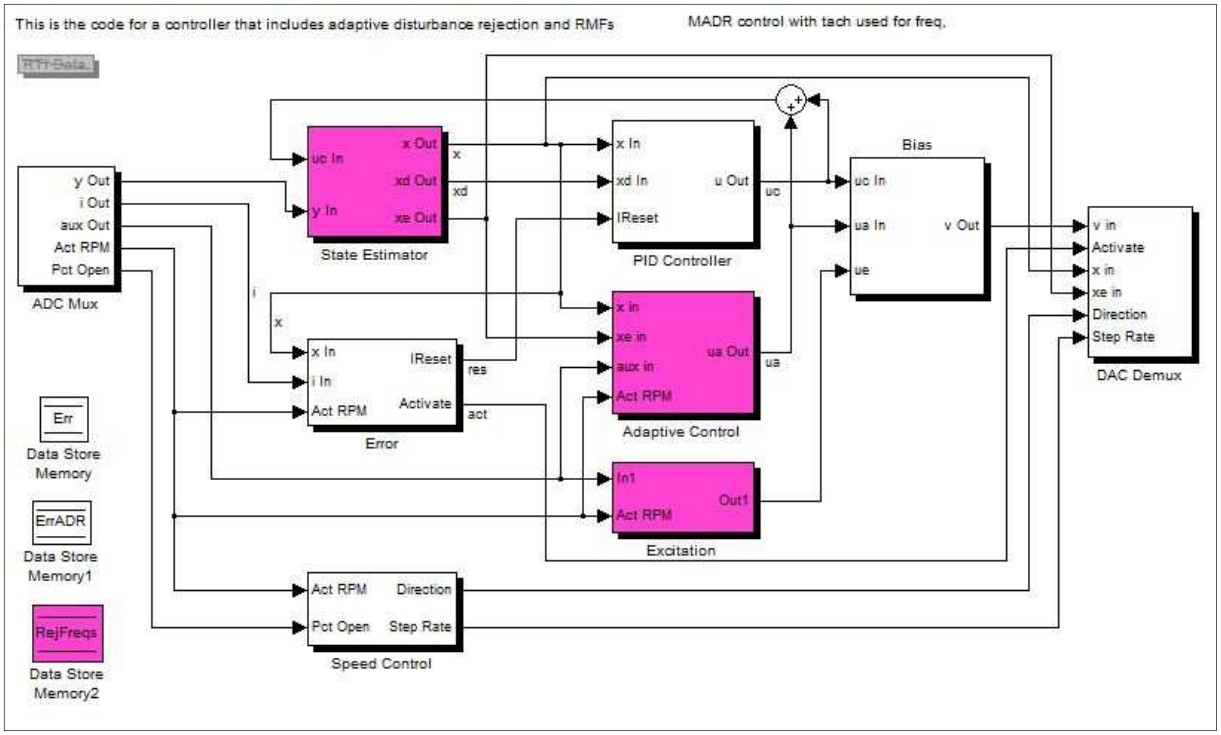


Figure 3-5 Bearing/Speed Controller

The diagram pictured here represents the controller as it appeared at the beginning of this research with two exceptions: the rotor speed signals added as inputs to the *Adaptive Control* and *Excitation* subsystems and the third data store added to support the Discrete Fourier Transform (DFT) driven adaptive controller. The colored subsystems or blocks represent those that were changed or added over the course of this research.

In this section, the purpose of each subsystem is briefly described, and the changes made to each are briefly summarized. Detailed discussions of the changes to the subsystems are provided in the following Chapter.

Adaptive Control

The *Adaptive Control* subsystem calculates the adaptive gains and the controller output necessary to suppress the effects of potentially damaging disturbances. The inputs to this subsystem are measured rotor position, estimated states (rotor position and speed), any

auxiliary inputs applied through the *ADC Mux* block and the actual speed of the rotor. The outputs of this block are the adaptive control signals for the bearings.

This subsystem was changed so that adaptive control signals are calculated and applied to all four axes of the bearing system, and this change was made to extend the research capabilities of the system. With adaptive control available on all axes, the predictive capability of the adaptive gains regarding moments produced by imbalance forces could also be investigated.

Safety features were also added to protect the system from damage and to calm the nerves of researchers in case of miscalculated or unanticipated controller behavior. These features null the contribution of the adaptive gains to the controller output if either or both gains saturate and disable the adaptive controller if calculated currents or measured displacements exceed thresholds.

ADC Mux

Subsystem *ADC Mux* is the Simulink interface to the dSPACE hardware analog-to-digital (A/D) converter. This block collects the inputs to the system and routes them to other subsystems for processing. The inputs include eight from the Proximitor probes that measure the position of the rotor with respect to the magnetic bearings, eight from the amplifiers that provide the current to power the bearings, two from the tachometers that measure the speed of the rotor and one from the flow-control valve that indicates the position of the valve from fully closed to fully open. No changes were made to this subsystem.

Bias

Bias outputs the control voltages applied to the bearings by combining inputs from several different sources. The PID and adaptive controllers provide inputs necessary to keep the rotor suspended. Other sources include user-specified bias voltages that can be used to shift the rotor towards one magnet or the other. Yet another source is the *Excitation* subsystem. Voltages input from this block are used to alter the orbit of the rotor to simulate various types of disturbances. No changes were made to this block.

DAC Demur

Subsystem *DAC Demux* is the Simulink interface to the dSPACE digital-to-analog (D/A) converter. Inputs to this subsystem are bearing control voltages output from *Bias*, a rotor status signal from *Error*, the measured and estimated positions of the rotor with respect to the magnets and two control signals for the flow-control valve. These signals determine the direction of valve motion (open or close) and the rate of opening or closing.

There are 10 control voltages output from *DAC Demux*. Eight are routed to the amplifiers shown in Figure 3-1 and converted to the currents required by the magnetic bearings to float the rotor. The other two are sent to the project box that provides the interface between the connector panel and the flow-control valve. No changes were made to this subsystem.

Error

The *Error* block detects current and position errors and takes precautionary actions to prevent damage to the bearing system if thresholds for current or position are exceeded. Typically, the actions taken are informational; status messages warn the user of deteriorating conditions. However, error states are numerical coded and stored in the *Err* and *ErrADR* data storage areas shown in Figure 3-5. These codes can be used by other subsystems to alter their behavior.

Inputs to *Error* are the measured position of the rotor, the currents flowing to the bearings and the actual speed of the rotor. Outputs include a signal to reset the integral gain and another signal that prior to being neutralized by [30] could be used to deactivate the bearings. This signal was disabled to prevent damage to the bearings that can result if deactivation occurs while the rotor is spinning. No changes were made to this subsystem.

Excitation

The *Excitation* subsystem as briefly noted in an earlier paragraph provides a means to excite the rotor with sinusoidal signals. These are either generated by this block based on user-specified values for amplitude and frequency or are input to the block via a dynamic signal

analyzer. In the latter case, *Excitation* simply adds the input directly to any signals generated within the subsystem to form the output.

This subsystem was changed to address two problems identified by [28] in his research on the predictive capability of adaptive gain H_p. One difficulty was synchronizing the excitation frequency output from this block with the actual speed of the rotor. The other was establishing the exact time when the excitation was applied by the block. With the changes made to *Excitation*, the frequency of the sinusoidal signals can be updated continuously to exactly match the speed of the rotor. The changes also make it possible to capture the complete duty cycle of the signals, so the precise moment when the excitation is applied and the precise moment when it is removed are both known.

Phi

The *Phi* subsystem computes the disturbance vector required to determine the adaptive gain H_p. The input to this block is the actual RPM of the rotor, and the output is the disturbance vector. *Phi* is actually a subsystem within the *Adaptive Control* block and does not appear in the top-level Simulink diagram shown in Figure 3-5.

This subsystem evolved over the life of the research. The block was originally enhanced so that the disturbance vector could be computed on the basis of the actual speed of the rotor and multiples of that speed or on the basis of the dominant frequencies calculated by the real-time DFT. These methods were in addition to the original one where the frequencies to reject were entered manually through ControlDesk. Regardless of the basis, three frequencies were used in the construction of the vector.

However, during development and testing, task overrun errors were encountered that destabilized the bearing controller. Thus, this subsystem was streamlined to use just a single frequency as just one measure taken to eliminate the overrun errors. (Overrun errors as well as all of the steps taken to overcome them are discussed in detail in Chapter 5.) This frequency could still be entered manually or it could be based on the speed of the rotor or on the output of the DFT.

Reducing the disturbance vector to a single frequency did not compromise the usefulness of the system significantly. The primary focus of this and similar research is often the imbalances that occur at the rotational speed of the system. However, further testing has indicated that the refinement and tuning that were done to eliminate task overruns may well have created a sufficient time margin that would allow the vector to be expanded beyond a single frequency without reintroducing overrun errors.

PID Controller

The *PID Controller* block implements proportional-integral-derivative control on the active magnetic bearings. The measured and integrated positions of the rotor and the estimated velocity of the rotor are the inputs to the subsystem. The sole output is a control signal. This subsystem was constructed so that different controller gains could be applied to each bearing. No changes were made to this block.

Speed Control

Speed Control is the subsystem that controls the speed of the rotor by implementing proportional, derivative and integral control. Inputs to this block are the actual speed of the rotor and the position of the flow control valve expressed as a percentage of fully open. Outputs from the block are the direction to move the valve (open or close) and the rate at which to move it.

This subsystem provides a number of user-configurable options such as those that determine the minimum and maximum opening and closing rates and the maximum sizes of the dead bands above and below the set operating speed. In addition, *Speed Control* supports full manual operation of the valve for instances where placing and maintaining the valve at a fixed position are needed. No changes were made to this block.

State Estimator

Inputs to *State Estimator* are the rotor position from *ADC Mux* and the control current to the bearings. Other inputs are calibration values, and these are added to the current inputs in this

block. Calibration is the procedure that centers the rotor exactly between the magnets and calculates any additional currents necessary to do so. This subsystem determines actual rotor position and estimates the position and velocity states for input to other subsystems.

A real-time Discrete Fourier Transform (DFT) was incorporated into this subsystem to provide among other things the capability of identifying disturbances that result from base motion. Matras [26] demonstrated that the adaptive control laws summarized in Chapter 2 can be applied to suppress disturbances produced by base motion. These would more than likely occur at unknown frequencies and possibly bear no relation at all to the speed of the rotor. One method to determine these frequencies would be to perform a spectral analysis or Fourier transform of the rotor's position relative to a bearing axis.

The DFT was included in this block since the actual position or displacement of the rotor, measured in thousandths of an inch, along each bearing axis (the rotor position) is available here. Outputs from the DFT can be and were used to drive the *Phi* subsystem, the block responsible for forming the disturbance vector required by the adaptive controller. In addition, the DFT provides a direct and nearly immediate measurement of the effectiveness of disturbance rejection.

3.4 ControlDesk Instrument Panels

The ControlDesk instrument panels used to operate and control the FACETS system are well documented and illustrated in [30]. Three additional panels were developed for this project to primarily support the configuration and management of the ADR controller. The purpose of each additional panel is discussed in the following Chapter, and Figures illustrating each are shown in this Chapter as well.

Chapter 4 System Enhancements

Major components of the FACETS system, including the bearing hardware, the Simulink bearing/speed controller and the ControlDesk instrument panels, were changed or enhanced to:

- improve the behavior of the rotor,
- extend the capabilities of the adaptive controller,
- ensure the safer operation of the system and reduce the potential for damage to it,
- increase the capacity of the system to serve as a research platform and
- maintain the consistency of operation through the use of established norms,

and these changes and enhancements are discussed in detail in this Chapter.

4.1 Bearing Hardware

Changes were made to the bearing hardware to improve the orbits of the rotor within each bearing. Ideally, the orbits should be small, centered and circular as shown in Figure 4-1 and should remain nearly so as the speed of the rotor increases.

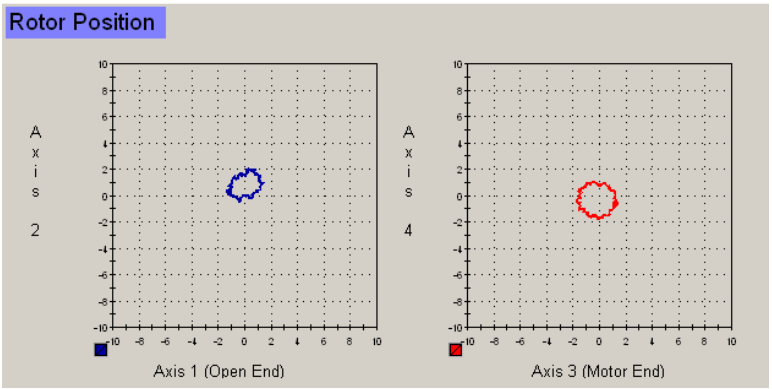


Figure 4-1 Ideal Bearing Orbits

However, as this project progressed, the orbit at the motor or driven end deteriorated, becoming large, off-centered and irregularly shaped as shown in Figure 4-2. (Both Figures

illustrate the distance, measured in thousandths of an inch (mils), that the rotor is displaced along each bearing axis.)

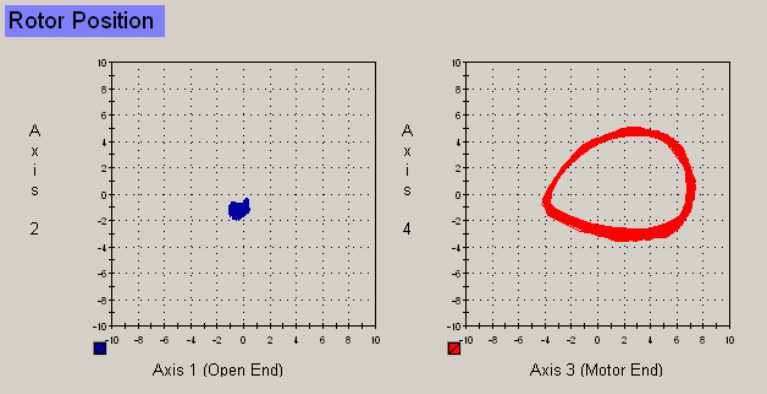


Figure 4-2 Less Than Ideal Bearing Orbits

In addition, the orbit at the motor end grew alarmingly in size as the rotor speed increased, becoming large enough at 5000 RPM to nearly contact the magnets and proximitors. Orbits of this size triggered a limit exception resulting in the display of a warning to the user through ControlDesk.

Investigations revealed that the causes of the poor orbits were misalignment between the air turbine and the rotor and a flex coupler that could not accommodate the misalignment and that also influenced the orbits negatively. Some misalignment, both parallel and angular, will nearly always be present between the turbine and the rotor. The turbine is solidly mounted, but the rotor orbits within the bearings. Even if the two are perfectly aligned when the rotor is suspended but not turning, they will become slightly misaligned as soon as the rotor begins to spin. Therefore, the turbine and rotor were carefully aligned first. Then, tests were performed with several different types of flex couplers to determine which one would result in the best orbits possible.

The turbine and rotor were aligned (with the flex coupler that was ultimately chosen in place) by placing shims under the driving end of the turbine and under the brackets that supported the turbine. The shimming was done with the rotor suspended but idle. When finished, the

driving end of the turbine was raised 0.002 inch, and the brackets were raised 0.003 inch at the driving end and 0.005 inch at the driven end. The shim stock used was non-conductive, non-magnetic brass of 0.001 inch thickness.

Experiments were performed with four different types of flex couplers, including double-loop, flexible-spider (the existing one), helical-beam and pinhole-disc, to identify an optimal one. The first three are shown in Figure 4-3.

Figure 4-3 Flex Couplers

In qualitative terms, the coupler must be stiff torsionally to reduce the risk of inducing torsional vibrations, stiff axially to control any thrust, however small, that may develop and compliant laterally to prevent small misalignments from affecting the bearing orbits. Only the pinhole-disc coupler met these requirements; the other three did not. Figure 4-4 shows this coupler in place on the FACETS system.

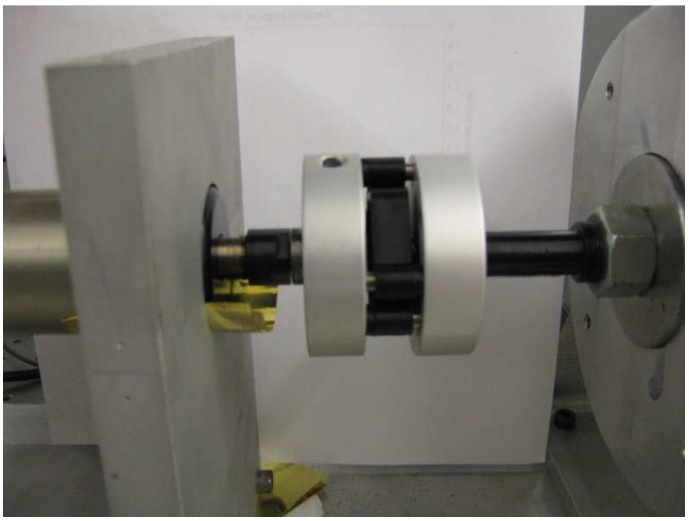


Figure 4-4 Pinhole-Disc Flex Coupler

4.2 Simulink Bearing/Speed Controller

As noted in the previous Chapter, the four subsystems of the Simulink bearing/speed controller that were changed significantly are *Adaptive Control*, *Excitation*, *Phi* and *State Estimator*. The changes made to each are detailed in this Section. In addition, the Simulink diagrams for these subsystems before and after the changes were made are shown. The diagrams for the subsystems that remained as they were can be found in [27].

Adaptive Control

Modifications were made to the *Adaptive Control* block to:

- extend adaptive control to all four axes,
- null the contribution of each adaptive gain to the controller output if the gain saturates,
- disable the adaptive controller if current or position limits are exceeded,
- reduce the composition of the disturbance vector to a single frequency and
- properly calculate the magnitude of adaptive gain H_p.

The original *Adaptive Control* subsystem as developed and refined by [27] is shown if Figure 4-5.

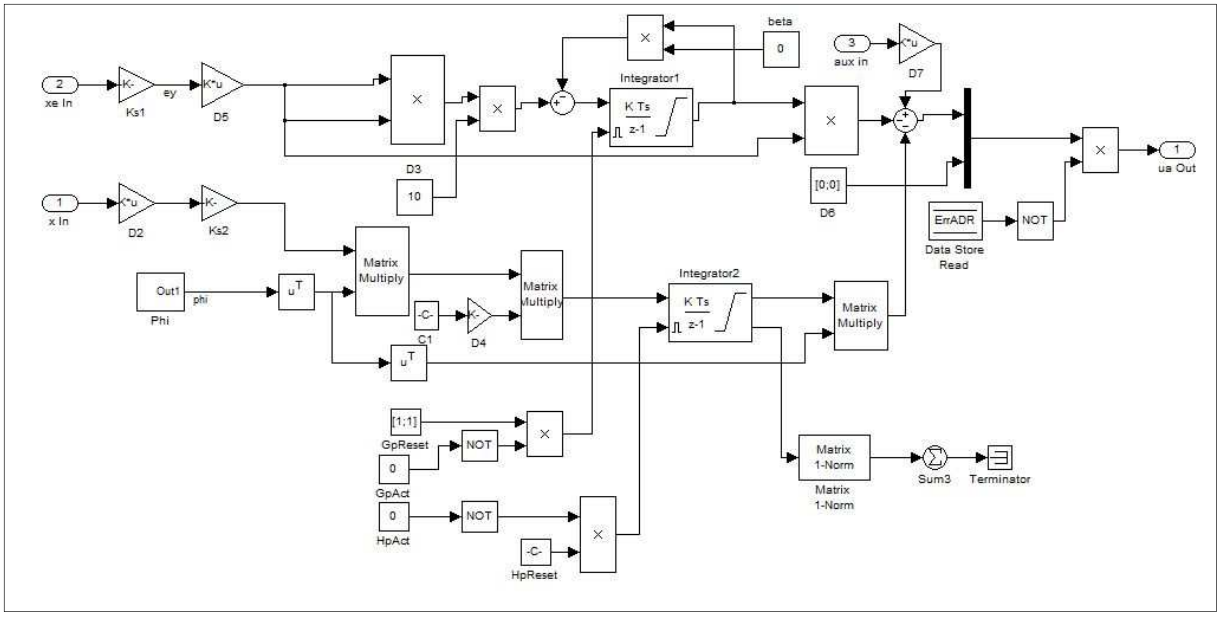


Figure 4-5 Original *Adaptive Control* Subsystem

The current *Adaptive Control* subsystem is pictured in Figure 4-6, and the colored blocks identify those that were changed from or added to the original subsystem.

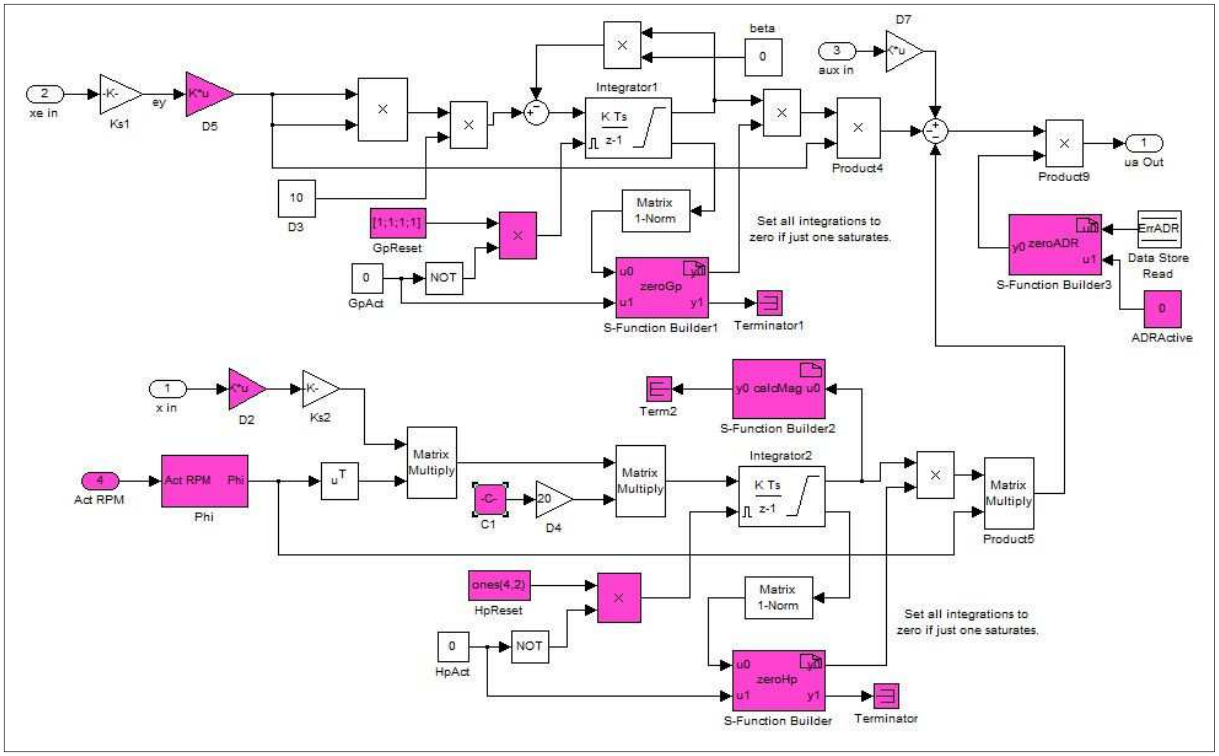


Figure 4-6 Current *Adaptive Control* Subsystem

The capability of applying adaptive control to all four axes was largely in place on the existing system. The gain blocks were already sized such that the outputs from the adaptive controller would match those from the other subsystems that acted on all axes. Therefore, it was only necessary to identify the matrix elements in each gain block that when set to their proper values would enable adaptive control on axes three and four. Also, the blocks necessary to reset the integrators to their initial condition, the method used by [27] to disable the adaptive gains, had to be similarly modified to accommodate the additional axes.

The bearing system experienced instability at very low speeds during the initial experiments with the predictive capability of the adaptive gains. The instability occurred immediately after the adaptive controller was activated and was independent of the mode (manual, DFT or speed) used to drive the controller. It appeared that the adaptive gains saturated very quickly, and it was thought that rapid saturation may have caused or contributed to the instability. (The actual cause of the instability and the steps taken to eliminate it are discussed in Chapter 5.)

Thus, the adaptive controller needed to be disabled as quickly as possible if the gains did indeed saturate, and S-functions *zeroGp* and *zeroHp* were developed to accomplish this.

S-functions or system functions are one way to greatly extend the capabilities of a Simulink model by providing a method for including user-written C programs into a model. Languages other than C can be used as well. The programs can be added through a standard Simulink block, typically S-Function Builder or S-Function. Both are available in the Simulink/User Defined Functions library.

S-Functions like other blocks in Simulink require that initialization and termination tasks be performed at the beginning and end of a simulation. Examples of the former include initializing the configuration structure, setting the number and size of input and output ports and allocating storage. An example of the latter includes freeing any memory allocated specifically for the block. The S-Function Builder block automatically adds the code necessary to perform these tasks. The S-Function block does not.

S-functions *zeroGp* and *zeroHp* were written using the S-Function Builder since no special tasks exclusive of those provided by the Builder were required to initialize or terminate the blocks at the beginning or end of a simulation. In fact, all S-functions developed for this project were incorporated into the bearing/speed controller model using the Builder rather than the block. If the Builder cannot be used, significantly more effort is required on the part of the programmer.

S-Functions *zeroGp* and *zeroHp* work identically. Each monitors the output from an integrator's saturation port. If the integrator saturates, the output from it is nulled and remains nulled, even if the integrator becomes unsaturated, until it is reset manually. See Appendix B for listings of both programs and detailed comments on the programs' logic.

The *Adaptive Control* subsystem was further modified to support explicit enabling and disabling of the adaptive controller through ControlDesk. Previously, the controller was enabled or disabled by sending the proper signals to the integrators' reset ports. Explicit enabling/disabling was combined with the limit detect function developed by [27] to maintain

the automatic disabling capability of the controller through the *Error* subsystem. The changes to *Adaptive Control* described in this paragraph were implemented via S-function *zeroADR*. See Appendix B for a listing of this program.

The last two changes made to the *Adaptive Control* subsystem were made to accommodate the resized disturbance vector and to calculate the magnitude of gain H_p. (Section 3.3 explains why this vector was resized to include only a single frequency.) The resized vector entailed changes to those blocks that compute H_p to ensure that matrix algebra is performed properly. Gain H_p was computed directly (by S-function *calcMag*) so that it would be available for data capture and subsequent analysis. A listing of *calcMag* appears in Appendix B.

Program *calcMag* could have been more easily implemented as an embedded MATLAB function, and in fact, it was initially. The program is straight forward, simply calculating the square root of the sum of squares. In addition, the program's computations do not depend on those from previous invocations. If they did, an S-function would be needed to provide the necessary storage class for the variables used. However, embedded MATLAB functions execute much more slowly than S-functions do. The embedded functions are interpreted rather than compiled and placed in-line as the S-functions are. Since the real-time constraints of the bearing/speed controller were becoming increasingly difficult to satisfy as the controller grew in scope and complexity, the execution time of each subsystem and each block had to be as fast as possible. Therefore, whenever user-developed programs were required, they were added as S-functions rather than embedded MATLAB functions.

Section 5.2 discusses the profiling of subsystems that was done to help identify ways to reduce the execution or run time of each. The results shown in this Section demonstrate the significant reduction in the run time of subsystem *Adaptive Control* when embedded MATLAB function *calcMag* was replaced with an equivalent S-function.

Excitation

Modifications were made to the *Excitation* block to:

- provide the capability to exactly match the frequency of the excitation to the speed of the rotor and
- exactly identify the time when the excitation is applied to and removed from the rotor.

The original *Excitation* subsystem is shown if Figure 4-7.

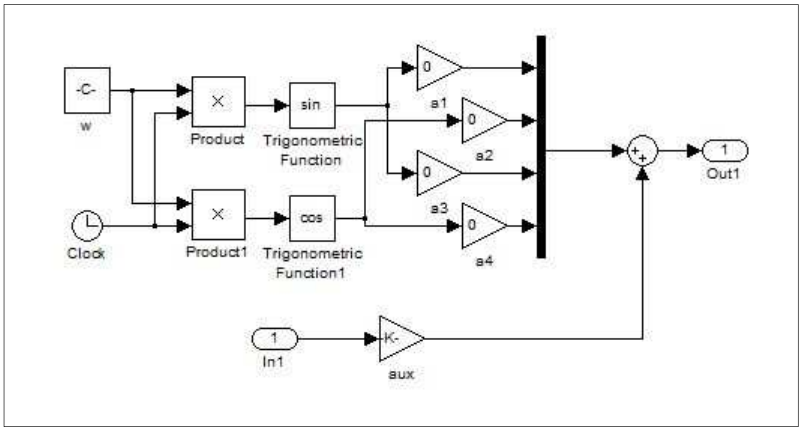


Figure 4-7 Original *Excitation* Subsystem

The current *Excitation* subsystem is pictured in Figure 4-8, and the colored blocks identify those that were changed from or added to the original subsystem.

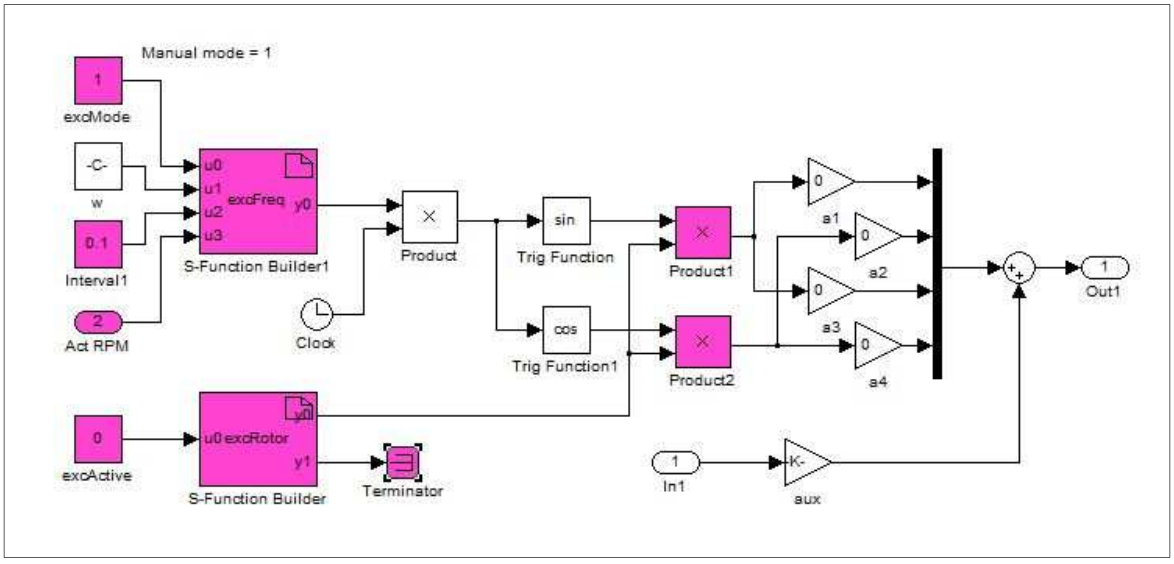


Figure 4-8 Current *Excitation* Subsystem

The original subsystem applied sinusoidal excitations to one or more axes at a single frequency. The excitation consisted of both sine and cosine terms, thus simulating a rotating imbalance. The frequency was chosen by the user and remained constant unless explicitly changed. The amplitudes of the sinusoid signals were also chosen by the user and could be the same or different for each axis.

The capability of *Excitation* was extended so that the frequency of the applied signals could be exactly the same as the speed or frequency of the rotor. Ensuring that the two frequencies are the same or nearly so is critical to determining the effectiveness of the adaptive controller. [28]. The addition of speed control to the FACETS system helped immeasurably in this regard since maintaining a constant rotor speed lessens or eliminates the need to adjust the excitation frequency dynamically. However, small variations in rotor speed still occurred regardless of how well the speed control functioned, so the ability to match the frequencies of the excitation and the rotor speed was still needed.

Dynamic frequency adjustment was implemented by S-function *excFreq* shown in Figure 4-4 (A listing of this function appears in Appendix B.). S-function *excFreq* can also update the frequency of the excitation at a user-specified interval in addition to adjusting it at the speed of the simulation. Intervals of any length can be chosen, limited only by the configuration of ControlDesk, and the length can be changed at any time. This "adjust and hold" capability was added in case instability resulted from adjustments that were made too rapidly. In fact, "adjust and hold" was initially developed for another subsystem which did become unstable when driven at the simulation frequency.

Another capability developed for subsystem *Excitation* permitted the recording of the exact time when excitation was both applied to and removed from an axis. Again, [27] had noted the importance of this to the determination of the responsiveness of the adaptive gains to changes in the imbalance state of the flywheel. S-function *excRotor* implements this capability by monitoring the current and previous states of the subsystem, enabling or disabling the excitation accordingly and updating the current status of the excitation through an output that is available for data capture. This output can then be plotted as a function of

time so that the exact moments of the application and removal can be found. See appendix B for a complete listing of this S-function.

Phi

Modifications were made to the *Phi* subsystem so that the disturbance vector could also be composed on the basis of:

- the actual speed of the rotor or
- the dominant frequencies calculated by a Fourier analysis.

The original *Phi* subsystem is shown if Figure 4-9.

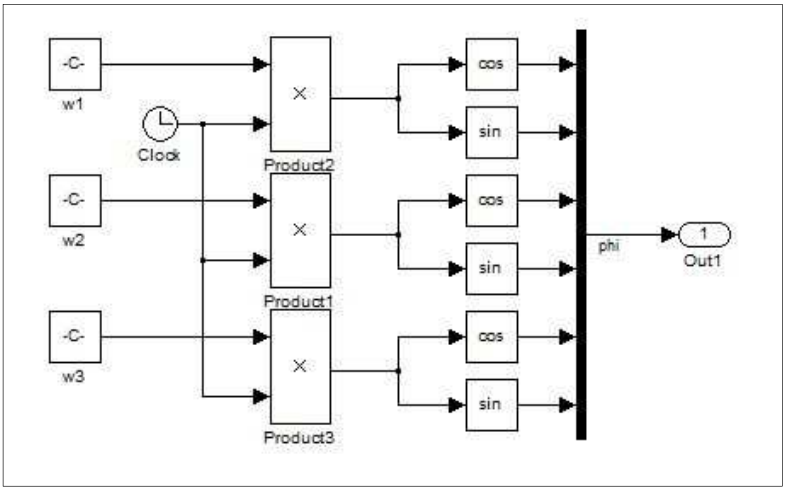


Figure 4-9 Original *Phi* Subsystem

As can be seen from the Figure, three frequencies rather than one were used to construct the disturbance vector to improve the effectiveness of adaptive control when the primary frequency to reject was not precisely known or when frequencies that were multiples of the rotor speed were to be rejected too [28]. The three frequencies were entered as constant values through ControlDesk.

The Phi subsystem was revised twice from the original, and the first version is pictured in Figure 4-10. The colored blocks identify those that were added to the original subsystem.

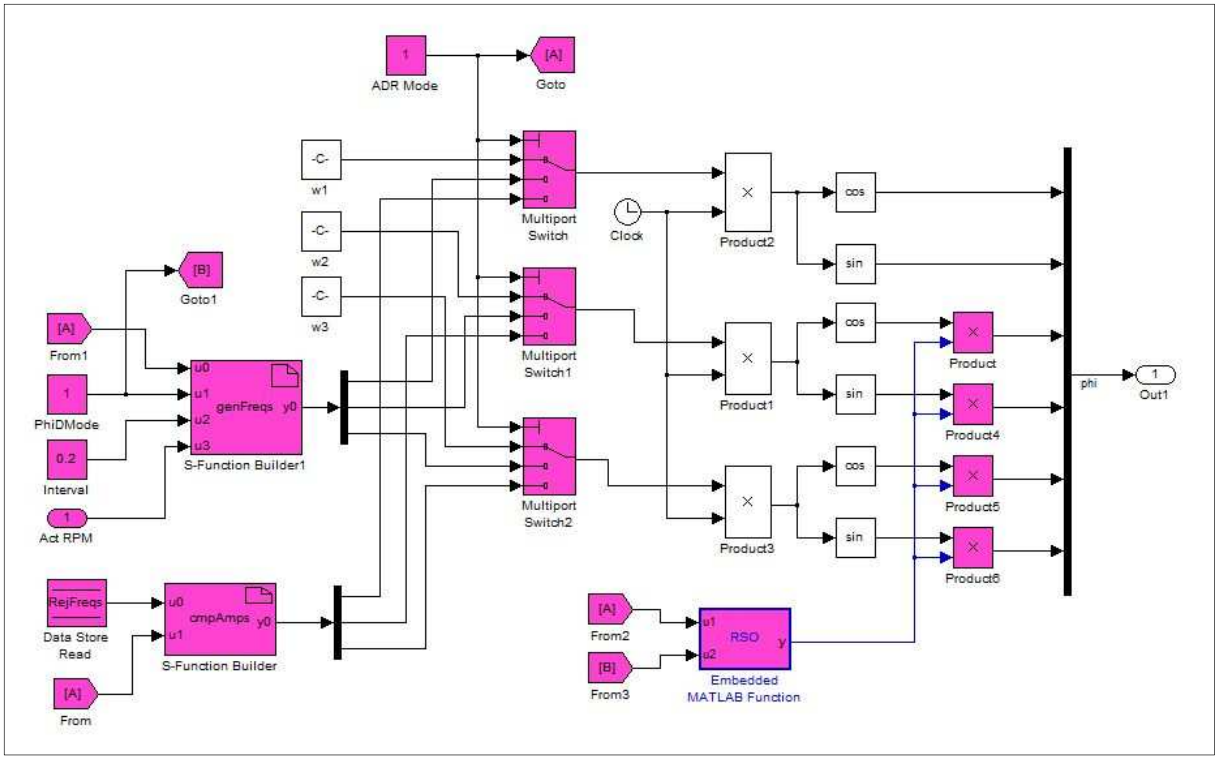


Figure 4-10 *Phi* Subsystem - Version 1

The first changes to the *Phi* subsystem maintained the existing size of the disturbance vector (three) but also introduced dynamic composition whereby the frequencies to reject could be based on the actual speed of the rotor, either measured directly by the tachometer or calculated indirectly via a Fourier transform. With dynamic composition, the actual frequencies that appear in the disturbance vector and the desired frequencies to reject are the same or nearly so improving the effectiveness and the responsiveness of the adaptive controller. As the actual and desired values diverge, the disturbance rejection becomes less complete, and the adaptive gains become less predictive (responsive) [28].

S-function *genFreqs* determines the frequencies to reject based on the measured speed of the rotor. This speed and others chosen by the user are placed in the disturbance vector. The user-chosen values can be multiples or fractions of the rotor speed or just about anything related to it depending only on the configuration of ControlDesk and the complexity of the S-function. In addition, *genFreqs* can update the disturbance vector at the simulation speed or at longer,

user-specified intervals, the "adjust and hold" capability discussed earlier, if instability results as a consequence of updates that occur too often. See Appendix B for a listing of function *genFreqs*.

S-function *cmpAmps* determines the frequencies to reject based on a Fourier analysis provided by S-function *DFT* that is included in subsystem *State Estimator*. The frequencies used for the disturbance vector are the dominant ones, those with the greatest amplitudes. The frequencies are made available to this function through a Data Store, a Simulink block that provides a vehicle for exchanging data between subsystems. This S-function updates the frequencies at whatever interval is required to calculate the next Fourier transform. This interval depends on the sampling frequency and the frequency resolution of the transform. Both parameters can be chosen and changed at will by the user through ControlDesk.

In addition, *cmpAmps* can be configured to only update the frequencies if ones calculated subsequently correspond to amplitudes that exceed a user-defined threshold. This feature was added to prevent the immediate reappearance of persistent frequencies following their rejection. When the composition of the disturbance vector is dynamic and driven by the Fourier analysis, once a persistent disturbance is rejected, its frequency is removed from the vector, and the disturbance then reappears only to be rejected again where it once more reappears *ad infinitum*. Establishing an amplitude threshold ensures that a disturbance once detected will be rejected until something more dominant unbalances the system. See Appendix B for a listing of S-function *cmpAmps*.

Following the decision to reduce the disturbance vector to a single frequency (See Section 3.3), the *Phi* subsystem had to be changed accordingly. These changes produced the second or current version shown in Figure 4-11. The colored blocks are the ones that were added to the original subsystem.

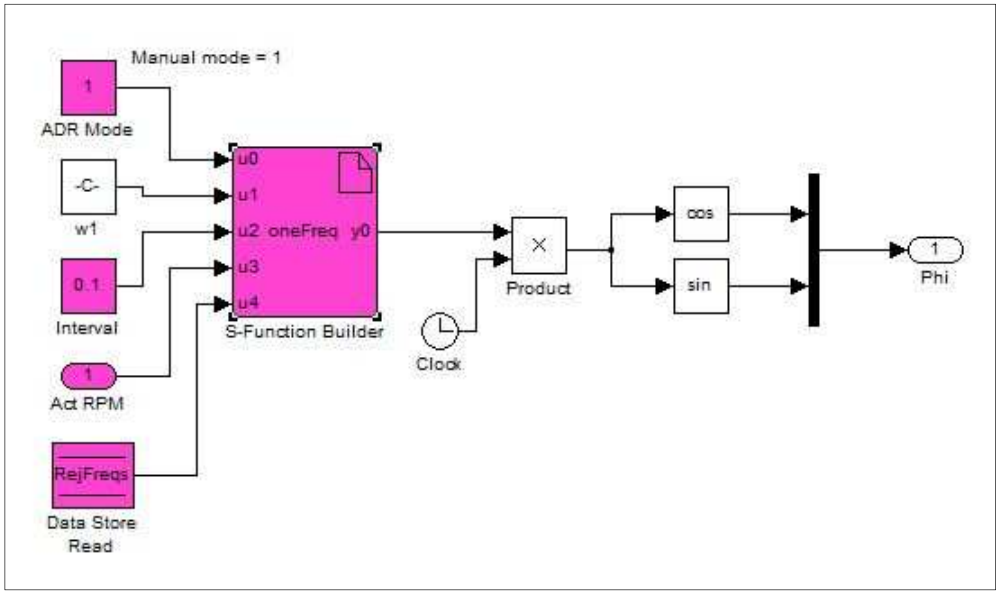


Figure 4-11 Current *Phi* Subsystem - Version 2

Although Version 2 is greatly simplified, it still provides the same functionality as the earlier Version. The disturbance vector can still be composed in the same three ways, by providing a single frequency manually or by determining it dynamically based on either the speed of the rotor or the dominant frequency returned by the Fourier transform. The S-function *oneFreq* chooses the method for composing the vector based on a switch set by the user. This function is basically a condensed version of a combination of the S-functions *genFreqs* and *cmpAmps* described earlier, and it also incorporates all of the switching that was done previously through several Simulink blocks. A listing for S-function *oneFreq* is shown in Appendix B.

S-function *oneFreq* provides the "adjust and hold" capability for the rotor-speed driven case or mode just as *genFreqs* did. In addition, amplitude and frequency thresholds can be established in the case of the Fourier-driven mode. The purpose of the amplitude threshold is the same in both Versions of the subsystem. The ability to establish a frequency threshold was a feature added to Version 2. The purpose of the frequency threshold is to prevent a very low or zero frequency from being used to compose the disturbance vector. A dominant frequency can occur at or near zero Hz if the rotor's orbits are large or off-center.

State Estimator

The *State Estimator* subsystem was enhanced to include a real-time Discrete Fourier Transform (DFT) that extends the capability of the adaptive controller. The DFT provides a spectral or frequency analysis of the rotor displacements, allowing the controller to identify and reject frequencies that are not known *a priori* and also provides a useful tool for the analysis of experimental data.

The original *State Estimator* subsystem is shown in Figure 4-12.

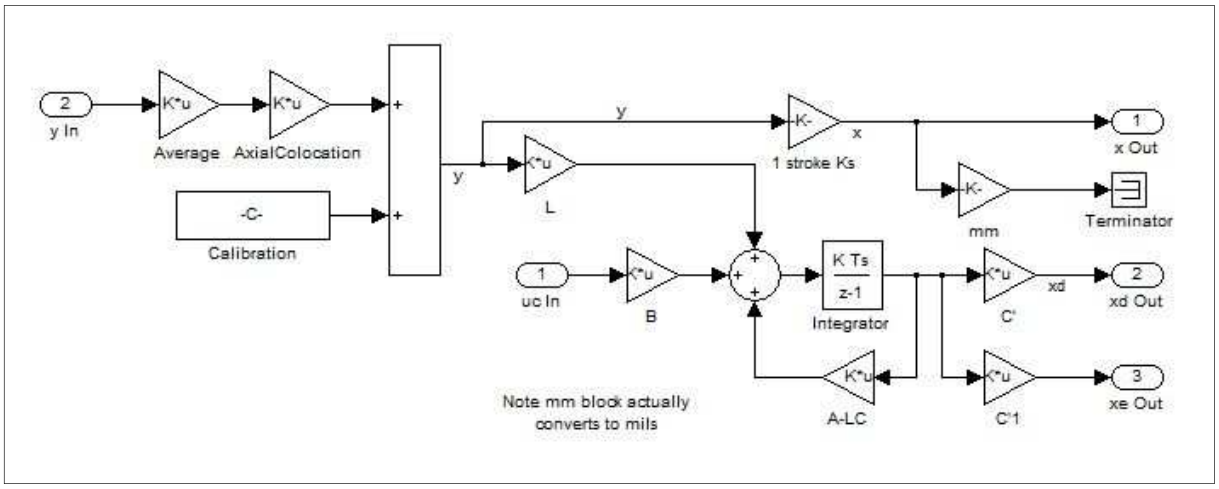


Figure 4-12 Original *State Estimator* Subsystem

The current *State Estimator* subsystem appears in Figure 4-13, and the colored blocks identify those that were added to the original subsystem.

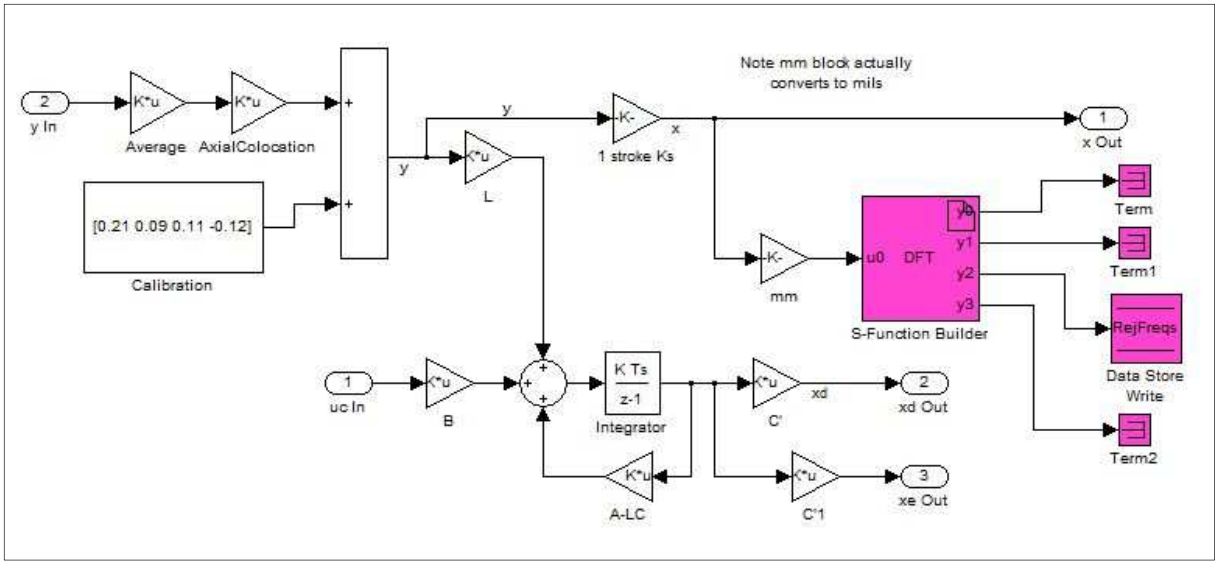


Figure 4-13 Current *State Estimator* Subsystem

The S-function *DFT* computes the discrete Fourier transform of rotor displacements along one axis according to the following equations given in Beckwith et.al.[32]:

$$A_n = \frac{2}{N}\sum_{r=1}^{N} y(r\Delta t)\cos\left(\frac{2\pi rn}{N}\right) \quad n = 0, 1, \cdots, \frac{N}{2} \tag{4.1}$$

$$B_n = \frac{2}{N}\sum_{r=1}^{N} y(r\Delta t)\sin\left(\frac{2\pi rn}{N}\right) \quad n = 1, 2, \cdots, \frac{N}{2} - 1 \tag{4.2}$$

$$C_n = \sqrt{{A_n}^2 + {B_n}^2} \quad n = 0, 1, \cdots, \frac{N}{2} \tag{4.3}$$

where A_n and B_n represent the Fourier coefficients, C_n represents the Fourier amplitudes, N equals the total number of data points and $y(r\Delta t)$ equals the magnitude of the r^{th} data point (displacement). The Fourier frequencies f_n are calculated using the equation:

$$f_n = n\Delta f \quad n = 0, 1, \cdots, \frac{N}{2} \tag{4.4}$$

where Δf is the fundamental cyclic frequency or frequency resolution of the transform.

Calculating Fourier transforms in a real-time environment presents serious challenges. The FACETS system is a genuine real-time computing or reactive computing system where the controller outputs must be available to the bearings within strict time constraints. If they are not, bearing instability can and does result. Functions added to the Simulink bearing/speed controller must not destabilize the bearings by either executing too slowly or imposing "shock" loads either of which could result in the failure to meet real-time constraints. Therefore, S-function *DFT* was written to run efficiently and consistently, requiring a nearly constant amount of time to execute during each invocation.

An examination of the equations used to determine the Fourier coefficients shows that each data point is used in the calculation of each coefficient. Since the number of data points to collect is known prior to the start of the transform, there is no need to wait until all data are collected before beginning the computations of the Fourier coefficients. S-function *DFT* performs all calculations that depend on a data point as that point becomes available. Calculating "as you go" ensures that the number of computations made each time the function is called is the same, avoiding shocks to the system and making the transform available to other subsystems as quickly as possible.

S-function *DFT* also uses a simple algorithm to produce an ordered sort of the Fourier frequencies, based on the corresponding Fourier amplitudes, that minimizes execution time and reduces risks to the real-time requirements of the system. Sorting is not done completely during a single execution of the function. Instead, just one data point is sorted and located properly during each invocation, but the sort occurs at simulation speed. The simple algorithm used and its unique implementation ensure that more sophisticated algorithms are not needed to increase the speed of the sort [33].

The discrete Fourier transform is configurable through a ControlDesk panel discussed in the following Section. The sampling frequency and the frequency resolution are chosen by the user and can be changed at any time during a simulation. The S-function recognizes if either or both parameters have changed during the computation of a transform, and if they have, the function will terminate the current transform and begin the calculation of a new one. In

addition, any one of the bearing's four axes can be selected as the basis of the Fourier analysis. Appendix B provides a complete listing of S-function *DFT*.

4.3 ControlDesk Instrument Panels

The ControlDesk instrument panels for this project were developed using the conventions for appearance and function established by [30]. These conventions ensure that instruments appear the same when placed within a frame or on a panel and that instruments are used consistently with regard to purpose. The “look and feel” of the panels created for this project is the same as it is for those created previously. Users are presented with a graphical interface for operating and controlling the rotor that gives no indication that it was developed by more than one researcher.

Each of the new panels, *More*, *DFT/ADR*, *ADR/Exc* and *Diag*, will be discussed in detail in this Section, and each with the exception of *Diag* will be shown.

More

The *More* panel, pictured in Figure 4-14, was created to ensure that new panels could be easily added to the existing system without changing the top-level *layout_start* panel.

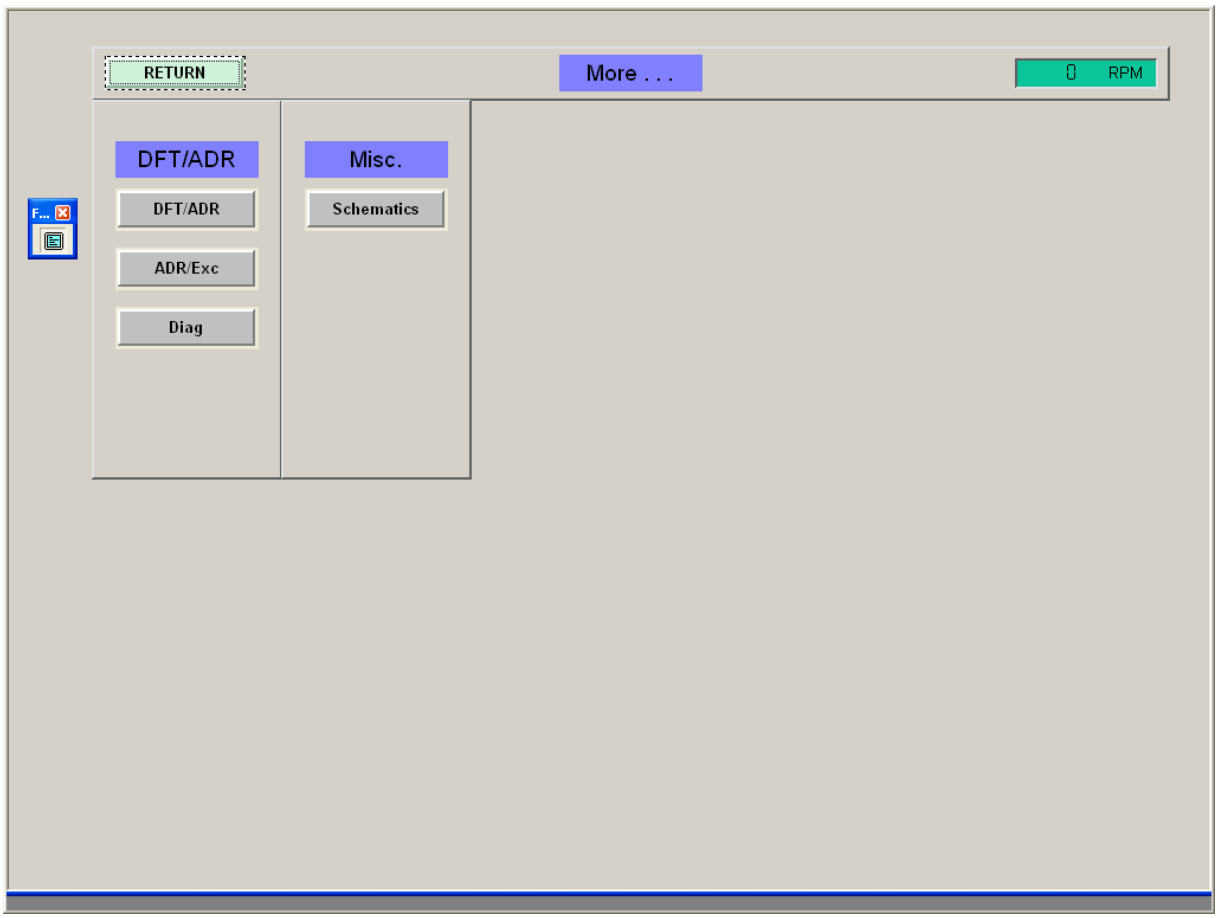


Figure 4-14 *More* Instrument Panel

This panel is actually a portal that provides access to other panels. Access to a new panel can be constructed by simply adding a PushButton instrument to the *More* panel and dropping the new panel's object onto that button. The return path from the new panel is just as easily created using complementary steps.

Reaching other panels through the portal adds to the time required to navigate the system, but the extra time is insignificant as long as the conventions for building panels are followed. If they are, the PushButtons required for navigation are placed consistently, ensuring that any panel can be accessed from any other using no more than a few mouse clicks. Other methods could have been used to move between panels, but they would have quickly cluttered the existing panels without providing a consistent, systematic way to add new ones.

DFT/ADR

The *DFT/ADR* instrument panel was developed to allow the user to configure and operate both the discrete Fourier transform and the adaptive controller as well as to observe the outputs from the transform and the frequencies of the disturbances to reject. The *DFT/ADR* panel is shown in Figure 4-15.

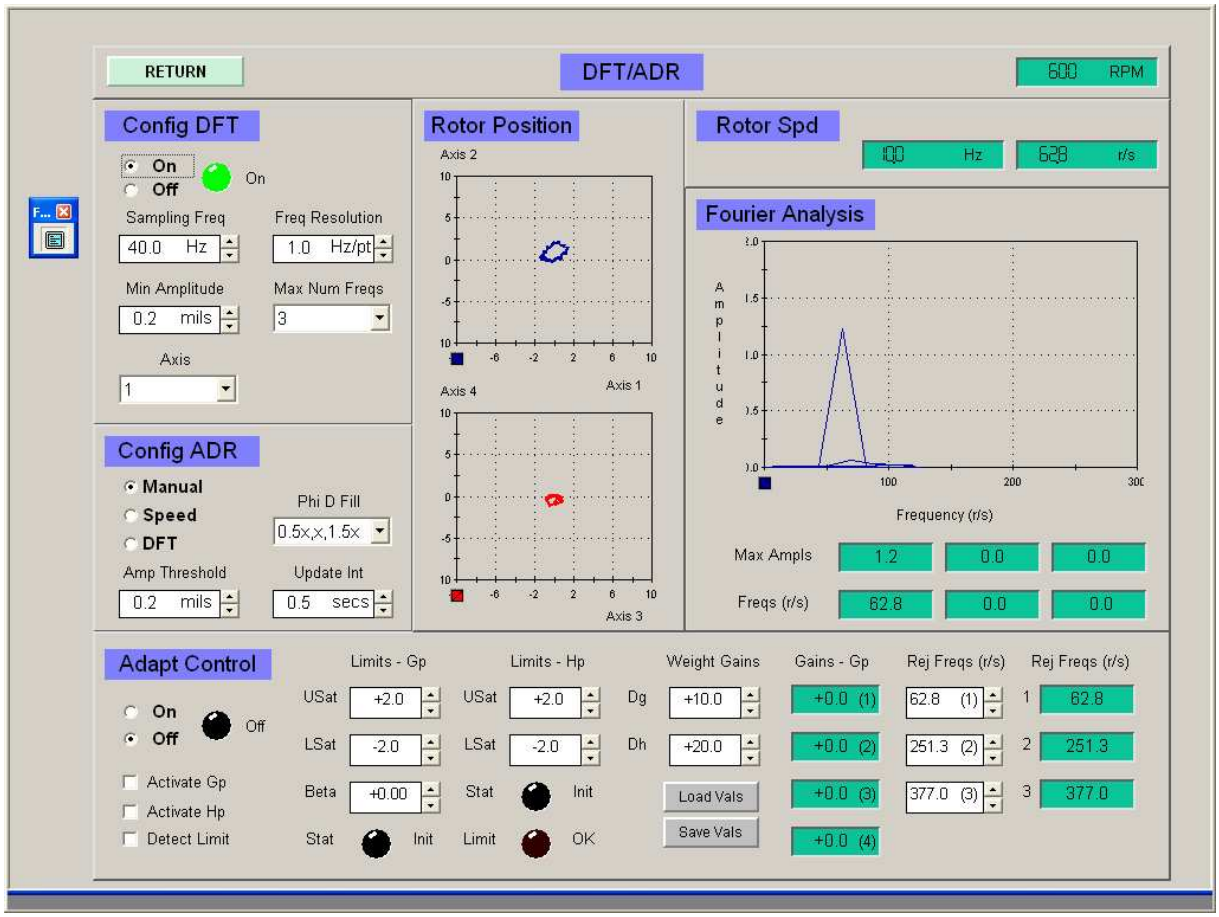


Figure 4-15 *DFT/ADR* Instrument Panel

Values for several parameters must be chosen to properly configure the system to perform a Fourier analysis. Although the parameters *Sampling Freq*, *Freq Resolution* and *Axis* are self-explanatory, further notes concerning the first two will be beneficial. The size of the structures used by S-function *DFT* to calculate a Fourier transform is based on the extreme values allowed for these parameters through ControlDesk. If these values are changed, the S-

function may not allocate adequate space to accurately compute transforms. A value defined in header file *mbcntrl_a.h* limits the number of data points that can be used to calculate a Fourier transform, and this value must be carefully considered before changes to expand the range of either parameter are made. See Appendix B for a listing of the header file.

The *Min Amplitude* parameter can be used to limit the frequencies output by the *DFT* function to only those that correspond to amplitudes equal to or greater than the parameter's value. This value or threshold prevents frequencies other than the most dominant from being used in the composition of the disturbance vector. Similarly, the *Max Num Freqs* parameter can also be used to limit the number of frequencies appearing in this vector. Note that with Version 2 of the *Phi* subsystem, neither parameter has any effect on the elements of this vector.

Notice that the Fourier analysis pictured identifies a single frequency that matches the rotor speed exactly. Even in the absence of any other excitation, this frequency will always appear in the spectral analysis of any rotating machinery that isn’t perfectly balanced. Although this frequency is present, it will not be located exactly at the rotor speed unless the actual speed of the rotor is an integer multiple of the frequency resolution [33]. For the case shown in Figure 4-15, the rotor speed was exactly ten times the resolution.

Configuration of the adaptive controller begins with the selection of the driving mode or method used to determine the frequency or frequencies, depending on the Version of the adaptive controller, to reject. The mode, chosen with a radio button, can be *Manual*, *Speed* or *DFT*. In *Manual* mode, the frequencies are simply entered by the user at the lower right of the panel. In *Speed* and *DFT* modes, the frequencies are determined dynamically, based on either the speed of the rotor or the output of the real-time DFT.

Values for parameters *Amp Threshold*, *Update Int* and *Phi D Fill* must also be chosen. Parameter *Amp Threshold* applies to *DFT* mode only, while parameters *Update Int* and *Phi D Fill* apply to *Speed* mode only. The purposes of the first two have already been discussed in Section 4.2. The remaining one, *Phi D Fill*, was used to add frequencies to the disturbance vector when the adaptive controller was driven by the speed of the rotor.

Recall that with the original system, the disturbance vector required three frequencies. When the controller is placed in *Speed* mode, only a single one, the rotor speed, is available to compose the vector. Therefore, the other two frequencies had to be calculated or chosen in some fashion to complete the vector. The parameter *Phi D Fill* provided several different methods for determining the remaining frequencies. The methods were straight-forward and computed the values of the additional frequencies as multiples of the rotor speed, fractions of the rotor speed or zero. The former method had been used in prior research of disturbance rejection since dominant frequencies occurring at two times and four times the rotor speed exist in the frequency spectrum of the FACETS system [28]. The latter method was included to support research that focused solely on disturbances that occur at rotor speed. Note again that with Version 2 of the *Phi* subsystem, the *Phi D Fill* parameter has no effect on the composition of the disturbance vector.

The bottom part of the DFT/ADR panel provides controls for operating and tuning the adaptive controller. Many of the instruments that appear here were already in place on existing panel *Control Parms* [30]. They were also placed on panel *DFT/ADR* so that both configuration and operation of the adaptive controller could be done from a single panel. Values of several parameters can be chosen to tune the controller and affect its responsiveness. Upper and lower saturation limits can be placed on the integrators that determine the adaptive gains G_p and H_p to prevent the gains from becoming too great and potentially causing instability in the bearings. In addition, values for the weighting matrices ΔG and ΔH can be chosen to affect how quickly the controller responds to disturbances.

Note that the calculation of the adaptive gains and the adaptive controller itself are disabled by default. The gains will only be calculated if the checkboxes are selected, and the output from the controller will only be applied to the bearings if the controller is enabled (turned on). In summary, nothing is output from the controller unless it is turned on, and the gains are activated.

ADR/Exc

The *ADR/Exc* panel provides the capability to configure and operate the adaptive controller just as the *DFT/ADR* panel did and also allows the user to enable the internal excitation and to choose either a static or dynamic method for selecting the excitation frequency. In addition, the values of the adaptive gains applied to each axis, the outputs from the adaptive controller and the orbits of the rotor are all displayed. The *ADR/Exc* panel is shown in Figure 4-16.

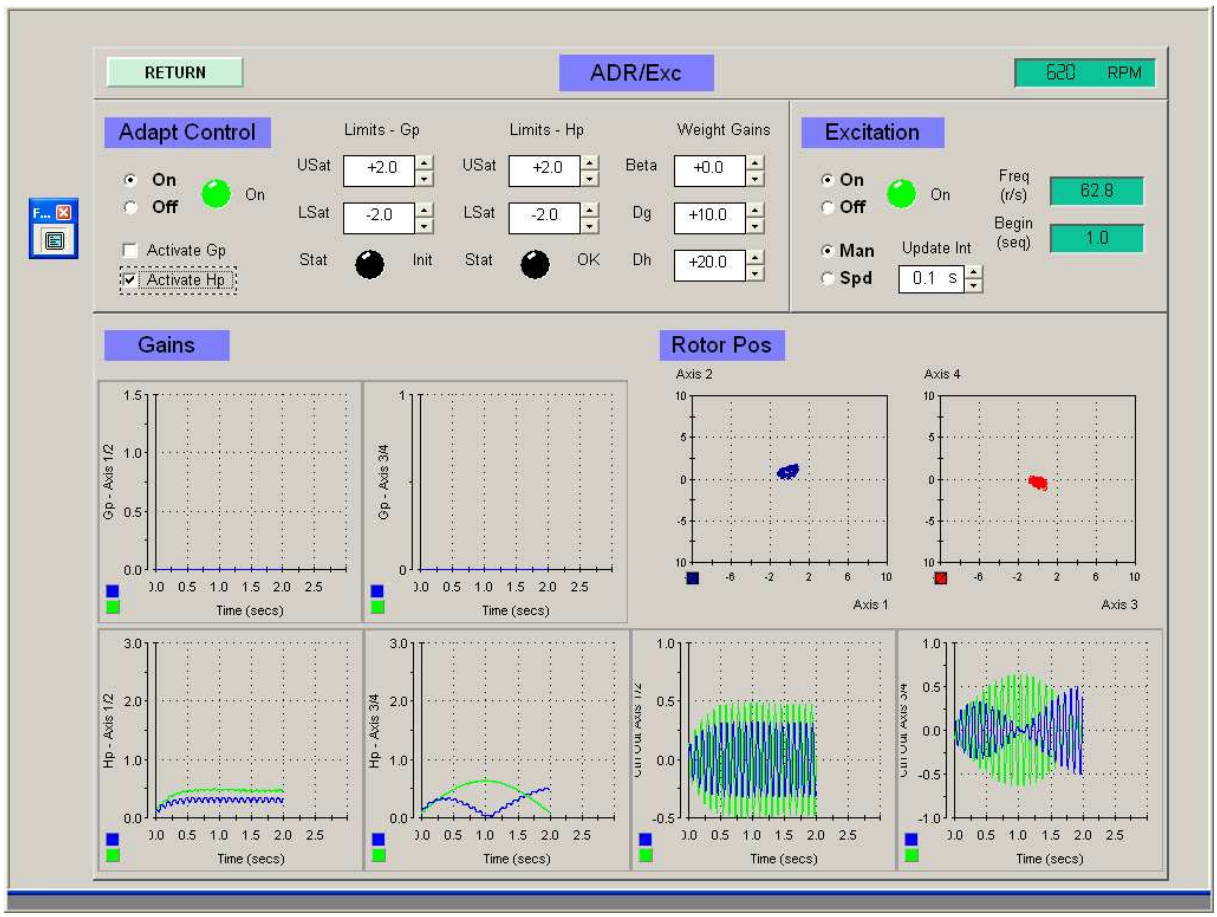


Figure 4-16 *ADR/Exc* Instrument Panel

This panel locates the controls essential to the safe investigation of adaptive-disturbance-rejection techniques onto a single screen. The adaptive gains and the controller outputs can be closely monitored and the controller quickly disabled if potentially unstable behavior is observed. Similarly, the effects of the excitation can also be closely watched and the

excitation instantly disabled if it's observed to be driving the bearing system towards instability.

Note from Figure 4-16 that the excitation must be enabled here and on existing panel *Input/Output* to be effective. The need to control the excitation from the *ADR/Exc* panel combined with the desire to not change any of the functions provided by existing panels required that the excitation be applied this way. Also, note that the frequency of the excitation can be static, chosen by the user and remaining constant until explicitly changed, or dynamic, following the speed of the rotor exactly or at "adjust and hold" intervals as described in Section 4.2.

Diag

The *Diag* panel was constructed so that diagnostic information could be easily output using an existing framework. This panel provides a means to display information that is necessary for a temporary purpose (e.g. troubleshooting problems or verifying outputs from new blocks) but that is not needed on a permanent basis. The *Diag* panel is not shown since it basically consists of the outer frame common to all panels and plotter array instruments.

Chapter 5 System Tuning

The Simulink bearing/speed controller has grown in size and complexity since it was originally conceived and built by [27]. Additions and enhancements have been made to the model to support the needs of subsequent researchers. During the current work with the FACETS system, the cumulative effects of the changes became significant enough to prevent the real-time requirements of the system from being met. Although this was not immediately apparent, it became so after attempts to resolve bearing stability issues actually led to increased instability. Once it was realized that real-time constraints were not being satisfied, several steps were taken to reduce the execution time of the controller. Specifically, subsystems were streamlined and profiled, fundamental sample sizes were varied and tested and tools provided by Simulink were used to produce a useful model that executed rapidly enough to ensure stability of the magnetic bearings.

5.1 Task Overruns

Section 3.3 briefly introduced the problem with task overrun errors that occurred with the bearing/speed controller built with Version 1 of the *Phi* subsystem. Initial testing with this controller began at low speeds, 600 RPM or less, and occasional bearing instability was observed with the adaptive controller active. The instability was present regardless of the mode used to drive the controller. At first, the adaptive control laws themselves were thought to be one possible cause of the instability. Perhaps, they did not apply well at very low speeds despite their solid theoretical development [25]. The laws were tested thoroughly using the simulated system and MATLAB program discussed in Chapter 2. The tests showed that the control laws worked perfectly at all speeds. The controller reacted to and rejected frequencies as low as 1 Hz.

Investigation then focused on the actual calculation of the adaptive gains by the Simulink bearing/speed controller. The Simulink model was modified to provide diagnostic information at several points along the data paths in the *Adaptive Control* and *Phi* subsystems. With these modifications in place, the instability became worse, occurring more frequently than before.

Collecting information that would hopefully provide insight into the problem was actually destabilizing the system further. At this point, the adaptive controller was still suspect. It was thought that possibly the calculation of the adaptive gains or the rates at which the gains adapted were causing instability. The Simulink model was modified to closely monitor the gains and the controller's output and to disable the controller if the potential for instability was detected. With these changes, the bearing/speed controller immediately became unstable when the adaptive controller was activated with the rotor suspended but not turning.

Finally, it became obvious that the stability problems steadily worsened as the Simulink model grew in size and complexity. It was ultimately concluded and correctly so that the dSPACE processor was overloaded. The Simulink bearing/speed controller could not be executed in the time allotted for it resulting in task overrun errors and attendant bearing instability during operation of the FACETS system.

When a Simulink model is compiled and loaded by Real-Time Workshop, the resulting executable program will be scheduled and run as one or more tasks on the dSPACE processor. In the case of the bearing/speed controller, the entire model is run as a single task. This task must execute completely from start to finish within the time allocated for it, and this time is defined by the fundamental sample size or fixed step size (FSS) of the simulation. If the task cannot be completed in a time less than or equal to the FSS, task overrun errors occur, and when they do, the results are unpredictable [34]. In the case of the FACETS system, the result was bearing instability.

Overrun errors can be eliminated several different ways including:

- decreasing the complexity of the model,
- increasing the efficiency of the model,
- increasing the fundamental sample size of the simulation and
- using a faster processor.

The first three of the four methods were used to overcome the overrun errors and restore the stability of the system without sacrificing functionality. Economic realities prevented the use of the last method listed.

The changes made to the existing subsystems during this research have already been discussed in Chapter 4. Note that the subsystems were changed to both enhance the capabilities of the FACETS system and to reduce the potential for overrun errors. In summary, the changes made to help eliminate errors by decreasing complexity included restructuring subsystems to maintain functionality with fewer blocks and reducing data widths where possible. Changes made to increase efficiency included reducing the number of trigonometric functions, replacing embedded MATLAB functions with S-functions when possible and exclusively using S-functions when user-defined functions were needed.

5.2 Subsystem Profiles

To demonstrate how significantly the execution time of a subsystem can be affected by the steps taken to satisfy both the functional and real-time requirements of the magnetic bearing system, stand-alone models of the *Adaptive Control* subsystem were built and profiled. Profiling is a method used to determine the time spent executing each block in a Simulink model during a simulation. The Profiler is an excellent tool for evaluating the relative effects on the overall execution time of a model caused by adding, changing, deleting and reconfiguring blocks.

The base model used for profiling the *Adaptive Control* subsystem is shown in Figure 5.1.

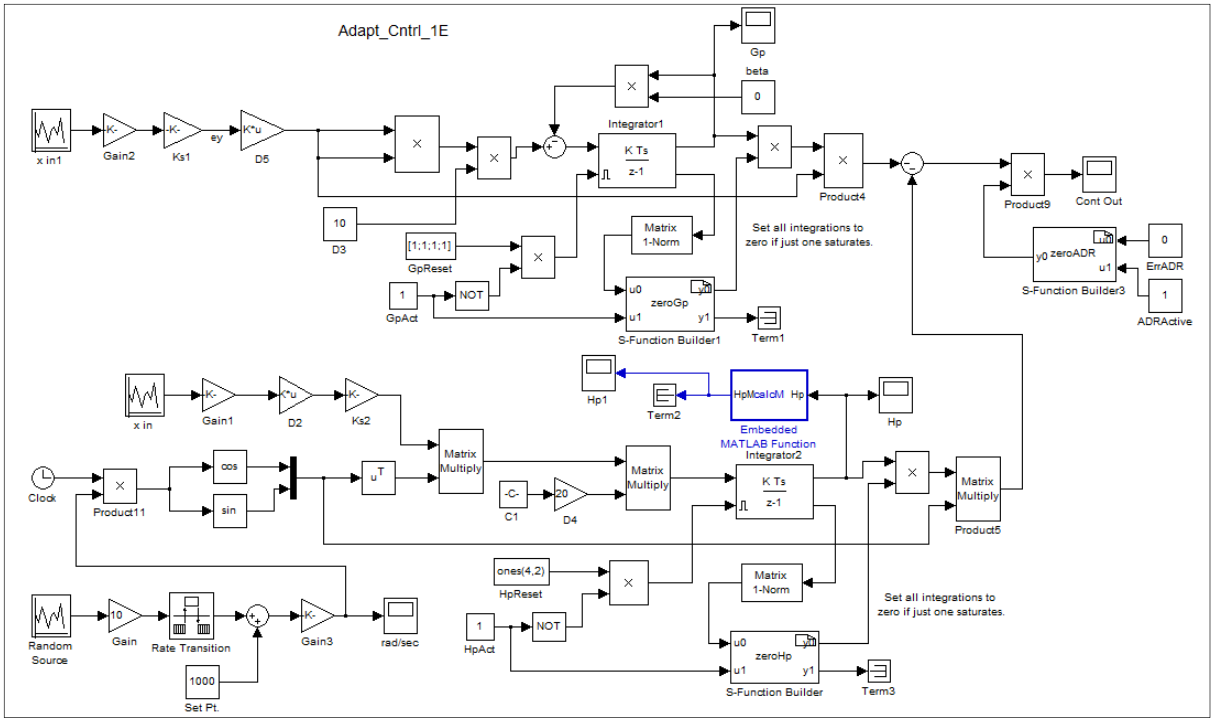


Figure 5-1 Base Profile Model – Adaptive Control Subsystem

This model is nearly identical to the actual *Adaptive Control* subsystem with two exceptions: the inputs from the other subsystems necessary for calculating the adaptive gains and composing the disturbance vector were simulated rather than input from hardware devices or state estimators and the blocks required to compose the disturbance vector were incorporated directly into the adaptive controller rather than assembled into a separate subsystem. The inputs for the adaptive gains were generated using Random Source blocks executing at simulation frequency. The input for the disturbance vector was also created using a random source block, but this block ran at 10% of the simulation frequency to closely resemble the small, slow fluctuations about a set point that are observed in the speed of the rotor during operation. Note that for profiling purposes, the adaptive controller was placed in speed mode.

The profile results for the base model, Adapt_Cntrl_1E, and for all variations of it are shown in Table 5-1. (See Appendix C for samples of profile reports for each model.)

Adapt_Cntrl_1E		Adapt_Cntrl_1S		Adapt_Cntrl_3E		Adapt_Cntrl_3S	
Tot Time (secs)	Blk Time (%)	Tot Time (secs)	Blk Time (%)	Tot Time (secs)	Blk Time (%)	Tot Time (secs)	Blk Time (%)
1.33	4.7	1.17	1.3	6.91	56.2	1.33	4.7
1.33	4.7	1.17	1.3	6.48	56.1	1.27	4.9
1.33	5.9	1.19	1.3	6.72	55.6	1.30	4.8

Table 5-1 Profile Results - Adaptive Controller

The base model as its name implies used a single frequency to construct the disturbance vector and an embedded MATLAB function to calculate the magnitude of the adaptive gain H_p. Note that this model also has just two trigonometric functions and a relatively narrow data width. The maximum width is a 4x2 matrix, and it occurs in the H_p calculation path. For this model as well as for all others listed in Table 5-1, the duration of the simulation was 60 seconds, and the simulation period or fundamental sample size was 0.01 second.

As Table 5-1 shows, the base model required 1.33 seconds to execute (total time or Tot Time), and the time spent executing the embedded MATLAB function (block time or Blk Time) varied from 4.7 to 5.9% of the total. The total time is not equal to the duration of the simulation in any of the results because the models were run entirely within Simulink, and when they are, there is no synchronization between the simulation and a real-time or wall clock. Simulink simply runs the model as fast as it can, and the number of times the model is run is equal to the product of the duration and frequency (1/FSS) of the simulation. Therefore, all models were executed exactly the same number of times (6000) during profiling, and all times appearing in the Table provide excellent comparative data.

The second model of the *Adaptive Control* subsystem profiled, Adapt_Cntrl_1S, was the same as the first or base model except that the embedded MATLAB function was replaced with a

compiled S-function. The total times for the simulations and the block times for the S-function are shown in Table 5-1. Note that the substitution of an S-function reduced the total times 10 to 12% and the block times from 75 to 80%. The reductions in block times were calculated from the actual time spent executing the block rather than from the percentages appearing in the Table.

The third model profiled, Adapt_Cntrl_3E, was similar to the first, but three frequencies were used to construct the disturbance vector. This entailed the addition of four trigonometric blocks and expanded the maximum data width to 4x6, the size of the matrix required to calculate the H_p gains. Again, the profile results for this model are shown in Table 5-1, and as can be seen, the total times increased substantially over those recorded for the other models. In addition, the time spent executing the embedded MATLAB function represented over 55% of the total simulation time.

The last model profiled, Adapt_Cntrl_3E, differed from the previous in that an S-function was substituted for the embedded MATLAB function. The reductions in total simulation times and in the time spent calculating the magnitude of the adaptive gains were substantial when compared to the results of the previous model and as illustrated by the data in Table 5-1. Interestingly, times recorded for the last model and for the first were nearly identical.

In summary, the steps taken to reduce the execution time of the *Adaptive Control* subsystem were shown by the profile results to be those that do indeed reduce the complexity and improve the efficiency of the subsystem. As a result, the execution time of the entire Simulink bearing/speed controller model is reduced, and the risks of task overrun errors are lessened.

A stand-alone model of the *State Estimator* subsystem was also built and profiled to determine how significantly the real-time calculation of Fourier transforms affected the execution time of the subsystem. The stand-alone model is shown in Figure 5-2.

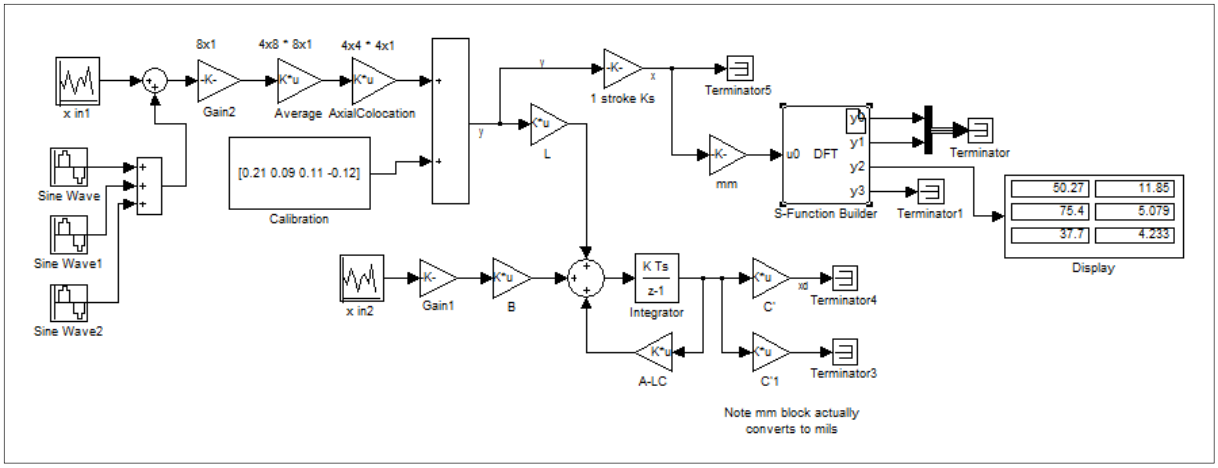


Figure 5-2 Profile Model – State Estimator Subsystem

This model is identical to the actual subsystem with the exception of the inputs. The *State Estimator* subsystem receives inputs on rotor position and control currents from other subsystems while inputs for the profiled model were simulated, just as they were with the profiled model of the *Adaptive Control* subsystem, with Random Source blocks. In addition, to properly test the calculation of the discrete Fourier transforms, three sinusoidal excitations of varying frequencies and amplitudes were added to the simulated rotor position inputs.

The profile results for the stand-alone model are shown in Table 5-2. (See Appendix C for a sample profile report.)

Tot Time (secs)	Blk Time (%)	Smp Freq (Hz)	Freq Res (Hz/pt)	Tot Time (secs)	Blk Time (%)	Smp Freq (Hz)	Freq Res (Hz/pt)
5.27	1.8	0	0	5.38	2.0	100	1
5.28	2.1	0	0	5.56	6.7	100	0.2
5.33	2.6	40	1	5.55	6.5	200	1
5.34	2.9	40	0.6	6.31	18.6	200	0.2

5.34	4.4	40	0.2	7.88	33.9	500	0.4

Table 5-2 Profile Results - State Estimator

All simulations profiled were executed at the same frequency and for the same duration. The first two results indicate the time necessary to execute the model without calculating a Fourier transform. For the rest of the results, Fourier transforms were computed with the combinations of sampling frequency and frequency resolution shown in the Table. Again, the times shown are those required to complete the simulation (total time) and those required to complete the execution of the DFT S-function block (block time). The latter are given as a percentage of the total.

There is a small amount of overhead required by the *DFT* S-function when it's disabled to ensure that the function is properly initialized once it is enabled. The time necessary to perform this overhead explains why the block times for the first two results shown in Table 5-2 are nonzero. The results also indicate that transforms requiring 100 or fewer data points have little effect on the execution time (total time) of the subsystem. Transforms that require 200 to 500 points do cause a small but noticeable increase in the total time by about 4% over that observed when the DFT is inactive. However, the data also show that when the DFT is enabled, the execution time of the subsystem can increase by 30% or more given combinations of high sampling frequency and fine frequency resolution.

In summary, the real-time calculation of a discrete Fourier transform has a negligible effect on the performance of the subsystem as long as the sampling frequencies are no greater than 200 Hz and the frequency resolutions are no less than 1 Hz/pt.

5.3 Fundamental Sample Sizes

The fundamental sample size or fixed step size of a simulation has already been defined as the maximum amount of time allowed for a Simulink model to execute from start to finish on the dSPACE processor. This time was originally set to 0.00015 second by [27] and was not

changed by [30], so it was the step size in use at the outset of this project. As the bearing/speed controller model evolved, it eventually could not be executed completely in the time allocated for it without task overrun errors and the resulting bearing instability. Experiments were conducted in parallel with other efforts to eliminate the errors to determine if a larger FSS could be used that would still satisfy the bearing's real-time constraints. Finding an optimal sample size can be difficult since it must be large (long) enough to accommodate the model but small (short) enough to ensure stability.

The existing sample size was very small and the resulting simulation frequency was very fast (6667 Hz). Given this speed, a good sample size for the first experiment was chosen as twice the existing one (0.0003 second). The FACETS system was tested with this over a range of speeds beginning with the rotor suspended but stationary and with nearly all of the changes to the model described earlier in place. The system performed perfectly with no stability problems.

Next, the FSS was increased to 0.0004 second, and the same tests were conducted as before. The system was operated over the range of speeds commonly used, and all facilities of the model were exercised. The system was stable over the course of all tests. For the next series of tests, the FSS was again increased by 0.0001 second, and here, the upper bound for the fixed step size was reached. With the FSS set to 0.0005 second, the FACETS system was just barely stable when suspended but stationary. The bearings protested audibly, though not loudly, emitting ominous sounds indicating that a step size any larger would lead to immediate instability.

A summary of the tests shows that the system was stable with sample sizes of 0.0003 and 0.0004 second. The upper limit on the sample size was clearly determined to be 0.0005 second. A lower limit was not precisely identified given the stressful nature of the testing, but an extrapolation of the results suggests that step sizes much less than 0.0003 second are too small. Accordingly, the FSS of the simulation was set to 0.0003 second for the duration of the project. In addition, several Simulink blocks used in the bearing/speed controller have their

sample times set explicitly. For all of these blocks, the sample times were also set to 0.0003 second.

By choosing an FSS at the lower end of the stable range, room remains for the FSS to be increased if the model grows in scope and/or complexity. However, the FACETS system is only stable over a very narrow range of fundamental sample sizes. If changes are made to the model, they should only be made after their effect on the overall simulation time of the model is known.

5.4 Model Advisor

Simulink provides a tool, the Model Advisor, which can be used to check models and subsystems for conditions and configuration settings that can result in inefficient simulations and poor code generation. The Advisor produces a report that details suboptimal conditions and settings and suggests changes to improve the model or subsystem. The individual checks performed by the Advisor will not be discussed here. They are simply too numerous and many were not relevant to the performance of the bearing/speed controller. For a detailed discussion of the checks, consult the online Simulink documentation.

All of the subsystems in the Simulink model were checked with the Model Advisor, and the results showed that all were well constructed and configured. The Advisor found very little that could be changed to improve the performance of the controller. However, the tool did suggest enabling compiler optimizations that would improve the efficiency of the code generated by Real-Time Workshop. These changes were made to the Simulink configuration parameters for the bearing/speed controller.

Chapter 6 Experimental Results

The primary objective of the experiments was to determine if the adaptive gain H_p.could be used to identify a change in the balance state of a rotating system. The magnetic bearing system was subjected to both simulated disturbances and actual changes in balance to investigate the behavior of the gain. Variations in the speed of the rotor affected the gain significantly, and a relationship between the gain and the balance change could not be established. Several methods were considered and tested to determine if synchronizing the disturbance frequency (rotor speed) with the reject frequency could eliminate the influence of the speed variations on the gain. Computer simulations demonstrated that the methods were promising and worth implementing on the actual magnetic bearing system.

6.1 Simulated Imbalances

The effectiveness of adaptive control can be demonstrated by applying a sinusoidal disturbance to the rotor and observing the change in the rotor displacements along the bearing axes when the adaptive controller is activated. Figure 6-1 shows the displacement of the rotor along a single axis both before and after adaptive control is applied.

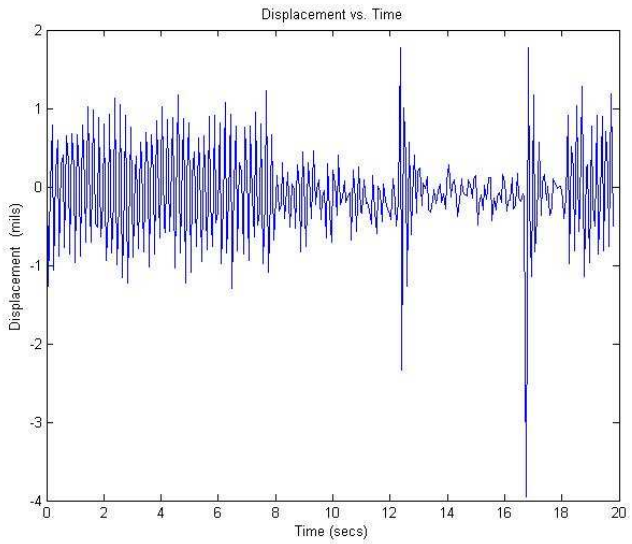


Figure 6-1 Rotor Displacement with Adaptive Control

The rotor was spinning at 600 RPM (10 Hz), and it was excited by a sinusoid, generated internally by the Simulink subsystem *Excitation*, with a magnitude of 0.3V and a frequency of 10 Hz. The excitation simulated a rotating imbalance since the disturbance frequency was the same as the rotor speed. The adaptive controller was activated at 7.78 seconds, and the excitation was applied at 12.22 seconds and removed at 16.76 seconds. Finally, the adaptive controller was deactivated at 18.18 seconds.

Figure 6-2 illustrates the variation of the adaptive gain H_p with time.

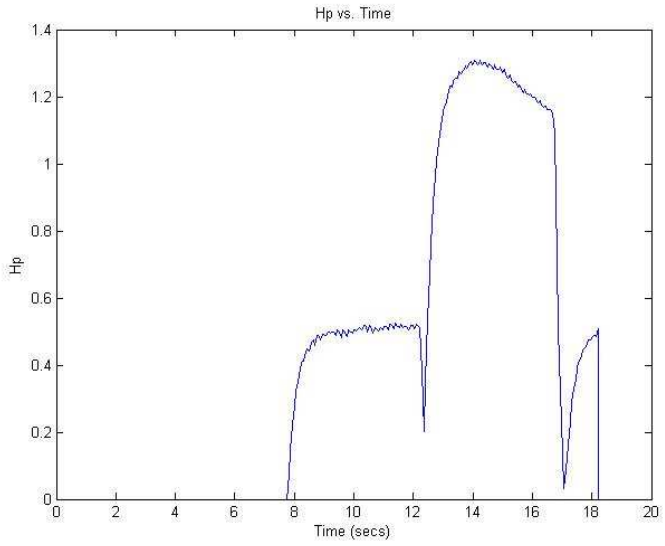


Figure 6-2 Gain Response to Synchronous Excitation (10 Hz)

The initial response of the gain is attributable to the inherent imbalance in the system. Once the gain has adapted to this and the controller has largely rejected the synchronous disturbance, the excitation was applied. The gain again responds almost immediately and adapts to the application of the sinusoid reducing the rotor displacements significantly.

The responsiveness of the gain and the effectiveness of the disturbance rejection can be seen more clearly when the trace of the rotor displacement, the trace of the gain and the duty cycle of the excitation are plotted together as a function of time. These plots are shown in Figure 6-3.

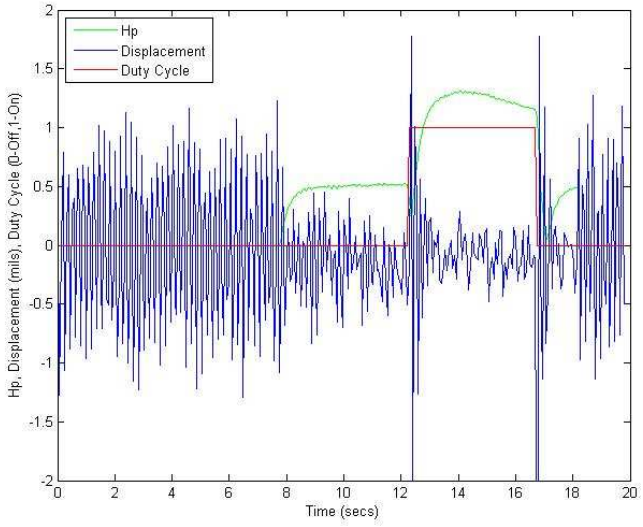


Figure 6-3 System Response to Synchronous Excitation (10 Hz)

Notice the spikes in rotor displacement that occur immediately after the sinusoid is applied and removed, the nearly simultaneous response of the gain H_p to the rapid change in displacements and the return of the displacements to their inherent imbalance levels when the adaptive controller is turned off.

The use of internal excitation clearly shows how well the adaptive controller can suppress synchronous disturbances. The successful rejection of asynchronous disturbances, such as those introduced via base motion, can also be shown using this method. In addition, the results presented in this section were obtained with the adaptive controller operating in manual mode whereby the frequency of the disturbance to reject was known and entered manually through ControlDesk. Similar, nearly identical, results can also be obtained with the controller operating in speed or DFT mode. In the former, the disturbance frequency would be derived from the actual speed of the rotor, and in the latter, from the dominant frequency calculated by the discrete Fourier transform. DFT mode is the only mode that can be used to construct the disturbance vector if the disturbance occurs at an unknown, asynchronous frequency.

6.2 Physical Imbalances

Adaptive control is only genuinely useful if it can detect and suppress disturbances that occur as a result of an actual change in the balance condition of a rotor or flywheel. Researchers have tried many different approaches to experimentally changing the balance of a rotating system. These range from the simple to the complex. Simpler methods typically involve affixing a small weight to a rotating device and then causing the weight to separate from the device through some means as described by Shiue et.al. [35]. The more complex methods attempt to move a small weight from one location on a rotating system to another often through mechanical and/or magnetic means as discussed in [28].

A simple method was chosen to change the balance of the system so that disturbance rejection could be adequately tested. Conceptually, the method selected was straightforward. A small weight (3 grams or less) would be attached to the circumference of the flywheel and then dislodged once the rotor was spinning at the desired speed. However, the physical realization of the method proved to be a tedious and time consuming task that tested our collective imaginations.

Several different versions of the method were tried. Initially, a small weight was secured to the outer surface of the flywheel with a thin strip of paper that was attached to the flywheel with an adhesive. Once the machinery was in motion, the paper would be either cut with a razor or burned off with a micro torch, releasing the weight. Both the razor and the torch worked well at very low speeds, but at speeds above those that could be achieved by turning the rotor by hand, neither worked acceptably. The razor proved dangerous, and the micro torch was unable to heat the paper sufficiently.

Next, a weight was attached to the inner circumference of the flywheel with a weak adhesive, and the weight was dislodged with compressed air. Again, this version proved successful while testing at low speeds, but it was unreliable at speeds above 300 RPM where the compressed air could not be uninterruptedly directed at the weight for a long enough time to separate it from the flywheel.

For the third variation of the method, a weight was affixed to the outer surface of the flywheel with an adhesive. Once the rotor was spinning, the weight was struck with a small, rigid bar to break the adhesive bond and detach the weight. This version worked perfectly once a suitable adhesive was found. Several were tried, and all were too elastic except one: hot melt glue. This adhesive, when applied, cures rapidly and becomes brittle. It's sufficiently strong to keep the weight firmly attached to the flywheel at all speeds used during testing, and it breaks quickly and cleanly when the weight is struck. This version of the basic method worked very well and was used to economically generate all of the experimental results. Figure 6-4 shows the flywheel with the imbalance weight attached and the detachment tool.

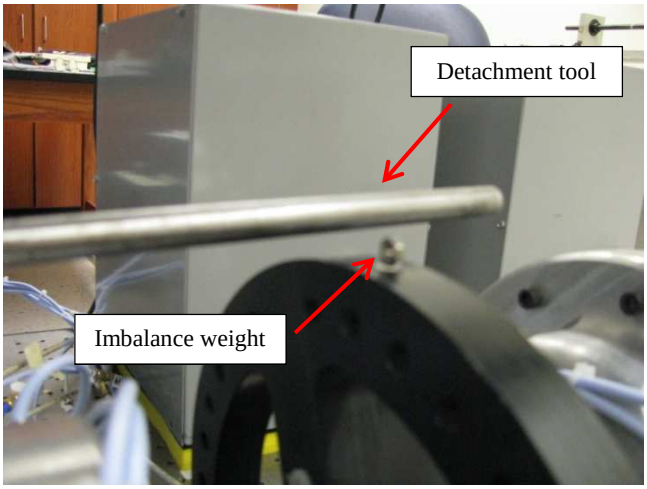


Figure 6-4 Flywheel with Imbalance Weight

6.3 Constant Frequency Disturbance

Several tests were conducted using the method just described to emphatically show the responsiveness or predictive capability of adaptive gain H_p when the balance state of the flywheel/rotor changed. The results from one test are shown graphically in Figures 6-5, 6-6 and 6-7. For this test, a 2.9 gram imbalance weight was used; the rotor was turning at 1200 RPM, and the adaptive controller was operating in manual mode.

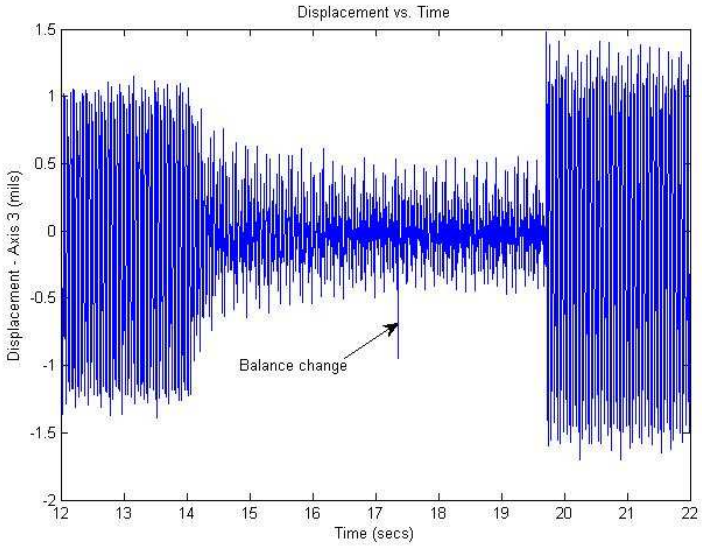


Figure 6-5 Rotor Displacement with Balance Change (2.9 g)

Figure 6-5 clearly shows the change in displacement along one bearing axis that occurs when the balance state changes (i.e. when the weight is separated from the flywheel).

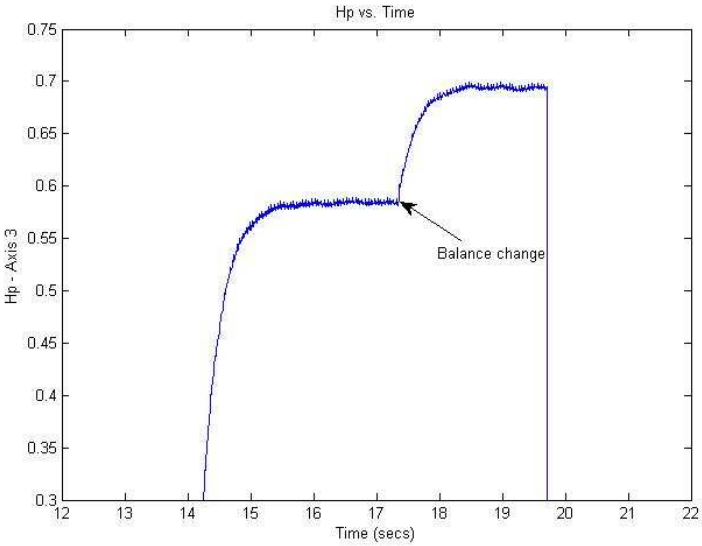


Figure 6-6 Gain Response to Balance Change (2.9 g)

Figure 6-6 illustrates the response of the adaptive gain to the change in imbalance. Overlaying the Figures as shown in Figure 6-7 indicates how quickly the adaptive controller reacts. The change in the balance state occurs at 17.35 seconds, and the controller responds at 17.36

seconds. Note that the plot for the adaptive gain has been inverted so that the immediate response of the controller to the change in displacement is obvious from the intersection of the two graphs.

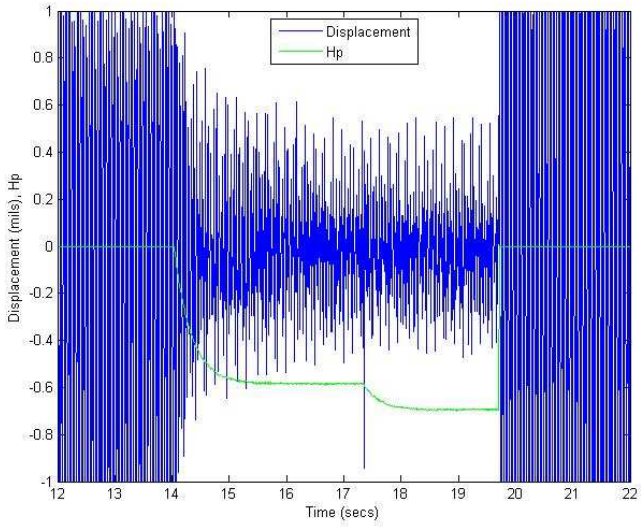


Figure 6-7 System Response to Balance Change (2.9 g)

For adaptive gain H_p to be truly useful as a predictor of a change in balance that can occur for example when a crack or other defect develops in a rotating device, it must react when the balance change does not cause a discernible change in the displacement of the rotor. If H_p only responds to measurable changes in displacement, it does not provide any indication of a change in balance that isn't already provided by the displacement measurements. Therefore, experiments were conducted to establish whether the variation in the adaptive gain with time indicated a state change when the variation of displacement with time did not. The same testing procedure was followed as before. However, the magnitude of the force created by the rotating imbalance weight had to be chosen such that no observable change in rotor displacements occurred when the weight was knocked loose from the flywheel.

Experiments indicated that an imbalance force of less than 0.6 N would not affect the rotor displacements. The imbalance weights used in the experiments were regularly shaped metal objects that weighed from 0.2 to 1.0 gram. Although nearly anything could be used to unbalance the system, objects less massive than 0.2 gram were either too small or

insufficiently rigid to withstand being struck and dislodged from the flywheel. Since the imbalance weight was always located 100 mm from the center of the rotor, rotor speed was limited to less than 1700 RPM during testing, otherwise detectable changes in the rotor displacements occurred.

With the basic parameters of imbalance weight, imbalance location and rotor speed established, experiments were then conducted to verify the disturbance rejection and investigate the predictive capability of adaptive gain H_p. Again, the adaptive controller was operating in manual mode. Representative results from these experiments for displacement and gain are shown in Figures 6-8 and 6-9.

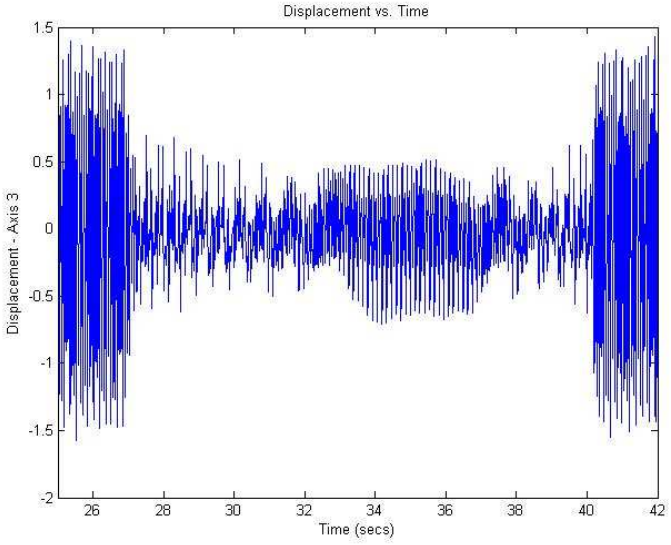


Figure 6-8 Rotor Displacement with Balance Change (0.5 g)

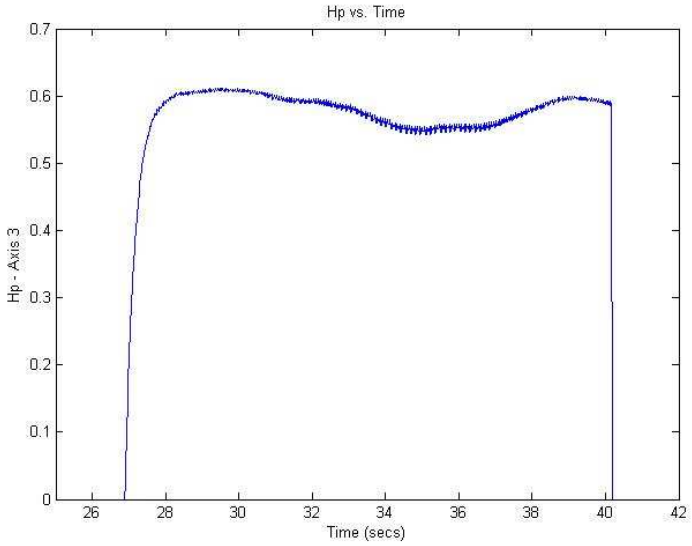


Figure 6-9 Gain Response to Balance Change (0.5 g)

The Figures show that the disturbance is adequately rejected, but they do not show whether H_p is predictive regarding the change in balance. Figure 6-9 is a coarse plot of the magnitude of the gain, but a close examination of a greatly magnified gain also gives little indication of a relationship between H_p and the balance state. A comparison of Figures 2-8 and 6-9 may illustrate why this is so. Figure 2-8 shows that H_p settles to an equilibrium value for the modeled system once the disturbance is rejected while Figure 6-9 shows that it does not settle during testing on the physical system. The variation in H_p which may either disguise or eliminate the predictive character of the gain has been discussed by previous researchers most notably [28] and is attributable to the small changes in rotor speed that occur on the test system. These changes cause a divergence between the frequency to reject, the frequency used to construct the disturbance functions, and the actual frequency of the disturbance, the speed of the rotor, resulting in an unsteady gain. Recall that for the modeled system, the two frequencies were exactly the same, and once the gain adapted, its value was nearly constant.

The speed of the rotor is well controlled, and the variations in speed about the set point are small, typically ±20 RPM [30]. However, the influence of the change in speed on the adaptive gain can be seen in Figure 6-10. In this Figure, rotor speed, normalized by the set point, and H_p are plotted. The set point was 900 RPM, and the range of speeds over the test interval was

875 to 916 RPM. In addition, the gain was offset by a constant value to help illustrate the correlation between the change in speed and the change in the gain.

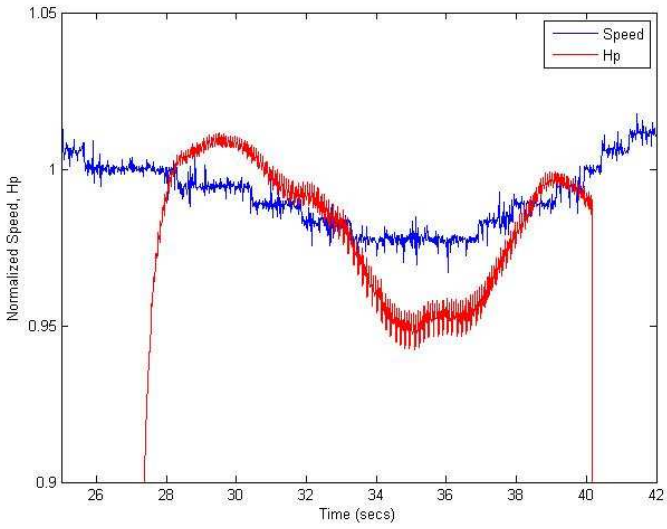


Figure 6-10 Gain Response to Balance Change (0.5 g) and Rotor Speed Variation

The Figure demonstrates that H_p tends to follow the speed, only assuming a fairly constant value when the speed of the rotor remains nearly constant.

Results identical to those just discussed were also obtained with the adaptive controller operating in DFT mode. Driving the controller with a discrete Fourier transform produced a reject frequency identical to one entered manually for the case where the rotor speed is nearly constant and the rotor is subjected to synchronous disturbances only. Theoretically, the Fourier transform is capable of calculating nearly the exact frequency (rotor speed) of the disturbance, so the identical behavior of the controller operating in either mode is not necessarily expected. However, the discussion of subsystem profiles in Chapter 5 shows that for the actual magnetic bearing system, an upper bound of 1 Hz/pt is placed on the frequency resolution of the transform by the simulation. Finer resolutions can greatly increase both the time required to complete the simulation and the risk of task overruns. Limiting the resolution to 1Hz/pt guarantees that the dominant frequency returned by the transform will always be the same for the rotor when its speed varies by no more than ±30 RPM about its set point.

Therefore, the controller behaves exactly the same, whether driven by a Fourier transform or by a frequency entered manually, when the rotor spins at a nearly constant speed.

6.4 Varying Frequency Disturbance

One possible method for stabilizing H_p was to synchronize the frequency used to construct the disturbance vector with the actual frequency of the disturbance. In fact, the need to synchronize the two frequencies was the reason for the development of the multi-mode adaptive controller discussed in Chapter 4. Therefore, several tests were performed to determine if good disturbance rejection characteristics could be maintained and if the predictive capability of H_p could be established with the adaptive controller operating in both speed and DFT modes.

In speed mode, the controller must be configured to update the frequency to reject at some interval as explained in Chapter 4. Testing indicated that the controller was insensitive to the interval size with the exception of intervals on the order of the Fundamental Sample Size (FSS) of the simulation. These would sometime result in negligible controller outputs and an ineffective adaptive controller but would more often result in large and growing outputs and an overly aggressive controller. Intervals three of more times greater than the FSS all produced similar results, though an interval size of two seconds or more would nullify the anticipated benefits of speed mode since larger intervals did not allow the rejection frequency to closely track the disturbance frequency. Regardless of the interval chosen, the results from the experiments were nearly identical and not as expected.

A sample of the results for the adaptive gain is shown in Figure 6-11.

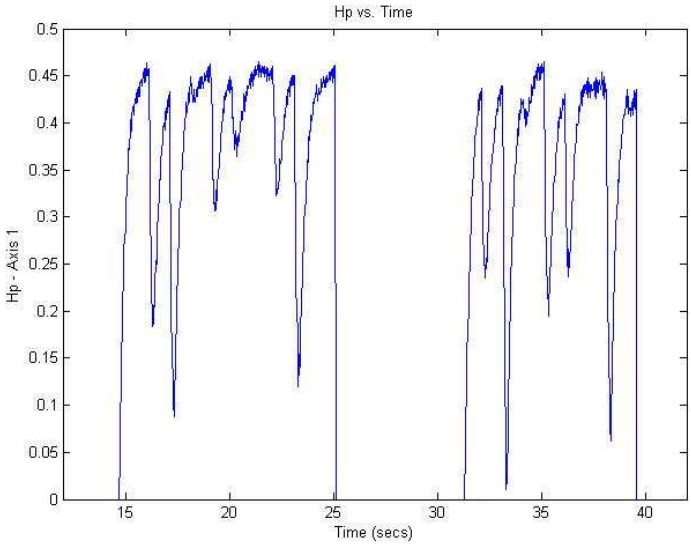


Figure 6-11 Gain Response to Reject Frequency Updates (1.0 sec)

Figure 6-11 demonstrates that each time the reject frequency is updated with the current speed of the rotor, the value of H_p changes abruptly. Note that the Figure shows two instances of the adaptive controller being activated and deactivated. The cause of the abrupt change can be inferred by overlaying the update profile on top of the gain curve as shown in Figure 6-12.

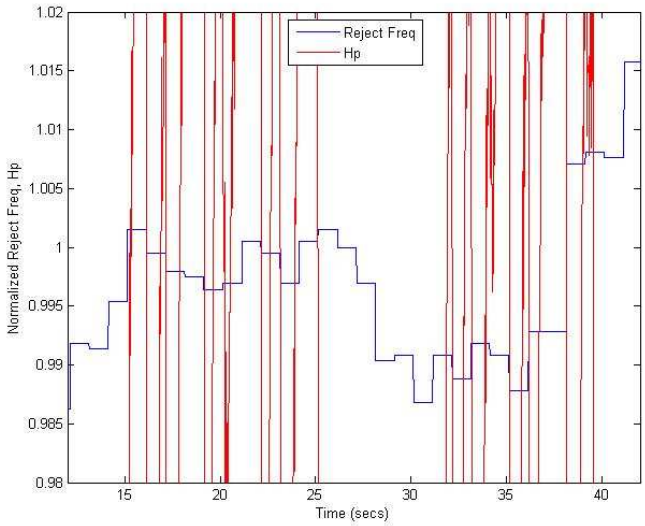


Figure 6-12 Gain Response to Reject Freq. Updates (1.0 sec) and Reject Freq. Variation

The update profile illustrates the frequency to reject as a function of time. As shown, this frequency is simply the rotor speed normalized by the set point, and it is updated at the beginning of each interval and maintained for the length of that interval. Note that each time the frequency changes, the adaptive gain changes immediately and significantly. Again, the gain plotted has been offset by a constant value so that the change in gain can be shown intersecting the change in frequency.

Two possible causes for the behavior of the adaptive gain were errors in the calculation of the gain and errors in the application of the control laws. To investigate the former, the output from each block of the *Phi* and *Adaptive Control* subsystems was closely examined to determine if all calculations were being performed correctly. To investigate the latter, the literature was again reviewed.

The gain H_p is found from the integration of Equation 2.5, so different methods of integrating this equation were tried, including Backward Euler and Trapezoidal, in place of Forward Euler which had always been used previously. In addition, the Discrete Integrator block in the Simulink model was replaced with an S-function that implemented a Forward Euler integration method using a C program. Regardless of the method used, the values calculated for the gain were identical. Also, different sample rates for the integration were tried, each one being an integer multiple of the Fundamental Sample Size, and each produced the same result.

Several other modifications were made to the subsystems, and none changed the results. A purely continuous system was constructed and added to *Adaptive Control* to calculate H_p in parallel with the discrete system. Each determined the gain identically. The *Phi* subsystem was eliminated and its function incorporated directly into *Adaptive Control*, and this did not change the behavior of the adaptive gain at all. Also, blocks were added to the adaptive controller that generated a known profile of the rotor speed to check the accuracy and reliability of the outputs from the tachometer. Actual and generated speeds produced the same results. Throughout the testing, Simulink was impressively consistent. The values calculated for the adaptive gain as well as the gain's behavior when the reject frequency was updated

were always the same regardless of the changes and additions made to the bearing/speed controller model.

Further review of the control laws and further examination of the outputs from the blocks in the Simulink model finally identified the cause of the problem (See Figure 6-13.).

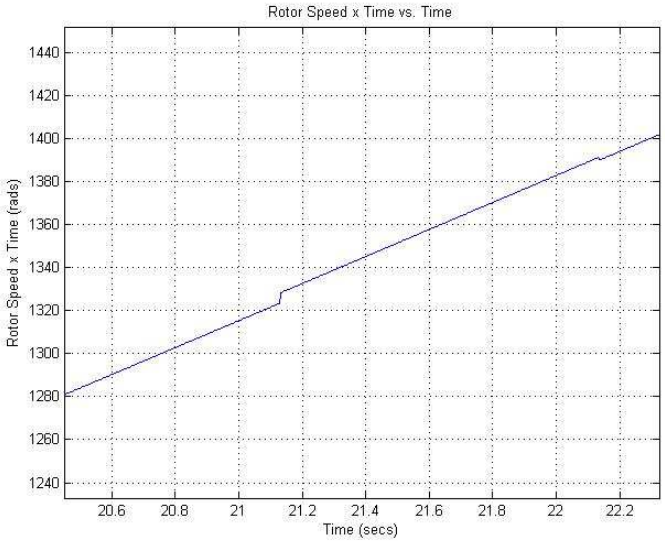


Figure 6-13 Angle Variation with Reject Frequency Updates (1.0 sec)

Figure 6-13 plots the angle used by the disturbance function (Equation 2.11) as a function of time for a small part of the results shown in Figures 6-11 and 6-12. The angle is discontinuous at the points where the reject frequency is updated. The updates occurred at 21.13 and 22.13 seconds. Components of the disturbance function based on this angle are also discontinuous as is the function itself. The sine component of the function appears in Figure 6-14.

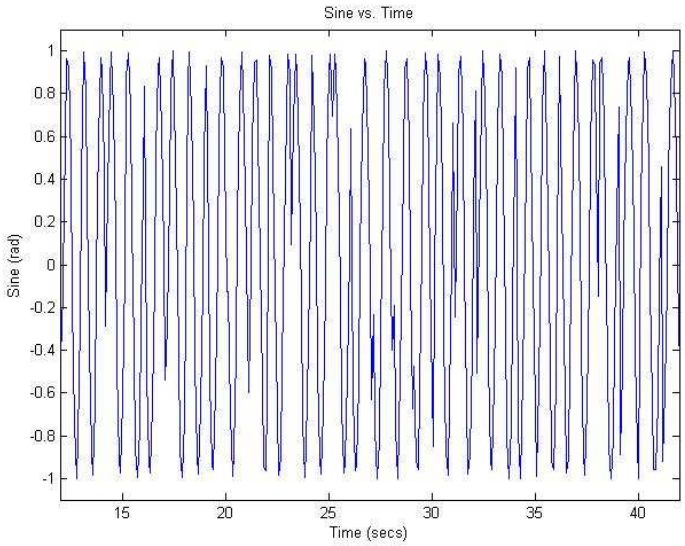


Figure 6-14 Sine Component of Disturbance Function with Reject Freq. Updates (1.0 sec)

As can be seen from this Figure, the discontinuities only occur when the frequency is updated, though they do not always occur when the frequency changes. Since the control laws implemented by the adaptive controller and summarized in [36] require that the disturbance function be continuous, the behavior of H_p when the frequency changes is not surprising.

6.5 Incrementally Varying Frequency Disturbance - Simulated

One possible way to satisfy the control laws and provide for dynamic adjustment of the frequency to reject would be to change the frequency over an interval of time using many small increments rather than changing the frequency all at once during a single sample step. In speed mode, the controller samples the speed of the rotor over a pre-selected interval, and if the speed changes between successive samples, the frequency to reject is updated by the magnitude of the change. If the magnitude could be made very small, almost infinitesimal, a continuous disturbance function could be approximated, and the effects of small variations in rotor speed on the adaptive gain H_p possibly eliminated.

To test the feasibility of approximating the disturbance function, a Simulink model was built that implemented a refined version of the rotor-speed driven adaptive controller. The model appears in Figure 6-15.

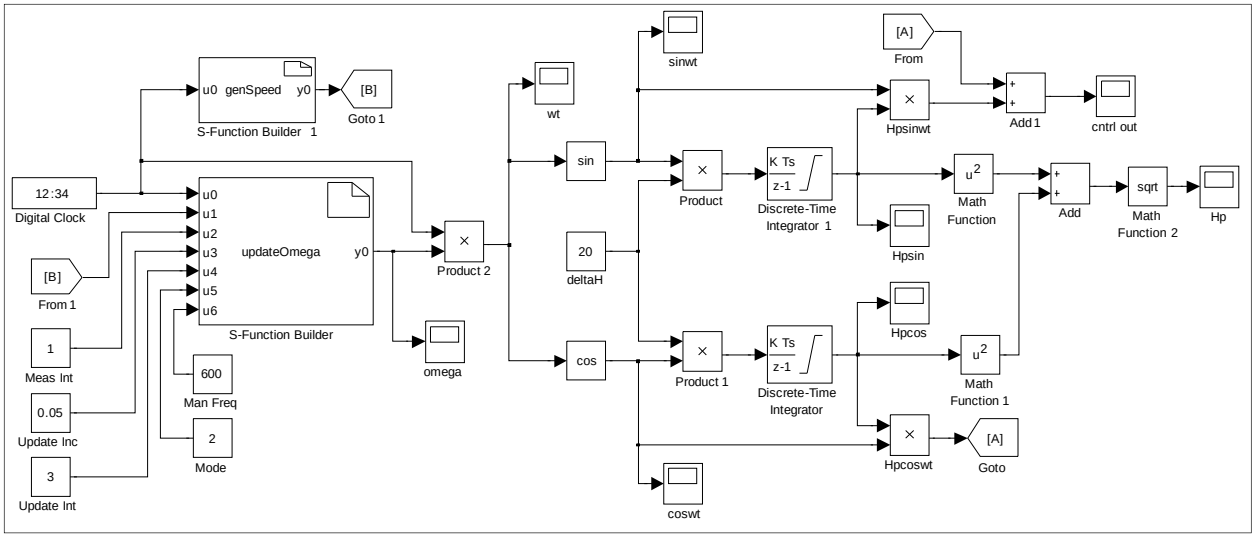


Figure 6-15 Simulink Model for Approximating a Continuous Function

This model determines the adaptive gain and the controller output identically to the bearing/speed controller operating in speed mode. The updates to the reject frequency are still computed by an S-function, in this case S-function *updateOmega*, but they are computed much differently than they are in the current controller. (See Appendix B for a listing of *updateOmega*.) User specified values for the speed measurement interval, speed update increment and speed update interval are required to properly configure the model. The speed measurement interval is simply the time between one speed measurement and the next just as it is in the current rotor-speed driven adaptive controller. The speed update increment is the magnitude of the change made to the reject frequency during a sample step and is expressed in terms of RPM. The speed update interval is the rate at which updates are applied to the reject frequency, and this interval is expressed as an integer multiple of the Fundamental Sample Size of the simulation rather than as a time per se. This model also implements a manual mode whereby the reject frequency is maintained at a constant value and generates a rotor-speed profile that simulates the small variations in speed of the actual rotor about its set point. The profile is created by S-function *genSpeed*, and a listing of the function is given in Appendix B.

Several simulations were performed to determine if values of the configuration parameters could be found that would result in an adaptive gain insensitive to small changes in the reject frequency. As long as the gain reacts to these changes, it cannot be determined whether the behavior of the gain is predictive of a change in the balance state of the system.

For all simulations, the Fundamental Sample Size (FSS), the speed measurement interval and the speed update interval were held constant, and the respective values for each were 0.001 second, 1.0 second and four times the FSS. Only the speed update increment was varied since it solely affects the continuity of the disturbance function. The other parameters can also affect the behavior of the controller but not directly. For example, if the speed measurement interval or speed update interval is too large, the controller will tract changes in rotor speed poorly whether the speed varies consistently within a range or whether the speed is increasing or decreasing steadily. In addition, the same profile for rotor speed was used in all simulations.

Prior to discussing the results of the simulations, Figure 6-16 is shown to illustrate how the frequency used to compose the disturbance function is updated incrementally.

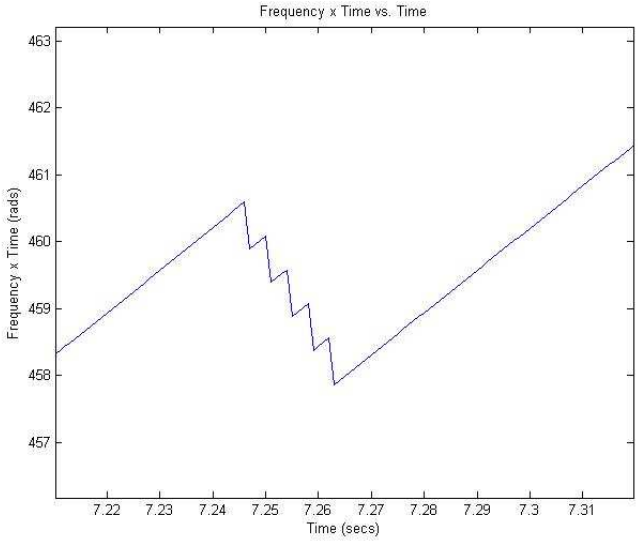


Figure 6-16 Angle Variation with Incremental Updates

For the results shown, the speed update increment was set to 1.0 RPM, and all other parameters were set to their aforementioned values. The frequency to reject was decreased from 607 RPM to 602 RPM in five equal increments over a period of 0.16 second. Note that the product of frequency and time is plotted on the vertical axis rather than frequency.

In summary, the simulations demonstrated that speed update increments greater than 0.05 RPM (0.0052 rad/sec) always created discontinuities in the disturbance function, adaptive gain and controller outputs. Figures 6-17 and 6-18 plot the latter two for a speed update increment of 0.1 RPM.

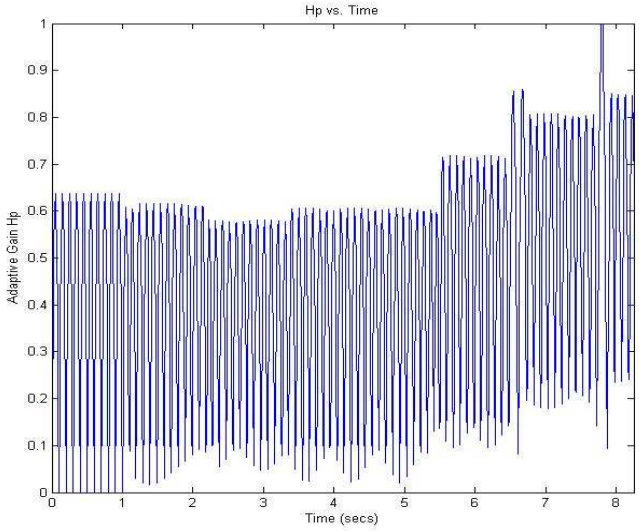


Figure 6-17 Discontinuous Adaptive Gain with Incremental Updates

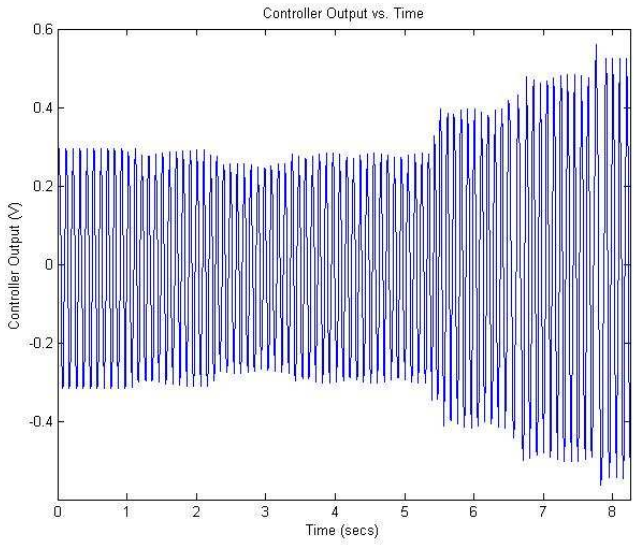


Figure 6-18 Discontinuous Controller Output with Incremental Updates

For each Figure, updates to the reject frequency began at seconds 1.0, 2.16, 3.33, 5.37, 6.50 and 7.70, corresponding to the discontinuities in each Figure.

Speed update increments of 0.01 RPM or less produced very good results as can be seen in Figures 6-19 and 6-20.

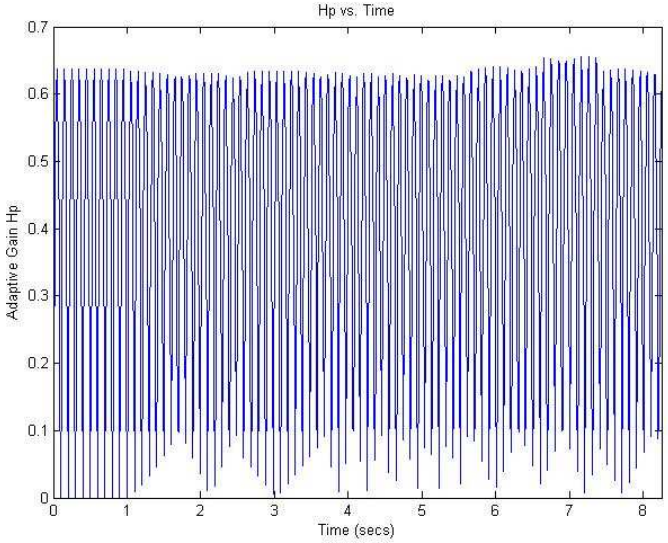


Figure 6-19 Continuous Adaptive Gain with Incremental Updates

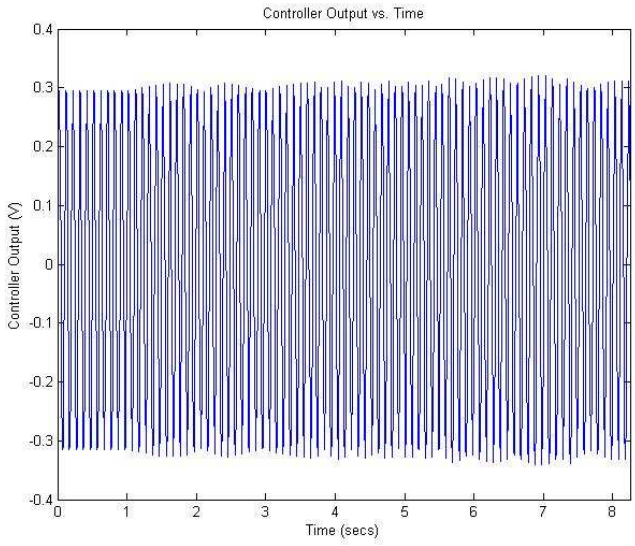


Figure 6-20 Continuous Controller Output with Incremental Updates

Increments this small created no discernible discontinuities in the graph of H_p with one possible exception at the update that began at 6.62 seconds or in the graph of controller output. Other updates to the reject frequency began at seconds 1.0 and 3.61. Variations in the amplitude of both the adaptive gain and the controller output were seen, but nothing sharp or discontinuous was detected. Overall, the results indicated that a continuous disturbance function could be approximated and that using the adaptive gain as a predictor of a change in the balance state of a system gaining or losing speed was possible.

Note that encouraging results were obtained when the speed update increment was an order of magnitude greater than the sample size of the simulation (0.01 vs. 0.001 second). Reject frequency updates of even 3 to 4 RPM require a substantial amount of time to complete when increments of this size are used even when the update interval is just four times the FSS. An update from 600 to 604 RPM required 0.16 second for the results shown in Figures 6-17 and 6-18, and the same update consumed 1.60 seconds for the results illustrated in Figures 6-19 and 6-20. An order of magnitude decrease in the update increment resulted in an order increase in the time needed to change the reject frequency. If the time to update the frequency becomes too large, the controller will follow the change in rotor speed poorly just as it will if the speed measurement and speed update intervals are too large.

One possible way to reduce the time required to update the reject frequency would be to perform updates at the speed of the simulation. The capability to do this was included in the controller operating in speed mode and is also included in the simulated controller shown in Figure 6-15. However, testing updates at simulation frequency with the original multi-mode controller could cause the controller to drive the rotor dangerously close to the bearings. The system did not become unstable, but controller outputs were calculated incorrectly and became too large. The simulated controller demonstrated the same behavior when updates were applied every sample step. Controller outputs could suddenly increase, though they always remained bounded, and then settle at a higher level before increasing again. Best results for both the real and simulated controllers were obtained when the reject frequency was updated no faster than three times the Fundamental Sample Size of the simulation.

6.6 Incrementally Varying Frequency Disturbance - Actual

Given the encouraging results obtained from the simulations, the *Adaptive Control* subsystem of the bearing/speed controller was modified to calculate the adaptive gains and controller outputs two ways using the existing path through the controller as well as a second parallel path added to the controller that incrementally updates the reject frequency. The blocks added to the modified subsystem are shown in Figure 6-21.

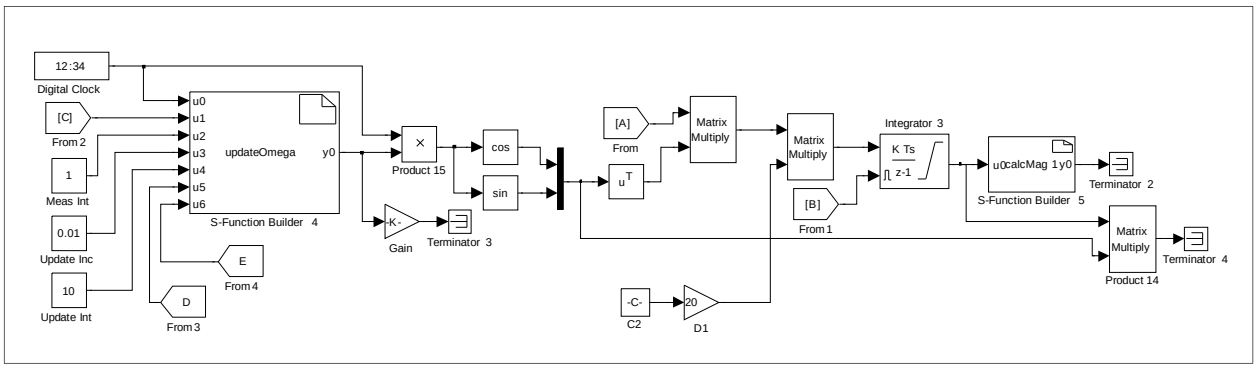
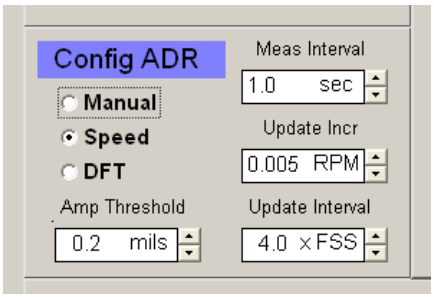


Figure 6-21 Modified *Adaptive Control* for Incremental Updates

Note that only the outputs calculated from the trusted, established path were applied to the bearings and that the rotor displacements used to compute the gains and outputs both ways

were the same. This “dual path” approach allowed for an accurate appraisal of the controller’s behavior when the reject frequency was updated without exposing the bearings to potential damage if the controller behaved differently that it did when simulated.

The ControlDesk panel *ADR/DFT* was also modified to accommodate the additional parameters needed to properly configure the controller to incrementally update the reject frequency. In particular, three instruments were added to the *Configure ADR* pane, and the modified pane is shown in Figure 6-22.



6-22 Modified Pane for Incremental Updates

Once the Simulink model and ControlDesk panel were changed, experiments were conducted to test the modified controller on the magnetic bearing system. The controller was initially tested with values for the configuration parameters that worked well in the simulations. Specifically, the speed measurement interval, speed update increment and speed update interval were chosen as 1.0 second, 0.005 RPM and four times the FSS, respectively. Results for this configuration were not as good as they were for the simulated controller. Figure 6-23 illustrates the adaptive gain calculated with a constant and a varying frequency as well as the rotor speed.

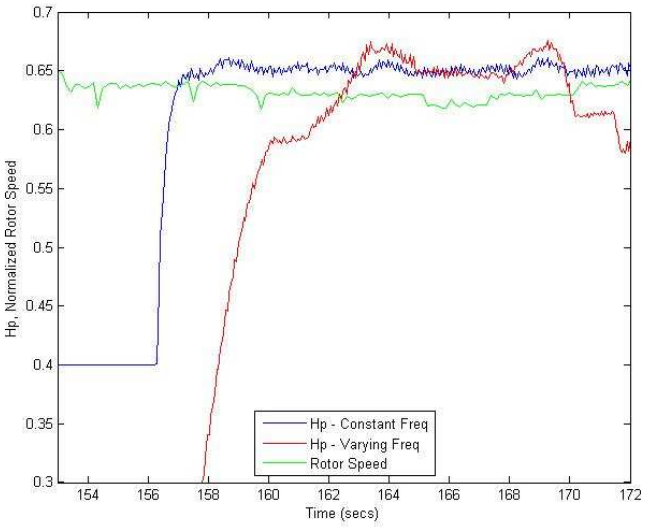


Figure 6-23 Gain Response to Incremental Updates and Rotor Speed Variation

The gain for the constant frequency case has been offset by a constant value so that the gains can be more easily compared. Also, the rotor speed has been normalized by the set point and offset so its relation to the gains can be more easily seen. Note that the varying frequency gain adapts more slowly and that it reacts more strongly to changes in rotor speed than does the constant frequency gain. Overall, the gains follow similar paths once they adapt if changes in rotor speed are small (±5 RPM). Over the interval from 165 to 167 seconds, speed variations are small, and the gain curves are similar.

The deviation in rotor speed from its set point and the frequency used in computing the gain for the varying case are shown in Figure 6-24.

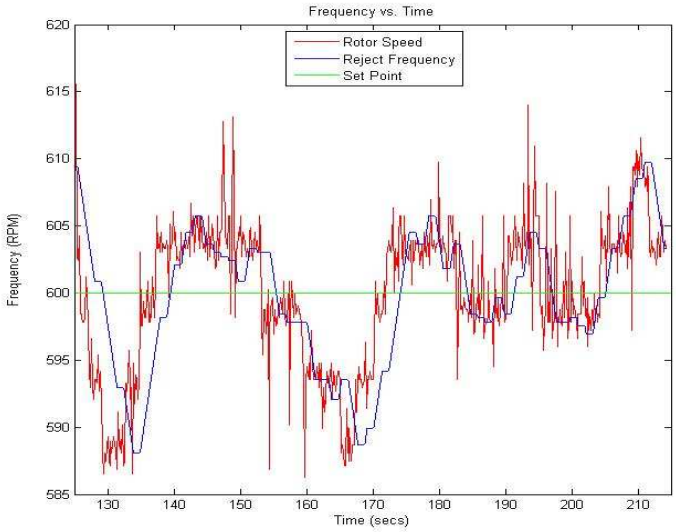


Figure 6-24 Reject Frequency and Rotor Speed Variations with Incremental Updates

The reject frequency follows the rotor speed very closely and tracks it nearly perfectly when changes in rotor speed are confined to a narrow range as they are from 175 to 209 seconds. When the changes are larger, the reject frequency lags the rotor speed. This can be easily seen for the interval from 125 to 141 seconds where the rotor speed decreases from 616 to 586 RPM and then increases to 602 RPM. Part of the lag is a consequence of the small increment that is used to update the reject frequency. The small size results in a frequency that is changed at a slower rate than the rate at which the rotor gains or loses speed. The difference in rates can be seen in the divergence of the slopes of the rotor speed and reject frequency curves. This difference is most notable for the interval from 125 to 134 seconds. Part of the lag is also caused by the method used to measure rotor speed and the length of the measurement interval. Measurement of rotor speed and updates to the reject frequency occur sequentially. Therefore, the reject frequency remains constant over the measurement interval. This can be seen in the Figure where the slope of the reject frequency curve is zero for a duration of one second. If the slope is zero for more than one second, no change in rotor speed was detected over the preceding measurement interval. Efforts to reduce or eliminate this lag are discussed later.

The controller outputs calculated from each of the gains appear in Figure 6-25 (The controller was activated at 156 seconds and deactivated at 173 seconds.).

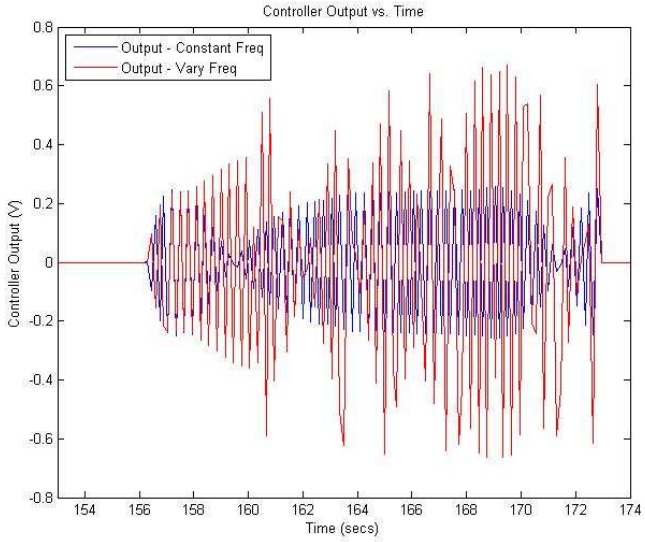


Figure 6-25 Controller Output with Incremental Updates

The control outputs are more variable and never attain a steady-state value when the reject frequency is changing rather than when it is constant. Since the outputs in both cases are computed using the same rotor displacements and weighting matrix, the difference in them can only be attributed to the incremental changes in frequency that continually occur in the composition of the disturbance vector in the one case.

Further tests were conducted with different configuration parameters to determine if improvements in the behavior of the adaptive gain could be found by reducing the lag between the reject frequency and rotor speed. The speed measurement interval was decreased to the order of the FSS so that frequency updates occurred nearly continuously with fewer than ten sample steps between any two. The speed update increment was increased to the largest size that worked successfully for the simulated controller so that frequency updates were made more rapidly. The speed update interval was also reduced but only by a single sample step since simulations and testing on the actual system have shown without exception that smaller intervals result in controller instability. In summary, speed measurement intervals from 0.0015 to 1.0 second, speed update increments from 0.001 to 0.02 RPM and speed

update intervals of three and four times the FSS were tried. No combination of configuration parameters improved the possibility of using the adaptive gain as an indicator of a change in the balance of the magnetic bearing system. In fact, all combinations produced results very similar to those presented in Figures 6-23, 24 and 25.

Chapter 7 Conclusions and Future Work

The adaptive controller for the magnetic bearing system was originally designed to reject persistent disturbances acting at synchronous (rotor speed) frequencies. As part of this research, the controller was redesigned so that it would also work effectively when rotor speed was varying and when disturbances were transient and asynchronous in character. The redesign allowed the controller to operate in three different modes, DFT, Speed and Manual, the mode indicating the method used to determine the frequency of the disturbance to reject.

Disturbance rejection was successful when the controller was operated in any of the modes. Although experimental results were not presented for the controller operating in DFT mode, tests showed that rejection was very good when this mode was used to suppress software generated disturbances. Results were mixed when the rotor speed was used to drive the controller. If the speed was nearly constant, disturbance suppression was very good. If the speed varied, rejection would range from very effective to completely ineffective. The controller operated as designed, but changes in rotor speed introduced discontinuities in the disturbance functions causing momentary loss of control effort.

The modeling done to improve the behavior of the controller in Speed mode suggested that this mode could be used effectively when rotor speed is changing. Tests show that rejection of a single frequency can significantly reduce rotor displacements when rotor speed varies over a small range. A controller could be built that incorporates two sub controllers that operate alternately with each controller using a constant reject frequency to provide disturbance rejection for a system operating over a wide range of speeds. The first sub controller could be used until suppression was no longer adequate. It would then be deactivated, and the second would be activated and operated until it was no longer effective and so on. Alternating between the two would eliminate the problems induced by discontinuities in the disturbance functions while maintaining controller effectiveness as rotor speed changes.

During development of the multi-mode adaptive controller, the increasing complexity of the Simulink model eventually overwhelmed the dSPACE system, and the model would not run

in real time. Careful refinement of the model and careful reconfiguration of the run-time environment resolved problems with simulation speed and processor load and ensured potential for future development of the magnetic bearing system. However, changes or additions to the model should still only be made with a knowledge of their impact on processing time.

Attempts to establish the predictive capability of the adaptive gain H_p proved inconclusive. Previous [28] as well as current research have demonstrated the gain's sensitivity to small variations in rotor speed. The reaction of the gain to the change in speed makes it very difficult to determine if a change in the gain is caused by a change in the balance of the rotating system or in the speed of the system. It was shown that the gain reacts strongly to a change in balance that is also observable in the rotor displacements, but it has yet to be shown conclusively that the gain reacts reliably and predictably to a change in balance that cannot be detected any other way.

The magnetic bearing system provides opportunities for continuing research with the currently implemented adaptive controller and provides a versatile platform that could be used to investigate other adaptive strategies discussed in the literature. To establish whether gain H_p is truly predictive or not, the speed of the rotor must be controlled very precisely. Methods to reduce oscillations in rotor speed about the set point are presented in [30]. Mechanical limitations of the flow control valve that may limit precision are also discussed here. Other changes to the regulation of the air supply to the turbine in addition to the control of it may be needed to nearly eliminate variations in rotor speed.

A knowledge of the exact time when the balance of the system changes would also benefit the investigation of gain variations. The results presented in Chapter 6 for simulated imbalances show the exact time when the balance of the rotor changed. Devising a method to physically unbalance the rotor and to capture the precise moment when it occurs would be worthwhile. Colleagues have suggested that a small electro-mechanical device could be designed and built that would mount to the flywheel. The device would retain and then release a known weight to unbalance the system. Radio signals would be used to communicate to the device and

overall control of it would be through the dSPACE system allowing the exact time of the balance change to be recorded. Of course, balance changes seldom occur at predefined moments on any system, but knowing exactly when they do occur on a research system would be very helpful in identifying an attendant but subtle variation in the adaptive gain curve.

The need to know how changes to the Simulink bearing/speed controller affect the execution time of the model on the dSPACE processor has been emphasized. Profiling was discussed as one method of determining the relative impact of a change on processing time. The dSPACE system also provides a number of real-time variables that record how heavily each task uses the processor and indicate whether a task places the system at risk of overrun errors [31]. These variables can be monitored through standard ControlDesk instruments. A ControlDesk panel should be created or a pane should be added to an existing panel, following the established conventions, to display these variables if any changes are made to the current system that could affect processing load.

Monitoring processor load is one way to more safely operate the FACETS system and avoid possible damage to it. An additional way would be to design, build and incorporate retainer bearings into the current system. Retainer bearings are often if not always used together with magnetic bearings in industrial applications. Retainer bearings are normal rolling element bearings that support the rotor when the magnetic bearings are de-energized or if the magnetic bearings should fail for some reason [37]. Retainer bearings prevent damage to the rotor and magnets by eliminating contact between the two. The small air gap between the rotor and bearings on the current system would create challenges in the design of retainers, but any research use of the system would proceed much more quickly if possible damage to it was not always an immediate consideration.

Given the development of the multi-mode adaptive controller, the magnetic bearing system can now be used to study the effects of disturbances acting at unknown frequencies and resulting from base motion. In fact, [26] had suggested investigating the effects of base motion on the system as a future task. Other adaptive control methods would also be worth investigating as suggested again by [26] especially if the work to include the monitoring of

processor use and to incorporate retainer bearings into the system were done to provide protection for the system during research into the unknown.

References

[1] L.Ting and J. Tichy, “Stiffness and Damping of an Eddy Current Magnetic Bearing System,” *Journal of Tribology*, July, 1992, pp. 600-604.

[2] J.K. Fremerey, “A 500-Wh Power Flywheel on Permanent Magnetic Bearings,” *Fifth International Symposium on Magnetic Suspension Technology*, 2000, pp. 287-295.

[3] R. Jansen and E. DiRusso, “Passive Magnetic Bearing with Ferrofluid Stabilization,” *NASA*, 1996, TM-107154

[4] M. Mekhiche, S. Nichols, J. Oleksy, J. Young, J. Kiley and D. Havenhill, “50 KRPM, 1100^{o} F Magnetic Bearings for Jet Turbine Engines,” *Seventh International Symposium on Magnetic Bearings*, 2000, pp.123-128.

[5] G. Schweitzer, “Active Magnetic Bearings – Changes and Limitations,” *Sixth International Conference on Rotor Dynamics*, 2002.

[6] D.C. Pang, J.L. Lin, B.Y. Shew and R.B. Zmood, “Design, Fabrication and Testing of a Millimeter-Level Magnetically Suspended Micro-Motor,” *Seventh International Symposium on Magnetic Bearings*, 2000, pp. 105-110.

[7] J. Inoue, Y. Araki and S. Miyaura, “On the Self Synchronization of Mechanical Vibrators,” *Bulletin of the Japan Society of Mechanical Engineers*, 1967, pp. 755-762.

[8] P. Bovik and C. Hogfors, “Autobalancing of Rotors,” *Journal of Sound and Vibration*, December, 1986, pp. 429-440.

[9] C.R. Burrows and M.N. Sahinkaya, “Vibration Control of Multi-Mode Rotor-Bearing System,” *Proceedings of the Royal Society of London*, March, 1983, pp.77-94.

[10] C.R. Knospe, “Stability and Performance of Notch Filter Control for Unbalance Response,” *International Symposium on Magnetic Suspension Technology*, 1991, pp. 181-205.

[11] R. Herzog, P. Buhler, C. Gahler and R. Larsonneur, “Unbalance Compensation Using Generalized Notch Filters in the Multivariable Feedback of Magnetic Bearings,” *IEEE Transactions on Control Systems Technology*, September, 1996, pp. 580-586.

[12] B. Shafai, S. Beale, P. LaRocca and E. Cusson, “Magnetic Bearing Control Systems and Adaptive Forced Balancing,” *IEEE Control Systems*, April, 1994, pp. 4-13.

[13] R. Larsonneur and R. Herzog, “Feedforward Compensation of Unbalance: New Results and Application Examples,” *IUTAM Symposium on Active Control and Vibrations*, September, 1994, pp. 45-52.

[14] K. Lum, V.T. Coppola and D.S. Bernstein, “Adaptive Autocentering Control for an Active Magnetic Bearing with Unknown Mass Imbalance,” *IEEE Transactions on Control Systems Technology*, 1996, pp. 587-597.

[15] S. Huang and L. Lin, “Fuzzy Dynamic Output Feedback Control with Adaptive Rotor Imbalance Compensation for Magnetic Bearing Systems,” *IEEE Transactions on Systems, Man and Cybernetics*, August, 2004, pp. 1854-1864.

[16] T. Grochmal and A.F. Lynch, “Precision Tracking of a Rotating Shaft with Magnetic Bearings by Nonlinear Decoupled Disturbance Observers,” *IEEE Transactions on Control Systems Technology*, November, 2007, pp. 1112-1121.

[17] X. Wen, “Enhanced Disturbance Observer Based Control for a Class of Time Delay Systems with Uncertain Sinusoidal Disturbances,” *Mathematical Problems in Engineering*, 2013, pp. 1-8.

[18] C.R. Burrows, P.S. Keogh and M.N. Sahinkaya, “Progress Towards Smart Rotating Machinery Through the Use of Active Bearings,” *Journal of Mechanical Engineering Science*, December, 2009, pp. 2849-2859.

[19] A.G. Abulrub, M.N. Sahinkaya, P.S. Keogh and C.R. Burrows, “Adaptive Control of Active Magnetic Bearings to Prevent Rotor-Bearing Contact,” *Proceedings of ASME International Mechanical Engineering Congress and Exposition*, November, 2006.

[20] M.O.T. Cole, P.S. Keogh and C.R. Burrows, “Robust Control of Multiple Discrete Frequency Vibration Components in Rotor-Magnetic Bearing Systems,” *JSME International Journal*, 2003, pp. 891-899.

[21] P.S. Keogh, M.O.T. Cole and C.R. Burrows, “Multi-state Transient Rotor Vibration Control Using Sampled Harmonics,” *Journal of Vibration and Acoustics*, 2002, pp. 186-197.

[22] M.N. Sahinkaya and A.E. Hartavi, “Variable Bias Current in Magnetic Bearings for Energy Optimization,” *IEEE Transactions on Magnetics*, March 2007, pp. 1052-1060.

[23] M.O.T. Cole, P.S. Keogh and C.R. Burrows, “Vibration Control of a Flexible Rotor/Magnetic Bearing System Subject to Direct Forcing and Base Motion Disturbances,” *Journal of Mechanical Engineering Science*, July, 1998, pp. 535-546.

[24] M.S. Kang, J. Lyou and J. Lee, “Sliding Mode Control for an Active Magnetic Bearing System Subject to Base Motion,” *Mechatronics*, February, 2010, pp. 171-178.

[25] R.J. Fuentes and M.J. Balas, “Direct Adaptive Rejection of Persistent Disturbances,” *Journal of Mathematical Analysis and Applications*, 2000, pp. 28-39.

[26] A. Matras, “Implementation of Adaptive Disturbance Rejection Control in an Active Magnetic Bearing System,” *Master's Thesis*, Auburn University, Auburn, AL, 2003.

[27] A. Matras, “Applied Adaptive Disturbance Rejection Using Output Redefinition on Magnetic Bearings,” *PhD Dissertation*, University of Colorado, Boulder, CO, 2005

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

[29] A. Matras, G. Flowers, R. Fuentes, M. Balas and J. Fausz, "Suppression of Persistent Rotor Vibrations Using Adaptive Techniques," *Transactions of the ASME*, December, 2006, pp. 682-689.

[30] R. Jantz, "Controlling the Speed of a Magnetically Suspended Rotor with Compressed Air," *Master's Thesis*, Auburn University, Auburn, AL, 2011.

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

[32] T.G. Beckwith, R.D. Marangoni and J.H. Lienhard, *Mechanical Measurements*, Addison-Wesley, New York, 1993, pp.146, 150 and 155.

[33] L.H. Miller and A.E. Quilici, *Programming in C*, John Wiley and Sons, New York, 1986, pg. 159.

[34] *Real-Time Interface Implementation Guide*, dSPACE, Paderborn, Germany 2002.

[35] F. Shiue, G.A. Lesieutre and C.E. Bakis, "Condition Monitoring of Filament-Wound Composite Flywheels Having Circumferential Cracks," *Journal of Spacecraft and Rockets*, March-April 2002, pp. 306-312.

[36] K. Barber, G.T. Flowers, A. Matras, M. Balas and J. Fausz, "Imbalance Detection and Health Monitoring from Gain Variations in an Adaptive Disturbance Rejection Controller," *ASME*, 2007.

[37] H. Dell, J. Engel, R. Faber, D. Glass, "Developments and Tests on Retainer Bearings for Large Active Magnetic Bearings," *Magnetic Bearings*, 1989, pp. 211-224.

Appendix A

MATLAB Program

ADR_three.m

```
%{
James Jantz

Last revision: November 15, 2012

ADR_three - solves the equations of motion for a spring-mass-damper system
and plots displacements; calculates controller inputs based on adaptive
techniques and plots displacements for the same system; plots controller
inputs.
%}

function ADR_three
  global useADR;
  fprintf('\n==> Rotating System - ADR Analysis <== \n')

  while true  % Execute as many analyses as the user desires.
    close all;  % Close open figure windows.

    % Define default char. of the sytem.
    M = 5.0;          % Mass (kg)
    w_natl = 10.0;    % Natural frequency (Hz)
    zeta = 0.01;      % Damping ratio
    w_dist = 20.0;    % Disturbance frequency (Hz)
    imbal_pct = 0.1;  % Imbalance as a percent of the total mass
    G = 1.0;          % Constant for gain Gp
    H = 100.0;        % Constant for gain Hp
    t_end = 60.0;     % Simulation length (sec)

    fprintf('\nDefault values for the systems are:\n')
    fprintf('  Mass = %4.1f kg\n', M)
    fprintf('  Natural freq. = %4.1f Hz\n', w_natl)
    fprintf('  Damping ratio = %4.2f\n', zeta)
    fprintf('  Disturb. freq. = %4.1f Hz\n', w_dist)
    fprintf('  Imbalance pct. = %4.2f \n', imbal_pct)
    fprintf('  Gp constant = %4.1f \n', G)
    fprintf('  Hp constant = %4.1f \n', H)
    fprintf('  Simulation len. = %4.1f sec\n', t_end)

    fprintf('\nDo you want to use the default values?')
    msg = sprintf('\nPress "Enter" for yes or type "n" for no:  ');
    reply = input(msg, 's');  % Read it as a string.

    if strcmp(reply, 'n') || strcmp(reply, 'N')
      msg = sprintf('\nEnter the mass (kg) of the system [%4.1f]:  ', M);
```

```
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        M = sscanf(reply, '%f', inf);
      end

      clear msg;
      msg = sprintf('\nEnter the nat''l freq. (Hz) of the system [%4.1f]:
', w_natl);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        w_natl = sscanf(reply, '%f', inf);
      end

      clear msg;
      msg = sprintf('\nEnter the damping ratio of the system [%4.2f]:  ',
zeta);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        zeta = sscanf(reply, '%f', inf);
      end

      clear msg;
      msg = sprintf('\nEnter the disturb. freq. (Hz) of the system [%4.1f]:
',...
        w_dist);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        w_dist = sscanf(reply, '%f', inf);
      end

      clear msg;
      msg = sprintf('\nEnter the imbalance (pct. of mass) of the system
[%4.2f]:  ',...
        imbal_pct);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        imbal_pct = sscanf(reply, '%f', inf);
      end

      clear msg;
      msg = sprintf('\nEnter the constant for gain Gp [%4.1f]:  ', G);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        G = sscanf(reply, '%f', inf);
      end

      clear msg;
      msg = sprintf('\nEnter the constant for gain Hp [%4.1f]:  ', H);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        H = sscanf(reply, '%f', inf);
      end
```

```
      clear msg;
      msg = sprintf('\nEnter the length (sec) of the simulation [%4.1f]:
', t_end);
      reply = input(msg, 's');  % Read it as a string.
      if ~isempty(reply)        % Use default unless a value is entered.
        t_end = sscanf(reply, '%f', inf);
      end
    end

    % Change the sign on delta H if the dist. freq. is less than the nat'l
    % freq.
    if (w_dist < w_natl)
      H = -abs(H);  % In case the user entered a negative value.
    end

    fprintf('\nValues used for the simulation are:\n')
    fprintf('  Mass = %4.1f kg\n', M)
    fprintf('  Natural freq. = %4.1f Hz\n', w_natl)
    w_natl = w_natl*2*pi;     % Convert to rad/s.
    fprintf('  Damping ratio = %4.2f\n', zeta)
    fprintf('  Disturb. freq. = %6.3f Hz\n', w_dist)
    w_dist = w_dist*2*pi;     % Convert to rad/s.
    fprintf('  Imbalance pct. = %4.2f \n', imbal_pct)
    fprintf('  Gp constant = %4.1f \n', G)
    fprintf('  Hp constant = %4.1f \n', H)
    fprintf('  Simulation len. = %4.1f sec\n', t_end)

    % Calculate constants.
    K = M*w_natl^2;     % Spring constant (N/m)
    Ccr = 2*sqrt(K*M);  % Critical damping coeff. (kg/s)
    C = zeta*Ccr;       % Damping coeff. (kg/s)

    % Define char. of imbalance.
    m = imbal_pct/100 * M;  % Mass (kg)
    r = 10.0;               % Location - distance from center of rotation (mm)
    F = m*(r/1000)*w_dist^2;  % Force (N)
    fprintf('  Imbalance force = %4.3f N\n', F)

    sys_char = [M K C w_dist F G H];  % Create a pseudo structure.

    % Time interval.
    t_beg = 0;
    tspan = [t_beg t_end];

    % Initial conditions.
    x_i = 0;  % Displacements
    y_i = 0;
    xDot_i = 0;  % Velocities
    yDot_i = 0;
    % Expand the initial conditions vector.  The last 4 elements are needed
    % to solve the equations for GpDot and HpDot.
    i_cond = [x_i y_i xDot_i yDot_i 0 0 0 0];
```

```
    useADR = false;

    % Calculate displacements without ADR control.
    [T z] = ode45(@calcDispl,tspan,i_cond,[],sys_char);

    % Graph them.
    subplot(2,1,1), plot(T,z(:,1)*1000, 'r')  % x-displacement (mm)
    title('X-Displacement vs. Time')
    xlabel('Time (sec)'), ylabel('Displacement (mm)')
    legend('no ADR','location', 'southeast')
    grid on
    hold on

    subplot(2,1,2), plot(T,z(:,2)*1000, 'r')  % y-displacement (mm)
    title('Y-Displacement vs. Time')
    xlabel('Time (sec)'), ylabel('Displacement (mm)')
    legend('no ADR','location', 'southeast')
    grid on
    hold on

    fprintf('\nPress any key to continue and graph displacements with ADR
control.\n')
    pause

    % Calculate displacements with ADR control.
    useADR = true;
    [T z] = ode45(@calcDispl,tspan,i_cond,[],sys_char);

    % Extract values.  These are returned by the solver.
    x_disp = z(:,1);
    y_disp = z(:,2);
    Gpx = z(:,5);
    Gpy = z(:,6);
    Hpx = z(:,7);
    Hpy = z(:,8);
    s = sin(w_dist*T);
    c = cos(w_dist*T);

    % Calculate Gp, Hp, Upx and Upy.
    Gp = zeros(length(T),1);  % Preallocate for a happy MATLAB.
    Hp = zeros(length(T),1);
    Upx = zeros(length(T),1);
    Upy = zeros(length(T),1);

    for i = 1:length(T)
      Gp(i) = sqrt(Gpx(i)^2 + Gpy(i)^2);
      Hp(i) = sqrt(Hpx(i)^2 + Hpy(i)^2);
      Upx(i) = Gpx(i)*x_disp(i) + Hpx(i)*s(i);
      Upy(i) = Gpy(i)*y_disp(i) + Hpy(i)*c(i);
    end

    % Graph displacements with ADR control.
```

```
    subplot(2,1,1), plot(T,z(:,1)*1000, 'b')  % x-displacement (mm)
    title('X-Displacement vs. Time')
    xlabel('Time (sec)'), ylabel('Displacement (mm)')
    legend('no ADR','ADR','location', 'southeast')

    subplot(2,1,2), plot(T,z(:,2)*1000, 'b')  % y-displacement (mm)
    title('Y-Displacement vs. Time')
    xlabel('Time (sec)'), ylabel('Displacement (mm)')
    legend('no ADR','ADR','location', 'southeast')

    fprintf('\nPress any key to continue and plot Upx and Upy.\n')
    pause

    % Plot adaptive gains (Gp and Hp).
    close all
    subplot(2,1,1), plot(T,Gp,'b')
    title('Gp vs. Time')
    xlabel('Time (sec)'), ylabel('Gp')
    grid on

    subplot(2,1,2), plot(T,Hp,'b')
    title('Hp vs. Time')
    xlabel('Time (sec)'), ylabel('Hp')
    grid on

    fprintf('\nPress any key to clear the figure and continue.\n')
    pause

    % Plot controller inputs (Upx and Upy) to plant.
    close all
    subplot(2,1,1), plot(T,Upx, 'b')
    title('Upx vs. Time')
    xlabel('Time (sec)'), ylabel('Upx')
    grid on

    subplot(2,1,2), plot(T,Upy, 'b')
    title('Upy vs. Time')
    xlabel('Time (sec)'), ylabel('Upy')
    grid on

    fprintf('\nPress any key to clear the figure and continue.\n')
    pause
    close all
    fprintf('\nDo you want to perform another analysis?')
    msg = sprintf('\nPress "Enter" for yes or type "n" for no:  ');
    choice1 = input(msg, 's'); % Read the user's response.
    switch choice1 % Decide what to do.
    case { 'N', 'NO', 'n', 'no'} % Quit this program.
      break
    end
  end
end
```

```
function dz = calcDispl(t,z,sys_char)
  global useADR;

  % Assign variables.
  M = sys_char(1);
  K = sys_char(2);
  C = sys_char(3);
  w_dist = sys_char(4);
  F = sys_char(5);
  G = sys_char(6);
  H = sys_char(7);

  % Basis vectors.
  s = sin(w_dist*t);
  c = cos(w_dist*t);

  dz = zeros(8,1);  % Preallocate, emphasizing style here.

  % Equations of motion.
  dz(1) = z(3);  % dZx1 = Zx2
  dz(2) = z(4);  % dZy1 = Zy2

  if ~useADR
    % Equations of motion 3 and 4.
    dz(3) = -(K/M * z(1) + C/M * z(3)) + 1/M * F*s;
    dz(4) = -(K/M * z(2) + C/M * z(4)) + 1/M * F*c;
  end

  if useADR
    % Calculate Gpxdot and Gpydot.
    dz(5) = -G * (z(1)^2 + z(1)*z(2));
    dz(6) = -G * (z(2)*z(1) + z(2)^2);

    % Calculate HpxDot and HpyDot.
    dz(7) = H * (z(1)*s + z(1)*c);
    dz(8) = H * (z(2)*s + z(2)*c);

    Upx = z(5)*z(1) + z(7)*s;
    Upy = z(6)*z(2) + z(8)*c;

    dz(3) = -(K/M * z(1) + C/M * z(3)) + (1/M * F*s) + Upx;
    dz(4) = -(K/M * z(2) + C/M * z(4)) + (1/M * F*c) + Upy;
  end
end
```

Appendix B

S-Functions

mbcntrl_a.h

```
/*
 * Name:    mbcntrl_a.h
 * Author:  Jimbo
 * Created: 3/15/13
 * Revision History:
 *
 * Purpose: provides definitions for the S-functions.
 */

/* M_PI is defined in <math.h>, but the RTW compiler can't seem to find it,
   so we'll set it up like this. */
#ifndef M_PI
  #define M_PI 3.14159265358979323846
#endif
#define TWO_PI 2 * M_PI

/* Keep an eye on these. They may well have been defined elsewhere. If so,
   surround them with an ifndef construct. */
#define TRUE 1
#define FALSE 0

/* The states of the internal excitation subsystem and the ways it can be
   driven. */
#define ACTIVE 1
#define INACTIVE 0
#define MAN_EXC 1
#define SPEED_EXC 2

/* The various ways the adaptive controller can be driven. */
#define MAN_MODE 1
#define SPEED_MODE 2
#define DFT_MODE 3

/* The fundamental sample size of the simulation. See note in oneFreq.c */
#define FSS 0.0003

/* Max. sample size used to allocate space. Actual size is computed based
   on the DFT freq. and freq. resolution set via ControlDesk. */
#define MAX_DFT_SIZE 2400
#define MAX_NUM_FREQ 3
#define FREQ_THRESHOLD 1 * TWO_PI  /* rad/sec */
```

Appendix B

S-Functions

zeroGp_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void zeroGp_Outputs_wrapper(constreal_T *u0,
                        constreal_T *u1,
                        real_T *y0,
                        real_T *y1, SimStruct *S)
{

/* This function monitors the output from the integrator's saturation port.
   If the integrator saturates, the output from it is nulled and remains
   nulled, even if the integrator becomes unsaturated, until it is reset
   manually. */

/* Inputs are as follow:
     u0[0] - Saturation signal.
     u1[0] - Gp active signal. */

/* Outputs are as follow:
     y0[0] - Gain multiplier - 0 if the integrator has sat'd, 1 otherwise.
     y1[0] - Status signal - 0 if the int. has sat'd, 1 if it hasn't, 2 if
             the gain is inactive. */

int satSig, GpAct;
static int firstSat = FALSE;

satSig = (int) u0[0];
GpAct = (int) u1[0];

if (GpAct == FALSE)  /* If Gp isn't active, reset the saturation flag, */
{
  firstSat = FALSE;
  y0[0] = 0;  /* output zero just to be safe, */
  y1[0] = 2;  /* and set the status port to inactive. */
  return;
}
```

```
if (satSig == 0)  /* If the integrator isn't saturated */
{
  if (firstSat == TRUE)  /* but it has been, */
  {
    y0[0] = 0;  /* continue to zero its output, */
    y1[0] = 0;  /* and set the status port to sat. */
  }
  else
  {
    y0[0] = 1;  /* If it isn't saturated and never has been, integrate! */
    y1[0] = 1;
  }
  return;
}

if (satSig != 0)  /* If it is saturated */
{
  if (firstSat == FALSE)  /* but it hasn't been */
  {
    firstSat = TRUE;  /* set the flag */
    y0[0] = 0;  /* and zero the output. */
    y1[0] = 0;
  }
  else
  {
    y0[0] = 0;  /* If it is saturated and has been, continue to zero the */
    y1[0] = 0;  /* output. */
  }
  return;
}
}
```

Appendix B

S-Functions

zeroHp_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void zeroHp_Outputs_wrapper(const real_T *u0,
                        const real_T *u1,
                        real_T *y0,
                        real_T *y1, SimStruct *S)
{

/* This function monitors the output from the integrator's saturation port.
   If the integrator saturates, the output from it is nulled and remains
   nulled, even if the integrator becomes unsaturated, until it is reset
   manually. */

/* Inputs are as follow:
     u0[0] - Saturation signal.
     u1[0] - Hp active signal. */

/* Outputs are as follow:
     y0[0] - Gain multiplier - 0 if the integrator has sat'd, 1 otherwise.
     y1[0] - Status signal - 0 if the int. has sat'd, 1 if it hasn't, 2 if
     the gain is inactive. */

int satSig, HpAct;
static int firstSat = FALSE;

satSig = (int) u0[0];
HpAct = (int) u1[0];

if (HpAct == FALSE)  /* If Hp isn't active, reset the saturation flag, */
{
  firstSat = FALSE;
  y0[0] = 0;  /* output zero just to be safe, */
  y1[0] = 2;  /* and set the status port to inactive. */
  return;
}
```

```
if (satSig == 0)  /* If the integrator isn't saturated */
{
  if (firstSat == TRUE)  /* but it has been, */
  {
    y0[0] = 0;  /* continue to zero its output, */
    y1[0] = 0;  /* and set the status port to sat. */
  }
  else
  {
    y0[0] = 1;  /* If it isn't saturated and never has been, integrate! */
    y1[0] = 1;
  }
  return;
}

if (satSig != 0)  /* If it is saturated */
{
  if (firstSat == FALSE)  /* but it hasn't been */
  {
    firstSat = TRUE;  /* set the flag */
    y0[0] = 0;  /* and zero the output. */
    y1[0] = 0;
  }
  else
  {
    y0[0] = 0;  /* If it is saturated and has been, continue to zero the */
    y1[0] = 0;  /* output. */
  }
  return;
}
}
```

Appendix B

S-Functions

zeroADR_wrapper.c

```
/*--- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---   */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void zeroADR_Outputs_wrapper(constreal_T *u0,
                          constreal_T *u1,
                          real_T *y0, SimStruct *S)
{

/* This function controls the output of the adaptive controller. If the
   controller is active (turned on) and no limit errors have been detected,
   the controller's output is passed through, otherwise it is nulled. */

/* Inputs are as follow:
     u0[0] - ADR error signal.
     u1[0] - ADR active switch. */

/* Output is as follows:
     y0[0] - 0 if a current or position limit has been exceeded or if the
             ADR controller is inactive, 1 otherwise. */

int errSig, ADRAct;

errSig = (int) u0[0];
ADRAct = (int) u1[0];

if (errSig == TRUE)  /* If the error flag is set, output 0. */
{
  y0[0] = 0;
  return;
}

if (ADRAct == TRUE)  /* If the ADR controller is active and there is no */
  y0[0] = 1;         /* limit error, output 1. */
else
  y0[0] = 0;  /* If the ADR controller is inactive, output 0. */

return;
}
```

Appendix B

S-Functions

calcMag_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

#include <math.h>

#define u_width 4
#define y_width 1

void calcMag_Outputs_wrapper(constreal_T *u0,
                             real_T *y0)
{

/* This function calculates the magnitude of the Hp gain for each of the
   four axes. */

/* Inputs are as follow:
     u0[0-7] - orthogonal components of Hp for all four axes. */

/* Outputs are as follow:
     y0[0-3] - Hp for all four axes. */

int i;
double HpMag[4] = {0};

for (i = 0; i<= 3; i++)
{
  HpMag[i] = sqrt(pow(u0[i], 2.0) + pow(u0[i+4], 2.0));
  y0[i] = HpMag[i];
}

return;

}
```

Appendix B

S-Functions

excFreq_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void excFreq_Outputs_wrapper(const real_T *u0,
                          const real_T *u1,
                          const real_T *u2,
                          const real_T *u3,
                          real_T *y0, SimStruct *S)
{

/* This function controls the frequency of the excitation applied to the
   bearings. The frequency can be a static value, or it can follow the
   speed of the rotor. In the latter case, the frequency can be updated at
   each simulation step or at a longer interval depending on configuration.
*/

/* The fundamental sample size (FSS) of the simulation is used to control
   the timer for the frequency output, and this value is set in header file
   mbcntrl_a.h. */

/* Enable printing by uncommenting the following #define. */
/* #define ENABLE_PRTG */

/* Inputs are:
     u0[0] - Excitation mode - 1 manual, 2 speed.
     u1[0] - Excitation frequency (rad/sec) - manual mode.
     u2[0] - Constant-output interval (sec).
     u3[0] - Speed of the rotor (RPM). */

/* Output is:
     y0[0] - Excitation frequency (rad/sec). */

int excMode;
static int newInt = TRUE;
double manExcFreq, intT, speed;
static double intTimer, curFreq = 0;
```

```
excMode = (int) u0[0];  /* Fetch values from ControlDesk. */
manExcFreq = u1[0];
intT = u2[0];
speed = u3[0] * 2.0 * M_PI / 60.0;  /* Convert rotor speed to rad/sec. */

/* Ensure that an interval of 0 (i.e. excitation freq. follows the rotor
   speed exactly) is handled properly. */
if (excMode == SPEED_MODE && intT <= FSS)
{
  y0[0] = speed;
  newInt = TRUE;
  return;
}

if (newInt == TRUE)  /* Begin a new interval. */
{
  intTimer = intT;  /* Set the interval timer to the interval. */
  /* Choose the excitation freq. based on the value of the exc. mode
     switch. */
  switch (excMode)
  {
    case 1: curFreq = manExcFreq;  /* Manual Mode. */
            break;
    case 2: curFreq = speed;  /* Speed Mode. */
            break;
    default: curFreq = manExcFreq;  /* In case something gets through
             break;                 /* the net. */
  }
  newInt = FALSE;  /* Forget this, and you're toast. */
}

#ifdef ENABLE_PRTG
  printf("intTimer = %7.3f \n", intTimer);
  printf("speed = %6.2f ADR mode = %2i \n", speed, excMode);
  printf("Freq = %6.2f \n", curFreq);
#endif

y0[0] = curFreq;  /* Output the freq. */

intTimer = intTimer - FSS;  /* Decrement the timer and check. */
if (intTimer > 0.0)
  newInt = FALSE;
else
  newInt = TRUE;

return;

}
```

Appendix B

S-Functions

excRotor_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void excRotor_Outputs_wrapper(const real_T *u0,
                          real_T *y0,
                          real_T *y1, SimStruct *S)
{

/* This function monitors the current and previous states of the
   Excitation subsystem, enables or disables the excitation
   accordingly and outputs the current state of the subsystem. */

/* Input is:
     u0[0] - Excitation active signal. */

/* Outputs are:
     y0[0] - u0[0]
     y1[0] - Status - TRUE (excitation on) or FALSE (excitation off) */

static int excActive = FALSE, lastState = INACTIVE;
static double taskT;

excActive = (int) u0[0];

if (excActive == FALSE)  /* If the excitation is inactive */
  if (lastState == INACTIVE)  /* and has been, */
    y0[0] = 0;  /* do not excite. */
  else  /* If the excitation is inactive and hasn't been, */
  {
    y0[0] = 0;  /* stop the excitation, */
    y1[0] = FALSE;  /* update the status for ControlDesk */
    lastState = INACTIVE;  /* and change the state. */
  }

if (excActive == TRUE)  /* If the excitation is active */
  if (lastState == ACTIVE)  /* and has been,*/
    y0[0] = 1;  /* continue exciting. */
```

```
  else  /* If the excitation is active and hasn't been, */
  {
    y0[0] = 1;  /* excite, */
    y1[0] = TRUE;  /* update the status for ControlDesk and */
    lastState = ACTIVE;  /* change the state. */
  }

return;

}
```

Appendix B

S-Functions

genFreqs_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void genFreqs_Outputs_wrapper(const real_T *u0,
                          const real_T *u1,
                          const real_T *u2,
                          const real_T *u3,
                          real_T *y0, SimStruct *S)
{

/* This function calculates the frequencies to use in the composition of
   the disturbance vector. The frequencies are based on the speed of the
   rotor and always include at least this speed. Others freq's can also be
   included that are either multiples or fractions of this speed. The
   freq's are either updated at the simulation speed or at longer intervals
   depending on configuration. */

/* The fundamental sample size (FSS) of the simulation is used to control
   the timer for the frequency output, and this value is set in header file
   mbcntrl_a.h. */

/* Enable printing by uncommenting the following #define. */
/* #define ENABLE_PRTG */

/* Inputs are as follow:
     u0[0] - ADR mode.
     u1[0] - Disturbance vector fill mode.
     u2[0] - Constant-output interval (sec).
     u3[0] - Speed of the rotor (RPM). */

/* Outputs are as follow:
     y0[0,1,2] - Reject frequencies (rad/sec). */

int i, ADRMode, PhiDMode;
static int newInt = TRUE;
double intT, speed;
```

```
static double intTimer, curFreqs[MAX_NUM_FREQ] = {0};

ADRMode = (int) u0[0];  /* Fetch values from ControlDesk. */
PhiDMode = (int) u1[0];
intT = u2[0];
speed = u3[0] * 2.0 * M_PI / 60.0;  /* Convert rotor speed to rad/sec. */

/* Ensure that a new interval is established the next time SPEED MODE is
   selected. */
if (ADRMode != SPEED_MODE)
{
  newInt = TRUE;
  return;
}

if (newInt == TRUE)  /* Begin a new interval. */
{
  intTimer = intT;  /* Set the interval timer to the interval. */
  /* Calculate the freq's to reject based on the value of the Phi D mode
     switch. */
  switch (PhiDMode)
  {
    case 1: curFreqs[0] = 0.5 * speed;
            curFreqs[1] = speed;
            curFreqs[2] = 1.5 * speed;
            break;
    case 2: curFreqs[0] = 1.5 * speed;
            curFreqs[1] = speed;
            curFreqs[2] = 0.5 * speed;
            break;
    case 3: curFreqs[0] = speed;
            curFreqs[1] = 2.0 * speed;
            curFreqs[2] = 3.0 * speed;
            break;
    case 4: curFreqs[0] = 3.0 * speed;
            curFreqs[1] = 2.0 * speed;
            curFreqs[2] = speed;
            break;
    case 5: curFreqs[0] = speed;  /* Rotor speed only (RSO) mode. */
            curFreqs[1] = 0.0;
            curFreqs[2] = 0.0;
            break;
    default: curFreqs[0] = 0.5 * speed;  /* Set default to case 1. */
             curFreqs[1] = speed;
             curFreqs[2] = 1.5 * speed;
             break;
  }
  newInt = FALSE;  /* Forget this, and you're toast. */
}

#ifdef ENABLE_PRTG
  printf("intTimer = %7.3f \n", intTimer);
  printf("speed = %6.2f PhiDMode = %2i \n", speed, PhiDMode);
```

```
  printf("Freqs = %6.2f %6.2f %6.2f \n", curFreqs[0], curFreqs[1],
curFreqs[2]);
#endif

for (i = 0; i < MAX_NUM_FREQ; i++)  /* Output the freq's. */
  y0[i] = curFreqs[i];

intTimer = intTimer - FSS;  /* Decrement the timer and check. */
if (intTimer > 0.0)
  newInt = FALSE;
else
  newInt = TRUE;

return;

}
```

Appendix B

S-Functions

cmpAmps_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 3
#define y_width 1

void cmpAmps_Outputs_wrapper(const real_T *u0,
                         const real_T *u1,
                         real_T *y0,
                         const real_T  *ampThreshold,
                         const int_T p_width0, SimStruct *S)
{

/* This function collects the dominant freq's calculated by the DFT for
   use in composing the disturbance vector. The freq's used are the current
   ones if they exceed a threshold value or the previous ones if they do
   not. */

/* Inputs are as follow:
     u0[0,1,2] - Fourier freq's (rad/sec) corresponding to the maximum
                 amplitudes.
     u0[3,4,5] – The maximum amplitudes (mils).
     u1[0] - ADR mode. */

/* Parameter is as follows:
     ampThreshold - See comment below (mils). */

/* Outputs are as follow:
     y0[0,1,2] - Reject frequencies (rad/sec). */

int i, ADRMode;
double freqs[MAX_NUM_FREQ], ampls[MAX_NUM_FREQ];
static double curFreqs[MAX_NUM_FREQ];

ADRMode = (int) u1[0];
if (ADRMode == DFT_MODE)
  for (i = 0; i < MAX_NUM_FREQ; i++)
  {
    freqs[i] = u0[i];  /* Fetch the latest. */
```

```
    ampls[i] = u0[i + MAX_NUM_FREQ];
    /* If the latest ampl. exceeds the threshold, output the latest freq.
       If the latest ampl. is less than or equal to the threshold, output
       the current freq. This approach is used to eliminate the on/off
       rejection of persistent disturb's. Recall, that in DFT mode, once a
       disturbance is rejected, its freq. is removed from the Phi D vector,
       and the disturbance then reappears only to be rejected again ad
       infinitum. */
    if (ampls[i] > *ampThreshold)
    {
      y0[i] = freqs[i];
      curFreqs[i] = freqs[i];  /* Update the current frequency. */
    }
    else
      y0[i] = curFreqs[i];
  }

return;  /* Return and burn. */

}
```

Appendix B

S-Functions

oneFreq_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void oneFreq_Outputs_wrapper(const real_T *u0,
                          const real_T *u1,
                          const real_T *u2,
                          const real_T *u3,
                          const real_T *u4,
                          real_T *y0,
                          const real_T  *ampThreshold,
                          const int_T p_width0, SimStruct *S)
{

/* This function determines the freq. to use in the composition of the
   disturbance vector. This freq. can either be static, entered through
   ControlDesk, or dynamic, based on either the speed of the rotor or the
   output of the DFT. If the freq. is based on the former, it can be
   updated at the simulation speed or at longer intervals depending on
   configuration. If the freq. is based on the latter, it will be updated
   each time a Fourier transform has been completed. */

/* The fundamental sample size (FSS) of the simulation is used to control
   the timer for the frequency output, and this value is set in header file
   mbcntrl_a.h. */

/* Enable printing by uncommenting the following #define. */
/* #define ENABLE_PRTG */

/* Inputs are as follow:
     u0[0] - ADR mode.
     u1[0] - Reject frequency (rad/sec) - manual mode.
     u2[0] - Constant-output interval (sec).
     u3[0] - Speed of the rotor (RPM).
     u4[0,1,2] - Fourier freq's (rad/sec) corresponding to the maximum
                 amplitudes.
     u4[3,4,5] - The maximum amplitudes (mils). */

/* Parameter is as follows:
```

```
     ampThreshold - See comment below (mils). */

/* Output is as follows:
     y0[0] - Reject frequency (rad/sec). */

int i, ADRMode;
static int newInt = TRUE;
double manRejFreq, intT, speed,
       newFreq, newAmpl;
static double intTimer, rejFreq = 0;

ADRMode = (int) u0[0];  /* Fetch values from ControlDesk. */
manRejFreq = u1[0];
intT = u2[0];
speed = u3[0] * 2.0 * M_PI / 60.0;  /* Convert rotor speed to rad/sec. */

if (ADRMode == MAN_MODE)
{
  y0[0] = manRejFreq;  /* Output the manual-mode reject freq. */
  newInt = TRUE;  /* Set things up properly for speed mode. */
  #ifdef ENABLE_PRTG
    printf("ADR mode = %2i Freq. = %6.2f \n", ADRMode, manRejFreq);
  #endif
  return;
}

/* In DFT Mode, the frequency to reject must be above a user-defined
   threshold to prevent the use of a very low frequency. (0 Hz or so)
   dominant component. */
if (ADRMode == DFT_MODE)
{
  for (i = 0; i < MAX_NUM_FREQ; i++)
    if (u4[i] > FREQ_THRESHOLD)  /* Ignore the components at very low */
    {                            /* freq's. */
      newFreq = u4[i];           /* FREQ_THRESHOLD - units are rad/sec. */
      newAmpl = u4[i + MAX_NUM_FREQ];
      break;
    }
  /* Update the freq. to reject only if the ampl. threshold has been
     exceeded. See S-Function cmpAmps for further details. */
  if (newAmpl > *ampThreshold)
        rejFreq = newFreq;

  y0[0] = rejFreq;
  newInt = TRUE;  /* Set things up properly for speed mode. */
  #ifdef ENABLE_PRTG
    printf("ADR mode = %2i Freq. = %6.2f \n", ADRMode, rejFreq);
  #endif
  return;
}

if (ADRMode == SPEED_MODE)
{
```

```
  /* Ensure that an interval of 0 (i.e. freq. to reject follows the rotor
     speed exactly) is handled properly. */
  if (intT <= FSS)
  {
    y0[0] = speed;
    newInt = TRUE;
    return;
  }
  if (newInt == TRUE)  /* Begin a new interval. */
  {
    intTimer = intT;  /* Set the interval timer to the interval. */
    rejFreq = speed;
    newInt = FALSE;  /* Forget this, and you're toast. */
  }

  y0[0] = rejFreq;  /* Output the freq. */

  intTimer = intTimer - FSS;  /* Decrement the timer and check. */
  if (intTimer > 0)
    newInt = FALSE;
  else
    newInt = TRUE;

  #ifdef ENABLE_PRTG
    printf("ADR mode = %2i Speed = %6.2f \n", ADRMode, speed);
    printf("Timer = %7.5f Freq = %6.2f \n", intTimer, rejFreq);
  #endif

  return;
}

}
```

Appendix B

S-Functions

DFT_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 4
#define y_width 1

void DFT_Outputs_wrapper(const real_T *u0,
                         real_T *y0,
                         real_T *y1,
                         real_T *y2,
                         real_T *y3  ,
                         const real_T  *DFTFreq, const int_T  p_width0,
                         const real_T  *DFTFreqRes, const int_T  p_width1,
                         const real_T  *minAmp, const int_T  p_width2,
                         const real_T  *numFreq, const int_T  p_width3,
                         const real_T  *actDFT, const int_T  p_width4,
                         const real_T  *axis,  const int_T p_width5,
                         SimStruct *S)
{

/* This function calculates the Discrete Fourier Transform (DFT) from the
   displacements measured along one axis. The contribution of each data
   point to each Fourier coeff. is calculated as each data point is
   received to maintain the number of computations per invocation and thus,
   the load on the dSPACE box at nearly constant levels. The Fourier ampl's
   and freq's are calculated and sorted in ascending order of amplitude.
   The ampl's and freq's are output in descending order of amplitude.
*/

/* Set the following to 1 to sort and output at simulation speed and keep
   the DFT humming. Values greater than 1 slow things down a bit. */
#define DOWN_SAMP_SORT 1

/* Enable printing by uncommenting the following #define. */
/* #define ENABLE_PRTG */

/* Input is as follows:
     u0[0] - Displacement of one axis (mils). */
```

```
/* Parameters are as follow:
     DFTFreq - Sampling frequency (Hz).
     DFTFreqRes - Frequency resolution (Hz).
     minAmp - Amplitude threshold (mils) - determines if a dominant freq.
              is retained in the output or not.
     numFreq - Number of freq's from the maximum to output.
     actDFT - DFT active flag - 1 if the DFT is active, 0 otherwise.
     axis - Axis for the DFT calculation. */

/* Outputs are as follow:
     y0[0] - Fourier frequency (Hz).
     y1[0] - Fourier amplitude (mils).
     y2[0,1,2] - Fourier freq's corresponding to the max. amplitudes (Hz).
     y2[3,4,5] - The max. amplitudes (mils).
     y3[0] - DFT status (on or off) - used to update LED on the control
             panel. */

int i, pos;
static int firstCall = TRUE, calcActive = TRUE, sortActive = FALSE,
           numCalls = 0, numPts = 1, outCntr = 0, axis_a = 1,
           downSamp, DFTSize,
           lastDFTFreq, lastDFTFreqRes;
real_T ampVal, freqVal, conTerm;
static real_T ACoeffs[MAX_DFT_SIZE/2 + 1] = {0},
              BCoeffs[MAX_DFT_SIZE/2 + 1] = {0},
              CCoeffs[MAX_DFT_SIZE/2 + 1],
              FourFreqs[MAX_DFT_SIZE/2 + 1] = {0},
              sortAmps[MAX_DFT_SIZE/2 + 1] = {0},
              sortFreqs[MAX_DFT_SIZE/2 + 1] = {0},
              simFreq;
time_T fss;

/* If DFT is turned off (the default when animation begins), set all flags
   and counters to their initial states, send the proper signal to the
   status port, send zero to all other output ports and set the axis for
   the DFT calc. */
if ((int) *actDFT == FALSE)
{
  calcActive = TRUE;  /* Flags */
  sortActive = FALSE;

  numCalls = 0;  /* Counters */
  numPts = 1;
  outCntr = 0;

  y0[0] = 0;  /* Need comment here. */
  y1[0] = 0;

  for (i = 0; i < MAX_NUM_FREQ; i++)
  {
    y2[i] = 0;
    y2[i + MAX_NUM_FREQ] = 0;
  }
```

```
  y3[0] = 0;

  axis_a = (int) *axis;  /* Axis can only be chosen when the DFT is */
  return;                /* inactive. */
}

/* If DFT is turned on, haul the mail. */
if ((int) *actDFT == TRUE)
{
  y3[0] = 1;  /* Update status on control panel. */
  numCalls++;  /* Records the number of times this routine has been called
                  since the last DFT calculation. Used to implement down
                  sampling. */

  if (firstCall)  /* Calculate only once per simulation. */
  {
    fss = ssGetFixedStepSize(S);  /* or just use FSS from mbcntrl_a.h. */
    simFreq = 1.0 / fss;  /* Hz */
    firstCall = FALSE;
  }
  /* Reset counters if the input parameters have changed since the last
     call. */
  if (lastDFTFreq != (int) *DFTFreq || lastDFTFreqRes != (int) *DFTFreqRes)
  {
    numCalls = 0;
    numPts = 1;
    outCntr = 0;
  }
  downSamp = (int)(simFreq / *DFTFreq);
  DFTSize = (int)(*DFTFreq / *DFTFreqRes);

  /* Fetch the amplitude value from the correct axis. */
  switch (axis_a)
  {
    case 1: ampVal = u0[0];  /* from axis 1. */
            break;
    case 2: ampVal = u0[1];  /* from axis 2. */
            break;
    case 3: ampVal = u0[2];  /* from axis 3. */
            break;
    case 4: ampVal = u0[3];  /* from axis 4. */
            break;
    default: ampVal = u0[0];  /* Use axis 1 if something out of bounds */
             break;           /* slips through the net. */
  }

  /* numPts records the number of the point used in the current calc. It
     represents the variable "r" in Eqs. 4.14 and 4.15 in [ ]. I had to
     place this comment somewhere. */

  /* Calculate the Fourier coeff's - multiply by 2/N later. */
  if (calcActive == TRUE)
  {
```

```
  y0[0] = 0;  /* Keep the graph cleaner. */
  y1[0] = 0;
  if (numPts == 1 || numCalls == downSamp)
  {
    /* Note the automatic type conversion in the following expression. */
    conTerm = (2.0 * M_PI * numPts) / DFTSize;
    if (numPts == 1)  /* Reinitialize the arrays. Otherwise . . . */
      for (i = 0; i <= DFTSize/2; i++)  /* i represents var. "n" in */
      {                                 /* Eqs. 4.14 and 4.15 [ ]. */
        ACoeffs[i] = 0;
        BCoeffs[i] = 0;
      }
    for (i = 0; i <= DFTSize/2; i++)
    {
      ACoeffs[i] = ampVal * cos(conTerm * i) + ACoeffs[i];
      BCoeffs[i] = ampVal * sin(conTerm * i) + BCoeffs[i];
    }
    numPts++;  /* Increment the number of the point used. */
    numCalls = 0;
  }

  /* Finish the calculation of the Fourier coeff's.  Discard the first
     and last B Coeff's and calculate C Coeff's (the Fourier ampl's). */
  if (numPts == DFTSize + 1)  /* We're one ahead. */
  {
    BCoeffs[0] = 0;
    BCoeffs[DFTSize/2] = 0;
    FourFreqs[0] = 0;
    for (i = 0; i <= DFTSize/2; i++)
    {
      ACoeffs[i] = 2.0/DFTSize * ACoeffs[i];
      BCoeffs[i] = 2.0/DFTSize * BCoeffs[i];
      CCoeffs[i] = sqrt(pow(ACoeffs[i], 2.0) + pow(BCoeffs[i], 2.0));
    }
    /* Calculate Fourier freq's. The first has already been set to 0. */
    for (i = 1; i <= DFTSize/2; i++)
      FourFreqs[i] = i * (*DFTFreqRes);  /* Hz */

    /* Finished the DFT interval.  Set things up for the output/sort */
    /* interval. */
    calcActive = FALSE;
    sortActive = TRUE;
    numPts = 1;
    numCalls = 0;
  }
  /* Preserve the input parameters so that we can check to see if they've
     changed since the last invocation. */
  lastDFTFreq = (int) *DFTFreq;
  lastDFTFreqRes = (int) *DFTFreqRes;
  return;
}

/* May wish to include some sort of delay here or make some sort of */
/* contact. */
```

```
  /* Print, sort and print the values. */
  if (sortActive == TRUE)
  {
    if (numPts == 1)
    {
      #ifdef ENABLE_PRTG
        printf("\nDFT freq = %3i Hz\n", (int) *DFTFreq);
        printf("Down samp = %3i \nDFT size = %4i Pts\n\n", downSamp,
DFTSize);
        printf("   n    Freq(r/s)     Amp \n");
      #endif
      for (i = 0; i <= DFTSize/2; i++)  /* Reinitialize the arrays. */
      {
        sortAmps[i] = 0;
        sortFreqs[i] = 0;
      }
    }
    /* Sort the Fourier ampl's and freq's in ascending order. The insertion
       sort described in [ ] is used, and each point is sorted vis-a-vis
       the current ones as it is received. */
    if (numPts == 1 || numCalls == DOWN_SAMP_SORT)
    {
      freqVal = FourFreqs[outCntr] * 2.0 * M_PI;  /* rad/sec */
      ampVal = CCoeffs[outCntr++];
      for (pos = numPts - 1; pos > 0 && ampVal < sortAmps[pos - 1]; pos--)
      {
        sortAmps[pos] = sortAmps[pos - 1];
        sortFreqs[pos] = sortFreqs[pos - 1];
      }
      sortAmps[pos] = ampVal;
      sortFreqs[pos] = freqVal;

      y0[0] = freqVal;  /* Output the freq. (rad/sec). */
      y1[0] = ampVal;   /* Ditto the amplitude. */
      #ifdef ENABLE_PRTG
        printf("%4i     %6.3f     %6.3f \n", numPts - 1, freqVal, ampVal);
      #endif
      numPts++;
      numCalls = 0;
    }

    /* Finished the output interval.  Print the sorted values in descending
       order, output the dominant freq's and set things up for the next DFT
       calculation. */
    if (numPts == (int)(DFTSize/2 + 2))  /* Again, we're one ahead. */
    {
      #ifdef ENABLE_PRTG
        printf("   n    Freq(r/s)     Amp \n");
        for (i = DFTSize/2; i >= 0; i--)
          printf("%4i     %6.3f     %6.3f \n", i, sortFreqs[i],
sortAmps[i]);
      #endif
      /* Set the amplitude to 0 to avoid drawing a long, angled line across
         the graph. */
```

```
      y1[0] = 0;
      /* Output only those ampl's (though no more than the max. number)
         greater than or equal to the threshold and their corresponding
         freq's, but first write zero to all of the output ports to keep
         the panel cleaner. */
      for (i = 0; i < MAX_NUM_FREQ; i++)
      {
        y2[i] = 0;
        y2[i + MAX_NUM_FREQ] = 0;
      }
      for (i = 0; i < (int) *numFreq; i++)
        if (sortAmps[DFTSize/2 - i] >= *minAmp)
        {
          y2[i] = sortFreqs[DFTSize/2 - i];  /* Frequency (Hz) */
          y2[i + MAX_NUM_FREQ] = sortAmps[DFTSize/2 - i]; /* Ampl (mils) */
        }
        else
        {
          y2[i] = 0;
          y2[i + MAX_NUM_FREQ] = 0;
        }
      #ifdef ENABLE_PRTG
        printf("Max. amps      : \n %5.2f \n %5.2f \n %5.2f \n", y2[3],
y2[4], y2[5]);
        printf("    freqs (r/s): \n %5.2f \n %5.2f \n %5.2f \n", y2[0],
y2[1], y2[2]);
      #endif
      sortActive = FALSE;
      calcActive = TRUE;
      numPts = 1;
      numCalls = 0;
      outCntr = 0;
    }
    /* Preserve the input parameters as before. */
    lastDFTFreq = (int) *DFTFreq;
    lastDFTFreqRes = (int) *DFTFreqRes;
    return;
  }
}
}
```

Appendix B

S-Functions

updateOmega_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#include "simstruc.h"
#include <math.h>
#include "mbcntrl_a.h"

#define u_width 1
#define y_width 1

void updateOmega_Outputs_wrapper(const real_T *u0,
                          const real_T *u1,
                          const real_T *u2,
                          const real_T *u3,
                          const real_T *u4,
                          const real_T *u5,
                          const real_T *u6,
                          real_T *y0, SimStruct *S)
{

/* This function determines the freq. to use in the composition of the
   disturb. vector. This freq. can either be constant or dynamic, based
   the speed of the rotor. If it's dynamic, values for the speed
   measurement interval, speed update increment and speed update interval
   must be chosen. */

/* Enable printing by uncommenting the following #define. */
#define ENABLE_PRTG

/* Inputs are as follow:
     u0[0] - Simulation time (sec).
     u1[0] - Actual rotor speed (RPM).
     u2[0] - Speed measurement interval (secs).
     u3[0] - Speed update increment (RPM).
     u4[0] - Speed update interval (multiples of FSS).
     u5[0] - Adaptive controller mode.
     u6[0] - Manual reject frequency (rad/sec). */

/* Output is as follows:
     y0[0] - Speed (rad/sec). */

int currMode;
static int newMeas = TRUE, measAct = FALSE, updateAct = FALSE, incrRPM,
```

```
          prevMode = MAN_MODE;
double simT, manRejFreq;
static double measTimer, begRPM, nextRPM, endRPM, RPMIncr, updateTimer;

simT = u0[0];  /* Fetch the simulation time, */
currMode = (int) u5[0];  /* controller mode and */
manRejFreq = u6[0];      /* manual reject frequency. */

/* Ensure that things are initialized properly. */
if (prevMode != SPEED_MODE)  /* If the mode was not and is not SPEED, */
  if (currMode != SPEED_MODE)
  {
      y0[0] = manRejFreq;  /* output the manual-mode reject freq, */
      prevMode = currMode;  /* preserve the current mode and return. */
      #ifdef ENABLE_PRTG
        printf(" Other mode, speed = %6.3f \n", manRejFreq);
      #endif
      return;
  }
  else  /* If the mode was not SPEED but now it is, begin a measurement */
  {
    begRPM = u1[0];  /* interval. */
    newMeas = TRUE;
    prevMode = currMode;
    #ifdef ENABLE_PRTG
      printf("\n\n ----- A New Analysis -----\n\n");
    #endif
  }
else if (currMode != SPEED_MODE)  /* If the mode was SPEED and now it is */
{
  prevMode = currMode;  /* not, preserve the current mode and return. */
  return;
}

/* If previous and currents modes are SPEED, continue onward. */
if (newMeas == TRUE)  /* Begin the measurement interval. */
{
  measTimer = u2[0];  /* Fetch the meas. interval from ControlDesk. */
  y0[0] = begRPM * (2*M_PI/60.0);
  #ifdef ENABLE_PRTG
    printf("\n Begin meas. interval, time = %7.3f sec,", simT);
    printf(" speed = %6.3f \n", begRPM);
  #endif
  nextRPM = begRPM;  /* Preserve it for later use. */
  newMeas = FALSE;
  measAct = TRUE;
  return;
}

if (newMeas == FALSE && measAct == TRUE)
{
  y0[0] = begRPM * (2*M_PI/60.0);
  measTimer = measTimer - FSS;  /* Decrement the timer and check. */
```

```
  if (measTimer > 0.0)  /* If the meas. interval hasn't expired, return. */
    return;
                    /* If it has, */
  endRPM = u1[0];  /* read the current speed. */
  #ifdef ENABLE_PRTG
    printf(" End meas. interval, time = %7.3f sec,", simT);
    printf(" speed = %6.3f \n", endRPM);
  #endif
  if (begRPM != endRPM)  /* If the speed has changed over the interval, */
  {
    #ifdef ENABLE_PRTG
      printf(" Begin update interval, time = %7.3f sec \n", simT);
    #endif
    RPMIncr = u3[0];  /* fetch the update increment and interval values */
    updateTimer = u4[0] * FSS;  /* from ControlDesk. */
    if (begRPM < endRPM) /* If the speed has increased, */
      incrRPM = TRUE;  /* increment. */
    else
      incrRPM = FALSE;  /* Otherwise, decrement. */
    measAct = FALSE;
    updateAct = TRUE;
  }
  else  /* If the meas. interval has expired and the speed hasn't changed,
*/
  {
    newMeas = TRUE;  /* begin a new measurement interval. */
    return;
  }
}

if (updateAct == TRUE)  /* Update the reject frequency. */
{
  updateTimer = updateTimer - FSS;  /* Decrement the timer and check. */
  if (updateTimer > 0.0) /* If the update interval hasn't expired, */
    return;              /* return. */
  /* If it has and the speed is increasing and the last update is less
     than the final value, */
  if (incrRPM == TRUE && nextRPM < endRPM)
  {
    nextRPM = nextRPM + RPMIncr;  /* calculate the next update value. */
    if (nextRPM > endRPM)
      nextRPM = endRPM;
    y0[0] = nextRPM * (2*M_PI/60.0);  /* Update it in rads/sec. */
    #ifdef ENABLE_PRTG
      printf(" End update interval, time = %7.3f sec,", simT);
      printf(" speed = %6.3f\n", nextRPM);
    #endif
    updateTimer = u4[0] * FSS;
  }
  /* If it has and the speed is decreasing and the last update is more
     than the final value, */
  else if (incrRPM == FALSE && endRPM < nextRPM)
  {
    nextRPM = nextRPM - RPMIncr;  /* calculate the next update value. */
```

```
    if (nextRPM < endRPM)
      nextRPM = endRPM;
    y0[0] = nextRPM * (2*M_PI/60.0);  /* Update it in rads/sec. */
    #ifdef ENABLE_PRTG
      printf(" End update interval, time = %7.3f sec,", simT);
      printf(" speed = %6.3f\n", nextRPM);
    #endif
    updateTimer = u4[0] * FSS;
  }
  else  /* If the update interval has expired and updating is complete, */
  {
    begRPM = nextRPM;
    updateAct = FALSE;  /* begin a new speed measurement interval. */
    newMeas = TRUE;
  }

  return;
}

}
```

Appendix B

S-Functions

genSpeed_wrapper.c

```
/*
 *   --- THIS FILE GENERATED BY S-FUNCTION BUILDER: 3.0 ---
 */

#if defined(MATLAB_MEX_FILE)
  #include "tmwtypes.h"
  #include "simstruc_types.h"
#else
  #include "rtwtypes.h"
#endif

#include <math.h>

#define u_width 1
#define y_width 1

void genSpeed_Outputs_wrapper(const real_T *u0,
                              real_T *y0)
{

/* This function generates a profile of the rotor speed using your basic
   multiway if. */

/* Input is as follows:
     u0[0] - Simulation time (sec).

 /* Output is as follows:
     y0[0] - Rotor speed (RPM). */

double simT, speed;

simT = u0[0];  /* Fetch the simulation time, */

if (simT >= 0.0 && simT < 0.9)  /* and generate the speed profile. */
  speed = 600;
if (simT >= 0.9 && simT < 1.9)
  speed = 604;
if (simT >= 1.9 && simT < 2.9)
  speed = 608;
if (simT >= 2.9 && simT < 3.8)
  speed = 609;
if (simT >= 3.8 && simT < 4.9)
  speed = 609;
if (simT >= 4.9 && simT < 5.9)
```

```
  speed = 612;
if (simT >= 5.9 && simT < 6.9)
  speed = 607;
if (simT >= 6.9 && simT < 7.9)
  speed = 602;
if (simT >= 7.9 && simT < 8.9)
  speed = 599;
if (simT >= 8.9 && simT < 9.9)
  speed = 603;
if (simT >= 9.9)
  speed = 606;

y0[0] = speed;

return;

}
```

Appendix C

Profile Reports

Simulink Model Adapt_Cntrl_1E

Simulink Profile Report: Summary

Report generated 11-Jun-2013 13:56:08

Total recorded time:	1.33 s
Number of Block Methods:	61
Number of Internal Methods:	7
Number of Nonvirtual Subsystem Methods:	2
Clock precision:	0.00000003 s
Clock Speed:	3000 Mhz

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Web Browser to view)
sim	1.32812500	100.0%	1	1.3281250000000	0.00000000	0.0%	Adapt_Cntrl_1E
ModelExecute	1.23437500	92.9%	1	1.2343750000000	0.07812500	5.9%	Adapt_Cntrl_1E
MajorOutputs	1.14062500	85.9%	6302	0.0001809941288	0.01562500	1.2%	Adapt_Cntrl_1E
Adapt_Cntrl_1E (Output)	1.12500000	84.7%	6302	0.0001785147572	0.57812500	43.5%	Adapt_Cntrl_1E
ModelInitialize	0.09375000	7.1%	1	0.0937500000000	0.09375000	7.1%	Adapt_Cntrl_1E
Adapt_Cntrl_1E/Embedded MATLAB Function (Output)	0.06250000	4.7%	6001	0.0000104149308	0.01562500	1.2%	Adapt_Cntrl_1E/Embedded MATLAB Function
Adapt_Cntrl_1E/Embedded MATLAB Function/ SFunction (Output)	0.04687500	3.5%	6001	0.0000078111981	0.04687500	3.5%	Adapt_Cntrl_1E/Embedded MATLAB Function/ SFunction
Adapt_Cntrl_1E/Sum1 (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_1E/Sum1
Adapt_Cntrl_1E/D2 (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_1E/D2
Adapt_Cntrl_1E/Cont Out (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_1E/Cont Out
Adapt_Cntrl_1E/Sum (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_1E/Sum
Adapt_Cntrl_1E/S-Function Builder (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_1E/S-Function Builder
MajorUpdate	0.01562500	1.2%	6302	0.0000024793716	0.01562500	1.2%	Adapt_Cntrl_1E
Adapt_Cntrl_1E/beta (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/beta
Adapt_Cntrl_1E/Product10 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Product10
Adapt_Cntrl_1E/Ks2 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Ks2
Adapt_Cntrl_1E/D3 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/D3
Adapt_Cntrl_1E/rad//sec (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/rad//sec
Adapt_Cntrl_1E/Hp (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Hp
Adapt_Cntrl_1E/Gp (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Gp
Adapt_Cntrl_1E/Product9 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Product9
Adapt_Cntrl_1E/Sum2 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Sum2
Adapt_Cntrl_1E/Product5 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Product5
Adapt_Cntrl_1E/Trigonometric Function (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Trigonometric Function
Adapt_Cntrl_1E/Rate Transition (Output)	0.01562500	1.2%	6302	0.0000024793716	0.01562500	1.2%	Adapt_Cntrl_1E/Rate Transition
Adapt_Cntrl_1E/Matrix 1-Norm (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Matrix 1-Norm
Adapt_Cntrl_1E/Integrator2 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Integrator2
Adapt_Cntrl_1E/Product7 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_1E/Product7

Appendix C

Profile Reports

Simulink Model Adapt_Cntrl_1S

Simulink Profile Report: Summary

Report generated 11-Jun-2013 14:14:29

Total recorded time:	1.17 s
Number of Block Methods:	60
Number of Internal Methods:	7
Number of Nonvirtual Subsystem Methods:	1
Clock precision:	0.00000003 s
Clock Speed:	3000 Mhz

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Web Browser to view)
sim	1.17187500	100.0%	1	1.1718750000000	0.00000000	0.0%	Adapt_Cntrl_1S
ModelExecute	1.10937500	94.7%	1	1.1093750000000	0.03125000	2.7%	Adapt_Cntrl_1S
MajorOutputs	1.00000000	85.3%	6302	0.0001586797842	0.03125000	2.7%	Adapt_Cntrl_1S
Adapt_Cntrl_1S (Output)	0.96875000	82.7%	6302	0.0001537210409	0.29687500	25.3%	Adapt_Cntrl_1S
MajorUpdate	0.07812500	6.7%	6302	0.0000123968581	0.03125000	2.7%	Adapt_Cntrl_1S
Adapt_Cntrl_1S/Math Function1 (Output)	0.06250000	5.3%	12002	0.0000052074654	0.06250000	5.3%	Adapt_Cntrl_1S/Math Function1
ModelInitialize	0.06250000	5.3%	1	0.0625000000000	0.06250000	5.3%	Adapt_Cntrl_1S
Adapt_Cntrl_1S/Hp (Output)	0.04687500	4.0%	6001	0.0000078111981	0.04687500	4.0%	Adapt_Cntrl_1S/Hp
Adapt_Cntrl_1S/GpAct (Output)	0.04687500	4.0%	6001	0.0000078111981	0.04687500	4.0%	Adapt_Cntrl_1S/GpAct
Adapt_Cntrl_1S/Integrator1 (Update)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/Integrator1
Adapt_Cntrl_1S/Cont Out (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/Cont Out
Adapt_Cntrl_1S/S-Function Builder3 (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/S-Function Builder3
Adapt_Cntrl_1S/Trigonometric Function1 (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/Trigonometric Function1
Adapt_Cntrl_1S/S-Function Builder (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/S-Function Builder
Adapt_Cntrl_1S/Ks1 (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/Ks1
Adapt_Cntrl_1S/x in1 (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/x in1
Adapt_Cntrl_1S/Matrix 1-Norm1 (Output)	0.03125000	2.7%	6001	0.0000052074654	0.03125000	2.7%	Adapt_Cntrl_1S/Matrix 1-Norm1
Adapt_Cntrl_1S/Integrator2 (Update)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/Integrator2
Adapt_Cntrl_1S/S-Function Builder2 (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/S-Function Builder2
Adapt_Cntrl_1S/Product3 (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/Product3
Adapt_Cntrl_1S/beta (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/beta
Adapt_Cntrl_1S/Product (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/Product
Adapt_Cntrl_1S/Ks2 (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/Ks2
Adapt_Cntrl_1S/D4 (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/D4
Adapt_Cntrl_1S/D2 (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/D2
Adapt_Cntrl_1S/C1 (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/C1
Adapt_Cntrl_1S/Gp (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/Gp
Adapt_Cntrl_1S/ADRActive (Output)	0.01562500	1.3%	6001	0.0000026037327	0.01562500	1.3%	Adapt_Cntrl_1S/ADRActive

Appendix C

Profile Reports

Simulink Model Adapt_Cntrl_3E

Simulink Profile Report: Summary

Report generated 11-Jun-2013 15:13:09

Total recorded time:	6.72 s
Number of Block Methods:	65
Number of Internal Methods:	7
Number of Nonvirtual Subsystem Methods:	2
Clock precision:	0.00000003 s
Clock Speed:	3000 Mhz

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Web Browser to view)
sim	6.71875000	100.0%	1	6.7187500000000	0.00000000	0.0%	Adapt_Cntrl_3E
ModelExecute	6.59375000	98.1%	1	6.5937500000000	0.17187500	2.6%	Adapt_Cntrl_3E
MajorOutputs	6.39062500	95.1%	6302	0.0010140629959	0.03125000	0.5%	Adapt_Cntrl_3E
Adapt_Cntrl_3E (Output)	6.35937500	94.7%	6302	0.0010091042526	1.23437500	18.4%	Adapt_Cntrl_3E
Adapt_Cntrl_3E/Embedded MATLAB Function (Output)	3.73437500	55.6%	6001	0.0006222921180	0.03125000	0.5%	Adapt_Cntrl_3E/Embedded MATLAB Function
Adapt_Cntrl_3E/Embedded MATLAB Function/ SFunction (Output)	3.70312500	55.1%	6001	0.0006170846526	3.70312500	55.1%	Adapt_Cntrl_3E/Embedded MATLAB Function/ SFunction
Adapt_Cntrl_3E/Rate Transition (Output)	0.12500000	1.9%	6302	0.0000198349730	0.12500000	1.9%	Adapt_Cntrl_3E/Rate Transition
Adapt_Cntrl_3E/Product6 (Output)	0.12500000	1.9%	6001	0.0000208298617	0.12500000	1.9%	Adapt_Cntrl_3E/Product6
ModelInitialize	0.10937500	1.6%	1	0.1093750000000	0.10937500	1.6%	Adapt_Cntrl_3E
Adapt_Cntrl_3E/Trigonometric Function3 (Output)	0.09375000	1.4%	6001	0.0000156223963	0.09375000	1.4%	Adapt_Cntrl_3E/Trigonometric Function3
Adapt_Cntrl_3E/Product2 (Output)	0.09375000	1.4%	6001	0.0000156223963	0.09375000	1.4%	Adapt_Cntrl_3E/Product2
Adapt_Cntrl_3E/Hp (Output)	0.06250000	0.9%	6001	0.0000104149308	0.06250000	0.9%	Adapt_Cntrl_3E/Hp
Adapt_Cntrl_3E/Matrix 1-Norm (Output)	0.06250000	0.9%	6001	0.0000104149308	0.06250000	0.9%	Adapt_Cntrl_3E/Matrix 1-Norm
Adapt_Cntrl_3E/Logical Operator (Output)	0.06250000	0.9%	6001	0.0000104149308	0.06250000	0.9%	Adapt_Cntrl_3E/Logical Operator
Adapt_Cntrl_3E/Integrator1 (Update)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Integrator1
MajorUpdate	0.03125000	0.5%	6302	0.0000049587433	0.00000000	0.0%	Adapt_Cntrl_3E
Adapt_Cntrl_3E/Product12 (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Product12
Adapt_Cntrl_3E/Product10 (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Product10
Adapt_Cntrl_3E/Product1 (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Product1
Adapt_Cntrl_3E/rad//sec (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/rad//sec
Adapt_Cntrl_3E/Hp1 (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Hp1
Adapt_Cntrl_3E/Gp (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Gp
Adapt_Cntrl_3E/Cont Out (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/Cont Out
Adapt_Cntrl_3E/S-Function Builder3 (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/S-Function Builder3
Adapt_Cntrl_3E/ADRActive (Output)	0.03125000	0.5%	6001	0.0000052074654	0.03125000	0.5%	Adapt_Cntrl_3E/ADRActive

Appendix C

Profile Reports

Simulink Model Adapt_Cntrl_3S

Simulink Profile Report: Summary

Report generated 11-Jun-2013 15:31:54

Total recorded time:	1.30 s
Number of Block Methods:	64
Number of Internal Methods:	7
Number of Nonvirtual Subsystem Methods:	1
Clock precision:	0.00000003 s
Clock Speed:	3000 Mhz

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Web Browser to view)
sim	1.29687500	100.0%	1	1.2968750000000	0.00000000	0.0%	Adapt_Cntrl_3S
ModelExecute	1.25000000	96.4%	1	1.2500000000000	0.01562500	1.2%	Adapt_Cntrl_3S
MajorOutputs	1.15625000	89.2%	6302	0.0001834735005	0.01562500	1.2%	Adapt_Cntrl_3S
Adapt_Cntrl_3S (Output)	1.14062500	88.0%	6302	0.0001809941288	0.46875000	36.1%	Adapt_Cntrl_3S
MajorUpdate	0.07812500	6.0%	6302	0.0000123968581	0.04687500	3.6%	Adapt_Cntrl_3S
Adapt_Cntrl_3S/S-Function Builder2 (Output)	0.06250000	4.8%	6001	0.0000104149308	0.06250000	4.8%	Adapt_Cntrl_3S/S-Function Builder2
Adapt_Cntrl_3S/S-Function Builder (Output)	0.04687500	3.6%	6001	0.0000078111981	0.04687500	3.6%	Adapt_Cntrl_3S/S-Function Builder
ModelInitialize	0.04687500	3.6%	1	0.0468750000000	0.04687500	3.6%	Adapt_Cntrl_3S
Adapt_Cntrl_3S/Product12 (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/Product12
Adapt_Cntrl_3S/D4 (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/D4
Adapt_Cntrl_3S/Trigonometric Function4 (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/Trigonometric Function4
Adapt_Cntrl_3S/Product2 (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/Product2
Adapt_Cntrl_3S/Matrix 1-Norm (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/Matrix 1-Norm
Adapt_Cntrl_3S/HpAct (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/HpAct
Adapt_Cntrl_3S/Logical Operator (Output)	0.03125000	2.4%	6001	0.0000052074654	0.03125000	2.4%	Adapt_Cntrl_3S/Logical Operator
Adapt_Cntrl_3S/Integrator2 (Update)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/Integrator2
Adapt_Cntrl_3S/Integrator1 (Update)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/Integrator1
Adapt_Cntrl_3S/Sum1 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/Sum1
Adapt_Cntrl_3S/Product (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/Product
Adapt_Cntrl_3S/Ks2 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/Ks2
Adapt_Cntrl_3S/D3 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/D3
Adapt_Cntrl_3S/D2 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/D2
Adapt_Cntrl_3S/x in (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/x in
Adapt_Cntrl_3S/S-Function Builder3 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/S-Function Builder3
Adapt_Cntrl_3S/ADRActive (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/ADRActive
Adapt_Cntrl_3S/Product5 (Output)	0.01562500	1.2%	6001	0.0000026037327	0.01562500	1.2%	Adapt_Cntrl_3S/Product5
Adapt_Cntrl_3S/Math Function1 (Output)	0.01562500	1.2%	12002	0.0000013018664	0.01562500	1.2%	Adapt_Cntrl_3S/Math Function1

Appendix C

Profile Reports

Simulink Model State_Estimator_prof

Simulink Profile Report: Summary

Report generated 11-Jun-2013 19:04:36

Total recorded time:	5.38 s
Number of Block Methods:	28
Number of Internal Methods:	6
Number of Nonvirtual Subsystem Methods:	2
Clock precision:	0.00000003 s
Clock Speed:	3000 Mhz

Function List

Name	Time		Calls	Time/call	Self time		Location (must use MATLAB Web Browser to view)
sim	5.37500000	100.0%	1	5.37500000000000	0.00000000	0.0%	State_Estimator_prof
ModelExecute	5.35937500	99.7%	1	5.35937500000000	0.18750000	3.5%	State_Estimator_prof
MajorOutputs	3.92187500	73.0%	60001	0.0000653634939	0.06250000	1.2%	State_Estimator_prof
State_Estimator_prof (Output)	3.85937500	71.8%	60001	0.0000643218446	1.76562500	32.8%	State_Estimator_prof
MajorUpdate	1.25000000	23.3%	60001	0.0000208329861	0.12500000	2.3%	State_Estimator_prof
State_Estimator_prof (Update)	1.12500000	20.9%	60001	0.0000187496875	0.45312500	8.4%	State_Estimator_prof
State_Estimator_prof/Sine Wave1 (Update)	0.26562500	4.9%	60001	0.0000044270095	0.26562500	4.9%	State_Estimator_prof/Sine Wave1
State_Estimator_prof/Sine Wave (Update)	0.17187500	3.2%	60001	0.0000028645356	0.17187500	3.2%	State_Estimator_prof/Sine Wave
State_Estimator_prof/Calibration (Output)	0.17187500	3.2%	60001	0.0000028645356	0.17187500	3.2%	State_Estimator_prof/Calibration
State_Estimator_prof/x in1 (Output)	0.17187500	3.2%	60001	0.0000028645356	0.17187500	3.2%	State_Estimator_prof/x in1
State_Estimator_prof/Sum1 (Output)	0.14062500	2.6%	60001	0.0000023437109	0.14062500	2.6%	State_Estimator_prof/Sum1
State_Estimator_prof/x in2 (Output)	0.14062500	2.6%	60001	0.0000023437109	0.14062500	2.6%	State_Estimator_prof/x in2
State_Estimator_prof/Display (Output)	0.14062500	2.6%	60001	0.0000023437109	0.14062500	2.6%	State_Estimator_prof/Display
State_Estimator_prof/Integrator (Update)	0.12500000	2.3%	60001	0.0000020832986	0.12500000	2.3%	State_Estimator_prof/Integrator
State_Estimator_prof/Sine Wave2 (Update)	0.10937500	2.0%	60001	0.0000018228863	0.10937500	2.0%	State_Estimator_prof/Sine Wave2
State_Estimator_prof/L (Output)	0.10937500	2.0%	60001	0.0000018228863	0.10937500	2.0%	State_Estimator_prof/L
State_Estimator_prof/S-Function Builder (Output)	0.10937500	2.0%	60001	0.0000018228863	0.10937500	2.0%	State_Estimator_prof/S-Function Builder
State_Estimator_prof/Sine Wave2 (Output)	0.10937500	2.0%	60001	0.0000018228863	0.10937500	2.0%	State_Estimator_prof/Sine Wave2
State_Estimator_prof/Integrator (Output)	0.09375000	1.7%	60001	0.0000015624740	0.09375000	1.7%	State_Estimator_prof/Integrator
State_Estimator_prof/AxialColocation (Output)	0.09375000	1.7%	60001	0.0000015624740	0.09375000	1.7%	State_Estimator_prof/AxialColocation
State_Estimator_prof/Sine Wave1 (Output)	0.09375000	1.7%	60001	0.0000015624740	0.09375000	1.7%	State_Estimator_prof/Sine Wave1
State_Estimator_prof/A-LC (Output)	0.07812500	1.5%	60001	0.0000013020616	0.07812500	1.5%	State_Estimator_prof/A-LC
State_Estimator_prof/mm (Output)	0.07812500	1.5%	60001	0.0000013020616	0.07812500	1.5%	State_Estimator_prof/mm
State_Estimator_prof/C' (Output)	0.06250000	1.2%	60001	0.0000010416493	0.06250000	1.2%	State_Estimator_prof/C'
State_Estimator_prof/Gain1 (Output)	0.06250000	1.2%	60001	0.0000010416493	0.06250000	1.2%	State_Estimator_prof/Gain1
State_Estimator_prof/1 stroke Ks (Output)	0.06250000	1.2%	60001	0.0000010416493	0.06250000	1.2%	State_Estimator_prof/1 stroke Ks
State_Estimator_prof/Sum (Output)	0.06250000	1.2%	60001	0.0000010416493	0.06250000	1.2%	State_Estimator_prof/Sum
State_Estimator_prof/Sum3 (Output)	0.06250000	1.2%	60001	0.0000010416493	0.06250000	1.2%	State_Estimator_prof/Sum3
State_Estimator_prof/C'1 (Output)	0.04687500	0.9%	60001	0.0000007812370	0.04687500	0.9%	State_Estimator_prof/C'1
State_Estimator_prof/B (Output)	0.04687500	0.9%	60001	0.0000007812370	0.04687500	0.9%	State_Estimator_prof/B
State_Estimator_prof/Average (Output)	0.04687500	0.9%	60001	0.0000007812370	0.04687500	0.9%	State_Estimator_prof/Average
State_Estimator_prof/Sine Wave (Output)	0.04687500	0.9%	60001	0.0000007812370	0.04687500	0.9%	State_Estimator_prof/Sine Wave
State_Estimator_prof/Gain2 (Output)	0.03125000	0.6%	60001	0.0000005208247	0.03125000	0.6%	State_Estimator_prof/Gain2
State_Estimator_prof/Sum2 (Output)	0.03125000	0.6%	60001	0.0000005208247	0.03125000	0.6%	State_Estimator_prof/Sum2
ModelInitialize	0.01562500	0.3%	1	0.0156250000000	0.01562500	0.3%	State_Estimator_prof
ModelTerminate	0.00000000	0.0%	1	0.0000000000000	0.00000000	0.0%	State_Estimator_prof